Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 17 –Árvore Heap (Definição e Implementação)

Agenda

- Motivação
- Funcionamento
- Inserção
- Remoção

- **PS:** Parte do conteúdo retirado do material do Prof. Flávio B. Gonzaga

Motivação

- A árvore Heap é uma árvore binária, **mas não uma árvore binária de busca**.
  - Portanto, não existe a garantia de que todos os elementos na subárvore esquerda serão menores que o nó raiz dessa subárvore.
  - Seguindo a mesma ideia, não existe garantia de que todos os elementos da subárvore direita serão maiores que o nó raiz dessa árvore.

Motivação

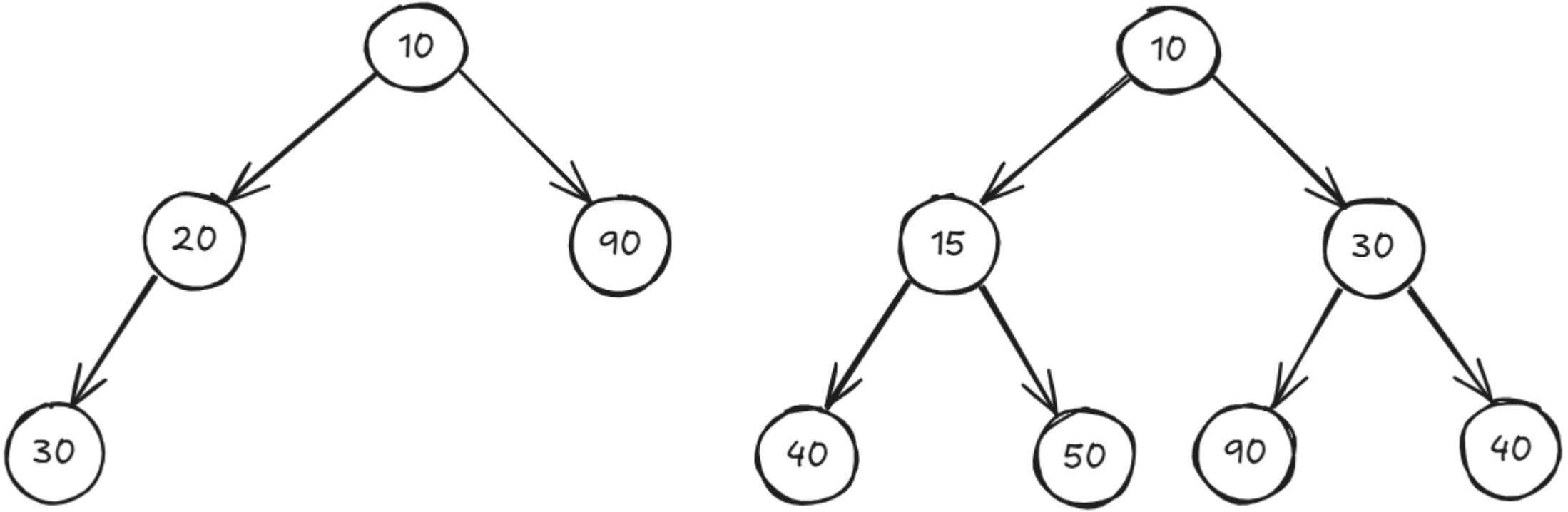
- A característica garantida pela árvore Heap no entanto é que o menor (ou o maior) elemento da árvore estará sempre na raiz da mesma;

- A propriedade acima também se aplica recursivamente aos nós internos da árvore.

Motivação

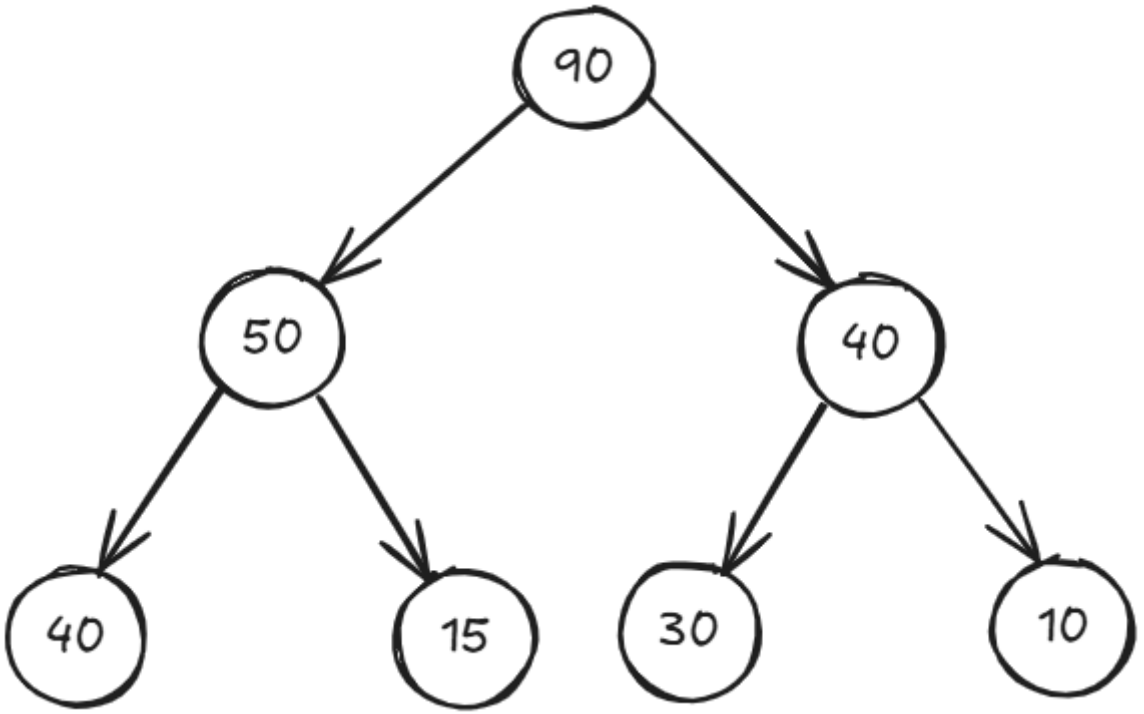
- A árvore Heap será portanto uma **Min Heap** ou uma **Max Heap**, de acordo com a natureza do valor oferecido na raiz (mínimo ou máximo dentre os valores presentes na árvore).
  - Ou seja, ela será uma **Min Heap** se a raiz for o menor valor da árvore ou;
  - Ela será uma **Max Heap** se a raiz for o maior valor da árvore.

Motivação



Exemplos de Min Heap

Motivação



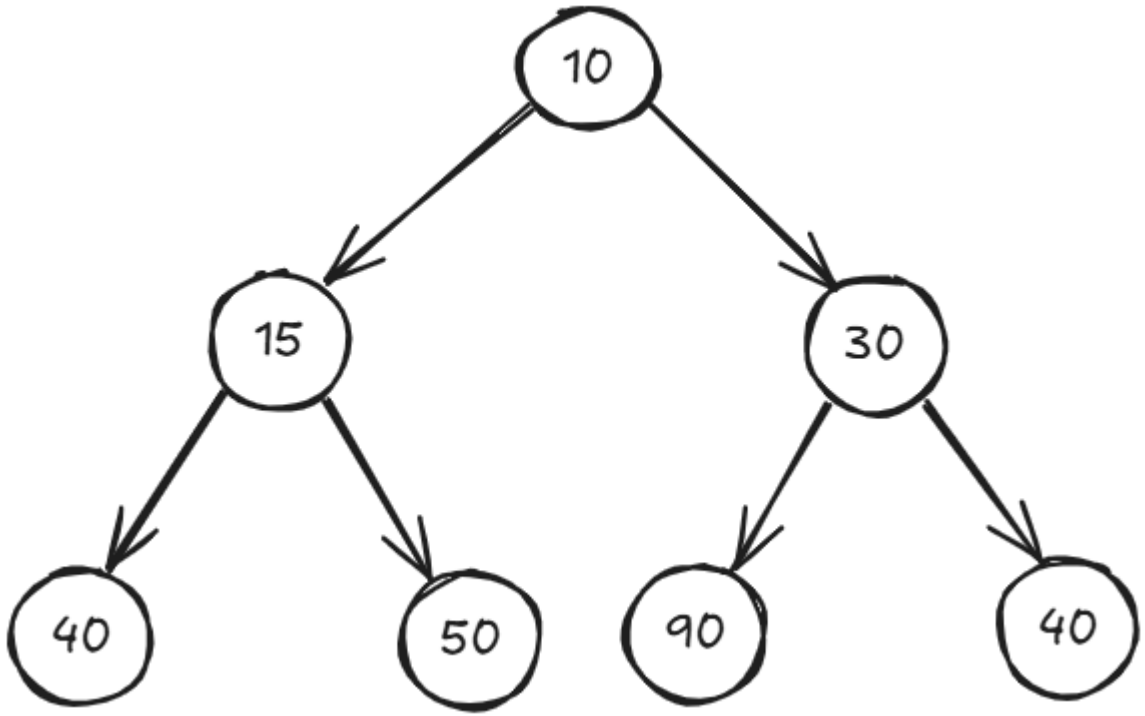
Exemplos de Max Heap

Funcionamento

- Geralmente, a árvore heap é implementada como um vetor.

