

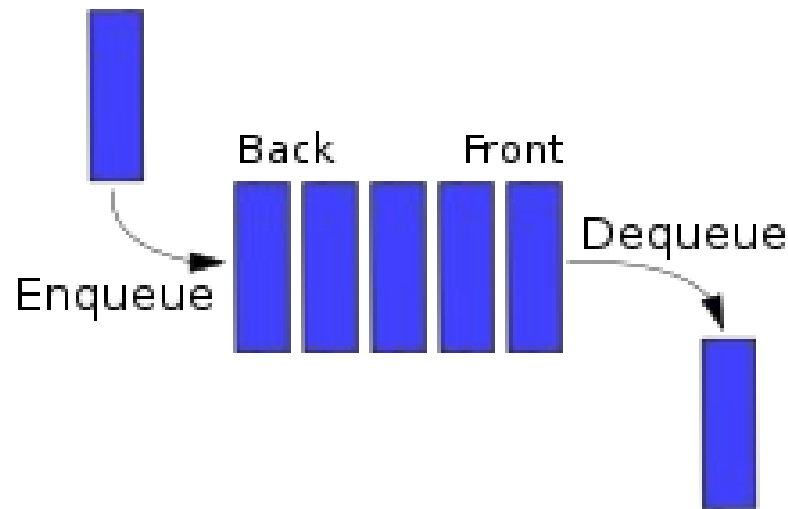
Fila Priorizada

Eduardo Furlan Miranda
2024-02-01

Adaptado do material do Prof. P. Feofiloff

Fila priorizada (Priority Queues - PQ)

- Tipo de dado abstrato que generaliza **fila**, **pilha**, e **fila randomizada**
- Remove o menor (MinPQ) ou maior item (MaxPQ)



Fila



Pilha

- Coleção

- Insere e apaga itens

- Fila

- Entra de uma lado, sai do outro

- Pilha

- Remove o item recém adicionado

- Pilha aleatória

- Remove um item aleatório

- Pilha priorizada

- Remove o menor ou maior item

operation	argument	return
insert	P	
insert	Q	
insert	E	
remove max		Q
insert	X	
insert	A	
insert	M	
remove max		X
insert	P	
insert	L	
insert	E	
remove max		P

Remove o maior (ou o menor) item

operation	argument	return value	size	contents (unordered)
insert	P		1	P
insert	Q		2	P Q
insert	E		3	P Q E
remove max		Q	2	P E
insert	X		3	P E X
insert	A		4	P E X A
insert	M		5	P E X A M
remove max		X	4	P E M A
insert	P		5	P E M A P
insert	L		6	P E M A P L
insert	E		7	P E M A P L E
remove max		P	6	E E M A P L

A sequence of operations on a priority queue

	MaxPQ()	cria uma PQ de máximo
	MaxPQ(int cap)	cria uma PQ de máximo com capacidade cap
	MaxPQ(Item[] a)	cria uma PQ de máximo com os itens que estão em a[]
void	insert(Item v)	insere o item v nesta PQ
Item	max()	devolve um item máximo deste PQ
Item	delMax()	remove e devolve um item máximo desta PQ
boolean	isEmpty()	esta PQ está vazia?
int	size()	número de itens desta PQ

Coleção

- Tipo de dado que armazena um grupo de itens

data type	core operations	data structure
stack	PUSH, POP	<i>linked list, resizing array</i>
queue	ENQUEUE, DEQUEUE	<i>linked list, resizing array</i>
priority queue	INSERT, DELETEMAX	<i>binary heap</i>
symbol table	PUT, GET, DELETE	<i>binary search tree, hash table</i>
set	ADD, CONTAINS, DELETE	<i>binary search tree, hash table</i>

Implementação elementar

```
insert(node)
{
    list.append(node)
}

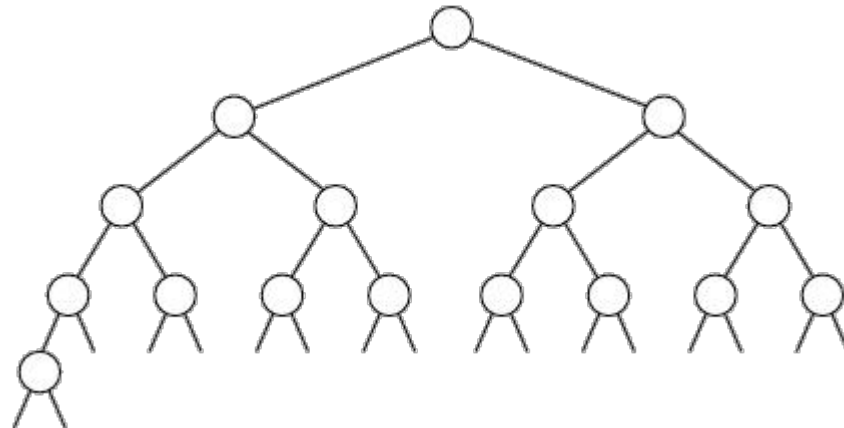
pull()
{
    highest = list.get_first_element()
    foreach node in list
    {
        if highest.priority < node.priority
        {
            highest = node
        }
    }
    list.remove(highest)
    return highest
}
```

Implementação elementar

- Implementação fácil: armazenar os itens em um vetor ou lista
- Se o vetor for mantido em ordem
 - `delMin()` é mais rápido
 - Consome tempo constante
 - `insert()` é lento
 - Consome tempo proporcional a N (nº de elementos)
- Se o vetor não estiver em ordem
 - `insert()` é rápido
 - Consome tempo constante
 - `delMin()` é lento
 - Consome tempo proporcional a N

