Aula 14
Coleções “Informadas”

Algoritmos e Estruturas de Dados

Coleções

“Informadas”

Coleções Cegas vs Informadas

- Até agora, temos aprendido a implementar e trabalhar com coleções cegas
  - Listas
  - Filas
  - Pilhas

Observação: Uma coleção "cega" não olha para os elementos inseridos para decidir como os guardar.

A sua organização não depende dos elementos inseridos.

Coleções Cegas vs Informadas

- Até agora, temos aprendido a implementar e trabalhar com coleções cegas

UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Coleções Cegas vs Informadas

- Agora iremos aprender uma série de coleções que tiram partido de informação sobre os objetos inseridos para decidir como os organizar
  - Relação de ordem
  - Relação hierárquica
  - Relação espacial
  - Chave

Observação: Uma coleção informada vai olhar para informação de cada objeto, e a relação com outros objetos já existentes, para decidir como o organizar e guardar internamente.

Coleções Cegas vs Informadas

- Vantagem coleções informadas

  - Ao organizarmos os objetos guardados de uma forma mais inteligente

  - Conseguimos tornar muito mais eficientes uma série de operações sobre os dados guardados

    Ex: pesquisas

Array ordenado

Array ordenado

- Exemplo mais simples de uma coleção informada

- *Array* que se encontra sempre ordenado

  - Quando inserimos, devemos inserir o elemento de forma a que o *array* continue ordenado

  - Quando queremos procurar um elemento, ou ver se ele existe, conseguimos fazê-lo de forma muito eficiente

Array ordenado

```
public class SortedArray<T extends Comparable<T>> implements Iterable<T> {

    private static final int DEFAULT_INITIAL_SIZE = 10;
    private T[] items;
    private int size;
    private int maxSize;

    public SortedArray()
    {
        //chama o construtor SortedArray(int initialSize)
        this(DEFAULT_INITIAL_SIZE);
    }

    @SuppressWarnings("unchecked")
    public SortedArray(int initialSize)
    {
        this.maxSize = initialSize;
        this.items = (T[]) new Comparable[this.maxSize];
        this.size = 0;
    }
```

Método *rank*

- *rank*(T item)
  - Assumindo que o array de itens está ordenado
  - Retorna o número de elementos < item

    Ou a posição do item no array

  - O valor retornado i pode ser interpretado da seguinte forma:

    Se o item recebido existe no array, o valor i retornado corresponde à posição desse item no array

    Se o item recebido não existe no array, o valor i retornado corresponde à posição onde o elemento pode ser colocado, de forma a que o array continue ordenado

Método *rank e pesquisa binária*

- *rank*(T item)
  - Assumindo que o array de itens está ordenado
  - Retorna o número de elementos < item

    Ou a posição do item no array

  - O método *rank* pode ser implementado de forma eficiente usando uma pesquisa binária

    Ideia da pesquisa binária:

    dividir o problema original num problema com metade do tamanho

Pesquisa binária

- Sabendo que o array se encontra ordenado

    Começamos por procurar o item no meio do array

    Encontrámos o item, item == items[mid]?

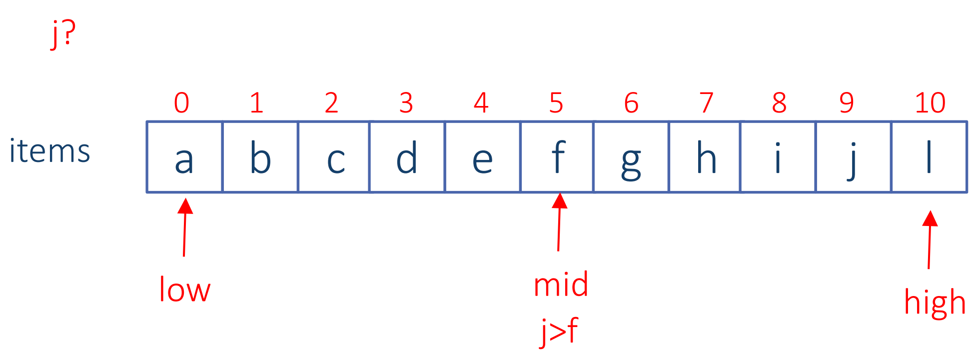
        Então retornamos o índice *mid*.

    item < items[mid]?

