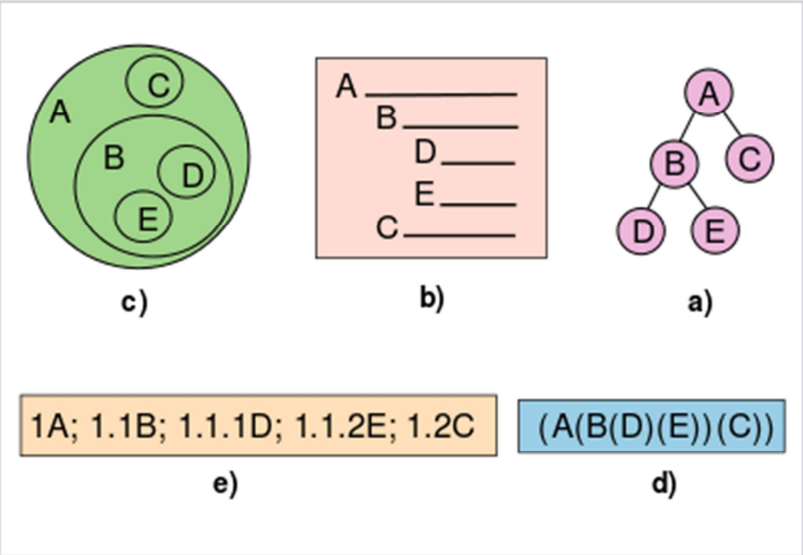
1. ÁRVORES - CONCEITOS BÁSICOS

São estruturas de dados não lineares que seguem o conceito de hierarquia que herdam as características das topologias em árvore, e.g., estruturas de pastas de um sistema operacional, banco de dados, interfaces gráficas, sites de internet, representação de expressão aritmética, hierarquia universitária, etc., são exemplos de árvore. Uma árvore é formada por um conjunto de elementos que armazenam informações chamadas de nodos.

Formas de representação de uma árvore: hierárquica, diagrama de inclusão, diagrama de barras, numeração por níveis, por aninhamento ou parênteses.



Diferentes representações de uma árvore: a) hierárquica, b) diagrama de barras, c) diagrama de inclusão, d) aninhamento e e) numeração por níveis

1.1. **Buscando Elementos**

Imagine que queremos buscar um elemento (15) no array abaixo. Como fazer ?

2	8	12	15	20	23	30

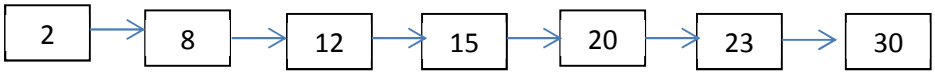
E o elemento 17? Será que não poderíamos aproveitar o fato de que o arranjo está ordenado?

Buscando o elemento 17?

Mas, se o número buscado for maior que todos (e.g. 35), neste caso, correremos o arranjo todo. Teria como melhorar?

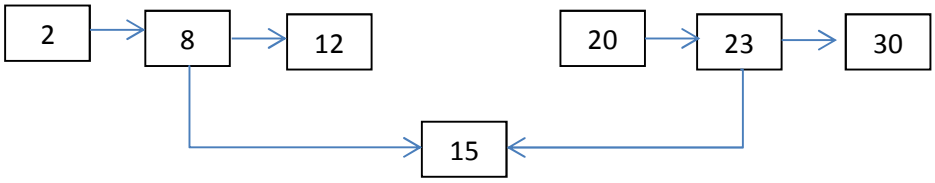
1.2. **Busca Binária**... É mais eficiente, mas depende de arranjos estáticos.

E se tivéssemos uma lista ligada?

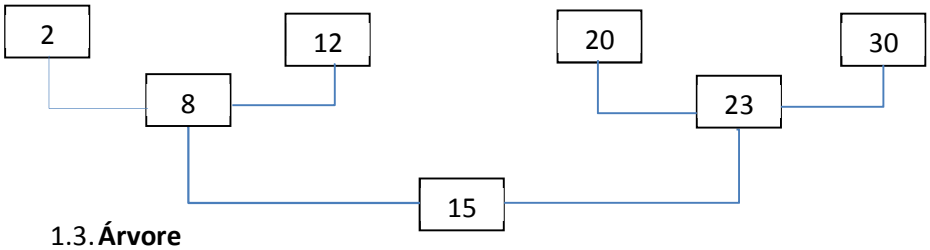


Teríamos um problema, pois não saberíamos o elemento do meio. Teríamos que fazer um arranjo de ponteiros para esta lista. Será que não poderíamos ter uma estrutura dinâmica que nos ajude nesta tarefa?

Se aplicássemos uma busca binária na lista ligada?

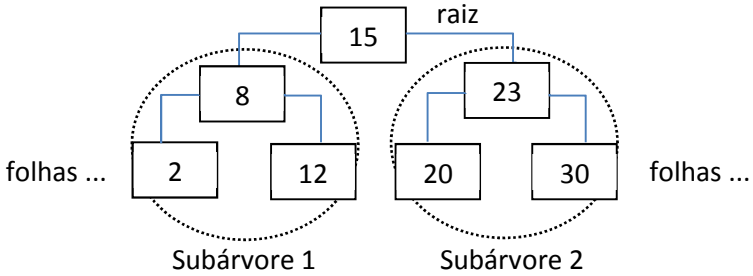


Eis nossa estrutura. Que nome, daremos a ela? Uma árvore.



1.3. **Árvore**

Em computação costumamos representar a árvore de forma invertida.



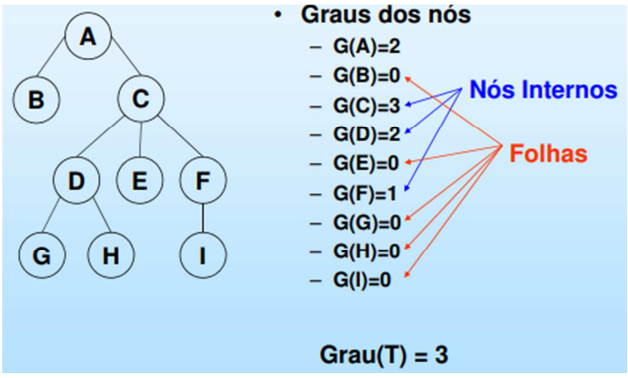
1.3.1. **Subárvores (Ramos)**

Um conjunto de nós consistindo de um nó chamado raiz, abaixo do qual estão as subárvores que compõem essa árvore.

O número de subárvores de cada nó é chamado de grau desse nó. No exemplo ao lado, todo nó tem grau 02, exceto os de base, que tem grau 0.

