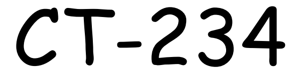
CT-234

Estruturas de Dados, Análise de Algoritmos e Complexidade Estrutural

Carlos Alberto Alonso Sanches



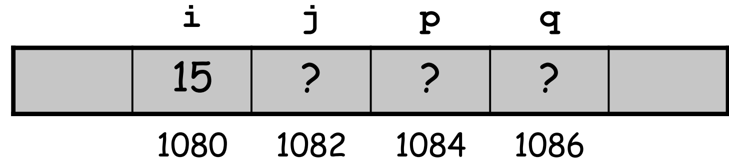
3) Estruturas de dados elementares

Filas, pilhas e árvores

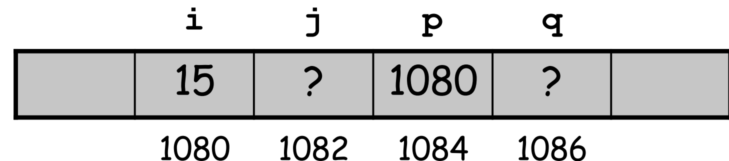
Ponteiros

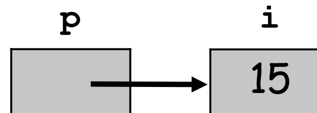
- *Ponteiros* (ou *apontadores*) são variáveis que armazenam endereços, ou seja, apontam para uma posição da memória
- São necessárias quando ocorre alocação dinâmica
- Exemplos:

```
int i = 15, j, *p, *q;
```



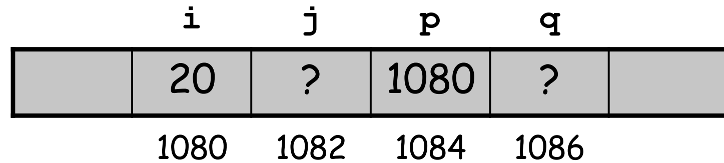

```
p = &i;
```

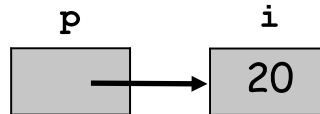




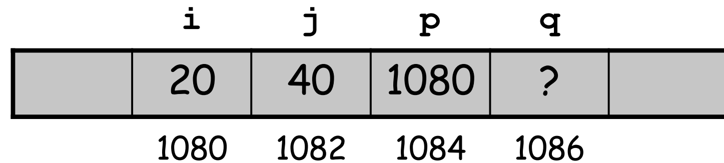
Ponteiros

```
*p = 20;
```

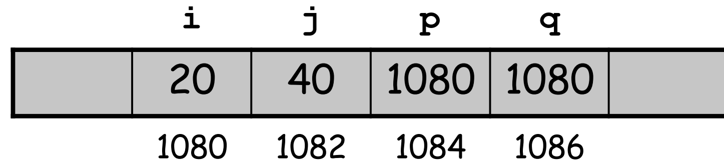


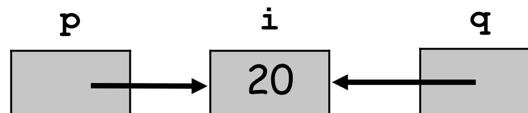



```
j = 2 * *p;
```




```
q = &i;
```



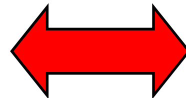


Tipos abstratos de dados

- Os tipos abstratos de dados (TAD), definidos através das suas operações, encapsulam a implementação
- Exemplos de operações abstratas:
  - Inserção
  - Remoção
  - Mínimo ou máximo
  - Sucessor ou predecessor
  - Busca
- As estruturas de dados utilizadas nas implementações dessas operações têm impacto direto no desempenho, tanto no tempo de execução como no consumo de espaço

TADs x implementações x desempenho

Pilhas
Filas
Árvores
Grafos
etc.

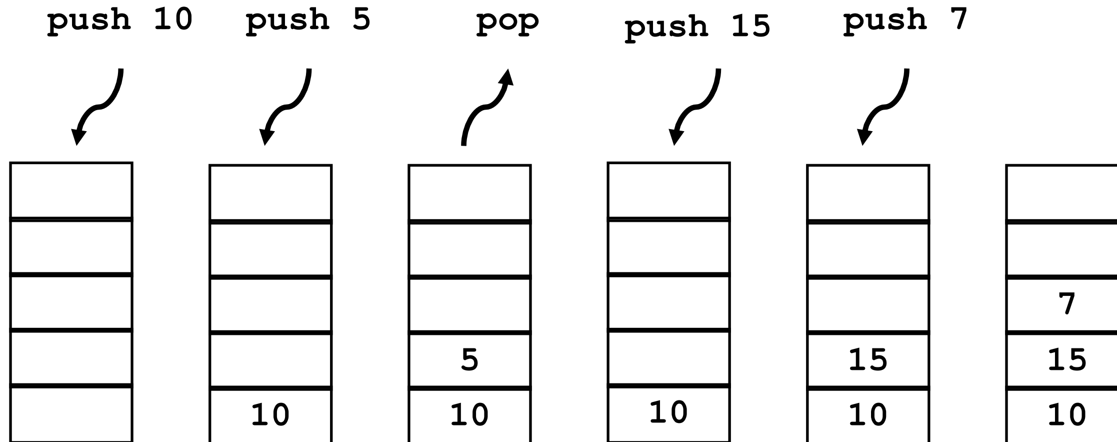


Variáveis indexadas
Ponteiros

Pilhas (*stacks*)

