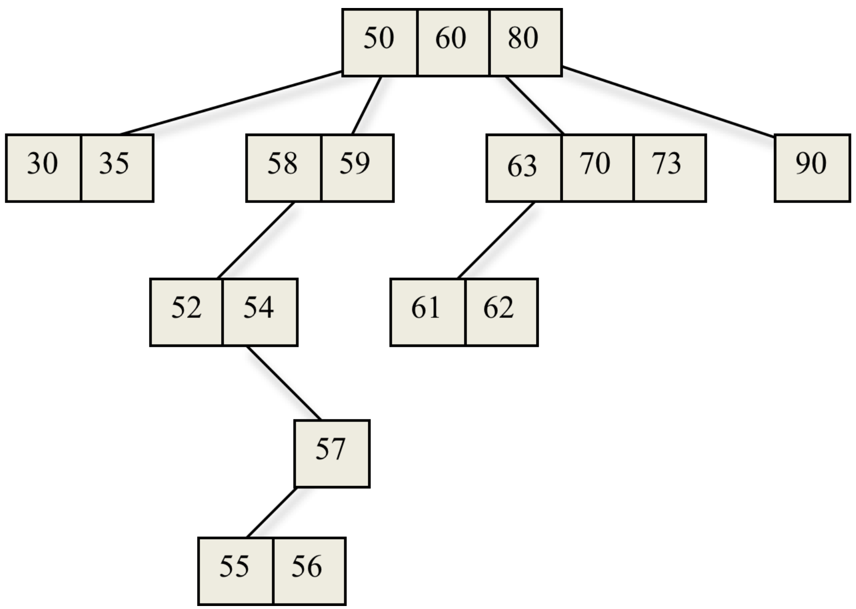
Árvores B

- Uma **árvore múltipla de procura** de ordem m é uma árvore tal que:
  - cada nó tem até m filhos e até m-1 chaves;
  - as chaves em cada nó estão em ordem crescente;
  - as chaves dos nós de um mesmo nível estão em ordem crescente.
- Exemplo: árvore múltipla de ordem 4



Árvores B

- As árvores múltiplas são usadas com o mesmo propósito das árvores binárias de procura: rápida atualização e recuperação de dados. No entanto, as árvores múltiplas são usadas, geralmente, para processar dados armazenados em dispositivos de memória secundária (como HDs, por exemplo), onde cada acesso é dispendioso.

Operações de acesso em discos

- Unidade básica para operações de E/S em disco: bloco. Quando um dado é lido de um disco, o bloco inteiro que contém esse dado é lido e transferido para a memória principal. Da mesma forma, dados são escritos em uma área de memória (*buffer*) até que completem um bloco, quando então são transferidos para o disco.

Árvores B

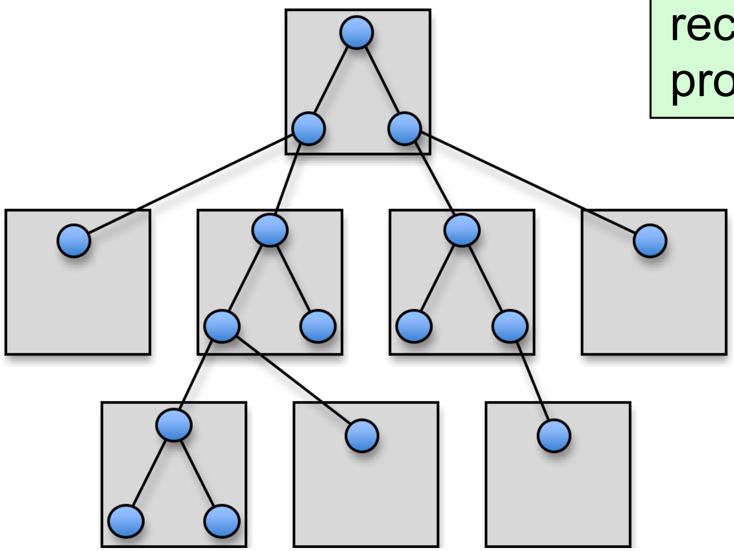
- Se um dado é solicitado de um disco:
  - o dado precisa ser localizado,
  - a cabeça precisa ser posicionada sobre a trilha do disco onde o dado reside, e
  - o disco precisa girar de modo que o bloco inteiro passe sob a cabeça para ser transferido para a memória.
- Portanto: tempo de acesso = tempo de procura (movimento mecânico da cabeça) + tempo de rotação (meia volta, em média) + tempo de transferência
- Exemplo: para transferir 5 KB de um disco que exige 40 ms para localizar uma trilha, trabalha a 3000 rpm e tem taxa de transferência de dados de 1000 KB/s:

  tempo = 40 ms + 0.5 rotação/(50 rps) + 5 KB/(1000 KB/s) =
  = 40 ms + 10 ms + 5 ms = 55 ms

  CPU: ordem de 10^{-9} segundos (10^6 vezes mais rápido).

Árvores B

- Se um programa usa constantemente dados armazenados em disco (como os sistemas de bancos de dados, por exemplo), este programa necessita de uma estrutura de dados que leve em conta as características deste meio de armazenamento.
- Exemplo: árvore binária de procura armazenada em vários blocos de disco.



Se for necessário acessar **muitos blocos** para recuperar um dado nesta árvore, o processamento será **muito lento**.

É melhor acessar uma grande quantidade de dados de uma só vez do que acessar muitos blocos.

Exemplo: transferir 10 KB

ta = 40 ms + 10 ms + 10 ms = 60 ms

Em dois blocos de 5 KB:

ta = 2 × (40 ms + 10 ms + 5 ms) = 110 ms

Árvores B

- Portanto, se os dados residem em disco, a estrutura de dados deve ser organizada de modo a minimizar o número de acessos ao disco.

