Índice

Capítulo 4 – Estrutura de Dados não sequencial com armazenamento não sequencial ("Árvore")

Capítulo 4 – Estrutura de Dados não sequencial com armazenamento não sequencial ("Árvore")

1. Introdução

1.1. Definição

Uma árvore é uma abstração matemática que serve para especificar relações, descrever organizações e armazenar informação, que se apresenta estruturada de forma hierárquica. Portanto, as árvores são esquemas utilizados para representar estruturas hierárquicas, como árvores genealógicas de famílias, campeonatos de modalidades desportivas e organização de grandes empresas. Nas ciências da computação, as árvores podem ser utilizadas para representar decisões, definições formais de linguagem ou mesmo para representar a hierarquia entre elementos.

No contexto das ciências da computação, uma árvore é uma estrutura de dados onde os dados estão dispostos de forma hierárquica (um conjunto de dados é hierarquicamente subordinado a um outro conjunto de dados).

Uma árvore pode definir-se como um conjunto finito de um ou mais **nodos** (nós ou vértices), tais que:

- existe um nodo particular chamado **raíz** (nodo principal);
- os restantes nodos (denominados de **filhos**) estão divididos em **n** ≥ 0 conjuntos disjuntos T_1, T_2, …, T_M, os quais estão ligados à raiz através de **ramificações**;
- cada conjunto de nodos T_k, k = 1, …, M é uma árvore, denominada **subárvore da raiz**;
- cada nodo filho é também raiz de subárvore;
- um nodo sem filhos é denominado de **folha** (ou nodo terminal).

A Figura 1 representa uma analogia entre árvore no contexto da biológica e árvore no contexto das ciências da computação (como estrutura de dados).

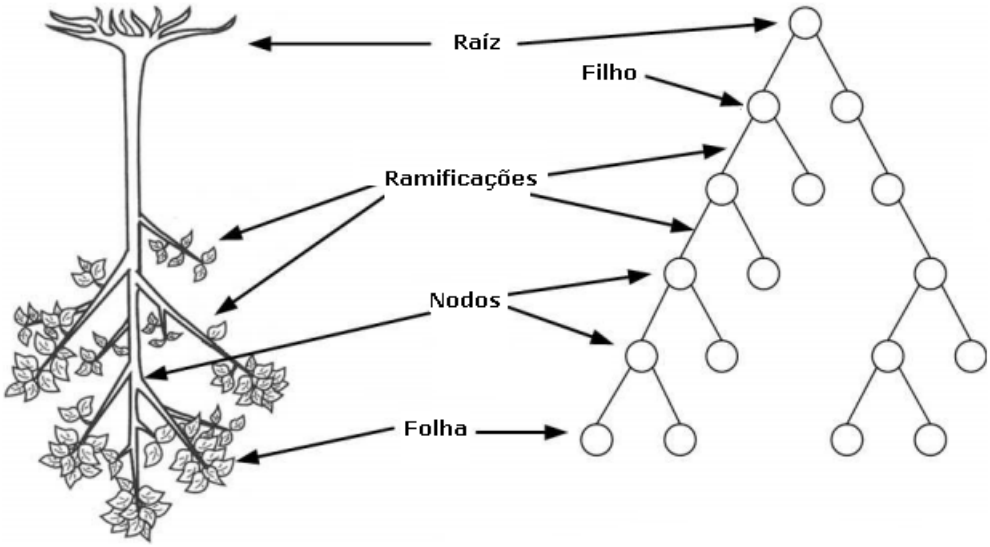


Figura 1 – Analogia entre árvores (biológica e como estruturas de dados).

1.2. Conceitos relacionados

Existe alguns termos que se podem usar, tais como: pai, filho, irmão, antepassado, descendente, folha, etc..

O grau de um nodo define-se como o número de subárvores desse nodo. Da mesma maneira, o grau de uma árvore é o grau máximo dos seus nodos. O número de filhos de um nodo é chamado de grau de saída de um nodo.

O nível de um nodo pode ser definido da seguinte forma: a) o nodo raiz tem nível 0; b) os níveis dos restantes nodo são uma unidade mais do que o nível dos seus pais.

A altura (profundidade) de uma árvore é o nível máximo dos seus nodos.

2. Árvores binárias

2.1. Definição

Uma árvore binária é um conjunto finito de elementos (nodos) que pode ser vazio ou particionado em três subconjuntos:

- raiz da árvore (elemento inicial, que é único);
- subárvore esquerda (se vista isoladamente forma uma outra árvore);
- subárvore direita (se vista isoladamente forma uma outra árvore).

Como a definição de árvore é recursiva, muitas operações sobre árvores binárias são definidas de forma recursiva.

As árvores onde cada nodo, que não seja folha, tem subárvores esquerda e direita não vazias são conhecidas como árvores estritamente binárias. Uma árvore estritamente binária com **n** folhas tem **2n-1** nodos.

A Figura 2 apresenta um método convencional para representar uma árvore. O nodo A é a raiz da árvore, a subárvore esquerda tem como raiz o nodo B e a direita é representada pelo nodo C. Um nodo sem filhos é chamado de **folha** (são os casos dos nodos D, G, H e I). Sendo A a raiz de uma árvore binária e B uma sua subárvore, então diz-se que A é **pai** de B e que B é **filho** de A. Os nodos que se encontram no mesmo nível denominam-se de **irmãos**.

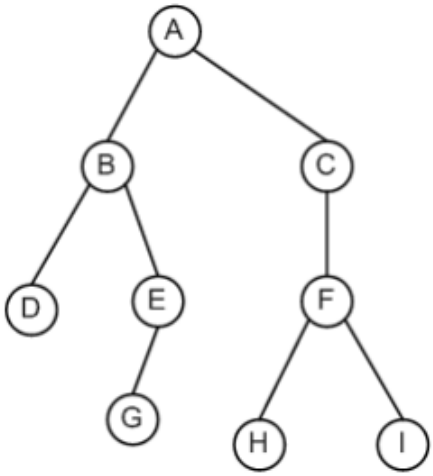


Figura 2 – Exemplo de uma árvore binária.

Relações (e conceitos) que podem ser observados na Figura 2:

- B e C são filhos de A.
- B e C são irmãos. D e E são irmãos. H e I são irmãos.
- T_A é a subárvore enraizada em A, portanto toda a árvore.
- T_F é a subárvore enraizada em F, que contém os nós F, H e I.
- Nodos sem filhos são chamados de folhas: os nodos D, G, H e I são folhas.
- Os graus de saída do nodo B é 2 e do nodo C é 1.
- O nodo A tem nível 0; os nodos B e C têm nível 1; os nodos D, E e F têm nível 2; os nodos G, H e I têm nível 3.
- Altura de A é 4, altura de C é 3, altura de E e F é 2, e altura de D, G, H e I é 1.

2.2. Caminho duma árvore

Um caminho da árvore é formado por uma sequência de nodos consecutivos (n_1, n_2, …,n_{k-1}, n_k) tal que existe sempre a relação: n_j é pai de n_{j+1}; os k nodos formam um

caminho de comprimento k-1. O comprimento entre o nodo A e o nodo H é 3 (Figura 2). A altura de um nodo pode-se também definir como o comprimento do maior caminho deste nodo até alguns de seus descendentes. Descendentes de um nodo são todos os nodos que podem ser alcançados caminhando-se para baixo a partir daquele nodo.

2.3. Árvore Binária Completa

Uma árvore completa é uma árvore estritamente binária na qual todas as folhas estão no mesmo nível k. Sendo k a profundidade da árvore, o número total de nodos é 2^{k+1}-1 e o número total de folhas é 2^k. A Figura 3 representa uma árvore completa de nível 3.

Embora uma árvore binária completa possua muitos nodos (o máximo para cada profundidade), a distância da raiz a uma folha qualquer é relativamente pequena. A árvore da Figura 3 tem profundidade 3 com 15 nodos (2^4-1) e 8 folhas (2^3).

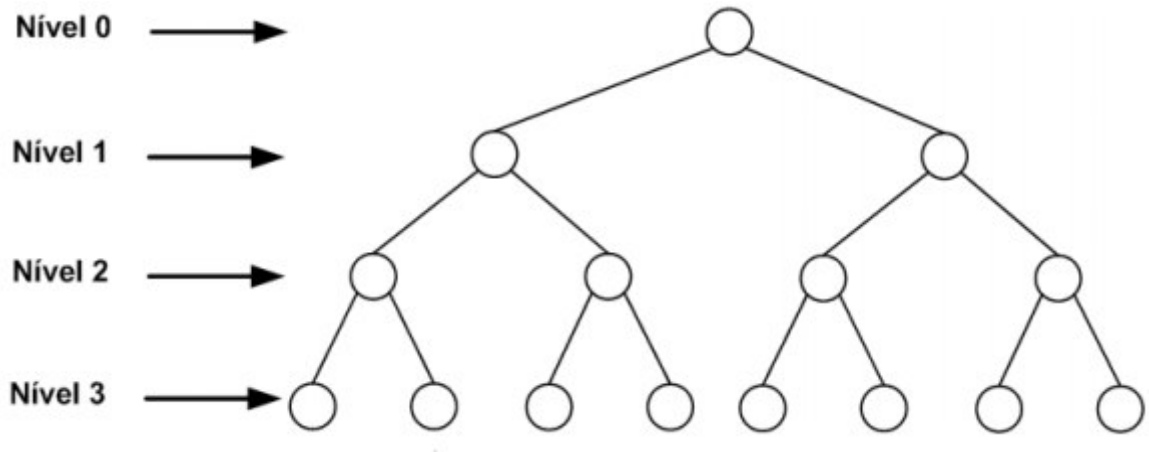


Figura 3 – Indicação dos níveis numa árvore.

2.4. Representação de uma árvore binária usando listas ligadas

Cada nodo duma árvore binária pode ser representado por uma estrutura contendo a informação e as ligações a outros nodos (às raizes das subárvores esquerda e direita). Por exemplo, a declaração seguinte:

```
struct  NodoAB {
        Info  Elemento;
        struct  NodoAB *Esquerda;
        struct  NodoAB *Direita;
};
typedef  struct  NodoAB  *PNodoAB;
```

Elemento	
*Esquerda	*Direita

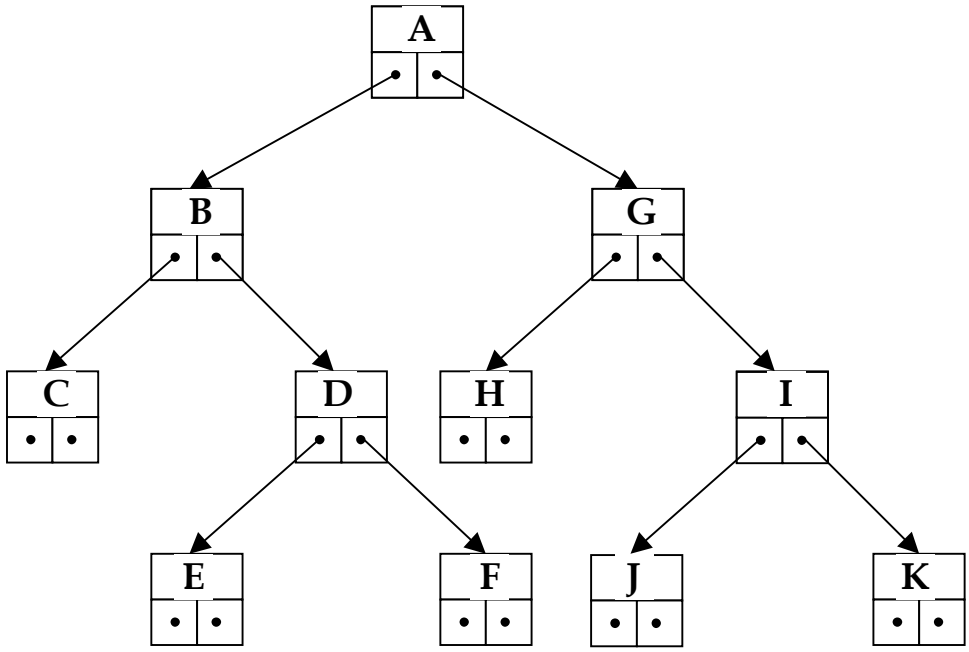


Figura 4 – Representação de uma árvore binária.

