Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Prática de Algoritmos e Estruturas de Dados II • DIM0111.1
◁ Implementação de uma Fila de Prioridade com um Binary Heap ▷
27 de setembro de 2017

Apresentação

O objetivo deste trabalho é implementar a estrutura de dados *fila de prioridade* (ou *priority queue* em Inglês) que permite acesso apenas ao menor/maior item (ou chave) armazenado. A fila de prioridade será implementada através de um *binary heap*, cujo armazenamento será feito via *representação implícita*[1].

1 Introdução

Filas de prioridades são utilizadas em diversas áreas de computação, em especial no escalonamento de processos em sistemas operacionais. Considere, por exemplo, um sistema de gerenciamento de processos[2] de impressão enviados para a uma impressora compartilhada. Em geral, espera-se que as requisições de impressão sejam colocadas em um fila de impressão — porém este comportamento nem sempre é a melhor opção de ação. Por exemplo, uma processo de impressão em particular pode ser muito importante, portanto seria interessante priorizar sua impressão sobre as demais, fazendo com que tal processo fosse impresso assim que a impressora finalizasse a impressão atual — em outras palavras, deveria haver um mecanismo formal de "furar a fila" quando necessário.

Similarmente, considere uma situação na qual a impressora acabou uma impressão e existem vários processos de 1 página e um de 100 páginas esperando por sua vez; neste caso pode ser razoável imprimir primeiramente todos os processos menores, deixando o de 100 páginas por último, mesmo que a impressão de 100 páginas tenha chegado antes dos demais processos de 1 página.

Se atribuirmos a cada processo um número para representar sua prioridade, então o *menor* número (ex.: número de páginas a imprimir) tende a indicar *maior* importância. Portanto queremos ser capazes de prover um mecanismo para acessar o menor ou maior item em uma coleção de itens, e removê-lo da coleção — estas são, respectivamente, as operações de `top` e `pop`.

A *fila de prioridade* é uma estrutura de dados que implementa este comportamento. Figura 1 ilustra as operações básica de uma fila de prioridade.

A determinação se a fila de prioridade deve retornar o maior ou menor item deve ser determinada pela *função de comparação* que precisa ser passada para a estrutura de dados. A função de comparação é quem determina a *ordem total estrita* utilizada pela fila de prioridade na hora de organizar seus elementos internamente.

[1] Uso de arranjo unidimensional.
[2] Também conhecidos como *jobs*.

Figura 1: Modelo de uma fila de prioridade: apenas o elemento mínimo (ou máximo, dependendo da implementação) é acessível.

2 Binary Heap

A implementação sugerida para a fila de prioridade é o *binary heap* (BH). Um *binary heap* suporta inserção de novos itens e remoção do menor item em tempo logarítmico no pior caso: $O(\lg N)$. Essa implementação não utiliza apontadores[3] para definir sua estrutura, sendo, portanto, mais simples de implementar.

Um BH tem duas propriedades básicas: *estrutural* e de *ordem*. Em casos especiais, após certas operações, essas propriedades básicas podem ser violadas. Porém, antes de qualquer tipo de consulta, as propriedades básicas devem ser restauradas, de forma a garantir o correto comportamento do BH.

2.1 Propriedade Estrutural do Binary Heap

Um BH é uma árvore binária *completa*[4]. É esta característica que permite que uma implementação por arranjo unidimensional possa ser utilizada ao invés da implementação tradicional encadeada através de apontadores.

O arranjo é preenchido na ordem de percorrimento *em-nível*, sendo a raiz armazenada na posição 1 ao invés de 0 (zero). A posição 0 do arranjo será utilziada como elemento *sentinela*, necessário para facilitar a implementação de algumas operações da fila de prioridade.

As propriedades estruturais são:

1. Um elemento do BH armazenado na posição i do arranjo tem um filho à esquerda em $2i$; porém, se $2i$ ultrapassar o número de elementos do BH isso significa que o filho à esquerda de i não existe.
2. Similarmente, o filho à direita pode ser encontrado imediatamente após o filho à esquerda em $2i + 1$. Novamente se $2i + 1$ ultrapassar o número de elementos armazenados do BH então i não possui filho à sua direita.
3. Finalmente, o pai de um certo nó i está na posição $\lfloor i/2 \rfloor$; dessa forma todos os nós têm pai, exceto o nó raiz (por isso a posição 0 não é utilizada).

