

Análise temporal:

Quanto tempo demora uma função?

Existem muitos fatores que fazem com que a resposta seja diversificada (dependendo do hardware, do compilador, potencialmente variável)

O algoritmo não depende do hardware, e não depende da entrada

Mas o número de execuções de cada linha depende da entrada fornecida à função (dimensão do array, valores do array)

Tempo de execução vs Complexidade do algoritmo

A complexidade do algoritmo indica de que forma o seu tempo de execução varia com tamanho de entrada

Como representar a complexidade?

Com a notação de O-grande

$O(n)$ – linear ou melhor, o tempo pode aumentar para o dobro ou pode ser menos do que isso

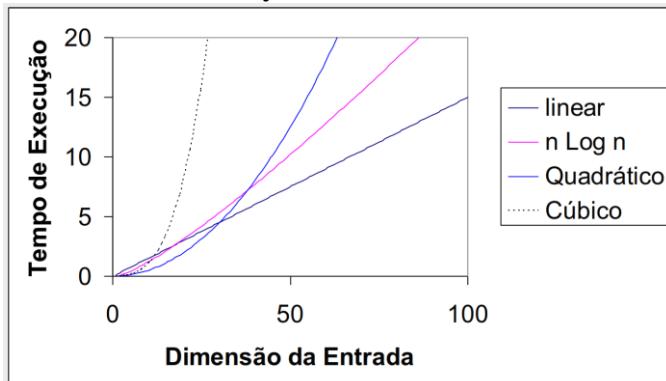
$\Theta(n)$ – exatamente linear (Θ, θ, Ω), o tempo vai realmente demorar o dobro se o tamanho do array passar o dobro.

Qualquer coisa $\Theta(n)$ é obviamente $O(n)$, qualquer coisa que é exatamente linear, é linear

Classes de complexidade:

- $\Theta(1), O(1)$
 - Constante. É um algoritmo cujo tempo de execução é constante, ou tem tendência a manter-se constante quando a dimensão da entrada aumenta.
- $\Theta(N), O(N)$
 - Linear.
- $\Theta(N^2), \Theta(N^3), \dots, O(N^2)$
 - Quadrática, cubica, .. Por exemplo: quadrático corresponde a um algoritmo quando a complexidade da entrada aumenta dez vezes o tempo de execução aumenta cem vezes, e se fosse cubico o tempo de execução aumentava mil vezes
- $\Theta(\log(N)), O(\log(N))$
 - Logarítmica
- $\Theta(N * \log(N)), O(N * \log(N))$
 - N-Log

O crescimento de funções:



Curva linear:

É melhor este algoritmo porque existe um ponto de dimensão de entrada a partir do qual o algoritmo linear é sempre melhor

Esse ponto existe sempre

Há-de existir um determinado valor de entrada a partir do qual o algoritmo linear vai ser sempre melhor do que o quadrático

Curva $n \log n$:

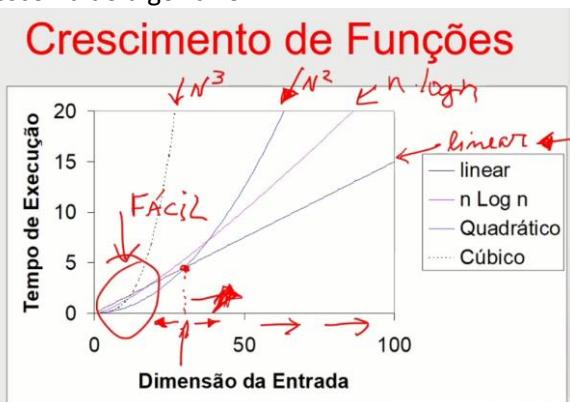
Também $n \log n$, vai ser mais baixo do que N^3 a partir de um determinado ponto

Curva quadrática:

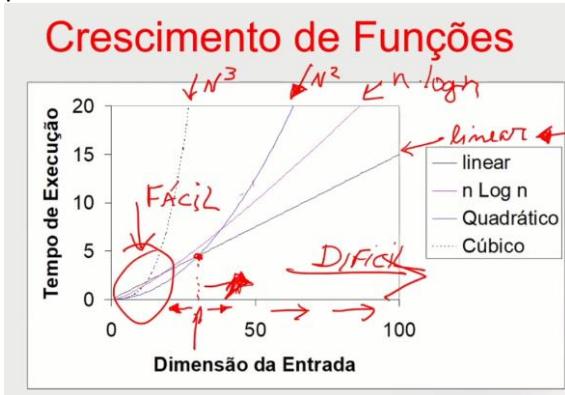
Curva cúbica:

Assim para uma determinada dimensão e entrada “suficientemente grande” o algoritmo vai ser melhor.. o linear.

Existe a zona fácil: onde qualquer coisa funciona, processando poucos dados, indiferente a escolha do algoritmo



Os problemas relacionados com desempenho surgem, quando as dimensões dos dados processados aumentam.



A notação O-grande não indica um valor específico de tempo, mas vai indicar como a função cresce para valores suficientemente elevados da dimensão da entrada

Exemplo:

O tempo de execução de um algoritmo vem:

$$2n^3 + 20n$$

Temos um desempenho cubico, $O(n^3)$

E porque se ignora $20n$, porque para valores grandes de n , os termos de maior valor como é o caso de n^3 é o termo que vão ser mais relevantes para o valor final

Situação do exemplo 1:

Se $n=1$

$2*1^3 + 20*1$, dá valores de 22

A maior parte do valor vem de $20n$

Mas se $n=2$

$2*2^3+20*2$, dá valores de 48 (distribuição de 8+40)

Mas se $n=3$

(distribuição de 16+60)

Mas se $n=10$

(distribuição de 2000+200)

Assim desprezamos os valores de menor ordem, assim $2n^3+20n$ é uma função de $O(n^3)$

O 2 não acrescenta nada, pois dizer $O(n) \Leftrightarrow o(2n)$

Exemplo:

O tempo de execução de um algoritmo vem:

$\frac{1}{2} * n^2 + 200n - n \log n$

Temos um desempenho quadrático, $O(n^2)$

A notação de O-Grande, $O(n)$, consegue abstrair-se do hardware,

Exemplo:

Qual é a complexidade de procurar o menor elemento num array de N elementos: $\Theta(N)$, porque é exatamente linear, eu não consigo saber o valor mais baixo sem ver todos os valores. $O(n)$ não está errado, mas não conseguimos ser melhor do que $\Theta(N)$, É exatamente linear

Exemplo:

Qual é a complexidade de procurar os dois pontos (x,y) mais próximos num plano num conjunto de N pontos. Quando temos uma situação de soma de valores, de procurar, $1..2..3..4..5..6....$ é quadrático, algoritmo é de complexidade quadrática, eu não sei se o último par é o mais próximo, vou ter que ver tudo.

Existem $N(n-1)/2$

O tempo de execução de um algoritmo quadrático cresce muito depressa, para problemas a sério não é possível usar algoritmos quadráticos, a sua ajuda é nula. A única situação é quando o número de dados processados for estritamente limitado. Se a quantidade de dados poder ser variável e aumentar existe zero algoritmos quadráticos com utilidade prática.

É exatamente quadrática,

Exemplo:

Verificar se num conjunto de N pontos há três que sejam colineares (que forem uma linha)

Existem $N(n-1)(n-2) / 6$, complexidade cubica, polinómio de terceiro grau

$O(n^3)$, não preciso de analisar todos os pontos, basta encontrar o primeiro, mas no pior caso se não existir nenhum eu tenho que percorrer todos

É no pior caso cubica

(ED03 - Complexidade (Parte 2).pdf)

Só o tempo não é suficiente para analisar um algoritmo, para comparar algoritmos.

O tempo é variável e difícil de definir de forma rigorosa

Usamos a analise de complexidade, que colocar os algoritmos em diferentes classes, como por exemplo, constante, linear, quadrática, etc..

A notação para identificar a complexidade que usamos é O-Grande em que:

Algo linear, é $O(N)$

Algo quadrático, é $O(N^2)$

Em que N é o tamanho que estamos a resolver, tipicamente é a quantidade de dados que estamos a processar

Exemplo:

$N \log N$ é $\Omega(N)$?

Sabendo eu $O(n)$ é linear ou melhor que é o mesmo que dizer que é \leq

$\Theta(N)$, representa o igual

$\Omega(N)$, Maior ou igual, e quer dizer que é linear ou pior

Exemplo:

Considere que um algoritmo tem complexidade de ordem $O(N)$. Quanto é que o tempo de execução aumenta se o tamanho da entrada aumentar 10 vezes?

R:

O tempo de execução aumenta na mesma proporção da ordem que a complexidade

Algo quadrático que aumente 10x a dimensão de entrada, vai passar 100x mais

Algo quadrático que aumente 100x a dimensão de entrada, vai passar 10000x mais

#Logaritmos

Um logaritmo é o número de bits necessários para representar o número n

No logaritmo de base 2 indica: "o número de vezes que N deve ser dividido para atingir 1"

$O(\log_b N) = O(\log N)$ é o mesmo que $O(5^*N) = O(N)$

Exemplo:

Dado um número X e um array A , devolver a posição de X em A ou uma indicação que X não existe.

R:

Usando a Pesquisa Sequencial

Quantos valores temos que analisar numa pesquisa sem sucesso? $\Theta(N)$

De uma pesquisa com sucesso, no pior caso? Percorrer o array todo, $\Theta(N)$

De uma pesquisa com sucesso, no caso médio? .. o que é médio?..

A pesquisa sequencial é uma pesquisa linear

R:

Usando a Pesquisa Binária

Existe uma restrição: os dados têm que estar ordenados

O que interessa é eliminar uma % da pesquisa, tipicamente queremos eliminar 50%

Qual a complexidade de uma pesquisa falhada? É reduzirmos a pesquisa a 1 valor e esse valor não é o que queremos procurar

E de uma pesquisa com sucesso, no pior caso? É ser o último valor

E de uma pesquisa com sucesso, no caso médio? É uma somatória de log..

E de uma pesquisa com sucesso, no melhor caso? O valor que quero é o que está no meio

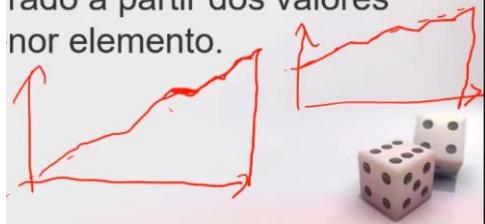
O algoritmo de pesquisa Binária é mais eficiente do que o de Pesquisa Sequencial, se der para ser aplicado

R:

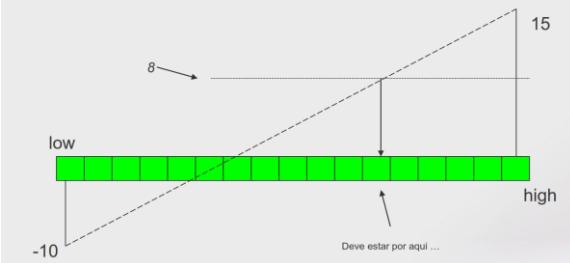
Usando a Pesquisa Interpolada

O array está ordenado e E o

s valores estão uniformemente distribuídos
rando a partir dos valores
nor elemento.



- Procurar o número 8



Qual é o pior caso? Os valores não estão uniformemente distribuídos, e podemos ter que pesquisar todos

Qual é o caso médio? $O(\log \log N)$

Exemplo:

Qual a complexidade do seguinte algoritmo

```
1. public static int maxSeqCont(int [] m)
2. {
3.     int maxSoma=0;
4.     Int N=m.length;
5.     for(int i=0;i<N;i++)
6.         for(j=i;j<N;j++){
7.             int estaSoma=0;
8.             for(k=i;k<j;k++)
9.                 estaSoma+=m[k];
10.            if(estaSoma>maxSoma)
11.                maxSoma=estaSoma;
12.        }
13.    return maxSoma;
14. }
```

Complexidade cubica, $O(n^3)$, pois o terceiro for pode implicar percorrer todos os números
A maior parte do tempo vai ser para executar a linha 9

Podemos obter a versão 2:

```

1. public static int maxSeqCont(int [] m)
2. {
3.     int maxSoma=0;
4.     int N=m.length;
5.     for(int i=0;i<N;i++){
6.         int estaSoma=0;
7.         for(j=i;j<N;j++){
8.             estaSoma+=m[j];
9.             if(estaSoma>maxSoma) ←
10.                 maxSoma=estaSoma;
11.             }
12.     }
13. return maxSoma;
14. }
```

A passa a ser de complexidade quadrática, $O(N^2)$

Podemos obter a versão 3:

```

1. public static int maxSeqCont(int [] m)
2. {
3.     int maxSoma=0;
4.     int N=m.length;
5.     int estaSoma=0;
6.     for(int i=0;i<N;i++){
7.         estaSoma+=m[i];
8.         if(estaSoma>maxSoma)
9.             maxSoma=estaSoma;
10.        if(estaSoma<0)
11.            estaSoma=0;
12.    }
13. return maxSoma;
14. }
```

A passa a ser de complexidade linear, $O(N)$

(ED04c- Genéricos.pdf)

Os métodos genéricos

Exemplo:

Pretende-se um método que imprime o conteúdo de um array:

```

public static void imprime(String [] m)
{
for(String a : m)
    System.out.println(a);
}

public static void main (String args[]){
String m[]={“José”, “António”};
imprime(m);
}
```

E generalizando para qualquer tipo de dados:

```

public static <T> void imprime(T [] m)
{
    for(T a: m)
        System.out.println(a);
}

```

Surge antes do valor de retorno <T>, tipo ou classe genérico

Exemplo:

```

public static <T> T escolheAleat(T a, T b)
{
    T ret;
    if(Math.random()>0.5)
        ret=a;
    else
        ret=b;
    return ret;
}

```

Aqui o valor de retorno tem que ser da classe T
O tipo de valor que recebe é irrelevante
A lista <T> tem que aparecer antes do valor de retorno

Exemplo:

Podem também haver classes genéricas

```

public class Par <T,S> {
    T primeiro;
    S segundo;
    public Par(T a, S b){
        primeiro=a;
        segundo=b;
    }
    public T getPrimeiro(){return primeiro;}
    public S getSegundo(){return segundo;}
    public setPrimeiro(T a){primeiro=a;}
    public setSegundo(S b){segundo=b;}
}

```

E porque não usar o Object?

No Object não existe validação dos tipos

Nas funções get devolvo um object e terei que fazer um cast para o que lá está

```

public static void main(String args[])
{
//Java 7+
Par<String, Integer> p1=new Par<>("Par1",1);
Integer i1=p1.getSegundo(); //ok

```

É possível colocar restrições aos parâmetros formais.

Exemplo:

```

public static <T extends Number> boolean maior (T p1, T p2){
    return p1.doubleValue()>p2.doubleValue();
}

```

A restrição é feita usando extends Number, e talvez deixando em aberto a classe T pode acontecer que uma arvore ou a class object não tenha o método doubleValue e dar erro/não funciona

Assim adicionamos restrições aos parâmetros formais como por exemplo com o extends Number (este Number pode também ser uma interface) e na classe Number existe o método doubleValue

#Wildcards e restrições

As wildcards podem ser especificadas da seguinte forma:

Ou X extends Y, em que, X é igual a ou estende a classe Y (ou implementa interface Y), x é derivado de Y

Ou X super Y, em que, Y é igual a ou estende a classe X (ou implementa interface X), Y é derivado de X

Em algumas circunstâncias podemos:

X ou Y podem ser substituídos por '?', o que significa "qualquer classe", "qualquer coisa"

Exemplo:

```
public static <T,S> void f(par<T,S> p)
```

Pode ser escrito como

```
public static void f(par<?,?> p)
```

Exemplo:

```
public static <T, S extends T> void f(par<T,S> p)
```

Pode ser escrito como

```
public static <T> void f(par<T, ? extends T> p)
```

em que por exemplo T pode ser number, e S integer, já que integer é derivado de T

Exemplo:

```
public static <T> void f(par<T, ? extends T> p)
```

```
{
```

```
T p1 = p.getFirst(); //funciona
```

```
T p2 = p.getSecond(); //funciona porque existe o extends T
```

```
}
```

pois o que temos aqui é a=b, qual é a relação entre a e b?

têm que ser do mesmo tipo mas não necessariamente porque pode acontecer que

Figura a = new Reactangle();

Assim a = b, se a for do tipo T, tenho que garantir que seja qualquer classe que seja igual a T ou derivada e isso faz-se com ? extends T

#Suporte de Polimorfismo com Wildcards

Exemplo:

```
public class Par <T,S> {
```

```
    T primeiro;
```

```
    S segundo;
```

```
    public Par(T a, S b){
```

```
        primeiro=a;
```

```
        segundo=b;
```

```
}
```

```
    public T getPrimeiro(){return primeiro;}
```

```
    public S getSegundo(){return segundo;}
```

```
...
```

```
    public void copia(Par<T,S> outroPar)
```

```
{
```

```
    primeiro=outroPar.getPrimeiro();
```

```
    Segundo=outroPar.getSegundo();
```

```

        }
    }

class Rect{
...
};

class Quad extends Rect{
...
}

public static void main(String args[])
{
    Quad q1=new Quad(1), q2=new Quad(2);
    Rect r1=new Rect(2,3), r2= new Rect (3,4);
    Par<Quad,Rect> p1=new Par<>(q1,r1);
    Par<Quad,Rect> p2=new Par<>(q2,r2);
    p2.copia(p1); //ok
    Par<Rect,Rect> p3=new Par<>(q1,r1);
    Par<Rect,Rect> p4=new Par<>(q2,r2);
    p3.copia(p4); //ok
    p3.copia(p1); // erro de compilação:
    //Par<Quad,Rect> não pode ser usado
    // no lugar de Par<Rect,Rect>
}

```

Não compila pela forma como foi definido o parâmetro
A função copia está definida de forma demasiado restritiva pois não foram usados wildcards

Outra versão do copia:

```

public void copia
(Par<? extends T,? extends S> outroPar)
{
    primeiro=outroPar.getPrimeiro();
    Segundo=outroPar.getSegundo();
}

```

Desta forma quando se faz

```

primeiro=outroPar.getPrimeiro();
se a=b, ou pode ser qualquer coisa que seja derivado de T -> ? extends S
o primeiro parâmetro seja qualquer coisa T ou derivado de T -> ? extends T

```

estas duas restrições

```

<? extends T,? extends S>
vão permitir que o código assim compile
permitem uma maior gama de classes

```

E a comparação entre coisas?

Se forem tipos básicos de dados, por exemplo a > b vai funcionar

Mas e se forem objetos?

É usado o interface comparable

Exemplo:

Class Figura com,

```
class Figura implements Comparable<Figura>{
    int compareTo(Figura f){...}
}
Class Rect extends Figura{...}
...
Rect r=new Rect(2,3), q=new Rect(3,4);
r.compareTo(q);
```

Tem que se implementar a interface Comparable
E assim Figura pode ser comparada com outras figuras
E tenho que implementar o método compareTo
<, se se for menor,
0 se for igual
>0, se for maior
E não esquecer que Rect extends Figura

E isto é válido que o Rect é uma figura, e assim todas as figuras são comparáveis entre si, até podem ser rect ou círculos

Exemplo:

Temos,

```
class Figura implements Comparable<Figura>{
    int compareTo(Figura f){...}
}
<T> int compara (Comparable<T> a , Tb ){
    return a.compareTo(b);
}
Rect r=new Rect(2,3), q=new Rect(3,4);
r.compareTo(q);
Compara(r,q); //funciona? Não..
```

O retângulo é comparable com rect?
Não, figura é comparable com figura
Só funciona se o r e o q forem figuras

Será que funciona se usarmos

Comparable<? extends T> T

Não é verdade! Errado.. porque não se vai comparar círculos com retangulos

E se for:

Comparable<Object> T

Sim funciona.

Assim o que está correto é comparar com uma classe qualquer, que pode ser T ou Super ficando assim:

Comparable<? Super T> T

Exemplo:

Temos,

```
class Figura implements Comparable<Figura>{
    int compareTo(Figura f){...}
```

```

    }
<T> int compara (Comparable<? super T> a , Tb ){
return a.compareTo(b);
}
Rect r=new Rect(2,3), q=new Rect(3,4);
r.compareTo(q);
Compara(r,q); //funciona?

```

Análise do uso do super:

R é um rectangulo, Rect

Q é um rectangulo, Rect

R tem que ser comparable com um tipo qualquer que é Rect ou que está acima de Rect

Isso verifica-se porque Rect é comparable com Figura, que é um tipo que está acima de Rect

Super:

Aplica-se em situações em que queremos garantir que o tipo que usamos pode receber do tipo original

Em qualquer situação em que existam genéricos, por exemplo:

<T, S extends Comparable <T>> void f() não é isto que se pretende escrever mas sim

<T, S extends Comparable <? Super T>> void f()

A intenção é a de fazer a comparação com T ou classe ou acima dela

Exemplo:

<T> void find(List<? super T> l, T t)

{

..

l.add(t);

}

Faz sentido? Sim já o seguinte não faz

<T> void find(List<T> l, T t)

{

..

l.add(t);

}

Já que só conseguia procurar por exemplo uma string numa lista de strings

Com o uso do super eu olho para cima, isto é, posso colocar os Rect numa lista de Figuras

#Limitações dos genéricos

Os genéricos só existem durante a compilação

Exemplo:

Como definir uma classe ParComparável, sabendo que esta representa um Par cujos elementos são comparáveis. A classe ParComparável deve por sua vez ser também comparável: a comparação deve ser decidida através da comparação dos dois primeiros elementos de cada par, desempatando com a comparação dos segundos.

Parte 1:

uma classe ParComparável, sabendo que esta representa um Par cujos elementos são comparáveis

```
public class ParComparavel <S extends Comparable<? super S>,
T extends Comparable<? super T>>
extends Par<S,T>
{...}
```

Parte 2:

A classe ParComparável deve por sua vez ser também comparável

```
public class ParComparavel <S extends Comparable<? Super S>,
T extends Comparable<? Super T>>
extends Par<S,T>
implements Comparable<ParComparavel<? extends S,>? extends T>>
{...}
```

Parte 3:

a comparação deve ser decidida através da comparação dos dois primeiros elementos de cada par, desempatando com a comparação dos segundos

```
class ParComparável ...{
```

```
...
int compareTo(ParComparavel<? extends S,>? extends T> p)
{
int compPrimeiro= p.getPrimeiro().compareTo(getPrimeiro());
if(compPrimeiro==0)
    return p.getSegundo(getSegundo());
else
return compPrimeiro;
}
...
}
```

ED05-iteradores.pdf

#iteradores

Server para isolar as estruturas de dados dos algoritmos que usam as estruturas de dados
Os iteradores são objetos utilizados para percorrer estruturas de dados

Para se implementar por exemplo um iterador:

```
public interface Iterator<E>{
    boolean hasNext();
    E next(); // gera exceção caso não exista
}
```

Em que E é o tipo de objeto que estamos a percorrer

E dois métodos importantes: hasNext e o next

Em que o

hasNext

devolve verdadeiro ou falso se existem mais valores para percorrer

e o next

avança para o próximo valor E devolve o próximo valor

Exemplo:

```
public class Par <T> implements Iterable<T>
{
T p1,p2;
Iterator<T> iterator(){
```

```

        return new IteratorPar<T>(this);
    }
    ...
};
```

É importante:

implements Iterable<T>
pois é uma classe iterável, que nos fornece um iterador para percorrer os valores que lá tem dentro
e com Iterator<T> iterator(){
que é o método que devolve o iterador para objetos do tipo T

```

e
public class IteratorPar<T> implements Iterator<T>{
int counter=0;
Par<T> par;
IteratorPar( Par<T> p){par=p;}
Boolean hasNext() {return counter!=2;}
T next(){
switch(counter)
{
case 0: counter++; return par.p1;
case 1: counter++; return par.p2;
default: throw new NoSuchElementException();
};
}
}
```

Iterator diferente de Iterable, cuidado
E na classe IteratorPar tem implementado os métodos
hasNext e next

Com algoritmos também se pode:

```

public <T> boolean procura(Iterable<T> m, T o){
Iterator<T> it=m.iterator();
boolean proximo=it.hasNext();
while(proximo){
    if(it.next()==o) // compara referência, não conteúdo
        return true;
    proximo=it.hasNext();
}
return false;
}
```

Inclui-se o Iterable

Em

```

public <T> boolean procura(Iterable<T> m, T o){
chamar a função iterator para se ir buscar o iterador
Iterator<T> it=m.iterator();
Depois chamar hasNext para saber se existe o próximo valor
boolean proximo=it.hasNext();
chamar o next para devolver o valor se existir
```

```
if(it.next()==0)
e continuar a procurar o próximo valor
proximo=it.hasNext();
```

este método funcionar com a classe Par
Par<Integer> p = ...
Procurar (p,x);
Ou funciona se tiver uma lista
List<Integer> l = ..
Procurar(l,x)

Nos ciclos for abreviados

```
for(integer i: l)
{
...
}
for(integer: p)
{
...
}
```

#Iteradores e Excepções

Os iteradores podem/devem gerar as seguintes excepções:

- NoSuchElementException – Tentativa de acesso a um elemento que não existe
- UnsupportedOperationException – A operação (p.ex: remoção) não é suportada.

Existem também o método remove, mas que não faz sentido para este tipo de dados

Mas,

O método remove, remove um valor

O valor que vai ser removido não é o que é apontado pelo iterador já que o iterador aponta para o meio de dois números

O método remove, remove o último valor devolvido pelo next

Exemplo:

Temos uma lista l

Uso um iterador:

It = l.iterator();

E faz-se um remove

It.remove();

Que valor é que remove?

Como não existe o next, porque o next nunca devolveu um valor, não vai remover nada e cria a excepção IllegalStateException

Exemplo: Se quero remover o valor

Uso um iterador:

It = l.iterator();

Faz o next

It.next() //para devolver neste caso o primeiro valor

E faz-se um remove

It.remove();

Exemplo: e se surgir o seguinte:

It = l.iterator();

It.next()

```
It.remove();
It.remove();
```

O valor que ele vai remover, não é nada, porque não houve next, surge então o IllegalStateException. Apos o primeiro remove a seguir ao next não está lá nada.

- IllegalStateException – Tentativa de remoção sem avançar para primeiro elemento ou tentativa de remover o mesmo elemento mais do que uma vez.
- ConcurrentModificationException – Quando se tenta usar um iterador inválido. Um iterador é inválido quando a coleção foi alterada externamente (através de um outro iterador, por exemplo).

Exemplo:

```
Valores guardados 1 2 3
next() devolve 1
next() devolve 2
remove() apaga 2
next() devolve 3
```

faltou ver o ConcurrentModificationException

#API de Colecções

Todas as estruturas de dados implementam a interface Collection<T>.

Inserção/Remoção: add, addAll, remove, removeAll, retainAll, clear.

Conversão para array: toArray

Suporte para iteradores: iterator

Informação e pesquisa: isEmpty, size, contains, containsAll

todas as implementações de Collection deverão possuir os seguintes construtores:

- Construtor sem parâmetros
- Construtor que cria uma coleção que referencia os mesmos elementos que outra coleção

Exemplo:

```
class Pacote <X> implements Collection<X>
{
    Pacote(){
        //cria coleção vazia;
        ...
    }

    Pacote(Collection<? extends X> c){
        //cria uma coleção que referencia
        //os mesmos elementos que c
        ...
    }

    ...
}
```

#Tipos de Colecções – List

Implementações de List:

LinkedList

ArrayList (ou Vector)

Têm métodos para as posições dos objetos:

indexOf
set
get

e iteradores bidirecional:

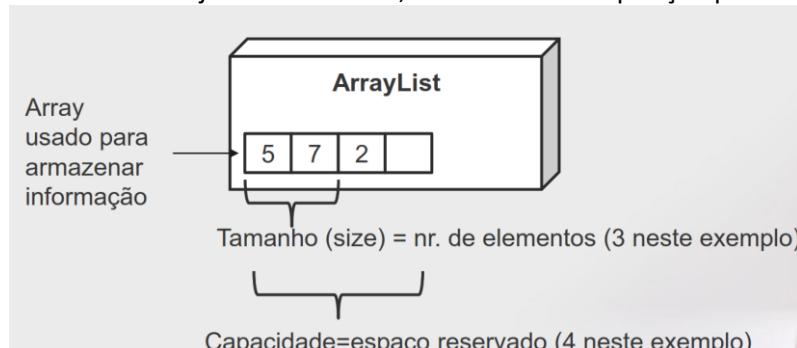
listIterador() e listIterador(int index)

e o ListIterator tem os métodos:

hasNext
next
add //insere elemento na lista
hasPrevious
nextIndex//retorna o índice do próximo elemento
previous //retorna o elemento anterior
set //modifica o valor que foi devolvido

Operações sobre ArrayLists

A ordem dos objetos é relevante, onde existe uma posição para cada objeto



Conceitos importantes: tamanho e capacidade

Qual é a complexidade das funções sobre as ArrayLists?

Qual a melhor estrutura de dados adequada para um determinado cenário?

Acrescentar um valor

Só funciona se a capacidade for maior do que o tamanho, a inserção é feita em tempo constante $O(1)$

Exemplo:

```
a[size] = v;  
size++;
```

e se não existir espaço livre?

Recriar o novo espaço maior, $O(1)$

Copiar todo o conteúdo para o novo espaço, $O(N)$ (vou demorar o dobro do tempo a copiar, demora tempo proporcional a N)

E adicionar o novo valor

A complexidade de add(N) vem:

$O(1)$, operação de tempo constante, capacidade > tamanho (size) ou tamanho < capacidade, $O(N)$, operação de tempo linear, capacidade = tamanho (size)

E se tiver que realocar, qual é novo espaço que se deve alocar?

Acrescentar mais um valor na capacidade fica com complexidade $O(N^2)$ que é muito mau. O número de células a aumentar não pode ser constante, assim se o valor atual, N, for 10 eu por exemplo devo aumentar para 2x mais, para 20... pois desta forma atinge-se comportamento constante amortizado.