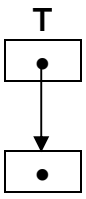
3. Operações sobre árvores binárias

3.1. Criar uma árvore binária (vazia)

Entrada: nada

Saída: devolve um ponteiro para uma árvore vazia (sem elementos)

```
PNodoAB CriarAB () {
    PNodoAB T;
    T = NULL;
    return T;
}
```



3.2. Verificar se uma árvore está vazia

Entrada: ponteiro para a raíz da árvore binária

Saída: 1 (se vazia); 0 (se não vazia)

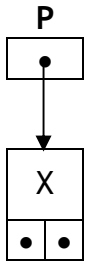
```
int ABVazia (PNodoAB T) {
    if (T == NULL)
        return 1;
    return 0;
}
```

3.3. Criar um nodo de uma árvore

Entrada: a informação que se pretende guardar

Saída: um ponteiro para um nodo (informação + ligações)

```
PNodoAB CriarNodoAB (Info X) {
    PNodoAB P = (PNodoAB) malloc(sizeof(struct NodoAB));
    if (P == NULL)
        return  NULL;
    P→Elemento = X;
    P→Esquerda = NULL;
    P→Direita = NULL;
    return P;
}
```



3.4. Destruir um nodo de uma árvore

Entrada: um ponteiro para o nodo que se pretende destruir

Saída: o ponteiro a apontar para NULL

```
PNodoAB LibertarNodoAB (PNodoAB P) {
    P→Esquerda = NULL;
    P→Direita = NULL;
    free(P);
    P = NULL;
    return P;
}
```

3.5. Destruir uma árvore

Como a linguagem C não liberta automaticamente a memória dinâmica alocada que já não está a ser usada, é muito importante desenvolver-se uma operação para remover todos os elementos de uma árvore, ou seja, providenciar uma operação de destruição da árvore para que o programador liberte a memória dinâmica que a árvore já não precisa de usar. Desta forma, a seguir apresenta-se uma função recursiva para este efeito.

Entrada: um ponteiro para a raiz da árvore que se pretende destruir

Saída: o ponteiro da raiz a apontar para NULL

```
PNodoAB DestruirAB (PNodoAB T) {
    if (T == NULL)
        return NULL;
    T→Esquerda = DestruirAB(T→Esquerda);
    T→Direita = DestruirAB(T→Direita);
    return LibertarNodoAB(T);
}
```

3.6. Copiar uma árvore

A cópia de uma árvore consiste em construir uma outra árvore a partir dos valores existentes numa outra árvore. Uma possível função recursiva é a que se segue.

Entrada: um ponteiro para a raiz da árvore que se pretende copiar

Saída: um ponteiro para a nova árvore

```
PNodoAB CopiarAB (PNodoAB T) {
    PNodoAB E, D, R;          // R é a nova árvore, que é uma cópia de T
    R = CriarAB();
    if (T == NULL)
        return R;
    E = CopiarAB(T→Esquerda);
    D = CopiarAB(T→Direita);
    R = CriarNodoAB(T→Elemento);
    R→Esquerda = E;
    R→Direita = D;
    return R;
}
```

3.7. Verificar a igualdade entre (duas) árvores

Dadas duas árvores binárias, P e Q, pretende-se verificar a igualdade entre elas. A igualdade não implica apenas terem os mesmos valores, mas também estes valores estarem colocados nas mesmas "posições" (isto é, terem a mesma estrutura). Uma possível função recursiva é a que se segue.

Entrada: ponteiros para as duas raízes das árvores que se pretendem comparar

Saída: 1 (se iguais); 0 (se diferentes)

```
int ABIguais (PNodoAB T, PNodoAB R) {
    int  ig;
    if (T == NULL && R == NULL)     // se são ambas nulas, então são iguais
        return 1;
    if (T == NULL || R == NULL)     // se uma delas é nula, então são diferentes
        return 0;
    if (T→Elemento != R→Elemento)  // raízes diferentes, árvores diferentes
        return 0;
    if (ABIguais(T→Esquerda, R→Esquerda) == 0) // subárvores esq. diferentes
        return 0;
    if (ABIguais(T→Direita, R→Direita) == 0)  // subárvores direitas diferentes
        return 0;
    return 1;         // caso contrário, são iguais
}
```

3.8. Determinar o número de nodos de uma árvore

Entrada: ponteiro para a raiz duma árvore

Saída: valor inteiro correspondente ao número de nodos da árvore

```
int NumeroNodosAB (PNodoAB T) {
    int  e, d;
    if (T == NULL)
        return 0;
    e = NumeroNodosAB(T→Esquerda);
    d = NumeroNodosAB(T→Direita);
    return (e + d + 1);
}
```

3.9. Determinar a altura de uma árvore

A altura de uma árvore binária corresponde ao nível mais elevado desta. Uma função recursiva possível para esta operação é a que se segue.

Entrada: ponteiro para a raiz duma árvore

Saída: valor inteiro correspondente ao altura/nível da árvore

```
int AlturaAB (PNodoAB T) {
    int  e, d;
    if (T == NULL)
        return -1;
    e = AlturaAB(T→Esquerda);
    d = AlturaAB(T→Direita);
    if (e > d)
        return (e + 1);  // e (altura da esquerda), 1 a raiz
    else
        return (d + 1);  // d (altura da direita), 1 a raiz
}
```

3.10. Listar os elementos de uma árvore

Uma operação comum é mostrar/imprimir todos os elementos de uma árvore binária, que consiste em visitar todos os nodos desta árvore segundo determinado critério e imprimir os seus elementos. De facto, e ao contrário do que acontece com as listas em que esta é uma operação simples por ser uma estrutura de dados linear, no caso da árvore binária, que é uma estrutura de dados hierárquica, é necessário definir um método sistemático de visitar cada nodo apenas uma vez. Esta operação, também chamada de travessia da árvore, pode ser feita de três formas: pré-ordem, em-ordem e pós-ordem.

Na caso da travessia se realizar em ***pré-ordem***, os nodos da árvore são visitados pela seguinte ordem: raiz, subárvore esquerda e subárvore direita. Tomando como exemplo a árvore da Figura 4, a sequência é a seguinte: A, B, C, D, E, F, G, H, I, J, K.

Se a travessia se realizar em ***em-ordem***, os nodos da árvore são visitados pela seguinte ordem: subárvore esquerda, raiz e subárvore direita. Tomando como exemplo a árvore da Figura 4, a sequência é a seguinte: C, B, E, D, F, A, H, G, J, I, K.

Se a travessia se realizar em ***pós-ordem***, os nodos da árvore são visitados pela seguinte ordem: subárvore esquerda, subárvore direita e raiz. Tomando como exemplo a árvore da Figura 4, a sequência é a seguinte: C, E, F, D, B, H, J, K, I, G, A.

A seguir, apresentam-se três possíveis funções recursivas, uma para cada uma das formas de realizar a travessia numa árvore.

Entrada: ponteiro para a raiz duma árvore

Saída: nada (mostra os elementos da árvores seguidos)

```
void ListarPreOrdemAB (PNodoAB T) {
    if (T != NULL) {
        printf(T→Elemento);
        ListarPreOrdemAB(T→Esquerda);
        ListarPreOrdemAB(T→Direita);
    }
}
void ListarEmOrdemAB (PNodoAB T) {
    if (T != NULL) {
        ListarEmOrdemAB(T→Esquerda);
        printf(T→Elemento);
        ListarEmOrdemAB(T→Direita);
    }
}
void ListarPosOrdemAB (PNodoAB T) {
    if (T != NULL) {
        ListarPosOrdemAB(T→Esquerda);
        ListarPosOrdemAB(T→Direita);
        printf(T→Elemento);
    }
}
```

3.11. Pesquisar um elemento numa árvore

Para procurar numa árvore binária um elemento específico deve-se examinar a raiz. Se o valor for igual à raiz, o elemento existe na árvore; caso contrário, deve-se pesquisar na subárvore da esquerda e, caso não existe nesta subárvore, deve-se então pesquisar na subárvore direita, e assim recursivamente em todos os nodos da subárvore. Uma possível função recursiva é a que se segue.

Entrada: ponteiro para a raiz duma árvore e o elemento a procurar

Saída: ponteiro para o nodo com o elemento (se existe); NULL (se não existe)

```
PNodoAB PesquisarAB (PNodoAB T, Info X) {
    PNodoAB P;
    if (T == NULL)
        return NULL;
```

```
        if (X == T→Elemento)
            return T;
        P = PesquisarAB(T→Esquerda, X);
        if (P != NULL)
            return P;
        return PesquisarAB(T→Direita, X);
}
```

3.12. Inserir um elemento numa árvore

O algoritmo de inserção de um elemento numa árvore binária começa com uma pesquisa deste elemento na árvore, no sentido de procurar a posição de inserção, sendo que o nodo com o novo elemento é sempre inserido como folha da árvore. De facto, como a árvore não está organizado por nenhuma critério, a pesquisa alcança uma folha, sendo então inserido nodo com o novo elemento nesta posição. No entanto, esta pesquisa terá que obedecer a algum critério, para que a distribuição dos nodos pela árvore seja o máximo possível uniforme (de forma a que árvore não fique totalmente desequilibrada). Ou seja, é examinada a raiz e introduzido um novo nodo na subárvore da esquerda ou na subárvore da direita, consoante o valor do critério de inserção for verdadeiro ou falso. Um critério possível pode ser a altura das subárvores de cada nodo, sendo uma possível função recursiva é a que se segue.

Entrada: ponteiro para a raíz duma árvore e o elemento a inserir

Saída: ponteiro para a raíz da árvore (que pode ter sofrido alteração)

```
PNodoAB InserirPorAlturaAB (PNodoAB T, Info X) {
        if (T == NULL) {
            T = CriarNodoAB(X);
            return T;
        }
        if (AlturaAB(T→Esquerda) > AlturaAB(T→Direita))
            T→Direita = InserirPorAlturaAB(T→Direita, X);
        else
            T→Esquerda = InserirPorAlturaAB(T→Esquerda, X);
        return T;
}
```

O algoritmo deixa claro o processo de inserção: primeiro, verifica-se se a altura da subárvore esquerda é superior à altura da subárvore direita (subárvore esquerda mais profundado que a direita); caso seja verdade insere-se o elemento na subárvore direita e, caso seja falso, insere-se na subárvore esquerda. Este processo continua até que se chegue a um nodo cuja subárvore esquerda ou direita seja vazia, adicionando-se, então, como filho direito ou esquerdo, dependendo da que for vazia (à esquerda se forem ambas).

3.13. Remover um elemento de uma árvore

A operação de remoção de um elemento duma árvore binária é mais complexo que as operações anteriores. Para se remover um elemento de uma árvore binária, deve-se considerar três casos distintos, em função das características do nodo a remover: se é uma folha, se é um nodo com um único filho ou se é um nodo com dois filhos.

3.13.1. Remoção de uma folha

A remoção de um nodo que se encontra no fim da árvore, isto é, que seja uma folha, é o caso mais simples de remoção. Basta remover o nodo da árvore (Figura 5).

