

# Árvores Binárias de Busca

Prof. Dr. Jonas Lopes de Vilas Boas

# Estruturas de dados

- Processos particulares para **armazenamento, organização e manipulação** de dados.
- São dinâmicas e tornam o código mais organizado.
- Podem ser classificadas em:
  - lineares ou não lineares;
  - homogêneas ou heterogêneas;
  - estáticas ou dinâmicas;

# Busca em estruturas de dados

- Algoritmos estratégicos para encontrar dados em uma estrutura de dados.
- Cada tipo de organização necessita de uma estratégia específica.
- A busca termina quando os dados buscados são encontrados.

# Busca sequencial

- Nessa estratégia, cada elemento da estrutura de dados é visitado de maneira sequencial (de acordo com a ordem de inserção, por exemplo).
- Exemplo: Buscar uma carta em um baralho embaralhado.
- Complexidade:  **$O(n)$** , onde  **$n$**  é a quantidade de elementos

Buscar a carta 5♥

A♥	10♠	K♦	Q♣	2♠	5♥	6♣
----	-----	----	----	----	----	----

Buscar a carta 5♥

A♥	10♠	K♦	Q♣	2♠	5♥	6♣
----	-----	----	----	----	----	----



Buscar a carta 5♥

A♥	10♠	K♦	Q♣	2♠	5♥	6♣
----	-----	----	----	----	----	----



Buscar a carta 5♥

A♥	10♠	K♦	Q♣	2♠	5♥	6♣
----	-----	----	----	----	----	----





Buscar a carta 5♥

A♥	10♠	K♦	Q♣	2♠	5♥	6♣
----	-----	----	----	----	----	----



Buscar a carta 5♥

A♥	10♠	K♦	Q♣	2♠	5♥	6♣
----	-----	----	----	----	----	----



Buscar a carta 5♥

A♥	10♠	K♦	Q♣	2♠	5♥	6♣
----	-----	----	----	----	----	----



# Busca binária

- Nessa estratégia, os elementos devem estar previamente ordenados.
- A busca começa por um elemento central.
- Caso o elemento buscado seja maior que o elemento central, a busca começa novamente considerando apenas a metade superior da estrutura, ou a metade inferior, caso o elemento seja menor.
- Exemplo: Buscar uma carta em um baralho ordenado.
- Complexidade:  $O(\log_2 n)$ , onde  $n$  é o número de elementos.

# Buscar a carta 5♥

A♥	10♠	K♦	Q♣	2♠	5♥	6♣
----	-----	----	----	----	----	----

Vamos considerar que A é o menor valor e K é o maior.

Os naipes são colocados na ordem: ♦ ♠ ♥ ♣

Buscar a carta 5♥

A♥	2♠	5♥	6♣	10♠	Q♣	K♦
----	----	----	----	-----	----	----

Buscar a carta 5♥

A♥	2♠	5♥	6♣	10♠	Q♣	K♦
----	----	----	----	-----	----	----



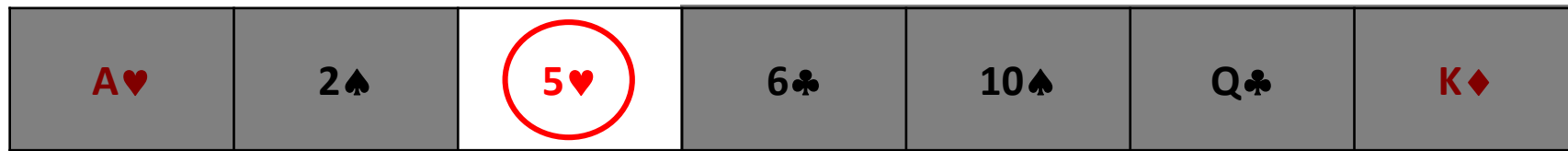
Buscar a carta 5♥

A♥	2♠	5♥	6♣	10♠	Q♣	K♦
----	----	----	----	-----	----	----





Buscar a carta 5♥



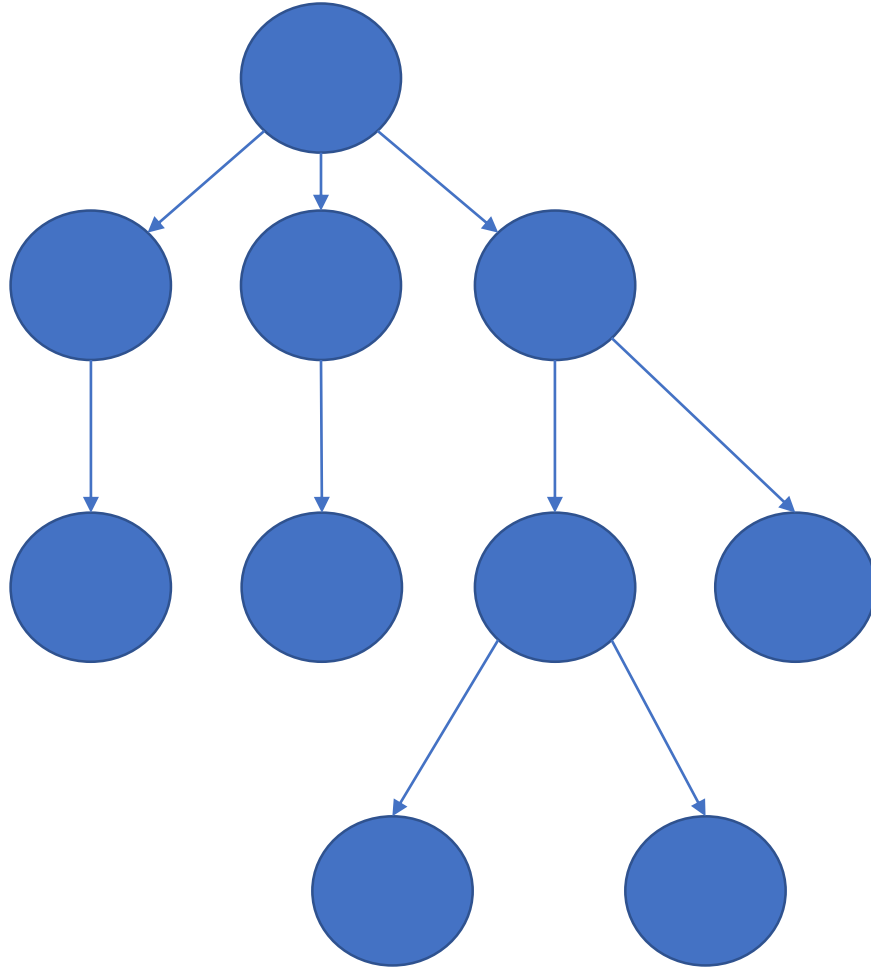
# Buscas em estruturas de dados dinâmicas

- Diferente das estruturas de dados estáticas, as estruturas de dados dinâmicas não são necessariamente lineares.
- Mesmo quando são lineares (as listas encadeadas, por exemplo), na prática essas estruturas podem dificultar a indexação por posição.
  - Por exemplo, pode ser difícil saber qual é o elemento central.
- Para isso outras estruturas dinâmicas podem ser usadas para melhorar a eficiência das buscas.

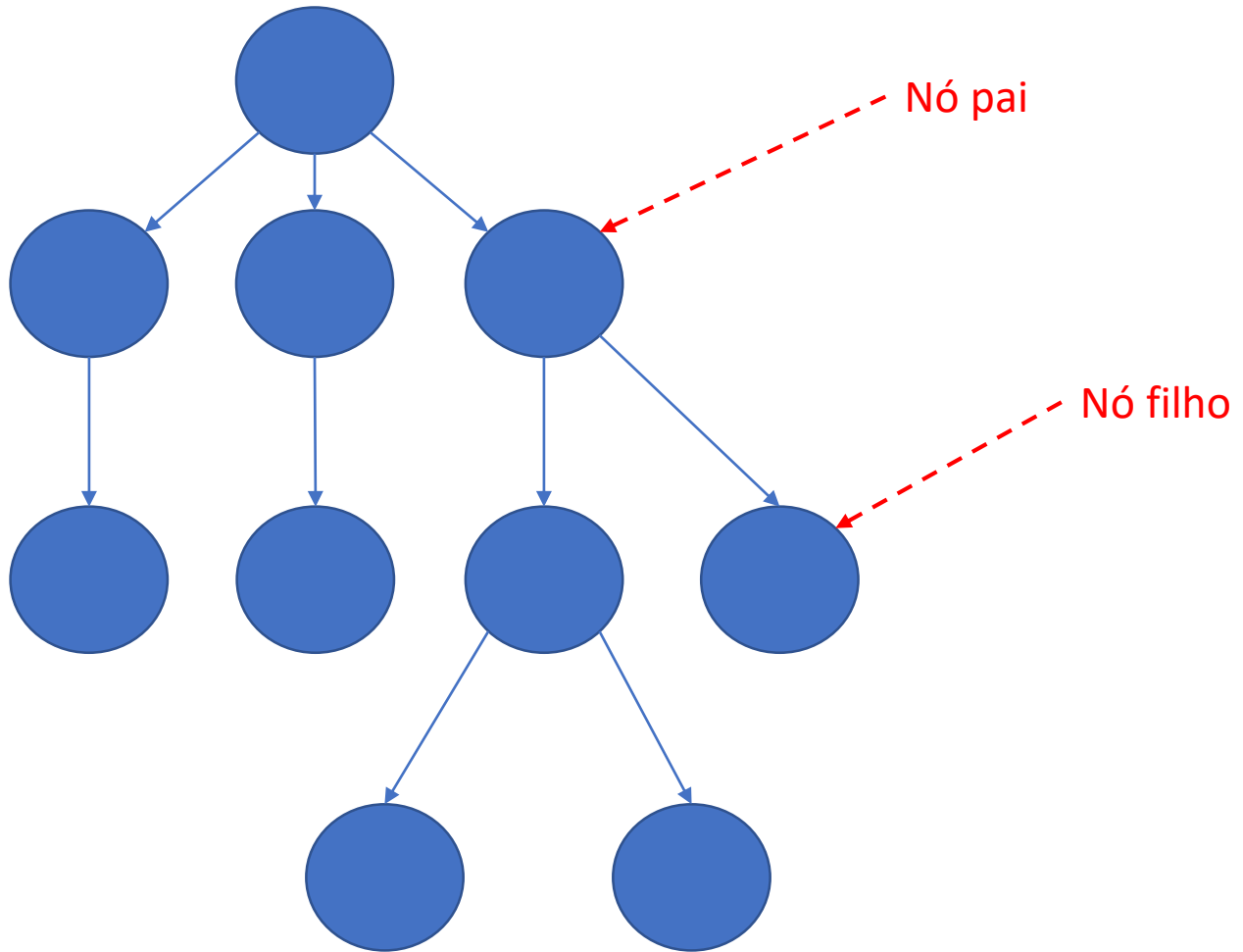
# Árvore

- É um tipo de estrutura de dados não linear e dinâmica.
- É um Grafo Acíclico e Direcionado.
- Armazena os dados de forma hierárquica.

