Gabarito Lista AED

1.

a) consumo de memória em cada uma das abordagens.

Na lista sequencial, por usamos um array, há uma alocação contígua, ou seja, os elementos são armazenados em posições consecutivas e sequenciais na memória, formando um bloco contínuo. Caso seja alocada com uma capacidade maior que a necessária, teremos um desperdício de memória, por conta disso, só vale a pena usar esta abordagem quanto o tamanho que o array deve ter seja conhecido ou possível de prever.

Já na lista ligada, termos a grande vantagem da alocação ser dinâmica e sobre demanda, não correndo mais o risco de alocamos uma capacidade maior do que aquela que iremos usar. Um ponto negativo porém, é o overhead de ponteiros, pois cada nó irá armazenar o dado/elemento + o ponteiro.
Exemplo: Para int (4 bytes) + ponteiro (8 bytes) = 12 bytes por elemento (66% é overhead)

OBS: só relembrando, overhead é um "peso extra" ou "custo adicional" que uma estrutura ou operação carrega, além do que é estritamente necessário. Em lista ligadas, é o custo dos ponteiros que nos dá a flexibilidade de inserção/remoção eficiente.

b) acesso a um elemento dado um índice.

Na lista sequencial, a complexidade de acesso a um elemento dado um índice será constante, ou seja, O(1), pois o tempo de acesso não depende do tamanho da lista ou da posição do elemento. Dado o índice, o acesso a qualquer elemento será sempre constante.

Já na lista ligada, será O(n), pois no pior caso teremos que percorrer a lista inteira até chegamos no índice desejado.

Alguns exemplos:

- Para acessar o elemento de índice 0 (primeiro): 0 passos.
- Para acessar o elemento de índice 1 (segundo): 1 passo, pois teremos que passar primeiro elemento.
- Para acessar o elemento de índice 2 (terceiro): 2 passos, pois teremos que passar pelos primeiros dois elementos.
- Para acessar o elemento de índice 99 (centésimo): 99 passos, pois teremos que passar pelos primeiros 99 elementos.
  E assim sucessivamente

c) eficiência da operação de inserção

Inserção por posição:

Lista Sequencial:

- **Início:** Caso seja no início, no pior caso teremos que deslocar todos os elementos, logo O(n)
- **Meio:** No pior caso também teremos que deslocar os elementos a partir da posição, logo O(n)
- **Fim (com espaço):** Caso haja espaço, podemos simplesmente no espaço vazio no fim, logo O(1)
- **Fim (sem espaço):** Caso não haja espaço, teremos que criar um novo array (por exemplo, com o dobro de tamanho do array anterior), e realoca os elementos do array antigo para esse novo array, logo O(n).

Lista Ligada:

- **Início:** No início, teremos que apenas ajustar 2 ponteiros, o que é uma operação constante, logo O(1).
- **Meio:** O(n) para buscar posição + O(1) para inserir, logo será O(n) + O(1) = O(n)
- **Fim (sem ponteiro último):** Se não tivermos um ponteiro para o último, teremos que percorrer todos os elementos, logo O(n).
- **Fim (com ponteiro último):** Com ponteiro para o último elemento teremos acesso direto à ele, logo O(1)

Inserção Ordenada:

Lista Sequencial:

- **Busca da posição:** Pode ser O(n) ou O(log n) com busca binária
- **Inserção:** Precisa deslocar elementos para abrir espaço, logo O(n)
- **Total:** O(n) porque o **deslocamento domina**

Lista Ligada:

- Busca da posição: Precisa percorrer a lista até achar onde inserir, logo O(n)
- Inserção: Na inserção apenas os ponteiros são ajustados, logo O(1)
- **Total:** O(n) por conta que a busca domina

d) Eficiência da operação de remoção

Lista Sequencial:

- **Início:** Desloca todos os elementos restantes -> O(n)
- **Meio:** Desloca elementos após a posição -> O(n)
- **Fim:** Apenas decrementa contador -> O(1)
- **Por valor:** O(n) para buscar + O(n) para remover = O(n)

Lista Ligada:

- **Início:** Ajusta ponteiro do início -> O(1)
- **Meio/Fim:** O(n) para buscar elemento + O(1) para remover = O(n)
- **Por valor:** O(n) para buscar + O(1) para remover = O(n)
- **Observação:** Se já se tem o nó a ser removido, a remoção é O(1)

2.
A organização ordenada pode de fato acelerar a busca, mas o benefício irá variar conforme o tipo de lista.
Em listas sequenciais a ordenação nos permite usar a busca binária, dividindo o espaço de busca pela metade a cada comparação, o que reduz a complexidade O(n) de para O(log n), uma melhoria bastante significativa.
Já em listas ligadas não é possível aplicar a busca binária, logo a ordenação não oferece melhoria significativa, pois a busca continua sendo O(n). A única vantagem que a gente acaba tendo é a possibilidade poder interromper a busca mais cedo quando encontramos um valor maior do que aquele que estamos procurando.

3.
A busca com sentinela colocaremos o elemento procurado como último elemento da lista, garantindo que a busca sempre encontrará o elemento. Isso nos elimina a necessidade de verificar se chegou ao final da lista a cada iteração.
A grande vantagem é justamente redução de comparações: de 2 para 1 por elemento no pior caso, tornando a busca aproximadamente 2x mais rápida em listas grandes.

4.

a) Uso de um nó cabeça (sem conteúdo útil)

A principal vantagem é a eliminação de casos especiais para operações no início da lista. Em uma lista ligada comum, precisamos verificar se a lista está vazia ou se estamos manipulando o primeiro elemento, o que adiciona complexidade ao código. Com o nó cabeça, todas as operações se tornam consistentes, simplificando a lógica e reduzindo a chance de erros, pois sempre teremos um nó anterior para referenciar.

b) Último nó apontando para o primeiro, tornando a lista circular

A vantagem aqui é a possibilidade de navegação contínua pela lista, além de facilitar o acesso rápido ao último elemento a partir do primeiro, otimizando algumas operações.

c) Uso de um ponteiro adicional que aponta para o último nó

Esta modificação torna a inserção no final da lista uma operação de tempo constante O(1), em vez de O(n). Sem este ponteiro, precisamos percorrer toda a lista até encontrar o último elemento para realizar uma inserção no fim, o que se torna ineficiente para listas grandes ou quando há muitas inserções no final.

d) Nós apontando para o elemento anterior, além do próximo, tornando a lista duplamente ligada

As vantagens são bem significativas, permite navegação nos dois sentidos (do início para o fim e vice-versa), remove qualquer nó em tempo constante O(1) quando já se tem sua referência, e facilita operações que precisam acessar elementos anteriores. O custo é o aumento no consumo de memória, pois cada nó agora armazena dois ponteiros em vez de um.

5.

a) Fila

Para implementar uma fila, escolheria a lista ligada como base, pois ela segue o princípio FIFO (First-In-First-Out), onde inserções ocorrem no final e remoções no início.
Com uma lista ligada simples, a remoção no início é O(1), mas a inserção no final seria O(n). No entanto, com uma modificação mínima , que seria adicionar um ponteiro para o último nó, conseguimos inserção no final em O(1) também. Dessa forma, ambas as operações principais da fila tornam-se O(1).
A lista sequencial seria menos adequada devido ao problema de deslocamento de elementos nas remoções no início (O(n) ou à complexidade de implementar uma fila circular.

b) Pilha

Para implementar uma pilha, escolheria a lista sequencial como base, pois ela segue o princípio LIFO (Last-In-First-Out), onde todas as operações (inserção e remoção) ocorrem no mesmo lado (topo).
Com uma lista sequencial, tanto o push (inserção) quanto o pop (remoção) podem ser implementados em O(1) mantendo um índice que aponta para o topo. A localidade de referência dos arrays também melhora o desempenho devido ao cache.
A lista ligada também funcionaria com complexidade O(1), mas a simplicidade e eficiência de cache da lista sequencial a tornam mais adequada para este propósito.

