Árvore:

Seções 5.1 e 7.2 do livro "Estruturas de Dados Usando C" de Tenembaum, Langsam e Augenstein
Seções 5.4 e 5.5 do livro do Sedgewick

Dado um conjunto de vértices e arestas, um caminho é uma lista de vértices distintos na qual cada vértice na lista é conectado ao próximo por uma aresta.

Árvore (livre): Um conjunto de vértices (nodos) e arestas que satisfaz a seguinte condição: existe exatamente um caminho conectando qualquer par de nodos.
Se houver algum par de nodos para o qual existe mais de um caminho ou nenhum caminho temos um grafo.

Floresta: Um conjunto de árvores disjuntas.

Em computação:

Em geral, árvores referem-se a estruturas que possuem um nodo designado como raiz. Nestas árvores, cada nodo é a raiz de uma subárvore.

Desenho da árvore:

Raiz no topo:

- existe a noção de um nodo estar acima (mais próximo da raiz) ou abaixo dele (mais longe da raiz)
- PAI: todo nodo, exceto a raiz tem um único pai, que é o nodo logo acima dele
- FILHOS: são os nodos logo abaixo de um determinado nodo
- IRMAO / AVO / ASCESTRAL/ DESCENDENTE
- FOLHAS ou nodos terminais: nodos que não possuem filhos
- nodos INTERNOS ou não terminais: que possuem filhos
- árvores ORDENADAS: árvores nas quais a ordem dos filhos é significativa
- árvores n-aria: árvores nas quais todos os nodos internos obrigatoriamente tem "n" filhos. Ex: árvore binária.

Nestas árvores em geral é utilizado o conceito de "nodo externo" (que não possui filhos), referenciado por nodos internos que não tem o número especificado de filhos. Neste caso, FOLHA é um nodo interno cujos filhos são todos nodos externos.

Nível de um nó:

Nível da raiz = 0
Nível de outros nós = nível do pai + 1

Altura da árvore:

Nível máximo de um nodo (interno ou externo) da árvore.

Árvore Binária:

Um árvore binária é ou um nodo externo ou um nodo interno conectado a um par de árvores binárias, chamadas de subárvore esquerda e subárvore direita do nodo.

Representação:

```
typedef struct nodo *ApNodo;
struct nodo {
  Item item;
  ApNodo esq, dir;
}
```

Propriedades:

1) uma arv. binária com N nodos internos tem N+1 nodos externos.

Prova por indução:
$N = 0$ --> árvore vazia com apenas um nodo externo
$N > 0$ --> a raiz de uma árvore binária tem na
subarv. esq: k nodos internos, $0 <= k <= N-1$
subarv. dir: $N-k-1$ nodos internos
Por hip.ind. a subarv. esq tem k+1 nodos externos e a subarv. dir $N-k-1+1$ nodos externos.
Assim, a árvore tem $(k+1) + (N-k-1+1) = N+1$ nodos externos.

Árvore binária completa de altura *d* é uma árvore binária na qual todos os nodos externos estão no nível d.

2) a quantidade de nodos externos em uma arv.bin.compl de altura $d = 2^{d}$

3) quantidade de nodos em uma árvore binária completa de altura $d = 2^{d+1} - 1$

Prova por indução em d:
Base: n = 0 (árvore vazia)
$\#nodos = 2^{0+1} - 1 = 1$ *(1 nodo externo)*
Hipótese da indução:
Uma árvore binária completa de altura d tem $2^{d+1} - 1$ nodos.
Passo da indução:
Uma arv.bin.compl de altura d tem 2^d nodos externos. Assim, aumentando 1 a altura da árvore, cada um destes nodos passa a ser pai de dois outros nos. Assim, # nodos em uma árvore de altura d+1
$= (2^{d+1} - 1) + (2^d * 2)$ os que já existiam + novos nodos
$= 2^{d+1} - 1 + 2^{d+1}$
$= 2 * 2^{d+1} - 1$
$= 2^{d+2} - 1$

4) quantidade de nodos internos em uma árvore binária completa de altura $d = 2^{d} - 1$. Consequência direta das propriedades 1 e 3.

Altura de uma arv.bin.completa com n "chaves" (nodos internos) = $log_2 (n+1)$

Árvore Binária quase Completa de Altura *d*:

Uma arv. binária na qual:

1. todos os nodos externos estão no nível d ou d-1
2. se um nó nd na árvore tem algum descendente direito no nível d (o máximo da árvore), então todos os nodos externos que são descendentes esquerdos de nd estão também no nível d.

Numeração dos nodos:

num(raiz) = 1
num(n) = *2 * num(np)*, se n é filho esquerdo de np
*2 * num(np) + 1*, se n é filho direito de np

Altura de uma arv.bin.quase completa com n nodos internos:

teto((log_2 (n+1))

Isso representa uma árvore que tem a quantidade de nodos internos entre uma árvore binária completa de altura *d-1 (2^{d-1} - 1 nodos)* e uma árvore binária completa de altura *d = (2^{d} - 1 nos)*.

Árvore Binária - Implementação

Seções 5.6, 5.7, 12.2 (Sedgewick)

Representações de Árvore Binária:

- vetores
- apontadores

```
typedef struct no *Apontador;
typedef struct no{
  long chave;
  ...
  Apontador esq, dir;
} No;
```

Percurso:

Pre-ordem: visita o nodo, subarv. esq, subarv. direita
Em ordem: visita a subarv. esq., nodo, subarv. direita
Pós-ordem: visita a subarv. esq, subarv. direita, nodo

Função Recursiva:

```
void preOrdem( Apontador p )
{
  if( p == NULL ) return;
  printf("%ld\n", p->chave);
  preOrdem( p->esq );
  preOrdem( p->dir );
}
```

Função Não Recursiva:

```
void preOrdemPilha( Apontador p )
{
  pilha s;

  inicializaPilha( s );
  push( s, p );
  while( !vazio(s) ){
    p = pop( s );
    if( p != NULL ){
      printf("%ld\n", p->chave);
      push( p->dir, s );
      push( p->esq, s );
    }
  }
}
```

Observe que esta função não pode ser diretamente estendida para percursos em ordem e pós-ordem. Uma possibilidade para resolver o problema é fazer distinção entre empilhar "árvore" e empilhar "chave".

Percurso por nível (utilizando uma fila)

```
void percursoPornível( Apontador p ){
  if( p == null )
    return;
  inicializaFila( f );
  putFila( f, p );
  while( !vazio( f )){
    p = getFila( f );
    printf( "%ld\n", p->chave );
    if( p->esq != null )
      putFila( f, p->esq );
    if( p->dir != null )
      putFila( f, p->dir );
  }
}
```

Contagem de Número de Nodos (internos) da Árvore:

```
int contaNodo( Apontador p ){
  if( p == NULL ) return 1;
  return contaNodo( p->esq ) + contaNodo( p->dir ) + 1;
}
```

Altura de uma árvore:

```
int altura( Apontador p ){
  int he, hd;

  if( p == NULL ) return 0;
  he = altura( p->esq );  hd = altura( p->dir );
  if (he > hd )
    return he+1;
 else
    return hd+1;
}
```

Nível do nodo que contém uma chave k:

- a função retorna -1 se k não existir na árvore
- chamada: nívelChave(raiz, k, 0)

```
int nívelChave( Apontador p, long k, int nível){
  int result;

  if( p == null )
    return -1;
  else if( p->chave == k )
   return nível;
  else {
    result = nívelChave( p->esq, k, nível+1 );
    if( result >= 0 )
      return result;
    else
      return nívelChave( p->dir, k, nível+1 );
  }
}
```

Tipo abstrato de dados (TAD):

É um conjunto de operações associado a uma estrutura de dados, de tal forma que haja independência de implementação para as operações

Dicionário ou Tabela de Símbolos:

Um dicionario é um TAD que contem itens com chaves e que dá suporte a seguintes operações básicas: inicializa, insere um novo elemento, remove um elemento e pesquisa o item que contem uma determinada chave.

- analogia com dicionario da língua portuguesa: chave = palavra, registro= pronuncia, sinônimo, definição, etc.

Árvore Binária de Busca

Seção 12.5, 12.6, 12.8, 12.9 (Sedgewick)

É uma árvore binária na qual para todo no n, os elementos da subárvore a esquerda contem chaves com valores menores que a chave de n e a subárvore a direita contem chaves maiores que a chave de n.

Implementação:

A implementação abaixo usa um nodo (nodoNull) para a representação de nodos externos ao invés do valor NULL:

```
#include <stdio.h>

typedef struct nodo *Apontador;
typedef struct nodo {
  longInt chave;
  ...
  Apontador dir, esq;
  int contaNodo;
} Nodo;

static Apontador raiz, nodoNull;

Apontador criaNodo( longint chave, Apontador esq, Apontador dir, int N ){
  Apontador p;

  p = malloc( sizeof *p );
  p->chave = chave; p->esq = esq; p->dir = dir; p->contNodo = N;
  return p;
}

void inicializaDic() {
  nodoNull = criaNodo( 0, 0, 0, 0 );
  raiz = nodoNull;
  return;
}

Apontador buscaArv( Apontador p, longInt chave ){
  if (p == nodoNull) return nodoNull;
  if (chave == p->chave) return p;
  if (chave < p->chave)
    return buscaArv( p->esq, chave )
  else return buscaArv( p->dir, chave );
}

Apontador insereArv( Apontador p, longint chave){
  if( p == nodoNull) return criaNodo( chave, nodoNull, nodoNull, 1 );
  if( chave < p->chave)
    p->esq = insereArv(p->esq, chave);
  else
    p->dir = insereArv(p->dir, chave);
  (p->contaNodo)++;
  return p;
}
```

Inserção - Função Iterativa:

```
Apontador insereArv( longint chave ){
  Apontador p, paiP;
```

```
    if( raiz == nodoNull ){
      raiz = criaNodo( chave, nodoNull, nodoNull, 1 );
      return raiz;
    }
    p = raiz;
    while( p != nodoNull ){
      paiP = p;
      if( chave < p->chave )
        p = p->esq;
      else
        p = p->dir;
    }
    p = criaNodo( chave, nodoNull, nodoNull, 1 );
    if( chave < paiP->chave )
      paiP->esq = p;
    else
      paiP->dir = p;
    return p;
  }
```

Ordenação por Árvore:

Composta por 2 passos:

1. pre-processamento: geração da árvore de binária de busca
2. percurso da árvore em-ordem

```
void ordena( longint v[], int tamV ){
  int i;

  inicializaDic();
  for( i=0; i < tamV; i++ )
    insereArv( raiz, v[i] );
  emOrdem( raiz );
  return;
}
```

Custo:

Pior caso:

n^2, no caso dos elementos serem lidos em ordem ascendente ou descendente

Caso médio:

n log(n), a altura de uma árvore balanceada é *chão(log_2 (n))*. Assim, para inserir cada um dos n elementos na árvore são necessárias no máximo *log(n)* comparações.

Para obter uma árvore balanceada:

Após a entrada de uma chave k, metade dos elementos tem chave menor que k e metade tem chaves maiores que k.

Custo médio de busca:

(em uma árvore binária com n nos, considerando que todos os nos tem igual probabilidade de serem acessados)

(s+n) / n, onde s é o comprimento do caminho interno, ou seja, a soma do comprimentos dos caminhos entre a raiz e cada um dos nos da árvore. Isto se deve ao fato de para acessar um determinado no de nível l, são necessárias l+1 comparações.

- pode ser mostrado que o número esperado de comparações em uma arv. de pesquisa randômica é 1.39 log(n), ou seja, somente 39% pior que a árvore completamente balanceada.

Inserção na raiz:

Exemplo:

```
              20
         10        30
               25     35
              21 27
```

Inserção da chave 26:

```
             26
         20        30
      10        25     35     que viola a def. de Arv.Bin.Busca
             21 27
```

Solução:

Série de rotações após a inserção do elemento na folha:

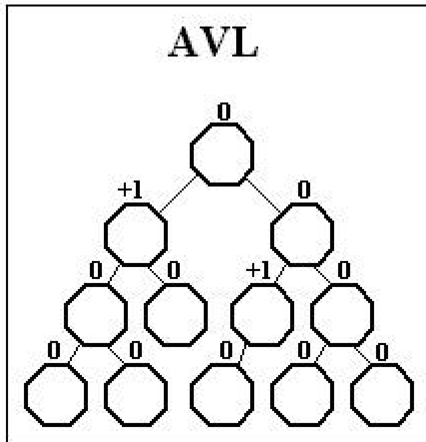
```
              20
         10          30
                 25     35
              21    27
                  26
```

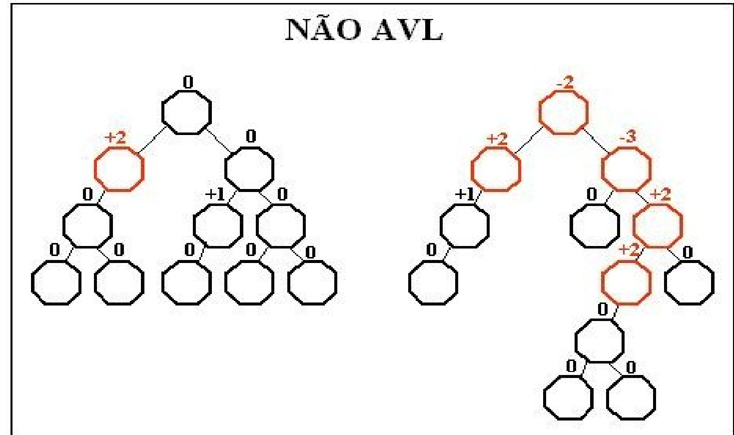
1. rotação a direita de 27
2. rotação a esq. de 25
3. rotação a direita de 30
4. rotação a esquerda de 20

```
Apontador rotDir( Apontador p ){
  Apontador q;

  q = p->esq;  p->esq = q->dir; q->dir = p;
  return q;
}

Apontador rotEsq( Apontador p ){
  Apontador q;

  q = p->dir;  p->dir = q->esq; q->esq = p;
  return q;
}

Apontador insereArv( Apontador p, longint chave ){
  if( p == nodoNull ) return criaNodo( chave, nodoNull, nodoNull, 1 );
  if( chave < p->chave ){
    p->esq = insereArv( p->esq, chave );
    p = rotDir( p );
  } else {
    p->dir = insereArv( p->dir, chave );
    p = rotEsq( p );
  }
  return p;
}

void insereDic( longint chave ){
  raiz = insereArv( raiz, chave );
  return;
}
```

Árvore Balanceada AVL (Árvore de Adelson, Velskii e Landis)

http://www.youtube.com/watch?v=mKMwi691rs8 (árv. Binárias, descrição, fator de balanceamento)
http://www.youtube.com/watch?v=s5w_Gny8B4A (inserção, rotação, remoção)
http://www.lcad.icmc.usp.br/~nonato/ED/
http://www.youtube.com/watch?v=1PGtyt7_EAw (exemplo de inserção e rotação)





Motivação:

Garantir custo O(log n) para busca, inserção e remoção

Abordagem:

Manter a árvore balanceada após cada operação

Descrição:

Uma árvore binária de busca na qual, para todos os nós, as alturas de suas subárvores não diferem em mais de 1.

