Estruturas de Dados I

Filas de Prioridade

Prof. Bruno Azevedo

Instituto Federal de São Paulo

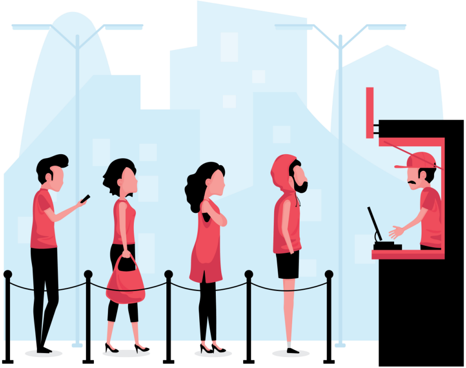
INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Catanduva

Filas de Prioridade

- Filas de prioridade são filas nas quais cada elemento possui uma prioridade associada.
- A prioridade determina a ordem de atendimento dos elementos. Ou seja, quando eles saem da fila de prioridade.
- Elementos de maior prioridade são atendidos antes dos elementos de menor prioridade.
- Se dois elementos têm a mesma prioridade, eles são atendidos na ordem em que foram enfileirados.
- A prioridade de um elemento pode ser o valor de sua chave, ou não, dependendo da implementação.

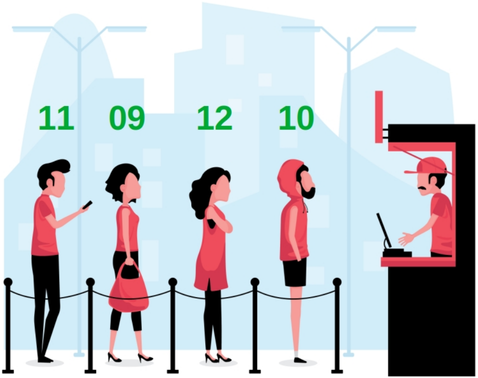
Filas de Prioridade

- Em uma fila tradicional, o elemento mais antigo na fila será o primeiro a sair (ou ser atendido).



Filas de Prioridade

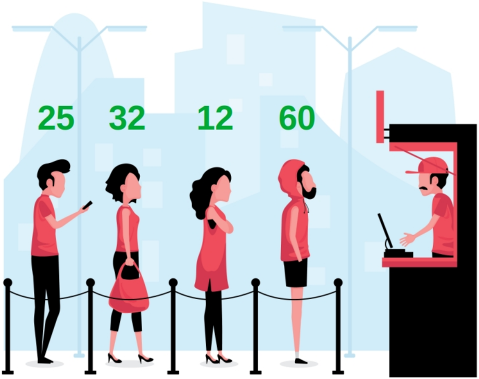
- Em uma fila de prioridade, a prioridade do elemento determina quando ele sairá, sendo comparada Às prioridades dos outros elementos da fila.



- Neste exemplo, a segunda pessoa será a próxima a ser atendida.

Filas de Prioridade

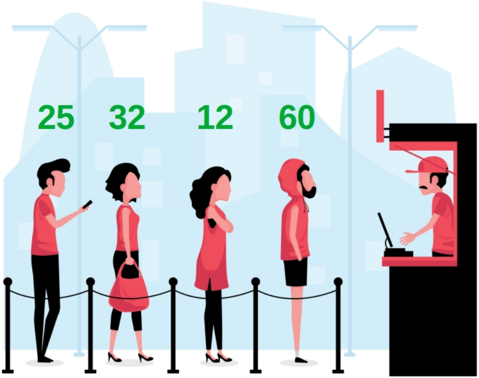
- Mas percebam que a prioridade deve ser comparada às prioridades dos outros elementos da fila.
- Então este valor 12 pode não ser tão alto em uma fila com elementos de prioridades maiores.



- Neste exemplo, a segunda pessoa será a última a ser atendida.

Filas de Prioridade

- Portanto, o elemento de **maior prioridade** da fila inteira será sempre o próximo elemento a ser atendido (sair) da fila.
- Utilizaremos a ordem de chegada como critério de desempate.



Usos de Filas de Prioridade

- São usadas em diversos contextos.
- Temos aplicações óbvias que envolvem a necessidade de uma fila de prioridades como a alocação de tarefas em sistemas operacionais e sistemas de mensagens em redes de computadores.
- Mas podemos destacar seu uso em implementações eficientes de diversos algoritmos clássicos da computação, como por exemplo:
  - Algoritmo de Dijkstra: Algoritmo para encontrar o caminho mínimo em um grafo ponderado não direcionado.

  $\rightarrow$ O problema de caminho mínimo consiste na minimização do custo de travessia de um grafo entre dois nós, custo este dado pela soma dos pesos de cada aresta percorrida.
  - Algoritmo de Prim: Algoritmo para encontrar árvore geradora mínima em um grafo conectado, valorado e não direcionado.

