Apontamentos de Estruturas de Dados

Heaps e Filas de Prioridade

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

Última versão: Novembro de 2022

Índice

14. Heaps e Filas de Prioridade

14.1. Introdução

Existem muitas situações, tanto na vida real como em aplicações em que desejamos escolher o próximo "mais importante" de uma coleção de pessoas, tarefas ou objetos. Por exemplo, os médicos numa sala de emergência de um hospital geralmente optam por ver o paciente "mais crítico" de seguida em vez daquele que chegou primeiro. Ao agendar programas para execução num sistema operativo multitarefa podem existir a qualquer momento vários programas (geralmente chamados de jobs) prontos para serem executados. O próximo trabalho selecionado é aquele com a prioridade mais alta. A prioridade é indicada por um valor específico associado à tarefa (e pode ser alterada enquanto a tarefa permanecer na lista de espera).

Quando uma coleção de objetos é organizada por importância ou prioridade damos-lhe o nome de fila de prioridade. Uma estrutura de dados que faça uso de uma fila normal não permite implementar uma fila de prioridade eficientemente porque a pesquisa pelo elemento com prioridade mais alta levará O(n) tempo (além desse problema temos ainda o problema do espaço auxiliar que será necessário para ir armazenado os elementos). Uma lista, ordenada ou não, também irá exigir O(n) de tempo para inserção ou remoção. Uma árvore binária de pesquisa que organiza os elementos por prioridade poderia ser usada, com o total de n inserções e n operações de remoção requerendo O(n log n) de tempo no caso médio. No entanto, há sempre a possibilidade da árvore binária de pesquisa ficar desequilibrada, levando a um mau desempenho. Em vez disso, gostaríamos de encontrar uma estrutura de dados que garantisse um bom desempenho para esta necessidade especial.

Neste capítulo são introduzidas as *heaps* (acervos). Uma *heap* é definida por duas propriedades. Primeiro, é uma árvore binária completa, então as *heaps* são quase sempre implementados através de uma representação de *array*. Segundo, os valores armazenados numa *heap* são parcialmente ordenados. Isso significa que existe uma relação entre o valor

armazenado em qualquer nó e os valores dos seus filhos. Existem duas variantes da *heap*, dependendo da definição desse relacionamento.

Uma ***maxheap*** que tem a propriedade de que cada nó armazena um valor maior ou igual ao valor de qualquer um de seus filhos. Como a raiz tem um valor maior ou igual aos seus filhos, que por sua vez têm valores maiores ou iguais a seus filhos. A raiz armazena o máximo de todos os valores na árvore.

Uma ***minheap*** que tem a propriedade de que cada nó armazena um valor menor ou igual aos seus filhos. Como a raiz tem um valor menor ou igual aos seus filhos, que por sua vez têm valores menores ou iguais aos seus filhos, a raiz armazena o mínimo de todos os valores na árvore.

Observe que não há qualquer relação entre o valor de um nó e o do seu irmão na *minheap* ou na *maxheap*. Por exemplo, é possível que os valores de todos os nós da subárvore esquerda da raiz sejam maiores que os valores de cada nó da subárvore direita. Podemos contrastar árvores binárias de pesquisa e *heaps* pela força das suas relações de ordenação. Uma árvores binárias de pesquisa define uma ordem total nos seus nós em que dadas as posições de quaisquer dois nós na árvore o da "esquerda" (equivalentemente, o que aparece mais cedo numa travessia em-ordem) tem um valor chave menor que o da "direita". Em contraste, uma *heap* implementa uma ordem parcial. Dadas as suas posições, podemos determinar a ordem relativa dos valores de chave de dois nós na *heap* apenas se um for descendente do outro.

Tenha cuidado para não confundir a representação lógica de uma *heap* com sua implementação física através de uma árvore binária completa baseada em *array*. Os dois não são sinónimos porque a visão lógica da *heap* é na verdade uma estrutura em árvore, enquanto a implementação física típica usa um *array*.

