

Estrutura de Dados

Paulo Torrens

paulotorrens@gnu.org

Departamento de Ciência da Computação
Centro de Ciências e Tecnologias
Universidade do Estado de Santa Catarina

2020/1

- Variáveis dentro de um programa são classificadas pelo seu **tipo**: inteiros, caracteres, *strings*...
- Frequentemente em programas há a necessidade de se salvar **coleções** (do inglês, *containers*) de dados em uma única variável (e.g., um vetor de inteiros, uma lista de usuários, etc)
- Muitas vezes, essas coleções precisam ser feitas de forma **dinâmica** dentro de um sistema
 - Não se sabe quantos objetos existirão dentro da coleção em tempo de compilação, ou o número de objetos será alterado durante a execução do programa
 - Precisamos da capacidade de **adicionar** e **remover** objetos dessa coleção
 - Em linguagens como C, tais coleções podem ser representados na forma de **tipos de dados abstratos**

- Variáveis dentro de um programa são classificadas pelo seu **tipo**: inteiros, caracteres, *strings*...
- Frequentemente em programas há a necessidade de se salvar **coleções** (do inglês, *containers*) de dados em uma única variável (e.g., um vetor de inteiros, uma lista de usuários, etc)
- Muitas vezes, essas coleções precisam ser feitas de forma **dinâmica** dentro de um sistema
 - Não se sabe quantos objetos existirão dentro da coleção em tempo de compilação, ou o número de objetos será alterado durante a execução do programa
 - Precisamos da capacidade de **adicionar** e **remover** objetos dessa coleção
 - Em linguagens como C, tais coleções podem ser representados na forma de **tipos de dados abstratos**

- Variáveis dentro de um programa são classificadas pelo seu **tipo**: inteiros, caracteres, *strings*...
- Frequentemente em programas há a necessidade de se salvar **coleções** (do inglês, *containers*) de dados em uma única variável (e.g., um vetor de inteiros, uma lista de usuários, etc)
- Muitas vezes, essas coleções precisam ser feitas de forma **dinâmica** dentro de um sistema
 - Não se sabe quantos objetos existirão dentro da coleção em tempo de compilação, ou o número de objetos será alterado durante a execução do programa
 - Precisamos da capacidade de **adicionar** e **remover** objetos dessa coleção
 - Em linguagens como C, tais coleções podem ser representados na forma de **tipos de dados abstratos**

- Variáveis dentro de um programa são classificadas pelo seu **tipo**: inteiros, caracteres, *strings*...
- Frequentemente em programas há a necessidade de se salvar **coleções** (do inglês, *containers*) de dados em uma única variável (e.g., um vetor de inteiros, uma lista de usuários, etc)
- Muitas vezes, essas coleções precisam ser feitas de forma **dinâmica** dentro de um sistema
 - Não se sabe quantos objetos existirão dentro da coleção em tempo de compilação, ou o número de objetos será alterado durante a execução do programa
 - Precisamos da capacidade de **adicionar** e **remover** objetos dessa coleção
 - Em linguagens como C, tais coleções podem ser representados na forma de **tipos de dados abstratos**

- Variáveis dentro de um programa são classificadas pelo seu **tipo**: inteiros, caracteres, *strings*...
- Frequentemente em programas há a necessidade de se salvar **coleções** (do inglês, *containers*) de dados em uma única variável (e.g., um vetor de inteiros, uma lista de usuários, etc)
- Muitas vezes, essas coleções precisam ser feitas de forma **dinâmica** dentro de um sistema
 - Não se sabe quantos objetos existirão dentro da coleção em tempo de compilação, ou o número de objetos será alterado durante a execução do programa
 - Precisamos da capacidade de **adicionar** e **remover** objetos dessa coleção
 - Em linguagens como C, tais coleções podem ser representados na forma de **tipos de dados abstratos**

- Variáveis dentro de um programa são classificadas pelo seu **tipo**: inteiros, caracteres, *strings*...
- Frequentemente em programas há a necessidade de se salvar **coleções** (do inglês, *containers*) de dados em uma única variável (e.g., um vetor de inteiros, uma lista de usuários, etc)
- Muitas vezes, essas coleções precisam ser feitas de forma **dinâmica** dentro de um sistema
 - Não se sabe quantos objetos existirão dentro da coleção em tempo de compilação, ou o número de objetos será alterado durante a execução do programa
 - Precisamos da capacidade de **adicionar** e **remover** objetos dessa coleção
 - Em linguagens como C, tais coleções podem ser representados na forma de **tipos de dados abstratos**

Filas, pilhas e listas

- Dentre as coleções clássica estudadas, podemos citar as **filas** e as **pilhas**
- Uma fila representa uma estrutura de dados **FIFO** (do inglês, *first-in, first-out*)
 - Podemos **enfileirar** um objeto (*enqueue*), o adicionando no **fim** da fila
 - E podemos **desenfileirar** um objeto (*dequeue*), removendo o item no **início** da fila
- Uma pilha representa uma estrutura de dados **LIFO** (do inglês, *last-in, first-out*)
 - Podemos **empilhar** um objeto (*push*), o adicionando no **topo** da pilha
 - E podemos **desempilhar** um objeto (*pop*), removendo o item do **topo** da pilha

- Dentre as coleções clássica estudadas, podemos citar as **filas** e as **pilhas**
- Uma fila representa uma estrutura de dados **FIFO** (do inglês, *first-in, first-out*)
 - Podemos **enfileirar** um objeto (*enqueue*), o adicionando no **fim** da fila
 - E podemos **desenfileirar** um objeto (*dequeue*), removendo o item no **início** da fila
- Uma pilha representa uma estrutura de dados **LIFO** (do inglês, *last-in, first-out*)
 - Podemos **empilhar** um objeto (*push*), o adicionando no **topo** da pilha
 - E podemos **desempilhar** um objeto (*pop*), removendo o item do **topo** da pilha

Filas, pilhas e listas

- Dentre as coleções clássica estudadas, podemos citar as **filas** e as **pilhas**
- Uma fila representa uma estrutura de dados **FIFO** (do inglês, *first-in, first-out*)
 - Podemos **enfileirar** um objeto (*enqueue*), o adicionando no **fim** da fila
 - E podemos **desenfileirar** um objeto (*dequeue*), removendo o item no **início** da fila
- Uma pilha representa uma estrutura de dados **LIFO** (do inglês, *last-in, first-out*)
 - Podemos **empilhar** um objeto (*push*), o adicionando no **topo** da pilha
 - E podemos **desempilhar** um objeto (*pop*), removendo o item do **topo** da pilha

Filas, pilhas e listas

- Dentre as coleções clássica estudadas, podemos citar as **filas** e as **pilhas**
- Uma fila representa uma estrutura de dados **FIFO** (do inglês, *first-in, first-out*)
 - Podemos **enfileirar** um objeto (*enqueue*), o adicionando no **fim** da fila
 - E podemos **desenfileirar** um objeto (*dequeue*), removendo o item no **início** da fila
- Uma pilha representa uma estrutura de dados **LIFO** (do inglês, *last-in, first-out*)
 - Podemos **empilhar** um objeto (*push*), o adicionando no **topo** da pilha
 - E podemos **desempilhar** um objeto (*pop*), removendo o item do **topo** da pilha

