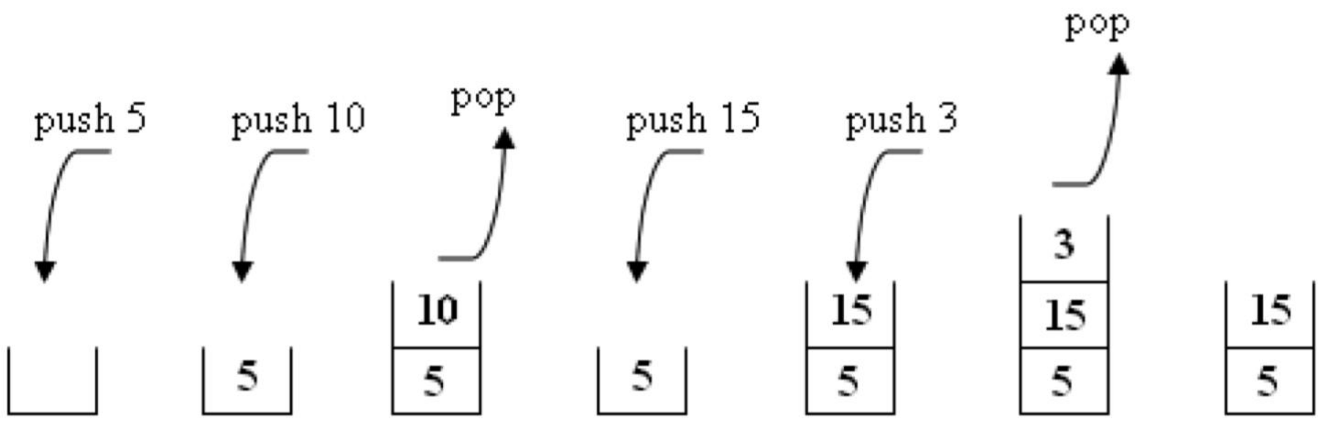
Pilhas e Filas

- Como vimos na aula anterior, uma lista encadeada é uma sequência de elementos (nós da lista), ordenados logicamente por meio de ponteiros.
- Estas estruturas de dados gerais podem ser usadas para implementar tipos abstratos de dados mais específicos, como a **pilha** e a **fila**.
- Na pilha, um ponteiro indica o **topo da pilha** e as operações de inclusão de novos elementos e exclusão de elementos da lista é realizada em relação ao topo da pilha.
  - Inclusão (**empilhamento** ou ***push***) coloca um novo elemento no topo da pilha.
  - Exclusão (**desempilhamento** ou ***pop***) retira o elemento que está no topo da pilha.

Pilhas e Filas

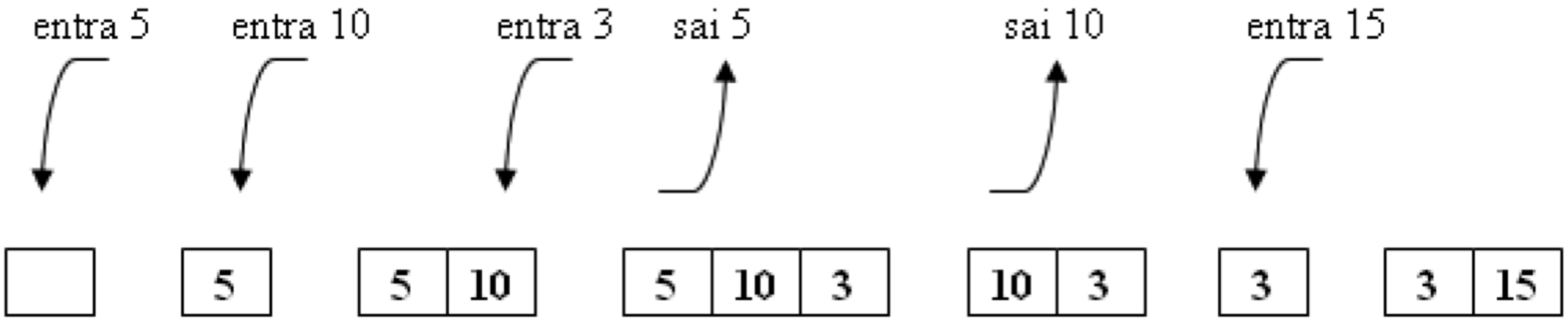
- Numa pilha, o regime de permanência de elementos na estrutura é da forma LIFO (“*last-in, first-out*”): “o último elemento que entra é o primeiro elemento que sai”.



- Na fila, um ponteiro indica o **início da fila** e outro ponteiro indica o **final da fila**. A inclusão de novos elementos é feita sempre no final da fila e a exclusão de elementos é realizada sempre no início da fila.

Pilhas e Filas

- Numa fila, o regime de permanência de elementos na estrutura é da forma FIFO ("*first-in, first-out*"): "o primeiro elemento que entra é o primeiro elemento que sai".

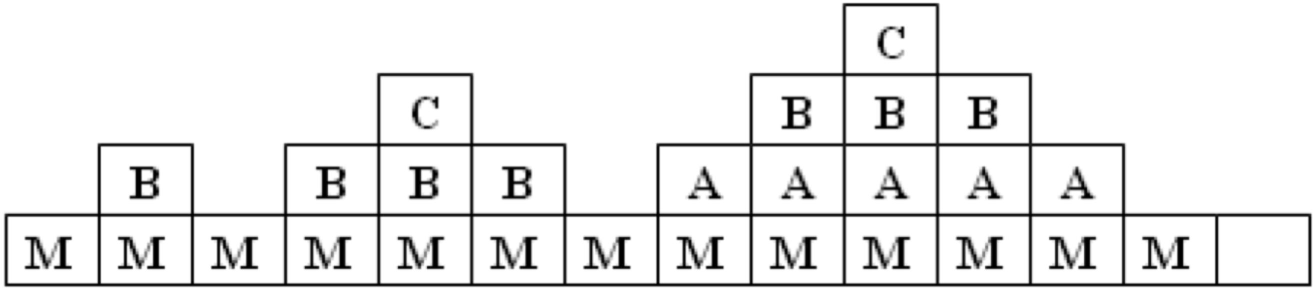


- As estruturas de pilha e de fila são muito importantes na Ciência da Computação. Muitos problemas computacionais podem ser tratados como operações de inclusão e exclusão nestas estruturas.

