AULA 9

Árvores binárias

Fonte: https://www.tumblr.com/

Referências: Árvores binárias de busca (PF); Binary Search Trees (S&W); slides (S&W)

Mais tabela de símbolos

Uma **tabela de símbolos** (= *symbol table* = *dictionary*) é um conjunto de objetos (itens), cada um dotado de uma chave (= `key`) e de um valor (= `val`).

As chaves podem ser números inteiros ou *strings* ou outro tipo de dados.

Uma tabela de símbolos está sujeito a dois tipos de operações, entre possíveis outras:

- inserção (= `put()`): consiste em introduzir um objeto na tabela;
- busca (= `get()`): consiste em encontrar um elemento que tenha uma dada chave.

Problema

Problema: Organizar uma tabela de símbolos de maneira que as operações de inserção e busca sejam *razoavelmente eficientes*.

Em geral, uma organização que permite inserções rápidas impede buscas rápidas e vice-versa.

Já vimos como organizar tabelas de símbols através de vetores e listas encadeadas e skip lists.

Hoje: mais uma maneira de organizar uma tabela de símbolos.

Árvore binárias

Uma **árvore binária** (= *binary tree*) é um conjunto de nós/células que satisfaz certas condições.
Cada nó terá três campos:

```
public class BT {
  private Node r;
  private class Node {
      private Item item; // conteúdo
      private Node left, right;
      public Node(Item item,
             Node left, Node right) {
          this.item = item;
          this.left = left;
          this.right = right; }
  }
```

Pais e filhos

Os campos `left` e `right` dão estrutura à árvore.

Se `x.left == y`, `y` é o **filho esquerdo** de `x`.

Se `x.right == y`, `y` é o **filho direito** de `x`.

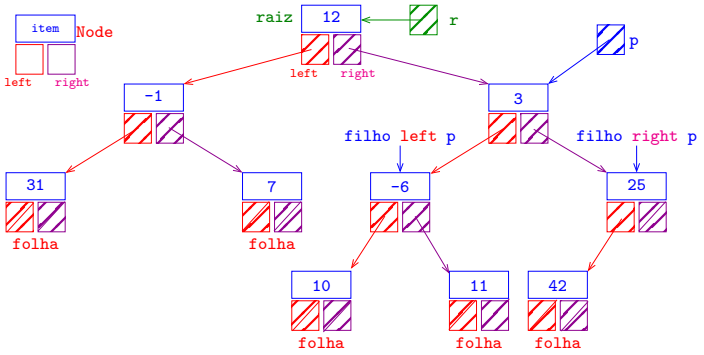
Assim, `x` é o **pai** de `y` se `x.left == y` ou `x.right == y`.

Folhas

Uma **folha** é um nó sem filhos.
Ou seja, se `x.left == null` e `x.right == null` então `x` é um folha.

```
private class Node {
   private Item item; // conteudo
   private Node left, right;
   public Node(Item item,
               Node left, Node right) {...}
   public boolean isLeaf() {
       return left == null
              && right == null;
   }
}
```

Ilustração de uma árvore binária



Árvores e subárvores

Suponha que `r` e `p` são nós.
`p` é **descendente** de `r` se `p` pode ser alcançada pela iteração dos comandos

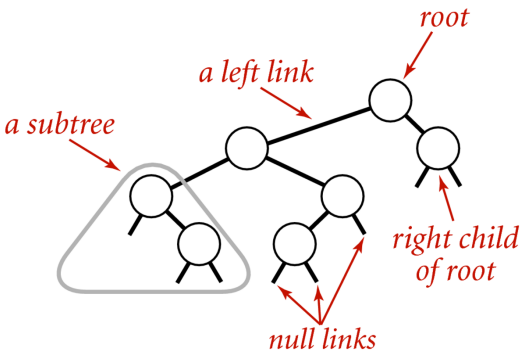
```
p = p.left;        p = p.right;
```

em qualquer ordem.

Um nó `r` juntamente com todos os seus descendentes é uma **árvore binária** e `r` é dito a **raiz** (=*root*) da árvore.

Para qualquer nó `p`, `p.left` é a raiz da subárvore esquerda de `p` e `p.right` é a raiz da subárvore direita de `p`.

