PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

(baseado nas notas de aula do prof. Túlio A. M. Toffolo)
Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

Licença

Filas de Prioridade

Descrição

São uma abstração de dados em que a **chave** de cada **elemento** reflete sua habilidade relativa de abandonar o conjunto de elementos rapidamente.

Aplicações:

- Sistemas operacionais usam filas de prioridades, nas quais as chaves representam o tempo em que eventos devem ocorrer.
- Métodos numéricos iterativos são baseados na seleção repetida de um elemento com maior (menor) valor.
- Sistemas de gerência de memória usam a técnica de substituir a página menos utilizada na memória principal por uma nova página.

Operações

- Construir uma fila de prioridades a partir de um conjunto com n elementos.
- Informar qual é o elemento do conjunto com maior prioridade.
- Remover o elemento de maior prioridade.
- Inserir um novo elemento.
- Alterar o valor da chave do elemento i para um novo valor que indique maior prioridade que a atual.

Filas de Prioridade

Operações

- Substituir o elemento de maior prioridade por um novo elemento, a não ser que o novo elemento tenha maior prioridade.
- Alterar a prioridade de um elemento.
- Remover um elemento qualquer.
- Agrupar duas filas de prioridades em uma única.

Complexidade: Lista linear ordenada

- Construir é $O(nlogn)$.
- Inserir é $O(n)$.
- Remover é $O(1)$.
- Alterar é $O(n)$.

Complexidade: Lista linear não ordenada

- Construir é $O(n)$.
- Inserir é $O(1)$.
- Remover é $O(n)$.
- Alterar é $O(n)$.

Heap

A melhor representação de uma fila de prioridade é por meio de uma estrutura de dados chamada **heap**:

- Neste caso, construir um heap tem custo $O(n)$.
- Operações de inserção, remoção, substituição e alteração possuem custo $O(logn)$.

Para implementar a operação agrupar de forma eficiente e ainda preservar um custo logarítmico para as demais operações, é necessário utilizar estruturas de dados mais sofisticadas, tais como árvores binomiais (Vuillemin, 1978).

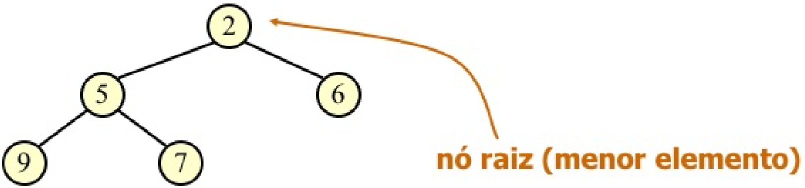
Heap

Um **heap** é uma árvore binária em que a chave de um nó filho é sempre:

1. menor ou igual do que a chave do nó pai, em um MaxHeap.
2. maior ou igual do que a chave do nó pai, em um MinHeap.

ou seja:

1. chave(v) $\leq$ chave(pai(v)) em um MaxHeap.
2. chave(v) $\geq$ chave(pai(v)) em um MinHeap.



Filas de Prioridade

Heap - Características

- Árvore binária quase completa.
- O primeiro nó é chamado raiz.
- Os nós são numerados de 0 a n-1 ou de 1 a n, dependendo do autor.

Nó raiz igual a zero

- Os nós $2k$+1 e $2k$+2 são os filhos à esquerda e à direita do nó k, para $0 < k \leq n/2$.
- O nó (k-1)/2 é o pai do nó k, para 0 < k < n.

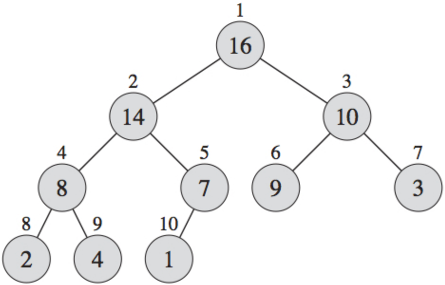
Nó raiz igual a um

- Os nós $2k$ e $2k$+1 são os filhos à esquerda e à direita do nó k, para $1 < k \leq n/2$.
- O nó (k)/2 é o pai do nó k, para 1 < $k \leq n$.

