Heap

Estruturas de Dados
Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-CompartilhaIgual 4.0 Internacional.
http://github.com/malbarbo/na-ed

Junto com o problema de busca, o problema de ordenação é um dos mais estudados da Computação.

O problema de ordenação consiste em, dada uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$, determinar uma permutação (reordenação) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ da sequência de entrada tal que, $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Para avaliarmos os algoritmos de ordenação, além da complexidade de tempo, consideramos o uso extra de memória e se a ordenação é estável.

Um algoritmo é **in-loco** se a quantidade de memória que ele precisa para executar é $O(1)$, ou seja, não depende do tamanho da entrada.

Um algoritmo de ordenação é **estável** se a ordem relativa dos elementos com chaves iguais é preservada.

Comparação entre os algoritmos de ordenação

Algoritmo	Estável?	Local?	Melhor	Médio	Pior
Inserção	Sim	Sim	$O(n)$	$O(n^2)$	$O(n^2)$
Seleção	Não	Sim	$O(n^2)$	$O(n^2)$	$O(n^2)$
Intercalação	Sim	Não	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$
Particionamento	Não	Sim	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$

Quais combinações de características poderíamos querer que não são atendidas por esses algoritmos?

Estável, local e pior caso $O(n \lg n)$. Existem alguns, mas não são viáveis na práticas pois são compilados de implementar e têm constantes altas.

Local e pior caso $O(n \lg n)$. Vamos ver como fazer isso usando uma estrutura de dados.

Mas antes, vamos fazer uma revisão.

A ideia de um algoritmo incremental é:

- Iniciar com a solução para um problema trivial;
- Estender a solução iterativamente para um problema maior até obter a solução do problema que queremos resolver;

Como projetar um algoritmo incremental para somar os elementos de um arranjo?

- Começamos com a soma do arranjo vazio que é 0;
- Estendemos a soma adicionando um elemento do arranjo por vez até que todos os elementos tenham sido somados.

Ordenação incremental

Como projetar um algoritmo incremental para ordenar os elementos de um arranjo?

- Iniciamos com um subarranjo vazio já ordenado;
- Estendemos o subarranjo já ordenado com um elemento da parte restante por vez até que todos os elementos tenham sido selecionados.

Temos que tomar duas decisões para tornar o processo concreto:

- Como selecionar o próximo elemento?
- Como estender o subarranjo ordenado?

Como selecionar o próximo elemento?

- Pegamos o primeiro elemento do restante.
- Qual é o custo? $O(1)$

Como estender o subarranjo ordenado?

- Inserindo o elemento selecionado na parte ordenada.
- Qual é o custo? $O(j)$, onde j é a quantidade de elementos já inseridos.

Qual é a complexidade de tempo da ordenação por inserção?

Qual é o melhor caso? A lista está em ordem não decrescente. A complexidade de tempo é $O(n)$.

Qual é o pior caso? A lista está em ordem não crescente, cada elemento deve ser levado até a posição 0 do arranjo. A complexidade de tempo é $O(n^2)$.

A implementação é in-loco? Sim.

A ordenação é estável? Sim.

Como selecionar o próximo elemento?

- Pegamos o menor elemento do restante.
- Qual é o custo? $O(n - j)$, onde j e de elementos já processados.

Como estender o subarranjo ordenado?

- Trocando de posição o menor elemento com o primeiro do restante.
- Qual é o custo? $O(1)$

Qual é a complexidade de tempo da ordenação por seleção? A complexidade de tempo é $O(n^2)$.

A implementação é in-loco? Sim.

A ordenação é estável? Não.

A ordenação por seleção é mais eficiente do que a ordenação por inserção? Não...

Quando projetamos o algoritmo de ordenação por seleção nós movemos o custo de inserção para a seleção e vice e versa... Parece que não ganhamos nada!

Mas isso não é verdade, agora podemos abordar o problema por outro ângulo!

