Filas de Priodidades com Heaps Binários

Conceitos, exemplos e aplicações implementados na Linguagem C

Santa Maria, Rio Grande do Sul, Brasil
Universidade Federal de Santa Maria

Laura Righi Boemo – Julho de 2022

Data Type (DTs) & Abstract Data Type (ADTs)

Para compreender as Filas de Prioridades e, por conseguinte, sua atuação com os Heaps Binários, é necessário reduzir a abstração do conhecimento abordado neste documento ao seu nível mais baixo, ou seja, à aonde tudo se origina: no conceito de “Data Types” – ou, em português, “Tipos de Dados”.

Os “Data Types” são os Modelos de Dados previamente oferecidos pela Linguagem C, como, por exemplo, *int*, *float* e *double*, dos quais são capazes de performar operações simples como *soma*, *subtração* e *adição*. Contudo, tais “Data Types”, sozinhos, não são capazes de suprir certas questões, como em situações em que necessitamos que o nosso “Data Type” venha a ser definido pelo usuário, onde as operações só podem ser nomeadas após solicitarmos por elas.

Em casos como o citado acima, a solução é criar e manipular uma Estrutura de Dados concomitantemente às operações solicitadas, tal tipo de manobra sobre as informações fornecidas, que não poderiam ser previamente definidas, é conhecida como “Abstract Data Type” – ou, em português, “Tipo Abstrato de Dado”.

Ou seja, os “Abstract Data Types” são um tipo de objeto cujo comportamento é definido por um conjunto de valores e um conjunto de operações dos quais atuam concomitantemente. Por definição, os ADTs mencionam quais operações devem e podem ser executadas, mas não como serão implementadas: como os dados serão organizados na memória e quais algoritmos serão utilizados para implementar suas operações não são especificados. E, então, por tal motivo, são chamados de “Abstratos” uma vez que fornece uma visão independente de sua implementação.

Um bom exemplo a ser comentado de ADT é a Fila. Uma estrutura da qual os elementos se alinham em um formato similar ao de uma lista, contudo, seguindo a regra de inserção e exclusão “FIFO” – ou seja, o primeir a entrar (“First In”), é, também, o primeiro a sair (“First Out”).

Filas de Prioridades

O tipo Fila, como mensionado anteriormente, é um ADT com dados encadeados em forma de uma lista, possuindo uma política de inserção e exclusão do tipo FIFO (First In – First Out). Em sequência, existe um tipo especial de Filas: as Filas de Prioridades, das quais seguem o mesmo conceito de uma Fila, contudo, colocando ao final da lista os elementos de menor prioridade – e, assim, deixando mais à frente os elementos de maior prioridade.

Uma analogia consistente às Filas de Prioridades são as filas de um hospital que sempre deixarão priorizarão os pacientes em situação emergencial, para que sejam atendidos antes. Ou seja, colocando os elementos de maior prioridade mais à frente para que sejam atendidos primeiro.

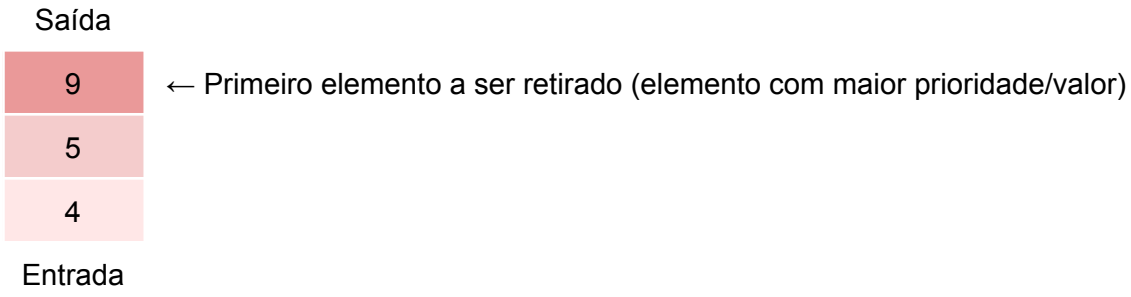
Filas & Filas de Prioridades: Qual a diferença?