Eu tenho um pico quando crio o novo espaço e copio a primeira vez, mas depois eu sei que a seguir não vou demorar tanto tempo, ele vai ser amortizado, as próximas inserções vão ser eficientes.

Se eu inserir n valores o tempo vai ser $O(N)$,

A inserção de 1 é $O(1)$ e vamos ter o custo de $O(1)$ amortizado (os tempos eficientes diuense)

Se inserirmos um numero, suficientemente grande de números da valores, a operação é de tempo $O(1)$ amortizado

Na pior das situações o último valor a inserir é o valor que provoca o aumento da capacidade, e se isso acontece vou ter mais o dobro do espaço, e esse espaço não vai ser usado e é um custo

Exemplo:

Se eu precisar de uma estrutura em que vou adicionando valores, se calhar um array list é uma estrutura boa, porque o desempenho vai ser de $O(1)$ amortizado para uma utilização normal, em termos de desempenho natural, usando adds

Complexidade de aceder a um valor de um arrayList:

Tem tempo Constante, vou para a posição usando o get

Complexidade de remover um valor de um arrayList:

Remover um valor de uma posição,

Terei que copiar os seguintes para as posições anteriores

//isto é operação de tempo linear, $O(N)$

E decrementar o size

E,

A pior posição para remover num arrayList é a primeira

A melhor posição para remover num arrayList é a ultima

Exemplo:

```
for(i=0; i <N; i++) //N vezes
    if(l.get(i) == 0){ //O(1), linear
        l.remove(i); //O(N)
        i--;
    }
}
```

O tempo deste código é $O(N^2)$, o que é mau..

Mas existem variações no remove, por exemplo se eu for remover o último da posição do array, vamos ter que percorrer o array todo para procurar o objecto a remover

A operação remove object vai demorar sempre um tempo linear

Uma forma melhorada do remove é começar a procurar do fim para o inicio, é uma ligeira otimização para tentar a manter o desempenho constante.

Situações no arrayList:

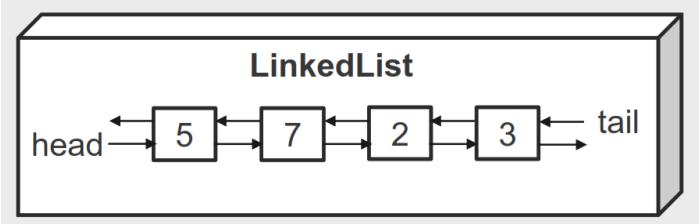
Operação	Complexidade
Add(obj)	O(1) amortizado. O(N) se capacidade esgotada, O(1) caso contrário.
Get(pos)	O(1)
Set(pos,obj)	O(1)
Remove(obj)	O(N) – É preciso procurar o obj
Add(pos,obj)	O(N)
Iterador.add(obj)	O(N)
Iterador.set(obj)	O(1)
Iterador.remove()	O(N)
Remove(ultima posição)	O(1) – Não há valores à frente...

O(1), tempo constante
O(N), pelo menos é linear

As operações que quero evitar no arrayList são:

Operação
Add(obj)
Get(pos)
Set(pos,obj)
Remove(obj)
Add(pos,obj)
Iterador.add(obj)
Iterador.set(obj)
Iterador.remove()
Remove(ultima posição)

Operações sobre LinkedLists
Usando listas duplamente ligadas



Qual a complexidade do get para aceder a uma posição aleatória?
Tenho que começar de uma das pontas e chegar à posição pretendida, complexidade linear, O(N)

Qual a complexidade do adicionar um novo valor?
Tempo constante, não interessa o tamanho da lista, O(1) verdadeiro e não amortizado

Qual a complexidade do adicionar um novo valor no inicio?

Tempo constante, não interessa o tamanho da lista, O(1) verdadeiro e não amortizado

Qual a complexidade do adicionar um novo valor no meio?

Tempo linear, tenho que ir até à posição onde quero inserir, não é melhor do que um arrayList

Exemplo:

```
for(i=0; i < N; i++){ //N vezes  
    print(l.get(i)); //O(N)  
}
```

Demora então N^2

Situações na LinkedList:

Operação	Complexidade
Get(pos)	O(N) – acesso a posição explícita
Set(pos,obj)	O(N) – acesso a posição explícita
Remove(obj)	O(N) – É preciso procurar...
Add(pos,obj)	O(N) – acesso a posição explícita
Iterador.add(obj)	O(1)
Iterador.set(obj)	O(1)
Iterador.remove()	O(1)
Get, Set, Remove, Add (primeira ou ultima posição)	O(1) – Acesso direto a primeiro e ultimo

Usando iteradores neste caso é sempre tempo constante, O(1)

Exemplo:

```
for(i=0; i < N; i++){ //N vezes  
    if(i.get(i) == 0) //O(N)  
        l.remove(i) //O(N)  
}
```

Demora então N^2 .. comportamento quadrático

Não esquecer que $O(N) + O(N) = 2 * O(N^2)$ mas que fica $O(N^2)$

Exemplo:

```
while(it.hasNext){ //percorrer os valores todos  
    if(i.next() == 0) //O(1) //operação que demora tempo constante  
        l.remove(i) //O(1) //operação que demora tempo constante  
}
```

Comportamento linear

Quando se faz uma operação por exemplo de procurar algo na linkedList, estou a fazer O(N), contudo com esta operação o iterador já lá fica “nessa posição” e assim sendo quando voltar a fazer outra operação o iterador “já pode lá estar”

Se eu tiver um objecto que recebe um objeto do tipo List, por exemplo:

List<T> L

Não sabemos o que temos, pode ser um arrayList ou pode ser um linkedList, devemos usar o iterador, pois não sabemos que tipo de lista que vamos receber

Faltou: especialização do interface Collection Set, especialização do interface Collection Queue, e a hierarquia de colecções Map,

[aula 05]

ED07-Arvores Binarias.pdf

#Árvores Binárias

Nas Árvores Binárias temos, zero um ou dois descendentes em cada nó

Operações básicas nas Árvores Binárias:

```
getNumeroNodos()  
getAltura()  
imprimePreOrdem()  
imprimePosOrdem()  
imprimeEmOrdem()
```

existem três formas básicas de percorrer a árvore: PreOrdem, PosOrdem e EmOrdem

A recursividade e o número de nodos de uma árvore com raiz

As funções recursivas têm que ter:

- 1) Condição de paragem (encontrar a situação em que encontramos o valor a devolver)
- 2) Construir a função como se ela estivesse a funcionar (“fazer função como se já funcionasse”)

Uma arvore vazia tem zero nós, se o nodo R (raiz) for Null então o número de nós é zero.

O número de nodos de uma árvore com raiz R é dado pela soma do número de nodos na subárvore esquerda com o número de nodos na subárvore direita mais um (a própria raiz)!

Exemplo:

```
int numeroNodos(Nodo raiz){  
    if(raiz == null) return 0; //condição de paragem  
    return 1+numeroNodos(raiz.esquerda)+numeroNodos(raiz.direita);  
}
```

ED (cálculo) <- Percorrer os descendentes da raiz e depois faz algo (determinar número de nodos) à raiz

Em pós-ordem: primeiro vou percorrer as subárvores do lado esquerdo e do lado direito e depois é que vou ao nodo da raiz

Exemplo: de pós-ordem void imprimir(Nodo R){ if(r==NULL) return; imprimir(r.esq); imprimir(r.dir); println(r.valor); }	Exemplo: de pre-ordem void imprimir(Nodo R){ if(r==NULL) return; println(r.valor); imprimir(r.esq); imprimir(r.dir); }	Exemplo: em ordem void imprimir(Nodo R){ if(r==NULL) return; imprimir(r.esq); println(r.valor); imprimir(r.dir); }
--	--	--

Exemplo:

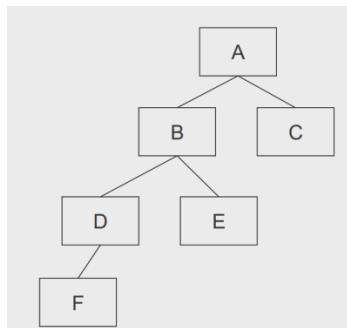
Calcular o número de nodos da subárvore da raiz, sem saber o número de nodos da arvore
Tenho que saber sempre o número de nodos

Exemplo: copiar uma arvore em pre-ordem, faz sentido porque preciso de ancorar anyes de fazer a copia das subárvore

```
void copia(Nodo R){  
    if(r==NULL) return null;  
    Nodo novo= new Nodo(r.valor);  
    novo.esq = copia(r.esq);  
    novo.dir= copia(r.dir);  
    return novo;  
}
```

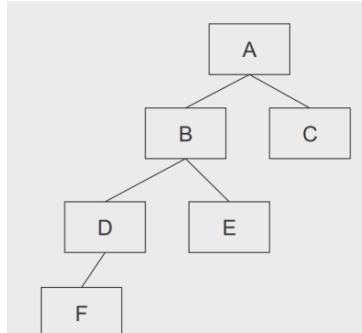
#Travessia de Árvore

Sequencia de percorrer a arvore em preordem, visito o nodo, depois o lado esquerdo e depois o lado direito



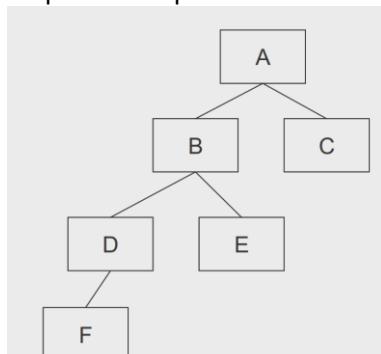
ABDFEC

Sequencia de percorrer a arvore em posordem, lado esquerdo e depois o lado direito e nodo



FDEBCA

Sequencia de percorrer a arvore em ordem, lado esquerdo, nodo, e depois o lado direito



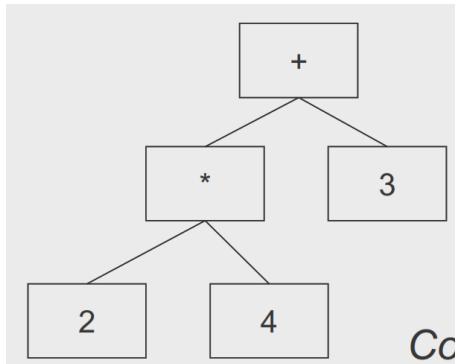
FDBEAC

#Aplicações de Árvores Binárias

Servem para representar expressões:

Exemplo:

Nodos internos representam as operações
As folhas os valores



$2*4+3$, foi obtido por ordem

```
void imprimir(Nodo n){  
if(nodo== null) return;  
imprimir(n.esq)  
print(n.valor)  
imprimir(n.dir)  
}
```

Mas para calcular é obtido por posordem, por exemplo:

```
Integer CalculaValor(){  
if (esquerda==null)&&(direita==null)  
    return Integer.valueOf(valor);  
int valEsq=esquerda.CalculaValor().intValue();  
int valDir=0;  
if(direita!=null)  
    direita=direita.CalculaValor().intValue();  
if (valor.equals("+"))  
    return valDir+ValEsq;  
if (valor.equals("*"))  
    return valDir*ValEsq;  
if (valor.equals("-"))  
    return valDir-ValEsq;  
...  
}
```

A construção de uma árvore é mais simples se tivermos a expressão representada em pos-fix. Notação pos-fix, os operadores estão depois dos argumentos e não entre eles (não se usam parêntesis)

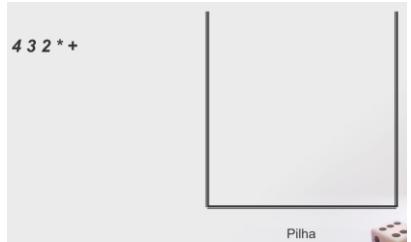
Notação in-fix, os operadores estão entre argumentos (podem usar-se parêntesis)

Exemplo:

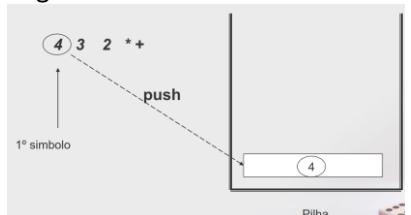
$4+3*2$ (infix) é equivalente a $4\ 3\ 2\ *\ +$ (postfix)

Exemplo: converter post-fix para árvore de $4\ 3\ 2\ *\ +$

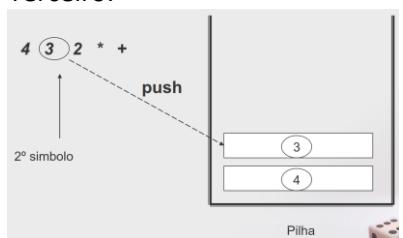
Primeiro:



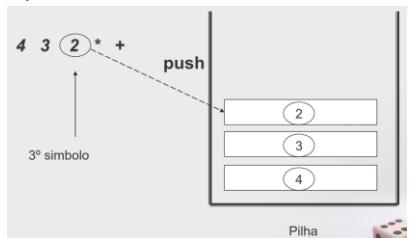
Segundo:



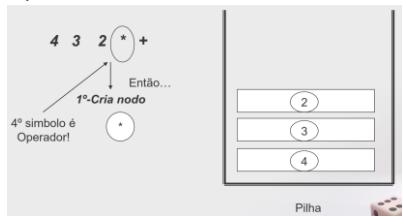
Terceiro:



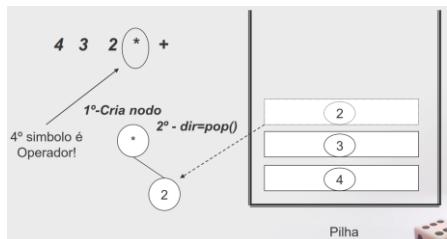
Quarto:



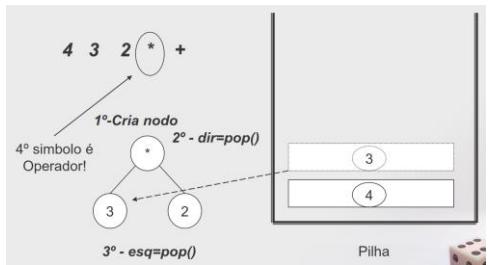
Quinto:



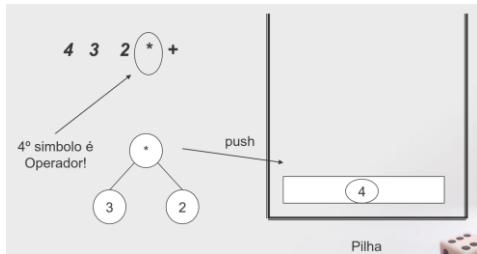
Sexto:



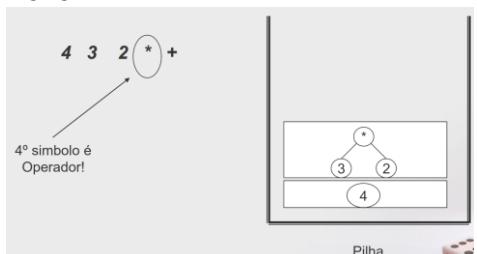
Sétimo:



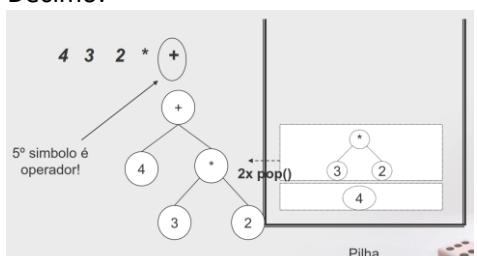
Oitavo:



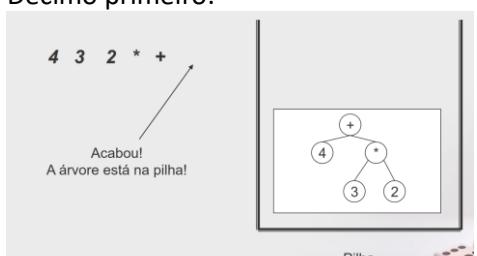
Nono:



Décimo:

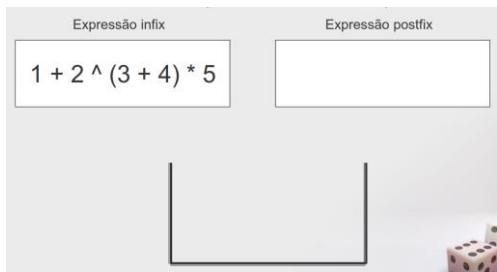


Décimo primeiro:

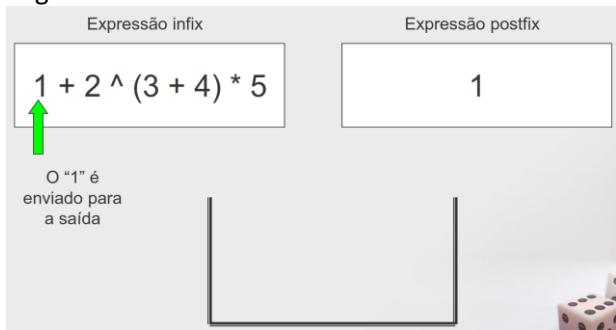


Exemplo: converter de infix para postfix, sabendo que 2 (infix) é equivalente a 4 3 2 * + (postfix)

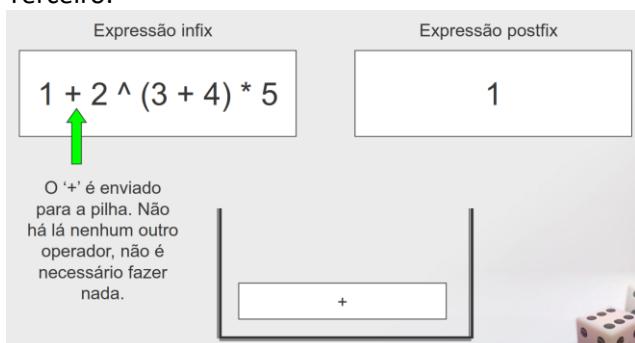
Primeiro:



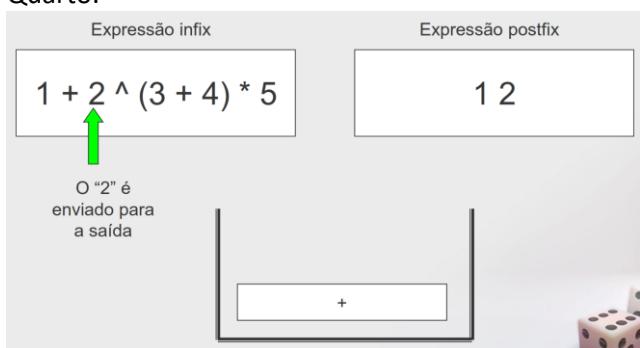
Segundo:



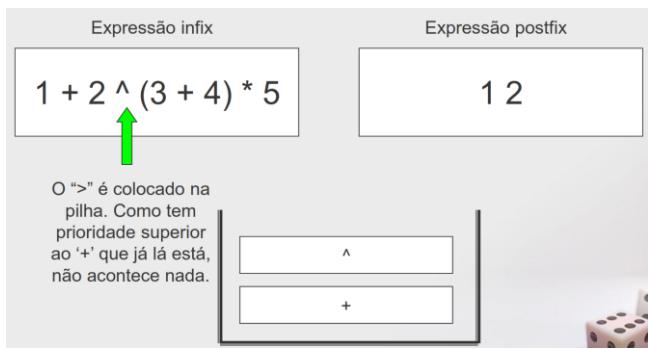
Terceiro:



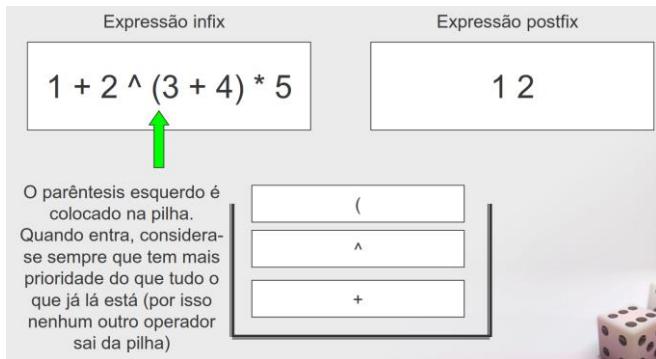
Quarto:



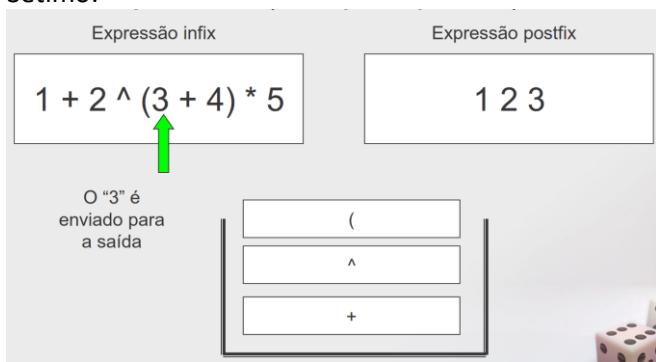
Quinto:



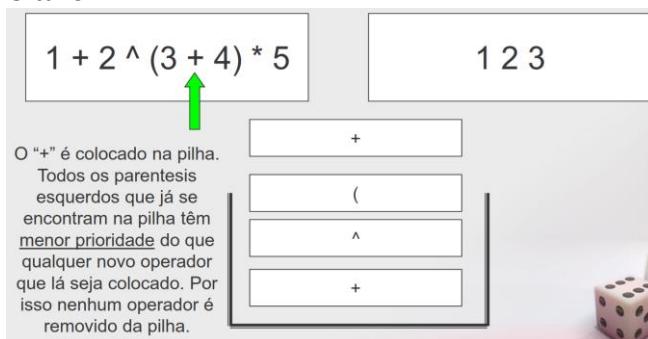
Sexto:



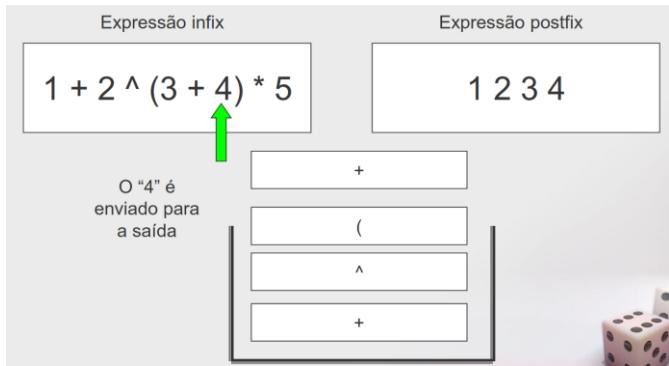
Sétimo:



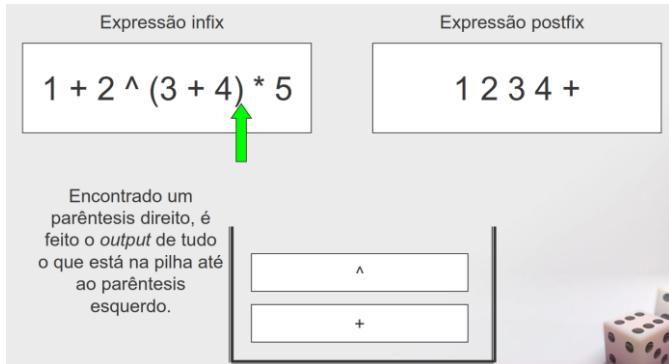
Oitavo:



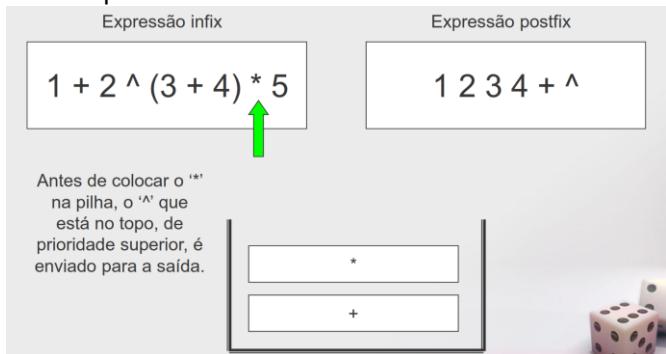
Nona:



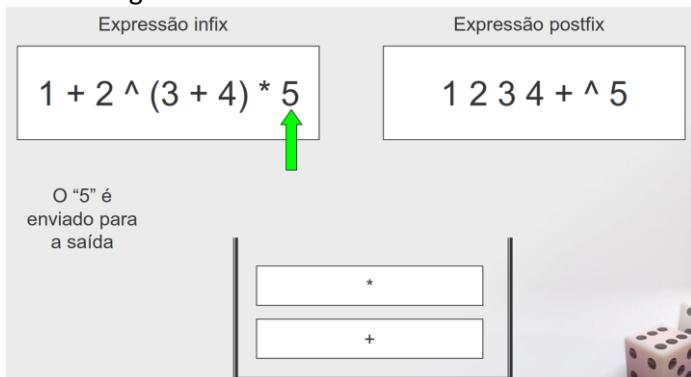
Décima:



Decima primeira:



Decima segunda:



Decima terceira:



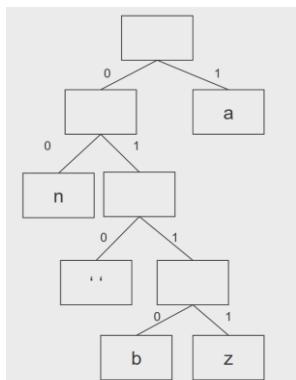
As árvores podem ser usadas na compressão de dados, árvore de Huffman
 Não existe uma só arvore, podem ter codificações diferentes

Exemplo:

Texto a comprimir:
 "banana ananaz"

Em que posso codificar:

'a' (6 ocorrências) =1
 'n' (4 ocorrências)=00
 'z' (1 ocorrência) =0111
 'b' (1 ocorrência) =0110
 '' (1 ocorrência)=010



Texto recodificado:
 "0110100100101010010010111"

A árvore de Huffman é construída de baixo para cima, e agregamos subárvores de acordo com a frequênciados símbolos que a subárvore representa.

E quando se escolhe que duas subárvores que devemos juntar numa nova arvore, vamos escolher as duas subárvores que tenham menos símbolos. Os empates são resolvidos ao calhas

Exemplo:
 ORANGOTANGO

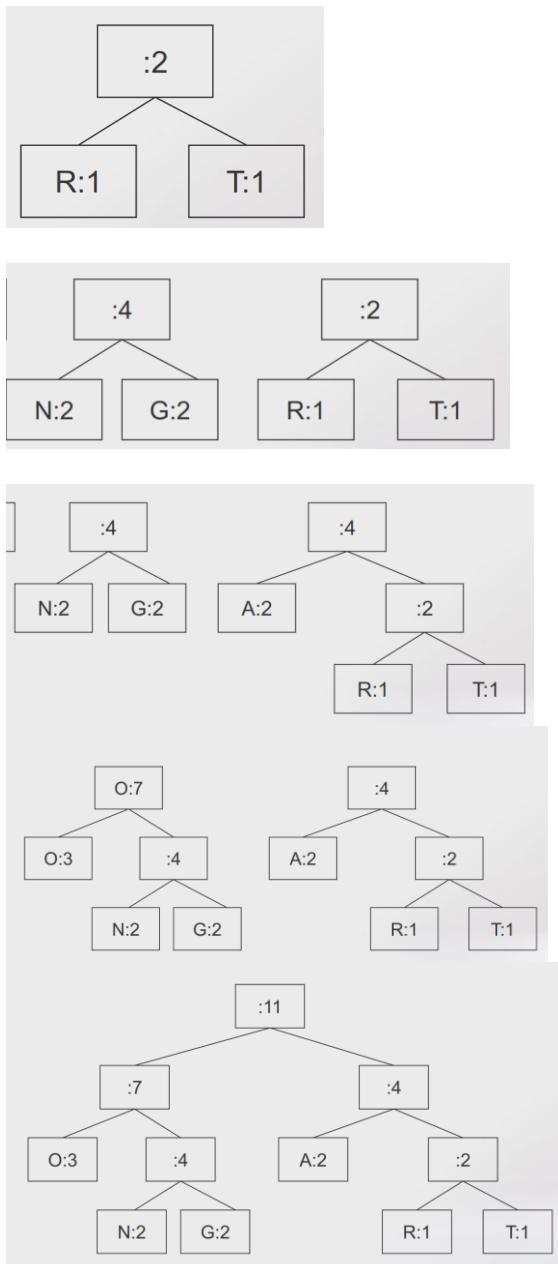
1) efetuar o calculo a frequênciadas letras

O 3

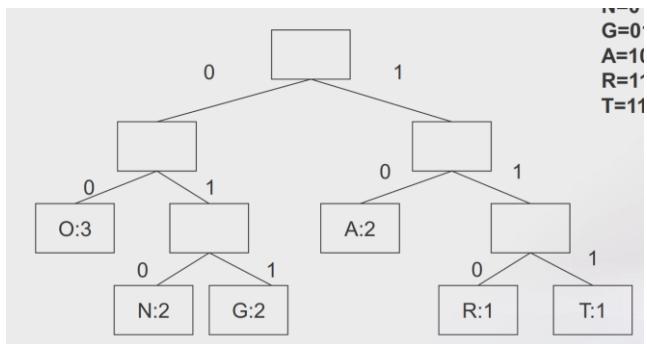
A 2

N 2
G 2
R 1
T 1

2) Agrupam-se dois dos elementos com menor frequência numa única árvore



As folhas da árvore vão corresponder sempre a símbolos do texto
A codificação é dado por 0 (esquerda) e 1 (direita)



O=00
N=010
G=011
A=10
R=110
T=111

#Arvores binárias de pesquisa

São usadas para criar estrutura de dados que tem tempos de inserção e pesquisa tempo logarítmico, porque a altura de uma árvore de N nodos está relacionado com Log de N

É uma arvore binária tem como critério inicial que é:

propriedade de ordem,

Todos os nodos na sub-árvore esquerda de um nodo raiz são menores do que a essa raiz

Todos os nodos na sub-árvore direita de um nodo raiz são maiores do que essa raiz.