O uso de arranjo para representar um árvore binária é chamado de *representação implícita*. A Figura 2 apresenta uma árvore binária (ou BH) e sua correspondente representação na forma de um arranjo unidimensional.

[3] Exceto em alocação dinâmica.

[4] Árvore binária completamente preenchida, sendo uma possível exceção o nível das folhas, que deve ser preenchido *em nível*, da esquerda para a direita.

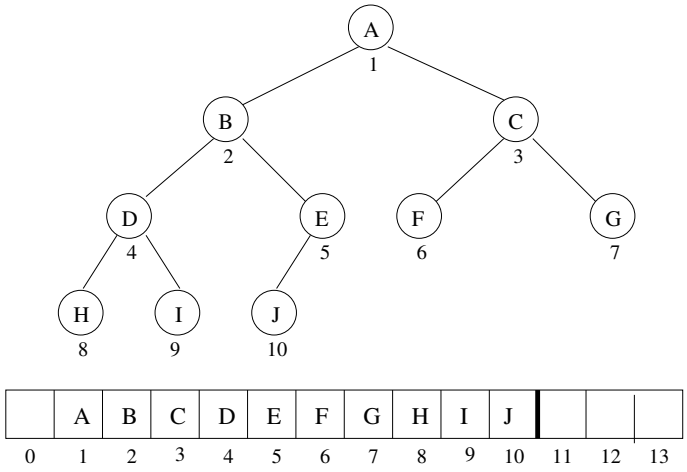

Figura 2: Exemplo de um *Binary Heap* e sua representação implícita. Note que a árvore *não* está completa.

2.2 Propriedade de Ordem do Binary Heap

A propriedade de ordenação do *heap* diz que:

> *Em um* heap, *para cada nó X com pai P, a chave em P é menor ou igual a chave armazenada em X.*

Um *binary heap* deve manter essa propriedade sempre verdadeira. Note que como a raiz da árvore não tem pai então é possível atribui $-\infty$ na posição zero do vetor. Porém essa posição deverá ser usada na inserção para guardar um *valor sentinela* para representar a condição de parada quando subirmos pela árvore — até, no máximo, a raiz — procurando o local apropriado para a nova chave.

Note que o termo *menor* utilizado na definição da propridade de ordem significa o menor item de acordo com a ordem estrita total estabelecida. Se desejarmos um fila de prioridade que retorna o maior elemento basta utilizarmos a relação de ordem apropriada. Por exemplo, se considerarmos que desejamos armazenar números inteiros, uma possível função de comparação para estabalecer uma fila de prioridade de mínimo seria

```
bool compare( int a, int b ) { return a < b; }
```

contudo, se desejarmos uma fila de prioridade de máximo, então a função de comparação deveria ser:

```
bool compare( int a, int b ) { return b < a; }
```

3 Implementação

A seguir estão descritos alguns dos métodos básicos mais importante da classe `PQ` que implementa uma fila de prioridade. Uma descrição mais detalhadas dos métodos da classe `PQ` pode ser encontrada aqui.

3.1 Inserção: `push`

Esse método insere um elemento na BH de forma a preservar sua propriedade de ordem. Inicialmente, o método simplesmente insere o novo elemento no final do vetor (última folha disponível mais a direita), posição denominada de `hole`. Se a inserção do novo elemento não violar a propriedade de ordem do BH então o processo de inserção está encerrado; caso contrário, é necessário realizar um movimento para cima (método `move_up`) que sucessivamente troca o valor x armazenado da posição `hole` com o de seu pai, até que seja encontrado um pai[5] que seja menor que x e, por conseguinte, preserve a propriedade de ordem do BH (i.e. nenhum filho pode ser menor do que o seu pai). Esse processo é ilustrado na sequência da Figura 3.