Anatomia de uma árvore binária



Anatomy of a binary tree

Fonte: `algs4`

Endereço de uma árvore

O endereço de uma árvore binária é o endereço de sua raiz.

```
Node r;
```

Um objeto `r` é uma árvore binária se

- `r == null` ou
- `r->left` e `r->right` são árvores binárias.

Maneiras de varrer uma árvore

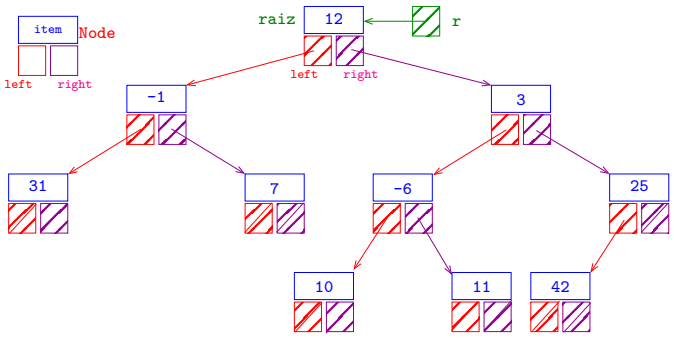
Existem várias maneiras de percorrermos uma árvore binária. Talvez as mais tradicionais sejam:

- *inorder traversal*: esquerda-raiz-direita (e-r-d);
- *preorder traversal*: raiz-esquerda-direita (r-e-d);
- *posorder traversal*: esquerda-direita-raiz (e-d-r);

`preOrdem()`, `inOrdem()` e `posOrdem()` retornam todos os itens da árvore como `Iterable`. Para iterar sobre todos os itens de uma `BT` de nome `st` basta fazermos

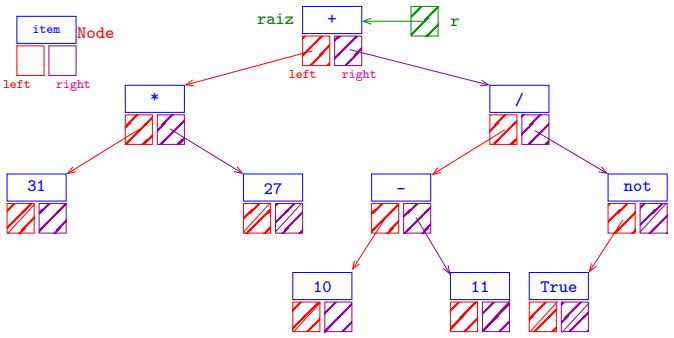
```
for (Item item:  st.preOrdem()) {
    StdOut.println(item);
}
```

Ilustração de percursos em árvores binárias



in-ordem (e-r-d): 31 -1 7 12 10 -6 11 3 42 25
pré-ordem (r-e-d): 12 -1 31 7 3 -6 10 11 25 42
pós-ordem (e-d-r): 31 7 -1 10 11 -6 42 25 3 12

Ilustração de percursos em árvores binárias



in-ordem (e-r-d): 31 * 27 + 10 - 11 / True not
pré-ordem (r-e-d): + * 31 27 / - 10 11 not True
pós-ordem (e-d-r): 31 27 * 10 11 - True not / +

esquerda-raiz-direita

Visitamos

1. a subárvore esquerda da raiz, em ordem e-r-d;
2. depois a raiz;
3. depois a subárvore direita da raiz, em ordem e-r-d;

```
public Iterable<Item> inOrdem() {
   Queue<Item> queue = new Queue<Item>();
   inOrdem(r, queue)
   return queue;
}
```

esquerda-raiz-direita

Visitamos

1. a subárvore esquerda da raiz, em ordem e-r-d;
2. depois a raiz;
3. depois a subárvore direita da raiz, em ordem e-r-d;

```
private void inOrdem(Node r,
                     Queue<Item> queue) {
   if (r != null) {
      inOrdem(r.left, queue);
      queue.enqueue(r.item);
      inOrdem(r.right,queue);
   }
}
```

esquerda-raiz-direita versão iterativa

```
private void inOrdem(Node r,
                     Queue<Item> queue) {
   Stack<Node> s = new Stack<Node>();
   while (r != null || !s.isEmpty()) {
      if (r != null) {
         s.push(r); r = r.left;
      }
      else {
         r = s.pop();
         queue.enqueue(r.item);
         r = r.right;
      }
   }
}
```