Filas, pilhas e listas

- Dentre as coleções clássica estudadas, podemos citar as **filas** e as **pilhas**
- Uma fila representa uma estrutura de dados **FIFO** (do inglês, *first-in, first-out*)
 - Podemos **enfileirar** um objeto (*enqueue*), o adicionando no **fim** da fila
 - E podemos **desenfileirar** um objeto (*dequeue*), removendo o item no **início** da fila
- Uma pilha representa uma estrutura de dados **LIFO** (do inglês, *last-in, first-out*)
 - Podemos **empilhar** um objeto (*push*), o adicionando no **topo** da pilha
 - E podemos **desempilhar** um objeto (*pop*), removendo o item do **topo** da pilha

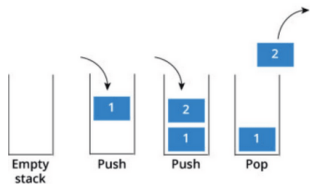
Filas, pilhas e listas

- Dentre as coleções clássica estudadas, podemos citar as **filas** e as **pilhas**
- Uma fila representa uma estrutura de dados **FIFO** (do inglês, *first-in, first-out*)
 - Podemos **enfileirar** um objeto (*enqueue*), o adicionando no **fim** da fila
 - E podemos **desenfileirar** um objeto (*dequeue*), removendo o item no **início** da fila
- Uma pilha representa uma estrutura de dados **LIFO** (do inglês, *last-in, first-out*)
 - Podemos **empilhar** um objeto (*push*), o adicionando no **topo** da pilha
 - E podemos **desempilhar** um objeto (*pop*), removendo o item do **topo** da pilha

Filas, pilhas e listas

- Dentre as coleções clássica estudadas, podemos citar as **filas** e as **pilhas**
- Uma fila representa uma estrutura de dados **FIFO** (do inglês, *first-in, first-out*)
 - Podemos **enfileirar** um objeto (*enqueue*), o adicionando no **fim** da fila
 - E podemos **desenfileirar** um objeto (*dequeue*), removendo o item no **início** da fila
- Uma pilha representa uma estrutura de dados **LIFO** (do inglês, *last-in, first-out*)
 - Podemos **empilhar** um objeto (*push*), o adicionando no **topo** da pilha
 - E podemos **desempilhar** um objeto (*pop*), removendo o item do **topo** da pilha

Data Structure Basics



Stack



Queue

Listas duplamente linkadas

- Uma das formas de se implementar filas e pilhas é através de listas linkadas
 - Porém, como as estruturas são **abstratas**, não é a única forma: por exemplo, vetores dinâmicos e listas circulares podem ser utilizados
- A ideia de uma lista linkada é representar os dados como uma **corrente** através de elos, pequenas estruturas contendo um valor que pertence à coleção
 - Elos salvam uma referência para o **próximo** elo da corrente
 - Além disso, a estrutura se lembra quem é seu primeiro elo, chamado de **cabeça**
 - Caso cada elo **também** se lembre do elo **anterior**, podemos nos mover em ambas as direções
 - Nesse caso, a lista também deverá se lembrar de sua **cauda**, e teremos uma lista **duplamente encadeada**

Listas duplamente linkadas

- Uma das formas de se implementar filas e pilhas é através de listas linkadas
 - Porém, como as estruturas são **abstratas**, não é a única forma: por exemplo, vetores dinâmicos e listas circulares podem ser utilizados
- A ideia de uma lista linkada é representar os dados como uma **corrente** através de elos, pequenas estruturas contendo um valor que pertence à coleção
 - Elos salvam uma referência para o **próximo** elo da corrente
 - Além disso, a estrutura se lembra quem é seu primeiro elo, chamado de **cabeça**
 - Caso cada elo **também** se lembre do elo **anterior**, podemos nos mover em ambas as direções
 - Nesse caso, a lista também deverá se lembrar de sua **cauda**, e teremos uma lista **duplamente encadeada**

Listas duplamente linkadas

- Uma das formas de se implementar filas e pilhas é através de listas linkadas
 - Porém, como as estruturas são **abstratas**, não é a única forma: por exemplo, vetores dinâmicos e listas circulares podem ser utilizados
- A ideia de uma lista linkada é representar os dados como uma **corrente** através de elos, pequenas estruturas contendo um valor que pertence à coleção
 - Elos salvam uma referência para o **próximo** elo da corrente
 - Além disso, a estrutura se lembra quem é seu primeiro elo, chamado de **cabeça**
 - Caso cada elo **também** se lembre do elo **anterior**, podemos nos mover em ambas as direções
 - Nesse caso, a lista também deverá se lembrar de sua **cauda**, e teremos uma lista **duplamente encadeada**

Listas duplamente linkadas

- Uma das formas de se implementar filas e pilhas é através de listas linkadas
 - Porém, como as estruturas são **abstratas**, não é a única forma: por exemplo, vetores dinâmicos e listas circulares podem ser utilizados
- A ideia de uma lista linkada é representar os dados como uma **corrente** através de elos, pequenas estruturas contendo um valor que pertence à coleção
 - Elos salvam uma referência para o **próximo** elo da corrente
 - Além disso, a estrutura se lembra quem é seu primeiro elo, chamado de **cabeça**
 - Caso cada elo **também** se lembre do elo **anterior**, podemos nos mover em ambas as direções
 - Nesse caso, a lista também deverá se lembrar de sua **cauda**, e teremos uma lista **duplamente encadeada**

Listas duplamente linkadas

- Uma das formas de se implementar filas e pilhas é através de listas linkadas
 - Porém, como as estruturas são **abstratas**, não é a única forma: por exemplo, vetores dinâmicos e listas circulares podem ser utilizados
- A ideia de uma lista linkada é representar os dados como uma **corrente** através de elos, pequenas estruturas contendo um valor que pertence à coleção
 - Elos salvam uma referência para o **próximo** elo da corrente
 - Além disso, a estrutura se lembra quem é seu primeiro elo, chamado de **cabeça**
 - Caso cada elo **também** se lembre do elo **anterior**, podemos nos mover em ambas as direções
 - Nesse caso, a lista também deverá se lembrar de sua **cauda**, e teremos uma lista **duplamente encadeada**