A estrutura de pilha

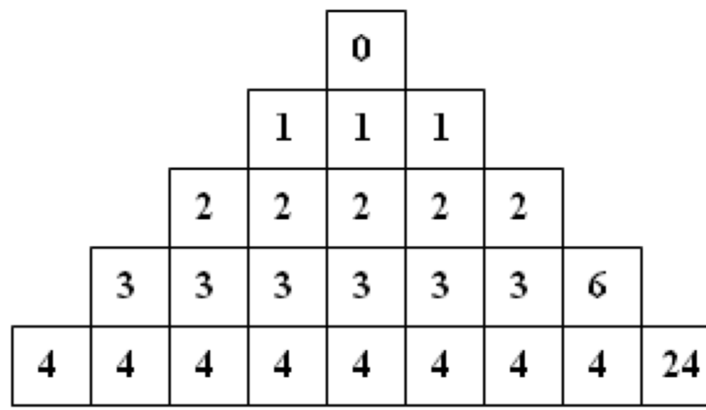
- A pilha é uma das estruturas de dados mais importantes na Ciência da Computação. Imagine, por exemplo, que um programa contém três funções: A, B e C. Considere que a função A chama B e B chama a função C.
- Evidentemente, a função B não pode terminar seu processamento enquanto a função C não retornar. Analogamente, A não pode terminar enquanto B não retornar de seu processamento. Portanto, as funções A, B e C têm a propriedade “a última função que começou sua execução será a primeira função a terminar seu processamento”.
- Como cada função necessita de seus próprios dados, esses dados devem ser alocados em uma estrutura que apresenta esta mesma propriedade, ou seja, em uma **pilha**.

A estrutura de pilha



- Numa pilha de chamadas de funções não importa se as áreas de dados colocadas na pilha vêm de funções diferentes ou de chamadas recursivas de uma mesma função.
- Exemplo: fatorial(4)

```
int fatorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * fatorial(n-1);
}
```



A estrutura de pilha

- Portanto, para qualquer problema em que se aplique a propriedade "o último a entrar é o primeiro a sair", a **pilha** é a estrutura de dados adequada.
- Uma pilha de, no máximo, N elementos pode ser implementada como um vetor p[0..N-1].
- A parte efetivamente ocupada pela pilha é p[0..n-1], onde n é o número de elementos da pilha (o índice t = n-1 corresponde ao **topo da pilha**). A pilha estará **vazia** se n = 0 e a pilha estará **cheia** se n = N.
- Algoritmo POP:

```
n = n - 1;
x = p[n];
```

Algoritmo PUSH:

```
p[n] = x;
n = n + 1;
```

- Cuidados especiais: pilha vazia (POP) ou cheia (PUSH).

A estrutura de pilha

- No entanto, sendo a pilha uma estrutura de dados dinâmica (sujeita a muitas operações de inclusão e exclusão), a lista encadeada é uma forma conveniente de implementação.
- Como um tipo abstrato de dados (TAD), uma pilha **p** é vista como uma sequência ordenada de elementos, sendo **p[0]** o primeiro elemento e **p[topo]**, o último (que corresponde ao topo da pilha). Em termos abstratos, é sempre possível incluir um novo elemento na pilha, ou seja, não há limite para o tamanho da pilha.
- Operações básicas deste TAD:

  empty(): verifica se a pilha está vazia ou não.

  push(): empilhar.

  pop(): desempilhar.

A estrutura de pilha

- Outras operações:
  - **clear()**, para excluir todos os elementos da pilha;
  - **elementoTopo()**, para retornar o elemento que está no topo da pilha sem excluí-lo.
- Uma aplicação importante de pilhas: avaliação de expressões aritméticas escritas na forma **pósfixa** (também conhecida como **notação polonesa**).
- Normalmente, escrevemos uma expressão aritmética usando a notação **infixa**, em que os operadores aparecem entre seus operandos. Por exemplo: A + B.
- Na notação **pósfixa**, o operador aparece após seus operandos. Portanto, para indicar a aplicação do operador + sobre os operandos A e B, escrevemos A B +.

A estrutura de pilha

- A **notação infixa** requer o conhecimento sobre a precedência dos operadores. Por exemplo: A + B * C, qual das operações deve ser feita em primeiro lugar?
- Normalmente, existe uma precedência estabelecida para as operações e, para quebrar a precedência, é preciso usar parênteses: (A + B) * C.
- Para a **notação pósfixa**, os parênteses são desnecessários. Exemplo: A B + C *.
- Outros exemplos:

Forma infixa	Forma pósfixa
A + (B * C)	A B C * +
(A + B) / (C – D)	A B + C D – /
A – B / (C + D * E)	A B C D E * + / –
A + B * C – D + E / F / (G + H)	A B C * D E F G H + / / + – +
((A + B) * C – (D / E)) * (F + G)	A B + C * D E / – F G + *

A estrutura de pilha

- A avaliação de expressões aritméticas escritas na forma pósfixa pode ser feita facilmente examinando-se a expressão da esquerda para a direita, pois nesta notação, os operadores aparecem na ordem em que devem ser executados (notar que isso não ocorre na notação infixa).
- Algoritmo para avaliação de expressões na forma posfixa (usa uma **pilha**):
  - Examinar a expressão da esquerda para a direita;
  - Toda vez que for encontrado um operando, **empilhar**;
  - Se for encontrado um operador, **desempilhar** os seus operandos (é preciso saber quantos operandos tem cada operador), aplicar o operador a estes operandos e empilhar o resultado da operação.

A estrutura de pilha

- **Exemplo**: 5 3 + 2 * 4 2 / – 6 3 + *

Símbolo	Operando2	Operando1	Resultado	Pilha (topo à direita)
5				5
3				5, 3
+	3	5	8	8
2				8, 2
*	2	8	16	16
4				16, 4
2				16, 4, 2
/	2	4	2	16, 2
–	2	16	14	14
6				14, 6
3				14, 6, 3
+	3	6	9	14, 9
*	9	14	126	126

- Ao final, o resultado da avaliação da expressão encontra-se no topo da pilha.

A estrutura de pilha

- E como converter para a forma pósfixa?
- Para converter da forma infixa para a forma pósfixa:
  - converter em primeiro lugar as operações de maior precedência,
  - tratar como um único operando uma parte da expressão já convertida para a forma pósfixa.
- Neste processo, as expressões entre parênteses mais internos precisam ser primeiro convertidas em pósfixas. O último par de parênteses dentro de um grupo de parênteses encerra a primeira expressão a ser convertida.
- Este comportamento "última expressão do grupo, primeira a ser convertida" sugere que o processo de conversão pode fazer uso de uma pilha.

A estrutura de pilha

- Para construir um algoritmo de conversão devemos notar que a ordem dos operandos na expressão é a mesma tanto na forma infixa como na forma posfixa.
- A questão é como inserir os operadores para obter a forma posfixa, o que depende da precedência dos operadores.
- Seja a função **precede(op1, op2)**:
  - precede(op1, op2) = 1, se **op1** tem precedência maior do que **op2**
  - precede(op1, op2) = 0, caso contrário.
- Os operadores devem ser empilhados de tal forma que, em qualquer instante, a partir do topo, os operadores presentes na pilha estão em ordem descrescente de precedência (ou seja, o operador de maior precedência está no topo da pilha).