- Ela é uma árvore completa ou seja, possui todos os níveis preenchidos, exceto possivelmente o último nível (que terá os nós sempre o mais à esquerda possível).

Funcionamento



Para o i-ésimo nó:

vet[(i-1)/2] retornará o nó pai
*vet[(2*i)+1]* retornará o filho a esquerda
*vet[(2*i)+2]* retornará o filho a direita

	0	1	2	3	4	5	6
vet	10	15	30	40	50	90	40

Funcionamento

- As inserções são sempre feitas no final da árvore, com complexidade de tempo de *O(log n)*.

- Se o novo nó for menor do que seu pai, troca-se a posição de um com o outro.
  - Essa operação é repetida subindo-se na árvore, até que o nó seja maior do que seu pai, ou se torne a raiz.
  - Essa descrição é valida para a Min Heap. A Max Heap é o contrário (maior valor).

Funcionamento

- As remoções são sempre feitas na raiz, que possui o menor ou o maior valor, dependendo do tipo da árvore. A complexidade de tempo também é *O(log n)*.
  - O último elemento inserido na árvore assume o lugar na raiz temporariamente.
  - Se a nova raiz for maior do que algum dos seus filhos, troca-se de lugar com o menor deles.
  - Realiza-se essa operação recursivamente até que o nó se torne menor ou igual do que ambos os filhos, ou até que o mesmo se torne um nó folha.

Funcionamento

- A definição da árvore heap utiliza as bibliotecas stdio.h e limits.h (a última é usada para pegarmos o menor valor de um inteiro). Definimos também o tamanho da (*MAX*) como 100.

- A estrutura possui o array *arr* com o tamanho definido anteriormente, o *tamanho* (inicialmente zero) e a *capacidade* (tamanho máximo do vetor, inicialmente *MAX*).

```c
#include <stdio.h>
#include <limits.h>

#define MAX 100

typedef struct {
    int arr[MAX];
    int tam;
    int capacidade;
} MinHeap;

int pai(int i) {
    return (i - 1) / 2;
}

int esquerda(int i) {
    return 2 * i + 1;
}

int direita(int i) {
    return 2 * i + 2;
}

void troca(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Funcionamento

- A função pai, esquerda e direita funcionam exatamente como definido em slides passados:

Para o i-ésimo nó:

vet[(i-1)/2] retornará o nó pai
*vet[(2*i)+1]* retornará o filho a esquerda
*vet[(2*i)+2]* retornará o filho a direita

- A função *troca* realiza a troca de posição de dois elementos dentro do vetor que representa a árvore heap.

```c
#include <stdio.h>
#include <limits.h>

#define MAX 100

typedef struct {
    int arr[MAX];
    int tam;
    int capacidade;
} MinHeap;

int pai(int i) {
    return (i - 1) / 2;
}

int esquerda(int i) {
    return 2 * i + 1;
}

int direita(int i) {
    return 2 * i + 2;
}

void troca(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Inserção

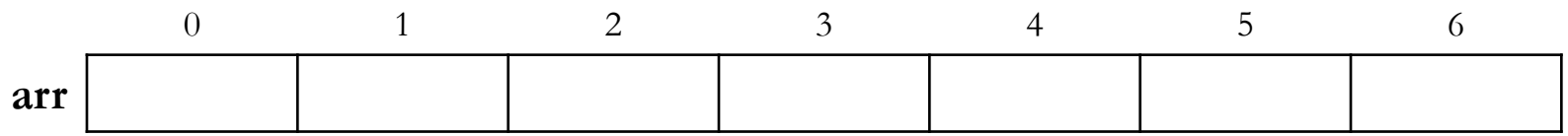
- Para entender como funciona a inserção em uma árvore heap, iremos inserir os seguintes valores em ordem:

5 – 2 – 4 – 15 – 7 – 3 – 1

- Para esta inserção, utilizaremos a estrutura de uma **Min Heap**.