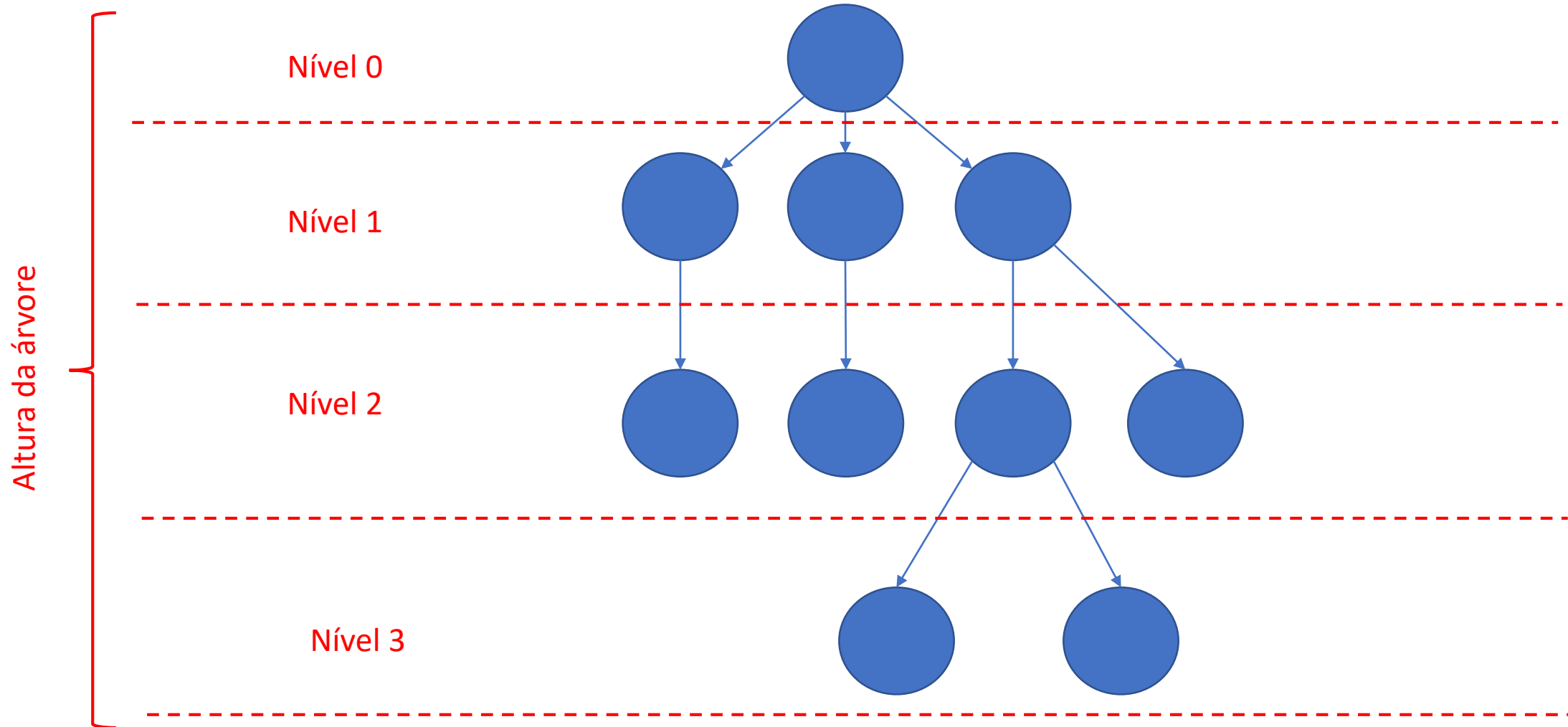
# Exemplo de Árvore



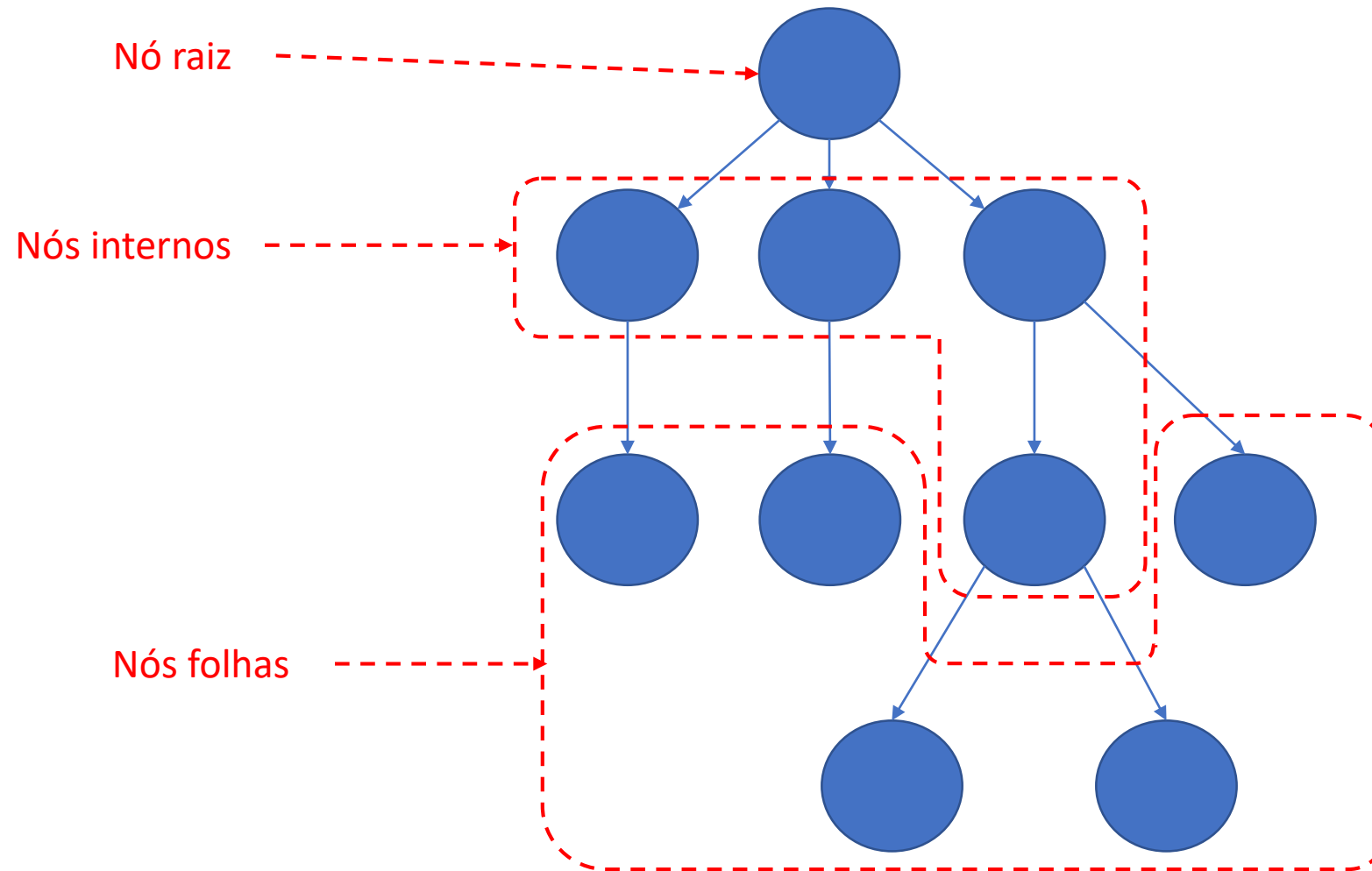
# Exemplo de Árvore



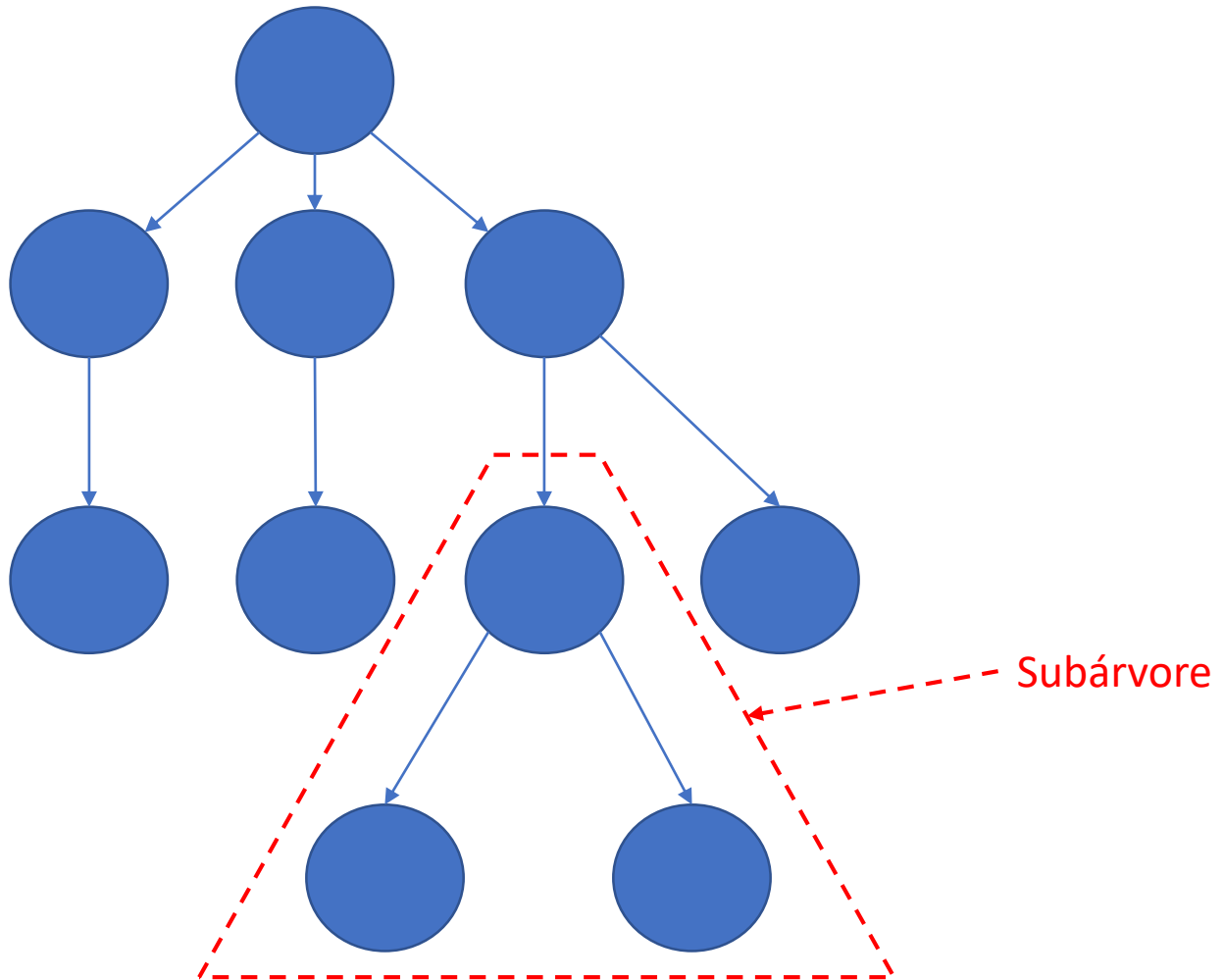
# Exemplo de Árvore



# Exemplo de Árvore



# Exemplo de Árvore

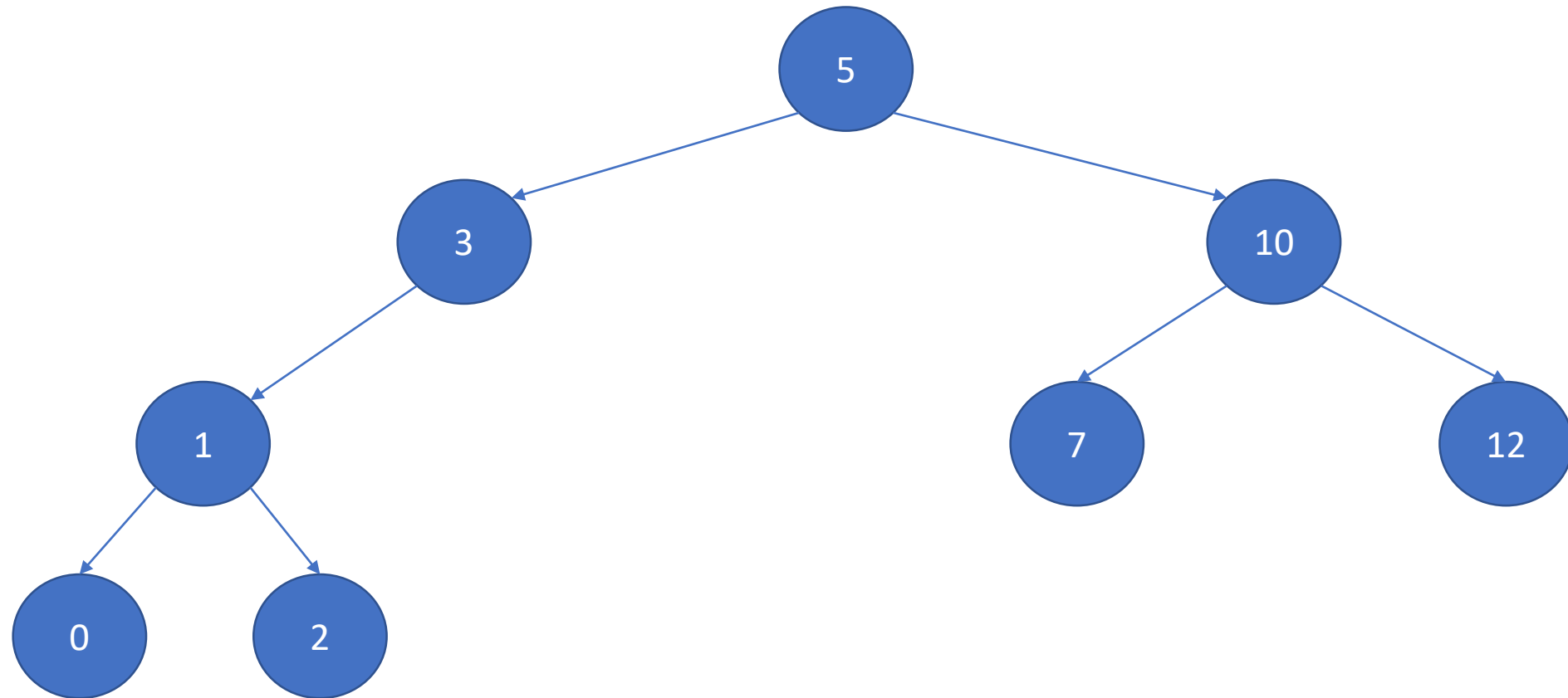




# Árvore Binária

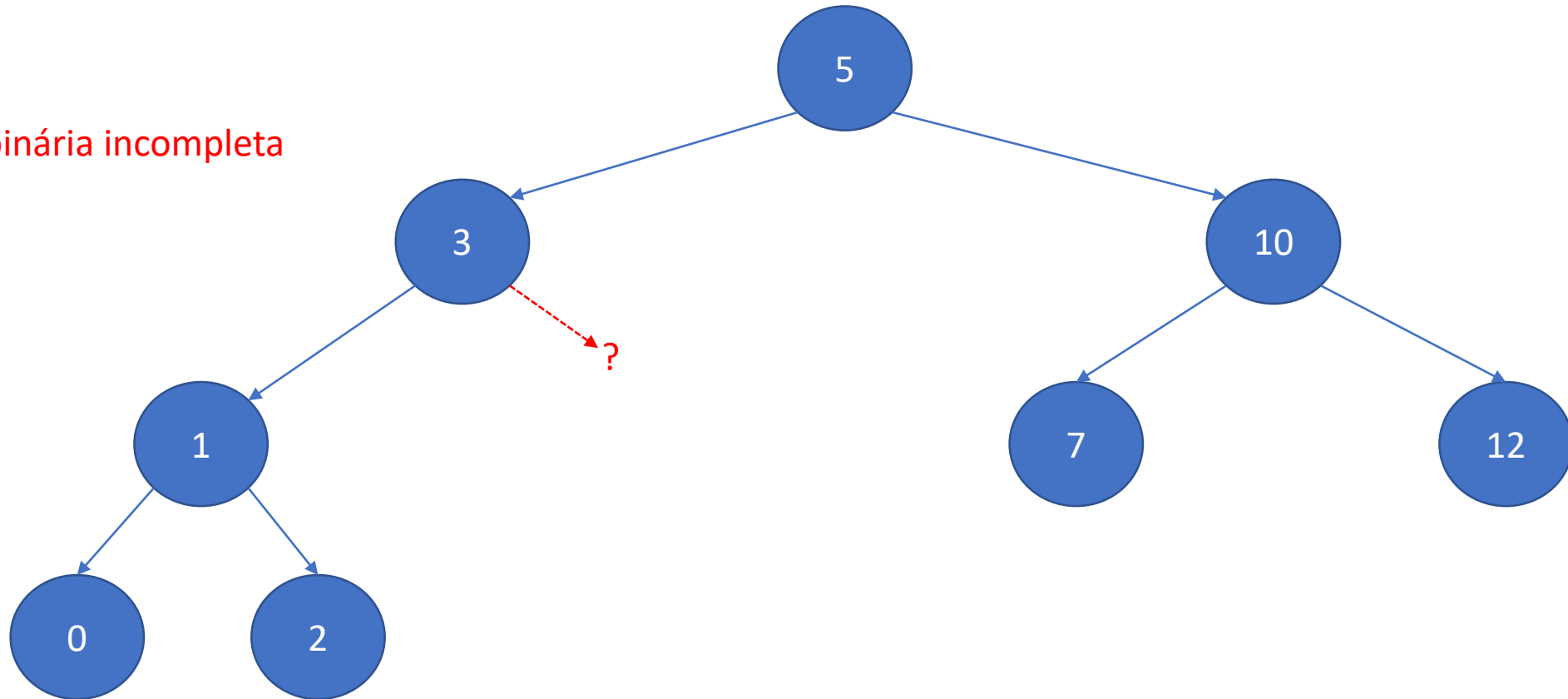
- É um tipo de árvore onde os nós podem ter no máximo dois filhos (filho da esquerda e filho da direita).