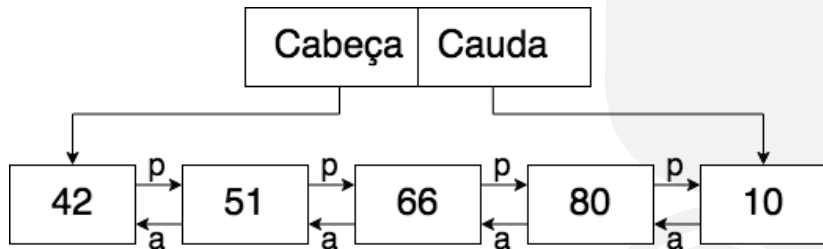
Listas duplamente linkadas

- Uma das formas de se implementar filas e pilhas é através de listas linkadas
 - Porém, como as estruturas são **abstratas**, não é a única forma: por exemplo, vetores dinâmicos e listas circulares podem ser utilizados
- A ideia de uma lista linkada é representar os dados como uma **corrente** através de elos, pequenas estruturas contendo um valor que pertence à coleção
 - Elos salvam uma referência para o **próximo** elo da corrente
 - Além disso, a estrutura se lembra quem é seu primeiro elo, chamado de **cabeça**
 - Caso cada elo **também** se lembre do elo **anterior**, podemos nos mover em ambas as direções
 - Nesse caso, a lista também deverá se lembrar de sua **cauda**, e teremos uma lista **duplamente encadeada**

Listas duplamente linkadas

- Uma das formas de se implementar filas e pilhas é através de listas linkadas
 - Porém, como as estruturas são **abstratas**, não é a única forma: por exemplo, vetores dinâmicos e listas circulares podem ser utilizados
- A ideia de uma lista linkada é representar os dados como uma **corrente** através de elos, pequenas estruturas contendo um valor que pertence à coleção
 - Elos salvam uma referência para o **próximo** elo da corrente
 - Além disso, a estrutura se lembra quem é seu primeiro elo, chamado de **cabeça**
 - Caso cada elo **também** se lembre do elo **anterior**, podemos nos mover em ambas as direções
 - Nesse caso, a lista também deverá se lembrar de sua **cauda**, e teremos uma lista **duplamente encadeada**

Listas duplamente linkadas



Vetores Dinâmicos

- Vetores são áreas contínuas de memória, cujo principal objetivo é o acesso randômico: itens podem ser verificados em qualquer posição de forma rápida
- Entretanto, o tamanho de vetores é fixo: não podemos adicionar, arbitrariamente, mais elementos
- Um vetor dinâmico é uma estrutura de dados abstrata definida a fim de se gerenciar um vetor internamente, podendo **redimensioná-lo** caso necessário
 - Para tal, um vetor dinâmico deve lembrar do endereço do vetor interno, do seu tamanho, e sua **capacidade**
- Além da vantagem de acesso randômico, é possível para um vetor dinâmico fornecer ao usuário do código o endereço para o **armazenamento interno**
 - Entretanto, **remover itens** de um vetor, exceto que na última posição, é **muito custoso**!

Vetores Dinâmicos

- Vetores são áreas contínuas de memória, cujo principal objetivo é o acesso randômico: itens podem ser verificados em qualquer posição de forma rápida
- Entretanto, o tamanho de vetores é fixo: não podemos adicionar, arbitrariamente, mais elementos
- Um vetor dinâmico é uma estrutura de dados abstrata definida a fim de se gerenciar um vetor internamente, podendo **redimensioná-lo** caso necessário
 - Para tal, um vetor dinâmico deve lembrar do endereço do vetor interno, do seu tamanho, e sua **capacidade**
- Além da vantagem de acesso randômico, é possível para um vetor dinâmico fornecer ao usuário do código o endereço para o **armazenamento interno**
 - Entretanto, **remover itens** de um vetor, exceto que na última posição, é **muito custoso**!

Vetores Dinâmicos

- Vetores são áreas contínuas de memória, cujo principal objetivo é o acesso randômico: itens podem ser verificados em qualquer posição de forma rápida
- Entretanto, o tamanho de vetores é fixo: não podemos adicionar, arbitrariamente, mais elementos
- Um vetor dinâmico é uma estrutura de dados abstrata definida a fim de se gerenciar um vetor internamente, podendo **redimensioná-lo** caso necessário
 - Para tal, um vetor dinâmico deve lembrar do endereço do vetor interno, do seu tamanho, e sua **capacidade**
- Além da vantagem de acesso randômico, é possível para um vetor dinâmico fornecer ao usuário do código o endereço para o **armazenamento interno**
 - Entretanto, **remover itens** de um vetor, exceto que na última posição, é **muito custoso**!

Vetores Dinâmicos

- Vetores são áreas contínuas de memória, cujo principal objetivo é o acesso randômico: itens podem ser verificados em qualquer posição de forma rápida
- Entretanto, o tamanho de vetores é fixo: não podemos adicionar, arbitrariamente, mais elementos
- Um vetor dinâmico é uma estrutura de dados abstrata definida a fim de se gerenciar um vetor internamente, podendo **redimensioná-lo** caso necessário
 - Para tal, um vetor dinâmico deve lembrar do endereço do vetor interno, do seu tamanho, e sua **capacidade**
- Além da vantagem de acesso randômico, é possível para um vetor dinâmico fornecer ao usuário do código o endereço para o **armazenamento interno**
 - Entretanto, **remover itens** de um vetor, exceto que na última posição, é **muito custoso**!

Vetores Dinâmicos

- Vetores são áreas contínuas de memória, cujo principal objetivo é o acesso randômico: itens podem ser verificados em qualquer posição de forma rápida
- Entretanto, o tamanho de vetores é fixo: não podemos adicionar, arbitrariamente, mais elementos
- Um vetor dinâmico é uma estrutura de dados abstrata definida a fim de se gerenciar um vetor internamente, podendo **redimensioná-lo** caso necessário
 - Para tal, um vetor dinâmico deve lembrar do endereço do vetor interno, do seu tamanho, e sua **capacidade**
- Além da vantagem de acesso randômico, é possível para um vetor dinâmico fornecer ao usuário do código o endereço para o **armazenamento interno**
 - Entretanto, **remover itens** de um vetor, exceto que na última posição, é **muito custoso**!

- Vetores são áreas contínuas de memória, cujo principal objetivo é o acesso randômico: itens podem ser verificados em qualquer posição de forma rápida
- Entretanto, o tamanho de vetores é fixo: não podemos adicionar, arbitrariamente, mais elementos
- Um vetor dinâmico é uma estrutura de dados abstrata definida a fim de se gerenciar um vetor internamente, podendo **redimensioná-lo** caso necessário
 - Para tal, um vetor dinâmico deve lembrar do endereço do vetor interno, do seu tamanho, e sua **capacidade**
- Além da vantagem de acesso randômico, é possível para um vetor dinâmico fornecer ao usuário do código o endereço para o **armazenamento interno**
 - Entretanto, **remover itens** de um vetor, exceto que na última posição, é **muito custoso**!