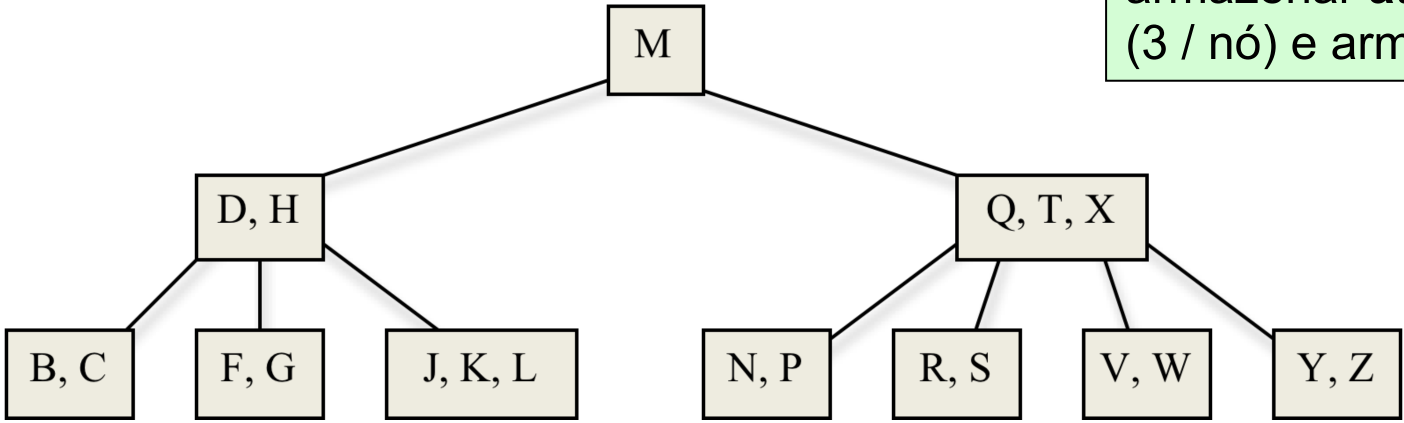
- Árvores B são estruturas de dados adequadas para organizar dados armazenados em discos. Uma propriedade importante de uma árvore B é o tamanho de seus nós, que pode ser tão grande quanto um bloco.

- Uma **árvore B** de ordem m é uma árvore múltipla de procura com as seguintes propriedades:
  - raiz (a menos que seja folha) tem, pelo menos, 2 filhos;
  - cada nó interno tem k filhos, com $\lceil m/2 \rceil \leq k \leq m$;
  - cada nó contém $k-1$ chaves, com $\lceil m/2 \rceil \leq k \leq m$;
  - todas as folhas da árvore estão no mesmo nível.

Árvores B

Exemplo: árvore B de ordem 4

Note que essa árvore pode armazenar **até 30 chaves** (3 / nó) e armazena 21.



- Observe que uma árvore B é uma árvore balanceada, de poucos níveis, e armazena, no mínimo, metade do total de chaves que pode armazenar.

- Usualmente: a ordem (m) de uma árvore B é grande (500, por exemplo), de modo que a quantidade de dados de um bloco caiba em um nó da árvore.

Busca em árvores B

- No caso de recuperação, o pior caso ocorre quando a árvore B tem o menor fator de ramificação em cada nó ($q = \lceil m/2 \rceil$) e a busca tem que atingir uma folha da árvore. Neste caso, em uma árvore B de altura h existem:

  1 chave na raiz +
  2 (q - 1) chaves no nível 1 +
  2 q (q - 1) chaves no nível 2 +
  2 q^2 (q - 1) chaves no nível 3 +
  ...
  2 q^{h-1} (q - 1) chaves no nível h =

  $$= 1 + 2(q-1)\left(\sum_{i=0}^{h-1} q^i\right) = -1 + 2q^h$$

Logo, o **número de chaves** (n) é tal que:

$$\boldsymbol{n \geq 2q^h - 1}$$

ou seja:

$$\boldsymbol{h \leq \log_q[(n+1)/2]}$$

- Portanto: para m suficientemente grande, h será pequeno mesmo que exista um grande número de chaves na árvore.

Busca em árvores B

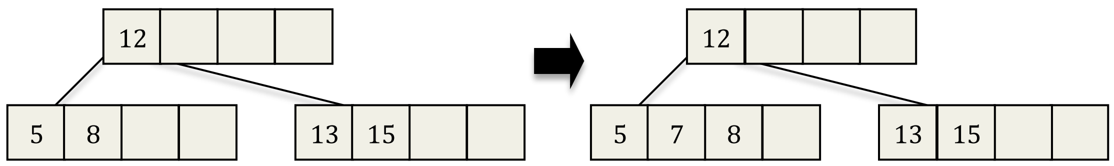
Exemplo:

- $m = 200$ e $n = 2000000$
- Então: $q = 100$ e $h \leq \log_{100}(2000001/2) = 3$
- Portanto, encontrar uma chave nessa árvore B exige, no pior caso, $h + 1 = 4$ buscas.

- Para compreender o algoritmo de recuperação é necessário entender com a árvore B é construída. Portanto, vamos analisar antes o algoritmo de inclusão de chaves em uma árvore B.
- Em **árvores binária de procura**, a inclusão de novas chaves é feita da raiz para as folhas, resultando em árvores desbalanceadas. Por exemplo, se a primeira chave incluída na árvore é a menor chave possível (e incluída na raiz), a raiz da árvore não terá uma subárvore esquerda.

Busca em árvores B

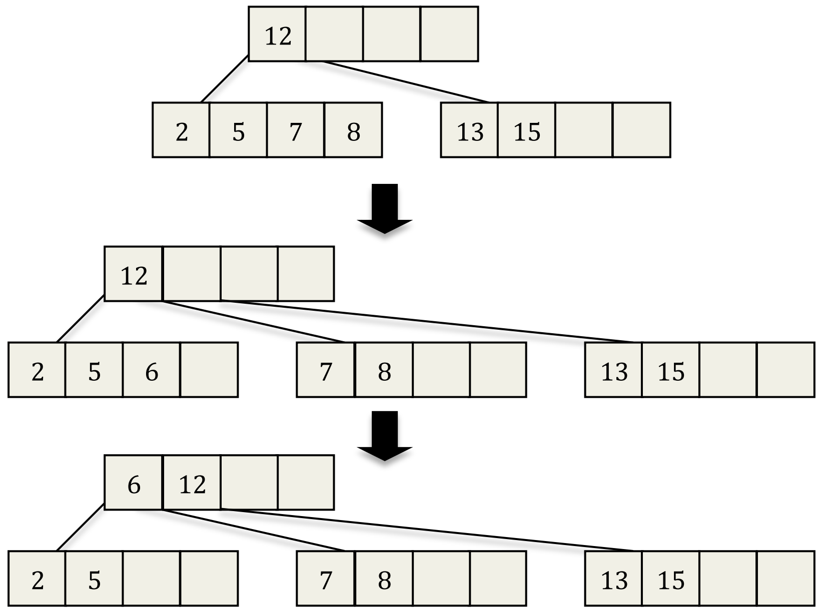
- Numa árvore B, a inclusão de novas chaves é feita das folhas para a raiz. Dessa forma, a raiz da árvore pode se alterar a cada inclusão.
- Três situações podem ocorrer ao inserir uma nova chave:
  - A nova chave é colocada em uma folha qua ainda tem espaço. **Exemplo**: incluir a chave 7.



