1 - Estruturas de Dados Dinâmicas

Um array (vetor) é uma estrutura de dados que mantém um conjunto de dados, considere como exemplo a representação da Figura 1, onde **v** é o nome (endereço) do array na memória.

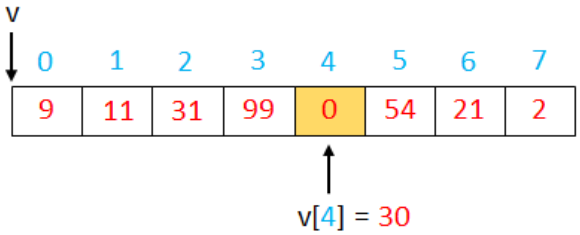

Figura 1 – Alocação e acesso a memória de um array.

Vantagem: no array qualquer posição pode ser recuperada independentemente das demais, veja que a quinta posição do array (v[4]) pode ser acessada independentemente das demais, tem-se apenas de fornecer o índice da posição no array.

Desvantagem: o array ocupa um espaço contíguo de memória e **não** permite ser redimensionado, ou seja, ele é uma estrutura de tamanho fixo que deve ser definida na sua criação. O array da Figura 1 sempre terá 8 elementos.

Ao contrário dos arrays, as estruturas de dados dinâmicas se redimensionam de acordo com os elementos armazenados, onde elas iniciam-se com zero elementos e novos elementos podem ser adicionados e removidos sem alterar a estrutura de armazenamento.

2 - Listas Encadeadas

Uma lista encadeada (linked list ou lista ligada) é um tipo de dado dinâmico, onde cada elemento é uma estrutura formada por 1 conteúdo e 1 endereço, assim como na representação da Figura 2. Os elementos são chamados de nós.

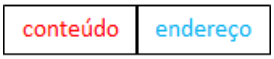

Figura 2 – Representação de 1 elemento ou nó da lista encadeada.

O conteúdo é um (atributo de um tipo de dado qualquer (primitivo ou objeto) já o endereço é uma variável que contém o endereço do próximo nó da lista, assim como no exemplo da Figura 3.

```
public class No {
    int conteudo;
    No proximo;
}
```

Figura 3 – Classe que representa os elementos da lista encadeada.

Lista é uma estrutura auto referenciada, pois o atributo proximo é um ponteiro para uma próxima estrutura do mesmo tipo.

A lista encadeada é formada pela sequência dos nós, assim como ilustrado na Figura 4, neste exemplo, tem-se uma lista encadeada para guardar os valores 9, 11, 31 e 99, obedecendo esta sequência. Veja que, cada elemento possui um endereço na memória RAM (Random Access Memory) do computador e esses endereços não são consecutivos, ou seja, o próximo elemento da sequência pode estar em qualquer posição da memória.

No exemplo da Figura 4, o primeiro nó da lista está no endereço e150 e os demais estão nos endereços e280, e070 e e210. O último nó da lista é aquele nó que possui o endereço null como próximo.

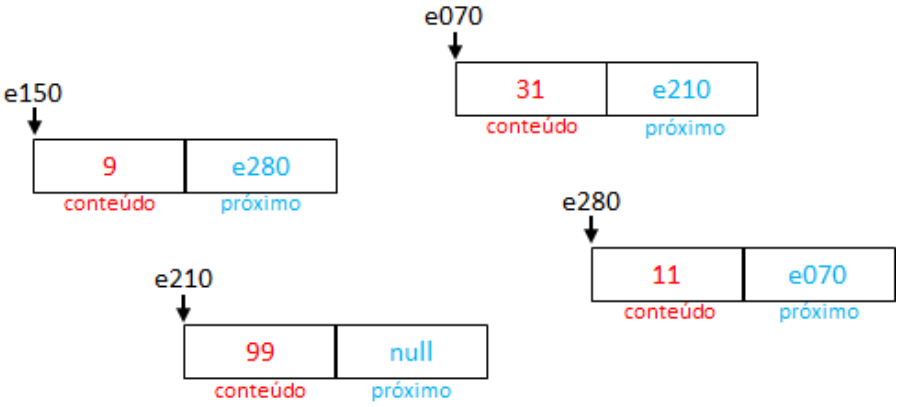

Figura 4 – Representação de uma lista encadeada na memória.

Somente os objetos da classe No não são capazes de formar a lista. É necessário ter uma estrutura (classe) para manter o endereço do primeiro nó da lista e fazer as operações na lista, tais como, inserir um novo nó e imprimir o conteúdo da lista. A Figura 5 mostra o diagrama UML das classes No e Lista.

