Filas de prioridade

Marcelo K. Albertini

3 de Dezembro de 2013

Filas de prioridade

O que é uma fila de prioridade?

Estrutura de dados que generaliza a ideia de ordenação.

Coleções de elementos: inserir e remover itens. Qual remover?

- Pilha - remover o item mais novo
- Fila - remover o item mais velho
- Fila aleatorizada - remover item mais aleatório
- **Fila de prioridade** - remover o maior ou menor item

Exemplo de uso de filas de prioridade

operação	item	valor obtido
inserir	P	
inserir	Q	
inserir	E	
remover máximo		Q
inserir	X	
inserir	A	
inserir	M	
remover máximo		X
inserir	P	
inserir	L	
inserir	E	
remover máximo		P

Aplicações

- Simulação orientada a eventos
  - clientes em uma fila
  - colisão de partículas
- Compressão de dados
  - Código de Huffman
- Busca em grafos
  - Algoritmo de Dijkstra
  - Algoritmo de Prim
- Inteligência artificial
  - Busca A*
- Sistemas operacionais
  - balanceamento de carga
  - manipulação de interrupções
- Filtragem de spam bayesiano
- Ranking web

Generalização de

pilhas, filas, filas aleatórias

Tipo abstrato de dados - API

public class	MaxFilaP <Chave extends Comparable<Chave> >	
	MaxFilaP()	criar fila de prioridade vazia
	MaxFilaP(Chave[] v)	criar fila com chaves
void	inserir(Chave c)	inserir chave na fila
Chave	removerMax()	retornar e remover chave máxima
boolean	estaVazio()	a fila está vazia?
Chave	max()	retorna chave máxima
int	size()	número de entradas na fila

Chave deve ser `Comparable`

Exemplo de uso de fila de prioridade

Encontrar os M maiores itens de uma sequência N itens

- Detecção de fraudes e ataques em redes de computadores
- Processamento de consultas em buscador web

Restrição

não há espaço de memória para guardar os N itens

Exemplo:detecção de fraudes

```
Marcelo $80
Joao    $30000
Maria   $9999
Paulo   $10000000
Beatriz $910
...
```

Suspeitos:

```
Paulo $10000000
Joao $30000
```

Meta

Encontrar os M maiores itens de uma sequência N itens: usar uma MinFilaP.

```
1  MinFilaP<Pagina> minfila = new MinFilaP<Pagina>();
2  BufferedReader teclado = new BufferedReader(new
       FileReader(System.in));
3  String consulta = teclado.nextLine();
4
5  while (teclado.hasNextLine()) {
6    String linha = teclado.nextLine();
7
8    Pagina p = new Pagina(linha, consulta);
9    // cada pagina guarda um valor de relevancia para a
         consulta
10
11   minfila.inserir(p);
12
13   if (minfila.size() > M)
14     minfile.removerMin();
15 }
```

Meta

Encontrar os M maiores itens de uma sequência N itens: usar uma MinFilaP.

Complexidades

implementação	tempo	espaço
fila elementar	$m \times n$	m
ordenação	$n \log n$	n
heap binária	$n \log m$	m
melhor teórico	n	m

```java
1 public class MaxFilaDesordenada<Chave extends
      Comparable<Chave>> {
2
3   private Chave[] fila; //fila[i] = i-esimo elemento
4   private int N;
5
6   public MaxFilaDesordenada(int capacidade) {
7     fila = (Chave[]) new Comparable[capacity];
8   }
9
10  public boolean estaVazio() { return N==0; }
11
12  public inserir(Chave c) { fila[N++] = c; }
13
14  public Chave removerMax() {
15    int max = 0;
16    for (int i =1; i < N; i++)
17      if (fila[max].compareTo(fila[i]) < 0)
18        max = i;
19
20    troca(max, N-1);
21    return fila[--N];
22  }
23 }
```

Implementação de filas de prioridade

Meta

Implementar todas as operações eficientemente.