raiz-esquerda-direita

Visitamos

1. a raiz;
2. depois a subárvore esquerda da raiz, em ordem r-e-d;
3. depois a subárvore direita da raiz, em ordem r-e-d;

```
public Iterable<Item> preOrdem() {
   Queue<Item> queue = new Queue<Item>();
   preOrdem(r, queue)
   return queue;
}
```

raiz-esquerda-direita

Visitamos

1. a raiz;
2. depois a subárvore esquerda da raiz, em ordem r-e-d;
3. depois a subárvore direita da raiz, em ordem r-e-d;

```
private void preOrdem(Node r,
                      Queue<Item> queue) {
   if (r != null) {
       queue.enqueue(r.item);
       preOrdem(r.left, queue);
       preOrdem(r.right,queue);
   }
}
```

esquerda-direita-raiz

Visitamos

1. a subárvore esquerda da raiz, em ordem e-d-r;
2. depois a subárvore direita da raiz, em ordem e-d-r;
3. depois a raiz;

```
public Iterable<Item> posOrdem() {
   Queue<Item> queue = new Queue<Item>();
   posOrdem(r, queue)
   return queue;
}
```

esquerda-direita-raiz

Visitamos

1. a subárvore esquerda da raiz, em ordem e-d-r;
2. depois a subárvore direita da raiz, em ordem e-d-r;
3. depois a raiz;

```
private void posOrdem(Node r,
                      Queue<Item> queue) {
   if (r != null) {
       posOrdem(r.left, queue);
       posOrdem(r.right,queue);
       queue.enqueue(r.item);
   }
}
```

Primeiro nó esquerda-raiz-direita

Recebe a raiz `r` de uma árvore binária não vazia e retorna o primeiro nó na ordem e-r-d

```
private Node primeiro(Node r)
{
   while (r.left != null)
       r = r.left;
   return r;
}
```

Altura

A **profundidade** (=*depth*) de um nó de uma BT é o número de links no caminho que vai da raiz até o nó.

A **altura** (=*height*) de uma BT é o máximo das profundidades dos nós, ou seja, a profundidade do nó mais profundo.

```
private int altura(Node r) {
   if (r == null) return -1;
   int hLeft = altura(r.left);
   int hRight = altura(r.right);
   return Math.max(hLeft, hRight) + 1
}
```

Árvores balanceadas

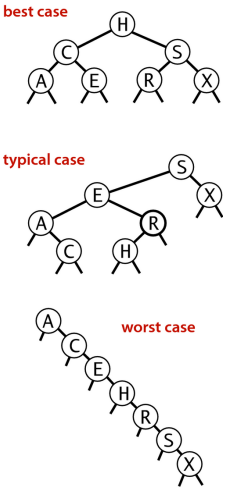
A altura de uma árvore com n nós é um número entre $\lg n$ e n.

Uma árvore binária é **balanceada** (ou **equilibrada**) se, em cada um de seus nós, as subárvores esquerda e direita tiverem *aproximadamente* a mesma altura.

Árvores balanceadas têm altura *próxima* de $\lg n$.

O consumo de tempo dos algoritmos que manipulam árvores binárias dependem frequentemente da altura da árvore.

Exemplos

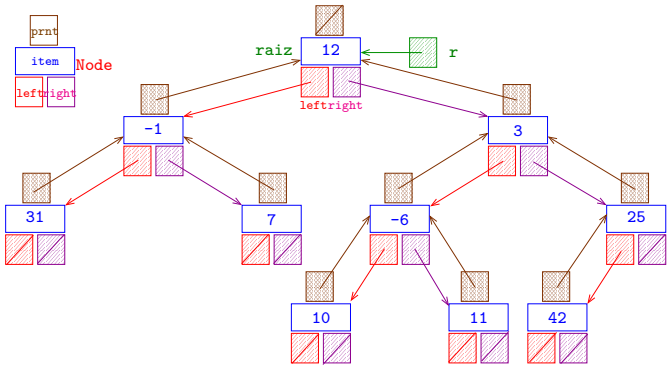


Nós com campo pai

Em algumas aplicações é conveniente ter acesso imediato ao pai de qualquer nó.

```
private class Node {
   private Item item; // conteudo
   private Node left, prnt, right;
   public Node(Item item) {
       this.item = item;
   }
}
```