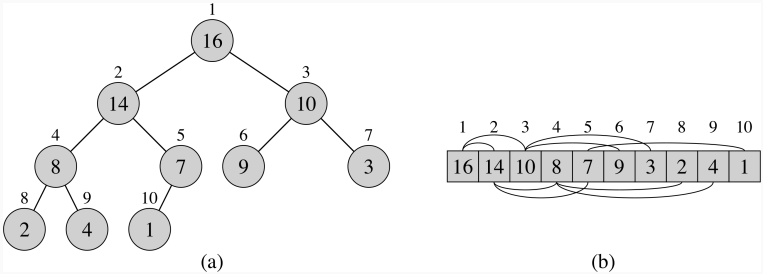
Antes

Tentamos diminuir o tempo para inserir em um arranjo ordenado (parece muito rígido).

Agora

Vamos tentar diminuir o tempo para selecionar o valor mínimo de um arranjo (parece mais flexível).

Um **heap** (binário) é um arranjo que pode ser visto como uma árvore binária quase completa:



Cada nó da árvore corresponde ao elemento do arranjo que armazena o valor do nó.

A árvore está preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda.

Note que nesse exemplo o arranjo é indexado a partir de 1!

Como o heap pode ser visto como árvore, ele também tem uma altura, que é $O(\lg n)$.

É essa característica que permite que as operações em um heap sejam eficientes.

Para arranjos indexados a partir de 0, a raiz do heap está na posição 0. Além disso, para cada nó no índice i, os índices do pai e dos filhos à direita e à esquerda podem ser calculados da seguinte forma:

$\text{PAI}(i) = \lfloor (i-1)/2 \rfloor$ para $i \neq 0$

$\text{ESQ}(i) = 2i + 1$

$\text{DIR}(i) = 2i + 2$

Em um **heap máximo** armazenado em um arranjo A, para cada nó i diferente da raiz

$$A[\text{PAI}(i)] \geq A[i]$$

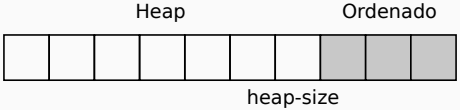
Em um **heap mínimo** armazenado em um arranjo A, para cada nó i diferente da raiz

$$A[\text{PAI}(i)] \leq A[i]$$

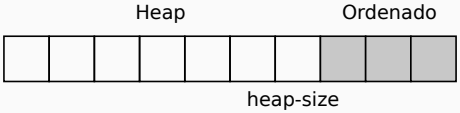
Em um heap máximo, onde está o maior elemento? Na raiz.

Em um heap mínimo, onde está o menor elemento? Na raiz.

Como utilizar um heap máximo em um processo de ordenação incremental?



- Mantemos a porção ordenada no final do arranjo;
- E o heap na porção inicial.



Como selecionar o próximo elemento?

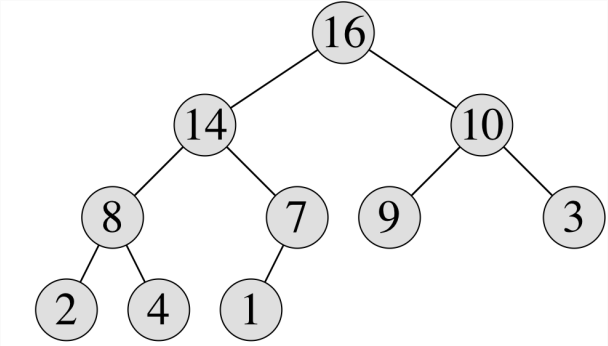
- Pegamos o maior elemento do heap.
- Qual é o custo? $O(1)$.

Como estender o subarranjo ordenado?

- Trocando de posição o maior elemento com o último do heap e consertando o heap.
- Qual é o custo? $O(\lg(\text{heap-size}))$ – veremos isso a seguir.

