DevTitans

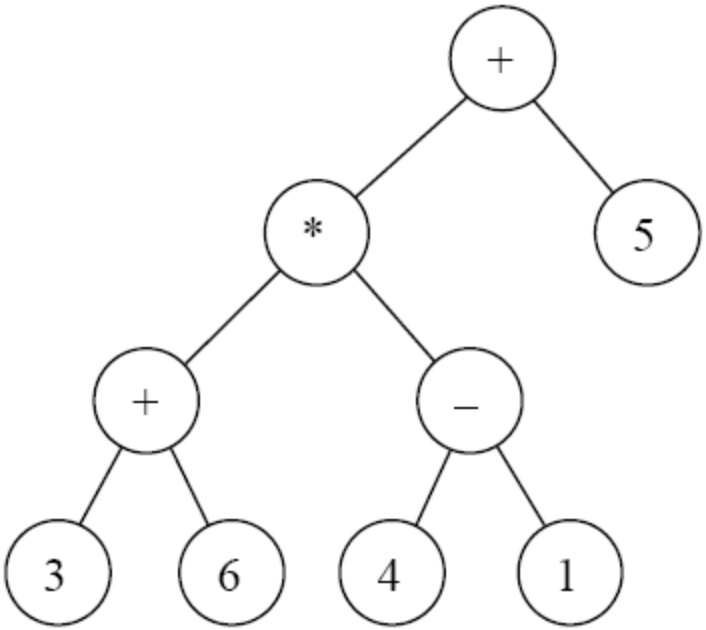
Árvores

Árvores Binárias

Árvore Binária

- Um exemplo de utilização de árvores binárias está na avaliação de expressões.
- Como trabalhamos com operadores que esperam um ou dois operandos, os nós da árvore para representar uma expressão têm no máximo dois filhos.
- Nessa árvore, os nós folhas representam operandos e os nós internos operadores.

Árvores Binárias

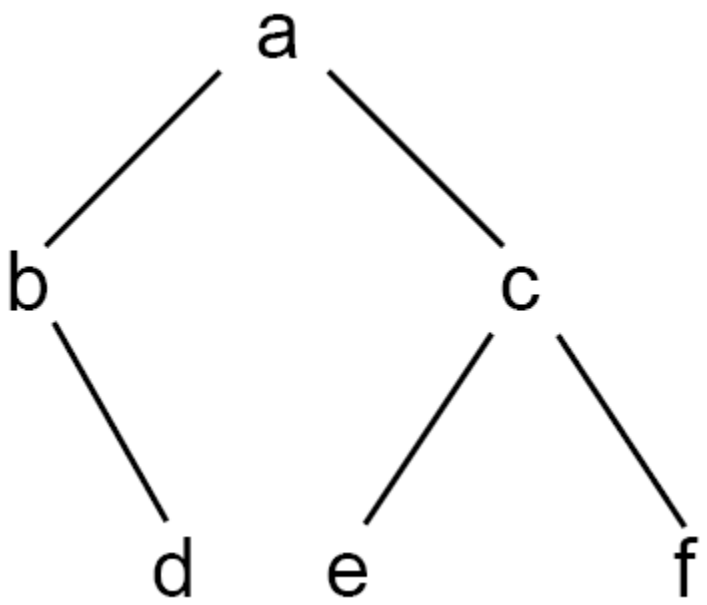


Esta árvore representa a expressão (3+6)*(4-1)+5

Estrutura de uma AB

- Numa árvore binária, cada nó tem zero, um ou dois filhos.
- De maneira recursiva, podemos definir uma árvore binária como sendo:
  - uma árvore vazia; ou
  - um nó raiz tendo duas sub-árvores, identificadas como a sub-árvore da direita (*sad*) e a sub-árvore da esquerda (*sae*).

Estrutura de uma AB



Raiz
Nós Internos
Folhas

Representação em C

```
struct arv {
   char info;
   struct arv* esq;
   struct arv* dir;
};

typedef struct arv Arv;
```