14.2. Heap ADT

Podemos então dizer que uma *minheap* é uma árvore binária com duas propriedades

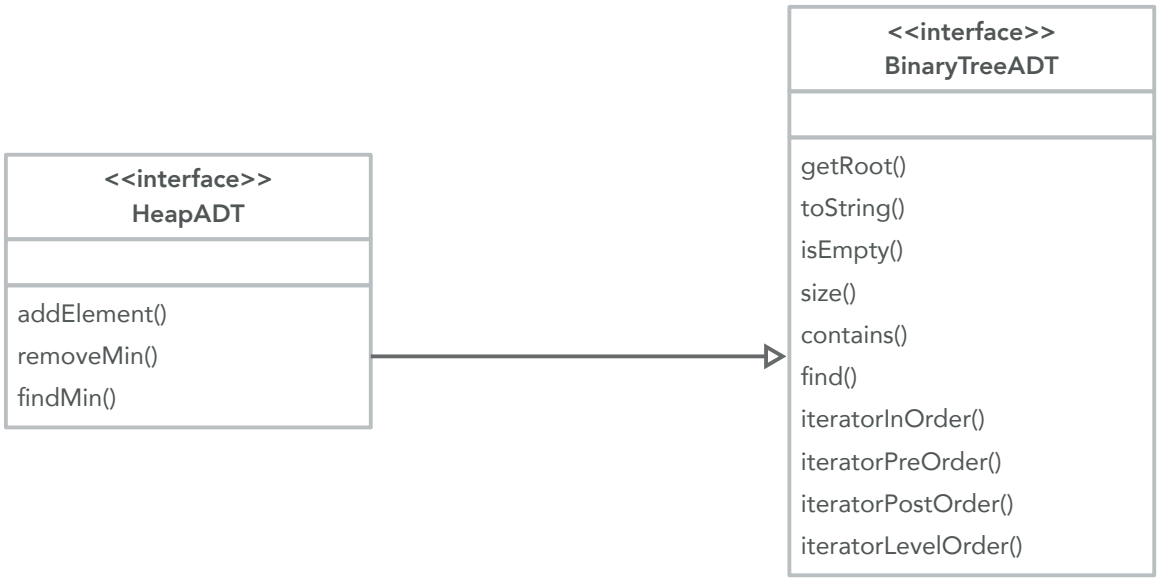
- É uma árvore completa;
- Para cada nó, o nó é menor ou igual ao filho esquerdo e ao filho direito.

Esta definição descreve uma *minheap*. Além das operações herdadas de uma árvore binária, uma *heap* tem as seguintes operações adicionais:

Operação	Descrição
`addElement`	Adiciona um determinado elemento à heap
`removeMin`	Remove o menor elemento na heap
`findMin`	Retorna uma referência do menor elemento na heap

14.3. Interface *Heap*

Uma *heap* é na essência uma árvore binária como uma ordenação fraca em que apenas existe a preocupação de ter o valor mínimo ou máximo (dependendo do tipo de *heap*) na raiz. Assim, podemos definir a interface `HeapADT` como uma extensão da interface `BinaryTreeADT` tal como temos vindo a fazer para outras coleções. Atenção que apesar do nome, a *heap* que está a especificar é uma *minheap* tal como já é referido na seção anterior.



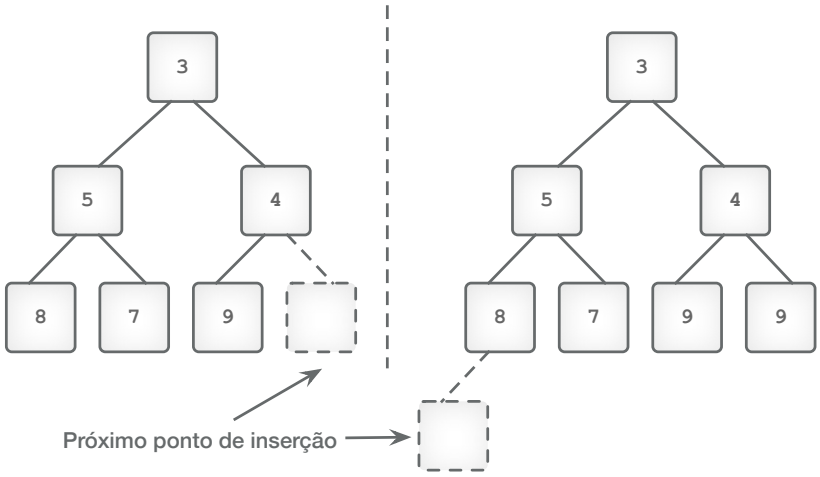

```
1. public interface HeapADT<T> extends BinaryTreeADT<T> {
2.     /**
3.      * Adds the specified object to this heap.
4.      *
```

```
5.      * @param obj  the element to added to this head
6.      */
7.     public void addElement (T obj);
8.
9.     /**
10.     * Removes element with the lowest value from this heap.
11.     *
12.     * @return  the element with the lowest value from this heap
13.     */
14.    public T removeMin();
15.
16.    /**
17.     * Returns a reference to the element with the lowest value in
18.     * this heap.
19.     *
20.     * @return  a reference to the element with the lowest value
21.     * in this heap
22.     */
23.    public T findMin();
24.}
```