E não existem valores iguais

E os valores iguais?

Não existe sítio para os colocar

Se for necessário: ter um contador no nodo, ou ter uma estrutura auxiliar no nodo que conta

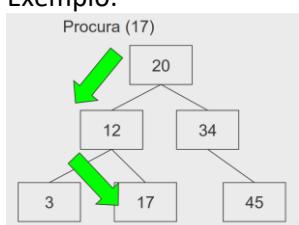
Travessia de árvore equilibrada:

Caso uma árvore de pesquisa equilibrada seja percorrida em ordem, os elementos são visitados por ordem crescente.

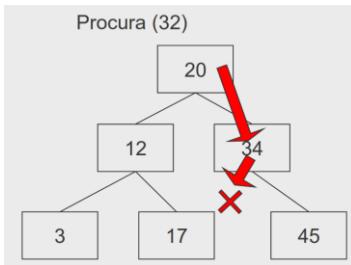
Como pesquisar um valor em árvores binárias equilibradas:

Para procurar o valor X numa árvore binária com raiz R, verificasse R contém X. Caso isso não aconteça, procura-se na sub-árvore esquerda ou direita conforme o elemento a procurar seja menor ou maior, respectivamente, do que o valor da raiz.

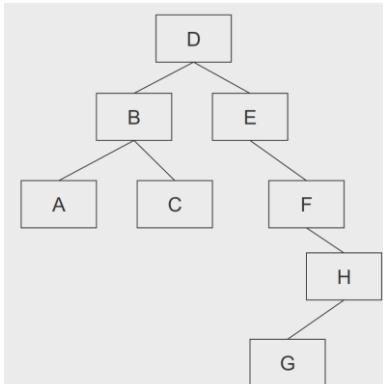
Exemplo:



Exemplo:



Onde está o menor valor:



A

Algoritmo de pesquisa do Elemento Mínimo:

```

private BinaryNode<AnyType> findMin(BinaryNode<AnyType> t )
{
    if (t==null)
        return null;
    if(t.left==null)
        return t;
    else
        return findMin(t.left);
}

```

Algoritmo de pesquisa:

```

private BinaryNode<AnyType> find( AnyType x, BinaryNode<AnyType> t )
{
    if( t != null )
    {
        if( x.compareTo( t.element ) < 0 )
            return find(x,t.left);
        else
            if( x.compareTo( t.element ) > 0 )
                return find(x,t.right);
            else
                return t;
                // Encontrado
                //(Resultado menos provável testado em último lugar)
    }
    return null;
}

```

[aula06]

Árvores de Pesquisa Binárias: a Inserção

Caso a árvore esteja vazia, cria uma raiz nova.

Caso contrário:

Se for igual à raiz, gera exceção.

Se for maior do que a raiz, insere na subárvore direita

Se for menor do que a raiz, insere na subárvore esquerda.

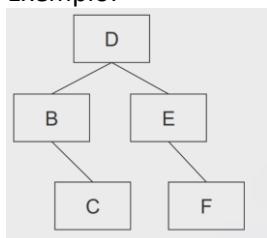
```
protected BinaryNode<AnyType> insert ( AnyType x, BinaryNode<AnyType> t )
{
    if( t == null )
        t = new BinaryNode<AnyType>( x ); //condição de saída
    else
        if( x.compareTo( t.element ) < 0 )
            t.left = insert( x, t.left );
        else if( x.compareTo( t.element ) > 0 )
            t.right = insert( x, t.right );
        else
            throw new DuplicateItemException( x.toString() ); // Duplicado, condição de
            saída
    return t; //nova raiz desta sub-árvore
}
```

“Nodo, é a nova raiz da arvores depois da inserção”

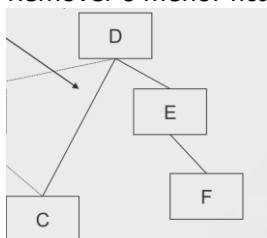
Árvores de Pesquisa Binárias: a remoção

O caso simples: remover o menor valor de uma subárvore

Exemplo:



Remover o menor fica



A função para remover o mínimo pode alterar a raiz da arvore?

Sim, se o menor for a raiz, assim a função devolve um Nodo e não Void, e vai ser usada a recursividade

```
public void removeMin()
```

```

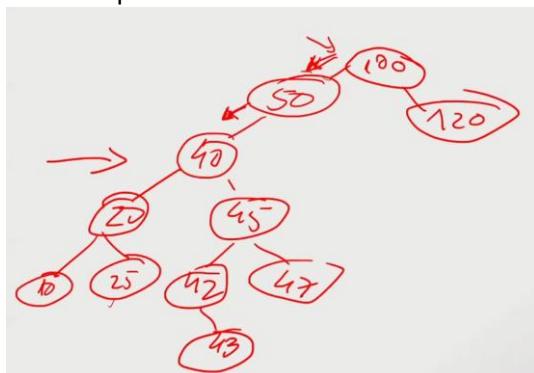
{
    root=removeMin(root);
}

private BinaryNode<AnyType> removeMin( BinaryNode<AnyType>t)
{
    if( t == null )
        throw new ItemNotFoundException( );
    else
        if( t.left != null )
        {
            t.left = removeMin( t.left );
            return t;
        }
        else
            return t.right;
}

```

Fases para remoção: do 40

Primeiro passo é encontrar onde está o Nodo



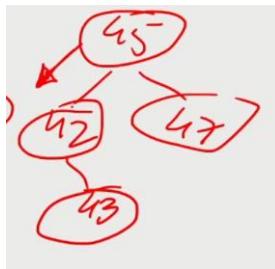
Procurar o valor a seguir ao 40:



Que está na subárvore da direita

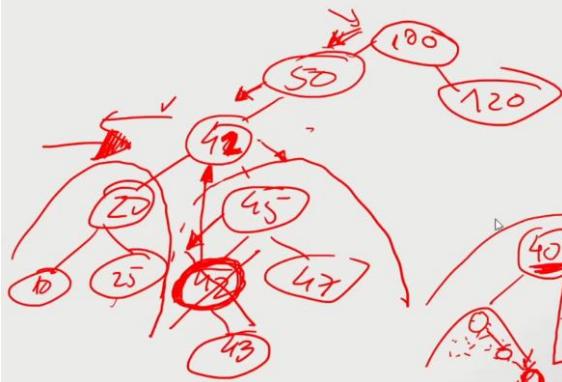


E procurar o menor valor subárvore da direita



Neste caso o 42 é o menor valor

Copiamos o 42 para o local do 40, não se cria um Nodo novo



Copiar nodo menor da subárvore direita, substituindo o 40 neste caso

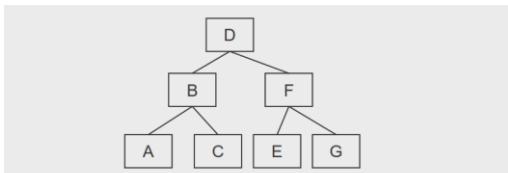
O algoritmo:

```
protected BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
{
    if( t == null )
        throw new ItemNotFoundException( x.toString() );
    if( x.compareTo( t.element ) < 0 )
        t.left = remove( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = remove( x, t.right );
    else
        if( t.left != null && t.right != null ) // Dois descendentes
        {
            t.element = findMin( t.right ).element;
            t.right = removeMin( t.right );
        }
        else
            t = ( t.left != null ) ? t.left : t.right; // Um só descendente
    return t;
}
```

Complexidade das operações

Proporcional ao número de Nodos por onde passamos.

Ou O(log N)



Árvore equilibrada: $O(\log N)$ no pior caso
Se queremos uma árvore equilibrada as operações são sempre $O(\log N)$

Ou $O(N)$, árvore desequilibrada

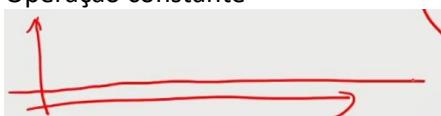
Árvore desequilibrada: $O(N)$ no pior caso



Esta árvore ficou assim porque os valores estão de forma crescente
E será um cenário comum? Sim infelizmente, a árvore binária então comporta-se mal
O principal problema reside no facto de a introdução de sequências ordenadas (ou semi-ordenadas) é uma acção relativamente frequente, resultando directamente na redução ao pior caso possível.

Relembrar que:

Operação constante



Operação linear



Operação logarítmica



Uma operação é constante numa árvore só quando accedemos à raiz

#Propriedade de ordem (relembrar)

Faz com que a inserção e a procura de valores seja eficiente ou que seja simples, permite fazer um percurso direto da raiz até aos valores que pretendo

#Propriedade de equilíbrio

Para além da propriedade de ordem, pode ser também adicionada uma propriedade adicional que assegura o equilíbrio da árvore.

Nenhum nodo poderá atingir uma profundidade demasiado elevada em relação aos outros.

Há diferentes variantes da propriedade de equilíbrio (por esta via surgem as árvores binárias auto balanceadas).

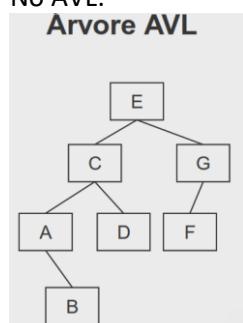
As operações de inserção e remoção tornam-se mais complicadas, mas a operação de pesquisa fica mais eficiente.

Existem várias propriedades de equilíbrio, sendo que:

A propriedade de equilíbrio mais simples é requerer que as subárvores direita e esquerda tenham a mesma profundidade se possível. Esta abordagem pode implicar modificar todos os Nodos da árvore, o que implica demorar muito e complicado de ser feito.

Surge então a propriedade de equilíbrio AVL. Para qualquer nodo na árvore, a profundidade das subárvores direita e esquerda só pode diferir em 1 (a profundidade de uma subárvore vazia é -1).

No AVL:



Profundidade do lado esquerdo de E é 3

Profundidade do lado direito de E é 2

3-2=1, esta equilibrado

Profundidade do lado esquerdo de C é 2

Profundidade do lado direito de C é 1

2-1=1, esta equilibrado

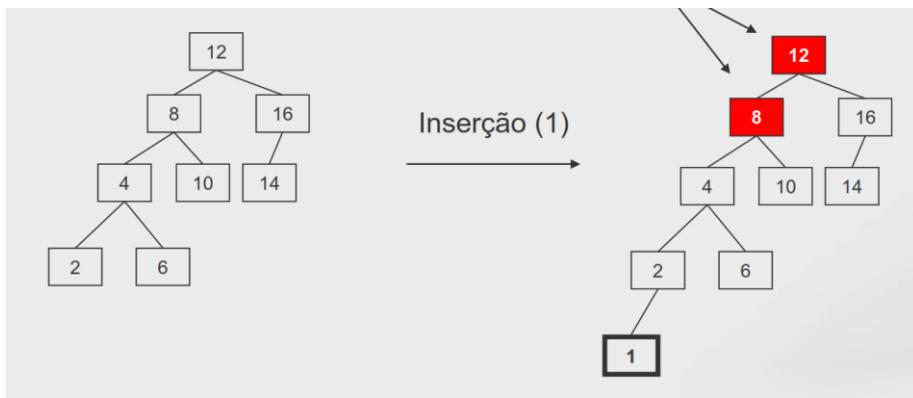
Profundidade do lado esquerdo de G é 1

Profundidade do lado direito de C é 0

1-0=1, esta equilibrado

Todos os nós estão equilibrados.

Exemplo: inserir o 1

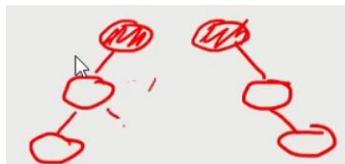


Deixou de haver profundidade com diferenças de 1

Depois da inserção, torna-se necessário efetuar uma operação adicional que assegura a preservação equilíbrio da árvore.

Os desequilíbrios (que surge do percurso que foi feito até chegar ao nó desequilibrado) podem resultar dos cenários:

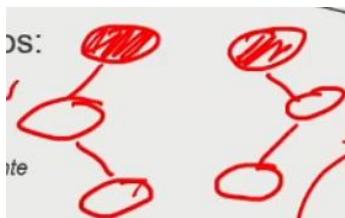
Ou



E vai ser uma inserção externa

O resultado de uma inserção exterior pode ser equilibrado com uma única rotação da árvore.

Ou

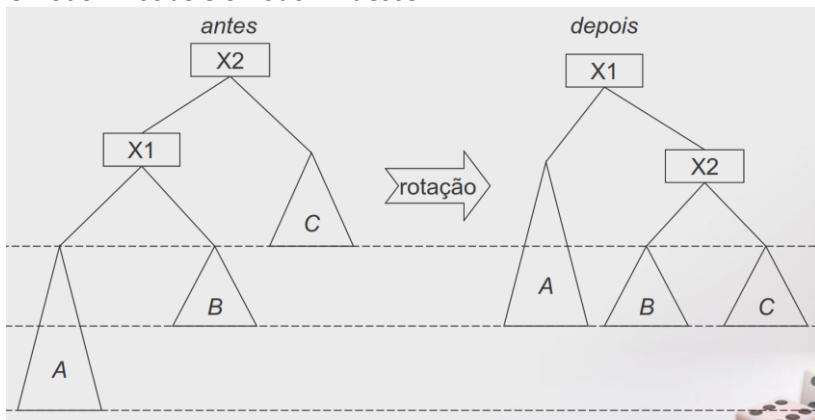


E vai ser uma inserção interna

O resultado de uma inserção interior pode ser equilibrado com duas rotações da árvore.

Face à inserção externa a rotação é feita entre o Nodo e o seu descendente

O nodo X1 sobe e o nodo X2 desce



E na rotação tudo o resto que se possa manter igual fica igual

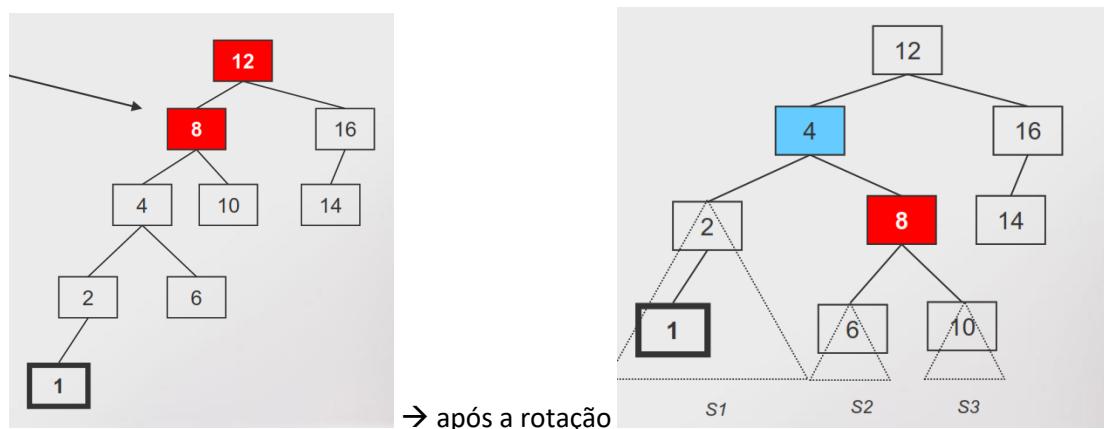
Exemplo:

Foi inserido o 1

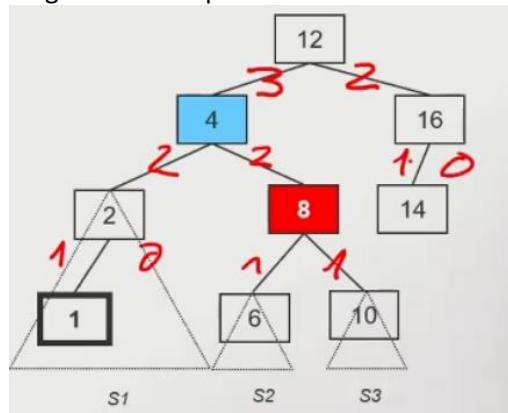
Faz-se o percurso para cima para verificar

Existe um desequilíbrio no Nodo 8

Rodar o 8 com o 4

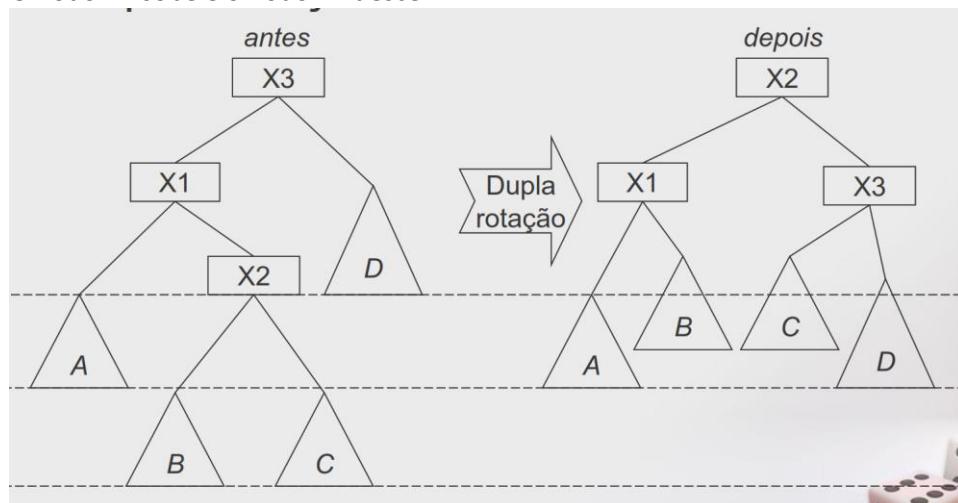


E agora existe equilíbrio



Face à inserção interna são feitas duas rotações consecutivas

O nodo X1 sobe e o nodo X2 desce



E na rotação tudo o resto que se possa manter igual fica igual

Exemplo:

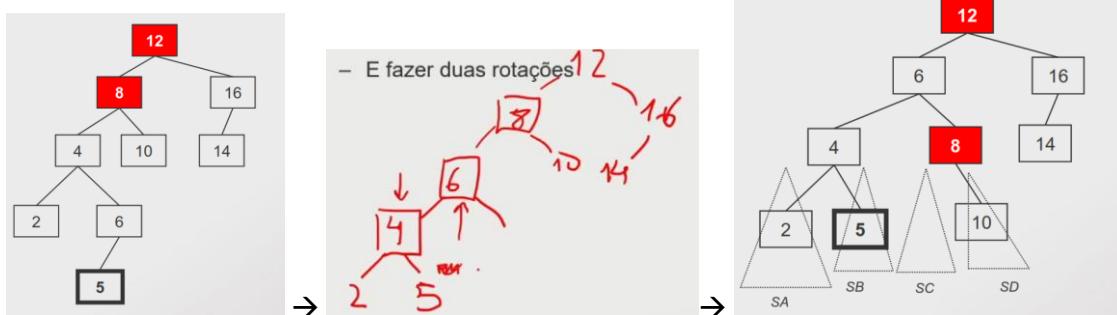
Foi inserido o 5

Faz-se o percurso para cima para verificar

Existe um desequilíbrio no Nodo 8

Rodar o 4 com o 6

Rodar o 6 com o 8



Exemplo:

1,2,3,4,5,6,7

Inserir 1

Inserir 2

Inserir 3

Está desequilibrada

O nó desequilibrado é 1

Rotação entre 1 e 2

Inserir o 4

Inserir o 5

Está desequilibrada

O nó desequilibrado é 3

Rotação entre 3 e 4

Inserir o 6

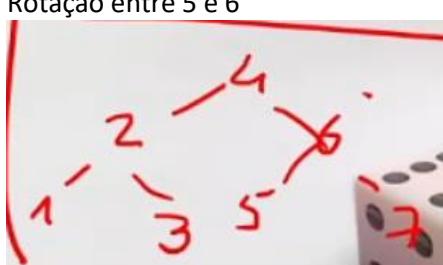
O nó desequilibrado é 2

Rotação entre 2 e 4

Inserir o 7

O nó desequilibrado é 5

Rotação entre 5 e 6



[aula07]

ED08 - Red Black.pdf

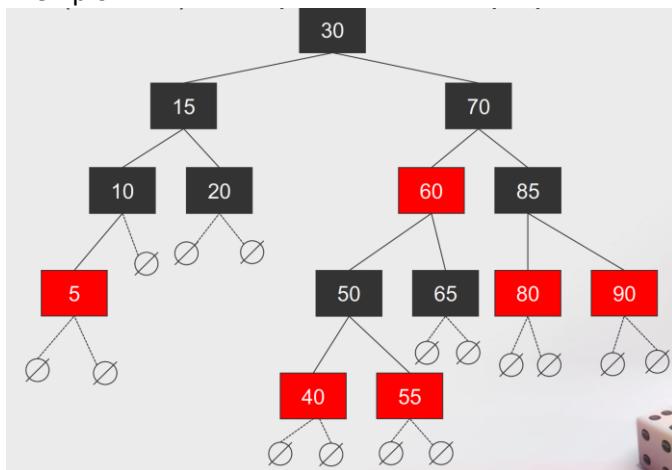
Quando mais exigente o critério de equilíbrio mais próximo estamos da situação ideal, mas vamos ter que fazer mais correções (rotações no ADL) para garantir que a árvore esta equilibrada

Em comparação com as árvores AVL, as árvores red-black têm uma maior eficiência de inserção, por via da utilização de uma propriedade de equilíbrio ligeiramente mais permissiva.

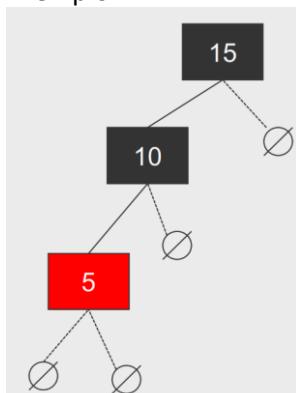
A propriedade de Equilíbrio (árvore red-black) implica que:

1. Todos os nodos são coloridos (vermelhos/pretos)
2. A raiz é preta.
3. Se um nodo é vermelho, ambos os descendentes devem ser pretos (os nodos nulos são considerados pretos).
4. Todos os percursos entre a raiz e qualquer um nodo nulo devem passar pelo mesmo número de nodos pretos

Exemplo:



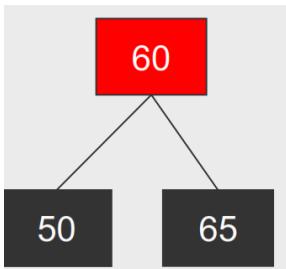
Exemplo:



Só passamos por um nodo preto para chegar ao nodo nulo

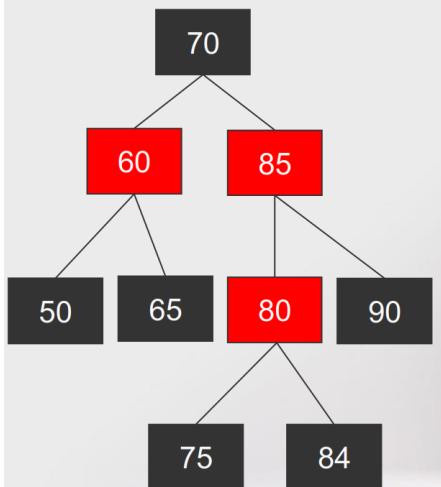
“Todos os percursos entre a raiz e qualquer um nodo nulo devem passar pelo mesmo número de nodos pretos”

Exemplo:



“A raiz não é preta”

Exemplo:



“Se um nodo é vermelho, ambos os descendentes devem ser pretos”

Na propriedade de equilíbrio:

A árvore tem que conter pelo menos $2^B - 1$ nodos pretos, onde B é o número de nodos pretos entre a raiz e uma folha.

A altura da árvore é, no máximo, $2 \log(N+1)$, A pesquisa é logarítmica.

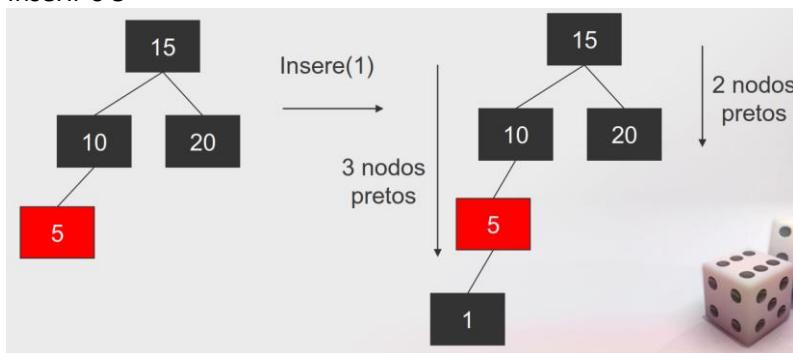
RB: Inserção de um nodo

A cor pode ser vermelha ou preto

Inserir um nó preto a arvore esta desequilibrada, porque vamos desequilibrar a árvore, já que não se verifica “Todos os percursos entre a raiz e qualquer um nodo nulo devem passar pelo mesmo número de nodos pretos” assim à partida os nos inseridos vão ser vermelhos

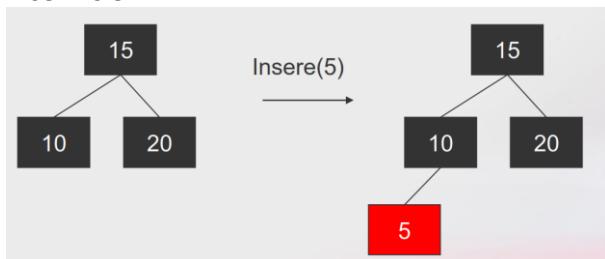
Exemplo:

Inserir o 5



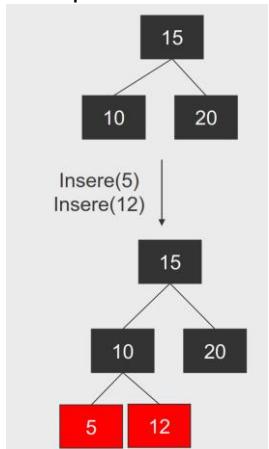
Exemplo:

Inserir o 5

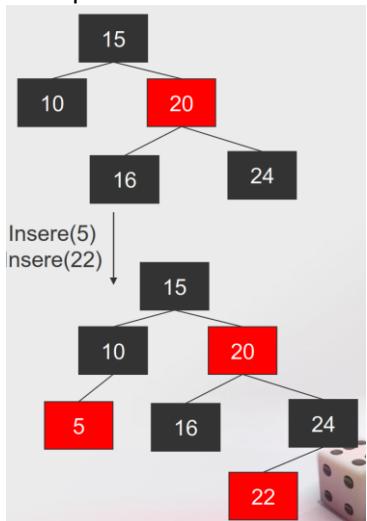


Todas as propriedades de equilíbrio estão verificadas

Exemplo:

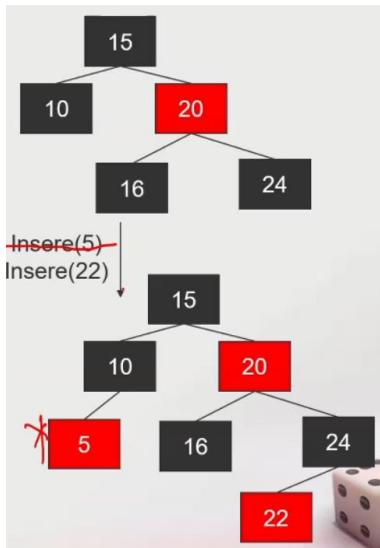


Exemplo:



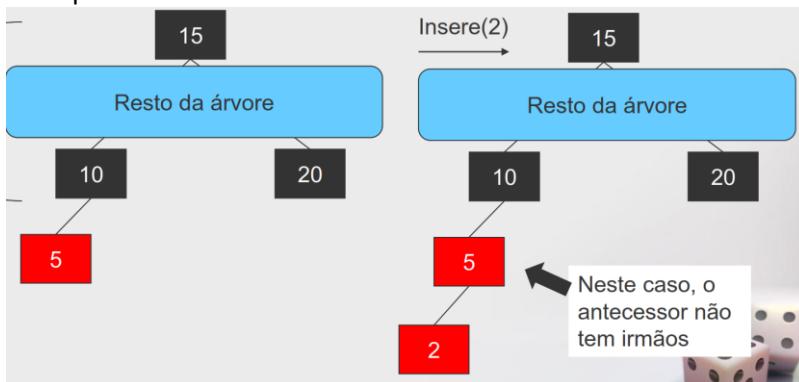
Exemplo 3:

Só inserimos o 22

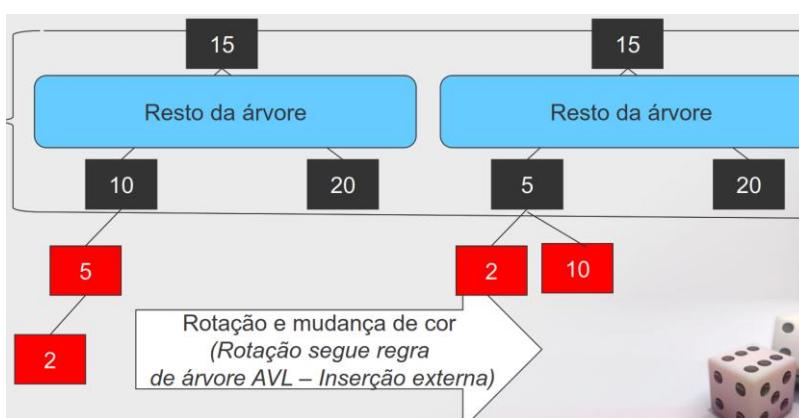


O caminho mais curto é 2 nos pretos
 O caminho mais comprido possível é 4
 isto ser o mais desequilibrado que a árvore pode ficar