- *Pilhas* são estruturas que permitem acessos em somente uma das suas extremidades
- Por essa razão, uma pilha é chamada de estrutura *LIFO* (*last in/ first out*)
- Operações:
  - push(x): empilha x
  - pop(): desempilha o topo
  - top(): retorna o topo sem desempilhá-lo
  - size(): retorna o tamanho atual da pilha
  - isEmpty(): verifica se a pilha está vazia

Exemplos com pilha



Implementação com vetor

```
size() {
return t+1; }

isEmpty() {
return (t<0); }

top() {
if (isEmpty())
   return Error;
return S[t]; }

push(x) {
if (size() == N)
   return Error;
S[++t]=x; }

pop() {
if (isEmpty())
   return Error;
S[t--]=null; }
```

- Na implementação com vetor, todas as operações anteriores podem ser executadas em tempo $\Theta(1)$
- Caso fosse necessário implementar uma operação de busca na pilha, a solução gastaria tempo de pior caso $\Theta(N)$, ou seja, seria ineficiente...

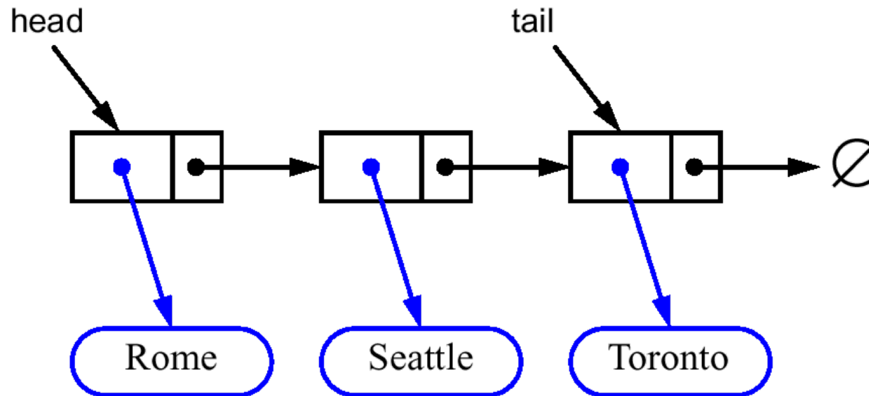
Uma aplicação típica

- Pilhas são utilizadas para verificar o emparelhamento de delimitadores:

```
Leia o próximo caractere ch;
while não é fim de arquivo {
     if ch é '(','[' ou '{'
        push(ch);
     else if ch é ')',']' ou '}' {
             x = top();
             pop();
             if ch e x não se casam
                 erro;
     }
     Leia o próximo caractere ch;
}
if isEmpty()
   sucesso;
else erro;
```

Implementação com lista ligada

- Os elementos da pilha podem estar conectados através de uma cadeia de ponteiros:

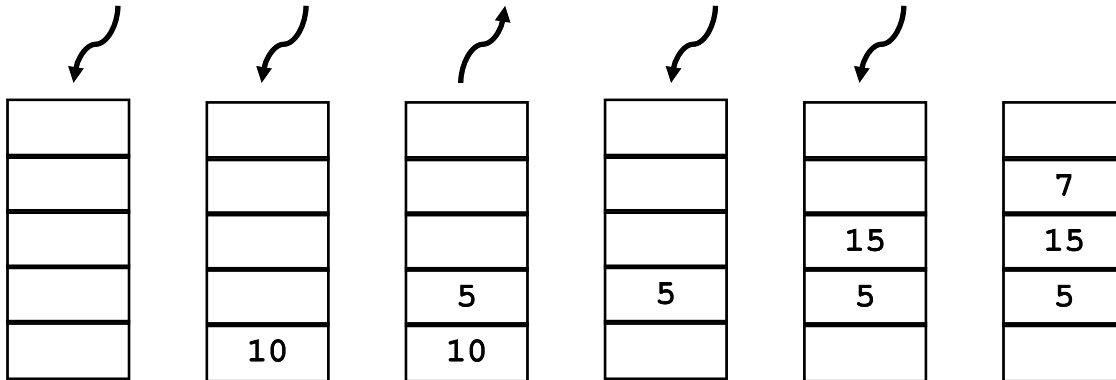


- O topo da pilha seria o nó apontado por *head*
- *tail* também poderia ser utilizado como topo da pilha?