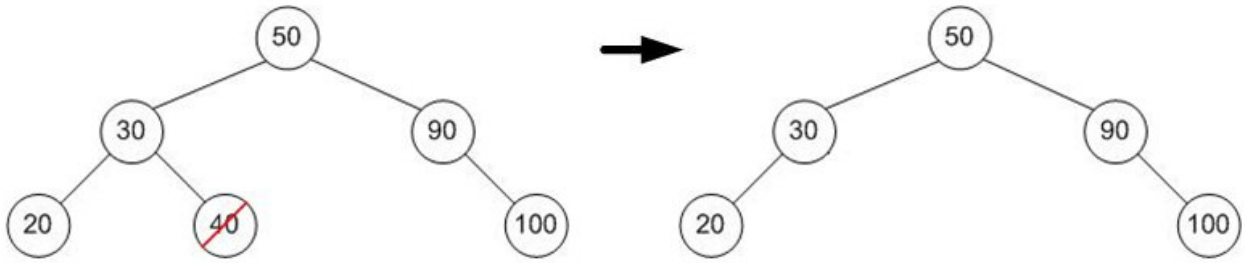


Figura 5 – Remoção de uma folha duma árvore.

3.13.2. Remoção de um nodo com um único filho

Caso o nodo a ser removido tenha um único filho, o pai do nodo (avô do filho) herda o filho. Isto é, o filho do nodo a remover assume a posição do pai na árvore (Figura 6).

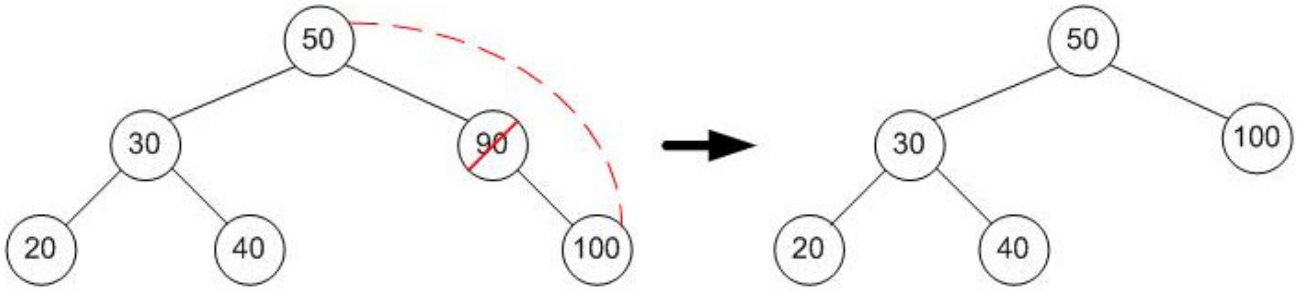


Figura 6 – Remoção de um nodo com um filho duma árvore.

3.13.3. Remoção de um nodo com dois filhos

Se o nodo a ser removido tiver dois filhos, o mecanismo de remoção implica substituir este nodo por uma das folhas sua descendente. Desta forma, esta operação realiza-se da seguinte forma:

1º) escolher uma das folhas descendente do nodo a remover que o irá substituir;

2º) substituir o elemento do nodo a ser removido pelo elemento da folha escolhida;

3º) remover a folha escolhida.

A Figura 7 exemplifica esta operação com a remoção do nodo com elemento 30. As folhas que podem substituir este nodo são as com os elementos 20, 37 e 45 (as três folhas suas descendentes), sendo um destes nodos que será verdadeiramente removido da árvore (na árvore da Figura 7, escolheu-se o nodo com o elemento 37 para substituir o nodo com o elemento 30).

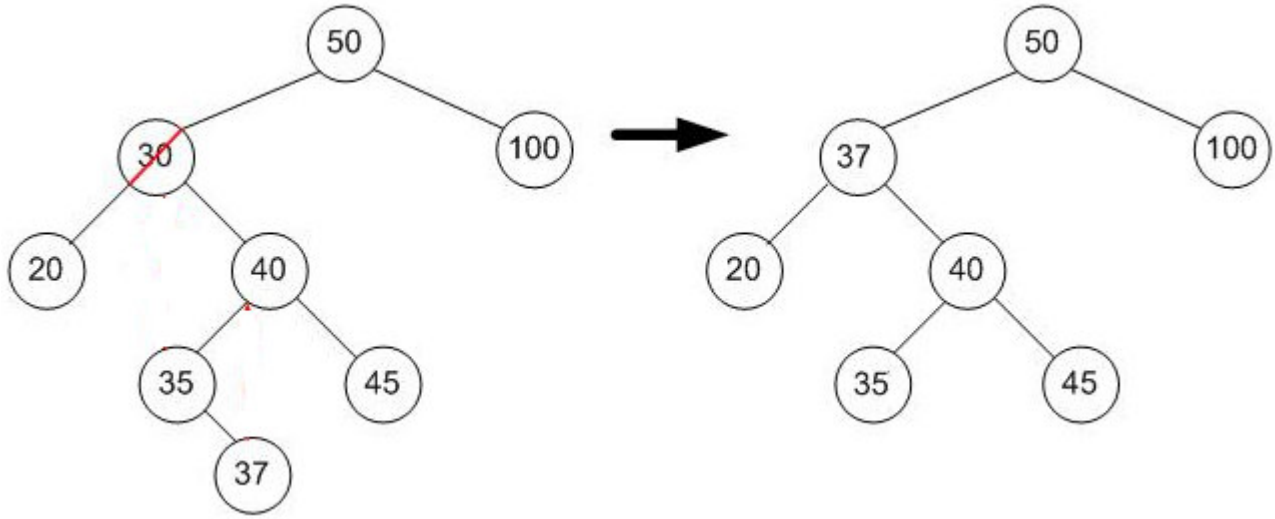


Figura 7 – Remoção de um nodo (30) com dois filhos de uma árvore.

3.13.4. Implementação

A operação de remoção de um elemento de uma árvore binária consiste em procurar o nodo com o elemento a remover e, após o encontrar (pois ele existe), aplicar uma outra operação que consiste em procurar a folha cujo elemento irá substituir o elemento a remover, realizando de seguida a sua substituição e, por fim, remover esta folha. Uma possível função recursiva é a que a seguir se apresenta.

Entrada: ponteiro para a raiz duma árvore e o elemento a remover

Saída: ponteiro para a raiz da árvore (que pode ter sofrido alteração)

Pré-requisito: o elemento a remover existe na árvore

```
PNodoAB RemoverAB (PNodoAB T, Info X) {
    PNodoAB P;
    if (T == NULL)
        return NULL;
    if (T→Elemento == X) {
        T = RemoverNodoAB(T);
        return T;
    }
    P = PesquisarAB(X, T→Esquerda);
    if (P != NULL)
        T→Esquerda = RemoverAB(T→Esquerda, X);
    else
        T→Direita = RemoverAB(T→Direita, X);
    return T;
}
```

Uma possível função para remover um nodo de uma árvore binária é a que se segue.

Entrada: ponteiro para o nodo que se pretende excluir (raiz de subárvore não vazia)

Saída: ponteiro para a raiz da subárvore sem um elemento (pode ter sido alterada)

```
PNodoAB RemoverNodoAB (PNodoAB T) {
    PNodoAB R;
    Info  X;
    if (T→Esquerda == NULL && T→Direita == NULL) {    // T é uma folha
        T = LibertarNodoAB(T);
        return T;
    }
    if (T→Esquerda == NULL) {       //  só um filho direito
        R = T;
        T = T→Direita;
        R = LibertarNodoAB(R);
        return T;
    }
```

```
        if (T→Direita == NULL) {          // só um filho esquerdo
            R = T;
            T = T→Esquerda;
            R = LibertarNodoAB(R);
            return T;
        }
        // 2 filhos: remover a folha escolhida e copiar a sua informação
        P = ProcurarFolhaAB(T, &X);
        T = LibertarNodoAB(P);
        T→Elemento = X;
        return T;
    }
```

A operação de remoção de um elemento com dois filhos implica a sua substituição por uma das folhas sua descendente. Uma possível função recursiva para procurar uma folha descendente do nodo a ser removido é a que se apresenta a seguir.

```
PNodoAB ProcurarFolhaAB (PNodoAB T, Info *X) {
    PNodoAB P;
    if (T→Esquerda == NULL  &&  T→Direita == NULL) {
        *X = T→Elemento;
        return T;
    }
    if (AlturaAB(T→Esquerda) > AlturaAB(T→ Direita))
        return  ProcurarFolhaAB(T→Esquerda, X);
    else
        return  ProcurarFolhaAB(T→ Direita, X);
}
```

3.14. Travessias não recursivos

Os algoritmos associados a travessias da árvore fazem as travessias em profundidade na árvore são recursivos, pois ao fazerem a travessia descendente de um caminho é necessário depois fazer a travessia ascendente para percorrerem outros caminhos ainda não percorridos e não existem ligações para os nodos anteriores (pai) para fazer o percurso de regresso. Para implementar as travessias de forma iterativa, é necessário utilizar uma memória para guardar os nodos já visitados e/ou por visitar, de forma a poder-se regressar

posteriormente a estes nodos. Como o caminho de regresso é normalmente o caminho percorrido no sentido contrário, devemos usar uma EAD Pilha ou uma EAD Fila para tal.

Nas travessia mais comuns, como sejam as em pré-ordem, em-ordem e pós-ordem, utiliza-se uma EAD Pilha para armazenar os nodos já visitados. A seguir apresentam-se três funções iterativas para estes três tipos de travessias, assim como a EAD Pilha necessária.

```
struct PilhaAB {
    PNodoAB Elemento;
    struct PilhaAB *Ant;
}
```

Entrada: ponteiro para a raiz de uma árvore

Saída: mostrar os elementos da árvores

```
void ListarPreOrdem (PNodoAB T) {
    struct PilhaAB S;
    PNodoAB P;
    if (ABVazia(T))
        return;
    S = CriarPilha();
    if (S == NULL)
        return;
    S = Push(S, T);
    while (!PilhaVazia(S)) {
        P = Topo(S);
        S = Pop(S);
        printf(P→Elemento);
        if (P→Direita != NULL)
            S = Push(S, P→Direita);
        if (P→Esquerda != NULL)
            S = Push(S, P→Esquerda);
    }
}
```

```
void ListarEmOrdemAB (PNodoAB T) {
    struct PilhaAB S;
    PNodoAB  P, Subir = (PNodoAB) malloc (sizeof(PNodoAB);
    if (ABVazia(T))
        return;
    S = CriarPilha();
    if (S == NULL)
        return;
    S = Push(S, T);
    while (!PilhaVazia(S)) {
        P = Topo(S);
        if (P == Subir) {
            S = Pop(S);
            if (PilhaVazia(S))
                return;
            P = Topo(S);
            S = Pop(S);
            printf(P→Elemento);
            if (P→Direita != NULL)
                S = Push(S, P→Direita);
            else
                S = Push(S, Subir);
        }
        else
            if (P→Esquerda != NULL)
                S = Push(S, P→Esquerda);
            else {
                printf(P→Elemento);
                S = Pop(S);
                S = Push(S, Subir);
            }
    }
}
```

```
void ListarPosOrdemAB (PNodoAB T) {
    struct PilhaAB  S;
    PNodoAB P, Subir = (PNodoAB) malloc(sizeof(PNodoAB);
    if (ABVazia(T))
        return;
    S = CriarPilha();
    if (S == NULL)
        return;
    S = Push(S, T);
    while (!PilhaVazia(S)) {
        P = Topo(S);
        if (P == Subir) {
            S = Pop(S);
            if (PilhaVazia(S))
                return;
            P = Topo(S);
            printf(P→Elemento);
            S = Pop(S);
        }
        else {
            S = Push(S, Subir);
            if (P→Direita != NULL) {
                S = Push(S, P→Direita);
            if (P→Esquerda != NULL)
                S = Push(S, P→Esquerda);
        }
    }
}
```