Em uma Fila, a regra de inserção e exclusão “FIFO” é aplicada indiferente da ordenação dos elementos.

Em uma Fila de Prioridade, o maior valor (maior prioridade) sempre ficará como o início da fila. Ou seja, a regra “FIFO” segue respeitada, contudo, a fila é reordenada de maneira que o primeiro a sair sempre seja o maior elemento – e não, necessáriamente, o primeiro elemento a entrar na Fila de Prioridade.

Usualmente, em uma Fila de Prioridade, o valor de um elemento costuma, também, ser seu próprio indicador de priorização. Ou seja, o elemento de maior valor é considerado o elemento de maior prioridade. Tal organização também pode ser feita em seu inverso, onde o elemento de menor prioridade fica em primeiro e o de maior por último ao longo da lista da Fila de Prioridade.

Exemplificação de uma lista cuja prioridade mais alta refere-se ao primeiro elemento a ser retirado.

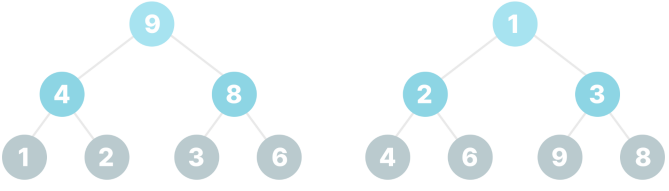


Filas de Prioridades com Heaps Binários

Uma Fila de Prioridades pode ser implementada utilizando Arrays, Listas Encadeadas, Heaps Binários ou Árvores Binárias. Dentre todas essas opções, os Heaps Binários oferecem uma implementação mais eficiente das Filas de Prioridades ao considerar o seu uso do processador em comparação à complexidade do Input: baseando na “Big O Notation”, para consulta, O(1), inserção, O(log n), e remoção, O(log n). E, dados tais fatos, estaremos utilizando tal tipo de Estrutura de Dados para implementar a Fila de Prioridades.

O Heap Binário é uma Árvore Binária completa que satisfaz as seguintes propriedades Heap: em caso de Max-Heap, os pais devem sempre ser maiores que os filhos, ou, em caso de Min-Heap, os pais devem sempre ser menores que os filhos (estaremos realizando o Max-Heap); a Árvore Binária deve sempre estar completa exceto, talvez, pela última linha na parte da direita.

Segue a ilustração comparando um Max-Heap (esquerda) com um Min-Heap (direita).



Funções para uma Fila de Prioridade com Heaps Binários

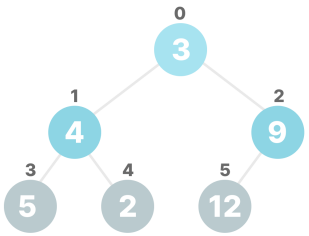
Para realizar e manipular uma Fila de Prioridade com Heaps Binários, são necessárias quatro funções: ordenar, inserir, excluir e imprimir.

1. Ordenar Elementos

A função de ordenar um Heap Binário serve, principalmente, para quando temos uma lista de elementos e desejamos ordená-la em um Max-Heap (seguindo a ideia de Fila de Prioridades). Podemos tomar como exemplo o seguinte Array:

3	4	9	5	2	12

Para isso, o primeiro passo é transformá-lo em uma Árvore Binária Completa.