1.3.2. **Grau; Internos/Externos**

Nós de grau ZERO são chamados de nós externos ou folhas. Os demais são chamados de nós internos.

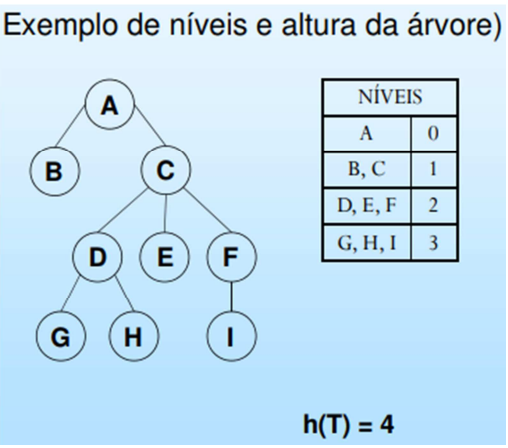


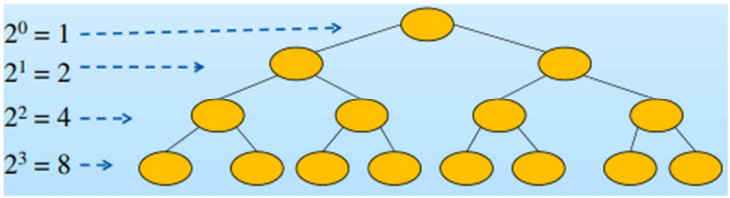
1.3.3. **Descendentes**

- Nós, abaixo de um determinado nó, são seus descendentes.
- Descendentes de 8: 2 e 12
- Descendentes de 15: todos os nós.
- Folha não tem descendente.

1.3.4. **Nível**

O nível do nó raiz é ZERO. A cada grupo de subárvores, nova geração, teremos novos níveis. Quantidade de nós da árvore será $2^n - 1$.





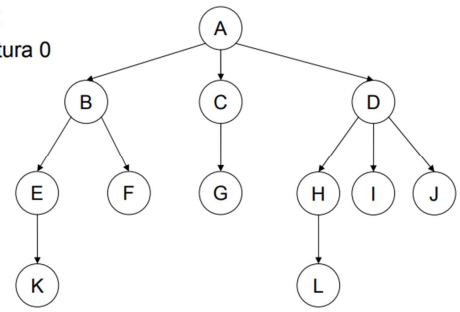
Em uma árvore binária cheia o número de nós do nível i é igual a 2^i. Consequentemente, em qualquer árvore binária existe no máximo 2^i nós no nível i.

1.3.5. **Altura**

- A **altura de uma árvore** é a altura da raiz.
- A **altura de um nó** será à distância desse nó, do maior caminho, até uma folha.
- Vale notar que a árvore é percorrida sempre da raiz às folhas.
- A altura (h) de um nó raiz é o comprimento do caminho mais longo entre ele e uma folha.
- Nem sempre a árvore estará balanceada. Ainda assim, as definições de altura, nível etc valem.
- A altura de uma árvore de uma raiz, da mesma forma, que o endereço de uma árvore na memória será o endereço de seu nó raiz.

No exemplo, os nós:

- K, F, G, L, I, J têm altura 0
- E, C e H têm altura 1
- B e D têm altura 2
- A tem altura 3



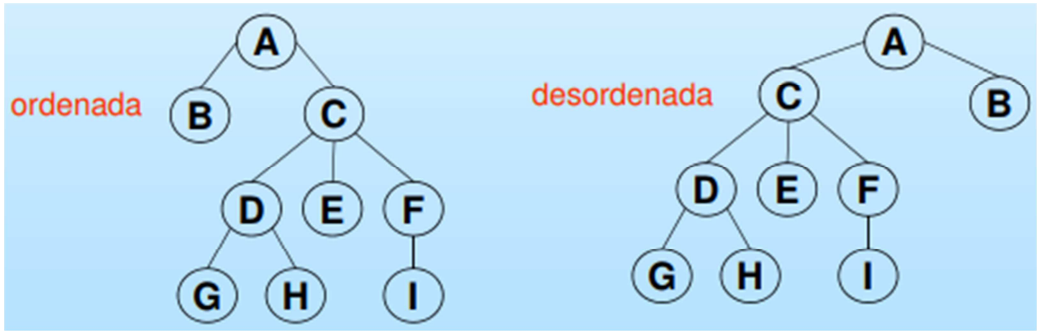
1.3.6. **Profundidade**

A profundidade da raiz é ZERO. E a profundidade de um nó é a distância percorrida da raiz a esse nó.

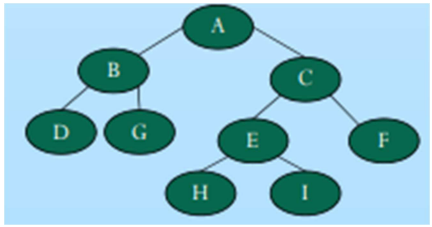
- Profundidade de 15 (raiz): ZERO;
- Profundidade de 8: 1;
- Profundidade de 12: 2.

1.3.7. Definições

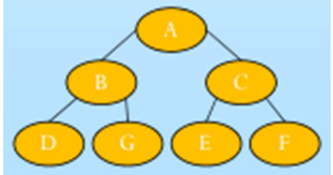
- ✓ Raiz => Não possui ancestral, só pode ter filhos.
- ✓ Folha => Não tem filhos (ou melhor, seus filhos são estruturas vazias)
- ✓ Nó ou vértice => Elemento que contém a informação
- ✓ Arco ou aresta => Liga dois nós
- ✓ Pai => Nó superior de uma aresta
- ✓ Raiz => Nó topo – não possui pai
- ✓ Folhas => Nós das extremidades inferiores – não tem nós filhos
- ✓ Grau => Representa o número de subárvores de um nó.
- ✓ Grau de uma árvore (aridade) : É definido como sendo igual ao máximo dos graus de todos os seus nós.
- ✓ Nível de um nó => É a distância da raiz da árvore. A raiz tem nível ZERO.
- ✓ **Altura (ou profundidade) de uma árvore** => É o nível máximo entre todos os nós da árvore ou, equivalente é a altura da raiz.
- ✓ **Altura de um nó** => É o número de arestas no maior caminho desde o nó até um dos seus descendentes. Portanto as folhas tem altura ZERO.
- ✓ A árvore vazia é uma árvore de altura -1, por definição.
- ✓ Uma árvore com um único nó tem altura 1 (o nó é raiz e folha ao mesmo tempo).
- ✓ Toda árvore com n > 1 nós possui no mínimo 1 e no máximo n-1 folhas.
- ✓ Árvore estritamente binária, cada nó possui 0 ou 2 filhos.
- ✓ Árvore binária completa. Se v é um nó tal que alguma subárvore de v é vazia, então v se localiza ou no último (maior) ou no penúltimo nível da árvore.
- ✓ Árvore binária cheia. Se v é um nó tal que alguma subárvore de v é vazia, então v se localiza no último (maior) nível da árvore, v é um nó folha.