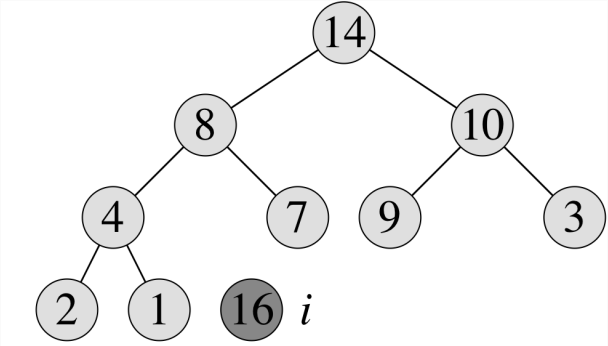
Este algoritmo é conhecido como **ordenação por heap** (*heap sort*).

Exemplo da ordenação por heap



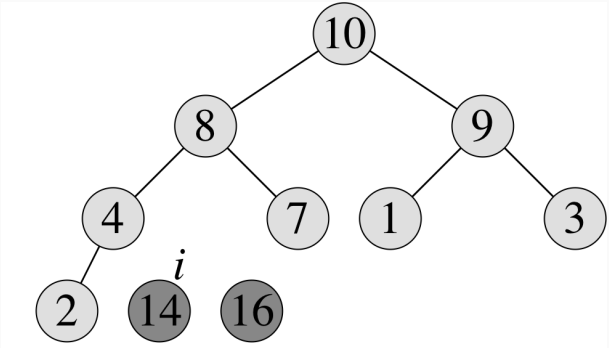

```
| 16 | 14 | 10 |  8 |  7 |  9 |  3 |  2 |  4 |  1 |
\_____________________ heap _____________________/
```

Exemplo da ordenação por heap

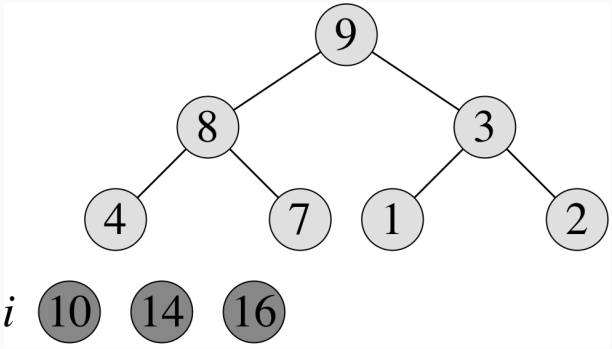



```
| 14 |  8 | 10 |  4 |  7 |  9 |  3 |  2 |  1 | 16 |
\_________________ heap _____________________/
```

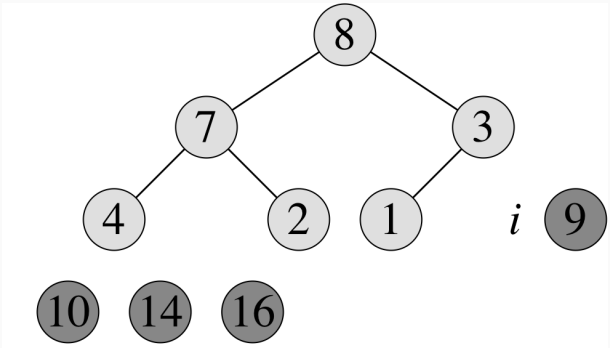
Exemplo da ordenação por heap



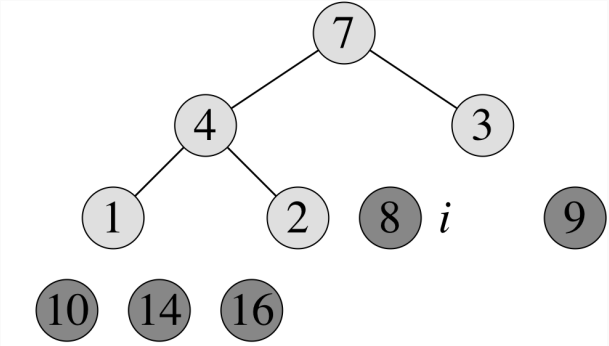

```
| 10 |  8 |  9 |  4 |  7 |  1 |  3 |  2 | 14 | 16 |
\_________________ heap _________________/
```



