Notas de aula: Análise de Algoritmos
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC
Profa. Carla Negri Lintzmayer

Estruturas de dados básicas
(*Última atualização: 28 de janeiro de 2019*)

Referências e materiais complementares desse tópico

Website Estruturas de Dados, do prof. Paulo Feofiloff, da USP.

Livro Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C.. Introduction to Algorithms. 2nd ed. MIT Press. 2002.
Capítulo 6.1, 6.2, 6.3, 6.5, 10, 11, 12, 13, 22.1.

Sumário

1 Estruturas de dados

- Servem para organizar dados de modo que eles possam ser processados de forma mais eficiente.
- Algoritmos e estruturas de dados andam juntos.
- Tipo abstrato de dados (TAD): conjunto de dados + relação entre eles + operações que podem ser aplicadas.
  - Exemplo: o conjunto de números inteiros e as operações de soma, subtração, multiplicação e divisão.
- Pode ser implementado de várias maneiras: estruturas de dados.
  - Existem várias estruturas de dados para um mesmo tipo abstrato de dados.
- Segredo de muitos programas rápidos: uma boa estrutura de dados pode ter um efeito dramático sobre a velocidade do programa.

  - Veja o exemplo do problema de calcular o n-ésimo número de Fibonacci.

- Estruturas diferentes suportam operações diferentes em tempos diferentes.
  - Não existe uma estrutura que é melhor para todas as circunstâncias.

2 Vetores e listas ligadas

- São as estruturas de dados mais simples e clássicas.
- Formam a base para muitos dos algoritmos vistos em um curso de análise de algoritmos.
- Ambas armazenam um conjunto de elementos *em sequência*.

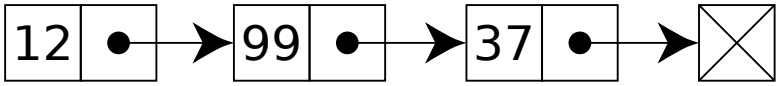
2.1 Vetores

- Coleção de elementos de um mesmo tipo que são referenciados por um identificador único.
  - Esses elementos ocupam posições *contíguas* na memória.
  - Isso permite acesso direto (em tempo constante – $\Theta(1)$) a qualquer dado por meio de um índice inteiro.
- Denotaremos um vetor A com capacidade para n elementos por $A[1..n]$ ou $A = (a_1, a_2, \ldots, a_n)$ e $A[i] = a_i$ é o elemento que está armazenado na posição i, para todo $1 \leq i \leq n$.
- Para quaisquer $1 \leq i < j \leq n$, denotamos por $A[i..j]$ o subvetor de A que contém os elementos $A[i], A[i+1], \ldots, A[j]$.
- Tempo das operações:
  - Busca: $O(n)$[1]
  - Inserção: $\Theta(1)$
  - Remoção: $O(n)$
  - Elemento mínimo/máximo: $\Theta(n)$
  - k-ésimo menor elemento: $\Theta(n \log n)$
  - Predecessor/sucessor de um elemento: $\Theta(n)$
- Tempo das operações se o vetor estiver ordenado:
  - Busca: $O(\log n)$
  - Inserção: $O(n)$
  - Remoção: $O(n)$
  - Elemento mínimo/máximo: $\Theta(1)$
  - k-ésimo menor elemento: $\Theta(1)$
  - Predecessor/sucessor de um elemento: $O(\log n)$
- Você pode ver implementações em C **aqui** e **aqui** ou em C, Java e Python **aqui**.

[1] Por que não podemos dizer que a busca é em tempo $\Theta(n)$?

2.2 Listas ligadas

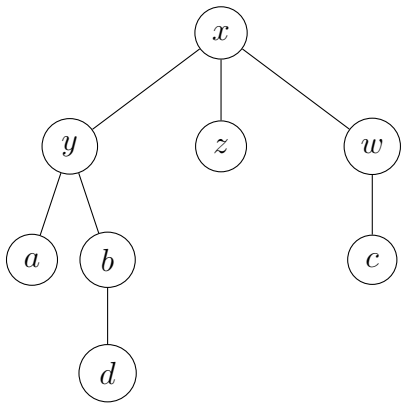
- Coleção de elementos de mesmo tipo.
  - Cada elemento é armazenado em um *nó*, que armazena também o *endereço* para o nó que armazena o próximo elemento.
  - Cada nó de uma lista pode estar em uma posição diferente da memória.
  - Têm-se acesso apenas ao primeiro nó da lista.
  - Listas não permitem acesso direto a um elemento: para acessar o k-ésimo elemento da lista, deve-se acessar o primeiro, que dá acesso ao segundo, que dá acesso ao terceiro, e assim sucessivamente, até que o $(k-1)$-ésimo elemento dá acesso ao k-ésimo.