- *Filas* são simples sequências de espera: crescem através do acréscimo de novos elementos no final e diminuem com a saída dos elementos da frente
- Os elementos são acrescentados em uma extremidade e removidos da outra
- Em relação à pilha, a principal diferença é que a fila é uma estrutura *FIFO (first in/first out)*

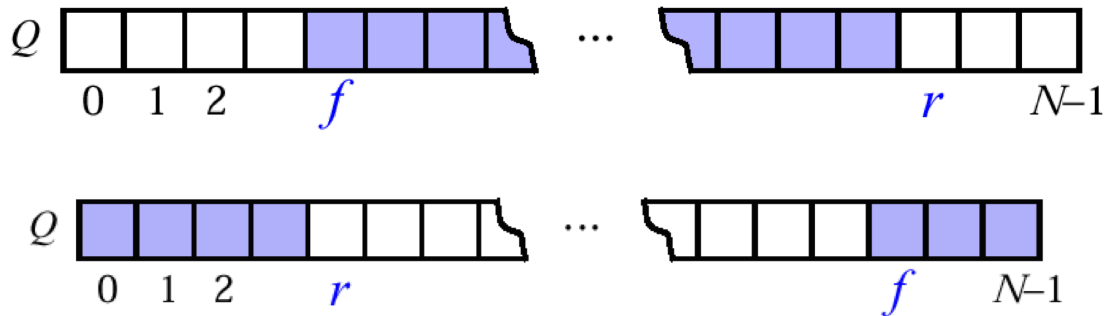
- size(): retorna o tamanho atual da fila
- isEmpty(): verifica se a fila está vazia
- first(): retorna o primeiro elemento da fila sem removê-lo
- dequeue(): retira o primeiro elemento da fila
- enqueue(x): coloca x no final da fila

Exemplos com fila



Implementação com vetor

- Convém utilizar o vetor de maneira circular:



- Nesse caso, o que significaria $f = r$?
- Como diferenciar fila vazia de fila cheia?

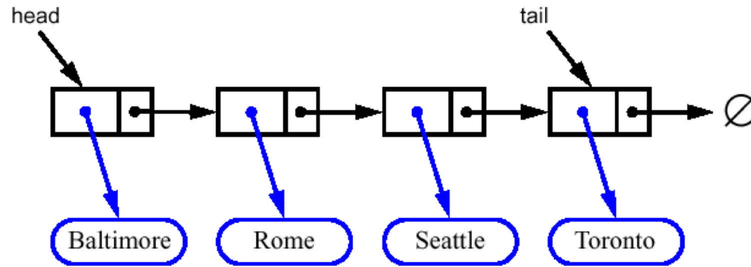
Implementação com vetor

```
size() {
return (N-f+r) % N; }

isEmpty() {
return (f==r); }

first() {
if (isEmpty())
     return Error;
return Q[f]; }
```

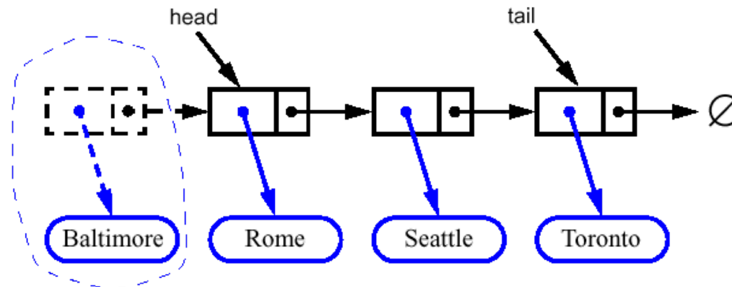
```
dequeue() {
if (isEmpty())
   return Error;
Q[f] = null;
f = (f+1) % N; }

enqueue(x) {
if (size() == N-1)
   return Error;
Q[r] = x;
r = (r+1) % N; }
```

- Na implementação com vetor, todas as as operações anteriores também podem ser executadas em tempo $\Theta(1)$
- Caso fosse necessário uma operação de busca, a solução óbvia gastaria tempo de pior caso $\Theta(N)$, ou seja, seria ineficiente. Seria possível realizá-la em tempo de pior caso $\Theta(\log N)$?

Implementação com lista ligada

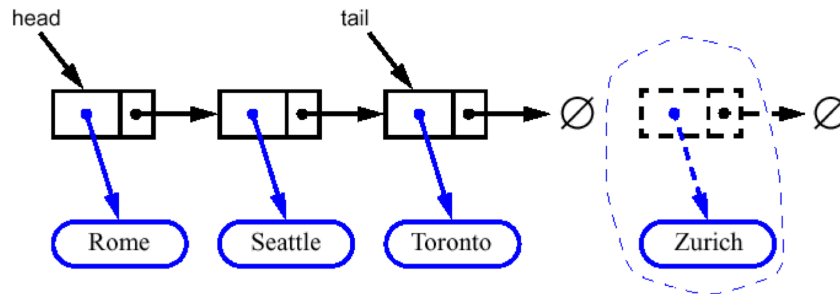


- Dequeue:

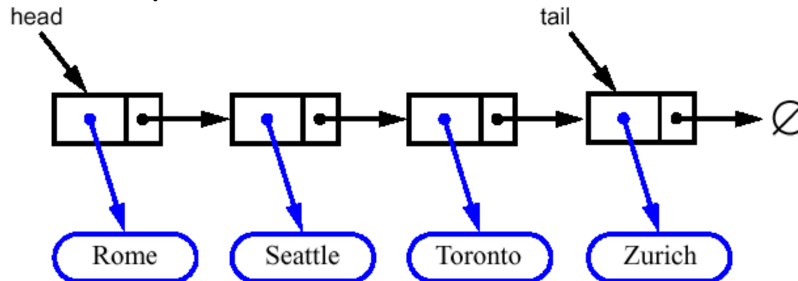


Implementação com lista ligada

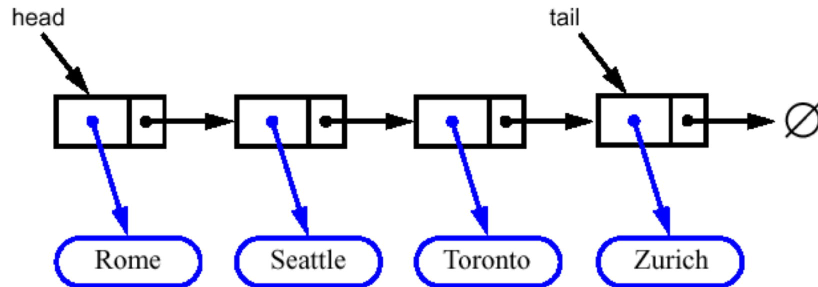
- Enqueue: crie um novo nó



- Adiante o ponteiro *tail* :



Implementação com lista ligada



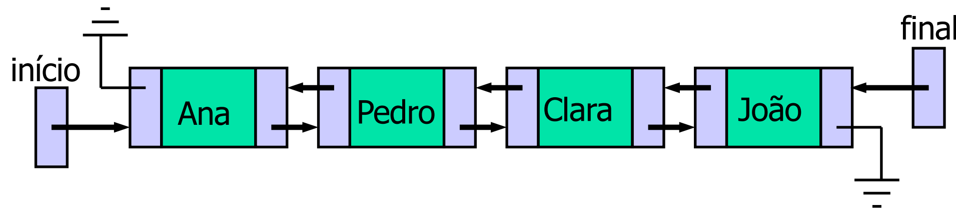
- Exercícios:
  - Numa fila de N elementos, implementada com lista ligada, seria possível realizar remoções no seu final em tempo constante?
  - Nessa mesma estrutura, seria possível realizar uma operação de busca em tempo de pior caso $\Theta(\log N)$?

Filas com duplo-fim (*deques*)

- Filas com duplo-fim (*doubled-ended queue*) permitem inserção e remoção, em tempo constante, tanto no início como no fim
- Operações:
  - insertFirst(x): insere x no início
  - insertLast(x): insere x no final
  - removeFirst(): remove o primeiro elemento
  - removeLast(): remove o último elemento
  - first(): retorna o primeiro elemento
  - last(): retorna o último elemento
  - size(): retorna o tamanho atual
  - isEmpty(): verifica se está vazia
- A *deque* pode simular tanto as operações de uma pilha como as de uma fila

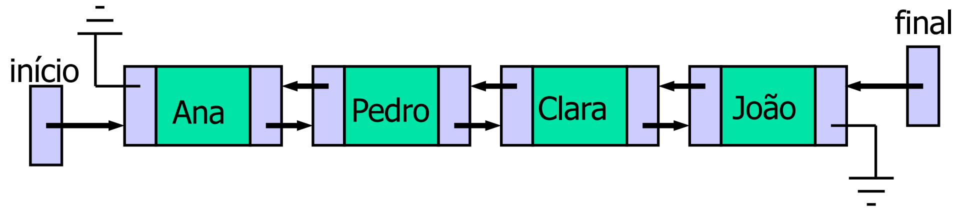
Listas duplamente ligadas

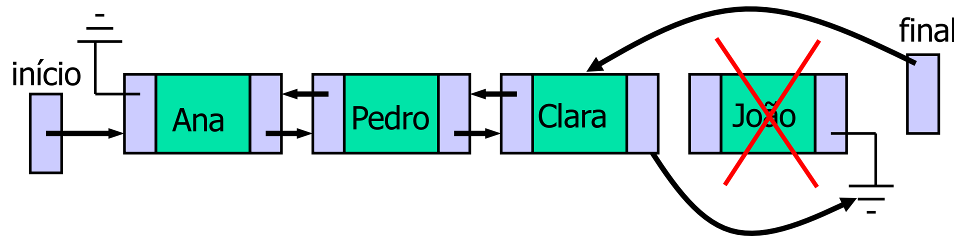
- Podemos implementar uma *deque* através de uma lista duplamente ligada:

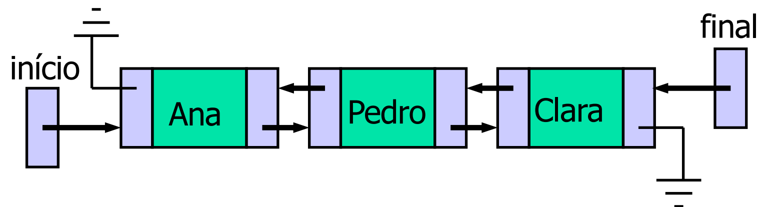


- Cada nó tem dois ponteiros: para o próximo e para o anterior.
- Para cada *deque*, é preciso guardar ponteiros para o seu início e o seu final

Remoção do último elemento







- Seria possível implementar uma *deque* através de um vetor?
- Quais as vantagens e as desvantagens em relação à implementação com listas duplamente ligadas?