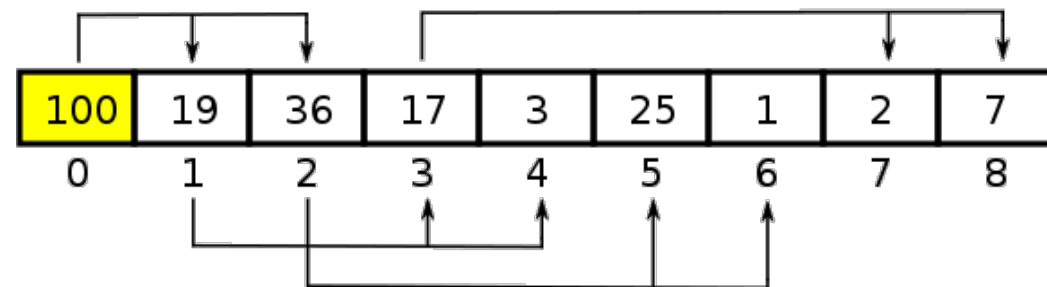
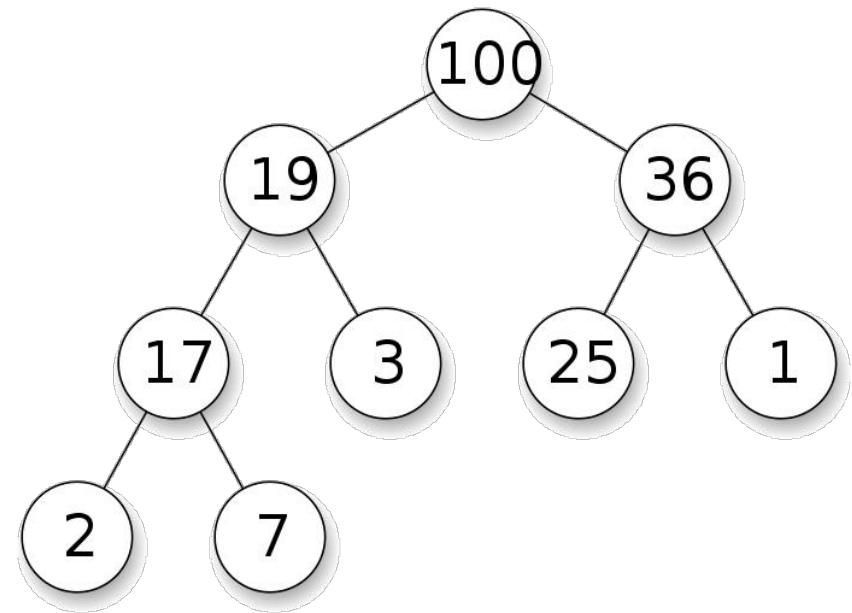
Árvore binária completa

- Árvore binária
 - Vazia ou nó com links para árvores binárias esquerda e direita
- Árvore completa
 - Perfeitamente balanceada, exceto no nível inferior



Implementação rápida, baseada em *heap*

- Heap é uma estrutura de dados baseada em árvore
 - **max-heap**: “os nós anteriores possuem valores maiores ou iguais”
 - **min-heap**: ... menores ou iguais
 - o nó no topo é chamado nó raiz
- O heap é uma implementação eficiente da **fila de prioridade**
- **Filas de prioridade** são frequentemente chamadas de "heaps"



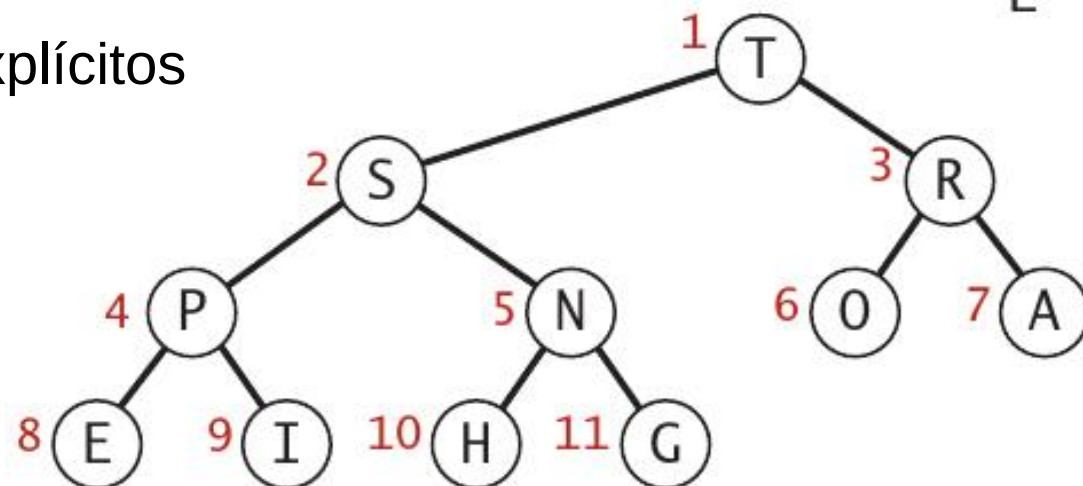
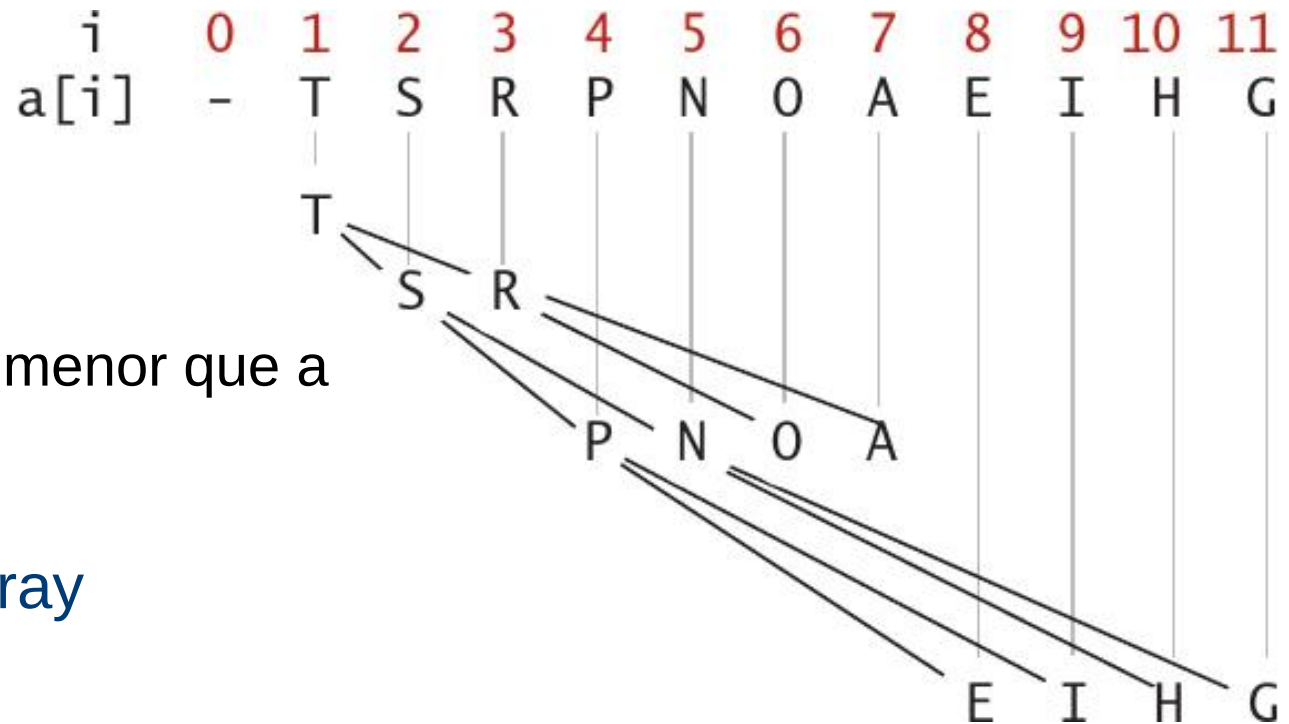
Representação de um heap binário