O balanceamento de um nó pode ser -1, 0, ou 1 dependendo de a altura da subárvore a esquerda ser menor, igual ou maior que a subárvore a direita, respectivamente. Ou seja, uma árvore está desbalanceada se contiver nodos com balanceamento menor que -1 ou maior que 1.

Balanceamento:

Rotação a direita e a esquerda:

Seja x o nodo com maior nível no qual ocorre um desbalanceamento. 4 casos a considerar. Inserção na:

a) subárvore esquerda do filho esquerdo de x
b) subárvore direita do filho direito de x
c) subárvore direita do filho esquerdo de x
d) subárvore esquerda do filho direito de x

Os casos a) e b) são resolvidos com UMA rotação para balancear a árvore.
Os casos c) e d) precisam de DUAS rotações.

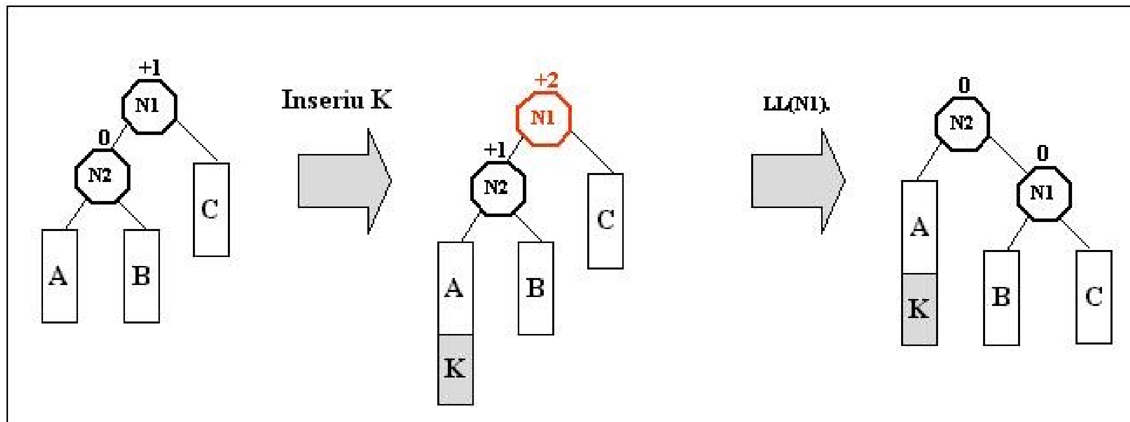
Rotação, inserção e altura das subárvores:

Rotação a direita do nó n:
-o balanceamento de n e de todos os seus ancestrais (depois da rotação) diminui de 1

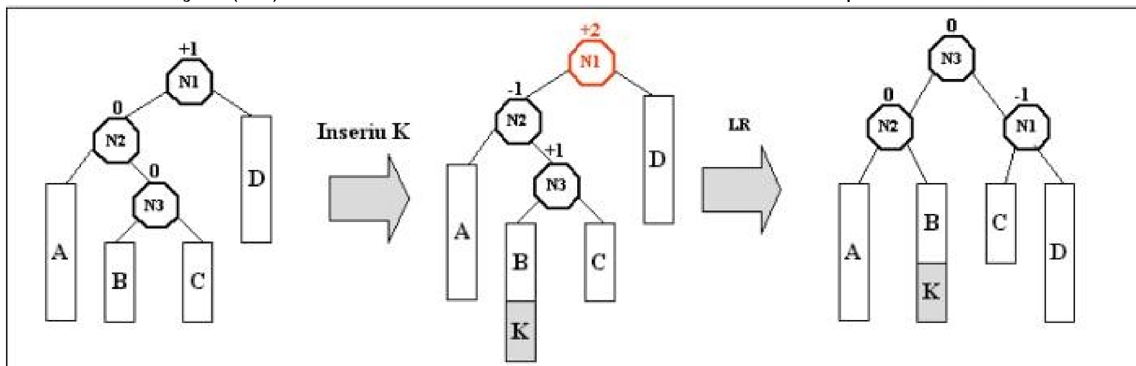
Rotação a esquerda do nó n:

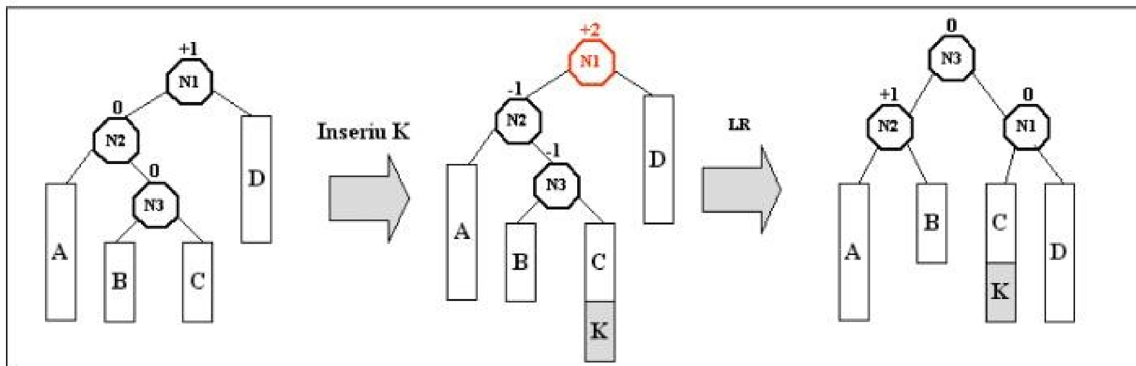
-o balanceamento de n e de todos os seus ancestrais (depois da rotação) aumenta de 1

- Rotação (LL): O novo nó X é inserido na sub-árvore da esquerda do filho esquerdo de A;

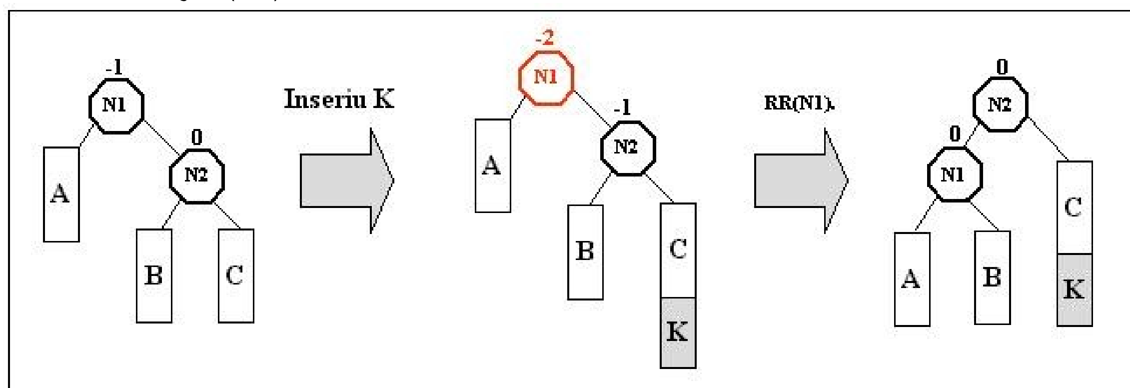


- Rotação (LR): X é inserido na sub-árvore da direita do filho esquerdo de A;

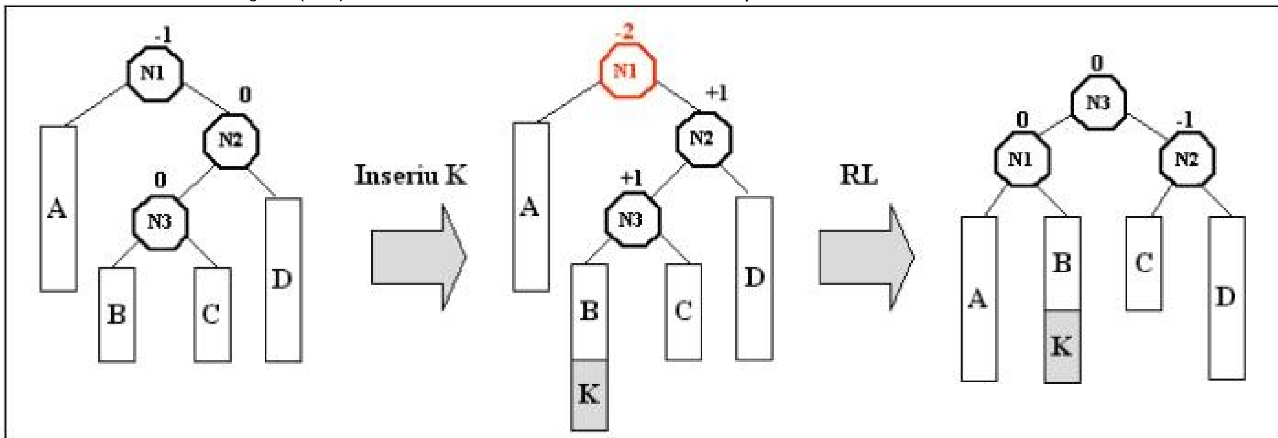


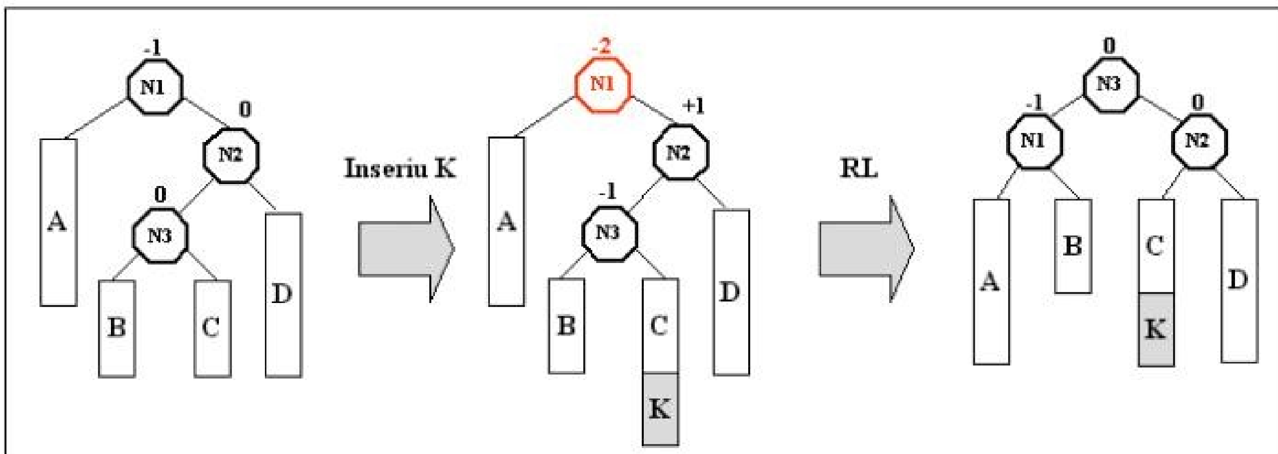


- Rotação (RR): X é inserido na sub-árvore da direita do filho direito de A;



- Rotação (RL): X é inserido na sub-árvore da esquerda do filho direito de A.