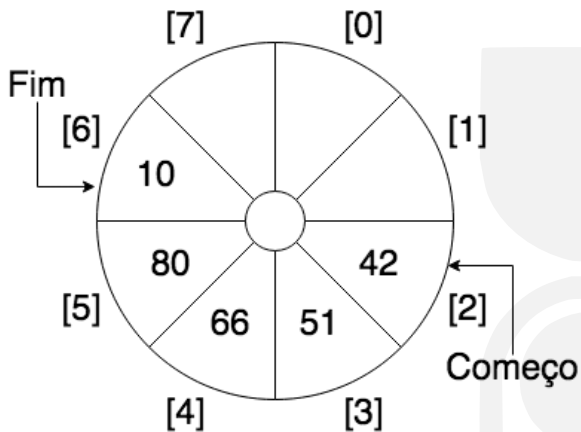
- Também chamados de listas circulares, possuem uma implementação similar à de vetores dinâmicos
- A ideia é que, além de salvar o seu tamanho (quantidade de itens), seja lembrada a **posição inicial**
- O vetor interno deve ser imaginado como um círculo: após passarmos da posição final, **voltamos à inicial**
- Graças a isso, podemos salvar itens no começo e no final do *buffer*, similar a uma lista duplamente linkada
- Caso a posição inicial e final sejam iguais, sabemos que o *buffer* está vazio

- Também chamados de listas circulares, possuem uma implementação similar à de vetores dinâmicos
- A ideia é que, além de salvar o seu tamanho (quantidade de itens), seja lembrada a **posição inicial**
- O vetor interno deve ser imaginado como um círculo: após passarmos da posição final, **voltamos à inicial**
- Graças a isso, podemos salvar itens no começo e no final do *buffer*, similar a uma lista duplamente linkada
- Caso a posição inicial e final sejam iguais, sabemos que o *buffer* está vazio

- Também chamados de listas circulares, possuem uma implementação similar à de vetores dinâmicos
- A ideia é que, além de salvar o seu tamanho (quantidade de itens), seja lembrada a **posição inicial**
- O vetor interno deve ser imaginado como um círculo: após passarmos da posição final, **voltamos à inicial**
- Graças a isso, podemos salvar itens no começo e no final do *buffer*, similar a uma lista duplamente linkada
- Caso a posição inicial e final sejam iguais, sabemos que o *buffer* está vazio

- Também chamados de listas circulares, possuem uma implementação similar à de vetores dinâmicos
- A ideia é que, além de salvar o seu tamanho (quantidade de itens), seja lembrada a **posição inicial**
- O vetor interno deve ser imaginado como um círculo: após passarmos da posição final, **voltamos à inicial**
- Graças a isso, podemos salvar itens no começo e no final do *buffer*, similar a uma lista duplamente linkada
- Caso a posição inicial e final sejam iguais, sabemos que o *buffer* está vazio

- Também chamados de listas circulares, possuem uma implementação similar à de vetores dinâmicos
- A ideia é que, além de salvar o seu tamanho (quantidade de itens), seja lembrada a **posição inicial**
- O vetor interno deve ser imaginado como um círculo: após passarmos da posição final, **voltamos à inicial**
- Graças a isso, podemos salvar itens no começo e no final do *buffer*, similar a uma lista duplamente linkada
- Caso a posição inicial e final sejam iguais, sabemos que o *buffer* está vazio



Algoritmos de ordenação

- Muitas vezes, durante a execução de um programa, temos a necessidade de verificar se um elemento está presente dentro de uma coleção (pilha, fila, vetor, etc)
- O algoritmo “óbvio”: percorra todos os elementos da coleção, e, um por um, verifique se ele é o elemento desejado
- Claramente, quanto mais elementos existirem na coleção, mais demorada será a busca ingênua no pior caso
- Caso a coleção esteja **ordenada**, temos uma alternativa: a **busca binária**; ao verificar o elemento no meio da coleção, sabemos que tudo à esquerda será menor, e à direita será maior
- Podemos repetir essa busca, sucessivamente diminuindo a quantidade de candidatos pela **metade**, até encontrarmos o elemento desejado, ou concluirmos que ele não está presente na coleção

- Muitas vezes, durante a execução de um programa, temos a necessidade de verificar se um elemento está presente dentro de uma coleção (pilha, fila, vetor, etc)
- O algoritmo “óbvio”: percorra todos os elementos da coleção, e, um por um, verifique se ele é o elemento desejado
- Claramente, quanto mais elementos existirem na coleção, mais demorada será a busca ingênua no pior caso
- Caso a coleção esteja **ordenada**, temos uma alternativa: a **busca binária**; ao verificar o elemento no meio da coleção, sabemos que tudo à esquerda será menor, e à direita será maior
- Podemos repetir essa busca, sucessivamente diminuindo a quantidade de candidatos pela **metade**, até encontrarmos o elemento desejado, ou concluirmos que ele não está presente na coleção

Algoritmos de ordenação

- Muitas vezes, durante a execução de um programa, temos a necessidade de verificar se um elemento está presente dentro de uma coleção (pilha, fila, vetor, etc)
- O algoritmo “óbvio”: percorra todos os elementos da coleção, e, um por um, verifique se ele é o elemento desejado
- Claramente, quanto mais elementos existirem na coleção, mais demorada será a busca ingênua no pior caso
- Caso a coleção esteja **ordenada**, temos uma alternativa: a **busca binária**; ao verificar o elemento no meio da coleção, sabemos que tudo à esquerda será menor, e à direita será maior
- Podemos repetir essa busca, sucessivamente diminuindo a quantidade de candidatos pela **metade**, até encontrarmos o elemento desejado, ou concluirmos que ele não está presente na coleção

Algoritmos de ordenação

- Muitas vezes, durante a execução de um programa, temos a necessidade de verificar se um elemento está presente dentro de uma coleção (pilha, fila, vetor, etc)
- O algoritmo “óbvio”: percorra todos os elementos da coleção, e, um por um, verifique se ele é o elemento desejado
- Claramente, quanto mais elementos existirem na coleção, mais demorada será a busca ingênua no pior caso
- Caso a coleção esteja **ordenada**, temos uma alternativa: a **busca binária**; ao verificar o elemento no meio da coleção, sabemos que tudo à esquerda será menor, e à direita será maior
- Podemos repetir essa busca, sucessivamente diminuindo a quantidade de candidatos pela **metade**, até encontrarmos o elemento desejado, ou concluirmos que ele não está presente na coleção

- Muitas vezes, durante a execução de um programa, temos a necessidade de verificar se um elemento está presente dentro de uma coleção (pilha, fila, vetor, etc)
- O algoritmo “óbvio”: percorra todos os elementos da coleção, e, um por um, verifique se ele é o elemento desejado
- Claramente, quanto mais elementos existirem na coleção, mais demorada será a busca ingênua no pior caso
- Caso a coleção esteja **ordenada**, temos uma alternativa: a **busca binária**; ao verificar o elemento no meio da coleção, sabemos que tudo à esquerda será menor, e à direita será maior
- Podemos repetir essa busca, sucessivamente diminuindo a quantidade de candidatos pela **metade**, até encontrarmos o elemento desejado, ou concluirmos que ele não está presente na coleção