- Árvore binária heap-ordenada

- Chaves em nós
- Chave do parente não é menor que a chave do filho

- Representação como array

- Índices iniciam em 1
- Nós em níveis ordenados
- Não tem necessidade de links explícitos
- Os 2 filhos de um índice k são:
 - $2 * k$ e $2 * k + 1$
- O pai de um índice k é:
 - $k / 2$

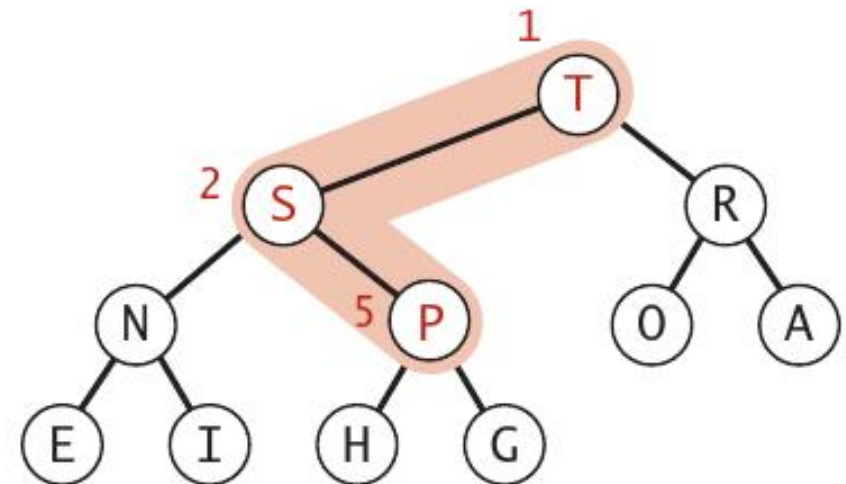
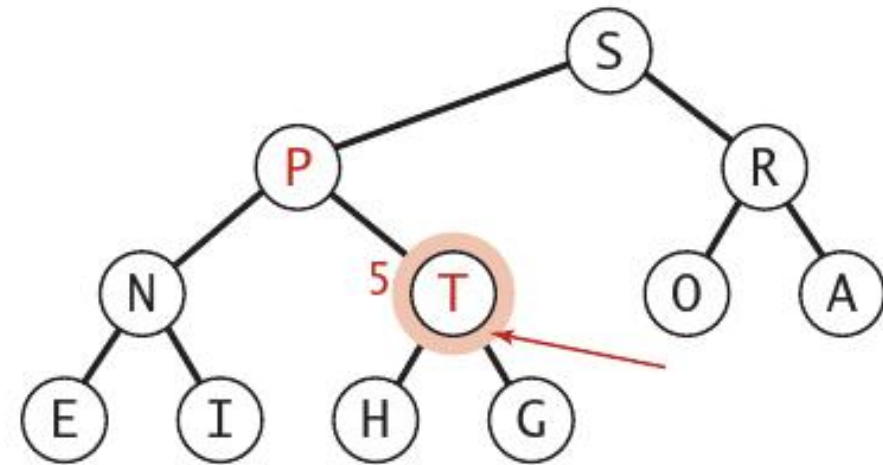


Heap binário: promoção

- Reheapify de baixo para cima (nadar) [*bottom-up reheapify (swim)*]

```
private void swim(int k) {
    while (k > 1 && less(k/2, k))
    {
        exch(k, k/2);
        k = k/2;
    }
}
```

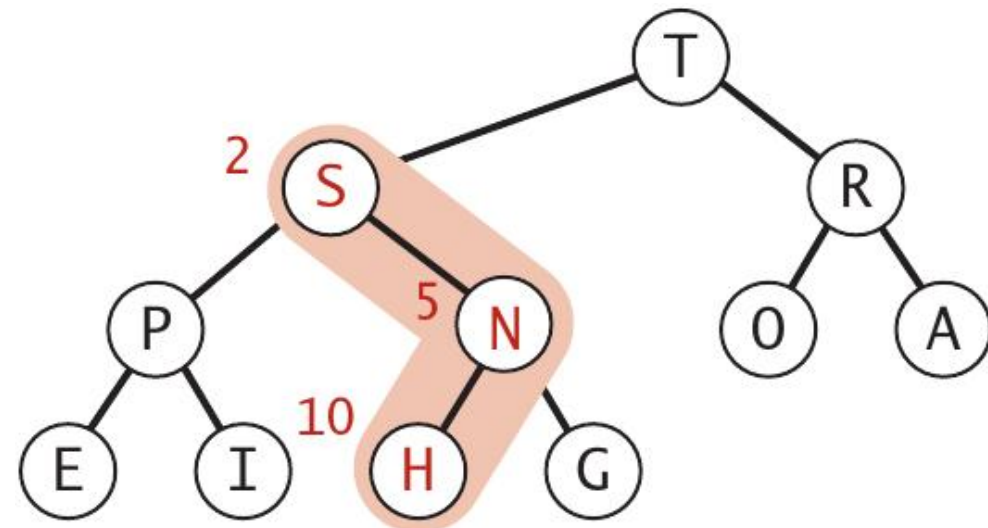
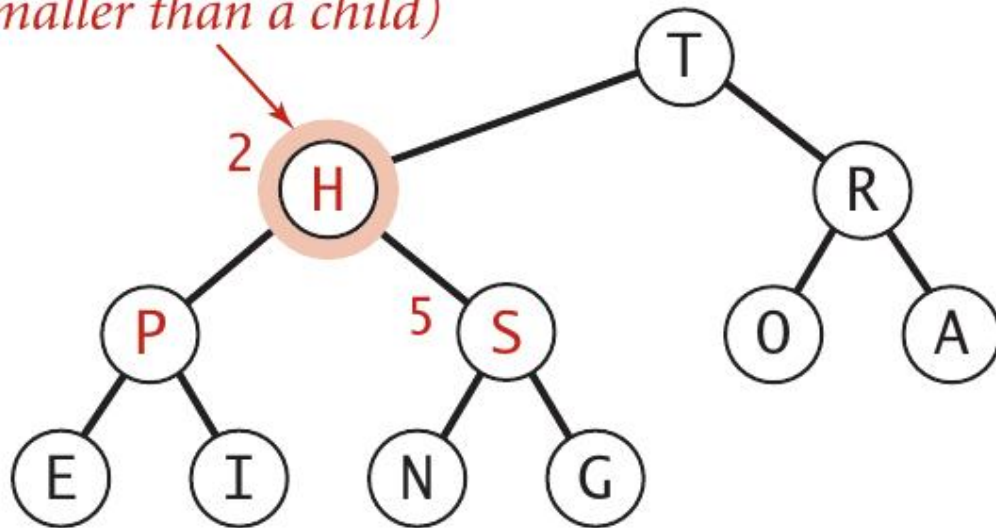
o parente de um
nó k é $k/2$