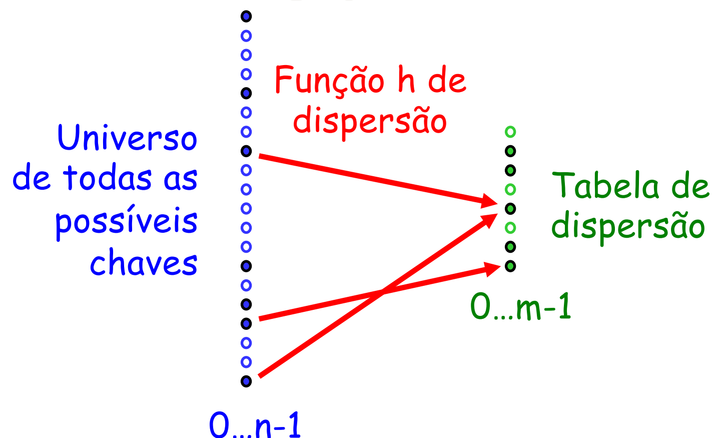
- Escreva um programa que leia uma expressão matemática com ( ), [ ] e { }, e informe se está balanceada
(não é preciso calculá-la, nem preocupar-se com a presença de outros caracteres).

- Escreva um programa para resolver qualquer instância do *Problema de Josephus*.
Faça uma versão que utilize vetor e outra com listas ligadas.

Tabelas de dispersão (*hashing*)

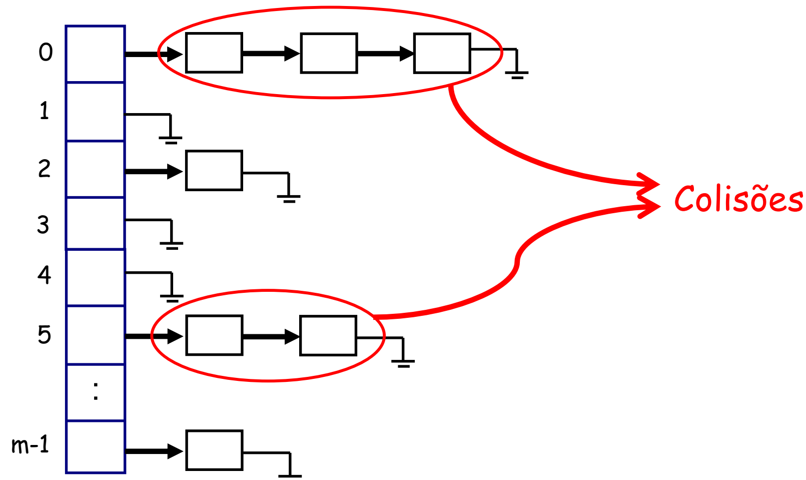
- *Dicionário* é o nome de uma TAD com as seguintes operações:
  - Inserção
  - Remoção
  - Busca através de uma chave
- Os dicionários podem ser implementados eficientemente com tabelas de dispersão, baseadas em listas ligadas (*open hashing*)
- Em condições adequadas, essas três operações podem ser realizadas em tempo constante

Ideia



- Naturalmente, m << n
- h: {0,...,n-1} $\rightarrow$ {0,...,m-1} é calculada em tempo $\Theta(1)$
- Colisão: h(a) = h(b) quando a $\neq$ b
- Na técnica *open hashing*, as colisões são tratadas com listas ligadas

Implementação da tabela



- Evidentemente, nas posições em que ocorrem colisões, a busca e a remoção passam a gastar mais tempo
- Nesses casos, a correspondente lista ligada precisa ser percorrida, nó a nó...

Comentários finais

- Fatores que alteram o desempenho das operações:
  - grau de ocupação da tabela
  - escolha da função de dispersão
- Dada uma chave x, uma conhecida função de dispersão é h(x) = x mod m. Geralmente, escolhe-se um valor de m primo
- Knuth oferece um bom estudo sobre diversas funções de dispersão

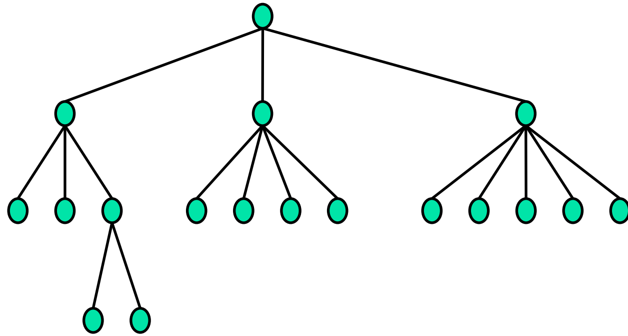
Árvores (*trees*)

- Tantos as pilhas como as filas são estruturas lineares, isto é, de uma única dimensão

- Na sua implementação, as listas ligadas possibilitam maior flexibilidade que os vetores, mas mesmo assim não permitem a representação hierárquica de dados

