Lista Encadeada

Conforme já apresentado, uma lista encadeada (ou ligada) consta de um número indeterminado de elementos dispostos em uma organização física não linear, ou seja, espalhados na memória, denominados **nós**. Para organizar a lista de maneira que essa possa ser utilizada como um conjunto linear, cada nó tem dois componentes (campos), um valor, que pode ser de qualquer tipo, e um endereço (uma referência) para o nó seguinte da lista. O último nó é representado de maneira diferente para significar que esse nó não se liga a nenhum outro, conforme mostrado na Figura 2.

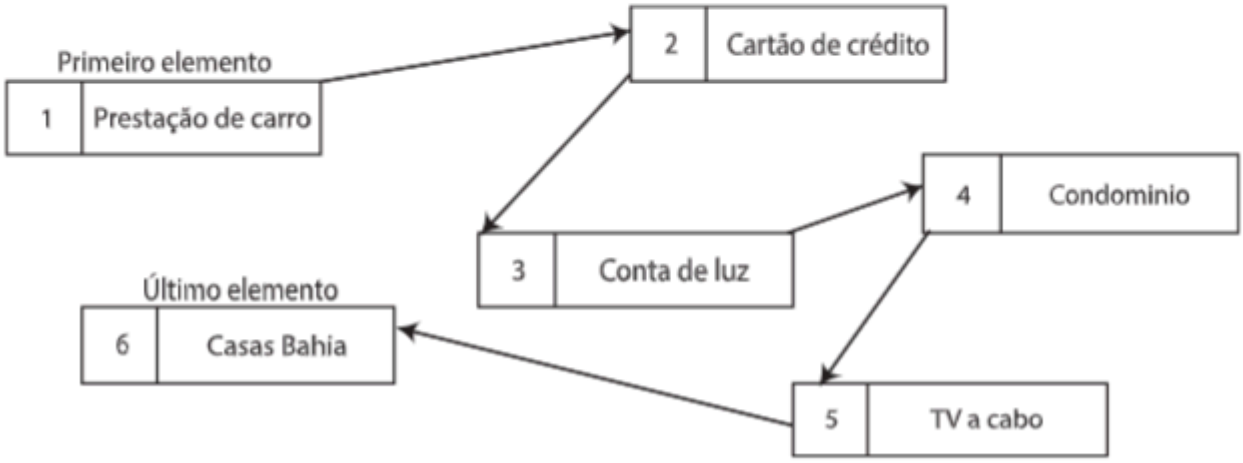


Figura 2 – Exemplo de lista encadeada de pagamentos
Fonte: Adaptado de Puga (2016)

As listas encadeadas podem ser divididas em quatro categorias:

·**Encadeada simples:** cada nó contém um único endereço que o conecta ao nó seguinte ou sucessor, conforme mostrado na Figura 2;

· **Duplamente encadeadas:** cada nó contém dois endereços, um ao seu nó antecessor e outro ao seu sucessor Veia a Figura 3.