Heap binário: rebaixamento

- Heapify de cima para baixo (afundar). [*Top-down heapify (sink)*]

*violates heap order
(smaller than a child)*



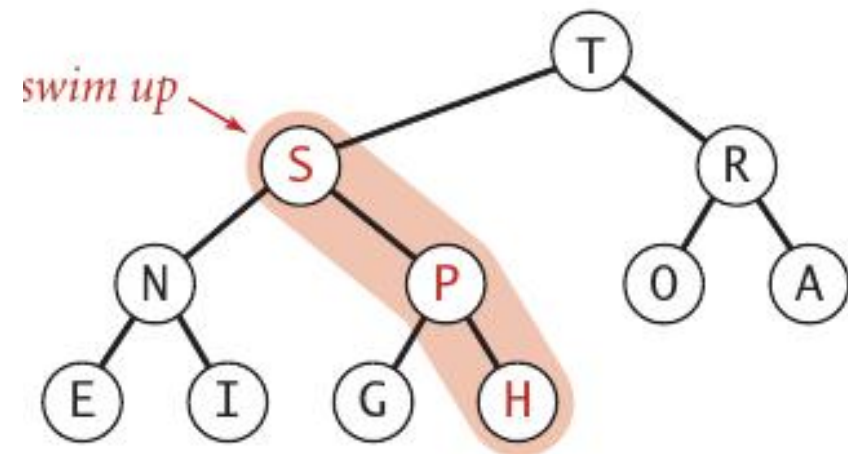
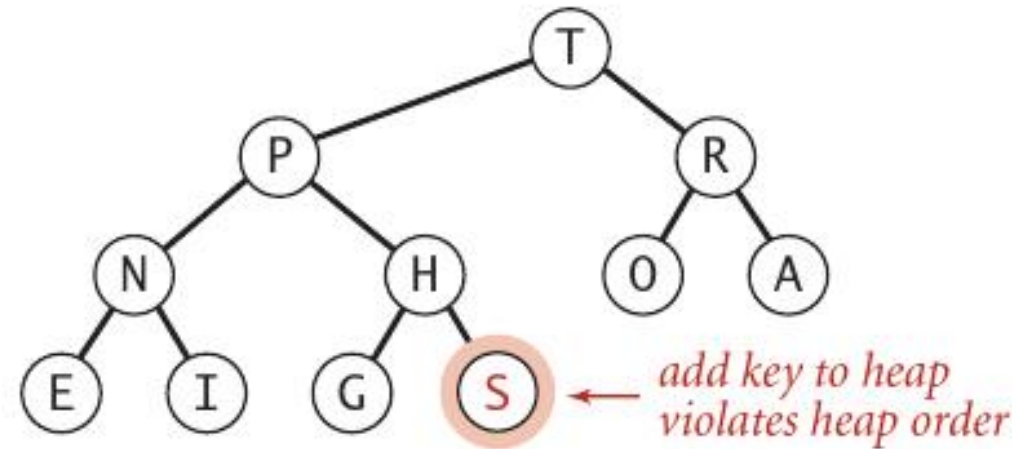
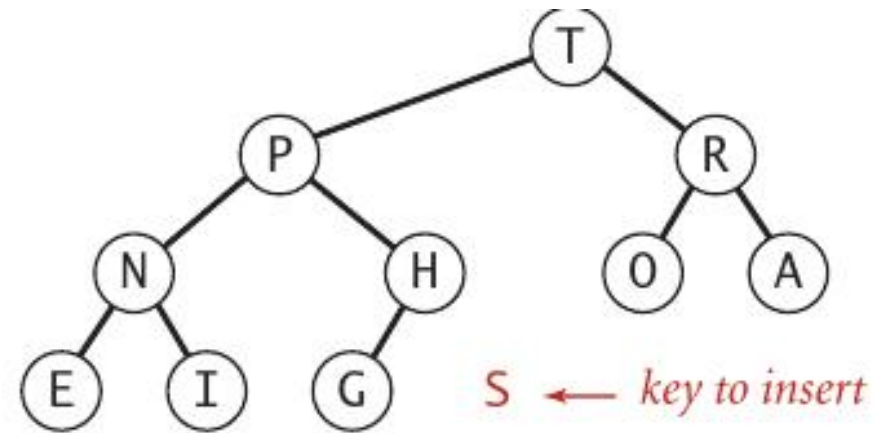
```
private void sink(int k) {
    while (2 * k <= N) {
        int j = 2 * k;
        if (j < N && less(j, j + 1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

os filhos do nó k
são $2 * k$ e $2 * k + 1$

Heap binário: inserção

- Adiciona um nó no final e nada para cima