A estrutura de pilha

- Se um operador **op** é lido e o símbolo presente no topo da pilha tem precedência maior do que **op**, os símbolos da pilha devem ser desempilhados (e incluídos na expressão pósfixa) até que o operador **op** possa ser empilhado.
- Os abre-parênteses também devem ser empilhados. A precedência dos "(", no entanto, deve ser a mais baixa de todos. Para isso, devemos fazer **precede("(", op) = precede(op, "(") = 0**, para todo **op** $\neq$ ")".
- Quando um fecha-parênteses for encontrado, todos os operadores até um "(" presentes na pilha devem ser desempilhados e incluídos na expressão pósfixa. O abre-parênteses deve ser removido da pilha e descartado. Para isso, devemos fazer **precede(op, ")") = 1**, para todo **op** $\neq$ "(".

A estrutura de pilha

- Algoritmo de conversão:

Nesta implementação, considera-se que os operandos são **inteiros de um único dígito** e que os operadores possíveis são: “+”,“–”,“*” e “/”.

```
void Converter(char infixa[], char posfixa[])
{
  int i,j;
  char simb,opTopo;

  j = 0;
  topo = NULL;

  for (i = 0; (simb = infixa[i]) != '\0'; i++)
  {
    if (umOperando(simb))
    {
      posfixa[j] = simb;
      j++;
    }
```

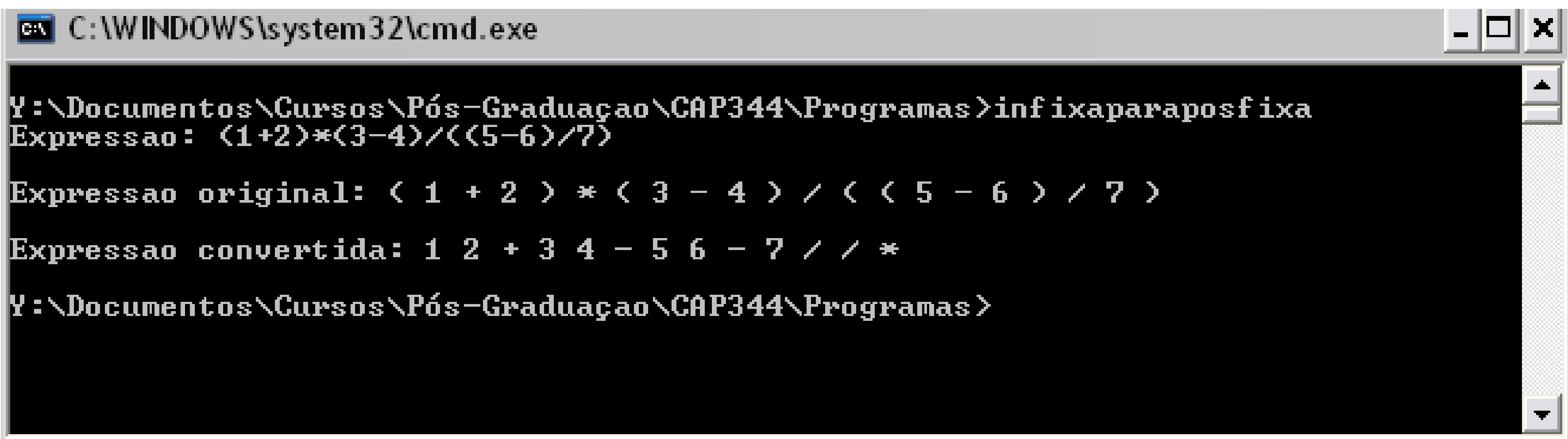
A estrutura de pilha

```
    else
    {
      if (topo != NULL)
      {
        opTopo = pop();
        while (precede(opTopo,simb))
        {
          posfixa[j] = opTopo;
          j++;
          opTopo = pop();
        }
        push(opTopo);
      }
      if (simb != ')')
        push(simb);
      else
        opTopo = pop();
    }
  }
```

A estrutura de pilha

```
  while (topo != NULL)
  {
    posfixa[j] = pop();
    j++;
  }
  posfixa[j] = '\0';
  return;
}
```

- Uma execução:



A estrutura de fila

- Como as pilhas, as filas também têm um papel muito importante na Ciência da Computação. Aplicações envolvendo filas são muito comuns em situações nas quais é preciso “esperar sua vez” para ter acesso a algum serviço.
- Exemplo: filas de processos para serem executados pelo sistema operacional. Normalmente, estes processos correspondem a tarefas que competem entre si por um determinado recurso, como tarefas esperando para serem impressas ou tarefas esperando por uma fatia de tempo do processador.
- Em alguns casos, existe uma ordem de prioridade para atendimento aos elementos de uma fila (fila de prioridades).

A estrutura de fila

- Como no caso da pilha, uma fila também pode ser implementada como um vetor.
- Exemplo: uma fila de, no máximo, N elementos pode ser implementada como um vetor f[0..N-1], em que a parte do vetor efetivamente ocupada pela fila é f[m..n-1], com $0 \leq m \leq n \leq N$. Neste caso, o primeiro elemento da fila está na posição **m** e o último elemento, na posição **n-1**. A fila estará **vazia** se m = n e estará **cheia** se n = N.
- Algoritmo de exclusão:

```
x = f[m];
m = m + 1;
```

Algoritmo de inclusão:

```
f[n] = x;
n = n + 1;
```

- Cuidados especiais: fila vazia e fila cheia. Além disso, movimentações de dados são necessárias para “arrumar” a fila, de modo que a capacidade da fila seja sempre N.

A estrutura de fila

- Sendo uma estrutura de dados dinâmica, é mais conveniente implementar a fila como uma lista encadeada.
- Como um TAD, uma fila **f** é vista como uma sequência ordenada de elementos, sendo **f[inicio]** o primeiro elemento da sequência e **f[final-1]**, o último elemento da sequência. Em termos abstratos, é sempre possível incluir um novo elemento na fila (não há limite para o tamanho da fila).
- Operações básicas:

  empty(): verifica se a fila está vazia ou não.

  incluir(): inclui um novo elemento no fim da fila.

  excluir(): exclui o elemento do início da fila.