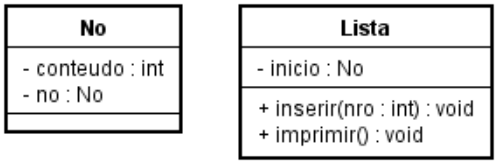

Figura 5 – Diagrama UML.

Na Figura 6 tem-se o código da classe Lista, veja que a classe mantém apenas o endereço do 1º nó da lista no atributo inicio, para inserir ou imprimir os elementos da lista é necessário começar pelo nó inicio e ir acessando o atributo proximo até encontrar o valor null, que significa o fim da lista.

A Figura 7 possui um exemplo de código para testar a classe Lista.

```
public class Lista {
   No inicio;

   Lista(){
      /* a lista está vazia */
      inicio = null;
   }

   void inserir(int nro){
      /* criar um nó */
      No no = new No();
      no.conteudo = nro;
      no.proximo = null; /* este será o último nó da lista */

      /* checa se a lista está vazia */
      if( inicio == null ){
         inicio = no;
      }
      else{
         /* percorrer a lista até encontrar o último nó */
         No ultimo = inicio;
         while( ultimo.proximo != null ){
            ultimo = ultimo.proximo;
         }
         /* alterar o próximo do último para o endereço do no */
         ultimo.proximo = no;
      }
   }

   void imprimir(){
      /* checa se a lista está vazia */
      if( inicio == null ){
         System.out.println("Lista vazia");
      }
```

```
public class Principal {
   public static void main(String[] args) {
      Lista lista = new Lista();
      lista.imprimir();
      lista.inserir(10);
      lista.imprimir();
      lista.inserir(20);
      lista.imprimir();
      lista.inserir(20);
      lista.imprimir();
      lista.inserir(40);
      lista.imprimir();
   }
}
```

Figura 7 – Código da classe Principal para testar a classe Lista.

```
        else{
            System.out.println(); /* quebra de linha na tela */
            /* percorrer a lista até encontrar o último nó */
            No ultimo = inicio;
            while( ultimo != null ){
                System.out.print( ultimo.conteudo +" ");
                ultimo = ultimo.proximo;
            }
        }
    }
}
```

Figura 6 – Código da classe Lista.

Na Figura 6, o método inserir insere novos elementos apenas no final da sequência, porém os elementos poderiam ser adicionados em qualquer posição. Para inserir um nó no meio da sequência é necessário somente alterar os endereços da variável próximo. Considere como exemplo a representação da Figura 8, na qual deseja-se inserir o nó e180 como próximo do nó atual. Para isso, tem-se de executar as instruções:

no.proximo = atual.próximo; /* para não perder a referência de quem é o próximo na sequência */