  $\rightarrow$ O problema de encontrar uma arvore geradora mínima consiste em encontrar um subgrafo, o qual é uma árvore, que conecta todos os vértices do grafo com custo total mínimo.
  - Codificação de Huffman: Algoritmo para compressão de dados sem perdas.

Como Implementar?

- Uma abordagem ingênua seria utilizar um vetor e efetuar deslocamentos de elementos para garantir que o elemento de maior prioridade sempre esteja na primeira posição.
- Vamos olhar essa implementação.

Implementação Ingênua

- Definimos estruturas para representar os elementos da fila e a própria fila.

```c
#define COMPRIMENTO_TOTAL_FILA 256

typedef struct {
    int valor;
    int prioridade;
} Elemento;

typedef struct {
    Elemento vetor[COMPRIMENTO_TOTAL_FILA];
    int tamanho; // Tamanho atual da fila
} FilaDePrioridade;
```

Implementação Ingênua

- Criamos uma função para criar uma fila de prioridade.

```
FilaDePrioridade* criarFilaDePrioridade() {
    FilaDePrioridade* fila = new FilaDePrioridade;
    fila->tamanho = 0;
    return fila;
}
```

Implementação Ingênua

- Também precisamos de uma função para inserir elementos na fila, ou seja, a operação de **enfileirar**.

```
int enfileirar(FilaDePrioridade* fila, int valor, int prioridade) {
    if(fila->tamanho >= COMPRIMENTO_TOTAL_FILA) {
        cout << "Erro: Fila de prioridade cheia." << endl;
        return -1;
    }
    int i = fila->tamanho;
    // Encontra a posição correta para inserir o novo elemento
    while(i > 0 && fila->vetor[i - 1].prioridade < prioridade) {
        fila->vetor[i] = fila->vetor[i - 1];
        i--;
    }
    // Insere o novo elemento
    fila->tamanho++;
    fila->vetor[i].valor = valor;
    fila->vetor[i].prioridade = prioridade;
}
```

Implementação Ingênua

- Por último, vamos criar uma função para remover elementos na fila, ou seja, a operação de **desinfileirar**.

```
Elemento desinfileirar(FilaDePrioridade* fila) {
    if(fila->tamanho <= 0) {
        cout << "Erro: Fila de prioridade vazia." << endl;
        Elemento elementoVazio = {0, 0};
        return elementoVazio;
    }
    // Remove o elemento de maior prioridade (primeiro do vetor)
    Elemento maiorPrioridade = fila->vetor[0];
    // Desloca os elementos restantes para preencher a lacuna
    for(int i = 1;i < fila->tamanho;i++) {
        fila->vetor[i - 1] = fila->vetor[i];
    }
    fila->tamanho--;
    return maiorPrioridade;
}
```

Avaliando a Implementação Ingênua

- É uma implementação eficiente?
- Podemos criar uma implementação mais eficiente?
- Qual a complexidade computacional de cada operação nesta implementação?
- Vamos iniciar abordando essa última pergunta.

Avaliando a Implementação Ingênua

- Vamos avaliar a operação de enfileiramento.
- No pior caso, quantos deslocamentos (cópias) precisaremos fazer?

```
int enfileirar(FilaDePrioridade* fila, int valor, int prioridade) {
    if(fila->tamanho >= COMPRIMENTO_TOTAL_FILA) {
        cout << "Erro: Fila de prioridade cheia." << endl;
        return -1;
    }
    int i = fila->tamanho;
    // Encontra a posição correta para inserir o novo elemento
    while(i > 0 && fila->vetor[i - 1].prioridade < prioridade) {
        fila->vetor[i] = fila->vetor[i - 1];
        i--;
    }
    // Insere o novo elemento
    fila->tamanho++;
    fila->vetor[i].valor = valor;
    fila->vetor[i].prioridade = prioridade;
    return 0;
}
```

Avaliando a Implementação Ingênua

- Precisaremos efetuar *n* deslocamentos, onde *n* é o tamanho da fila.
- Isto ocorrerá quando o elemento inserido tiver prioridade maior que qualquer elemento já existente na fila de prioridade.
- Ou seja, ele será inserido na primeira posição da fila de prioridade.

```
int enfileirar(FilaDePrioridade* fila, int valor, int prioridade) {
    if(fila->tamanho >= COMPRIMENTO_TOTAL_FILA) {
        cout << "Erro: Fila de prioridade cheia." << endl;
        return -1;
    }
    int i = fila->tamanho;
    // Encontra a posição correta para inserir o novo elemento
    while(i > 0 && fila->vetor[i - 1].prioridade < prioridade) {
        fila->vetor[i] = fila->vetor[i - 1];
        i--;
    }
    // Insere o novo elemento
    fila->tamanho++;
    fila->vetor[i].valor = valor;
    fila->vetor[i].prioridade = prioridade;
    return 0;
}
```