- É muito utilizada para otimização de buscas.

# Exemplo de Árvore Binária de Busca



# Exemplo de Árvore Binária de Busca

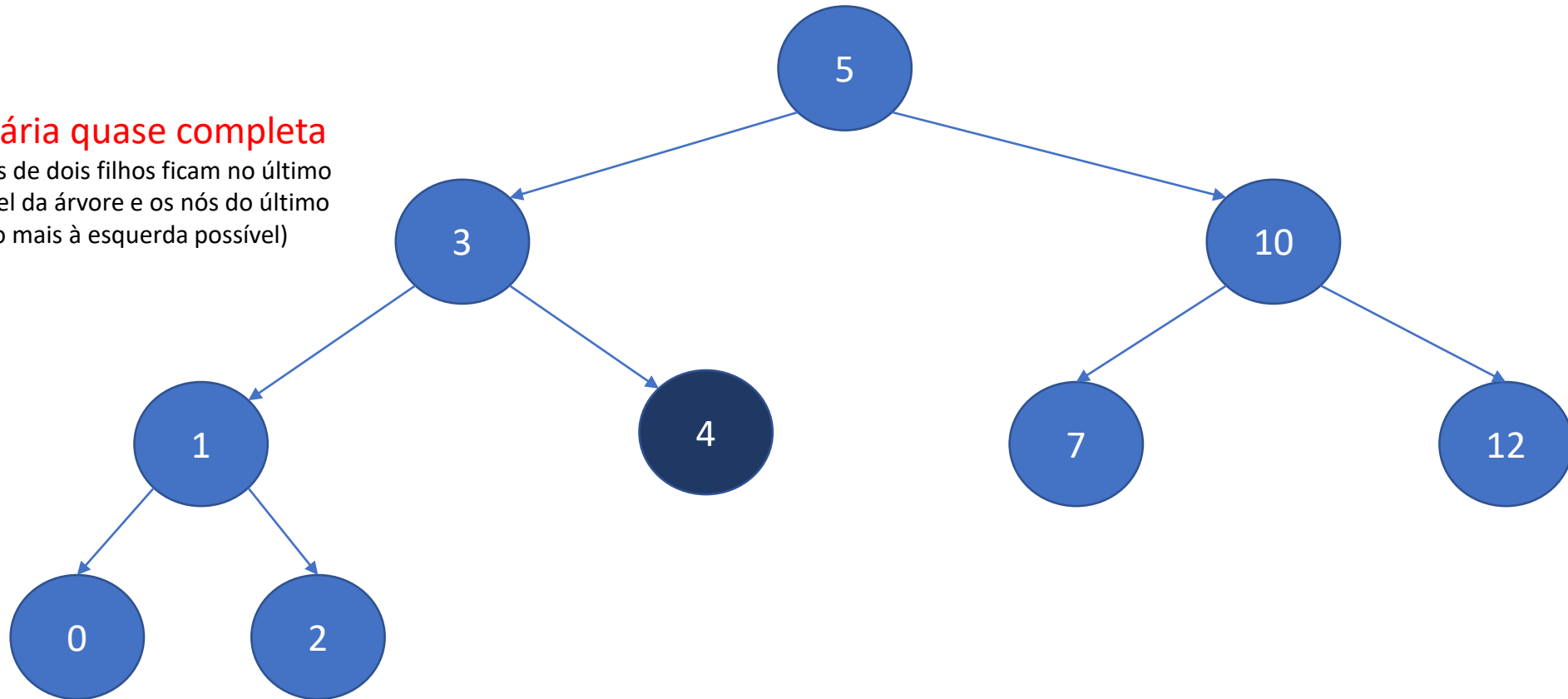
Árvore binária incompleta



# Exemplo de Árvore Binária de Busca

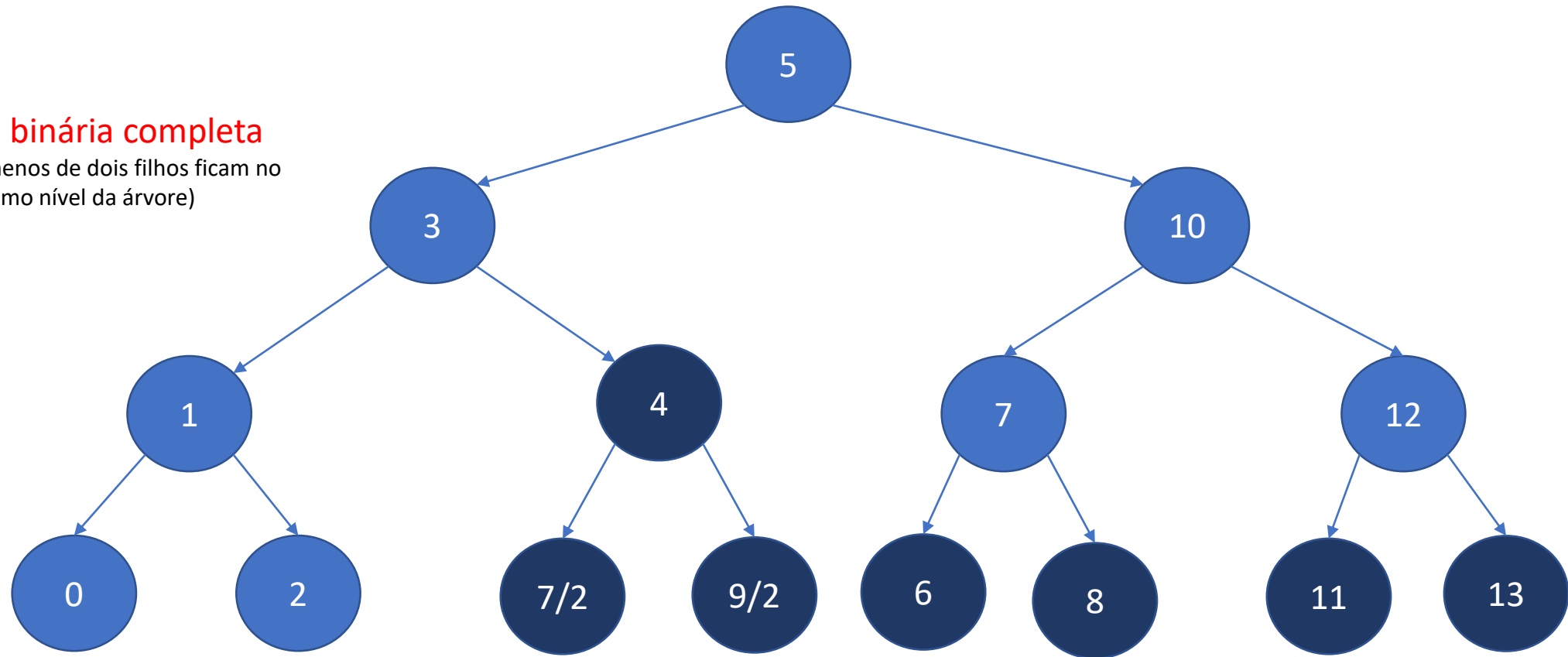
## Árvore binária quase completa

(Nós com menos de dois filhos ficam no último e penúltimo nível da árvore e os nós do último nível ficam o mais à esquerda possível)



