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

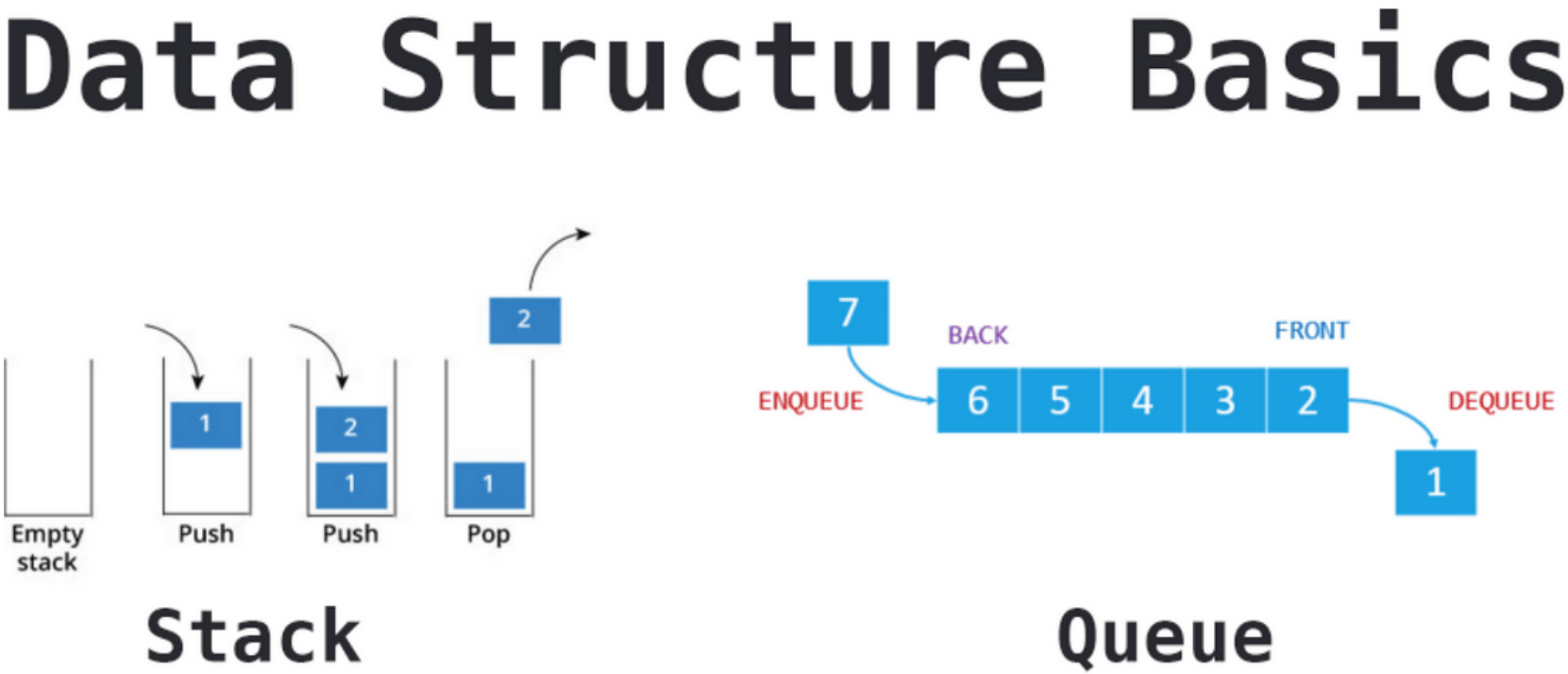
- Dentre as coleções clássica estudadas, podemos citas as **filas** e as **pilhas**
- Uma fila representa uma estrutura de dados **FIFO** (do inglês, *first-in, first-out*)
  - Podemos **enfileirar** um objeto (*enqueue*), o adicionando no **fim** da fila
  - E podemos **desenfileirar** um objeto (*dequeue*),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