```
public void insert(Key x) {  
    pq[++n] = x;  
    swim(n);  
}
```

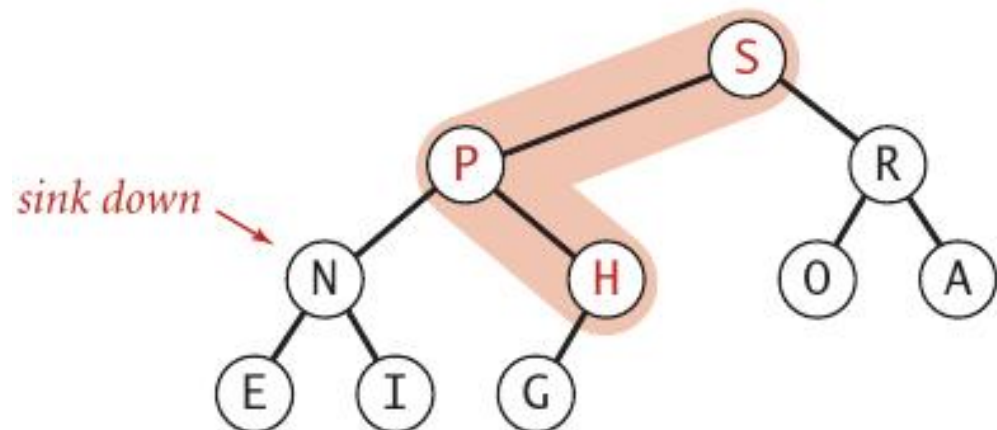
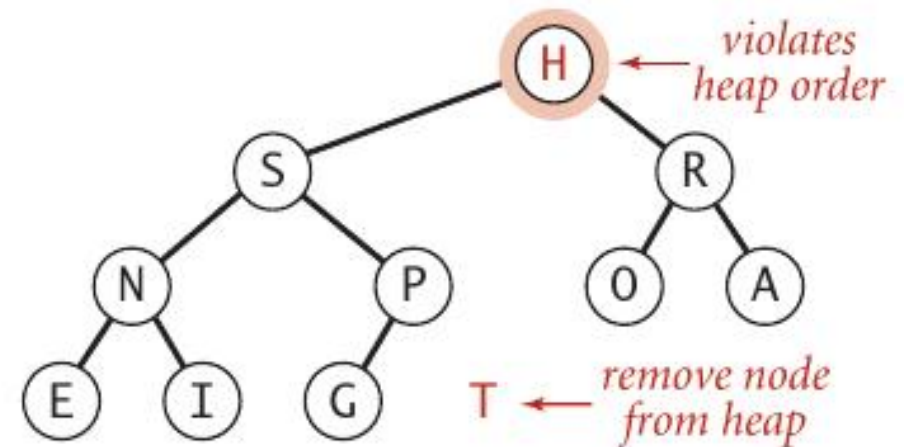
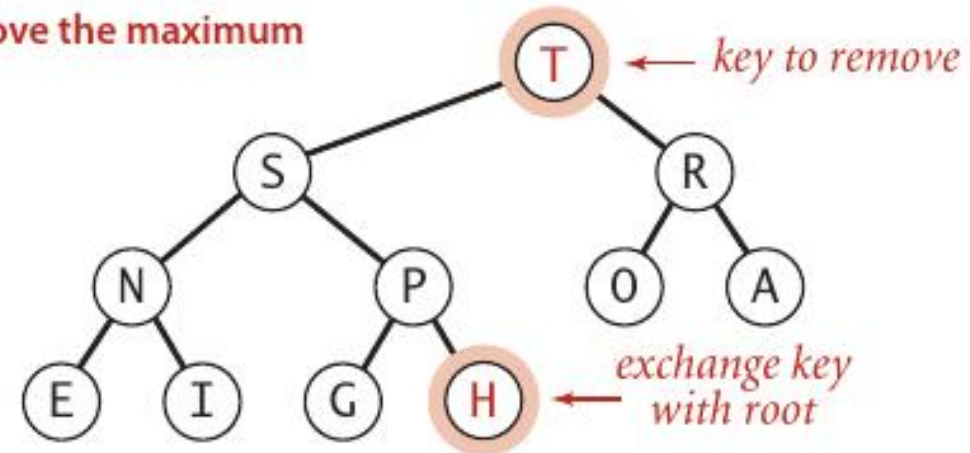


Heap binário: excluir o máximo

- Troca a raiz pelo nó no final e, em seguida, afunda

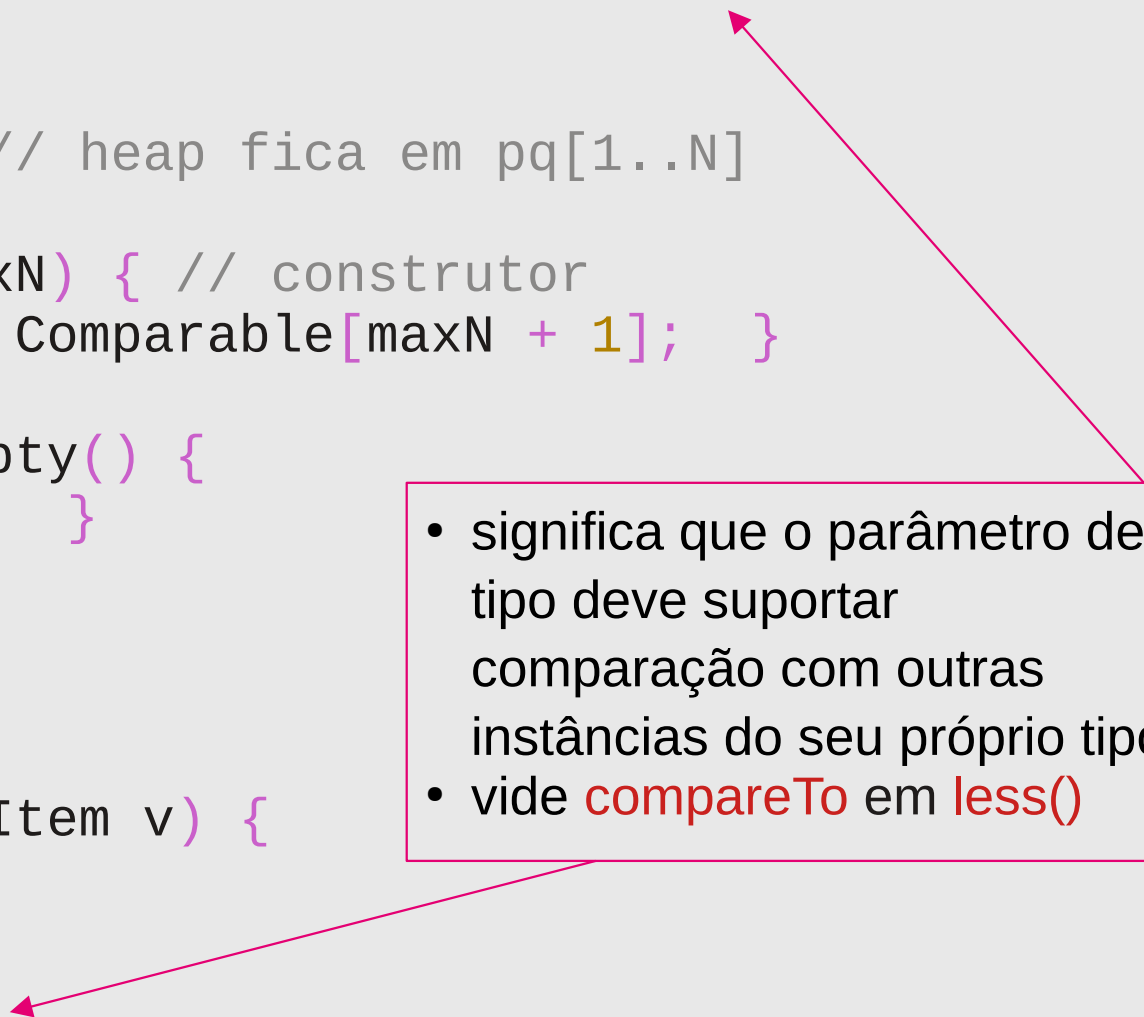
```
public Key delMax() {
    Key max = pq[1];
    exch(1, n--);
    sink(1);
    pq[n + 1] = null;
    return max;
}
```

remove the maximum



Implementação em Java

```
public class MaxPQ < Item extends Comparable < Item > > {  
  
    private Item[] pq;  
    private int N = 0; // heap fica em pq[1..N]  
  
    public MaxPQ(int maxN) { // construtor  
        pq = (Item[]) new Comparable[maxN + 1];  
    }  
  
    public boolean isEmpty() {  
        return N == 0;  
    }  
  
    public int size() {  
        return N;  
    }  
  
    public void insert(Item v) {  
        pq[++N] = v;  
        swim(N);  
    }  
  
    private boolean less(int i, int j){  
        return pq[i].compareTo(pq[j]) < 0;  
    }  
}
```



- significa que o parâmetro de tipo deve suportar comparação com outras instâncias do seu próprio tipo
- vide **compareTo** em **less()**