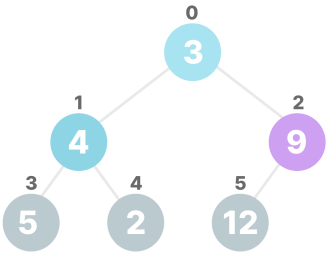


Neste momento, iremos localizar o índice de seu último nó que não esteja na sua última fileira, ou seja, **sendo n a quantidade total de nós ocupados no Heap**, aquele cujo índex pode ser localizado com a seguinte fórmula…

$$i = n/2 - 1$$

Neste caso, receberemos o número 9, na segunda posição.

$$i = 6/2 - 1$$
$$i = 3 - 1$$
$$i = 2$$



Em seguida, definimos o índice i como o Maior Elemento quando comparado a seus dois filhos, mesmo que tal fato não seja verdade.

$$maiorElemento = i$$

Ou seja,

```
maiorElemento = 2
```

Para encontrar o Filho da Esquerda e o Filho da Direita de qualquer nó que não esteja na última fileira do Heap, utilizamos as seguintes fórmulas…

Filho da Esquerda do elemento de índice i

```
filhoEsquerda = 2 * i + 1
```

Ou seja,

```
filhoEsquerda = 2 * 2 + 1
filhoEsquerda = 4 + 1
filhoEsquerda = 5
```

O Filho da Esquerda contém índice 5, sendo o elemento de valor 12.

Filho da Direita do elemento de índice i

```
filhoDireita = 2 * i + 2
```

Ou seja,

```
filhoDireita = 2 * 2 + 2
filhoDireita = 4 + 2
filhoDireita = 6
```

O Filho da Esquerda contém índice 6, sendo o elemento de valor nulo, não inserido no Heap.

Considerando tais fórmulas, podemos dar procedência ao processo de organização do Heap com os seguintes dois laços condicionais.

Caso o Filho da Esquerda seja maior que o Maior Elemento, então o definimos como o maior elemento…

```
if (filhoEsquerda  > maiorElemento ) {
    maiorElemento = filhoEsquerda;
}
```

Ou seja,

```
if (índice5 > índice2) {
    maiorElemento = índice5;
}

if (12 > 9) {
    maiorElemento = 12;
}
```

Neste caso, o laço é verdadeiro e a variável de Maior Elemento deixa de ser o índice da posição 9 e passa a ser a de posição 12.

Caso o Filho da Direita seja maior que o Maior Elemento, então o definimos como o maior elemento…

```
if (filhoDireita  > maiorElemento ) {
    maiorDireita = filhoEsquerda;
}
```

Ou seja,

```
if (índice6 > índice5) {
    maiorElemento = índice6;
}

if (0 > 12) {
    maiorElemento = 0;
}
```

Neste caso, o laço não é verdadeiro e o Maior Elemento segue com o índice do número 12.

Por conseguinte, ao final de ambas as condicionais, invertemos a posição do valor atual do índice i com o valor do Maior Elemento, assim fazendo o Maior Elemento ficar na posição de superioridade do Heap em relação àquele nó.

Para inverter, sendo “a” o índice de i e “b” o índice do Maior Elemento, chama-se a seguinte função…

```
void inverter(int *a, int *b) {
    int temp = *b;
    *b = *a;
    *a = temp;
}
```

Por fim, a função para ordenar o Heap é chamada novamente, porém, enviando o novo valor de i para que, desse modo, todos os elementos do Heap passem pela verificação de ordenação, parando a chamada recursiva apenas quando o índice do Maior Elemento seja o próprio índice de i.

Tanto a função de inverter quanto a de ordenar são chamadas sob tal condicional de “Maior Elemento diferente de i”.

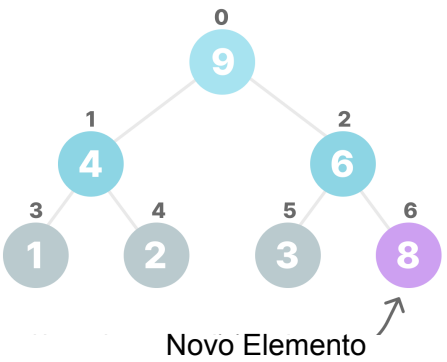
```
if (maiorElemento != i) {
   inverter(i, maiorElemento);
   ordenarHeap(h, maiorElemento);
 }
```