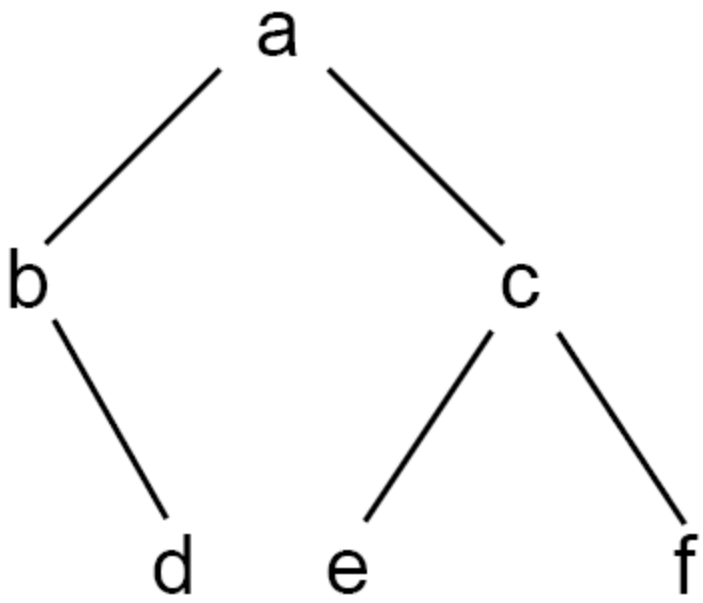
Da mesma forma que uma lista encadeada é representada por um ponteiro para o primeiro nó, a estrutura da árvore como um todo é representada por um ponteiro para o nó raiz.

Criando Árvores

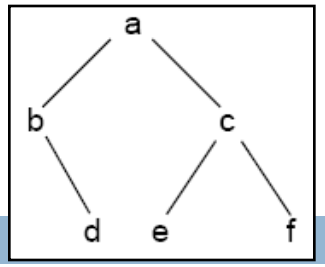
```
Arv* inicializa(void)
{
  return NULL;
}

Arv* cria(char c, Arv* sae, Arv* sad)
{
  Arv* p=(Arv*)malloc(sizeof(Arv));
  p->info = c;
  p->esq = sae;
  p->dir = sad;
  return p;
}
```

Exemplo

- Exemplo: Usando as operações `inicializa` e `cria`, crie uma estrutura que represente a seguinte árvore.

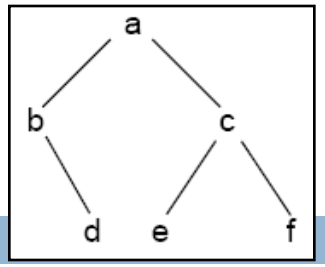


Exemplo




```
/* sub-árvore com 'd' */
Arv* a1= cria('d',inicializa(),inicializa());

/* sub-árvore com 'b' */
Arv* a2= cria('b',inicializa(),a1);

/* sub-árvore com 'e' */
Arv* a3= cria('e',inicializa(),inicializa());

/* sub-árvore com 'f' */
Arv* a4= cria('f',inicializa(),inicializa());

/* sub-árvore com 'c' */
Arv* a5= cria('c',a3,a4);

/* árvore com raiz 'a'*/
Arv* a = cria('a',a2,a5 );
```

Exemplo



A árvore poderia ser criada recursivamente com uma única atribuição, seguindo a sua estrutura

```
Arv* a = cria('a',
              cria('b',
                  inicializa(),
                  cria('d', inicializa(), inicializa())
                  ),
              cria('c',
                   cria('e', inicializa(), inicializa()),
                   cria('f', inicializa(), inicializa())
                   )
              );
```

Exibindo Conteúdo da Árvore

```
void imprime (Arv* a)
{
  if (!vazia(a)){
    printf("%c ", a->info); /* mostra raiz */
    imprime(a->esq); /* mostra sae */
    imprime(a->dir); /* mostra sad */
  }
}
```

Como é chamada essa forma de exibição?

E para exibir na forma in-fixada? E na pós-fixada?

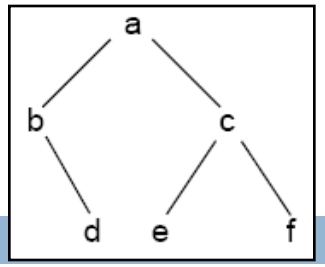
Liberando Memória

```
Arv* libera (Arv* a){
  if (!vazia(a)){
    libera(a->esq); /* libera sae */
    libera(a->dir); /* libera sad */
    free(a); /* libera raiz */
  }
  return NULL;
}
```

Enxerto e Poda

Vale a pena notar que a definição de árvore, por ser recursiva, não faz distinção entre árvores e sub-árvores. Assim, `cria` pode ser usada para acrescentar ("enxertar") uma sub-árvore em um ramo de uma árvore, e `libera` pode ser usada para remover ("podar") uma sub-árvore qualquer de uma árvore dada.