Inserção de um novo nodo n:

1. insere n na árvore binária, guardando o ancestral 'a' de nível mais alto que PODE ficar desbalanceado ('a' inicialmente é a raiz)
2. altera os balanceamentos de todos os nodos no caminho de 'a' até n

```
se chave(n) < chave(a)
   bal(a) = bal(a) + 1
senão
   bal(a) = bal(a) - 1
```

3. balanceia a árvore e acerta balanceamentos

```
se estiver desbalanceado para a direita -- bal(a) < -1
  f = direita(a)
  se bal(direita(a)) == -1
     rotaçãoEsquerda(a)
     bal(a) = bal(f) = 0
  senão
     neto = esquerda(f)
     rotaçãoDireita(direita(a))
     rotaçãoEsquerda(a)
     se  bal(neto) == 0
       bal(a) = bal(f) = 0
     senão se bal(neto) > 0
       bal(a)= 0
       bal(f) = -1;
     senão
       bal(a)  = 1;
       bal(f) = 0;
```

```
  bal(neto) = 0;

se estiver desbalanceado para a esquerda -- bal(a) > 1
  f = esquerda(a)
  se bal(f) == 1
     rotaçãoDireita(a)
     bal(a) = bal(f) = 0
  senão
     neto = direita(f)
     rotaçãoEsquerda(esquerda(a))
     rotaçãoDireita(a)

     se bal(n) == 0
       bal(a) = bal(f) = 0
     senão se bal(neto) > 0
       bal(a) = -1;
       bal(f) = 0;
     senão
       bal(a) = 0;
       bal(f) = 1;
     bal(neto) = 0
```

```
Apontador Insere(Registro r, Apontador p, int *mudouAltura){
  if( p == nodoNull ){
    *mudouAltura = TRUE;
    return criaNo( r, nodoNull, nodoNull );
  }
  if( r.Chave <= p->Reg.Chave ){
    p->Esq = Insere( r, p->Esq, mudouAltura );
    if( *mudouAltura ){
      p->bal++;
      if( p->bal == 0 )
        *mudouAltura = FALSE;
      else if( p->bal == 2){
        *mudouAltura = FALSE;
        balanceia( &p );
      }
    }
  } else {
    p->Dir = Insere( r, p->Dir, mudouAltura );
    if( *mudouAltura ){
      p->bal--;
      if( p->bal == 0 )
        *mudouAltura = FALSE;
      else if( p->bal == -2){
        *mudouAltura = FALSE;
        balanceia( &p );
      }
    }
  }
  return p;
}
```

Remoção de nodo:

Ideia:

1. fazer busca do nodo que contem a chave a ser removida (nodoK)
2. se nodoK for uma folha então remover nodoK, caso contrario:
   - encontrar o nodo com maior chave na subarv. esquerda ou
   - menor chave na subárvore direita (nodoRem)
   - remover nodoRem da árvore

- substituir a chave em nodoK pela chave em nodoRem

Cuidados na remoção do novo da árvore:

- caso o nodo ficar desbalanceado, balancear da mesma forma que na inserção
- a remoção de um nodo pode causar diversas operações de balanceamento a partir de nodoRem até a raiz
- a mesma estrategia utilizada pela inserção de manter um parâmetro (mudouAltura) pode ser usada na remoção: ele é verdadeiro caso a subárvore mudou de altura e falso caso contrario.
- a mudança de altura da subárvore com raiz em "n" pode ser verificada testando o balanceamento original de n: se o balanceamento for zero, uma remoção não vai alterar sua altura, uma vez que a remoção de um nodo na subárvore do filho esquerdo não altera a altura da subárvore do filho direito (e vice-versa). Portanto a altura da subárvore com raiz em n também não se altera.
- observe que uma operação de balanceamento em um nodo n também altera sua altura.

```
void Remove(TipoChave k, Apontador *raiz ){
  Apontador nodoK, nodoRem;
  Registro regRem;
  int mudouH;

  /* busca nodo que contem chave k */
  nodoK = busca( k, *raiz );
  if( nodoK == nodoNull )
    return;

  /* busca nodo com dados que vao substituir chave k que sera' removida */
  if( nodoK->Dir == nodoNull && nodoK->Esq == nodoNull )
    nodoRem = nodoK;
  else if(nodoK->bal > 0 )
    nodoRem = buscaMaior( nodoK->Esq );
  else
    nodoRem = buscaMenor( nodoK->Dir );
  regRem = nodoRem->Reg;

  /* remove nodoRem da árvore */
  /* nodoRem é folha ou tem um único filho */
  *raiz = removeR( nodoRem, *raiz, &mudouH );
  nodoK->Reg = regRem;
  return;
}

Apontador removeR( Apontador nodoRem, Apontador p, int *mudouH ){
  Apontador filho;

  /* remove nodoRem: se for folha retorna nodoNull;
     caso constrario retorna o endereço do seu único filho */
  if( p == nodoRem ){
    if( p->Dir != nodoNull )
      filho = p->Dir;
    else if( p->Esq != nodoNull )
      filho = p->Esq;
    else
      filho = nodoNull;
    free( p );
    *mudouH = TRUE;
    return filho;
  }
  else if( nodoRem->Reg.Chave < p->Reg.Chave ){
    p->Esq = removeR( nodoRem, p->Esq, mudouH );
    if( *mudouH ){
      if( p->bal == 0 )       /* se o balanceamento era originalmente = 0
*/
```

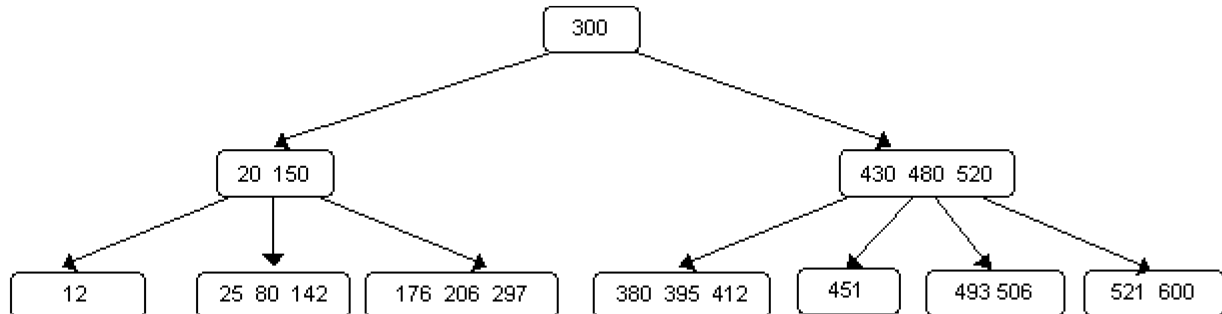
```
          *mudouH = FALSE;      /* a remoção não altera a altura da subarv.
*/
        p->bal--;
        if( p->bal == -2 )      /* mesmo balanceando a altura da subarv. muda
*/
          balanceia( &p );
      }
    }
    else {
      p->Dir = removeR( nodoRem, p->Dir, mudouH );
      if( *mudouH ){
        if( p->bal == 0 )
          *mudouH = FALSE;
        p->bal++;
        if( p->bal == 2 )
          balanceia( &p );
      }
    }
    return p;
}
```

Altura:

A altura de uma árvore AVL com n nodos é <= 1.44 log(n)

Árvores 2-3-4

Seção 13.3 (Sedgewick)
http://www.youtube.com/watch?v=bhKixY-cZHE *(Inserção em uma árv. 2-3)*
http://www.lcad.icmc.usp.br/~nonato/ED/



Definição:

Uma árvore 2-3-4 é uma árvore que está vazia ou que é composta por 3 tipos nodos:

-nodos-2: contem uma chave k1 e dois apontadores, p1 e p2. O apontador p1 aponta para uma árvore com valores de chave menores que k1 e o apontador p2 aponta para uma árvore com valores de chave maiores que k2.

-nodos-3: contem duas chaves k1, k2 e três apontadores, p1, p2 e p3. p1 aponta para uma árvore com valores de chave menores que k1, p2 aponta para uma árvore com valores de chave >k1 e <k2, e p3 aponta para uma árvore com valores de chave > k2

-nodos-4: contem três chaves k1, k2, k3 e quatro apontadores, p1, p2, p3 e p4. p1 aponta para uma árvore com valores de chave menores que k1, p2 aponta para uma árvore com valores de chave >k1 e <k2, p3 aponta para uma árvore com valores de chave >k2 e <k3, e p4 aponta para uma árvore com valores de chave >k3.

Definição:

Uma árvore 2-3-4 balanceada é uma árvore 2-3-4 na qual todos os nodos folha estão no mesmo nível.

Pesquisa:

- cada nodo possui no máximo 3 chaves (k[0],k[1],k[2]) e 4 apontadores (p[0],p[1],p[2],p[3]).
- cada nodo possui também o tamanho do nodo -- tam(n), que corresponde a quantidade de chaves no nodo n.

```
q = raiz( A );
if( q != NULL ){
  for( i=0; i< tam(n); i++)
    if( q.k[i] == chave )
      return q;
    else if( q.k[i] > chave )
      return busca( q.p[i], chave );
  return busca( q.p[tam(n)], chave );
}
```

Propriedade:

A busca de uma arv. 2-3-4 com n nodos visita no máximo log(n)+1 nodos .

Inserção:

A inserção de uma chave k pode ser feita da mesma forma que em uma árvore binária. Apos uma busca sem sucesso, inserir a chave na folha.
Problema: a árvore pode ficar desbalanceada

Exemplo:
```
        R
      /   \
     A     S   inserir C, H, I
```

Abordagem para balancear:

1. inserir sempre em um nodo folha. Caso o nodo contenha mais de 3 chaves, divide o nodo em dois e "sobe" com a chave do meio.

2. dividir os nodos com 3 chaves em todas as pesquisas realizadas na árvore.

Exemplo: inserir nodos A, S, E, R, C, H, I, N, G, X

```
typedef struct No *Apontador;
typedef struct No {
  Registro Reg[3];
  Apontador Ap[4];
  int NumReg;
} No;
```

- Insere(registro r, no corrente p)
- Percorre a árvore, dividindo nos-3 e inserindo o novo registro na folha

1. se árvore vazia

```
criaNo( r );
```

2. se p é um no-3, divide, jogando o registro do meio "para cima"

```
cria no com Reg[0]
    p1 = criaNo( Reg[0]);
    p1->Ap[0] = p->Ap[0];
    p1->Ap[1] = p->Ap[1];

cria no com Reg[2]
    p2 = criaNo( p->Reg[2]);
    p2->Ap[0] = p->Ap[2];
    p2->Ap[1] = p->Ap[3];

  se no dividido for a raiz
    novaRaiz = criaNo( Reg[1] )
    novaRaiz->Ap[0] = p1;
    novaRaiz->Ap[1] = p2;
  }
  senão insere Reg[1] no pai
    procura a posição pos de Reg[1] no pai
    desloca registros [pos, NumReg] em uma posição
    insere Reg[1] na posição pos
    pai->Ap[pos] = p1;
    pai->Ap[pos+1] = p2;
    pai->NumReg++;
```

3. continua percorrendo até achar uma folha da árvore

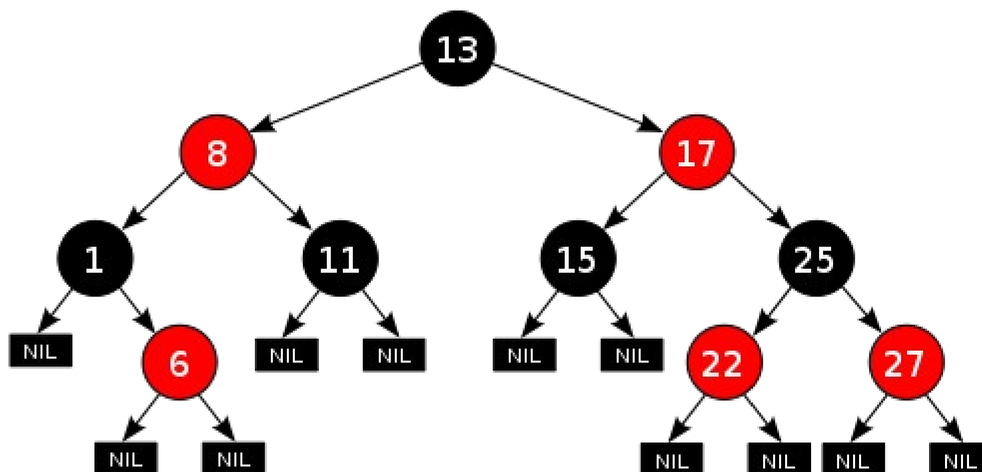
```
 se p não for folha
    para cada registro r em p
```

```
    se( r.Chave == p->Reg[i].Chave )   /* chave já existe */
      ERRO;
    senão se( p->Reg[i].Chave > r.Chave ){
      Insere( r, p->Ap[i] );

  Insere( r, p->Ap[q->NumReg])

se p for folha
  procura posição pos em p para inserir r
  desloca registros nas posições [pos, NumReg]
  insere registro r na posição r
 q->NumReg++;
```

Árvore Rubro-Negra

Cap. 14 de Cormen&Leiserson&Rivest

http://www.youtube.com/watch?v=DVpLMeGG-Qs (INGLÊS, definição, inserção, prova da altura)
http://www.ic.unicamp.br/~zanoni/mo637/aulas/arvoresRubroNegras.pdf



Definição:

Em uma arv. Rubro-negra:

1. todo nodo ou é preto ou é vermelho
2. a raiz é preta.
3. todo nodo externo (NIL) é preto
4. se um nodo é vermelho seus dois filhos são pretos (não podem existir dois nodos vermelhos consecutivos em um caminho)
5. todo caminho de um nodo até um nodo externo contem o mesmo número de nodos pretos

Coloração:

A cada iteração as propriedade da arv. RN são violadas se:

1. a raiz for vermelha, neste caso é só pintar a raiz de preto (ultima linha do código); ou
2. porque há dois nodos vermelhos consecutivos (a altura preta não é alterada porque o novo nodo é sempre pintado de vermelho) neste caso, "jogar o problema para cima", mantendo como invariante a altura (de nodos pretos) nas subárvores.