Árvore Completa



Árvore Cheia



2. ÁRVORES BINÁRIAS

Uma árvore binária é uma árvore em que, abaixo de cada nó, existem no máximo duas subárvores. Trata-se de uma árvore binária de pesquisa, árvore binária de busca, árvore de busca binária.

Uma árvore binária de pesquisa – ABP é uma árvore binária em que, a cada nó, todos os registros com chaves menores que a deste nó estão na subárvore da esquerda, enquanto os registros com chaves maiores estão na subárvore da direita.

Como representamos computacionalmente uma árvore binária? Unindo nós. E como representamos os nós? Com 02 ponteiros: um ponteiro para a subárvore da esquerda e um ponteiro para a subárvore da direita. Além de um campo para a chave e dados.

chave	
null	null

Para nossa ABP, utilizaremos as seguintes funções:

- Inicializar Árvore;
- Inserir Elemento
- Buscar Elemento
- Contar número de Elementos
- Imprimir Elementos
- Remover Elementos

2.1. **Definindo a estrutura de dados não linear – árvore binária em linguagem C**

```
#include <stdio.h>
#include <stdlib.h>
#include (malloc.h>
#define false 0
#define true 1
typedef int bool;
typedef int PRIMARY_KEY;
typedef struct pontNo {
        PRIMARY_KEY chave;
        struct pontNo *sae;
        struct pontNo *sad;
} NO;
typedef NO* arvore;
arvore Inicializar();
arvore adicionarElemento(arvore raiz, arvore no);
arvore criarNovoNo(PRIMARY_KEY ch);
int contarNo(arvore raiz);
void exibirArvore(arvore raiz);
arvore removerNo(arvore raiz, PRIMARY_KEY ch);
```

```
Int main(void){

  PRIMARY_KEY ch;

  // Inicializar Raiz
   arvore r = Inicializar();


// Criar Chave Novo Nó [Elemento]
//printf("Digite o elemento:");  scanf("%d", &ch);

arvore no = criarNovoNo(23); // exemplo 23, 12, 15, 25, 8


// Inserir Chave - Elemento(nó) na Raiz
r = adicionarElemento(r, no);


// Contar os Nós da Raiz
Printf("Total de nós: %d", contarNo(r));


// Exibir os Nós da Raiz
exibirArvore (r);


// Remover  Chave,  Nó da Raiz
removerNo (r, 15);

}
```

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

2.2. Inicializar Árvore

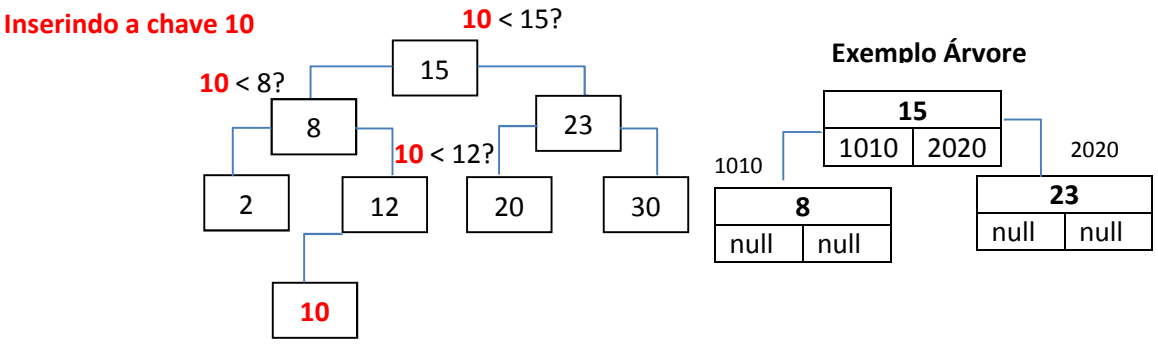
Para representarmos nossa árvore, precisaremos tão somente do endereço do nó raiz. E para inicializarmos a árvore basta tonarmos esse endereço NULL.

```
arvore Inicializar() {
  return NULL; }
```

Nota: Se a inserção fosse: 25, 23, 15, 12 e 8 (forma um galho, parece uma lista ligada .) A ordem define a forma da árvore.

2.3. Inserir Chave (Elemento)

Nesta árvore não será permitido a duplicação de chaves. Chaves menores à de um determinado nó ficam na subárvore da esquerda deste. E chaves maiores de um determinado nó ficam na subárvore da direita deste.




```
arvore criarNovoNo(PRIMARY_KEY ch){
        arvore* novo = (arvore*) malloc sizeof(arvore);
        novo->chave=ch;
        novo->sae=NULL;
        novo->sad= NULL;
        return arvore; }
```

- Algoritmo de Inserção

```
Se ( a raiz for null ) então
   Inserimos na raiz
Senão  Se ( a chave do elemento a ser inserido for menor que a da raiz) então
          Insere elemento na subárvore da esquerda
       Senão
         Insere elemento na subárvore da direita
       Fim Se
Fim Se
```

```
arvore adicionarElemento(arvore raiz, arvore no){
        if ( arvore == NULL ) return(no);
        if ( no->chave < raiz->chave )
                raiz->sae = adiciona(raiz->sae, no);
        else    raiz->sad = adiciona(raiz->sad, no);
        return(raiz); }
```

2.4. **Buscar de Elemento**

```
arvore buscarArvore(PRIMARY_KEY ch, arvore raiz){
        if ( arvore == NULL ) return NULL;
        if ( raiz->chave == ch ) return (raiz);
        if ( raiz->chave > ch ) return (buscaArvore(ch, raiz->sae);
        else return (buscaArvore(ch, raiz->sad);
}
```

2.4.1. **Algoritmos de Busca em árvores**

Uma das operações importantes consiste em percorrer cada elemento da árvore uma única vez, percurso, também chamado de travessia da árvore, em profundidade em árvores binárias, pode ser feita:

- **Pré-Ordem ou Pré-Fixado (VLR) raiz->sae->sad**
  Os filhos de um nó são processados após o nó

- **Em-Ordem (Central) ou in-Ordem (LVR) sae->raiz->sad**
  Em que se processa o filho à esquerda, o nó, e finalmente o filho à direita.