Heap

As chaves na árvore satisfazem a condição do [Max, Min]*Heap*:

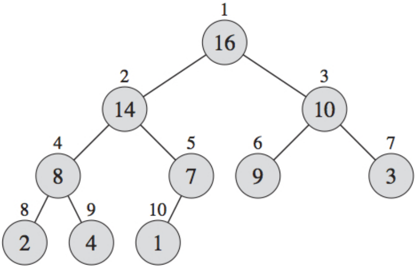
- A chave em cada nó é maior do que as chaves em seus filhos (MaxHeap).
- A chave no nó raiz é a maior chave do conjunto (MaxHeap).
- Uma árvore binária quase completa pode ser representada por um arranjo, conforme abaixo (note que a primeira posição é 1 na figura).

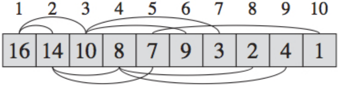


Heap

Esta representação é extremamente compacta:

- Permite caminhar pelos nós da árvore facilmente.
- Os filhos de um nó i estão nas posições 2i e 2i+1.
- O pai de um nó i está na posição (i) / 2.
- Na representação do *heap* em um arranjo, a menor (ou maior) chave está sempre na posição 1.





Heap

Um algoritmo elegante para construir o *heap* foi proposto por Floyd em 1964.

O algoritmo não necessita de nenhuma memória auxiliar.

Dado um vetor A[0], A[1], ..., A[n-1]:

- Os elementos A[$n/2$], A[$n/2+1$], ..., A[n-1] formam um *heap* válido pois são nós folhas (nós que não possuem filhos).
- Neste intervalo não existem dois índices i e j tais que $j = 2i{+}1$ ou $j = 2i{+}2$.
- O princípio do algoritmo é então incluir os elementos A[$n/2$ -1], A[$n/2$ -2], ..., A[0] no *heap* um a um, e, se necessário, reconstruir a propriedade do *heap* trocando os nós de lugar.

Exemplo 1

Vejamos a construção de um *MinHeap* a partir de um arranjo [9, 5, 6, 8, 3, 2, 7].

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

MinHeap - Construção

Os elementos de A[3] a A[6] formam um *heap* válido.

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

MinHeap - Construção

- O *heap* é estendido para a esquerda ($Esq = 2$), englobando o elemento 6 (A[2]), pai dos elementos A[5] e A[6].
- A condição de *heap* é violada:
  - O *heap* é refeito trocando os elementos A[2] e A[5].

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

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

MinHeap - Construção

- O *heap* é estendido para a esquerda ($Esq = 1$), englobando o elemento 5 (A[1]).
- A condição de *heap* é violada:
  - O *heap* é refeito trocando os elementos A[1] e A[4].

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

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

MinHeap - Construção

- O *heap* é estendido para a esquerda ($Esq = 0$), englobando o elemento 9 (A[0]).
- A condição de *heap* é violada:
  - O *heap* é refeito trocando os elementos A[0] e A[2].

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

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

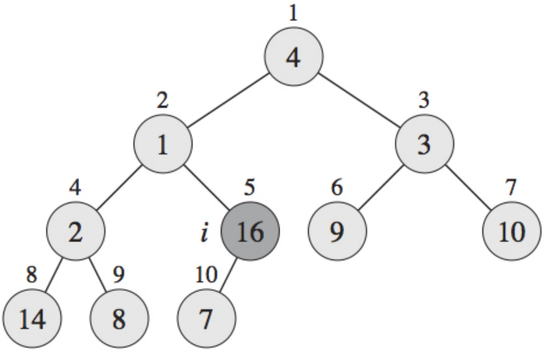
MinHeap - Construção

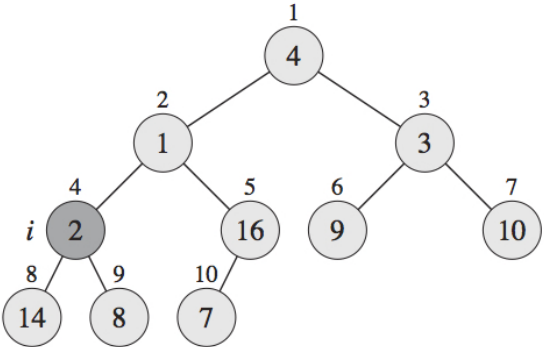
- Como a condição ainda está sendo violada:
  - O *heap* é refeito trocando os elementos A[2] e A[5].
- Como resultado, o *heap* foi construído.