Enxerto e Poda



Considerando a criação da árvore feita anteriormente, podemos acrescentar alguns nós, com:

```
a->esq->esq = cria('x',
  cria('y',inicializa(),inicializa()),
  cria('z',inicializa(),inicializa())
);
```

E podemos liberar alguns outros, com:

```
a->dir->esq = libera(a->dir->esq);
```

Buscando um Elemento

Essa função tem como retorno um valor booleano (um ou zero) indicando a ocorrência ou não do caractere na árvore.

```
int busca (Arv* a, char c){
  if (vazia(a))
    return 0;
  else
    return a->info==c ||
      busca(a->esq,c) ||
      busca(a->dir,c);
}
```

Buscando um Elemento

Podemos dizer que a expressão:

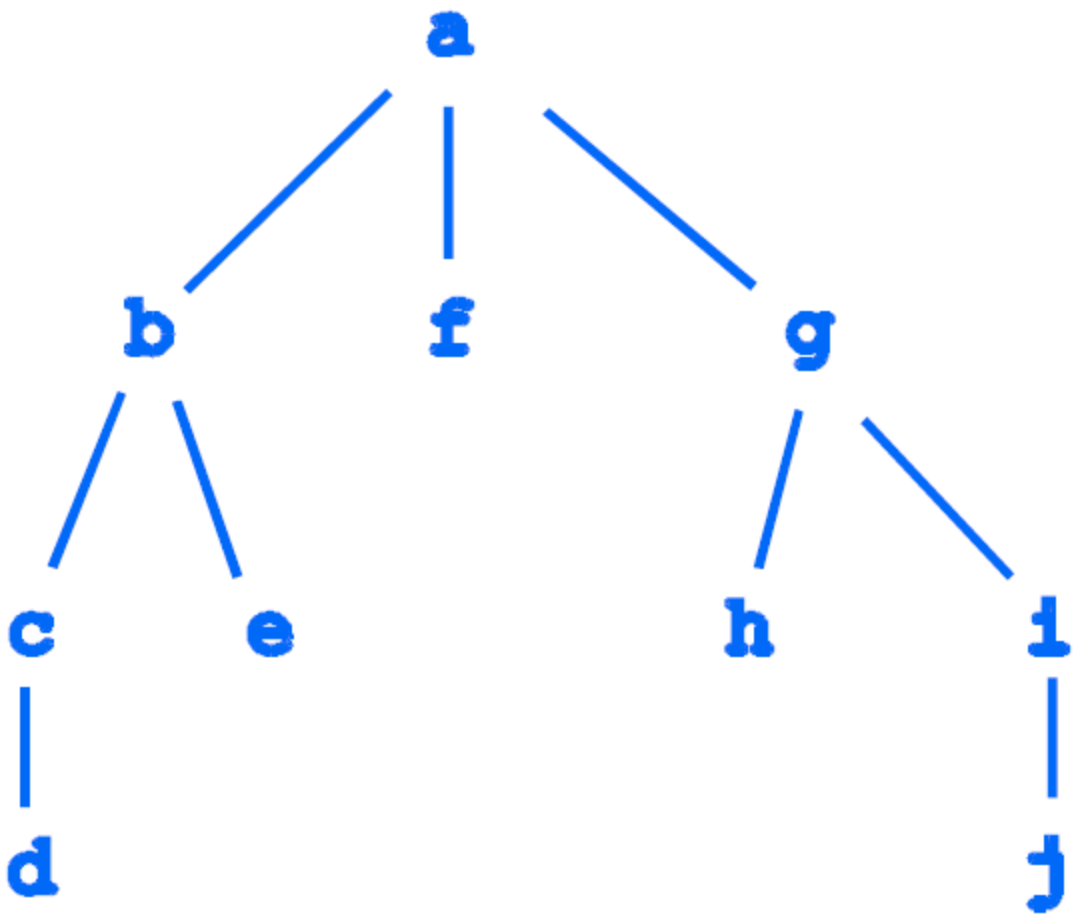
```
return c==a->info || busca(a->esq,c) || busca(a->dir,c);
```

é equivalente a:

```
if (c==a->info)
  return 1;
else if (busca(a->esq,c))
  return 1;
else
  return busca(a->dir,c);
```

