Estrutura de Dados

Paulo Torrens

paulotorrens@gnu.org

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

2020/1

Coleções Dinâmicas

- 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**

Coleções Dinâmicas

- 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**

Coleções Dinâmicas

- 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**

Coleções Dinâmicas

- 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**

Coleções Dinâmicas

- 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**

Coleções Dinâmicas

- 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 citas 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 **tipo** da pilha

Filas, pilhas e listas

- Dentre as coleções clássica estudadas, podemos citas 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 **tipo** da pilha

Filas, pilhas e listas

- Dentre as coleções clássica estudadas, podemos citas 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 **tipo** da pilha

Filas, pilhas e listas

- Dentre as coleções clássica estudadas, podemos citas 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 **tipo** da pilha

Filas, pilhas e listas

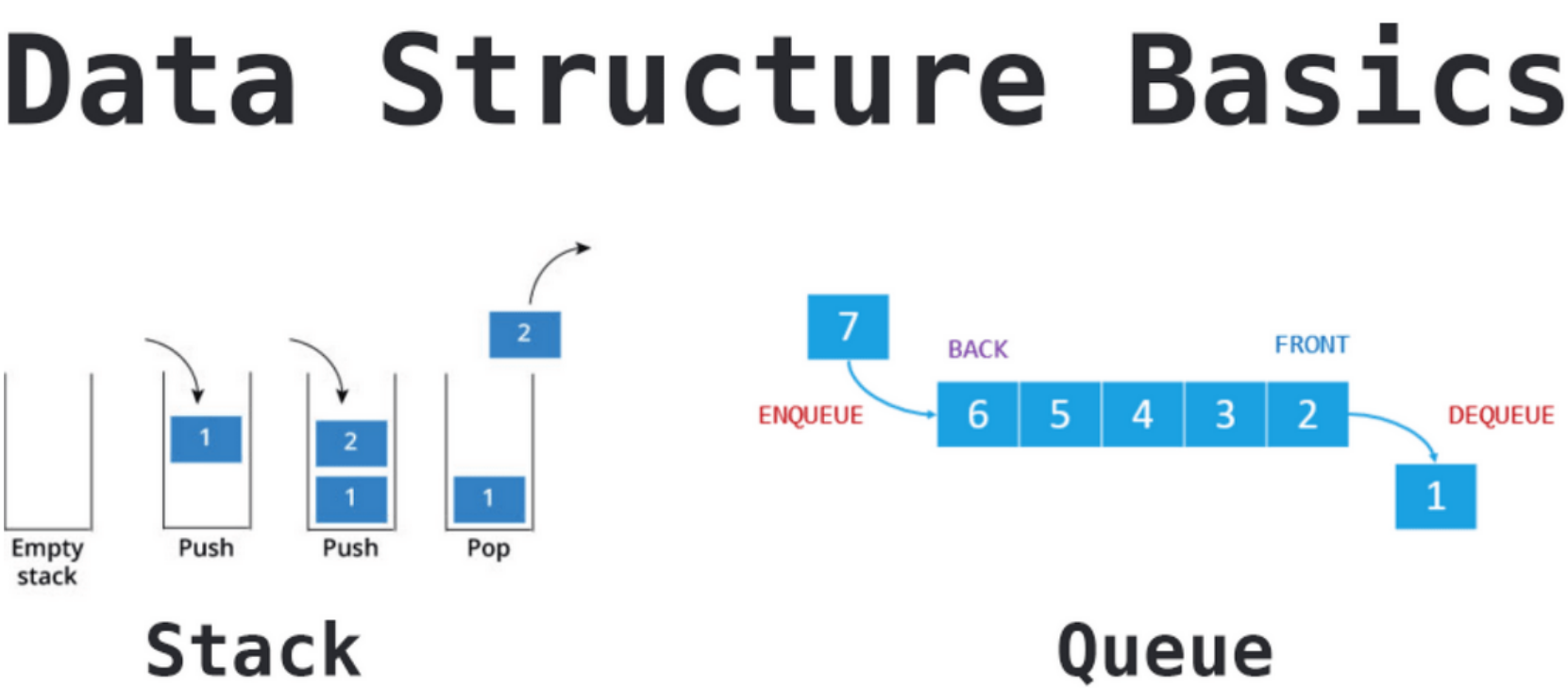
- Dentre as coleções clássica estudadas, podemos citas as **filas** e as **pilhas**
- Uma fila representa uma estrutura de dados **FIFO** (do inglês, *first-in, first-out*)
  - Podemos **enfileirar** um objeto (*enqueue*), o adicionando no **fim** da fila
  - E podemos **desenfileirar** um objeto (*dequeue*), 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 **tipo** da pilha