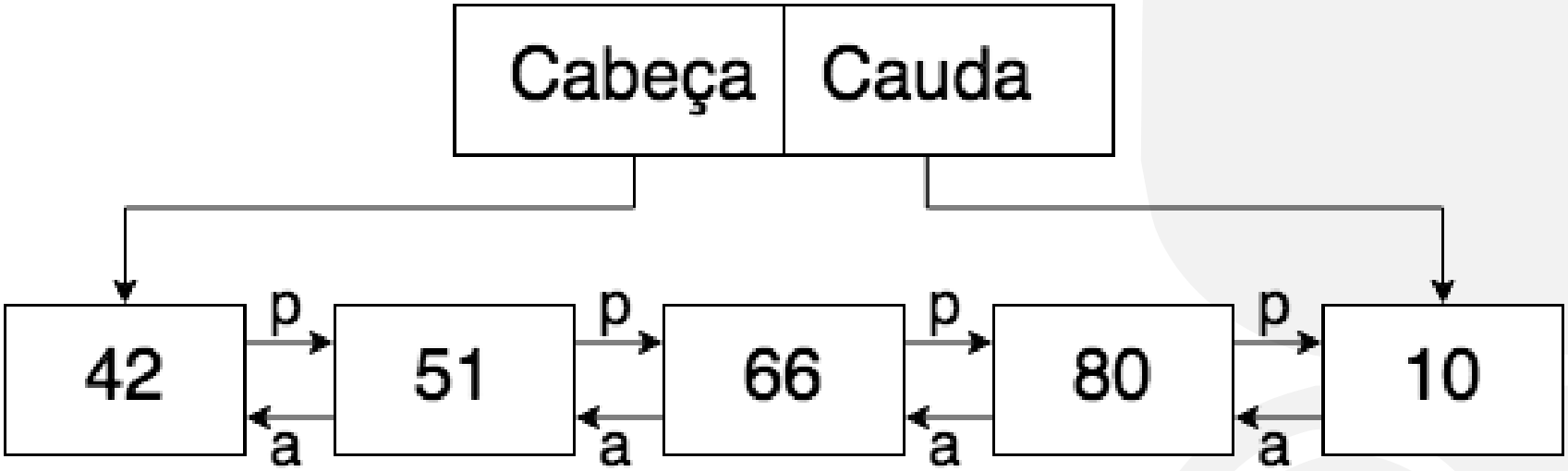
Listas duplamente linkadas

- 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

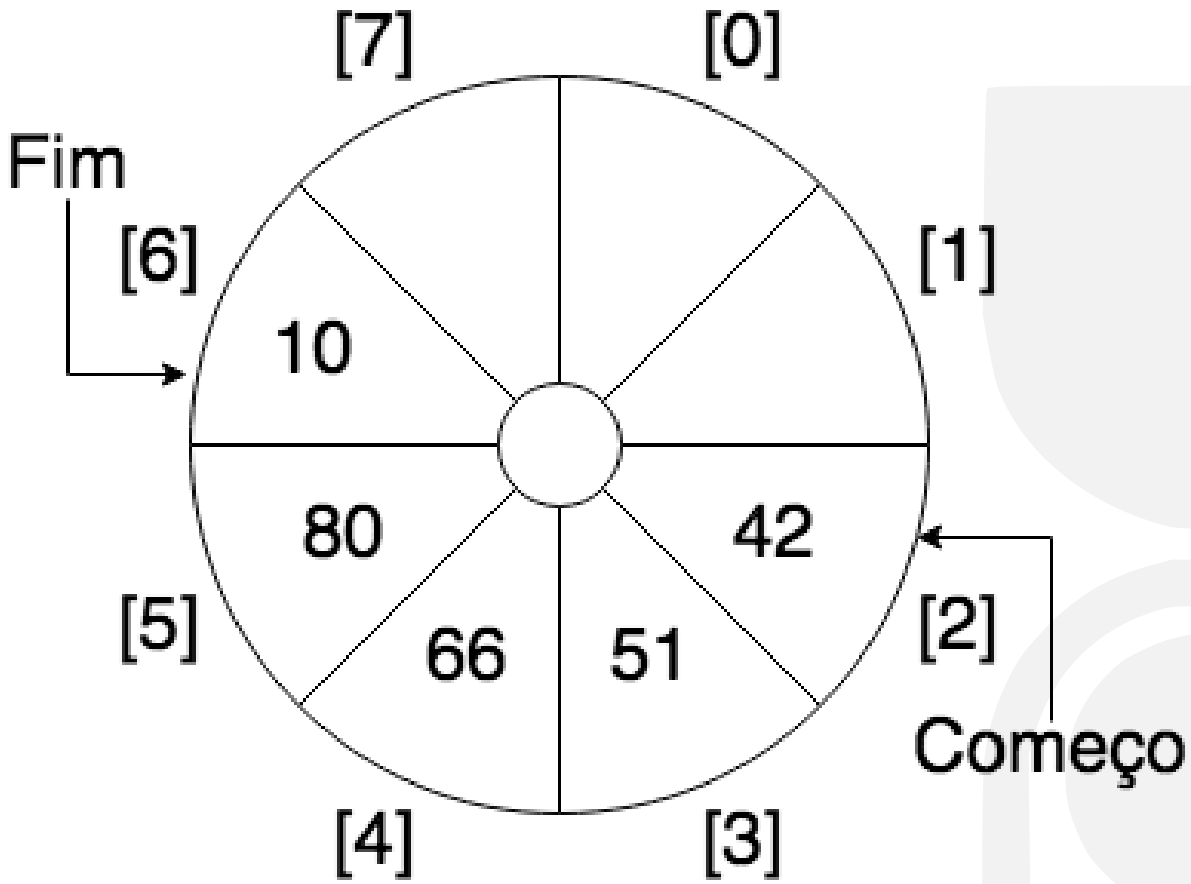
Buffers circulares

- 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
```

- 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**

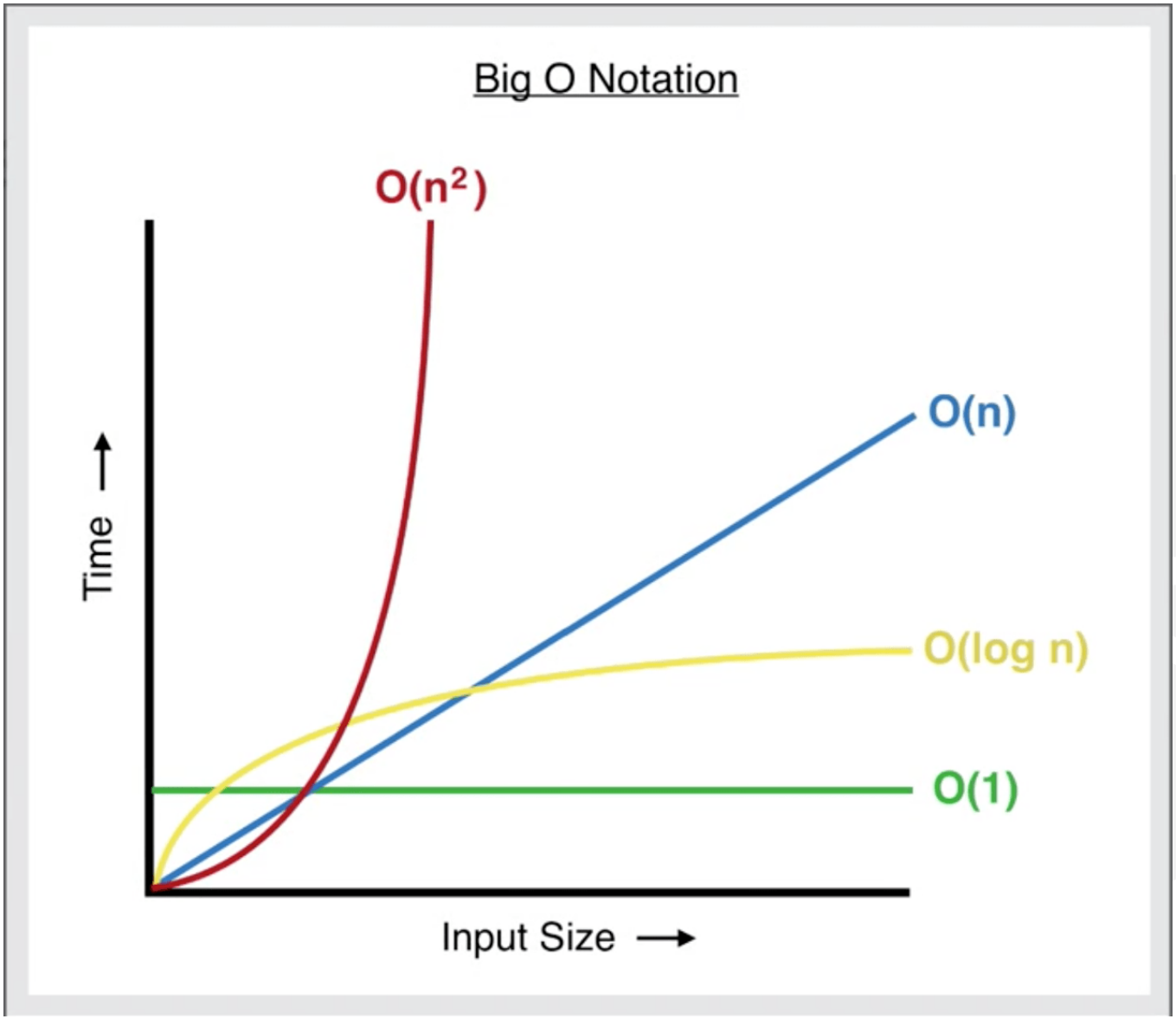
Complexidade de algoritmos

- 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**

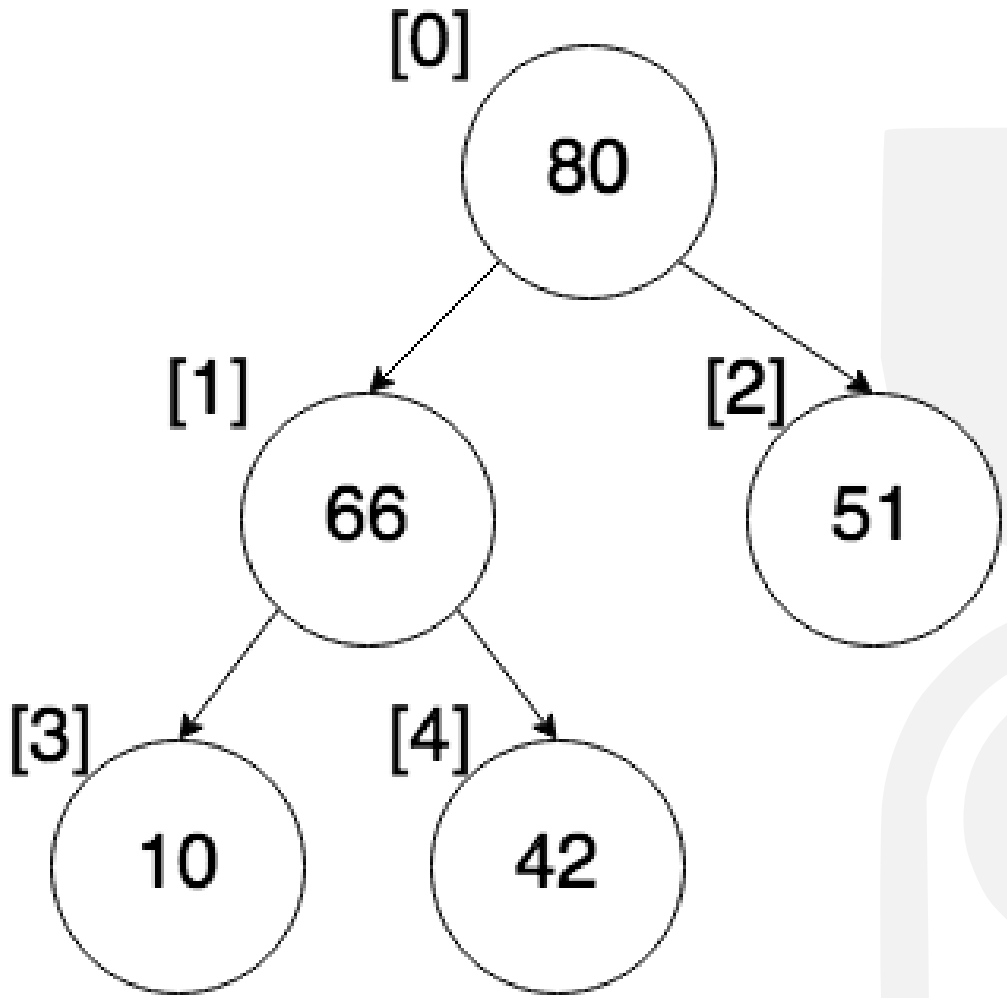
Monte binário

- 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



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)
```