Algoritmos de ordenação

- Diversos **algoritmos de ordenação** existem, com propriedades como tempo de execução variadas
- Para simplificar a apresentação, vamos assumir que a coleção que queremos ordenar possui **acesso randômico** (e.g., vetor)
- Exemplo trivial: **bubble sort**

```
ENTRADA  vetor v, tamanho n

ENQUANTO n > 2:
    PARA i DE 0 ATÉ n:
        SE v[i] > v[i + 1]:
            TROQUE v[i] E v[i + 1]
    DIMINUA n
```

- “Borbulhamos” os itens, movendo os maiores para as posições mais à direita, repetindo conforme necessário

Algoritmos de ordenação

- Diversos **algoritmos de ordenação** existem, com propriedades como tempo de execução variadas
- Para simplificar a apresentação, vamos assumir que a coleção que queremos ordenar possui **acesso randômico** (e.g., vetor)
- Exemplo trivial: **bubble sort**

```
ENTRADA  vetor v, tamanho n

ENQUANTO n > 2:
    PARA i DE 0 ATÉ n:
        SE v[i] > v[i + 1]:
            TROQUE v[i] E v[i + 1]
    DIMINUA n
```

- “Borbulhamos” os itens, movendo os maiores para as posições mais à direita, repetindo conforme necessário

Algoritmos de ordenação

- Diversos **algoritmos de ordenação** existem, com propriedades como tempo de execução variadas
- Para simplificar a apresentação, vamos assumir que a coleção que queremos ordenar possui **acesso randômico** (e.g., vetor)
- Exemplo trivial: **bubble sort**

```
ENTRADA  vetor v, tamanho n

ENQUANTO n > 2:
    PARA i DE 0 ATÉ n:
        SE v[i] > v[i + 1]:
            TROQUE v[i] E v[i + 1]
    DIMINUA n
```

- “Borbulhamos” os itens, movendo os maiores para as posições mais à direita, repetindo conforme necessário

- Diversos **algoritmos de ordenação** existem, com propriedades como tempo de execução variadas
- Para simplificar a apresentação, vamos assumir que a coleção que queremos ordenar possui **acesso randômico** (e.g., vetor)
- Exemplo trivial: **bubble sort**

```
ENTRADA  vetor v, tamanho n

ENQUANTO n > 2:
    PARA i DE 0 ATÉ n:
        SE v[i] > v[i + 1]:
            TROQUE v[i] E v[i + 1]
    DIMINUA n
```

- “Borbulhamos” os itens, movendo os maiores para as posições mais à direita, repetindo conforme necessário

- Algoritmo: **selection sort**

```
ENTRADA vetor v, tamanho n

PARA i DE 0 ATÉ n:
    min ← i
    PARA j DE i + 1 ATÉ n:
        SE v[j] < v[min]:
            min ← j
    SE min ≠ i:
        TROQUE v[i] E v[min]
```

- Algoritmo: **insertion sort**

```
ENTRADA vetor v, tamanho n

PARA i DE 1 ATÉ n:
    j ← i
    ENQUANTO j > 0 E v[j - 1] > v[j]:
        TROQUE v[j] E v[j - 1]
        DIMINUA j
```

- Algoritmo: **selection sort**

```
ENTRADA vetor v, tamanho n

PARA i DE 0 ATÉ n:
    min ← i
    PARA j DE i + 1 ATÉ n:
        SE v[j] < v[min]:
            min ← j
    SE min ≠ i:
        TROQUE v[i] E v[min]
```

- Algoritmo: **insertion sort**

```
ENTRADA vetor v, tamanho n

PARA i DE 1 ATÉ n:
    j ← i
    ENQUANTO j > 0 E v[j - 1] > v[j]:
        TROQUE v[j] E v[j - 1]
    DIMINUA j
```

Complexidade de algoritmos

- É possível quantificar o **custo** de um algoritmo em relação ao tamanho de sua **entrada**, o que chamamos de **complexidade**
- No caso de algoritmos de ordenação, o tamanho da entrada, n , é exatamente a quantidade de itens em uma coleção
- Além de ser uma valiosa informação do ponto de vista teórico, a complexidade de um algoritmo pode nos ajudar a escolher a melhor alternativa para solucionar um problema
- Podemos avaliar a complexidade de **tempo**, isto é, quanto tempo a execução do algoritmo leva, e a complexidade de **espaço**, quanta memória ele usa
- Medimos a complexidade em proporção à entrada, e não por valores absolutos, pois tais iriam depender do compilador, sistema operacional, processador, cache, etc

Complexidade de algoritmos

- É possível quantificar o **custo** de um algoritmo em relação ao tamanho de sua **entrada**, o que chamamos de **complexidade**
- No caso de algoritmos de ordenação, o tamanho da entrada, n , é exatamente a quantidade de itens em uma coleção
- Além de ser uma valiosa informação do ponto de vista teórico, a complexidade de um algoritmo pode nos ajudar a escolher a melhor alternativa para solucionar um problema
- Podemos avaliar a complexidade de **tempo**, isto é, quanto tempo a execução do algoritmo leva, e a complexidade de **espaço**, quanta memória ele usa
- Medimos a complexidade em proporção à entrada, e não por valores absolutos, pois tais iriam depender do compilador, sistema operacional, processador, cache, etc

Complexidade de algoritmos

- É possível quantificar o **custo** de um algoritmo em relação ao tamanho de sua **entrada**, o que chamamos de **complexidade**
- No caso de algoritmos de ordenação, o tamanho da entrada, n , é exatamente a quantidade de itens em uma coleção
- Além de ser uma valiosa informação do ponto de vista teórico, a complexidade de um algoritmo pode nos ajudar a escolher a melhor alternativa para solucionar um problema
- Podemos avaliar a complexidade de **tempo**, isto é, quanto tempo a execução do algoritmo leva, e a complexidade de **espaço**, quanta memória ele usa
- Medimos a complexidade em proporção à entrada, e não por valores absolutos, pois tais iriam depender do compilador, sistema operacional, processador, cache, etc

Complexidade de algoritmos

- É possível quantificar o **custo** de um algoritmo em relação ao tamanho de sua **entrada**, o que chamamos de **complexidade**
- No caso de algoritmos de ordenação, o tamanho da entrada, n , é exatamente a quantidade de itens em uma coleção
- Além de ser uma valiosa informação do ponto de vista teórico, a complexidade de um algoritmo pode nos ajudar a escolher a melhor alternativa para solucionar um problema
- Podemos avaliar a complexidade de **tempo**, isto é, quanto tempo a execução do algoritmo leva, e a complexidade de **espaço**, quanta memória ele usa
- Medimos a complexidade em proporção à entrada, e não por valores absolutos, pois tais iriam depender do compilador, sistema operacional, processador, cache, etc