Inserção

5 – 2 – 4 – 15 – 7 – 3 – 1



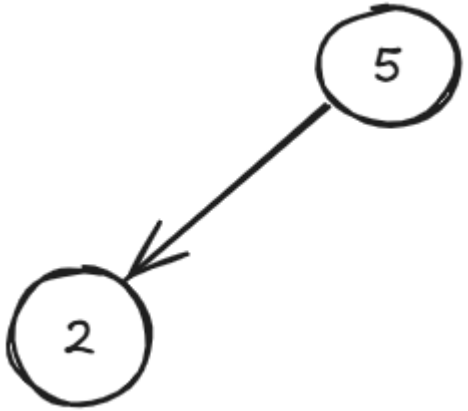
Inserção

~~5~~ – 2 – 4 – 15 – 7 – 3 – 1

	0	1	2	3	4	5	6
arr	5						

Inserção

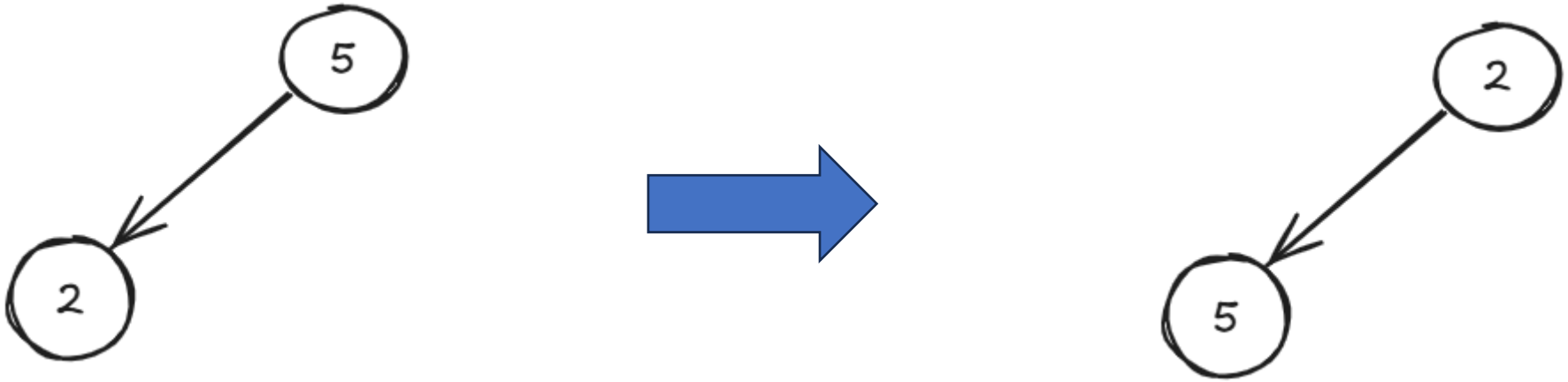
~~5~~ – ~~2~~ – 4 – 15 – 7 – 3 – 1



	0	1	2	3	4	5	6
arr	5	2					

Inserção

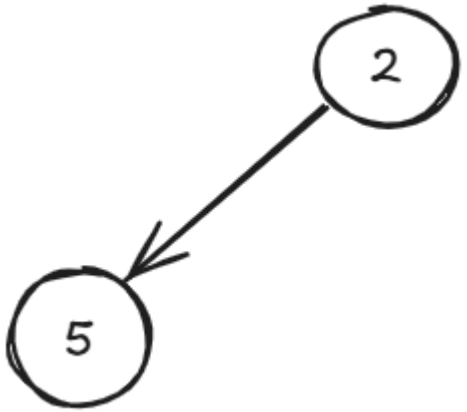
~~5~~ – ~~2~~ – 4 – 15 – 7 – 3 – 1



	0	1	2	3	4	5	6
arr	2	5					

Inserção

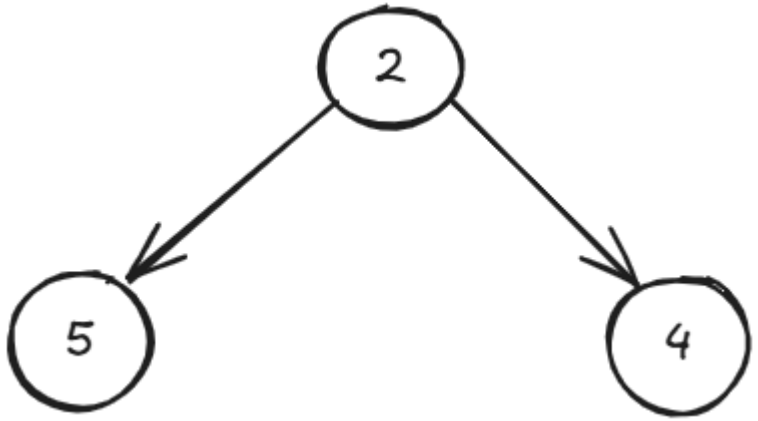
~~5~~ – ~~2~~ – 4 – 15 – 7 – 3 – 1



	0	1	2	3	4	5	6
arr	2	5					

Inserção

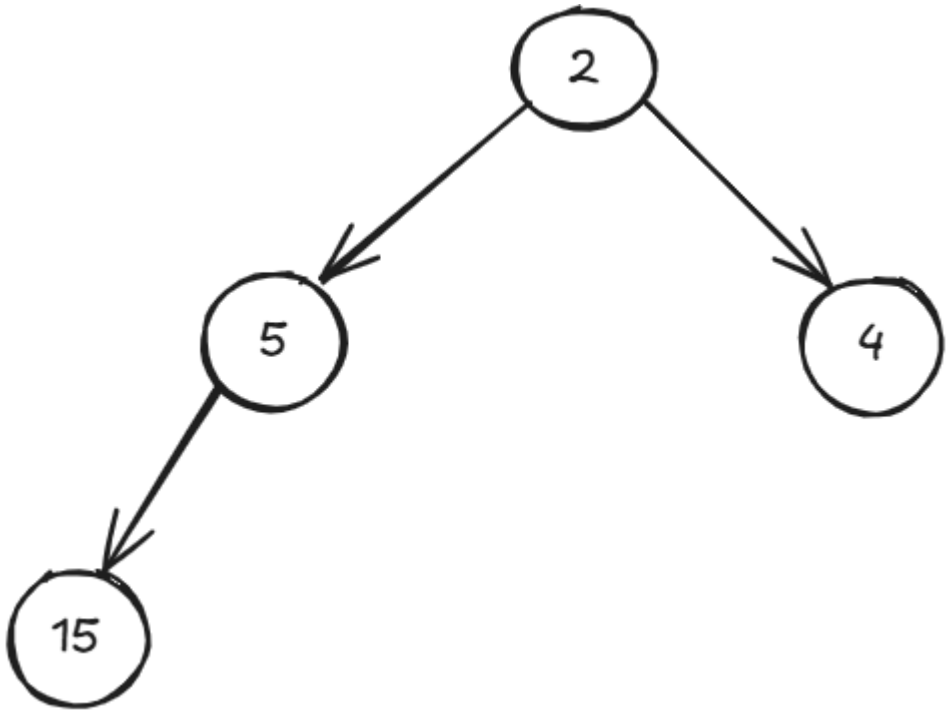
~~5~~ – ~~2~~ – 4 – 15 – 7 – 3 – 1



	0	1	2	3	4	5	6
arr	2	5	4				

Inserção

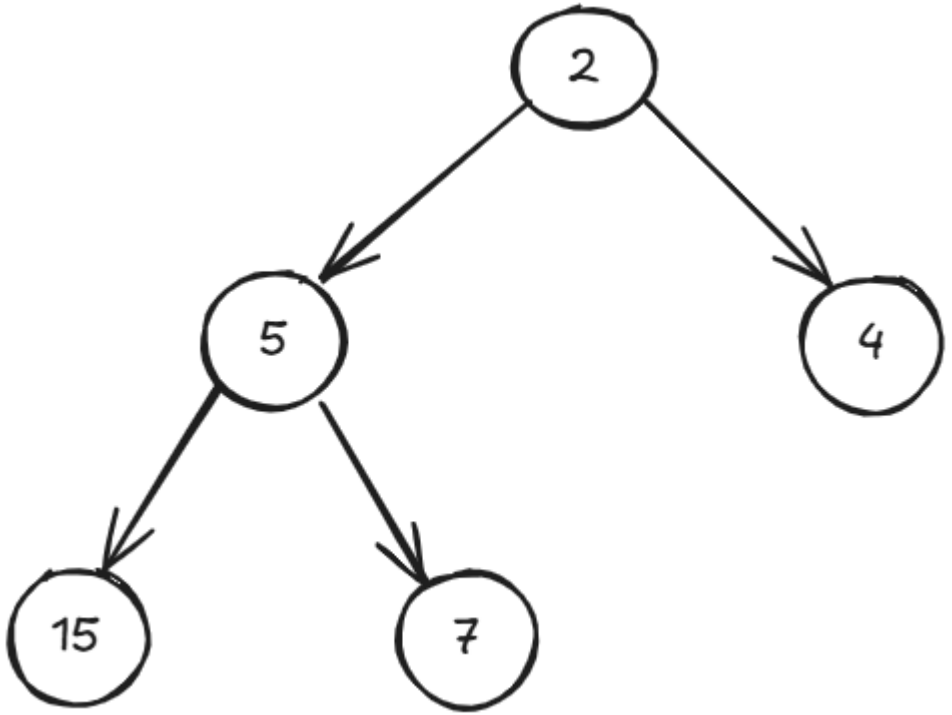
~~5~~ – ~~2~~ – 4 – ~~15~~ – 7 – 3 – 1



	0	1	2	3	4	5	6
arr	2	5	4	15			

Inserção

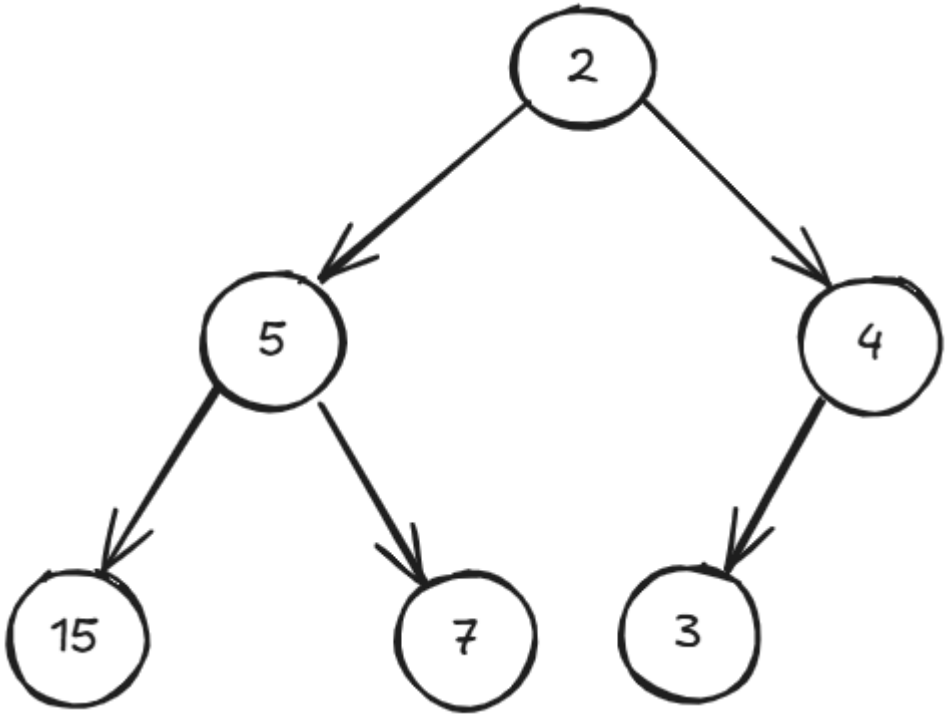
~~5~~ – ~~2~~ – 4 – ~~15~~ – ~~7~~ – 3 – 1