- o "problema" continua enquanto o nodo corrente for vermelho

Caso 1: o tio é vermelho: neste caso, "descer" a cor preta para as duas sub-árvores: pintar o tio e o pai do nodo corrente de preto, e para manter a invariante, pintar o avo de vermelho.
Caso 3: (esq-esq): o tio é preto e o nodo corrente é filho esquerdo do pai, que é filho esquerdo do avô: pinta o avô de vermelho, o pai de preto e faz a rotação a direita
Caso 2: (esq-dir): o tio é preto e o nodo corrente é filho direito do pai, que é filho esquerdo do avô: transforma no Caso 2, fazendo uma rotação a esquerda do pai do nodo corrente.

- o algoritmo assume que a raiz é sempre preta, de forma que se pai(b) é vermelho, pai(pai(n)) sempre existe.
- observe que em todos os casos pai(pai(n)) é preto, já que pai(n) é vermelho

Inserção:

Considere a seguinte estrutura de dados para a estrutura do no da árvore, sendo que:
- a cor do NodoNulo é BLACK

- pai da raiz de uma árvore não vazia é o nodoNulo
- todos os nodos externos também são representados pelo mesmo nodoNulo.

```
enum tipoCor {RED, BLACK};

typedef struct no{
  Apontador esq, dir;
  Apontador pai;
  longInt chave;
  tipoCor cor;
} No;

RN-insere(raiz, k){
  novoNodo = criaNodo( k )
  x = raiz;
  paiX = nodoNulo;
  enquanto x <> nodoNulo
    paiX = x;
    se k < chave(x)
       x = esq(x)
    senao x = dir(x)

  pai(novoNodo) = paiX
  se paiX = nodoNulo
    raiz = novoNodo
  senao se k < chave(paiX)
          esq(paiX) = novoNodo
        senao
          dir(paiX) = novoNodo
  cor(novoNodo) = RED
  arrumaArvRN( raiz, novoNodo )
}

arrumaArvRN( raiz, p ){
  enquanto cor(pai(p)) = RED
    se pai(p) == esq(pai(pai(p))){  /* insercao na subarv.esq */
      tio = dir(pai(pai(p)))
      se cor(tio)== RED{            /* cor a dir. do avo é  vermelho */
        cor(pai(p)) = BLACK         /* Caso 1 */
        cor(tio) = BLACK
        cor(pai(pai(p)) = RED
        p = pai(pai(p))
      }
      senao{                        /* cor a dir. do avo é preto */
        se p == dir(pai(p)){         /* desbal. na subarv.dir do filho esq
*/
          p = pai(p)                /* Caso 2: esq-dir */
          rotEsq(p)
        }
        cor(pai(p)) = BLACK         /* Caso 3  esq-esq*/
        cor(pai(pai(p))) = RED
        rotDir( pai(pai(p) )
    }
    senao{
              /* insercao na subarv. direita -- idem trocando dir <-> esq
*/
    }
  cor(raiz) = BLACK
}
```

Custo e Altura:

insereArvBin - *O(log n)* → while é executado no Caso 1 no maximo log(n) vezes, portanto, custo total O(log n)
Altura → *2lg(n + 1)*

Remoção em Arv. Rubro-Negra:

Lema:

A altura de uma arv. RN com n nodos tem altura de no máximo 2log(n+1).

Prova:

Seja x um nodo. Representamos por hp(x) a altura "preta" de x; ou seja, a quantidade de nodos pretos a partir de x (sem incluir x) até um nodo externo.
Primeiro mostramos que a quantidade de nodos internos em uma subárvore de x é no minimo 2^{hp(x)} - 1 por indução na altura(h) de x.

Se h=0, x é um nodo externo e bh(x)=0: 2^{0} - 1 = 0.

Se h>0 , x não é um nodo externo e tem 2 filhos com alturas menores que x.

As alturas pretas dos filhos de x podem ser hp(x)-1 ou hp(x), dependendo do filho ser um nodo preto ou vermelho, respectivamente. Como a altura das subarv. é menor que a altura de x, pela hipotese da indução a quantidade de nodos internos da subarv. com raiz em x é pelo menos

(2^{hp(x)-1} - 1) + (2^{hp(x)-1} - 1) + 1

ou seja, pelo menos

2*2^{hp(x)} - 2 + 1 = 2^{hp(x)} - 1.

Para terminar a prova, observe que em uma arv. RN de altura h pelo menos metade dos nodos em um caminho da raiz até um nodo externo são pretos. Assim

hp(x) >= h/2 e n >= 2^{h/2} -1.

Movendo 1 e aplicando log_2 em ambos os lados temos log(n+1) >= log(2^{h/2}) ou seja

h <= 2log(n+1).

Consequência:

busca, inserção, remoção em arv. RN é O(log(n))

Remoção de nodos pretos causam desbalanceamento de todos os seus ancestrais ==> um dos nodos pretos vira um "duplo preto"

Correção:

Tentar "jogar" o desbalanceamento para cima até que:

- -seja encontrado um nodo vermelho
- -encontre a raiz
- -possa executar rotações e mudanças de cor que restaurem o balanceamento

```
remove-RN( raiz, nodoK ){ /* nodoK é o nodo que tem a chave K a ser
removida */
  se esq(nodoK) == nodoNulo ou dir(nodoK) == nodoNulo
    nodoRem = nodoK               /* se nodoK tem 0 ou 1 filho, remove
nodoK */
  senão                           /* senão remove o sucessor */
    nodoRem = sucessor( nodoK )   /* neste caso o nodoRem não tem filho
esq */
  se esq( nodoRem ) <> nodoNulo
    filho = esq(nodoRem)
  senão
    filho = dir(nodoRem)
  pai(filho) = pai(nodoRem)
  se pai(nodoRem) == nodoNulo
```

```
     raiz = filho
   senão se nodoRem == esq(pai(nodoRem))
            esq(pai(nodoRem)) = filho
         senão
            dir(pai(nodoRem)) = filho
   se nodoK <> nodoRem
      /* copia chave e dados do nodoRem para nodoK */
   se cor(nodoRem) == BLACK
     arrumaRem-RN( raiz, filho )
}

arrumaRem-RN( raiz,  p ){
   enquanto p <> raiz e cor(p) == BLACK
     se n == esquerda(pai(p)){   /* extra BLACK a esquerda */
        d = direita(pai(p))

        se cor(d) == RED{        /* Caso 1 */
          cor(d) = BLACK
          cor(pai(p)) = RED
          rotaçãoEsq( pai(p) )
          d = direita(pai(p))
        }
        se cor(esquerda(d)) == BLACK e cor(direita(d)) == BLACK{
          cor(d) = RED     /* Caso 2 */
          p = pai(p)
        }
        senão{
            se esquerda(d)->cor == RED  /* direita(d)->cor == RED */
               cor(d) = RED             /* Caso 3 */
               cor(esquerda(d)) = BLACK
               rotaçãoDir( d )
               d = direita( pai(p) )
            }
            cor(d) = cor(pai(p))        /* Caso 4 */
            pai(p)->cor = BLACK
            cor(direita(d)) = BLACK
            rotaçãoEsq ( pai(p) )
            p = raiz;
     }
     senão{               /* extra BLACK a direita -- similar */
     }
   cor(p) = BLACK
}
```

Caso 1:
Se o irmão de p for vermelho, seus dois filhos são pretos. O objetivo do Caso 1 é transformar o irmão de p em preto (Casos 2, 3 ou 4). Para isso, troca-se a cor do irmão para preto, o pai para vermelho e faz uma rotação. Como os filhos do irmão são pretos, o novo irmão de p será preto.
Na árvore 2-3-4, este caso equivale ao pai de p ser um nodo do tipo-3. O Caso 1 corresponde a transformar uma representação deste tipo de nodo, com a chave de maior valor como raiz da subárvore na representação deste tipo de nodo na RB em outra representação com a chave de menor valor como raiz da subárvore; e vice-versa.

Caso 2:
Se o irmão direito é preto e seus dois filhos são pretos troca a cor do irmão par vermelho, isso já compensa o preto extra da subarv. esquerda.
Na arv. 2-3-4 este caso corresponde ao "merge" de nodos do tipo-2; ou seja, seu irmão é um nodo do tipo-2 e não tem chaves para emprestar.

Caso 4:
Se o irmão direito d é preto. Se o filho direito de d é vermelho transforma ele em preto para compensar o preto extra. Troca cores. Faz rotação a esquerda em pai(n)

Na arv. 2-3-4 este caso corresponde ao empréstimo de uma chave do irmão.

Caso 3:
Se o irmão direito d é preto. Se o filho esquerdo de d é vermelho, faz uma rotação para transformar este caso no Caso 4:

Troca as cores de w e esquerdo(w)
Rotação a direita de w
w = direita(pai(n))

Na arv. 2-3-4 este caso corresponde ao empréstimo de uma chave do irmão.
porém, o irmão na RB não está na representação dir-dir ou esq-esq. Portanto, é necessário fazer uma rotação para transforma-la no Caso 4.

Árvores B

Cap. 6 (Nivio) - Cap. 18 (Cormen)

http://www.youtube.com/watch?v=qXfPA6xqVlQ (definição)
http://www.youtube.com/watch?v=ANZBJw3a944 *(inserção, remoção) (Atenção: a definição de grau é diferente da que apresentada pela Carmem)*
http://www.lcad.icmc.usp.br/~nonato/ED/B_arvore/btree.htm (segue a definição de grau da Carmem)

Motivação:

- Indexação em memoria secundaria
- Generalização de uma árvore 2-3-4

Definição:

É uma árvore n-aria. Em uma árvore B com grau minimo m temos que:

1. cada nodo contem no **minimo *m-1*** chaves (e *m* filhos - *grau >= m*) e no **máximo *2m-1*** chaves*(2m filhos - grau <= 2m)*, **exceto o nodo raiz**, que pode conter entre *1* e *2m-1* chaves
2. todas os nodos folha aparecem no mesmo nível.

Altura:

h <= log_t((n+1)/2).

Implementação:

- cada nodo x mantem:
  - num[x] : quantidade de chaves em x
  - chave_1[x],... chave_{num[x]}[x] chaves em ordem não decrescente
  - folha[x]: verdadeiro se x é folha e falso, caso contrario
  - p_1[x],...p_{num[x]+1}[x] apontadores para os filhos

```
const t = 2;
typedef struct no_arvoreB arvoreB;

struct no_arvoreB {
   int num_chaves;
   char chaves[2*t-1];
   arvoreB *filhos[2*t];
   bool folha;
};
```

Busca:

- parecido com uma árvore binária, porém com nodos contendo entre *t-1* e *2t-1* chaves
- resultado: um par (x,i), onde x é o endereço do nodo que contem a chave procurada k e i é a posição da chave dentre do nodo

```
Busca (x, k)
  i = 0
  enquanto (i <= num[x] e k > chave_i[x])
    i = i+1
  se i <= num[x] e k = chave_i[x]
    retorna (x, i)
  se folha[x]
    retorna NIL
  else
    le_disco(p_i[x])
```

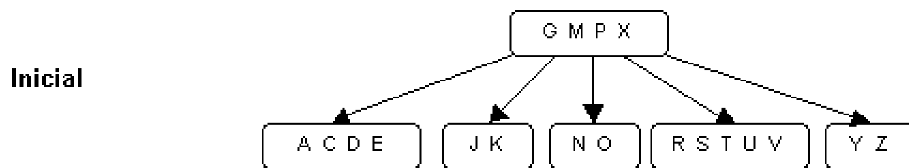
```
                retorna Busca(p_i[x], k)
```

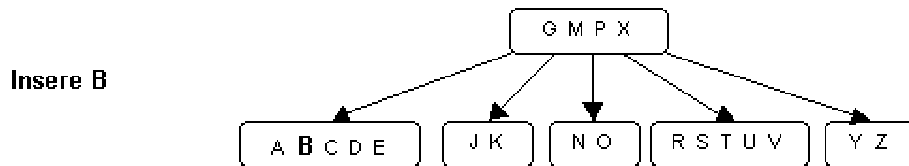
Número de acessos a disco: no pior caso *log_t(n)*

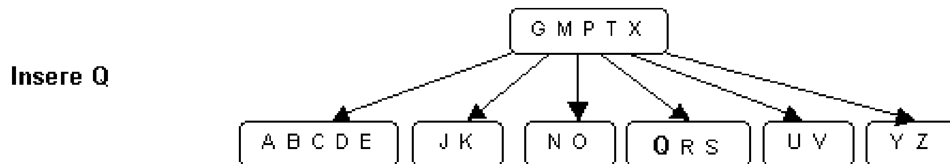
Tempo de CPU: *O(t log_t(n))*

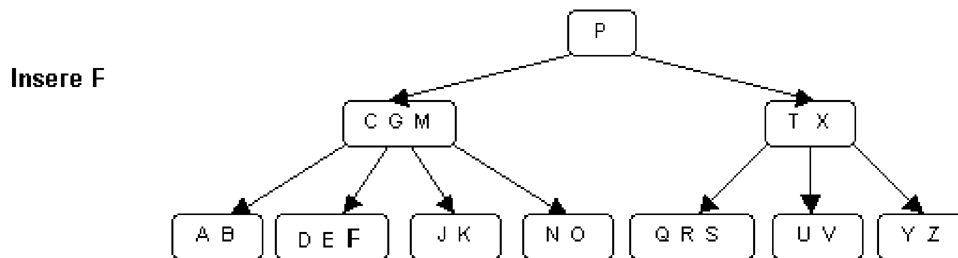
Inserção:

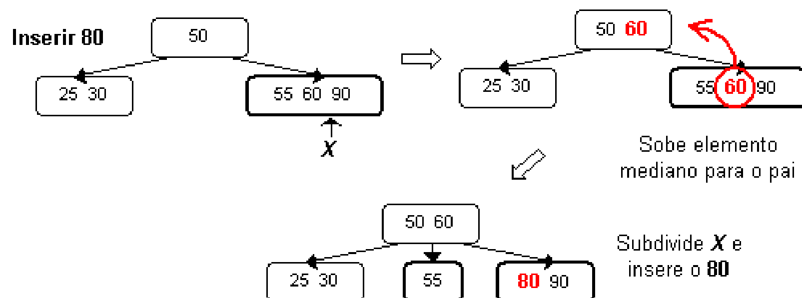
- a ideia é a mesma da arv. 2-3-4: a medida que desce na árvore, nodos cheios são divididos para que seja sempre possível inserir novas chaves em seus filhos (com a possibilidade de uma chave "subir")












```
Inicializa( T )
  x = alocaNodo()
  folha[x] = TRUE
  num[x] = 0
  escreve_disco( x )
  raiz[T] = x
```

Acessos a disco e tempo CPU: O(1)

```
DivideNodo( pai, ind, f )
 /* Divide nodo f que é filho de pai não cheio, sendo f o filho
"ind"
 de pai */
  z = alocaNodo()
  folha[z] = folha[f]
  num[z] = t-1
  para j=1 a t-1
    chave_j[z]= chave_{j+t}[f]
  se f não é folha
    para j=1 a t
```

```
            p_j[z] = p_{j+t}[f]
       num[f] = t-1
       para j = num[pai]+1 downto ind+1
         p_{j+1}[pai] = p_{j}[pai]
       p_{ind+1}[pai] = z
       para j = num[pai] downto ind
         chave_{j+1}[pai] = chave_j[pai]
       chave_ind[pai] = chave_t[f]
       num[pai] = num[pai]+1
       escreve_disco(f)
       escreve_disco(p)
       escreve_disco(z)
```

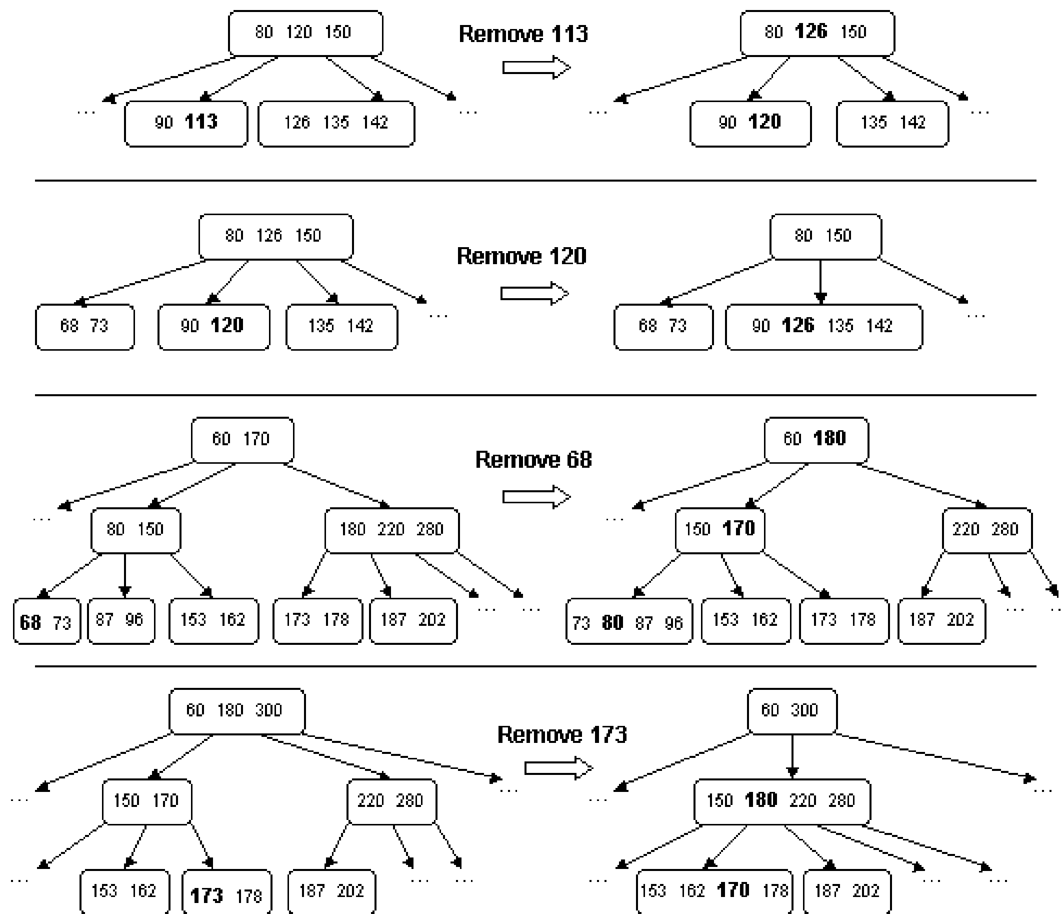
Acessos a Disco: O(1)
Tempo de CPU: O(t)

```
     insereArvB( raiz, k )
       r = raiz
       se num[r] = 2t-1  /* raiz cheia */
          z = alocaNodo()
          raiz = z
          folha[z] = false
          num[z] = 0
          p_1[1] = raiz
          divideNodo( z, 1, r )
          insereNodonãoCheio( z, k )
        senão
          insereNodonãoCheio( r, k )

     insereNodonãoCheio( x, k )  /* insere chave k no nodo x */
       i = num[x]
       se folha[x]
          enquanto i >=1 e k < chave_i[x]
             chave_{i+1}[x] = chave_i[x]
             i = i - 1
          chave_{i+1}[x] = k
          num[x] = num[x] + 1
          escreve_disco(x)
       senão
          enquanto i >= 1 e k < chave_i[x]
            i = i - 1
          i = i+1
          z = le_disco( p_i[x] )
          se num[z] = 2t-1
            divideNodo( x, i, z )
            se k > chave_i[x]
              i = i + 1
          inserenãoCheio( p_i[x], k )
```

Acessos a Disco: O(log_t(n))
Tempo de CPU: O(t log_t(n))

Remoção em Árvores B:

- problema que pode acontecer: um nodo pode ficar com menos que t-1 chaves
- solução: a medida que faz a busca pelo nodo que contem a chave, garantir que todos os nodos no caminho tenham pelo menos t chaves (1 a mais que o minimo). Isso é garantido da seguinte forma: se o nodo a ser visitado tem t-1 chaves




```
buscaEmNodo( x, k )
  i = 1
  enquanto( i <= num(x) e k < chave[i](x) )
    i = i+1;
  retorna i

removeNoNodo( x, indK )
  para j = indK até num(x)-1
    chave[j](x) = chave[j+1](x)
  para j= indK+1 até num(x)
    p[j](x) = p[j+1](x)
  num(x) = num(x) - 1

remove( Apontador raiz, Apontador x, Chave k )
  indK = buscaEmNodo( x, k )    /* busca ind. de k ou de chave > k
*/
  esq = p[indK](x)
  se indK <= num(x)
     dir = p[indK+1](x)
  senão
     dir = NULL

  se indK <= num(x) e chave[indK](x) == k   /* k é uma das chaves de
x */
     se folha(x)                            /* remove k em nodo
folha */
       removeNoNodo( x, indK )
     else                       /* remove k em nodo não folha */
       se (num(esq)) >=  t      /* filho esq. tem pelo menos t
chaves*/
         predK = predecessor( x, k );
```

```
         remove( raiz, x, predK );
         chave[indK](x) = predK;

       senão se dir <> NULL and (num(dir) >= t
                                /* filho dir. tem pelo menos t
chaves*/
         succK = sucessor( x, k )
         remove( raiz, x, succK );
         chave[indK](x) = succK

       senão                           /* dois filhos tem t-1 chaves
*/
         p = merge( x, indK )
         se x == raiz
            free( raiz )
            raiz = p
         remove( raiz, p, k )

  senão                         /* k não está em x */
     se folha(x)
        retorna ERRO
     senão                      /* continua descendo na arv. */
       se num(esq) >= t
           remove( raiz, esq, k )
       senão
         se dir <> NULL and num(dir) >= t
                                /* empresta chave do irmao direito
*/
            ...
            remove( raiz, esq, k )
         senão se indK > 1 e num(p[indK-1](x)) >= t
                                /* empresta do irmao esq */
            ...
            remove( raiz, esq, k )
         senão
           se dir == NULL
             p = merge(x, indK-1)
           senão
             p = merge(x, indK)
           se x == raiz
             free( raiz )
             raiz = p
           remove( raiz, p, k )

merge( Apontador x, índice indK )
  esq = p[indK](x)
  dir = p[indK+1](x)
  num(esq) = num(esq) + 1       /* junta (esq + k + dir)  em esq */
  chave[num(esq)](esq) = chave[indK](x);
  para i = 1 até num(dir)
     chave[num(esq)+i](esq) = chave[i](dir)
     p[num(esq)+i](esq) = p[i](dir)
  num(esq) = num(esq) + num(dir)
  p[num(esq)+1](esq) = p[num(dir)+1](dir)
  free( dir );

  para i = indK até num(x)-1     /* arruma chaves e apont. em x */
    chave[i](x) = chave[i+1](x)
  para i = indK+1 até num(x)
     p[i](x) = p[i+1](x)
  num(x) = num(x) - 1
  retorna esq;
```

Memória Secundária:

- conceito de prato, trilha, cilindro, cabeçote de leitura e gravação
- acesso é muito mais lento em mem. secundaria que na mem. primaria devido aos componente mecânicos (rotação do disco e cabeçote)
- rotação: +- 7200 RPM --> 8.33 milissegundos para uma rotação
- acesso a memoria em silício: +- 100 nanosegundos (5x mais rápido, dá para fazer 100.000 acessos à memoria e durante a espera por 1 rotação do disco)
- este tempo não é constante porque da localização do dado na trilha e o posicionamento do cabeçote
- forma de amortização: transferência em paginas ao invés de itens individuais. Paginas em geral tem tamanho em múltiplo de 512 (2^9)
- capacidade de mem. secundaria é em geral bem maior que a capacidade de mem. principal --> memoria virtual
- memoria virtual: baseada em uma função entre o espaço de endereçamento N (utilizado por um programa) e o espaço de memoria M f: N -> M
- a memoria principal é dividida em "Moldura de paginas", na qual cada moldura contem exatamente uma pagina
- mecanismo de paginação:
    1. determina qual a pagina que um programa está endereçando. O endereço é dividido da seguinte forma: se o espaço de endereçamento possui 24 bits, então a memoria virtual tem tamanho 2^24. Se o tamanho da pagina é 512 (2^9) então nove bits são utilizados para endereçar o byte dentro de uma pagina e o restante para representar o número da pagina.
    2. carrega a pagina na moldura de paginas. Precisa de uma tabela de paginas (que mapeia o número da pagina à moldura na qual ela está carregada e também uma politica para reposição de paginas (LRU-Least Recently Used, LFU-Least, Frequently Used, FIFO-First In First Out)

- o tamanho do nodo coincide com o tamanho da pagina
- quantidade de chave tipicamente está entre 50 a 2000

Árvore B+

Cap. 6, 6.3 (Nivio)

Ideia:

Separar nodos de índice de nodos de dados
- nodos internos contem apenas índices
- todos os registros nas folhas (sem a necessidade de apontadores)
- os nodos folha são encadeados (para facilitar a busca ordenada de valores)
- pode ter um grau distinto para nodos índice e folha

Objetivo:

- acesso sequencial mais eficiente
- facilitar o acesso concorrente as dados

Exemplos:
-inserção
-remoção
-busca

Acesso Concorrente:

- podem ocorrer problemas se um processo estiver lendo a estrutura e outro inserindo uma nova chave que causa divisão de um nodo
- uma pagina é segura se não houver possibilidade de mudança na estrutura da árvore devido a inserção ou remoção na pagina

    inserção: pagina é segura se #chaves < 2t-1
    remoção: pagina é segura de #chave > t-1

Protocolo de bloqueio:

lock-R : bloqueio para leitura
lock-W: bloqueio exclusivo para escrita

Leitura:

1. lock-R(raiz)
2. read(raiz) e torne-a pagina corrente
3. enquanto pag. corrente não for folha
    lock-R(descendente)
    unlock(pag corrente)
    read(descendente) e torne-a pagina corrente

Atualização:

1. lock-W(raiz)
2. read(raiz) e torne-a pagina corrente
3. enquanto pag. corrente não for folha
    lock-W(descendente)
    read(descedente) e torne-a paginal corrente
    Se pag. corrente for segura
        unlock(p) para todos os nodos p antecedentes que tenham sido bloqueados

Método de Acesso Sequencial Indexado (ISAM)

- parecido com o árvore B+, mas utiliza paginas de overflow
- há uma previsão inicial da quantidade de registros do arquivo, deixando cerca de 20% das paginas inicialmente livres
- vantagem: não ha' necessidade de bloqueio nas paginas de índice
- desvantagem: pode haver um "desequilíbrio" da quantidade de registros em cada intervalo

Heap

http://www.youtube.com/watch?v=QdRL3XLyiVc *(definição, Heapsort, voz do Google)*
http://www.youtube.com/watch?v=_9QXNFcrF4c *(INGLÊS, propriedades, árv. → array, parte 1)*
http://www.youtube.com/watch?v=DHhPg01rBGs *(INGLÊS, adição, remoção, parte 2)*
http://www.youtube.com/watch?v=8xJU2TZksWw *(INGLÊS, propriedades, construção, Heapfy)*

É a representação *em forma de vetor* de uma árvore binária em ordem-heap.
- Lembrando: é uma árvore binária quase completa de altura d

Arv. binária na qual:
1. todas as folhas estão no nível d ou d-1
2. se um nó nd na árvore tem algum descendente direito no nível d (o máximo da árvore), então todos os descendentes esquerdos de nd que forem folhas estão também no nível d.