Como se pode verificar, as versões recursivas dos algoritmos dos três tipos de travessias anteriores são muito mais fáceis de implementar do que as correspondentes versões iterativas. No entanto, num outro tipo de travessia isso não acontece, como é o caso da travessia por níveis, na qual se utiliza uma EAD Fila (e não uma EAD Pilha). A seguir apresenta-se uma função iterativa para este tipo de travessia, assim como a EAD Fila.

```
struct FilaAB {
        PNodoAB Elemento;
        struct FilaAB *Prox;
}
void ListarPorNiveisAB (PNodoAB T) {
        struct FilaAB  Q;
        PNodoAB  P;
        if (ABVazia(T))
                return;
        Q = CriarFila();
        if (Q == NULL)
                return;
        Q = Juntar(Q, T);
        while (!FilaVazia(Q)) {
                P = Frente(Q);
                printf(P→Elemento);
                Q = Remover(Q);
                if (P→Direita != NULL)
                        Q = Juntar(Q, P→Esquerda);
                if (P→Esquerda != NULL)
                        Q = Juntar(Q, P→Direita);
        }
}
```

4. Árvores Binárias de Pesquisa

4.1. Definição

Uma árvore binária de pesquisa (ABP) (ou "Binary Search Tree" - BST) é uma árvore binária em que os seus nodos têm associado uma chave, que determina a sua posição de colocação na árvore e que obedece às seguintes regras: a chave de um nodo é maior do que a chave de qualquer nodo da sua subárvore esquerda e menor do que a chave de qualquer nodo da sua subárvore direita (árvore em ordem crescente). Desta definição decorre que não podem existir nodos com chaves iguais.

Assim, uma ABP é uma estrutura de dados recursiva e dicotómica (dividida em 2 partes, e cada um destas duas partes pode também ser dividida em duas partes, e assim sucessivamente). Cada nodo pode ser visto como a raiz de duas árvores:

- a árvore esquerda com nodos cujas chaves são inferiores à chave da raiz, e
- a árvore direita com nodos cujas chaves são superiores à chave da raiz.

Desta forma, permite-se a utilização recursiva da pesquisa binária, a partir da sua raiz, até se encontrar o elemento pretendido, ou até se atingida uma árvore nula.

O objetivo de organizar dados em árvores binárias de pesquisa é facilitar a tarefa de procura de um determinado elemento. A partir da raiz e da chave a ser encontrada, é possível saber qual o caminho a ser percorrido até encontrar o elemento/nodo com aquela chave. Para tanto, basta verificar se a chave do elemento procurado é maior ou menor à chave do elemento na posição atual. Não existe uma forma única de organizar os dados (informação) numa árvore binária de pesquisa, pois, dependendo da escolha do nodo raiz, obtêm-se árvores diferentes. Na Figura 8 os elementos { 2, 3, 5, 6, 7, 8 } são organizados em árvores binárias de pesquisa de duas formas diferentes (árvores binárias diferentes).

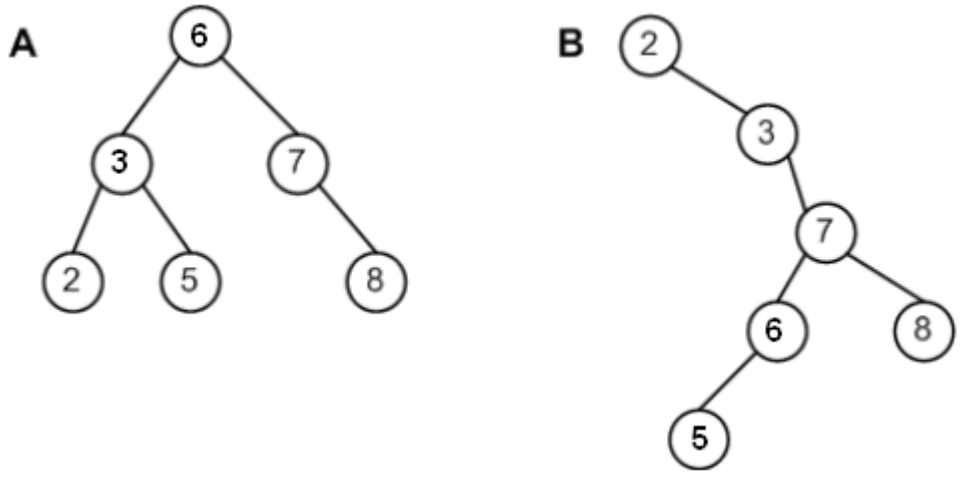


Figura 8 – Árvores binárias de pesquisa diferentes e com os mesmos elementos.

As duas árvores contêm exatamente os mesmos elementos, porém possuem estruturas diferentes. Enquanto a árvore A está enraizada num dos nodos com o elemento 5, a árvore B está enraizada no nodo com o elemento 2. Supondo que se está a pesquisar o elemento 8 nas árvores, as comparações seriam as apresentadas na tabela seguinte:

Árvore A	Árvore B
8 > 5 8 > 7 Encontrado!	8 > 2 8 > 3 8 > 7 Encontrado!

Na árvore A são realizadas menos comparações, relativamente à utilizada na árvore B. O melhor caso irá ocorrer quando a árvore estiver cheia, sendo que, neste caso, a procura de um elemento terá tempo proporcional a *log n*, enquanto que o pior caso ocorrerá quando todos os nodos das árvores apontarem somente para um dos lados, caso em que o tempo de processamento da procura será proporcional a *n*.

Tendo em conta a definição de árvore binária de pesquisa, que implica uma certa ordenação dos seus elementos, as operações sobre este tipo de estruturas de dados têm que obedecer às caraterísticas que lhes estão inerentes.

4.2. Pesquisar um elemento numa árvore

Para procurar numa árvore binária um elemento específico deve-se examinar a raiz. Se o valor for igual à raiz, o elemento existe na árvore. Se o elemento for menor do que a raiz, então deve-se pesquisar na subárvore da esquerda, e assim recursivamente em todos os nodos da subárvore. Da mesma forma, se o elemento for maior do que a raiz, então deve-se pesquisar na subárvore da direita. Até alcançar o nodo-folha da árvore, encontrando-se ou não o elemento requerido. Esta operação efetua *log n* operações no caso médio e *n* no pior caso quando a árvore está desequilibrada; neste caso, a árvore é considerada uma árvore degenerada. Uma possível função recursiva é a que se segue.

Entrada: ponteiro para a raiz duma árvore e o elemento a procurar

Saída: 1 (o elemento existe na árvore); 0 (o elemento não existe na árvore)

```
int PesquisarABP (PNodoAB T, Info X) {
    if (T == NULL)
        return 0;
    if (X == T→Elemento)
        return 1;
    if (X < T→Elemento)
        return PesquisarABP(T→Esquerda, X);
    return PesquisarABP(T→Direita, X);
}
```

4.3. Determinar o maior e o menor elementos de uma árvore

Pelo facto de uma ABP ser uma estrutura de dados que se pode dividir em duas partes, e cada uma destas duas parte se dividir também em duas partes, e assim sucessivamente, leva a que o maior e o menor elementos se encontrem nos nodos mais à

direita e mais à esquerda, respetivamente. A seguir apresenta-se duas possíveis funções recursivas para determinar o maior e o menor elementos de uma ABP.

Entrada: ponteiro para a raiz duma árvore

Saída: ponteiro para o nodo com o maior elemento da árvore

```
PNodoAB NodoMaiorElementoABP (PNodoAB T) {
    if (T == NULL)
        return NULL;
    if (T→Direita == NULL)
        return T;
    return  NodoMaiorElementoABP(T→ Direita);
}
```

Entrada: ponteiro para a raiz duma árvore

Saída: ponteiro para nodo com o menor elemento da árvore

```
PNodoAB NodoMenorElementoABP (PNodoAB T) {
    if (T == NULL)
        return NULL;
    if (T→Esquerda == NULL)
        return T;
    return  NodoMenorElementoABP(T→Esquerda);
}
```

4.4. Inserir um elemento numa árvore

O processo de inserção de um elemento numa ABP tem que obedecer à definição desta, ou seja, a chave do elemento a inserir não pode ainda existir na árvore e a sua posição de inserção tem que assegurar que a chave de um elemento é maior do que as chaves de todos os elementos da sua subárvore esquerda e menor do que as chaves de todos os elementos da sua subárvores direita.

O algoritmo de inserção de um elemento numa árvore binária começa com uma pesquisa deste elemento na árvore, no sentido de procurar a posição de inserção, sendo que o novo nodo com este elemento é sempre inserido como folha da árvore. De facto, como o elemento não existe na árvore, a pesquisa alcança uma folha, sendo então inserido um nodo com este elemento nesta posição. Ou seja, é examinada a raiz e introduzido um novo nodo na subárvore da esquerda (se o valor do novo nodo é menor do que o valor da

raiz) ou na subárvore da direita (se o valor do novo nodo for maior do que o valor da raiz). Uma possível função recursiva é a que se segue.

Entrada: o elemento a inserir e o ponteiro para a raiz duma árvore

Saída: ponteiro para a raiz da árvore (que pode ter sofrido alteração)

Pré-requisito: o elemento a inserir não existe na árvore

```
PNodoAB InserirABP (Info X, PNodoAB T) {
    if (T == NULL) {
        T = CriarNodoAB(X);
        return T;
    }
    if (X < T→Elemento)
        T→Esquerda = InserirABP(X, T→Esquerda);
    else
        T→Direita = InserirABP(X, T→Direita);
    return T;
}
```

Os algoritmos deixam claro o processo de inserção: primeiro, o novo valor é comparado com o valor da raiz; se seu valor for menor que a raiz, é comparado então com o valor do filho da esquerda da raiz; se seu valor for maior, então compara-se com o filho da direita da raiz. Este processo continua até que se chegue a um nodo folha, e então adiciona-se o filho à direita ou à esquerda, dependendo de seu valor ser maior ou menor que o valor da folha.

4.5. Remover um elemento de uma árvore

A remoção de elementos de uma ABP tem que assegurar que, após a remoção do elemento, a árvore continua a ser uma ABP. A operação de remoção de um elemento é mais complexo que as operações anteriores. Para se remover um elemento de uma ABP, deve-se considerar três casos distintos, em função das características do nodo a remover: se é uma folha, se é um nodo com um único filho ou se é um nodo com dois filhos.

Os mecanismos para tratar os dois primeiros casos são os mesmos descritos para o caso de árvores binárias genéricas (Remover um elemento de uma árvore, página 12). O mecanismo para tratar o terceiro caso, remoção de um nodo com dois filhos, tem em conta as características de uma árvore binária de pesquisa (os seus elementos estão organizados segundo um critério).

4.5.1. Remoção de um nodo com dois filhos

Se o nodo a ser removido tiver dois filhos, o mecanismo de remoção implica substituir este nodo pelo nodo com chave mais "próxima" (maior ou menor) da chave do elemento a ser removido. Desta forma, esta operação poderá realiza-se de duas formas diferentes:

- substituir a informação associada ao nodo a ser removido pela informação do seu sucessor, que é o nodo mais à esquerda da subárvore direita (é o menor elemento da subárvore direita, que é maior do que qualquer elemento da subárvore esquerda);
- substituir a informação associada ao nodo a ser removido pelo seu antecessor, que é o nodo mais à direita da subárvore esquerda (é o maior elemento da subárvore esquerda, que é menor do que qualquer elemento da subárvores direita).

Realizada a escolha, remove-se o nodo sucessor (ou antecessor).