# Exemplo de Árvore Binária de Busca

Árvore binária completa  
(Nós com menos de dois filhos ficam no último nível da árvore)



# Árvore Binária de Busca

- É um tipo de árvore binária que é construída com uma estratégia para facilitar a busca binária.
- A estratégia é já construir a estrutura de maneira ordenada:
  - Ao inserir um novo nó, ele é comparado com os nós já existentes;
  - Começando da raiz, se o nó a ser inserido tem valor menor que o nó visitado, ele continua a verificação para a esquerda, senão para a direita;
  - Se o nó visitado for um nó folha, o novo nó é inserido à esquerda se o valor for maior, ou à direita se for menor.
- Complexidade da busca:  $O(h)$ , onde  $h$  é a altura da árvore

# Árvore Binária de Busca

- Representação de um nó da árvore

```
struct treenode {  
    int info;  
    struct treenode * left;  
    struct treenode * right;  
};
```

# Árvore Binária de Busca

- Algoritmo de inserção:

```
void tInsert(treenodeptr &p, int x){  
    if (p == NULL) {  
        p = new treenode;  
        p->info = x;  
        p->left = NULL;  
        p->right = NULL;  
    } else if (x < p->info)  
        tInsert(p->left, x);  
    else  
        tInsert(p->right, x);  
}
```



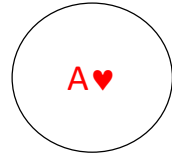
# Buscar a carta 5♥

A♥	10♠	K♦	Q♣	2♠	5♥	6♣
----	-----	----	----	----	----	----

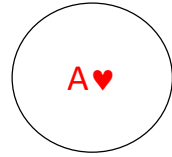
Vamos considerar que A é o menor valor e K é o maior.