```

private void swim(int k) {
    while (k > 1 && less(k / 2, k)) {
        exch(k / 2, k);
        k = k / 2;
    }
}

private void sink(int k) {
    while (2 * k <= N) {
        int j = 2 * k;
        if (j < N && less(j, j + 1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}

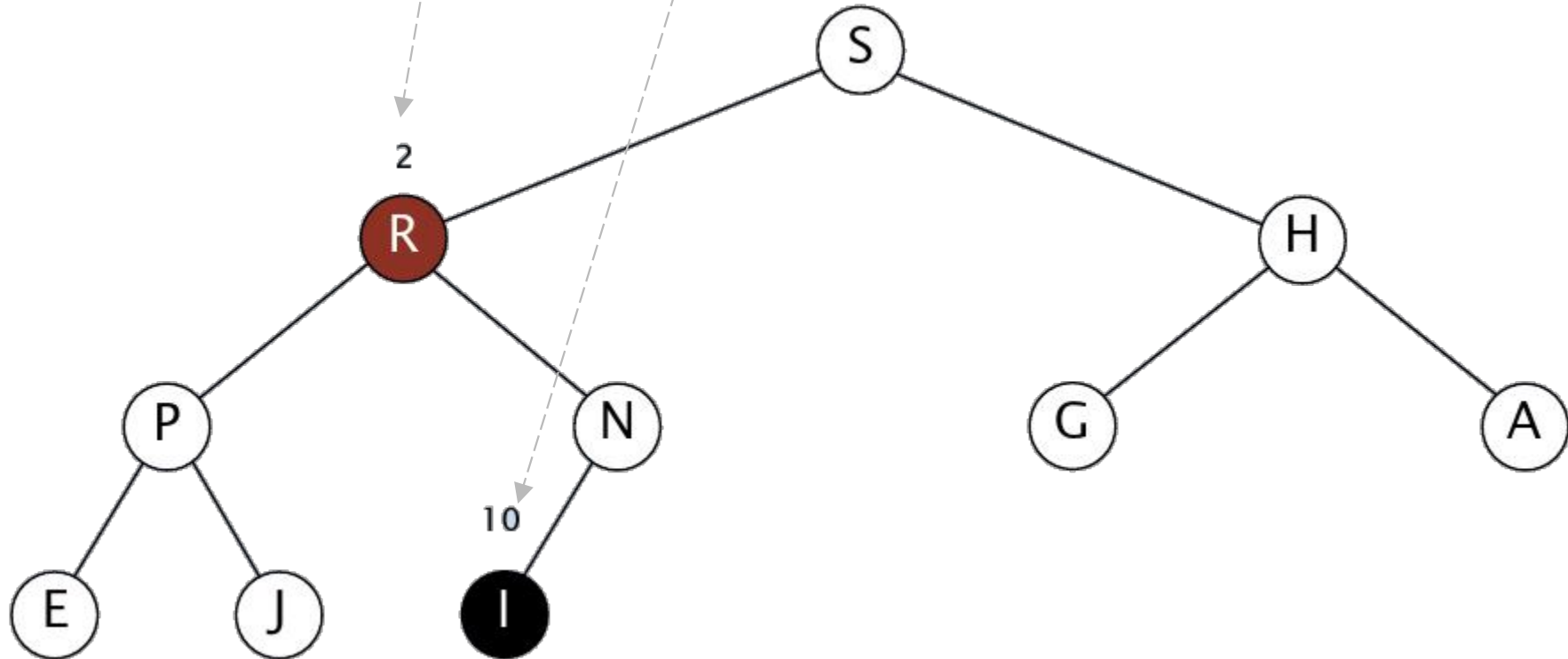
public Item delMax() {
    Item max = pq[1];
    exch(1, N--);
    pq[N + 1] = null;
    sink(1);
    return max;
}