        Procuramos a chave no lado esquerdo do array

    item > items[mid]?

        Procuramos a chave no lado direito do array



Pesquisa binária

- Sabendo que o array se encontra ordenado

  Começamos por procurar o item no meio do array

  Encontrámos o item, item == items[mid]?

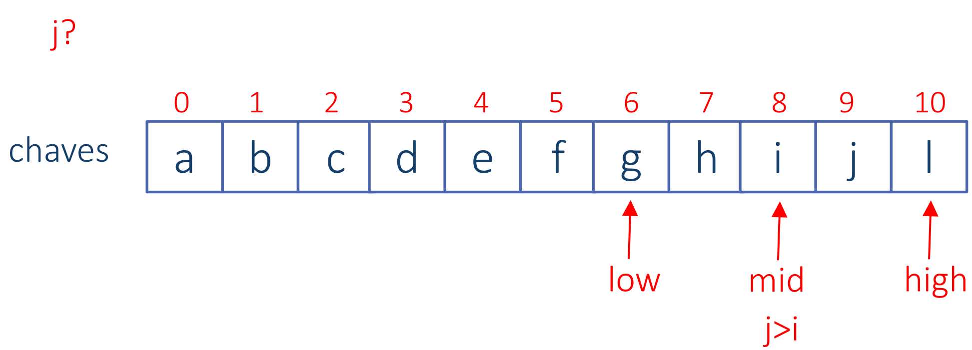
  Então retornamos o índice *mid*.

  item < items[mid]?

  Procuramos a chave no lado esquerdo do array

  item > items[mid]?

  Procuramos a chave no lado direito do array



Pesquisa binária

- Sabendo que o array se encontra ordenado

    Começamos por procurar o item no meio do array

    Encontrámos o item, item == items[mid]?

        Então retornamos o índice *mid*.

    item < items[mid]?

        Procuramos a chave no lado esquerdo do array

    item > items[mid]?

        Procuramos a chave no lado direito do array

j?

	0	1	2	3	4	5	6	7	8	9	10
chaves	a	b	c	d	e	f	g	h	i	j	l
										low, mid	high

j=j!!

Método rank c/ pesquisa binária

```java
private int rank(T item)
{
    int low = 0, high = this.size-1;
    int mid;
    int cmp;

    while(low <= high)
    {
        mid = low + (high - low)/2;
        cmp = item.compareTo(this.items[mid]);
        if (cmp < 0) high = mid-1;
        else if (cmp > 0) low = mid+1;
        else return mid;
    }

    return low;
}
```

Usamos 3 variáveis para representar:
low - o lado esquerdo
high – o lado direito
mid - meio
do *subarray* onde estamos a procurar no momento

Este cmp vai retornar 1 de 3 valores possíveis:
< 0, se item < item do meio
> 0, se item > item do meio
= 0, se item = item do meio

Se <0, continuar a procura do lado esquerdo
ignoramos o lado direito

Se >0, continuar a procura do lado direito
ignoramos o lado esquerdo

Se =0, podemos parar imediatamente,
encontrámos o item

Método rank c/ pesquisa binária

```
private int rank(T item)
{
    int low = 0, high = this.size-1;
    int mid;
    int cmp;

    while(low <= high)
    {
        mid = low + (high - low)/2;
        cmp = item.compareTo(this.items[mid]);
        if (cmp < 0) high = mid-1;
        else if (cmp > 0) low = mid+1;
        else return mid;
    }

    return low;
}
```

Usamos 3 variáveis para representar:
low - o lado esquerdo
high – o lado direito
mid - meio
do *subarray* onde estamos a procurar no momento

aqui podemos usar uma versão não recursiva, através deste ciclo

Este ciclo termina quando low = high+1, e isto acontece quando o item que estamos à procura não existe no array.