Ilustração de nós com campo pai



Sucessor e predecessor

Recebe um nó p de uma árvore binária não vazia e retorna o seu **sucessor** na ordem e-r-d.

```
private Node sucessor(Node p) {
   if (p.right != NULL) {
       Node q = p.right;
       while (q.left != null) q = q.left;
       return q;
   }
   while (p.prnt!=null && p.prnt.right==p)
       p = p.prnt;
   return p.prnt;
}
```

Exercício: função que retorna o **predecessor**.

Comprimento interno

O **comprimento interno** (=*internal path length*) de uma BT é a soma das profundidades dos seus nós, ou seja, a soma dos comprimentos de todos os caminhos que levam da raiz até um nó.

Esse conceito é usado para estimar o desempenho esperado de STs implementadas com BSTs

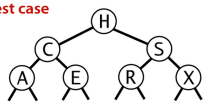
Fique atento

O código a seguir percorre a árvore em pré-ordem e imprime uma sequência de bits que a codifica.

```
private static void writeBT(Node x) {
   if (x == null) {
       StdOut.print(1);
       return;
   }
   StdOut.print(0);
   writeBT(x.left);
   writeBT(x.right);
}
```

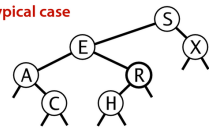
Fique atento!

best case



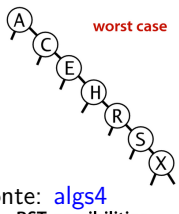

```
0 0 0 1 1 0 1 1 0 0 1 1 0 1 1
H C A - - E - - S R - - X - -
```

typical case




```
0 0 0 1 0 1 1 0 0 1 1 1 0 1 1
S E A - C - - R H - - - X - -
```

worst case




```
0 1 0 1 0 1 0 1 0 1 0 1 0 1 1
A - C - E - H - R - S - X - -
```

Fonte: algs4

Fique atento

Usaremos essa codificação mais a frente em umas coisas bem bacanas.

```
private static Node readBT() {
   if (StdIn.readInt()) {
      return null;
   }
   Node left = readBT();
   Node right = readBT();
   return new Node(null, left, right);
}
```

Análise de desempenho

Fonte: Pinterest

Referências: Analysis of algoritms (S&W))

Análise de desempenho

Estamos interessados em consumo de tempo e memória.

Para sabermos se podemos resolver problemas MAIORES.

Motivação:

- prever o comportamento de programas
- comparar algoritmos e suas implementações
- compreender o problema para desenvolver novos algortmos

Histórias de sucesso: *Discrete Fourier Transform*: processamento de sinais

Problema

3-sum: Dados n inteiros, listar os trios que somam zero.

Aplicações em geometria computacional:

- encontrar pontos colineares
- decidir se um polygono está dentro de outro
- planejar movimento
- ...

Força bruta

```
public static int count(int[] a) {
   int n = a.length;
   int count = 0;
   for (int i = 0; i < n; i++) {
      for (int j = i+1; j < n; j++) {
         for (int k = j+1; k< n; k++) {
            if (a[i] + a[j] + a[k] == 0)
            count++;
         }
      }
   }
   return count;
}
```

Examina todos os $\binom{n}{3}$ trios $= O(n^3)$.

Força bruta

```
public static int count(int[] a) {
   int n = a.length;
   int count = 0;
   for (int i = 0; i < n; i++) {
      for (int j = i+1; j < n; j++) {
         for (int k = j+1; k< n; k++) {
            if (a[i] + a[j] + a[k] == 0)
            count++;
         }
      }
   }
   return count;
}
```

Quanto tempo gasta para $n = 1$ milhão?

Método científico!

- observar a natureza
- fazer hipóteses de modelo consistente com a observação
- prever eventos usando as hipóteses
- verificar as predições e fazer mais observações
- validar repetindo até que as hipoteses e observações estejam de acordo

Formulação de hipóteses

Doubling hypothesis. Podemos formular uma hipótese se soubermos responder a pergunta:

> *Qual é o efeito no consumo de tempo se dobramos o tamanho da entrada.*

Análise experimetal. Podemos fazer experimentos dobrando a entrada e observando o consumo de tempo.

