CENTRO DE TECNOLOGIA - UNIVERSIDADE FEDERAL DE SANTA MARIA

CIÊNCIA DA COMPUTAÇÃO

GUILHERME MENEGHETTI EINLOFT
JAIME ANTONIO DANIEL FILHO
LUÍS GUSTAVO WERLE TOZEVICH

HEAPS BINÁRIOS: UMA IMPLEMENTAÇÃO PRAGMÁTICA PARA FILAS DE PRIORIDADE

SANTA MARIA/RS
2023

"A computação é sobre ideias - as máquinas apenas as executam." - Edsger Dijkstra

LISTA DE FIGURAS

LISTA DE SIGLAS

FIFO - First In First Out

LIFO - Last In First Out

SUMÁRIO

INTRODUÇÃO

Estruturas de dados, como filas de prioridade, são de extrema importância no campo da computação devido à sua capacidade de facilitar a manipulação eficiente de dados em vários cenários que exigem a priorização de elementos. A utilização dessas estruturas de dados permite a otimização de algoritmos e o aprimoramento do desempenho geral do sistema. O objetivo deste estudo é aprofundar a compreensão e a aplicação dessas estruturas de dados, com ênfase específica na implementação de filas prioritárias baseadas em *heaps* binários.

Além da teoria, este trabalho enfatiza o impacto prático dessas estruturas, também mostramos e detalhamos os passos das nossas implementações. Permitindo a compreensão do funcionamento interno dessas estruturas e como elas podem ser efetivamente utilizadas para resolver problemas computacionais.

1. OBJETIVOS

Nesta seção, descrevem-se os objetivos que orientam a elaboração deste projeto.

1.1 OBJETIVO GERAL

O objetivo geral deste trabalho consiste na elaboração de um material que possibilite ao leitor o pleno entendimento sobre filas de prioridade implementadas como *heaps* binários. Abordar-se-ão os conceitos fundamentais, fornecendo ilustrações das operações de inserção e remoção em *heaps* binários, além de apresentar os algoritmos associados a essas operações. Busca-se também a completa compreensão das operações de impressão e consulta em filas de prioridade com *heaps* binários.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos deste trabalho são:

a) Apresentar conceitos sobre filas de prioridade e *heaps* binários, destacando sua relevância na computação e ciência da informação.
b) Ilustrar, por meio de exemplos e representações visuais, o funcionamento prático das filas de prioridade com *heaps* binários, detalhando as etapas de inserção e remoção de elementos na estrutura.
c) Descrever os algoritmos em pseudocódigos relacionados às operações de inserção, remoção e impressão/consulta em filas de prioridade com *heaps* binários.
d) Fornecer informações acerca das complexidades de tempo associadas a operações em *heaps* binários, visando conhecer sua eficiência computacional.
e) Explorar aplicações práticas das filas de prioridade com *heaps* binários em contextos reais, evidenciando sua utilidade e relevância na resolução de problemas cotidianos e complexos.

2. FILAS DE PRIORIDADE

Uma fila de prioridade constitui-se como uma estrutura de dados que preserva uma coleção de elementos, sendo que a cada elemento é atribuída uma prioridade associada (IME-USP, s.d.). A ideia fundamental por trás de uma fila de prioridade é que os elementos sejam dispostos de forma que o elemento com a maior prioridade esteja sempre pronto para ser acessado e processado. Imaginemos que existem K classes de itens (1, 2, 3, …, K). Em um sistema para processarmos um item, o tipo i será selecionado com preferência acerca do tipo j se a classe de preferencia de i for maior que a classe de j, ou seja, o elemento i será escolhido mesmo que o elemento j tenha chegado a fila antes do item do tipo i (MILLER JR, 1960). Essa estrutura não respeita a política de saída das pilhas (LIFO) nem das filas (FIFO), a diferença entre essas três estruturas pode ser visualizada na imagem 01. Possui grande importância quando pensamos em algoritmos e sistemas que envolvem a ordenação e o processamento eficiente de elementos com base em suas prioridades relativas. Implicando na necessidade de operações eficientes de inserção, remoção e acesso ao elemento com a maior prioridade, de modo a otimizar o desempenho das aplicações que utilizam essa estrutura.

Figura 1 - Diferença entre pilhas,filas e filas de prioridade

Fonte: Autor (2023)

Há várias abordagens disponíveis para a implementação dessas estruturas, destacando-se as estruturas mais comuns, tais como a fila de prioridade utilizando *heaps* binários, *heap* binomial (Okasaki, 1999), *heap* de Fibonacci e vetores, entre outras. Cada uma dessas estruturas apresenta características distintas e diferentes níveis de desempenho, de acordo com as necessidades e contextos específicos de aplicação. Neste estudo, no entanto, nosso foco estará na compreensão da

implementação utilizando *heaps* binários. Uma fila de prioridade possui aplicação em uma diversidade de algoritmos e problemas computacionais, como: escalonamento de processos em servidores, conforme demosntrado por Gittins e Nash (1977) e sua vasta aplicação no gerenciamento de atendimentos médicos, como evidenciado por Gupta (2013), visto a necessidade de atendimento preferencial a alguns pacientes, um exemplo de fila de prioridade pode ser observada na figura 02. Estabelecendo-se como uma ferramenta crítica no desenvolvimento de sistemas e algoritmos que exigem a manipulação de elementos com base em suas prioridades.

Figura 2 - Exemplo de fila de prioridade

Fonte: Martins (2021)

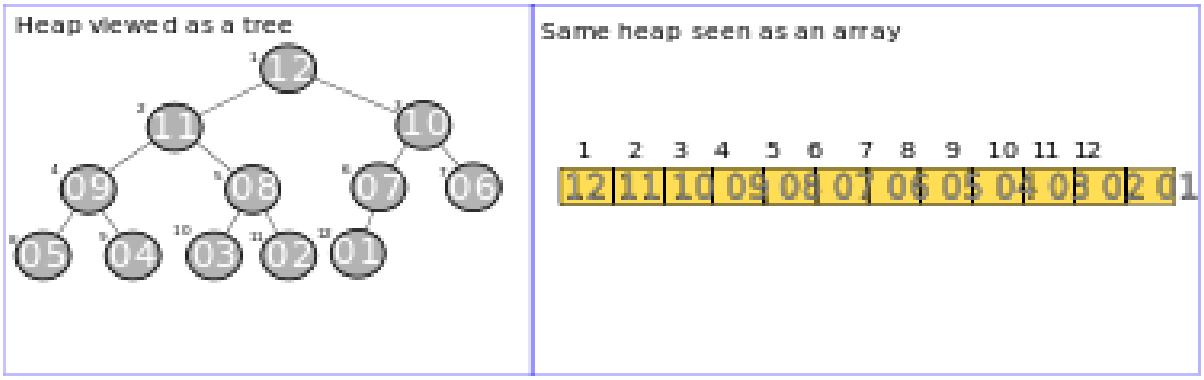
3. HEAPS BINÁRIOS

Esta seção busca firmar conhecimentos acerca dos *heaps* binários e sua aplicabilidade no contexto deste estudo.