Filas, pilhas e listas

- Dentre as coleções clássica estudadas, podemos citas 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 **tipo** da pilha

Filas, pilhas e listas

- Dentre as coleções clássica estudadas, podemos citas 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 **tipo** da pilha


Data Structure Basics
Empty stack
Push
Push
Pop
2
1
2
1
1
Stack
7
BACK
FRONT
ENQUEUE
6
5
4
3
2
DEQUEUE
1
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**

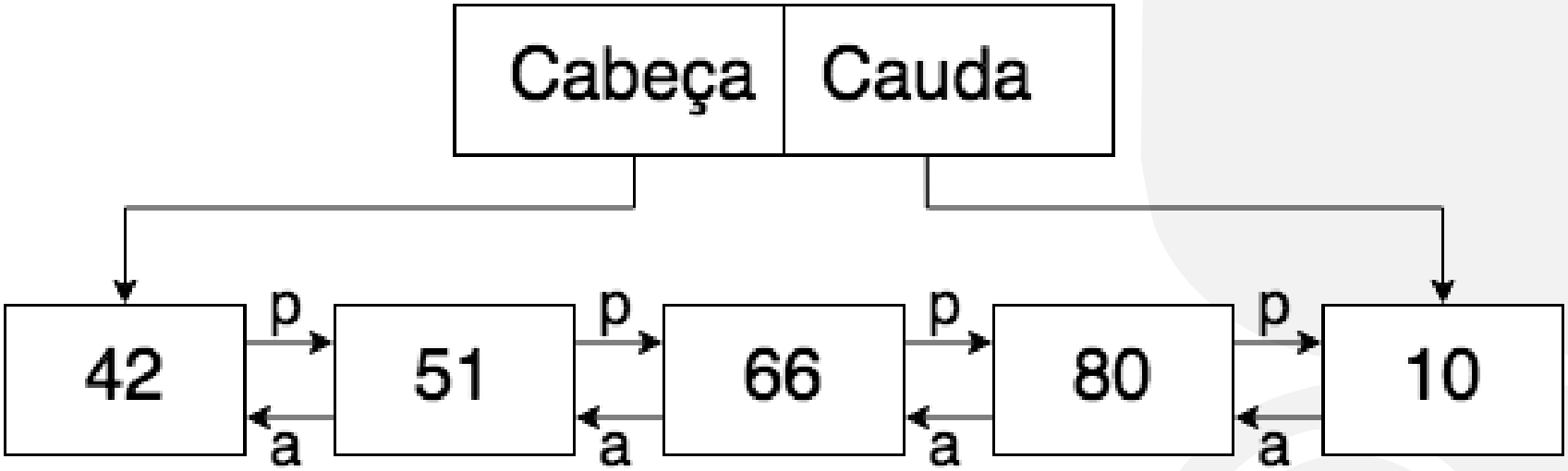
Listas duplamente linkadas

- 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 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**!