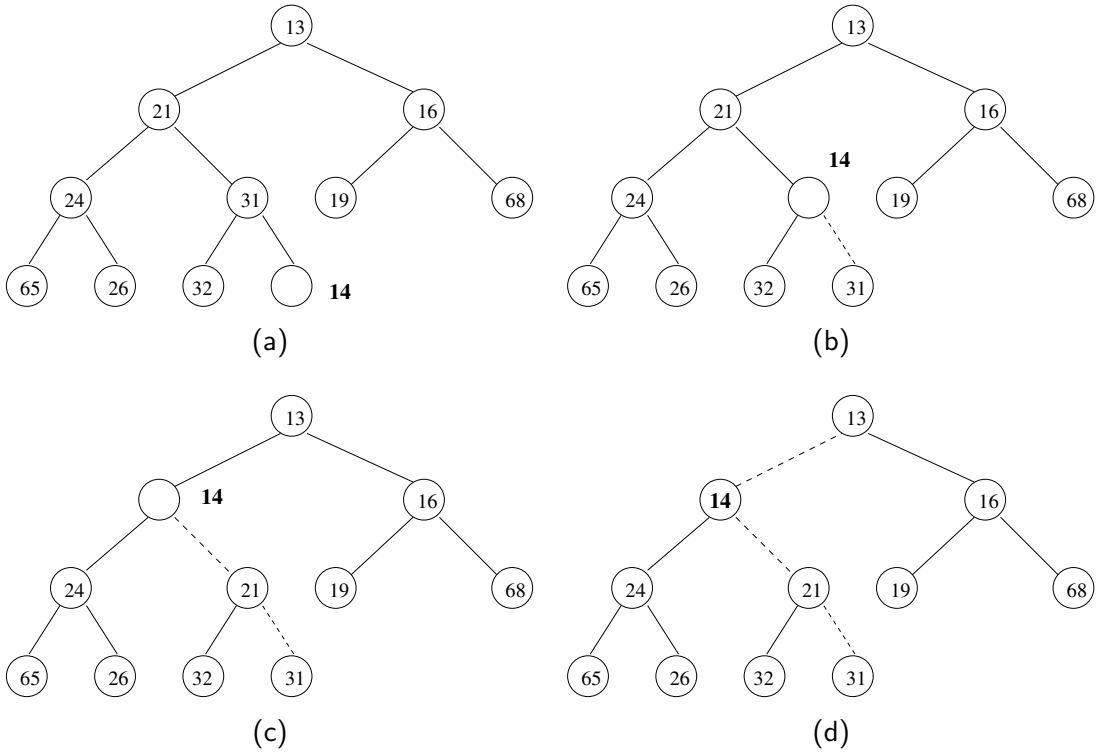

Figura 3: Inserção da chave 14 seguida do processo de movimento para cima. (a): O *hole* foi criado no final da árvore mas não é a posição correta para o 14; (b) a chave 14 sobe um nível e o pai do *hole* desce um nível, porém ainda não é o seus local correto; (c) a chave 14 sobe mais um nível, e; (d) por fim a chave 14 é confirmada em uma posição que não viola a propriedade de ordem do *binary heap*.

Note que o método `push` deve primeiramente verificar se a árvore suporta a inserção de um novo nó (existem folhas disponíveis). Caso contrário o método precisa aumentar a árvore, dobrando seu tamanho e, ao mesmo tempo, preservando os elemento já armazenados.

O método `toss` é semelhante ao método `push` pois também insere um elemento no BH. A diferença entre os dois é que o método `toss` **não faz** o ajuste de movimento para cima (`move_up`) se a ordem do BH for violada. O método `toss` simplesmente insere o novo elemento. Se a

[5] Isto é, no máximo até a raiz.

propriedade de ordem foi violada, o método atribui falso para o flag `m_sorted`. A vantagem desse método é um melhor desempenho (se comparado ao `push`), pois é possível inserir vários nós sem nos preocuparmos com a ordem. Assim, a re-ordenação é executada apenas quando uma operação de retirada (`pop`) ou de consulta (`top`) for solicitada pelo código cliente — não podemos esquecer de "ligar" (i.e. atribuir verdadeiro para) o flag `m_sorted` após a ordenação ter sido realizada.