atual.proximo = no; /* para incluir o novo nó na sequência */

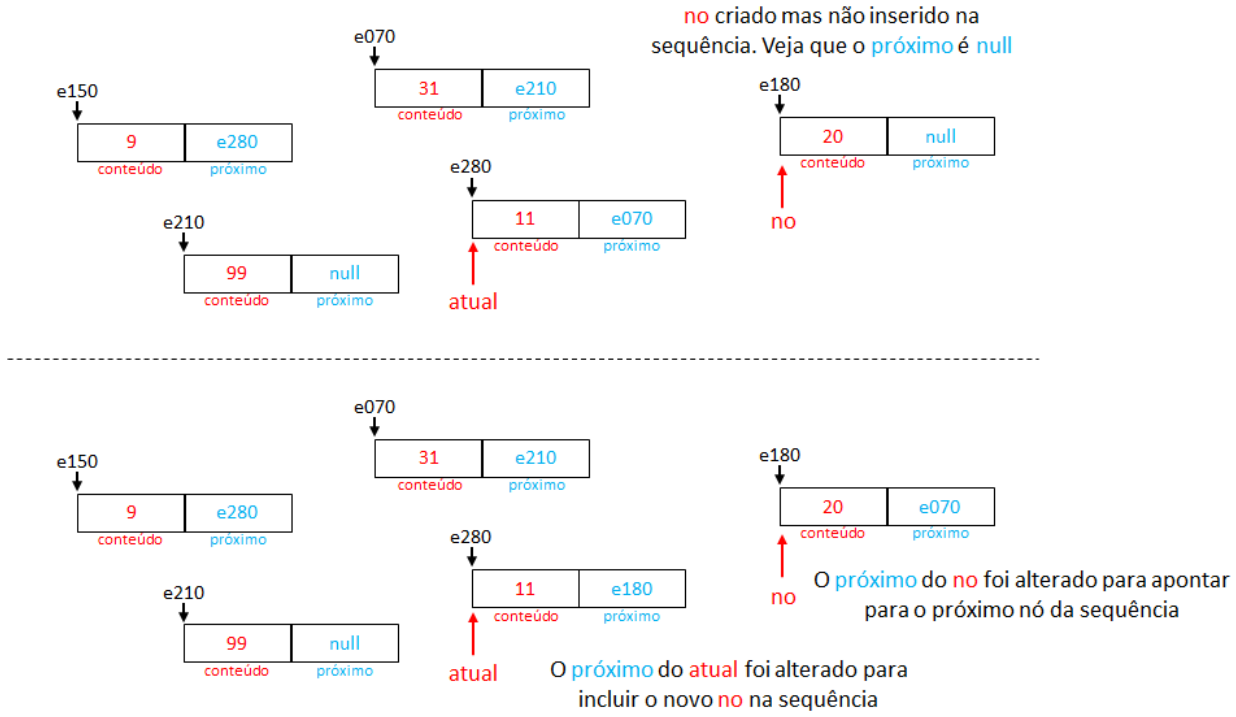

Figura 8 – Representação da inserção de um novo nó no meio da lista.

A Figura 9 mostra a implementação do método para inserir um registro em qualquer posição da lista, veja que foi necessário manter dois apontadores, caso contrário não seria possível inserir a esquerda (antes) do nó atual. A Figura 10 possui o código para testar a inserção, já a Figura 11 mostra o resultado.

A classe Lista da Figura 9 possui ainda os métodos buscar(int) e buscar(int,No) para procurar um elemento na lista e retorná-lo. Ambos os métodos são equivalentes, mas o buscar(int,No) utiliza recursividade.

A classe Lista da Figura 9 possui também o método remover(int) para remover da lista o nó que possui o valor passado como parâmetro.

```
public class Lista {
   No inicio;

   Lista(){
      /* a lista está vazia */
      inicio = null;
   }

   void inserir(int nro){
      No no = new No();/* criar um nó */
      no.conteudo = nro;

      No anterior = null; /* ponteiro para o elemento anterior*/
      No atual = inicio; /* ponteiro para o elemento atual*/

      /* procura a posição de inserção */
      while( atual != null && atual.conteudo < nro){
         anterior = atual;
         atual = atual.proximo;
      }

      if( anterior == null ){
         /* insere antes do início */
         no.proximo = inicio;
         inicio = no;
      }
      else{
         /* insere no meio ou final da sequência */
         no.proximo = anterior.proximo;
         anterior.proximo = no;
      }
   }

   void imprimir(){
      /* checa se a lista está vazia */
      if( inicio == null ){
         System.out.println("Lista vazia");
      }
      else{
         System.out.println(); /* quebra de linha na tela */
         /* percorrer a lista até encontrar o último nó */
         No ultimo = inicio;
         while( ultimo != null ){
            System.out.print( ultimo.conteudo +" ");
            ultimo = ultimo.proximo;
         }
      }
   }

   /*irá retornar o nó que possui o nro ou null caso contrário*/
   No buscar(int nro){
      No atual = inicio;
      /* percorre até encontrar o nro ou o fim da lista */
      while( atual != null && atual.conteudo != nro ){
         atual = atual.proximo;
      }
      return atual;
   }

   /* irá retornar o nó que possui o nro ou null caso contrário.
   *  A busca será por recursividade */
   No buscar(int nro, No no){
      if( no == null ) return null;
      else if( no.conteudo == nro ) return no;
      else return buscar(nro, no.proximo);
   }

   void remover(int nro){
      No anterior = null; /* ponteiro para o elemento anterior*/
      No atual = inicio; /* ponteiro para o elemento atual*/

      /* procura o nó que possui o nro */
      while( atual != null && atual.conteudo != nro){
         anterior = atual;
         atual = atual.proximo;
```

```
public class Principal {
   public static void main(String[] args) {
      Lista lista = new Lista();
      lista.inserir(20);
      lista.imprimir();
      lista.inserir(10);
      lista.imprimir();
      lista.inserir(15);
      lista.imprimir();
      lista.inserir(40);
      lista.imprimir();
      lista.inserir(8);
      lista.imprimir();
   }
}
```

Figura 10 – Código da classe Principal para testar a classe Lista.

```
20
10 20
10 15 20
10 15 20 40
8 10 15 20 40
```

Figura 11 – Resultado do código da Figura 10.

```
      }

      if( anterior == null && atual != null ){
         /* o nro está no 1o nó */
         inicio = atual.proximo;
      }
      /* atual será null quando o nro não existir */
      else if( atual != null ){
         /* remove do meio ou final da sequência */
         anterior.proximo = atual.proximo;
      }
   }
}
```

Figura 9 – Código da classe Lista para inserir um nó no meio da lista.