- **Pós-Ordem ou Pós_Fixado (LRV) sae->sad->raiz**
  os filhos são processados antes do nó

Outra operação utilizada nas árvores de pesquisa é a travessia da raiz até uma das folhas. Essa operação tem um custo computacional proporcional ao número de níveis da árvore. O pior caso é a travessia de todos os elementos até a folha de nível mais baixo. Árvores balanceadas presentam o melhor pior caso possível, para certo número de nós. O pior caso apresenta-se na árvore degenerada em que cada nó possui exatamente um filho, e a árvore tem o mesmo número de níveis que de nós, assemelhando-se a uma lista ligada.

2.5. **Contar Nós da Raiz**

- Algoritmo de Contagem

Se não houver árvore raiz, n = 0.
Se existir árvore, conte a subárvore à esquerda some a raiz.
Some a raiz
Some a subárvore à direita

```
int contarNo (arvore raiz){
        if ( raiz == NULL ) return 0;

        return ( contagemNo(raiz->sae)
                + 1
                + contagemNo(raiz->sad) );
}
```

2.6. Exibir Elementos da Árvore

Para imprimir a árvore, iremos adotar a ordem: **raiz – subárvore esquerda – subárvore direita**.

```
void exibirArvore (arvore raiz){
        if ( raiz != NULL ) {
                printf("%i", raiz->chave);

                printf("(");

                exibirArvore(raiz->sae);
                exibirArvore(raiz->sad);

                printf(")");
}
```

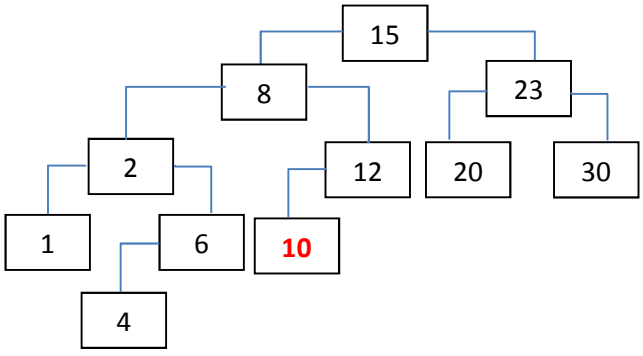
2.7. Remover Elementos da Árvore - ABP

Método para remover um elemento da árvore binária – ABP. Existem outras formas de implementar, entretanto apresentaremos uma maneira de como fazer.

Problemas na remoção de um nó: Temos que lidar com as subárvores desse nó, portanto, a árvore resultante deve continuar sendo de busca. Nós da subárvore da esquerda tem chave menor a do nó raiz e os nós da subárvore da direita tem chave maoir do que a do nó raiz.

Como fazer? Se o nó a ser retirado possui no máximo um descente, substitua-o por este. Se o nó a ser retirado for um nó folha, elimine o nó da subárvore. Agora, se o nó possuir 02 dependentes, substituímos o nó a ser retirado pelo nó mais à direita da subárvore da esquerda ou substituímos o nó a ser retirado pelo nó mais à esquerda da subárvore da direita.

Exemplo: Para removermos o elemento 15?



Ou substituímos pelo elemento 12 e o elemento 10 passa a ser filho de 8 ou substituímos por 20.

Para remover, precisamos então saber:

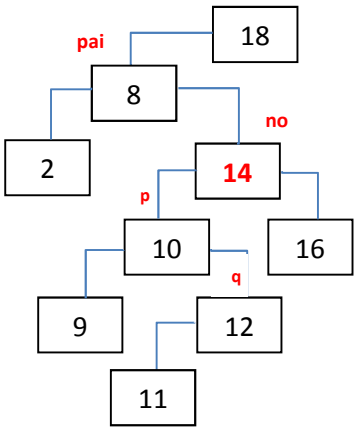
- O nó a ser removido;

- O pai do nó
- O nó substituto
- O pai do nó substituto

Utilizando um método auxiliar:
/* Busca binária não recursiva. Devolve o ponteiro do nó buscado. Abastece pai com ponteiro para o nó pai deste. */

```
arvore buscaNO (arvore raiz, PRIMARY_KEY ch, arvore *pai){
        arvore atual = raiz;
        *pai = NULL;

        while (atual) {
           if ( atual->chave == ch ) return(atual);
           *pai = atual;
           If ( ch < atual->chave) atual = atual->sae;
          else atual = atual->sad;
        }
        return (NULL);
}

arvore removerNo (arvore raiz, PRIMARY_KEY ch) {
        arvore pai, no, p, q;
        no = buscaNO(raiz, ch, &pai);
        if (no == NULL) return(raiz);

        if (!no->sae || !no->dir) {
             if (!no->sae) q = no->sad;
             else q = no->sae;
        }
        else {    p = no;
                  q = no->sae;
                  while ( q->sad ) {
                      p = q;
                      q = q->sad;
                  }
                  If ( p != no ) {
                     p->sad  = q->sae;
                     q->sae = no->sae;
                  }
                  q->sad = no->sad;
        }
        if (!pai) {
           free(no);
           return(q);
        }
        if (ch < pai->chave) pai->sae = q;
        else pai->sad = q;

        free(no);
        return(raiz);
}
```

Exemplo: Exclusão do elemento 14.



Então, tratamos o caso do nó removido ter no máximo um filho, ou ter dois filhos, do nó removido ser a raiz ou não ser a raiz. Além de tratarmos do caso do pai do substituto ser ou não o nó removido.

Importante lembrar que... A ordem de inserção determina a forma da árvore, ou seja, a forma de balanceamento da árvore. E isso determina o quão eficiente serão as buscas na árvore.

Podemos ter então a eficiência de uma busca binária, caso a árvore esteja balanceada ou aproximadamente balanceada, com a vantagem de ser uma estrutura dinâmica. Caso contrário, voltamos à busca sequencial, como em uma lista encadeada, árvore degenerada, só que usando mais memória, pelo ponteiro extra na estrutura de dados.

A boa notícia é que se os elementos que compõe a árvore forem obtidos aleatoriamente, espera-se um desempenho apenas de 39% pior do que a árvore completamente balanceada. Ou seja, a árvore em que as folhas aparecem no mesmo nível ou, no máximo, em dois níveis adjacentes.

Portanto, se seus dados vierem em ordem crescente ou decrescente a estrutura de dados árvore será a mais indicada.

Em suma:

- Grau = número de subárvores (quantidade de nós)
- Descendentes = nós que estão abaixo de um nó.
- h (altura) = Comprimento do caminho mais longo entre o nó e um nó folha.
- A altura de uma árvore é a altura de uma raiz. O endereço de uma árvore é o endereço de uma raiz.
- Altura do nó ao nó folha
- Profundidade da raiz ao nó
- A profundidade de uma árvore ZERO.
- Nós internos / nós externos / nível / descendentes; folha, raiz, subárvores.

