classificação e pesquisa de dados

ordenação
comparação

ordenação heapsort

1. Construir uma max-heap a partir do vetor (*in-place*)
2. Trocar a raiz da heap com o último elemento da heap
3. Desconsiderar da heap a última posição onde foi colocada a raiz
4. Processar o subvetor para restabelecer as características da heap
5. Executar os passos 2, 3 e 4 enquanto o subvetor for maior que 1

ordenação heapsort

1. Construir uma max-heap a partir do vetor (*in-place*)

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

maxheap

ordenação heapsort

2. Trocar a raiz da heap com o último elemento da heap

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

3. Desconsiderar da heap a última posição onde foi colocada a raiz

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

ordenação
heapsort
classificação e pesquisa de dados

4. Processar o subvetor para restabelecer as características da heap

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

ordenação
heapsort
classificação e pesquisa de dados

2. Trocar a raiz da heap com o último elemento da heap

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

3. Desconsiderar da heap a última posição onde foi colocada a raiz

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

ordenação
heapsort
classificação e pesquisa de dados

```
void heapsort(int *v, int n) {
    maxheap(v, n);
    while(n > 0) {
        trocar(&v[0], &v[n-1]);
        n--;
        heapify(v, n, 0);
    }
}
```

ordenação
heapsort
classificação e pesquisa de dados

```
void maxheap(int *v, int n) {
    for(int idx = n/2-1; idx >= 0; idx--)
        heapify(v, n, idx);
}
```

```
void heapify(int *v, int n, int idx) {
    int esq, dir, maior = idx;
    bool troca = true;
    while(troca) {
        troca = false;
        esq = 2*maior + 1; dir = 2*maior + 2;
        if(esq < n && v[maior] < v[esq])
            maior = esq;
        if(dir < n && v[maior] < v[dir])
            maior = dir;
        if(maior != idx) {
            trocar(&v[maior], &v[idx]);
            troca = true;
            idx = maior;
        }
    }
```

$O(\lg n)$


```
void maxheap(int *v, int n) {
    for(int idx = n/2-1; idx >= 0; idx--)
        heapify(v, n, idx);
}
```

$O(\lg n)$

$n \times O(\lg n) = O(n \lg n)$


```
void heapsort(int *v, int n) {
    maxheap(v, n);
    while(n > 0) {
        trocar(&v[0], &v[n-1]);
        n--;
        heapify(v, n, 0);
    }
}
```

$O(n \lg n)$

$O(\lg n)$

$n \times O(\lg n) = O(n \lg n)$

$O(n \lg n)$

Algumas características

- Algoritmo *in-place* (in loco)
  - ⇒ memória adicional constante: $O(1)$
- Algoritmo **não** estável
  - ⇒ o algoritmo **não** preserva a ordem de registros com mesma chave

ordenação
heapsort

classificação e pesquisa de dados

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

Demonstração (não estável)

ordenação
quicksort

classificação e pesquisa de dados

Baseado no paradigma Divisão e Conquista (DC)
Divide and Conquer

Emprega o conceito de partição para separar elementos “menores” e “maiores”

Utiliza um pivô durante a fase de partição

ordenação
quicksort

classificação e pesquisa de dados

Partição

Seleciona um índice do pivô para promover a divisão dos dados
⇒ usaremos o índice 0

O pivô será posicionado na sua posição final do vetor

Elementos anteriores ao pivô serão menores que ele

Elementos posteriores ao pivô serão maiores ou iguais a ele

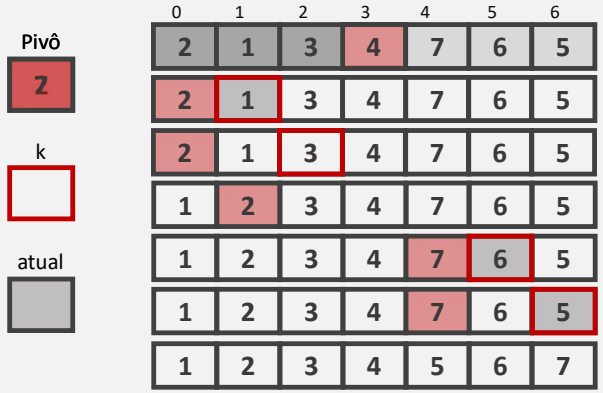
ordenação
quicksort

classificação e pesquisa de dados

Pivô: 4
k
atual

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

ordenação
quicksort
classificação e pesquisa de dados



ordenação
quicksort
classificação e pesquisa de dados

```
int particionamento(int *v, int esq, int dir) {
    int pivo = v[esq], pos = esq+1;
    for(int atual = pos; atual <= dir; atual++) {
        if(v[atual] < pivo) {
            troca(&v[atual], &v[pos]);
            pos++;
        }
    }                                          O(n)
    troca(&v[esq], &v[pos-1]);
    return pos-1;
}
```

ordenação
quicksort
classificação e pesquisa de dados

```
void quicksort(int *v, int esq, int dir) {
    stack<int> pilha; pilha.push(esq); pilha.push(dir);
    while(!pilha.empty()) {
        int dir, esq;
        dir = pilha.top(); pilha.pop();
        esq = pilha.top(); pilha.pop();
        int p = particionamento(v, esq, dir);  O(n)
        if(p-1 > esq) {
            pilha.push(esq); pilha.push(p-1);
        }
        if(p+1 < dir) {
            pilha.push(p+1); pilha.push(dir);
        }
    }
}
```

ordenação
quicksort
classificação e pesquisa de dados

```
void quicksort(int *v, int esq, int dir) {
    if(esq < dir) {
        int p = particionamento(v, esq, dir);
        quicksort(v, esq, p-1);
        quicksort(v, p+1, dir);
    }
}
```

Algumas características

Algoritmo *in-place* (in loco)

⇒ memória adicional constante: $O(1)$

Algoritmo **não** estável

⇒ o algoritmo **não** preserva a ordem de registros com mesma chave

ZIVIANI, Nivio. Projeto de Algoritmos com Implementação em Java e C++. São Paulo: Thomson Learning, 2007.

LEISERSON, C. E.; STEIN, C.; RIVEST, R. L., CORMEN, T.H. AlgoritmosTeoria e Prática. Tradução da 2ª edição americana. Rio de Janeiro: Campus, 2002.

LINTZMAYER, Carla Negri; MOTA, Guilherme Oliveira. Análise de Algoritmos e de Estruturas de Dados. Notas de aula. Versão: 29/06/2022.