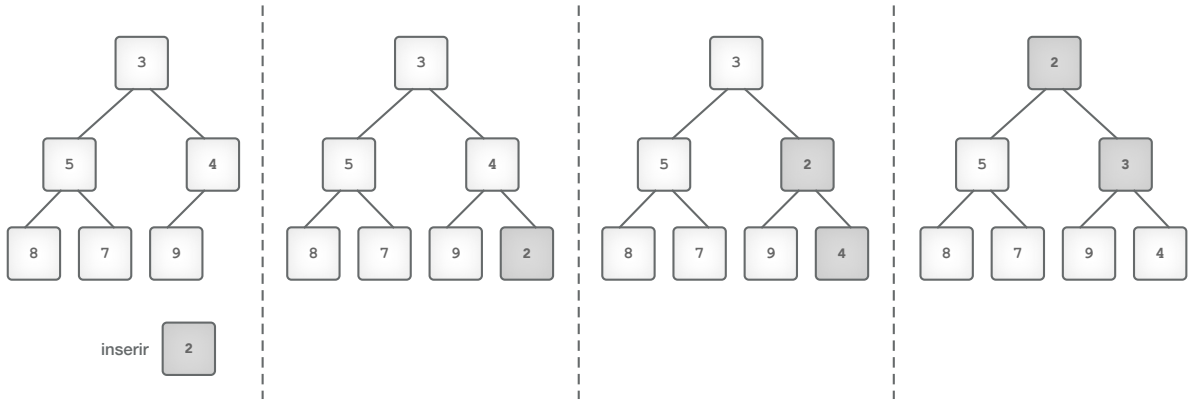
14.4. Adicionar elementos a uma heap

Para adicionarmos elementos precisamos primeiro especificar qual o tipo de *heap* a considerar. Neste caso iremos descrever como são adicionados elementos a uma *minheap*. Quando adicionamos um elemento este irá ser colocado no local apropriado da *heap*. O local adequado é o local que irá manter a integridade da árvore – neste caso *minheap*. Existe apenas um local correto para a inserção de um novo nó:

- Ou a próxima posição livre da esquerda no nível h;
- Ou a primeira posição no nível h+1 se o nível h estiver cheio.



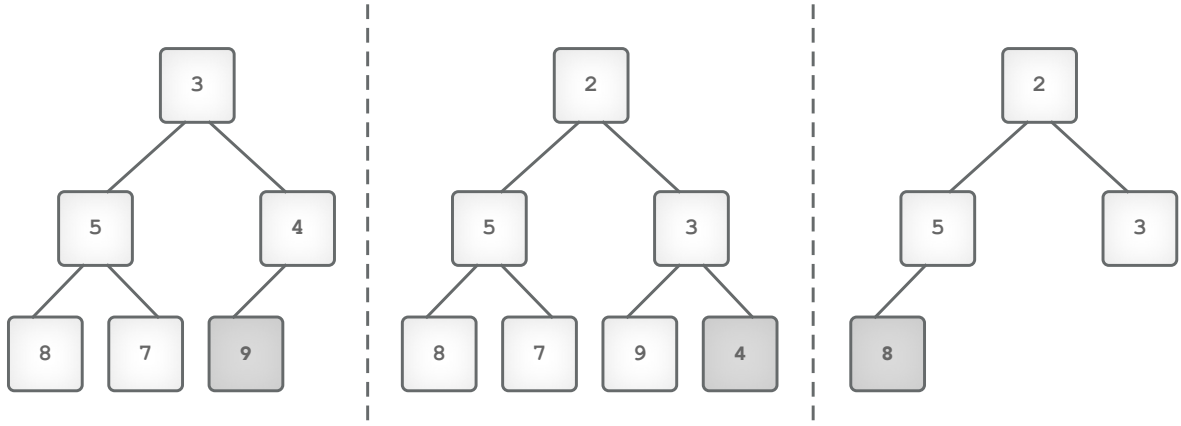
Depois de termos localizado a posição correta para o novo nó temos de ter em conta a propriedade da ordenação. Para isso temos apenas de comparar o valor do novo nó com o valor do pai e realizar a troca de valores caso seja necessário. Claro que esse processo irá ser realizado enquanto for necessário, isto é, continuamos o processo subindo pela árvore até que o novo valor seja maior que o seu pai ou então o novo valor passa a ser a raiz da *heap*. A figura seguinte apresenta um exemplo da inserção de um novo elemento na *heap* e a consequente reordenação da mesma.