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

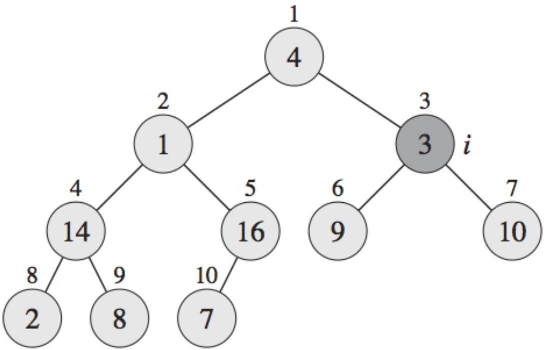
Exemplo 2 - *MaxHeap*

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

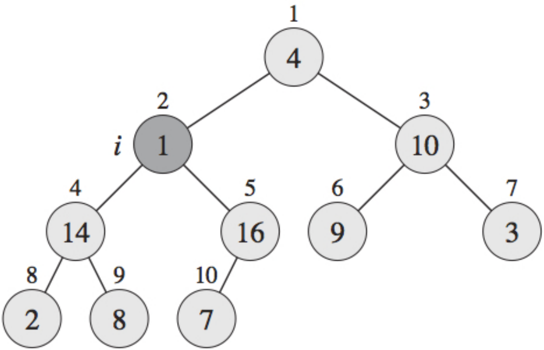




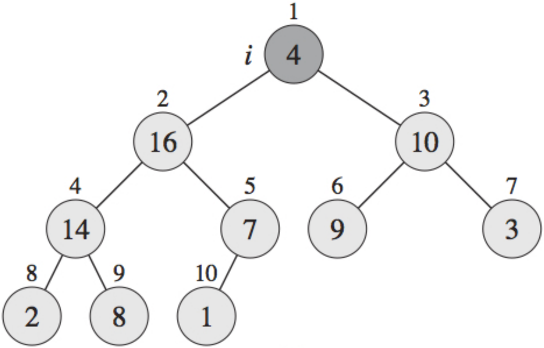
O nó A[4] viola a propriedade do *heap*, já que o pai é menor do que os filhos.



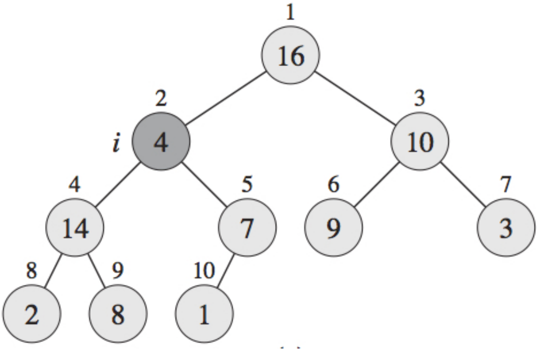
O nó A[3] viola a propriedade do *heap*, já que o pai é menor do que os filhos.



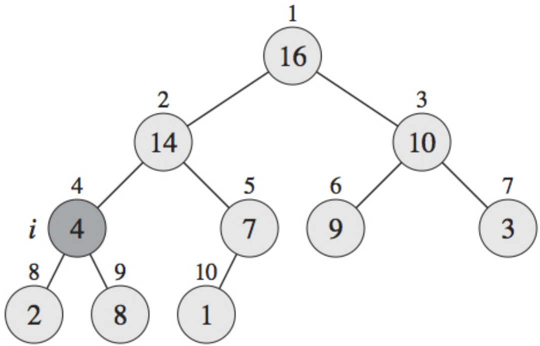
O nó A[2] viola a propriedade do *heap*, já que o pai é menor do que os filhos.



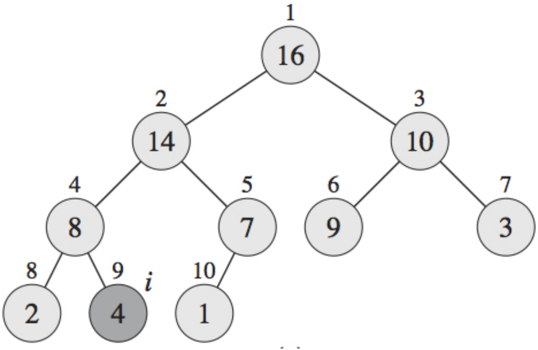
O nó A[1] viola a propriedade do *heap*, já que o pai é menor do que os filhos.