Exemplo:



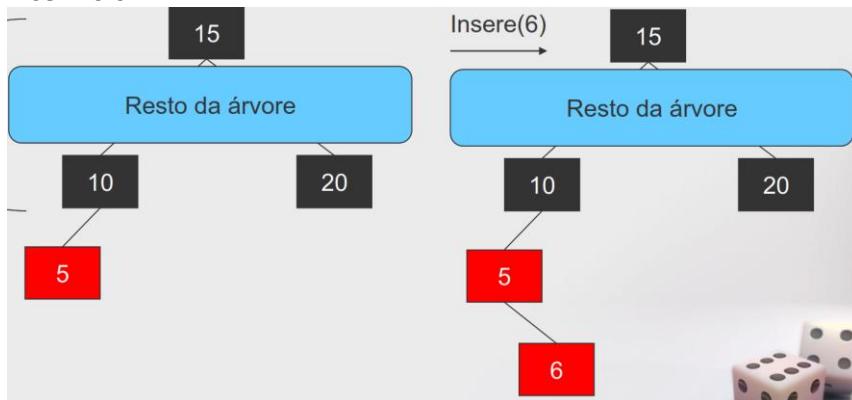
Temos um desequilibrio o 5 e 2 são vermelhos
 Existe alguma forma de arrumar os nodos 10-5-2 para equilibrar



Rotação do 10 com 5
 E mudar a cor de 5
 Mudar a cor de 10
 É a mesma regra da AVL, inserção externa

Exemplo:

Inserir o 6



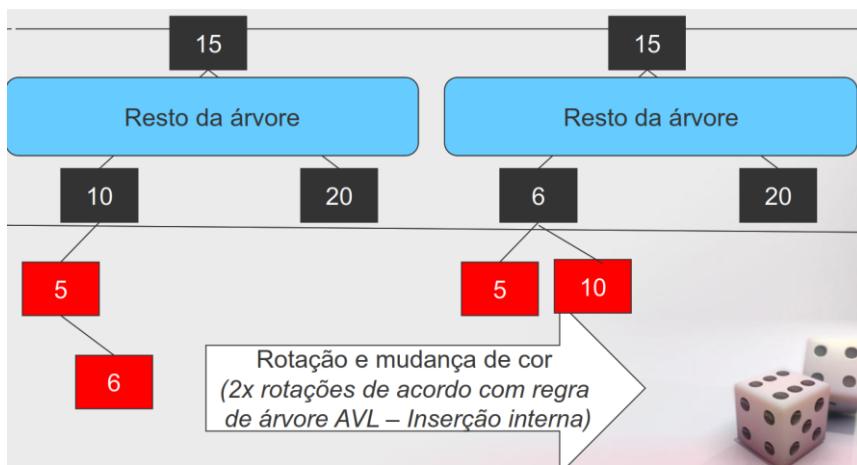
Aplicar rotação interna, duas rotações

Aplicar uma rotação entre o 5 e 6

Aplicar a rotação entre 10 e 6

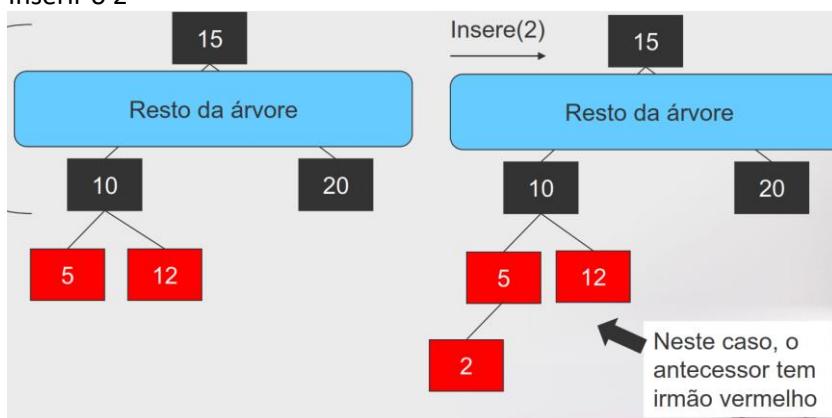
Muda de cor o 6

Muda de cor o 10



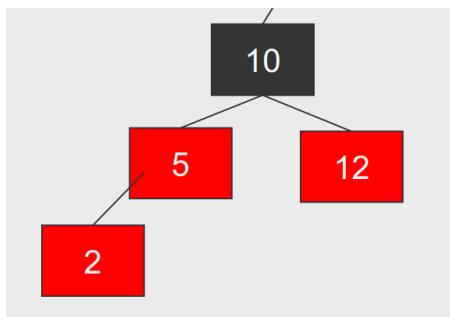
Exemplo:

Inserir o 2



Não existe nenhuma solução boa, temos que adotar e olhar para cima da arvore.. vamos ter que adotar outra solução.

Devemos antecipar as alterações à arvore, não vai poder acontecer algo do género:



Vamos então:

Fazer alterações à arvore depois de inserir o novo valor

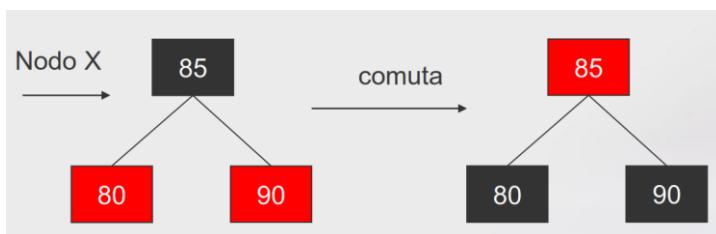
E fazer também alterações quando estamos à procura do local de inserção

Pois o problema normalmente acontece quanto um nodo tem dois descendentes vermelhos

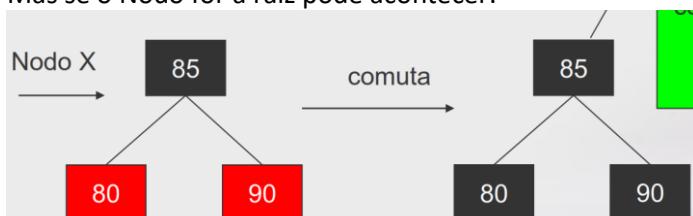
A estratégia:

À medida que procuramos no local de inserção fazemos modificações, comutamos as cores, sempre que passamos por Nodo preto com dois descendentes vermelhos

Se temos um Nodo preto não interessa o que vem acima dele

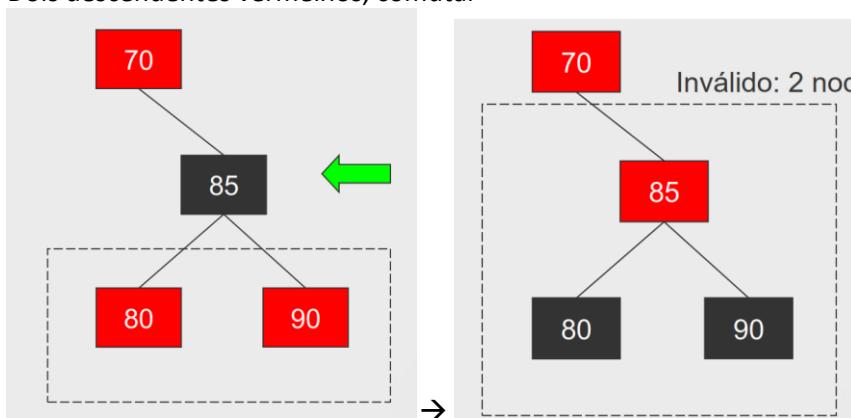


Mas se o Nodo for a raiz pode acontecer:



Exemplo:

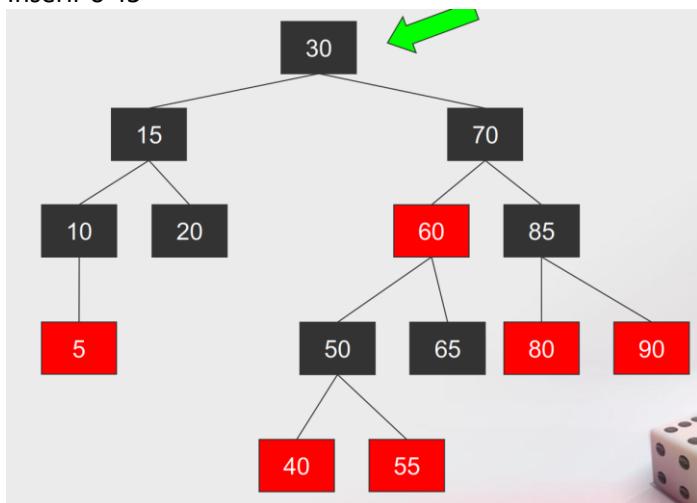
Dois descendentes vermelhos, comutar



Dois nodos vermelhos não é válido

Exemplo:

Inserir o 45



O 30 não é um Nodo preto com dois descendentes vermelhos
45 > 30, vamos para o lado direito

O 70 não é um Nodo preto com dois descendentes vermelhos
45 <70, vamos para o lado esquerdo

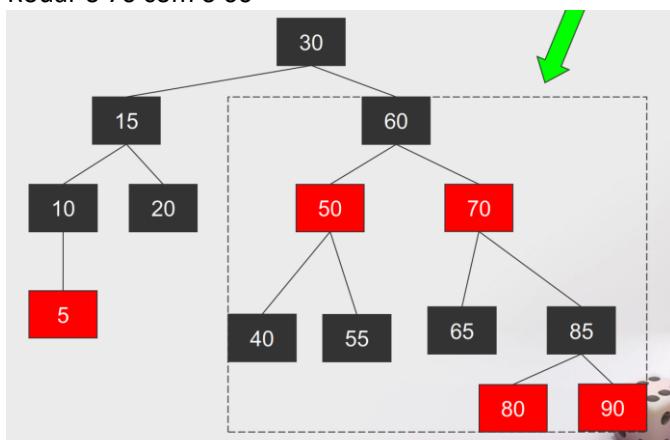
O 60 não é um Nodo preto com dois descendentes vermelhos
45 <50, vamos para o lado esquerdo

O 50 é um Nodo preto com dois descendentes vermelhos

Comutar as cores do 50, 40, e 55

Ficamos com o nodo 60 e 50 vermelhos

Rodar o 70 com o 60

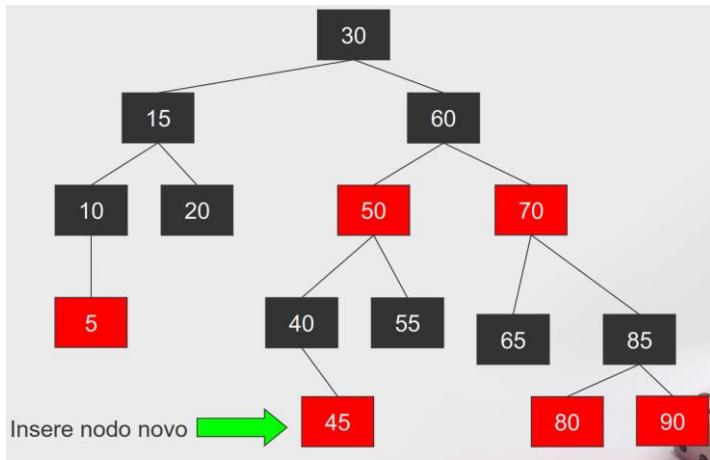


Continuar à procura para inserir o 45, estamos no nodo 50

45 <50, vamos para o lado esquerdo

O 40 é um Nodo preto sem descendentes

45 é inserido como nodo vermelho



Exemplo:

Árvore vazia

Inserir o 1,2,3,6,4,5,7,8

Todas as inserções são feitas com nodos vermelhos

Inserir o 1 vermelho

Mas a raiz é sempre preta

Comuta cor do 1

1 não tem descendentes RR

Inserir o 2, vermelho

2 > 1, direita

1B, 2R

1 não tem descendentes RR

2 não tem descendentes RR

Inserir o 3, vermelho

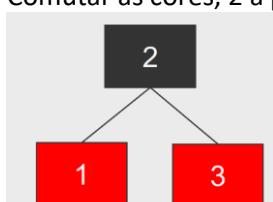
3 > 1, direita

3 > 2, direita

A arvore não esta equilibrada, dois vermelhos consecutivos

Aplicar uma rotação simples, no avô, 1 com o 2

Comutar as cores, 2 a preto, 1 a vermelho

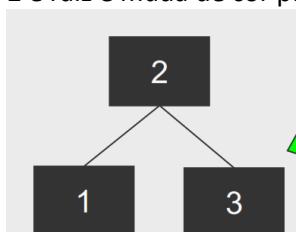


Inserir o 6

2, Dois descendentes vermelhos, é preciso comutar a cores

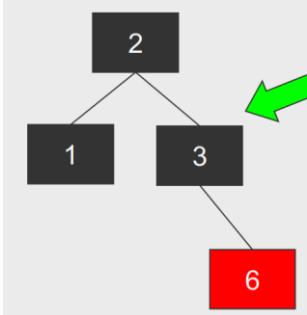
2 vermelho, 1 e 3 preto

2 é raiz e muda de cor para preto

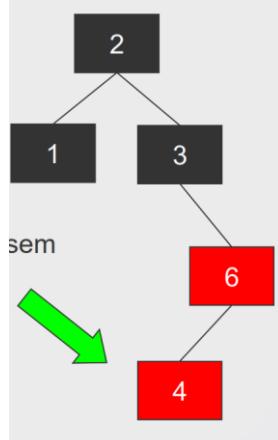


2 não tem descendentes RR

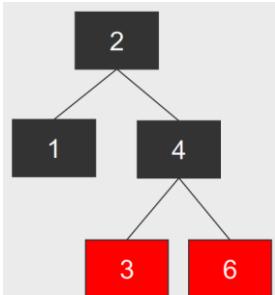
6 > 2, direita
 3 não tem descendentes RR
 6 > 3, direita
 Inserir o 6, vermelho



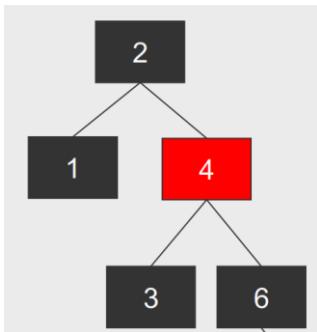
Inserir o 4
 2 não tem descendentes RR
 4 > 2, direita
 3 não tem descendentes RR
 4 > 3, direita
 6 não tem descendentes RR
 4 < 6, esquerda
 Inserir o 4, vermelho
 A arvore não esta equilibrada, dois vermelhos consecutivos



Aplicar primeira rotação, entre o 6 e o 4
 Aplicar segunda rotação, entre o 3 e o 4
 O Nodo do topo fica preto, comuta cor
 Inserir o 7



2 não tem descendentes RR
 7 > 2, direita
 4 tem descendentes RR
 Comutar as cores, o 4 passa a vermelho, e o 3 e 6 a preto



4 não tem descendentes RR

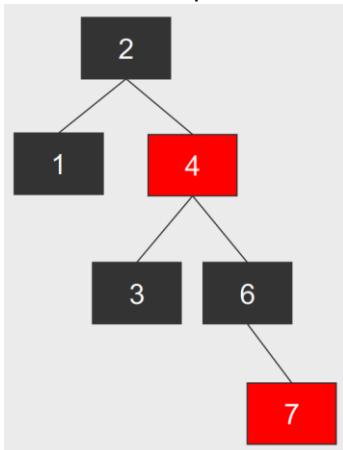
7 > 4, direita

6 não tem descendentes RR

7 > 6, direita

Inserir o 7, vermelho

A árvore está equilibrada: verificar as quatro propriedades



Inserir o 5

2 não tem descendentes RR

5 > 2, direita

4 não tem descendentes RR

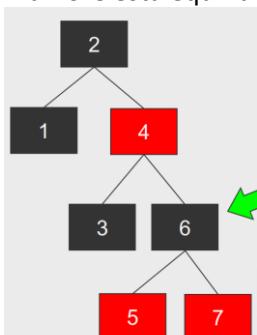
5 > 4, direita

6 não tem descendentes RR

5 < 6, esquerda

Inserir o 5, vermelho

A árvore está equilibrada: verificar as quatro propriedades



Inserir o 8

2 não tem descendentes RR

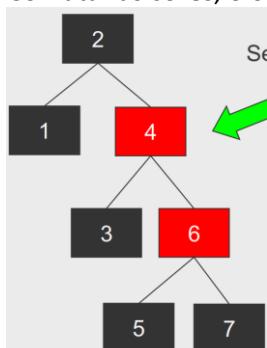
8 > 2, direita

4 não tem descendentes RR

8 > 4, direita

6 tem descendentes RR

Comutar as cores, o 6 passa a vermelho, e o 5 e 7 a preto



Existem dois nodos vermelhos, o 4 e 6

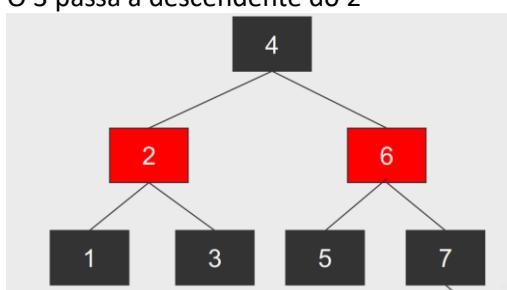
Aplicar uma rotação e rodar o 2 e o 4

4 passa a preto

O 2 passa a descendente do 4

O 3 não tem sitio para estar, e para ficar bem

O 3 passa a descendente do 2



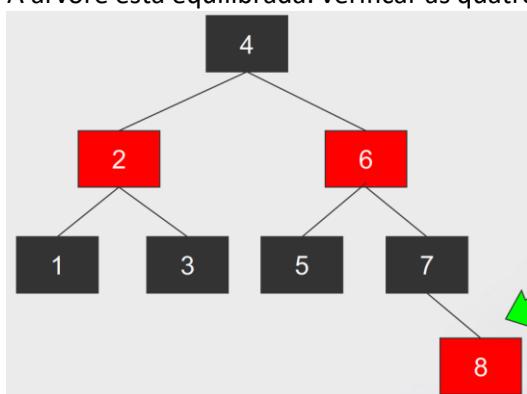
Estamos no 6

6 não tem descendentes RR

8 > 6, direita

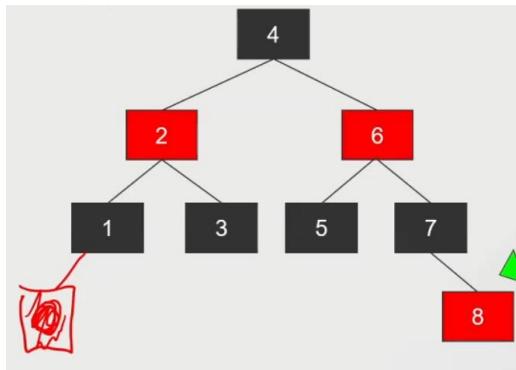
Inserir o 8, vermelho

A árvore está equilibrada: verificar as quatro propriedades



Exemplo:

O que acontece quando inserimos o zero



A árvore está equilibrada: ao verificar as quatro propriedades, mas quando começamos pela raiz temos dois descendentes vermelhos, 4 tem descendentes RR, e assim 2 e 6 passam para preto
 O 4 passa para vermelho
 Como o 4 é raiz, passa para preto
 E só depois é que inserimos o 0 vermelho

Resumo:

As árvores AVL têm uma profundidade ligeiramente menor, em termos teóricos

A inserção em árvores Red-Black é mais eficiente

Em Red-Black a implementação é complicada não só pela simetria como também pelos inúmeros casos especiais

Em Red-Black para simplificar a implementação, são usadas duas sentinelas

As duas sentinelas são:

NullNode: Existe um (único) nó que representa um nodo nulo. Este nodo é sempre preto.
 Logo, não há links nulos.

Header: Pseudo-raiz (raiz da raiz). Tem um valor de $-\infty$ e o seu descendente direito é a raiz da árvore.

Árvore Red-Black, pesquisa com sentinelas:

No inicio da pesquisa, o nodo nulo é inicializado com o valor a pesquisar!

Desta forma evitasse fazer uma verificação a necessidade de testar em todas iterações testar se é `!=Null`

Não é necessário ver o restante como é o caso da construção do algoritmo

Árvores auto balanciadas:

AVL

Red-Black

[aula08]

ED09-BTreeSplay.pdf

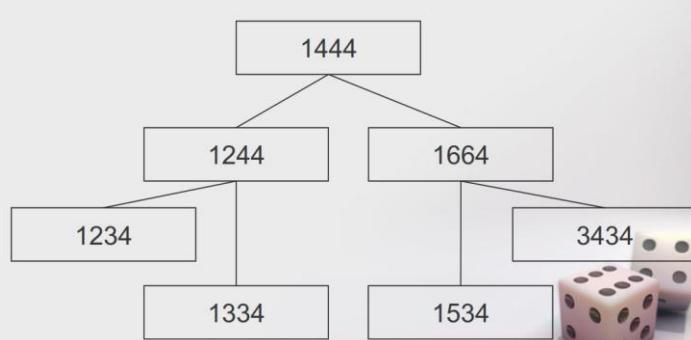
E uma árvore equilibrada para memoria persistente?

Exemplo:

Ficheiro Monolítico

1234
1244
1334
1444
1534
1664
3434

Estrutura em árvore



Contudo estar em memoria ou estar em disco os acessos são diferentes, em disco são acessos lentos. Então devemos algoritmos que reduzam a profundidade da arvore para se ter menos acessos.

Surgem então o tipo de árvore M-ária, onde são guardados vários (M) items de dados em cada nodo. Por exemplo a B-Tree

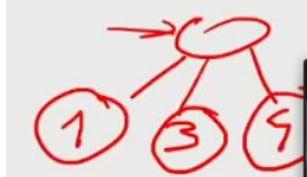
#B-Tree

Uma B-tree de ordem M é uma árvore M-ária que verifica as seguintes propriedades:

1. Os dados são guardados nas folhas
2. Cada nodo não-folha guarda até M-1 chaves de pesquisa: Cada chave i representa o menor índice na subárvore i+1.
3. A raiz é uma folha ou tem entre 2 e M descendentes.
4. Cada nodo não-folha tem entre M/2 (arredondado para cima) e M descendentes.
5. Todas as folhas têm entre L/2 (arredondado para cima) e L items de dados

Exemplo:

1. Os dados são guardados nas folhas



2. Cada nodo não-folha guarda até M-1 chaves de pesquisa



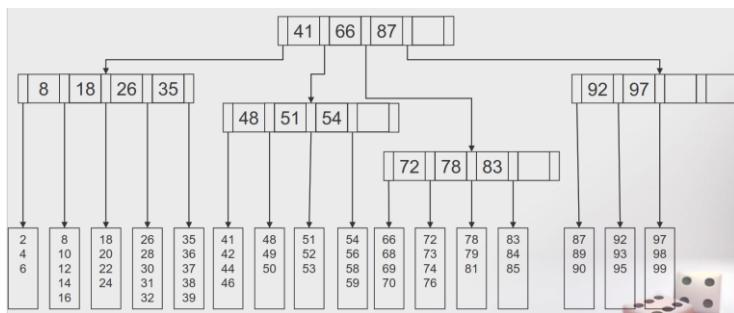
Exemplo:

Valores de 2 a 99

Em cada folha existem 5 valores

No nós intermédios surgem alguns dos valores

Descendentes, cada Nodo tem até 5 descendentes



#BTree

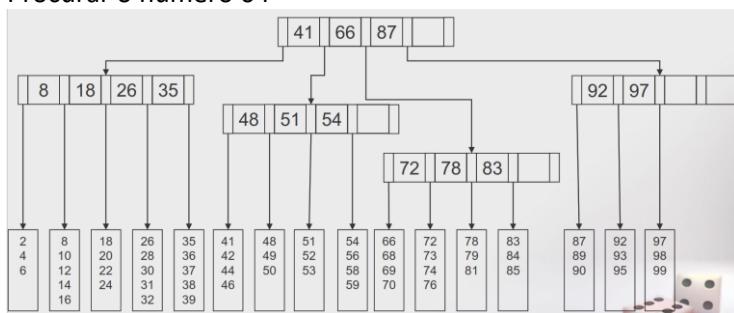
M número de descendentes

L número de valores em cada nodo

Objetivo maximizar os acessos aos discos

Exemplo:

Procurar o número 64



Começar na raiz

Para saber qual a subárvore procurar o índice

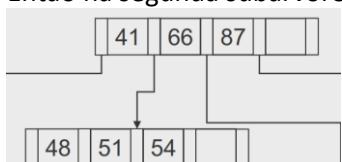


41 é o menor valor de uma subárvore

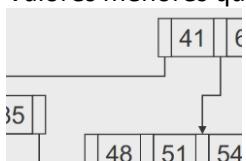
66 é o menor valor de uma subárvore

87 é o menor valor de uma subárvore

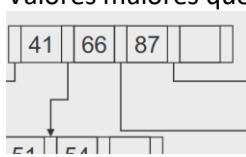
Então na segunda subárvore sabemos que os valores vão estar entre 41 e 65

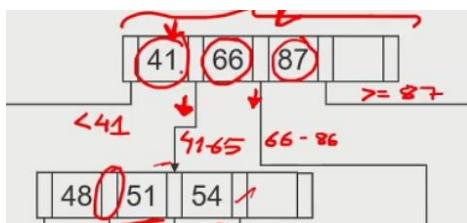


Valores menores que 41



Valores maiores que 61





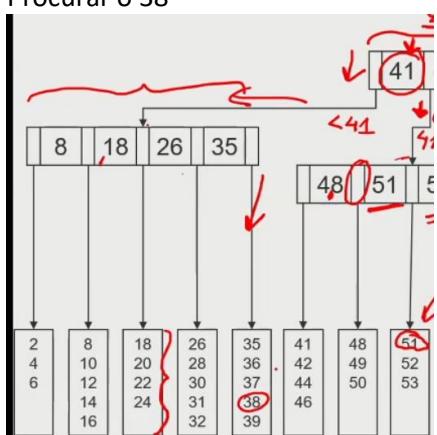
Vou usar o ramo de 41-65 para procurar o 64



Percorro a folha, 54, 56,... e verifico que o valor 64 não existe.