3.1 O QUE É UM HEAP BINÁRIO?

Os *heaps* binários, que foram introduzidos por Williams (1964), são estruturas de dados que possibilitam o acesso e a retirada do elemento de maior prioridade dentro de uma coleção de itens na qual novos itens podem ser inseridos a qualquer instante. Os *heaps* binários, assim como explicado por Kostyukov (2013), são representados como árvores binárias. Existem duas propriedades que definem um *heap* binário: a propriedade de forma e propriedade de *heap*. A propriedade de forma garante que os *heaps* binários são árvores completas, o que significa que todos os níveis da árvore estão preenchidos, exceto possivelmente o último, o qual é preenchido da esquerda para a direita. Isso possibilita a utilização de vetores para armazenar a estrutura do *heap* binário de forma eficiente, como pode ser visto na figura 03, o que simplifica diversas operações, como a inserção e remoção de elementos. A propriedade *heap* exige que o elemento armazenado em cada nó seja maior ou igual, ou menor ou igual aos elementos nos nós filhos de acordo com critérios previamente definidos de ordenação dos elementos, dependendo do tipo de *heap* binário empregado.

Figura 3 - Visualização de heaps binários como vetores



Fonte: Cormen (2001)

Os *heaps* binários são comumente empregados em algoritmos de ordenação, como o *HeapSort*, e em diversas aplicações que demandam uma fila de prioridade eficiente, já que permitem acesso rápido ao elemento de maior prioridade, presente no topo do *heap*.

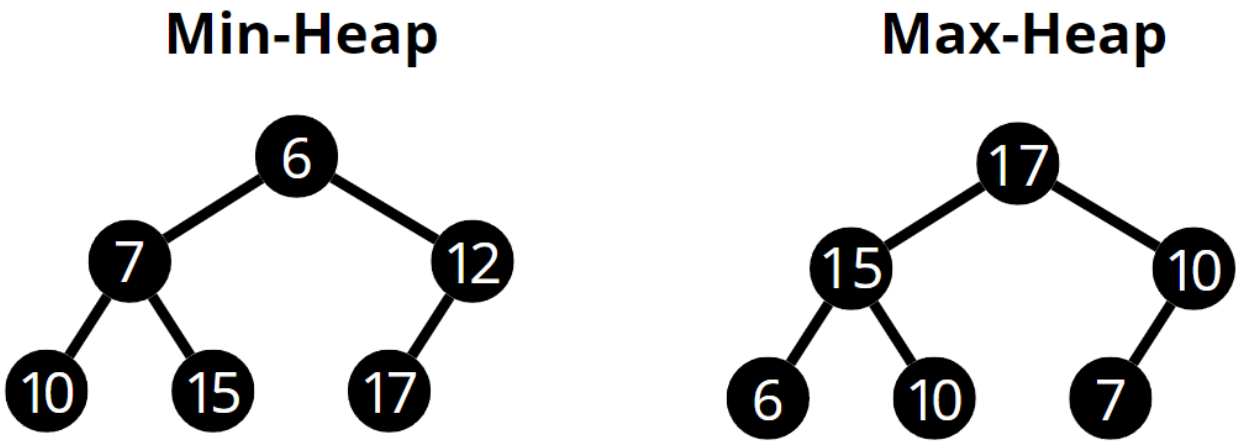
3.2 TIPOS DE HEAPS BINÁRIOS

Os *heaps* binários, divididos em *min-heap* e *max-heap*, são estruturas de dados que priorizam a organização do elemento mais extremo – mínimo ou máximo – no topo do *heap*. Um min-heap mantém o elemento mínimo no topo do heap. Cada nó possui um valor que é menor ou igual aos valores dos seus filhos. Consequentemente, o nó no topo do *heap*, a raiz, contém o menor valor presente na estrutura. Assim, o elemento de menor valor está localizado no topo de um min-*heap*. É eficiente para selecionar os K elementos principais em um conjunto (JELČICOVÁ, 2023).

Por sua vez, um *max-heap* mantém o elemento máximo no topo do *heap*. Neste caso, valor atribuído a cada nó é sempre maior ou igual ao valor dos seus filhos. Isso implica que o nó no topo do *heap*, a raiz, contém o maior valor em comparação com todos os outros nós na estrutura. Assim, em um *max-heap*, o elemento de maior valor está sempre no topo. É útil para ordenar itens de acordo com seus valores de ordenação e remover o primeiro item da fila (KOHUTKA, 2018).

A diferença entre *min-heap* e *max-heap* pode ser visualizada na figura 04.

Figura 4 - Diferença entre min-heap e max-heap



Fonte: Autor (2023)

3.3 COMPLEXIDADE DE TEMPO

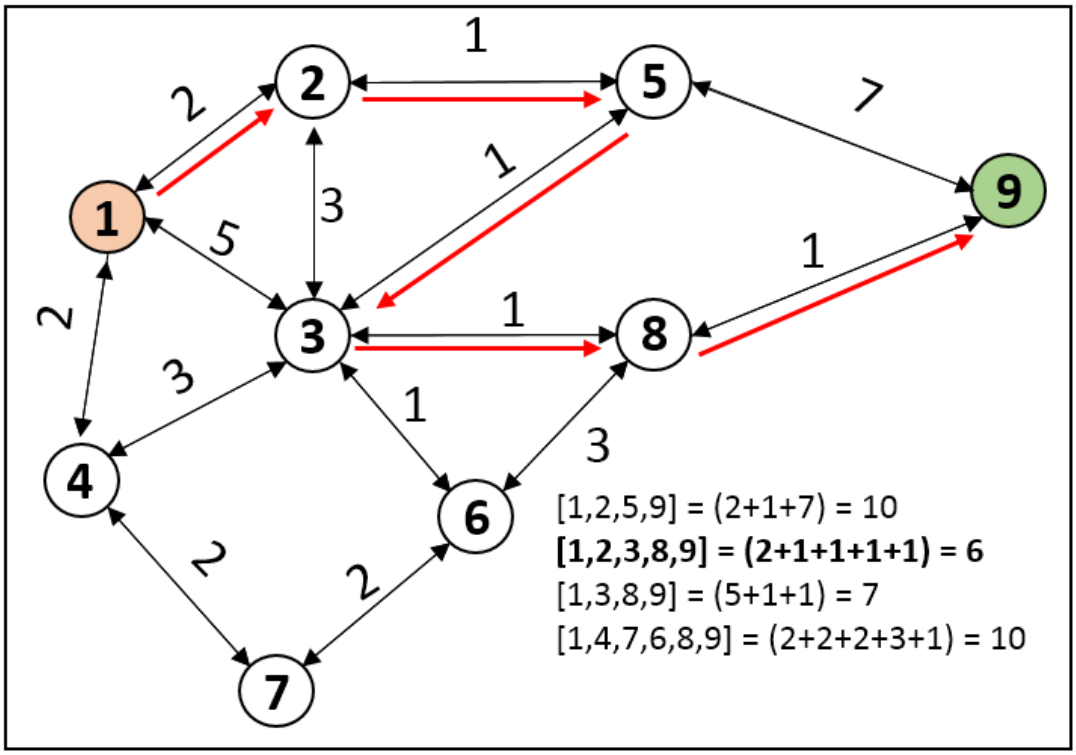
No algoritmo apresentado por Williamns (1964) a inserção de um elemento e a retirada do elemento mínimo possuía o tempo de $O(\log n)$, onde n é o número de elementos na fila de prioridade. Segundo Williams (1964) o tempo para construir um heap a partir de um vetor de n elementos é de $O(n \log n)$. Sack e Strothotte (1985) exemplificaram como realizar a junção de dois