Complexidade de algoritmos

- É possível quantificar o **custo** de um algoritmo em relação ao tamanho de sua **entrada**, o que chamamos de **complexidade**
- No caso de algoritmos de ordenação, o tamanho da entrada, n , é exatamente a quantidade de itens em uma coleção
- Além de ser uma valiosa informação do ponto de vista teórico, a complexidade de um algoritmo pode nos ajudar a escolher a melhor alternativa para solucionar um problema
- Podemos avaliar a complexidade de **tempo**, isto é, quanto tempo a execução do algoritmo leva, e a complexidade de **espaço**, quanta memória ele usa
- Medimos a complexidade em proporção à entrada, e não por valores absolutos, pois tais iriam depender do compilador, sistema operacional, processador, cache, etc

Complexidade de algoritmos

- Para definirmos um custo, imaginamos a execução do algoritmo em um **computador idealizado**
- Normalmente, utilizamos a notação **big O**, que representa o **pior caso** de execução de um algoritmo
 - $O(1)$: complexidade **constante**, isto é, não depende do tamanho da entrada
 - $O(\log n)$: complexidade **logarítmica**, proporcional a um logaritmo do tamanho da entrada
 - $O(n)$: complexidade **linear**, diretamente proporcional ao tamanho da entrada
 - $O(n^2)$: complexidade **quadrática**, proporcional ao quadrado do tamanho da entrada
- Utilizamos também o **big Ω** para representar o **melhor caso**

Complexidade de algoritmos

- Para definirmos um custo, imaginamos a execução do algoritmo em um **computador idealizado**
- Normalmente, utilizamos a notação **big O**, que representa o **pior caso** de execução de um algoritmo
 - $O(1)$: complexidade **constante**, isto é, não depende do tamanho da entrada
 - $O(\log n)$: complexidade **logarítmica**, proporcional a um logaritmo do tamanho da entrada
 - $O(n)$: complexidade **linear**, diretamente proporcional ao tamanho da entrada
 - $O(n^2)$: complexidade **quadrática**, proporcional ao quadrado do tamanho da entrada
- Utilizamos também o **big Ω** para representar o **melhor caso**

Complexidade de algoritmos

- Para definirmos um custo, imaginamos a execução do algoritmo em um **computador idealizado**
- Normalmente, utilizamos a notação **big O**, que representa o **pior caso** de execução de um algoritmo
 - $O(1)$: complexidade **constante**, isto é, não depende do tamanho da entrada
 - $O(\log n)$: complexidade **logarítmica**, proporcional a um logaritmo do tamanho da entrada
 - $O(n)$: complexidade **linear**, diretamente proporcional ao tamanho da entrada
 - $O(n^2)$: complexidade **quadrática**, proporcional ao quadrado do tamanho da entrada
- Utilizamos também o **big Ω** para representar o **melhor caso**

Complexidade de algoritmos

- Para definirmos um custo, imaginamos a execução do algoritmo em um **computador idealizado**
- Normalmente, utilizamos a notação **big O**, que representa o **pior caso** de execução de um algoritmo
 - $O(1)$: complexidade **constante**, isto é, não depende do tamanho da entrada
 - $O(\log n)$: complexidade **logarítmica**, proporcional a um logaritmo do tamanho da entrada
 - $O(n)$: complexidade **linear**, diretamente proporcional ao tamanho da entrada
 - $O(n^2)$: complexidade **quadrática**, proporcional ao quadrado do tamanho da entrada
- Utilizamos também o **big Ω** para representar o **melhor caso**

Complexidade de algoritmos

- Para definirmos um custo, imaginamos a execução do algoritmo em um **computador idealizado**
- Normalmente, utilizamos a notação **big O**, que representa o **pior caso** de execução de um algoritmo
 - $O(1)$: complexidade **constante**, isto é, não depende do tamanho da entrada
 - $O(\log n)$: complexidade **logarítmica**, proporcional a um logaritmo do tamanho da entrada
 - $O(n)$: complexidade **linear**, diretamente proporcional ao tamanho da entrada
 - $O(n^2)$: complexidade **quadrática**, proporcional ao quadrado do tamanho da entrada
- Utilizamos também o **big Ω** para representar o **melhor caso**

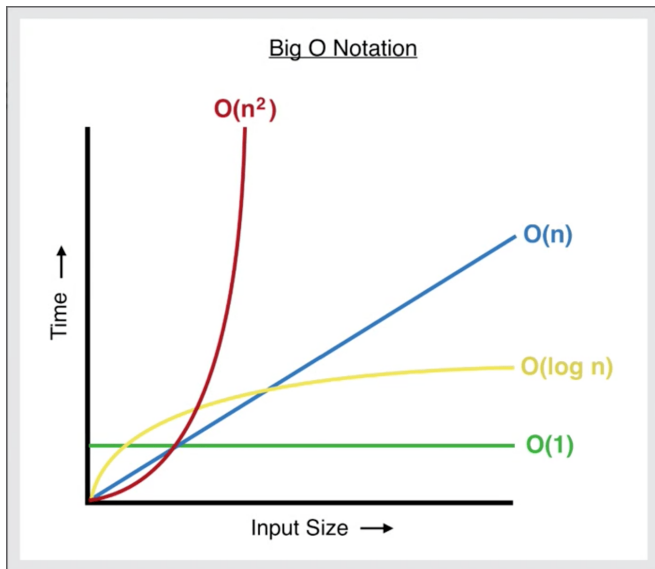
Complexidade de algoritmos

- Para definirmos um custo, imaginamos a execução do algoritmo em um **computador idealizado**
- Normalmente, utilizamos a notação **big O**, que representa o **pior caso** de execução de um algoritmo
 - $O(1)$: complexidade **constante**, isto é, não depende do tamanho da entrada
 - $O(\log n)$: complexidade **logarítmica**, proporcional a um logaritmo do tamanho da entrada
 - $O(n)$: complexidade **linear**, diretamente proporcional ao tamanho da entrada
 - $O(n^2)$: complexidade **quadrática**, proporcional ao quadrado do tamanho da entrada
- Utilizamos também o **big Ω** para representar o **melhor caso**

Complexidade de algoritmos

- Para definirmos um custo, imaginamos a execução do algoritmo em um **computador idealizado**
- Normalmente, utilizamos a notação **big O**, que representa o **pior caso** de execução de um algoritmo
 - $O(1)$: complexidade **constante**, isto é, não depende do tamanho da entrada
 - $O(\log n)$: complexidade **logarítmica**, proporcional a um logaritmo do tamanho da entrada
 - $O(n)$: complexidade **linear**, diretamente proporcional ao tamanho da entrada
 - $O(n^2)$: complexidade **quadrática**, proporcional ao quadrado do tamanho da entrada
- Utilizamos também o **big Ω** para representar o **melhor caso**

Complexidade de algoritmos



Monte binário

- Uma estrutura com propriedades úteis é o chamado **monte binário** (do inglês, *binary heap*), que nos permite rapidamente encontrar o maior (ou menor) elemento
- Visualizamos um *buffer* como uma *árvore*, com duas propriedades:
 - **Propriedade de formato**: a estrutura forma uma árvore completa, onde todos os galhos (exceto o último nível) tem a mesma altura, e os maiores galhos sempre estão à esquerda
 - **Propriedade de *heap***: nós na árvore sempre possuem um valor maior (ou menor) que seus filhos e descendentes
- Contamos a posição do vetor na nossa “árvore” de cima pra baixo, da esquerda para a direita
- Dado um *buffer* com a propriedade de *heap*, é possível ordená-lo utilizando o algoritmo **heapsort**