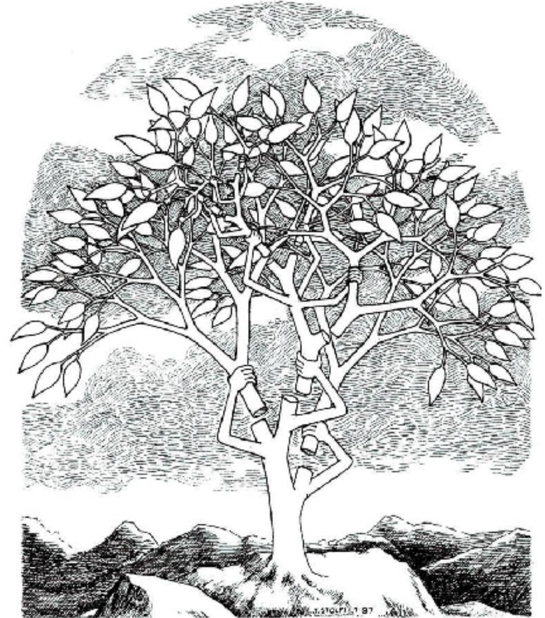
- *Árvores* são estruturas hierárquicas, formadas por vértices e arestas. Ao contrário das árvores naturais, são representadas de cima para baixo: a raiz está no topo e as folhas na base

Árvores

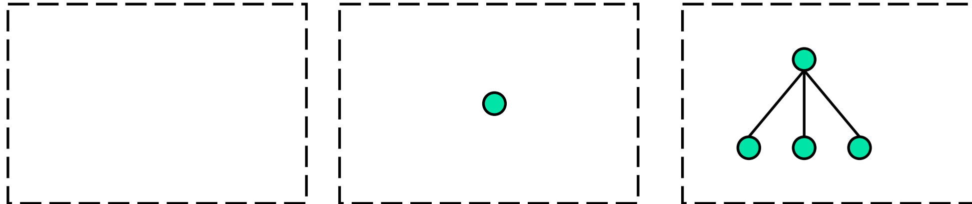


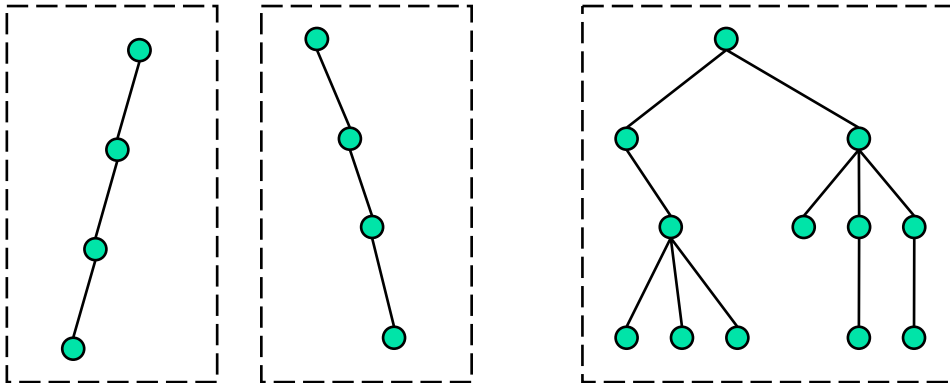
- A raiz é o único nó que não possui ancestrais
- As folhas são os nós sem filhos

- É uma estrutura recursiva: cada filho é também uma árvore



Exemplos de árvores





- Cada nó pode ser alcançado a partir da raiz através de uma sequência única de arestas, chamada de <u>caminho</u>

- O <u>comprimento de um caminho</u> é a quantidade de arestas que ele possui

- O <u>nível de um nó</u> é o comprimento do caminho da raiz até ele

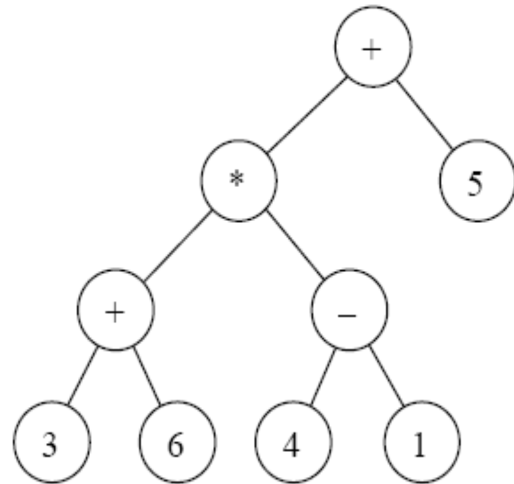
- A <u>altura de uma árvore</u> não vazia é o nível máximo dos nós dessa árvore

- O grau de um nó é o número de seus filhos
- As folhas têm grau nulo e são chamadas de nós terminais
- Um nó que não é folha (isto é, que possui grau não nulo) é chamado de nó interno ou nó não-terminal
- O grau de uma árvore é o máximo entre os graus dos seus nós
- Um conjunto de árvores é chamado de floresta