private void exch(int i, int j) {
    Item t = pq[i];
    pq[i] = pq[j];
    pq[j] = t;
}
}

```

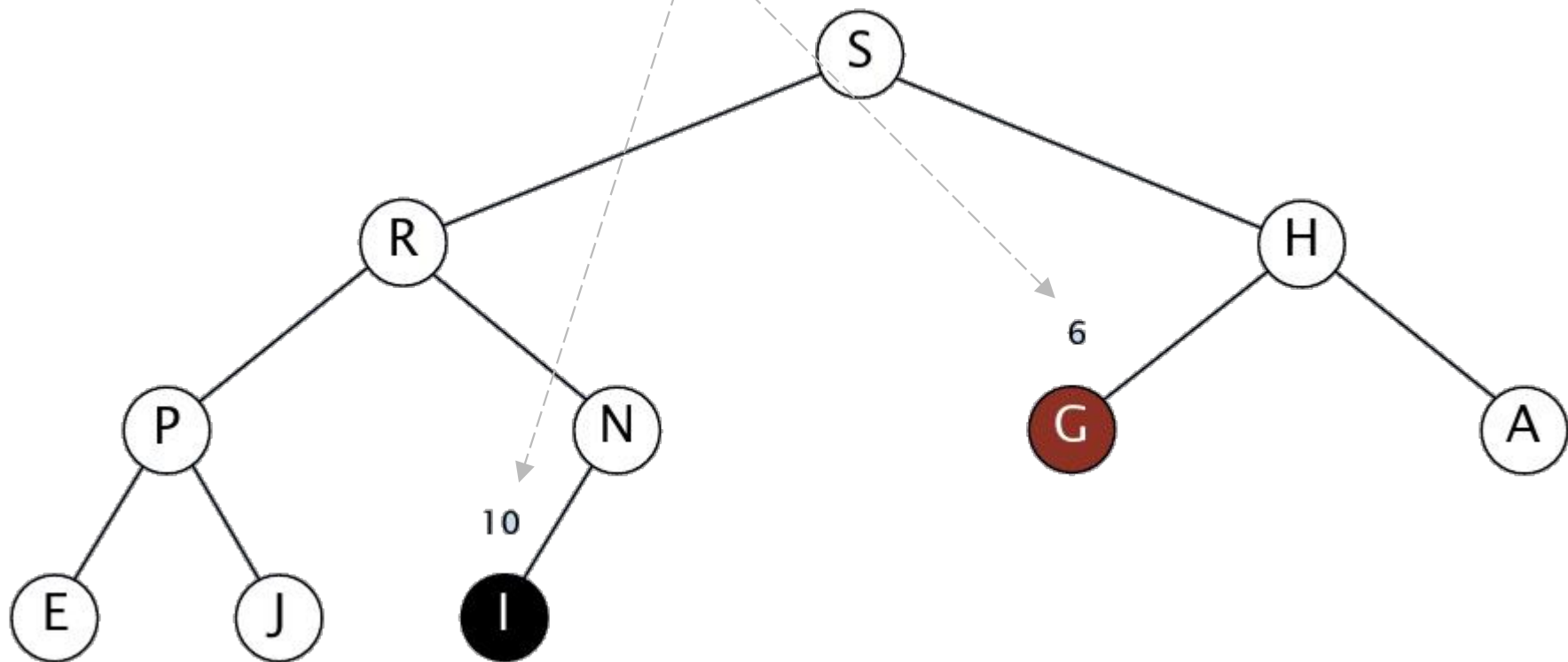
Apagar uma chave de um *heap* binário

- Escolher um índice r , entre 1 e n ← n = tamanho do array
- Executar $\text{exch}(r, n--)$ ← depois de trocar, o $--$ decrementa o valor de n
- Executar ou $\text{sink}(r)$ ou $\text{swim}(r)$



Em ramos diferentes

- Escolher um índice r , entre 1 e n
- Executar $\text{exch}(r, n--)$
- Executar ou $\text{sink}(r)$ ou $\text{swim}(r)$

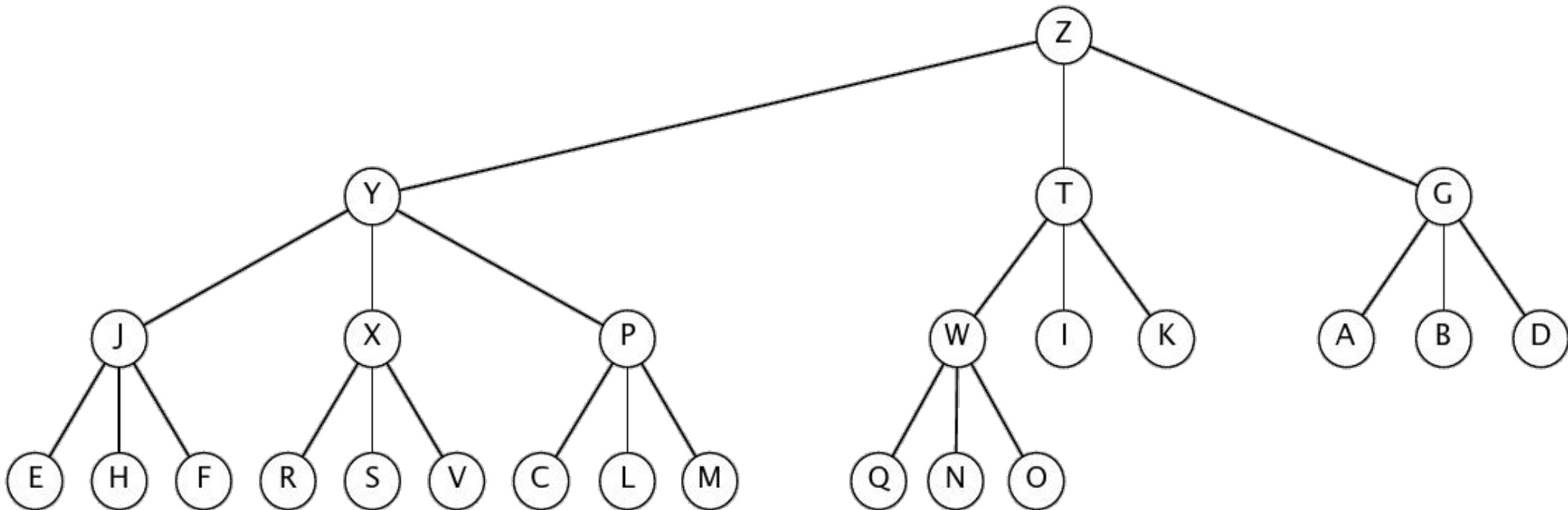


Heap binário: algoritmos melhorados

- Meias-trocas (*half-exchanges*) em sink e swin
 - Em um primeiro passo
 - Coloca-se o menor valor na raiz
 - No segundo passo
 - O laço interno pode ser simplificado
 - Mais adiante será mostrado uma aplicação em *Insertion-sort*
- Heurística de "salto" de Floyd
 - Afunda (sink) a raiz até o final da parte de baixo da árvore
 - Nada (swin) de volta
 - Quantidade menor de comparações
 - Mais trocas

Heap binário: algoritmos melhorados

- Heap de múltiplas vias (*Multiway heaps*)
 - Árvore completa *d-way* (ou *d-ary heaps*, ou *d-heaps*)
 - Não é árvore binária ou 2-way
 - A chave dos pais não é menor que as chaves dos filhos
 - Melhora cache misses e vm page faults



Exemplo de aplicação de *half-exchanges*

- O exemplo usa o Insertion-sort, primeiro sem half-exchanges, e depois com a aplicação da melhoria
- Basicamente a melhoria é do laço interno

Insertion-sort sem half-exchange