	0	1	2	3	4	5	6
arr	2	5	4	15	7		

Inserção

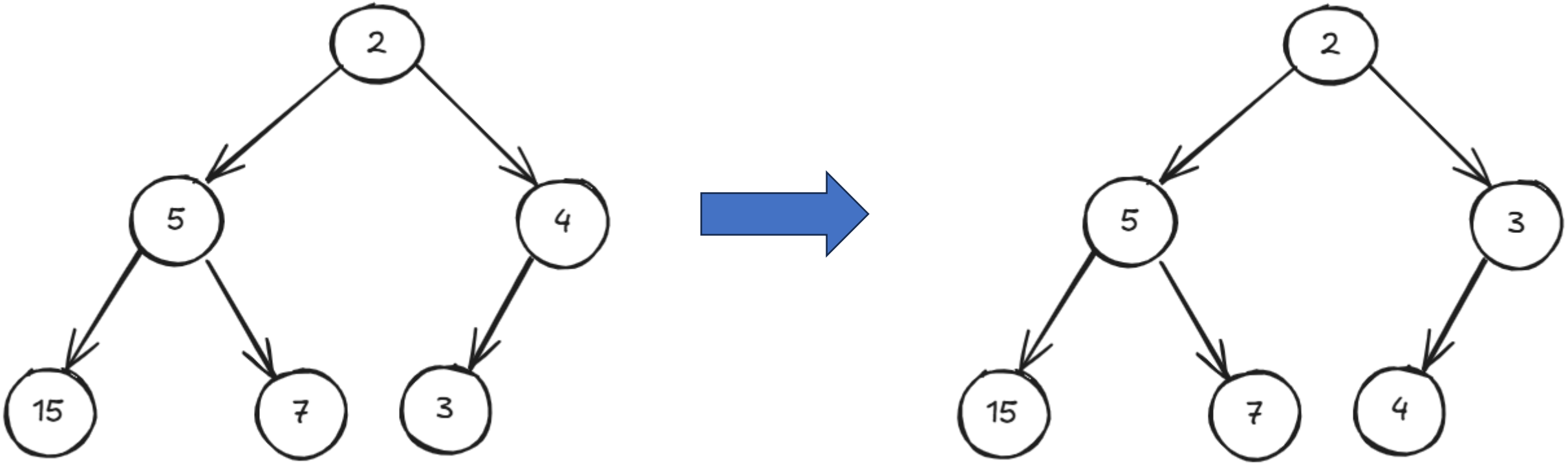
~~5~~ – ~~2~~ – 4 – ~~15~~ – ~~7~~ – ~~3~~ – 1



	0	1	2	3	4	5	6
arr	2	5	4	15	7	3	

Inserção

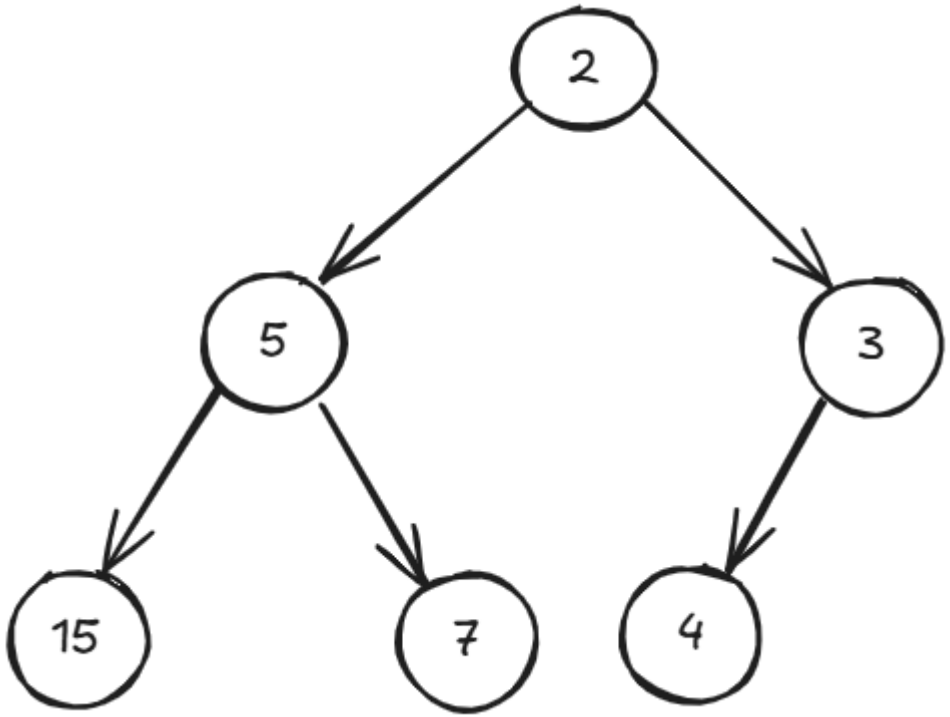
~~5~~ – ~~2~~ – ~~4~~ – ~~15~~ – ~~7~~ – ~~3~~ – 1



	0	1	2	3	4	5	6
arr	2	5	3	15	7	4	

Inserção

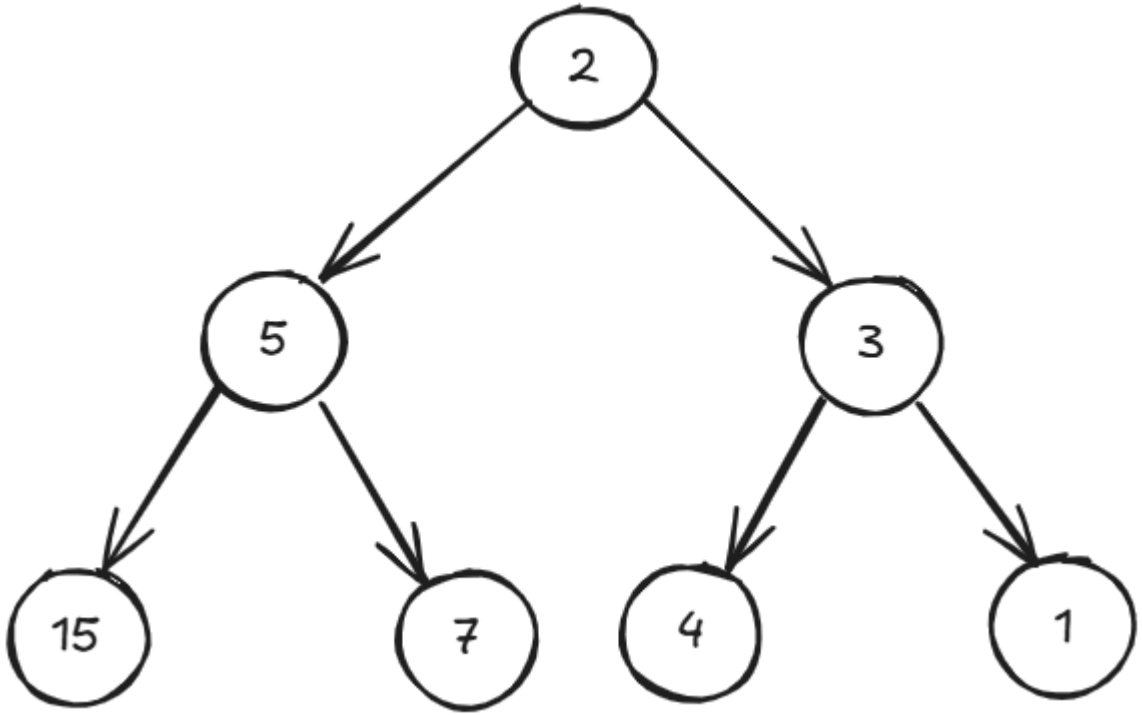
~~5~~ – ~~2~~ – 4 – ~~15~~ – ~~7~~ – ~~3~~ – 1



	0	1	2	3	4	5	6
arr	2	5	3	15	7	4	

Inserção

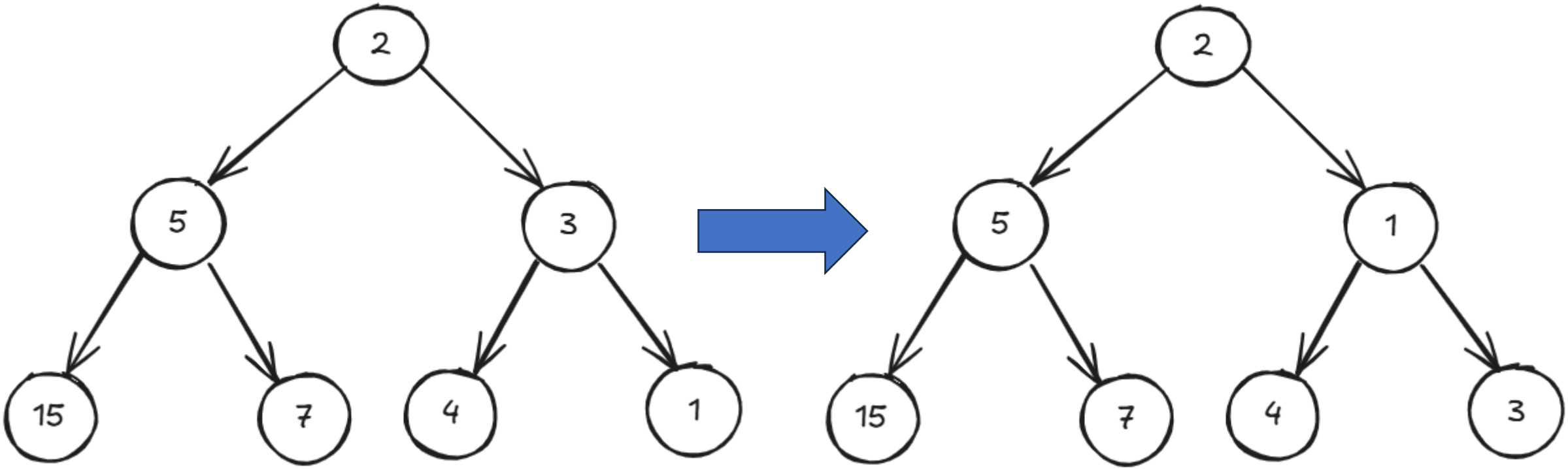
~~5~~ – ~~2~~ – 4 – ~~15~~ – ~~7~~ – ~~3~~ – ~~1~~



