Capítulo 3

Listas

Objetivos

- Existem certas estruturas clássicas que se comportam como padrões uma vez que são utilizadas na prática em diversos domínios de aplicação. Neste capítulo é apresentada a estrutura de dados Lista e o correspondente Tipo Abstrato, detalhando a sua interface e apresentando duas implementações: vetores e ponteiros. Variantes do tipo abstrato listas, na forma de Pilhas e Filas, de ampla utilização prática, são descritas.

Introdução

Um conjunto de elementos pode ser intuitivamente representado através de uma lista linear. Listas são estruturas extremamente flexíveis que possibilitam uma ampla manipulação das informações uma vez que inserções e remoções podem acontecer em qualquer posição.

Uma lista pode ser definida como uma estrutura linear[12], finita cuja ordem é dada a partir da inserção dos seus elementos componentes. As listas são estruturas compostas, constituídas por dados de forma a preservar a relação de ordem linear entre eles. Cada elemento na lista pode ser um dado primitivo ou arbitrariamente complexo.

Em geral, uma lista segue a forma a_1, a_2, a_3, ..., a_n, onde n determina o tamanho da lista. Quando $n = 0$ a lista é chamada *nula* ou *vazia*. Para toda lista, exceto a nula, a_{i+1} segue (ou sucede) a_i ($i < n$), e a_{i-1} precede a_i ($i > 1$). O primeiro elemento da lista é a_1, e o último a_n. A posição correspondente ao elemento a_i na lista é i. A lista pode ser representada visualizando-se um vetor, por exemplo.

As características básicas da estrutura de dados lista são as seguintes:

- Homogênea. Todos os elementos da lista são do mesmo tipo.
- A ordem nos elementos é decorrente da sua estrutura linear, no entanto os elementos não estão ordenados pelo seu conteúdo, mas pela posição ocupada a partir da sua inserção.

[12] Uma estrutura é dita de linear uma vez que seus elementos componentes se encontram organizados de forma que todos, a exceção do primeiro e último, possuem um elemento anterior e um posterior, somente.

- Para cada elemento existe anterior e seguinte, exceto o *primeiro*, que não possui anterior, e o *último*, que não possui seguinte.
- É possível acessar e consultar qualquer elemento na lista.
- É possível inserir e remover elementos em qualquer posição.

1. Definição do TAD Lista

Como apresentado na parte anterior, a definição de um Tipo Abstrato de Dados (TAD) envolve a especificação da interface de acesso para a manipulação adequada da estrutura, a partir da qual são definidas em detalhe, as operações permitidas e os parâmetros requeridos. O conjunto de operações depende fortemente das características de cada aplicação, no entanto, é possível definir um conjunto de operações mínimo, necessário e comum a todas as aplicações.

Interface do TAD Lista

```
// Cria uma lista vazia
Lista Criar ()

//insere numa dada posição na lista.
int Inserir (Lista l, tipo_base  dado, corrente pos)

// Retorna o elemento de uma dada posição
tipo_base  consultaElemento (Lista l, corrente pos);

// Remove o elemento de uma determinada posição
int Remover (Lista l, corrente pos);

// Retorna 1 a lista está vazia, ou 0 em caso contrario.
int Vazia (Lista l);

// Retorna 1 se a lista está cheia, ou 0 em caso contrario.
int Cheia (Lista l);

// Retorna a quantidade de elementos na lista.
int Tamanho (Lista l);

// Retorna o próximo elemento na lista a partir da posição corrente.
corrente proximoElemento (Lista l, corrente pos);
```

```
/*Busca por um determinado elemento e retorna sua posição corrente, ou -1 caso não seja encontrado.*/
corrente Busca (Lista l, tipo_base  dado);
```

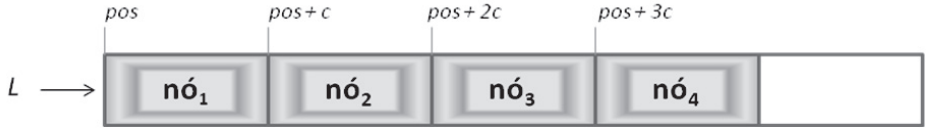
Uma vez definida a interface, esta pode ser implementada utilizando uma representação dos dados adequada. Existindo mais de uma estrutura adequada, a escolha depende principalmente das necessidades e características dos dados a serem manipulados pela aplicação.