Implementação	inserção	remoção
vetor desordenado	1	N
vetor ordenado	N	1
meta	log n	log n

Heap binária

O que é?

Heap binária é uma estrutura de dados que permite a execução eficiente das operação de inserção e remoção em filas de prioridade.

Heap binária é um tipo de árvore

Árvore binária

Vazia ou nós com ligações à direita ou à esquerda para outras árvores.

Árvore binária completa

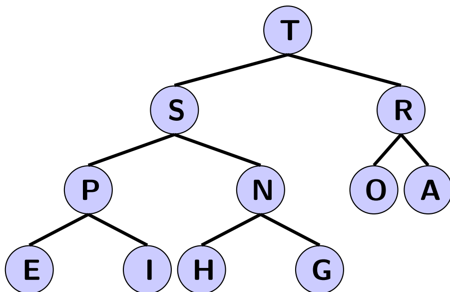
Perfeitamente balanceada. Cada nível possui todos os nós possíveis, exceto, possivelmente o último.

- altura de uma árvore binária é $\lfloor logN \rfloor$
- altura aumenta quando N for potência de 2:

Heap binária

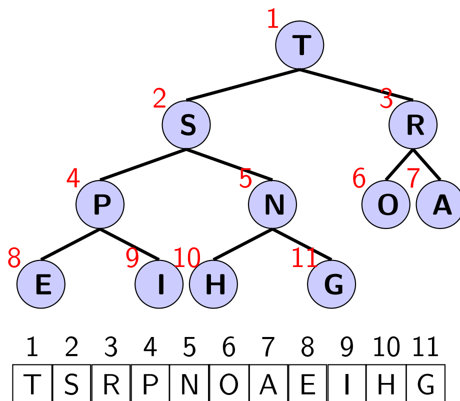
Uma árvore binária completa

- Chaves estão em nós
- Chaves-pais não são menores que chaves-filhas



Propriedades da heap binária

- Implementação em um vetor
- Maior chave está em v[1], que é a raiz da árvore binária
- Usa posições do vetor para ligações entre nós
  - Nó pai do nó na posição k está na posições $\lfloor (k/2) \rfloor$
    - Lembre-se $\lfloor 1.9 \rfloor = 1$
  - Filhos do nó na posição k estão em $2k$ e $2k + 1$



Implementação de operações da heap binária

- Propriedades invariantes da heap binária devem ser sempre mantidas
- Na inserção ou remoção de chaves, verificar e corrigir violações de propriedades

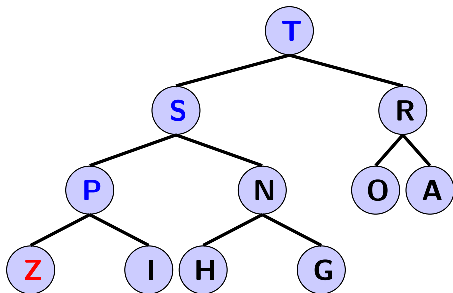
Promoção em heap binária

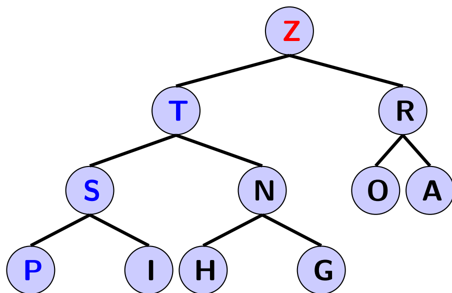
Cenário

Chave-filha torna-se maior que chave-pai. Isso provoca uma violação de regras da heap.