	0	1	2	3	4	5	6
arr	2	5	3	15	7	4	1

Inserção

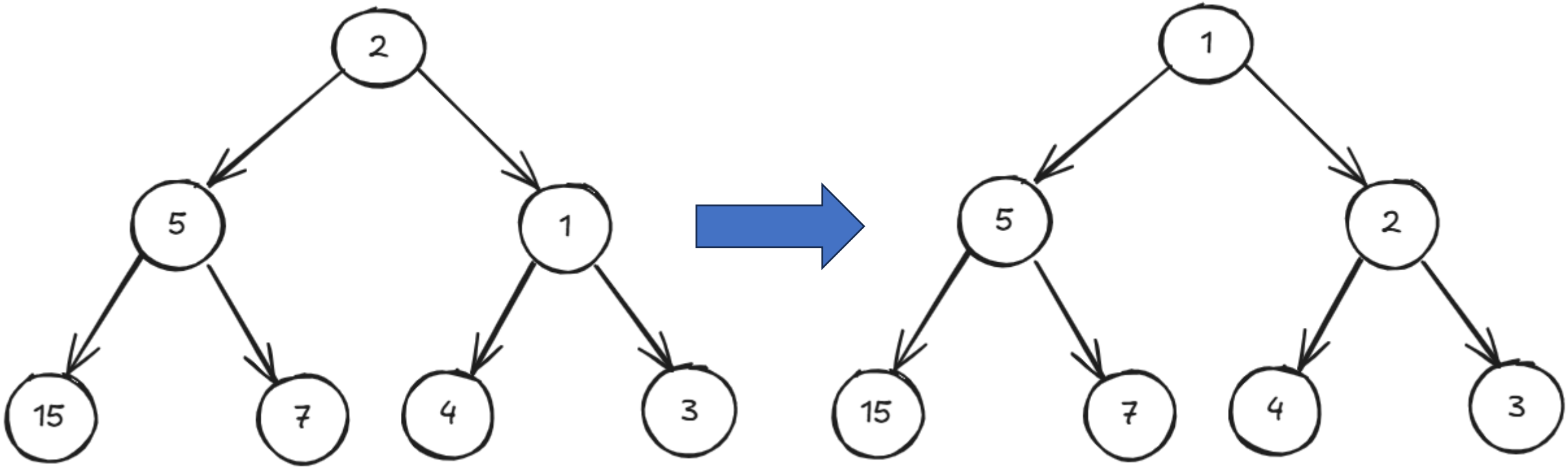
~~5~~ – ~~2~~ – ~~4~~ – ~~15~~ – ~~7~~ – ~~3~~ – ~~1~~



	0	1	2	3	4	5	6
arr	2	5	1	15	7	4	3

Inserção

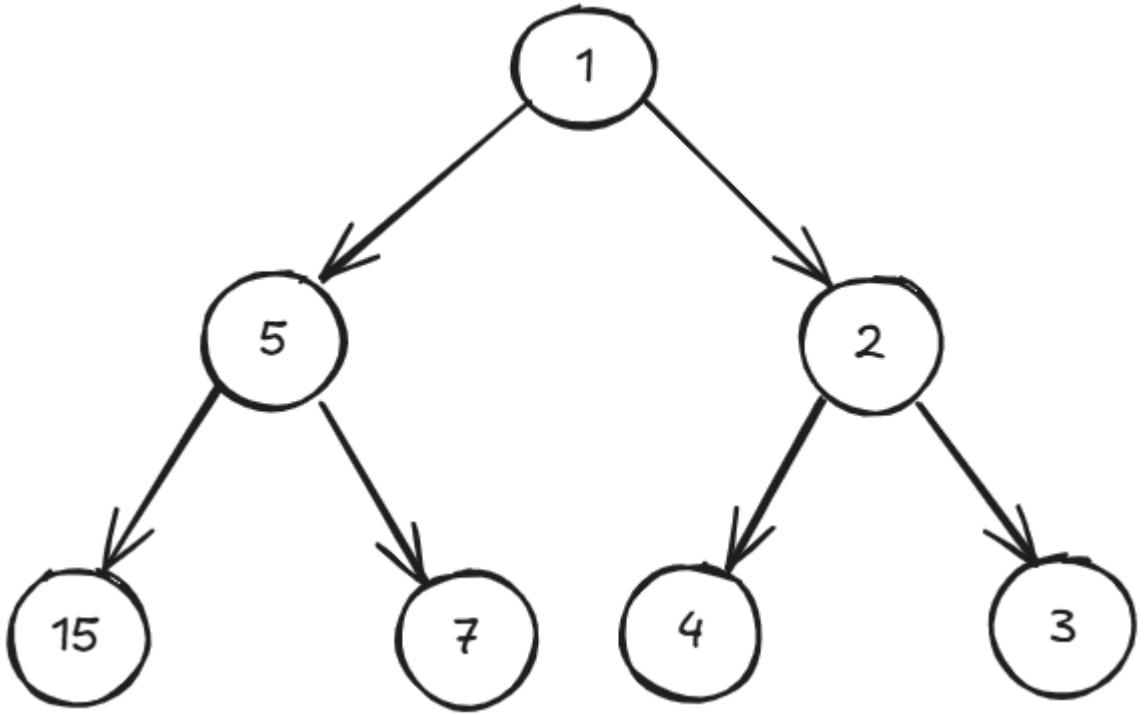
~~5~~ – ~~2~~ – ~~4~~ – ~~15~~ – ~~7~~ – ~~3~~ – ~~1~~



	0	1	2	3	4	5	6
arr	1	5	2	15	7	4	3

Inserção

~~5~~ – ~~2~~ – ~~4~~ – ~~15~~ – ~~7~~ – ~~3~~ – ~~1~~



	0	1	2	3	4	5	6
arr	1	5	2	15	7	4	3

Inserção

- A função *insere* insere um novo valor *k* na.
- Primeiramente, ela verifica se a heap atingiu sua capacidade máxima, caso em que imprime uma mensagem de erro e retorna sem realizar a inserção.
- Se houver espaço, a função aumenta o tamanho da heap, coloca o valor k na última posição do vetor e inicia um processo de "subida" para manter a propriedade da Min Heap.

```c
void insere(MinHeap* heap, int k) {
    if (heap->tam == heap->capacidade) {
        printf("\nOverflow: Nao eh possivel inserir\n");
        return;
    }

    heap->tam++;
    int i = heap->tam - 1;
    heap->arr[i] = k;

    while (i != 0 && heap->arr[pai(i)] > heap->arr[i]) {
        troca(&heap->arr[i], &heap->arr[pai(i)]);
        i = pai(i);
    }
}
```

Inserção

- Ela compara o valor inserido com seu pai (usando a função pai(i)), e enquanto o valor do nó for menor que o valor do seu pai, os dois valores são trocados (usando a função troca).
- Esse processo é repetido até que o valor do nó esteja na posição correta, garantindo que a Min Heap continue válida.