Monte binário

- 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 raíz)
- Assim, podemos definir o algoritmo **heapsort** em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raíz para o fim do *buffer*

Monte binário

- 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 raíz)
- Assim, podemos definir o algoritmo **heapsort** em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raíz para o fim do *buffer*

Monte binário

- 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 raíz)
- Assim, podemos definir o algoritmo **heapsort** em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raíz para o fim do *buffer*

Monte binário

- 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 raíz)
- Assim, podemos definir o algoritmo **heapsort** em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raíz para o fim do *buffer*

Monte binário

- 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 raíz)
- Assim, podemos definir o algoritmo **heapsort** em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raíz para o fim do *buffer*

Monte binário

- 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 raíz)
- Assim, podemos definir o algoritmo **heapsort** em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raíz para o fim do *buffer*

Monte binário

- 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 raíz)
- Assim, podemos definir o algoritmo **heapsort** em dois passos: transformando um *buffer* em um monte, e sucessivamente movendo o maior elemento da raíz 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	$\Omega(n \log n)$	$O(n \log n)$

Estruturas associativas

- Coleções como filas, pilhas, vetores, etc, são usadas para agrupar elementos (valores) isoladamente
- Muitas vezes, precisamos de estruturas **associativas**, isto é, representam pares entre **chaves** e **valores** dentro da estrutura
- Vários tipos de estruturas similares existem, dentre elas as listas associativas, árvores de busca binária, tabelas *hash*, etc
  - Dentre as operações definidas para essas estruturas estão formas de se encontrar e se atualizar valores associados a chaves específicas
  - As estruturas usadas com mais frequência são otimizadas para busca e acesso, fornecendo formas eficientes de se acessar o valor associado a uma chave
  - Algumas versões especializadas existem; por exemplo, árvores B+ são estruturas usadas para a implementação de sistemas de arquivos e bancos de dados

Estruturas associativas

- Coleções como filas, pilhas, vetores, etc, são usadas para agrupar elementos (valores) isoladamente
- Muitas vezes, precisamos de estruturas **associativas**, isto é, representam pares entre **chaves** e **valores** dentro da estrutura
- Vários tipos de estruturas similares existem, dentre elas as listas associativas, árvores de busca binária, tabelas *hash*, etc
  - Dentre as operações definidas para essas estruturas estão formas de se encontrar e se atualizar valores associados a chaves específicas
  - As estruturas usadas com mais frequência são otimizadas para busca e acesso, fornecendo formas eficientes de se acessar o valor associado a uma chave
  - Algumas versões especializadas existem; por exemplo, árvores B+ são estruturas usadas para a implementação de sistemas de arquivos e bancos de dados

Estruturas associativas

- Coleções como filas, pilhas, vetores, etc, são usadas para agrupar elementos (valores) isoladamente
- Muitas vezes, precisamos de estruturas **associativas**, isto é, representam pares entre **chaves** e **valores** dentro da estrutura
- Vários tipos de estruturas similares existem, dentre elas as listas associativas, árvores de busca binária, tabelas *hash*, etc
    - Dentre as operações definidas para essas estruturas estão formas de se encontrar e se atualizar valores associados a chaves específicas
    - As estruturas usadas com mais frequência são otimizadas para busca e acesso, fornecendo formas eficientes de se acessar o valor associado a uma chave
    - Algumas versões especializadas existem; por exemplo, árvores B+ são estruturas usadas para a implementação de sistemas de arquivos e bancos de dados

Estruturas associativas

- Coleções como filas, pilhas, vetores, etc, são usadas para agrupar elementos (valores) isoladamente
- Muitas vezes, precisamos de estruturas **associativas**, isto é, representam pares entre **chaves** e **valores** dentro da estrutura
- Vários tipos de estruturas similares existem, dentre elas as listas associativas, árvores de busca binária, tabelas *hash*, etc
  - Dentre as operações definidas para essas estruturas estão formas de se encontrar e se atualizar valores associados a chaves específicas
  - As estruturas usadas com mais frequência são otimizadas para busca e acesso, fornecendo formas eficientes de se acessar o valor associado a uma chave
  - Algumas versões especializadas existem; por exemplo, árvores B+ são estruturas usadas para a implementação de sistemas de arquivos e bancos de dados

Estruturas associativas

- Coleções como filas, pilhas, vetores, etc, são usadas para agrupar elementos (valores) isoladamente
- Muitas vezes, precisamos de estruturas **associativas**, isto é, representam pares entre **chaves** e **valores** dentro da estrutura
- Vários tipos de estruturas similares existem, dentre elas as listas associativas, árvores de busca binária, tabelas *hash*, etc
  - Dentre as operações definidas para essas estruturas estão formas de se encontrar e se atualizar valores associados a chaves específicas
  - As estruturas usadas com mais frequência são otimizadas para busca e acesso, fornecendo formas eficientes de se acessar o valor associado a uma chave
  - Algumas versões especializadas existem; por exemplo, árvores B+ são estruturas usadas para a implementação de sistemas de arquivos e bancos de dados

Estruturas associativas

- Coleções como filas, pilhas, vetores, etc, são usadas para agrupar elementos (valores) isoladamente
- Muitas vezes, precisamos de estruturas **associativas**, isto é, representam pares entre **chaves** e **valores** dentro da estrutura
- Vários tipos de estruturas similares existem, dentre elas as listas associativas, árvores de busca binária, tabelas *hash*, etc
  - Dentre as operações definidas para essas estruturas estão formas de se encontrar e se atualizar valores associados a chaves específicas
  - As estruturas usadas com mais frequência são otimizadas para busca e acesso, fornecendo formas eficientes de se acessar o valor associado a uma chave
  - Algumas versões especializadas existem; por exemplo, árvores B+ são estruturas usadas para a implementação de sistemas de arquivos e bancos de dados

Listas associativas

- **Listas associativas** (do inglês, *association lists*) são estruturas em forma de lista (tanto **simplesmente linkada** quanto **duplamente linkada**), que, além de salvarem um valor, como visto anteriormente, **também salvam a chave**
- A inserção, remoção e busca de itens numa lista associativa tem complexidade O(n), linear
  - Por exemplo, caso não desejemos ter itens repetidos, ao se inserir um novo item, precisamos primeiro percorrer toda a lista procurando se já existe um item na lista com a chave desejada
  - Se o item já existir, devemos apenas atualizar o valor
  - Entretanto, no pior caso, o item não existe, porém toda a lista precisou ser percorrida; o item então pode ser inserido no começo da lista (ou no final, visto que precisamos chegar ao último elemento de qualquer forma)
- Embora possua uma implementação simples, esse tipo de estrutura não é eficiente para grandes conjuntos de dados

Listas associativas

- **Listas associativas** (do inglês, *association lists*) são estruturas em forma de lista (tanto **simplesmente linkada** quanto **duplamente linkada**), que, além de salvarem um valor, como visto anteriormente, **também salvam a chave**
- A inserção, remoção e busca de itens numa lista associativa tem complexidade O(n), linear
  - Por exemplo, caso não desejemos ter itens repetidos, ao se inserir um novo item, precisamos primeiro percorrer toda a lista procurando se já existe um item na lista com a chave desejada
  - Se o item já existir, devemos apenas atualizar o valor
  - Entretanto, no pior caso, o item não existe, porém toda a lista precisou ser percorrida; o item então pode ser inserido no começo da lista (ou no final, visto que precisamos chegar ao último elemento de qualquer forma)
- Embora possua uma implementação simples, esse tipo de estrutura não é eficiente para grandes conjuntos de dados

Listas associativas

- **Listas associativas** (do inglês, *association lists*) são estruturas em forma de lista (tanto **simplesmente linkada** quanto **duplamente linkada**), que, além de salvarem um valor, como visto anteriormente, **também salvam a chave**
- A inserção, remoção e busca de itens numa lista associativa tem complexidade O(n), linear
  - Por exemplo, caso não desejemos ter itens repetidos, ao se inserir um novo item, precisamos primeiro percorrer toda a lista procurando se já existe um item na lista com a chave desejada
  - Se o item já existir, devemos apenas atualizar o valor
  - Entretanto, no pior caso, o item não existe, porém toda a lista precisou ser percorrida; o item então pode ser inserido no começo da lista (ou no final, visto que precisamos chegar ao último elemento de qualquer forma)
- Embora possua uma implementação simples, esse tipo de estrutura não é eficiente para grandes conjuntos de dados