- Tempo das operações:
  - Busca: $O(n)$
  - Inserção: $\Theta(1)$
  - Remoção: $O(n)$
  - Elemento mínimo/máximo: $\Theta(n)$
  - k-ésimo menor elemento: $\Theta(n \log n)$
  - Predecessor/sucessor de um elemento: $\Theta(n)$
- Você pode ver implementações em C **aqui** ou em C, Java e Python **aqui**.

3 Árvores

- Estrutura não linear de nós, onde cada nó contém o elemento armazenado e pode ter um ou mais ponteiros para outros nós.
  - De certa forma, são um conceito estendido de listas ligadas.
- Mais especificamente, são *estruturas hierárquicas* nas quais um nó aponta para os nós abaixo dele na hierarquia, chamados seus *nós filhos*.
- Um nó em especial é a *raiz*, que é o topo da hierarquia e está presente no *nível 0* da árvore.
- Nós filhos da raiz estão no nível 1; os nós filhos destes estão no nível 2; e assim por diante.
- Um nó sem filhos é chamado de *folha* da árvore.



nível 0

nível 1

nível 2

nível 3

- x é o nó raiz (nível 0)
- y, z e w são filhos de x
- y é pai de a e b
- a, d, z e c são folhas
- essa árvore tem 4 níveis
- ela tem altura 3

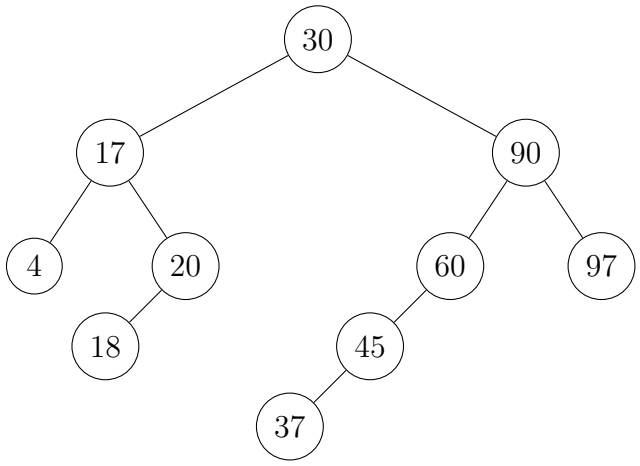
- Em uma árvore, só temos acesso direto ao nó raiz e qualquer manipulação, portanto, deve percorrer os ponteiros entre os nós.
- Note ainda que existe um único caminho entre a raiz e uma folha.
  - A distância do caminho raiz-folha mais longo, considerando todas as folhas, define a *altura* da árvore.
  - Equivalentemente, a altura de uma árvore é igual ao maior nível.
- Considerando apenas essas informações, vemos que qualquer busca deve ser feita percorrendo a árvore toda. Inserções/remoções não estão bem definidas também.
  - Essencialmente, não ganhamos muita coisa com relação a uma lista ligada.
- O tipo mais comum de árvore, e que define melhor as operações mencionadas, é a *árvore binária*.
- Árvores binárias são aquelas cujo maior número de filhos de qualquer nó é dois e, portanto, podemos distinguir os filhos entre direito e esquerdo.
- Árvores binárias também podem ser definidas recursivamente: ela é vazia ou é um nó raiz que é pai de uma árvore binária à direita e de outra árvore binária à esquerda.
  - Assim também dizemos que o filho direito (resp. esquerdo) do nó raiz é *raiz da subárvore direita* (resp. *esquerda*).
- Para facilitar a discussão, chamaremos de x o nó que contém o elemento x. Poderemos dizer então que um nó "é maior" do que outro nó (ou menor), ou mesmo falar sobre "o maior" (ou o menor) nó da árvore.

3.1 Árvores binárias de busca

- São árvore binárias especiais nas quais, para cada nó x, todos os nós da subárvore esquerda são menores do que x e todos os nós da subárvore direita são maiores do que x.
- Essa propriedade é usada justamente para guiar a operação de busca.
  - Se quisermos procurar um elemento k na árvore, primeiro o comparamos com a raiz: ou k é igual à raiz e a busca termina, ou k é menor do que a raiz e o problema se reduz a procurar k na subárvore esquerda, ou então k é maior do que a raiz e o problema se reduz a procurar k na subárvore direita.