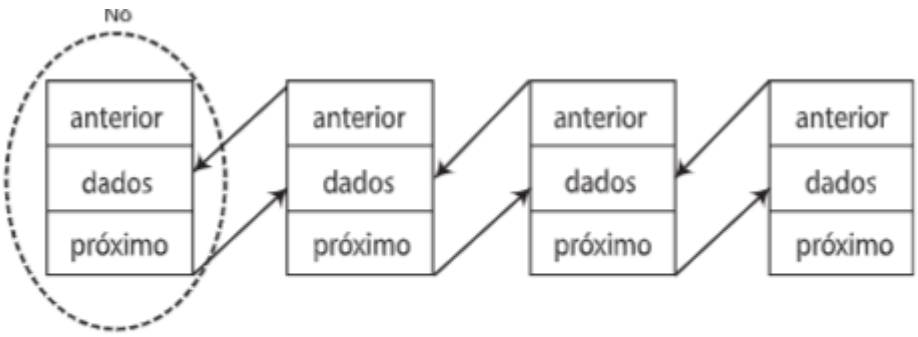
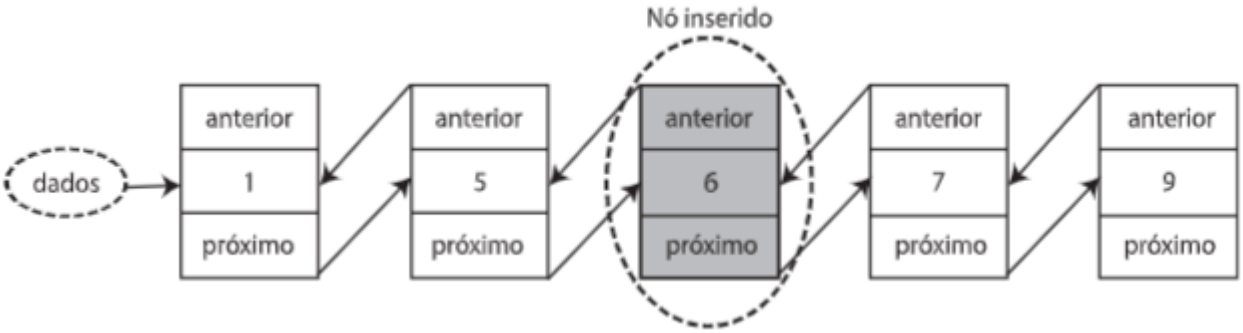
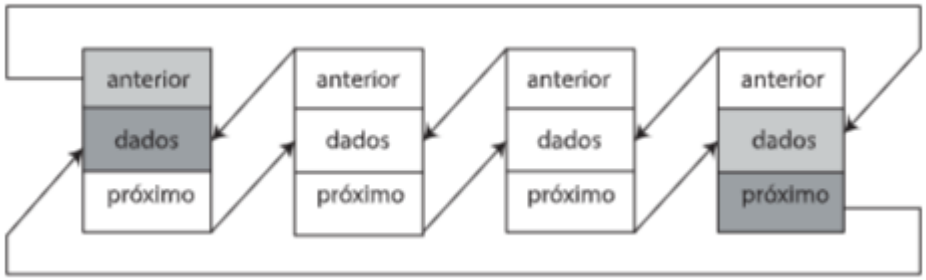


Figura 3 – Listas duplamente encadeadas
Fonte: Adaptado de Puga (2016)

· **Ordenadas:** a ordem linear da lista corresponde à ordem linear dos elementos. Assim, quando um novo elemento é inserido, o mesmo deve ser posicionado de tal modo que garanta que a ordem da lista será mantida (veja a Figura 4). Uma lista ordenada pode ter encadeamento simples ou duplo, porém, o princípio de ordenação é o mesmo;



Circulares: o último elemento se liga ao primeiro elemento, e vice-versa. Essa lista pode ser percorrida de modo circular, tanto da direita para esquerda quanto da esquerda para direita. Conforme mostra a Figura 5.



fonte. Muaplado de ruga (2010)

Sabendo que a alocação fixa de memória por meio de vetor é menos eficiente, então, adotaremos nesta Unidade a alocação de memória dinâmica e implementaremos a lista com ligações simples

O gerenciamento dos elementos de uma lista é realizado por operações, que terão as seguintes tarefas:

· **Inicialização ou criação, com declaração dos nós;**

· **Inserir elementos em uma lista**;

· **Eliminar elementos de uma lista;**

· **Buscar elementos de uma lista.**

Criação de um nó

Um nó será representado em C# por uma classe, denominada aqui de Node, a qual deve conter um valor (que nesta implementação será um objeto genérico denominado item) e a variável que referenciará o próximo elemento da lista (deno- minado prox do tipo Node), conforme apresentado na Figura 6.

```
class Node
{
    private Object item;
    private Node prox;

    Node(Object item)
    {
        this.item = item;
        prox = null;
    }
}
```