A Figura 9 exemplifica esta operação com a remoção do nodo/elemento com chave 30. Este nodo possui como sucessor e antecessor os nodos com chaves 35 e 20, respetivamente. A forma de remoção escolhida é a primeira (substituir pelo seu sucessor). Desta forma, o filho (35) do nodo com chave 40 será promovido ao lugar do nodo a ser excluído (30), o nodo com chave 40 continuará na sua posição e o filho do nodo com chave 35 (no caso o nodo com chave 37) passará a ser filho do nodo com chave 40. Desta forma, o nodo que é ser verdadeiramente removido da árvore é o nodo com chave 35, o que acontece quando o nodo com chave 40 liga-se ao nodo com chave 37 através do seu nodo esquerdo, libertando-se o nodo com chave 35.

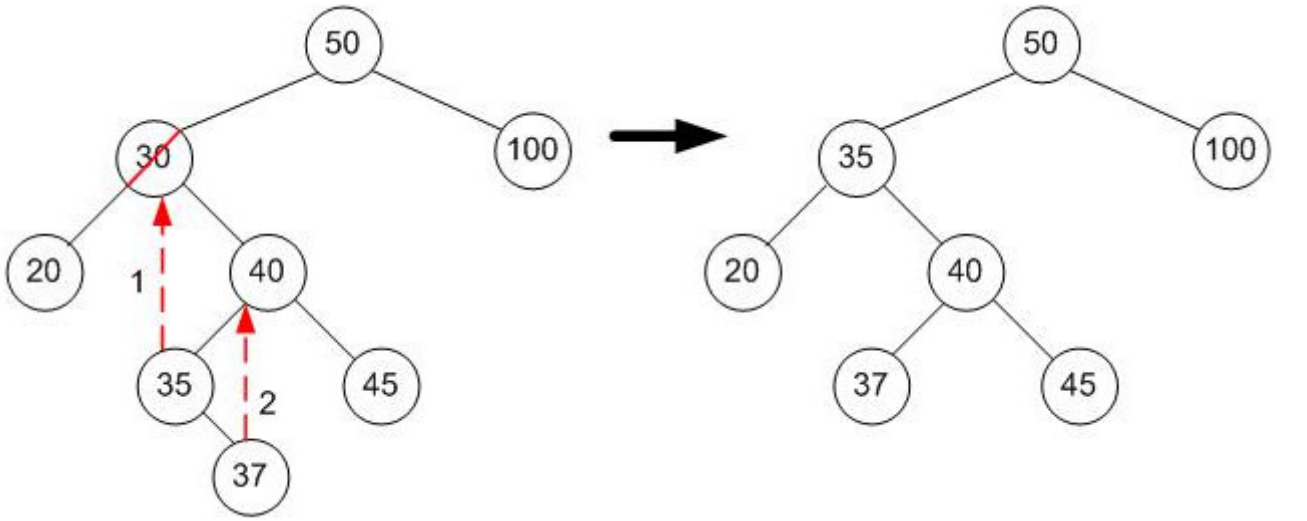


Figura 9 – Remoção de um nodo (30) com dois filhos duma ABP.

4.5.2. Implementação

A operação de remoção de um elemento de uma ABP consiste em procurar o elemento a remover e, após o encontrar (ele existe na árvore), aplicar uma outra operação para remover aquele nodo. Uma possível função recursiva é a que a seguir se apresenta.

Entrada: ponteiro para a raiz duma árvore e o elemento a remover (existe na árvore)
Saída: ponteiro para a raiz da árvore (que pode ter sofrido alteração)

```
PNodoAB RemoverABP (PNodoAB T, Info X) {
    if (T→Elemento == X) {
        T = RemoverNodoABP(T);
        return T;
    }
    if (X < T→Elemento)
        T→Esquerda = RemoverABP(T→Esquerda, X);
    else
        T→Direita = RemoverABP(T→Direita, X);
    return T;
}
```

Uma possível função para remover um nodo de uma ABP é a que a seguir se apresenta.

Entrada: ponteiro para o nodo que se pretende excluir (raiz de subárvore não vazia)
Saída: ponteiro para a raiz da subárvore sem um elemento (pode ter sido alterada)

```
PNodoAB RemoverNodoABP (PNodoAB T) {
    PNodoAB  R;
    Info  X;
    if (T→Esquerda == NULL && T→Direita == NULL) {    // T é uma folha
        T = LibertarNodoAB(T);    // liberta nodo
        return T;                 // devolve NULL
    }
    if (T→Esquerda == NULL) {       //  só um filho direito
        R = T;
        T = T→Direita;
        R = LibertarNodoAB(R);
        return T;
    }
```

```
        if (T→Direita == NULL) {          // só um filho esquerdo
            R = T;
            T = T→Esquerda;
            R = LibertarNodoAB(R);
            return T;
        }
        // 2 filhos: remover o nodo sucessor/antecessor e copiar a sua informação
        T→Direita = SubstituirNodoDireita(T→Direita, &X);          // 1º caso
        T→Esquerda = SubstituirNodoEsquerda(T→Esquerda, &X);       // 2º caso
        T→Elemento = X;
        return T;
    }
```

Como referido antes, a operação de remoção de um elemento com dois filhos implica a sua substituição pelo seu sucessor ou antecessor. Uma possível função recursiva para substituir a informação associada ao nodo a ser removido pela informação do seu *sucessor*, que é o nodo mais à esquerda da subárvore direita é a que se apresenta a seguir.

Entrada: ponteiro para o nodo que se pretende excluir (raiz de subárvore não vazia)

Saída: ponteiro para a raiz da árvore e o elemento do nodo removido

```
PNodoAB SubstituirNodoDireita (PNodoAB T, Info *X) {
    PNodoAB  PAux;
    if (T→Esquerda == NULL) {
        *X = T→Elemento;
        PAux = T;
        T = T→Direita;
        PAux = LibertarNodoAB(PAux);
        return T;
    }
    T→Esquerda = SubstituirNodoDireita(T→Esquerda, X);
    return T;
}
```

Uma possível função recursiva para substituir a informação associada ao nodo a ser removido pelo seu *antecessor*, que é o nodo mais à direita da subárvore esquerda é a que se apresenta a seguir.

Entrada: ponteiro para o nodo que se pretende excluir (raiz de subárvore não vazia)

Saída: ponteiro para a raiz da árvore e o elemento do nodo removido

```
PNodoAB SubstituirNodoEsquerda (PNodoAB T, Info *X) {
    PNodoAB  PAux;
    if (T→Direita == NULL) {
        *X = T→Elemento;
        PAux = T;
        T = T→Esquerda;
        PAux = LibertarNodoAB(PAux);
        return  T;
    }
    T→Direita = SubstituirNodoEsquerda(T→Direita, X);
    return T;
}
```

5. Árvores equilibradas

5.1. Introdução

A pesquisa de um elemento numa árvore binária de pesquisa implica fazer uma travessia da árvore até se encontrar o elemento procurado, ou então se concluir da sua inexistência. As travessias de uma árvore fazem-se em profundidade, pelo que quanto maior é a altura de uma árvore mais demorado será o processo de pesquisa de um elemento, no pior caso e no caso médio. Desta forma, é desejável que as árvores estejam equilibradas em altura, de forma que os algoritmos de manipulação de árvores tenham eficiência logarítmica.

5.2. Inserção "binária"

Uma forma de equilibrar uma árvore binária de pesquisa em altura, consiste em armazenar os elementos da árvore numa sequência linear (lista) usando a travessia em em-ordem e, depois, construir uma nova árvore fazendo uma inserção "binária" dos elementos da sequência. Uma possível função recursiva é a que se segue.

Entrada: um ponteiro para a raiz da árvore

Saída: um ponteiro para a árvore equilibrada

```
PNodoAB CriarArvoreEquilibrada (PNodoAB T) {
    Info *Lista;
    int  N = 0, Num = NumeroNodos(T);
    if (T == NULL)
        return NULL;
    Lista = (Info *) malloc(Num * sizeof(Info));
    if (Lista == NULL)
        return NULL;
    CriarSequenciaEmOrdem(T, Lista, &N);
    T = EquilibrarArvore(Lista, 0, N-1);
    return T;
}
```

Uma possível função recursiva para criar a sequência de elementos de uma árvore binária de pesquisa é a que se apresenta a seguir.

Entrada: um ponteiro para a raiz da árvore

Saída: um vetor do tipo **Info** e o número de elementos deste vetor

```
void CriarSequenciaEmOrdem (PNodoAB T, Info L[], int *N) {
    if (T != NULL) {
        CriarSequenciaEmOrdem(T→Esquerda, L, N);
        L[*N] = T→Elemento;
        *N = (*N) + 1;
        CriarSequenciaEmOrdem(T→Direita, L, N);
    }
}
```

Uma possível função recursiva para equilibrar uma árvore, a partir dos elementos de uma lista, é a que se apresenta a seguir.

Entrada: uma lista de elementos do tipo **Info** e os índices inicial e final desta lista

Saída: um ponteiro para a raiz da árvore equilibrada

```
PNodoAB EquilibrarArvore (Info L[], int inicio, int fim) {
    PNodoAB T;
    int  medio;
    if (inicio > fim)
        return NULL;
```

```
        if (inicio == fim) {
            T = InserirABP(L[inicio], T);
            return T;
        }
        medio = (inicio + fim) / 2;
        T = InserirABP(L[medio], T);
        T→Esquerda = EquilibrarArvore(L, inicio, medio-1);
        T→Direita = EquilibrarArvore(L, medio+1, fim);
        return T;
}
```

5.3. Árvore de Adelson-Velskii Landis (AVL)

Uma vez que o equilíbrio perfeito de uma árvore, que acontece quando as subárvores esquerda e direita têm a mesma altura, exigiria algoritmos de inserção e de remoção muito complexos, é conveniente definir uma condição de equilíbrio que seja fácil de manter.

Uma boa condição de equilíbrio consiste em definir uma árvore equilibrada em altura como sendo uma árvore em que a diferença das alturas entre as subárvores esquerda e direita, em cada nodo, não seja superior a uma unidade (Figura 10 – a), b)). Desta forma, a árvore com a melhor altura é aquela que tem o maior número de nodos para uma qualquer altura, ou seja, a árvore binária completa (Figura 10.(a)).

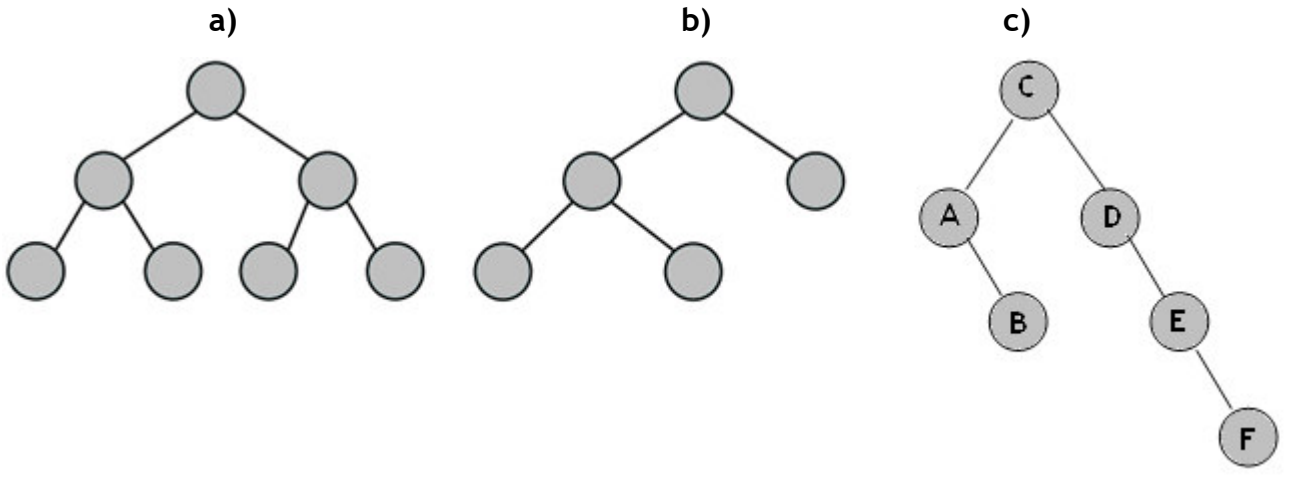


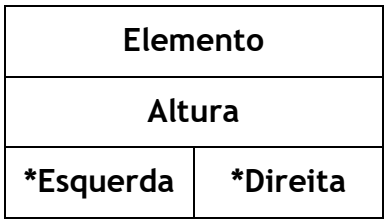
Figura 10 – Exemplo de árvores equilibradas e não equilibradas em altura.