Nessa situação, low corresponde à posição onde poderíamos colocar o novo item, pois todos os elementos à esquerda são < que item e todos os items à direita (se existirem) são > que item

Método *contains*

Este método retorna *true* se o item recebido existir no *array*, e *false* caso contrário

```
public boolean contains(T item)
{
    if (this.size == 0) return false;
    int i = rank(item);
    if(i < this.size && this.items[i].compareTo(item) == 0) return true;
    else return false;
}
```

O método *rank* retorna um valor entre 0 e *high*+1. Mas *high*+1 = *size*.

Quando o valor retornado é igual a *size*, quer dizer que o item que estamos à procura é maior que todos os outros, e portanto na realidade não está no *array*. Nesse caso, caso queiramos adicionar como novo elemento, a posição i adequada seria size.

Portanto consideramos que o item já existe se *i < size*, e se o item guardado na posição i é efetivamente o item de que estamos à procura

Método *add*

```
public void add(T item)
{
    if(this.size == this.maxSize)
    {
        resize(this.maxSize*2);
    }

    int i = rank(item);

    //shift right all elements to the right of i
    for(int j = this.size; j > i; j--)
    {
        this.items[j] = this.items[j-1];
    }

    this.items[i] = item;
    this.size++;
}
```

procura o lugar de inserção

"Shift Right" de 1 posição de todos os elementos à direita de i
Pois precisamos de arranjar espaço para colocar o novo item

Isto pode eventualmente ser trocado por uma cópia em bloco

agora que já temos espaço, podemos inserir o item

Método Iterador

```java
private class SAIterator implements Iterator<T>
{
    int index;
    SAIterator()
    {
        this.index = 0;
    }

    @Override
    public boolean hasNext() {
        return this.index < size;
    }

    @Override
    public T next() {
        return items[this.index++];
    }
}

@Override
public Iterator<T> iterator() {
    return new SAIterator();
}
```

SortedArray

Complexidade Temporal

Complexidade temporal rank

```
private int rank(T item)
{
    int low = 0, high = this.size-1;
    int mid;
    int cmp;

    while(low <= high)
    {
        mid = low + (high - low)/2;
        cmp = item.compareTo(this.items[mid]);
        if (cmp < 0) high = mid-1;
        else if (cmp > 0) low = mid+1;
        else return mid;
    }

    return low;
}
```

considerando n = (high – low) +1

(high – low) = n - 1

Em cada iteração ou retornamos imediatamente depois da comparação, ou reduzimos n para metade.

Se cmp < 0, n = $(\underbrace{\text{high}}_{\text{mid - 1}} – \text{low})+1$

= mid – 1 – low + 1 = mid – low

mid – low = (high – low)/2

= (high – low)/2 = **(n-1)/2**

Complexidade temporal rank

- No pior caso (o item não existe no *array* ordenado)
  - *Nunca saimos mais cedo*

comparação com elemento do meio

  - $T_{rank}(n) \leq 1 + T_{rank}\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$
  - $T_{rank}\ (n) \leq 1 + 1 + T_{rank}\left(\left\lfloor \frac{n}{4} \right\rfloor\right)$
  - $T_{rank}\ (n) \leq \underbrace{1 + 1 + \cdots + 1}_{\log_2 n}$
  - $T_{rank}\ (n) \leq 1 + \log_2 n$
  - $T_{rank}\ (n) \sim \log_2 n$

Complexidade temporal rank

- No caso médio (o item existe em posição incerta)
  - *Saimos mais cedo quando o encontramos*
  - *Podemos encontrá-lo na 1.ª tentativa, ou na última*
  - *Em média, vamos fazer metade das iterações do ciclo.*
  - $T_{rank}\ (n) \leq \frac{1+\log_2 n}{2}$
  - $T_{rank}\ (n) \sim \frac{\log_2 n}{2}$

Complexidade temporal contains

```
public boolean contains(T item)
{
    if (this.size == 0) return false;
    int i = rank(item);
    if(i < this.size && this.items[i].compareTo(item) == 0) return true;
    else return false;
}
```

A complexidade temporal do método *contains* é dada pela complexidade do método *rank*, pois todas as outra operações têm tempo constante e podem ser ignoradas