Árvores Genéricas

Exemplo

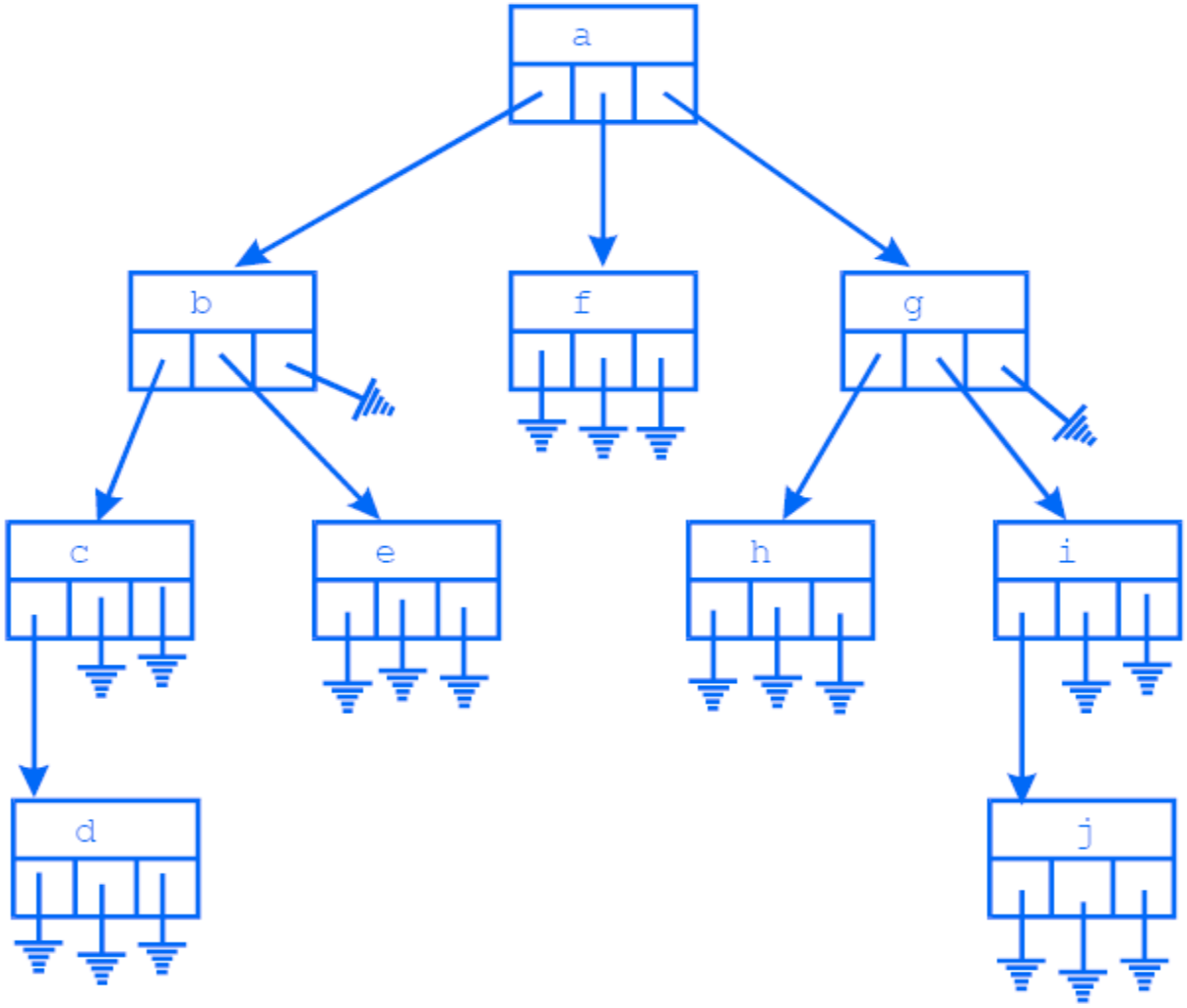


Representação em C

Prevendo um número máximo de filhos igual a 3,
e considerando a implementação de árvores para armazenar valores de caracteres simples, a declaração do tipo que representa o nó da árvore poderia ser:

```
struct arv3 {
  char val;
  struct arv3 *f1, *f2, *f3;
};
```