- Uma estrutura com propriedades úteis é o chamado **monte binário** (do inglês, *binary heap*), que nos permite rapidamente encontrar o maior (ou menor) elemento
- Visualizamos um *buffer* como uma **árvore**, com duas propriedades:
 - **Propriedade de formato**: a estrutura forma uma árvore completa, onde todos os galhos (exceto o último nível) tem a mesma altura, e os maiores galhos sempre estão à esquerda
 - **Propriedade de *heap***: nós na árvore sempre possuem um valor maior (ou menor) que seus filhos e descendentes
- Contamos a posição do vetor na nossa “árvore” de cima pra baixo, da esquerda para a direita
- Dado um *buffer* com a propriedade de *heap*, é possível ordená-lo utilizando o algoritmo **heapsort**

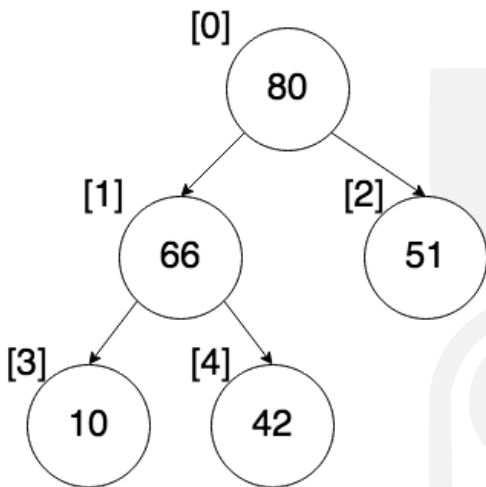
Monte binário

- Uma estrutura com propriedades úteis é o chamado **monte binário** (do inglês, *binary heap*), que nos permite rapidamente encontrar o maior (ou menor) elemento
- Visualizamos um *buffer* como uma **árvore**, com duas propriedades:
 - **Propriedade de formato**: a estrutura forma uma árvore completa, onde todos os galhos (exceto o último nível) tem a mesma altura, e os maiores galhos sempre estão à esquerda
 - **Propriedade de *heap***: nós na árvore sempre possuem um valor maior (ou menor) que seus filhos e descendentes
- Contamos a posição do vetor na nossa “árvore” de cima pra baixo, da esquerda para a direita
- Dado um *buffer* com a propriedade de *heap*, é possível ordená-lo utilizando o algoritmo **heapsort**

- Uma estrutura com propriedades úteis é o chamado **monte binário** (do inglês, *binary heap*), que nos permite rapidamente encontrar o maior (ou menor) elemento
- Visualizamos um *buffer* como uma **árvore**, com duas propriedades:
 - **Propriedade de formato**: a estrutura forma uma árvore completa, onde todos os galhos (exceto o último nível) tem a mesma altura, e os maiores galhos sempre estão à esquerda
 - **Propriedade de *heap***: nós na árvore sempre possuem um valor maior (ou menor) que seus filhos e descendentes
- Contamos a posição do vetor na nossa “árvore” de cima pra baixo, da esquerda para a direita
- Dado um *buffer* com a propriedade de *heap*, é possível ordená-lo utilizando o algoritmo **heapsort**

- Uma estrutura com propriedades úteis é o chamado **monte binário** (do inglês, *binary heap*), que nos permite rapidamente encontrar o maior (ou menor) elemento
- Visualizamos um *buffer* como uma **árvore**, com duas propriedades:
 - **Propriedade de formato**: a estrutura forma uma árvore completa, onde todos os galhos (exceto o último nível) tem a mesma altura, e os maiores galhos sempre estão à esquerda
 - **Propriedade de *heap***: nós na árvore sempre possuem um valor maior (ou menor) que seus filhos e descendentes
- Contamos a posição do vetor na nossa “árvore” **de cima pra baixo, da esquerda para a direita**
- Dado um *buffer* com a propriedade de *heap*, é possível ordená-lo utilizando o algoritmo **heapsort**

- Uma estrutura com propriedades úteis é o chamado **monte binário** (do inglês, *binary heap*), que nos permite rapidamente encontrar o maior (ou menor) elemento
- Visualizamos um *buffer* como uma **árvore**, com duas propriedades:
 - **Propriedade de formato**: a estrutura forma uma árvore completa, onde todos os galhos (exceto o último nível) tem a mesma altura, e os maiores galhos sempre estão à esquerda
 - **Propriedade de *heap***: nós na árvore sempre possuem um valor maior (ou menor) que seus filhos e descendentes
- Contamos a posição do vetor na nossa “árvore” **de cima pra baixo, da esquerda para a direita**
- Dado um *buffer* com a propriedade de *heap*, é possível ordená-lo utilizando o algoritmo **heapsort**



Monte binário

- Definimos uma função `heapify` que amontoa uma árvore...

```
ENTRADA  vetor v, tamanho n, índice i

m ← i
SE item à esquerda de i for maior que item em m:
    m ← esquerda de i
SE item à direita de i for maior que item em m:
    m ← direita de i
SE m ≠ i:
    TROQUE v[i] E v[m]
    heapify(v, n, m)
```

- E, dado um vetor, é possível transformá-lo em um *heap* em tempo linear, através da função `make-heap`...

```
ENTRADA  vetor v, tamanho n

PARA i DE n / 2 ATÉ 0:
    heapify(v, n, i)
```

Monte binário

- Definimos uma função `heapify` que amontoa uma árvore...

```
ENTRADA  vetor v, tamanho n, índice i

m ← i
SE item à esquerda de i for maior que item em m:
    m ← esquerda de i
SE item à direita de i for maior que item em m:
    m ← direita de i
SE m ≠ i:
    TROQUE v[i] E v[m]
    heapify(v, n, m)
```

- E, dado um vetor, é possível transformá-lo em um *heap* em tempo linear, através da função `make-heap`...

```
ENTRADA  vetor v, tamanho n

PARA i DE n / 2 ATÉ 0:
    heapify(v, n, i)
```

- Ao visualizarmos um *buffer* de memória como um monte, podemos facilmente encontrar índices para os nós relacionados na árvore (por exemplo, necessário para *heapify*)
 - Índice do nó pai do índice i : $\lfloor (i - 1) / 2 \rfloor$
 - Filho à esquerda de i : $i * 2 + 1$
 - Filho à direita de i : $i * 2 + 2$
- Note que a função *heapify* assume que as sub-árvores (à esquerda e direita) **já estejam** com propriedade de *heap*
- Assumimos também nos exemplos anteriores que estamos usando um **monte máximo**, e, portanto, sabemos que o **maior elemento** está sempre no **índice zero** (a raiz)
- Assim, podemos definir o algoritmo **heapsort** em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raiz para o fim do *buffer*