Para eliminar a violação

- Trocar chave-folha com chave-pai
- Repetir até corrigir todas as violações





Promoção em heap: algoritmo

Elemento *fila*$[k/2]$ deve ser maior que *fila*$[k]$, se não for devemos trocá-los.

```
1 void promocao(int k) {
2   while (k > 1 && (fila[k/2].compareTo(fila[k])< 0)) {
3     troca(k, k/2); // troca pois fila[k/2] < fila[k]
4     k = k/2; // sobe na árvore com divisão inteira
5   }
6 }
```

Inserção em heap binária

- Inserção: Inserir nó no fim do vetor e aplique a `promoção`
- Custo: No máximo $1 + \log N$ comparações

```
1 void insercao(Chave c) {
2   fila[++N] = c;
3   promocao(N);
4 }
```

```
1 void insercao(Chave c) {
2 fila[++N] = c; promocao(N);
3 }
```

```
1 void promocao(int k) {
2  while (k > 1 &&
3      (fila[k/2].compareTo(
           fila[k]) < 0)) {
4   troca(k, k/2);
5   k = k/2;
6  }
7 }
```

Começar com inserção no fim.

Inserção da *chave* $= 9$

1	2	3	4	5	6	7	8	9
8	7	5	6	1	3	2	4	9

```
1 void insercao(Chave c) {
2 fila[++N] = c; promocao(N);
3 }
```

```
1 void promocao(int k) {
2  while (k > 1 &&
3      (fila[k/2].compareTo(
           fila[k]) < 0)) {
4   troca(k, k/2);
5   k = k/2;
6  }
7 }
```

Inserção da *chave* $= 9$

1	2	3	4	5	6	7	8	9
8	7	5	6	1	3	2	4	9

1	2	3	4	5	6	7	8	9
8	7	5	9	1	3	2	4	6

Promoção no lugar do pai

$9/2 = 4$

```
1 void insercao(Chave c) {
2 fila[++N] = c; promocao(N);
3 }
```

```
1 void promocao(int k) {
2   while (k > 1 &&
3       (fila[k/2].compareTo(
              fila[k]) < 0)) {
4     troca(k, k/2);
5     k = k/2;
6   }
7 }
```

Inserção da *chave* = 9

1	2	3	4	5	6	7	8	9
8	7	5	6	1	3	2	4	9

1	2	3	4	5	6	7	8	9
8	7	5	9	1	3	2	4	6

1	2	3	4	5	6	7	8	9
8	7	5	9	1	3	2	4	6

1	2	3	4	5	6	7	8	9
8	9	5	7	1	3	2	4	6

Promoção no lugar do pai

$4/2 = 2$

```
void insercao(Chave c) {
fila[++N] = c; promocao(N);
}
```

```
void promocao(int k) {
 while (k > 1 &&
     (fila[k/2].compareTo(
         fila[k]) < 0)) {
  troca(k, k/2);
  k = k/2;
 }
}
```

Inserção da *chave* = 9

1	2	3	4	5	6	7	8	9
8	7	5	6	1	3	2	4	9

1	2	3	4	5	6	7	8	9
8	7	5	9	1	3	2	4	6

1	2	3	4	5	6	7	8	9
8	7	5	9	1	3	2	4	6

1	2	3	4	5	6	7	8	9
8	9	5	7	1	3	2	4	6

1	2	3	4	5	6	7	8	9
8	9	5	7	1	3	2	4	6

1	2	3	4	5	6	7	8	9
9	8	5	7	1	3	2	4	6

Promoção no lugar do pai

$1/2 = 1$

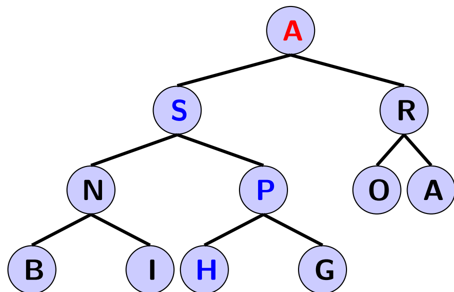
Demoção em heap binária

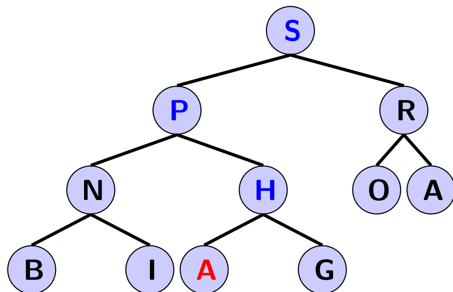
Cenário

Chave-pai se torna menor que alguma chave-filha. Violação!

