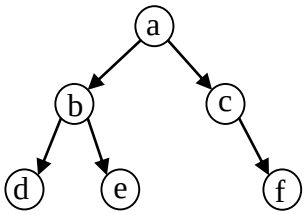
ÁRVORE BINÁRIA

Uma árvore binária A é um conjunto de elementos denominados nós tal que:

- A = ∅ ou
- A é constituído por um nó denominado raiz e duas árvores binárias identificadas por subárvore esquerda e subárvore direita.

Exemplo:

A = {a,{b,{d,∅,∅},{e,∅,∅}},{c,∅,{f,∅,∅}}}



Um nó y é filho de um nó x se y é raiz da subárvore esquerda ou da subárvore direita da árvore de raiz x. Um nó x é pai de um nó y se y é filho de x.

O nó raiz de uma árvore é o único nó que não tem pai.

Um nó x é uma folha se não tem filhos.

Um nó x é um nó interior de uma árvore se x tem pelo menos um filho.

Um caminho de um nó a até um nó b é uma sequência de nós $x_1,x_2,x_3...x_k$, tal que x_{i+1} é filho de x_i, com i = 1,2,3...k-1, x_1 = a e x_k = b. O nó a é denominado origem do caminho e o nó b é denominado término do caminho. O comprimento do caminho é igual a k-1.

Um nó z é descendente do nó x se existe um caminho de origem x e término z.

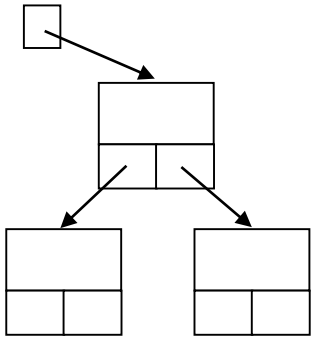
A altura de um nó x é o comprimento do caminho mais longo do nó x até uma folha. A altura da árvore é a altura do nó raiz.

O nível (ou profundidade) de um nó x é o comprimento do caminho do nó raiz até o nó x. O nível do nó raiz é igual a zero.

Estrutura de armazenamento de dados

```
typedef struct No{
        int elemento;
        struct No * esq;
        struct No * dir;
} No;

typedef No* Arvore;
```



O tipo **Arvore** é um pointer para um **No** ou seja, uma Arvore é a referência para o nó raiz de uma árvore.

O tipo **No** é um struct com três campos: um campo do tipo **int**, identificado por **elemento**, dois campos do tipo pointer para **No**, denominados, respectivamente, **esq** e **dir**. Esses são as referências para as raízes das duas subárvores Esquerda e Direita.

A função criarArvoreVazia deve criar uma árvore vazia. Para isso, basta colocar o valor NULL na referência da raiz da árvore.

Para percorrer os dados em uma árvore é preciso estabelecer uma ordem de varredura. Os quatro algoritmos sugeridos a seguir fazem varreduras em ordens diferentes. Se aplicados à árvore desenhada acima, produzem os acessos:

1) d, b, e, a, c, f 2) a, b, d, e, c, f 3) d, e, b, f, c, a 4) a, b, c, d, e, f

Algoritmos de varreduras em árvores binárias:

1) O algoritmo eRd varre os nós de uma árvore binária na ordem subárvore Esquerda,Raiz,subárvore Direita (caminhamento INORDER ou simétrico).

```
void eRd(Arvore a){
    if (a != NULL) {eRd(a->esq); mostrarElemento(a); eRd(a->dir);}
}
```

2) O algoritmo Red(a) varre os nós de uma árvore binária na ordem Raiz,subárvore Esquerda,subárvore Direita (caminhamento PREORDEM).

3) O algoritmo edR(a) varre os nós de uma árvore binária na ordem subárvore Esquerda,subárvore Direita,Raiz (caminhamento POSORDEM).

4) O algoritmo bfs(a) varre os nós de uma árvore binária por nível, a partir do nó raiz (nível 0) e na ordem esquerda-direita (caminhamento breadth first search).

Exercícios

1) Implementar o tipo Arvore escrevendo as 5 funções dadas a seguir.

```
#include <stdlib.h>
#include "Booleano.h"

typedef struct No{
        int elemento;
        struct No * esq;
        struct No * dir;
} No;
typedef No* Arvore;
```

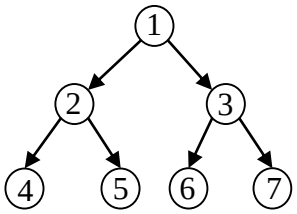


figura 1

```
// protótipos das funções
  Arvore criarArvoreVazia( );
  bool verificarArvoreVazia(Arvore);
  void mostrarArvore(Arvore);
  void mostrarRaiz(Arvore);
  Arvore construirArvore(int, Arvore, Arvore);
  //o primeiro parâmetro é um elemento do conjunto, os dois últimos são
  //ponteiros para as subárvores esquerda e direita.
```

2) Produzir um programa de teste para o tipo Arvore do exercício anterior de modo que seja construída a árvore apresentada na figura 1.

3) Implementar os algoritmos de varredura para substituição do corpo em mostrarArvore:
   a) Red(a) – executa a varredura preordem (raiz-subárvore esquerda-subárvore direita)
   b) edR(a) – executa a varredura posordem (subárvore esquerda-subárvore direita-raiz)

4) Implementar o algoritmo de varredura por nível para mostrarArvore, seguindo o esboço do algoritmo dado a seguir. Para testar a implementação é necessário utilizar o tipo Fila implementado em FilaPointer.h. O algoritmo é denominado bfs(a), iniciais de breadth first search. O parâmetro a é a referência para o nó raiz da árvore.