~½ $\log_2 n$ caso médio
~ $\log_2 n$ pior caso

Complexidade temporal *add*

```
public void add(T item)
{
    if(this.size == this.maxSize)
    {
        resize(this.maxSize*2);
    }

    int i = rank(item);

    //shift right all elements to the right of i
    for(int j = this.size; j > i; j--)
    {
        this.items[j] = this.items[j-1];
    }

    this.items[i] = item;
    this.size++;
}
```

~ n, mas se considerarmos o custo amortizado isto pode ser considerado constante

~½ $\log_2 n$ caso médio
~ $\log_2 n$ pior caso

~½ n caso médio
~ n pior caso

Complexidade temporal *add*

```java
public void add(T item)
{
    if(this.size == this.maxSize)
    {
        resize(this.maxSize*2);
    }

    int i = rank(item);

    //shift right all elements to the right of i
    for(int j = this.size; j > i; j--)
    {
        this.items[j] = this.items[j-1];
    }

    this.items[i] = item;
    this.size++;
}
```

~ n, mas se considerarmos o custo amortizado isto pode ser considerado constante

~½ $\log_2 n$ caso médio
~ $\log_2 n$ pior caso

~½ n caso médio
~ n pior caso

☹ a operação de "shift" estraga o que poderia ser uma boa complexidade temporal

Caso médio: ½ $\log_2 n$ + ½ n **~ ½ n**
Pior caso: $\log_2 n$ + n **~ n**

Complexidade temporal SortedArray

	Caso Médio		Pior Caso	
	Inserção	Pesquisa	Inserção	Pesquisa
Array Sequencial	1	n/2	1	n
Array Ordenado	n/2	½ $\log_2$ n	n	$\log_2$ n

A inserção não é tão
simpática por causa dos shifts

Complexidade temporal SortedArray

	Caso Médio		Pior Caso	
	Inserção	Pesquisa	Inserção	Pesquisa
Array Sequencial	1	n/2	1	n
Array Ordenado	n/2	½ $\log_2 n$	n	$\log_2 n$

Observação:
Embora a técnica de pesquisa binária seja muito eficiente, um Array Ordenado não é uma boa coleção informada devido ao custo elevado de fazermos inserções.

Dica: No entanto, se tivermos um array já construído e quisermos fazer muitas pesquisas sobre esse array, podemos usar um algoritmo de ordenação e pesquisas binárias sobre esse array.

Árvores Binárias

Disclaimer:
Na sua forma mais simples árvores binárias são consideradas coleções cegas. No entanto como são muito usadas para implementar coleções informadas, vamos começar por estudar o que é uma árvore binária.

Listas Ligadas

- Embora tenham algumas vantagens

- Listas ligadas são substancialmente menos usadas que os arrays
  - O dinamismo das listas pode ser facilmente obtido através de um array redimensionável
  - Os arrays têm a grande vantagem de serem um bloco contínuo de memória

Listas Ligadas

- Então pq raios perdemos tanto tempo a falar de listas ligadas?

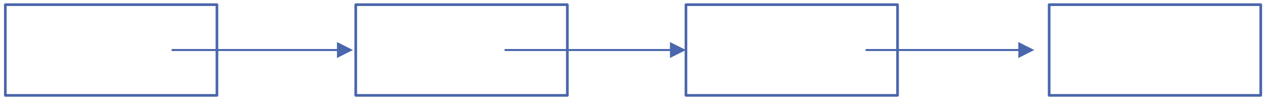
Listas Ligadas

- Então pq raios perdemos tanto tempo a falar de listas ligadas?