6.

a) Para pilha

Esta implementação seria adequada, porém não iria trazer vantagens significativas em relação a uma lista ligada simples.
Isso se deve por conta que uma pilha requer apenas operações em um extremo (topo). Com uma lista duplamente ligada circular mantendo apenas ponteiro para o primeiro nó:

- **Push (inserção):** Inserir no início → O(1) (mesmo que lista simples)
- **Pop (remoção):** Remover do início → O(1) (mesmo que lista simples)

Enquanto a complexidade de inserção e remoção se mantém a mesma que na lista simples, esta implementação acaba nos trazendo algumas desvantagens.
A primeira seria um overhead maior de memória (pois agora iriamos ter dois ponteiros por nó em vez de um), além da circularidade não ser aproveitada em operações de pilha
Logo a conclusão é que essa implementação para pilha até que funciona, porém é "exagero", com uma lista simples sendo mais eficiente para pilha.

b) Para fila

Esta implementação seria adequada e traria vantagens (ao contrário do caso anterior) em relação à lista ligada simples.
Isso se deve por conta que uma fila requer operações em dois extremos (inserção no fim, remoção no início). E com a lista duplamente ligada circular:

- **inserir no fim:** O(1) - conseguimos acessar diretamente o último elemento através do ponteiro primeiro→anterior
- **remover do início:** O(1) - temos acesso direto ao primeiro elemento
- **Acesso ao final:** Imediato através da dupla ligação circular

Logo teríamos algumas vantagens em usar lista duplamente ligada em detrimento de lista simples para implementação de uma fila, sendo elas a inserção no fim sendo O(1) sem necessitar de ponteiro adicional para o último elemento ele, e também um melhor aproveitamento da estrutura circular.
Assim, a conclusão é que essa implementação seria altamente adequada, aproveitando bem as características da estrutura para otimizar as operações de fila.

Exercícios Práticos

1.
Para lista sequencial dinâmica:

```
Elemento valor_indice(ListaSequencial *lista, int indice) {
    if (lista == NULL) return -1;
    // verificamos se o indice é um valor valido (maior ou igual a zero)
    // e verificamos se ele esta dentro do limite do tamanho da lista
```

```
   if (indice >= 0 && indice < lista->livre) return lista->a[indice];

    // supondo que -1 não seja um valor válido na lista, retornamos ele
   return -1; // para indicar um índice invalido.

}
```

Para lista ligada:

```
Elemento valor_indice(ListaLigada * lista, int indice) {
    if (lista == NULL) return -1;
    if (indice < 0)  return -1; // novamente supondo que -1 n seja um valor
válido                          // na lista, retornamos ele para indicar
índice inválido

    No *p = lista->primeiro;
    int i = 0;

    while (p != NULL && i < indice) {
        p = p->proximo;
        i++;
    }

    if (p != NULL) return p->valor;
    return -1; // indice fora dos limites
}
```

2. Pessoal essa aqui é meio extensa e acabei não conseguindo fazer ela por falta de tempo, perdão! :(

3.
a)

```
ListaLigada * junta_listas_1(ListaLigada *l1, ListaLigada *l2) {
if (l1 == NULL && l2 == NULL) return cria_lista();
if (l1 == NULL) return l2;
if (l2 == NULL) return l1;
 // criamos uma nova lista vazia para armazenar o resultado
 ListaLigada *resultado = cria_lista();
 No *p;

 // percorremos a primeira lista e copiamos todos os elementos
 p = l1->primeiro;
 while (p != NULL) {
     // inserimos cada elemento no final da nova lista
```

```
    insere(resultado, p->valor, tamanho(resultado));
    p = p->proximo;
 }

 // percorremos a segunda lista e copiamos todos os elementos
 p = l2->primeiro;
 while (p != NULL) {
    // inserimos cada elemento no final da nova lista
    insere(resultado, p->valor, tamanho(resultado));
    p = p->proximo;
 }

 // retornamos a nova lista com todos os elementos copiados
 return resultado;
}
```