3.2 Busca do Menor Elemento: `top`

A busca do menor elemento em uma BH é fácil, pois a menor chave está armazenada na primeira posição (válida) do vetor interno: `A[1]`. O problema ocorre quando essa chave é retirada, pois nesse caso um ajuste (`move_down`) tem que ser feito na árvore de forma a restabelecer a propriedade de estrutura e de ordem.

3.3 Retirada do Menor Elemento: `pop`

Esse método deve ser desenvolvido de forma similar à inserção. Nesse caso retira-se o menor elemento (raiz do BH) e em seguida tenta-se atribuir o último elemento da árvore à posição raiz. Esse tentativa raramente dará certo (somente se a árvore tiver apenas dois níveis) o que vai provocar um movimento para baixo (`move_down`) da raiz provisória da árvore. Figura 4 demonstra esse processo quando a chave 13 é removida do BH.

3.4 Re-organizando o Binary Heap: `fix_heap`

Esse método transforma uma árvore binária completa que não obedece a propriedade de ordem do *heap* em um BH correto (i.e. ordenado).

A base desse método é utilizar chamadas recursivas para ordenar suas sub-árvores, o que, no final, irá resultar em uma árvore totalmente ordenada. A rotina recursiva de re-ordenação funcionaria perfeitamente pois ela tem a garantia de que quando aplicarmos o método `move_down(i)` todos os descendentes de i já foram processados pelas suas próprias chamadas recursivas a `move_down`.

Porém a recursão não é necessária devido ao seguinte fato: se invocarmos `move_down` sobre os nós em-nível na ordem reversa, então no momento que `move_down(i)` for processado todos os descendentes do nó i terão sido processados por uma chamada anterior a `move_down`. Isso faz com que o algoritmo `fix_heap` seja extremamente simples de implementar. Note que não é necessário aplicar o método sobre os nós folhas, portanto o algoritmo deve começar pelo mais alto nó interno (não-folha) da árvore.

A Figura 5 apresenta um exemplo de uma árvore binária sem ordem que é sucessivamente ajustada pela rotina `fix_heap`.

4 Tarefas

Você deve executar duas tarefas para completar esta lista.