3. ÁRVORES N-ÁRIAS

Algumas vezes, a árvore binária não é a melhor escolha se lidarmos com grandes volumes de dados, vai aumentando a altura, e neste caso, a recursividade irá gastar bastante memória.

Se lidar com grandes volumes de dados, sua profundidade cresce e dependendo da modelagem do problema, cada nó precisa ter um número variado de filhos, de 0 a n, com n > 2.

O que fazer? Precisamos de mais filhos, mais filhos é o que teremos – árvore n-ária.

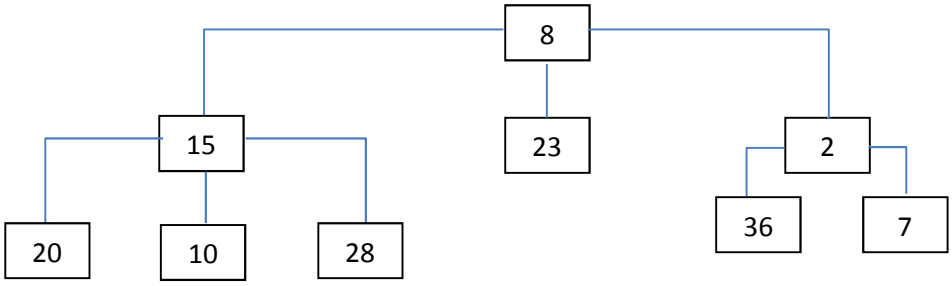
Definição:

- ✓ 1. Uma árvore n-ária é uma árvore em que cada nó pode ter até n filhos.
- ✓ 2. Uma árvore n-ária é uma árvore em que cada nó pode ter um número ilimitado de filhos.

Árvore n-ária trata-se de uma generalização das árvores binárias.

E qual definição usar? A que melhor se adaptar ao problema modelado. Exemplificando a definição 2, uso mais geral.

Suponha que queremos modelar a seguinte árvore:



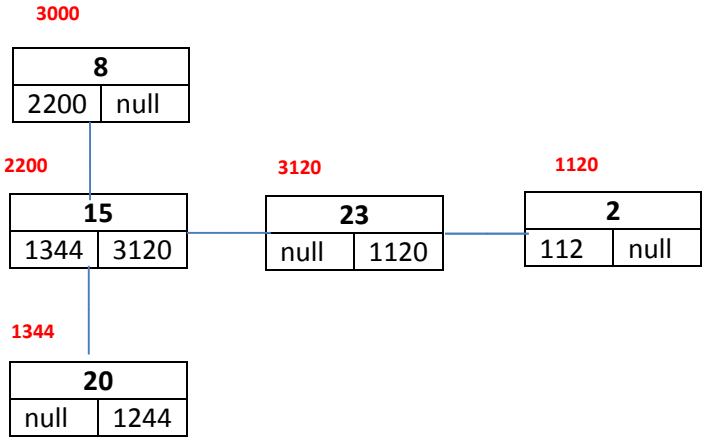
Neste tipo de árvore n-ária não há ordem nos nós... E como seria a representação computacional do nó?

Note: Numa árvore n-ária de pesquisa poderemos definir uma ordem.

Uma possibilidade seria:

chave	
filho	irmão

Então, nossa árvore ficaria...




```
#include <stdio.h>
#include <stdlib.h>
#define false 0
#define true 1
typedef int bool;
typedef int PRIMARY_KEY;
typedef struct pontNo {
        PRIMARY_KEY chave;
        struct pontNo *primFilho;
        struct pontNo *proxIrmao;
} NO;

typedef NO* PONT;
PONT Inicializar();
PONT adicionarElemento(PONT raiz, PONT no);
PONT criarNovoNo(PRIMARY_KEY ch);
void   exibirArvore(PONT raiz);
void   bucarChaveArvore(PRIMARY_KEY ch , PONT raiz);

PONT Inicializar(PRIMARY_KEY ch) {
  return(criaNovoNO(ch);
};

PONT criarNovoNo(PRIMARY_KEY ch){
        PONT novo = (PONT) malloc(sizeof(NO));
        novo->primFilho = NULL;
        novo->proxIrmao = NULL;
        novo->ch;
        return(novo);
}
```

```
bool adicionarElemento(PONT raiz, PRIMARY_KEY novaChave, PRIMARY_KEY chavePai){
        PONT pai = buscaChave(chavePai, raiz);
        if (!pai) return false;
        PONT filho = criaNovoNo(novaChave);
        PONT p = pai->primFilho;
        if (!p) pai->primFilho = filho;
        else {
                while ( p->proxIrmao)
                  p = p->proxIrmao;

                 p->proxIrmao = filho;
         }
        return (true);
};
```

Nota: Verificamos se o pai existe. Cria o nó para o filho e verifica-se o primogênito desse pai. Senão houver primogênito, o novo nó é o primeiro filho. Do contrário, vamos ao último filho, neste caso, inserimos o novo nó com caçula.

```
Int main(void){
   PONT r = Inicializar(8);
}
```

```
PONT buscarChaveArvore (PRIMARY_KEY ch, PONT raiz) {
        if (raiz == NULL) return NULL;

        if ( raiz->chave == ch) return raiz;

        PONT p = raiz->primFilho;

        while ( p ) {
                PONT resp = buscarChaveArvore (ch , p);
                If ( resp ) return resp;
                p = p->proxIrmao;
        }
        Return (NULL);
}
```

Efetuamos a busca. Senão existir a raiz, retorna NULO. Senão encontramos, olhamos os filhos. E, para cada filho, buscamos na subárvore de que ele é raiz.

```
void exibirArvore (PONT raiz) {
        if ( raiz == NULL ) return;

        printf("%d (", raiz->chave);
        PONT p = raiz->primFilho;
        while ( p ) {
             exibirArvore (p);
             p = p->proxIrmao;
        }
        printf(")");
```

}

Saída: 8(15(20()10()28())23()2(36()7()))

E a exclusão? A exclusão vai depender muito do contexto em que ela será usada. Pode até nem ser necessária. Em especial, temos que decidir o que fazer com os nós filhos do nó excluído. Serão adotados por alguém? Ou serão eliminados juntos? É uma decisão de projeto.

4. ÁRVORES N-ÁRIAS – Trie ANP

É um tipo específico de uma árvore n-ária. Uma trie (de retrieval), é uma árvore n-ária projetada para recuperação rápida de chaves de busca.