b)

```
  ListaLigada * junta_listas_2(ListaLigada *l1, ListaLigada *l2) {
  if (l1 == NULL) return l2;
  if (l2 == NULL) return l1;
  if (l1 == l2) return l1

   // verificamos se a primeira lista esta vazia
   if (l1->primeiro == NULL) {
   //se estiver vazia, simplesmente apontamos para o inicio da segunda lista
       l1->primeiro = l2->primeiro;
   } else {
       // se nao estiver vazia, encontramos o ultimo no da primeira lista
       No *ultimo = l1->primeiro;
       while (ultimo->proximo != NULL) {
           ultimo = ultimo->proximo;
       }
       // conectamos o ultimo no da primeira lista com o primeiro da segunda
       ultimo->proximo = l2->primeiro;
   }

   // limpamos a segunda lista para evitar acesso duplo aos mesmos nos
   l2->primeiro = NULL;

   // retornamos a primeira lista modificada
   return l1;
}
```

c)
junta_listas_1:
A grande vantagem é que as listas originais são preservadas intactas, porém temos a desvantagem do consumo de memória e tempo para copiar todos os elementos. No geral, só faremos uso quando precisamos manter as listas originais para uso posterior.

junta_listas_2:
A grande vantagem é a eficiência (operação O(n)) onde n é o tamanho da primeira lista. A desvantagem é que a segunda lista fica vazia após a operação. É ideal quando a segunda lista não será mais usada separadamente.

d) Sim! Podemos adicionar um ponteiro para o último nó na nossa estrutura:

```
// primeiro modificamos a estrutura da lista para incluir ponteiro para o
último nó
typedef struct {
    No *primeiro;
    No *ultimo;    // adicionamos este ponteiro para o ultimo no
} ListaLigada;

// agora implementamos a versao otimizada de junta_listas_2
ListaLigada * junta_listas_2_otimizada(ListaLigada *l1, ListaLigada *l2) {
    if (l1 == NULL) return l2;
    if (l2 == NULL) return l1;

    // verificamos se a primeira lista esta vazia
    if (l1->primeiro == NULL) {
        // se estiver vazia, copiamos toda a estrutura da segunda lista
        l1->primeiro = l2->primeiro;
        l1->ultimo = l2->ultimo;
    } else {
        // conectamos o ultimo no da primeira lista com o primeiro da segunda
        l1->ultimo->proximo = l2->primeiro;
 // atualizamos o ponteiro do ultimo para apontar para o ultimo da segunda
lista
        l1->ultimo = l2->ultimo;
    }

    // limpamos a segunda lista para evitar acesso duplo
    l2->primeiro = NULL;
    l2->ultimo = NULL;

    // retornamos a primeira lista modificada
    return l1;
}
```

e)

```
ListaLigada * divide_lista(ListaLigada *lista) {
    if (lista == NULL) return cria_lista();
    // calculamos o tamanho total da lista
    int total = tamanho(lista);

    // verificamos se a lista é muito pequena para dividir
    if (total <= 1) return cria_lista();
        // retornamos uma lista vazia caso não der para dividir

    // calculamos onde fazer a divisao
    int metade = total / 2;

    // se o total for impar, a primeira parte tem um elemento a mais
    if (total % 2 == 1)  metade++;

    // criamos uma nova lista para a segunda parte
    ListaLigada *segunda_parte = cria_lista();
    No *p = lista->primeiro;
    No *ant = NULL; // ponteiro para o no anterior

    // percorremos ate o ponto de divisao
    for (int i = 0; i < metade - 1; i++) {
        ant = p;
        p = p->proximo;
    }

    // definimos o inicio da segunda parte
    if (p != NULL) {
        segunda_parte->primeiro = p->proximo;
        // desconectamos as listas
        p->proximo = NULL;
    }

    // retornamos a segunda parte
    return segunda_parte;
}
```