Os naipes são colocados na ordem: ♦ ♠ ♥ ♣

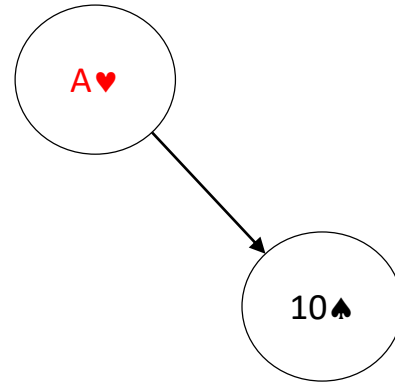
Inserindo a carta A♥



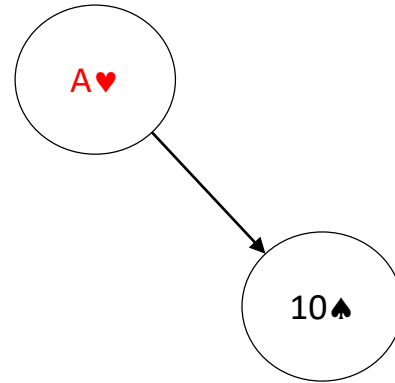
Inserindo a carta 10♠



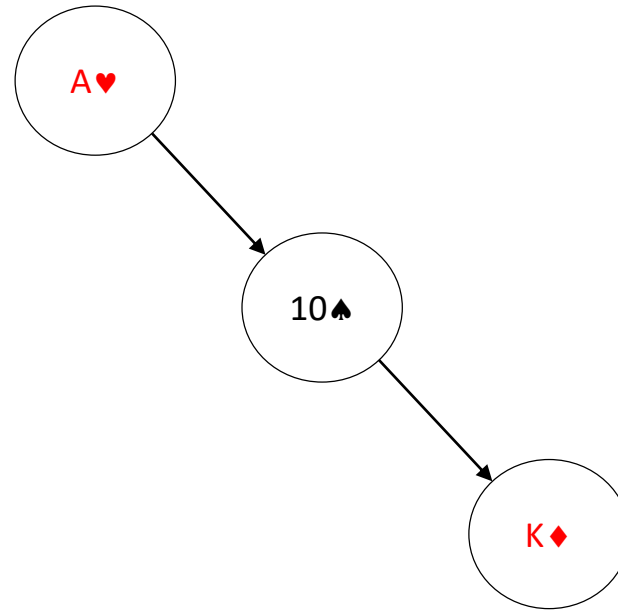
# Inserindo a carta 10♠



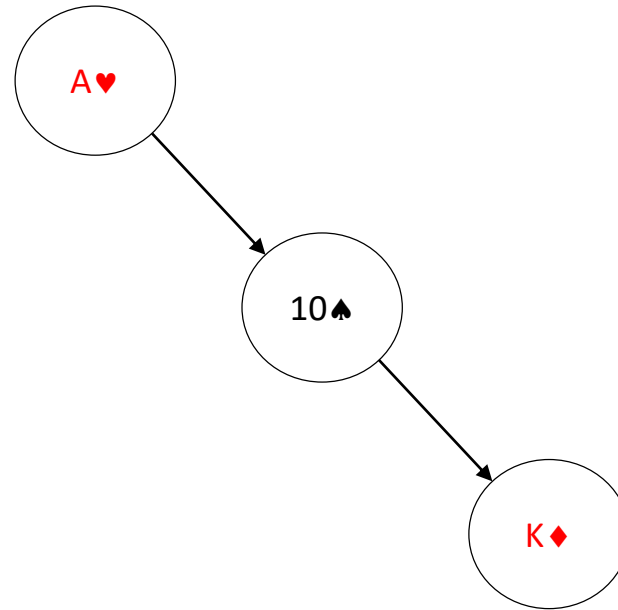
# Inserindo a carta K♦



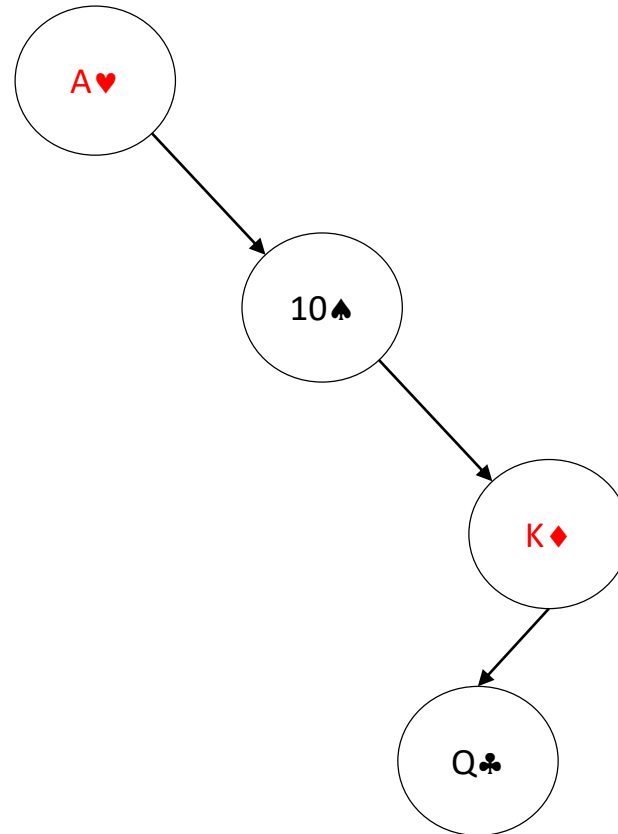
# Inserindo a carta **K♦**



# Inserindo a carta Q♣

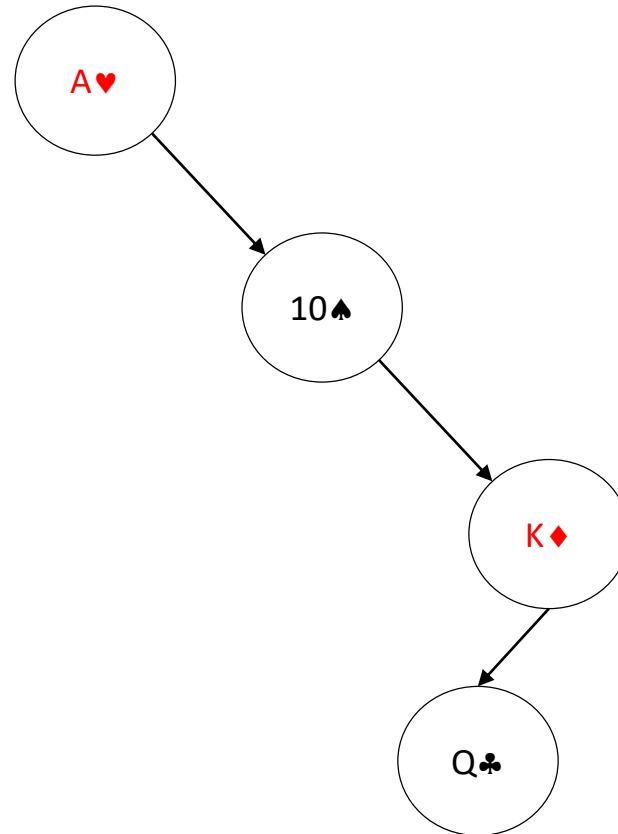


# Inserindo a carta Q♣

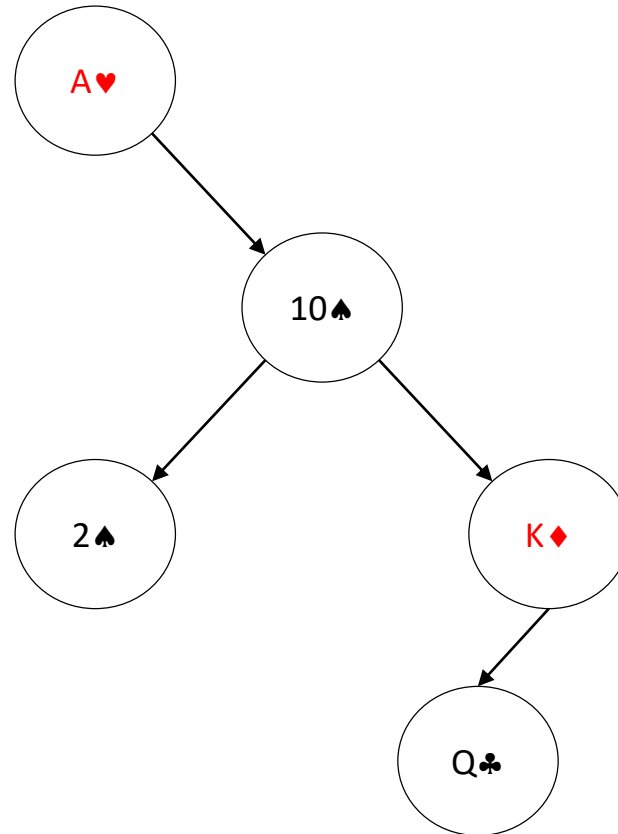




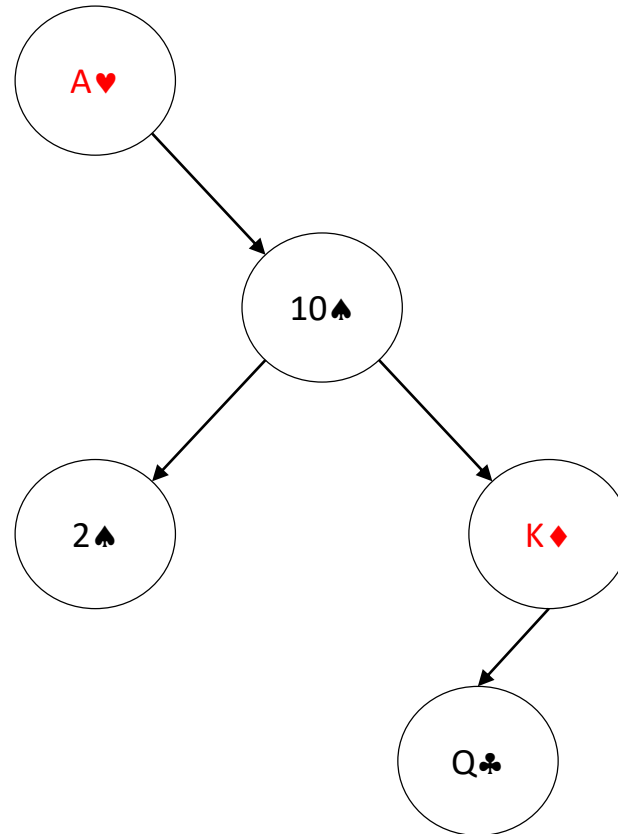
# Inserindo a carta 2♠



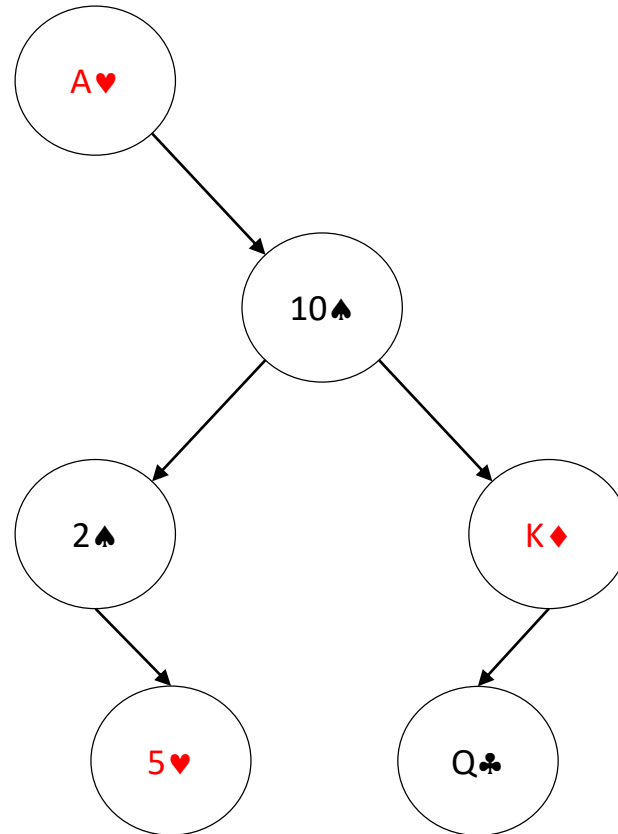
# Inserindo a carta 2♠



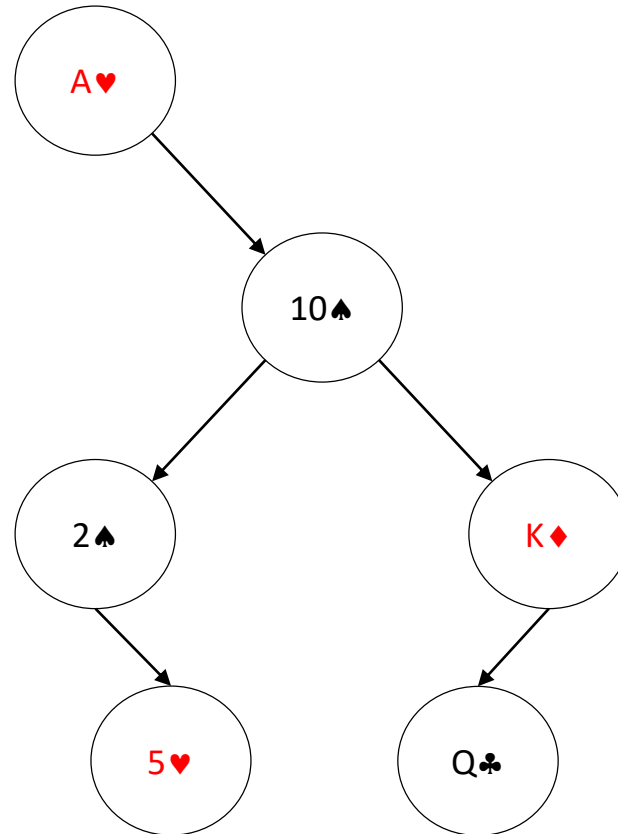
# Inserindo a carta 5♥



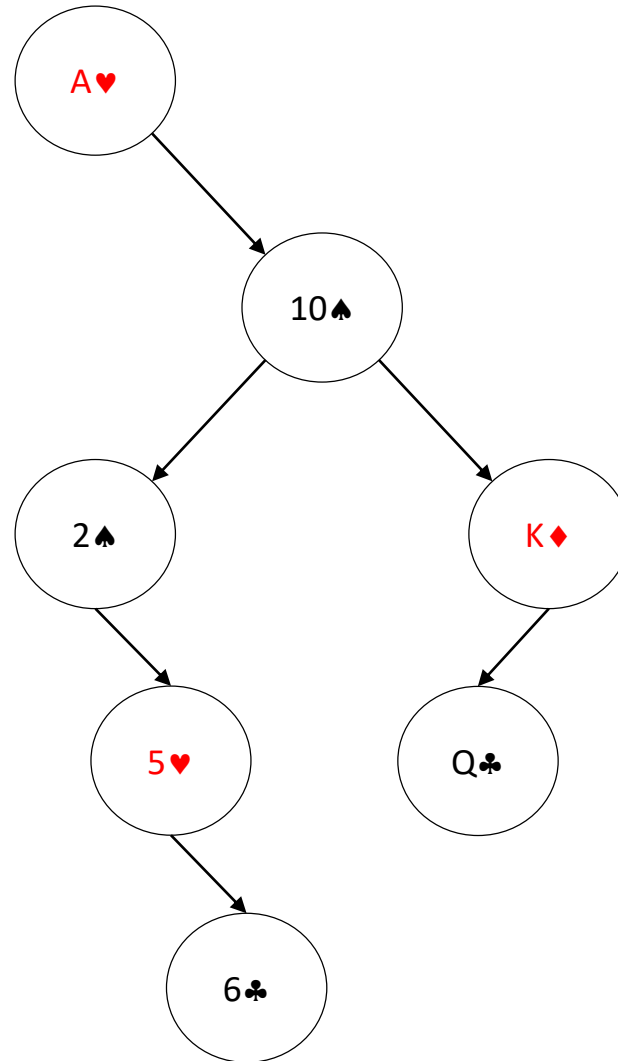
# Inserindo a carta 5♥



# Inserindo a carta 6♣



# Inserindo a carta 6♣

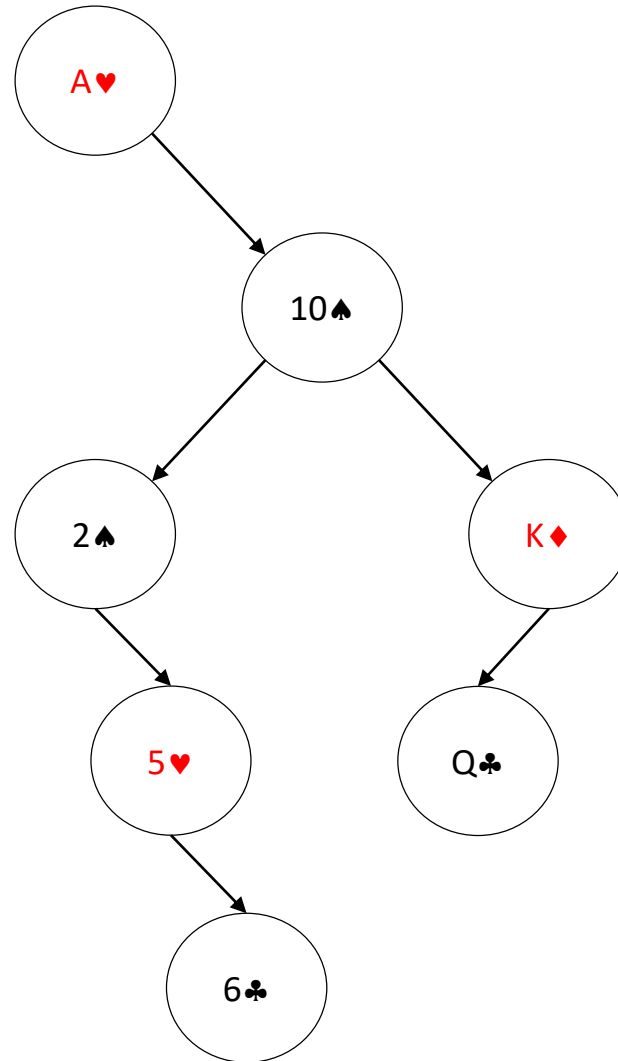


# Árvore Binária de Busca

- Algoritmo de busca:

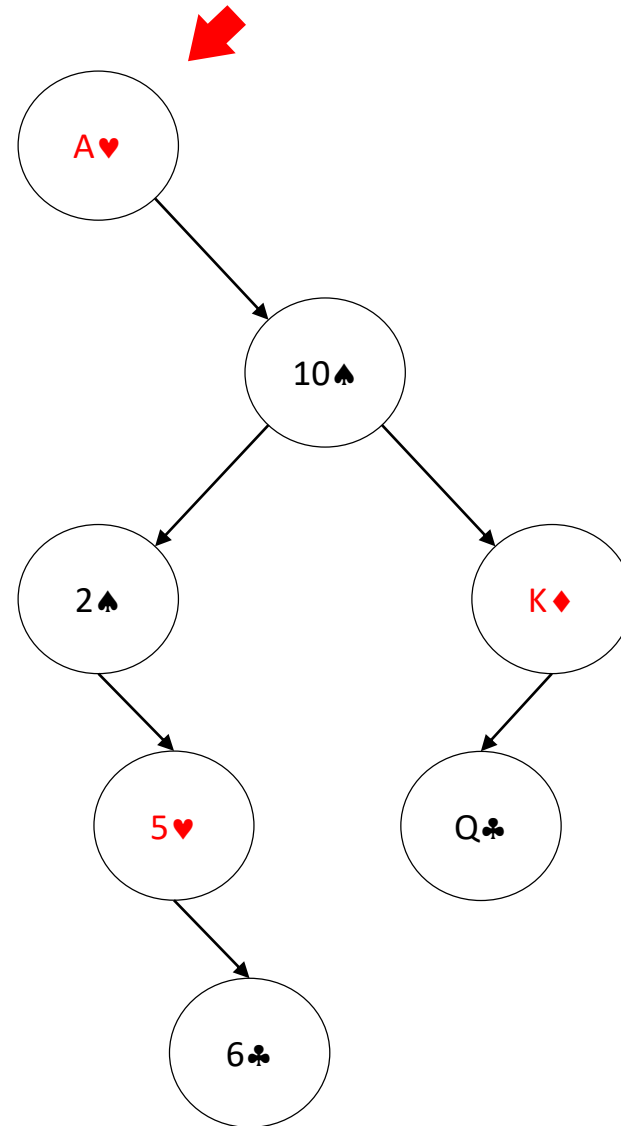
```
treenodeptr tSearch(treenodeptr p, int x) {  
    if (p == NULL)  
        return NULL;  
    else if (x == p->info)  
        return p;  
    else  
        if (x < p->info)  
            return tSearch(p->left,x);  
        else  
            return tSearch(p->right,x);  
}
```