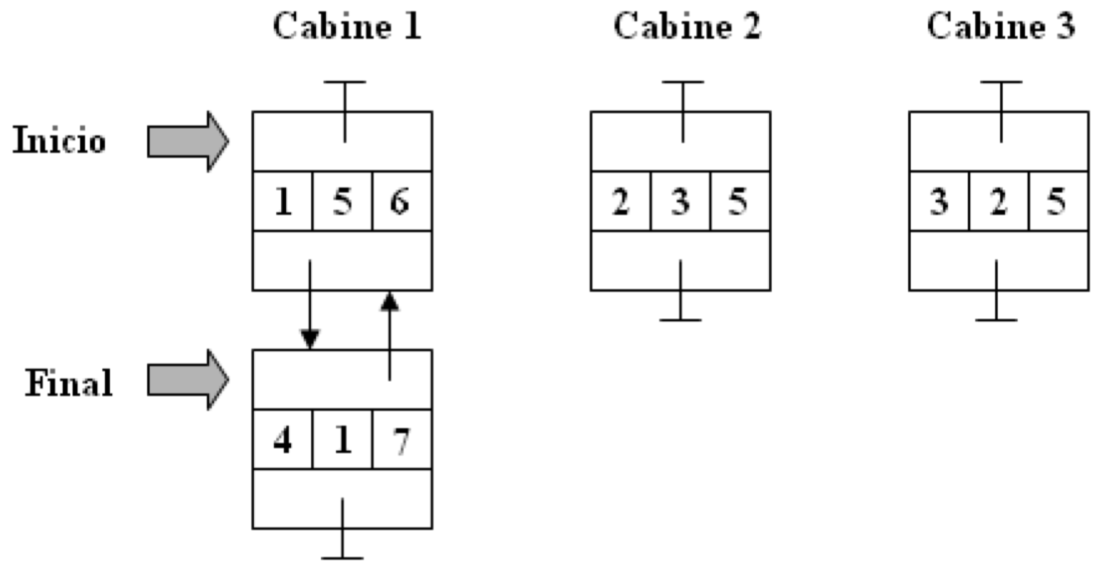
A estrutura de fila

- Outras operações:
  - **clear()**, para excluir todos os elementos da fila;
  - **full()**, para verificar se a fila está cheia;
  - **primeiro()**, para retornar o primeiro elemento da fila, sem excluí-lo;
  - **ultimo()**, para retornar o último elemento da fila, sem excluí-lo.
- Uma aplicação interessante de filas: simulação.

  Exemplo: uma praça de pedágio com 3 cabines. Imagine que um motorista, ao chegar no pedágio, escolhe sempre a cabine com a menor fila. Considere que este pedágio recebe um veículo a cada 1 minuto e que o tempo de atendimento, em minutos, é um inteiro aleatório uniformemente distribuído no intervalo [1, 5].

A estrutura de fila

- Considere que a fila de cada cabine é implementada como uma lista duplamente encadeada (cada lista mantém seus próprios ponteiros de início e final).
- Considere que cada célula destas listas contém: o número do veículo (que corresponde ao instante em que o veículo chega no pedágio), o tempo de atendimento (TA), gerado aleatoriamente, e o instante em que o veículo é servido e sai da fila.



A estrutura de fila

- Observe que o instante de saída de um veículo depende se existe ou não um veículo à frente na fila. Assim, o instante de saída de um veículo deve ser calculado como:
  - saída = chegada + TA, se não existe veículo à frente;
  - saída = máx(chegada, saída do veículo à frente) + TA, se existe veículo à frente.
- A lista, portanto, deve ser **duplamente encadeada**:
  - ponteiro para o próximo nó, para permitir a inclusão de novos nós na fila;
  - ponteiro para o nó anterior, para facilitar o cálculo do instante de saída.

Ver **programa de simulação** no livro:

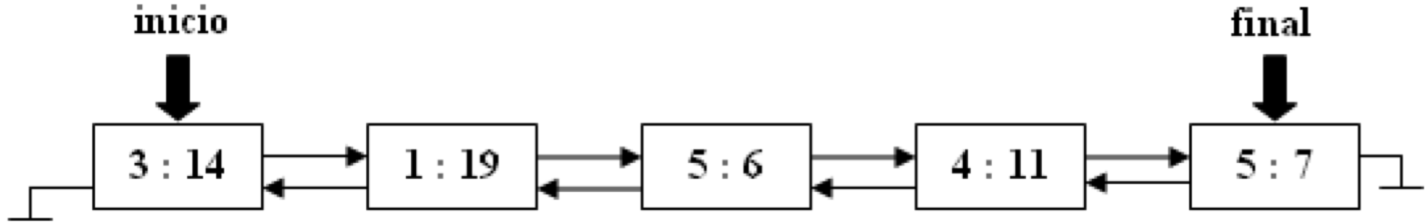
SENNE, E.L.F. *Primeiro Curso de Programação em C*, 3. ed., Florianópolis: Visual Books, 2009. p. 288-290.

A estrutura de fila

- Em uma fila, o primeiro elemento a ser servido é o primeiro elemento incluído na estrututa. Na pilha é o contrário: o primeiro elemento a ser servido é o último elemento incluído.
- Portanto, podemos imaginar que, tanto na fila como na pilha, a prioridade de atendimento dos elementos é dada pela ordem de inclusão dos elementos na estrutura:
  - pilha: **fila de prioridade descendente**, em que o primeiro elemento a ser servido é o que possui o maior valor do instante de inclusão (ou seja, o último a ser incluído).
  - fila: **fila de prioridade ascendente**, em que o primeiro elemento a ser servido é o que possui o menor valor do instante de inclusão (ou seja, o primeiro a ser incluído).

A estrutura de fila

- Numa **fila de prioridade** em geral, a prioridade de cada elemento é estabelecida por um atributo qualquer, independente do instante em que o elemento foi inserido na estrutura.
- **Exemplo**. Uma fila de processos para serem executados, em que cada elemento da fila possui os atributos:
  - um valor de prioridade (por exemplo, de 1 a 5);
  - o tempo necessário de processamento.



Ver **programa de simulação** no livro:

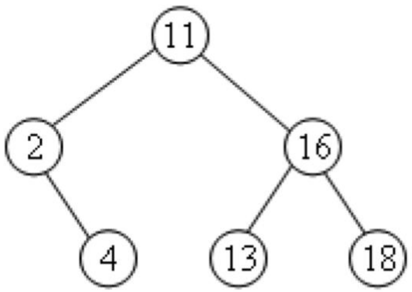
SENNE, E.L.F. *Primeiro Curso de Programação em C*, 3. ed., Florianópolis: Visual Books, 2009. p. 293-296.

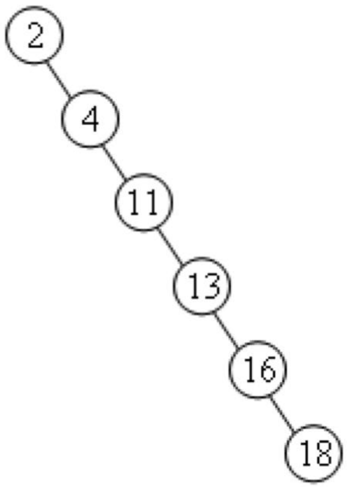
Árvores binárias de procura balanceadas

- Como vimos, em determinadas situações é conveniente estruturar os dados na forma de árvores.