A seguir, o Tipo Abstrato Lista é implementado utilizando duas estruturas de dados comumente utilizadas e adequadas às necessidades, cada uma com vantagens e desvantagens particulares.

1.1. Implementação do TAD Lista usando alocação estática

Na implementação de lista adotando alocação de memória estática os elementos componentes são organizados em posições contíguas de memória utilizando *arranjos* ou vetores.

Vantagens e desvantagens desta estrutura foram discutidas na parte 3. Em particular, a utilização de vetores se torna adequada no caso em que existe uma clara noção do tamanho da entrada a ser processada, e uma perspectiva que indica que as aplicações que irão utilizar o TAD não estarão executando muitas operações de inserção e remoção que possam vir a alterar significativamente o tamanho preestabelecido. A estruturação da lista utilizando alocação estática é apresentada graficamente a seguir. Note que, a partir do endereço correspondente ao primeiro elemento no vetor (*pos*), e conhecendo o tamanho (*c*) de cada componente na lista, é possível calcular o endereço na memória de qualquer elemento armazenado no vetor. Isso garante o acesso direto aos elemento em *O(1)*.



A seguir é apresentada a definição da estrutura de dados e implementação[13] das operações definidas na interface utilizando alocação estática de memória através da definição de vetores. Considerando a utilização de alocação estática de memória, o tamanho da estrutura de dados precisa ser determinado em tempo de compilação.

[13] A notação utilizada na implementação é próxima à linguagem C.

```
#define tamanho
```

Como o protótipo da lista é definido de modo a ser implementado usando vetor ou outro recurso, torna-se necessário definir um tipo que possa ser utilizado como elemento que é acessado nas operações (ou corrente). No caso desta implementação inicial, utilizando vetor, os elementos são acessados através de suas posições no vetor, sendo que estas posições são representadas como interiores que iniciam em 0 (zero) na primeira posição e seguem até n-1 (tamanho do vetor - 1). Portanto, torna-se necessário definir o tipo corrente como inteiro.

```
typedef int corrente;
```

Uma lista é um tipo de dado que estrutura elementos cujo tipo pode ser arbitrariamente complexo, envolvendo inclusive, a utilização de outros TADs. A definição a seguir especifica o tipo base dos elementos da lista como inteiros.

```
typedef int tipo_base;
```

Os elementos da lista são organizados em um vetor, de tamanho predefinido. Adicionalmente, um atributo contendo a quantidade de elementos da lista (*quant_Elem*) é incluído na estrutura no intuito de tornar mais fácil e ágil o acesso aos elementos e possibilitar o controle do crescimento da estrutura.

```
typedef struct {
        tipo_base v[tamanho];
        int quant_Elem;
} no_Lista;
```

A informação relativa à quantidade de elementos existente na lista pode ser útil em diversas situações uma vez que, conhecendo a posição na memória (endereço) do primeiro elemento da lista, é possível calcular o endereço do último elemento e, consequentemente, da primeira posição disponível. Isto é possível por conta da propriedade de armazenamento contíguo propiciada pela estrutura de dados.

Na linguagem C, a posição que corresponde ao primeiro elemento do vetor corresponde ao índice i = 0. Em outras linguagens, como por exemplo Pascal, o primeiro elemento no vetor se encontra na posição i=1.

Finalmente a lista é definida como um ponteiro à estrutura de dados onde os elementos são agregados.

```
typedef no_Lista *Lista;
```

A criação de uma lista inicialmente vazia envolve a definição do ponteiro correspondente, apontando a um endereço de memória reservado com tamanho adequado para o armazenamento da estrutura, em particular, o primeiro nó representando a cabeça da lista. O tamanho da lista é inicializado em zero uma vez que inicialmente não contém elementos.

```
Lista Criar () {
        Lista l = (Lista) malloc (sizeof (no_Lista));
        if (l != NULL){
```

```
                l -> quant_Elem = 0;
        return (l);
        }
        else printf ("Não existe memória suficiente");
        return;
}
```