  - A folha onde a chave precisa ser incluída está cheia, mas ainda é possível criar um novo filho para o pai desta folha. Neste caso, cria-se uma nova folha e metade das chaves é movida da folha cheia para a nova folha. Em seguida, a última chave da antiga folha cheia é transferida para o nó ascendente.

Busca em árvores B

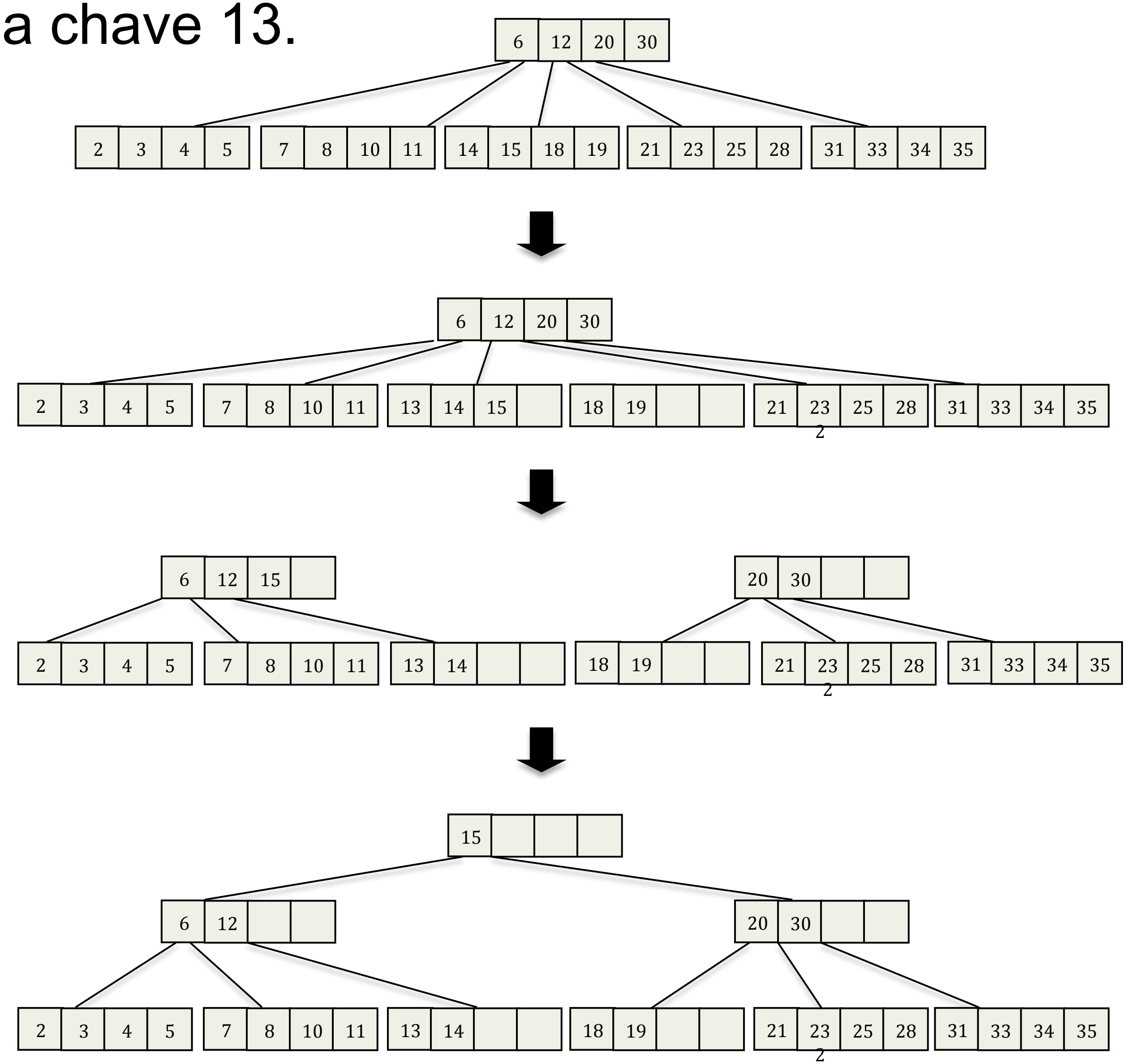
- **Exemplo**: incluir a chave 6.



- Não há possibilidade de criar um novo filho para o pai da folha cheia ou o nó ascendente também está cheio. Neste caso, será preciso criar um novo nível na árvore.

Busca em árvores B

Exemplo: incluir a chave 13.