Figura 6 – Classe que cria a estrutura de nó simples

Inserir elementos em uma lista

Agora que você já conhece a estrutura do nó, podemos inserir novos elemen- tos no início, no fim ou em uma posição específica da lista. Na lógica a seguir (Figura 7), temos, em uma classe chamada ListaSimples, os atributos do tipo Node primeiro e segundo, que referem-se ao primeiro e último nó da lista respectivamente, sendo que o último tem a referência para nulo (nulo), e três métodos de inserção de elemento na lista - um método que insere um nó no início da lista (insereInicio), um que insere em uma posição específica (inserePosicao), veja Figura 8, e outro que insere no final (insereFim) - e ainda um método que retorna a quantidade de elementos da lista (contaNos). Para que a operação possa ocorrer no início ou final da lista, você precisa saber qual é o primeiro nó e o último, ou seja, o nó cabeça, que conterá o primeiro ele- mento da lista, e o nó cauda, que contém o último elemento da lista.

```
public object primeiro { get; private set; }
public object ultimo { get; private set; }

class ListaSimples
{
    private Node primeiro, ultimo;
    private int qtdeNos;

    ListaSimples()
    {
        primeiro.setProx(null);
        ultimo.setProx(null);
    }
    void insereFim(Node novo)
    {
        novo.setProx(null);
        if (this.primeiro==null)
            this.primeiro = novo;
        if (this.ultimo != null)
            this.ultimo.setProx(novo);
        this.ultimo = novo;
    }
    void insereInicio(Node novo)
    {
        if (this.primeiro != null)
            novo.setProx(novo);
        else
        {
            if (this.primeiro == null)
                this.primeiro = novo;
            this.ultimo = novo;
        }
    }
    void inserePosicao(Node novo, int pos)
    {
        Node aux = primeiro;
        int qtde = contaNos();
        int pos_aux;
        if(pos == 0)
        {
            novo.setProx(primeiro);
            if (primeiro == ultimo)
            {
                ultimo = novo;
            }
            primeiro = novo;
```

```
            primeiro = novo;
        }
        else
        {
            if(pos <= qtde)
            {
                pos_aux = 1;
                while(aux != null && pos > pos_aux)
                {
                    aux = aux.getProx();
                    pos_aux++;
                }
                aux.setProx(novo);
            }
            else
            {
                if(pos > qtde)
                {
                    ultimo.setProx(novo);
                    ultimo = novo;
                }
            }
        }

    }
    public int contaNos()
    {
        int tam = 0;
        Node aux = primeiro;
        while (aux != null)
        {
            tam++;
            aux = aux.getProx();
        }
        return tam;
    }
}
```

Excluir elemento da lista

A partir da operação de exclusão, apresentaremos apenas o método correspondente, lembrando que esse e os demais foram desenvolvidos no escopo da classe ListaSimples, apresentada na Figura 7.

```
void excluiNo (Object item)
{
    Node aux = primeiro;
    while (aux != null && aux.getItem()!=item)
    {
        aux = aux.getProx();
    }
    aux.setProx(aux.getProx().getProx());
    if (ultimo == aux.getProx())
        ultimo = aux;
}
```

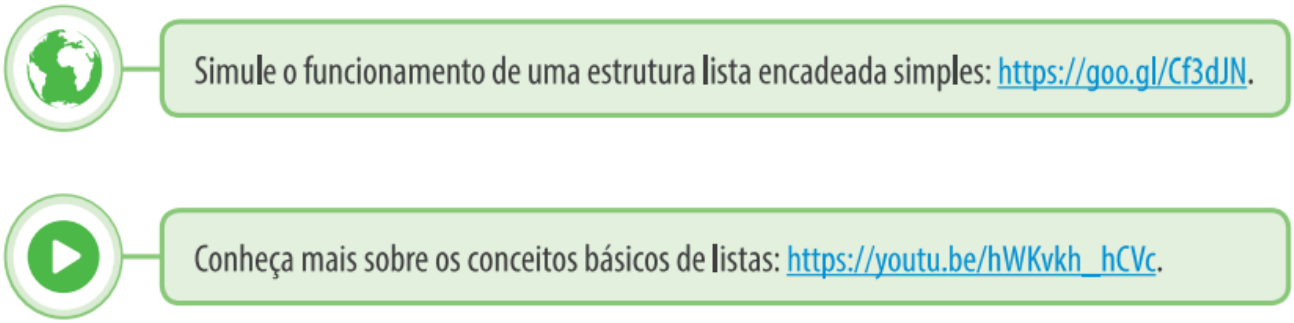
Buscar elemento em uma lista

O método de buscar de um elemento da lista (Figura 10) recebe como parâmetro o objeto genérico item que deverá ser pesquisado na lista. Para isso, é feita a varredura de todos os nós da lista por meio da estrutura de repetição while. Nessa, é comparado o campo item do nó da

lista com o item passado como parâmetro - quando são iguais, retorna o nó; caso não encontre, retorna nulo.

```
Node buscaNo(Object item)
{
    int i = 0;
    Node aux = primeiro;
    while (aux != null)
    {
        if(aux.getItem() == item)
        {
            return aux;
        }
        i++;
        aux = aux.getProx();
    }
    return null;
}
```