  - Note que o pior caso de uma busca será percorrer um caminho raiz-folha inteiro, de forma que a busca pode levar tempo $O(h)$, onde h é a altura da árvore.
  - E agora temos uma potencial melhora com relação a listas ligadas: pode ser que a árvore tenha altura menor do que o número n de elementos armazenados nela.

- Outras operações que não alteram a estrutura de uma árvore de busca são:
  - Encontrar o menor elemento: basta seguir os filhos esquerdos a partir da raiz até chegar em um nó que não tem filho esquerdo – este contém o menor elemento da árvore. Tempo necessário: $O(h)$.
  - Encontrar o maior elemento: basta seguir os filhos direitos a partir da raiz até chegar em um nó que não tem filho direito – este contém o maior elemento da árvore. Tempo necessário: $O(h)$.
  - O sucessor de um elemento k: é o menor elemento que é maior do que k. Pela estrutura da árvore, se o nó k tem um filho direito, então o sucessor de k é o menor elemento dessa subárvore direita. Caso o nó k não tenha filho direito, então o primeiro nó maior do que k deve ser um ancestral seu: é o menor nó cujo filho esquerdo também é ancestral de k. Tempo necessário: $O(h)$
  - O predecessor de um elemento k: é o maior elemento da subárvore enraizada no filho esquerdo de x ou então é o maior ancestral cujo filho direito também é ancestral de k. Tempo necessário: $O(h)$



  - o sucessor de 30 é o 37 (menor nó da subárvore enraizada em 90)
  - o sucessor de 20 é o 30 (menor ancestral do 20 cujo filho esquerdo, o 17, também é ancestral do 20)

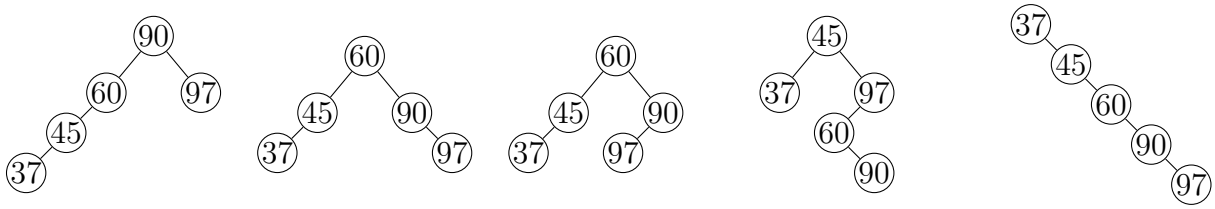
- Para inserir um novo nó x na árvore, podemos primeiro buscar por x na árvore.
  - Se x não estiver, então a busca terminou em um nó y que deverá ser o pai de x: se $x < y$ inserimos x à esquerda e caso contrário o inserimos à direita.
  - Note que qualquer busca posterior por x vai percorrer exatamente o mesmo caminho e chegar corretamente a x.
  - Portanto, essa inserção mantém a propriedade da árvore binária de busca.
  - Não é difícil perceber que o tempo de execução desse algoritmo também é $O(h)$.

```
1: função INSERE(raiz, x)
2:     se raiz é vazia então
3:         retorne x
4:     se x < raiz então
5:         raiz.esquerda ← INSERE(raiz.esquerda, x)
6:     senão se x > raiz então
```

7: $raiz.direita \leftarrow$ INSERE($raiz.direita$, x)

8: **devolve** $raiz$

- No caso de remoções, precisamos tomar alguns cuidados extras para garantir que a árvore continue sendo de busca.
  - Se o nó a ser removido é folha, então não há problemas e basta removê-lo.
  - Se o nó a ser removido tem um único filho, então temos um caso simples também e basta substituí-lo por esse filho.
  - Agora, se o nó x a ser removido tem dois filhos, precisamos substituí-lo por algum outro nó que tenha no máximo um filho e vá manter a propriedade de busca.
    - Um bom candidato para substituir x é seu sucessor: todos os nós à esquerda de x são menores do que o sucessor de x e todos os nós à direita são maiores.
    - Como o sucessor de x é o menor nó da subárvore direita de x (pois x tem dois filhos) e o menor nó de uma árvore tem no máximo um filho (à direita), podemos de fato trocar o nó sucessor com x e prosseguir removendo x, que passa a ter um único filho.
  - Note que o tempo de execução dessa operação depende basicamente da operação que encontra o sucessor de um nó (pois nos outros casos temos simples atualizações de ponteiros), de forma que ela também leva tempo $O(h)$.
- Tempo das operações dada uma árvore de altura h com n nós:
  - Busca: $O(h)$
  - Inserção: $O(h)$
  - Remoção: $O(h)$
  - Elemento mínimo/máximo: $O(h)$
  - k-ésimo menor elemento: $O(h)$
  - Predecessor/sucessor de um elemento: $O(h)$
- Note agora que a inserção é feita "de qualquer forma", apenas respeitando a propriedade de busca.
- Assim, a árvore gerada após n inserções pode ter qualquer formato.
  - Um mesmo conjunto de elementos, dependendo da ordem na qual são inseridos, pode dar origem a várias árvores diferentes.