```
a = [9, 5, 6, 7, 8, 4, 3, 2, 1, 0]
print(a)
for i in range(1, len(a)) :
    v = a[i]
    j = i - 1
    print(f"j={j}, v={v}")
    while j >= 0 and v < a[j] :
        a[j + 1] = a[j]
        a[j] = v
        print(j, v, a)
        j -= 1
print("Resultado:\n", a)
```

Oportunidades de melhorias - Laço interno

$j \geq 0$

$a[j] = v$

Saída

[9, 5, 6, 7, 8, 4, 3, 2, 1, 0]

j=0, v=5

0 5 [5, 9, 6, 7, 8, 4, 3, 2, 1, 0]

j=1, v=6

1 6 [5, 6, 9, 7, 8, 4, 3, 2, 1, 0]

j=2, v=7

2 7 [5, 6, 7, 9, 8, 4, 3, 2, 1, 0]

j=3, v=8

3 8 [5, 6, 7, 8, 9, 4, 3, 2, 1, 0]

j=4, v=4

4 4 [5, 6, 7, 8, 4, 9, 3, 2, 1, 0]

3 4 [5, 6, 7, 4, 8, 9, 3, 2, 1, 0]

2 4 [5, 6, 4, 7, 8, 9, 3, 2, 1, 0]

1 4 [5, 4, 6, 7, 8, 9, 3, 2, 1, 0]

0 4 [4, 5, 6, 7, 8, 9, 3, 2, 1, 0]

$j=5, v=3$

5 3 [4, 5, 6, 7, 8, 3, 9, 2, 1, 0]
 4 3 [4, 5, 6, 7, 3, 8, 9, 2, 1, 0]
 3 3 [4, 5, 6, 3, 7, 8, 9, 2, 1, 0]
 2 3 [4, 5, 3, 6, 7, 8, 9, 2, 1, 0]
 1 3 [4, 3, 5, 6, 7, 8, 9, 2, 1, 0]
 0 3 [3, 4, 5, 6, 7, 8, 9, 2, 1, 0]

$j=6, v=2$

6 2 [3, 4, 5, 6, 7, 8, 2, 9, 1, 0]
 5 2 [3, 4, 5, 6, 7, 2, 8, 9, 1, 0]
 4 2 [3, 4, 5, 6, 2, 7, 8, 9, 1, 0]
 3 2 [3, 4, 5, 2, 6, 7, 8, 9, 1, 0]
 2 2 [3, 4, 2, 5, 6, 7, 8, 9, 1, 0]
 1 2 [3, 2, 4, 5, 6, 7, 8, 9, 1, 0]
 0 2 [2, 3, 4, 5, 6, 7, 8, 9, 1, 0]

$j=7, v=1$

7 1 [2, 3, 4, 5, 6, 7, 8, 1, 9, 0]
 6 1 [2, 3, 4, 5, 6, 7, 1, 8, 9, 0]
 5 1 [2, 3, 4, 5, 6, 1, 7, 8, 9, 0]
 4 1 [2, 3, 4, 5, 1, 6, 7, 8, 9, 0]
 3 1 [2, 3, 4, 1, 5, 6, 7, 8, 9, 0]
 2 1 [2, 3, 1, 4, 5, 6, 7, 8, 9, 0]
 1 1 [2, 1, 3, 4, 5, 6, 7, 8, 9, 0]
 0 1 [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

$j=8, v=0$

8 0 [1, 2, 3, 4, 5, 6, 7, 8, 0, 9]
 7 0 [1, 2, 3, 4, 5, 6, 7, 0, 8, 9]
 6 0 [1, 2, 3, 4, 5, 6, 0, 7, 8, 9]
 5 0 [1, 2, 3, 4, 5, 0, 6, 7, 8, 9]
 4 0 [1, 2, 3, 4, 0, 5, 6, 7, 8, 9]
 3 0 [1, 2, 3, 0, 4, 5, 6, 7, 8, 9]
 2 0 [1, 2, 0, 3, 4, 5, 6, 7, 8, 9]
 1 0 [1, 0, 2, 3, 4, 5, 6, 7, 8, 9]
 0 0 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Algoritmo melhorado - com *half-exchanges*

```

a = [9, 5, 6, 7, 8, 4, 3, 2, 1, 0]

# put smallest element in position to serve as
sentinel
for i in range(len(a)-1, 0, -1) :
    if (a[i] < a[i-1]) :
        a[i], a[i-1] = a[i-1], a[i]
        print(i, a)

print(a)
for i in range(1, n) :
    v = a[i]
    j = i
    print(f"j={j}, v={v}")
    while v < a[j-1] :
        a[j] = a[j-1]
        print(j, v, a)
        j -= 1
    a[j] = v
    print(j, v, a, "LAÇO EXTERNO")
print("Resultado:\n", a)

```

Melhorias no laço interno

[0, 9, 5, 6, 7, 8, 4, 3, 2, 1]

j=1, v=9

j=2, v=5

2 5 [0, 9, 9, 6, 7, 8, 4, 3, 2, 1]

1 5 [0, 5, 9, 6, 7, 8, 4, 3, 2, 1] LAÇO EXTERNO

j=3, v=6

3 6 [0, 5, 9, 9, 7, 8, 4, 3, 2, 1]

2 6 [0, 5, 6, 9, 7, 8, 4, 3, 2, 1] LAÇO EXTERNO

j=4, v=7

4 7 [0, 5, 6, 9, 9, 8, 4, 3, 2, 1]

3 7 [0, 5, 6, 7, 9, 8, 4, 3, 2, 1] LAÇO EXTERNO

j=5, v=8

5 8 [0, 5, 6, 7, 9, 9, 4, 3, 2, 1]

4 8 [0, 5, 6, 7, 8, 9, 4, 3, 2, 1] LAÇO EXTERNO

j=6, v=4

6 4 [0, 5, 6, 7, 8, 9, 9, 3, 2, 1]

5 4 [0, 5, 6, 7, 8, 8, 9, 3, 2, 1]

4 4 [0, 5, 6, 7, 7, 8, 9, 3, 2, 1]

3 4 [0, 5, 6, 6, 7, 8, 9, 3, 2, 1]

2 4 [0, 5, 5, 6, 7, 8, 9, 3, 2, 1]

1 4 [0, 4, 5, 6, 7, 8, 9, 3, 2, 1] LAÇO EXTERNO

$j=7, v=3$

7	3	[0, 4, 5, 6, 7, 8, 9 , 9, 2, 1]
6	3	[0, 4, 5, 6, 7, 8 , 8, 9, 2, 1]
5	3	[0, 4, 5, 6, 7 , 7, 8, 9, 2, 1]
4	3	[0, 4, 5, 6 , 6, 7, 8, 9, 2, 1]
3	3	[0, 4, 5 , 5, 6, 7, 8, 9, 2, 1]
2	3	[0, 4 , 4, 5, 6, 7, 8, 9, 2, 1]
1	3	[0, 3, 4, 5, 6, 7, 8, 9, 2, 1]

LAÇO EXTERNO

$j=8, v=2$

8	2	[0, 3, 4, 5, 6, 7, 8, 9 , 9, 1]
7	2	[0, 3, 4, 5, 6, 7, 8 , 8, 9, 1]
6	2	[0, 3, 4, 5, 6, 7 , 7, 8, 9, 1]
5	2	[0, 3, 4, 5, 6 , 6, 7, 8, 9, 1]
4	2	[0, 3, 4, 5 , 5, 6, 7, 8, 9, 1]
3	2	[0, 3, 4 , 4, 5, 6, 7, 8, 9, 1]
2	2	[0, 3 , 3, 4, 5, 6, 7, 8, 9, 1]
1	2	[0, 2, 3, 4, 5, 6, 7, 8, 9, 1]

LAÇO EXTERNO

j=9, v=1

9	1	[0, 2, 3, 4, 5, 6, 7, 8, 9 , 9]
8	1	[0, 2, 3, 4, 5, 6, 7, 8 , 8, 9]
7	1	[0, 2, 3, 4, 5, 6, 7 , 7, 8, 9]
6	1	[0, 2, 3, 4, 5, 6 , 6, 7, 8, 9]
5	1	[0, 2, 3, 4, 5 , 5, 6, 7, 8, 9]
4	1	[0, 2, 3, 4 , 4, 5, 6, 7, 8, 9]
3	1	[0, 2, 3 , 3, 4, 5, 6, 7, 8, 9]
2	1	[0, 2 , 2, 3, 4, 5, 6, 7, 8, 9]
1	1	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

LAÇO EXTERNO

Resultado:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Referências

FEOFILOFF, P. Filas Priorizadas. [S. l.]: IME-USP, 2019. Disponível em: <https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/priority.html>

SEDGEWICK, R.; WAYNE, K. **Algorithms**. [S. l.]: Addison-Wesley Professional, 2011.