Buffers circulares

- 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

Buffers circulares

- 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

Buffers circulares

- 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

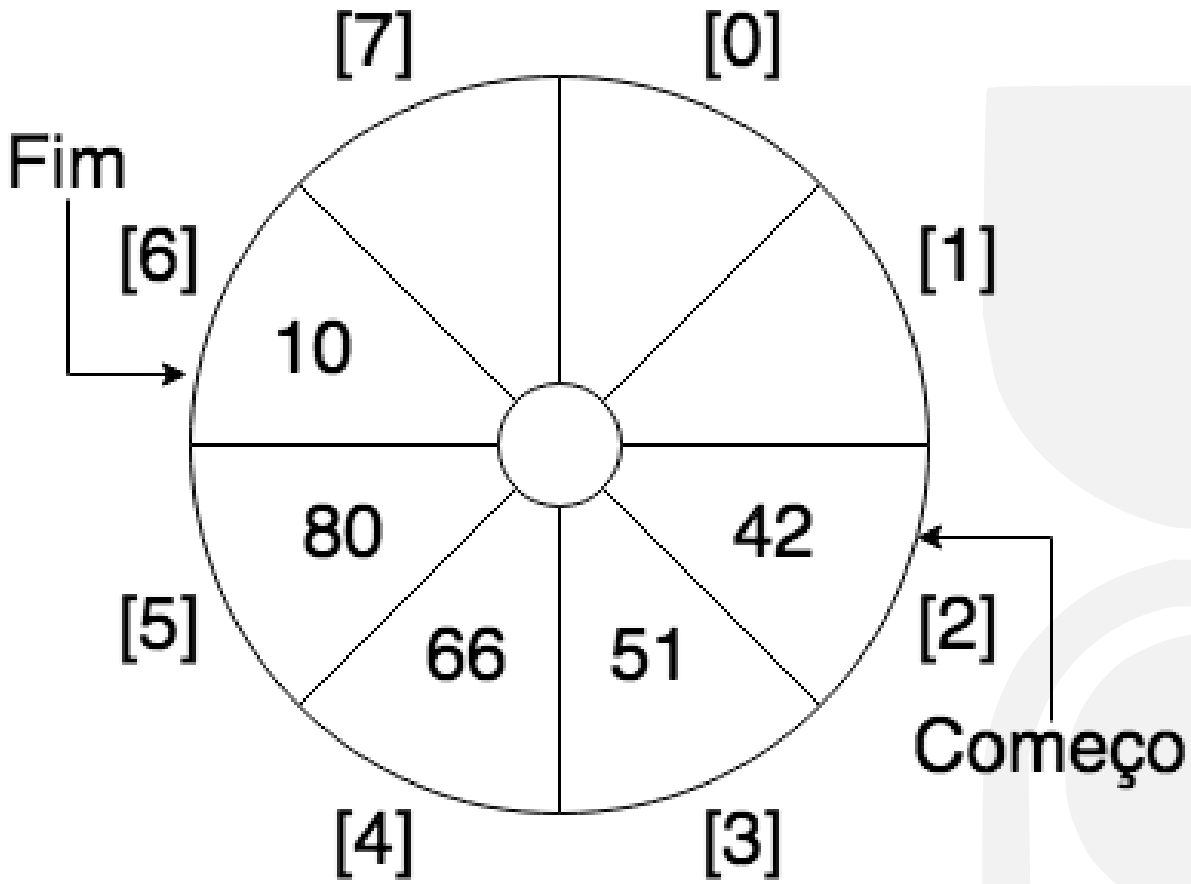
Buffers circulares

- 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

Buffers circulares

- 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

Buffers circulares



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

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

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

- 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

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