Avaliando a Implementação Ingênua

- Vamos avaliar a operação de desenfileiramento.
- Qual o pior caso nesta operação? Ela tem um pior caso?

```
Elemento desinfileirar(FilaDePrioridade* fila) {
    if(fila->tamanho <= 0) {
        cout << "Erro: Fila de prioridade vazia." << endl;
        Elemento elementoVazio = {0, 0};
        return elementoVazio;
    }
    // Remove o elemento de maior prioridade (primeiro do vetor)
    Elemento maiorPrioridade = fila->vetor[0];
    // Desloca os elementos restantes para preencher a lacuna
    for(int i = 1;i < fila->tamanho;i++) {
        fila->vetor[i - 1] = fila->vetor[i];
    }
    fila->tamanho--;
    return maiorPrioridade;
}
```

Avaliando a Implementação Ingênua

- Nesta operação, sempre efetuaremos *n* deslocamentos.
- Afinal, o primeiro elemento sempre será o elemento removido.

```
Elemento desinfileirar(FilaDePrioridade* fila) {
    if(fila->tamanho <= 0) {
        cout << "Erro: Fila de prioridade vazia." << endl;
        Elemento elementoVazio = {0, 0};
        return elementoVazio;
    }
    // Remove o elemento de maior prioridade (primeiro do vetor)
    Elemento maiorPrioridade = fila->vetor[0];
    // Desloca os elementos restantes para preencher a lacuna
    for(int i = 1;i < fila->tamanho;i++) {
        fila->vetor[i - 1] = fila->vetor[i];
    }
    fila->tamanho--;
    return maiorPrioridade;
}
```

Avaliando a Implementação Ingênua

- Temos n deslocamentos no pior caso para a operação de enfileiramento.
- E sempre teremos n deslocamentos na operação de desinfileiramento, portanto, este é o pior e melhor caso.
- E qual o melhor caso na operação de enfileiramento?

Avaliando a Implementação Ingênua

- Temos n deslocamentos no pior caso para a operação de enfileiramento.
- E sempre teremos n deslocamentos na operação de desinfileiramento, portanto, este é o pior e melhor caso.
- E qual o melhor caso na operação de enfileiramento? Zero deslocamentos se a fila está vazia (fila->tamanho == 0)

Poderíamos Utilizar Ordenação?

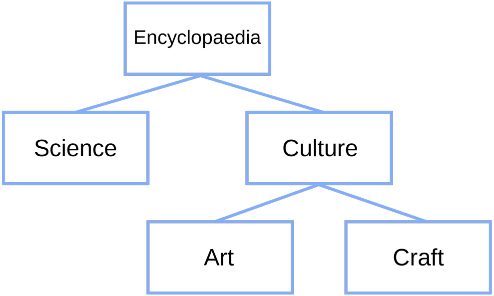
- Uma pergunta que poderiam fazer é: não seria melhor ordenar a fila a cada enfileiramento de um novo elemento?
- A resposta é não: algoritmos de ordenação possuem a complexidade de pior caso em $O(n \cdot \log_2 n)$.
- E o desinfileiramento necessitaria de n - 1 cópias para ajuste no vetor (assumindo o uso de um vetor, como fizemos na abordagem anterior).

Existe Uma Alternativa Mais Eficiente?

- Voltando a nossa implementação ingênua, ela não possui uma complexidade computacional alta, não é exponencial ou fatorial. É uma complexidade linear.
- Mas isso pode ser melhorado.
- Para esse fim, aprenderemos uma nova estrutura de dados que será usada na implementação da fila de prioridade.
- Mas antes, precisamos conhecer **Árvores**.

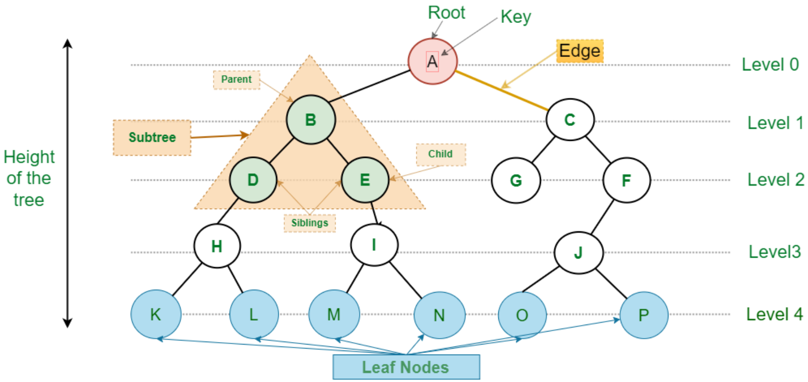
Árvores

- Uma estrutura em forma de árvore é uma maneira de representar a natureza hierárquica de uma estrutura em uma forma gráfica.
- Possuem este nome porque a representação se parece com uma árvore, mas de cabeça para baixo. As folhas encontram-se na parte inferior.
- Este tipo de representação não é restrita a computação.