Um exemplo de uma árvore que não cumpre com a condição de equilíbrio é a que se apresenta na Figura 10 – c). Apesar da árvore da Figura 10 – c) estar equilibrada na raiz (a altura das subárvores esquerda e direita é 2 e 3, respetivamente), existe desequilíbrio na subárvore com raiz no nodo D (a altura das subárvores esquerda e direita é 0 e 2,

respetivamente). Desta forma, é apenas neste nodo que se devem aplicar os mecanismos de reequilíbrio que serão estudados mais à frente.

A árvore de Adelson-Velskii Landis, simplesmente designada por árvore AVL, é uma árvore equilibrada em altura que obedece àquele critério de equilíbrio (Figura 10). Para implementá-la é preciso guardar em cada nodo a sua altura, que representa a altura da subárvore com raiz nesse nodo. Logo, o nodo da AVL é idêntico ao nodo de uma ABP, à qual se juntou o campo *Altura*. A seguir apresenta-se uma possível declaração de um nodo AVL.

```
struct NodoAVL {
    Info Elemento;
    int Altura;
    struct NodoAVL *Esquerda;
    struct NodoAVL *Direita;
};
typedef struct NodoAVL *PNodoAVL;
```



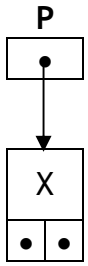
As operações a realizar sobre uma árvore AVL são semelhantes às das árvores binárias comuns. Desta forma, apenas se apresentam algumas, as que são referidas nas funções descritas neste documento.

5.3.1. Criar um nodo de uma árvore AVL

Entrada: a informação que se pretende guardar

Saída: um ponteiro para um nodo (informação + ligações)

```
PNodoAVL CriarNodoAVL (Info X) {
    PNodoAVL P = (PNodoAVL) malloc(sizeof(struct NodoAVL));
    if (P == NULL)
        return NULL;
    P→Elemento = X;
    P→Altura = 1;
    P→Esquerda = NULL;
    P→Direita = NULL;
    return P;
}
```



5.3.2. Destruir um nodo de uma árvore

Entrada: um ponteiro para o nodo que se pretende destruir

Saída: o ponteiro a apontar para NULL

```
PNodoAVL LibertarNodoAVL (PNodoAVL P) {
    P→Esquerda = NULL;
    P→Direita = NULL;
    free(P);
    P = NULL;
    return P;
}
```

5.3.3. Altura de uma árvore AVL

Atendendo à definição apresentada, os algoritmos de inserção e remoção de um nodo de uma árvore AVL implica a atualização da altura dos nodos da árvore afetados por aquelas operações. Desta forma, o algoritmo associado à operação de determinar a altura de uma árvore AVL fica simplificado, pois basta consultar o campo "Altura" da raiz da árvore. Uma possível função para realizar esta operação é a que se apresenta a seguir.

Entrada: um ponteiro para a raiz da árvore AVL

Saída: um valor inteiro correspondente à altura da árvore

```
int  AlturaAVL (PNodoAVL  T) {
    if (T == NULL)
        return 0;
    return T→Altura;
}
```

5.3.4. Inserção de um nodo numa árvore AVL

A inserção de um nodo numa árvore AVL é sempre feita como folha da árvore, o que implica realizar duas operações, após a sua inserção:

1º) atualizar a altura dos nodos já existentes na árvore;

2º) verificar a necessidade de equilibrar a árvore.

Para a atualização da altura dos nodos da árvore basta, no pior caso, atualizar a altura dos nodos que definem o caminho da raiz da árvore ao nodo (folha) inserido. No entanto, quando o nodo é inserido na subárvore de um nodo de menor altura, não é necessário atualizar a altura de todos os nodos daquele caminho (apenas os nodos desta subárvore).

A Figura 11 apresenta a inserção dos elementos 2, 5 e 9 numa árvore AVL. Após a inserção do elemento 2 na árvore vazia (Figura 11 - a)), segue-se a inserção do elemento 5 (Figura 11 – b)). Neste momento, a árvore obtida está equilibrada, sendo que a altura da raiz foi sucessivamente atualizada (inicialmente com, 0 passou para 1 e depois para 2). Quando o nodo 9 é inserido na árvore (Figura 11 – c)), a árvore fica desequilibrada na raiz, pois a altura das subárvores esquerda e direita da raiz é 0 e 2, respetivamente. De notar que a altura dos nodos com os elementos 2, 5 e 9 são 3, 2 e 1, respetivamente.

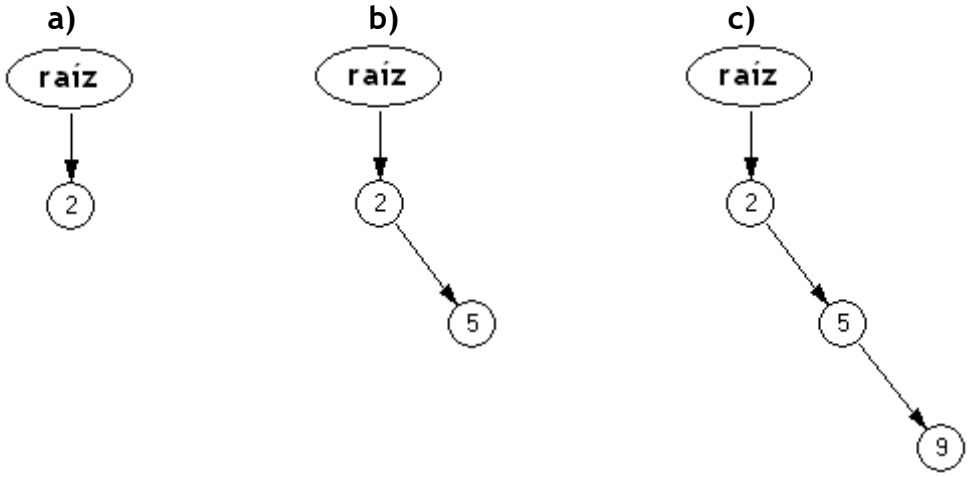


Figura 11 – Inserção de elementos numa árvore AVL.

Para reequilibrar a árvore, é necessário efetuar uma rotação simples à esquerda do nodo com o elemento 2, usando o seu filho direito, o nodo com elemento 5 (Figura 12).

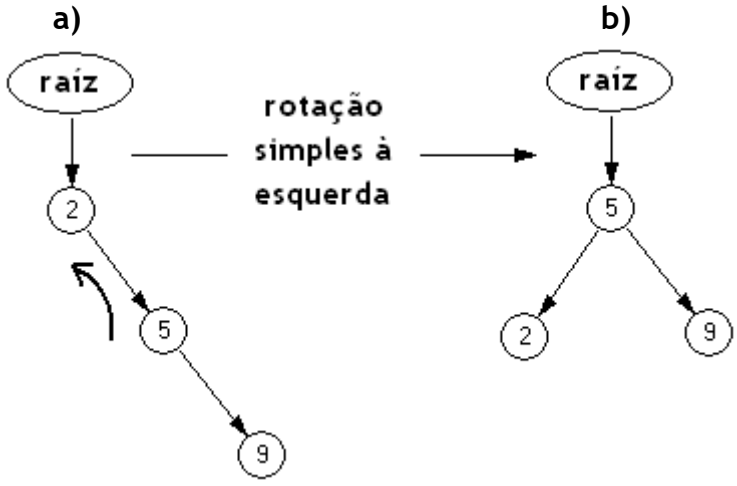


Figura 12 – Reequilíbrio de uma árvore AVL com rotação simples à esquerda.

Após a rotação, a árvore fica equilibrada com altura 2 na raiz, que passou a ser o filho direito do nodo 2, isto é, o nodo 5. Desta forma, a altura dos nodos da subárvore com raiz no elemento 5 é 1, para os nodos com os elementos 2 e 9, e 2 para o nodo com elemento 5.

Ao inserir-se um nodo com o elemento 8 (com altura 1) na árvore da Figura 12 a altura dos nodos com os elementos 5 e 9 é atualizada para 3 e 2, respetivamente (Figura 13 – a)), mantendo a árvore equilibrada. No entanto, com a inserção dum nodo com o elemento 7 (com altura 1) na árvore, o que implica a atualização da altura dos nodos com os elementos 5 (de 3 para 4), 9 (de 2 para 3) e 8 (de 1 para 2), esta fica desequilibrada nos nodos com os elementos 5 e 9 (Figura 13 – b)): a altura das subárvores esquerda e direita do nodo com o elemento 5 (raiz) têm altura 1 e 3, respetivamente, e a altura das subárvores esquerda e direita do nodo com elemento 9 têm altura 2 e 0, respetivamente.

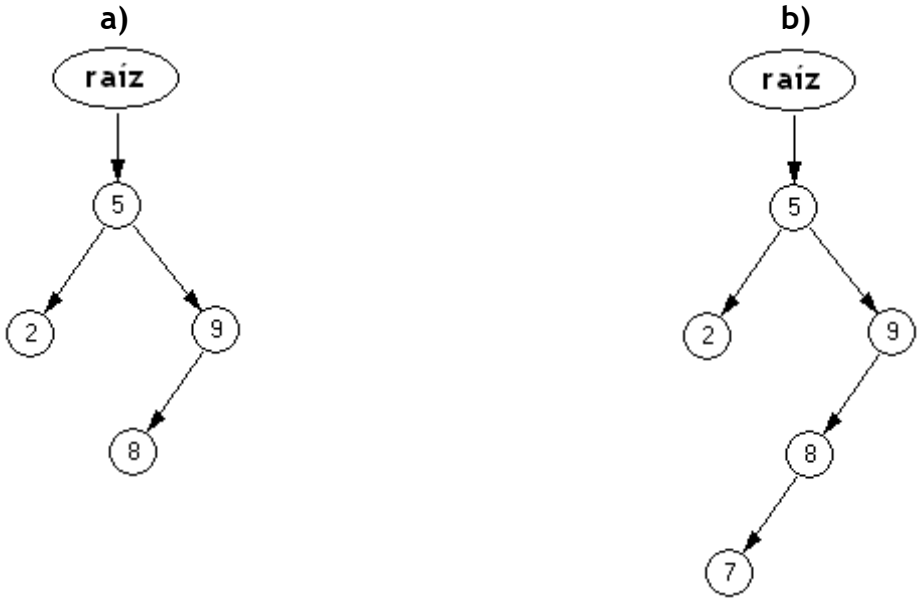


Figura 13 – Inserção de elementos numa árvore AVL.