Para eliminar a violação

- Trocar chave na posição pai com chave na posição filha com maior chave.
  - Porque não filho com menor chave?
- Repetir até resolver todas as violações





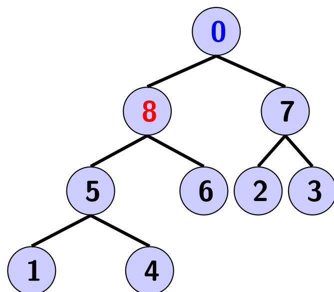
Implementação: democão

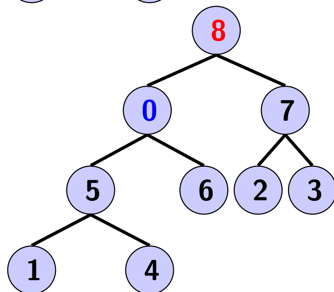
```
1  void democao(int k) {
2    while (2*k <= N) {
3      int j = 2*k; // filho da "esquerda" esta em j
4
5      // compara com filho da "direita" para pegar maior
6      if (j < N && (fila[j].compareTo(fila[j+1]) <0))
7        j++;
8
9      if (fila[k].compareTo(fila[j]) >= 0)
10       break; // interromper processo de democao
11
12     troca(k, j); // efetuar democao
13     k = j;
14   }
15 }
```

Execução da demoção

1	2	3	4	5	6	7	8	9	10
0	8	7	5	6	2	3	1	4	9

1	2	3	4	5	6	7	8	9	10
8	0	7	5	6	2	3	1	4	9

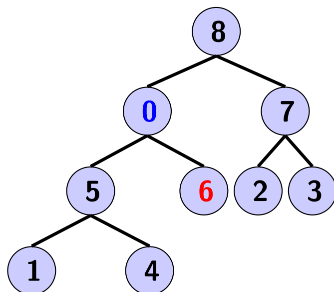


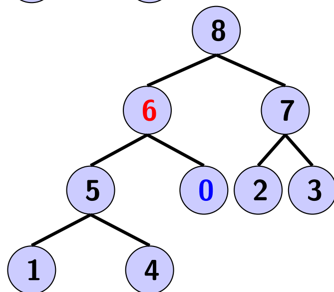


Execução da demoção

1	2	3	4	5	6	7	8	9	10
8	0	7	5	6	2	3	1	4	9

1	2	3	4	5	6	7	8	9	10
8	6	7	5	0	2	3	1	4	9

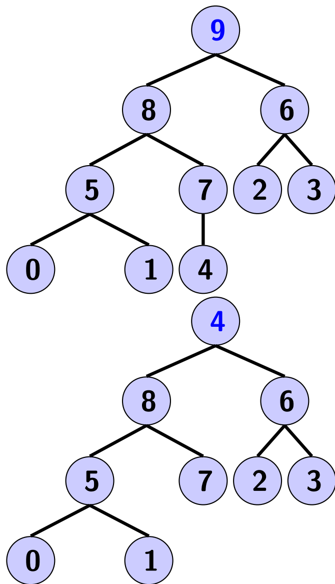




Remover max = 9

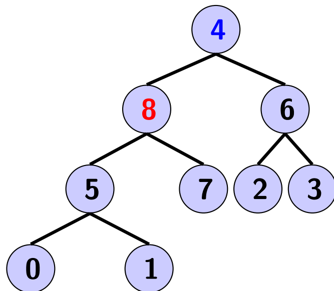
1	2	3	4	5	6	7	8	9	10
9	8	6	5	7	2	3	0	1	4