Listas associativas

- **Listas associativas** (do inglês, *association lists*) são estruturas em forma de lista (tanto **simplesmente linkada** quanto **duplamente linkada**), que, além de salvarem um valor, como visto anteriormente, **também salvam a chave**
- A inserção, remoção e busca de itens numa lista associativa tem complexidade O(n), linear
  - Por exemplo, caso não desejemos ter itens repetidos, ao se inserir um novo item, precisamos primeiro percorrer toda a lista procurando se já existe um item na lista com a chave desejada
  - Se o item já existir, devemos apenas atualizar o valor
  - Entretanto, no pior caso, o item não existe, porém toda a lista precisou ser percorrida; o item então pode ser inserido no começo da lista (ou no final, visto que precisamos chegar ao último elemento de qualquer forma)
- Embora possua uma implementação simples, esse tipo de estrutura não é eficiente para grandes conjuntos de dados

Listas associativas

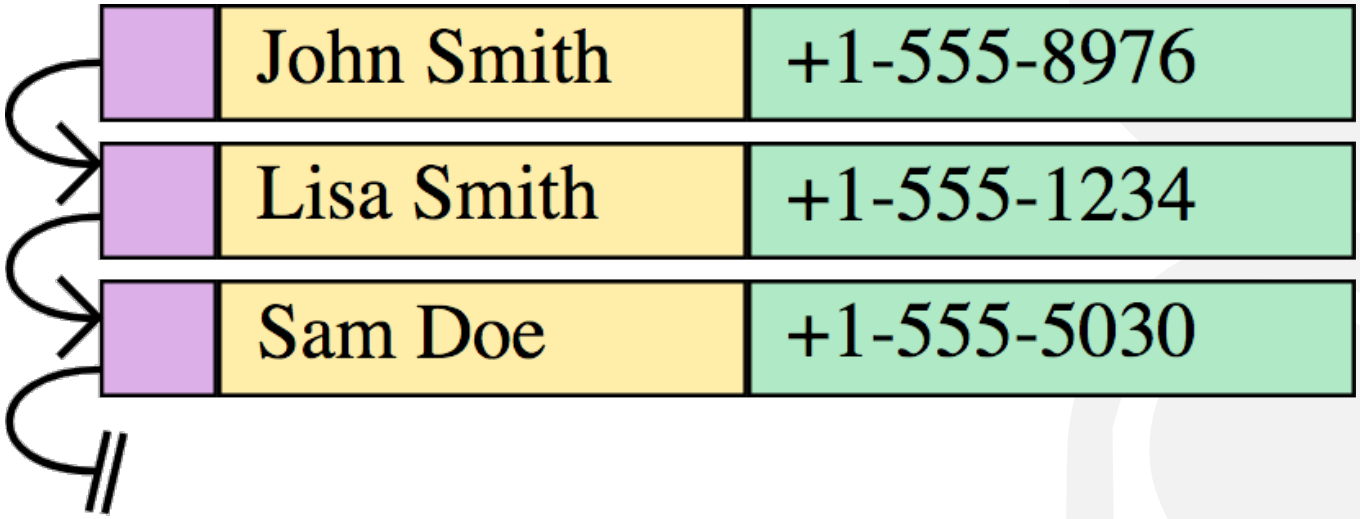
- **Listas associativas** (do inglês, *association lists*) são estruturas em forma de lista (tanto **simplesmente linkada** quanto **duplamente linkada**), que, além de salvarem um valor, como visto anteriormente, **também salvam a chave**
- A inserção, remoção e busca de itens numa lista associativa tem complexidade O(n), linear
  - Por exemplo, caso não desejemos ter itens repetidos, ao se inserir um novo item, precisamos primeiro percorrer toda a lista procurando se já existe um item na lista com a chave desejada
  - Se o item já existir, devemos apenas atualizar o valor
  - Entretanto, no pior caso, o item não existe, porém toda a lista precisou ser percorrida; o item então pode ser inserido no começo da lista (ou no final, visto que precisamos chegar ao último elemento de qualquer forma)
- Embora possua uma implementação simples, esse tipo de estrutura não é eficiente para grandes conjuntos de dados

Listas associativas

- **Listas associativas** (do inglês, *association lists*) são estruturas em forma de lista (tanto **simplesmente linkada** quanto **duplamente linkada**), que, além de salvarem um valor, como visto anteriormente, **também salvam a chave**
- A inserção, remoção e busca de itens numa lista associativa tem complexidade O(n), linear
  - Por exemplo, caso não desejemos ter itens repetidos, ao se inserir um novo item, precisamos primeiro percorrer toda a lista procurando se já existe um item na lista com a chave desejada
  - Se o item já existir, devemos apenas atualizar o valor
  - Entretanto, no pior caso, o item não existe, porém toda a lista precisou ser percorrida; o item então pode ser inserido no começo da lista (ou no final, visto que precisamos chegar ao último elemento de qualquer forma)
- Embora possua uma implementação simples, esse tipo de estrutura não é eficiente para grandes conjuntos de dados

Listas associativas

Cada item da lista salva uma chave, um valor (que está associado a essa chave), e uma referência para o próximo item, e, caso a lista seja duplamente linkada, uma referência para o item anterior



Árvores

- Árvores de busca binária (*binary search trees*), ou apenas **árvores**, apresentam uma forma hierárquica de armazenar os dados, respeitando algumas **invariantes**
- A **raíz** de uma árvore é representada por um **nó**, onde cada nó **salva um valor**, e potencialmente aponta para dois outros nós (à sua **esquerda** e à sua **direita**)
  - Caso o nó à esquerda exista, **sua chave será menor que a chave do nó pai**
  - Respectivamente, caso o nó à direita exista, **sua chave será maior que a chave do nó pai**
  - Idealmente, podemos fazer na árvore um procedimento similar ao da busca binária: se a chave que desejamos encontrar for menor que a chave de um nó, continuamos buscando apenas na sua esquerda (e, se for maior, na sua direita)
  - **Ao inserir um item novo**, respeitamos essa regra: **procuramos o lugar para inserí-lo, nos movendo na direção apropriada**

Árvores

- Árvores de busca binária (*binary search trees*), ou apenas **árvores**, apresentam uma forma hierárquica de armazenar os dados, respeitando algumas **invariantes**
- A **raíz** de uma árvore é representada por um **nó**, onde cada nó **salva um valor**, e potencialmente aponta para dois outros nós (à sua **esquerda** e à sua **direita**)
  - Caso o nó à esquerda exista, **sua chave será menor que a chave do nó pai**
  - Respectivamente, caso o nó à direita exista, **sua chave será maior que a chave do nó pai**
  - Idealmente, podemos fazer na árvore um procedimento similar ao da busca binária: se a chave que desejamos encontrar for menor que a chave de um nó, continuamos buscando apenas na sua esquerda (e, se for maior, na sua direita)
  - **Ao inserir um item novo**, respeitamos essa regra: **procuramos o lugar para inserí-lo, nos movendo na direção apropriada**

Árvores

- Árvores de busca binária (*binary search trees*), ou apenas **árvores**, apresentam uma forma hierárquica de armazenar os dados, respeitando algumas **invariantes**
- A **raíz** de uma árvore é representada por um **nó**, onde cada nó **salva um valor**, e potencialmente aponta para dois outros nós (à sua **esquerda** e à sua **direita**)
  - Caso o nó à esquerda exista, **sua chave será menor que a chave do nó pai**
  - Respectivamente, caso o nó à direita exista, **sua chave será maior que a chave do nó pai**
  - Idealmente, podemos fazer na árvore um procedimento similar ao da busca binária: se a chave que desejamos encontrar for menor que a chave de um nó, continuamos buscando apenas na sua esquerda (e, se for maior, na sua direita)
  - **Ao inserir um item novo**, respeitamos essa regra: **procuramos o lugar para inserí-lo, nos movendo na direção apropriada**

