# MC-202 — Unidade 12
# Filas de Prioridade e HeapSort

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

1º semestre/2017

# Voltando ao SelectionSort

Versão do SelectionSort que

# Voltando ao SelectionSort

Versão do SelectionSort que

- coloca o elemento máximo na posição `v[r]`

# Voltando ao SelectionSort

Versão do SelectionSort que

- coloca o elemento máximo na posição `v[r]`
- coloca o segundo maior elemento na posição `v[r-1]`

# Voltando ao SelectionSort

Versão do SelectionSort que

- coloca o elemento máximo na posição `v[r]`
- coloca o segundo maior elemento na posição `v[r-1]`
- etc...

# Voltando ao SelectionSort

Versão do SelectionSort que

- coloca o elemento máximo na posição `v[r]`
- coloca o segundo maior elemento na posição `v[r-1]`
- etc...

```
1  int selection_invertido(int *v, int l, int r) {
2    int i, j, max;
3    for (i = r; i > l; i--) {
4      max = i;
5      for (j = i-1; j >= l; j--)
6        if (v[j] > v[max])
7          max = j;
8      troca(&v[i],&v[max]);
9    }
10 }
```

# Rescrevendo...

Usamos uma função que acha o elemento máximo do vetor

```
1 int extrai_maximo(int *v, int l, int r) {
2   max = r;
3   for (j = r-1; j >= l; j--)
4     if (v[j] > v[max])
5       max = j;
6   return max;
7 }
```

# Rescrevendo...

Usamos uma função que acha o elemento máximo do vetor

```
1 int extrai_maximo(int *v, int l, int r) {
2   max = r;
3   for (j = r-1; j >= l; j--)
4     if (v[j] > v[max])
5       max = j;
6   return max;
7 }
```

E reescrevemos o SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

# Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

# Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

O tempo do `selection_invertido_v2` é:

# Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

O tempo do `selection_invertido_v2` é:

- o tempo de chamar $r - l - 1$ vezes `extrai_maximo(l, i)`

# Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

O tempo do `selection_invertido_v2` é:

- o tempo de chamar $r - l - 1$ vezes `extrai_maximo(l, i)`
- com `i` variando de `r` a `l+1`

# Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

O tempo do `selection_invertido_v2` é:

- o tempo de chamar $r - l - 1$ vezes `extrai_maximo(l, i)`
- com `i` variando de `r` a `l+1`

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

# Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

O tempo do `selection_invertido_v2` é:

- o tempo de chamar $r - l - 1$ vezes `extrai_maximo(l, i)`
- com `i` variando de `r` a `l+1`

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

# Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

O tempo do `selection_invertido_v2` é:

- o tempo de chamar $r - l - 1$ vezes `extrai_maximo(l, i)`
- com `i` variando de `r` a `l+1`

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) =$$

# Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

O tempo do `selection_invertido_v2` é:

- o tempo de chamar $r - l - 1$ vezes `extrai_maximo(l, i)`
- com `i` variando de `r` a `l+1`

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) =$$

# Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

O tempo do `selection_invertido_v2` é:

- o tempo de chamar $r - l - 1$ vezes `extrai_maximo(l, i)`
- com `i` variando de `r` a `l+1`

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot k =$$

# Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

O tempo do `selection_invertido_v2` é:

- o tempo de chamar $r - l - 1$ vezes `extrai_maximo(l, i)`
- com `i` variando de `r` a `l+1`

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot k = \frac{(n+2)(n-1)}{2} =$$

# Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int l, int r) {
2   int i, j, max;
3   for (i = r; i > l; i--) {
4     max = extrai_maximo(l, i);
5     troca(&v[i],&v[max]);
6   }
7 }
```

O tempo do `selection_invertido_v2` é:

- o tempo de chamar $r - l - 1$ vezes `extrai_maximo(l, i)`
- com `i` variando de `r` a `l+1`

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot k = \frac{(n+2)(n-1)}{2} = \mathrm{O}(n^2)$$

# Dá para fazer melhor?

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot k = \frac{(n+2)(n-1)}{2} = \mathrm{O}(n^2)$$

# Dá para fazer melhor?

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot k = \frac{(n+2)(n-1)}{2} = \mathrm{O}(n^2)$$

E se formos capazes de extrair o máximo em tempo $\mathrm{O}(\lg k)$?

# Dá para fazer melhor?

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot k = \frac{(n+2)(n-1)}{2} = \mathrm{O}(n^2)$$

E se formos capazes de extrair o máximo em tempo $\mathrm{O}(\lg k)$?

Tal algoritmo levaria tempo:

# Dá para fazer melhor?

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot k = \frac{(n+2)(n-1)}{2} = \mathrm{O}(n^2)$$

E se formos capazes de extrair o máximo em tempo $\mathrm{O}(\lg k)$?

Tal algoritmo levaria tempo:

$$\sum_{k=2}^{n} T(k) =$$

# Dá para fazer melhor?

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot k = \frac{(n+2)(n-1)}{2} = \mathrm{O}(n^2)$$

E se formos capazes de extrair o máximo em tempo $\mathrm{O}(\lg k)$?

Tal algoritmo levaria tempo:

$$\sum_{k=2}^{n} T(k) =$$

# Dá para fazer melhor?

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot k = \frac{(n+2)(n-1)}{2} = \mathrm{O}(n^2)$$

E se formos capazes de extrair o máximo em tempo $\mathrm{O}(\lg k)$?

Tal algoritmo levaria tempo:

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot \lg k \leq (n-2) \lg n =$$

# Dá para fazer melhor?

$T(k)$: tempo de extrair o máximo de um vetor com $k$ elementos

Para ordenar $n$ elementos, o SelectionSort gasta tempo

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot k = \frac{(n+2)(n-1)}{2} = \mathrm{O}(n^2)$$

E se formos capazes de extrair o máximo em tempo $\mathrm{O}(\lg k)$?

Tal algoritmo levaria tempo:

$$\sum_{k=2}^{n} T(k) = \sum_{k=2}^{n} c \cdot \lg k \leq (n-2) \lg n = \mathrm{O}(n \lg n)$$

Veremos esse algoritmo em breve...

# Fila de Prioridade

Uma fila de prioridades é uma estrutura de dados com duas operações básicas:

# Fila de Prioridade

Uma fila de prioridades é uma estrutura de dados com duas operações básicas:

- Inserir um novo elemento

# Fila de Prioridade

Uma fila de prioridades é uma estrutura de dados com duas operações básicas:

- Inserir um novo elemento
- Remover o elemento com maior *chave* (prioridade)

# Fila de Prioridade

Uma fila de prioridades é uma estrutura de dados com duas operações básicas:

- Inserir um novo elemento
- Remover o elemento com maior *chave* (prioridade)

Uma pilha é como uma fila de prioridades:

# Fila de Prioridade

Uma fila de prioridades é uma estrutura de dados com duas operações básicas:

- Inserir um novo elemento
- Remover o elemento com maior *chave* (prioridade)

Uma pilha é como uma fila de prioridades:

- o elemento com maior chave é sempre o último inserido

# Fila de Prioridade

Uma fila de prioridades é uma estrutura de dados com duas operações básicas:

- Inserir um novo elemento
- Remover o elemento com maior *chave* (prioridade)

Uma pilha é como uma fila de prioridades:

- o elemento com maior chave é sempre o último inserido

Uma fila é como uma fila de prioridades:

# Fila de Prioridade

Uma fila de prioridades é uma estrutura de dados com duas operações básicas:

- Inserir um novo elemento
- Remover o elemento com maior *chave* (prioridade)

Uma pilha é como uma fila de prioridades:

- o elemento com maior chave é sempre o último inserido

Uma fila é como uma fila de prioridades:

- o elemento com maior chave é sempre o primeiro inserido

# Fila de Prioridade

Uma fila de prioridades é uma estrutura de dados com duas operações básicas:

- Inserir um novo elemento
- Remover o elemento com maior *chave* (prioridade)

Uma pilha é como uma fila de prioridades:

- o elemento com maior chave é sempre o último inserido

Uma fila é como uma fila de prioridades:

- o elemento com maior chave é sempre o primeiro inserido

Primeira implementação: armazenar elementos em um vetor

# Fila de Prioridade (usando vetores) - TAD