Normalmente, a lista precisa ser percorrida de forma a realizar algum tipo de processamento sobre os dados que a compõem. Consequentemente se torna necessário um mecanismo que possibilite checar no momento em que não seja possível processar mais nenhum elemento. O método *ultimoElemento* é responsável por fazer esta checagem.

```
int ultimoElemento (Lista l, corrente pos) {
        if (pos + 1 == l -> quant_elem) return (1)
        else return (0);
}
```

A execução de uma operação de remoção requer a existencia de no mínimo um elemento na lista. A função *Vazia* é utilizada para informar se há elementos no vetor, retorna o valor 1 no caso da lista se encontrar vazia, e 0 em caso contrário.

```
int Vazia (Lista l) {
        if (l -> quant_Elem == 0) return (1)
        else return (0);
}
```

Outra operação que pode ser de utilidade é a checagem pelo caso em que a estrutura que armazena a lista possa estar cheia, uma vez que esta situação pode inviabilizar a inserção de um novo elemento. Este método é de fundamental importância, principalmente no caso de utilização de alocação de memória estática onde a tentativa de inserção de um novo elemento pode acarretar o estouro da memória, fazendo com que o programa termine com erro.

```
int Cheia (Lista l) {
        if (l -> quant_Elem == tamanho) return (1)
        else return (0);
}
```

Uma função que pode ser definida para auxiliar a verificação de próximo elemento e a inserção é uma função vailadPos. Esta recebe a lista e a posição atual e verifica se a posição é maior ou igual a zero e se a posição é maior que a quantidade de elementos -1.

```
int validaPos(Lista l, corrente pos){
        if(pos >=0&&pos< ((l-> quant_Elem) -1)){
                return (1);
        }else{
                return (0);
        }
}
```

Adicionalmente, o percurso ao longo dos elementos de uma lista requer de uma operação que possibilite a movimentação ao longo da estrutura, elemento a elemento. A função *proximoElemento* retorna o índice no vetor correspondente à posição do próximo elemento, se for o último e não tiver próximo, retorna -1.

```
corrente proximoElemento (Lista l, corrente pos) {
        pos = pos++;
                if (validaPos(l, pos)
                        return (pos);
                return (-1);
}
```

A operação de inserção é possivelmente uma das mais importantes, uma vez que é através dela que a lista será construída. Em particular, o tipo lista não possui nenhuma restrição em relação à inserção, podendo acontecer em qualquer posição. Desta forma, na hora de inserir um elemento na lista, é necessário informar qual a posição correspondente ao novo elemento, seja no inicio, meio ou fim da lista. Considerando que a inserção nem sempre é possível por conta da limitação de espaço da estrutura de dados utilizada, a função retorna 1 se a inserção foi realizada com sucesso e 0 em caso contrário

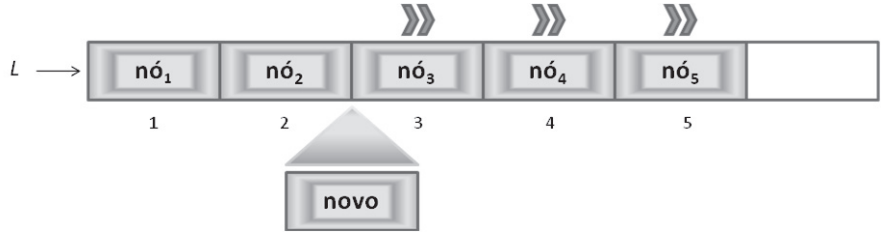
Algoritmo 0

```
int Inserir (Lista l, tipo_base  dado, corrente pos) {
        int i;
        if (!validaPos(l pos) || (cheia (l))) return (0);

        for (i = l->) quant_elem ; i > = pos; i--){
        l -> v[i] = l -> v[i-1];
        }
        l -> v[pos] = dado;
        l -> quant_elem = (l -> quant_elem)+1;

        return (1);
}
```

Dependendo da posição onde o elemento será inserido, o trabalho requerido pode ser estimado de forma a estabelecer o custo da função. Pelo fato de se utilizar alocação estática e contígua de memória para o armazenamento dos elementos da lista, a inserção de um elemento em uma execução requer que o espaço físico na memória seja providenciado em tempo de compilação. Para isso é necessário deslocar (*shift*) todos os elementos necessários, desde a posição requerida até o final da lista. Por exemplo, se a lista possui 5 elementos e o novo elemento precisa ser inserido na posição 3, os últimos 3 elementos precisarão ser deslocados à direita, para que a posição de inserção requerida fique disponível para o novo elemento a ser inserido. A situação é ilustrada na figura a seguir.