# Buscar a carta 5♥

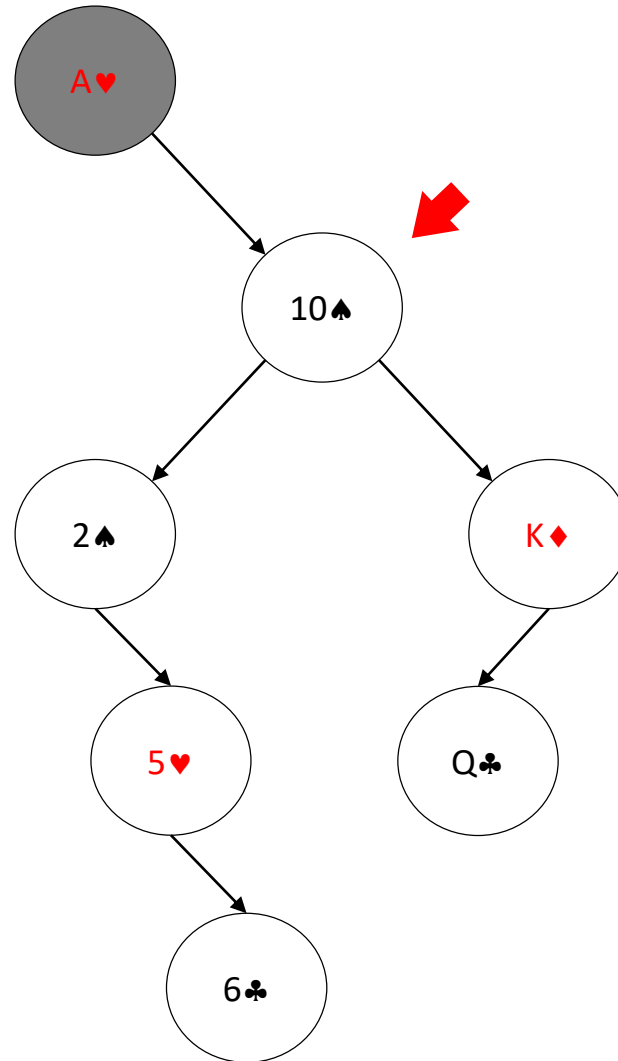




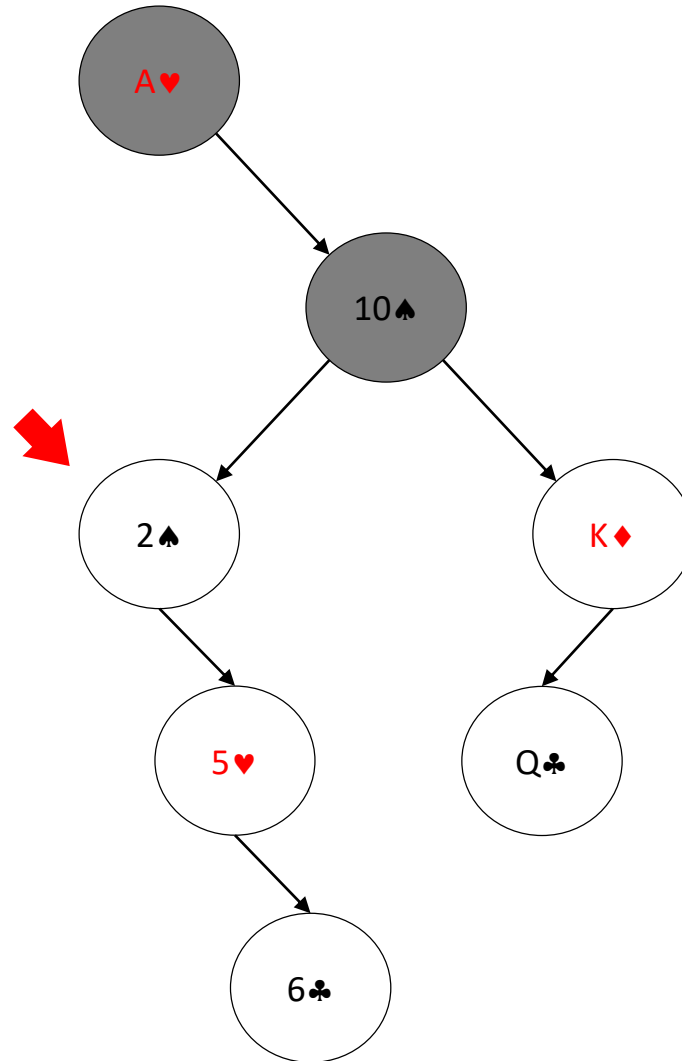
# Buscar a carta 5♥



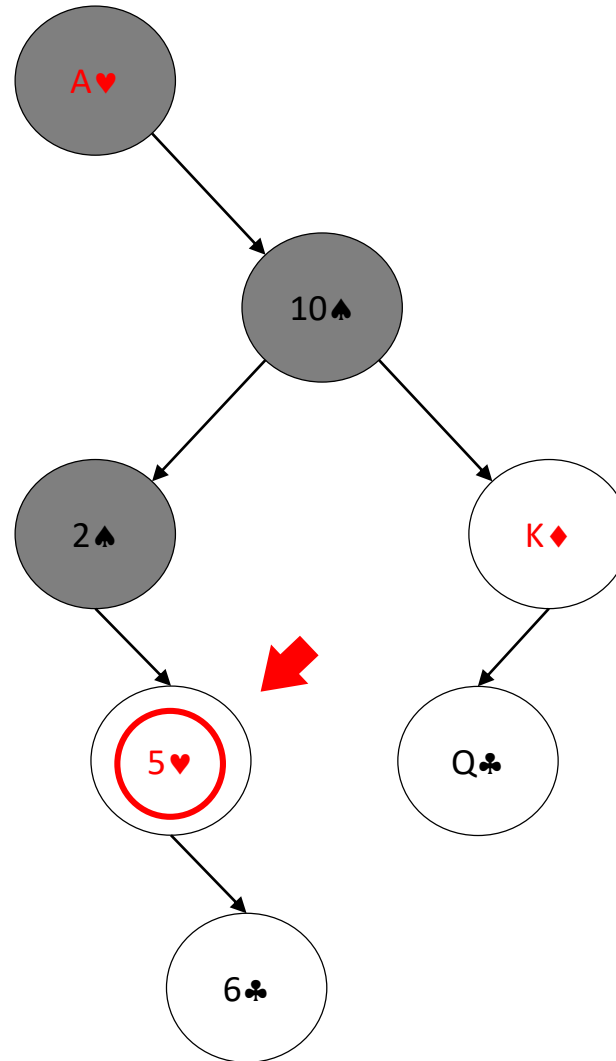
# Buscar a carta 5♥



# Buscar a carta 5♥



# Buscar a carta 5♥



# Complexidade da Árvore Binária de Busca

- Pode-se perceber que, no pior caso, a complexidade de busca vai ser igual ao número de elementos, ou seja,  $O(n)$ .
- Dessa forma, é necessário um mecanismo para garantir a construção da árvore de maneira a se aproximar ao máximo da complexidade da busca binária, ou seja,  $O(\log_2 n)$ .
- As árvores balanceadas são soluções para esse problema.

# Árvore Binária de Busca

- O algoritmo de **remoção** de elementos tem três casos de atuação.
  - Se o nó a ser removido não tem filhos: remove o nó e aponta a referência para o nó atual para NULL.
  - Se o nó a ser removido só tem filhos à direita **ou** à esquerda: remove o nó e aponta a referência para o nó atual para seu filho.
- Se o nó a ser removido tem dois filhos: encontra o nó M com o menor valor da subárvore da direita, aponta a esquerda do pai de M para a direita de M, modifica o valor do nó atual para o valor de M e remove o nó M.

# Árvore Binária de Busca

- Algoritmo de encontrar o menor e aponta para a direita dele:

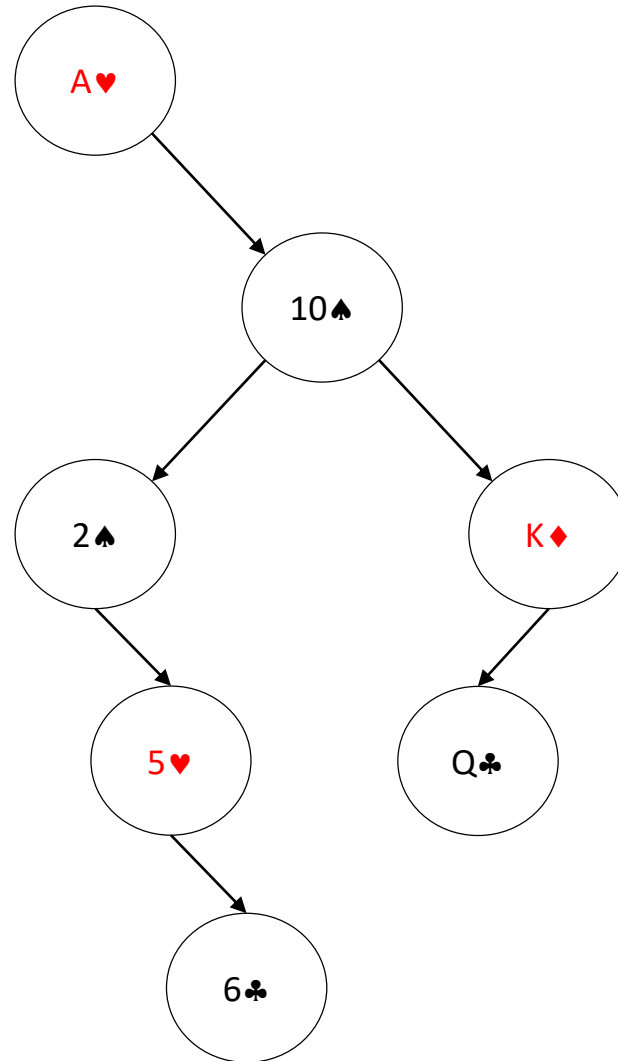
```
treenodeptr tPointSmaller(treenodeptr &p) {  
    treenodeptr t = p;  
    if (p->left == NULL){  
        p = p->right;  
        return t;  
    }  
    else  
        return tPointSmaller(p->left);  
}
```

- Algoritmo de remover um nó:

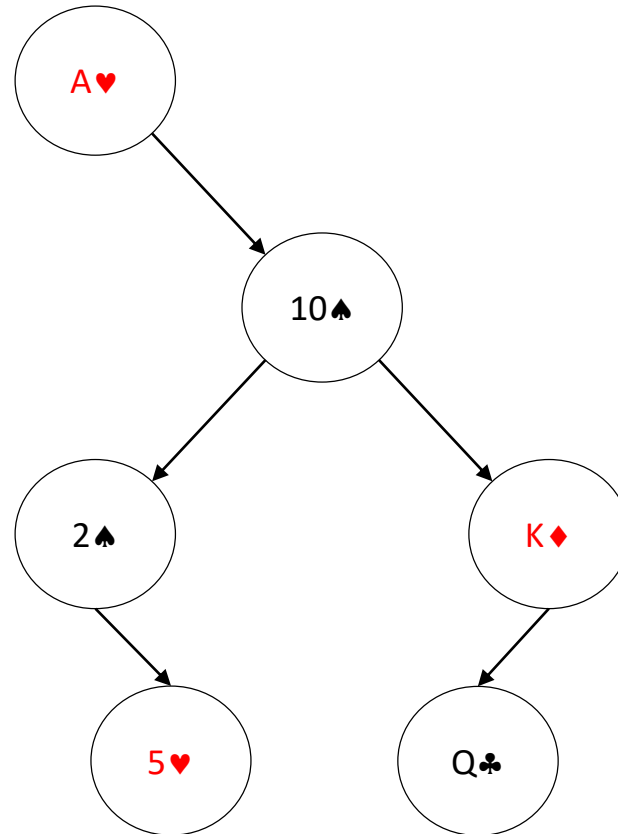
```
bool tRemove(treenodeptr &p, int x) {
    treenodeptr t;
    if (p == NULL) return false;
    if (x == p->info){
        t = p;
        if(p->left == NULL)
            p = p->right;
        else if(p->right == NULL)
            p = p->left;
        else {
            t = tPointSmaller(p->right);
            p->info = t->info;
        }
        delete t;
        return true;
    } else if (x < p->info) return tRemove(p->left,x);
    else return tRemove(p->right,x);
}
```