```
1 #ifndef FILA_PRIORIDADE_VETOR_H
2 #define FILA_PRIORIDADE_VETOR_H
3
4 typedef struct {
5   char nome[20];
6   int chave;
7 } Item;
```

# Fila de Prioridade (usando vetores) - TAD

```
 1 #ifndef FILA_PRIORIDADE_VETOR_H
 2 #define FILA_PRIORIDADE_VETOR_H
 3
 4 typedef struct {
 5   char nome[20];
 6   int chave;
 7 } Item;
 8
 9 typedef struct {
10   Item *v;
11   int n, tamanho;
12 } FilaP;
```

# Fila de Prioridade (usando vetores) - TAD

```
 1 #ifndef FILA_PRIORIDADE_VETOR_H
 2 #define FILA_PRIORIDADE_VETOR_H
 3
 4 typedef struct {
 5   char nome[20];
 6   int chave;
 7 } Item;
 8
 9 typedef struct {
10   Item *v;
11   int n, tamanho;
12 } FilaP;
13
14 void inicializa(FilaP *fp, int tamanho);
15 void insere(FilaP *fp, Item item);
16 Item extrai_maximo(FilaP *fp);
17 int vazia(FilaP fp);
18 int cheia(FilaP fp);
19
20 #endif
```

# Operações Básicas

```
1 void inicializa(FilaP *fp, int tamanho) {
2   fp->v = malloc(tamanho*sizeof(Item));
3   fp->n = 0;
4   fp->tamanho = tamanho;
5 }
```

# Operações Básicas

```
1 void inicializa(FilaP *fp, int tamanho) {
2   fp->v = malloc(tamanho*sizeof(Item));
3   fp->n = 0;
4   fp->tamanho = tamanho;
5 }
```

```
1 void insere(FilaP *fp, Item item) {
2   fp->v[fp->n] = item;
3   (fp->n)++;
4 }
```

# Operações Básicas

```
1 void inicializa(FilaP *fp, int tamanho) {
2   fp->v = malloc(tamanho*sizeof(Item));
3   fp->n = 0;
4   fp->tamanho = tamanho;
5 }
```

```
1 void insere(FilaP *fp, Item item) {
2   fp->v[fp->n] = item;
3   (fp->n)++;
4 }
```

```
1 Item extrai_maximo(FilaP *fp) {
2   int j, max = 0;
3   for (j = 1; j < fp->n; j++)
4     if (fp->v[max].chave < fp->v[j].chave)
5       max = j;
6   troca(&(fp->v[max]), &(fp->v[fp->n-1]));
7   (fp->n)--;
8   return fp->v[fp->n];
9 }
```

# Operações Básicas

```
1 void inicializa(FilaP *fp, int tamanho) {
2   fp->v = malloc(tamanho*sizeof(Item));
3   fp->n = 0;
4   fp->tamanho = tamanho;
5 }
```

```
1 void insere(FilaP *fp, Item item) {
2   fp->v[fp->n] = item;
3   (fp->n)++;
4 }
```

```
1 Item extrai_maximo(FilaP *fp) {
2   int j, max = 0;
3   for (j = 1; j < fp->n; j++)
4     if (fp->v[max].chave < fp->v[j].chave)
5       max = j;
6   troca(&(fp->v[max]), &(fp->v[fp->n-1]));
7   (fp->n)--;
8   return fp->v[fp->n];
9 }
```

Insere em $\mathrm{O}(1)$, extrai o máximo em $\mathrm{O}(n)$

# Operações Básicas

```
1 void inicializa(FilaP *fp, int tamanho) {
2   fp->v = malloc(tamanho*sizeof(Item));
3   fp->n = 0;
4   fp->tamanho = tamanho;
5 }
```