Algumas características

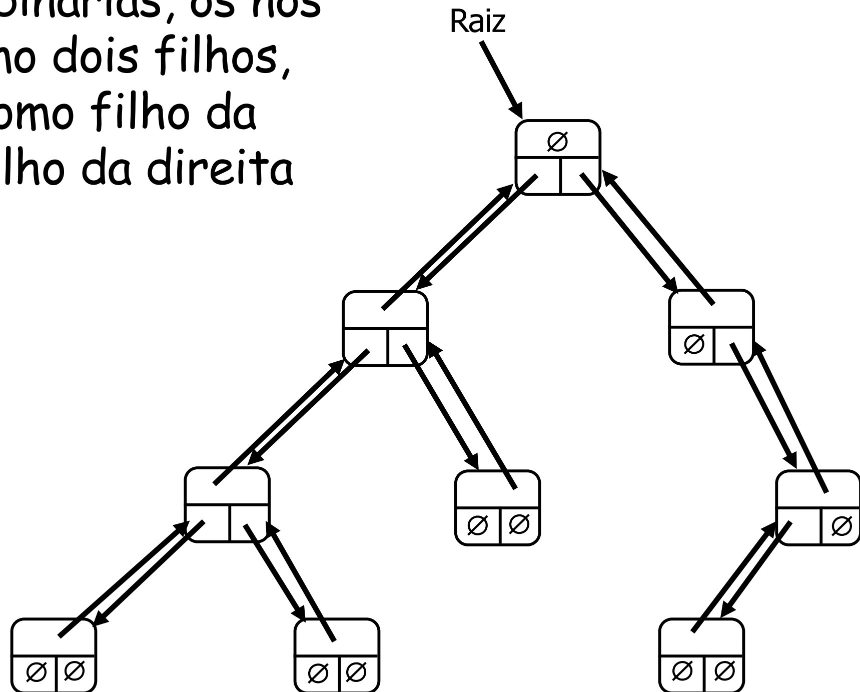
- O número de filhos por nó e as informações armazenadas diferenciam os tipos de árvores

- A árvore ao lado representa a expressão (3+6)*(4-1)+5: as folhas possuem valores e os nós intermediários, operadores matemáticos



Árvores binárias

- Nas árvores binárias, os nós têm no máximo dois filhos, designados como filho da esquerda e filho da direita



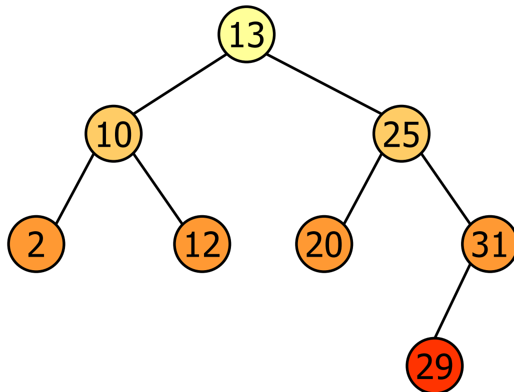
- Seria possível implementar uma árvore binária através de um vetor?

- Percurso em árvores é uma sequência em que cada um dos nós da árvore aparece exatamente uma vez (não precisam estar necessariamente unidos por arestas)
- Uma árvore com *N* nós possui *N!* percursos diferentes
- Mais importantes: *percursos em largura* (ou extensão) e *percursos em profundidade*

Percurso em largura

- O *percurso em largura* começa na raiz e passa ordenadamente por todos os níveis subsequentes, visitando-se primeiro os nós da esquerda e depois os da direita
- É possível implementá-lo com o uso de uma fila:
  - Depois que um nó é visitado, seus filhos são colocados no final dessa fila (primeiro, o da esquerda; depois, o da direita)
  - O próximo nó a ser visitado é o que está no início da fila
  - Desse modo, os nós de nível $i+1$ serão visitados somente depois que todos os nós do nível i já o foram

Exemplo



Fila: 13

Fila: 10, 25

Fila: 25, 2, 12

Fila: 2, 12, 20, 31

Fila: 12, 20, 31

Fila: 20, 31

Fila: 31

Fila: 29

Percurso em largura: 13, 10, 25, 2, 12, 20, 31, 29

```
void BreadthFirst() {
  Queue q;
  Node *p = root;
  if (p != 0) {
    q.enqueue(p);
    while (!q.isEmpty()){
      p = q.dequeue();
      visit(p);
      if (p->left != 0)
        q.enqueue(p->left);
      if (p->right != 0)
        q.enqueue(p->right);
    }
  }
}
```

Percurso em profundidade

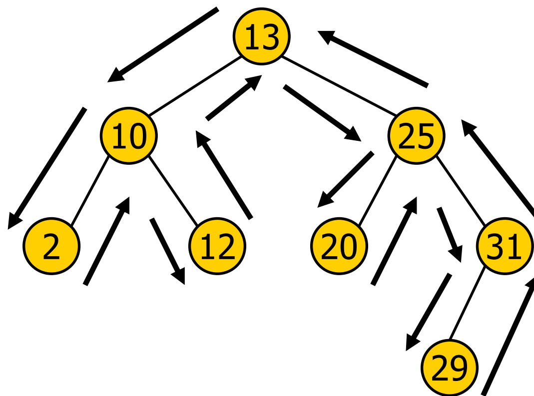
- *O percurso em profundidade* prossegue tanto quanto possível à esquerda (ou à direita). Em seguida, volta ao último nó visitado e recomeça
- Este processo é repetido até que todos os nós sejam visitados
- Pode ser implementado recursivamente ou com uma pilha explícita

Percursos em profundidade à esquerda

- Pré-ordem: raiz, filhos da esquerda, filhos da direita
- In-ordem: filhos da esquerda, raiz, filhos da direita
- Pós-ordem: filhos da esquerda, filhos da direita, raiz

Exemplo

- Percurso pré-ordem à esquerda:



- Qual seria o percurso pré-ordem à direita dessa árvore?