Mesmo Exemplo



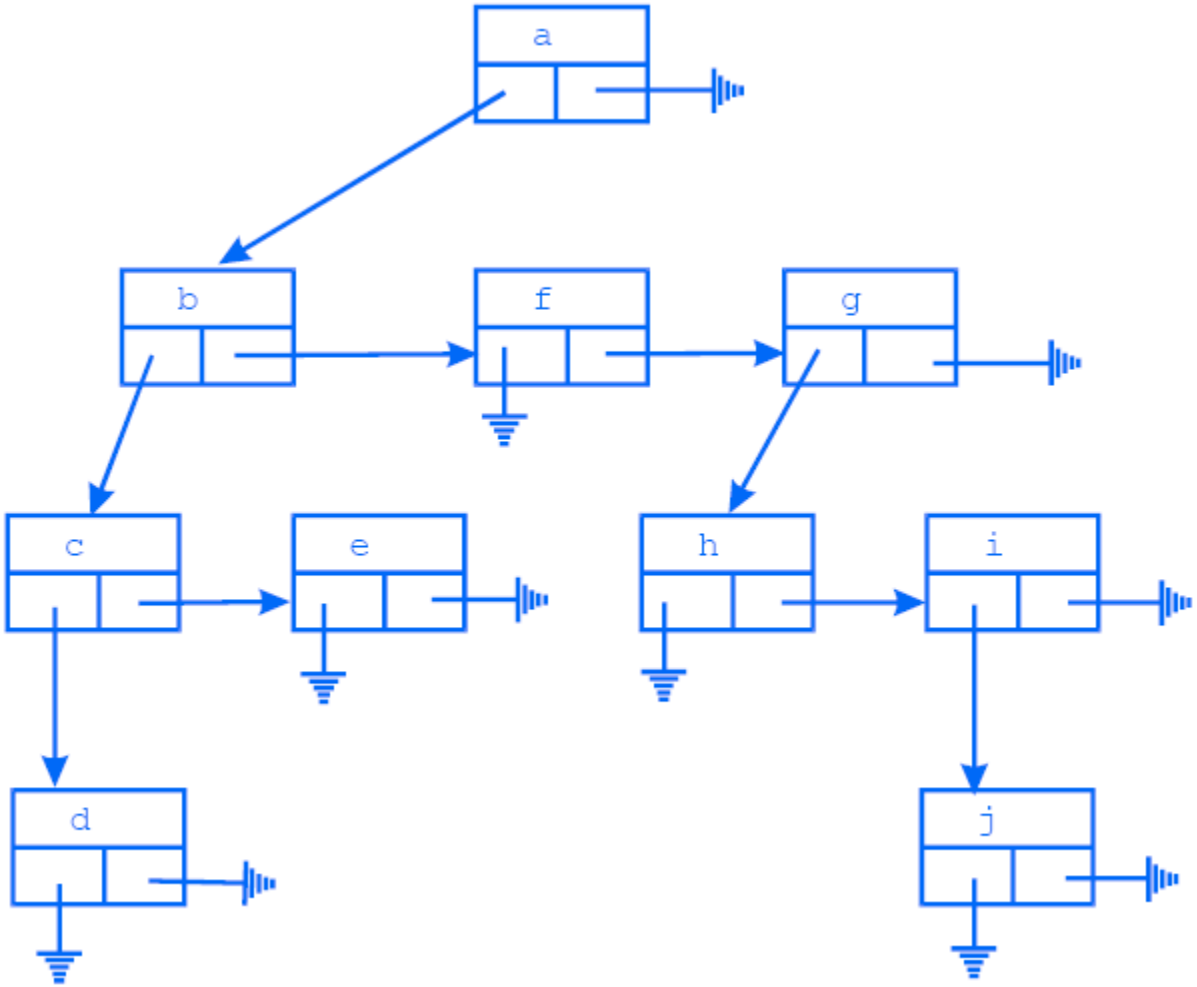
Solução

Uma solução que leva a um aproveitamento melhor do espaço utiliza uma "lista de filhos": um nó aponta apenas para seu primeiro (`prim`) filho, e cada um de seus filhos, exceto o último, aponta para o próximo (`prox`) irmão.

A declaração de um nó pode ser:

```
struct arvgen {
  char info;
  struct arvgen *primFilho;
  struct arvgen *proxIrmao;
};
typedef struct arvgen ArvGen;
```

Exemplo da Solução



Exemplo da Solução

Uma das vantagens dessa representação é que podemos percorrer os filhos de um nó de **forma sistemática**, de maneira análoga ao que fizemos para percorrer os nós de uma lista simples

Com o uso dessa representação, a generalização da árvore é apenas conceitual, pois, concretamente, a árvore foi transformada em uma **árvore binária**, com filhos esquerdos apontados por `primFilho` e direitos apontados por `proxIrmao`

Criação de uma ÁrvoreGen

```
ArvGen* cria (char c)
{
  ArvGen *a =(ArvGen *)malloc(sizeof(ArvGen));
  a->info = c;
  a->primFilho = NULL;
  a->proxIrmao = NULL;
  return a;
}
```