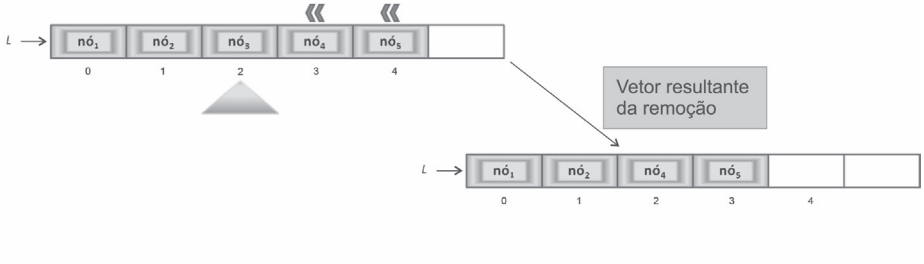
O pior cenário neste caso acontece quando é requerida a inserção na primeira posição da lista, obrigando o deslocamento de todos os elementos da lista em uma posição à direita. Nete caso, o custo requerido é *O(n)*.

Analogamente à operação de inserção, a operação de remoção exclui um elemento em qualquer posição, portanto esta posição precisa ser informada explicitamente. O algoritmo é responsável por checar se a posição informada representa uma posição válida dentro da estrutura.

Algoritmo 0.1

```
int Remover (Lista l, corrente pos) {
        int i;
        if (vazia(l) || (!validaPos(l)) return (0);

        int dado = l -> v[pos];
        for (i = pos + 1 ; i < l -> quant_elem; i++)
                l -> v[i-1] = l -> v[i];
          l -> quant_elem = (l -> quant_elem)-1;
        return (1);
}
```



Realizando uma análise análoga ao caso da inserção, o custo para a remoção é *O(n)*.

No caso em que seja necessário remover um elemento de acordo com algum conteúdo específico, o elemento em questão precisa ser previamente localizado através da operação de *Busca*. A partir da posição retornada pela busca, caso o elemento seja efetivamente encontrado na estrutura, a remoção poderá ser efetivamente realizada.

A busca por um determinado elemento pode ser originada de duas formas. Na primeira variante, a busca pode ser orientada por um determinado conteúdo, retornando como resultado a sua posição na lista, no caso em que o elemento for efetivamente encontrado, caso contrário retorna -1.

```
corrente Busca (Lista l, tipo_base  dado) {
        int i;
        for (i = 0; i <= tamanho(l); i++)
        if (l -> v[i] == dado) return (i);
        return (-1);
}
```

O pior caso possível para esta busca, consiste na situação onde o elemento procurado não se encontra na lista. Nesta situação a lista será percorrida na totalidade, sem sucesso, passando pelos *n* elementos que determinam seu tamanho. Consequentemente, o custo desta operação no seu pior caso é *O(n),* ou seja linear.

Na segunda variante, a busca pode acontecer a partir de uma determinada posição na lista, retornando o elemento contido nessa posição.

```
tipo_base  Consulta (Lista l, corrente pos) {
return (l -> v[pos-1]);
}
```

A complexidade da busca neste caso é *O(1)* uma vez que consiste no acesso direto à posição correspondente, demandando para isso tempo constante.

1. Considerando a implementação do TAD utilizando alocação estática de memória resolva as questões a seguir:

a) Explique por que o custo da remoção em uma lista implementada utilizando alocação estática e contígua de memória é *O(n)*.

b) Implemente a operação que retorna a quantidade de elementos na lista, cujo cabeçalho é: int tamanho (lista l). Determine a complexidade da operação implementada.

c) Implemente o método auxiliar que verifique se uma determinada posição é válida, isto é, se encontra dentro dos limites do vetor. O cabeçalho da operação é int validaPos (corrente pos).