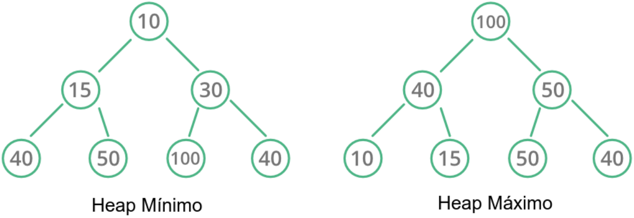
Árvores

- Na computação, uma árvore é um tipo de dado abstrato que representa uma estrutura hierárquica em forma de árvore com um conjunto de nós conectados.
- Cada nó na árvore pode estar conectado a múltiplos filhos, mas deve estar conectado exatamente a um pai.
- Exceto pelo nó raiz, que não tem pai.



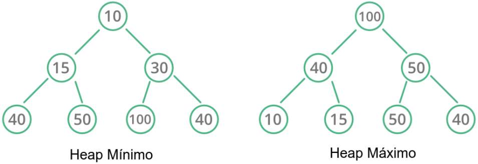
Heap

- O **Heap** é uma estrutura de dados baseada em árvore que satisfaz a propriedade de Heap.
- Em um Heap Máximo, para qualquer nó filho, a chave (valor) do nó pai é maior ou igual a chave do filho.
- Em um Heap Mínimo, a chave do pai é menor ou igual à chave do filho.
- O nó no topo do Heap é chamado de nó raiz.

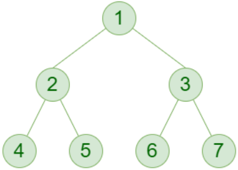


Heap

- Existem vários tipos de Heap, este exemplificado na figura abaixo é um **Heap Binário**.



- Um Heap Binário é um Heap que assume a forma de uma árvore binária.
- Uma árvore binária é uma estrutura de dados em árvore na qual cada nó tem no máximo dois filhos.

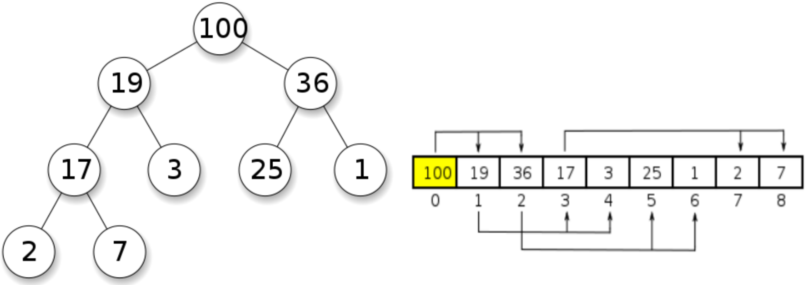


Heap

- Voltando a fila de prioridade, podemos implementar essa estrutura de modo mais eficiente do que a implementação ingênua utilizando um Heap.
- Utilizaremos um Heap Binário para este fim, mas existem implementações ainda mais eficientes utilizando outros tipos de Heap.
- Heaps geralmente são implementados utilizando um vetor, o que é interessante para iniciantes em Estrutura de Dados.
- Afinal, vetores são estruturas que programadores iniciantes geralmente estão acostumados a utilizar.

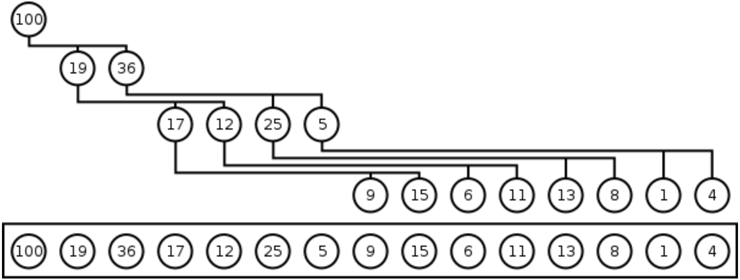
Heap

- Abaixo podemos visualizar como um Heap Binário é armazenado em um vetor.



Heap

- Outra visualização.



Heap

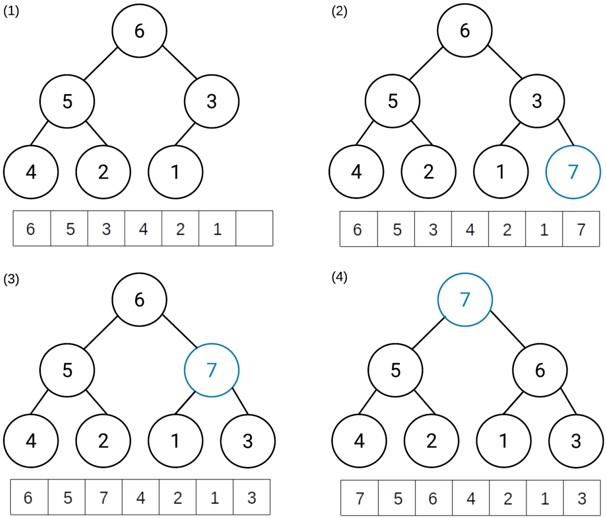
- Utilizaremos um Heap Máximo em nossa implementação.
- Existem diversas operações possíveis em um Heap, mas iremos nos concentrar em duas:
  - Inserção de um novo elemento no Heap.
  - Extração do elemento de valor máximo (ou seja, a raiz).
- São operações que utilizaremos na fila de prioridade.