Árvores

- Árvores de busca binária (*binary search trees*), ou apenas **árvores**, apresentam uma forma hierárquica de armazenar os dados, respeitando algumas **invariantes**
- A **raíz** de uma árvore é representada por um **nó**, onde cada nó **salva um valor**, e potencialmente aponta para dois outros nós (à sua **esquerda** e à sua **direita**)
  - Caso o nó à esquerda exista, **sua chave será menor que a chave do nó pai**
  - Respectivamente, caso o nó à direita exista, **sua chave será maior que a chave do nó pai**
  - Idealmente, podemos fazer na árvore um procedimento similar ao da busca binária: se a chave que desejamos encontrar for menor que a chave de um nó, continuamos buscando apenas na sua esquerda (e, se for maior, na sua direita)
  - **Ao inserir um item novo**, respeitamos essa regra: **procuramos o lugar para inserí-lo, nos movendo na direção apropriada**

Árvores

- Árvores de busca binária (*binary search trees*), ou apenas **árvores**, apresentam uma forma hierárquica de armazenar os dados, respeitando algumas **invariantes**
- A **raíz** de uma árvore é representada por um **nó**, onde cada nó **salva um valor**, e potencialmente aponta para dois outros nós (à sua **esquerda** e à sua **direita**)
  - Caso o nó à esquerda exista, **sua chave será menor que a chave do nó pai**
  - Respectivamente, caso o nó à direita exista, **sua chave será maior que a chave do nó pai**
  - Idealmente, podemos fazer na árvore um procedimento similar ao da busca binária: se a chave que desejamos encontrar for menor que a chave de um nó, continuamos buscando apenas na sua esquerda (e, se for maior, na sua direita)
  - **Ao inserir um item novo**, respeitamos essa regra: **procuramos o lugar para inserí-lo, nos movendo na direção apropriada**

Árvores

- Árvores de busca binária (*binary search trees*), ou apenas **árvores**, apresentam uma forma hierárquica de armazenar os dados, respeitando algumas **invariantes**
- A **raíz** de uma árvore é representada por um **nó**, onde cada nó **salva um valor**, e potencialmente aponta para dois outros nós (à sua **esquerda** e à sua **direita**)
  - Caso o nó à esquerda exista, **sua chave será menor que a chave do nó pai**
  - Respectivamente, caso o nó à direita exista, **sua chave será maior que a chave do nó pai**
  - Idealmente, podemos fazer na árvore um procedimento similar ao da busca binária: se a chave que desejamos encontrar for menor que a chave de um nó, continuamos buscando apenas na sua esquerda (e, se for maior, na sua direita)
  - **Ao inserir um item novo**, respeitamos essa regra: **procuramos o lugar para inserí-lo, nos movendo na direção apropriada**

Árvores

- Dizemos que a **altura de uma árvore** é a altura da sua raíz (que é um nó), sendo zero se a árvore estiver vazia
- Dizemos que a **altura de um nó**, se ele existir, é de 1 mais o tamanho do seu maior nó filho, considerando altura zero para o nós filhos que não existem
- Uma árvore é considerada **balanceada** se a **diferença entre a altura dos filhos** (à esquerda e à direita) **de todos os nós**, também chamada de fator, for no máximo 1
  - Por exemplo, ao visualizarmos um monte como uma árvore, notamos que ele sempre está balanceado
  - Uma árvore é dita **completa** (ou **cheia**) se os filhos de todos os nós tem a mesma altura entre si (fator zero)
- Pelos mesmos motivos da busca binária, operações em árvores passam a ser logarítmicas, O(log n), **caso a árvore esteja balanceada**, pois **sabemos que metade dos dados está para cada um dos lados!**, mas linear caso ela não esteja

Árvores

- Dizemos que a **altura de uma árvore** é a altura da sua raíz (que é um nó), sendo zero se a árvore estiver vazia
- Dizemos que a **altura de um nó**, se ele existir, é de 1 mais o tamanho do seu maior nó filho, considerando altura zero para o nós filhos que não existem
- Uma árvore é considerada **balanceada** se a **diferença entre a altura dos filhos** (à esquerda e à direita) **de todos os nós**, também chamada de fator, for no máximo 1
  - Por exemplo, ao visualizarmos um monte como uma árvore, notamos que ele sempre está balanceado
  - Uma árvore é dita **completa** (ou **cheia**) se os filhos de todos os nós tem a mesma altura entre si (fator zero)
- Pelos mesmos motivos da busca binária, operações em árvores passam a ser logarítmicas, O(log n), **caso a árvore esteja balanceada**, pois **sabemos que metade dos dados está para cada um dos lados!**, mas linear caso ela não esteja

Árvores

- Dizemos que a **altura de uma árvore** é a altura da sua raíz (que é um nó), sendo zero se a árvore estiver vazia
- Dizemos que a **altura de um nó**, se ele existir, é de 1 mais o tamanho do seu maior nó filho, considerando altura zero para o nós filhos que não existem
- Uma árvore é considerada **balanceada** se a **diferença entre a altura dos filhos** (à esquerda e à direita) **de todos os nós**, também chamada de fator, for no máximo 1
  - Por exemplo, ao visualizarmos um monte como uma árvore, notamos que ele sempre está balanceado
  - Uma árvore é dita **completa** (ou **cheia**) se os filhos de todos os nós tem a mesma altura entre si (fator zero)
- Pelos mesmos motivos da busca binária, operações em árvores passam a ser logarítmicas, O(log n), **caso a árvore esteja balanceada**, pois **sabemos que metade dos dados está para cada um dos lados!**, mas linear caso ela não esteja

Árvores

- Dizemos que a **altura de uma árvore** é a altura da sua raíz (que é um nó), sendo zero se a árvore estiver vazia
- Dizemos que a **altura de um nó**, se ele existir, é de 1 mais o tamanho do seu maior nó filho, considerando altura zero para o nós filhos que não existem
- Uma árvore é considerada **balanceada** se a **diferença entre a altura dos filhos** (à esquerda e à direita) **de todos os nós**, também chamada de fator, for no máximo 1
  - Por exemplo, ao visualizarmos um monte como uma árvore, notamos que ele sempre está balanceado
  - Uma árvore é dita **completa** (ou **cheia**) se os filhos de todos os nós tem a mesma altura entre si (fator zero)
- Pelos mesmos motivos da busca binária, operações em árvores passam a ser logarítmicas, O(log n), **caso a árvore esteja balanceada**, pois **sabemos que metade dos dados está para cada um dos lados!**, mas linear caso ela não esteja