Ao trocar A[1] por A[2], a propriedade do *heap* é novamente violada.



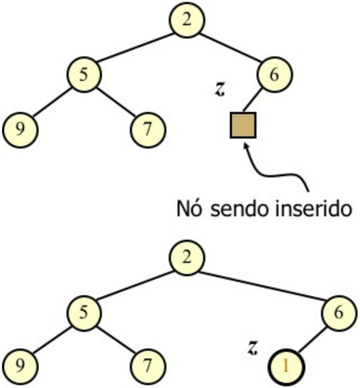
Ao trocar A[2] por A[4], a propriedade do *heap* é novamente violada.



A troca de A[4] por A[9] não gera mais violações e temos um *heap* válido.

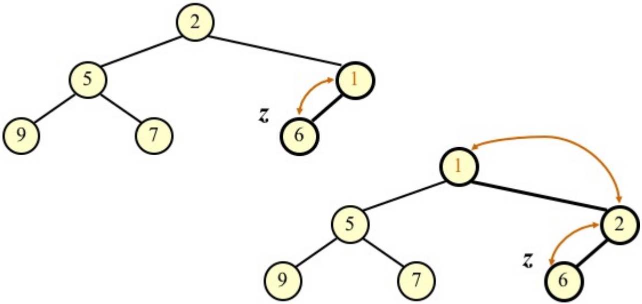
MinHeap - Inserção

É necessário comparar o nó inserido com os pais e refazer enquanto ele for menor que o pai ou até que ele seja o nó raiz.



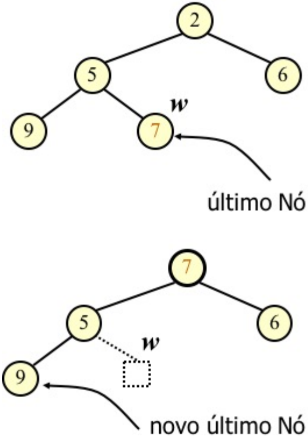
MinHeap - Inserção

Na pior das hipóteses o custo de uma inserção será $O(logn)$, equivalente à altura da árvore.



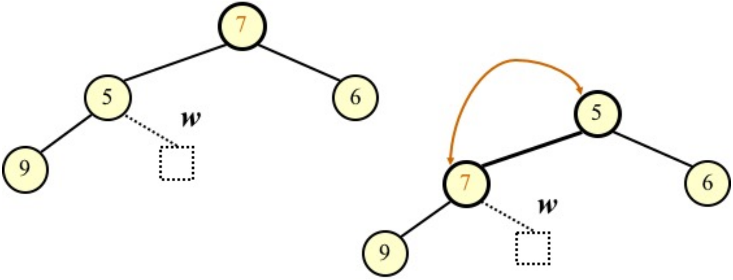
MinHeap - Remoção

É necessário trocar o nó raiz pelo último nó do *heap* e remover o último nó.



MinHeap - Remoção

É necessário refazer o *heap*! Na pior das hipóteses o custo de uma remoção será $O(logn)$, equivalente à altura da árvore



Construção

```
void heapConstroi(Telemento *v, int n) {
    int esq;
    esq = (n / 2) - 1; // esq = antecessor do primeiro
        no folha do heap
    while (esq >= 0) {
        heapRefaz(v, esq, n-1);
        esq--;
    }
}
```

Reconstrução

```c
void heapRefaz(Telemento *v, int esq, int dir) {
    int i = esq;
    int j = i*2 + 1; // j = primeiro filho de i

    Telemento aux = v[i]; // aux = no i (pai de j)

    while (j <= dir) {
        if (j < dir && v[j].chave < v[j+1].chave)
            j++; // j recebe o outro filho de i
        if (aux.chave >= v[j].chave)
            break; // o heap foi refeito corretamente
        v[i] = v[j];
        i = j;
        j = i*2 + 1; // j = primeiro filho de i
    }
    v[i] = aux;
}
```

Exercício

Dada a sequência de números 3 4 9 2 5 1 8, construa um *max heap*, passo-a-passo.

Em seguida, remova os 3 elementos prioritários.

Dúvidas?