Numeração dos Nodos (Índice do Vetor):

num(raiz) = 1
num(n) = *2 * num(np)* se n é filho esquerdo de np
*2 * num(np) + 1* se n é filho direito de np

Assim, dado um heap A e um índice i:

```
pai(i) = chão(i/2)
esq(i) = i*2
dir(i) = i*2+1
```

tamHeap(A) : índice do maior elemento em A que está preenchido com elementos do heap
tam(A) : tamanho do vetor

- arv. estritamente binária quase completa com n folhas tem 2n-1 nós
- arv. binária quase completa (que não seja estritamente binária) tem 2n nós
- altura de uma arv.bin quase completa de n nos = *floor (log_2 (n))*

Propriedade da Heap:

max-heap: A[pai(i)] >= A[i] ==> o maior elemento é a raiz
min-heap: A[pai(i)] <= A[i] ==> o menor elemento é a raiz

Funções:

```
arrumaHeapDown (A, i): supoe que as subarv. dir e esq do elemento i
já satisfazem a propriedade max-heap, mas A[i] pode ser menor que
seus filhos (viola max-heap).
  {

    e = esq(i);  d = dir(i);
    se e < tamHeap(A) e e < d
      maior = d
    senão
      maior = e
    se A[maior] > A[i]
      troca(A[i], A[maior])
      arrumaHeap( maior )
   }
  }
```

Custo: *O(log(n))*

```
arrumaHeapUp (A, i):
  {
    enquanto i > 1 e A[i/2] < A[i]
      troca(A[i], A[i/2])
      i = i/2
   }
  }


constróiMaxHeap( A ):
  {  tamHeap( A ) = tam(A);
     para i = tamHeap(A) / 2  ate 1 em ordem decrescente
       arrumaHeapDown( A, i );
  }
```

Observação: todos os elemento com índice maior que floor(tamHeap(A)) são folhas. Portanto a iteração só precisa tratar os elementos armazenados nos índices menores que este (nodos internos).

Custo: *n/2 * log(n) = O(n log(n))*

Heapsort:

```
heapSort (A)
   { constróiMaxHeap( A );
     para  i = tam( A ) até 2 {
       troca( A[1],  A[i];
       tamHeap( A ) = tamHeap( A ) - 1;
       arrumaHeapDown( A , 1 )
     }
   }
```

Custo: *constróiHeap: O(n log(n)) + n * arrumaHeap : n * O(log (n))*
Total: *O(n log(n))*

Embora o custo do heapsort seja o mesmo do quicksort, na pratica o quicksort em geral é mais rapido. Mas uma das aplicações para o Heap é a implementação de uma lista de prioridades.

Lista de Propriedades :

(pode ser lista-max ou lista-min, aqui é considerado lista-max)

É uma estrutura para manter um conjunto de elementos S, cada um com um valor associado, chamado de chave. Deve prover as seguintes funções:

- insere(S, x): insere o elemento x em S
- máximo(S): retorna o maior elemento em S
- extraiMax(S): remove e retorna o maior elemento de S

Exemplo de aplicação:
- Escalonamento de processos:
  Huffman (lista-min)

Implementação:

(representando o conjunto S em um Heap A)

```
máximo( A ){
     retorna A[1]
   }
```

Custo: *O(1)*

```
extraiMax( A ){
```

```
    se tamHeap( A ) < 1 retorna erro;
    max = A[1]
    troca( A[1], A[tamHeap(A)]
    tamHeap(A) = tamHeap(A) - 1
    arrumaHeapDown( A, 1 );
    retorna max
  }
```

Custo: *arrumaHeap = O(log(n))*

```
insere( A, k ){
     tamHeap(A) = tamHeap(A) + 1;
     A[i] = k;
     arrumaHeapUp( A, tamHeap(A));
```

Custo: *O(log(n))*

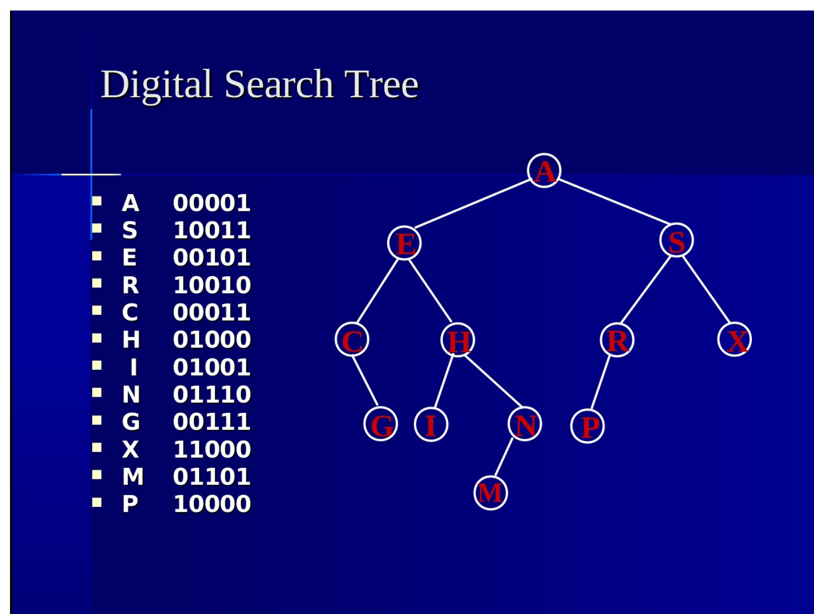
Árvores de Pesquisa Digitais

Cap. 15 (Sedgewick)

Para aplicações nas quais a busca é feita em apenas uma parte da chave. Ou seja, quando a chave pode ser decomposta em pedações de tamanho fixo (bytes ou bits).
É necessário haver uma operação eficiente para obter a i-ésima parte da chave

Vantagens:
- custo do pior caso razoável, sem necessidade de balanceamento
- permite chaves de tamanho variável

Árvore de Pesquisa Digital Binária:



A arv. digital tem custo quase ótimo para aplicações com grande volume de chaves, com fácil implementação (+ fácil que avl, rubro-negra). O desempenho é muito bom desde que exista uma operação eficiente de acesso aos bits que compõe a chave.

- as chaves são representadas por sequência de bits
- cada nó pode ter dois filhos
- os bits são analisados do mais significativo para o menos significativo. Se for igual a zero, a chave é armazenada no filho esquerdo, caso contrario, no filho direito
- a árvore não mantem as chaves em ordem. A ordem somente é garantida para chaves no mesmo nível.
- a característica da árvore é que uma chave está armazenada em algum nodo no caminho determinado pela sua sequencia de bits
- considerando chaves de tamanho fixo com w bits (e sem repetição de chaves), a quantidade de chaves N a ser inserida na árvore é <= 2^w.
- a árvore digital é apropriada se a quantidade de chaves for significativamente menor que 2^w. Caso contrario, uma árvore de pesquisa AVL ou rubro-negra seria mais apropriada.
-para chaves de 32 bits, a arv. digital seria apropriada se o número de chaves for no máximo 100.000, e para chaves de 64 bits, a arv. digital pode ser apropriada para qualquer quantidade de chaves
-o tempo de busca é limitado pelo tamanho da chave
-o caminho mais longo tende a ser pequeno em diversas aplicações.

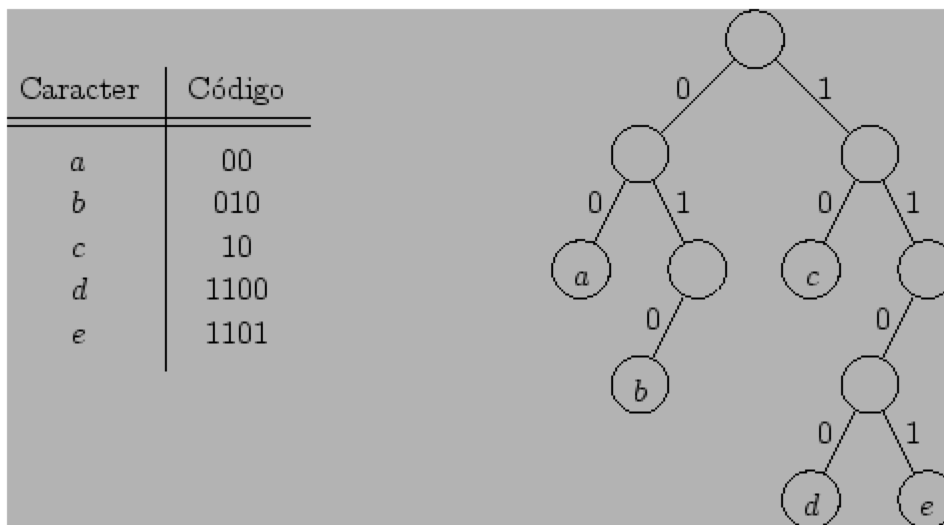
Implementação:

Estrutura de Dados: (Idêntica a árvore binária)

```
typedef long TipoChave;
typedef struct Registro {
  TipoChave Chave;
  /* outros componentes */
} Registro;
typedef struct No *Apontador;
typedef struct No {
  Registro Reg;
  Apontador Esq, Dir;
} No;
```

```
busca:  busca(raiz, chave, 0)  /* nodo, valor, ordem do bit */

busca( p, k, w )
  kNodo = p->reg.Chave;
  se( p == NULL) retorna NULL;
  se( kNodo == k ) retorna p->reg;
  se( digito(k, w) == 0 )
    retorna busca( esquerda(p), chave, w+1 )
  senão
    retorna busca( direita(p), chave, w+1 )
```

Trie



Similar as arv. de busca digitais, mas mantem as chaves em ordem e armazena chaves somente nas folhas.

Definição:

Uma trie é uma árvore binária que possui chaves associadas aos nodos folhas e definida recursivamente da seguinte forma:

a) a trie para um conjunto vazio de chaves é apenas um apontador NULL;
b) a trie para apenas uma chave é composta apenas por um nodo folha que contem esta chave
c) a trie para um conjunto de chaves maior que um é composta por um nodo interno, sendo o filho esquerdo uma trie contendo chaves cujo bit inicial é 0 e o filho direito uma trie contendo chaves cujo bit inicial é 1. O primeiro bit é então removido para a construção das subárvores direita e esquerda.

Característica:

Existe uma única trie para um determinado conjunto de chaves. Ou seja, a estrutura da árvore independe da ordem de inserção.

Implementação:

A estrutura pode ser igual a arv. binária, mas pode ser melhorado para que os nodos internos contenham somente apontadores, e as folhas apenas chaves.

```
busca(raiz, chave, 0)
-----
busca( p, k, w )
  se p == nodoNulo retorna nodoNulo;
  se p->esq == nodoNulo e p->dir == nodoNulo
    se p->reg.Chave == k
       retorna p->reg
     senão retorna nodoNulo;
  se digito(k, w) == 0
     retorna busca( p->esq), chave, w+1)
  senão
     retorna busca( p->dir, chave, w+1)

inicializa()
  return criaNodo( chaveNula )
```

```
chamada: insert(raiz, chave, 0)

insert(p, k, w)
  kNodo = p->reg.Chave;
  se p == nodoNulo retorna criaNodo( k );
  se p é folha então retorna split( criaNodo(k), p, w );
  se digito(k, w) == 0
    p->esq = insert( p->esq, chave, w+1 )
  senão
    p->dir = insert( p->dir, chave, w+1 )
  retorna p;

split( p1, p2, w)
  n = criaNodo( itemNulo );
  d1 = digito(p1, w);
  d2 = digito(p2, w);
  se d1 == d2
    se d1 == 0
      n->esq = split( p1, p2, w+1 )
    senão
      n->dir = split( p1, p2, w+1 )
  senão
    se d1 == 0
      n->esq = p1;  n->dir = p2;
    senão
      n->esq = p2;  n->dir = p1;
  retorna n;
```

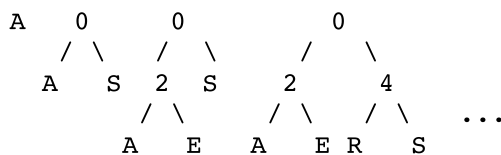
Árvore Patricia (Practical Algorithm to Retrieve Information coded in Alphanumeric)

Seção 15.3 (Sedgewick) – Seção 5.4.2 (Nivio)

Características:

- ao contrario das TRIES, não requer a criação de múltiplos nodos quando as chaves diferem apenas nos bits no final da chave.
- implementação com um único tipo de nodo (nas tries os nodos internos e folhas possuem estruturas distintas, já que as chaves estão armazenadas somente nas folhas)
- similarmente as árvores digitais, uma arv. Patricia para armazenar n chaves contem exatamente n nodos
- requer em media log(n) comparações de bits por busca e apenas uma busca da chave como um todo
- não depende do tamanho da chave como as tries e pode ser usada para chaves de tamanho variável

Exemplo da árvore Patricia "simplificada" (de acordo com a definição no livro do Ziviani):



Ideia:

- nodos armazenam qual o bit que o diferencia do pai
- nodos armazenam uma chave, da mesma forma que nas árvores digitais.