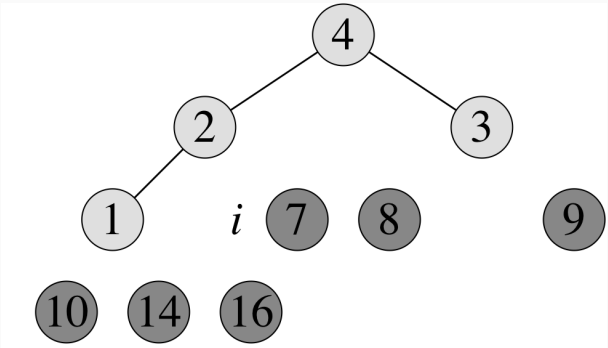

```
|  9 |  8 |  3 |  4 |  7 |  1 |  2 | 10 | 14 | 16 |
\_______________ heap _______________/
```



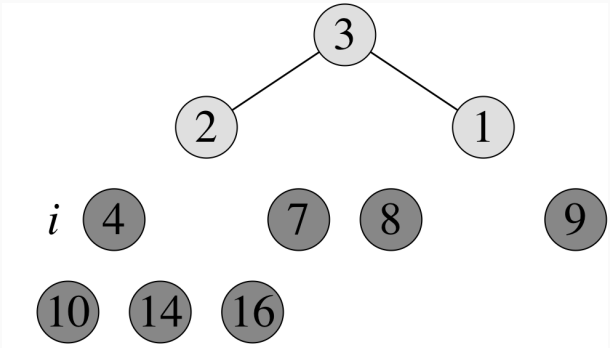

```
|  8 |  7 |  3 |  4 |  2 |  1 |  9 | 10 | 14 | 16 |
\___________ heap _____________/
```



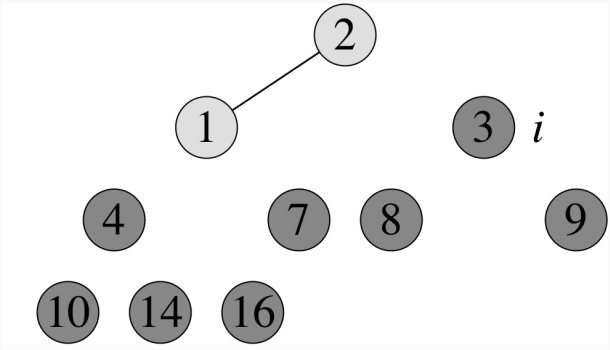

```
|  7 |  4 |  3 |  1 |  2 |  8 |  9 | 10 | 14 | 16 |
\_________ heap _________/
```



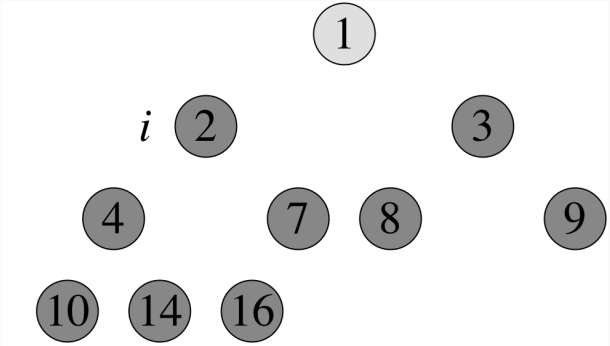

```
|  4 |  2 |  3 |  1 |  7 |  8 |  9 | 10 | 14 | 16 |
\______ heap _______/
```




```
|  3 |  2 |  1 |  4 |  7 |  8 |  9 | 10 | 14 | 16 |
\____ heap ____/
```




```
|  2 |  1 |  3 |  4 |  7 |  8 |  9 | 10 | 14 | 16 |
\_ heap __/
```



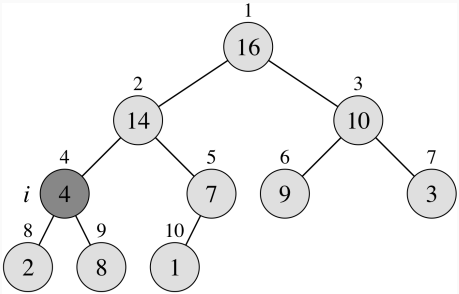

```
|  1 |  2 |  3 |  4 |  7 |  8 |  9 | 10 | 14 | 16 |
\heap/
```

Operações com heap

Que operações precisamos para implementar a ordenação por heap?