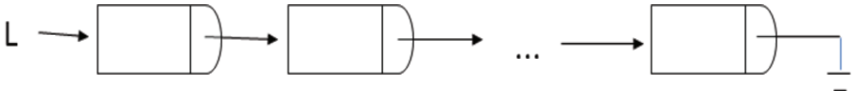
1.2. Implementação do TAD Lista usando alocação dinâmica

Na implementação do Tipo Abstrato lista adotando alocação de memória dinâmica, a alocação de memória é gerenciada sob demanda em tempo de execução. Esta característica determina que os elementos componentes são organizados em posições não-contíguas, ficando espalhados ao longo da memória. Consequentemente, não é preciso estimar *a priori* o tamanho da estrutura uma vez que o espaço é alocado na medida da necessidade, dependendo das operações de inserção e remoção realizadas.

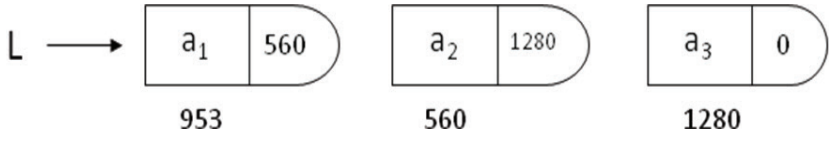
Vantagens e desvantagens desta estrutura foram discutidas na Parte

3. Em particular, esta estrutura se torna adequada quando o tamanho da estrutura é desconhecido e pode variar de forma imprevisível. No entanto, a gerência da memória torna a implementação mais trabalhosa e propensa a erros, podendo acarretar em perda de informação. Adicionalmente, o acesso aos dados é seqüencial, no sentido que para acessar o elemento na posição *m*, se torna necessário percorrer os *m - 1* elementos anteriores. Com isso, no pior caso, a busca por um elemento na lista demanda custo *O(n).*

A seguir é apresentada a definição da estrutura de dados e implementação das operações definidas na interface para o TAD Lista utilizando alocação dinâmica de memória através de ponteiros. A notação utilizada na implementação é próxima à linguagem C. Esta estrutura representa uma seqüência de elementos encadeados por ponteiros, ou seja, cada elemento deve conter, além do dado propriamente dito, uma referência para o próximo elemento da lista. Graficamente, a estrutura de dados para a implementação de listas utilizando ponteiros é a seguinte:



Por exemplo, uma lista com valores de ponteiros a endereços reais tem a seguinte forma:



O espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenados: para cada novo elemento inserido na estrutura é alocado um espaço de memória para armazená-lo. Consequentemente, não é possível garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, e, portanto não é possível ter acesso direto aos elementos da lista. Isto implica que, para percorrer todos os elementos da lista, devemos explicitamente seguir o encadeamento dos elementos. Para isto, é preciso definir a estrutura do nó, como uma estrutura auto-referenciada contendo, além do conteúdo de informação, um ponteiro ao próximo elemento na sequência. As definições a seguir implementam a estrutura correspondente.

```
typedef struct node *no_ptr;
```

```
struct node {
        tipo_base  elemento;
        no_ptr prox;
};
```

Finalmente a lista é definida como um ponteiro ao primeiro nó da lista, a partir do qual a sequência de nós é encadeada.

```
typedef no_PTR Lista;
```

Uma lista é um tipo de dado que estrutura elementos cujo tipo pode ser arbitrariamente complexo, envolvendo inclusive, a utilização de outros TADs. A definição a seguir especifica o tipo base dos elementos da lista como inteiros.

```
typedef int tipo_base;
```

A implementação de listas com ponteiros requer um cuidado especial uma vez que qualquer erro na manipulação dos ponteiros pode acarretar em perda parcial ou total da lista de elementos. Assim sendo, a utilização de um ponteiro auxiliar para o percurso ao longo da lista pode ser de grande utilidade. Com esse objetivo definimos um tipo *Corrente*, a ser utilizado como cópia da lista de forma a possibilitar a sua manipulação com segurança.

```
typedef no_ptr Corrente;
```

A função que *cria* uma lista vazia utilizando alocação dinâmica de memória é apresentada a seguir. A função tem como valor de retorno a lista vazia inicializada, isto é, o valor de retorno é *NULL,* pois não existem elementos na lista.

```
Lista Criar (){
  return (NULL);
}
```

O método Inicializar posiciona o índice da posição corrente no inicio da lista. Desta forma o ponteiro *pos* aponta para o mesmo local onde se encontra o primeiro elemento da lista. Este método é útil quando a lista precisa ser percorrida desde o inicio.

```
corrente Inicializar (Lista L){
        corrente pos = L;
        return (pos);
}
```

O deslocamento de um elemento para o seguinte na lista é dado pelo percurso ao longo dos ponteiros, onde, a partir do nó atual, a função a seguir retorna o ponteiro onde se localiza o próximo elemento na lista.

```
corrente proximoElemento (corrente p) {
        return (p -> prox);
}
```

A procura por um conteúdo na lista é realizada através da função *Busca*. Esta função percorre a lista desde o inicio, enquanto o elemento não for encontrado. A função retorna a posição do elemento na lista em caso de sucesso na procura, ou *NULL* se o elemento não for encontrado.