Inserção em um Heap

- A inserção em um Heap Binário Máximo segue os seguintes passos:
  1. O novo elemento é adicionado ao final do vetor que representa o Heap.
  2. Em seguida, o elemento é comparado com seu pai. Se o novo elemento for maior que seu pai, eles são trocados.
  3. Esse processo é repetido até que o elemento esteja na posição correta ou até que ele se torne a raiz do heap.

Inserção em um Heap

- Vamos inserir o número 7 no Heap abaixo, sendo ilustrado em sua representação de árvore e como vetor.



Inserção em um Heap

- Vamos implementar essa operação.
- Definimos estruturas para representar os elementos do Heap e o próprio Heap.
- Também criamos uma função para trocar dois nós de posição.

```c
#define COMPRIMENTO_TOTAL_HEAP 256

typedef struct No {
    int chave; // Chave do nó
} No;

typedef struct Heap {
    No Nos[COMPRIMENTO_TOTAL_HEAP];
    int tamanho; // Tamanho atual do heap
} Heap;

void troca(No *x, No *y) {
    No temp = *x;
    *x = *y;
    *y = temp;
}
```

Inserção em um Heap

- A operação de inserção em um Heap Binário Máximo.

```
int insereHeap(Heap *heap, int valor) {
    if(heap->tamanho >= COMPRIMENTO_TOTAL_HEAP) {
        printf("Heap cheio. Não é possível inserir mais elementos.\n");
        return -1;
    }
    int i = heap->tamanho;
    heap->Nos[i].chave = valor;
    heap->tamanho++;
    // Ajuste para manter a propriedade do heap
    while(i > 0 && heap->Nos[(i-1)/2].chave < heap->Nos[i].chave) {
        troca(&heap->Nos[i], &heap->Nos[(i-1)/2]);
        i = (i-1)/2;
    }
    return 0;
}
```

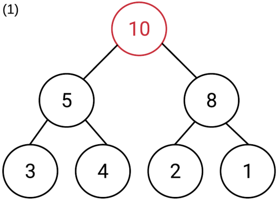
Extração do elemento de valor máximo em um Heap

- A extração do elemento de valor máximo em um Heap Binário Máximo segue os seguintes passos:
    1. Substituir a raiz pelo último nó do Heap de modo a manter a estrutura de árvore binária.
    2. O novo nó raiz pode violar a propriedade de Heap, portanto, devemos efetuar a operação *heapify-down*. Comparamos o novo nó raiz com seus filhos e, se necessário, trocamos o novo nó raiz com o filho máximo, garantindo que o maior valor esteja na posição correta.
    3. Esse processo é repetido até a propriedade do Heap máximo seja restaurada, ou seja, até que o nó atual seja maior ou igual aos seus filhos.

Extração do elemento de valor máximo em um Heap

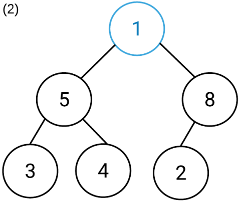
- Vamos extrair o elemento máximo do Heap abaixo (sua raiz), sendo ilustrado em sua representação de árvore e como vetor.

(1)



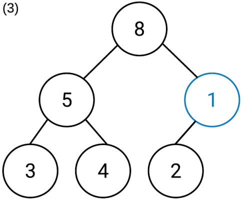
10	5	8	3	4	2	1

(2)



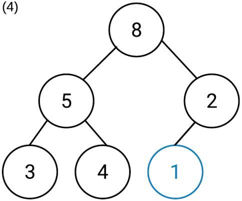
1	5	8	3	4	2	

(3)



8	5	1	3	4	2	

(4)



8	5	2	3	4	1	

Extração do elemento de valor máximo em um Heap

- A operação de extração do elemento de valor máximo em um Heap;

```
No extraiMaximo(Heap *heap) {
    No raiz;
    if(heap->tamanho <= 0) {
        printf("Heap vazio. Não é possível extrair máximo.\n");
        raiz.chave = -1;
        return raiz;
    }
    // Salvar a raiz removida
    raiz = heap->Nos[0];
    // Substituir a raiz pelo último nó do heap
    heap->Nos[0] = heap->Nos[--heap->tamanho];
    // Aplicar Max-Heapify para restaurar a propriedade do heap
    maxHeapify(heap->Nos, heap->tamanho, 0);
    return raiz;
}
```

- Como a substituição da raiz pelo último nó temos uma estrutura em árvore mas a propriedade de Heap pode estar violada.
- A função maxHeapify restaura a propriedade de Heap máximo.