  Exemplo: a busca pode ser realizada mais eficientemente em uma árvore binária (algoritmo O(log n)) do que em uma lista encadeada (algoritmo O(n)).

- Mas essa vantagem depende da árvore. Sejam as árvores binárias de procura contendo a mesma informação:





A árvore da esquerda é melhor do que a da direita. Por que? Quantos testes são necessários na primeira e na segunda árvores, para localizar uma chave de procura, no pior caso?

Árvores binárias de procura balanceadas

- O problema com a segunda árvore (que é equivalente a uma lista encadeada) é o desbalanceamento: a subárvore direita de qualquer nó é muito maior do que a subárvore esquerda correspondente.
- Uma árvore binária é **balanceada** se a diferença nas alturas das subárvores esquerda e direita de qualquer nó da árvore é menor ou igual a 1.
- Para o processo de busca é importante que a árvore seja balanceada.

  Exemplo: uma árvore binária perfeitamente balanceada contendo 30000 nós. Qual será a altura *h* desta árvore?Quantos testes serão necessários no pior caso?

  E se for uma lista encadeada com 30000 nós? Quantos testes serão necessários no pior caso?

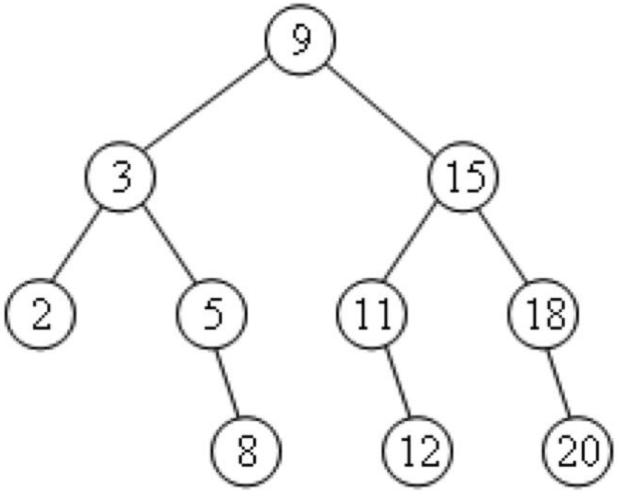
Árvores binárias de procura balanceadas

- Uma forma simples (mas ineficiente) de construir uma árvore binária de procura balanceada é ordenar os dados que a árvore deve conter e escolher (recursivamente) como raiz de cada subárvore o valor que estiver na posição mais central possível na ordenação.
- Algoritmo:

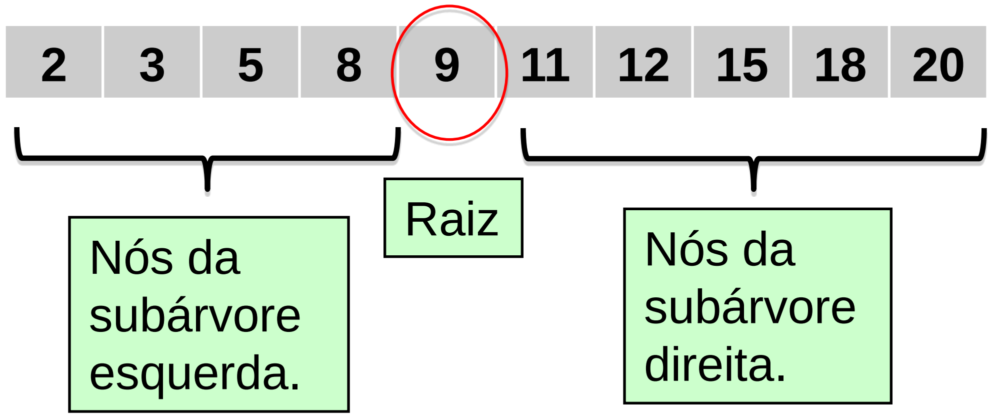
```
void ConstruirArvoreBalanceada(int dados[], int ini, int fim)
{
  int med;

  if (ini <= fim)
  {
    med = (ini + fim)/2;
    IncluirNaArvore(dados[med]);
    ConstruirArvoreBalanceada(dados,ini,med-1);
    ConstruirArvoreBalanceada(dados,med+1,fim);
  }
}
```

Árvores binárias de procura balanceadas

- Exemplo: árvore binária resultante da aplicação deste algoritmo ao conjunto: {20, 12, 5, 2, 8, 15, 11, 9, 3, 18}.



Inicialmente:



- Inconvenientes deste algoritmo:
  - Os dados que a árvore contém devem ser conhecidos a priori;
  - A ordenação desses dados.
- E se a árvore tiver que ser construída à medida que os dados forem chegando?

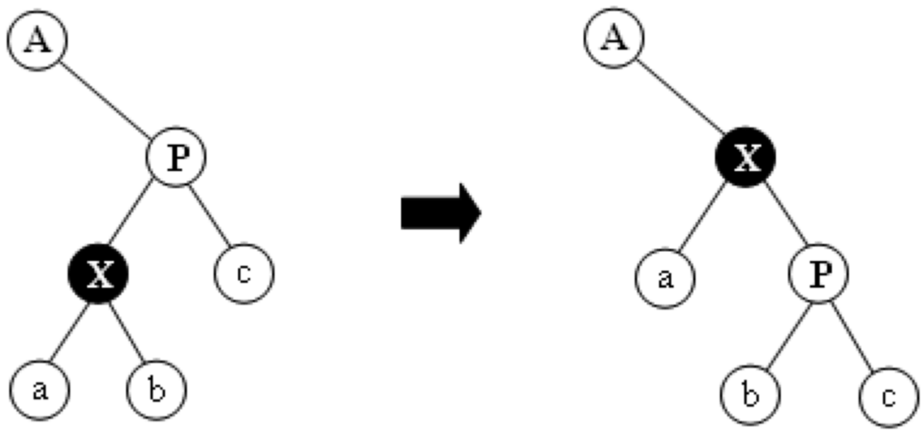
Árvores binárias de procura balanceadas

- Uma possibilidade:
  - transferir os dados da árvore (possivelmente desbalanceada) para um vetor por meio da **travessia em ordem** (o que garante a ordenação dos dados);
  - destruir a árvore atual;
  - reconstruir a árvore usando o algoritmo.
- Este processo pode ser muito oneroso para árvores grandes.
- Vamos discutir dois algoritmos mais eficientes para balanceamento de árvores binárias de procura:
  - O algoritmo DSW (Day, Stout e Warren)
  - O algoritmo AVL (Adelson-Velsky e Landis)

O algoritmo DSW

- O algoritmo DSW baseia-se na transformação de árvores binárias por meio da operação de rotação.
- Existem dois tipos de rotação (simétricos entre si): rotação à esquerda e rotação à direita.