Percurso: 13, 10, 2, 12, 25, 20, 31, 29

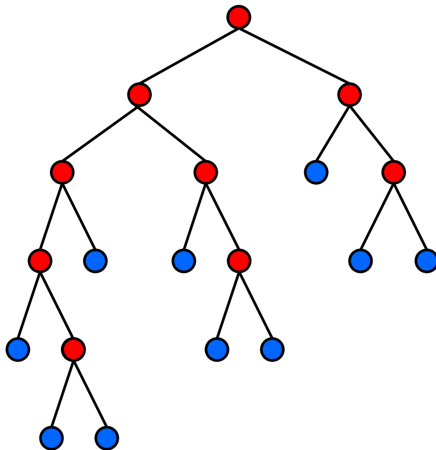
Percursos à esquerda

```
void inorder(Node *p) {
  if (p != 0) {
    inorder(p->left);
    visit(p);
    inorder(p->right);
  }
}

void preorder(Node *p) {
  if (p != 0) {
    visit(p);
    preorder(p->left);
    preorder(p->right);
  }
}
```

```
void postorder(Node *p) {
  if (p != 0) {
    postorder(p->left);
    postorder(p->right);
    visit(p);
  }
}
```

Um caso particular

- Nas árvores binárias não-vazias cujos *nós internos tenham exatamente dois filhos*, sabe-se que f = i + 1, onde f e i são respectivamente o número de folhas e o número de nós internos



i nós internos

f folhas

$f = i + 1$

i+1 nós internos

f+1 folhas

$f+1 = i+1 + 1$

Demonstração

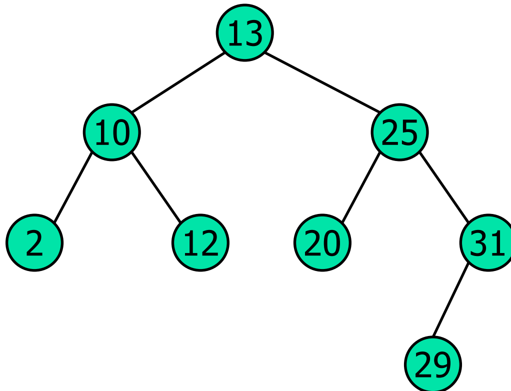
- Como as árvores são estruturas recursivas, suas propriedades geralmente são demonstradas através de indução
- No caso considerado, a argumentação é construtiva, isto é, prova-se para todas as possíveis árvores, de forma incremental no tamanho:
  - Se a árvore é apenas a raiz: $i = 0$, $f = 1$. Portanto, $f = i + 1$
  - Considere uma árvore com a propriedade $f = i + 1$
  - A única maneira dessa árvore crescer, mantendo a propriedade, é anexar duas folhas a uma folha já existente, que se tornará nó interno. Logo, tanto o número de folhas como o número de nós internos é acrescido de um, mantendo-se válida a propriedade $f = i + 1$

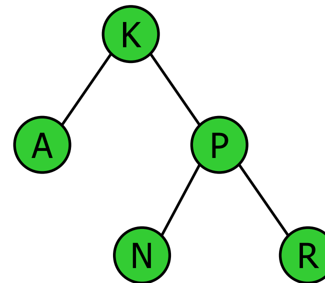
Árvores binárias completas

- Uma árvore binária é *completa* quando os nós internos têm exatamente dois filhos e as folhas têm a mesma distância da raiz
- Uma árvore binária completa de altura h tem exatamente 2^h-1 nós internos e 2^h folhas
- Numa árvore binária completa com 2^h folhas, a distância da raiz até qualquer folha é h
- Prove!!

Árvores binárias de busca

- Definição: Uma árvore binária é de busca se *em cada um de seus nós* :
  - todas as chaves armazenadas na sua sub-árvore esquerda são menores que a chave do próprio nó;
  - todas as chaves armazenadas na sua sub-árvore direita são maiores que a chave do próprio nó.
- Exemplos:





Árvores binárias de busca

- Algoritmo de busca é simples e direto
- Repetição que começa na raiz:
  - Compare com o valor armazenado no nó
  - Se for igual, a busca chegou ao fim
  - Se for menor, vá para a sub-árvore esquerda
  - Se for maior, vá para a sub-árvore direita
  - Se não houver como continuar, o valor não está na árvore
- No pior caso, gastará tempo proporcional à altura da árvore

- Como pode ser feita a inserção e a remoção de valores?
- Objetivos:
  - Garantir a ordenação
  - Manter um balanceamento: é desejável que a altura dessa árvore binária de busca seja proporcional ao logaritmo da quantidade de valores armazenados
- Há diversos algoritmos...