```
1 void insere(FilaP *fp, Item item) {
2   fp->v[fp->n] = item;
3   (fp->n)++;
4 }
```

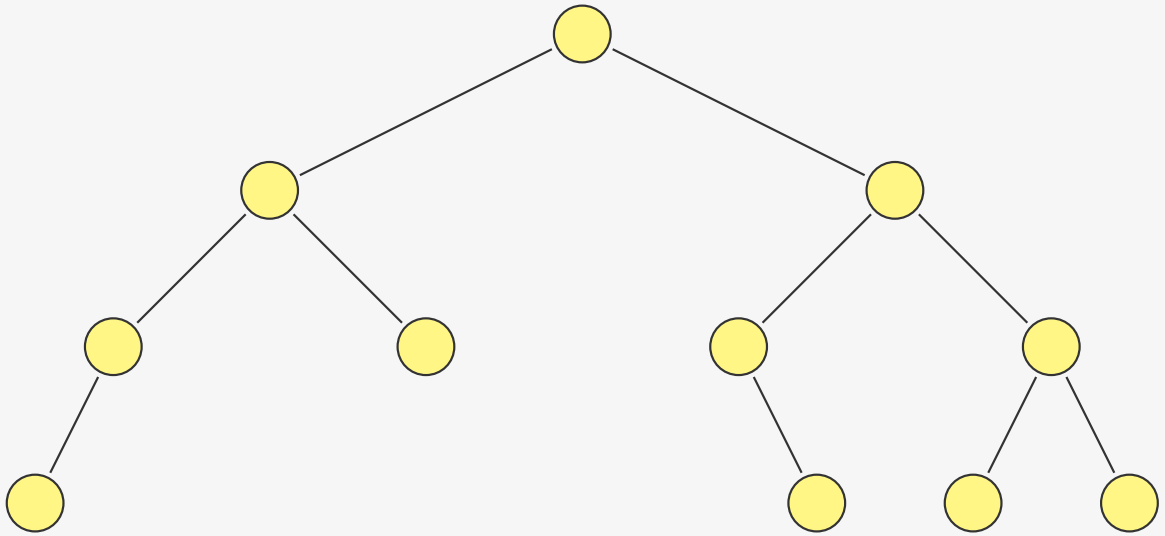
```
1 Item extrai_maximo(FilaP *fp) {
2   int j, max = 0;
3   for (j = 1; j < fp->n; j++)