Os nodos externos, ao invés de serem somente NULL, podem apontar para o nodo na árvore que contem a chave com o prefixo determinado pelo caminho da raiz até o nodo externo.

Definição:

Uma árvore Patricia é uma árvore binária na qual cada nodo N possui uma chave e um índice de bit k, cujo valor é definido da seguinte forma:

- k é o primeiro bit no qual a chave difere da chave do seu pai. Seja kp o índice bit do pai. Se o bit kp da chave é igual a 0 então N é o filho esquerdo do pai; caso contrario, ele é o filho direito.

Um nodo externo pode corresponder a NULL ou ao endereço de um nodo A, que é igual a N ou um ancestral de N. Sejam n_1,...,n_q os nodos da raiz até A com índices bit b_1,...,b_q, respectivamente. A chave de A é a unica da árvore que satisfaz a seguinte condição: Para todo i em [2,q]:

- se n_i é filho esquerdo de n_{i-1} então o bit b_{i-1} da chave é igual a 0;
- se n_i é filho direito de n_{i-1} então o bit b_{i-1} da chave é igual a 1.

Exemplo:

```
bits de 0 a 4
A 00001
S 10011
E 00101
R 10010
C 00011
H 01000
I 01001
N 01110
```

```
G 00111
X 11000
M 01101
P 10000
L 01100
```

Implementação:

Estrutura de dados:

```
typedef long TipoChave;
typedef struct Registro {
  TipoChave chave;
  /* outros componentes */
} Registro;
typedef struct No *Apontador;
typedef struct No {
  Registro reg;
  Int bit;
  Apontador esq, dir;
} No;
```

Obs: para simplificar a implementação, a raiz da árvore é sempre um nodo R com "chave nula" (todos os bits iguais a 0, um valor não utilizado como valor de chave) e campo bit igual -1.
Caso a árvore contenha pelo menos uma chave, R->esq aponta para um nodo que contem uma chave.

Inicialização:

```
raiz = criaNodo( itemNulo, -1)
raiz->esq = raiz;
raiz->dir = raiz;
```

Busca:

Chamada: busca(raiz, chave)
Retorna endereço do registro com a chave ou nodoNulo

```
busca(raiz, k)
  p = buscaR( raiz->esq, k, -1 );
  se p->reg.Chave == k retorna p
  senão retorna nodoNulo

buscaR( p, k, bit )
  se p->bit <= bit retorna p;
  se digito( k, p->bit) == 0
     retorna buscaR( p->esq, k, p->bit )
  senão
     retorna buscaR( p->dir, k, p->bit )
```

Inserção:

Chamada: insere(raiz, reg)

```
insere( raiz, reg )
  k = reg.chave
  p = buscaR( raiz->esq, k, -1 )
  pk = p->chave
  se k == pk retorna;   /* chave já está na árvore */
  i = 0;                /* procura bit que diferencia k da chave de
p */
  enquanto digito(k, i)==digito(pk, i)
```

```
    i++;
  raiz->esq = insereR (raiz->esq, reg, i, raiz)

insereR( p, reg, bit, paiP )
  se p->bit >= bit ou p->bit <= paiP->bit
      n = criaNodo( reg, bit )
      se digito(reg.chave, bit) == 0
         n->esq = n;  n->dir = p;
      senão
         n->esq = p;  n->dir = n;
      retorna n
  se digito(reg.chave, bit) == 0
     p->esq = insereR( p->esq, reg, p->bit, p )
  senão
     p->dir = insereR( p->dir, reg, p->bit, p )
  retorna p
```

Obs: a inserção de um nodo com bit menor do que um já existente corresponde a inserção na trie no lugar de um filho "nulo" de um nodo interno que não foi criado na arv. Patricia. Se o bit for igual ao de um nodo existente, esta inserção corresponde na inserção na trie com "split" de uma folha.

Característica:

Todos os nodos externos abaixo de um determinado nodo n com bit de índice k tem como prefixo os mesmos k bits.

Assim, para obter as chaves ordenadas, basta imprimir as chaves dos nodos externos, percorrendo a árvore em-ordem.

Chamada: ordenado(raiz->esq, -1)

```
ordenado( p, bit )
  se p == nodoNulo retorna;
  se p->bit <= bit
    escreve p->reg.chave; retorna;
  ordenado( p->esq, p->bit )
  ordenado( p->dir, p->bit )
```

Custo:

Inserção: número médio de comparações = log(n)
número máximo de comparações = 2*log(n) <= sizeOf(k)

Arv. Patricia são especialmente indicadas para chaves grandes, pois evitam a comparação de todos os bits que a compõe.

Tries n-arias

Seção 15.4 (Sedgwick)

Generalização de tries, na qual chaves são codificadas em uma base qualquer, não necessariamente binária.

Uma trie n-aria possui chaves armazenadas nas folhas. Ela é def. rec. Da seguinte forma: uma trie para um conjunto vazio de chaves corresponde ao apontador nulo; uma trie com uma única chave corresponde a uma folha contendo esta chave; uma trie com cardinalidade maior que um é um nodo interno com apontadores referentes a trie com chaves começando com cada um dos dígitos possíveis, com este digito desconsiderado na construção das subárvores.

- Ex1: números na base decimal com 5 dígitos

```
39646                              ||||||||||
39555       \   \   \(2)              \(3)        44477   \(5) ....
21745                21745 23745  .....\(9)
23745                              39646  39555
44477
```

Ex2: casa, bela, rua

Inserção: cara, número

Trie Existencial:

Não guarda informação sobre o registro, apenas se uma determinada chave esta presente ou não na trie (n-aria).
Uma trie existencial para um conjunto de chaves é def. rec. Da seguinte forma: uma trie para um conjunto vazio de chaves corresponde ao apontador nulo; uma trie para um conjunto não vazio de chaves corresponde a um nodo interno com apontadores para nodos filhos contendo valores para cada valor de digito possível. Nestas subárvores o primeiro digito é removido para sua construção, de forma recursiva.

Ex: casa, bela, rua
Inserção: cara, número

- assumimos que nenhuma chave é prefixo de outra. Isso pode ser garantido de duas formas distintas:

1. chaves de tamanho fixo
2. um "marcador" de final de chave: neste caso este marcador é um valor que não aparece em nenhuma chave e é considerado como um dos dígitos possíveis de serem encontrados na construção da trie.

Implementação:

```
typedef struct nodo *Apontador;
  struct nodo {Apontador prox[R];}

  static Apontador raiz;

  void init() { raiz = null; }

  Apontador criaNodo(){
    int i;
    Apontador x = malloc( sizeof *x );
    for( i = 0; i < R; i++) x->prox = null;
    return x;
  }

  Item buscaR( Apontador p, Chave v, int w ){
    int i = digito(v, w);
    if (p == null) return NULLItem;
    if (i == NULLdigit) return v;
    return buscaR( p->prox[i], v, w+1);
  }

  busca( Chave v ){ return buscaR( raiz, v, 0 ); }

  Apontador insereR( Apontador p, Item item, int w ){
    Chave v = chave(item);
    int i = digito(v, w);
    if (p == null) h = criaNodo();
    if (i == NULLdigit) return p;
    p->prox[i] = insereR( p->prox[i], v, w+1);
    return p;
  }

  void insere(item){ raiz = insereR( raiz, item, 0); }
```

Características:

- árvore com altura baixa
- grande número de apontadores nulos

Consequências:

- baixo tempo de busca / inserção
- grande desperdício de espaço

Exemplo:
Éramos jovens e, como tal, sempre a buscar acelerar o tempo, indagando-nos sobre temas que os anos certamente se encarregariam de responder - mal sabíamos que, para entender certas coisas, bastava envelhecer.

Na forma de trie existencial, trie existencial abstrata e trie existencial ternaria

Trie Ternaria

- similar as arv. de busca binárias, mas que utiliza carácteres (dígitos) como chave do nodo
- cada nodo tem 3 apontadores: para chaves que começam com o digito menor que o corrente, iguais e maiores

Característica:

- tempo de busca: tamanho da chave
- número de links: no máximo 3 vezes o tamanho total do conjunto de chaves

Implementação:

```
typedef struct nodo *Apontador;
struct nodo {int d;
             Apontador dir, meio, esq;};
Apontador raiz;

void init() {raiz = NULL};

Apontador criaNodo( int d ){
  Apontador x = malloc( sizeof *x);
  x->d = d;
  x->dir = x->meio = x->esq = NULL;
  return x;
}

TipoChave buscaR( Apontador p, TipoChave k, int w ){
  int i = digito(k, w);
  se (p == NULL) return chaveNULA;
  se (p == digitoNULO) return k;
  se (i < h->d) return buscaR( p->esq, k, w );
  senão se (i == h->d) return buscaR( p->meio, k, w+1 );
  senão buscaR( p->dir, k, w );
}

TipoChave busca( TipoChave k ){
  return buscaR( raiz, k, 0 );
}
```

Vantagens:

- adapta-se as irregularidades (desbalanceamento dos carácteres que aparecem) nas chaves de pesquisa
- não dependem da quantidade de dígitos (carácteres) possíveis
- quando a chave não está armazenada na árvore, a quantidade de dígitos comparados tende a ser pequena (mesmo quando a chave de busca é longa)
- ela é flexível:
  . pode ser usada para obter chaves que casam com dígitos específicos da chave de pesquisa
  . pode ser usada para obter chaves que diferem em no máximo uma posição da chave de pesquisa
- arv. Patricia oferece vantagens similares, mas comparando bits ao invés de bytes.

Exemplo:
Busca de todas as palavras que casam com "ca*", onde "*" pode ser qualquer carácter

```
char s[MAXTAM];
void casaR( Apontador p, char *k, int i ){
```

```
  se p == NULL return;
  se ((*k == '\0') && (p->d == '\0'))
  {  s[i] = p->d; escreve s; }
  se ((*k == '*') || (*k == p->d))
  {  s[i] = p->d; casaR( p->meio, v+1, i+1 ); }
  se ((*k == '*') || (*k < p->d))
    casaR( p->esq, k, i );
  se ((*k == '*') || (*k > p->d))
    casaR( p->dir, k, i );
}

void casa( char *k )
{ casaR( raiz, k, 0 ); }
```

Melhoramentos possíveis na árvore:

1. como a maioria dos nodos próximos das folhas possuem apenas um único filho, utilizar a mesma ideia das arv. trie n-arias de manter a chave na folha no nível que a distingue das demais chaves. Isso torna a árvore *independente* do tamanho das chaves
2. utilizar a ideia das arv. Patricia de manter as chaves nos nodos internos, usando os apontadores "para cima".
3. usar um vetor com N elementos, um para cada digito possivel somente na raiz. -- busca similar a um catalogo telefônico. O objetivo é diminuir a altura da árvore e o número de comparações em uma busca.

Exemplo:
Se
- conjunto de chaves = 1 bilhão de chaves
- conjunto de dígitos = 256
- raiz com vetor de 256^2 = 65.536

#comparações de bytes é em torno de 10.

Compressão de Dados

Seção 17.3 (Cormem)

Motivação:

- Um texto contendo 100.000 carácteres em [a,f]:

```
                  a      b     c     d     e     f
---------------------------------------------
freq. (*1000)    45     13    12    16     9     5
cod. tam fixo   000    001   010   011   100   101
cod. tam var.     0    101   100   111  1101  1100
```

==> tamanho total com codificação fixa = 100.000 * 3 = 300.000
==> tamanho total com codificação variável = 45000*1 + 13000*3 + 12000*3 + 16000*3 + 9000*4 + 5000*4 = 224.000

Ganho de aprox. 25%.

Desafio:
- Geração da codificação de tamanho variável ótima.
  --> codificação de Huffman

Exemplo:

```
  cabaaed  100.0.101.0.0.1101.111
  bebada
```

Custo de uma codificação:

somatório_{c no texto} (freq(c) * tam(c)}, onde freq(c) é a frequencia do carácter c no texto e tam(c) é o tamanho da codificação de c (em bits)

Codificação Ótima:

É dada pela representação na forma de uma árvore binária *completa*. Para um conjunto C de carácteres, a árvore tem |C| folhas, |C|-1 e nodos internos.

- para que o separador "." não seja necessário, a codificação deve ser uma codificação de prefixo. Ou seja, nenhum código pode ser prefixo de outro.

- Ideia de Huffman: códigos menores são gerados para carácteres que aparecem no texto com maior frequência.
  --> geração de uma trie, na qual as chaves são os carácteres

Implementação:

Estrutura de dados:

```
typedef struct No *Apontador;
typedef struct No {
  Char k;             /* carácter */
  Int f;              /* frequencia */
  Apontador Esq, Dir;
} No;
```

```
Huffman( C )   /* C é um conjunto de nodos contendo as chaves e
frequências preenchidas */
```

```
n = |C|    /* tamanho do conjunto C *
Q = C      /* Q é uma fila de prioridades */
para i= 1 a n-1
  z = criaNodo();
  p1 = extraiMinimo( Q )
  p2 = extraiMinimo( Q )
  esq(z) = p1;  dir(z) = p2;
  z->f = p1->f + p2->f
  insere(Q, z)
retorna extraiMinimo( Q )
```

Codificação:
(arquivo texto T, com código Huffman na trie com raiz)

```
codifica( T, raiz )
  l = inicializaLista();
  geraVetorCod( raiz, v, l );
  enquanto não for fim de arquivo( Tc ){
    c = ler( Tc );
    escreve( v[c] );
  }

geraVetorCod( p, v, lista ){   /* gera um vetor indexado pelo
carácter */
   se p é folha {             /* contendo a codificao */
       v[p->k] = "conteudo da lista"
    }
    senão {
       insereFim( lista, 0 );
       geraVetorCod( esq(p), v, lista );
       removeFim( lista );

       insereFim( lista, 1 );
       geraVetorCod( dir(p), v, lista );
       removeFim( lista );
    }
```

Decodificação:
(arquivo codificado Tc, raiz da trie)

```
decodifica( Tc, raiz )
  enquanto não for fim de arquivo( Tc ){
    p = raiz;
    enquanto p não for folha {
      b = ler(Tc);
      se  b == 0
        p =  esquerda(p)
      senão
        p = direita (p)
    }
    escreve p->k
  }
```

Exemplo:
Processo de construção para a palavra ABRACADABRA.