f)

```
void processa_lista(ListaLigada *lista, Elemento max) {
    if (lista == NULL) return;

    No *p = lista->primeiro;
```

```
    No *ant = NULL;

    while (p != NULL) {
        if (p->valor > max) {
            Elemento resto = p->valor;
            No *prox = p->proximo;

            // remove p da lista
            if (ant == NULL) lista->primeiro = prox;
            else ant->proximo = prox;
            free(p);

            // inser_pos aponta para o no depois do qual vamos inserir as
parcelas
            // (NULL significa que vamos inserir no início)
            No *inser_pos = ant;

            // inserimos parcelas na ordem correta
            while (resto > 0) {
                Elemento parcela = (resto > max) ? max : resto;
                No *novo = (No *) malloc(sizeof(No));
                novo->valor = parcela;
                novo->proximo = NULL;

                if (inser_pos == NULL) {
                    // inserir no inicio
                    novo->proximo = lista->primeiro;
                    lista->primeiro = novo;
                    inser_pos = novo;
                } else {
                    // inserir depois de inser_pos
                    novo->proximo = inser_pos->proximo;
                    inser_pos->proximo = novo;
                    inser_pos = novo;
                }
                resto -= parcela;
            }

            // ant deve apontar para o ultimo no inserido
            ant = inser_pos;
            // continuamos a partir do proximo original
            p = prox;
        } else {
            ant = p;
            p = p->proximo;
        }
```

```
    }
}
```

g)

```
No * no_indice(ListaLigada *lista, int indice) {
    if (lista == NULL) return NULL;
    // verificamos se o indice é invalido
    if (indice < 0) return NULL;

    // comecamos do primeiro no
    No *p = lista->primeiro;
    int i = 0;

    // percorremos a lista ata chegar no indice desejado
    while (p != NULL && i < indice) {
        p = p->proximo;
        i++;
    }
    // retornamos o no na posicao do indice (ou NULL se nao existir)
    return p;
}
```

h)

```
Boolean insere_com_no_indice(ListaLigada *lista, Elemento e, int indice) {
    if ( lista == NULL) return FALSE;
    // verificamos se o indice é valido
    if (indice < 0 || indice > tamanho(lista))  return FALSE;

    No *novo = (No *)malloc(sizeof(No));   // criamos um novo no
    novo->valor = e;

    // verificamos se é insercao no inicio
    if (indice == 0) {
        novo->proximo = lista->primeiro;
        lista->primeiro = novo;
    } else {
        // encontramos o no anterior à posicao de insercao
        No *ant = no_indice(lista, indice-1);
        // inserimos apos o no anterior
        novo->proximo = ant->proximo;
        ant->proximo = novo;
    }
```

```
    return TRUE;
}
```

i)

```
Elemento valor_indice_com_no_indice(ListaLigada *lista, int indice) {
    if (lista == NULL) return -1;
    // usamos a funcao no_indice para encontrar o no
    No *no = no_indice(lista, indice);

    // se encontramos o no, simplesmente retornamos seu valor
    if (no != NULL) return no->valor;

    // se nao encontramos, simplesmente retornamos um valor invalido
    return -1; // (novamente supondo que -1 não seja um valor valido na lista)
}
```

j)

```
No * busca_no_anterior(ListaLigada *lista, Elemento e) {
    if (lista == NULL) return NULL;
    No *ant = NULL; // ponteiro para o no anterior
    No *atual = lista->primeiro;

    // percorremos a lista procurando pelo elemento
    while (atual != NULL && atual->valor != e) {
        ant = atual;
        atual = atual->proximo;
    }

    // verificamos se o elemento nao foi encontrado
    if (atual == NULL) {
        // elemento nao encontrado, encontramos o ultimo no
        No *ultimo = lista->primeiro;
        while(ultimo!=NULL && ultimo->proximo != NULL) ultimo=ultimo->proximo;
        // retornamos o ultimo no
        return ultimo;
    }

    // retornamos o no anterior ao elemento encontrado
    return ant;
}
```