4     if (fp->v[max].chave < fp->v[j].chave)
5       max = j;
6   troca(&(fp->v[max]), &(fp->v[fp->n-1]));
7   (fp->n)--;
8   return fp->v[fp->n];
9 }
```

Insere em $\mathrm{O}(1)$, extrai o máximo em $\mathrm{O}(n)$

- Se mantiver o vetor ordenado, os tempos se invertem

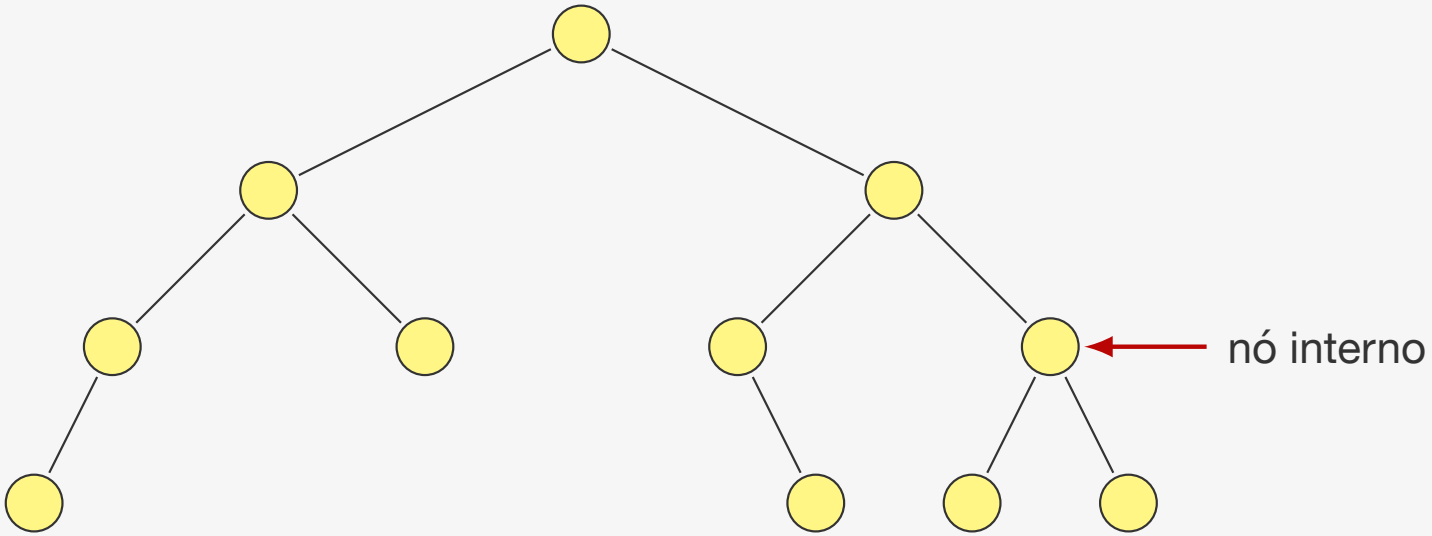
# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



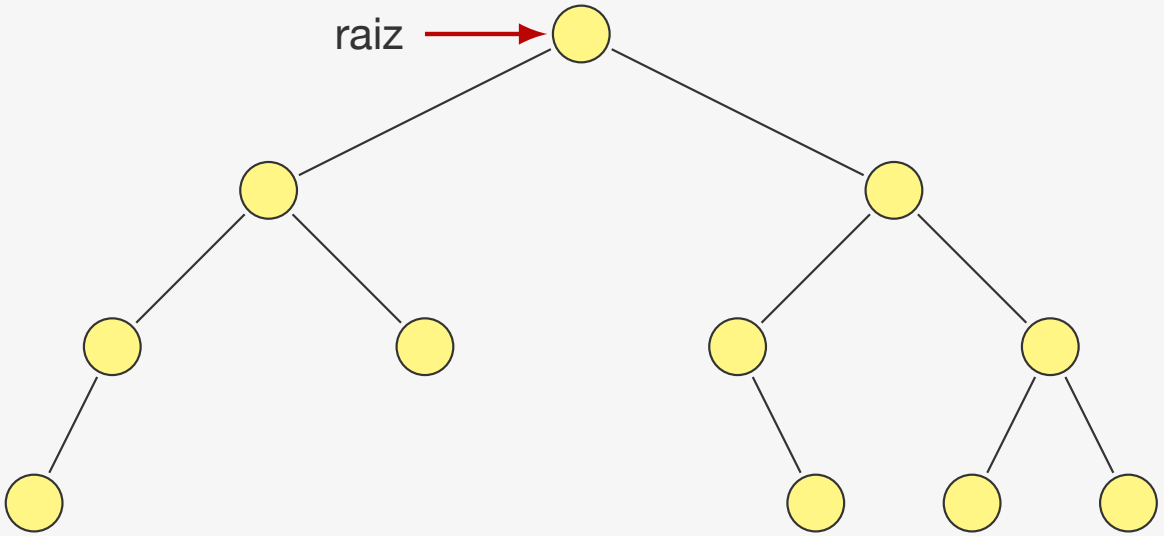
# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