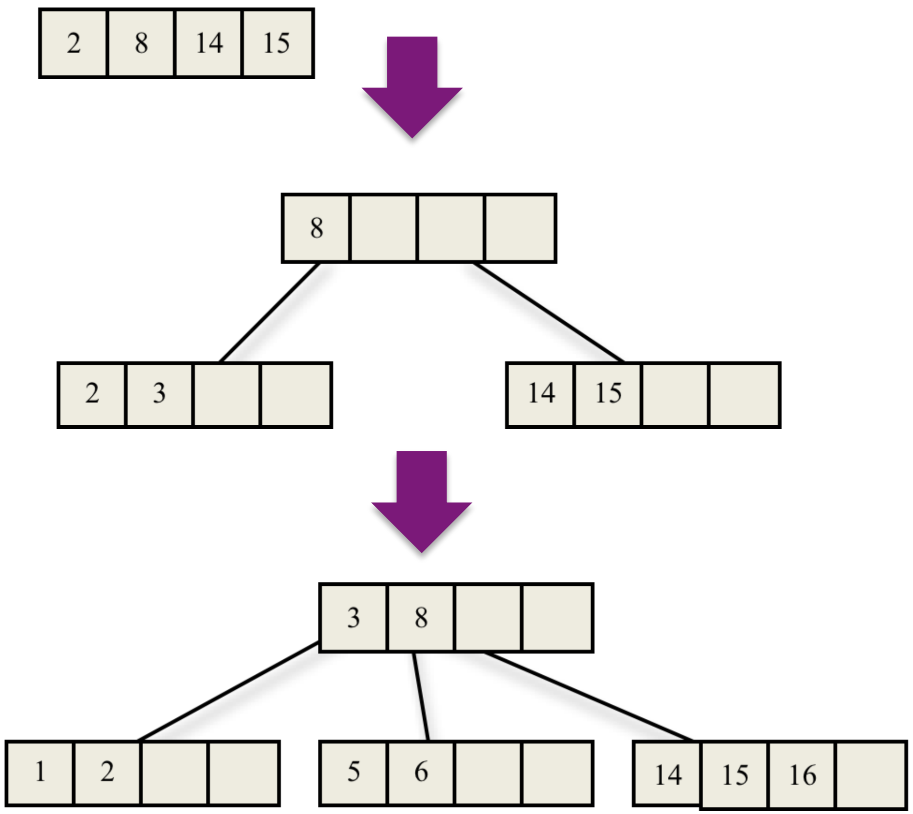
Busca em árvores B

Algoritmo de inclusão:

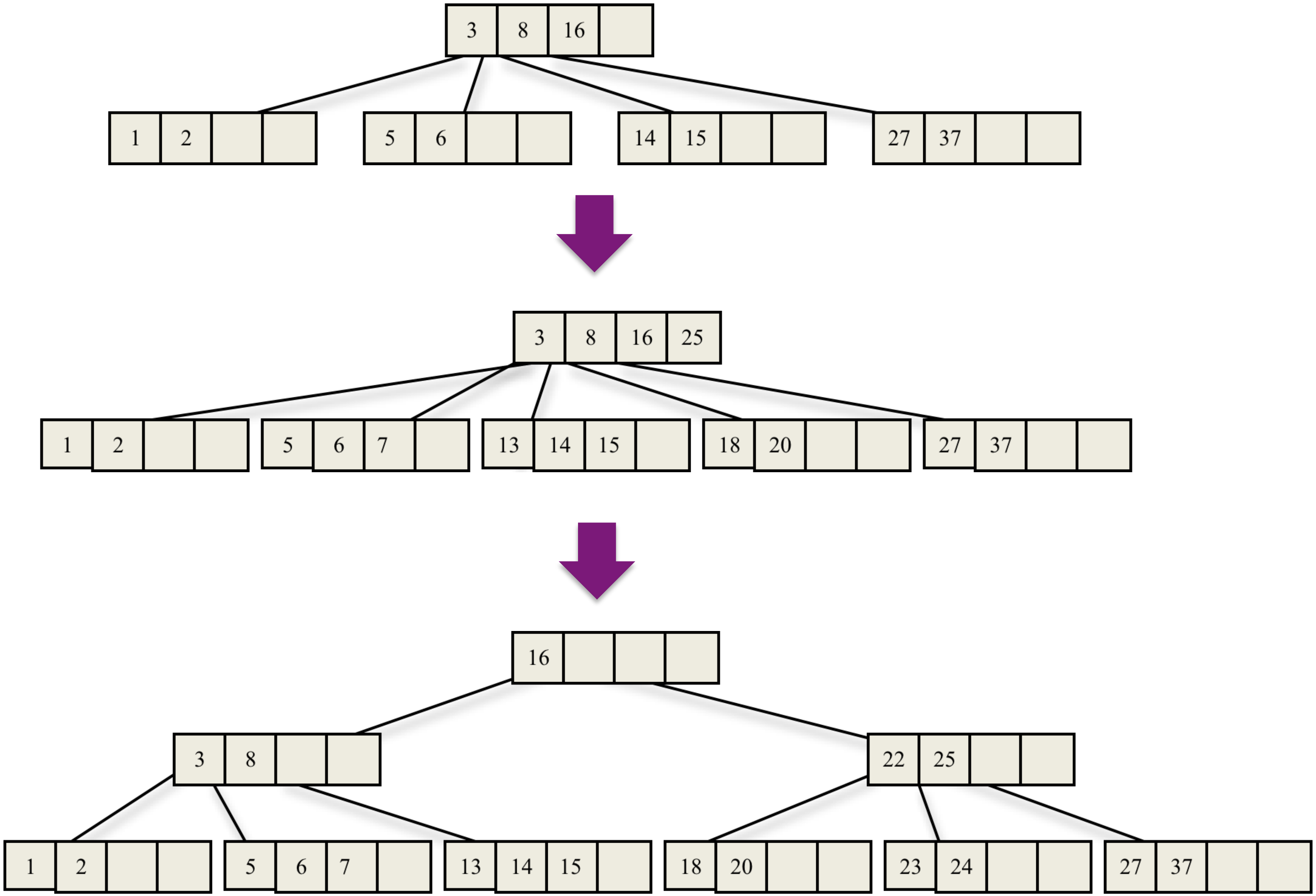
```
void Incluir(int k)
{
  node = folha onde inserir k;
  while (true)
  {
    Encontrar a posição apropriada para k em node;
    if (! cheia(node))
    {
      Inserir k;
      return;
    }
    Dividir node em node1 (velho) e node2 (novo);
    Distribuir as chaves igualmente entre node1 e node2;
    if (node == raiz)
    {
      Criar uma nova raiz como ascendente de node1 e node2;
      Inserir k na raiz;
      return;
    }
    node = pai(node);
  }
}
```

Busca em árvores B

- **Exemplo**: sequência de inclusão em árvore B de ordem 5 para o seguinte conjunto de chaves: {8, 14, 2, 15, 3, 1, 16, 6, 5, 27, 37, 18, 25, 7, 13, 20, 22, 23, 24}. Note que a todo momento a árvore está balanceada.