- Todas as operações que mencionamos têm tempo $O(h)$ e, como vimos acima, uma árvore binária de busca pode ter altura $h = n$ e, portanto, ser tão ruim quanto uma lista ligada.
  - Uma forma de melhorar o tempo, portanto, é garantir que a altura da árvore não seja tão grande.

- Uma *árvore balanceada* garante que sua altura vai ser sempre pequena o suficiente mesmo depois de várias inserções e remoções.
  - No caso de árvores binárias, se ela tem altura h então existem no máximo $2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$ nós.
  - Se n é o total de nós, então $n \leq 2^{h+1} - 1$, o que implica que $h \geq \lfloor \log n + 2 \rfloor - 1$.
  - Ou seja, a menor altura de qualquer árvore binária com n nós é $O(\log n)$.
- *Árvore AVL* é uma árvore binária de busca balanceada que mantém a seguinte propriedade: a diferença entre as alturas da subárvore esquerda e direita de qualquer nó é no máximo 1.
  - Isso garante que a altura h de qualquer árvore AVL é sempre $O(\log n)$.
- *Árvore red-black* é uma árvore binária de busca balanceada que também tem algura $O(\log n)$.
- *Árvore-B* é uma árvore de busca balanceada mas que não é binária: cada nó tem m elementos e $m + 1$ filhos. Ela garante altura $O(\log n)$ também.
- Vídeo aulas do prof. Tim Roughgarden (em inglês, com legendas): **13.1**, **13.2**, **13.3** e **13.5**. Sobre árvores red-black: **13.4** e **13.6**.
- Você pode ver implementações em C **aqui** ou em C, Java e Python **aqui**.

4 Fila (TAD)

- Coleção de elementos cuja operação de remoção é especial:
  - o elemento removido deve ser o que está na estrutura há mais tempo.
- Em outras palavras: o primeiro elemento a entrar é o primeiro a sair (*first in, first out* – FIFO).
- Útil para manter a ordem de documentos para impressão ou o buffer do teclado.
- Adicionar (*enqueue*) e remover (*dequeue*) itens são as operações mais utilizadas.
- Tempo das operações (independente da implementação por vetor ou por lista ligada):
  - Inserção (*enqueue*): $\Theta(1)$
  - Remoção (*dequeue*): $\Theta(1)$
- Você pode ver implementações em C **aqui** ou em C, Java e Python **aqui**.

5 Pilha (TAD)

- Coleção de elementos cuja operação de remoção é especial:
  - o elemento removido deve ser o que está na estrutura há menos tempo.
- Em outras palavras: o último elemento a entrar é o primeiro a sair (*last in, first out* – LIFO).
- Útil para manter balanceamento de símbolos, operações de desfazer e refazer em aplicativos, botões "voltar" em *browsers* ou cálculo de expressões aritméticas.
- Adicionar (*push*) e remover (*pop*) itens são as operações mais utilizadas.
- Tempo das operações (independente da implementação por vetor ou por lista ligada):
  - Inserção (*enqueue*): $\Theta(1)$
  - Remoção (*dequeue*): $\Theta(1)$
- Você pode ver implementações em C **aqui** ou em C, Java e Python **aqui**.

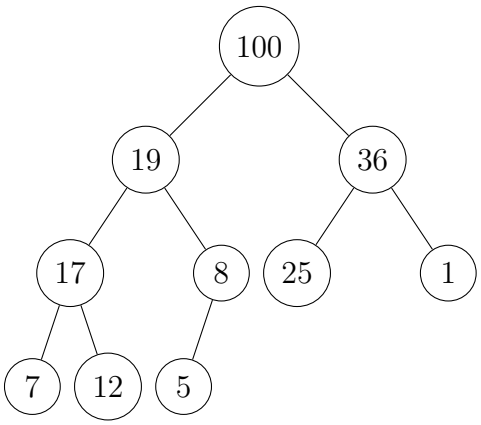
6 Filas de prioridade (TAD)