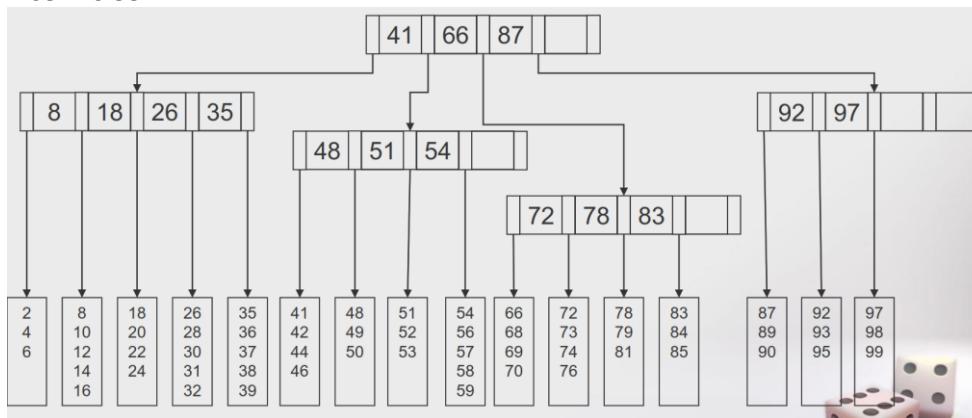
Exemplo:

Procurar o 38



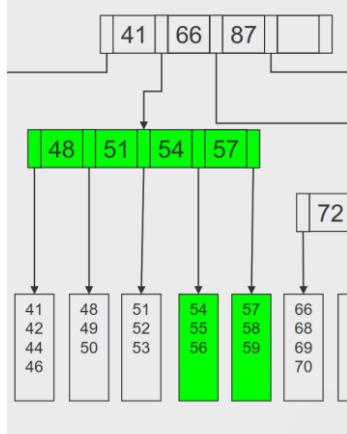
Exemplo:

Inserir o 55



Não existe espaço

Partir o bloco a meio, a folha de 54,56,57,58,59

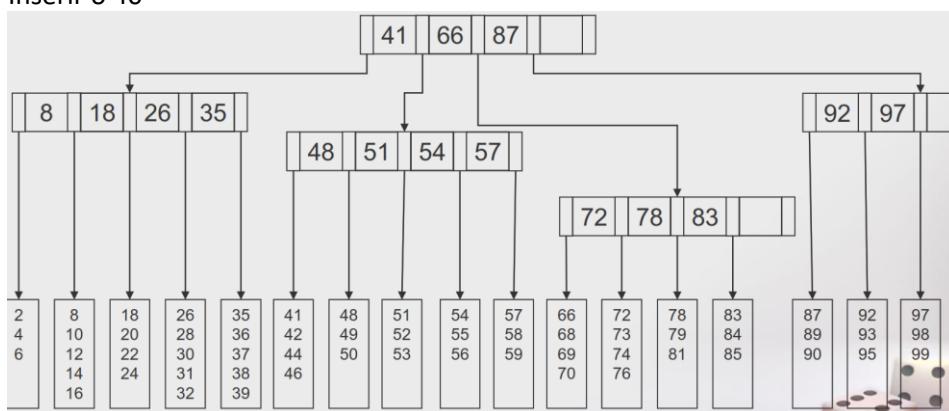


Se uma folha está cheia, divide-se e surge uma nova folha

O número de folhas varia entre L e $L/2$, se a L está cheia então dividiu-se e fica $L/2$

Exemplo:

Inserir o 40



$40 < 41$

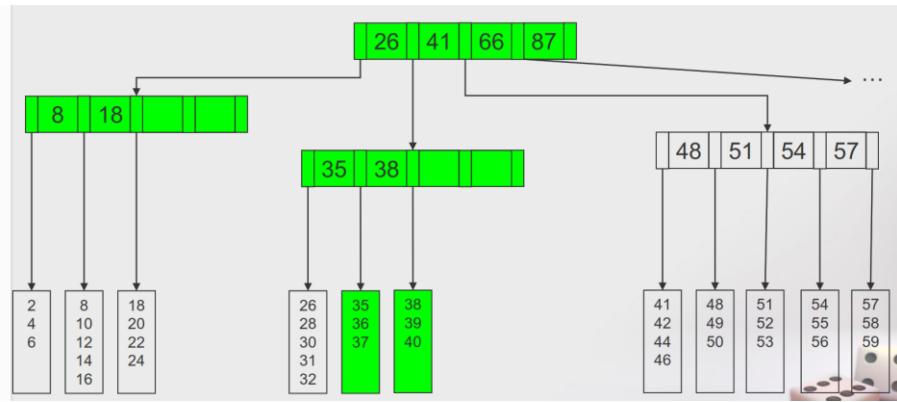
$40 > 35$

E a folha está cheia

Dividir a meio o 35,36,37,38,39

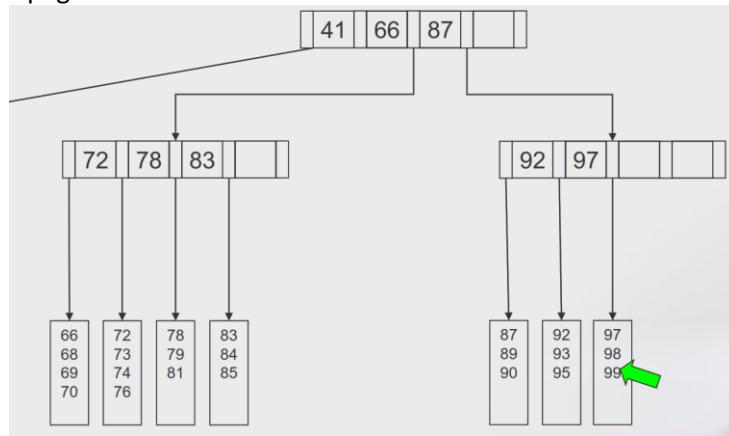
Os descendentes já são cinco: 8, 18, 26, 35

Temos que dividir a meio o nodo intermédio



Exemplo:

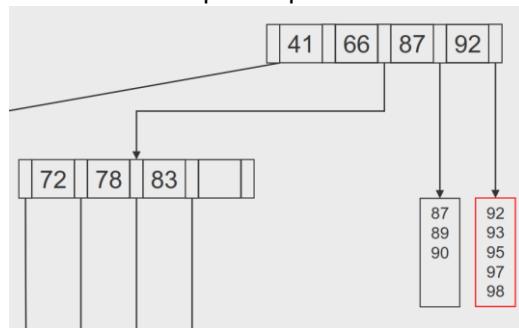
Apagar o 99



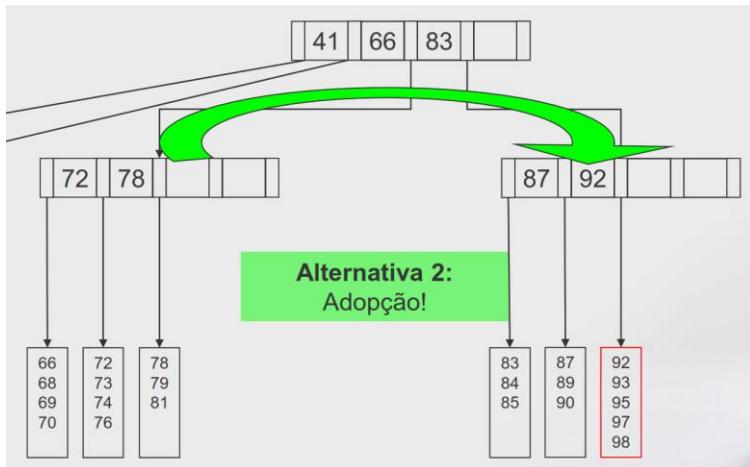
O número mínimo de registros é três

O vizinho pode assumir os valores, mas o nodo já não tem os três descendentes que devia ter

Alternativa 1: repetir o processo de consolidação



Alternativa 2: fazer uma adoção, havendo margem para isso



#Estruturas de Dados Estáticas

As árvores de pesquisa equilibradas são estruturas estáticas:

O tempo de pesquisa de um determinado elemento depende da sua profundidade na árvore.
A profundidade é estática, ou seja, é determinada no momento de inserção e não varia nunca mais.

Não oferecem vantagens para padrões de acesso comuns:

Regra 90-10: 90% dos acessos são a 10% dos dados

Acesso sequencial

#Estruturas de Dados Dinâmicas

As estruturas de dados dinâmicas (ou auto-organizadas) pretendem dar resposta a este problema:

A organização interna dos dados é modificada de acordo com os padrões de pesquisa para maximizar o desempenho.

Vamos analisar duas estruturas de dados dinâmicas:

Listas auto-organizadas – Usadas para implementar conjuntos.

Splay Trees

A posição da informação muda de acordo com os acessos nas estruturas de Dados Dinâmicas

#Lista Auto-Organizada

O acesso aos elementos de uma lista próximos das localizações conhecidas (início, fim e último acesso) é mais eficiente do que o acesso a elementos mais remotos.

Idealmente, os nodos deveriam estar organizados em ordem de probabilidade de acesso

Várias estratégias podem ser adoptadas para atingir este fim:

- MTF – Move to Front – Copia cada elemento acedido para a cabeça da lista.
- Transposição – Troca um elemento acedido com o elemento anterior.
- Contagem – O número de acessos é armazenado em cada nodo, e são efetuadas trocas quando tal é necessário.

Mais estável: MTF – Move to Front, Transposição, Contagem

Mais reactivo: Contagem, Transposição, MTF – Move to Front

Exemplo:

Lista com 1 → 10 → 100 → 20

Estratégia MTF – Move to Front (mais reactiva)

Quando acedemos ao valor ele passa a ser a cabeça da lista

Acedo ao 100

Passamos a ter 100 → 1 → 10 → 20

Exemplo:

Lista com 1 → 10 → 100 → 20

Estratégia Transposição (vamos aproximando do início)

Acedo ao 100

Passamos a ter 1 → 100 → 10 → 20

Exemplo:

Lista com 1 → 10 → 100 → 20

Estratégia Contagem

Tem-se guardado o número de acessos

E quando o número de acessos foi superior ao que está antes, trocam-se de posições

O desempenho, pode ser calculado pelo número médio dos saltos

$S = p(0)*0 + p(1)*1 + p(2)*2 + \dots$

Exemplo:

% de acessos: 10%, 30%, 50%, 10%

Mas se organizarmos, na ordem ideal: 50%, 30%, 10%, 10%

$p(0)=0.5$ $p(1)=0.3$ $p(2)=0.1$ e $p(3)=0.1$ vem que $S = 0.8$

ao invés de Se $p(0)=0.1$ $p(1)=0.3$ $p(2)=0.5$ e $p(3)=0.1$, com $S = 1.6$

#Splay Tree

As Splay Trees são árvores com comportamento dinâmico

São árvores que respeitam a ordem, valores menores para a esquerda e maiores para a direita

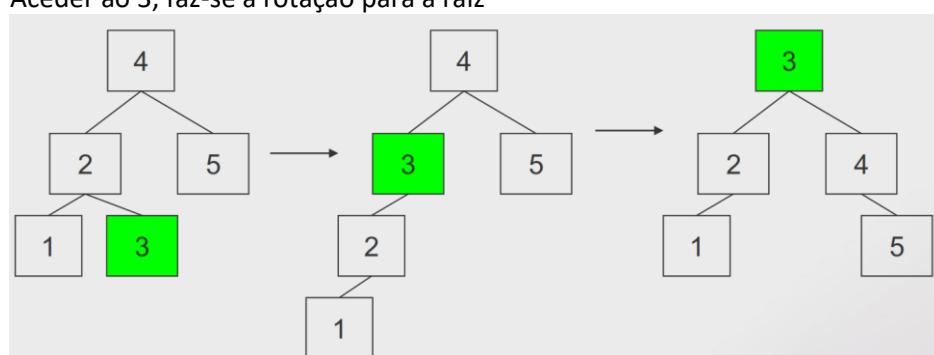
Qual é o desempenho?

Garantia de desempenho $O(N)$ no pior caso.

Mas têm-se garantia de desempenho $O(\log N)$ amortizado.

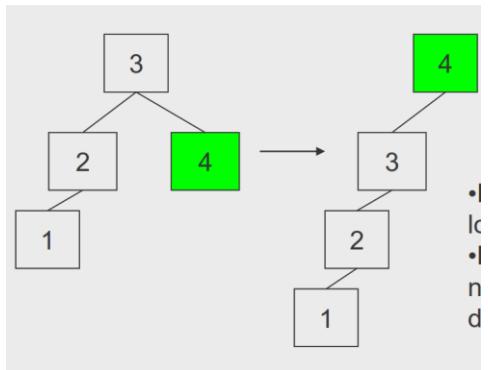
Exemplo:

Aceder ao 3, faz-se a rotação para a raiz



Exemplo:

Inserir o 4



Exemplo:

Inserir o 1,2,3,4

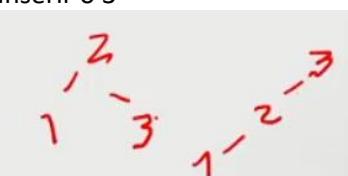
Inserir o 1



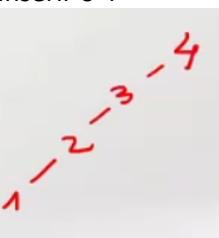
Inserir o 2



Inserir o 3

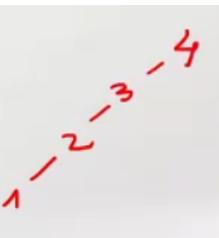


Inserir o 4



Exemplo:

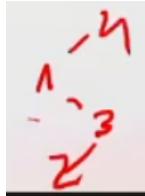
aceder ao 1



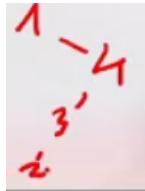
O 1 roda com o 2



O 1 roda com o 3



O 1 roda com o 4



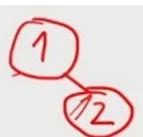
Demorei N valores até chegar ao 1
Complexidade quadrática!! $O(N^2)$, mau.

O processo básico de rotação para raiz pode ser modificado ligeiramente com uma técnica chamada splaying para assegurar garantias de desempenho amortizado!
splay: espalhar ou dispersar

Quando um nodo é acedido em vez de rodar com o nó acima,

Exemplo:

Temos os nodos



Rodo o 2 com a raiz



Exemplo:

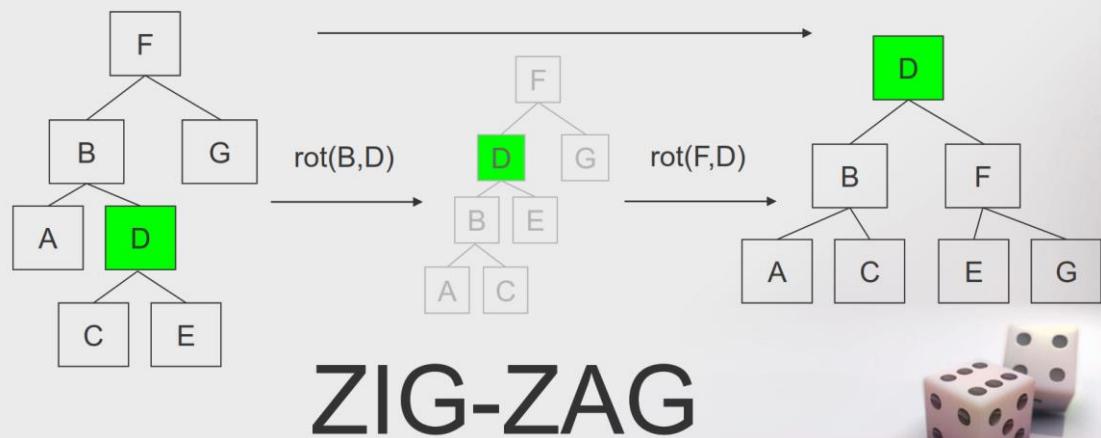
Acesso a um descendente da raiz



Exemplo:

O nodo que está a subir é interior, Nodo interior

Dupla rotação
rot(B,D) e depois rot(F,D)

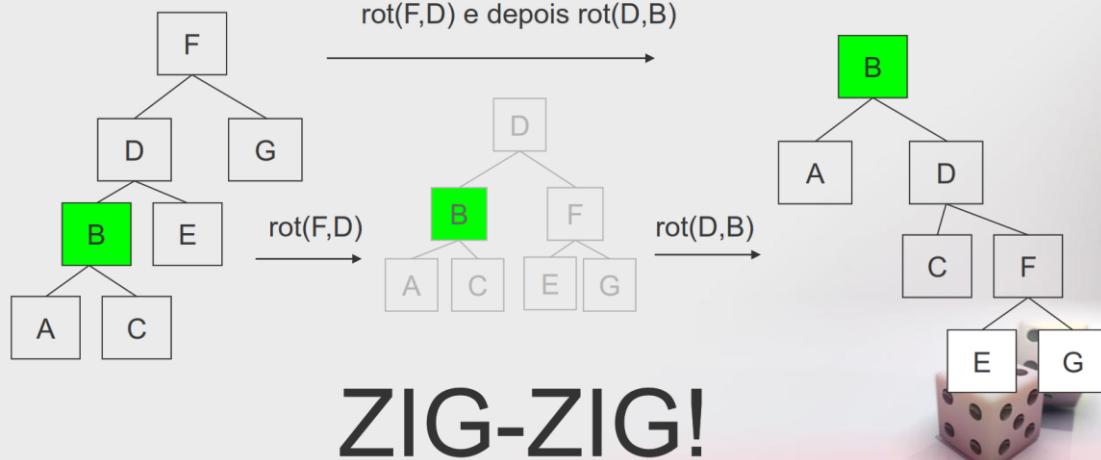


Fez-se uma dupla rotação, Zig-Zag

Exemplo:

Nodo exterior,

Splaying!!!
dupla rotação:
rot(F,D) e depois rot(D,B)

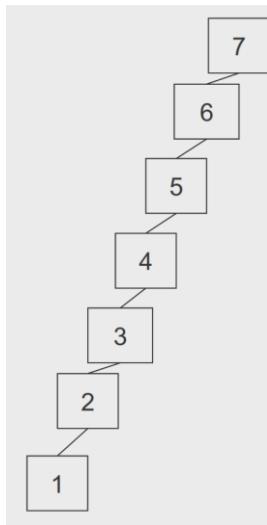


Primeiro foi feita a rotação do F com D

E depois a rotação do D com o B

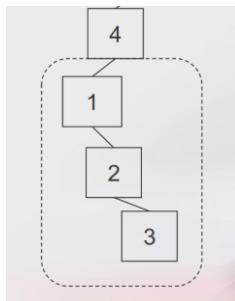
Exemplo:

Aceder ao 1



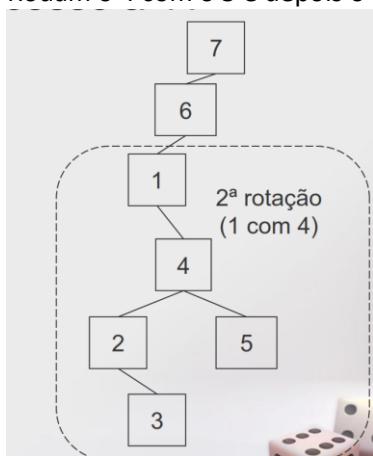
Observar o 2 e o 3

Rodam o 3 com o 2 e depois o 2 com o 1



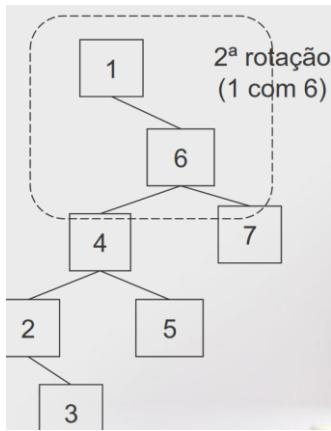
Observar o 4 e o 5

Rodam o 4 com o 5 e depois o 4 com o 1



Observar o 6 e o 6

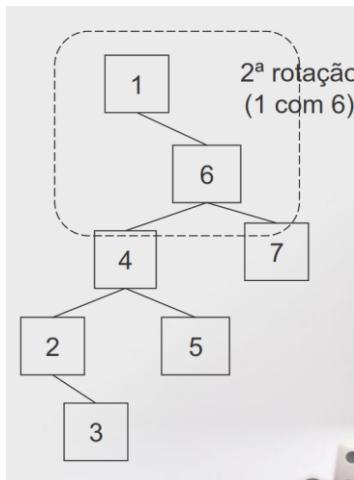
Rodam o 7 com o 6 e depois o 6 com o 1



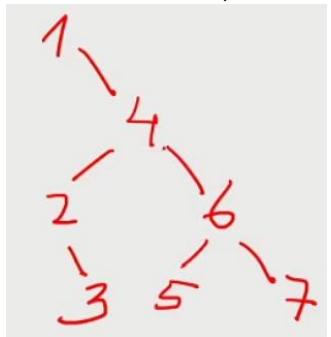
Melhorámos a situação

Exemplo

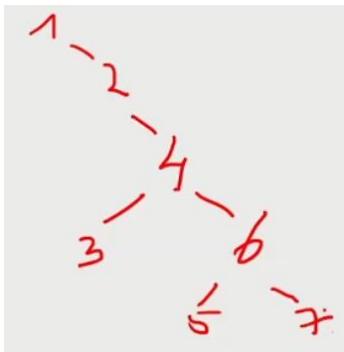
Acesso ao 2



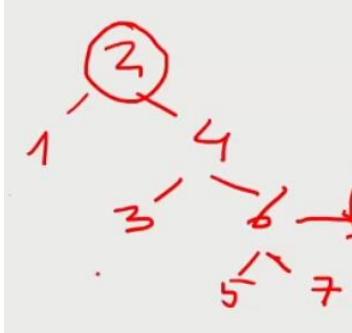
Rodar o 4 com o 6,



e rodar o 4 com o 2

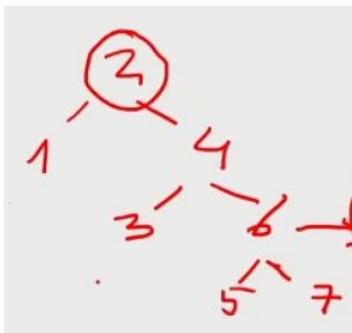


Rotação do nodo com a raiz



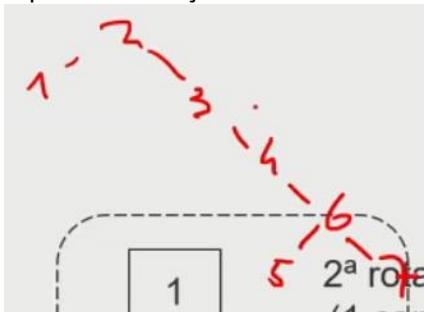
Exemplo

Acesso ao 3



Temos que fazer duas rotações: rodar o 3 com o 4, e rodar o 3 com o 2

A primeira rotação:



A segunda rotação:



Remoção?

1. Acesso a nodo a remover (é colocado na raiz)
2. É removido, sobrando as duas árvores descendentes desconexas (L e R).
3. Procura-se o maior elemento da subárvore esquerda (é colocado na raiz)
4. A subárvore direita é colocada como descendente direito da nova raiz.

Exemplo:

Remover o 3



Ficam duas árvores



Procurar o maior valor da subárvore esquerda

Ir sempre para a direita até não conseguir ir mais

E a subárvore direita é colocada como descendente direito da nova raiz.



[aula09]

ED10-HashTables.pdf

#Hash Tables

As árvores binárias de pesquisa são vocacionadas para o armazenamento de informação ordenada.

Uma hash table é uma estrutura de dados que pode ser usada quando a ordem da informação não é relevante

Inserções, procura e remoções são feitas em tempo médio constante!

A ideia é encontrar uma função que transforma os dados em números de dimensão mais reduzida... Qualquer função deste género é denominada por função de hash (ou função de escrutínio).

O problema da dimensão gigante da tabela, podemos ter outro problema, pois limita o tamanho da tabela, e a gama de valores for superior existem coisas diferentes que ficam mapeados no mesmo local. Existe assim a possibilidade de ocorrerem colisões, ou seja, strings diferentes que são mapeadas para o mesmo índice

Lidar com as colisões pode ser feito com:

Sondagem linear

Sondagem quadrática

Encadeamento

Mas uma propriedade da função de hash é que deve ser simples (e rápida) de calcular

```
//função de hash – versão final
public static int hash( String key, int tableSize )
{
    int hashVal = 0;
    for( int i = 0; i < key.length( ); i++ )
        hashVal = 37 * hashVal + key.charAt( i );
    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;
    return hashVal;
}
```

#Colisões - Sondagem Linear

A estratégia mais simples consiste em pesquisar sequencialmente o array a partir da posição original até que seja encontrada uma posição livre.

Esta pesquisa dá a volta do final para o início do array se necessário.

Exemplo:

Função de hash $H(N,M)=N\%M$ para:

```
Insere(89);
Insere(18);
Insere(49);
Insere(58);
Insere(9);
```

Insere(89); $H(89,10)=9$

Insere(18); $H(18,10)=8$

Insere(49); $H(49,10)=9 \rightarrow$ colisão

Procurar a próxima posição livre

Insere(58); $H(58,10)=8 \rightarrow$ colisão

Procurar a próxima posição livre

Insere(9); $H(9,10)=9 \rightarrow$ colisão

Procurar a próxima posição livre

Resultado final:

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	8
9	89

Desde que a tabela não esteja cheia, será sempre encontrada uma célula livre.
Se a tabela estiver muito cheia, poderá ser necessário percorrer muitos elementos

A taxa de ocupação é um factor que permite calcular o numero médio de saltos para encontrar um espaço vazio

Relembrar o resto da divisão

A sondagem linear:

Funciona bem com tacas de ocupação reduzidas

Não funciona com tacas de ocupação elevadas (efeito de clustering)

#Colisões - Sondagem Quadrática

Serve para evitar o clustering

A ideia da sondagem quadrática a base é efectuar a pesquisa de forma não sequencial de forma a evitar a agregação de items de dados

Caso a posição H esteja ocupada (colisão):

Na pesquisa sequencial procura-se nas posições: H+1, H+2, H+3, ...

Na sondagem quadrática procura-se nas posições: H+1^2, H+2^2, H+3^2 ,...

Exemplo:

Função de hash $H(N,M)=N\%M$ para:

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insere(49); $H(49,10)=9 \rightarrow$ colisão

$(9+1^2) \bmod 10 = 0$

Posição 0 livre fica o 49

Insere(58); $H(58,10)=8 \rightarrow$ colisão

$(8+1^2) \bmod 10 = 9 \rightarrow$ colisão

$(8+2^2) \bmod 10 = 2$

Posição 2 livre fica o 58

Insere(9); H(9,10)=9 -> colisão

$(9+1^2) \bmod 10 = 0$ -> colisão

$(9+2^2) \bmod 10 = 3$

Posição 3 livre fica o 9

0	49
1	
2	58
3	9
4	
5	
6	
7	
8	8
9	89

O critério que tem de ser comprido: a dimensão da tabela tem que ser um número primo e a carga de % de ocupação tem que se menor que 50%

Não pode acontecer uma tabela com Sondagem Quadrática crescer acima de uma ocupação de 50%

#Redimensionamento

A hash table pode ser redimensionada quando atinge uma taxa de ocupação de 0.5

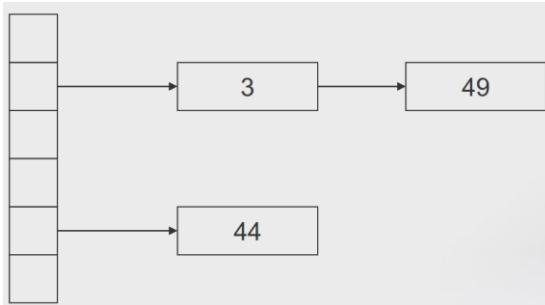
É necessário recalcular de novo os valores de hash e reintroduzir a informação, esta operação chama-se rehashing

#Encadeamento

As técnicas de sondagem não funcionam bem para taxas de ocupação elevadas

Caso o redimensionamento seja indesejável, é possível usar a técnica de encadeamento para lidar com colisões

Neste caso, em cada posição da tabela, há uma lista que contém todos os elementos que partilham esse índice hash



[aula10]

ED11-Heap.pdf

Heap Binária, é uma estrutura de dados e usada para implementar priority queues

Suporta a inserção de elementos e remoção do menor em tempo logarítmico (no pior caso)

Como implementar um priority queue?

Numa lista?
inserção O(1)
pesquisa de mínimo O(N) (temos que percorrer todos)
não resolve o problema

Lista ordenada?
inserção O(N)
pesquisa de mínimo O(1)
não resolve o problema

Árvore Binária Equilibrada? Red Black, AVL..
Inserção O(log N)
pesquisa mínimo O(log N)

A heap binária é uma árvore que:

Pode ser implementada com um array.
Suporta inserção e remoção do mínimo em O(log N) (pior caso)
Suporta inserção em tempo médio O(1) (amortizados) e pesquisa do mínimo em tempo em O(1) (pior caso)

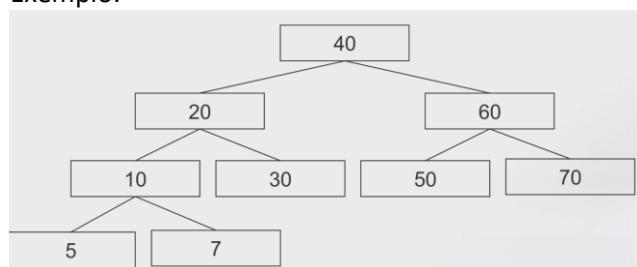
Na heap binária tem como propriedades:

A propriedade de Ordem
A propriedade de Estrutura

#Propriedade de Estrutura

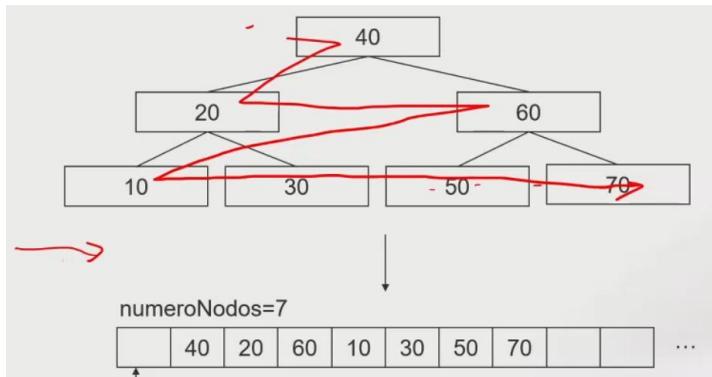
A heap binária, tem que ser uma árvore binária completa, isto é, uma árvore completamente preenchida (exceto, eventualmente, último nível)
É a árvore mais condensada possível, a melhor em altura
O comportamento dos algoritmos que percorrem a árvore entre a raiz e uma folha são log(N) no pior caso.

Exemplo:



Como não existem espaços no meio os nodos podem ser guardados num array em ordem de nível.

Exemplo:
Posso representar uma árvore binária completa num array



A posição 0 é deixada vazia de forma a que o mecanismo de acesso funcione para todas as posições

Exemplo:

`numeroNodos=7`

	40	20	60	10	30	50	70		
--	----	----	----	----	----	----	----	--	--

Qual é a posição do descendente esquerdo do nodo na posição i ?

$$2 * i$$

Exemplo:

`numeroNodos=7`

	40	20	60	10	30	50	70		
--	----	----	----	----	----	----	----	--	--

Qual é a posição do descendente esquerdo do nodo na posição i ?

$$2*i$$

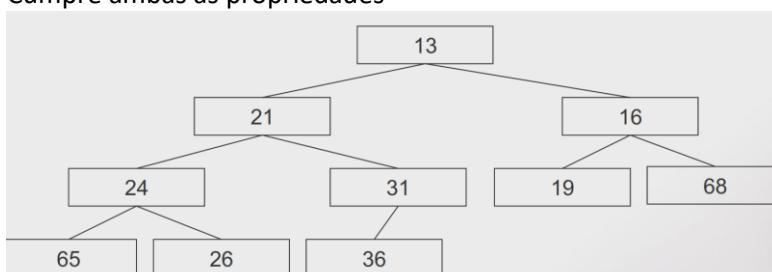
#Propriedade de Ordem-Heap

Queremos procurar eficientemente o menor elemento, qual o melhor sítio para o colocar?

Numa heap binária, para qualquer nodo X com ascendente P, verifica-se que a chave de P é menor ou igual à chave de X.

Exemplo:

Cumpre ambas as propriedades



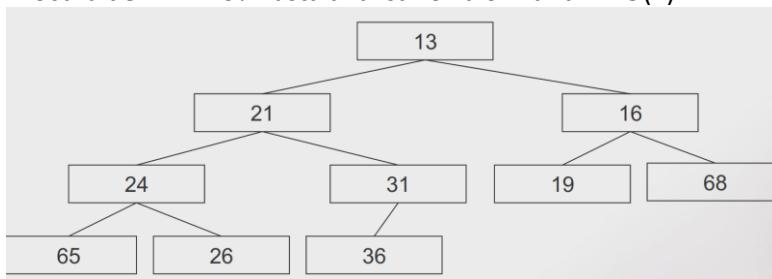
Relembrar que aqui a questão de esquerda/direita não se aplica

Propriedade de Estrutura: O conteúdo deve estar organizado numa ABC.

Propriedade de Ordem (Heap): O ascendente deve sempre ser menor ou igual aos seus descendentes.