Com esses métodos, você é capaz de utilizar a estrutura de dados lista para resolver problemas na área de jogos que sejam possíveis de serem aplicados à lógica aqui apresentada. Observe que, após você entender a da lista encadeada simples, os outros tipos de listas podem ser implementados.



Listas Ordenadas

Na lista ordenada, os elementos permanecem ordenados, de acordo com algum critério, após a operação de inserção ou exclusão, ou seja, os nós podem ser inseridos e eliminados na posição correta, sem realocação dos demais elementos. Pela definição de lista ordenada, nada impede que haja elementos repetidos, como a lógica de busca e de exclusão irá operar sobre esses elementos depende do problema.

As operacoes de criacao, remoção e busca podem ser as mesmas apresentadas na lista simples encadeada. A lógica se diferenciará nos métodos de inserção. No caso da lista

ordenada, o elemento a ser inserido será posicionado antes do elemento que possui o valor maior do que ele. Assim, o nó que possuir o valor menor irá se referenciar ao novo nó, e o novo para o nó que possuir o maior valor quando comparado a ele - no caso da ordenação crescente; já no caso da decrescente, acontecerá o inverso.

Para que você entenda na prática a lógica da lista ordenada, a Figura 11 apresenta o método de inserção de nó em uma lista simples encadeada. Nessa, consideramos que a lista está manipulando como conteúdo números inteiros.

```
void insereNoOrdenado(Node novo)
{
    Node aux;
    //Verificando se a lista está vazia
    if (primeiro == null || contaNos() == 0)
    {
        novo.setProx(primeiro);
        primeiro = novo;
    }
    else
    {
        aux = primeiro;
        int valorNovo = Convert.ToInt32(novo.getItem());
        int valorAux = Convert.ToInt32(aux.getItem());
        while (aux.getProx() != null &&  valorNovo > valorAux)
        {
            aux = aux.getProx();
        }
        novo.setProx(aux.getProx());
        aux.setProx(novo);
    }
}
```

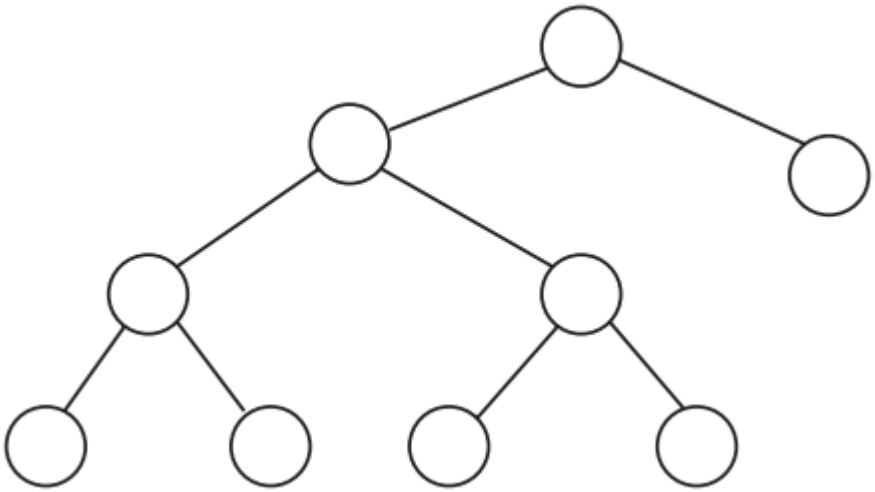
Veja mais sobre listas encadeadas em C# em: https://youtu.be/oHJBLaD5fQk.

Árvores

As estruturas apresentadas até aqui são chamadas de lineares. A importância dessas estruturas é inegável, porém, elas não são adequadas para representar dados que devem ser

dispostos de maneira hierárquica. A estrutura que preenche essa lacuna é a árvore.