Doubling Method

Hipótese. O consumo de tempo do programa é $T(n) = an^b$.

Consequência Quando n crescem $T(2n)/T(n)$ se aproxima de 2^b.

Doubling method:

- comece com um n moderado
- registre o consumo de tempo
- dobre o valor de n
- repita enquanto for possível
- verifique que a razão de tempos consecutivos se aproxima de 2^b
- prediga e extrapole: multiplique por 2^b para estimar $T(2n)$

Encontrar dados representativos

- dados reais
- escrever um programa para gerar os dados

```
public class Generator {
public static void main(String[] args) {
   int m = Integer.parseInt(args[0]);
   int n = Integer.parseInt(args[1]);
   for (int i = 0; i < n; i++) {
      StdOut.println(StdRandom.uniform(-m,m));
   }
}
}
```

Encontrar dados representativos

```
% java Generator 100000 1000 | java ThreeSum
elapsed time = 0.276
609

% java Generator 100000 2000 | java ThreeSum
elapsed time = 1.969
4860

% java Generator 100000 4000 | java ThreeSum
elapsed time = 16.051
39968
```

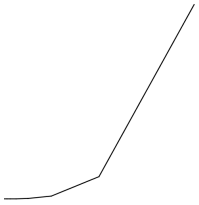
Execute experimentos

```
% java DoublingTest
```

n	tempo (s)	T(2n)/T(n)
512	0.04	2.47
1024	0.36	9.78
2048	0.74	2.03
4096	5.83	7.92
8192	46.06	7.91
16384	363.02	7.88
32768	3157.43	8.70

Plot os resultados

```
% java PlotFilter < dados/doubling.txt
```



Análise de dados

- Plotar em escala log-log
- se pontos estão em uma reta a curva é da forma $a \times n^b$
- o expoente b é a inclinação da reta
- encontre a resolvendo a equação com os dados

Análise de dados

```
% java PlotFilter < dados/doubling.txt
```

$$\lg T(n) = \lg a + 3 \lg n \Rightarrow$$
$$T(n) = an^3 \Rightarrow$$
$$3157.43 = a \times 32768^3 \Rightarrow$$
$$a = 8.97 \times 10^{-11}$$

Predição e verificação

Logo, $T(n) = 8.97 \times 10^{-11} \times n^3$.

$n = 16384 \Rightarrow T(n) = 394.50s$

$n = 8192 \Rightarrow T(n) = 49.31s$

$n = 1000000 \Rightarrow T(n) = 89700000s$
aproximadamente 1038 dias

Predição e verificação

Em 1970, $a = 5.2 \times 10^-6$

$$T(n) = a \times n^3 \Rightarrow$$
$$T(1000000) = 170 \text{ anos}$$

Em 2010, $a = 4.84^-10$

$$T(n) = a \times n^3 \Rightarrow$$
$$T(1000000) = 15.7 \text{ anos}$$

Meu computador, $a = 8.97 \times 10^{-11}$

$$T(n) = a \times n^3 \Rightarrow$$
$$T(1000000) = 2.9 \text{ anos}$$

Modelo matemático

Consumo de tempo depende de:

- consumo de tempo de cada operação (depende do computador)
- frequência que cada operação é executada (depende do algoritmo)

Modelo matemático

Cálculo da frequência pode ser difícil...

Usamos simplificações!

- notação tio (*tilde notation*)
- notação assintótica ($O(\ldots)$)

Notação $\sim$: Escrevemos $\sim f(n)$ para representar qualquer função $g(n)$ tal que $g(n)/f(n) \to 1$ quando $n \to \infty$.

Análises experimental e matemática

Análise experimental:

- tempo gasto pelo programa em um computador
- ajustar uma curva aos dados para obter uma fórmula que descreve o consumo de tempo em função de n
- útil para prever mas não para explicar

Análise matemática:

- analiza o algoritmo para encontrar uma fórmula para o consumo de tempo em função de n
- útil para prever e explicar
- pode envolver matemática avançada
- independe do computador