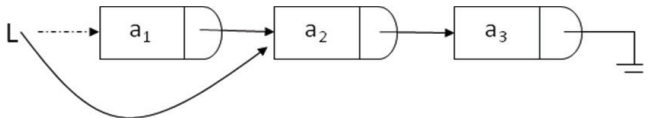
```
corrente Busca (Lista L, tipo_base  x) {
        corrente p = L;
        while ((p != NULL) && (p -> elemento != x))
        p = p -> prox;
  return p;
}
```

O acesso às informações contidas nos nós da lista é realizado através da função *Consulta*, que retorna o conteúdo de informação armazenado no nó referenciado pelo ponteiro corrente.

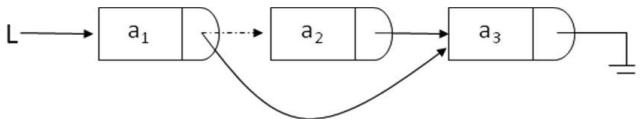
```
tipo_base Consulta (Lista L, corrente p){
        if (p != NULL) return (p -> elemento);
}
```

A remoção de um elemento da lista envolve a análise de duas situações: a remoção do primeiro nó da lista ou de um nó em outra posição qualquer, sem ser no inicio da lista. A seguir, o processo de remoção em cada caso é ilustrado.

No caso da remoção do primeiro elemento da lista é necessário que o ponteiro à lista seja atualizado, indicando o novo primeiro elemento.



No caso de remoção de um elemento, sem ser o primeiro, o processo consiste em fazer um *by pass* sobre esse elemento através do ajuste correto dos ponteiros, para posteriormente liberar a memória correspondente. Para efetivar a remoção é preciso o nó anterior ao nó a ser removido, que pode ser obtido a partir da função auxiliar *Anterior*.



A função *Remover* apresentada a seguir, remove o elemento correspondente a uma determinada posição *pos*, passada por parâmetro. Esta posição pode ser resultado de um processo de busca, a partir de um determinado conteúdo. Em ambos os casos é preciso liberar a memória correspondente ao nó removido.

A seguir é apresentado o algoritmo que implementa o processo de remoção.

Algoritmo 0.2

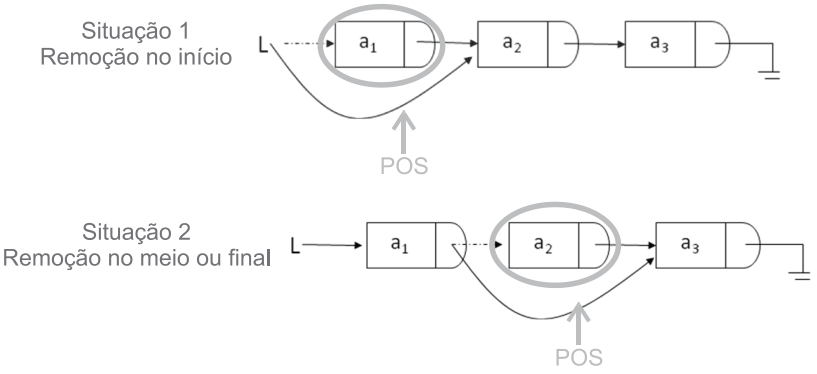
A função inerior retorna o nó anterior ao nó passado (pos). Ela percorre a toda a lista verificando se o próximo elemento é o elemento pos.

```
corrente Anterior(Lista l, corrente pos){
        corrente ant = null;

  if(pos!= l){
        corrente atual = l;
        while(atual != null & atual -> prox != pos){
              atual = atual -> prox;
        }
        ant = atual
}

return (1);
}
```