Busca em árvores B



Busca em árvores B

- Observe a relação entre os valores das chaves de um nó e os valores das chaves dos filhos deste nó: se x é o valor da i-ésima chave de um nó (x = chave[i]), então os valores das chaves do i-ésimo filho deste nó são todos menores do que x e os valores das chaves do (i+1)-ésimo filho deste nó são todos maiores do que x.

Exemplo: Seja o nó (3, 8, _, _).

- 1ª chave = 3. Logo, todas as chaves do 1º filho deste nó são menores do que 3 (1 e 2) e todas as chaves do 2º filho são maiores do que 3 (5, 6 e 7).
- 2ª chave = 8. Logo, todas as chaves do 2º filho deste nó são menores do que 8 (5, 6 e 7) e todas as chaves do 3º filho deste nó são maiores do que 8 (13, 14 e 15).
- E assim por diante, para qualquer nó da árvore.

Busca em árvores B

- Esta observação facilita a compreensão do algoritmo de recuperação. O algoritmo procura pela chave k e retorna um ponteiro para o nó que contém esta chave, ou NULL, caso a chave não exista na árvore B.

- Algoritmo:

```
arvoreB* Procurar(int k, arvoreB* p)
{
  if (p != NULL)
  {
    for (i = 0; i < p->numChaves && p->chave[i] < k; i++);

    if (i > p->numChaves || p->chave[i] > k)
      return Procurar(k,p->filho[i]);
    else
      return p;
  }
  else
    return NULL;
}
```

Exclusão de chaves em árvores B

- A operação de exclusão é, em grande parte, o inverso da operação de inclusão, embora existam mais casos particulares.
- Após uma exclusão, é preciso evitar que um nó esteja menos da metade cheio. Isto significa que, em alguns casos, dois nós tenham que ser fundidos em um único.
- A operação de exclusão pode ser dividida em dois casos principais:
  - exclusão de chave de uma folha;
  - exclusão de chave de um nó interno da árvore B.

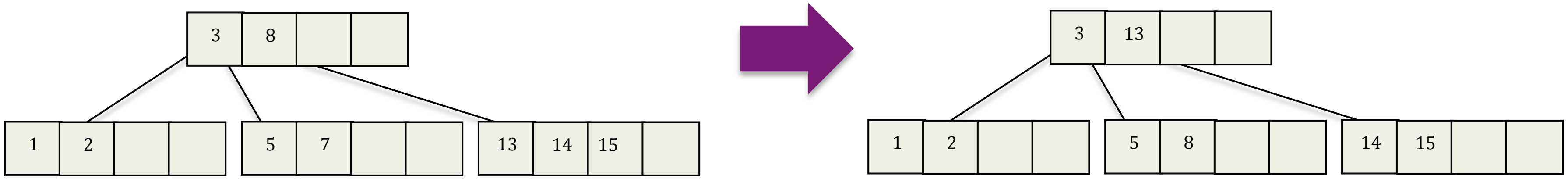
Exclusão de chaves em árvores B

Exclusão de chave de uma folha:

1. Se depois de excluir uma chave k, a folha está pelo menos metade cheia, basta mover as demais chaves da folha (maiores do que k) uma posição para a esquerda. Este caso é o inverso do 1º caso da operação de inclusão.
2. Se depois de excluir uma chave k, a folha torna-se subutilizada (com menos de $\lceil m/2 \rceil$-1 chaves), dois casos devem ser considerados:
   - 2(a): Se existir um irmão à esquerda ou à direita com mais chaves do que o mínimo, redistribuem-se as chaves da folha e do irmão, move-se a chave correspondente do nó ascendente para a folha e uma chave do irmão para o ascendente.

Exclusão de chaves em árvores B

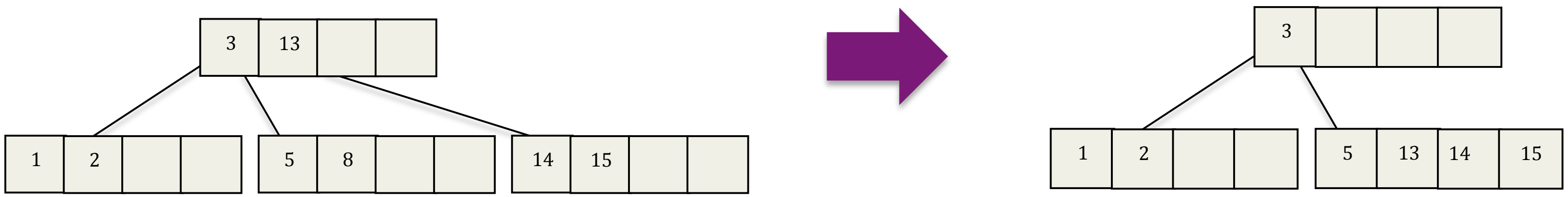
Exemplo: excluir a chave 7.



- 2(b): A folha está subutilizada e o irmão tem o mínimo de chaves. Então esses nós são fundidos, as chaves da folha e do irmão e mais a chave correspondente do ascendente são colocadas na folha e o irmão é descartado. Isto pode gerar uma cadeia de operações, se o ascendente também se tornar subutilizado. Neste caso, trata-se o ascendente como uma folha e repete-se o passo (2b) até que o passo (2a) possa ser executado ou a raiz da árvore seja alcançada. Isto é o inverso do 2º caso da operação de inclusão.

Exclusão de chaves em árvores B

Exemplo: excluir a chave 8.