- Inicialização do heap
- Consertar o heap

Consertando um heap



Seja A um arranjo que armazena um heap máximo.

Considerando que o elemento da posição i foi alterado, como podemos verificar se a propriedade do heap se mantém, e, caso contrário, como podemos "consertar" o heap?

Verificamos se $A[i]$ é menor que algum dos dois filhos, se sim, trocamos $A[i]$ de lugar com o maior filho, depois executamos o processo recursivamente para o filho que foi trocado.

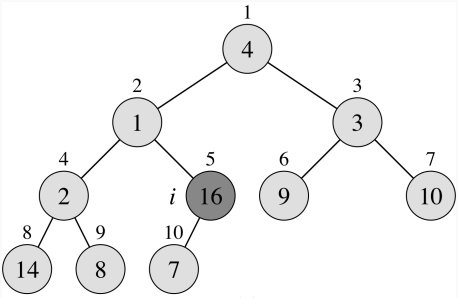
Consertando um heap

Projete uma função que receba como parâmetro um arranjo *A*, a quantidade de elementos *n* de *A* que estão sendo usados e um índice *i*, onde os elementos `esq(i)` e `dir(i)` são raízes de heap máximo, e "conserte" o arranjo, se necessário, para que a árvore com raiz *i* seja um heap máximo.

```python
def conserta_heap(A: list[int], n: int, i: int):
    assert i < n <= len(A)
    # Encontra o índice do maior entre
    # A[i], A[esq(i)] e A[dir(i)]
    fesq = esq(i)
    fdir = dir(i)
    imax = i
    if fesq < n and A[fesq] > A[imax]:
        imax = fesq
    if fdir < n and A[fdir] > A[imax]:
        imax = fdir
    # Se o maior não é A[i], ajusta e repete
    # o processo.
    if imax != i:
        A[i], A[imax] = A[imax], A[i]
        conserta_heap(A, n, imax)
```

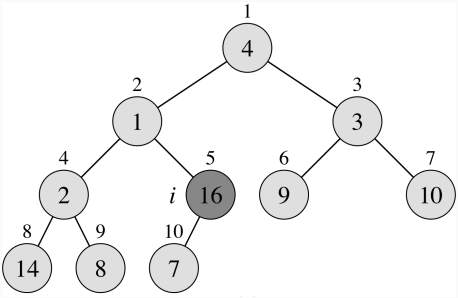
Qual é a complexidade de tempo? $O(h)$, onde h é a altura do heap, ou seja, $O(\lg n)$.

Como construir um heap? Vamos começar com o que está certo e ir “consertando” até que todo o heap fique certo.

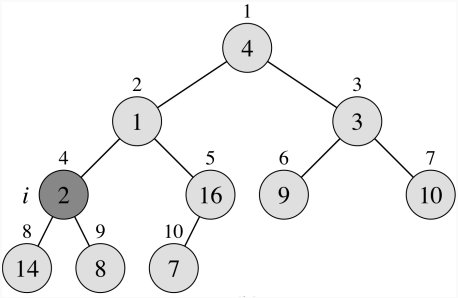


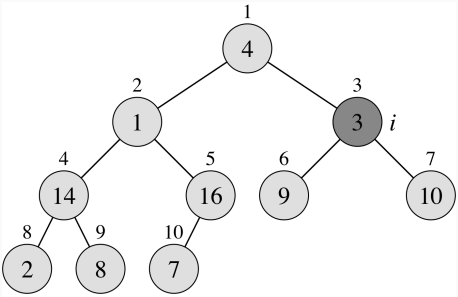
Dado um arranjo qualquer, que vamos transformar em um heap, quais elementos sabemos que são raízes de heaps válidos? As folhas. Note que em um heap o número de folhas nunca é menor do que o número de nós internos.