heaps binários de tamanho k e n em um tempo de O(k + log n . log k). Uma modificação de um *heap* binário proposta por Edelkamp et al. (2017) demonstra uma eficiência de tempo de O(1) no pior caso para as operações de inserção e extração do elemento mínimo. Esta variante mantém a complexidade temporal no pior caso de *O*(log n) para a operação de extração. Além disso, implementações avançadas, tais como os *heaps* de emparelhamento, são capazes de atingir uma complexidade de tempo de *O*(1) para a operação de inserção (BRODAL, 2013).

4. ALGORITMO DE DIJKSTRA

O algoritmo de Dijkstra é um algoritmo frequentemente empregado em problemas de roteamento de rede e cálculos complexos de teoria dos grafos (SZCZEŚNIAK et. al., 2023; HOU, 2022). Ele pode identificar o caminho mais curto em uma área (CHEK, 2023) e, por esse motivo, é comumente utilizado para o cálculo de menor distância entre dois vértices, um exemplo dessa aplicação pode ser visto na figura 05. Pode-se utilizar uma fila de prioridade para manter e gerenciar os vértices a serem explorados, privilegiando aqueles com menor custo até o momento, oferecendo um acesso eficiente ao elemento de menor custo. Pois a fila de prioridade, implementada com *heaps* binários, permite que o algoritmo acesse e atualize o elemento em tempo logarítmico. No entanto, em alguns casos o algoritmo pode ser ineficiente devido ao grande número de nós percorridos (BHATTACHARYYA e KARMAKAR, 2022). Assim, diversos estudos propuseram otimizações para aprimorar a eficiência do algoritmo para determinados usos, como normalização de regressão linear e técnicas inteligentes de gerenciamento de energia. Essas otimizações visam reduzir o tempo de computação, o consumo de memória e de energia, além de lidar com problemas como a prevenção de colisões e *deadlock*.

Figura 5 - Exemplo de uso de algoritmo de Dijktra



Fonte: Rehman et. al. (2019)

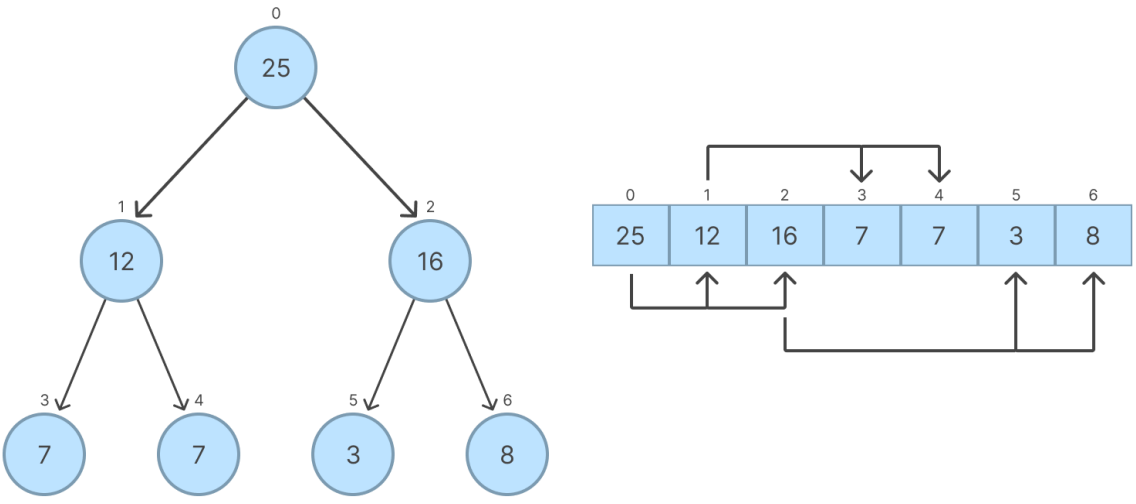
5. IMPLEMENTAÇÃO DE HEAPS BINÁRIOS

Nesta seção, apresentaremos nossa implementação de filas de prioridade utilizando *heaps* binários e o seu funcionamento. Todas as operações são realizadas de modo a garantir as propriedades de forma e de *heap*, necessárias em um *heap* binário.

5.1 ESTRUTURA DE DADOS IMPLÍCITA

O conceito de estruturas de dados implícitas refere-se a metodologias nas quais os dados são organizados de uma maneira que não é explicitamente declarada. Em vez disso, a estrutura de dados é derivada de uma forma mais flexível e adaptável, geralmente aproveitando à maneira pela qual os elementos são organizados na memória e à forma como as operações são executadas, sem necessariamente ser explicitamente definida como uma estrutura de dados tradicional. Quando pensamos em *heap* binários, notamos que seus elementos preenchem completamente um nível de uma árvore binária antes de partir para o próximo. Dessa forma, os *heaps* binários, embora possam ser implementados utilizando árvores que armazenam referências aos nós filhos, podem ser representados por um vetor unidimensional. Como pode ser visto na figura 06.

Figura 6 - Exemplo estrutura de dados implícita



Fonte: Autor (2023)

Nessa representação, o primeiro elemento do vetor é a raiz e os nós seguintes são adicionados da esquerda para a direita percorrendo cada um dos níveis da árvore sequencialmente. A próxima posição livre do vetor sempre representará o nó livre mais a esquerda do último nível da árvore binária. Isso permite que os novos elementos sempre sejam inseridos ao final do vetor para

conformar com a forma de uma árvore binária completa, além de operações eficientes por meio de cálculos simples para acessar os filhos e os pais de um nó específico. Segundo Zapata-Carratala, Arsiwalla e Beynon (2022), tal implementação é uma das melhores opções para armazenar eficientemente *heaps* binários. Utiliza-se os índices do vetor para determinar as relações entre os nós (por exemplo, o filho esquerdo de um nó no índice i está localizado em 2 * i + 1 e o filho direito está em 2 * i + 2).