Uma árvore é uma estrutura bidimensional, não linear, que possui propriedades especiais e admite muitas operações de conjuntos dinâmicos, tais como a consulta, inserção, remoção, entre outros.



As características de árvores são :

---

nó raiz: nó no topo da árvore, do qual descendem o os demais nós; é o primeiro nó da árvore;

---

nó interior: nó do interior da árvore (que possui descendentes);

---

nó terminal: nó que não possui descendentes;

---

trajetória: número de nós que devem ser percorridos até o nó determinado;

---

grau do nó: número de nós descendentes do nó, ou seja, o número de subárvores de um nó;

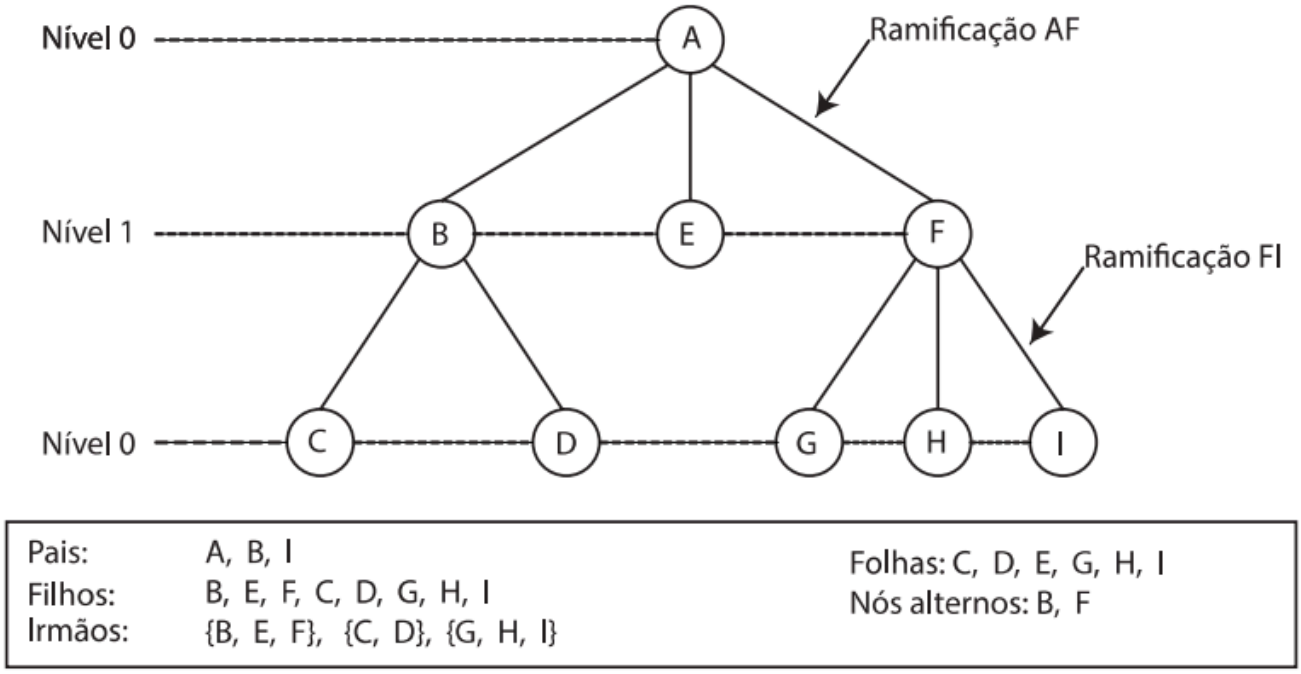
---

grau da árvore: número máximo de subárvores de um nó;

---

altura da árvore: número máximo de níveis dos seus nós.