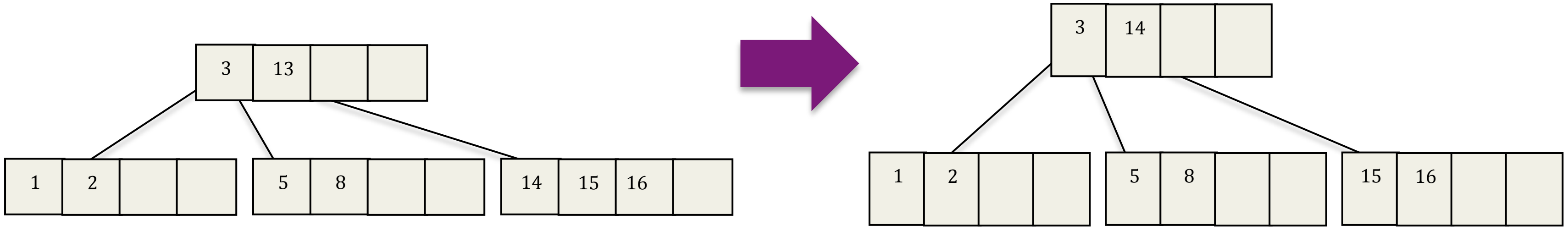
3. Um cuidado especial deve ser tomado quando o ascendente é a raiz e tem somente uma chave. Neste caso, as chaves da folha, do irmão e da raiz são todas colocadas na raiz e tanto a folha como o irmão são descartados. Com isso, a altura da árvore será diminuída de 1. Este caso é o inverso do 3º caso da operação de inclusão.

Exclusão de chaves em árvores B

Exclusão de chave de um nó interno:

- Excluir uma chave de um nó interno da árvore pode levar a problemas com a reorganização da árvore. Por isto, este caso é transformado em excluir uma chave de uma folha.
- A chave a ser excluída é substituída por seu sucessor (ou predecessor) imediato, que pode aparecer somente em uma folha. Esta chave é excluída da folha e colocada na posição da chave a ser excluída efetivamente.

Exemplo: excluir a chave 13.



Exclusão de chaves em árvores B

- Existem algumas variações de árvore B. Por exemplo:
  - **Árvores B***. Exige-se que todos os nós, exceto a raiz, estejam pelo menos 2/3 cheios (em vez de 1/2 cheios).
  - **Árvores B+**. O que acontece com a travessia em-ordem em uma árvore B? Como cada nó da árvore está em um bloco, para os nós internos seria feito um acesso a um bloco, mas somente uma chave deste nó seria incluído na listagem e então um outro bloco teria que ser acessado. Com isto, seriam necessários vários acessos a um mesmo bloco para os nós internos da árvore. As árvores B+ oferecem uma solução para este problema. Os dados são colocados nas folhas (implementadas como listas encadeadas) e os nós internos são índices.

Ver **Capítulo 7** do livro: DROZDEK, A. *Estruturas de Dados e Algoritmos em C++*, São Paulo: Thomson, 2002.

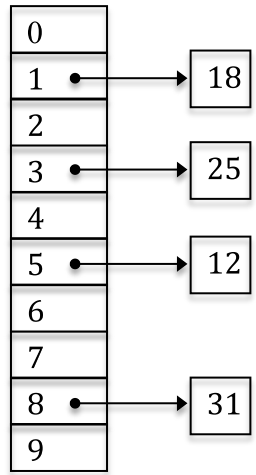
Tabelas *hash*

- Uma tabela *hash* é uma estrutura de dados eficiente para implementar dicionários, ou seja, conjuntos dinâmicos que admitem apenas as operações **Incluir**, **Excluir** e **Recuperar**. Uma tabela hash é uma generalização do endereçamento direto, como existe em um vetor.

- Exemplo: um conjunto dinâmico no qual cada elemento possui uma chave (distinta) definida em C = {0, 1, ..., m-1} pode ser representado por um vetor T[0 .. m-1] (conhecido como **tabela de endereçamento direto**), no qual cada posição corresponde a uma chave de C. Cada elemento de T é um ponteiro para uma estrutura de dados que contém o elemento correspondente a esta chave. T[k] = NULL, se o conjunto não contém um elemento com chave igual a k.

Tabelas *hash*

- Exemplo: conjunto dinâmico D = {18, 25, 12, 31}, com chaves do conjunto K = {1, 3, 5, 8}, respectivamente, onde o conjunto possível de chaves é C = {0, 1, ..., 9}.



Neste caso, a implementação de cada operação de dicionário é trivial e pode ser realizada em tempo O(1):

```
int Recuperar(int T[], int k)
{
  return T[k];
}
```

```
void Incluir(int T[], int x)
{
  T[chave(x)] = x;
}
```

```
void Excluir(int T[], int x)
{
  T[chave(x)] = NULL;
}
```

Tabelas *hash*

- E se o valor de m (número possível de chaves) for muito grande? Neste caso, armazenar uma tabela de tamanho m pode ser impraticável.
- Quando | K | << | C |, ou seja o número de chaves realmente utilizadas é muito menor do que o número de chaves possíveis, uma tabela *hash* exige muito menos espaço (O(| K |)) do que uma tabela de endereçamento direto, embora as operações de dicionário possam ser feitas em tempo O(1), em média.
- Numa tabela *hash* H[0 .. *m*-1], o elemento com chave *k* é armazenado na posição h(*k*) (e não na posição *k*, como na tabela de endereçamento direto), em que h é denominada **função *hash*** e tal que:

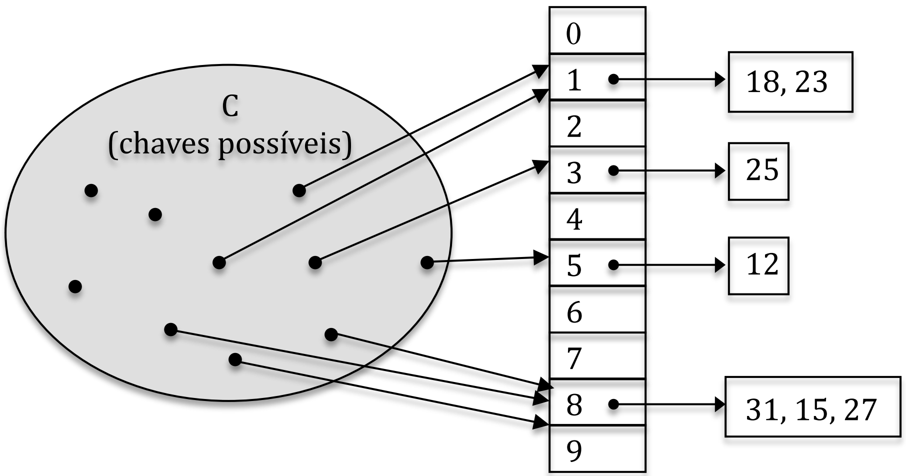
$$h : C \rightarrow \{0, 1, \ldots, m\text{-}1\}$$

h(*k*) é o **valor *hash*** da chave *k*

Tabelas *hash*

- Quando duas chaves k_1 e k_2, com $k_1 \neq k_2$, são tais que h (k_1) = h(k_2), diz-se haver uma **colisão**. Notar que, como | C | > m, as colisões são inevitáveis.