- Ao visualizarmos um *buffer* de memória como um monte, podemos facilmente encontrar índices para os nós relacionados na árvore (por exemplo, necessário para *heapify*)
 - Índice do nó pai do índice i : $\lfloor (i - 1) / 2 \rfloor$
 - Filho à esquerda de i : $i * 2 + 1$
 - Filho à direita de i : $i * 2 + 2$
- Note que a função *heapify* assume que as sub-árvores (à esquerda e direita) **já estejam** com propriedade de *heap*
- Assumimos também nos exemplos anteriores que estamos usando um **monte máximo**, e, portanto, sabemos que o **maior elemento** está sempre no **índice zero** (a raiz)
- Assim, podemos definir o algoritmo *heapsort* em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raiz para o fim do *buffer*

- Ao visualizarmos um *buffer* de memória como um monte, podemos facilmente encontrar índices para os nós relacionados na árvore (por exemplo, necessário para *heapify*)
 - Índice do nó pai do índice i : $\lfloor (i - 1) / 2 \rfloor$
 - Filho à esquerda de i : $i * 2 + 1$
 - Filho à direita de i : $i * 2 + 2$
- Note que a função *heapify* assume que as sub-árvores (à esquerda e direita) **já estejam** com propriedade de *heap*
- Assumimos também nos exemplos anteriores que estamos usando um **monte máximo**, e, portanto, sabemos que o **maior elemento** está sempre no **índice zero** (a raiz)
- Assim, podemos definir o algoritmo **heapsort** em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raiz para o fim do *buffer*

- Ao visualizarmos um *buffer* de memória como um monte, podemos facilmente encontrar índices para os nós relacionados na árvore (por exemplo, necessário para *heapify*)
 - Índice do nó pai do índice i : $\lfloor (i - 1) / 2 \rfloor$
 - Filho à esquerda de i : $i * 2 + 1$
 - Filho à direita de i : $i * 2 + 2$
- Note que a função *heapify* assume que as sub-árvores (à esquerda e direita) **já estejam** com propriedade de *heap*
- Assumimos também nos exemplos anteriores que estamos usando um **monte máximo**, e, portanto, sabemos que o **maior elemento** está sempre no **índice zero** (a raiz)
- Assim, podemos definir o algoritmo **heapsort** em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raiz para o fim do *buffer*

- Ao visualizarmos um *buffer* de memória como um monte, podemos facilmente encontrar índices para os nós relacionados na árvore (por exemplo, necessário para *heapify*)
 - Índice do nó pai do índice i : $\lfloor (i - 1) / 2 \rfloor$
 - Filho à esquerda de i : $i * 2 + 1$
 - Filho à direita de i : $i * 2 + 2$
- Note que a função *heapify* assume que as sub-árvores (à esquerda e direita) **já estejam** com propriedade de *heap*
- Assumimos também nos exemplos anteriores que estamos usando um **monte máximo**, e, portanto, sabemos que o **maior elemento** está sempre no **índice zero** (a raiz)
- Assim, podemos definir o algoritmo *heapsort* em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raiz para o fim do *buffer*

- Ao visualizarmos um *buffer* de memória como um monte, podemos facilmente encontrar índices para os nós relacionados na árvore (por exemplo, necessário para *heapify*)
 - Índice do nó pai do índice i : $\lfloor (i - 1) / 2 \rfloor$
 - Filho à esquerda de i : $i * 2 + 1$
 - Filho à direita de i : $i * 2 + 2$
- Note que a função *heapify* assume que as sub-árvores (à esquerda e direita) **já estejam** com propriedade de *heap*
- Assumimos também nos exemplos anteriores que estamos usando um **monte máximo**, e, portanto, sabemos que o **maior elemento** está sempre no **índice zero** (a raiz)
- Assim, podemos definir o algoritmo *heapsort* em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raiz para o fim do *buffer*

- Ao visualizarmos um *buffer* de memória como um monte, podemos facilmente encontrar índices para os nós relacionados na árvore (por exemplo, necessário para *heapify*)
 - Índice do nó pai do índice i : $\lfloor (i - 1) / 2 \rfloor$
 - Filho à esquerda de i : $i * 2 + 1$
 - Filho à direita de i : $i * 2 + 2$
- Note que a função *heapify* assume que as sub-árvores (à esquerda e direita) **já estejam** com propriedade de *heap*
- Assumimos também nos exemplos anteriores que estamos usando um **monte máximo**, e, portanto, sabemos que o **maior elemento** está sempre no **índice zero** (a raiz)
- Assim, podemos definir o algoritmo **heapsort** em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raiz para o fim do *buffer*

Monte binário

- Ao remover o maior item, movemos o item mais abaixo e à direita (o último item do *buffer*!) para a primeira posição, e corrigimos a propriedade de *heap* com a função *heapify*
- A complexidade do algoritmo, então, é $O(n \log n)$, estando entre a complexidade linear e a complexidade quadrática

```
ENTRADA  vetor v, tamanho n
```

```
make-heap(v, n)
```

```
ENQUANTO n > 0:
```

```
    max ← v[0]
```

```
    remove-max(v, n)
```

```
    DIMINUA n
```

```
    v[n] ← max
```

Monte binário

- Ao remover o maior item, movemos o item mais abaixo e à direita (o último item do *buffer*!) para a primeira posição, e corrigimos a propriedade de *heap* com a função *heapify*
- A complexidade do algoritmo, então, é $O(n \log n)$, estando entre a complexidade linear e a complexidade quadrática

```
ENTRADA  vetor v, tamanho n
```

```
make-heap(v, n)
```

```
ENQUANTO n > 0:
```

```
    max ← v[0]
```

```
    remove-max(v, n)
```

```
    DIMINUA n
```

```
    v[n] ← max
```

Complexidade de algoritmos, continuando...

Operação	Estrutura	Pilha	Fila	Vetor	Circular
Inserir no começo		$O(1)$			$O(1)$
Inserir no final			$O(1)$	$O(1)$	$O(1)$
Inserir em um índice		$O(n)$	$O(n)$	$O(n)$	$O(n)$
Remover no começo		$O(1)$	$O(1)$		$O(1)$
Remover no final				$O(1)$	$O(1)$
Remover em um índice		$O(n)$	$O(n)$	$O(n)$	$O(n)$
Acesso por índice		$O(n)$	$O(n)$	$O(1)$	$O(1)$
Procurar um elemento		$O(n)$	$O(n)$	$O(n)$	$O(n)$

Complexidade de algoritmos, continuando...

Algoritmo	Melhor caso	Pior caso
Bubblesort	$\Omega(n^2)$	$O(n^2)$
Selection sort	$\Omega(n^2)$	$O(n^2)$
Insertion sort	$\Omega(n)$	$O(n^2)$
Heapsort	$\Omega(n \log n)$	$O(n \log n)$