- Coleção de elementos que possuem prioridades associadas e a operação de remoção é especial:
  - o elemento removido deve ser o que possui maior prioridade[2].
- Útil para escalonamento de tarefas pelo processador, algoritmos clássicos como Dijkstra, Prim e Huffman.
- Adicionar um elemento, acessar o elemento de maior prioridade, remover o elemento de maior prioridade e alterar a prioridade de um elemento são as operações mais utilizadas.
- Mais comumente implementada por meio de heaps (veja Seção 6.1).
- Tempo das operações com implementação em vetor ou lista ligada ordenados por prioridade:
  - Inserção: $O(n)$
  - Acesso ao elemento de maior prioridade: $\Theta(1)$
  - Remoção (do elemento de maior prioridade): $\Theta(1)$
  - Alterar a prioridade de um elemento: $O(n)$
- Tempo das operações com implementação em heap:
  - Inserção: $O(\log n)$
  - Acesso ao elemento de maior prioridade: $\Theta(1)$
  - Remoção (do elemento de maior prioridade): $O(\log n)$
  - Alterar a prioridade de um elemento: $O(\log n)$

[2]Ter maior prioridade não significa necessariamente que o valor indicativo da prioridade é o maior. Manteremos a palavra prioridade para deixar o texto genérico.

6.1 Heap

- São estruturas de dados que conceitualmente podem ser vistas como uma árvore binária quase completa: todos os níveis estão cheios, exceto talvez pelo último, que é preenchido da esquerda para a direita.
- Em uma heap, a seguinte propriedade deve ser sempre mantida:
  - um nó tem prioridade maior do que a prioridade de seus dois filhos (se eles existirem).
- Isso garante que o elemento de maior prioridade armazenado está na raiz: acesso direto.
- Uma heap de tamanho n é geralmente representada por um vetor de n posições: o elemento na posição i tem filho esquerdo na posição $2i$ (se $2i \leq n$), seu filho direito na posição $2i+1$ (se $2i+1 \leq n$), e seu pai na posição $\lfloor i/2 \rfloor$ (se $i > 1$).
  - Assim, percorrendo o vetor da esquerda para a direita, vemos todos os elementos do nível x consecutivamente antes dos elementos do nível $x+1$.
  - Um nó na posição i tem altura $\lfloor \log(n/i) \rfloor$ e está no nível $\lfloor \log i \rfloor$.

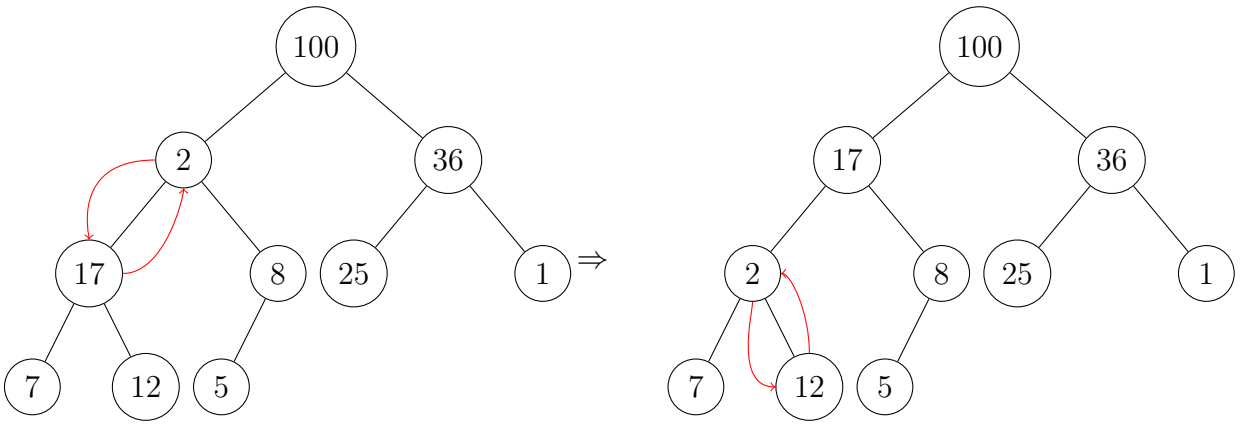


$$A = (\underbrace{100}_{\text{nível 0}}, \underbrace{19,\ 36}_{\text{nível 1}}, \underbrace{17,\ 8,\ 25,\ 1}_{\text{nível 2}}, \underbrace{2,\ 7,\ 5}_{\text{nível 3}})$$