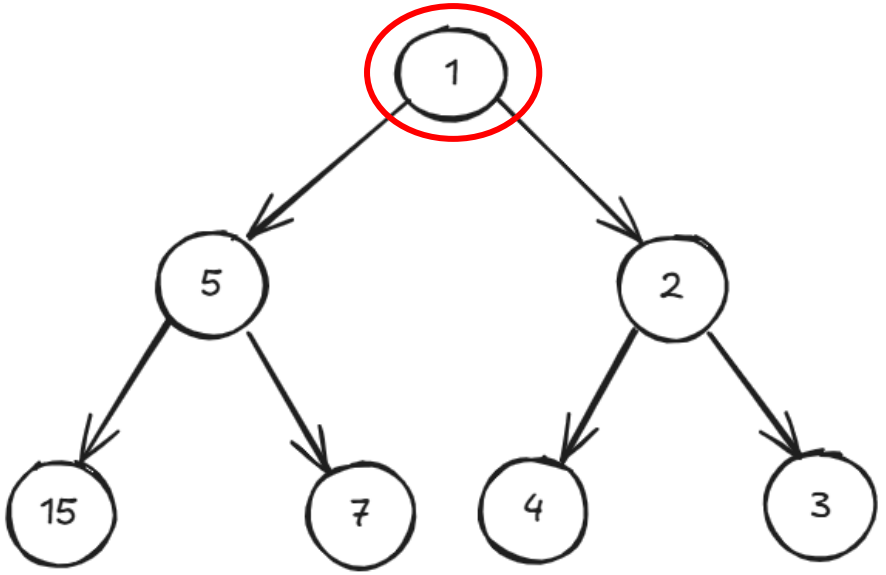
```c
void insere(MinHeap* heap, int k) {
    if (heap->tam == heap->capacidade) {
        printf("\nOverflow: Nao eh possivel inserir\n");
        return;
    }

    heap->tam++;
    int i = heap->tam - 1;
    heap->arr[i] = k;

    while (i != 0 && heap->arr[pai(i)] > heap->arr[i]) {
        troca(&heap->arr[i], &heap->arr[pai(i)]);
        i = pai(i);
    }
}
```

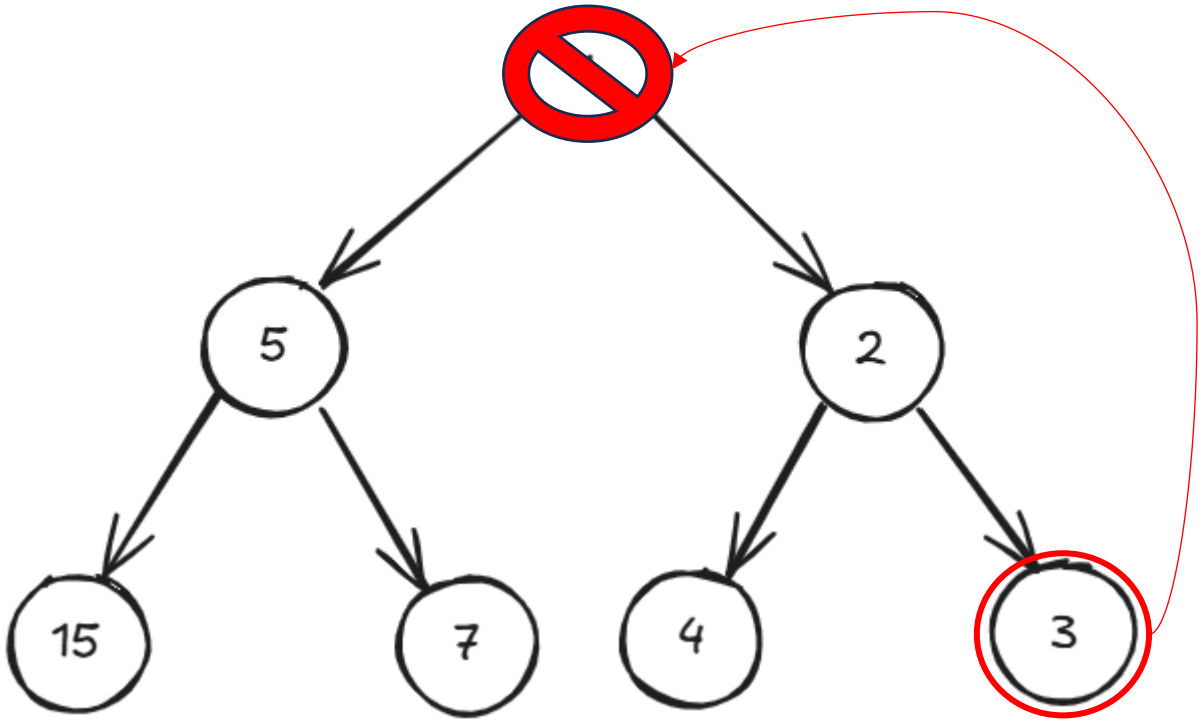
Remoção

- Para entender como funciona a remoção em uma árvore heap, iremos remover os valores 1 e 2 (em sequência) da árvore criada anteriormente.
  - Para esta remoção, utilizaremos a estrutura de uma **Min Heap**.