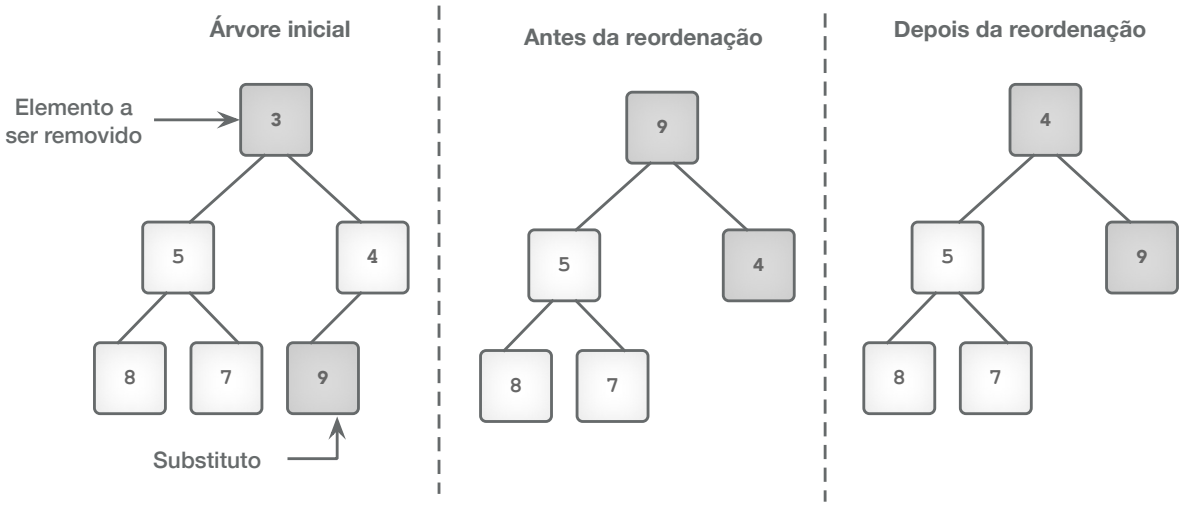
14.5. Remover elementos de uma *heap*

Normalmente na maior parte das coleções temos de saber à partida qual o elemento que pretendemos remover. No caso de uma *heap* é um pouco diferente já que apenas podemos remover os elementos de um local em particular que é raiz. No caso de ser uma *minheap* removemos o menor elemento da *heap*, no caso de se tratar de um *maxheap* removemos o

maior elemento. Neste caso em particular iremos descrever como remover o elemento mínimo da *heap*. Como já foi referido o elemento mínimo é sempre armazenado na raiz. Assim, temos de devolver o elemento raiz e substituí-lo por um outro elemento. O elemento de substituição é sempre a última folha. A última folha é sempre o último elemento no nível h. De seguida podemos ver alguns exemplos da última folha da *heap*.