Na remoção encontramos duas situações: 1) quando o elemento a ser removido é a primeira posição (Nó anterior é null); e 2) quando o elemento a ser removido não é o elemento da primeira posição (Nó anterior não é null).




```
int Remover(Lista l, corrente pos){
        corrente noAnterior = Anterior (L,pos);

        corrente tmp_cell = pos;

        if(noAnterior == NULL){
                l = l -> prox;
        }else{
                noAnterior -> prox = pos -> prox;
        }

        free (tmp_cell);

        return (1);
}
```

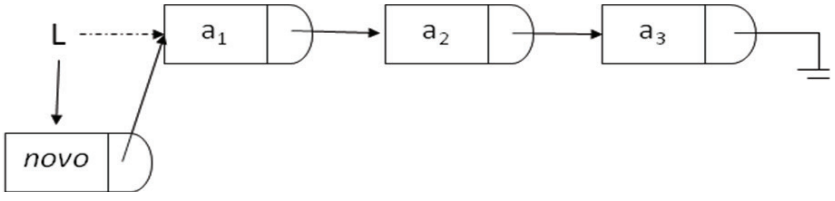
O algoritmo apresentado para a função remover utiliza uma função anterior. Esta função anterior tem o papel de retornar o elemento que antecede a posição atual. Tendo em vista que para o elemento atual ser removido, basta ligar o elemento anterior ao próximo. A seguir a função anterior é apresentada:

```
corrente Anterior(Lista l, corrente pos){;
        corrente ant = null;
```

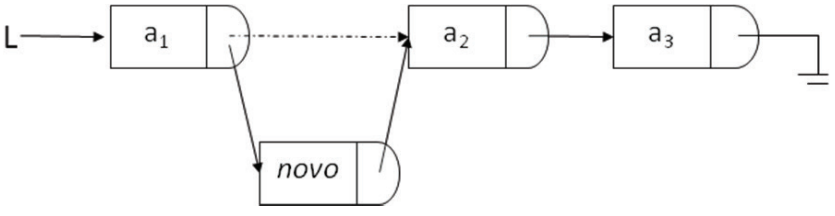
```
        if(pos != NULL){
         corrente atual = l;
        while(atual != null & atual ->prox !=pos){
                atual = atual -> prox;
        }
                ant = atual;

        return (ant);
}
```

A inserção em uma lista pode acontecer em qualquer posição, que pode ser no inicio, no final ou qualquer outra posição no meio da lista. O conteúdo do parâmetro *pos* representa a posição de inserção requerida para o elemento *novo* a ser inserido, no caso de inserção na cabeça da lista *pos é null*. Neste caso, o ponteiro L que apontava ao primeiro elemento da lista aponta agora para o novo elemento inserido.



Para qualquer outro valor de *pos*, o processo de inserção acontece como ilustrado a seguir. A lista precisa ser percorrida até a posição de inserção requerida. Nesse ponto, o novo elemento será inserido atualizando os ponteiros correspondentes.




```
int Inserir (Lista l, tipo_base  dado, corrente pos){
  int i;
  Corrente atual = Inicializar (l);
```

```
    Lista novo = (Lista) malloc(sizeof(struct node));
    novo -> elemento = dado;
    if (pos == NULL){
            novo -> prox = l;
            l = novo;
    }

    else {
      while (atual -> next != NULL) and (atual ->
                    next != pos)
                    atual = atual -> prox;
                    novo -> prox = atual -> prox;
                    atual -> prox = novo;
    }
    else return (1);
}
```

A função *Vazia* é utilizada para verificar se a lista possui ou não elementos armazenados. A verificação consiste em checar se o ponteiro ao primeiro elemento é *null*.

```
int Vazia (Lista L) {
        if (L == NULL) return (1)
        else return (0);
}
```

A função *ultimoElemento* é utilizada para verificar o final da lista. Esta checagem consiste em determinar se o próximo elemento que segue ao atual é *NULL*.

```
int ultimoElemento (corrente p) {
        if (p -> prox == NULL) return (1)
        return (0);
}
```

Com exceção de *Busca* e *Anterior* todas as operações consomem tempo *O(1)*. Isto por que somente um número fixo de instruções é executado sem

levar em conta o tamanho da lista. Para *Busca* e *Anterior* o custo é *O*(*n*), pois a lista inteira pode precisar ser percorrida se o elemento não se encontra ou for o último da lista.