5.2 INSERÇÃO DE ELEMENTOS

O pseudocódigo a seguir representa a lógica utilizada para realizar as operações de inserção de elementos no *heap* binário.

```
procedimento push_heap(heap, valor)
   garantir_capacidade(heap, heap.size + 1)

   heap.size = heap.size + 1

   i = heap.size - 1
   pai = obter_nó_pai(i)

   heap.array[i] = valor

    enquanto i ≠ 0 e heap.compare(heap.array[pai], heap.array[i]) faça
      trocar(heap.array[i], heap.array[pai])
      i = pai
      pai = obter_nó_pai(i)
   fim enquanto
fim procedimento
```

Primeiramente, verifica-se a capacidade do *heap* para garantir que exista espaço para o novo elemento. Após isso, aumenta-se o tamanho do *heap* para ocorrer a inserção. O valor é inserido em sua posição apropriada (no final), para garantir a propriedade de forma de uma árvore binária completa, e após isso, inicia-se um ciclo para ordenar a estrutura, comparando novo elemento com seu pai e, se o novo elemento violar a propriedade do *heap*, isto é, se for maior (ou menor) que seu

pai, os elementos são trocados de posição. Esta troca continua subindo na árvore até que a propriedade do *heap* seja restaurada para o novo elemento inserido.

A fim de visualizarmos como ocorrem as inserções na fila de prioridade utilizando este algoritmo, realizaremos uma simulação dos procedimentos realizados e verificaremos o seu estado após cada inserção. A figura 07 detalha as inserções realizadas.

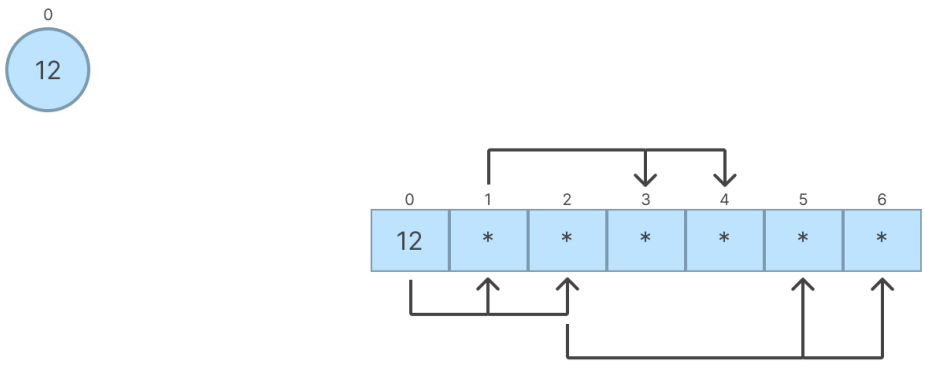
Figura 7 - Comando de inserção executados

```
push_heap(heap, 12);
push_heap(heap, 7);
push_heap(heap, 8);
push_heap(heap, 7);
push_heap(heap, 16);
push_heap(heap, 25);
push_heap(heap, 3);
```

Fonte: Autor (2023)

Inicialmente, a inserção começa com o *heap* vazio. O valor 12 é inserido na posição inicial, tornando-se a raiz. O resultado da inserção pode ser visualizado na figura 08.

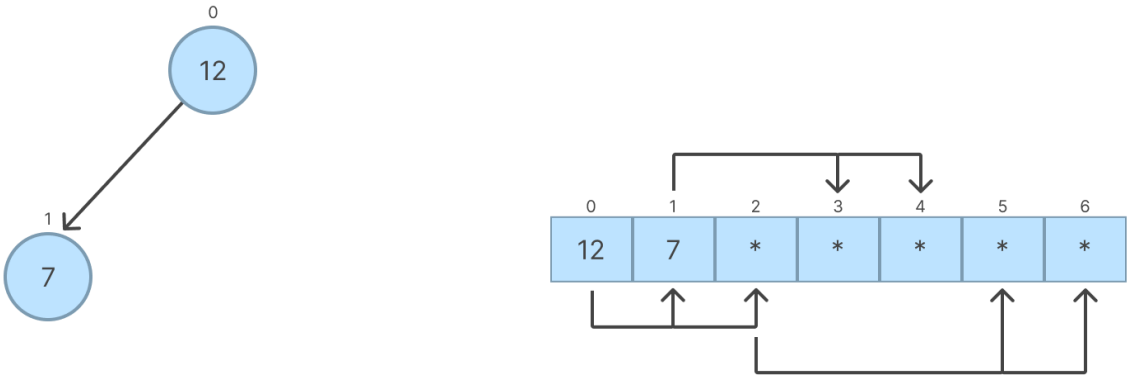
Figura 8 - Primeira inserção



Fonte: Autor (2023)

O 7 é inserido como um novo nó à esquerda do nó 0, pois mantém a propriedade do *heap* (nó pai > nó filho). O resultado da inserção pode ser visualizado na figura 09.

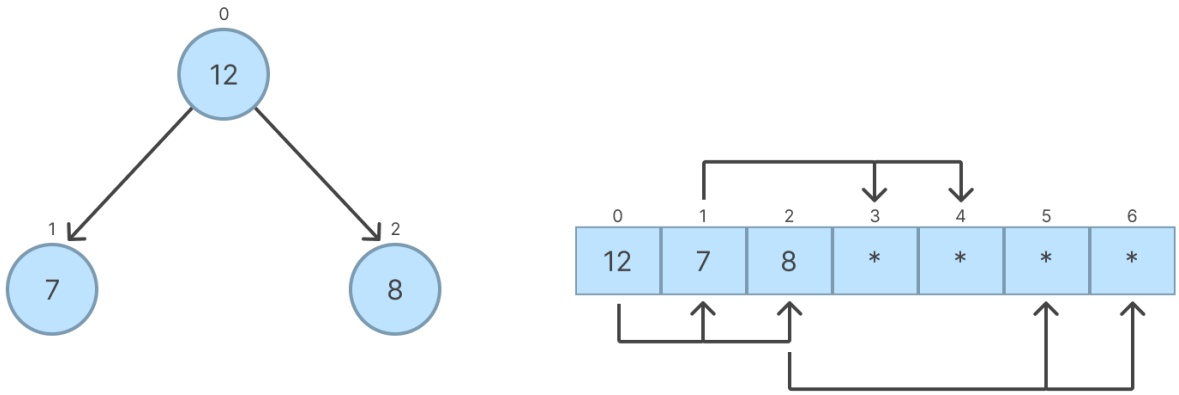
Figura 9 - Segunda inserção



Fonte: Autor (2023)

Agora, insere-se o valor 8 a direita do nó 0. O resultado da inserção pode ser visualizado na figura 10.