- Uma vez que temos uma heap, a operação de remover o elemento de maior prioridade é feita da seguinte forma: trocamos o primeiro elemento, que é o de maior prioridade, (raiz) com o último (que é o nó mais à direita do último nível) e restauramos a propriedade de heap fazendo trocas entre pais e filhos, começando da raiz e descendo pelo caminho onde as trocas vão sendo feitas, podendo chegar a uma folha.

```
1: função REMOVE(A, n)
2:     troque A[1] com A[n]
3:     CORRIGE-PARA-BAIXO(A, 1, n − 1)
4:     n ← n − 1
```

  - Essas trocas sucessivas são feitas pelo algoritmo CORRIGE-PARA-BAIXO, cujo pseudocódigo é mostrado a seguir. O algoritmo recebe um índice i e o vetor A tal que as subárvores enraizadas em $2i$ e $2i+1$ são heaps. Seu objetivo é fazer com que a subárvore enraizada em i seja heap também.



1: **função** CORRIGE-PARA-BAIXO(A, i, n)
2: $maior \leftarrow i$
3: **se** $2i \leq n$ e a prioridade de $A[2i]$ é maior do que a de $A[maior]$ **então**
4: $maior \leftarrow 2i$
5: **senão se** $2i + 1 \leq n$ e a prioridade de $A[2i + 1]$ é maior do que a de $A[maior]$ **então**
6: $maior \leftarrow 2i + 1$
7: **se** $maior \neq i$ **então**
8: troque $A[i]$ com $A[maior]$
9: CORRIGE-PARA-BAIXO(A, $maior$, n)

- O algoritmo CORRIGE-PARA-BAIXO está certo?
  - Podemos provar que sim por indução na altura do i-ésimo elemento. Nota: a altura do elemento $A[i]$ é $\lfloor \log(n/i) \rfloor$, número que denotaremos por h_i.
  - Caso base: $h_i = 0$. Nesse caso, $A[i]$ deve ser uma folha e podemos ver que o algoritmo não faz nada. De fato, folhas já são heaps.
  - Hipótese: o algoritmo CORRIGE-PARA-BAIXO corretamente faz com que a árvore enraizada em $A[x]$, de altura $h_x < h_i$, seja uma heap, considerando que inicialmente as árvores enraizadas em $A[2x]$ e $A[2x + 1]$ são heaps, se elas existirem.
  - Passo: temos o elemento $A[i]$ de altura h_i e sabemos que $A[2i]$ e $A[2i + 1]$ são raízes de heaps, se elas existirem.
    - Primeiro o algoritmo encontra $maior$, que é o índica do elemento de maior prioridade dentre $\{A[i], A[2i]$ se $2i \leq n, A[2i + 1]$ se $2i + 1 \leq n\}$.
    - Se $maior = i$, então $A[i]$ é maior que seus filhos e portanto a árvore enraizada em $A[i]$ já é heap. O algoritmo de fato não faz nada nesse caso e portanto termina corretamente.
    - Se $maior \neq i$, suponha que $maior = 2i$, isto é $A[2i]$ tem prioridade maior do que $A[i]$ e do que $A[2i + 1]$ (se $2i + 1 \leq n$). O algoritmo então troca $A[i]$ com $A[2i]$ e faz uma chamada recursiva para corrigir a posição $2i$. Agora, os filhos do novo elemento $A[i]$ têm prioridade menor do que ele e, como $h_{maior} < h_i$, por hipótese de indução sabemos que a árvore enraizada em $2i$ será corrigida.
    - O caso é similar se $maior = 2i + 1$.

- Qual o tempo de execução do CORRIGE-PARA-BAIXO?
  - No pior caso, vamos acessar todos os nós que fazem parte de um caminho que vai de i até uma folha.

  - Logo, o algoritmo tem tempo proporcional à altura do nó i na árvore, isto é, $O(\log(n/i))$.
  - Disso vemos que o tempo do REMOVE é $O(\log n)$.

- Já a operação de inserção de um novo elemento é feita da seguinte forma: insira-o à direita do último elemento (o mais à esquerda possível no último nível) e restaure a propriedade de heap fazendo trocas entre filhos e pais, começando da posição da inserção e subindo pelo caminho onde as trocas vão sendo feitas, podendo chegar à raiz.

```
1: função INSERE(A, k, n)
2:     A[n + 1] ← k
3:     CORRIGE-PARA-CIMA(A, n + 1)
4:     n ← n + 1
```