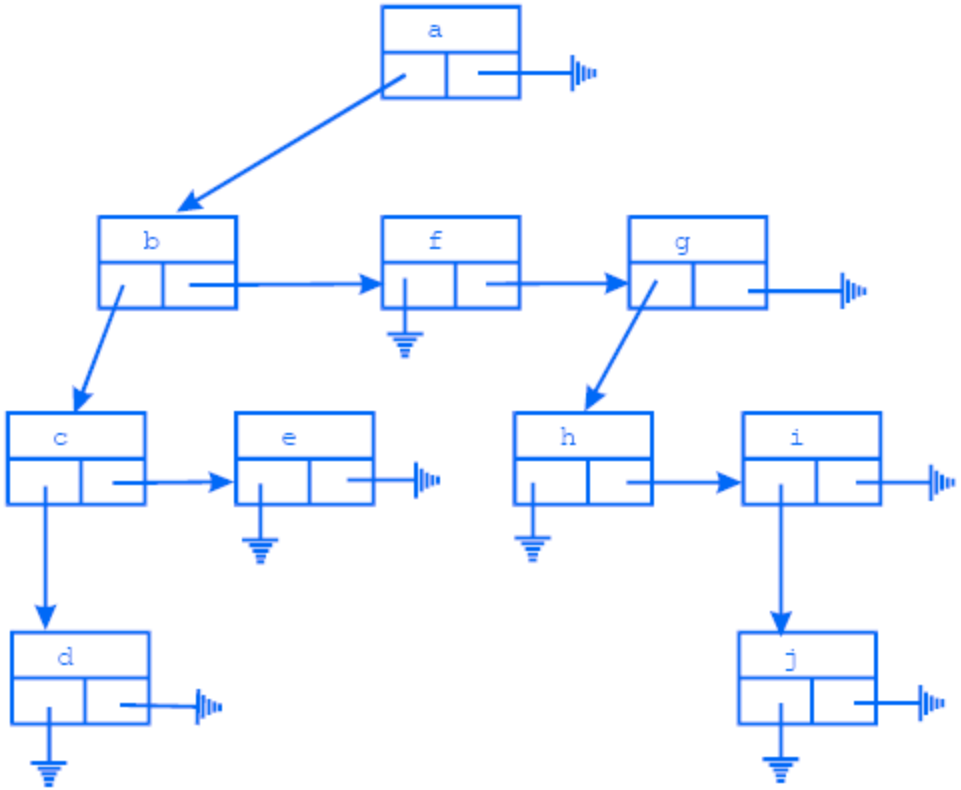
Inserção

Optamos por inserir sempre no início da lista

```
void insere (ArvGen* a, ArvGen* f)
{
  f->proxIrmao = a->primFilho;
  a->primFilho = f;
}
```

Exemplo Criação ÁrvoreGen

Com as funções `cria` e `insere` como construir a árvore abaixo?



Exemplo Criação ÁrvoreGen

```
/* cria nós como folhas */
ArvGen* a = cria('a');
ArvGen* b = cria('b');
ArvGen* c = cria('c');
ArvGen* d = cria('d');
ArvGen* e = cria('e');
ArvGen* f = cria('f');
ArvGen* g = cria('g');
ArvGen* h = cria('h');
ArvGen* i = cria('i');
ArvGen* j = cria('j');
...
```

```
...
/* monta a hierarquia */
insere(c,d);
insere(b,e);
insere(b,c);
insere(i,j);
insere(g,i);
insere(g,h);
insere(a,g);
insere(a,f);
insere(a,b);
```

Impressão (pré-ordem)

```
void imprime (ArvGen* a)
{
  ArvGen* p;
  printf("%c\n",a->info);
  for (p=a->primFilho; p!=NULL; p=p->proxIrmao)
    imprime(p);
}
```

Busca de Elemento

```
int busca (ArvGen* a, char c)
{
  ArvGen* p;
  if (a->info==c)
    return 1;
  else {
    for (p=a->primFilho; p!=NULL; p=p->proxIrmao)
    {
      if (busca(p,c))
        return 1;
    }
  }
  return 0;
}
```

Liberação de Memória

O único cuidado que precisamos tomar na programação dessa função é a de liberar as subárvores antes de liberar o espaço associado a um nó (isto é, usar pós-ordem).

```
void libera (ArvGen* a) {
  ArvGen* p = a->primFilho;
  while (p!=NULL) {
    ArvGen* t = p->proxIrmao;
    libera(p);
    p = t;
  }
  free(a);
}
```