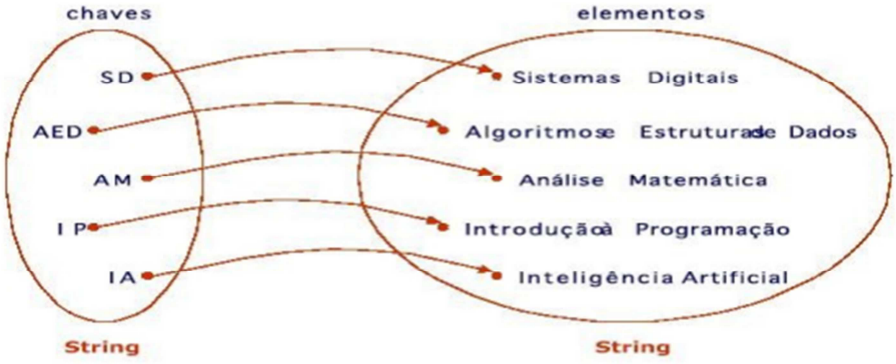
TRIE vem de RETRIEVAL – RECUPERAÇÃO; Pronúncia: TRI ou TRAI; É um tipo de árvore de busca. Idéia geral: usar partes das CHAVES como caminho busca.

Vantagens de uma ANP, indexação de um grande volume de dados.

Características:

Ela não armazena nenhuma chave explicitamente. Chaves são codificadas nos caminhos a partir da raiz. Todo nó tem um campo valor, retornado caso a chave termine nesse nó.

Cada chave formada por palavras sobre um alfabeto. Palavras com tamanho variável e ilimitado. Em geral associam-se chaves a elementos ou registros, como na tabela Hash.



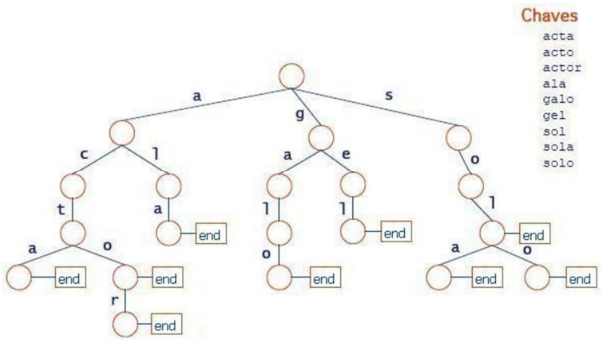
Cada chave é formada a partir de alfabeto de símbolos

Exemplos de alfabetos: {0,1}, {A, B, C, D, E,...Z}, {0,1,2,3,4,5,...,9}
Exemplos de chaves: ABABBBABABA 19034717 Maria 01010101000000000010100000000010 10
Chaves parcialmente partilhadas entre os elementos.

Trie – Estrutura

- Descendentes de mesmo nó com mesmo prefixo
- Raiz: cadeia vazia
- Valores ou elementos associados a folhas ou a alguns nós internos de interesse
- O caminho da raiz para qualquer outro nó é um prefixo de uma string
- O grau corresponde ao tamanho do alfabeto
- A trie pode ser vista como um autômato finito
- Cada nível percorrido corresponde a avançar um elemento na chave

Árvore ordenada e n-ária



5. ÁRVORES AVL – Algoritmos de Adelson-Velsky e Landis

É uma árvore binária de busca balanceada com relação à altura de suas subárvores. São as árvores que minimizam o número de comparações efetuadas no pior caso para uma busca com chaves de probabilidades de ocorrências idênticas. Todas as operações de buscas, inserção e remoção com complexidade O(logn), no qual n é o número de elementos da árvore, que são aplicados a árvore de busca binária.

Vimos que a ordem de inserção determina o formato de uma árvore de busca binária. Pode ficar balanceada, perfeitamente balanceada ou degenerada. Vimos também que isso era a diferença entre ela se comportar como numa busca binária ou uma busca sequencial.

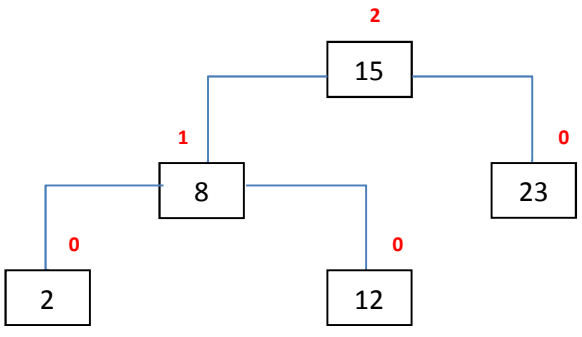
Balanceamento é importante. Contudo, um balanceamento perfeito é algo computacionalmente caro. Podemos permitir um bom balanceamento, em que pode haver um pouco de desbalanceamento.

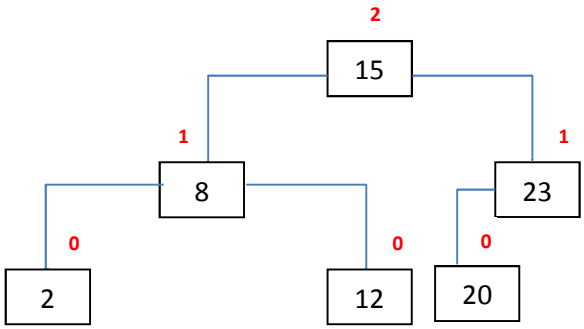
Numa AVL verifica a altura das subárvores da esquerda e da direita, garantindo que essa diferença não seja maior que +-1, esta diferença é o fator de balanceamento do nó, fB = hEsquerda – hDireita.

Nota: A altura de um nó é o comprimento do maior caminho a partir da subárvore até o nó folha.

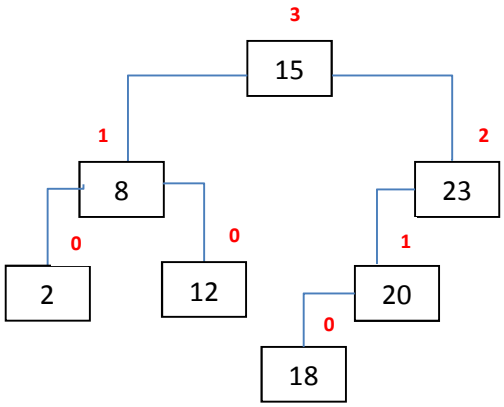
O fator de balanceamento é calculado a cada nó. E, para cada nó, a diferença de altura entre a subárvore da esquerda e da direita não pode passar +-1. Sendo que, **a altura de uma árvore VAZIA é -1**. O fator de balanceamento, ou alternativamente a altura do nó, é armazenada no próprio nó.