- Foi um bom treino para a próxima fase

Lista Ligada

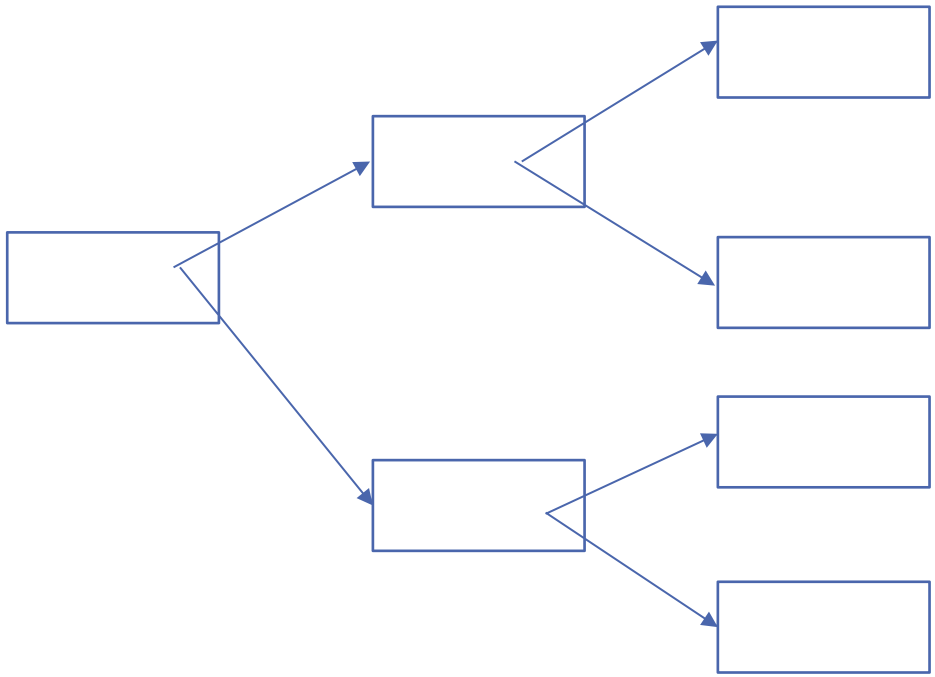
- Sequência de nós ligados entre si



- Então e se cada nó puder apontar para 2 nós?