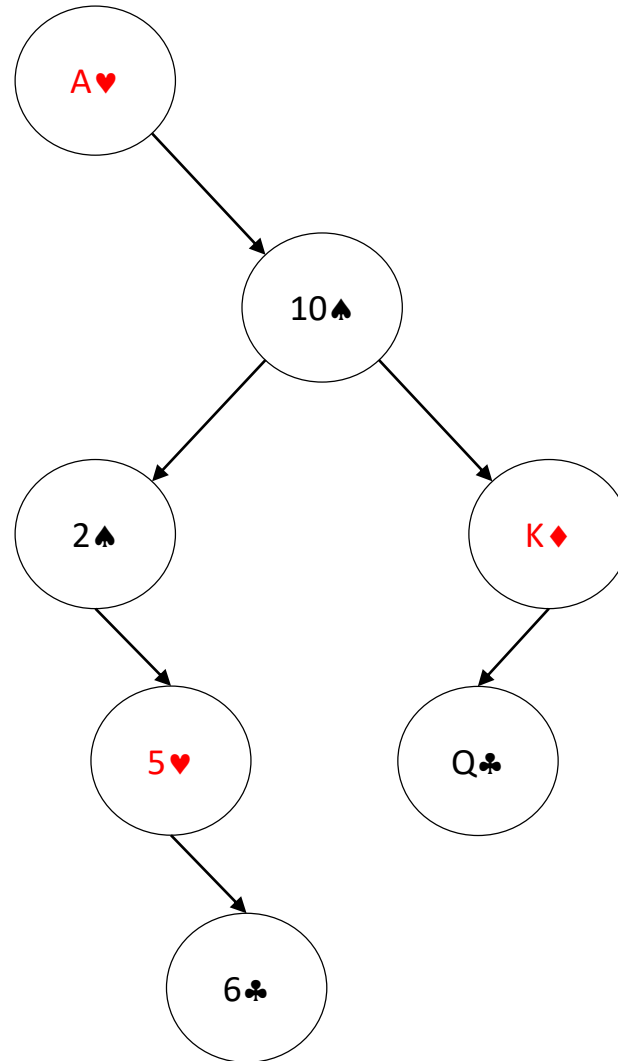
# Removendo a carta 6♣



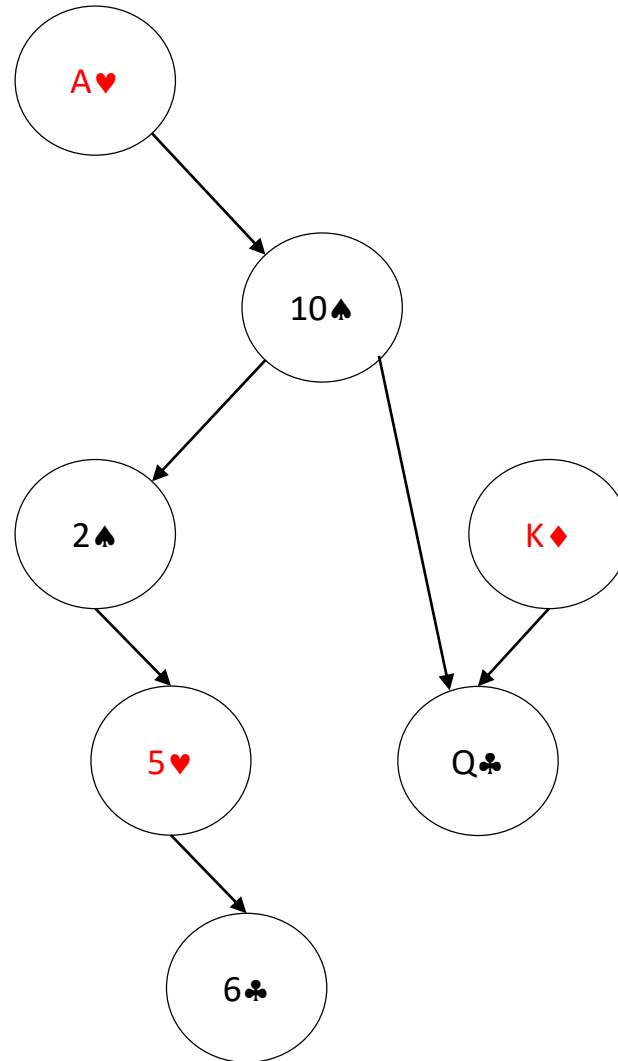
# Removendo a carta 6♣



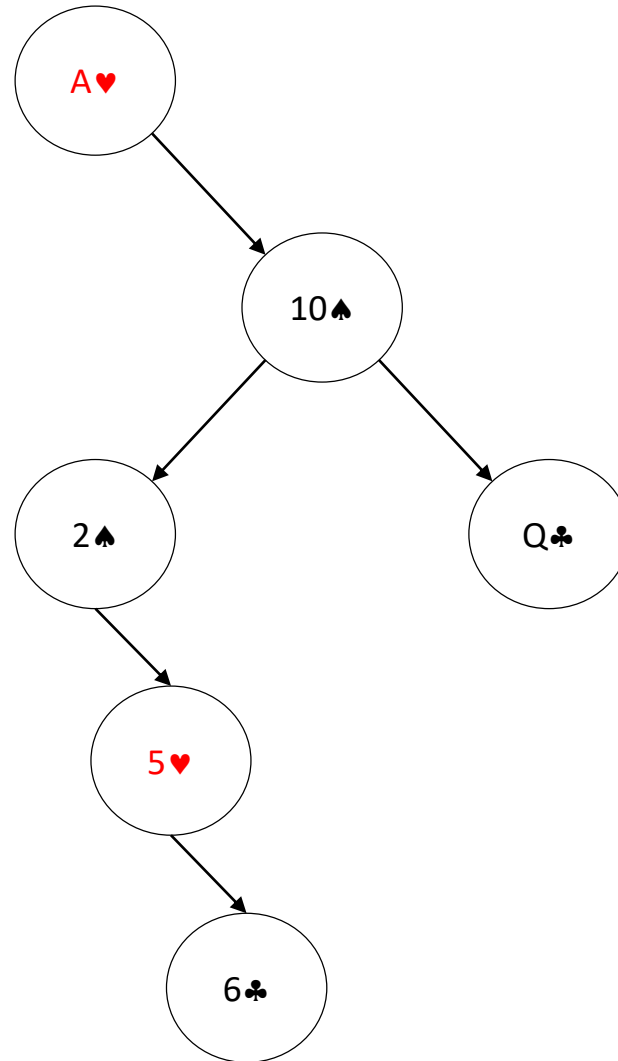
# Removendo a carta K♦



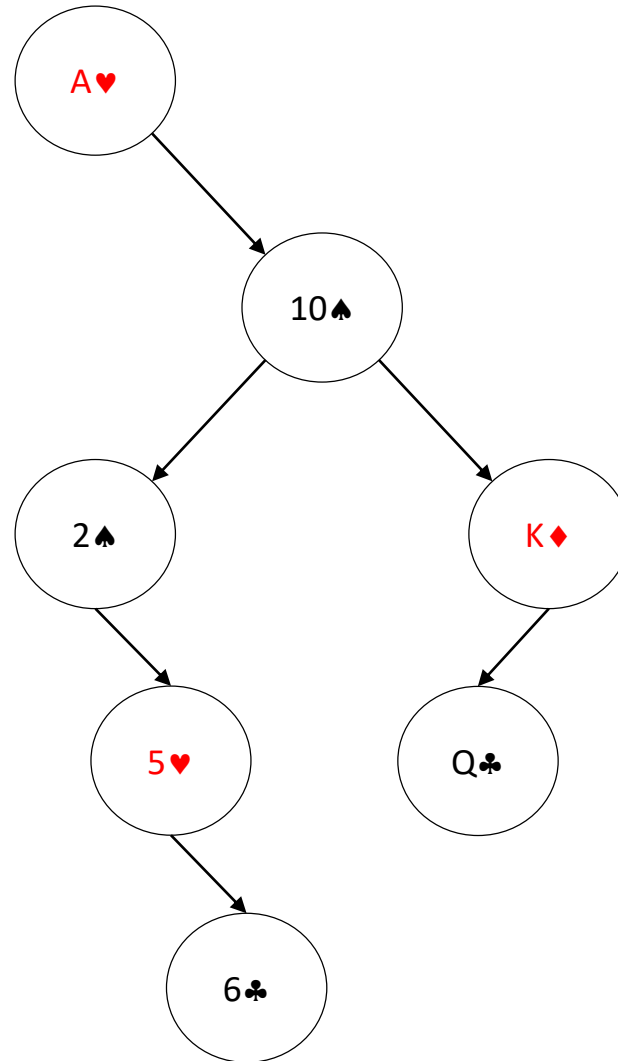
# Removendo a carta K♦



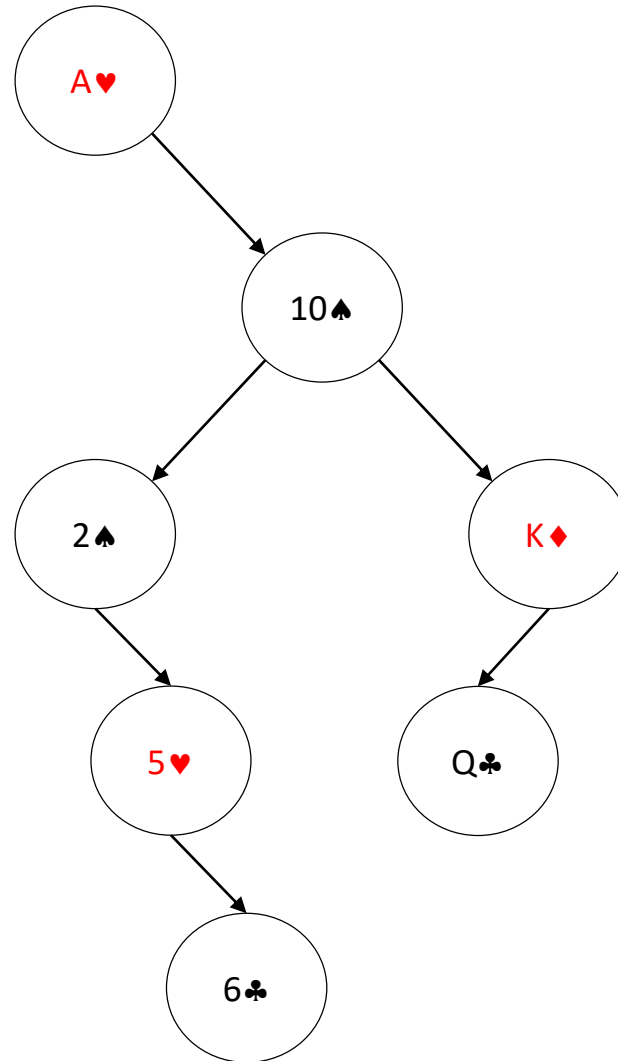
# Removendo a carta K♦



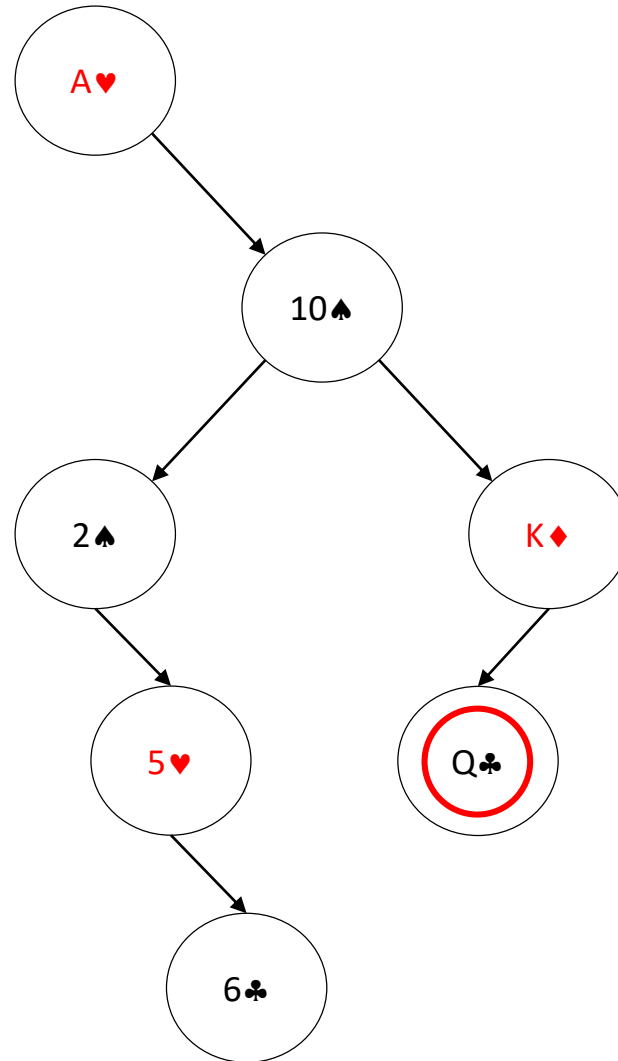
# Removendo a carta 10♠



# Removendo a carta 10♠

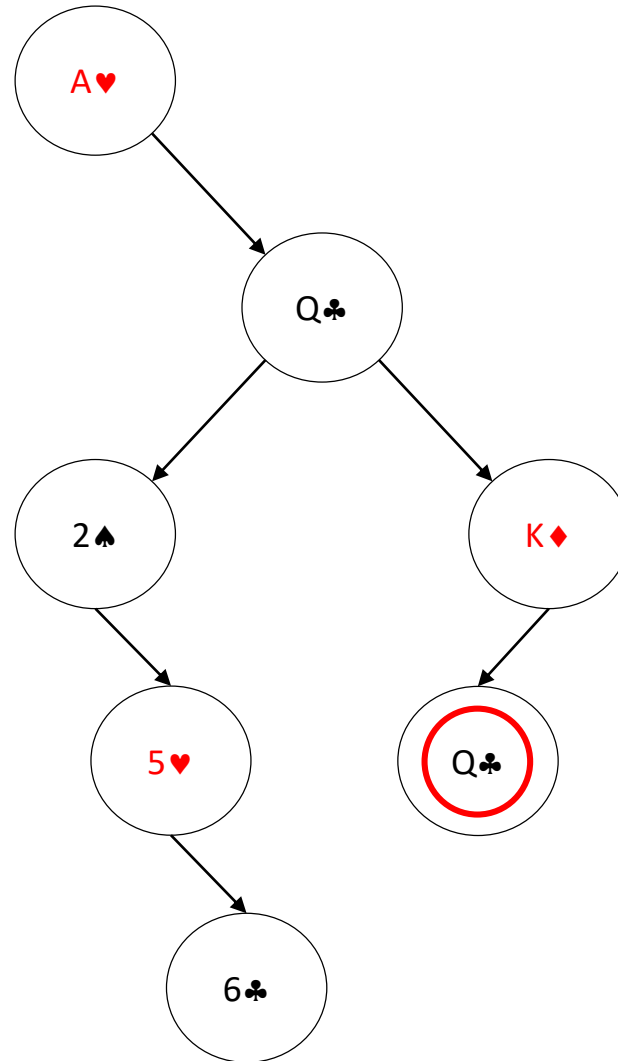


# Removendo a carta 10♠

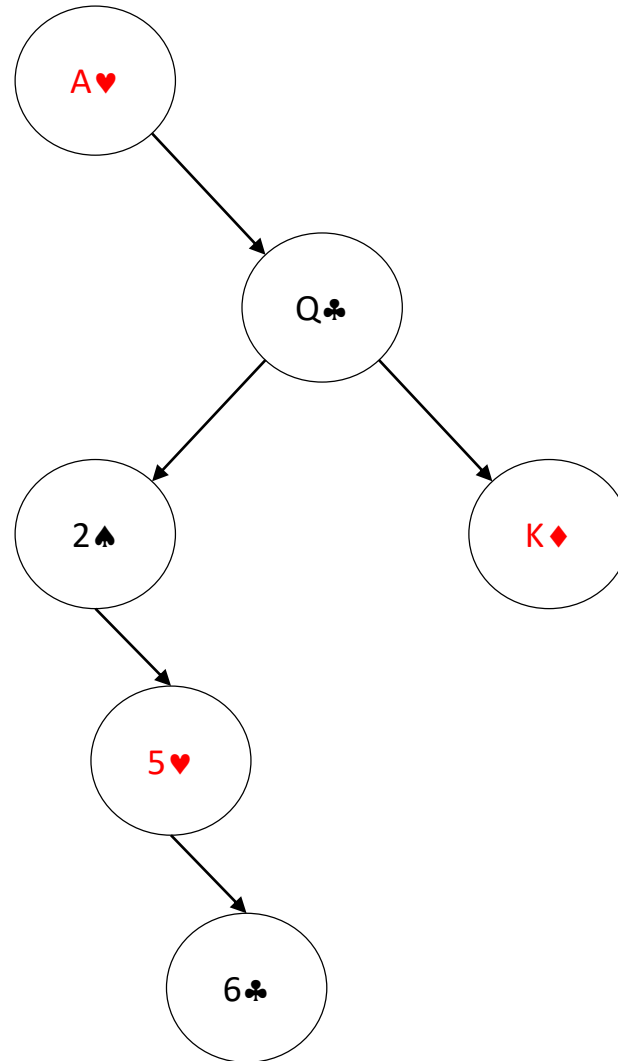




# Removendo a carta 10♠



# Removendo a carta 10♠



# Percursos

- O algoritmo de **varredura** em uma árvore binária de busca, assim como em um grafo qualquer, pode ser uma BFS ou uma DFS.
- Porém, como nessa estrutura de dados temos no máximo dois nós adjacentes (filho da direita e filho da esquerda) temos algumas particularidades em cada um desses algoritmos.

# Percurso em profundidade

- Nessa varredura, os nós descendentes de um determinado nó são visitados antes do próximo filho de um nó.
- Pode ocorrer de três formas:
  - Pré-ordem;
  - Em-ordem;
  - Pós-ordem.

# Percurso em profundidade em pré-ordem

```
void tPreOrder(treenodeptr p) {  
    if (p != NULL){  
        cout << p->info << endl;  
        tPreOrder(p->left);  
        tPreOrder(p->right);  
    }  
}
```

# Percurso em profundidade em-ordem

```
void tInOrder(treenodeptr p) {  
    if (p != NULL){  
        tInOrder(p->left);  
        cout << p->info << endl;  
        tInOrder(p->right);  
    }  
}
```