Remoção

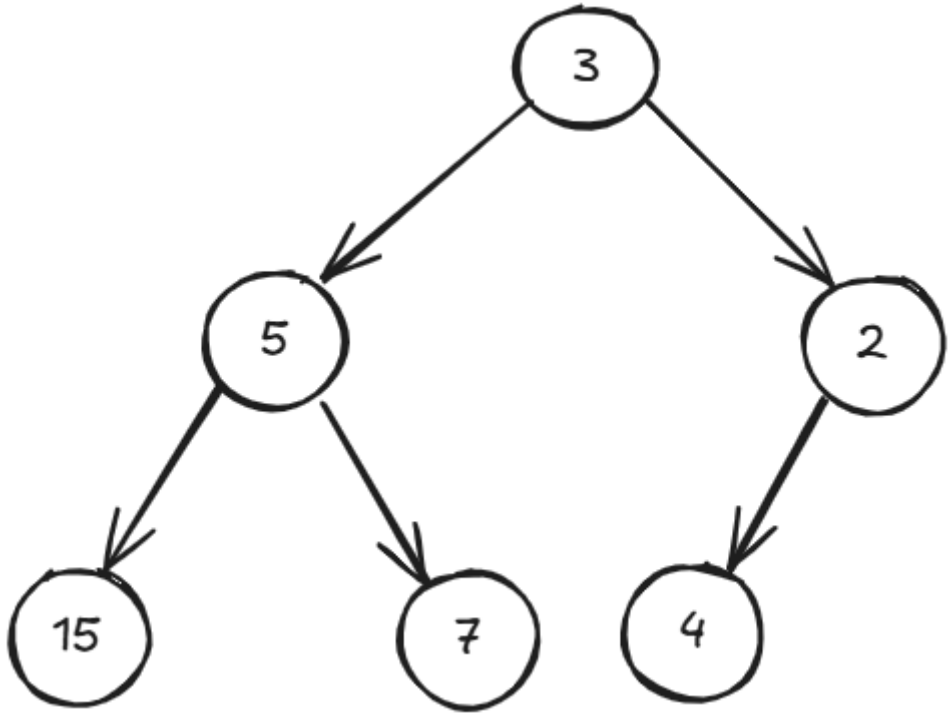
Remoção dos valores 1 e 2



	0	1	2	3	4	5	6
arr	1	5	2	15	7	4	3

Remoção

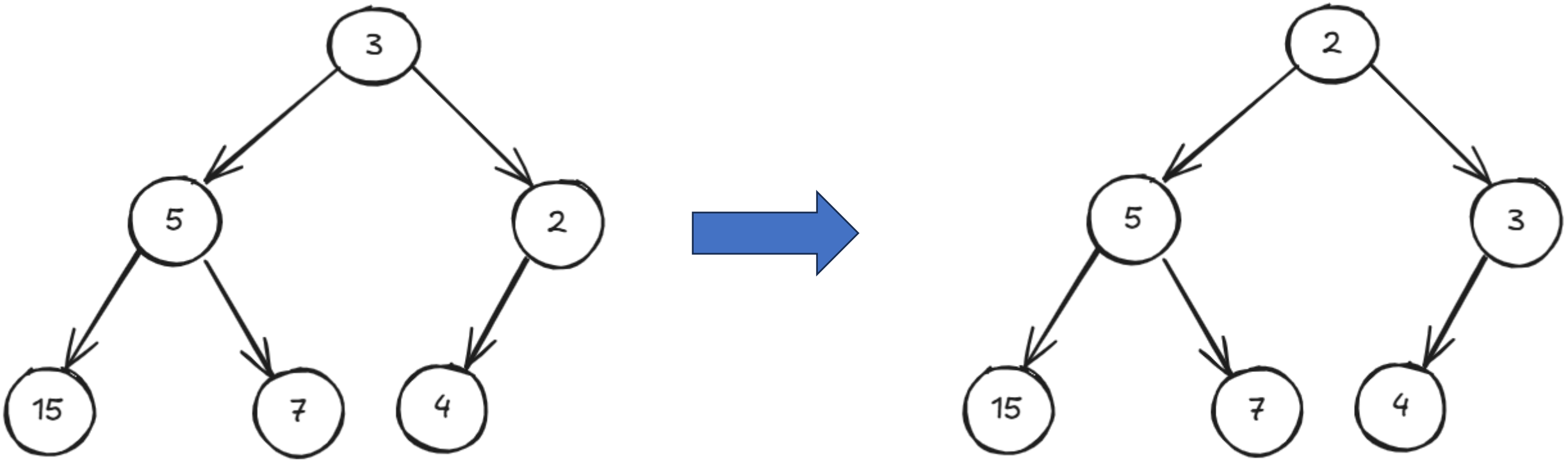
Remoção dos valores 1 e 2



	0	1	2	3	4	5	6
arr	3	5	2	15	7	4	

Remoção

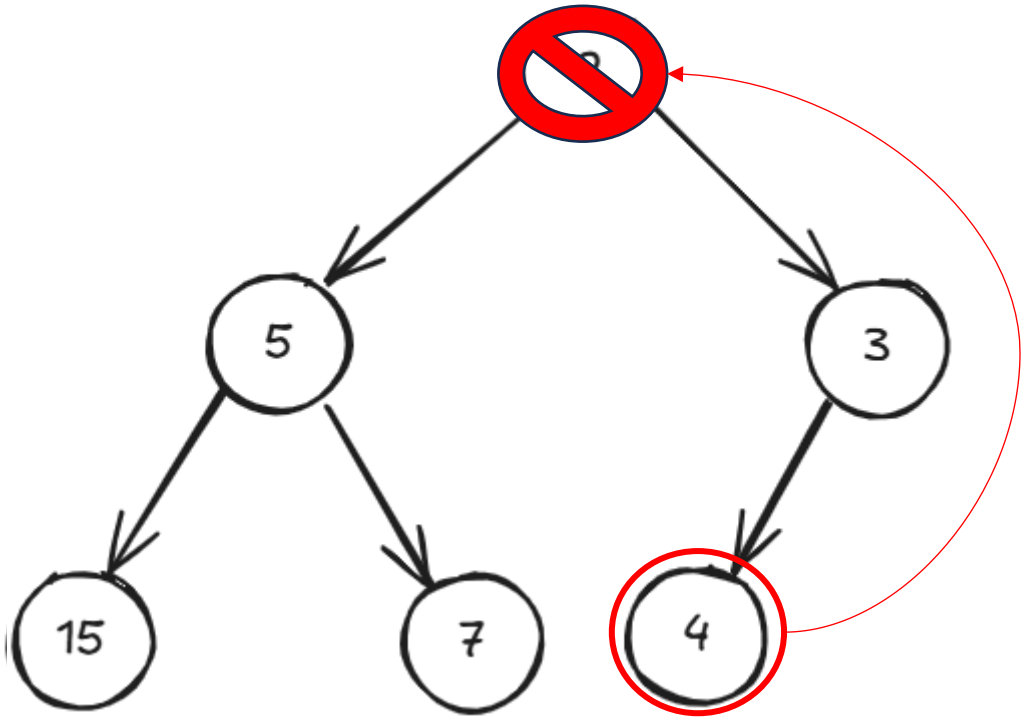
Remoção dos valores 1 e 2



	0	1	2	3	4	5	6
arr	2	5	3	15	7	4	

Remoção

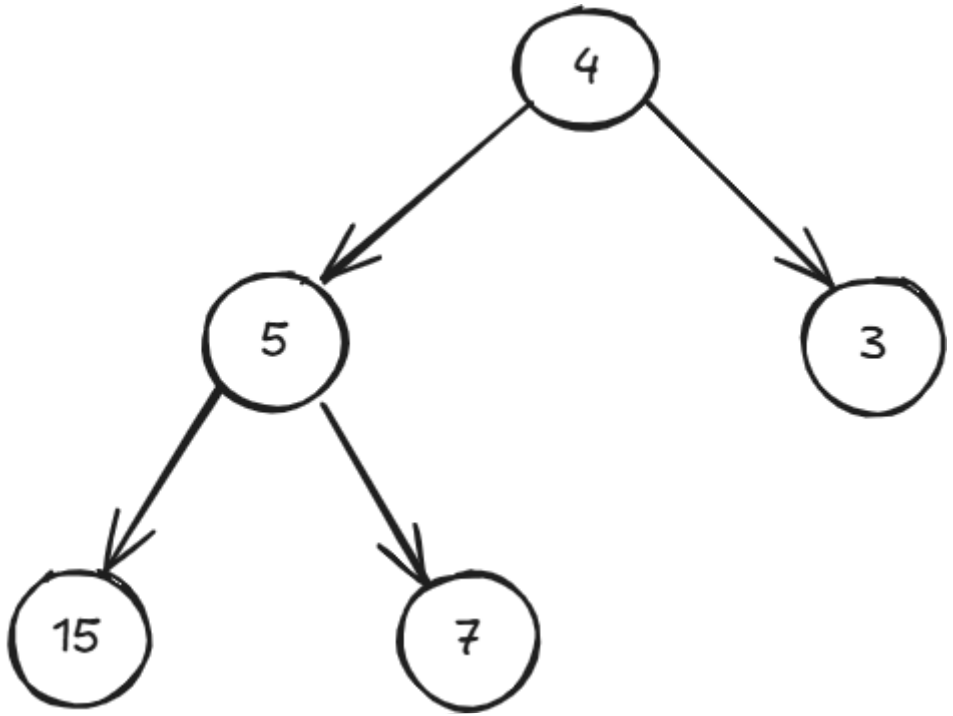
Remoção dos valores 1 e 2



	0	1	2	3	4	5	6
arr	2	5	3	15	7	4	

Remoção

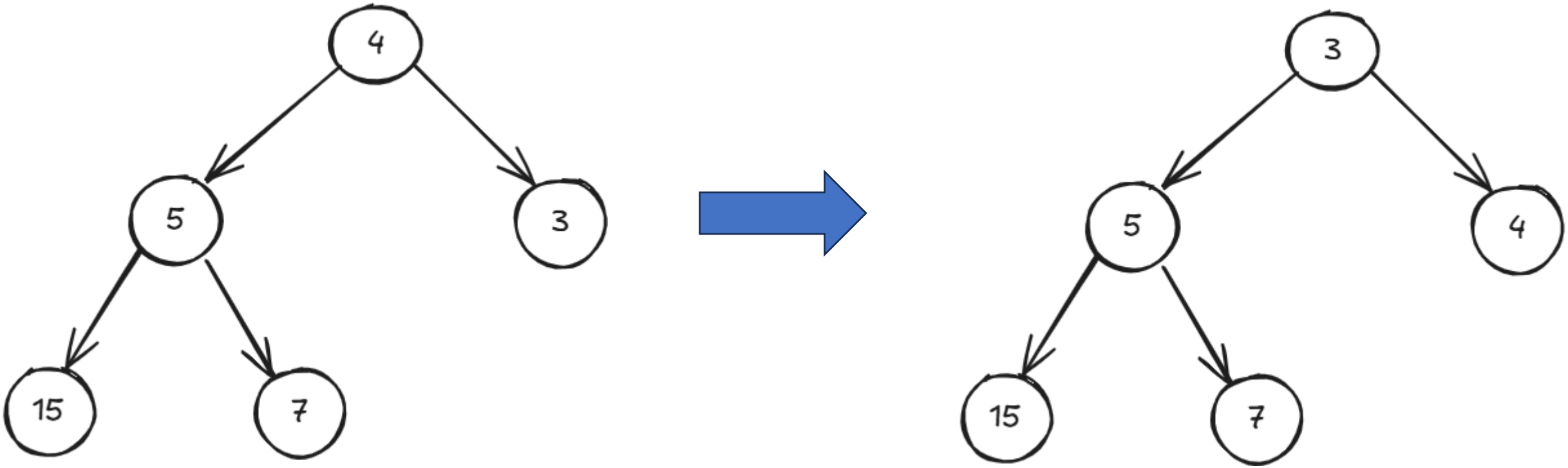
Remoção dos valores 1 e 2



	0	1	2	3	4	5	6
arr	4	5	3	15	7		

Remoção

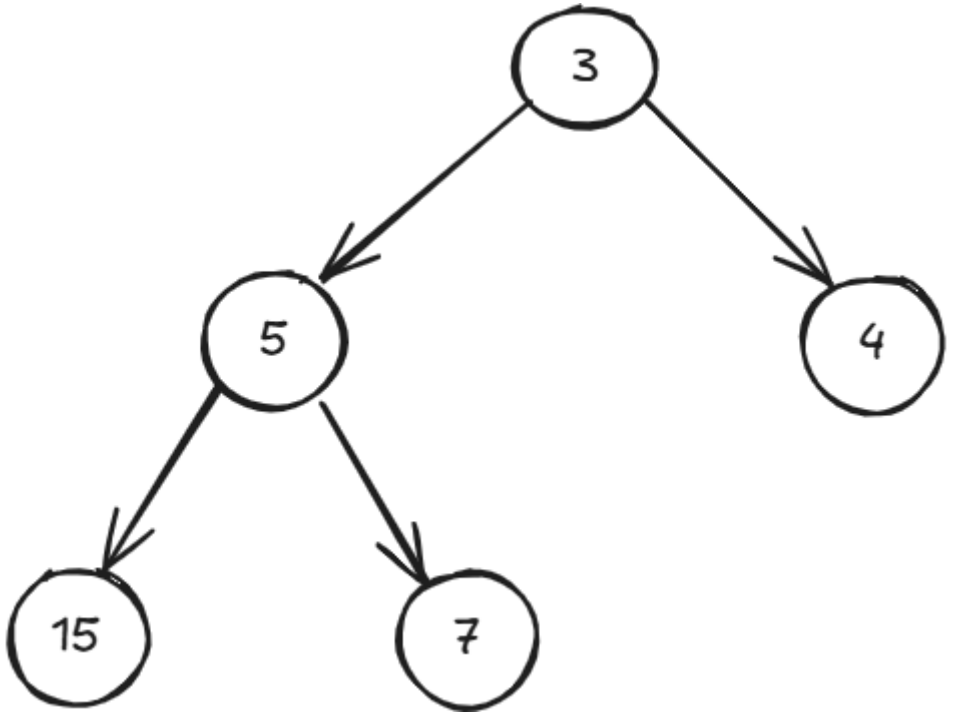
Remoção dos valores 1 e 2



	0	1	2	3	4	5	6
arr	3	5	4	15	7		

Remoção

Remoção dos valores 1 e 2



	0	1	2	3	4	5	6
arr	3	5	4	15	7		

Remoção

- A função *restaurarMinHeap* é responsável por garantir que a estrutura de dados Min Heap continue válida a partir de um índice *i* após uma operação que possa ter violado a propriedade da heap.
- Ela começa calculando os índices dos filhos esquerdo e direito do nó *i* usando as funções *esquerda(i)* e *direita(i)*.
- Em seguida, compara os valores dos filhos com o valor do nó atual para identificar qual dos filhos tem o menor valor.

```c
void restaurarMinHeap(MinHeap* heap, int i) {
    int l = esquerda(i);
    int r = direita(i);
    int menor = i;

    if (l < heap->tam && heap->arr[l] < heap->arr[menor]) {
        menor = l;
    }

    if (r < heap->tam && heap->arr[r] < heap->arr[menor]) {
        menor = r;
    }

    if (menor != i) {
        troca(&heap->arr[i], &heap->arr[menor]);
        restaurarMinHeap(heap, menor);
    }
}
```

Remoção

- Se um dos filhos tiver um valor menor que o nó atual, o índice de menor é atualizado para o índice do filho.
- Se o nó atual não for o menor, os valores do nó e do filho com o menor valor são trocados (usando a função troca), e a função é chamada recursivamente para o novo índice do nó trocado.
- Esse processo continua até que a propriedade da Min Heap seja restaurada, ou seja, até que o nó atual não precise mais ser trocado com seus filhos.

```c
void restaurarMinHeap(MinHeap* heap, int i) {
    int l = esquerda(i);
    int r = direita(i);
    int menor = i;

    if (l < heap->tam && heap->arr[l] < heap->arr[menor]) {
        menor = l;
    }

    if (r < heap->tam && heap->arr[r] < heap->arr[menor]) {
        menor = r;
    }

    if (menor != i) {
        troca(&heap->arr[i], &heap->arr[menor]);
        restaurarMinHeap(heap, menor);
    }
}
```

Remoção

- A função minimo é responsável por extrair o valor mínimo de uma Min Heap, ou seja, a raiz da heap, que sempre contém o menor valor.
- Ela começa verificando se a heap está vazia, retornando -1 em caso afirmativo. Se a heap contiver apenas um elemento, o valor da raiz é retornado e a heap é ajustada para um tamanho zero.

```c
int minimo(MinHeap* heap) {
    if (heap->tam <= 0) {
        return -1;
    }

    if (heap->tam == 1) {
        heap->tam--;
        return heap->arr[0];
    }

    int raiz = heap->arr[0];
    heap->arr[0] = heap->arr[heap->tam - 1];
    heap->tam--;
    restaurarMinHeap(heap, 0);
    return raiz;
}
```

Remoção