Desta forma, é necessário realizar uma rotação simples à direita do nodo com o elemento 9, usando o seu filho esquerdo, o nodo com o elemento 8 (Figura 14), passando este a ser a raiz desta subárvore. Após esta rotação, a árvore fica equilibrada no nodo com o elemento 8 com altura 2 (a altura do nodo com o elemento 9 passou de 2 para 1, mantendo-se a altura do nodo com o elemento 7 em 1). Com esta rotação a árvore também fica equilibrada na raiz (nodo com o elemento 5), o que não acontecia antes, pois a altura da sua subárvore direita passou de 3 para 2 e a subárvore esquerda mantém a altura em 1.

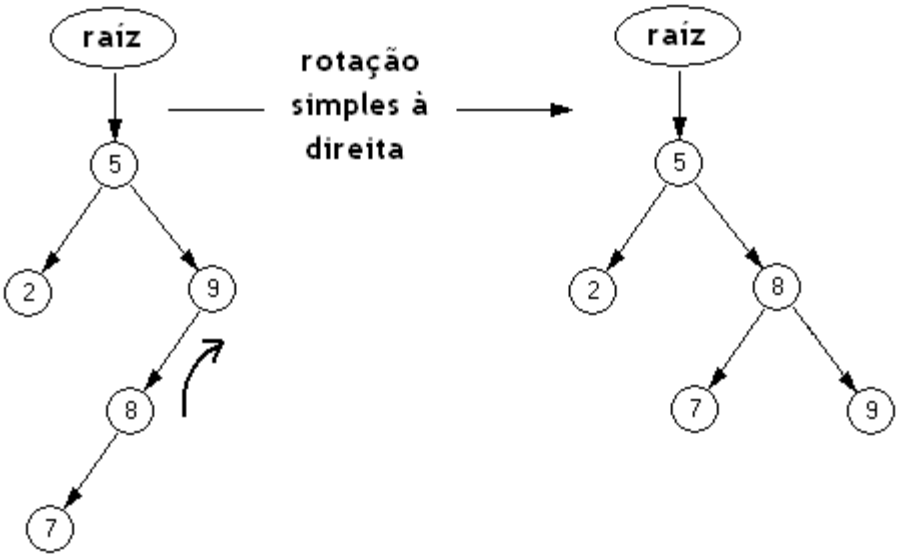


Figura 14 - Reequilíbrio de uma árvore AVL com rotação simples à direita.

Se de seguida forem, por exemplo, inseridos nodos com os elementos 1 e 3, por qualquer ordem, a árvore mantém-se equilibrada e com altura 3, pois estes nodos serão filhos do nodo com o elemento 2.

Nestes casos apresentados está-se perante uma **inserção externa**, pois o nodo é inserido à esquerda do filho esquerdo ou à direita do filho direito, sendo que a árvore facilmente se torna equilibrada (quando o seu desequilíbrio acontece após esta inserção) com uma única rotação, à direita ou à esquerda, de um único nodo. Esta operação designa-se por **rotação simples**.

Uma possível função para realizar a operação de rotação simples à esquerda é a que se apresenta a seguir.

Entrada: ponteiro para o nodo da árvore AVL onde se deteta desequilíbrio

Saída: ponteiro anterior atualizado

```
PNodoAVL  RotacaoSimplesEsquerda (PNodoAVL  P) {
    int  AltEsq, AltDir;
    PNodoAVL  NodoAux = P→Direita;
    P→Direita = NodoAux→Esquerda;
    NodoAux→Esquerda = P;
    // atualizar a altura dos nodos envolvidos na rotação
    AltEsq = AlturaAVL(P→Esquerda);
    AltDir = AlturaAVL(P→Direita);
```

```
        if (AltEsq > AltDir)
            P→Altura = AltEsq + 1;
        else
            P→Altura = AltDir + 1;
        AltEsq = P→Altura;
        AltDir = AlturaAVL(NodoAux→Direita);
        if (AltEsq > AltDir)
            NodoAux→Altura = AltEsq + 1;
        else
            NodoAux→Altura = AltDir + 1;
        return NodoAux;
    }
```

Uma possível função para realizar a operação de rotação simples à direita é a que se apresenta a seguir.

Entrada: ponteiro para o nodo da árvore AVL onde se deteta desequilíbrio

Saída: ponteiro anterior atualizado

```
    PNodoAVL RotacaoSimplesDireita (PNodoAVL P) {
        int AltEsq, AltDir;
        PNodoAVL NodoAux = P→Esquerda;
        P→Esquerda = NodoAux→Direita;
        NodoAux→Direita = P;
        // atualizar a altura dos nodos envolvidos na rotação
        AltEsq = AlturaAVL(P→Esquerda);
        AltDir = AlturaAVL(P→Direita);
        if (AltEsq > AltDir)
            P→Altura = AltEsq + 1;
        else
            P→Altura = AltDir + 1;
        AltEsq = AlturaAVL(NodoAux→Esquerda);
        AltDir = P→Altura;
        if (AltEsq > AltDir)
            NodoAux→Altura = AltEsq + 1;
```

```
        else
            NodoAux→Altura = AltDir + 1;
        return NodoAux;
    }
```

Ao inserir-se um nodo com o elemento 6 na AVL da Figura 14, este será colocado à esquerda do nodo com o elemento 7, sendo que analisando o caminho em direção à raiz da árvore verifica-se que esta continua equilibrada no nodo com o elemento 8 (a diferença de alturas das suas subárvores é apenas 1 unidade), mas não no nodo com o elemento 5, a raiz da árvore (ver Figura 15-a)). De facto, neste nodo a árvore passou a estar desequilibrada, pois a diferença de alturas das suas subárvores é de 2 unidades (a esquerda tem 1 e a direita 3). Desta forma, é necessário realizar uma rotação simples à direita do nodo com o elemento 8, usando o seu filho esquerdo, o nodo com o elemento 7 (ver Figura 15-b)). Mas, mesmo depois desta rotação a árvore continua desequilibrada na raiz (nodo com o elemento 5), sendo necessário realizar uma segunda rotação simples, agora à esquerda, do filho direito da raiz, ou seja, do nodo com o elemento 7. Após esta segunda rotação, a árvore fica também equilibrada na raiz, que agora contém o elemento 7, e não o 5 como inicialmente, mantendo a altura 3 (ver Figura 15-c)). Nesta situação, uma rotação simples não resolveu o problema, tendo sido necessário uma segunda rotação simples em sentido contrário; ou seja, foi necessário uma rotação dupla, que neste caso foi direita-esquerda.

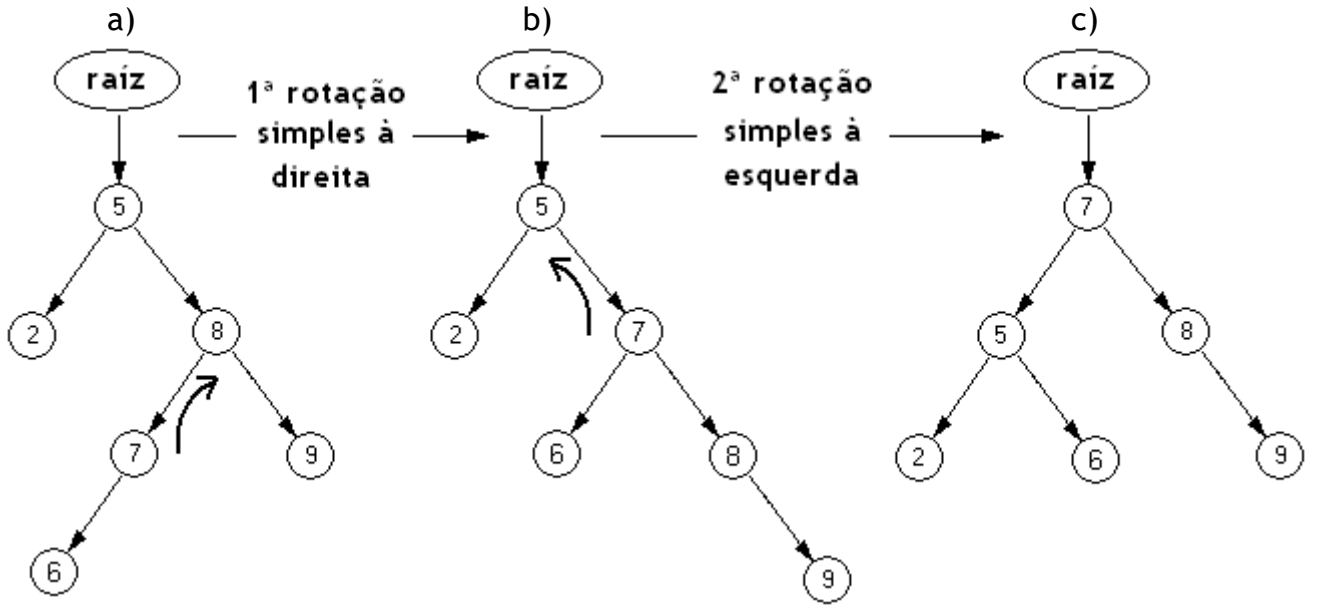


Figura 15 – Inserção de um nodo numa AVL com rotação dupla direita-esquerda.

Tal como existem rotações duplas direita-esquerda, também existem rotações duplas esquerda-direita. A Figura 16 apresenta o exemplo da inserção do nodo com o elemento 6 numa AVL, na qual foram previamente inseridos nodos com os elementos 7, 4, 8, 1 e 5.

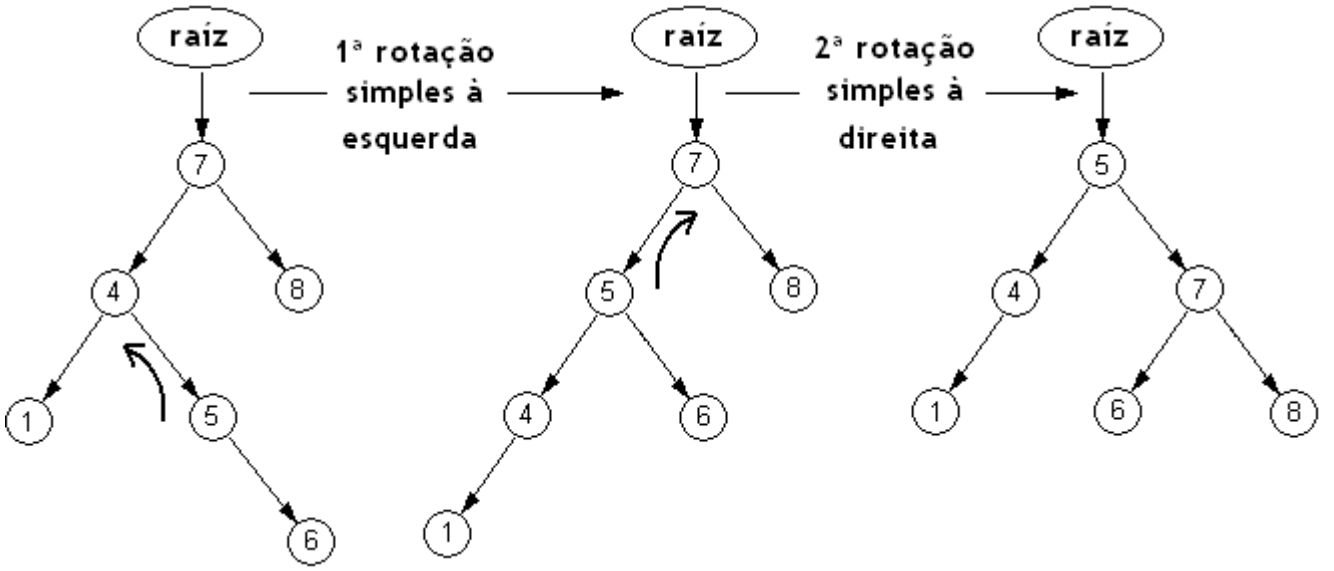


Figura 16 – Inserção de um nodo numa AVL com rotação dupla esquerda-direita.