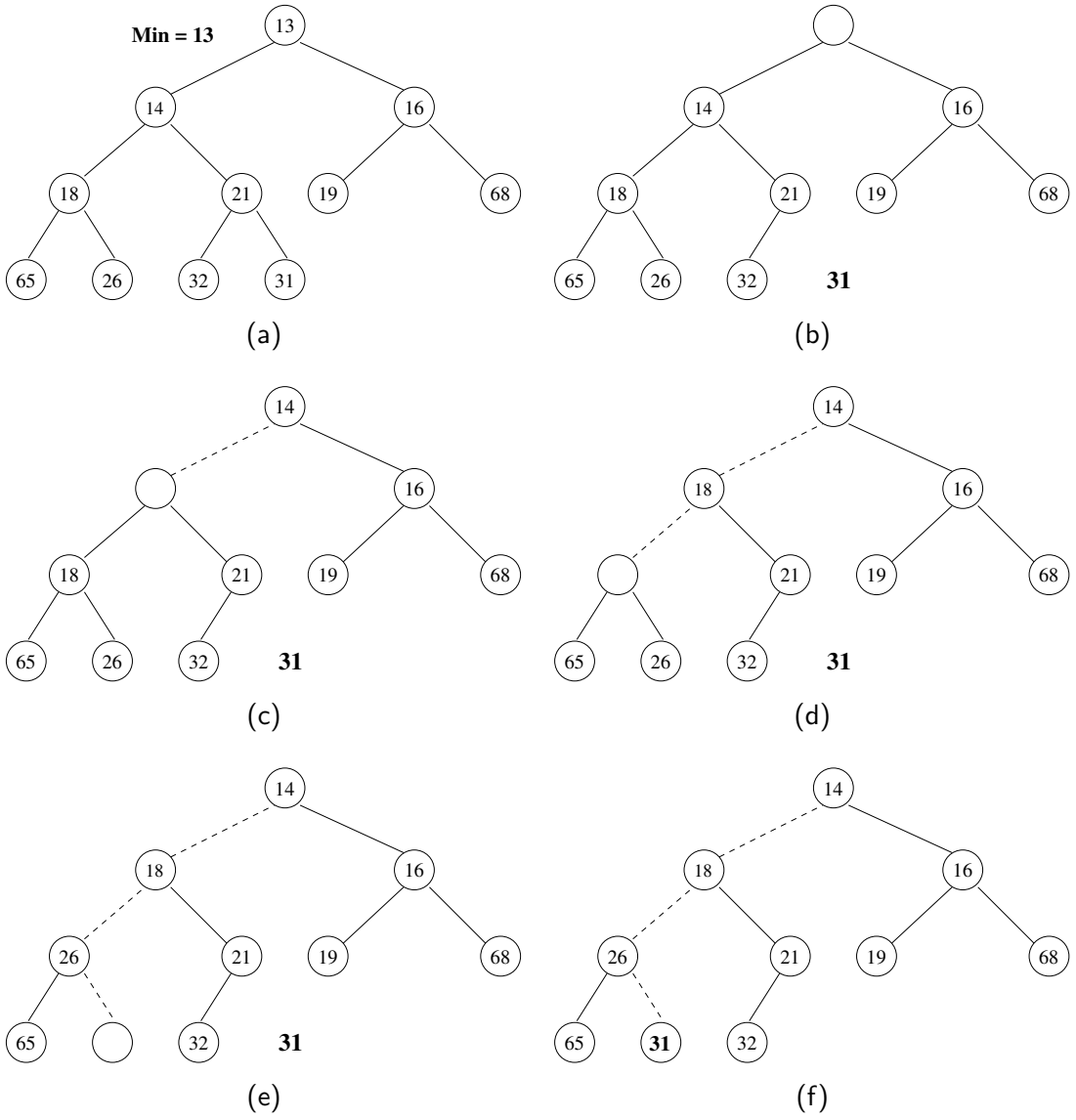

Figura 4: Remoção da menor chave (13), seguida do processo de movimento para baixo (`move_down`) do elemento 31. Figuras de (a) à (f) demonstram o movimento do *hole* para baixo até que a última chave (31) ache sua nova posição. Note como o elemento 31 desce sempre pelo filho com menor chave (você sabe por que?).

- Implementar a classe `PQ`, conforme a interface descrita na Seção 3.
- Implementar o algoritmo *heap sort* de forma que o mesmo realize uma ordenação em ordem não-crescente.

  Um opção é utilizar o seguinte algoritmo:

  1. Criar um `PQ` usando o construtor que recebe uma lista inicializadora como parâmetro;
  2. Enquanto o número de elemento do `PQ` for maior do que 1 faça:
     (a) Troque o último elemento do `PQ` com o primeiro (posição 1 do vetor);

(b) Decremente em 1 o tamanho total do `PQ`;

(c) Realize o `move_down` sobre o novo nó raiz.

5 Avaliação

O trabalho pode ser desenvolvido em duplas, tentando, dentro do possível, dividir as tarefas igualmente entre os componentes. Porém os componentes devem ser capazes de explicar qualquer trecho de código do programa, mesmo que o mesmo tenha sido desenvolvido pelo outro membro da equipe. Após a entrega, poderá ser solicitado às equipes, através de entrevista, que expliquem o funcionamento de seus programas.