Cabeça de lista: uma lista pode ser implementada usando o 1º elemento da lista encadeada apenas como um marcador de início, ou seja, o seu conteúdo é ignorado. Nesse caso, dizemos que o 1º elemento é a cabeça da lista encadeada. A Figura 12 mostra uma implementação de lista encadeada com cabeça, esse código é equivalente a classe Lista da Figura 6. A Figura 13 mostra o código para testar classe Lista e a Figura 14 o resultado.

```
public class Lista {
   No inicio;

   Lista(){
      /* o início é a cabeça da lista */
      inicio = new No();
   }

   void inserir(int nro){
      /* criar um nó */
      No no = new No();
      no.conteudo = nro;
      no.proximo = null; /* este será o último nó da lista */

      /* percorrer a lista até encontrar o último nó */
      No ultimo = inicio;
      while( ultimo.proximo != null ){
         ultimo = ultimo.proximo;
      }
      /* alterar o próximo do último para o endereço do no */
      ultimo.proximo = no;
   }

   void imprimir(){
      /* checa se a lista está vazia */
      if( inicio.proximo == null ){
         System.out.println("Lista vazia");
      }
      else{
         System.out.println(); /* quebra de linha na tela */
         /* percorrer a lista até encontrar o último nó */
         No ultimo = inicio.proximo;
         while( ultimo != null ){
            System.out.print( ultimo.conteudo +" ");
            ultimo = ultimo.proximo;
         }
      }
   }
}
```

Figura 12 – Código da classe Lista com cabeça.

```
public class Principal {
   public static void main(String[] args) {
      Lista lista = new Lista();
      lista.imprimir();
      lista.inserir(20);
      lista.imprimir();
      lista.inserir(10);
      lista.imprimir();
      lista.inserir(15);
      lista.imprimir();
      lista.inserir(40);
      lista.imprimir();
      lista.inserir(8);
      lista.imprimir();
   }
}
```

Figura 13 – Código da classe Principal para testar a classe Lista.

```
Lista vazia

20
20 10
20 10 15
20 10 15 40
20 10 15 40 8
```

Figura 14 – Resultado do código da Figura 13.

3 - Exercícios

1 – Programar na classe Lista, da Figura 9, o método last():No para retornar o último nó da lista. Observação: os demais métodos **não** podem ser alterados.

2 – Programar na classe Lista, da Figura 9, os métodos count():int e countR(quant:int, no:No):int para retornar a quantidade de nós que possui a lista. O método count():int deverá ter uma implementação iterativa e countR():int uma implementação recursiva. Observação: os demais métodos **não** podem ser alterados.

3 – Programar na classe Lista, da Figura 9, o método sumR(soma:int, no:No):int para retornar o somatório dos conteúdos dos nós da lista. O método retorna zero se a lista estiver vazia. A implementação deverá ser recursiva. Observação: os demais métodos **não** podem ser alterados.

4 – A altura de um no em uma lista encadeada é a distância entre no e o fim da lista. Mais precisamente, a altura do no é o número de passos do caminho que leva do no até a última célula da lista. Programar na classe Lista, da Figura 9, um método de nome height para retornar a altura de um elemento recebido como parâmetro.

5 – A profundidade de um elemento no em uma lista encadeada é a distância entre o no e o início da lista. Programar na classe Lista, da Figura 9, um método de nome depth para retornar a profundidade de um elemento recebido como parâmetro.

6 – Programar na classe Lista, da Figura 6, o método sort para ordenar a lista. Observações: os demais métodos **não** podem ser alterados e não deverá ser criada outra lista, ou seja, terá de usar a própria lista para ordenar.

7 – Programar na classe Lista, da Figura 6, o método unsort para ordenar os elementos da lista em ordem decrescente. Observações: os demais métodos **não** podem ser alterados e não deverá ser criada outra lista, ou seja, terá de usar a própria lista para ordenar.

8 – Programar na classe Lista, da Figura 6, o método toArray que retorna todos os elementos da lista como um array de inteiros. Observações: os demais métodos **não** podem ser alterados e os elementos do array deverão estar na mesma ordem da lista.

9 – Programar na classe Lista, da Figura 6, o método inserir(v:int[]), ele carrega todos os elementos do array **v** na lista. Observação: os demais métodos **não** podem ser alterados.