Extração do elemento de valor máximo em um Heap

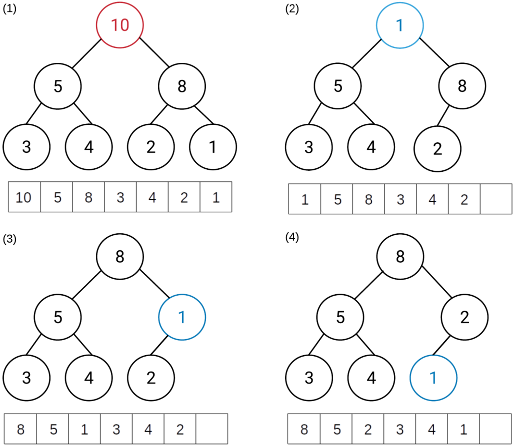
- A função maxHeapify.
- Notem que trocamos cada nó que for menor que seus filhos por seu filho máximo, garantindo que o maior valor esteja na posição correta.

```
void maxHeapify(No* A, int tamanho, int i) {
    int esquerda = 2 * i + 1;
    int direita = 2 * i + 2;
    int maior = i;
    if(esquerda < tamanho && A[esquerda].chave > A[maior].chave)
        maior = esquerda;
    if(direita < tamanho && A[direita].chave > A[maior].chave)
        maior = direita;
    if(maior != i) {
        troca(&A[i], &A[maior]);
        maxHeapify(A, tamanho, maior);
    }
}
```

- A função maxHeapify é **recursiva**.
- Ela é chamada recursivamente para o Nó filho que foi trocado para garantir que a propriedade do Heap máximo seja mantida nessa subárvore.

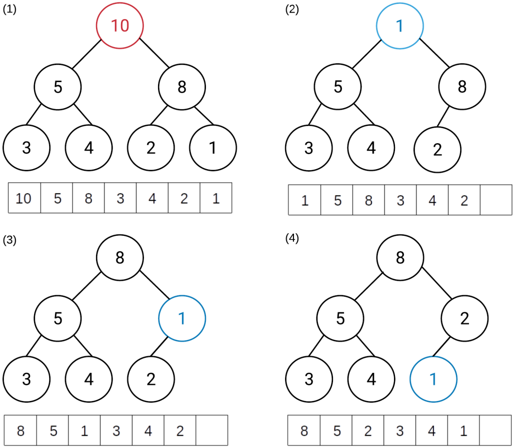
Extração do elemento de valor máximo em um Heap

- Por exemplo, no Heap da figura 2, o “maior” será o nó de valor 8, o elemento da direita.
- Este é maior que o elemento pai, o nó de valor 1. Portanto, é efetuada a troca entre os dois (figura 3).
- Em seguida, a função maxHeapify é chamada recursivamente para o nó de valor 1.
- Como resultado, haverá mais uma troca, do elemento “esquerda”, de valor 2, e do nó de valor 1.



Extração do elemento de valor máximo em um Heap

- Notem também que a última chamada de maxHeapify não terá efeito, já que não entrará em nenhum dos if, já que este não terá filhos.



Implementação com Heap da Fila de Prioridade

- Devem ter percebido que o Heap Binário Máximo implementa perfeitamente a fila de prioridade.
- Encontramos facilmente o elemento de maior prioridade em um Heap, sendo a raiz.
- Portanto, podemos usar o Heap para implementar a fila de prioridade, em vez do vetor da implementação ingênua.
- Vamos avaliar a **eficiência** desta implementação.

Avaliando a Implementação Com Heap Binário Máximo

- Na operação de enfileirar, temos que efetuar diversas trocas para garantir a propriedade de Heap a cada inserção.
- Quantas trocas precisaremos efetuar no pior caso, considerando que temos *n* elementos em nosso Heap?

```
int insereHeap(Heap *heap, int valor) {
    if(heap->tamanho >= COMPRIMENTO_TOTAL_HEAP) {
        printf("Heap cheio. Não é possível inserir mais elementos.\n");
        return -1;
    }
    int i = heap->tamanho;
    heap->Nos[i].chave = valor;
    heap->tamanho++;
    // Ajuste para manter a propriedade do heap
    while(i > 0 && heap->Nos[(i-1)/2].chave < heap->Nos[i].chave) {
        troca(&heap->Nos[i], &heap->Nos[(i-1)/2]);
        i = (i-1)/2;
    }
    return 0;
}
```

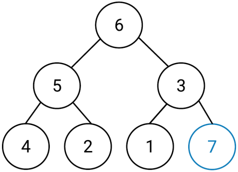
Avaliando a Implementação Com Heap Binário Máximo