5) Escrever uma versão iterativa do algoritmo de caminhamento eRd. Para isso, use uma pilha auxiliar (implementada em PilhaPointer.h).

AJUDAS

EXERCICIO 1 - implementações

```
Arvore criarArvoreVazia( ){
   Arvore ap;
   ap = NULL;
   return ap;
   }

bool verificarArvoreVazia(Arvore ap){
   bool ok;
   if (ap == NULL) ok = TRUE; else ok = FALSE;
   return ok;
   }

void mostrarArvore(Arvore ap){
   if (ap != NULL){
        mostrarArvore(ap->esq);
        mostrarRaiz(ap)
        mostrarArvore(ap->dir);
        }
   }

Arvore construirArvore(int item, Arvore e, Arvore d){
     No *novo, *raiz;
     novo = (struct No*)malloc(sizeof(struct No));
     novo->elemento = item;
     novo->esq = e;
     novo->dir = d;
     raiz = novo;
     return raiz;
}

void mostrarRaiz(Arvore ap){
     printf(" %d  \n",ap->elemento);
}
```

EXERCICIO 4 - algoritmo

Esboço do algoritmo bfs(a):

```
se (a ≠ terra)
   então  p ← a;  colocar p na fila f
          repita
             p ← acessar fila f;  mostrarElemento(p); retirar da fila f;
             e ← raiz da subárvore esquerda de p;
             se (e ≠ terra)  então  colocar e na fila;
             d ← raiz da subárvore direita de p;
             se (d ≠ terra)  então  colocar d na fila f;
          até que (fila f seja vazia)
```

EXERCICIO 5 - algoritmo

Esboço do algoritmo eRd(a)

```
se (a ≠ terra)
    então  p ← a;   fim ← false;
           repita
              enquanto (p ≠ terra) faça
                    coloca p na pilha
                    avança p para a esquerda
              se (pilha ≠ ∅)
                 então  p ⇐ acessarTopo da pilha
                        mostrar No p
                        pop pilha
                        avança p para a direita
                 senão  fim ← true
           até que (fim = true)
```

implementações - Pilha e Fila

```
/* Pilha de pointer */
#define Max 20
typedef struct {
      void * apont;
      int item;
}Casa;
typedef struct {
     int topo;
     Casa tabela[Max];
} Pilha;

//protótipos
void criarPilhaVazia(Pilha *);
void *acessarTopo(Pilha *);
int verificarPilhaVazia(Pilha *);
int verificarPilhaCheia(Pilha *);
void pushPilha(Pilha *, void *);
void popPilha(Pilha *);

// implementações
void criarPilhaVazia(Pilha *ap){
    ap->topo = 0;
}
void *acessarTopo(Pilha *ap){
   int k;
   void *t;
   k = ap->topo - 1;
   t = ap->tabela[k].apont;
   return t;
}
```

```
int verificarPilhaVazia(Pilha *ap){
    int vazia;
    if (ap->topo == 0) vazia = 1; else vazia = 0;
    return vazia;
}
int verificarPilhaCheia(Pilha *ap){
    int cheia;
    if (ap->topo == Max) cheia = 1; else cheia = 0;
    return cheia;
}
void pushPilha(Pilha *a, void *novo){
    int k;
    k = a->topo;
    a->tabela[k].item = 0;
    a->tabela[k].apont = novo;
    a->topo = k + 1;
}
void popPilha(Pilha *a){
    int k;
    k = a->topo;
    a->topo = k-1;
}
```

```
/* fila de pointer */
typedef struct Celula{
      void * item;                /* item é um pointer */
      struct Celula *next;       /* o campo next guarda o endereço do elemento seguinte da fila */
} Celula;
typedef struct{
      Celula *inicio;     /* inicio tem o endereço do primeiro da fila */
      Celula *fim;        /* fim guarda o endereço do ultimo da fila */
} Fila;

// prototipos das funções
void criarFilaVazia(Fila *);         /* o construtor cria a fila vazia */
void *acessarFila(Fila *);           /* devolve o primeiro da fila */
int verificarFilaVazia(Fila *);      /* devolve 1 se a fila estiver vazia */
void pushFila(Fila *, void *);       /* coloca o valor dado no fim da fila */
void popFila(Fila *);                /* retira o primeiro da fila */

// implementações
void criarFilaVazia(Fila *fi){
    fi->inicio = NULL;
    fi->fim = NULL;
}
 void *acessarFila(Fila *fi){
   Celula *aux;
   void * valor;
   aux = fi->inicio;
   if (aux == NULL) valor = NULL;
   else valor = aux->item;
   return valor;
}
int verificarFilaVazia(Fila *fi){
    int vazia;
    Celula *aux;
    aux = fi->inicio;
    if (aux == NULL) vazia = 1;
    else vazia = 0;
    return vazia;
}
void pushFila(Fila *fi, void *ap){
    Celula *p;
    Celula *aux;
    p = (Celula *)malloc(sizeof(Celula));
    p->item = ap; p->next = NULL;
    if (fi->inicio == NULL) fi->inicio = p;
    else {aux = fi->fim;  aux->next = p;}
    fi->fim = p;
}
void popFila(Fila *fi){
    Celula *seg;
    Celula *pri;
    pri = fi->inicio;
    if (fi->inicio != NULL){
      if (fi->inicio == fi->fim) {fi->inicio = NULL; fi->fim = NULL;}
      else {seg = pri->next; fi->inicio = seg;}
    }
}
```