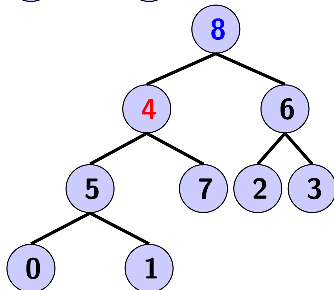
1	2	3	4	5	6	7	8	9
4	8	6	5	7	2	3	0	1



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

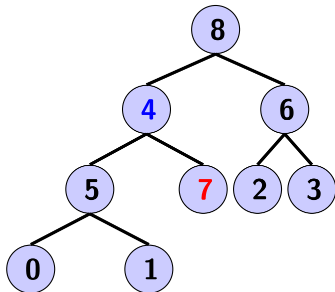
1	2	3	4	5	6	7	8	9
8	4	6	5	7	2	3	0	1

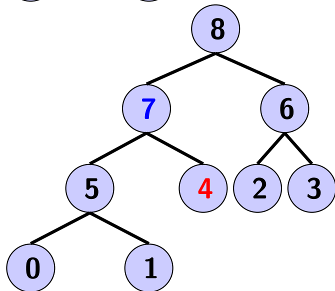




1	2	3	4	5	6	7	8	9
8	4	6	5	7	2	3	0	1

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

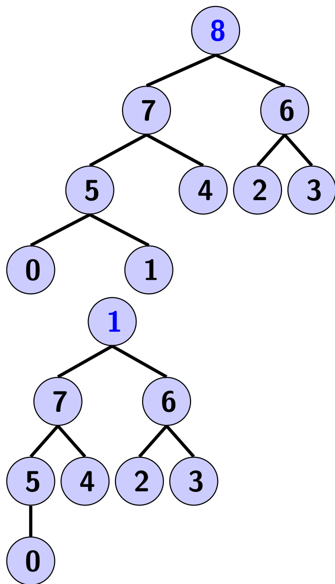




Remover max = 8

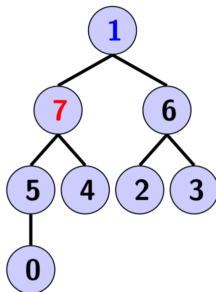
1	2	3	4	5	6	7	8	9
8	7	6	5	4	2	3	0	1

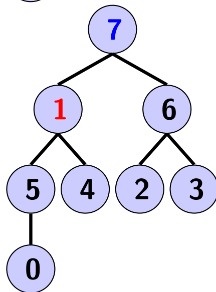
1	2	3	4	5	6	7	8
1	7	6	5	4	2	3	0



1	2	3	4	5	6	7	8
1	7	6	5	4	2	3	0

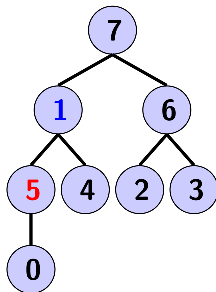
1	2	3	4	5	6	7	8
7	1	6	5	4	2	3	0

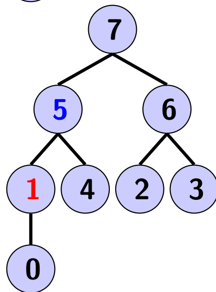




1	2	3	4	5	6	7	8
7	1	6	5	4	2	3	0

1	2	3	4	5	6	7	8
7	5	6	1	4	2	3	0

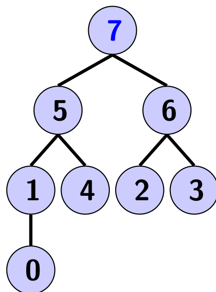


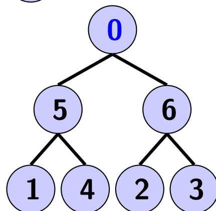


Remover max = 7

1	2	3	4	5	6	7	8
7	5	6	1	4	2	3	0

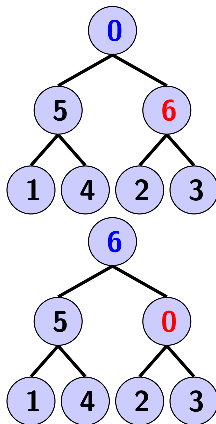
1	2	3	4	5	6	7
0	5	6	1	4	2	3





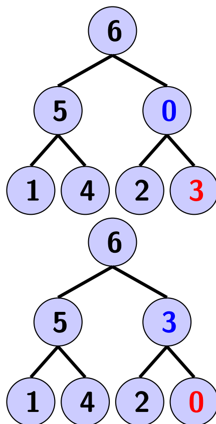
1	2	3	4	5	6	7
0	5	6	1	4	2	3

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



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

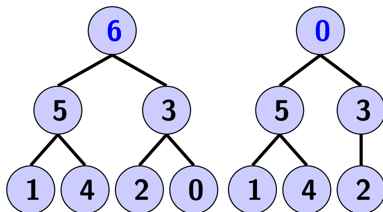
1	2	3	4	5	6	7
6	5	3	1	4	2	0



Remover max = 6

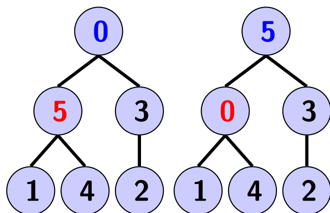
1	2	3	4	5	6	7
6	5	3	1	4	2	0

1	2	3	4	5	6
0	5	3	1	4	2



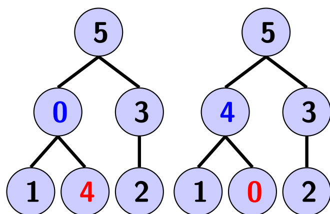
1	2	3	4	5	6
0	5	3	1	4	2

1	2	3	4	5	6
5	0	3	1	4	2



1	2	3	4	5	6
5	0	3	1	4	2

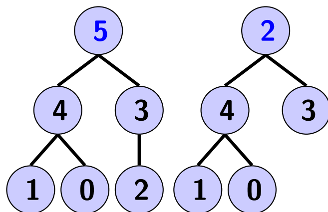
1	2	3	4	5	6
5	4	3	1	0	2



Remover max = 5

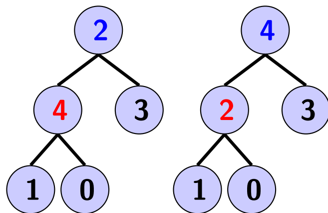
1	2	3	4	5	6
5	4	3	1	0	2

1	2	3	4	5
2	4	3	1	0

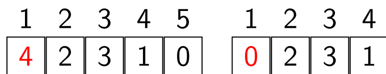


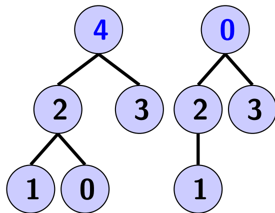
1	2	3	4	5