  Exemplo:



- A figura mostra a operação de **rotação à direita** do nó **X** em relação ao nó **P**. Observe que, após a operação de rotação, a árvore resultante continua sendo uma árvore binária de procura.

O algoritmo DSW

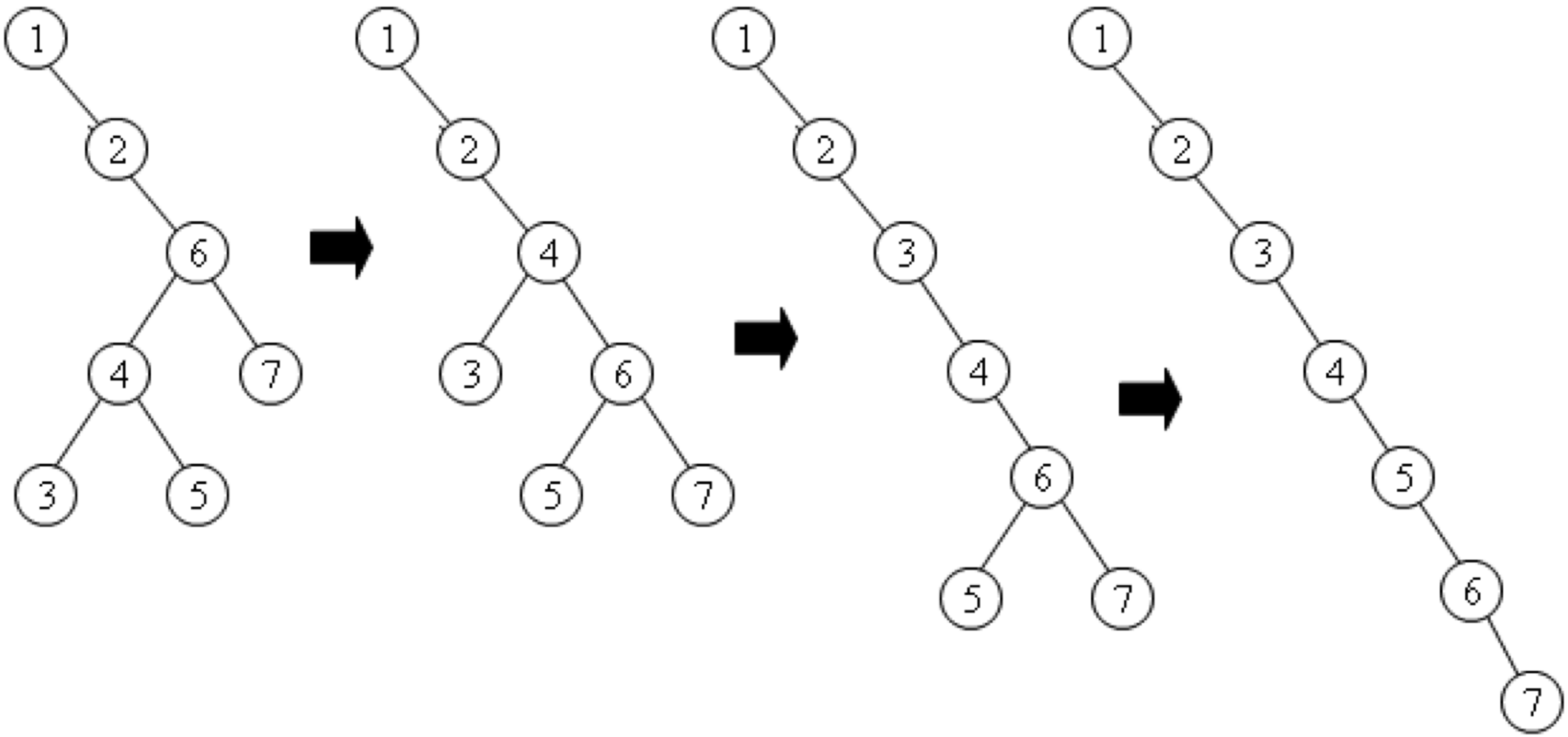
- O balanceamento de uma árvore binária de procura com o algoritmo DSW se dá em duas etapas:
  - 1) transformar a árvore binária em uma lista encadeada;
  - 2) transformar a lista encadeada em uma árvore binária balanceada.

- Em ambas as etapas, o algoritmo utiliza a operação de rotação:
  - Na etapa 1, a rotação à direita é usada repetidamente até que a árvore original se transforme em uma lista.
  - Na etapa 2, a rotação à esquerda é usada repetidamente. A cada aplicação desta operação a árvore (originalmente, uma lista) vai se tornando cada vez mais balanceada.

O algoritmo DSW

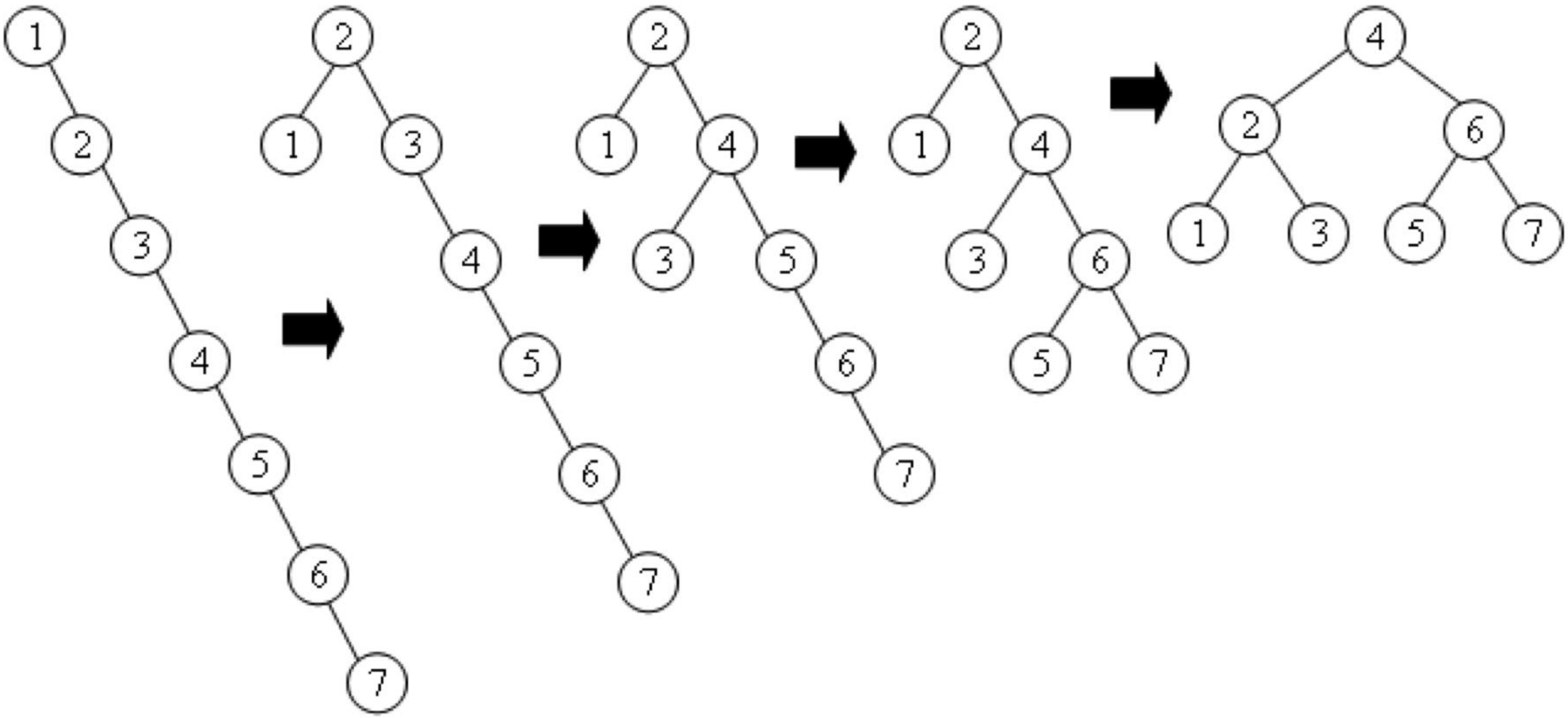
Exemplo:

- Etapa 1: Transformação da árvore original (desbalanceada) em uma lista.



O algoritmo DSW

- Etapa 2: Transformação da lista em uma árvore binária de procura balanceada.



O algoritmo DSW

Algoritmo:

Nesta implementação, presume-se que a árvore contém **N** nós e que o ponteiro **raiz** aponta para a raiz da árvore.

```
void AlgoritmoDSW()
{
  TransformarArvoreEmLista();
  TransformaListaEmArvoreBalanceada(N);
}
```

```
void RotacaoADireita(arvore *A, arvore *P, arvore *X)
{
  if (A != NULL)
  {
    A->direita = X;
    P->esquerda = X->direita;
    X->direita = P;
  }
}
```

O algoritmo DSW

```
void TransformarArvoreEmLista()
{
  arvore *avo,*pai,*filho;

  avo = NULL;
  pai = raiz;
  while (pai != NULL)
  {
    if (pai->esquerda != NULL)
    {
      filho = pai->esquerda;
      RotacaoADireita(avo,pai,filho);
      pai = avo->direita;
    }
    else
    {
      avo = pai;
      pai = pai->direita;
    }
  }
}
```

O algoritmo DSW

```
void RotacoesAEsquerda(int n)
{
  int i;
  arvore *pai,*filho;

  pai = pseudoRaiz;
  for (i = 0; i < n; i++)
  {
    filho = pai->direita;
    pai->direita = filho->direita;
    pai = pai->direita;
    filho->direita = pai->esquerda;
    pai->esquerda = filho;
  }
}
```

Notar que para as operações de **rotação à esquerda** considera-se uma pseudo-raiz para a árvore: um nó sem filho à esquerda e cujo filho à direita é a raiz da árvore.

O algoritmo DSW

```
void TransformaListaEmArvoreBalanceada(int N)
{
  int k,m,n;

  pseudoRaiz = calloc(1,sizeof(arvore));
  pseudoRaiz->info = 0;
  pseudoRaiz->esquerda = NULL;
  pseudoRaiz->direita = raiz;

  n = N;
  k = pow(2,(int)(log2(n+1)))-1;
  m = n - k;
  RotacoesAEsquerda(m);
  n = n - m;
  while (n > 1)
  {
    n = n/2;
    RotacoesAEsquerda(n);
  }
  raiz = pseudoRaiz->direita;
}
```

Criação da pseudo-raiz da árvore.

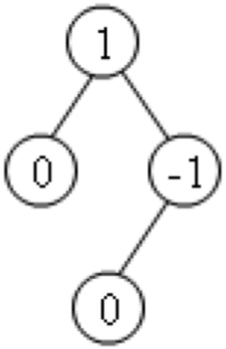
O algoritmo AVL

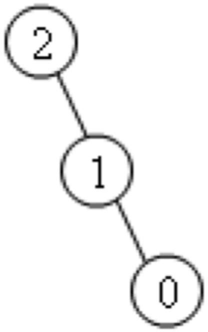
- O algoritmo DSW exige a reconstrução completa da árvore original. Isto, no entanto, não é necessário se apenas uma parte da árvore apresenta desbalanceamento.
- A ideia do algoritmo AVL é modificar apenas uma parte da árvore, dependendo dos fatores de balanceamento dos nós da árvore.
- O **fator de balanceamento** de um nó é dado pela diferença entre as alturas de suas subárvores direita e esquerda. Portanto, uma árvore está balanceada se os fatores de balanceamento de todos seus nós são -1, 0 ou 1.

Exemplo:

fatores de

balanceamento

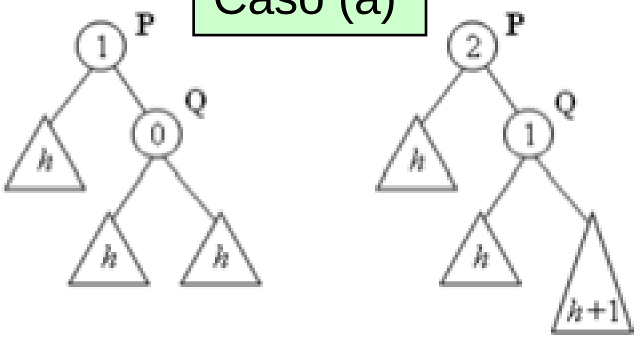




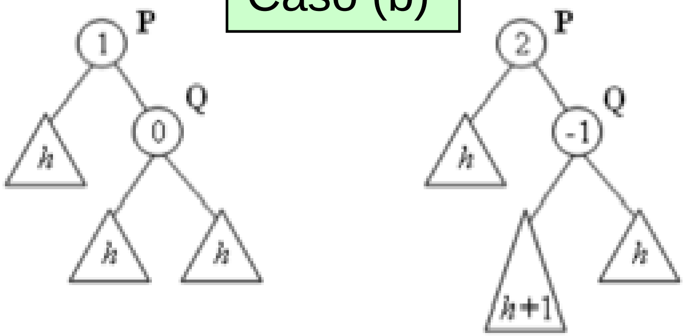
O algoritmo AVL

- Se o fator de balanceamento de um nó qualquer de uma árvore for menor do que -1 ou maior do que 1, a árvore encontra-se desbalanceada.
- Uma árvore balanceada pode tornar-se desbalanceada em 2 situações (Na verdade, são 4 situações, simétricas 2 a 2. Portanto, somente 2 precisam ser analisadas):
  - (a) O desbalanceamento resulta da inclusão de um nó na subárvore direita do filho à direita.
  - (b) O desbalanceamento resulta da inclusão de um nó na subárvore esquerda do filho à direita.