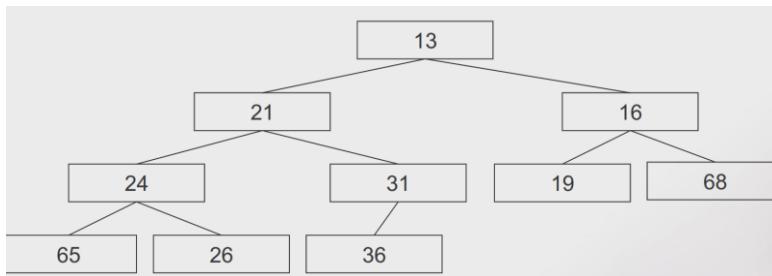
Exemplo:

Procura de Mínimo? Basta analisar o valor na raiz – O(1)

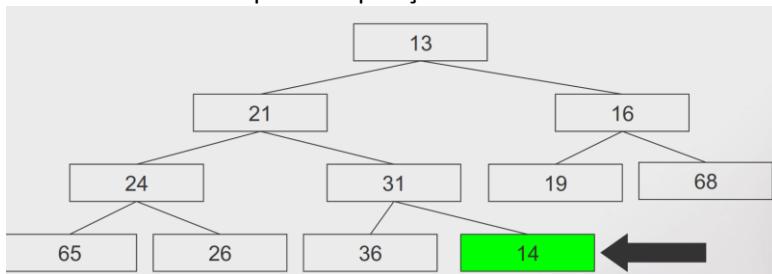


Exemplo:

Inserir 14



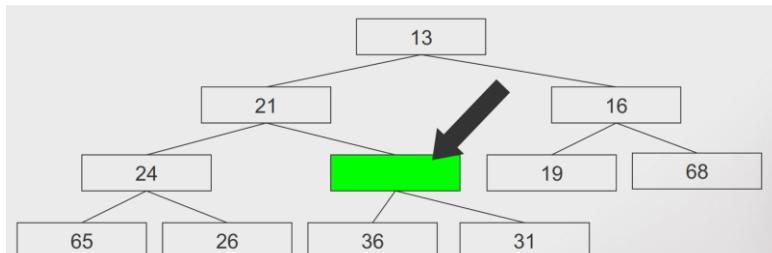
Coloca-se o valor na próxima posição livre.



Mas não acontece ainda a propriedade de ordem, o $31 > 14$

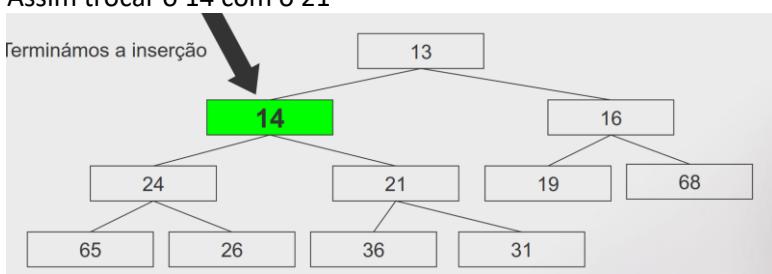
O novo elemento é propagado para cima até que a propriedade de ordem de heap seja verificada.

Assim trocar o 14 com o 31



Mas não acontece ainda a propriedade de ordem, o $21 > 14$

Assim trocar o 14 com o 21

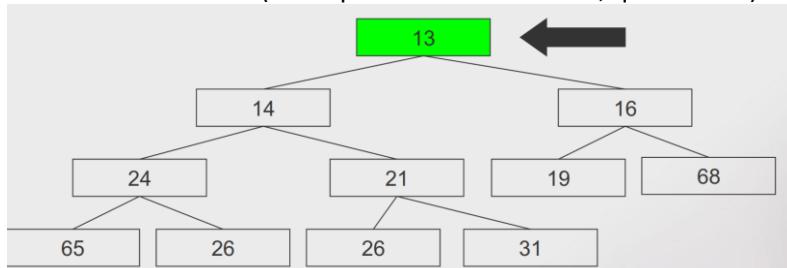


Complexidade $O(\log N)$ em termos de máxima: inserção do novo mínimo faz percurso completo até raiz

Complexidade $O(1)$ em termo de média: É necessário, em média, subir 1.6 níveis.

Exemplo:

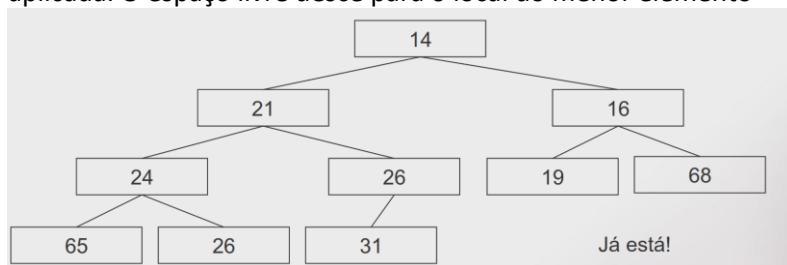
A remover o mínimo (é sempre o menor de todos, que é a raiz)



Não faz sentido remover o 26, porque não sei onde ele está, e posso ter que percorrer muitos dos nodos, como por exemplo o maior de todos (procurar o 100).

O valor que vai ter que ser reposicionado é o 31 que é a última folha

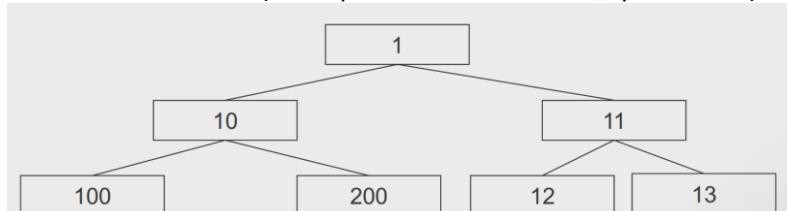
Vai-se removendo e ocupando os espaços, verificando sempre se a propriedade da ordem está aplicada. O espaço livre desce para o local do menor elemento



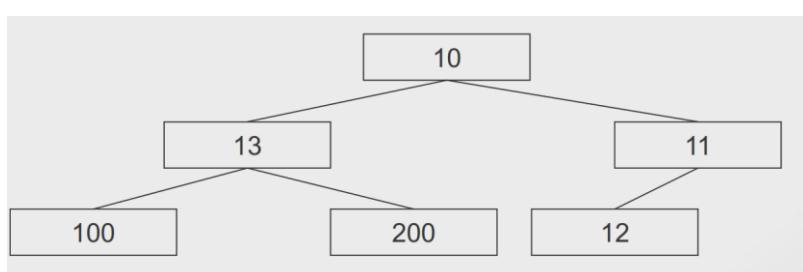
Complexidade $O(\log N)$ máxima e média!

Exemplo:

A remover o mínimo (é sempre o menor de todos, que é a raiz)



O valor que vai ter que ser reposicionado é o 13 que é a última folha



#Inicialização de Heap Binária

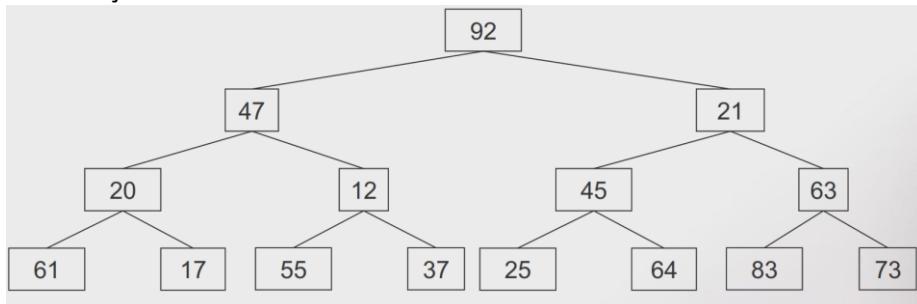
É possível inicializar de forma eficiente uma heap binária com N elementos.

Dada uma ABC qualquer, a ordem de heap pode ser obtida em $O(N)$

Inserir cada elemento individualmente numa heap vazia demora, naturalmente, $O(N \log N)$ no pior caso

Exemplo:

Inicialização

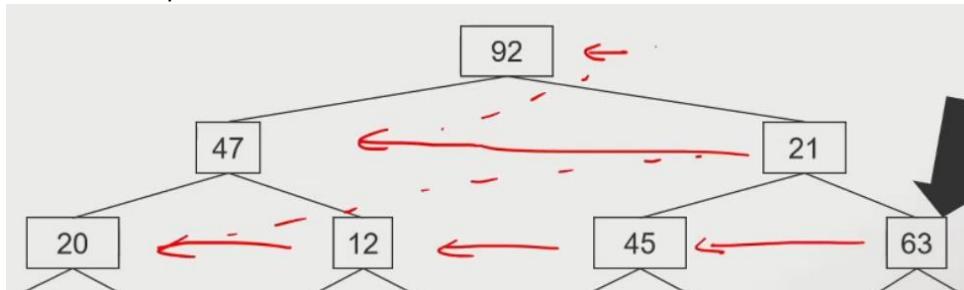


Processar os valores de baixo para cima

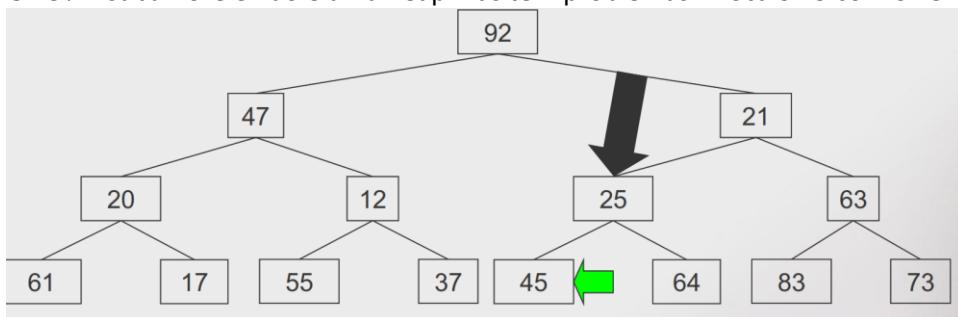
As folhas não vão ser processadas, já estão no último nível, já são uma heap, não é aplicada nenhuma correção

O primeiro Nodo é o 63, $n/2$, $15/2$, posição 7, logo surge o 63

E o sentido a percorrer será:

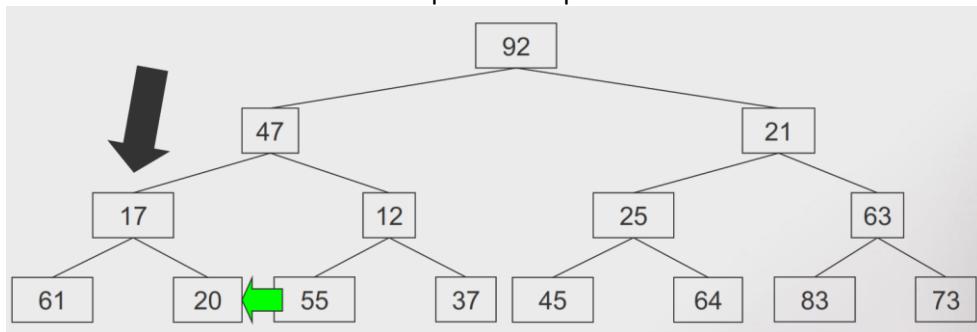


O 63 está no local correto? precisa de descer? Não. Não se faz nada. A subárvore é uma heap
O 45? A subárvore é não é uma heap mas tem problemas. Troca o 25 com o 45



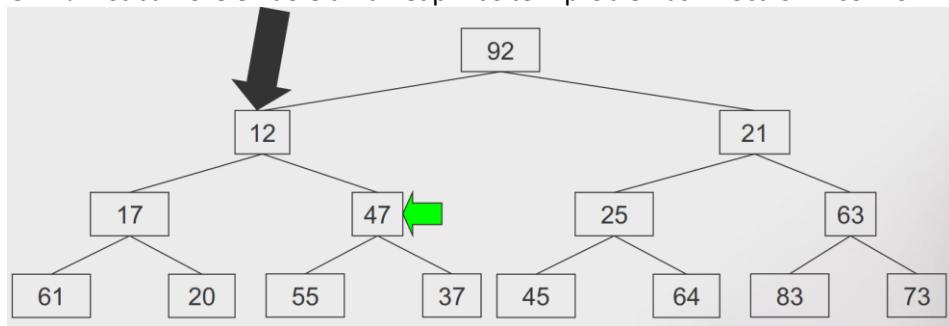
O 12? A subárvore é uma heap

O 20? A subárvore é não é uma heap mas tem problemas. Troca o 20 com o 17

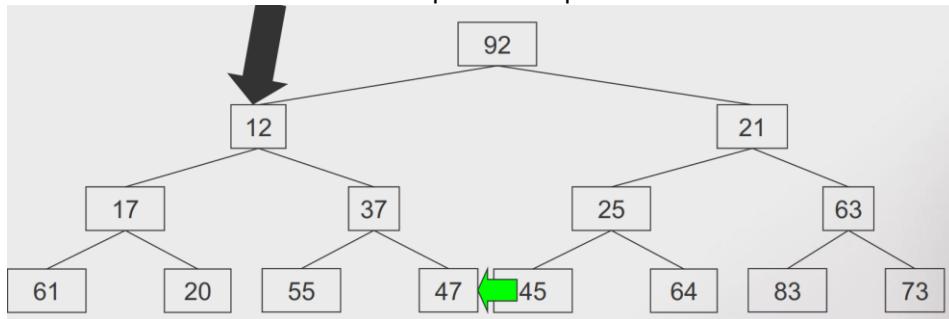


O 21? A subárvore é uma heap

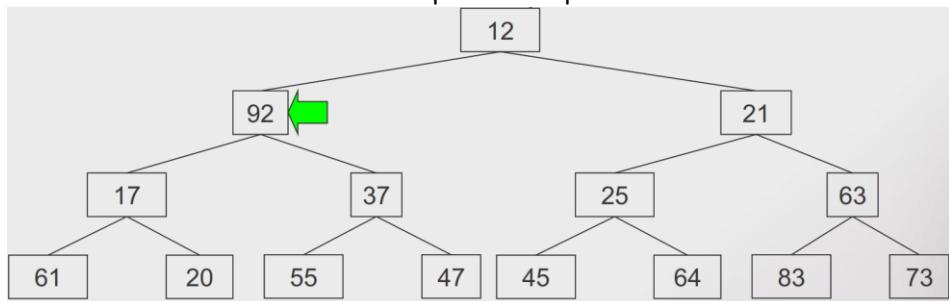
O 47? A subárvore é não é uma heap mas tem problemas. Troca o 47 com o 12



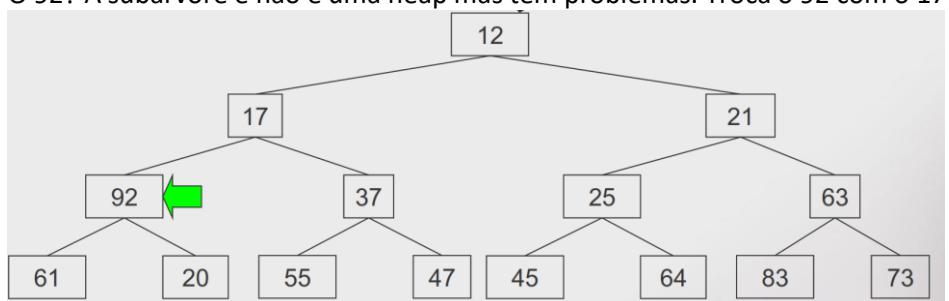
O 47? A subárvore é não é uma heap mas tem problemas. Troca o 47 com o 37



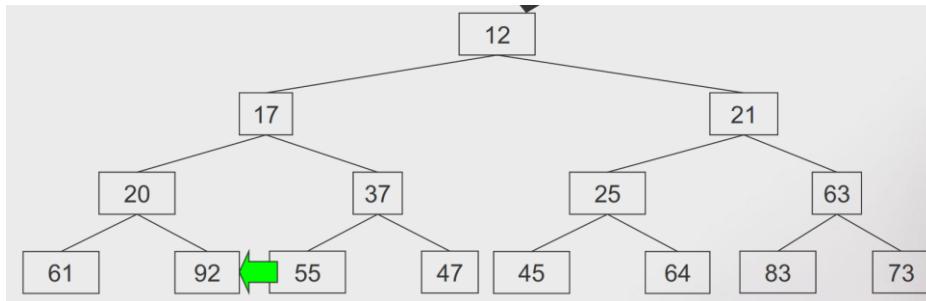
O 92? A subárvore é não é uma heap mas tem problemas. Troca o 92 com o 12



O 92? A subárvore é não é uma heap mas tem problemas. Troca o 92 com o 17



O 92? A subárvore é não é uma heap mas tem problemas. Troca o 92 com o 20



Este processo teve complexidade linear, $O(N)$, mais eficiente que se tivesse a inserir um valor de cada vez

ED12- Skewed and Paired heaps.pdf

#Skew Heap

A Skew Heap ou Heap Desequilibrada, é similar a Heap Binária, mas sem condição de equilíbrio:

Verifica propriedade de ordem de heap.

Não é uma ABC.

Vantagens para uma heap binária:

Suportam de forma eficiente a junção de duas heaps

A partir da operação de junção é possível fazer todas as outras operações fundamentais que podem ser expressas a partir da função de junção:

Inserção (X) – Junção da árvore original com uma árvore contendo apenas o nodo X.

removeMinimo() – Remove raiz e faz junção das subárvore esquerda e direita.

diminuiValor(p, novoValor) – Separa a subárvore com raiz no nodo p, diminui o valor do nodo

Junção simplista: Recursivamente, o resultado da junção de duas árvores com raizes r_1 e r_2 , com $r_1 \leq r_2$, é dado por:

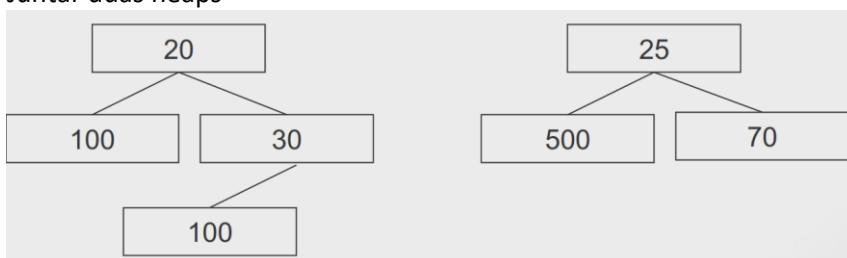
A outra subárvore, se r_1 ou r_2 estiverem vazias.

Se $r_1 \leq r_2$, a árvore com raiz em r_1 após junção da árvore r_2 com o descendente direito de r_1 .

Se $r_2 < r_1$, a árvore com raiz em r_2 após junção da árvore r_1 com o descendente direito de r_2 .

Exemplo:

Juntar duas heaps

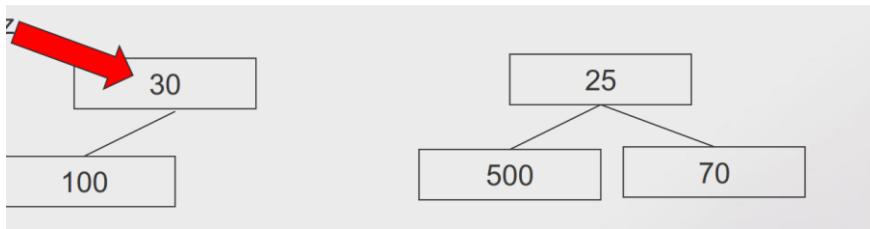


Qual é a menor raiz?

Nodo 20

Identifico a subárvore direita de menor valor a juntar com a outra Heap

Repete-se o processo

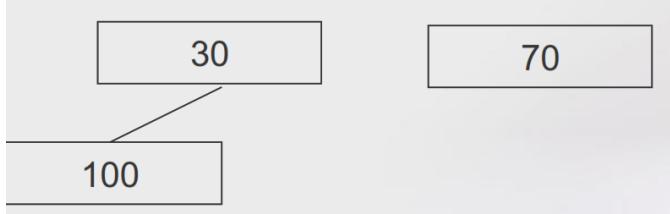


Qual é a menor raiz?

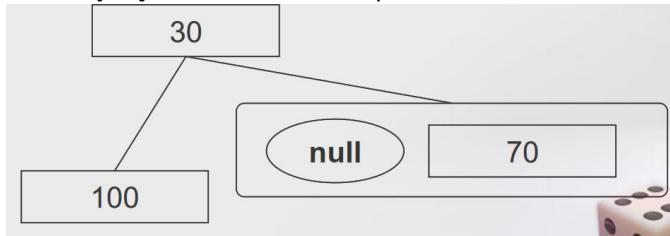
Nodo 25

25 vai ser a raiz da sub heap

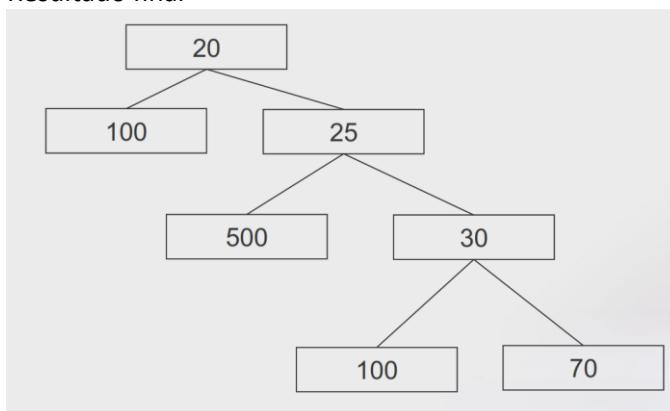
Identifico a subárvore direita de menor valor a juntar com a outra Heap, que neste caso é o 70



Fazer a junção de 70 com null que dá 70



Resultado final



Trata-se de um algoritmo recursivo simples, relativamente eficiente

Estamos sempre a juntar o lado direito, nunca muda o lado esquerdo. Todos os Nodos vão preservar a sua arvore esquerda original

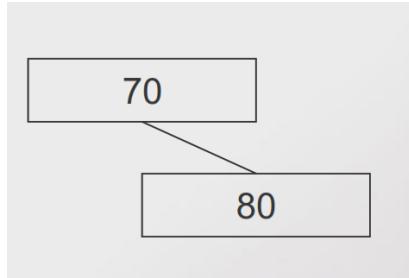
Exemplo:

Inserir os valores 70, 80, 60

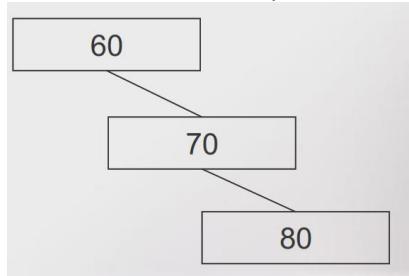
Juntar o null com 70



Juntar o 80 com o 70



Juntar o 60 com a heap



Dado que o lado esquerdo dos nodos não é alterado, nenhum nodo da heap resultante terá descendentes esquerdos. Ou seja, esta será uma lista linear. Uma lista ligada.

Assim para resolver, em cada etapa da junção, trocamos a ordem dos descendentes direitos e esquerdos. Esta operação de junção é aquilo que define a utilização de uma SKEW Heap

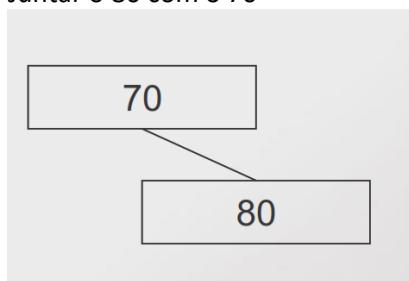
Exemplo:

Inserir os valores 70, 80, 90, 75

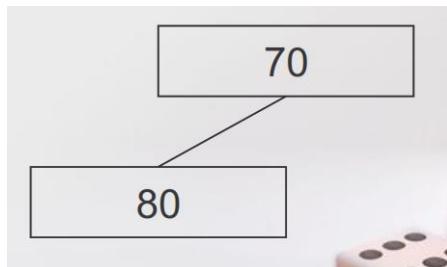
Juntar o null com 70



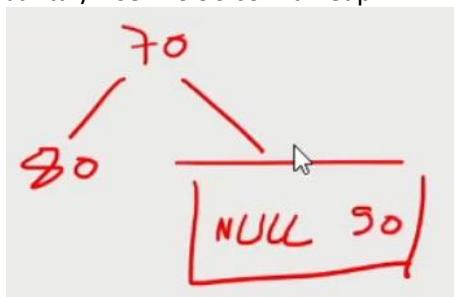
Juntar o 80 com o 70



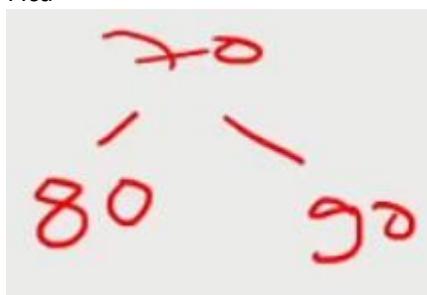
Troca



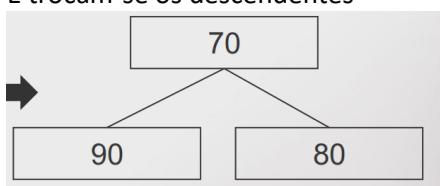
Juntar/Inserir o 90 com a heap



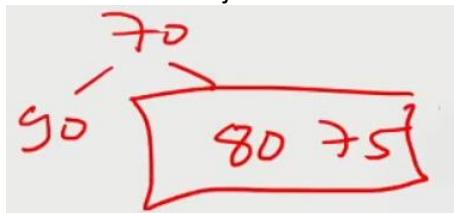
Fica



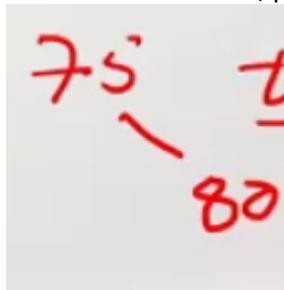
E trocam-se os descendentes

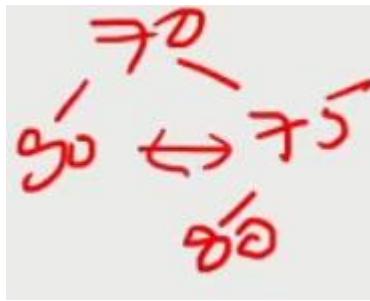


Vou inserir o 75 e juntar com o 80

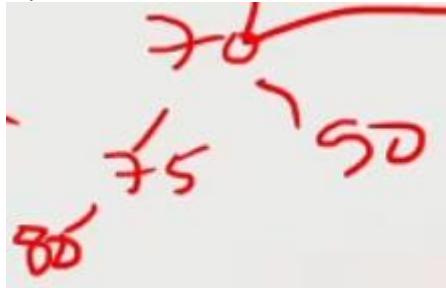


Trocar o 75 com o 80, porque 75 é menor





E por último troco o 75 com o 90, troca da direita com a esquerda



Não se evita completamente a possibilidade de serem criadas árvores degeneradas. Mas o comportamento resultante é amortizado pelo grande número de operações eficientes que precedem a criação dessa árvore.

O custo amortizado para as operações de inserção, remoção do mínimo e junção é: $4 \log N$ no pior dos casos.

A implementação recursiva da junção numa Skew Heap é desaconselhada.

#Heap Emparelhada

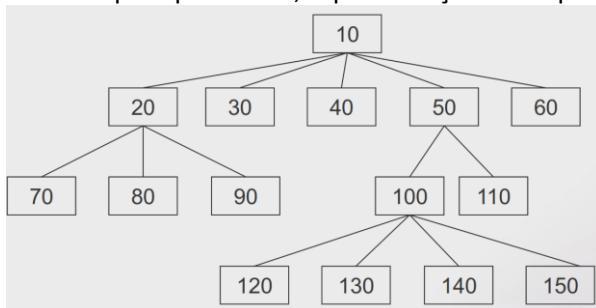
Heap M-ária, sem restrições de equilíbrio.

Todas as operações são suportadas em tempo constante, no pior caso, excepto a remoção.

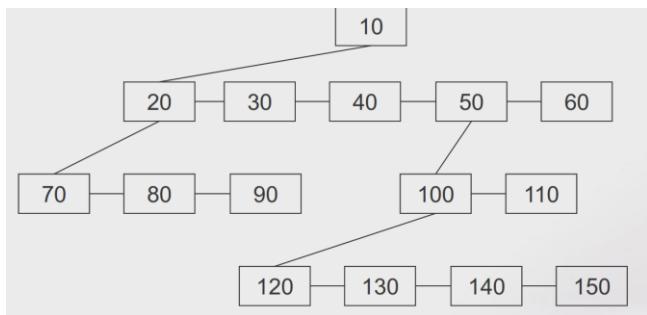
A remoção demora tempo linear no pior caso, mas sequências de operações têm desempenho logarítmico amortizado.