Atividades de avaliação

Utilizando o TAD Lista definido nesta parte, implemente como aplicação um programa que, dadas duas listas A e B, crie uma terceira lista L intercalando os elementos das duas listas A e B.

```
Lista Intercala (Lista A , Lista B) {
  Lista L = cria ();
  corrente pos_L = Inicializar (L);
  corrente pos_A = Inicializar (A);
  corrente pos_B = Inicializar (B);

  /*Assumimos que A e A tem o mesmo tamanho
  while (not ultimoElemento(pos_A)) &&
      (not ultimoElemento (pos_B)){
    Inserir (L, Consulta (pos_A), null);
    pos_A = proximoElemento (pos_A);
    Inserir (L, Consulta (pos_B), null);
    pos_B = proximoElemento (pos_B);
  }
  return(L);
}
```

Atividades de avaliação

Considerando a implementação do TAD utilizando alocação dinâmica de memória resolva as questões a seguir:

1. Implemente a operação que retorna a quantidade de elementos na lista, cujo cabeçalho é: int Tamanho (Lista l). Determine a complexidade da operação implementada.

2. Implemente uma rotina para a remoção de uma lista desalocando a memória utilizada. O cabeçalho da rotina é void Remove_list (Lista L).
3. Implemente a rotina auxiliar chamada *anterior* usada na remoção, de acordo com o seguinte cabeçalho corrente anterior (Lista L, corrente pos). Esta rotina retorna a posição do elemento anterior a uma outra posição pos. Se o elemento não for encontrado retorna NULL.
4. Utilizando as operações definidas na interface do TAD Lista implemente um método que dadas duas listas L1 e L2, calcule L1 ∪ L2 (união) e L1 ∩ L2 (interseção). O resultado das operações deve ser retornado em uma terceira lista L3.
5. Utilizando as operações definidas na interface do TAD Lista implemente um método que dada uma lista retorne uma segunda lista onde os elementos pertencentes à primeira estejam ordenados em forma crescente. Determine a complexidade do seu algoritmo.
6. Escreva um programa que, utilizando o TAD Lista, faça o seguinte:

a) Crie quatro listas (L1, L2, L3 e L4);

b) Insira sequencialmente, na lista L1, 10 números inteiros obtidos de forma randômica (entre 0 e 99);

c) Idem para a lista L2;

d) Concatene as listas L1 e L2, armazenando o resultado na lista L3;

e) Armazene na lista L4 os elementos da lista L3 (na ordem inversa);

f) Exiba o conteúdo das listas L1, L2, L3 e L4.

Texto Complementar

Variações sobre o TAD Lista

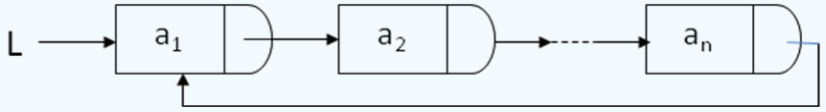
Lista ordenada

Uma lista ordenada é uma lista onde seus elementos componentes são organizados de acordo a um critério de ordenação com base em um campo chave. A ordem estabelecida determina que a inserção de um determinado elemento na lista irá a acontecer no lugar correto. A lista pode ser ordenada de forma crescente ou decrescente.
A partir da existência de um critério de ordenação na lista, a função responsável pela busca por um determinado conteúdo na lista pode ser adaptado de forma a tornar a busca mais eficiente.

Lista circular

A convenção consiste em manter a última célula apontando para a primeira. Desta forma, o teste por fim de lista nunca é satisfeito. Com isso, precisa ser estabelecido

um critério de parada de forma a evitar que o percurso na lista não encontre nunca o fim. Uma forma padrão é estabelecido com base no número de elementos existentes na lista.



Lista duplamente encadeada

Em alguns casos pode ser conveniente o percurso da lista de trás para frente através da adição de um atributo extra na estrutura de dados, contendo um ponteiro para a célula anterior. Esta mudança na estrutura física acarreta um custo extra no espaço requerido e também aumenta o trabalho requerido nas inserções e remoções, uma vez que existem mais ponteiros a serem ajustados. Por outro lado simplifica a remoção, pois não mais precisamos procurar a célula anterior *(O(n))*, uma vez que esta pode ser acessada diretamente através do ponteiro correspondente.