- Caso contrário, o valor da raiz (que é o menor valor da heap) é armazenado em raiz, a raiz é substituída pelo último elemento da heap (o último elemento do vetor, o tamanho da heap é diminuído, e a função restaurarMinHeap é chamada a partir do índice 0 para reestruturar a heap e garantir que a propriedade da Min Heap seja mantida.
- O valor da raiz (o valor mínimo) é então retornado.

```c
int minimo(MinHeap* heap) {
    if (heap->tam <= 0) {
        return -1;
    }

    if (heap->tam == 1) {
        heap->tam--;
        return heap->arr[0];
    }

    int raiz = heap->arr[0];
    heap->arr[0] = heap->arr[heap->tam - 1];
    heap->tam--;
    restaurarMinHeap(heap, 0);
    return raiz;
}
```

Remoção

```c
void reduzirChave(MinHeap* heap, int i, int valor) {
    heap->arr[i] = valor;

    while (i != 0 && heap->arr[pai(i)] > heap->arr[i]) {
        troca(&heap->arr[i], &heap->arr[pai(i)]);
        i = pai(i);
    }
}
```

- A função *reduzirChave* é responsável por diminuir o valor de uma chave específica em uma Min Heap e garantir que a propriedade da heap seja mantida após a alteração.
- Ela recebe como parâmetros a heap, o índice do elemento a ser alterado (i) e o novo valor para esse elemento (valor).
- Primeiramente, o valor do elemento no índice i é atualizado para o novo valor fornecido.

Remoção

```c
void reduzirChave(MinHeap* heap, int i, int valor) {
    heap->arr[i] = valor;

    while (i != 0 && heap->arr[pai(i)] > heap->arr[i]) {
        troca(&heap->arr[i], &heap->arr[pai(i)]);
        i = pai(i);
    }
}
```

- Em seguida, a função entra em um loop no qual verifica se o valor do elemento na posição i ainda é maior que o valor de seu pai (usando a função pai(i)).
- Se for, a função troca os dois valores para garantir que a propriedade da Min Heap seja preservada, ou seja, o valor do nó sempre deve ser menor ou igual aos seus filhos.
- Esse processo de troca e "subida" do elemento é repetido enquanto o valor do nó for maior que o valor do seu pai, até que o nó atinja uma posição correta na heap ou se torne a raiz.

Remoção

```c
void deletar(MinHeap* heap, int i) {
    if (i >= heap->tam) {
        printf("\nIndice fora dos limites\n");
        return;
    }

    reduzirChave(heap, i, INT_MIN);
    minimo(heap);
}
```

- A função deletar é responsável por remover um elemento específico de uma Min Heap, dado o seu índice i.
- Inicialmente, a função verifica se o índice i está dentro dos limites da heap, ou seja, se é menor que o tamanho da heap.
- Caso contrário, imprime uma mensagem de erro e retorna. Se o índice for válido, a função chama a função *reduzirChave* para diminuir o valor do elemento na posição i para *INT_MIN* (o menor valor possível para um inteiro), garantindo que esse elemento se torne o novo valor mínimo da heap.

Remoção

```c
void deletar(MinHeap* heap, int i) {
    if (i >= heap->tam) {
        printf("\nIndice fora dos limites\n");
        return;
    }

    reduzirChave(heap, i, INT_MIN);
    minimo(heap);
}
```

- Após essa alteração, a função chama minimo para extrair a raiz (o valor mínimo), que agora será o elemento que originalmente estava na posição i, efetivamente removendo-o da heap.
- Assim, a função utiliza uma combinação de diminuição da chave e remoção do mínimo para excluir um elemento específico da Min Heap.

Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 17 –Árvore Heap (Definição e Implementação)