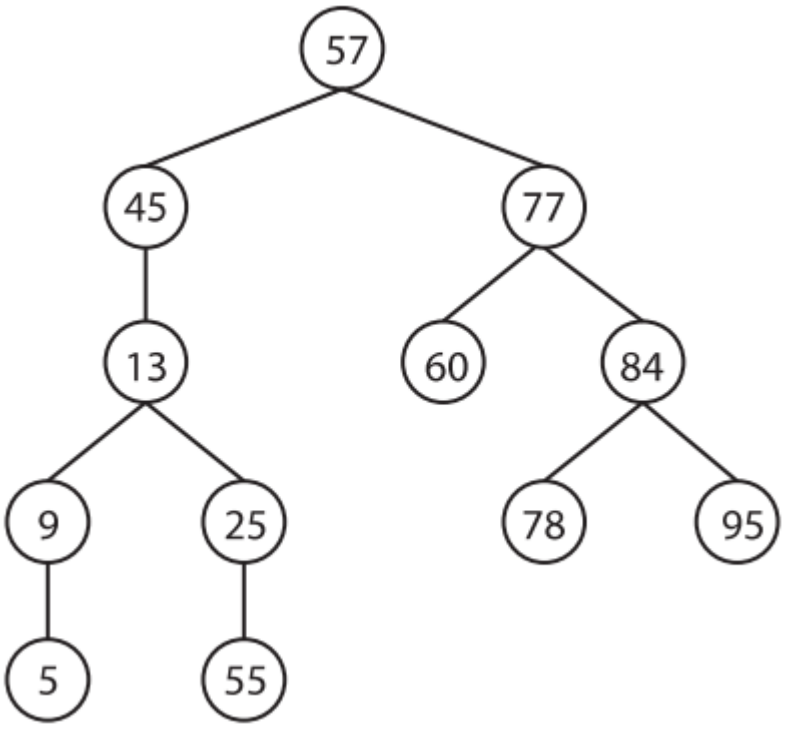
Podemos utilizar árvores binárias para armazenamento temporário de conjunto le elementos e pode ser implementada com armazenamento dinâmico, através de listas duplamente encadeadas. Com relação ao tipo, elas podem ser do tipo listas generalizadas ou binárias. Nas árvores binárias, cada nó possui, no máximo, dois filhos. Nesta Unidade, abordaremos apenas as árvores binárias.



Árvores binárias

Uma árvore binária pode ser nula, e assim como qualquer árvore possui um elemen- to denominado raiz, e os demais elementos são particionados em T1 e T2, estruturas disjuntas de árvore binária. A subárvore da esquerda é denominada T1 e a da direita T2. Nesse tipo de árvore também existe a particularidade quanto à posição dos nós. Os nós da direita sempre possuem valor superior aos do nó pai, e os da esquerda nó pai .

sempre possuem valor inferior ao do nó pai



Para a manipulação de árvore, existe uma grande similaridade com os nós criados para manipular listas, por isso utilizaremos os mesmos princípios, criaremos uma classe BTreeNo, a qual implementa o nó da árvore e contém os atributos item do tipo int, o que possibilita manipular números inteiros com o objetivo de facilitar o entendimento da lógica, mas vale lembrar que a árvore manipula qualquer tipo de dados e os atributos esq e dir do tipo BTreeNo. A cada novo nó inserido na árvore, uma instância da classe BTreeNo será criada, ou seja, um novo objeto nó.

```
class BTreeNo
{
    private int valor;
    private BTreeNo esq;
    private BTreeNo dir;

    BTreeNo(int valor)
    {
        this.valor = valor;
    }
    public void setValor(int valor)...
    public void setEsq(BTreeNo esq)...
    public void setDir(BTreeNo dir)...
    public int getValor()...
    public BTreeNo getEsq()...
    public BTreeNo getDir()...
}
```

Inserir nó em uma árvore

Após definir a estrutura do nó da árvore, o próximo passo é criar a classe árvore, que será instanciada toda vez que uma nova árvore for criada, contendo os métodos que possibilitem operações de inserção e exclusão de nós. Os métodos de inserção são apresentados na Figura 19.

```
class BTree
{
    private BTreeNo raiz;

    private BTreeNo inserir(BTreeNo arvore, int novo)
    {
        BTreeNo aux = null;
        if (arvore == null)
        {
            aux.setValor(novo);
            return aux;
        }
        else if (novo < arvore.getValor())
            arvore.setEsq(inserir(arvore.getEsq(), novo));
        else
            arvore.setDir(inserir(arvore.getDir(), novo));

        return arvore;
    }
    public void inserirNo(int novo)
    {
        raiz = inserir(raiz, novo);
    }
}
```