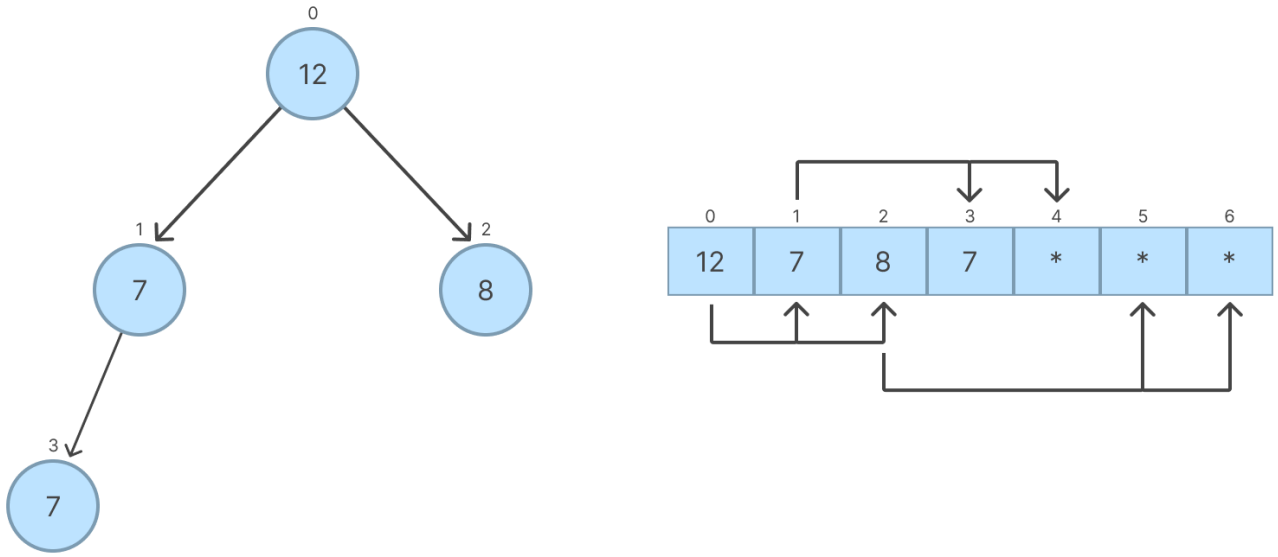
Figura 10 - Segunda inserção



Fonte: Autor (2023)

Inserção do valor 7 à esquerda do nó 1 (valor 7). Mantém-se a propriedade do heap, então nenhuma troca é realizada. O resultado da inserção pode ser visualizado na figura 11.

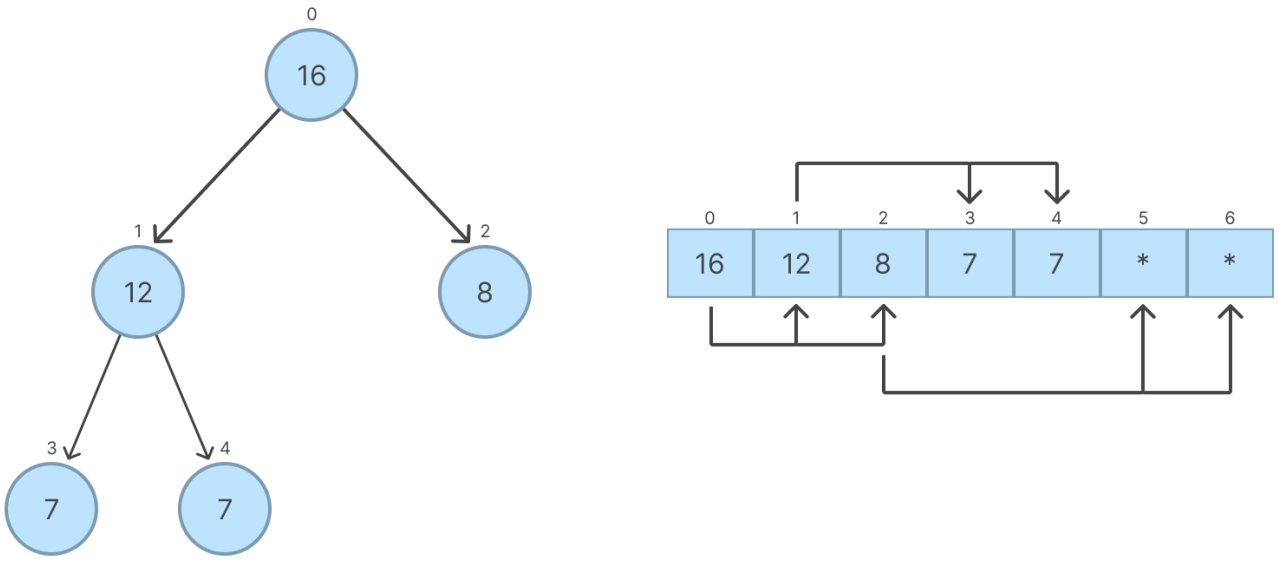
Figura 11 - Terceira inserção



Fonte: Autor (2023)

Inserção do valor 16. O resultado da inserção pode ser visualizado na figura 12.

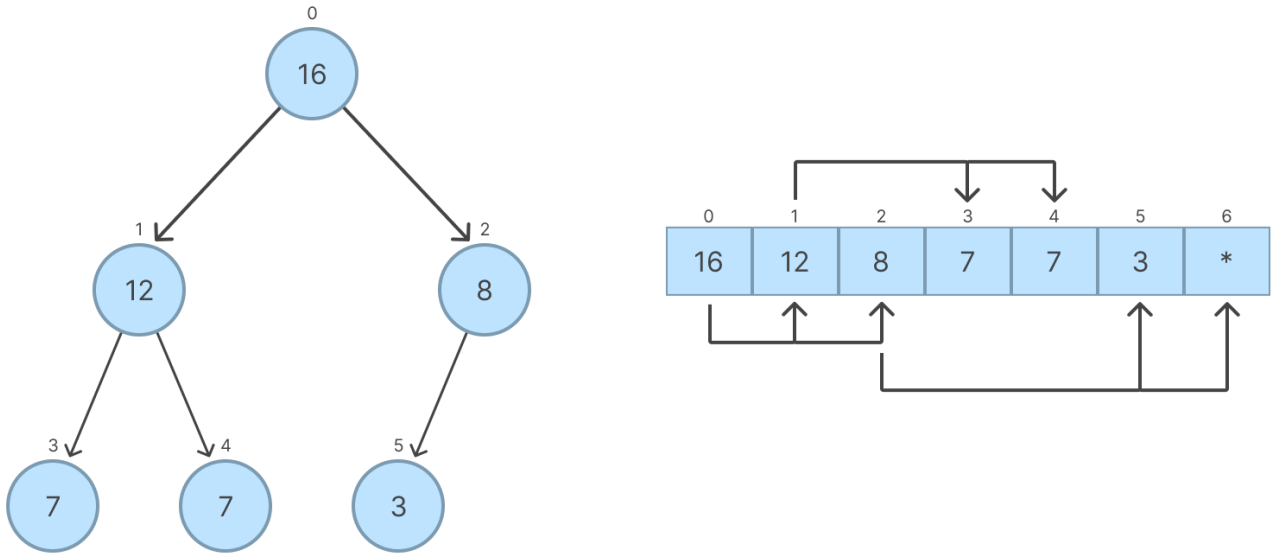
Figura 12 - Quarta inserção



Fonte: Autor (2023)