Exemplo:

Uma heap emparelhada, representação conceptual



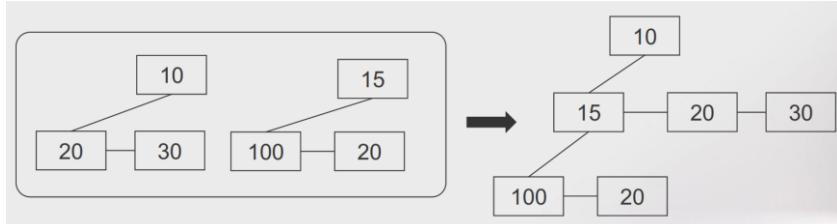
A heap emparelhada, representação binária



Exemplo:

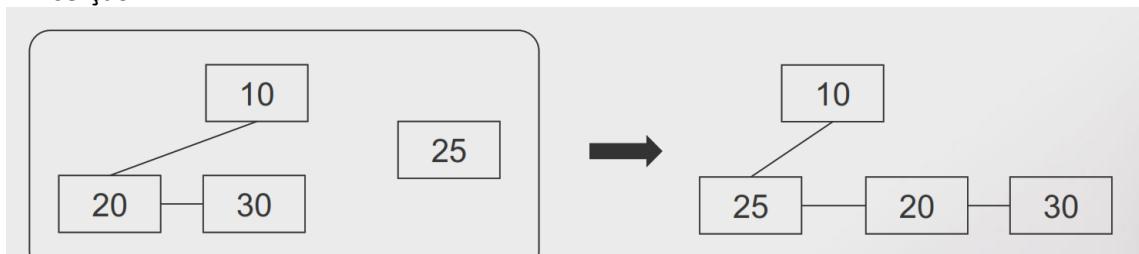
A junção, é feita em tempo constante

A heap a juntar passa a ser o primeiro descendente do lado esquerdo



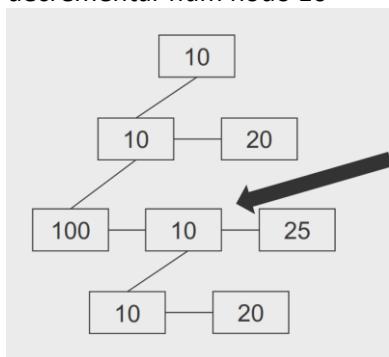
Exemplo:

A inserção

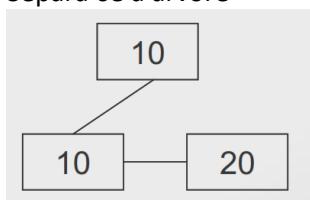


Exemplo:

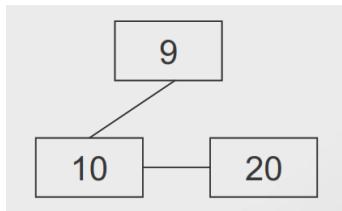
decrementar num nodo 10



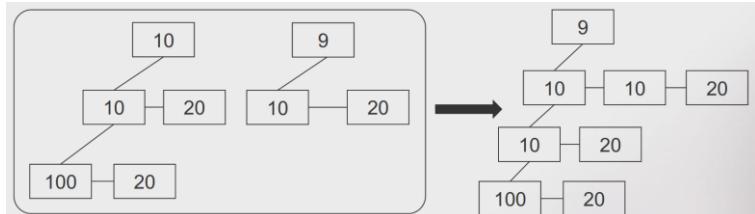
Separa-se a árvore



Decrementa-se



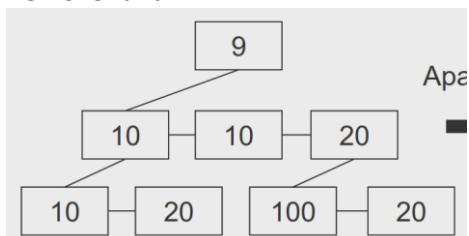
E junta-se as árvores



Exemplo:

Remover o mínimo

Remover a raiz



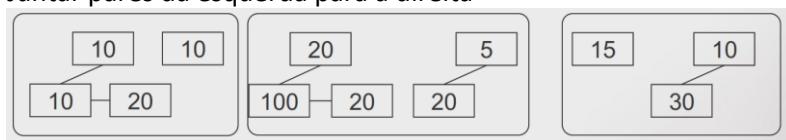
Não existe limite para o número de descendentes

Fica-se com N árvores para se juntar

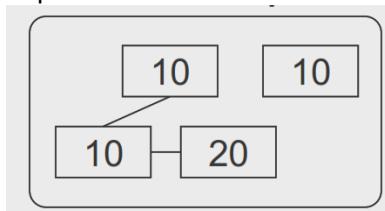


A remoção demora $O(N)$ no pior caso e a ordem de junção das subárvores é escolhida de forma a minimizar a recorrência destas situações.

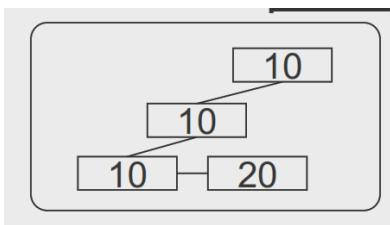
Juntar pares da esquerda para a direita



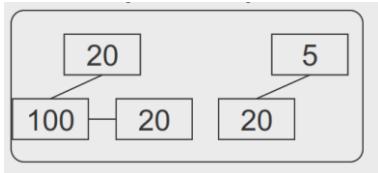
E que fica:



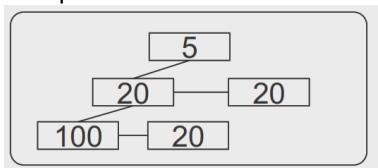
Em:



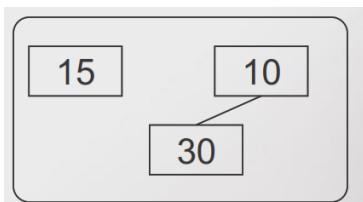
E fica:



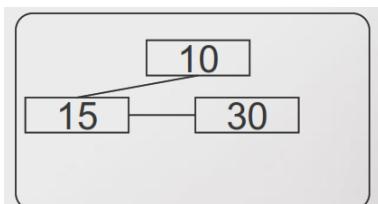
Em que o menor da árvore é o 5, e é essa que vai ser a raiz



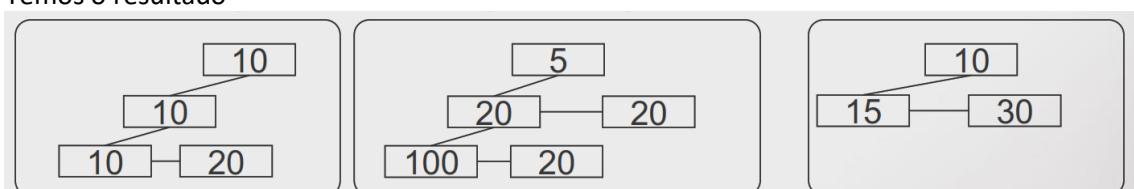
E fica:



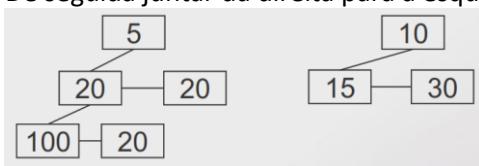
Em:



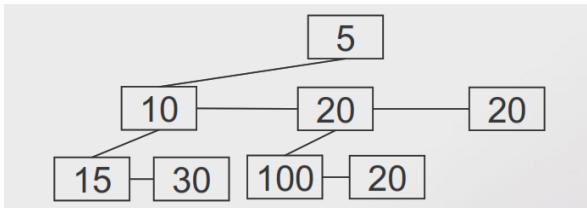
Temos o resultado



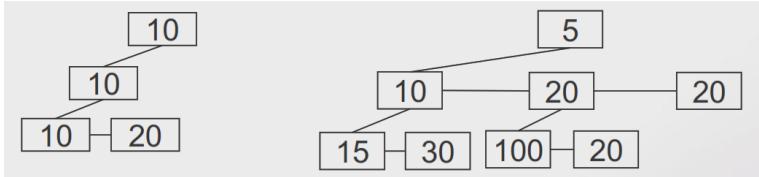
De seguida juntar da direita para a esquerda



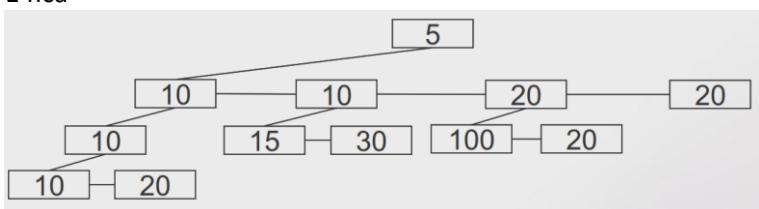
Fica em:



E novamente juntar da direita para a esquerda



E fica



A junção em dois passos assegura comportamento logarítmico amortizado. O número de descendentes da raiz é controlado.

[aula 11]

ED013- Multidimensional and Spatial Information.pdf

#Informação Multidimensional

O armazenamento de informação multidimensional e espacial.

Informação multidimensional pode ser definida como sendo um conjunto de pontos num espaço de dimensão elevada, com uma dimensão por cada atributo

A informação pode ter um carácter continuo, relaciona com informação espacial, e as coisas podem tornar-se difíceis de ser mapeados

Se a informação não possui características de localidade espacial pode normalmente ser representada “confortavelmente” por um único ponto

Se tivermos estruturas que nos permitam armazenar pontos multidimensionais, conseguimos representar tanto informação de natureza discreta como informação espacial (contínua)

A transformação descrita funciona bem se o objetivo é apenas consultar a informação

Mas não facilita a realização de pesquisas sobre esses objetos espaciais

Assim têm que existir estruturas de dados que armazenam a ocupação de espaço e não pontos de dados discretos multidimensionais

Informação Multidimensional vs. Informação Espacial

Existem assim estruturas que permitem armazenar pontos multidimensionais e que são por isso adequadas para informação de natureza discreta (sem localidade espacial) e

estruturas que permitem armazenar informação espacial de natureza contínua, que permitem não só aceder à informação mas também realizar pesquisas complexas de natureza espacial (ver se um objecto intersecta ou contém outro objeto, etc) de forma eficiente e eficaz

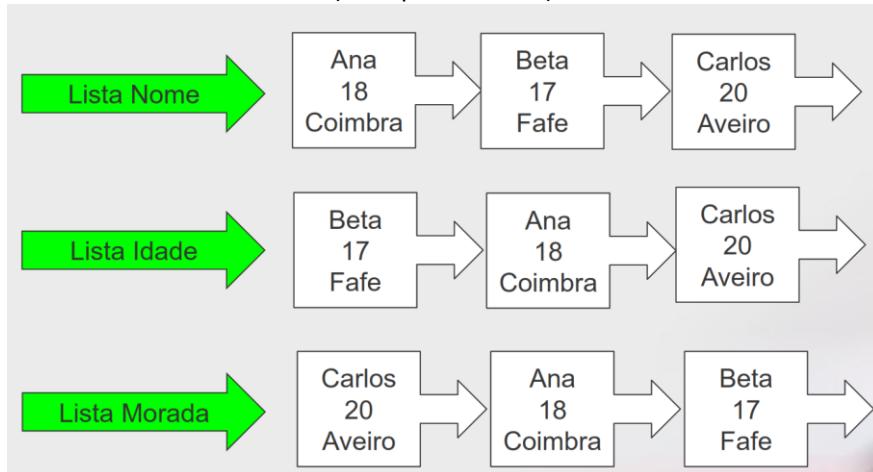
#Estruturas para Pontos Multidimensionais

As listas invertidas: indexar cada uma das dimensões através de uma lista

#Listas Invertidas

Exemplo:

pretende-se armazenar informação sobre pessoas definidas por (nome, idade, morada). Neste caso seriam usadas 3 listas (uma por atributo)



Cada lista está ordenada por diferentes dimensões

Só é eficiente a informação sobre o nome, mas se for mais complexa com diferentes atributos, surge o problema que só é eficiente quando separadas

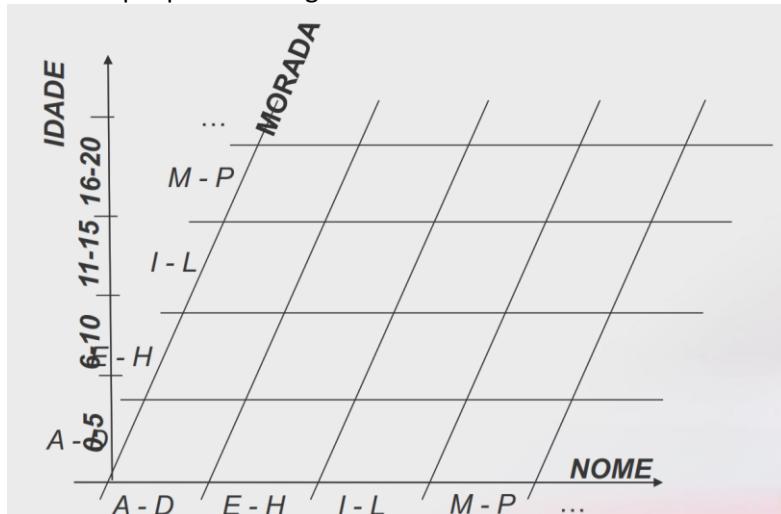
Surge o método de grelha fixa:

Subdividir o espaço dos pontos por uma grelha, sendo usada uma estrutura secundária (p.ex. uma lista) por cada espaço da grelha, para armazenar os pontos nele contidos.

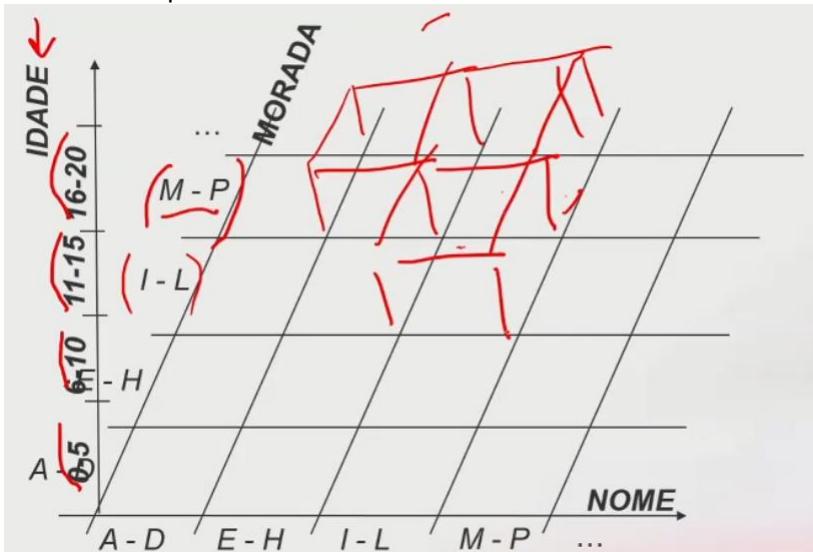
Usar um diretório (a grelha é gerida pelo diretório) que identifica qual o espaço ao qual pertence um ponto e que redireciona para a estrutura secundária correspondente.

Exemplo:

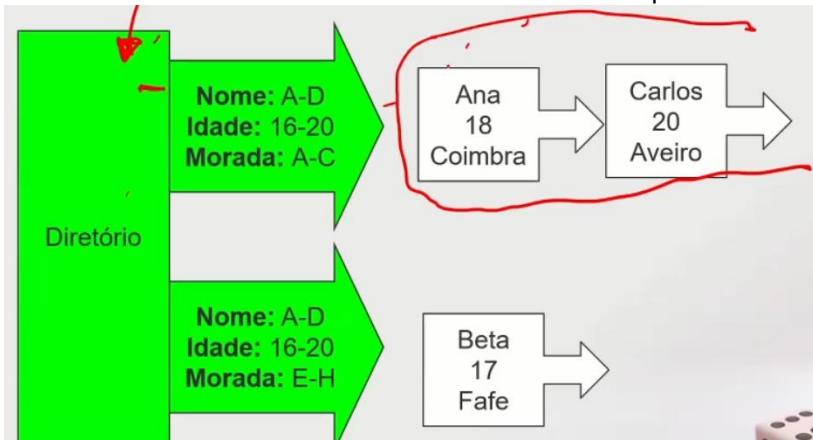
Exemplo: Pretende-se armazenar informação sobre pessoas definidas por (nome, idade, morada). Um exemplo possível de grelha seria:



O diretório é a parte da estrutura de dados que sabe em qual dos "cubos" é que um determinado ponto vai estar



E dentro dos cubos existe uma estrutura secundária que tem todo a informação que lá está



#Grelha Uniforme vs. Grelha não Uniforme

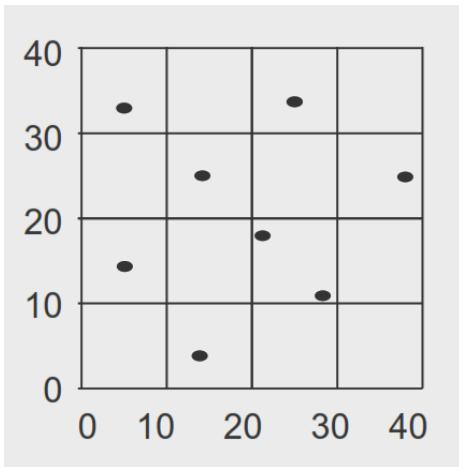
A grelha pode ser definida de acordo com duas abordagens diferentes:

Ou dividir o espaço equitativamente em cada uma das dimensões. Resulta numa grelha uniforme

- Pretende organizar o espaço.
 - O diretório é facilmente implementado com um array multidimensional.
- Ou tentar dividir os dados equitativamente, colocando as divisões em posições arbitrárias e não regulares.
- Pretende organizar os dados.

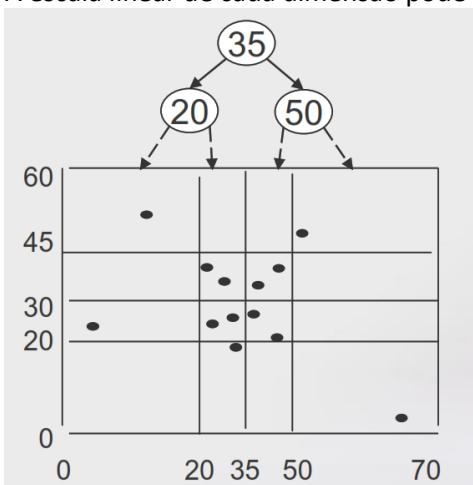
Grelha Uniforme

Correspondência direta entre posição e array



Grelha Não- Uniforme

A escala linear de cada dimensão pode ser representada por uma árvore (binária)



Se a informação for dinâmica (p.ex. os pontos podem mover-se) ou não estiverem dispersos de forma uniforme pelo espaço, então podem algumas células com muita informação e muitas outras vazias.

#Grelhas dinâmicas

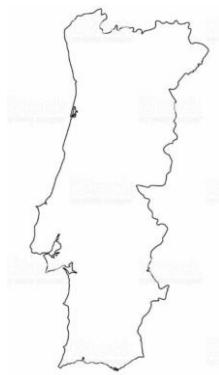
Uma forma de implementar grelhas dinâmicas passa pela utilização de uma das seguintes generalizações multidimensionais de árvores binárias (quadtree – árvore de quadrantes):

- Quadtree de Pontos – Nesta árvore, as fronteiras são determinadas pelos dados inseridos.
- Quadtree PR – Nesta árvore, as fronteiras são escolhidas de forma predeterminada.

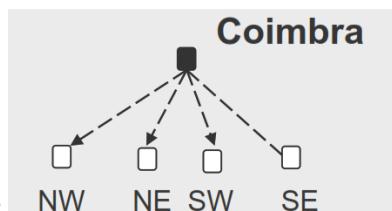
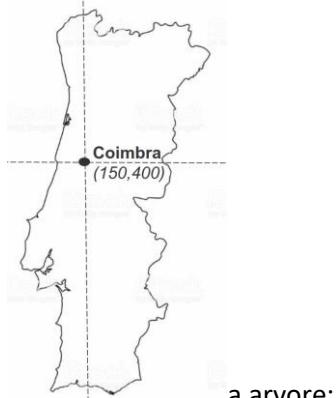
#Quadtree de Pontos

Exemplo:

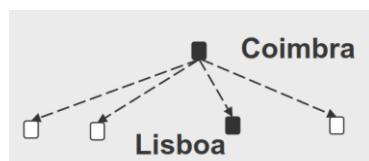
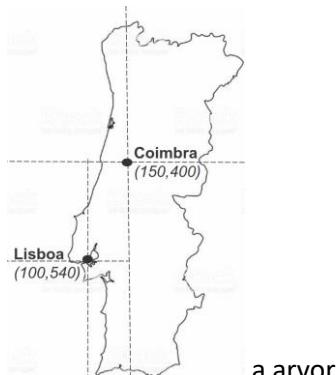
Nesta árvore, cada ponto inserido vai separar o espaço em vários quadrantes



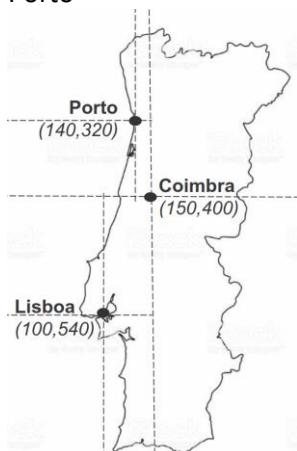
Quero guardar o ponto Coimbra (150, 400)



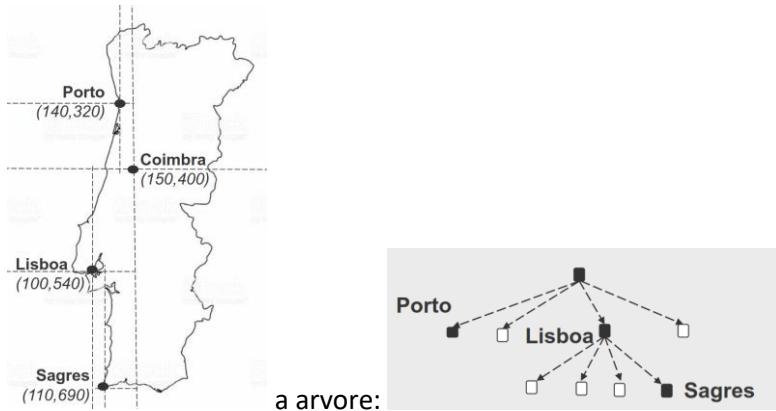
Inserir Lisboa



Porto



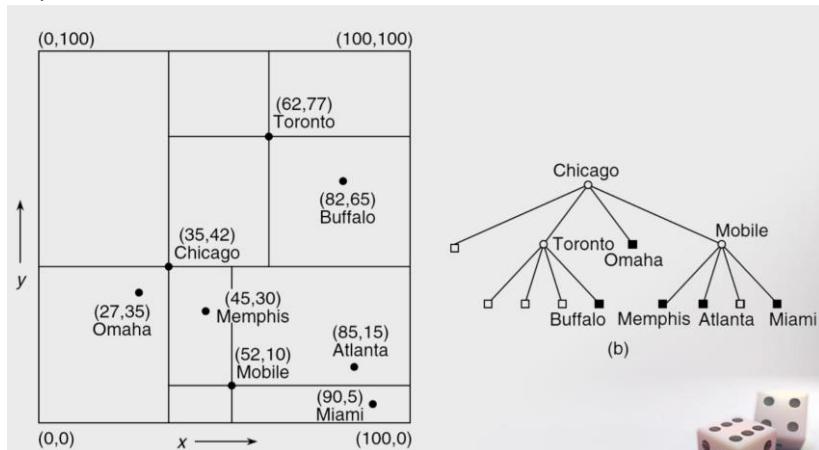
e Sagres



Se o primeiro ponto fosse Sagres a árvore também seria diferente
Nesta abordagem os pontos vão estando em todas as posições da árvore

Exemplo:

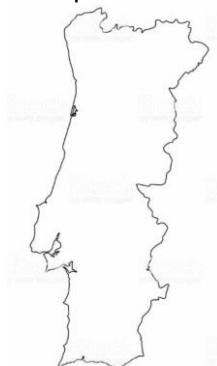
#Quadtree de Pontos



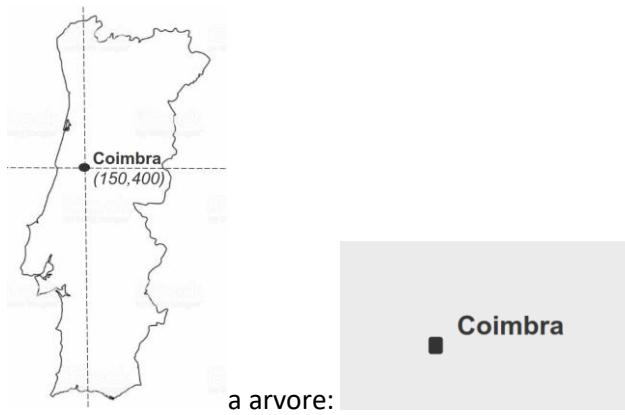
#Quadtree PR

Vamos encontrar a partição do espaço mais adequado para o conjunto de dados
A forma da árvore não depende da ordem de inserção
Os pontos nesta abordagem estão nas folhas da árvore

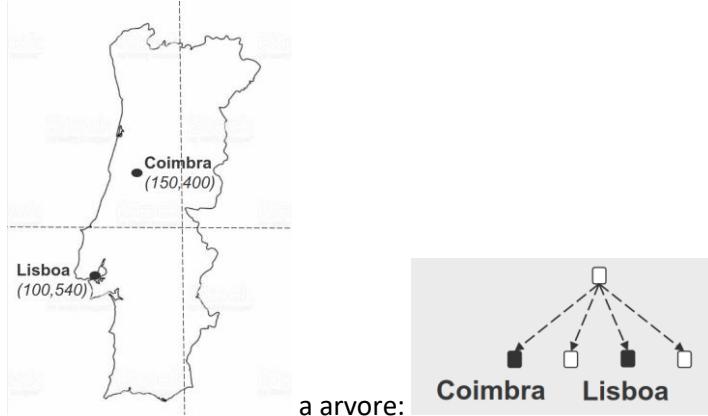
Exemplo:



Quero guardar o ponto Coimbra (150, 400)



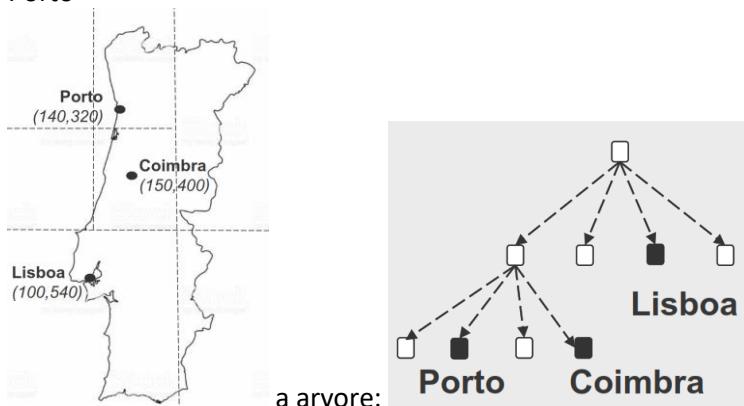
Não existe nenhuma subdivisão ela só existe quando forem inseridos mais pontos
Inserir Lisboa



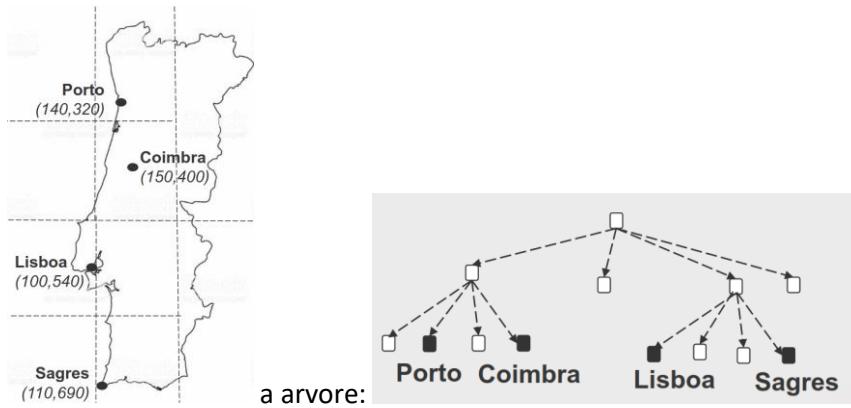
Pois há apenas uma célula que contém dois pontos. Vai ser subdividido para corrigir essa situação. E divide-se e não se usam os pontos dos dados que estão a ser inseridos (existe uma subdivisão de um ponto).

Não se usa a posição dos dados inseridos, e esta subdivisão permite então ter um ponto em cada quadrante e os dados inseridos estão em folhas

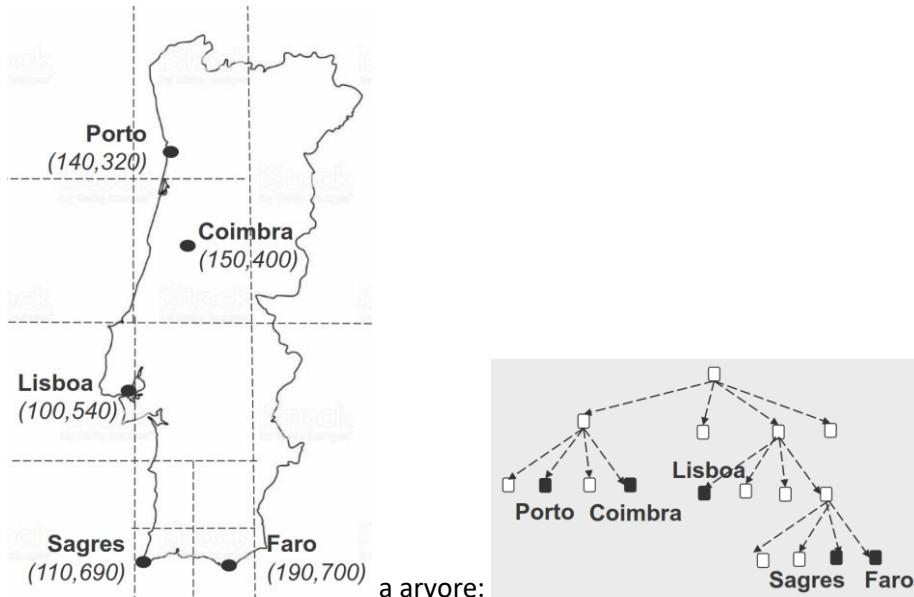
Porto



e Sagres



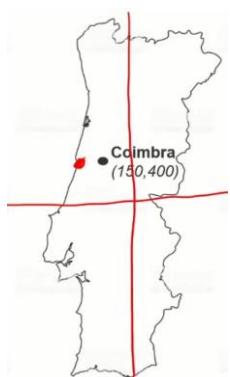
E inserir Faro



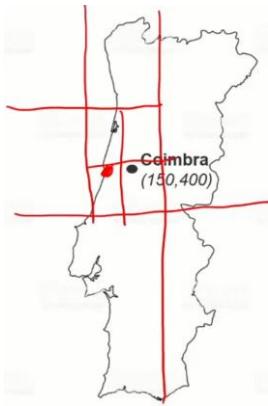
A subdivisão vai ser sendo feita até ter o número desejado em cada uma das células

Exemplo:

Coimbra e Figueira da Foz



E o processo vai dividir até..