De acordo com a lógica acima apresentada, o objeto BTreeNo será instanciado toda vez que um novo nó for inserido na árvore. O método inserirNo é o responsável em receber o valor a ser inserido e, então, ele chama o método inserir, que recebe como parâmetro um valor do tipo BTreeNo (chamado no código de árvore) e um valor inteiro. Esse percorre recursivamente a árvore a partir da raiz, buscando uma posição de referência nula para inserir o novo elemento. Quando a estrutura de dados está vazia, o método insere o novo elemento na raiz; quando a estrutura já tem elementos armazenados, o método verifica se o va- lor a ser inserido é maior ou menor que o nó. Caso seja menor, será inserido à es- querda; caso contrário, insere-se à direita. A recursividade é utilizada para percorrer os nós até encontrar um nó vazio (null) que possibilita a inserção de um novo nó.

Exibir nós de uma árvore A partir da operação para exibir os valores, apresentaremos apenas o método correspondente, lembrando que esse e os demais foram desenvolvidos no escopo da classe BTree, apresentada na Figura 19.

```
public void exibirEsquerdo(BTreeNo arv)
{
    if (arv != null)
    {
        exibirEsquerdo(arv.getEsq());
        Console.WriteLine(arv.getValor());
    }
}
public void exibirDireito(BTreeNo arv)
{
    if (arv != null)
    {
        exibirDireito(arv.getEsq());
        Console.WriteLine(arv.getValor());
    }
}
public void ExibirRaiz()
{
    Console.WriteLine("Raiz: " + raiz.getValor());
}
public void exibirNoEsq()
{
    exibirEsquerdo(raiz);
}
public void exibirNoDir()
{
    exibirDireito(raiz);
}
```

Assim como o método inserirNo, os métodos exibirNoEsq e exibirNoDir não recebem parâmetros e chamam os métodos exibirEsquerdo ou exibiDireito, e passa como parâmetro o nó denominado raiz. Os métodos exibirEsquerdo e exibirDireito recebem como parâmetro um objeto do tipo BTreeNo, que é passa- do pelo método exibirNoEsq ou exibirNoDir; e que, por meio de chamadas recursivas, buscam os nós à esquerda ou à direita da árvore, até que não encontrem um nó nulo. Esse percurso garante a impressão sempre na ordem ascendente de valor.

Excluir nó de uma árvore

A exclusão de um nó da árvore requer uma lógica mais complexa, uma vez que as referências ao nó excluído e seus filhos precisam ser devidamente ajustados. A lógica a seguir, Figura 21, apresenta o método de exclusão, que recebe como parâmetro o valor do elemento a ser excluído.

```
    public void exluirNo(int item)
    {
        BTreeNo aux=raiz, pai=null, filho=raiz, temp;
        while(aux != null && aux.getValor() != item)
        {
            pai = aux;
            if (item < aux.getValor())
                aux = aux.getEsq();
            else
                aux = aux.getDir();
        }
        }else if (aux.getDir() == null)
        {
            if (pai.getEsq() == aux)
                pai.setEsq(aux.getEsq());
            else
                pai.setDir(aux.getEsq());
        }else if(aux.getEsq() == null)
        {
            if (pai.getEsq() == aux)
                pai.setEsq(aux.getDir());
            else
                pai.setDir(aux.getDir());
        }
        if (aux == null)
          {
            for (temp = aux, filho = aux.getEsq(); filho.getDir() != null;
                temp = filho, filho = filho.getDir());
            if(filho != aux.getEsq())
            {
                temp.setDir(filho.getEsq());
                filho.setEsq(raiz.getEsq());
            }
            filho.setDir(raiz.getDir());
            raiz = filho;
          }
        else
        {
            for (temp = aux, filho = aux.getEsq(); filho.getDir() != null;
                    temp = filho, filho = filho.getDir()) ;
            if (filho != aux.getEsq())
            {
                temp.setDir(filho.getEsq());
                filho.setEsq(aux.getEsq());
            }
            filho.setDir(aux.getDir());
            if(pai.getEsq() == aux)
                pai.setEsq(filho);
            else
                pai.setDir(filho);
        }
    }
}
```

Primeiramente, é feita uma busca nos diversos nós da árvore, comparando o valor passado como parâmetro com o valor do nó e, após isso, são tratados os diversos cenários que podem acontecer, tais como percorrer totalmente a árvore e não encontrar o valor; o nó pesquisado ser