  - Essas trocas sucessivas são feitas pelo algoritmo CORRIGE-PARA-CIMA, cujo pseudocódigo é mostrado a seguir. O algoritmo recebe um índice i e o vetor A tal que a subárvore que tem os elementos até a posição $i - 1$ é uma heap. Seu objetivo é fazer com que a subárvore que vai até i seja heap também.

```
1: função CORRIGE-PARA-CIMA(A, i)
2:     se i ≥ 2 e a prioridade de A[i] é maior do que a de A[⌊i/2⌋] então
3:         troque A[i] com A[⌊i/2⌋]
4:         CORRIGE-PARA-CIMA(A, ⌊i/2⌋)
```

- O algoritmo CORRIGE-PARA-CIMA está certo?
  - Sim. EXERCÍCIO: por indução no nível do i-ésimo elemento.
- Qual o tempo de execução do CORRIGE-PARA-CIMA?
  - No pior caso, vamos acessar todos os nós que fazem parte de um caminho que vai de i até a raiz.
  - Logo, o algoritmo tem tempo proporcional ao nível do nó i na árvore, isto é, $O(\log i)$.
  - Disso vemos que o tempo do INSERE é $O(\log n)$.
- Outra operação importante é a de alterar a prioridade de algum elemento que já está na heap.
  - Ele provavelmente vai "estragar" a heap, mas isso depende do novo valor que ele assumiu.
  - Se o elemento está na posição i, atualize $A[i]$ e use CORRIGE-PARA-CIMA caso a prioridade tenha aumentado ou então CORRIGE-PARA-BAIXO caso a prioridade tenha diminuído.
  - Assim, o tempo é $O(\log n)$.
- Uma última questão importante é: como construir uma heap a partir de um conjunto de elementos?
  - Seja $A[1..n]$ um vetor qualquer. Podemos visualizá-lo como a estrutura de árvore.
  - Note que os elementos de $A[\lfloor n/2 \rfloor + 1..n]$ são folhas e, portanto, são heaps triviais.

  - A árvore enraizada em $A[\lfloor n/2 \rfloor]$ (provavelmente) não é heap, mas seus filhos com certeza são (pois são folhas), então podemos usar Corrige-Para-Baixo a partir desse elemento.
  - Podemos então repetir esse procedimento com todo elemento da posição $\lfloor n/2 \rfloor$ até a posição 1: toda vez que aplicarmos Corrige-Para-Baixo sobre um elemento, já temos a garantia de que seus dois filhos são heap, o que é justamente o que o algoritmo exige.

```
1: função Constroi-Heap(A, n)
2:     para i ← ⌊n/2⌋ até 1 faça
3:         Corrige-Para-Baixo(A, i, n)
```

- Qual o tempo do Constroi-Heap?
  - Fazendo uma análise rápida: são $\Theta(n)$ chamadas ao Corrige-Para-Baixo, que podemos dizer que executa em tempo $O(\log n)$, e, portanto, temos tempo total $O(n \log n)$.
  - Essa análise não está errada, mas é possível fazer uma análise mais esperta e melhorar essa garantia de tempo.
  - Note que o algoritmo começa no penúltimo nível, digamos $p-1$, da árvore, corrige todos os seus 2^{p-1} nós e vai para o nível $p-2$, corrigindo seus 2^{p-2} nós e indo para o nível $p-3$, e assim por diante até o nível 0.
  - O tempo do Corrige-Para-Baixo é proporcional à altura do nó no qual ele é chamado (o limite $O(\log n)$ anterior era, portanto, um chute muito alto): no nível $p-1$, o tempo é c, no nível $p-2$ é $2c$, e assim por diante, para alguma constante c.
  - Então o tempo total $T(n)$ do Constroi-Heap é

    $$T(n) \leq c \times 2^{p-1} + 2c \times 2^{p-2} + 3c \times 2^{p-3} + \cdots + (p-1)c \times 2^1 + pc \times 2^0 < 2 \times 2^p \times c \ .$$

  - Como $p \leq \log n$, $T(n) = O(n)$.
- Veja uma implementação em C **aqui** ou em C e Python **aqui**.
- Vídeo aulas do prof. Tim Roughgarden (em inglês, com legendas): **12.2** e **12.3**.

7 Tabelas Hash

- Suponha que queremos projetar um sistema que armazena dados de funcionários usando como chave seus CPFs.
  - Basicamente esse sistema vai precisar fazer inserções, remoções e buscas (todas dependentes do CPF dos funcionários).
- Note que podemos usar um vetor ou lista ligada para isso: neste caso a busca é feita em tempo linear, o que pode ser custoso na prática se o número n de funcionários armazenados for muito grande.
  - Se usarmos um vetor ordenado, a busca pode ser melhorada para ter tempo $O(\log n)$, mas inserções e remoções passam a ser custosas.