Exemplos: **AVL** **AVL**





Se inserirmos o elemento 18? Não é uma árvore AVL

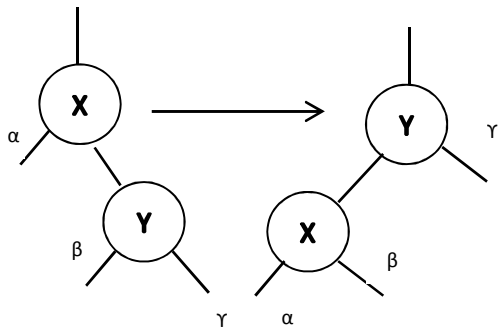


Uma inserção pode fazer com que o fator de balanceamento de um nó vire +-2. Contudo somente nós no caminho do ponto de inserção até a raiz pode ter mudado sua altura.

Então, após a inserção, voltamos até a raiz, nó por nó, atualizando as alturas. Se um novo fator de balanceamento para um determinado nó for 2 ou -2, ajustamos a árvore rotacionando em torno desse nó.

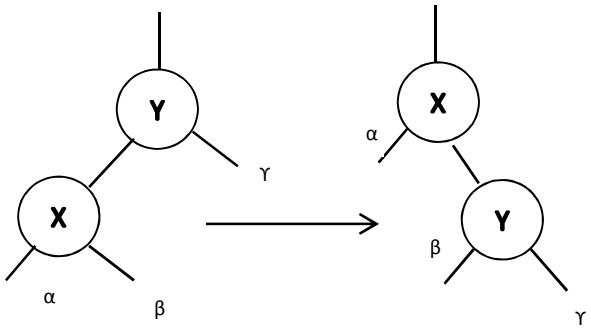
Existem dois tipos de rotação:

Rotação à esquerda

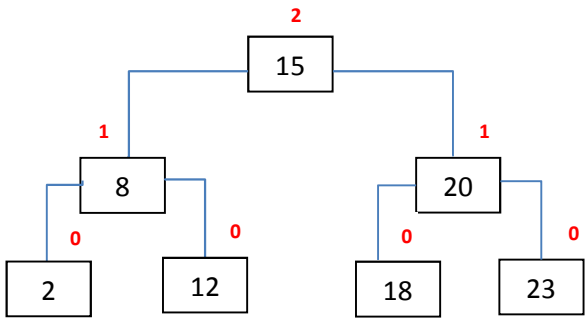


Rotação à direita

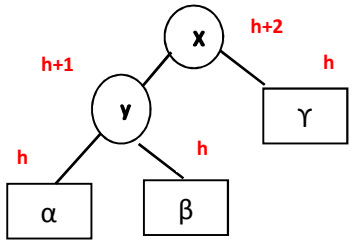
c



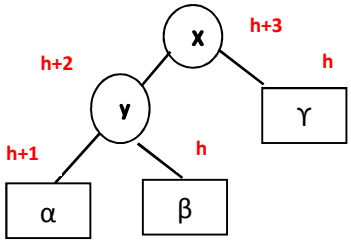
Balanceando rotacionando à direita à **árvore não AVL acima** em torno do nó



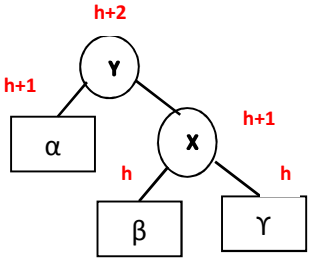
Considere a árvore AVL abaixo:



O que pode acontecer se inserirmos um elemento em α? Quebramos o balanceamento em X. Como consertamos o balanceamento?



Fazendo a rotação à direita.



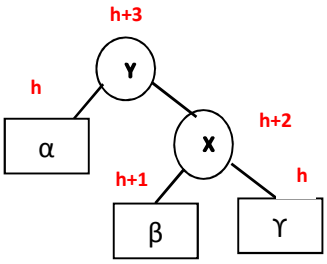
E assim, “erguendo” a parte que havia ficado maior.

Inserções na parte mais externa da árvore são resolvidas com rotações simples. Ou seja, na subárvore esquerda do filho esquerdo do nó desbalanceado. Ou na subárvore direita do filho direito desse nó.

E quando a inserção é na parte mais interna? Na subárvore direita do filho esquerdo do nó desbalanceado, ou na subárvore esquerda do filho direito desse nó.

O que pode acontecer inserirmos um elemento em β? Quebramos novamente o balanceamento em X.

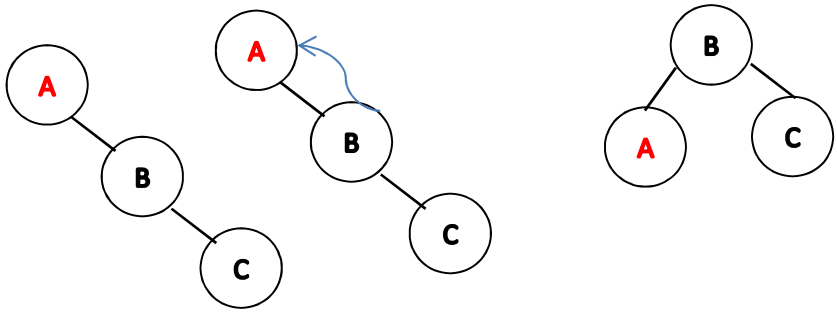
Como resolvemos? Tentemos a rotação à direita.



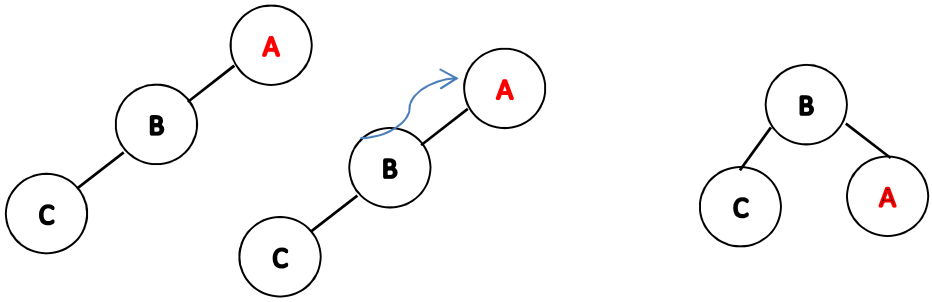
Então, enquanto inserções na parte mais externa da árvore são resolvidas com rotações simples. Inserções na parte mais interna da árvore são resolvidas com rotações duplas.