- O ideal é que a função *hash* distribua as colisões igualmente para cada valor de chave (a palavra *hash* significa "recortar em muitos pedaços").



- Como resolver as colisões?

Tabelas *hash*

Resolução de colisões por encadeamento

- Utilizar uma lista encadeada para armazenar todos os elementos que efetuam *hash* para uma mesma posição.
- As operações de dicionário continuam fáceis de implementar:

```
int Recuperar(int H[], int k)
{
  procurar pelo elemento com chave k
  na lista apontada por H[h(k)];
}

void Incluir(int H[], int x)
{
  incluir x no início da lista apontada por H[h(chave(x))];
}

void Excluir(int H[], int x)
{
  excluir x da lista apontada por H[h(chave(x))];
}
```

Tabelas *hash*

- Notar que:
  - **Incluir**: pode ser realizada em tempo O(1).
  - **Excluir**: pode ser feita em tempo O(1), caso exista um ponteiro para o elemento *x* a ser excluído e a lista for duplamente encadeada (se a lista for simples, mesmo havendo um ponteiro para *x*, será preciso percorrer a lista para determinar o predecessor de *x*, o que no pior caso, pode significar percorrer quase a lista inteira).
  - **Recuperar**: o tempo de execução é proporcional ao tamanho da lista que contém o elemento com chave igual a *k*.

Tabelas *hash*

- H: tabela *hash* com *m* posições e que armazena *n* elementos. Então, $\alpha = n/m$ é o **fator de carga** para H.
- Note que o fator de carga de H é o número médio de elementos armazenados em cada uma das listas encadeadas de H.
- No pior caso, todas as *n* chaves executam *hash* para a mesma posição. Neste caso, existe apenas uma lista encadeada de *n* elementos.
- Portanto, o tempo da operação **Recuperar**, no pior caso, é O(*n*) (mais o tempo necessário para calcular a função *hash*).
- O desempenho médio de uma tabela *hash* depende de como a função *hash* distribui, em média, as *n* chaves nas *m* listas.

Tabelas *hash*

- Supondo que:
  - Qualquer das *n* chaves tem igual probabilidade de efetuar *hash* para qualquer das *m* listas (hipótese conhecida como ***hash* uniforme simples**), ou seja, o tamanho médio de qualquer lista é $n/m = \alpha$;
  - O valor *hash* h(*k*) pode ser calculado em tempo O(1), ou seja, o tempo necessário para recuperar um elemento com chave *k* depende linearmente do comprimento $n_{h(k)}$ da lista H[h(*k*)],

  pode-se mostrar que o tempo esperado para a operação **Recuperar** em uma tabela *hash* em que as colisões são resolvidas por lista encadeada é $O(1 + \alpha)$.

Ver **Seção 11.2** do livro: CORMEN, T.H.; LEISERSON, C.E.; RIVEST, R.L.; STEIN, C. *Algoritmos: Teoria e Prática*, Rio de Janeiro: Elsevier, 2002. p. 181-185.

Tabelas *hash*

- Portanto, se o número de posições da tabela *hash* (m) for proporcional ao número de elementos na tabela (*n*), ou seja, se n = O(m), então α = n/m = O(m)/m = O(1). Assim, a recuperação pode ser feita, em média, em tempo O(1).
- Resumindo, para uma tabela *hash* na qual as colisões são resolvidas por lista encadeada, as operações:
  - **Incluir**: pode ser realizada em tempo O(1).
  - **Excluir**: pode ser realizada em tempo O(1), se existir um ponteiro para o elemento a ser excluído e a lista usada for duplamente encadeada.
  - **Recuperar**: pode ser realizada em tempo O(1), se vale a hipótese de *hash* uniforme simples, n = O(m) e a função *hash* puder ser calculada em tempo O(1).

Funções *hash*

- Uma boa função *hash* distribui uniformemente as *n* chaves nas *m* listas.

  Exemplo: se as chaves são números reais distribuídos independente e uniformemente no intervalo [0, 1), então a função: $h(k) = \lfloor k\, m \rfloor$ satisfaz à condição de *hash* uniforme simples.
- Nos métodos que veremos para calcular funções *hash* supõe-se que as chaves possíveis são números naturais. Se as chaves não pertencem a N = {0, 1, 2, ... }, deve-se encontrar uma maneira de interpretá-las desta forma.

  Exemplo: chaves são alfabéticas.

  "CAP" pode ser vista como a lista (67, 65, 80).

  Base 128: "CAP" = $67 \times 128^2 + 65 \times 128^1 + 80 \times 128^0 = 1106128$.

Funções *hash*

O método de divisão

- Neste método, função *hash* é dada por: **h(*k*) = *k* mod *m***.
  Exemplo: $m = 100$ e $k = 1106128$. Então: $h(k) = 28$.
- Para este método, em geral, evitam-se certos valores de *m*. Por exemplo, se $m = 2^p$, então $h(k)$ será o inteiro formado pelos *p* bits de mais baixa ordem de *k*.

  Exemplo: $m = 2^5 = 32$
  - Para $k = 6128$, $h(k) = 16$.

    Notar que: $6128 = (1011111110000)_2$ e, portanto, os 5 bits de mais baixa ordem de 6128 são: $(10000)_2 = 16$.
  - Portanto, qualquer outra chave, cuja representação binária termine com 10000, por exemplo, $(1010101010000)_2 = 5456$, também terá o valor *hash* 16. Note que $h(5456) = (5456 \bmod 32) = 16$.

Funções *hash*

- Portanto, se escolhermos $m = 2^p$, para algum valor de p, os valores *hash* não vão depender de todos os *bits* da chave como seria desejável.
- Um número primo não próximo de uma potência de 2 é, frequentemente, uma boa escolha para m.