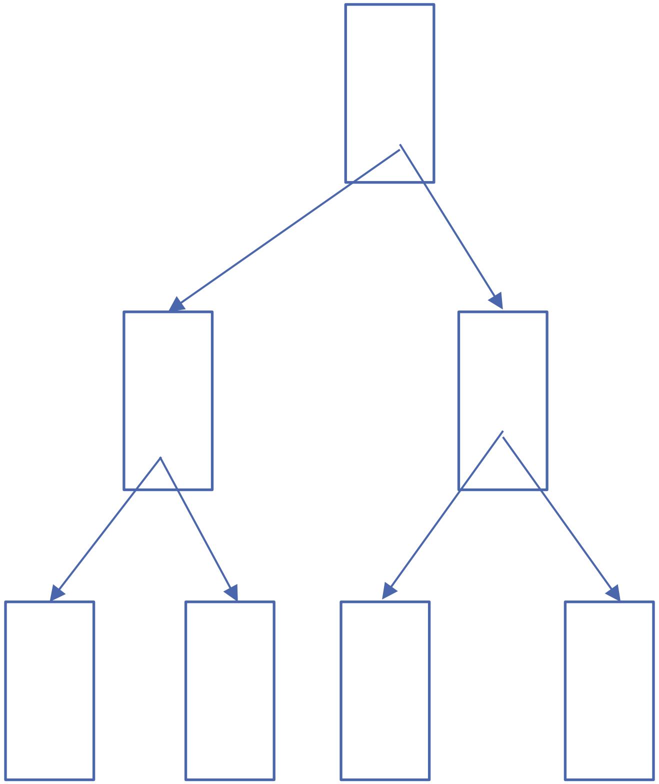
Árvore Binária

- Sequência de nós ligados entre si



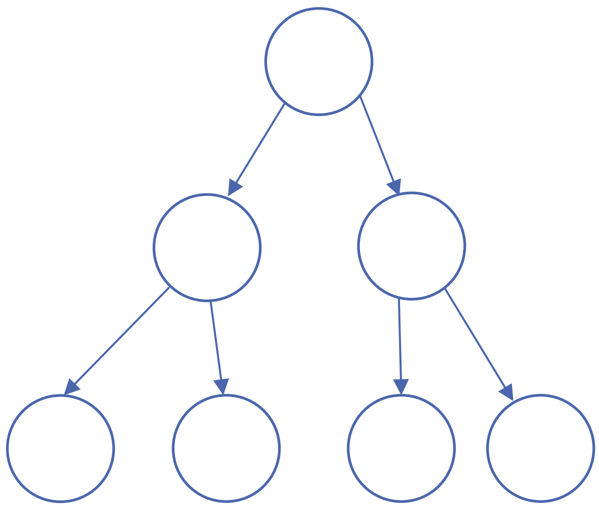
Árvore Binária

- Costuma-se representar uma árvore de cima para baixo



Árvore Binária

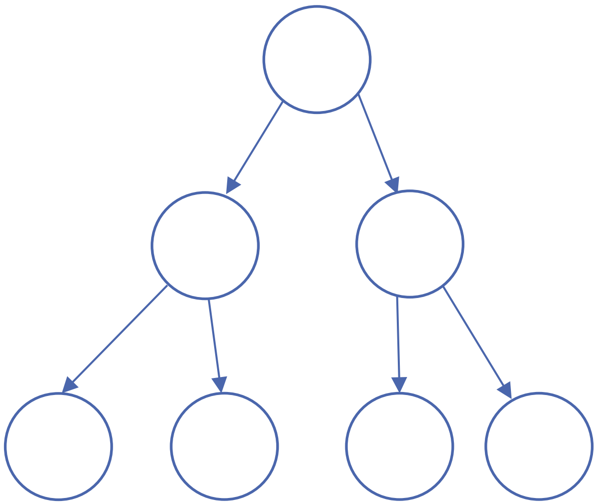
- E vamos usar círculos para representar os nós



Definição Árvore Binária

Definição recursiva:

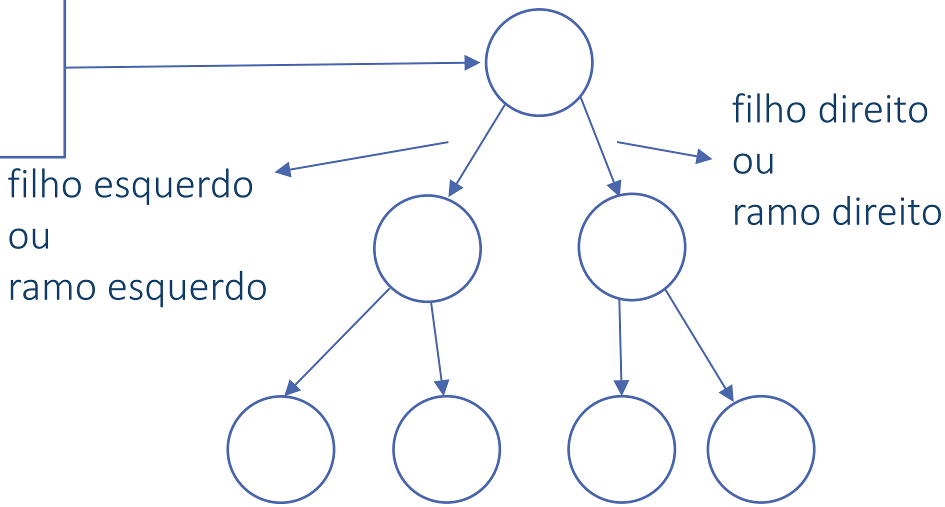
- *Uma árvore vazia é uma árvore binária*
- *Um nó com um valor e dois ponteiros para outras 2 árvores (filhas) é uma árvore binária*



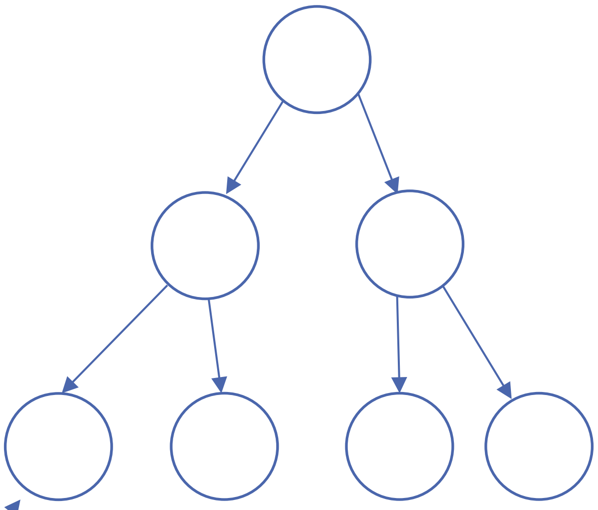
```
private class Node<T>
{
    private T item;
    private Node<T> left;
    private Node<T> right;
    private int size;
```