Resumindo:

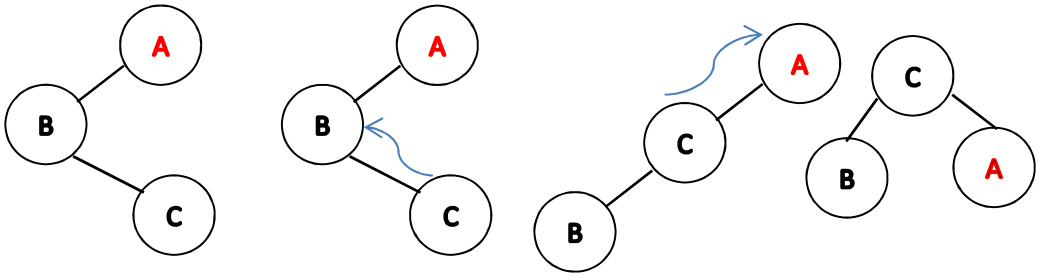
- Fazemos uma rotação à esquerda se um nó foi inserido na subárvore direita do filho à direita.



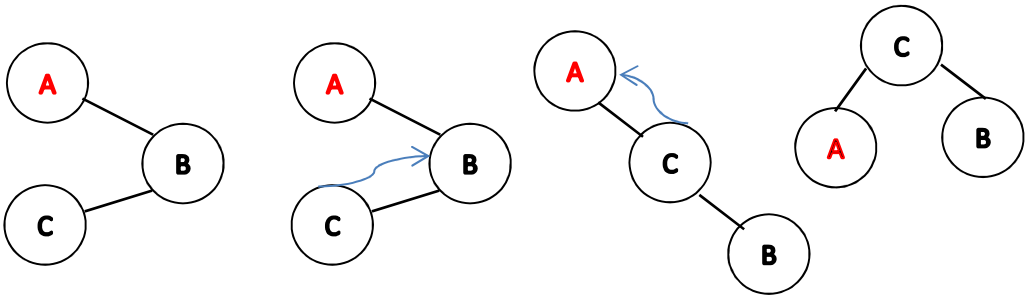
- Fazemos uma rotação à direita se um nó foi inserido na subárvore esquerda do filho à esquerda.



- Fazemos uma rotação esquerda-direita se um nó foi inserido na subárvore direita do filho à esquerda.



- Fazemos uma rotação direita-esquerda se um nó foi inserido na subárvore esquerda do filho à direita.



Por fim, as árvores de busca binárias tem um grande problema porque elas podem ficar podem ficar desbalanceadas, conforme vão inserindo elementos. E dependendo da ordem elas podem ficar tão ruins, quanto uma lista ligada comum, porque elas perdem todas as características que nós gostaríamos que elas tivessem de realizar um busca binária nesta estrutura de dados. Uma solução para mantê-la aproximadamente balanceada é utilizar árvores AVL.

Árvores AVL, são árvores de buscas binárias em que, para cada nó, a altura de sua subárvore esquerda difere da sua subárvore direita em no máximo +1 ou -1.

Vimos que, para manter essa propriedade da árvore AVL, fazíamos rotação de vértices. Fazemos uma rotação em torno do nó desbalanceado. Ainda assim, por vezes é necessário fazer duas rotações em sentidos opostos.

Para manter a propriedade da árvore AVL, fazemos rotação vértices, sempre em torno do nó desbalanceado.

Ainda assim, por vezes é necessário fazer duas rotações em sentidos opostos, uma em torno do nó logo abaixo do nó desbalanceado, outra em torno do nó desbalanceado.

6. ÁRVORES B

Em ciência da computação, uma árvore B é uma estrutura de dados em árvore, auto-balanceada, que armazena dados classificados e permite pesquisas, acesso sequencial, inserções e remoções em tempo logarítmico. A árvore B é uma generalização de uma árvore de pesquisa binária em que um nó pode ter mais que dois filhos.[1] Diferente das árvores de pesquisa binária auto-balanceadas, a árvore B é bem adaptada para sistemas de armazenamento que leem e escrevem blocos de dados relativamente grandes, como discos.

Árvores B são estruturas usada para implementar TSs (tabelas de símbolos) muito grandes. Uma árvore B pode ser vista como um índice (análogo ao índice de um livro) para uma coleção de pequenas TSs: o índice diz em qual das pequenas TSs está a chave que você procura. Pode-se dizer que uma árvore B é uma TS de TSs.

Resumo:

- memória rápida e memória lenta (HDD ou SSD) do computador
- páginas e a classe Page
- páginas externas e páginas internas
- busca e inserção
- a classe BTreeSET

Definição (Bayer e McCreight, 1972): Para qualquer inteiro positivo par M, uma árvore B (B-tree) de ordem M é uma árvore com as seguintes propriedades:

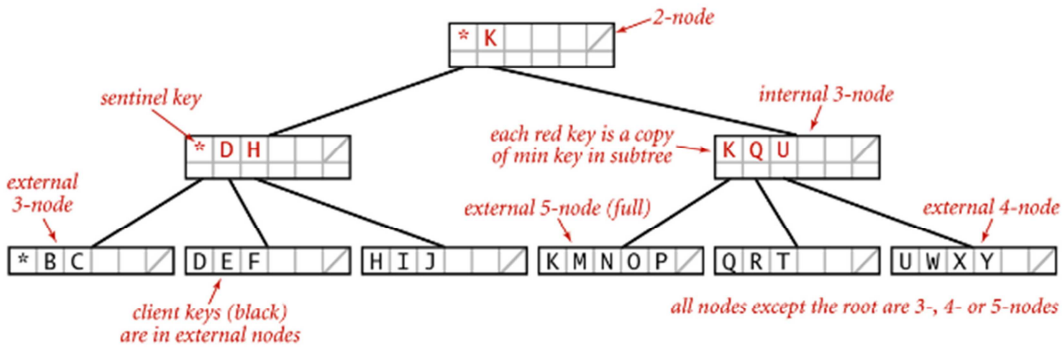
- cada nó contém no máximo M−1 chaves,
- a raiz contém no mínimo 2 chaves e cada um dos demais nós contém no mínimo M/2 chaves,
- cada nó que não seja uma folha tem um filho para cada uma de suas chaves,
- todos os caminhos da raiz até uma folha têm o mesmo comprimento (ou seja, a árvore é perfeitamente balanceada).
-

Uma árvore B de ordem 4 é essencialmente uma árvore 2-3 (embora existam algumas difereças que destacaremos adiante).

Aplicações. Árvores B são a estrutura subjacente a muitos sistemas de arquivos e bancos de dados. Por exemplo,

- o sistema de arquivos NTFS do Windows,
- o sistema de arquivos HFS do Mac,
- os sistemas de arquivos ReiserFS, XFS, Ext3FS, JFS do Linux, e
- os bancos de dados ORACLE, DB2, INGRES, SQL e PostgreSQL.

Exemplo:



Anatomy of a B-tree set (M = 6)