Insere-se o valor 3 a esquerda do nó 2. O resultado final da fila de prioridade pode ser visto na figura 13.

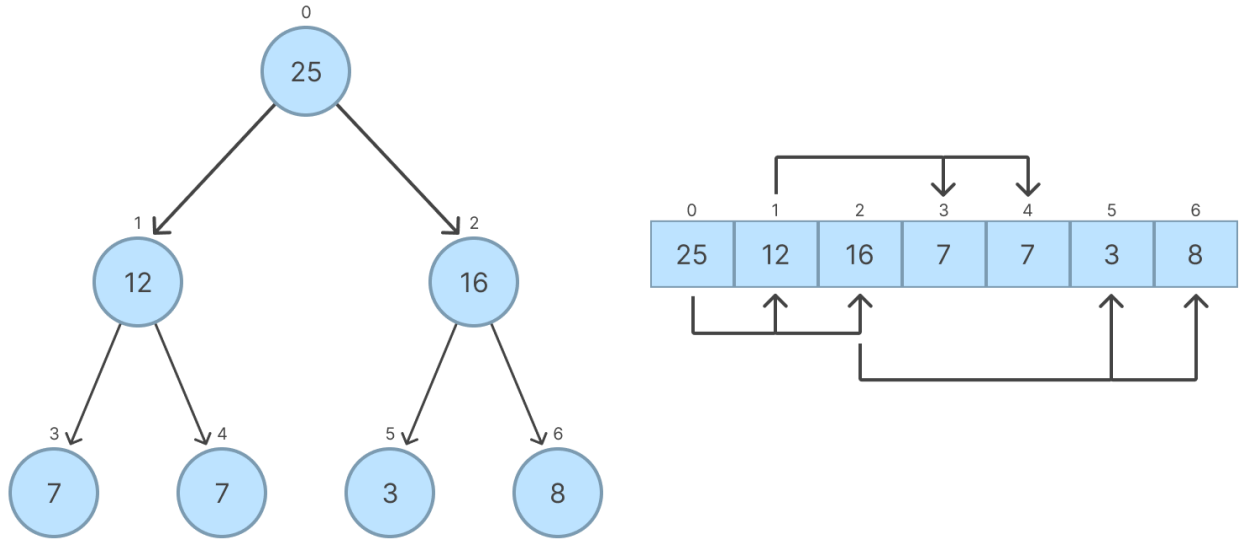
Figura 13 - Quinta inserção



Fonte: Autor (2023)

Enfim, o valor 25 é inserido no topo do *heap*. Após comparações com o 8 e 16, é movido para a raiz. O resultado da inserção pode ser visualizado na figura 14.

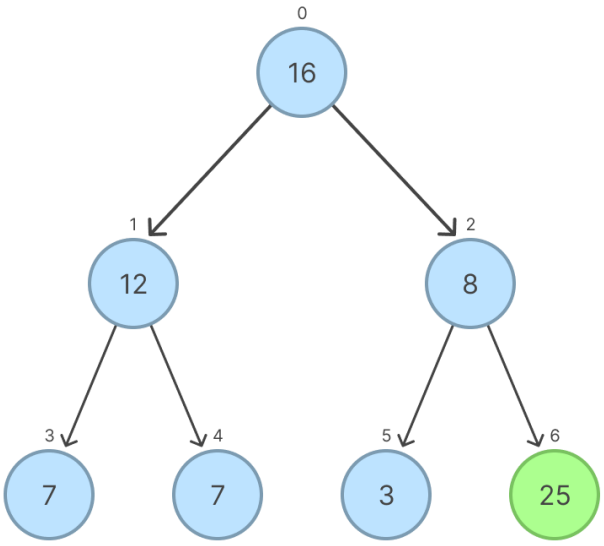
Figura 14 - Sexta inserção



Fonte: Autor (2023)

Para melhor compreensão da inserção de um elemento, será apresentado o passo a passo que ocorreu com a inserção do elemento 25 na árvore apresentada na figura 15. Primeiramente, o elemento 25 é inserido no nó livre mais à esquerda do último nível. Pode ser visualizado na figura 15.

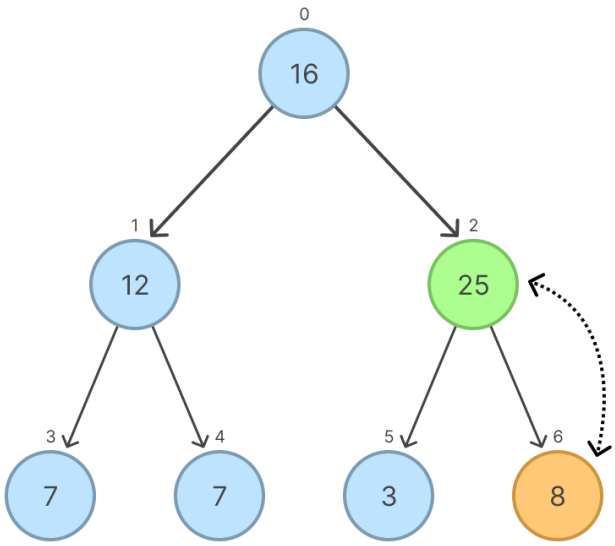
Figura 15 - Início da inserção



Fonte: Autor (2023)

Logo depois, troca-se o nó 25 com o seu pai, o nó número 2, de modo a manter a ordenação do *heap*, pois o valor 25 é maior que o valor 8. O resultado pode ser visto na figura 16.