Nomenclatura Árvores Binárias

O 1.º nó da árvore (não tem pai) é designado de raiz da árvore.



Nomenclatura Árvores Binárias

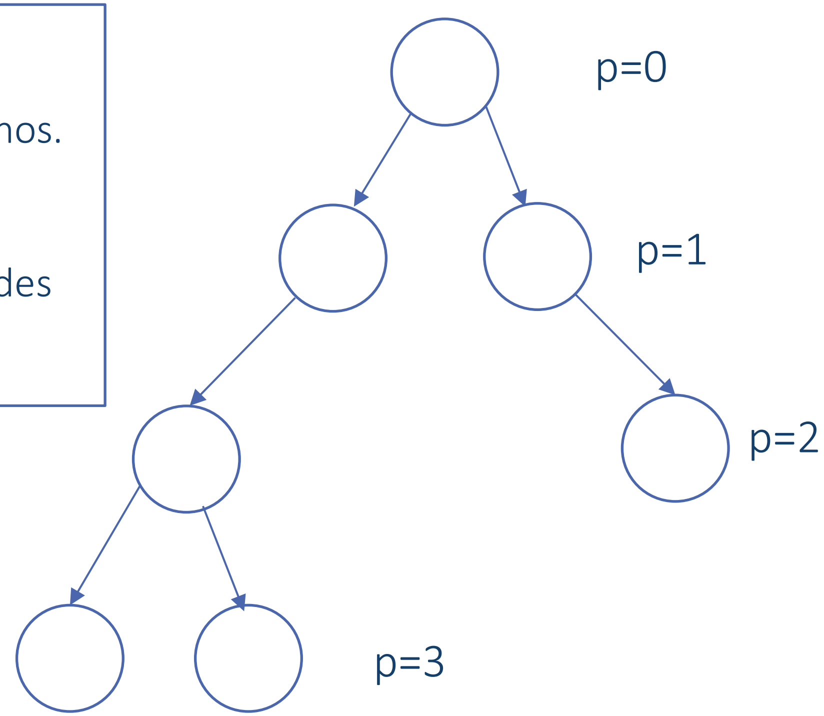


Um nó sem filhos (ou seja, os filhos são vazios) é designado de folha

Características Árvores Binárias

Observação:
Não é obrigatório que um nó tenha sempre os dois filhos.
Um nó pode ter 0 filhos (folha), 1 filho, ou dois filhos.