Uma possível função para realizar a operação de rotação dupla direita-esquerda é a que se apresenta a seguir.

Entrada: ponteiro para o nodo da árvore AVL onde se deteta desequilíbrio

Saída: ponteiro anterior atualizado

```
PNodoAVL  RotacaoDuplaDireitaEsquerda (PNodoAVL  P) {
    P→Direita = RotacaoSimplesDireita(P→Direita);
    P = RotacaoSimplesEsquerda(P);
    return P;
}
```

Uma possível função para realizar a operação de rotação dupla esquerda-direita é a que se apresenta a seguir.

Entrada: ponteiro para o nodo da árvore AVL onde se deteta desequilíbrio

Saída: ponteiro anterior atualizado

```
PNodoAVL  RotacaoDuplaEsquerdaDireita (PNodoAVL  P) {
    P→Esquerda = RotacaoSimplesEsquerda(P→Esquerda);
    P = RotacaoSimplesDireita(P);
    return P;
}
```

Depois de se estudar as várias possibilidades de rotação dos nodos para reequilibrar uma AVL, vai-se estudar o processo global deste reequilíbrio. Este processo começa por calcular as alturas das subárvores do nodo onde poderá existir desequilíbrio, para depois determinar se existe um desequilíbrio na árvore naquele nodo. Caso existe, realiza a rotação adequada, a qual também atualiza a altura dos nodos envolvidos na rotação afetuada. Pelo contrário, se a árvore continua equilibrada, então é necessário atualizar a altura do nodo inspecionado.

Existem dois tipos de desequilíbrios num nodo de uma árvore AVL: **externo** e **interno**. O ***desequilíbrio externo*** acontece após uma inserção externa ou uma remoção interna, sendo corrigido com uma rotação simples. O ***desequilíbrio interno*** acontece após uma inserção interna ou uma remoção externa, sendo corrigido com uma rotação dupla.

Se num dado nodo existe desequilíbrio na sua subárvore esquerda, então é necessário realizar uma rotação à direita; se a altura da subárvore esquerda é maior ou igual à altura da subárvore direita, do filho esquerdo deste nodo, então existe um desequilíbrio externo, senão existe um desequilíbrio interno. Da mesma maneira, se o desequilíbrio ocorrer na subárvore direita daquele nodo, é necessário fazer uma rotação à esquerda; se a altura da subárvore direita é maior ou igual à altura da subárvore esquerda, do filho direito do nodo, então existe um desequilíbrio externo, senão existe um desequilíbrio interno. Pelo contrário, se não existe qualquer desequilíbrio no nodo, então apenas será atualizado a sua altura.

Uma possível função para realizar a operação de reequilíbrio ou atualização da altura de um dado nodo é a que se apresenta a seguir.

Entrada: ponteiro para o nodo da árvore AVL onde se deteta desequilíbrio (subárvore)
Saída: ponteiro anterior atualizado (se realizou reequilíbrio)

```
PNodoAVL  EqulibrarAVL (PNodoAVL  T) {
    int  AltEsq, AltDir;
    if (T == NULL)
        return T;
    AltEsq = T→Esquerda;
    AltDir = T→Direita;
    if (AltEsq – AltDir == 2) {   // subárvore esquerda desequilibrada
        AltEsq = AlturaAVL(T→Esquerda→Esquerda);
        AltDir = AlturaAVL(T→Esquerda→Direita);
```

```
            if (AltEsq > AltDir)
                T = RotacaoSimplesDireita(T);
            else
                T = RotacaoDuplaEsquerdaDireita(T);
            return T;
        }
        if (AltDir – AltEsq == 2) {   // subárvore direita desequilibrada
            AltEsq = AlturaAVL(T→Direita→Esquerda);
            AltDir = AlturaAVL(T→Direita→Direita);
            if (AltDir > AltEsq)
                T = RotacaoSimplesEsquerda(T);
            else
                T = RotacaoDuplaDireitaEsquerda(T);
            return T;
        }
        if (AltDir > AltEsq)
            T = RotacaoSimplesEsquerda(T);
        else
            T = RotacaoDuplaDireitaEsquerda(T);
        return T;
    }
```

A operação de inserção de um elemento numa árvore AVL é muito semelhante à apresentada para o caso de uma árvore binária de pesquisa (secção 4.4, pág. 22), à qual se acrescentou a operação de reequilíbrio. Na travessia da árvore é usada a pesquisa binária recursiva para determinar se o elemento a inserir ainda não existe na árvore, parando no nodo exterior de inserção, caso não exista. Depois de inserir um nodo com o elemento na árvore, que é uma folha, e no caminho de retorno desta folha até à raiz, a operação de reequilíbrio (função ***EquilibrarAVL***) é realizada de forma a reequilibrar a árvore e a atualizar as alturas dos nodos envolvidos, caso sejam necessários.

Uma possível função para realizar a operação de inserção de um elemento numa AVL é a que se apresenta a seguir.

Entrada: ponteiro para o raíz da árvore AVL e o elemento a inserir

Saída: ponteiro da raiz da árvore AVL atualizado

```
PNodoAVL  InserirElementoAVL (PNodoAVL  T, Info X) {
    if (T == NULL) {
        T = CriarNodoAVL(X);
        return T;
    }
    if (T→Elemento > X)
        T→Esquerda = InserirElementoAVL(T→Esquerda, X);
    else
        if (T→Elemento < X)
            T→Direita = InserirElementoAVL(T→Direita, X);
        else
            return T;        // X já existe na árvore
    T = EqulibrarAVL(T);     // reequilibrar a árvore
    return T;
}
```

5.3.5. Remoção de um nodo de uma árvore AVL

A remoção de nodos/elementos de uma árvore AVL provoca o efeito simétrico da inserção de nodos/elementos. Desta forma, a **remoção interna** (remoção à direita do filho esquerdo ou à esquerda do filho direito) de um nodo, desequilibra a árvore da mesma forma que a inserção externa, pelo que é reequilibrada com uma rotação simples. A **remoção externa** (remoção à esquerda do filho esquerdo ou à direita do filho direito) de um nodo, desequilibra a árvore da mesma forma que a inserção interna, pelo que é reequilibrada com uma rotação dupla.

O mecanismo de remoção de um elemento de uma árvore AVL é muito mais complexo do que o de inserção, pois, enquanto que a inserção requer, quanto muito, uma rotação (simples ou dupla) para reequilibrar a árvore, a remoção pode requerer uma rotação em cada nodo visitado. Desta forma, este processo implica, para além da remoção do nodo, a atualização das alturas dos nodos pertencentes ao caminho entre o nodo removido e a raiz da árvore e, se necessário, aplicar as rotações adequadas para reequilibrar a árvore. Apesar disto, com exceção do reequilíbrio da árvore, este mecanismo é semelhante ao associado à remoção de um elemento de uma árvore binária de pesquisa (secção 3.13, pág. 12) quer relativamente à remoção do nodo, quer ao ajuste das ligações.

Assim sendo, o mecanismo de remoção de um elemento de uma árvore AVL combina os mecanismos de remoção de um elemento duma árvore binária de pesquisa e de equilíbrio de uma árvore AVL. Uma possível função para realizar a operação de remoção de um elemento de uma AVL é a que se apresenta a seguir.

Entrada: ponteiro para o raiz da árvore AVL e o elemento a remover

Saída: ponteiro da raiz da árvore AVL atualizado

```
PNodoAVL  RemoverElementoAVL (PNodoAVL  T, Info *X) {
    if (T == NULL)   // a árvore está vazia ou X não está na árvore
        return T;
    if (T→Elemento > *X)
        T→Esquerda = RemoverElementoAVL(T→Esquerda, X);
    else
        if (T→Elemento < *X)
            T→Direita = RemoverElementoAVL(T→Direita, X);
        else {      // copiar e eliminar o elemento
            *X = T→Elemento;
            T = RemoverNodoAVL(T);
        }
    T = EqulibrarAVL(T);       // reequilibrar a árvore
    return T;
}
```

O mecanismo de remover um nodo de uma árvore AVL (função ***RemoverNodoAVL*** usada na remoção) é muito semelhante ao usado em árvores binárias de pesquisa (secção 4.5.2, pág. 25). Quando o nodo com o elemento a remover tem dois filhos, então é necessário substituir o elemento a remover pelo menor elemento da sua subárvore direita (usando a função ***SubstituirNodoMinAVL***).

Uma possível função para realizar a operação de remoção de um nodo de uma AVL é a que se apresenta a seguir.

Entrada: ponteiro para o pai do nodo com o elemento da árvore AVL a remover

Saída: ponteiro para o pai do nodo removido atualizado

```
PNodoAVL  RemoverElementoAVL (PNodoAVL  T) {
    PNodoAVL  NodoAux = T;
    if (T→Esquerda == NULL  && T→Direita == NULL)     // remover uma folha
        T = DestruirNodoAVL(T);
    else
        if (T→Esquerda == NULL) {       // só subárvore direita
            T = T→Direita;
            NodoAux = DestruirNodoAVL(NodoAux);
        }
        else
            if (T→Direita == NULL) {   // só subárvore esquerda
                T = T→Esquerda;
                NodoAux = DestruirNodoAVL(NodoAux);
            }
            else  // 2 subárvores não vazias:
                //  substituir pelo menor elemento da subárvore direita
                T = SubstituirNodoMinAVL(T→Direita, T→Elemento);
    return T;
}
```

A operação de substituir um elemento pelo menor elemento da subárvore direita do nodo com aquele elemento, consiste em pesquisar recursivamente o nodo mais à esquerda das subárvores direitas desde aquele nodo, em copiar o elemento do nodo substituto para o elemento do nodo a remover, ligar a subárvore direita do nodo substituto à subárvore esquerda do nodo anterior e destruir o nodo substituto. Como esta operação pode desequilibrar a árvore, é usada no fim a operação de reequilíbrio de uma árvore AVL. Uma possível função para realizar a operação de substituir um elemento pelo menor elemento da subárvore direita de um nodo de uma AVL é a que se apresenta a seguir.

Entrada: ponteiro para o raiz da árvore AVL e o elemento a substituir

Saída: ponteiro da raiz da árvore AVL atualizado

```
PNodoAVL  SubstituirNodoMinAVL (PNodoAVL  T, Info *X) {
    PNodoAVL  NodoAux = T;
    if (T→Esquerda == NULL) {
        *X = T→Elemento;
        T = T→Direita;
        NodoAux = DestruirNodoAVL(NodoAux);
    }
    else
        T→Esquerda = SubstituirNodoAVL(T→Esquerda, X);
    T = EqulibrarAVL(T);
    return T;
}
```

6. Árvores N-árias

Uma **árvore N-aria T** é um conjunto finito de *nodos* com as seguintes propriedades:

1. O conjunto é vazio, $T = \varnothing$; ou
2. O conjunto consiste numa raiz, *R*, e exatamente *N* árvores *N*-arias distintas. Os restantes nós são repartidos em $N \geq 0$ subconjuntos, T_0, T_1, ..., T_{N-1}, em que cada qual é uma árvore *N*-aria tal que

$$T = \{ R; T_0; T_1; ..., T_{N-1} \}$$

Segundo esta definição, uma ***árvore binária T*** define-se como um conjunto finito de *nodos* com as propriedades:

1. O conjunto T é vazio, $T = \varnothing$; ou
2. O conjunto T consiste numa raiz, *R*, e exatamente *duas* árvores *binárias* distintas $\mathbf{T_L}$ e $\mathbf{T_R}$. A árvore T_L é chamada *árvore esquerda* de *T* e a árvore T_R é chamada de *árvore direita* de *T*.

$$T = \{ R; T_L; T_R \}$$