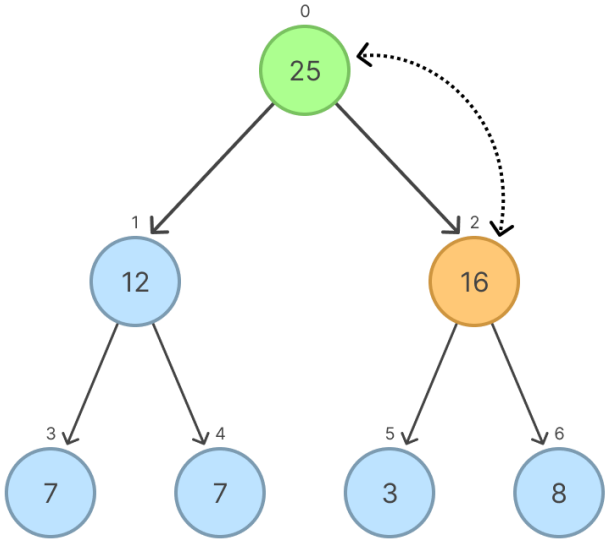
Figura 16 - Trocas



Fonte: Autor (2023)

Por fim, troca-se o nó 25 com o seu pai, o nó número 0, de modo a manter a ordenação do heap, pois o valor 25 é maior que o valor 16, como pode ser visto na figura 17.

Figura 17 - Fim da inserção



Fonte: Autor (2023)

5.3 REMOÇÃO DE ELEMENTOS

O pseudocódigo a seguir representa a lógica utilizada para realizar as operações de remoção de elementos no *heap* binário.

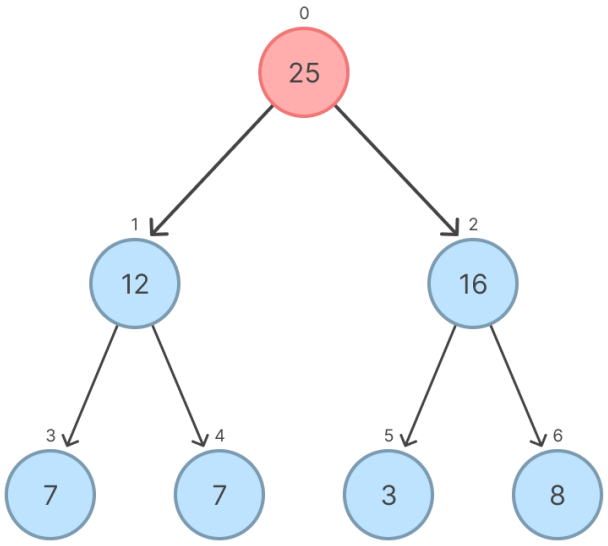
```
função pop_heap(heap)
    valor = heap.array[0]
    heap.size = heap.size - 1
    trocar(heap.array[0], heap.array[heap.size])

    i = 0

    repita
        mínimo = i
        esquerda = obter_nó_esquerdo(i)
        direita = obter_nó_direito(i)

        se existe_nó(heap, esquerda) e heap.compare(heap.array[mínimo], heap.array[esquerda]) então
            mínimo = esquerda

        se existe_nó(heap, direita) e heap.compare(heap.array[mínimo], heap.array[direita]) então
            mínimo = direita
```

```
        se i for igual a mínimo então
            interromper o loop

        trocar(heap.array[i], heap.array[mínimo])

    fim repita
    retornar valor
fim função
```

Esse algoritmo tem como objetivo remover o elemento de maior valor do *heap*. Primeiramente, armazena-se o valor do elemento na posição inicial para posterior retorno. O elemento na posição inicial é então substituído pelo último elemento do *heap*, mantendo a árvore completa. Após isso, inicia-se um processo de reorganização para restaurar as propriedades necessárias. Compara-se o elemento atual com seus filhos (esquerdo e direito) para determinar qual é o menor (ou maior) entre eles. Assim, encontra-se o índice do nó que deverá ser trocado com o nó atual e, se necessário, realiza-se a troca. Se nenhum dos filhos violar o critério de ordem, isso significa que o heap foi restaurado e o laço é interrompido.

Para visualizarmos como ocorrem as remoções na fila de prioridade utilizando este algoritmo, realizaremos uma simulação dos procedimentos realizados e verificaremos o seu estado após cada inserção. Será realizado uma remoção na fila de prioridade construída no item 5.2, que pode ser visualizada na figura 17. Primeiramente, localizamos e salvamos o valor 25 que está na raiz 0, como pode ser visto na figura 18.

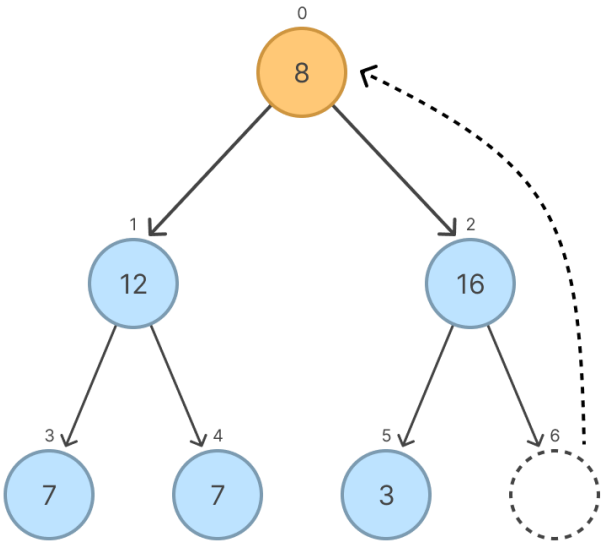
Figura 18 - Início da remoção



Fonte: Autor (2023)

Posteriormente, colocamos o valor do último nó (neste caso o nó número 6), na posição 0. Pode ser visualizado na figura 19.

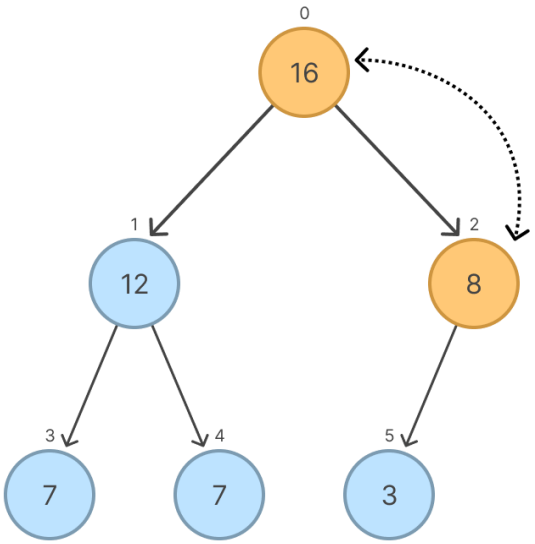
Figura 19 - Movendo último elemento para a raiz



Fonte: Autor (2023)

Por fim, realizamos a ordenação, trocando o nó de valor 8 com o nó de valor 16. Desta forma, mantemos as propriedades do *heap*. O resultado pode ser visualizado na figura 20.