O tempo de execução da construção da codificação de Huffman depende do tempo para obter os carácteres em ordem ascendente de frequência e inserir novos elementos no conjunto. Utilizando uma lista ordenada este tempo é *O(n)*, e a função Huffman teria gastaria então n-1 * n tempo, ou seja, *O(n^2)*.

Uma alternativa seria a utilização de uma árvore binária balanceada (como AVL ou RN). porém, no problema

em questão também há uma limitação no valor de n, que é previamente conhecido. Ou seja, há uma quantidade previamente sabida de carácteres que um arquivo pode conter. Assim, introduzimos mais uma estrutura de dados, chamada de heap para a implementação da lista de prioridades.

Uma árvore está *em ordem-maxheap* se a chave em cada nodo é maior ou igual às chaves armazenadas em todos os seus filhos.

Hash

Cap. 12 (Cormen)

- suporte ao tipo dicionario (insert, busca e remove) -- note sem ordenação
- é uma generalização do tipo vetor, na qual o intervalo dos valores de chave é muito maior que a quantidade de valores que serão armazenados. Ex: chaves no intervalo [0,10.000], mas apenas 1000 elementos no vetor.

- endereçamento direto: para chave no intervalo [0,n], alocar um vetor de *n+1* posições.

Problema: n pode ser muito grande

- abordagem:
    1. computar o valor de uma função de espalhamento (ou hash) no intervalo [0,m-1] -- h(k)
    2. armazenamento o elemento no elemento h(k) do vetor

Problema: pode haver colisões, ou seja, mais de uma chave com o mesmo valor de h(k). Paradoxo do aniversario: em um grupo de 23 ou + pessoas, existe uma chance de mais de 50% que duas pessoas façam aniversario no mesmo dia.

Exemplo: M = m
    h = k mod 7

Chaves: {4, 7, 13, 8, 9, 2}
    [4][0][6] [1][2][2]

Resolução de colisão por lista encadeada:
Constrói uma lista encadeada para cada endereço da tabela.

Custo de Busca:

Para n chaves:
- pior caso: O(n) se a função h mapeia todas as chaves para o mesmo elemento do vetor
- caso médio: O(n/m) n/m é o fator de carga (número médio de elementos em cada posição do vetor ==> para valores de m próximos de n: O(1)
- melhor caso: O(1)

Funções de Espalhamento:

Ideal:
- simples de ser computada
- para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer

Exemplo: para chaves uniformemente distribuídas no intervalo [0,1], *h(k) = floor(km)*

Para chaves não numéricas é preciso primeiro transformar a chave em inteiro:

Shift Folding: (deslocamento)

Somatório do código ASCII dos carácteres *K = sum_{i=0}^{n-1} Chave[i]*

Limit Folding: (dobramento ou sanfona)

Inverte o código a cada carácter: *K = sum_{i=0,2,4,6,..} Chave[i] + sum_{i=1,3,5,7,...} inverso(Chave[i])*

Usando o código ASCII e a posição do carácter:

*K = sum_{i=0}^{n-1} Chave[i] * 128^{n-i-1}*

Exemplo: pt = (112 * 128^1) + (116 * 128^0) = 14452

Usando pesos:

*K = sum_{i=1}^{n} Chave[i] * peso[i]*, onde
- Chave[i] é a representação ASCII do i-ésimo carácter da chave
- peso[i] é um inteiro randomicamente gerado.

Vantagem de usar pesos: conjuntos de pesos distintos geram funções de espalhamento distintos.

Exemplo: now: (110 * 128^2*3) + (111 * 128^1*4) + (119 * 128^0*1)

Funções de Espalhamento:

Método da Divisão:

h(k) = k mod m

- a escolha de m é importante. Em geral é escolhido um primo.
- motivo: no exemplo acima, se m for igual a 64 (2^6), então o resultado da função h é simplesmente os 6 bits menos significativos de k, enquanto é melhor considerar a chave como um todo.
- bom valor para m: primos não muito próximos a potencias de 2

Ex: n=2000, e queremos buscar uma chave examinando em media 3 elementos. Assim, 2000/3=666 e uma boa escolha pode ser 701, que é primo e não é prox. a uma potencia de 2.

Método da Multiplicação:

*h(k) = floor(m * (kA mod 1))*,
onde A é uma constante entre 0 e 1 "mod 1" é a parte fracionaria de kA

- vantagem: o método não é muito dependente do valor de m
- pode ser implementado de forma eficiente quando m = 2^p, da seguinte forma: considere uma chave de w bits e um valor s no intervalo (0, 2^w), tal que A = s / 2^w. Assim, primeiro obtêm-se k*s, que é um valor de 2w bits (r1,r0). Como m=2^p, o valor de h(k) corresponde aos p bits mais significativos de r0.

Exemplo:
A = 0.6180339887 ("razao de ouro", (sqrt(5)-1)/2) sugerido por Knuth
p = 14
m = 2^14 = 16384
w = 32
k = 123456

Procura-se um valor de A, ou seja, uma fração da forma s/2^32 que seja próxima de (sqrt(5)-1/2) -->
A=2.654.435.769 / 2^32.

Assim, k*s = 123.456 * 2.654.435.769 = 327.706.022.297.664 = 76300 * 2^32 + 17.612.864
Ou seja, r1 = 76300 e r0 = 17.612.864.
Os p (14) bits mais significativos de r0 resulta na valor de h(k) = 67.

Meio do Quadrado:

- Multiplica-se a chave por ela mesma e trunca-se as duas extremidades do resultado até o número de dígitos ser igual ao número de dígitos do endereço desejado (no intervalo da tabela hash).

Exemplo:
Endereçamento de 3 dígitos
Chave 134675 â†’ quadrado = 18137355625
Trucando â†’ 735

Chave 436987 â†’ quadrado = 190957638169
Trucando â†’ 763

Shift Folding: (deslocamento)

A mesma ideia da conversão de carácter para inteiro, mas dividindo o número de dígitos ou bits da chave para obter inteiros menores.

Ex: k = 12345678 para obter um índice de 3 dígitos da tabela hash:

```
 12 +
345 +
678
----
925   <--- ind. da tabela
```

Limit Folding: (dobramento ou sanfona)

A mesma ideia da conversão, mas para obter índices de tamanho menor. Neste caso em geral não sem leva em conta o "carry"
Ex: k = 12345678 para obter um índice de 3 dígitos

```
21 +
345 +
876
---
321 <--- ind. da tabela
```

Hashing universal:

- escolha de uma função hash randomicamente, que seja independente do valor da chave, em tempo de execução. Isso faz com que o sistema que está fornecendo as chaves não possa provocar o hash de chegar ao pior caso.
- o algoritmo pode ter comportamento distinto em cada execução

Exemplo (pesos para formação de chave numérica a partir de uma seq. de carácteres)

Ex: sabemos que o DDD e os 3 primeiros dígitos de um número telefônico não são distribuídos de forma uniforme: não servem para a função hash. Podemos supor que os 4 últimos dígitos de um número telefônico são distribuídos mais ou menos uniformemente, tornando-se uma boa opção. Assim, podemos dar peso 0 para os 3 primeiros dígitos.

-$h_{a,b}(k) = ((ak+b) \bmod p) \bmod m$, onde p é primo, a em [0,p) e b [1,p)

Tratamento de Colisão:

Lista encadeada:

Custo médio da busca sem sucesso:

Custo = $1/m \sum_{i=0}^{m-1}$ (tamanho lista do elemento i)

Endereçamento Aberto:

- usado quando a quantidade de registros a serem armazenados é previamente sabido --> escolhe-se m > n   e assim todas as chaves podem ser armazenadas na própria tabela

A função *h: U x {0,..., m-1} --> {0,...,m-1}*

Assim, para achar a posição de armazenamento de uma chave é realizada uma busca nas posições h(k,0), h(k,1), h(k,2) até achar uma que esteja vazia.

```
hash_insert(T, k)   /* T é a tabela hash */
  i = 0
  repita
     j = h(k,i)
     se T[j] == nil então T[j] = k; retorna
     i = i+1
  até que i = m
  retorna "erro: tabela cheia"
```

Busca similar
Remoção: não pode colocar nil na posição, mas sim um outro valor "REMOVIDO" para que a posição possa ser usada novamente por uma inserção ou a busca continuar o processo até encontrar a chave procurada.

Hashing Linear:

h(k,i) = (h'(k) + i) mod m

Exemplo:
chaves = 12,21,14,5,19   m=7

Problema:
- agrupamento -- a media que a tabela vai ficando cheia, uma nova chave tende a ocupar uma posição continua a uma já ocupada, piorando assim o tempo de pesquisa.
- consequência: o valor de h(k,0) determina a sequencia que será analisada

Custo da busca:
- pior caso O(n)
- melhor e caso médio O(1)

Hashing Duplo:

h(k,i) = (h1(k) + i*h2(k)) mod m

Neste caso, a primeira posição investigada é T[h1(k)]. As posições investigadas depois tem um deslocamento variável de h2(k) (modulo m). Assim, a sequencia de posições investigadas depende duplamente da chave. Assim, para cada par de valores (h1(k), h2(k)), a sequencia de posições do vetor investigada muda, gerando assim m^2 sequencias distintas.

Ex: m=13
h1(k) = k mod 13
h2(k) = 1 + (k mod 11)

inserção de 14 na tabela:

```
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]
   79        69  98   72             50
```

- o método linear é mais simples e portanto melhor para tab. esparsas
- se n/m for próximo de 1: melhor o duplo para minimizar a quantidade de comparações em casos de colisão

Ordenação Externa

Seção 4,2 (Nivio)

Necessária quando a quantidade a ser ordenada não cabe na memoria principal

Considerações:

- o custo para acessar um item é algumas ordens de grandeza maior que o os custos de processamento na memoria interna. Assim, o custo de um algoritmo de ordenação externa considera apenas a quantidade de leituras e escritas em memoria secundaria, ignorando o custo de processamento em memoria principal.
- podem existir restrições quanto ao acesso aos dados (sequencial / randômico)
- o desenvolvimento de algoritmos é muito dependente do estado atual da tecnologia

Estrategia Principal para Ordenação:

1. primeira passada sobre o arquivo quebrando em blocos do tamanho da memoria interna disponível. Cada bloco é ordenado na memoria interna.
2. os blocos ordenados são intercalados, fazendo varias passadas sobre o arquivo até que ele esteja completamente ordenado.

Objetivo:

Reduzir o número de passadas sobre o arquivo.

Entrada:
(considerando o arquivo armazenado em fita magnética)
- N registros para serem ordenados
- espaço em memoria principal para armazenar M registros
- 2P dispositivos externos

Intercalação Balanceada de Vários Caminhos:

Exemplo: intercalaçãoBALANCEADA (n=22)
Considerando M=3 e P=3 (intercalação de 3 caminhos)

Etapa 1: o arquivo é lido do dispositivo 0 de 3 em 3 (M) registros e armazenados em blocos de 3 nos dispositivos P a 2P-1
Resultado: N/M blocos de M registros ordenados

Etapa 2: intercalação dos blocos ordenados, escrevendo o resultado nos dispositivos 0 a P-1, repetindo até que todos os registros estejam ordenados.

Número de passos: 1 + log_{P} (N/M)
Exemplo: N=1.000.000.000 M=1.000.000 P=3 precisa de apenas 9 passos

Seleção por Substituição:

Objetivo:
Obtenção de sequencias maiores que M no primeiro passo utilizando uma lista de prioridades.

Ideia:
Retira-se o menor elemento dentre os M, e insere o próximo elemento x. Se x for menor que o ultimo elemento retirado, ele é marcado como maior que todos os demais para iniciar uma nova sequencia.

==> segundo Knuth, para números randômicos, o tamanho da sequencia gerada é em media igual a 2M.

Exemplo com heap size = 3

```
ent 1  2  3   ordenado
e   i  n  t   i
r   n  e* t   n
c   t  e* c*  t
a   a* e* c*  a*
l   c* e* l*  c*
a   e* a  l*  e*
c   l* a  c   l*
a   a  a  c   a
o   a  c  c   a
b   b  o  c   b
a   c  o  a*  c
l   l  o  a*  l
a   o  a* a*  o
n   a* n* a*  a*
c   a* n* c*  a*
e   c* n* e*  c*
a   e* n* a   e*
d   n* d  a   n*
a   a  d  a   a
    a  d      a
    d         d
```

O heap pode também ser utilizado para fazer as intercalações, mas só é vantajoso quando a quantidade de blocos gerados na primeira fase for grande (p. ex. >= 8). Neste caso, é necessário log_2(8) comparações para obter o menor elemento.

Exemplo:
entradas: int cer aal

```
ent  1  2  3  sai
     a  c  i   a
a    a  c  i   a
l    c  l  i   c
e    e  l  i   e
r    i  l  r   i
n    l  n  r   l
     n  r      n
t    r  t      r
     t         t
```