#Árvore k-d

Uma árvore k-d indexa a informação de acordo com uma dimensão (atributo) distinta em cada nível

"em níveis diferentes vou ter informação de níveis diferentes"

Exemplo:

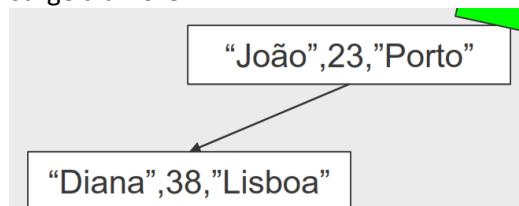
Avore k-d com pessoas (nome, idade, morada)

1. Insere ("João", 23, "Porto")
surge a arvore com nível 1:

"João", 23, "Porto"

2. Insere ("Diana", 38, "Lisboa")

Nível 1: Comparado o nome diana < João
surge a arvore:

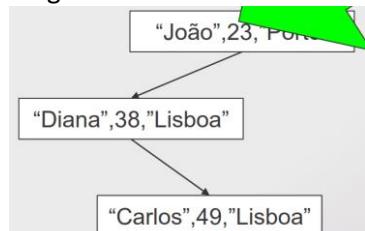


3. Insere ("Carlos", 49, "Lisboa")

Nível 1: Comparado o nome carlos < João

Nível 2: Comparado a idade (é a segunda dimensão) 49 > 38

surge a arvore:

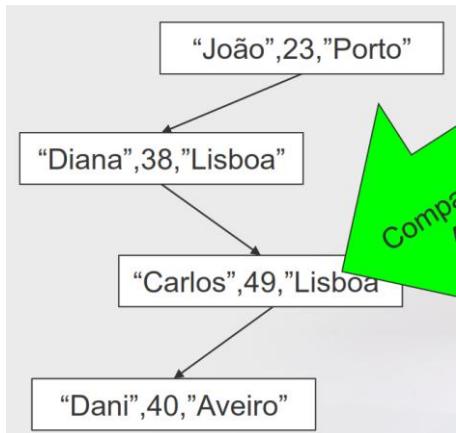


4. Insere ("Dani", 40, "Aveiro")

Nível 1: Comparado o nome dani < João

Nível 2: Comparado a idade (é a segunda dimensão) 40 < 38

Nível 3: Comparado a cidade (é a terceira dimensão) aveiro < Lisboa



5. Insere ("Rui",12,"Coimbra")

Nível 1: Comparado o nome rui > João

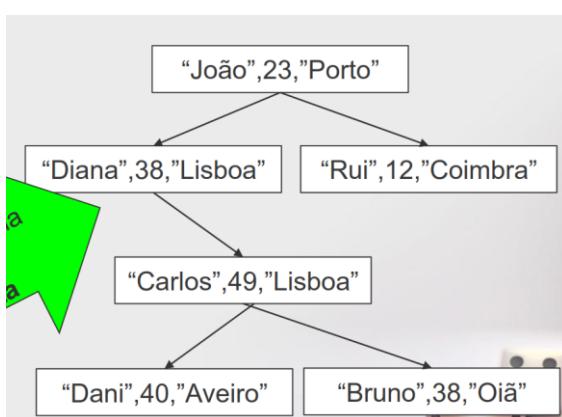


6. Insere ("Bruno",38,"Oiã")

Nível 1: Comparado o nome bruno < João

Nível 2: Comparado a idade (é a segunda dimensão) 38 <= 38 (vai para a direita)

Nível 3: Comparado a cidade (é a terceira dimensão) oiã > Lisboa



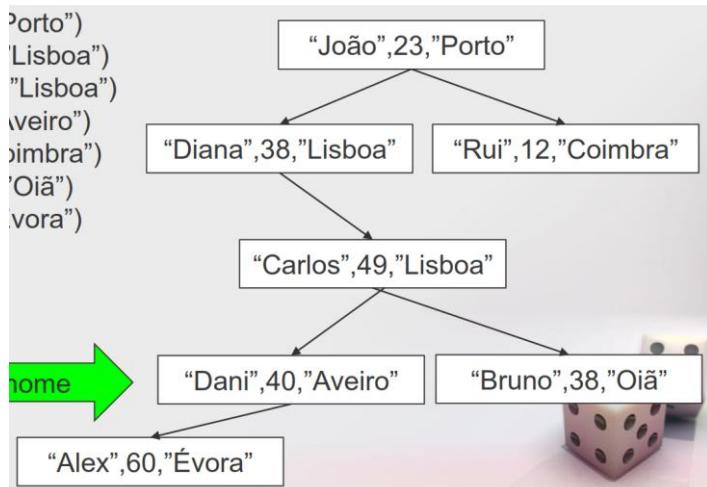
6. Insere ("Alex",60,"Évora")

Nível 1: Comparado o nome alex < João

Nível 2: Comparado a idade (é a segunda dimensão) 60 > 38

Nível 3: Comparado a cidade (é a terceira dimensão) evora < Lisboa

Nível 4: Comparado o nome alex < Carlos



As árvores k-d podem ser usadas independentemente, ou então serem usadas para implementar a decomposição do espaço das quadtrees

Árvore PR k-d

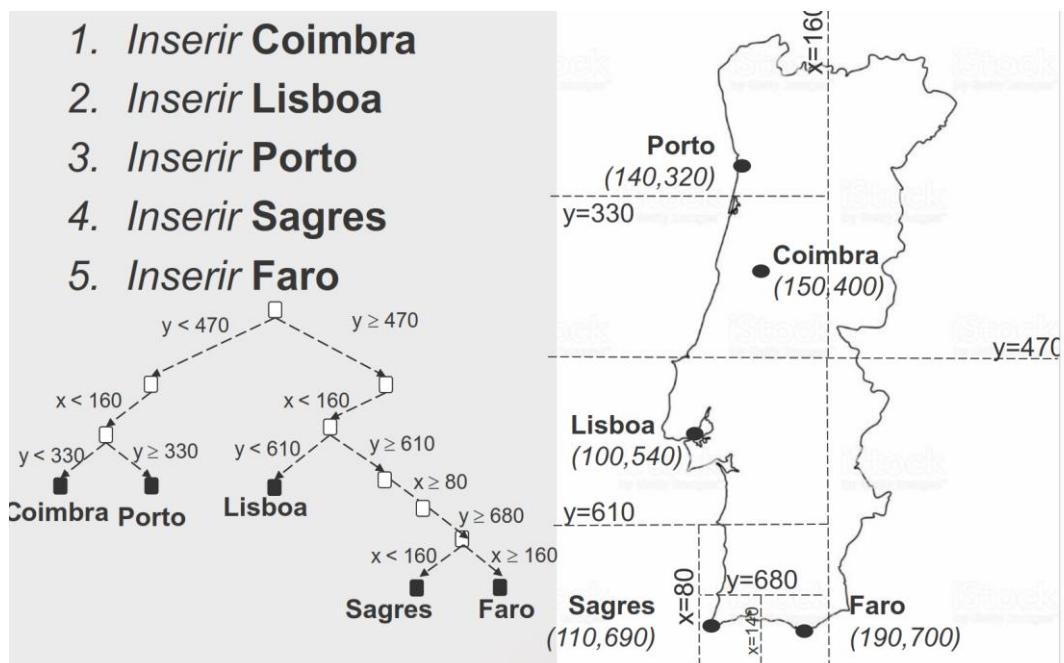
Continuam a não poder haver pontos na mesma célula

Exemplo:

No nível um da arvore considero o Y

No nível dois da arvore considero o X

E repete-se a operação



#Pesquisas

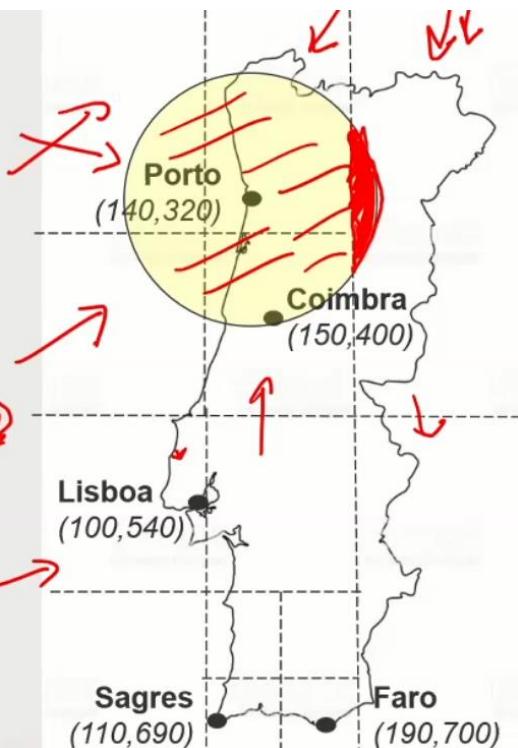
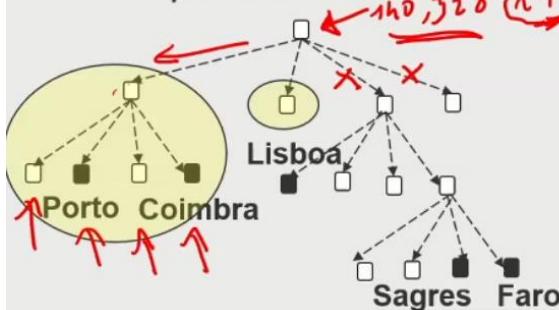
Grelhas fixas, quadtrees e árvores k-d permitem realizar de forma eficiente pesquisas relacionadas com os pontos armazenados

Exemplo:

Quais as cidades a menos de 150 km do Porto?

- Quais as cidades a menos de 150 km do Porto?

– É possível eliminar da pesquisa nodos que não podem conter os pontos indicados



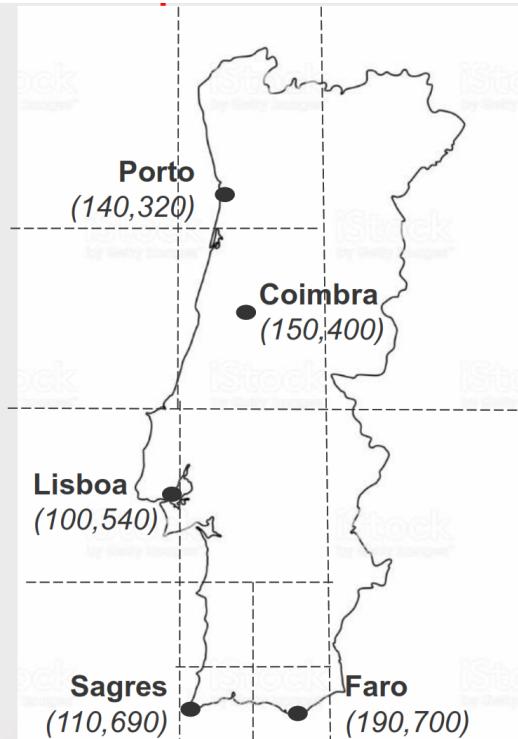
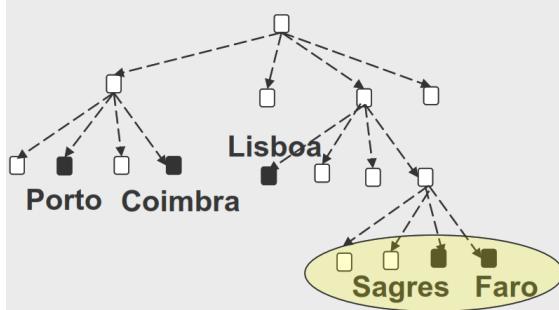
Exemplo:

Qual é a cidade mais próxima de Sagres?

Pesquiso nos quadrantes as cidades candidatas

- Qual a cidade mais próxima de Sagres?

– Pesquisa-se a cidade mais próxima dentro de cada quadrante
– limita-se a pesquisa em função do melhor candidato apurado.



#R-Tree

Como representar as áreas?

A R-Tree é um exemplo de uma árvore que utiliza uma hierarquia de objetos para decompor o espaço

Cada nodo contém uma descrição de uma região que contém os pontos contidos na sua subárvore

As R-Tree seguem uma estrutura análoga à das B-Tree:

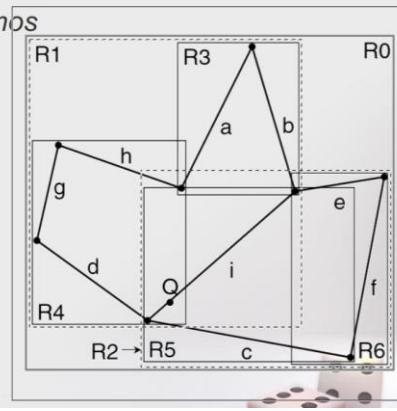
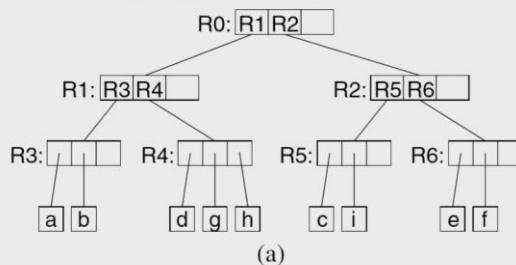
Regiões são índices nos nodos internos

Dados estão nas folhas

- As R-Tree seguem uma estrutura análoga à das B-Tree:

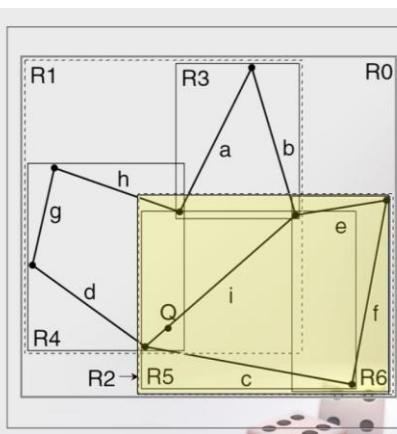
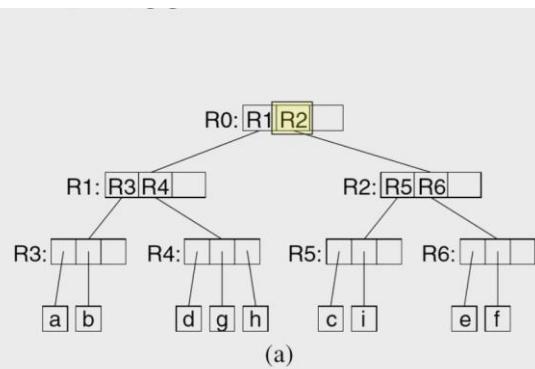
– Regiões são índices nos nodos internos

– Dados estão nas folhas



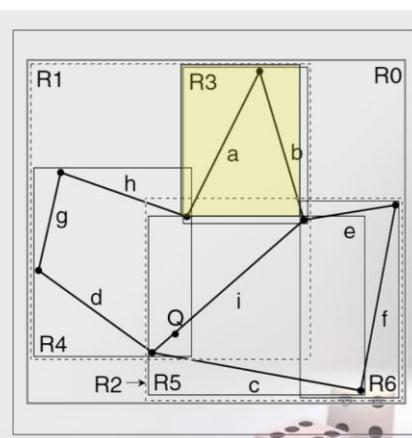
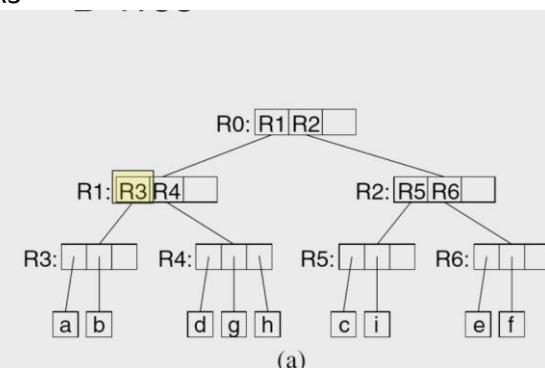
Exemplo:

R2



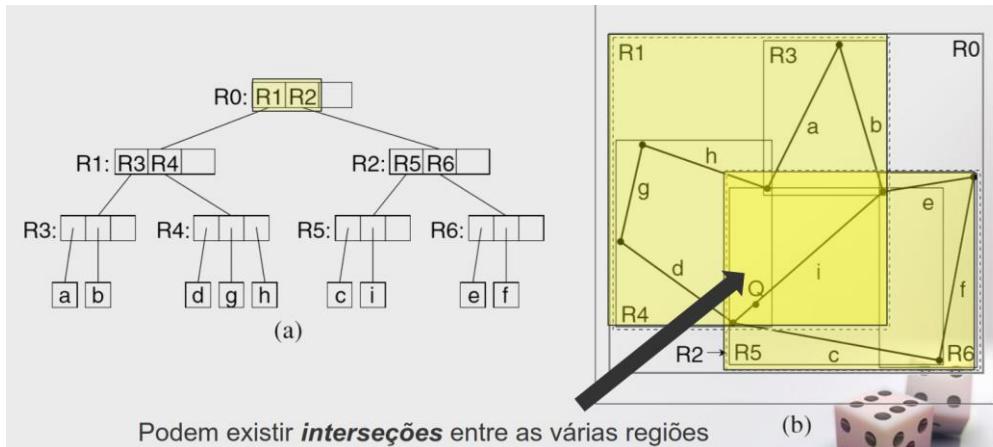
Exemplo:

R3



Exemplo:

R1 e R2



#Pesquisa na R-Tree

A pesquisa de informação num R-Tree é feita verificando se cada uma das regiões pode ou não conter o ponto de interesse

A eficácia da R-Tree depende da sua capacidade de discriminar entre espaço vazio e espaço ocupado

o objetivo nas R-Trees é:

Minimizar a cobertura e as sobreposições das regiões.

Estes objetivos são tidos em conta nos processos de criação e atualização da árvore (que não vamos analisar aqui em detalhe)

Há outras estruturas que tentam resolver estas questões.

Na R+-Tree

a decomposição de uma célula resulta em células disjuntas, evitando assim o problema de ineficiência que pode afetar as R-Trees normais.

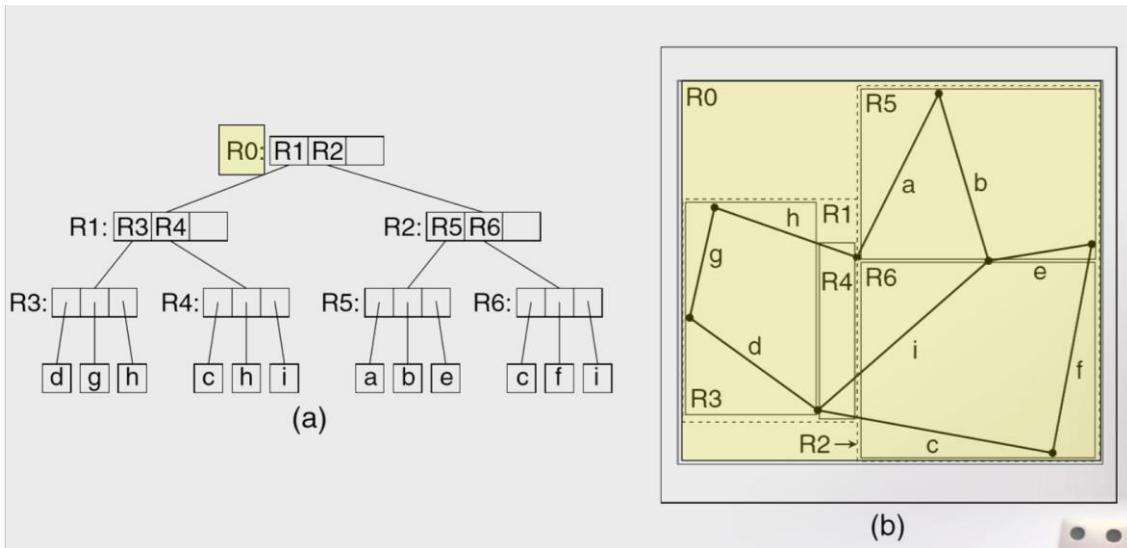
Esta propriedade pode ser violada nas folhas em casos degenerados (p.ex. linhas)

Como consequência desta restrição, a mesma informação pode aparecer em mais do que uma folha e a árvore torna-se mais profunda.

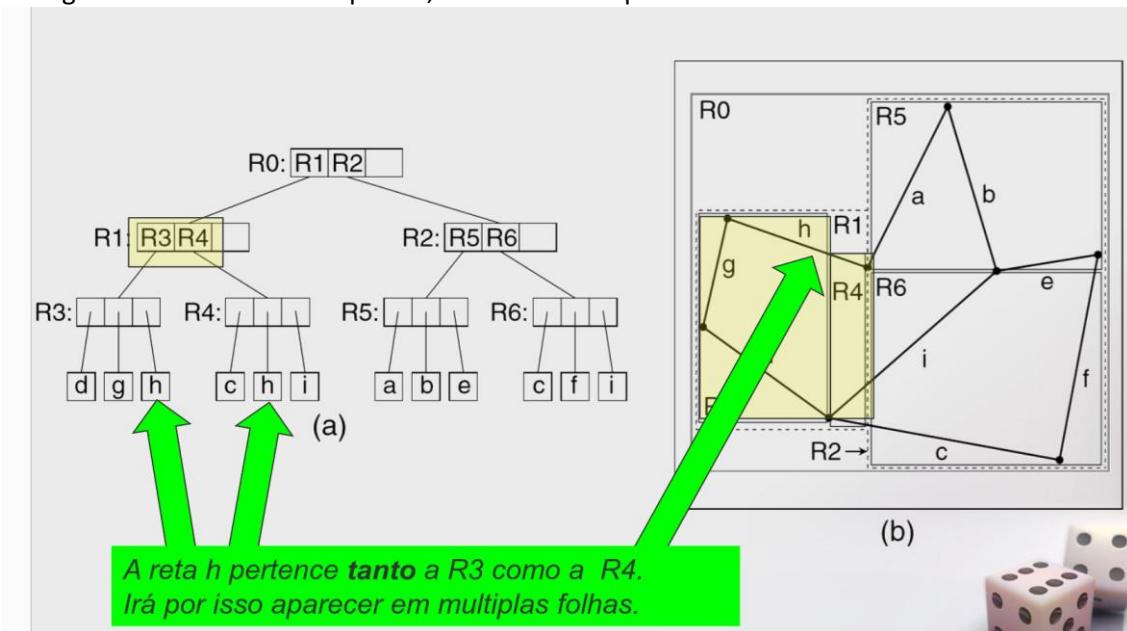
As pesquisas são, no entanto mais rápidas.

(a mesma informação pode aparecer em diferentes partes da árvore, tornando desta forma a árvore mais profunda)

Exemplo:



As regiões nunca estão sobrepostas, mas a recta H aparece em dois locais.



#Representação de Regiões

Há diferentes formas de representar regiões:

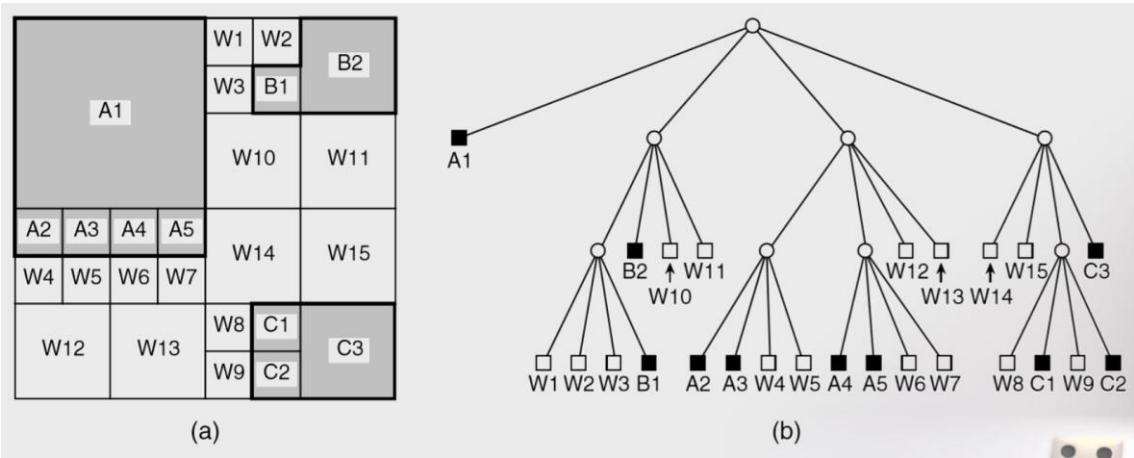
Através da sua fronteira

Através do seu interior

#Quadtree de Regiões

Exemplo:

Uma arvore que representa uma região

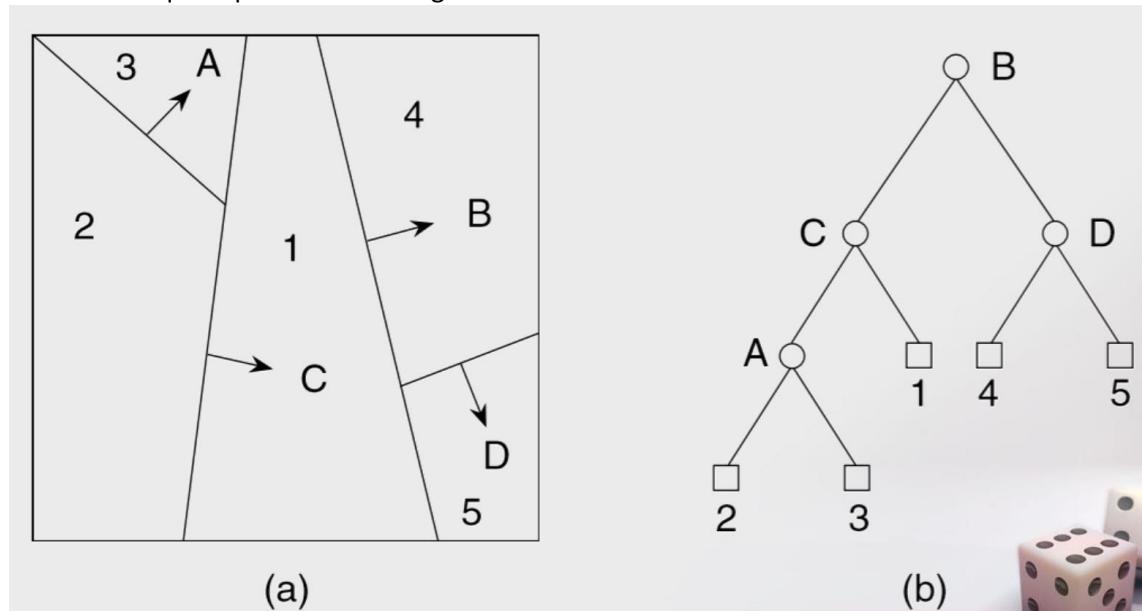


#BSP-Trees

Separar o espaço de forma diferente

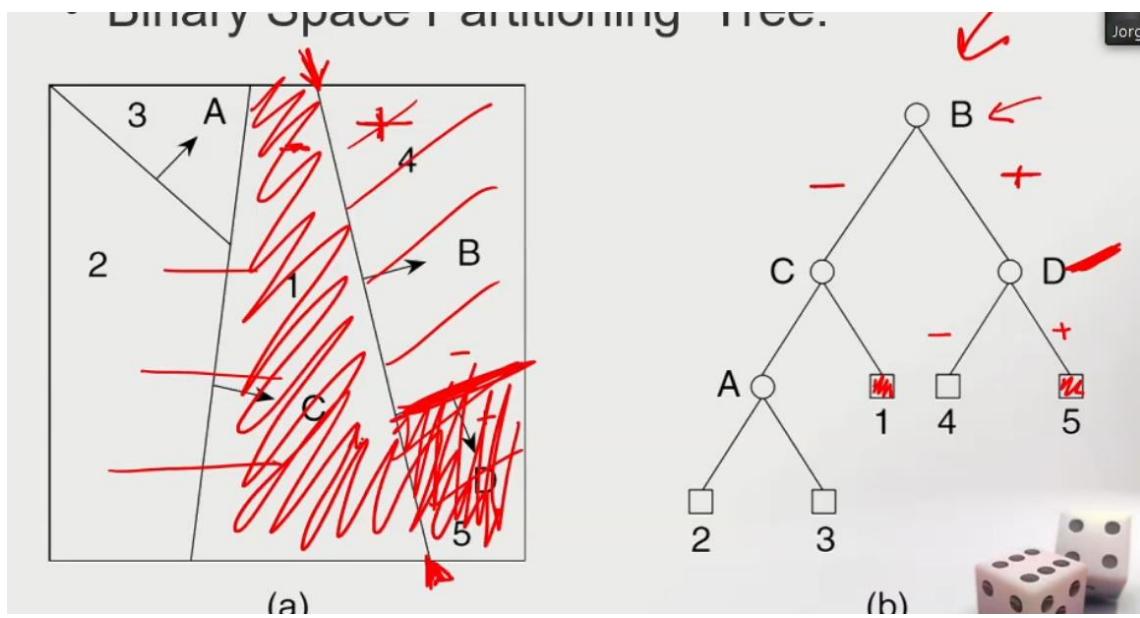
Exemplo:

Uma arvore que representa uma região



Exemplo:

Representar o 1 e o 5



Não falamos de mais...