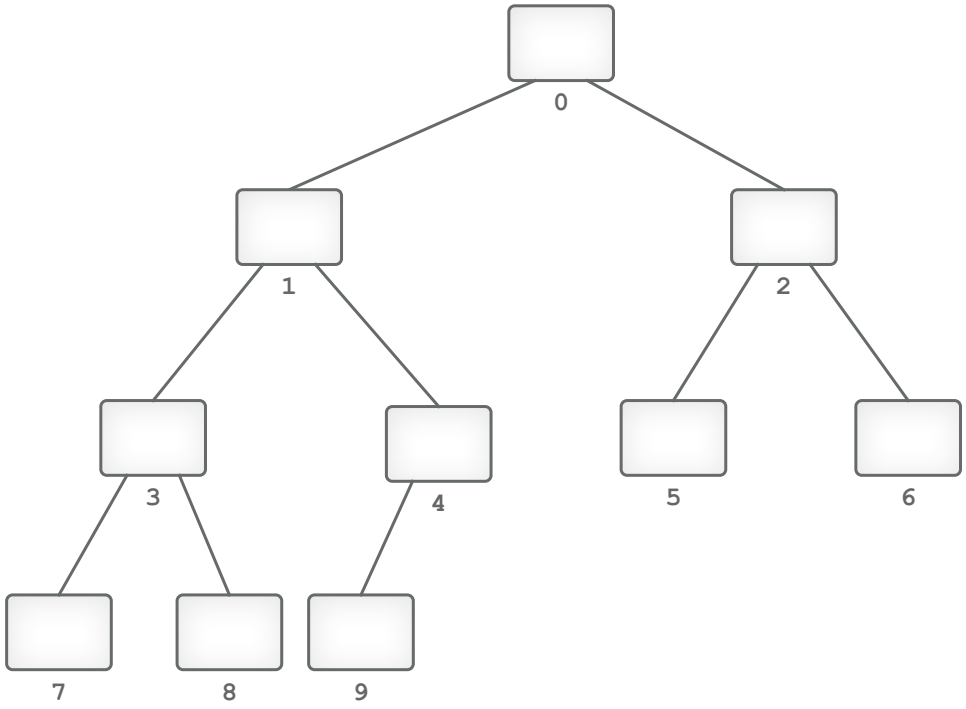


Uma vez que o elemento armazenado na última folha tenha sido transferido para a raiz, a *heap* deve ser reordenada. Isso é feito comparando o novo elemento de raiz com os seus filhos de valor menor (direito e esquerdo) e trocando-os com a raiz caso seja necessário. Este processo é repetido pela árvore a baixo até que o elemento seja uma folha ou menor que ambos os seus filhos. De seguida podemos ver um exemplo de remoção de um elemento e respetiva reordenação da *heap*.



14.6. Implementar heaps

As *heaps* podem ser implementadas de várias formas mas o fato de serem árvores binárias completas torna possível armazená-las de forma muito eficiente em locais de memória contíguos. Considere os números atribuídos aos nós da árvore binária completa na figura seguinte. Observe que os números são atribuídos da esquerda para a direita nos níveis e de cima para baixo na árvore.



Este esquema de numeração pode ser usado para identificar cada nó numa árvore binária completa: o nó zero é a raiz, o nó um é o filho esquerdo da raiz, o nó dois é o filho direito da raiz e assim por diante. Observe em particular que

- O filho esquerdo do nó k é o nó 2k+1;
- O filho direito do nó k é o nó 2k+2;
- O nó pai de k é o nó ((k-1)/2);
- Se houver n nós na árvore, o último com um filho é o nó (n/2) – 1.

Agora, se permitirmos esses números nos nós serem os índices de um *array*, então cada localização de *array* fica associada a um nó na árvore e podemos armazenar os valores nos

nós da árvore no *array*: o valor do nó k é armazenado no *array* na localização k. A correspondência entre os índices do *array* e as localizações dos nós torna possível representar árvores binárias completas em *arrays*. O fato de a árvore binária estar completa significa que cada localização do *array* armazena o valor num nó, de modo a que nenhum espaço é utilizado no *array*.

Claro que a implementação pode ser feita com recurso a listas ligadas tal como já temos implementado outras coleções, mas a implementação mais direta e natural de uma *heap* é com recurso a um *array*. Para a implementação de uma *heap* com recurso a uma lista ligada a implementação seria simplesmente uma extensão da nossa classe `LinkedBinaryTree`. No entanto, uma vez que cada nó precisa ter uma referência para o pai, vamos criar uma classe `HeapNode` para estender à nossa classe `BinaryTreeNode` que usámos anteriormente.

```
1. public class HeapNode<T> extends BinaryTreeNode<T>{
2.    protected HeapNode<T> parent;
3.
4.    /**
5.     * Creates a new heap node with the specified data.
6.     *
7.     * @param obj  the data to be contained within
8.     *             the new heap nodes
9.     */
10.   HeapNode (T obj) {
11.      super(obj);
12.      parent = null;
13.   }
14.}
```

14.7. Notas Finais

Uma árvore é um tipo especial de grafo que é um tópico muito importante nas ciências da computação. Uma aplicação de árvores é para ordenação: um *array* pode ser tratado como uma árvore binária completa e depois transformado numa pilha. A *heap* pode então ser manipulada para ordenar um *array* num tempo de O(n log n).

14.8. Exercícios Propostos

Exercício 1

O que entende por fila de prioridade?

Exercício 2

Uma fila de prioridade pode ser implementada com uma *heap*? Justifique a sua resposta.

Exercício 3

Onde está o maior valor numa *heap*?

Exercício 4

Qual a abordagem para a implementação de uma *heap*? Justifique a sua resposta.

Exercício 5

Porque temos de reordenar a *heap* sempre que adicionamos ou removemos um elemento?

Exercício 6

Desenhe a *minheap* binária resultante da inserção de 7, 5, 6, 3, 8, 1 nessa ordem numa *minheap* inicialmente vazia. Não precisa mostrar a representação em *array* da *heap*. Precisa apenas apresentar as várias iterações da construção da árvore até à final.

Exercício 7

Desenhe o resultado de uma chamada `removeMin` na sua *minheap* desenhada no exercício 6.