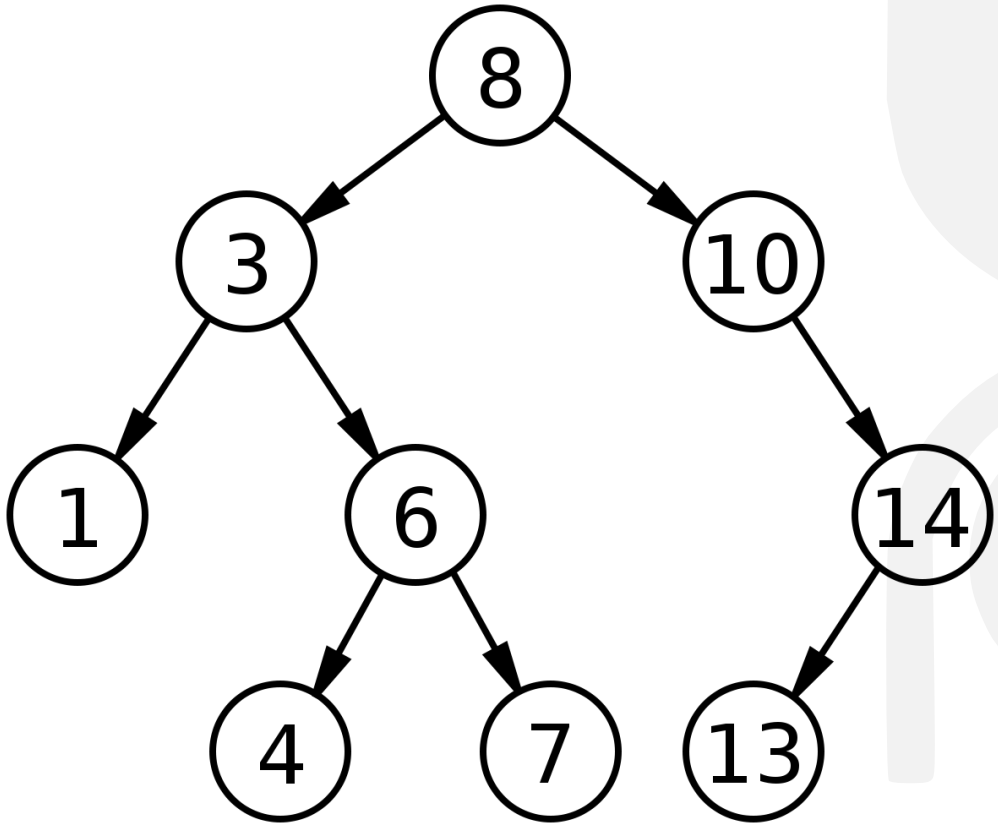
Árvores

- Dizemos que a **altura de uma árvore** é a altura da sua raíz (que é um nó), sendo zero se a árvore estiver vazia
- Dizemos que a **altura de um nó**, se ele existir, é de 1 mais o tamanho do seu maior nó filho, considerando altura zero para o nós filhos que não existem
- Uma árvore é considerada **balanceada** se a **diferença entre a altura dos filhos** (à esquerda e à direita) **de todos os nós**, também chamada de fator, for no máximo 1
  - Por exemplo, ao visualizarmos um monte como uma árvore, notamos que ele sempre está balanceado
  - Uma árvore é dita **completa** (ou **cheia**) se os filhos de todos os nós tem a mesma altura entre si (fator zero)
- Pelos mesmos motivos da busca binária, operações em árvores passam a ser logarítmicas, O(log n), **caso a árvore esteja balanceada**, pois **sabemos que metade dos dados está para cada um dos lados!**, mas linear caso ela não esteja

Árvores

- Dizemos que a **altura de uma árvore** é a altura da sua raíz (que é um nó), sendo zero se a árvore estiver vazia
- Dizemos que a **altura de um nó**, se ele existir, é de 1 mais o tamanho do seu maior nó filho, considerando altura zero para o nós filhos que não existem
- Uma árvore é considerada **balanceada** se a **diferença entre a altura dos filhos** (à esquerda e à direita) **de todos os nós**, também chamada de fator, for no máximo 1
  - Por exemplo, ao visualizarmos um monte como uma árvore, notamos que ele sempre está balanceado
  - Uma árvore é dita **completa** (ou **cheia**) se os filhos de todos os nós tem a mesma altura entre si (fator zero)
- Pelos mesmos motivos da busca binária, operações em árvores passam a ser logarítmicas, O(log n), **caso a árvore esteja balanceada**, pois **sabemos que metade dos dados está para cada um dos lados!**, mas linear caso ela não esteja

Um exemplo de árvore, usando números como chaves (e omitindo valores). Note que a árvore não se encontra balanceada: a esquerda do nó de chave 10 tem altura 0, porém sua direita tem altura 2



Árvores balanceadas

- Árvores, se implementadas de forma ingênua, deixam de ser balanceadas facilmente
  - Imagine, por exemplo, que você está **inserindo** 100 **elementos ordenados**: eles sempre serão colocados à direita (visto que o primeiro elemento a ser inserido era o menor), obtendo performance similar a de uma lista associativa no pior caso
  - É possível **balancear uma árvore manualmente**, porém isso **é feito em tempo linear**
- **Existem formas de se implementar uma árvore** que, ao se inserir um elemento (que ainda não exista), ou ao se remover um elemento, a árvore seja **automaticamente balanceada**
  - Podemos citar aqui, por exemplo, **árvores AVL** e **árvores rubro-negras** (*red-black*) como versões existentes
  - Considerando árvores como **tipos abstratos**, o algoritmo de balanceamento automático se torna um **detalhe de implementação** que não importa ao usuário do código
  - Em tais árvores, inserção e remoção (e busca) podem ser feitas em tempo logarítmico, pois **corrigir o balanceamento de uma árvore pode ser executado em O(log n)**

Árvores balanceadas

- Árvores, se implementadas de forma ingênua, deixam de ser balanceadas facilmente
  - Imagine, por exemplo, que você está **inserindo** 100 **elementos ordenados**: eles sempre serão colocados à direita (visto que o primeiro elemento a ser inserido era o menor), obtendo performance similar a de uma lista associativa no pior caso
  - É possível **balancear uma árvore manualmente**, porém isso **é feito em tempo linear**
- **Existem formas de se implementar uma árvore** que, ao se inserir um elemento (que ainda não exista), ou ao se remover um elemento, a árvore seja **automaticamente balanceada**
  - Podemos citar aqui, por exemplo, **árvores AVL** e **árvores rubro-negras** (*red-black*) como versões existentes
  - Considerando árvores como **tipos abstratos**, o algoritmo de balanceamento automático se torna um **detalhe de implementação** que não importa ao usuário do código
  - Em tais árvores, inserção e remoção (e busca) podem ser feitas em tempo logarítmico, pois **corrigir o balanceamento de uma árvore pode ser executado em O(log n)**

Árvores balanceadas

- Árvores, se implementadas de forma ingênua, deixam de ser balanceadas facilmente
  - Imagine, por exemplo, que você está **inserindo** 100 **elementos ordenados**: eles sempre serão colocados à direita (visto que o primeiro elemento a ser inserido era o menor), obtendo performance similar a de uma lista associativa no pior caso
  - É possível **balancear uma árvore manualmente**, porém isso **é feito em tempo linear**
- **Existem formas de se implementar uma árvore** que, ao se inserir um elemento (que ainda não exista), ou ao se remover um elemento, a árvore seja **automaticamente balanceada**
  - Podemos citar aqui, por exemplo, **árvores AVL** e **árvores rubro-negras** (*red-black*) como versões existentes
  - Considerando árvores como **tipos abstratos**, o algoritmo de balanceamento automático se torna um **detalhe de implementação** que não importa ao usuário do código
  - Em tais árvores, inserção e remoção (e busca) podem ser feitas em tempo logarítmico, pois **corrigir o balanceamento de uma árvore pode ser executado em O(log n)**

Árvores balanceadas

- Árvores, se implementadas de forma ingênua, deixam de ser balanceadas facilmente
  - Imagine, por exemplo, que você está **inserindo** 100 **elementos ordenados**: eles sempre serão colocados à direita (visto que o primeiro elemento a ser inserido era o menor), obtendo performance similar a de uma lista associativa no pior caso
  - É possível **balancear uma árvore manualmente**, porém isso **é feito em tempo linear**
- **Existem formas de se implementar uma árvore** que, ao se inserir um elemento (que ainda não exista), ou ao se remover um elemento, a árvore seja **automaticamente balanceada**
  - Podemos citar aqui, por exemplo, **árvores AVL** e **árvores rubro-negras** (*red-black*) como versões existentes
  - Considerando árvores como **tipos abstratos**, o algoritmo de balanceamento automático se torna um **detalhe de implementação** que não importa ao usuário do código
  - Em tais árvores, inserção e remoção (e busca) podem ser feitas em tempo logarítmico, pois **corrigir o balanceamento de uma árvore pode ser executado em O(log n)**

Árvores balanceadas

- Árvores, se implementadas de forma ingênua, deixam de ser balanceadas facilmente
  - Imagine, por exemplo, que você está **inserindo** 100 **elementos ordenados**: eles sempre serão colocados à direita (visto que o primeiro elemento a ser inserido era o menor), obtendo performance similar a de uma lista associativa no pior caso
  - É possível **balancear uma árvore manualmente**, porém isso **é feito em tempo linear**
- **Existem formas de se implementar uma árvore** que, ao se inserir um elemento (que ainda não exista), ou ao se remover um elemento, a árvore seja **automaticamente balanceada**
  - Podemos citar aqui, por exemplo, **árvores AVL** e **árvores rubro-negras** (*red-black*) como versões existentes
  - Considerando árvores como **tipos abstratos**, o algoritmo de balanceamento automático se torna um **detalhe de implementação** que não importa ao usuário do código
  - Em tais árvores, inserção e remoção (e busca) podem ser feitas em tempo logarítmico, pois **corrigir o balanceamento de uma árvore pode ser executado em O(log n)**