Figura 20 - Organizando a fila de prioridade



Fonte: Autor (2023)

5.4 CONSULTA DO ELEMENTO DE MAIOR PRIORIDADE

O pseudocódigo a seguir representa a lógica utilizada para consultar o elemento de maior de prioridade da fila.

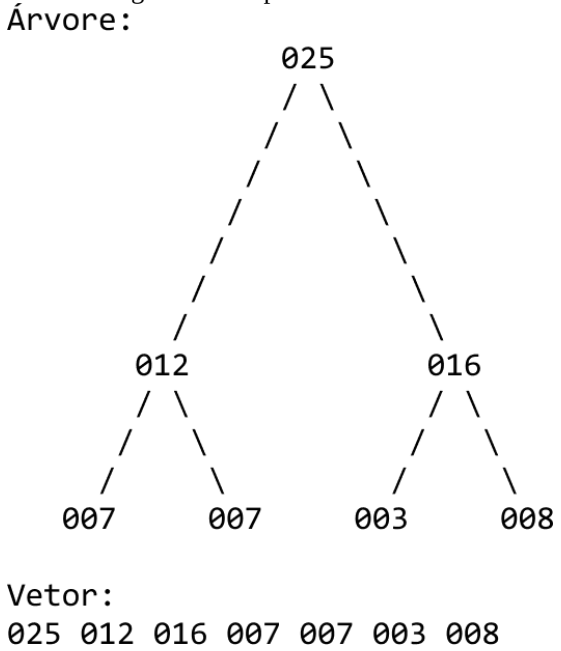
```
função peek_heap(heap)
    retornar heap.array[0]
fim função
```

Esse algoritmo acessa o elemento que está no topo da arvóre binária do *heap*, ou seja, na primeira posição da representação em vetor. A função retorna o elemento de maior (ou menor) valor dependendo do tipo de *heap*, sem removê-lo.

5.5 IMPRESSÃO DO HEAP BINÁRIO

A implementação realizada permite a apresentação de um *heap* binário de duas formas: uma exibição simplificada em formato de vetor e uma representação gráfica em forma de árvore. Isso permite visualizar a estrutura do *heap* e os valores armazenados, facilitando a compreensão de sua organização, como pode ser visto na figura 21. O código pode ser visualizado em: https://github.com/guikage/Trabalhos_ED_Lab2/blob/main/ed/t2/heap.c.

Figura 21 - Impressão da árvore



Fonte: Autor (2023)

CONCLUSÃO

A análise de filas de prioridade, em particular a implementação por meio de *heaps* binários, revela a importância e versatilidade desses conceitos na computação. Essas estruturas oferecem não apenas uma maneira eficiente de gerenciar e processar dados, mas também servem como alicerce para algoritmos vitais em diversas aplicações computacionais.

A comparação entre *min-heaps* e *max-heaps*, além das reflexões sobre o algoritmo de Dijkstra, evidencia não apenas a eficiência dessas estruturas, mas também os desafios que surgem em contextos específicos de aplicação. Essa compreensão acerca das filas de prioridade e *heaps* binários contribuem significativamente para estudantes e profissionais da área, promovendo conhecimentos sobre as complexidades inerentes aos sistemas computacionais. Ressaltando a importância contínua de sua implementação e aprimoramento em vários domínios da computação.

REFERÊNCIAS

BHATTACHARYYA, Subhajit; KARMAKAR, Mousumi. **Optimal Path Planning with Smart Energy Management Techniques Using Dijkstra's Algorithm**. In: Human-Centric Smart Computing: Proceedings of ICHCSC 2022. Singapore: Springer Nature Singapore, 2022

BRODAL, Gerth Stølting. A survey on priority queues. In: **Space-Efficient Data Structures, Streams, and Algorithms**: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013

CHEK, LEONG WEN. **Low Latency Extended Dijkstra Algorithm with Multiple Linear Regression for Optimal Path Planning of Multiple AGVs Network**. Engineering Innovations, v. 6, p. 31-36, 2023

CORMEN, T.H.; et al. **Introduction to algorithms.** [S.l.]: The MIT press 2001.

EDELKAMP, Stefan; ELMASRY, Amr; KATAJAINEN, Jyrki. **Optimizing binary heaps. Theory of Computing Systems,** v. 61, p. 606-636, 2017

FLOYD, R.W.: **Algorithm 245: Treesort3**. Commun, 1964

GITTINS, J. AND NASH, P. **Scheduling, queues and dynamic allocation indices**. In Prague Conference on Information Theory, Statistical Decision Functions and Random Processes, pages 191–202. Springer. (1977)

GUPTA, D. (2013). **Queueing models for healthcare operations**. In Handbook of Healthcare Operations Management, pages 19–44. Springer.

HOU, XUFANG. **Rural E-commerce Customer Path Planning Based on Dijkstra Algorithm**. In: 2022 International Conference on Knowledge Engineering and Communication Systems (ICKES). IEEE, 2022

IME-USP. (s.d.). **Fila de Prioridade**. Disponível em: https://www.ime.usp.br/~song/mac5710/slides/03prior.pdf.

JELČICOVÁ, Zuzana et al. **A Min-Heap-Based Accelerator for Deterministic On-the-Fly Pruning in Neural Networks**. In: 2023 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2023

KOHUTKA, Lukas; STOPJAKOVA, Viera. Heap Queue: **A Novel Efficient Hardware Architecture of MIN/MAX Queues for Real-Time Systems**. In: 2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS). IEEE, 2018.

KOSTYUKOV, Vladimir. **A Functional Approach to Standard Binary Heaps**. arXiv preprint arXiv:1312.4666, 2013.

MARTINS, G. HIG: **Pacientes mais graves tem prioridade no atendimento**. Retirado de: https://ilhacarioca.com/hig-pacientes-mais-graves-tem-prioridade-no-atendimento/.2021.

MILLER JR, Rupert G. **Priority queues. The Annals of Mathematical Statistics**, v. 31, n. 1, p. 86-103, 1960.

SACK, J.R., STROTHOTTE, T.: **An algorithm for merging heaps**. Acta Inf. 1985

SZCZEŚNIAK, IRENEUSZ; WOŹNA-SZCZEŚNIAK, Bożena. **Generic Dijkstra: correctness and tractability**. In: NOMS 2023 IEEE/IFIP Network Operations and Management Symposium. IEEE, 2023

UR REHMAN, ATTA & AWUAH-OFFEI, KWAME & BAKER, D. & BRISTOW, DOUGLAS .**Emergency Evacuation Guidance System For Underground Miners**. (2019).

WILLIAMS, J.W.J.: **Algorithm 232: Heapsort**. Commun.1964

ZAPATA-CARRATALA, Carlos; ARSIWALLA, Xerxes D.; BEYNON, Taliesin. **Heaps of Fish: arrays, generalized associativity and heapoids**. arXiv preprint arXiv:2205.05456, 2022.