  Exemplo:
  - Deseja-se armazenar $n = 2000$ elementos em uma tabela *hash*, com colisões resolvidas por listas de aproximadamente 3 elementos.
  - Portanto, uma boa escolha para m é um número primo próximo de 2000/3, mas não próximo de uma potência de 2.
  - Por exemplo, $m = 673$.

Funções *hash*

O método de multiplicação

- Neste método, a função *hash* é dada por:

  $$\mathbf{h}(k) = \lfloor m\ (k\mathbf{A}\ \mathbf{mod}\ 1)\rfloor$$

  onde A é uma constante (0 < A < 1). Note que (*k*A mod 1) significa "a parte fracionária de *k*A".
- Uma vantagem do método de multiplicação é que o valor de *m* não é crítico. Em geral, escolhe-se $m = 2^p$, para algum inteiro *p*, para facilitar os cálculos em computadores (que têm palavras de memória cujos comprimentos são potências de 2).
- Knuth sugere: $A \cong (\sqrt{5} - 1) = 0.61803398$

Ver o livro: KNUTH, D.E. *The Art of Computer Programming - Volume 3: Sorting and Searching*. Reading: Addison, 1973.

Tabelas *hash* com endereçamento aberto

- Nas tabelas *hash* com endereçamento aberto, todos os elementos são armazenados na própria tabela, ou seja, cada entrada da tabela contém um elemento ou NIL (indicando que a posição está vazia).
- Neste caso, a tabela *hash* pode ficar cheia (o fator de carga não pode exceder 1).
- A operação **Incluir** examina sucessivamente a tabela *hash* até encontrar uma posição vazia. Em vez de examinar as posições da tabela na ordem 0, 1, ..., m-1, a sequência de posições a ser examinada (sondagem) depende da chave a ser inserida.
- Para isso, estende-se a função *hash*, incluindo o número da sondagem como segundo parâmetro:

$$h : C \times \{0, 1, ..., m\text{-}1\} \rightarrow \{0, 1, ..., m\text{-}1\}$$

Tabelas *hash* com endereçamento aberto

- Para uma chave *k*, a sequência de sondagem será:

  $$h(k, 0), h(k, 1), ..., h(k, m\text{-}1)$$

  e deve ser uma permutação de (0, 1, ..., *m*-1), para permitir que qualquer posição da tabela possa ser eventualmente considerada para a nova chave.

Algoritmo:

```
int Incluir(int H[], int k)
{
  int i,j;

  for (i = 0; i < m; i++)
  {
    j = h(k,i);
    if (H[j] == NIL)
    {
      H[j] = k;
      return j;
    }
  }
  Erro("A tabela hash está cheia");
}
```

Nos algoritmos a seguir, supõe-se que o elemento que contém a chave *k* é igual à própria chave.

Tabelas *hash* com endereçamento aberto

- A operação **Recuperar** efetua mesma sequência de sondagem. A busca termina sem sucesso ao encontrar uma posição vazia (pois, pelo algoritmo de inclusão, a chave seria inserida nesta posição e não mais adiante na sequência de sondagem). Isso é válido se as chaves não são excluídas da tabela.

Algoritmo:

```
int Recuperar(int H[], int k)
{
  int i,j;

  for (i = 0; i < m; i++)
  {
    j = h(k,i);
    if (H[j] == NIL)
      return NIL;
    if (H[j] == k)
      return j;
  }
  return NIL;
}
```

Tabelas *hash* com endereçamento aberto

- Para a **exclusão** é necessário um cuidado especial. Ao excluir uma chave da posição *j*, se assinalarmos essa posição como NIL (posição vazia), será impossível recuperar uma chave *k* que foi incluída após a posição *j.*
- Uma solução é marcar a posição *j* como DEL (em vez de NIL), indicando que a chave desta posição foi excluída.
- Com isto, deve-se alterar a função **Incluir** para que as posições DEL também sejam consideradas como vazias, ou seja:

```
if ((H[j] == NIL) || (H[j] == DEL))
```

- Nenhuma alteração será necessária na função **Recuperar.**

Tabelas *hash* com endereçamento aberto

- A suposição de que, para uma chave, qualquer das m! permutações de (0, 1, ..., m-1) tem igual probabilidade de ser a sequência de sondagem desta chave é conhecida como ***hash* uniforme**.

- O *hash* uniforme é uma generalização do *hash* uniforme simples, pois a função *hash* não produz apenas uma posição única, mas uma sequência inteira de sondagem.

- As técnicas usadas para determinar uma sequência de sondagem garantem que a sequência:

$$(h(k, 0), h(k, 1), ..., h(k, m-1))$$

  é uma permutação de (0, 1, ..., m-1) para qualquer chave k. Mas, essas técnicas não produzem m! sequências distintas, ou seja, não implementam o *hash* uniforme.

-

Tabelas *hash* com endereçamento aberto

- O método mais simples é a **sondagem linear**, que utiliza uma função *hash* da forma:

  $$h(k, i) = (h'(k) + i) \bmod m \quad (i = 0, 1, ..., m\text{-}1)$$

  onde h' é uma função *hash* comum.

- Portanto, na sondagem linear a sequência começa na posição h'(k) e segue sequencialmente a partir daí.

- Este método tem uma tendência de criar agrupamentos na tabela (diversas posições ocupadas em sequência).

- Isso prejudica o desempenho da tabela *hash* porque as posições vazias que aparecem depois de um agrupamento têm mais chance de serem preenchidas do que as outras posições (e quanto maior for o agrupamento, maior é a probabilidade dele se tornar maior ainda).

Tabelas *hash* com endereçamento aberto

- Um dos melhores métodos para resolver colisões em tabelas *hash* com endereçamento aberto é o ***hash* duplo**:

  $$h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m \qquad (i = 0, 1, ..., m-1)$$

  onde h_1 e h_2 são funções *hash* comuns.
- Os valores de $h_2(k)$ e m devem ser primos entre si para que a tabela inteira seja pesquisada. Uma maneira de conseguir isto é escolher m como um número primo e fazer com que h_2 retorne um inteiro positivo menor do que m. **Exemplo**:
  - $h_1(k) = k \bmod m$
  - $h_2(k) = 1 + (k \bmod (m-1))$
- Este método consegue obter m^2 sequências de sondagem distintas e estas sequências se parecem com permutações escolhidas aleatoriamente.