Árvores balanceadas

- Árvores, se implementadas de forma ingênua, deixam de ser balanceadas facilmente
  - Imagine, por exemplo, que você está **inserindo** 100 **elementos ordenados**: eles sempre serão colocados à direita (visto que o primeiro elemento a ser inserido era o menor), obtendo performance similar a de uma lista associativa no pior caso
  - É possível **balancear uma árvore manualmente**, porém isso **é feito em tempo linear**
- **Existem formas de se implementar uma árvore** que, ao se inserir um elemento (que ainda não exista), ou ao se remover um elemento, a árvore seja **automaticamente balanceada**
  - Podemos citar aqui, por exemplo, **árvores AVL** e **árvores rubro-negras** (*red-black*) como versões existentes
  - Considerando árvores como **tipos abstratos**, o algoritmo de balanceamento automático se torna um **detalhe de implementação** que não importa ao usuário do código
  - Em tais árvores, inserção e remoção (e busca) podem ser feitas em tempo logarítmico, pois **corrigir o balanceamento de uma árvore pode ser executado em O(log n)**

Árvores balanceadas

- Árvores, se implementadas de forma ingênua, deixam de ser balanceadas facilmente
  - Imagine, por exemplo, que você está **inserindo** 100 **elementos ordenados**: eles sempre serão colocados à direita (visto que o primeiro elemento a ser inserido era o menor), obtendo performance similar a de uma lista associativa no pior caso
  - É possível **balancear uma árvore manualmente**, porém isso **é feito em tempo linear**
- **Existem formas de se implementar uma árvore** que, ao se inserir um elemento (que ainda não exista), ou ao se remover um elemento, a árvore seja **automaticamente balanceada**
  - Podemos citar aqui, por exemplo, **árvores AVL** e **árvores rubro-negras** (*red-black*) como versões existentes
  - Considerando árvores como **tipos abstratos**, o algoritmo de balanceamento automático se torna um **detalhe de implementação** que não importa ao usuário do código
  - Em tais árvores, inserção e remoção (e busca) podem ser feitas em tempo logarítmico, pois **corrigir o balanceamento de uma árvore pode ser executado em O(log n)**

Tabela *hash*

- Tabelas *hash* (do inglês, *hashtable* ou *hashmap*) são uma estrutura associativa, que, similar às árvores, armazenam valores associados a uma chave
- Como nas árvores, chaves podem ser qualquer valor que possa ser ordenado, incluindo números inteiros e *strings*
- Tabelas *hash* utilizam uma técnica chamada de *hashing* (dispersão, espalhamento, etc) para decidir **onde** armazenar seu conteúdo
  - A tabela possui uma função de *hash*, **que gera um valor numérico para cada possível chave**
  - A tabela, então, possui uma coleção de **baldes** (*buckets*), geralmente salvos como um vetor, representando sua memória
  - O índice de um elemento na memória é `hash(key) % len`, isto é, **aproximamos** o *hash* da chave para a quantidade de baldes disponíveis (ainda tendo um valor **determinístico**)
  - Por conta disso, o **tempo de acesso médio** de um item em uma tabela *hash* é **constante**

Tabela *hash*

- Tabelas *hash* (do inglês, *hashtable* ou *hashmap*) são uma estrutura associativa, que, similar às árvores, armazenam valores associados a uma chave
- Como nas árvores, chaves podem ser qualquer valor que possa ser ordenado, incluindo números inteiros e *strings*
- Tabelas *hash* utilizam uma técnica chamada de *hashing* (dispersão, espalhamento, etc) para decidir **onde** armazenar seu conteúdo
  - A tabela possui uma função de *hash*, **que gera um valor numérico para cada possível chave**
  - A tabela, então, possui uma coleção de **baldes** (*buckets*), geralmente salvos como um vetor, representando sua memória
  - O índice de um elemento na memória é `hash(key) % len`, isto é, **aproximamos** o *hash* da chave para a quantidade de baldes disponíveis (ainda tendo um valor **determinístico**)
  - Por conta disso, o **tempo de acesso médio** de um item em uma tabela *hash* é **constante**

Tabela *hash*

- Tabelas *hash* (do inglês, *hashtable* ou *hashmap*) são uma estrutura associativa, que, similar às árvores, armazenam valores associados a uma chave
- Como nas árvores, chaves podem ser qualquer valor que possa ser ordenado, incluindo números inteiros e *strings*
- Tabelas *hash* utilizam uma técnica chamada de *hashing* (dispersão, espalhamento, etc) para decidir **onde** armazenar seu conteúdo
  - A tabela possui uma função de *hash*, **que gera um valor numérico para cada possível chave**
  - A tabela, então, possui uma coleção de **baldes** (*buckets*), geralmente salvos como um vetor, representando sua memória
  - O índice de um elemento na memória é `hash(key) % len`, isto é, **aproximamos** o *hash* da chave para a quantidade de baldes disponíveis (ainda tendo um valor **determinístico**)
  - Por conta disso, o **tempo de acesso médio** de um item em uma tabela *hash* é **constante**

Tabela *hash*

- Tabelas *hash* (do inglês, *hashtable* ou *hashmap*) são uma estrutura associativa, que, similar às árvores, armazenam valores associados a uma chave
- Como nas árvores, chaves podem ser qualquer valor que possa ser ordenado, incluindo números inteiros e *strings*
- Tabelas *hash* utilizam uma técnica chamada de *hashing* (dispersão, espalhamento, etc) para decidir **onde** armazenar seu conteúdo
  - A tabela possui uma função de *hash*, **que gera um valor numérico para cada possível chave**
  - A tabela, então, possui uma coleção de **baldes** (*buckets*), geralmente salvos como um vetor, representando sua memória
  - O índice de um elemento na memória é `hash(key) % len`, isto é, **aproximamos** o *hash* da chave para a quantidade de baldes disponíveis (ainda tendo um valor **determinístico**)
  - Por conta disso, o **tempo de acesso médio** de um item em uma tabela *hash* é **constante**

Tabela *hash*

- Tabelas *hash* (do inglês, *hashtable* ou *hashmap*) são uma estrutura associativa, que, similar às árvores, armazenam valores associados a uma chave
- Como nas árvores, chaves podem ser qualquer valor que possa ser ordenado, incluindo números inteiros e *strings*
- Tabelas *hash* utilizam uma técnica chamada de *hashing* (dispersão, espalhamento, etc) para decidir **onde** armazenar seu conteúdo
  - A tabela possui uma função de *hash*, **que gera um valor numérico para cada possível chave**
  - A tabela, então, possui uma coleção de **baldes** (*buckets*), geralmente salvos como um vetor, representando sua memória
  - O índice de um elemento na memória é `hash(key) % len`, isto é, **aproximamos** o *hash* da chave para a quantidade de baldes disponíveis (ainda tendo um valor **determinístico**)
  - Por conta disso, o **tempo de acesso médio** de um item em uma tabela *hash* é **constante**

Tabela *hash*

- Tabelas *hash* (do inglês, *hashtable* ou *hashmap*) são uma estrutura associativa, que, similar às árvores, armazenam valores associados a uma chave
- Como nas árvores, chaves podem ser qualquer valor que possa ser ordenado, incluindo números inteiros e *strings*
- Tabelas *hash* utilizam uma técnica chamada de *hashing* (dispersão, espalhamento, etc) para decidir **onde** armazenar seu conteúdo
  - A tabela possui uma função de *hash*, **que gera um valor numérico para cada possível chave**
  - A tabela, então, possui uma coleção de **baldes** (*buckets*), geralmente salvos como um vetor, representando sua memória
  - O índice de um elemento na memória é `hash(key) % len`, isto é, **aproximamos** o *hash* da chave para a quantidade de baldes disponíveis (ainda tendo um valor **determinístico**)
  - Por conta disso, o **tempo de acesso médio** de um item em uma tabela *hash* é **constante**

Tabela *hash*

- Tabelas *hash* (do inglês, *hashtable* ou *hashmap*) são uma estrutura associativa, que, similar às árvores, armazenam valores associados a uma chave
- Como nas árvores, chaves podem ser qualquer valor que possa ser ordenado, incluindo números inteiros e *strings*
- Tabelas *hash* utilizam uma técnica chamada de *hashing* (dispersão, espalhamento, etc) para decidir **onde** armazenar seu conteúdo
  - A tabela possui uma função de *hash*, **que gera um valor numérico para cada possível chave**
  - A tabela, então, possui uma coleção de **baldes** (*buckets*), geralmente salvos como um vetor, representando sua memória
  - O índice de um elemento na memória é `hash(key) % len`, isto é, **aproximamos** o *hash* da chave para a quantidade de baldes disponíveis (ainda tendo um valor **determinístico**)
  - Por conta disso, o **tempo de acesso médio** de um item em uma tabela *hash* é **constante**