k)

```
int busca_com_anterior(ListaLigada *lista, Elemento e) {
    if (lista == NULL) return -1;
    // usamos a funcao busca_no_anterior para encontrar o no anterior ao
elemento
    No *ant = busca_no_anterior(lista, e);
    No *atual;

    // determinamos qual é o no atual (que contem o elemento)
    // se anterior é NULL, o elemento procurado é o primeiro
    if (ant == NULL) atual = lista->primeiro;
    else atual = ant->proximo;

    // verificamos se encontramos o elemento
    if (atual != NULL && atual->valor == e) {
        // calculamos o indice do elemento encontrado
        int indice = 0;
        No *p = lista->primeiro;

        // percorremos ate encontramos o no atual para saber seu indice
        while (p != atual) {
            p = p->proximo;
            indice++;
        }
        return indice; // retornamos o indice encontrado
    }

    return -1; // se nao encontramos o elemento, retornamos -1
}
```

l)

```
Boolean remove_elemento_com_anterior(ListaLigada *lista, Elemento e) {
    if (lista == NULL) return FALSE;
    // usamos a funcao busca_no_anterior para encontrar o no anterior ao
elemento
    No *ant = busca_no_anterior(lista, e);
    No *remover;

    // verificamos se o elemento é o primeiro da lista
    if (ant == NULL) {
        // pegamos o primeiro no como candidato a remocao
        remover = lista->primeiro;
```

```
        // verificamos se encontramos o elemento e ele é realmente o primeiro
        if (remover != NULL && remover->valor == e) {
            // atualizamos o ponteiro do primeiro para pular o no removido
            lista->primeiro = remover->proximo;

            // liberamos a memoria do no removido
            free(remover);

            return TRUE; // retornamos que tudo deu certo
        }
    } else {
        // o elemento esta no meio ou final da lista
        // pegamos o no apos o anterior (que deve ser o que queremos remover)
        remover = ant->proximo;

        // verificamos se encontramos o elemento
        if (remover != NULL && remover->valor == e) {
            // conectamos o no anterior com o proximo do no removido
            ant->proximo = remover->proximo;

            // liberamos a memoria do no removido
            free(remover);

            return TRUE; // retornamos que tudo deu certo
        }
    }
    return FALSE; // se chegamos aqui, entao o elemento nao foi encontrado
}
```

m)

a nova versão de remove_elemento que usa busca_no_anterior
não melhora a eficiência assintótica em relação à versão original.

análise da complexidade:

- a função busca_no_anterior tem complexidade O(n) porque precisa percorrer a lista
- a operação de remoção em si tem complexidade O(1) após encontrar o nó
- complexidade total: O(n) + O(1) = O(n)

comparação com a versão original:

- ambas as versões têm complexidade O(n)
- a versão original tbm percorre a lista para encontrar o elemento
- a diferença está apenas na organização do código, não na performance

vantagens da nova versão:

- código mais modular e reutilizável
- separação clara das responsabilidades
- a função busca_no_anterior pode ser usada em outras operações

Resumindo tudo, o veredito é que a refatoração melhora a organização e manutenibilidade do código, mas não altera a complexidade algorítmica da operação de remoção, pois a operação dominante continua sendo a busca linear pelo elemento na lista.

4.
A implementação funciona corretamente mas é extremamente ineficiente. Para uma pilha, ambas as operações principais de push (inserção) e pop (remover) deveriam ser O(1), mas nesta implementação são O(n), tornando-a inadequada para uso em aplicações reais com muitos elementos.
O problema está justamente em usar inserção/remoção no início da lista sequencial, que requer deslocamento de todos os elementos (O(n)). O correto seria implementar pilha com operações no final do array usando um índice "topo".

5.

```
ListaLigada * inverte_lista(ListaLigada *lista) {
    // criamos uma nova lista vazia para armazenar o resultado invertido
    ListaLigada *lista_invertida = cria_lista();

    // começamos pelo primeiro no da lista original
    No *atual = lista->primeiro;

    // percorremos todos os nos da nossa lista original
    while (atual != NULL) {
        // inserimos o valor do no atual no inicio da nova lista
        // isso faz com que o primeiro elemento vire o ultimo, e vice-versa
        insere(lista_invertida, atual->valor, 0);
        atual = atual->proximo; // avancamos para o poximo no da lista
original
    }

    // retornamos a nova lista com os elementos na ordem invertida
    return lista_invertida;
}
```