- 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
```

Algoritmos de ordenação

- 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**

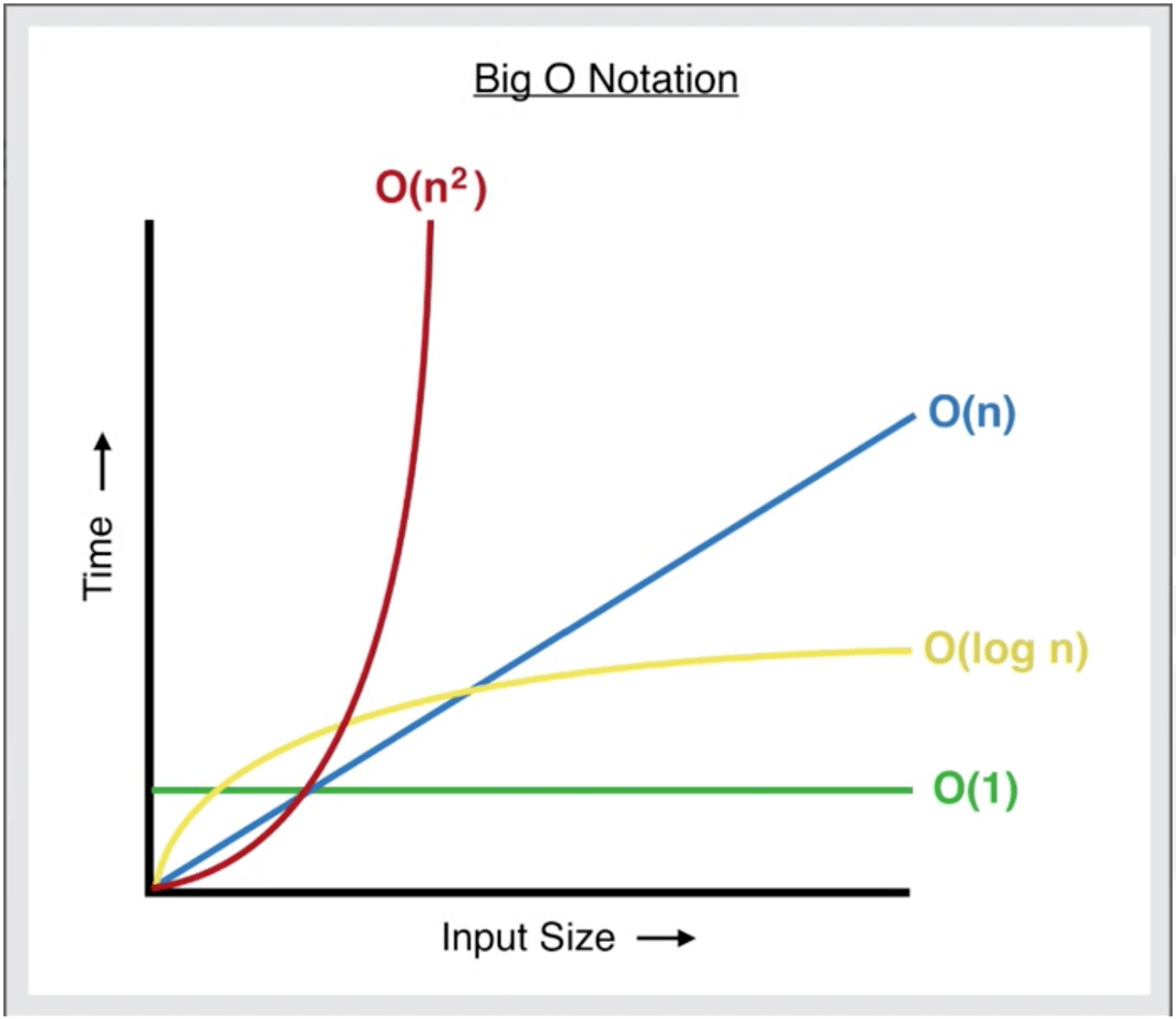
Complexidade de algoritmos

- 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**

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**

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**

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**

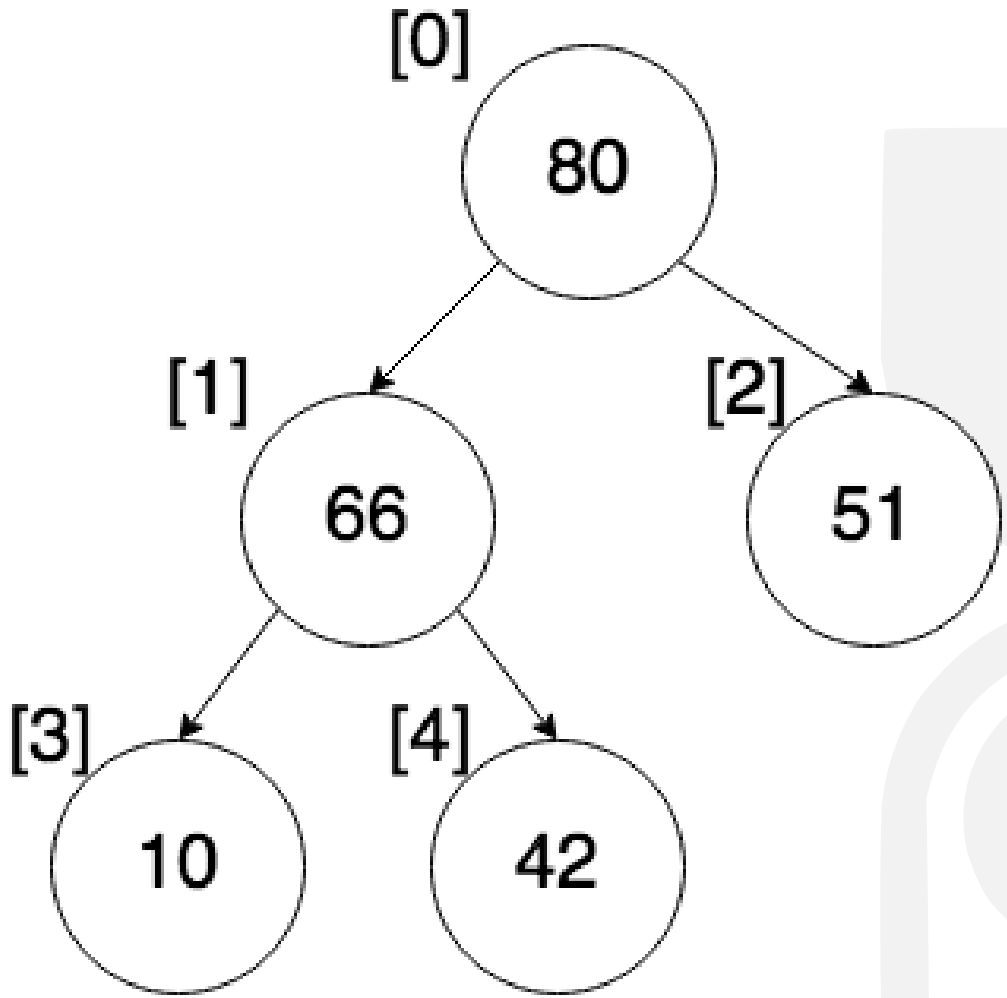
Monte binário

- 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**

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**

Monte binário



Complexidade de algoritmos, continuando

- ...