Construindo um heap

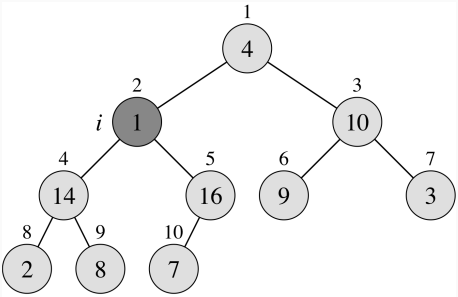


Construindo um heap

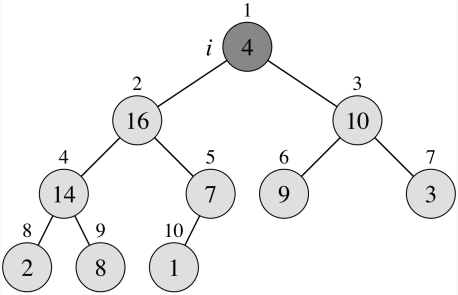



1
4
2
1
3
3
i
4
14
5
16
6
9
7
10
8
2
9
8
10
7

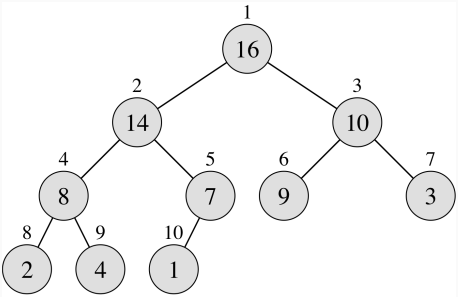
Construindo um heap



Construindo um heap



Construindo um heap



Construindo um heap

Projete uma função que receba como parâmetro um arranjo A, e rearranje os elementos de A para formar um heap máximo.

```python
def inicializa_heap(A: list[int]):
    for i in reversed(range(len(A) // 2)):
        conserta_heap(A, len(A), i)
```

Qual é a complexidade de tempo?

- Limite simples: a função é `conserta_heap` tem tempo $O(\lg n)$ e é chamada $n/2$ vezes, portanto, $O(n \lg n)$;
- Limite estrito: $O(n)$ – discutido em sala.

Implementação ordenação por heap

Projete uma função que implemente a ordenação por heap.

```python
def ordena_heap(lst: list[int]):
    inicializa_heap(lst)
    for i in reversed(range(1, len(lst))):
        # Troca o maior do heap com
        # o elemento da última posição do heap
        lst[0], lst[i] = lst[i], lst[0]
        # Conserta a raiz do heap
        conserta_heap(lst, i, 0)
```

Qual é a complexidade de tempo?

- `inicializa_heap`: $O(n)$
- `conserta_heap`: $\sum_{i=1}^{n-1} \lg(i) = O(n \lg n)$
- Total: $O(n \lg n)$

A implementação é in-loco? Sim (se `conserta_heap` não for recursiva)

A implementação é estável? Não.

Comparação entre os algoritmos de ordenação incrementais

Algoritmo	Estável?	Local?	Melhor	Médio	Pior
Inserção	Sim	Sim	$O(n)$	$O(n^2)$	$O(n^2)$
Seleção	Não	Sim	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heap	Não	Sim	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$

Fila de prioridades

Uma **fila de prioridades** é um TAD que armazena itens associados com prioridades e suas duas principais operações são:

- Inserir um item com uma determinada prioridade;
- Remover o item com maior prioridade.

Como implementar uma fila de prioridades?

- Usando uma ABB auto-balanceada; (inserção é remoção $O(\lg n)$)
- Usando um heap; (inserção é remoção $O(\lg n)$)

O heap é mais interessante pois é implementado com arranjo e por isso consome menos memória.

Referências

Thomas H. Cormen et al. Introduction to Algorithms. 3rd edition. Capítulos 6, 7 e 8.