2	4	3	1	0

1	2	3	4	5
4	2	3	1	0



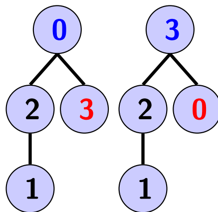
Remover max = 4



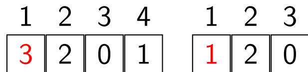


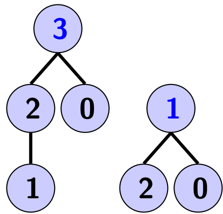
1	2	3	4
0	2	3	1

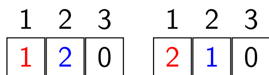
1	2	3	4
3	2	0	1

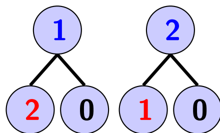


Remover max = 3



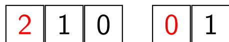


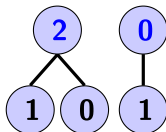

1 2 3
1 2 0
1 2 3
2 1 0

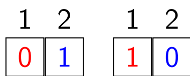

1
2 0
2
1 0

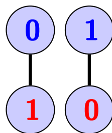
Remover max $= 2$

1 2 3   1 2

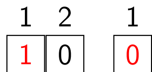


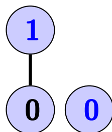



1 2
0 1
1 2
1 0

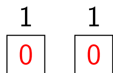

0
1
1
0

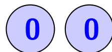
Remover max = 1





Remover max = 0





Operação removerMax()

Remover chave máxima

Trocar raiz com nó no fim e aplicar o processo de democão.

```
1 public Chave removerMax() {
2   Chave max = fila[1];
3   troca(1, N--);
4   democao(1);
5   fila[N+1] = null; // libera referência de objeto para
        coletor de lixo
6   return max;
7 }
```

Custo

No máximo $2 \log N$ comparações

Custos de operações de listas de prioridade

implementação	inserção	remover máximo	ver máximo
vetor desordenado	1	N	N
vetor ordenado	N	1	1
heap binário	$\log N$	$\log N$	1
heap d-ária	$\log_d N$	$d \log_d N$	1
heap de Fibonacci	1	$\log N$	1
impossível	1	1	1

Porque ter todas as operações lineares é impossível?

Observações gerais

Lista de prioridade orientada ao mínimo

Trocar operações de comparação `compareTo`

Outras operações

- Ao inserir, evitar overflow e usar operação de redimensionar vetor
- Ao remover, verificar e lançar exceção se fila está vazia
- Remoção de item arbitrário
- Mudar prioridade de item