- Uma outra opção é usar uma árvore binária de busca balanceada, que garante tempo $O(\log n)$ em qualquer uma das três operações.
- Uma terceira solução é criar um vetor grande o suficiente para que ele seja indexado pelos CPFs.
  - Essa estratégia, chamada *endereçamento direto*, é ótima pois garante que as três operações serão executadas em tempo $\Theta(1)$.
- Acontece que um CPF tem 11 dígitos, sendo 9 válidos e 2 de verificação, de forma que podemos ter 9^{10} possíveis números diferentes (algo na casa dos bilhões). Logo, endereçamento direto não é viável.
- Por outro lado, a empresa precisa armazenar a informação de n funcionários apenas, o que é um valor bem menor. Temos ainda uma quarta opção: tabelas hash.
- Uma *tabela hash* é uma estrutura de dados que basicamente mapeia chaves a elementos.
  - Ela implementa eficientemente – em tempo médio $O(1)$ – as operações de busca, inserção e remoção.
- Ela usa uma *função hash*, que recebe como entrada uma chave (um CPF, no exemplo acima) e devolve um número pequeno (entre 1 e m), que serve como índice da tabela que vai armazenar os elementos de fato (que tem tamanho m).
  - Assim, se h é uma função hash, um elemento de chave k vai ser armazenado (falando de forma bem geral) na posição $h(k)$.
- Note, no entanto, que sendo o universo U de chaves grande (tamanho M) e o tamanho m da tabela bem menor do que M, não importa como seja a função h: várias chaves serão mapeadas para a mesma posição – o que é chamado de *colisão*.
  - Aliás vale mencionar que mesmo se o contrário fosse verdade ainda teríamos colisões: por exemplo, se 2450 chaves forem mapeadas pela função hash para uma tabela de tamanho 1 milhão, mesmo com uma distribuição aleatória perfeitamente uniforme, de acordo com o **problema do aniversário**, existe uma chance de aproximadamente 95% de que pelo menos duas chaves serão mapeadas para a mesma posição.
- Temos então que lidar com dois problemas quando se fala em tabelas hash:
  - escolher uma função hash que minimize o número de colisões, e
  - lidar com as colisões, que são inevitáveis.
- Tempo das operações (numa boa implementação) considerando o caso médio:
  - Busca: $O(1)$
  - Inserção: $O(1)$
  - Remoção: $O(1)$
- Veja mais sobre tabelas hash **aqui** ou nas vídeo aulas do prof. Tim Roughgarden (em inglês, com legendas): **14.1**, **14.2**, **14.3**, **15.1**, **15.2**, **15.3** e **15.4**.

8 Union-Find

- Estrutura de dados que mantém uma partição de um conjunto de objetos.
  - Dizemos que $A_1, A_2, \ldots, A_m$ é uma *partição* de um conjunto B se para cada A_i temos que $A_i \subseteq B$, $A_i \cap A_j = \emptyset$ para todo $i \neq j$ e $A_1 \cup \cdots A_m = B$.
- Essa estrutura fornece duas operações:
  - FIND(x): retorna o identificador do grupo ao qual o objeto x pertence.
  - UNION(g_1, g_2): funde dois grupos g_1 e g_2 em um único grupo.
- O identificador de um grupo é um elemento pertencente ao grupo, chamado de representante do grupo.
- É possível implementar essa estrutura de forma a garantir o seguinte tempo para as operações:
  - Find: $O(\alpha(n))$[3]
  - Union: $O(\alpha(n))$
- Vídeo aulas do prof. Tim Roughgarden (em inglês, com legendas): **1**, **2** e **3**.

8.1 Uma possível implementação

- Sejam $x_1, x_2, \ldots, x_n$ objetos quaisquer.
- Sejam $REPR[1..n]$ e $TAM[1..n]$ dois vetores tais que $REPR[i]$ armazena o representante do grupo onde x_i está e $TAM[i]$ armazena o tamanho do grupo onde x_i está.
  - Assim, uma forma de implementar a função FIND(x_i) é simplesmente retornar $REPR[i]$: tempo $\Theta(1)$.
- Inicialmente, cada objeto está em um grupo sozinho:
  - $REPR[i] = x_i$ e $TAM[i] = 1$ para todo $i \in 1..n$.
- Quando dois grupos se fundem, fazemos com que o grupo de menor tamanho passe a ter o mesmo representante que o grupo de maior tamanho: acessamos os objetos do grupo de menor tamanho[4] e atualizamos a devida entrada em $REPR$ e TAM.

[3] $\alpha(n)$ é conhecida como a **inversa da função Ackermann**.
[4] Como fazer isso de maneira eficiente?