Tabela *hash*

- Exemplo, **elf hash**:

```
unsigned long elf_hash(const char *s) {
  unsigned long h = 0;
  for(int i = 0; i < strlen(s); i++) {
    h = (h << 4) + s[i];
    unsigned long x = h & 0xF0000000;
    if(x != 0) {
      h ^= x >> 24;
      h &= ~x;
    }
  }
  return h;
}
```

Tabela *hash*

- Baldes são, similar aos elos de uma lista linkada, sub-estruturas necessárias para o armazenamento
- Como há um tamanho limitado para índices dentro da tabela, há a possibilidade de **colisões**
  - Funções de *hash* diferentes possuem propriedades diferentes quanto à frequência de colisões
- É possível salvarmos um balde, por exemplo, como uma lista associativa, isto é, uma lista de pares e valores
  - Neste caso, após encontrarmos um índice para uma chave, procuramos todos os itens salvos naquele balde pela chave
  - Caso o balde **esteja vazio ou contenha apenas um item**, a busca ocorre, como desejado, **em tempo constante**
  - Mas cuidado: a técnica conhecida como HashDoS tenta produzir muitas chaves com o mesmo *hash*, fazendo com que a busca seja **linear** no número de colisões!
- Idealmente, baldes devem ser implementados como árvores de busca binária, tornando o melhor tempo constante, e o **pior tempo logarítmico** em relação ao número de colisões

Tabela *hash*

- Baldes são, similar aos elos de uma lista linkada, sub-estruturas necessárias para o armazenamento
- Como há um tamanho limitado para índices dentro da tabela, há a possibilidade de **colisões**
  - Funções de *hash* diferentes possuem propriedades diferentes quanto à frequência de colisões
- É possível salvarmos um balde, por exemplo, como uma lista associativa, isto é, uma lista de pares e valores
  - Neste caso, após encontrarmos um índice para uma chave, procuramos todos os itens salvos naquele balde pela chave
  - Caso o balde **esteja vazio ou contenha apenas um item**, a busca ocorre, como desejado, **em tempo constante**
  - Mas cuidado: a técnica conhecida como HashDoS tenta produzir muitas chaves com o mesmo *hash*, fazendo com que a busca seja **linear** no número de colisões!
- Idealmente, baldes devem ser implementados como árvores de busca binária, tornando o melhor tempo constante, e o **pior tempo logarítmico** em relação ao número de colisões

Tabela *hash*

- Baldes são, similar aos elos de uma lista linkada, sub-estruturas necessárias para o armazenamento
- Como há um tamanho limitado para índices dentro da tabela, há a possibilidade de **colisões**
  - Funções de *hash* diferentes possuem propriedades diferentes quanto à frequência de colisões
- É possível salvarmos um balde, por exemplo, como uma lista associativa, isto é, uma lista de pares e valores
  - Neste caso, após encontrarmos um índice para uma chave, procuramos todos os itens salvos naquele balde pela chave
  - Caso o balde **esteja vazio ou contenha apenas um item**, a busca ocorre, como desejado, **em tempo constante**
  - Mas cuidado: a técnica conhecida como HashDoS tenta produzir muitas chaves com o mesmo *hash*, fazendo com que a busca seja **linear** no número de colisões!
- Idealmente, baldes devem ser implementados como árvores de busca binária, tornando o melhor tempo constante, e o **pior tempo logarítmico** em relação ao número de colisões

Tabela *hash*

- Baldes são, similar aos elos de uma lista linkada, sub-estruturas necessárias para o armazenamento
- Como há um tamanho limitado para índices dentro da tabela, há a possibilidade de **colisões**
  - Funções de *hash* diferentes possuem propriedades diferentes quanto à frequência de colisões
- É possível salvarmos um balde, por exemplo, como uma lista associativa, isto é, uma lista de pares e valores
  - Neste caso, após encontrarmos um índice para uma chave, procuramos todos os itens salvos naquele balde pela chave
  - Caso o balde **esteja vazio ou contenha apenas um item**, a busca ocorre, como desejado, **em tempo constante**
  - Mas cuidado: a técnica conhecida como HashDoS tenta produzir muitas chaves com o mesmo *hash*, fazendo com que a busca seja **linear** no número de colisões!
- Idealmente, baldes devem ser implementados como árvores de busca binária, tornando o melhor tempo constante, e o **pior tempo logarítmico** em relação ao número de colisões

Tabela *hash*

- Baldes são, similar aos elos de uma lista linkada, sub-estruturas necessárias para o armazenamento
- Como há um tamanho limitado para índices dentro da tabela, há a possibilidade de **colisões**
  - Funções de *hash* diferentes possuem propriedades diferentes quanto à frequência de colisões
- É possível salvarmos um balde, por exemplo, como uma lista associativa, isto é, uma lista de pares e valores
  - Neste caso, após encontrarmos um índice para uma chave, procuramos todos os itens salvos naquele balde pela chave
  - Caso o balde **esteja vazio ou contenha apenas um item**, a busca ocorre, como desejado, **em tempo constante**
  - Mas cuidado: a técnica conhecida como HashDoS tenta produzir muitas chaves com o mesmo *hash*, fazendo com que a busca seja **linear** no número de colisões!
- Idealmente, baldes devem ser implementados como árvores de busca binária, tornando o melhor tempo constante, e o **pior tempo logarítmico** em relação ao número de colisões

Tabela *hash*

- Baldes são, similar aos elos de uma lista linkada, sub-estruturas necessárias para o armazenamento
- Como há um tamanho limitado para índices dentro da tabela, há a possibilidade de **colisões**
  - Funções de *hash* diferentes possuem propriedades diferentes quanto à frequência de colisões
- É possível salvarmos um balde, por exemplo, como uma lista associativa, isto é, uma lista de pares e valores
  - Neste caso, após encontrarmos um índice para uma chave, procuramos todos os itens salvos naquele balde pela chave
  - Caso o balde **esteja vazio ou contenha apenas um item**, a busca ocorre, como desejado, **em tempo constante**
  - Mas cuidado: a técnica conhecida como HashDoS tenta produzir muitas chaves com o mesmo *hash*, fazendo com que a busca seja **linear** no número de colisões!
- Idealmente, baldes devem ser implementados como árvores de busca binária, tornando o melhor tempo constante, e o **pior tempo logarítmico** em relação ao número de colisões

Tabela *hash*

- Baldes são, similar aos elos de uma lista linkada, sub-estruturas necessárias para o armazenamento
- Como há um tamanho limitado para índices dentro da tabela, há a possibilidade de **colisões**
  - Funções de *hash* diferentes possuem propriedades diferentes quanto à frequência de colisões
- É possível salvarmos um balde, por exemplo, como uma lista associativa, isto é, uma lista de pares e valores
  - Neste caso, após encontrarmos um índice para uma chave, procuramos todos os itens salvos naquele balde pela chave
  - Caso o balde **esteja vazio ou contenha apenas um item**, a busca ocorre, como desejado, **em tempo constante**
  - Mas cuidado: a técnica conhecida como HashDoS tenta produzir muitas chaves com o mesmo *hash*, fazendo com que a busca seja **linear** no número de colisões!
- Idealmente, baldes devem ser implementados como árvores de busca binária, tornando o melhor tempo constante, e o **pior tempo logarítmico** em relação ao número de colisões

Tabela *hash*

- Baldes são, similar aos elos de uma lista linkada, sub-estruturas necessárias para o armazenamento
- Como há um tamanho limitado para índices dentro da tabela, há a possibilidade de **colisões**
  - Funções de *hash* diferentes possuem propriedades diferentes quanto à frequência de colisões
- É possível salvarmos um balde, por exemplo, como uma lista associativa, isto é, uma lista de pares e valores
  - Neste caso, após encontrarmos um índice para uma chave, procuramos todos os itens salvos naquele balde pela chave
  - Caso o balde **esteja vazio ou contenha apenas um item**, a busca ocorre, como desejado, **em tempo constante**
  - Mas cuidado: a técnica conhecida como HashDoS tenta produzir muitas chaves com o mesmo *hash*, fazendo com que a busca seja **linear** no número de colisões!
- Idealmente, baldes devem ser implementados como árvores de busca binária, tornando o melhor tempo constante, e o **pior tempo logarítmico** em relação ao número de colisões

Tabela *hash*