Usar operações: `promocao` e `remocao`

Heapsort com ordenação no lugar (in-place)

Ideia

- fazer **in-place**, ou seja, sem usar vetor auxiliar
- criar uma heap binária orientada ao máximo (max heap) com todas as chaves disponíveis
- aplicar `removerMax` para cada uma das chaves, até acabar a heap

Criação da heap **in-place**

Primeira passada: criar heap usando método bottom-up.

```
1 for (int k = N/2; k >= 1; k--)
2   democao(vet, k, N);
```

Segunda passada: remover o máximo, um por vez, deixar elemento no vetor em vez de liberar memória (com `null`).

```
1 while (N > 1) {
2   troca(vet, 1, N-1);
3   democao(vet, 1, N);
4 }
```

Análise matemática

- Construção da heap usa até $2N$ comparações e trocas
- Heapsort usa até $2N \log N$ comparações e trocas
- Heapsort é um algoritmo in-place com pior caso $N \log N$

Comparações

- Mergesort: não é in-place (possível, porém não é prático) e usa $O(n)$ espaço extra
- Quicksort: quadrático no pior caso
- Heapsort: $N \log N$, sem espaço extra
- iteração interna mais longa que do Quicksort (constante de complexidade maior no caso médio)
- Mal uso de caching pelo uso frequente de referências em posições não contínuas da memória
- Heapsort não é estável

Tabela de algoritmos

	no lugar	estável	pior	médio	melhor	comentário
seleção	S		$N^2/2$	$N^2/2$	$N^2/2$	N trocas
inserção	S	S	$N^2/2$	$N^2/4$	N	bom p/ pequeno N
shell	S		?	?	N	fácil e $< N^2$
quick	S		$N^2/2$	$2N \log N$	$N \log N$	mais rápido na prática
quick 3	S		$N^2/2$	$2N \log N$	N	chaves duplicadas
merge		S	$N \log N$	$N \log N$	$N \log N$	estável e garantido
heap	S		$2N \log N$	$2N \log N$	$N \log N$	no lugar e garantido
ideal	S	S	$N \log N$	$N \log N$	$N \log N$	

quick 3 = quicksort em três partes