- Na operação de enfileirar, podemos ter que efetuar diversas trocas para garantir a propriedade de Heap a cada inserção.
- Quantas trocas precisaremos efetuar no pior caso, considerando que temos n elementos em nosso Heap? $\log_2 n$ trocas. Vamos entender isso.

```c
int insereHeap(Heap *heap, int valor) {
    if(heap->tamanho >= COMPRIMENTO_TOTAL_HEAP) {
        printf("Heap cheio. Não é possível inserir mais elementos.\n");
        return -1;
    }
    int i = heap->tamanho;
    heap->Nos[i].chave = valor;
    heap->tamanho++;
    // Ajuste para manter a propriedade do heap
    while(i > 0 && heap->Nos[(i-1)/2].chave < heap->Nos[i].chave) {
        troca(&heap->Nos[i], &heap->Nos[(i-1)/2]);
        i = (i-1)/2;
    }
    return 0;
}
```

Avaliando a Implementação Com Heap Binário Máximo

- No exemplo ilustrado na figura, a inserção do 7 resulta em duas trocas, com o 3 e com o 6.

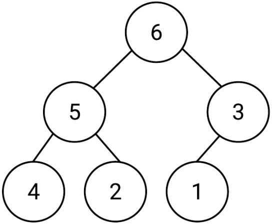


6	5	3	4	2	1	7

- Ou seja, o pior caso será equivalente ao número de arestas da maior distância existente entre um nó folha e a raiz.
- E qual será essa distância, dado um Heap de n elementos?

Avaliando a Implementação Com Heap Binário Máximo

- Vamos entender a altura do Heap.
- Um heap binário é uma árvore binária completa, ou seja, todos os níveis da árvore estão totalmente preenchidos, exceto possivelmente o último nível, que é preenchido da esquerda para a direita.



- Considere os níveis iniciando em zero na raiz. Em um nível i do Heap pode haver até 2^i nós.
- Ou seja, temos um crescimento exponencial de nós a cada nível do Heap.

Avaliando a Implementação Com Heap Binário Máximo

- Queremos descobrir, dado um número de *n* nós de um Heap, qual será o número máximo de níveis que ele pode possuir.
- Com esta informação, saberemos o número máximo de trocas, que será o número de níveis menos um.

Avaliando a Implementação Com Heap Binário Máximo

- Sabemos que cada nível i terá até 2^i nós. Portanto, se temos k níveis em um Heap, teremos $2^0 + 2^1 + 2^2 + \ldots + 2^{k-1}$ nós no Heap.
- Isso é a soma de uma progressão geométrica (P.G.) finita. A soma dos termos de uma P.G. é dada por:

$$S = a_1 \left(\frac{r^n - 1}{r - 1} \right)$$

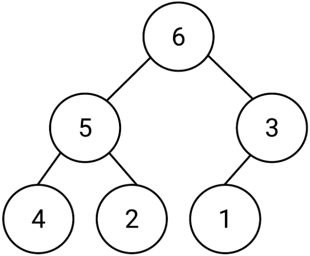
- Onde a_1 é o primeiro termo, r é a razão da P.G., e n é o número de termos, respectivamente.
- Substituindo na fórmula:

$$S = 2^0 \left(\frac{2^k - 1}{2 - 1} \right) = 2^k - 1$$

- Portanto, um Heap de k níveis possui até $2^k - 1$ nós.
- Exemplificando, se um Heap possui 3 níveis, ele terá até $2^3 - 1 = 7$ nós.

Avaliando a Implementação Com Heap Binário Máximo

- Se um Heap possui 3 níveis, ele terá até $2^3 - 1 = 7$ nós.
- Observem um Heap de 3 níveis.



Avaliando a Implementação Com Heap Binário Máximo

- Em um Heap de n nós: $n = 2^k - 1$, onde k é o número de níveis. Mas precisamos obter o valor de k em função de n.
- Para este fim, vamos isolar k.

$$n = 2^k - 1$$
$$n + 1 = 2^k$$

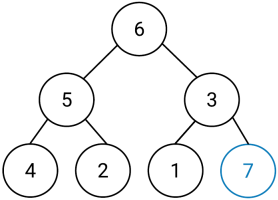
- Vocês devem lembrar que a operação logarítmica é a operação inversa da operação exponencial.
- A função logarítmica na base 2, denotada por $\log_2 n$, é a operação inversa de 2^n. Deste modo:

$$\log_2(n+1) = k$$

- Portanto, um heap com n nós possui $\log_2(n+1)$ níveis.
- Por exemplo, se um Heap possuir 7 nós, ele terá $\log_2 8 = 3$ níveis, porque $2^3 = 8$.

Avaliando a Implementação Com Heap Binário Máximo

- O número máximo de trocas que nosso algoritmo fará é o número de níveis menos 1.
- No exemplo ilustrado pela figura abaixo, a inserção do nó de valor 7 resultará em $\log_2(n+1)$ - $1 = 2$ trocas.