Caminhos diferentes na árvore podem ter profundidades diferentes

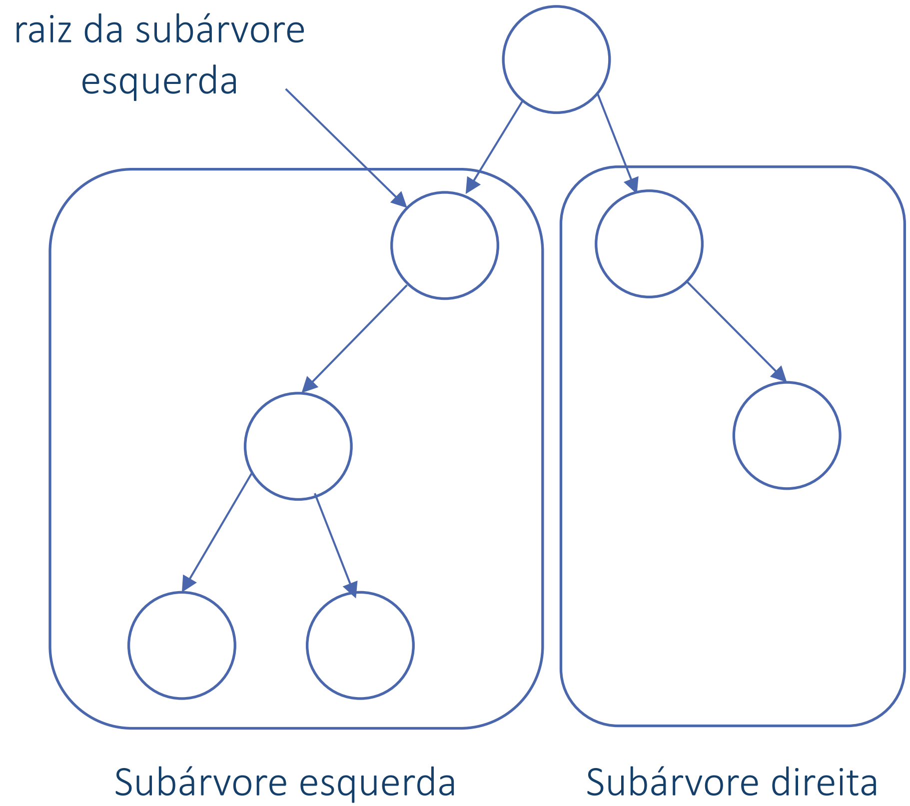


Def:
A profundidade de um nó numa árvore corresponde à distância (número de ramos) do nó à raiz da árvore

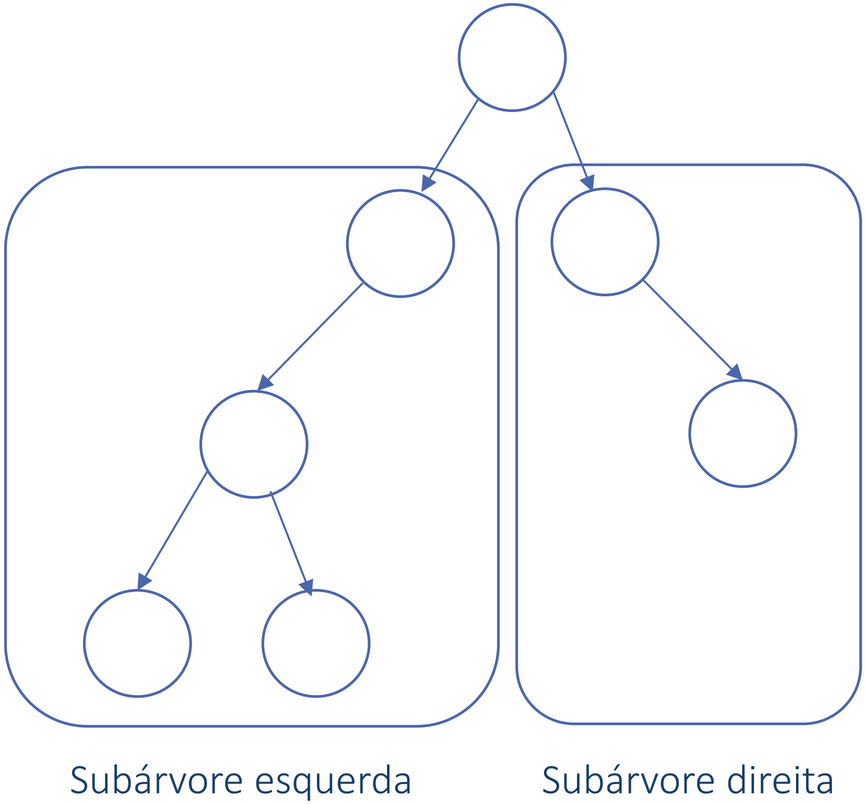
Esta árvore é uma árvore binária perfeitamente válida

Características Árvores Binárias

Observação:
Dada a definição recursiva, o filho de uma árvore é também ele considerado uma árvore (ou subárvore).



Características Árvores Binárias



Observação:
Agora que atingimos o nível seguinte, podemos fazer algumas coisas engraçadas com árvores binárias.