# Percurso em profundidade pós-ordem

```
void tPostOrder(treenodeptr p) {  
    if (p != NULL){  
        tPostOrder(p->left);  
        tPostOrder(p->right);  
        cout << p->info << endl;  
    }  
}
```

# Varredura em largura (em nível)

- Apresenta os nós da árvore por ordem de nível.

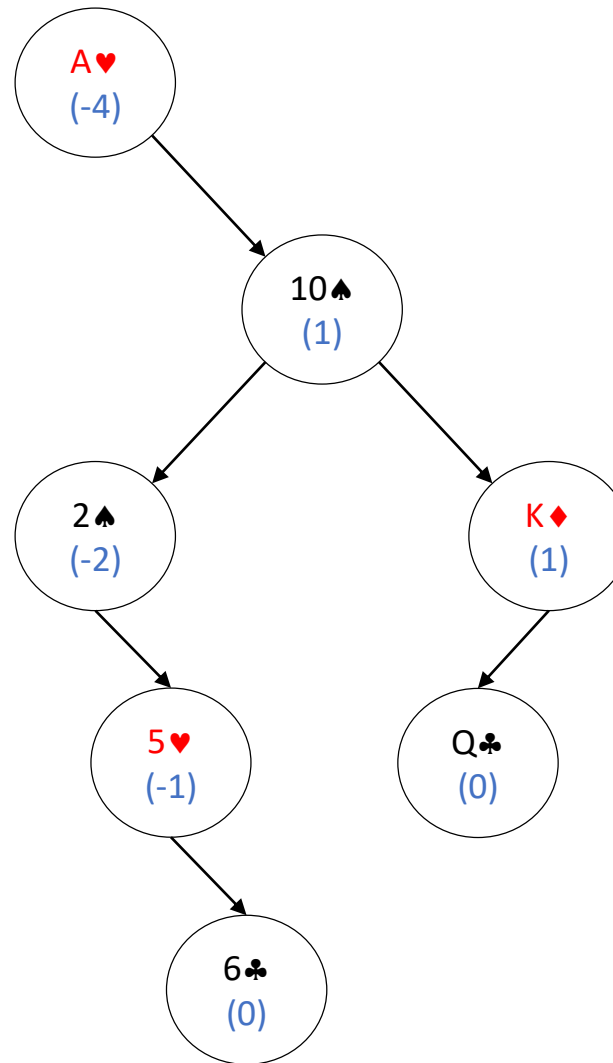
```
void tLevels(treenodeptr p) {  
    treenodeptr n;  
    list<treenodeptr> q;  
    if (p != NULL){  
        q.push_back(p);  
        while(!q.empty()){  
            n = q.front();  
            q.pop_front();  
            if (n->left != NULL)  
                q.push_back(n->left);  
            if (n->right != NULL)  
                q.push_back(n->right );  
            cout<<n->info<<endl;  
        }  
    }  
}
```



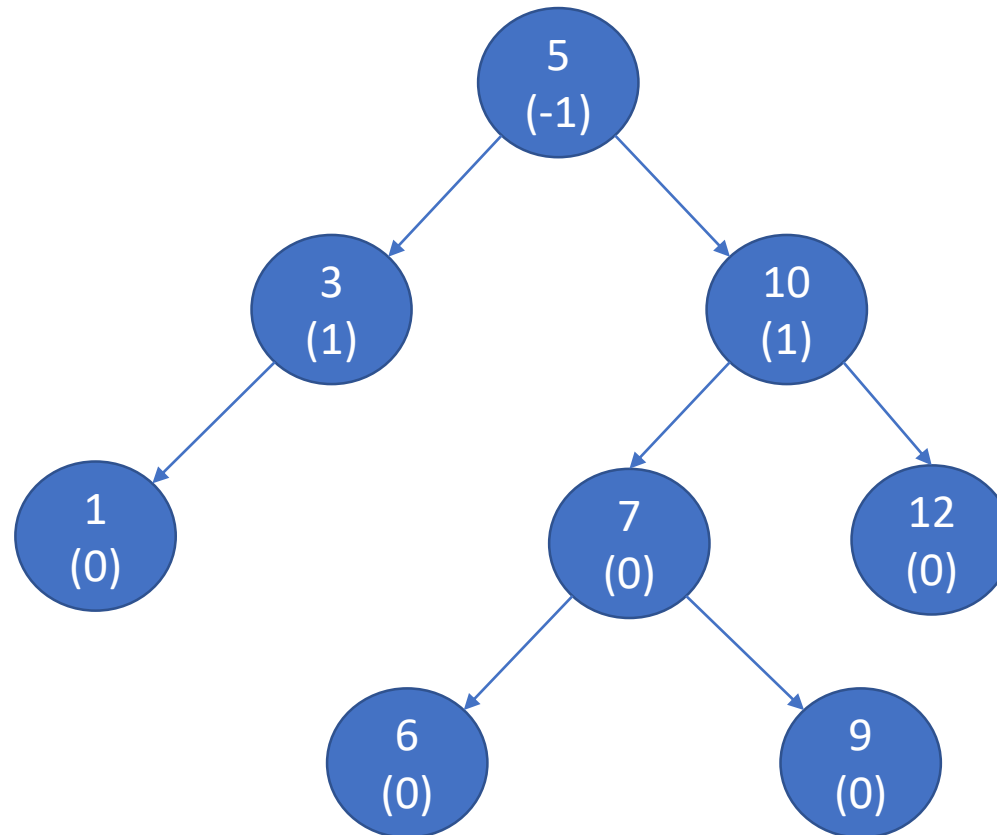
# Árvore AVL

- É um tipo de árvore binária na qual as alturas das duas subárvores de todos os nós nunca difere em mais de um.
- O balanceamento de um nó é definido como a diferença da altura de sua subárvore esquerda pela altura da sua subárvore direita.
- Dessa forma, os possíveis valores de balanceamento são -1, 0 ou 1.
- Uma árvore binária balanceada garante buscas mais eficientes.

# Exemplo de Árvore Binária desbalanceada



# Exemplo de Árvore AVL



# Rotação

- É um mecanismo para transformar a árvore binária para torna-la balanceada, sem mudar as características de navegação inicial.
- O ideal é realizar a rotação sempre que uma árvore se torna desbalanceada, ou seja, é inserido um nó a esquerda de um nó com balanceamento 1 ou é inserido