6	5	3	4	2	1	7

- No pior caso, o nosso algoritmo, utilizando Heap para implementar a fila de prioridade, fará $\log_2(n+1)$ - 1 trocas.
- Isso é melhor que n trocas. Tempo logarítimico é significativamente melhor que tempo linear.

Avaliando a Implementação Com Heap Binário Máximo

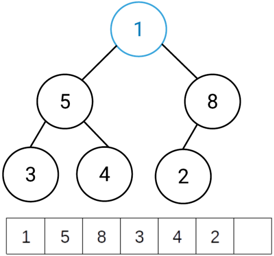
- Na operação de desinfileirar, podemos ter que efetuar diversas trocas para garantir a propriedade de Heap após a remoção da raiz.
- Quantas trocas precisaremos efetuar no pior caso, considerando que temos *n* elementos em nosso Heap? Vamos voltar ao nosso exemplo anterior.

```
No extraiMaximo(Heap *heap) {
  No raiz;
  if(heap->tamanho <= 0) {
      printf("Heap vazio.
      Não é possível extrair
      máximo.\n");
      raiz.chave = -1;
      return raiz;
  }
  raiz = heap->Nos[0];
  heap->Nos[0] = heap->Nos[--heap->tamanho];
  maxHeapify(heap->Nos, heap->tamanho, 0);
  return raiz;
}
```

```
void maxHeapify(No* A, int tamanho, int i) {
  int esquerda = 2 * i + 1;
  int direita = 2 * i + 2;
  int maior = i;
  if(esquerda < tamanho && A[esquerda].chave
     > A[maior].chave)
      maior = esquerda;
  if(direita < tamanho && A[direita].chave
     > A[maior].chave)
      maior = direita;
  if(maior != i) {
      troca(&A[i], &A[maior]);
      maxHeapify(A, tamanho, maior);
  }
}
```

Avaliando a Implementação Com Heap Binário Máximo

- No exemplo ilustrado na figura, a remoção da raiz e sua substituição pelo último elemento do vetor resulta em duas trocas, com o 8 e com o 2.



- O pior caso será equivalente ao número de arestas da maior distância existente entre um nó folha e a raiz.
- E já sabemos qie essa distância é o número de níveis menos um, que é dado por: $\log_2(n+1)$ - 1.
- Ou seja, no pior caso faremos o mesmo número de trocas que na operação de enfileiramento.

Comparando as Implementações

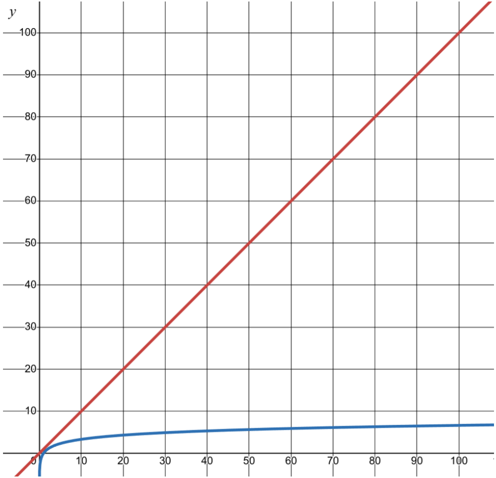
- Comparando as implementações da fila de prioridade usando as estruturas de dados Vetor e Heap Binário.

ED	Enfileiramento	Desinfileiramento
Vetor	n	n
Heap Binário	$\log_2(n+1)$ - 1	$\log_2(n+1)$ - 1

- Avaliando o pior caso, uma possui complexidade linear para ambas as operações e a outra possui complexidade logarítmica para ambas as operações.

Comparando as Implementações

- No gráfico abaixo podem observar que o crescimento logarítimico é muito menor que o crescimento linear.



- O eixo y representa *n*, o número de elementos da fila de prioridade, e o eixo x representa o número de trocas ou deslocamentos efetuados pelo algoritmo no pior caso.

Exercícios

- Escreva uma função para mesclar duas filas de prioridade, implementadas em um Heap Binário, em uma única fila de prioridade mantendo a ordem dos elementos de acordo com suas prioridades.
- Escreva uma função para imprimir os elementos de uma fila de prioridade, implementadas em um Heap Binário, em ordem de prioridade.
- Escreva uma função para imprimir os elementos de uma fila de prioridade, implementadas em um Heap Binário, em ordem reversa de prioridade. O elemento deve ser extraído diretamente do Heap, ou seja, não extraia todos os elementos e os ordene de forma crescente.
- Escreva uma função para atualizar a chave de um nó em um Heap Binário (aumentar ou decrementar).
- Escreva um texto explicando o funcionamento do Heap Binomial (pesquise), e comparando-o com o Heap Binário.