∽ FIM ∽

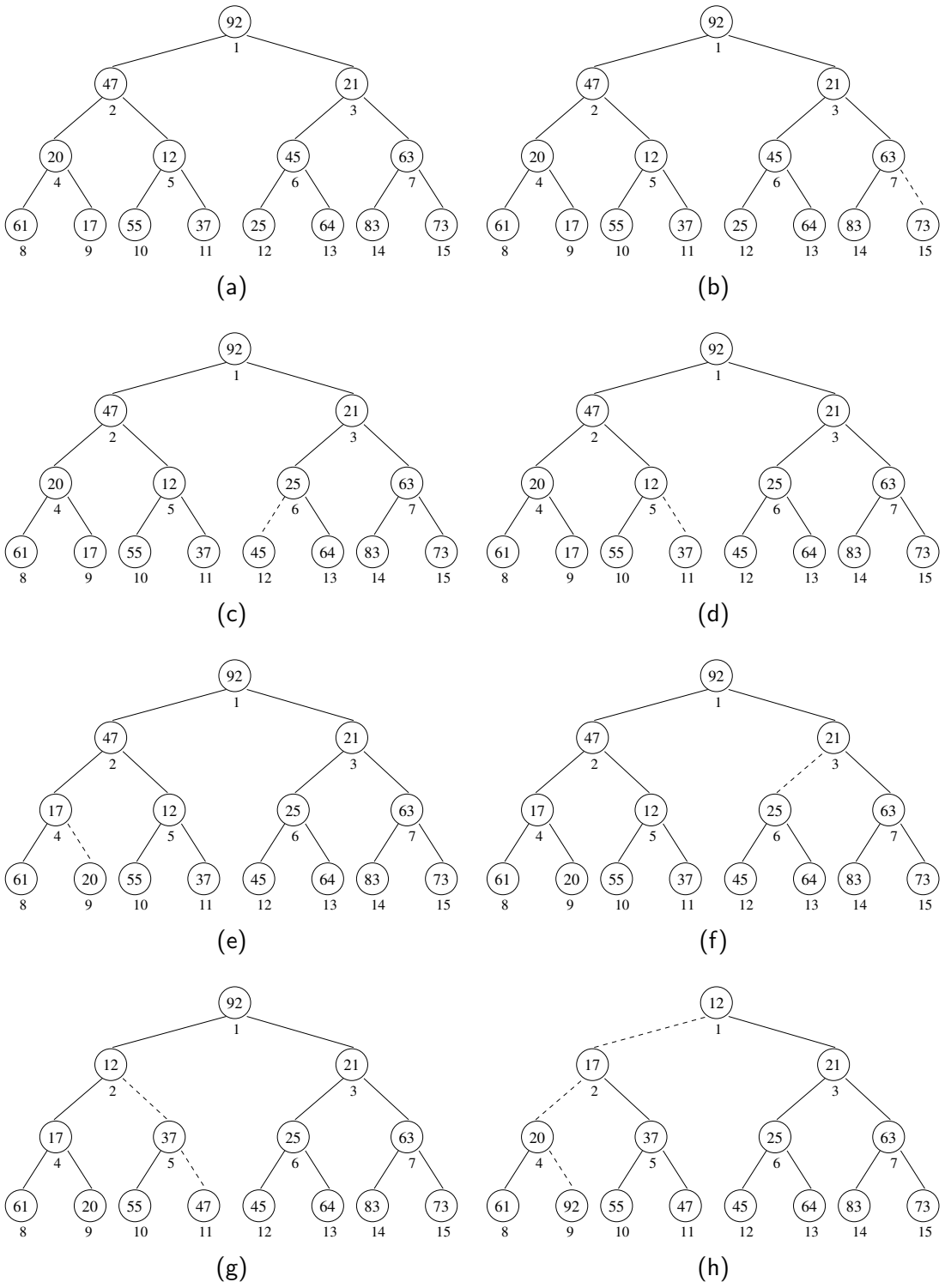

Figura 5: Re-ordenando a árvore com a chamada do `fix_heap`. Essa rotina invoca o método `move_down(i)` sobre os nós não folhas de trás para frente no percorrimento em nível, ou seja, do nó com índice 7 para o nó com índice 1, neste exemplo. Figuras de (a) à (h) representam chamadas sucessivas ao método `move_down(i)`. Perceba que houve troca de chaves na figura (c), `move_down(6)`; figura (e), `move_down(4)`; (g), `move_down(2)`, e; figura (h), `move_down(1)`.