Desse modo, em código na Linguagem C, para a função de ordenação do Heap, temos…

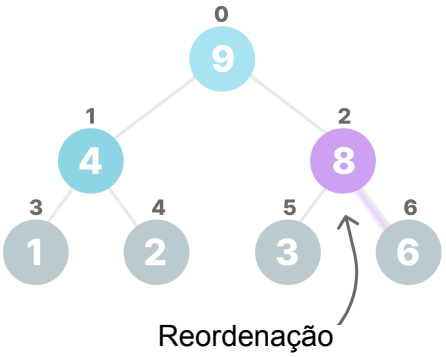
```c
void inverter(int *a, int *b) {
    int temp = *b;
    *b = *a;
    *a = temp;
}

void ordenarHeap(EstruturaDoHeap* h, int i) {
    if (h->size > 1) {
        int largest = i;
        int l = 2 * i + 1;
        int r = 2 * i + 2;

        if (l < h->size && h->heaparray[l] > h->heaparray[largest])
            largest = l;
        if (r < h->size && h->heaparray[r] > h->heaparray[largest])
            largest = r;

        if (largest != i) {
            inverter(&h->heaparray[i], &h->heaparray[largest]);
            ordenarHeap(h, largest);
        }
    }
}
```

2. Inserir Elemento

Para inserir um novo elemento no Heap, devemos, primeiramente, adicionar o novo valor na última posição disponível do Heap.



Então, é necessário chamar a função de Ordenar para organizar a posição em que o novo valor realmente deveria estar (mesmo que ela seja, de fato, a última).



Tal processo é feito conforme a seguinte lógica…

Função para inserir elemento no Heap(Recebe o ponteiro de Heap e o novo elemento) {
Adiciona no Array de Heap, em sua última posição (que é a quantidade de números ocupados, seu tamanho), o novo elemento.
Acresenta mais um ao tamanho do Heap.

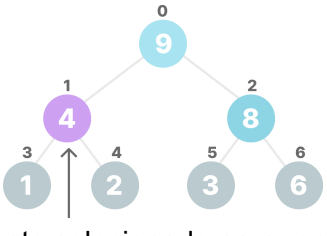
Laço de repetição for para percorrer todos os nós i (maior ao menor, de baixo para cima) e ordená-los.
}

Desse modo, em código na Linguagem C, para a função de inserção no Heap, temos…

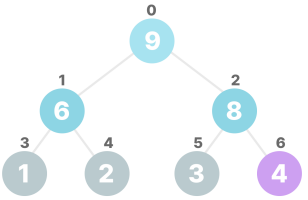
```c
void inserirNoHeap(EstruturaDoHeap* h, int newNum) {
    h->heaparray[h->size] = newNum;
    h->size += 1;

    for (int i = (((h->size)/2) - 1); i >= 0; i--) {
        ordenarHeap(h, i);
    }
}
```

3. Remover Elemento

Para remover um elemento do Heap, devemos encontrar o índice da posição do elemento que está sendo desejado eliminar e invertê-la com o último elemento.

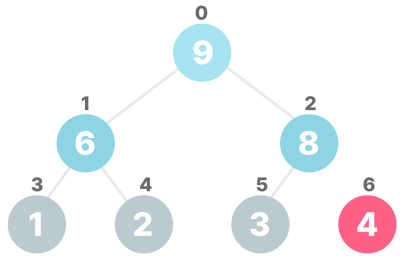


Elemento selecionado para remover

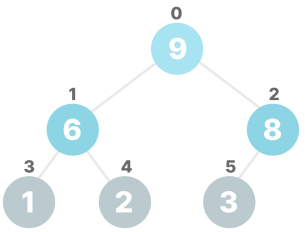


Elemento invertido para última posição

Após inverter o modelo de maneira que ele se localiza na última posição do Heap, eliminamos o último elemento do Heap, assim, removendo o elemento desejado. Bem como, em seguida, reordenamos o Heap chamando a função de ordenar caso a troca de posição tenha causado algum problema para com a prioridade dos elementos.



Elemento eliminado



Reordenando o Heap

Tal processo é feito conforme a seguinte lógica…

Função para remover elemento do Heap(Recebe o ponteiro de Heap e o elemento a remover) {
Declara-se variável auxiliar.