Caso (a)

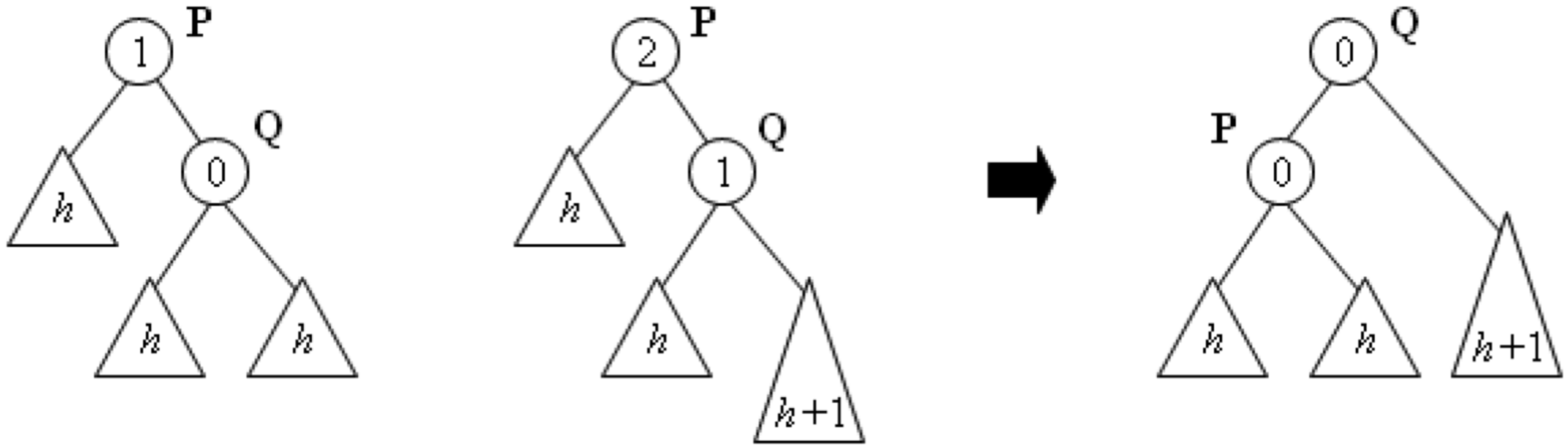


Caso (b)



O algoritmo AVL

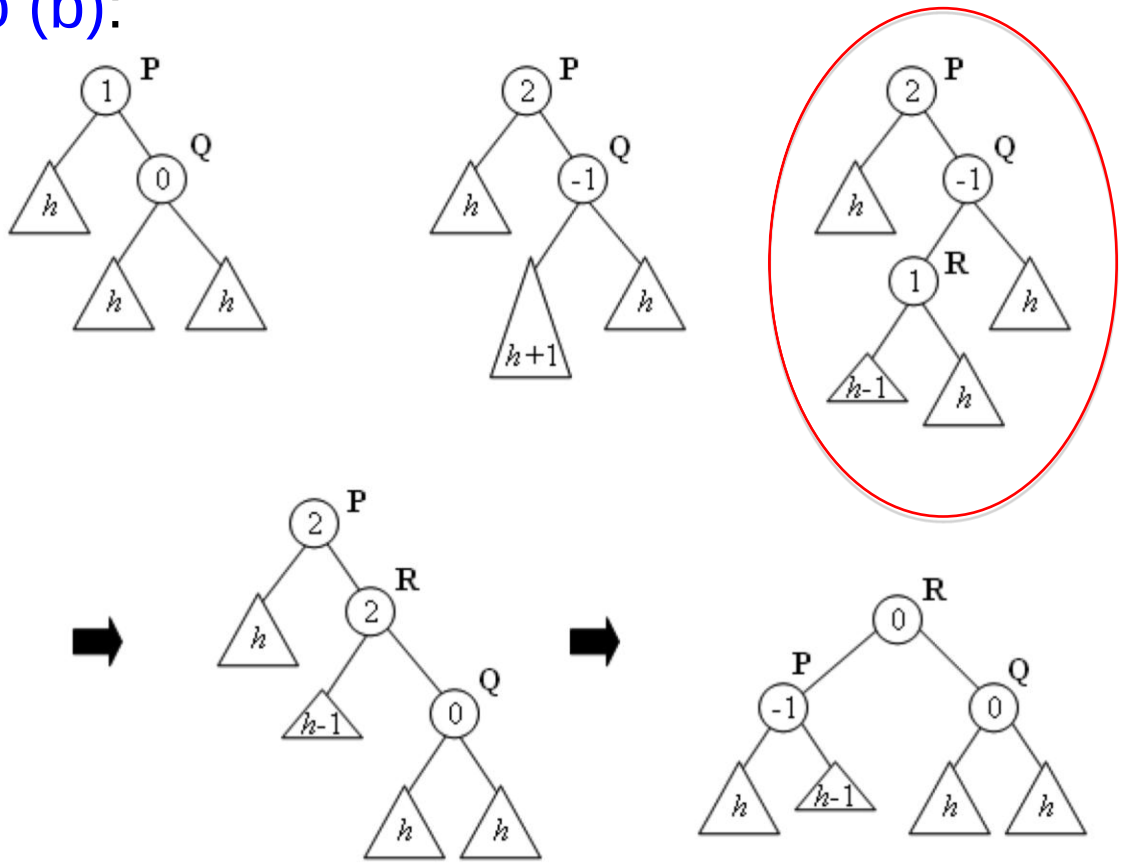
- Caso (a):



Note que o desbalanceamento ocorre no nó P devido à inclusão de um novo nó na subárvore direita de Q. Neste caso, a solução é simples: basta aplicar a operação de rotação à esquerda de Q em relação a P.

O algoritmo AVL

- Caso (b):



Árvore desbalanceada em mais detalhe.

- Neste caso, o balanceamento requer duas rotações:
  - rotação à direita de R em relação a Q, e
  - rotação à esquerda de R em relação a P.

O algoritmo AVL

- Note que nestes dois casos consideramos P como raiz de uma árvore única.

- No entanto, P pode ser parte de uma árvore balanceada maior. Neste caso, após o balanceamento de P não é necessário se preocupar com os predecessores de P, pois a altura da árvore balanceada final nos dois casos é igual à altura da árvore balanceada original (e vale $h + 2$).

- Isso significa que o fator de balanceamento do pai da raiz da árvore balanceada final permanece o mesmo.

- Portanto, as transformações feitas na subárvore P são suficientes para restaurar o balanceamento da árvore inteira.