Laço de repetição que fará com que a variável auxiliar assuma todos os índices possíveis dentro do tamanho do array, caso algum valor do array seja idêntico ao valor a ser removido, a variável auxiliar para de ser percorrida e armazena o índice do elemento a ser removido.

Chama-se a função inverter, enviando o índice da variável auxiliar e a última posição do Heap.
Diminui-se o tamanho do Heap, assim eliminando o último valor (que, com a inversão era o nosso elemento a ser removido).

Laço de repetição for para percorrer todos os nós i (maior ao menor, de baixo para cima) e ordená-los.
}

Desse modo, em código na Linguagem C, para a função de remoção do Heap, temos…

```c
void removerElementoDoHeap(EstruturaDoHeap* h, int num) {
    int i;

    for (i = 0; i < h->size; i++) {
        if (num == h->heaparray[i])
            break;
    }

    inverter(&h->heaparray[i], &h->heaparray[h->size - 1]);
    h->size -= 1;

    for (int i = (((h->size)/2) - 1); i >= 0; i--) {
        ordenarHeap(h, i);
    }
}
```

4. Imprimir Elementos

Para imprimir o Heap, imprimirei em dois formatos: um representando sua Fila Priorizada e outro representando seu Heap. Para o primeiro, simplesmente chamo a função “imprimeArray” e percorro os elementos do array; o que, escrito na Linguagem C, é:

```
void imprimirArray(EstruturaDoHeap* h) {
    for (int i = 0; i < h->size; ++i) {
        printf("%d ", h->heaparray[i]);
    }
}
```

```
Imprimindo a Lista Priorizada dos valores...

90 54 14 8 3 7 9 2
```

Em sequência, para imprimir o Heap numa representação de seu formato de Árvore, utilizei da seguinte lógica:

Função para imprimir Heap(Recebe o ponteiro de Heap) {
Declara-se variável auxiliar para contar o limite de elementos em uma linha e para contar a quantidade de elementos impressos na linha.

Laço de repetição com variável auxiliar i percorrendo todos os índices possíveis dentro do tamanho do Heap.

Laço condicional para, caso seja o primeiro elemento do Heap (de índice i 0), apenas será impresso seu valor. Do contrário, para índice diferente de 0, imprimimos uma nova linha de elementos, onde, para cada elemento, o auxiliar contador de elementos na linha soma um.

Após primeiro laço condicional, caso o contador de elementos contenha a mesma quantidade que a variável de limite de elementos em uma linha. Então a linha é quebrada e o tamanho da variável que armazena a quantidade duplica-se, já que agora estamos lidando com o dobro de elementos que a linha anterior, e é somada ao total da linha anterior (já que o mesmo é acumulativo).
}

Desse modo, em código na Linguagem C, para a função de inserção no Heap, temos…

```c
void imprimirHeap(EstruturaDoHeap* h) {
    int contadorLinhas = 2, auxContador = 0;

    for (int i = 0; i < h->size; ++i) {
        if(i == 0) {
            printf("%d\n\n", h->heaparray[i]);
        } else {
            printf("%d ", h->heaparray[i]);
            auxContador += 1;
        }

        if(auxContador == contadorLinhas) {
            printf("\n\n");
            contadorLinhas = contadorLinhas * 2 + contadorLinhas;
        }
    }

    printf("\n");
}
```

Ou seja, finalizando com o seguinte Output…

```
Imprimindo a Lista Priorizada em formato de Heap...

46

23 12

2 5 4 8
```

```
Imprimindo a Lista Priorizada em formato de Heap...

46

23 12

2 5 4 8
```

Referências

https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/priority.html
https://medium.com/swlh/abstract-date-types-binary-trees-2e2f2bedfeaf
https://www.edureka.co/blog/queue-in-c/
https://pt.slideshare.net/rodrigovmoraes/heap-8003774
https://www.youtube.com/watch?v=XDxLEUgVDMM