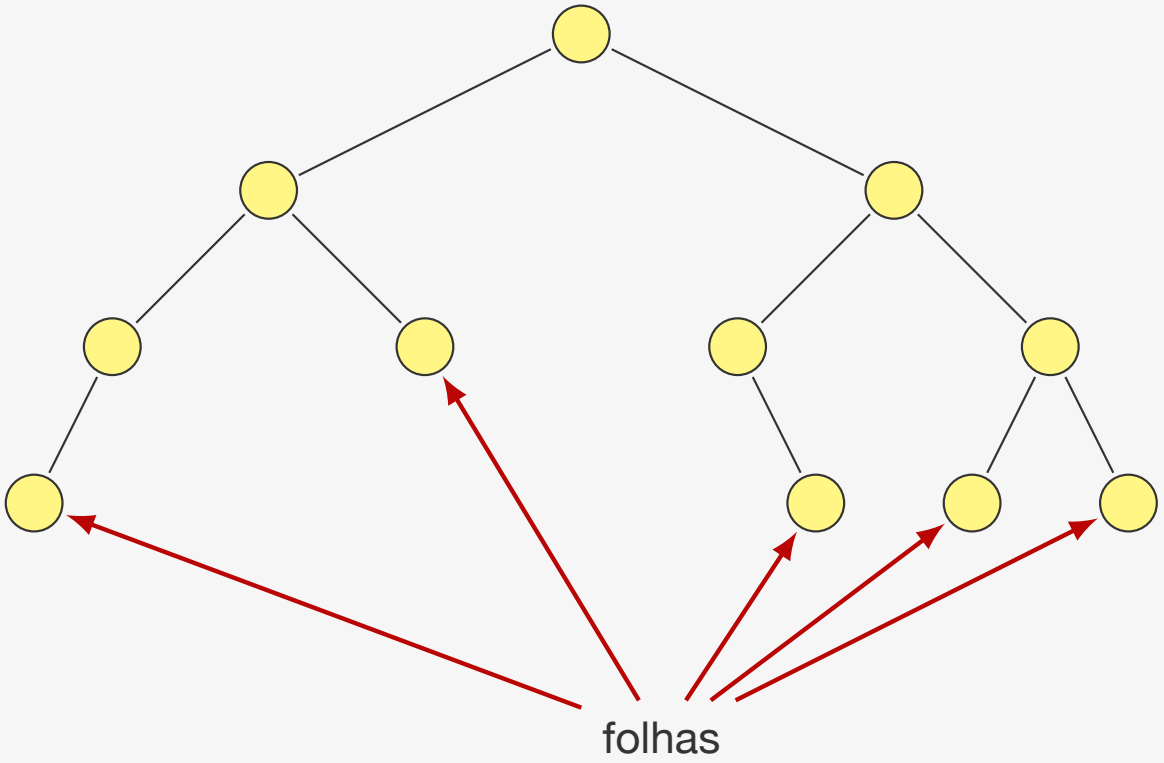
# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



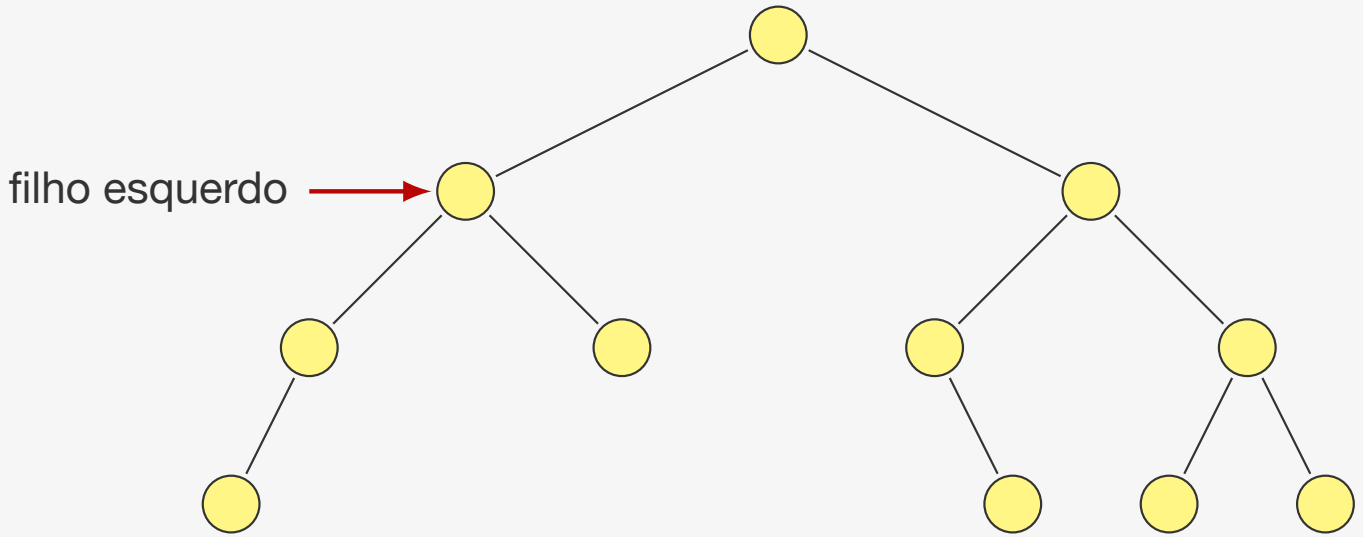
# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



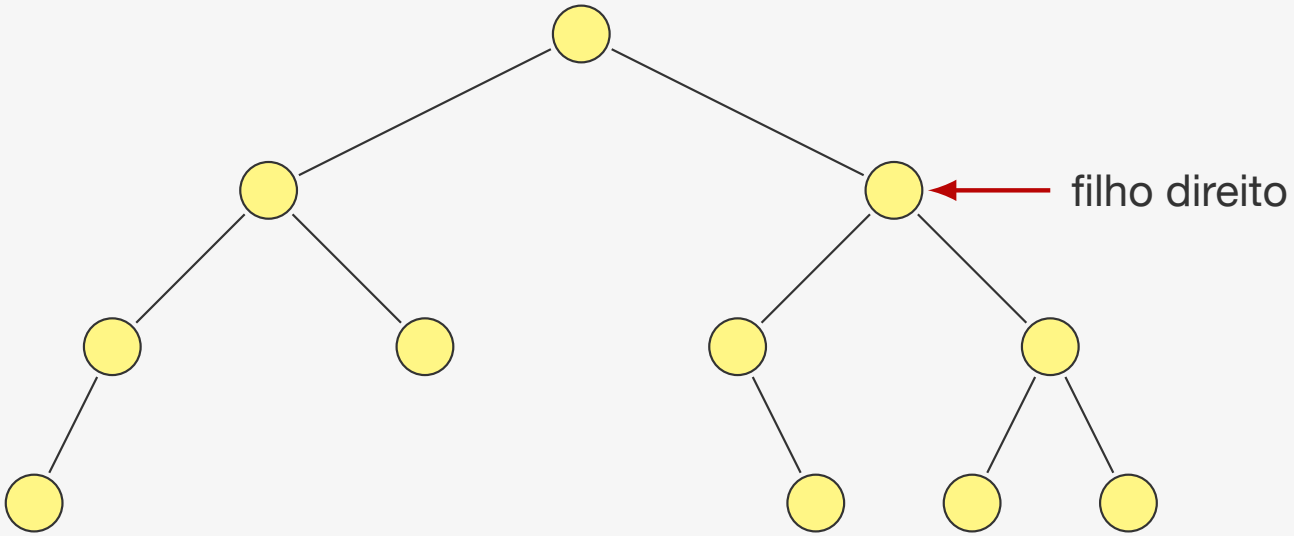
# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



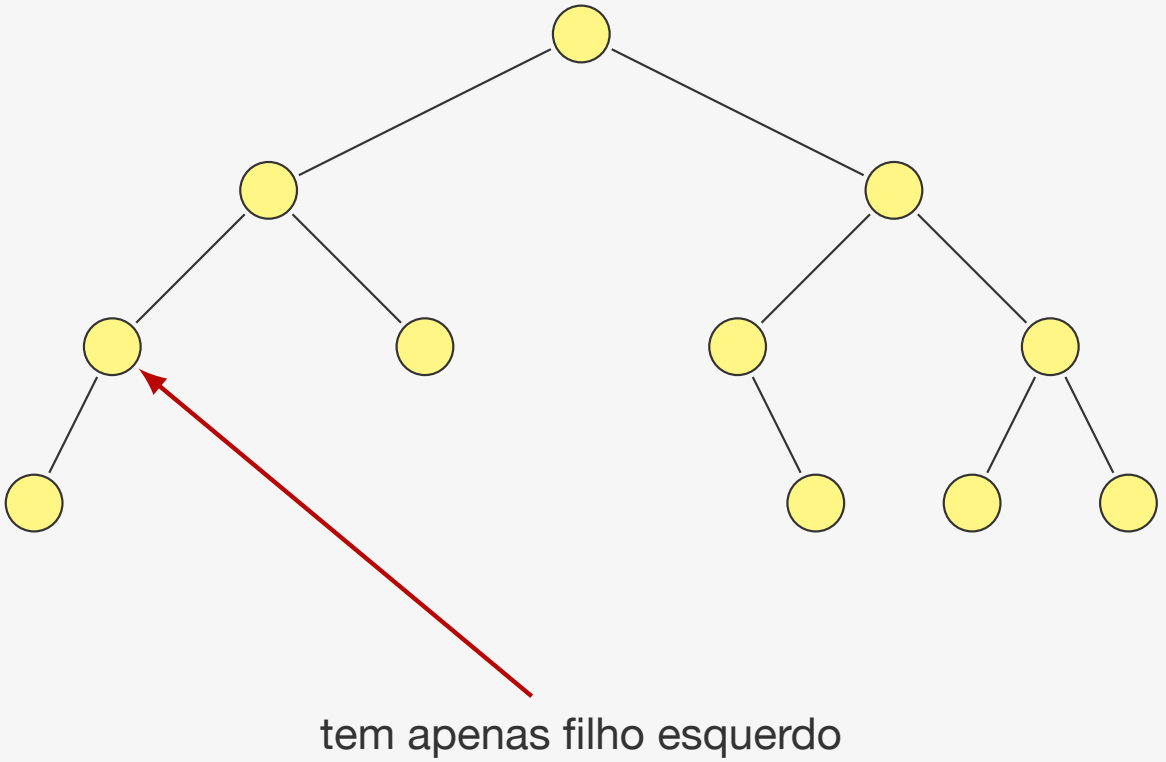
# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



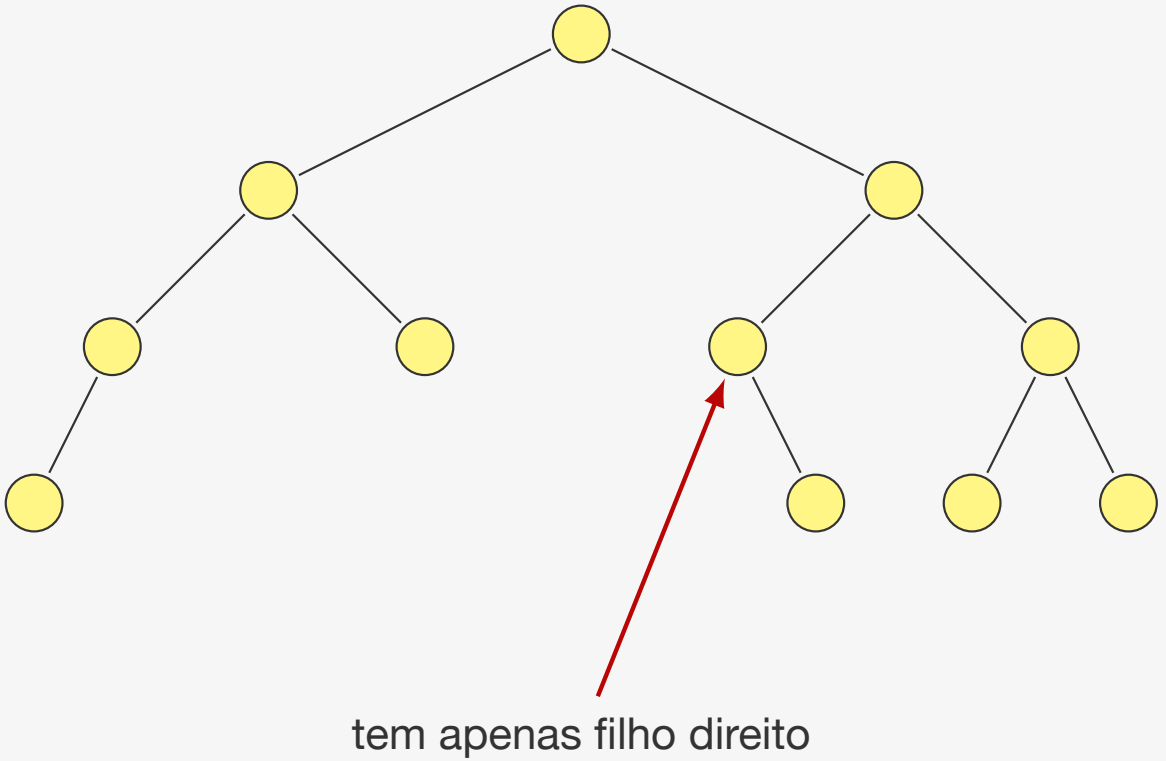
# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



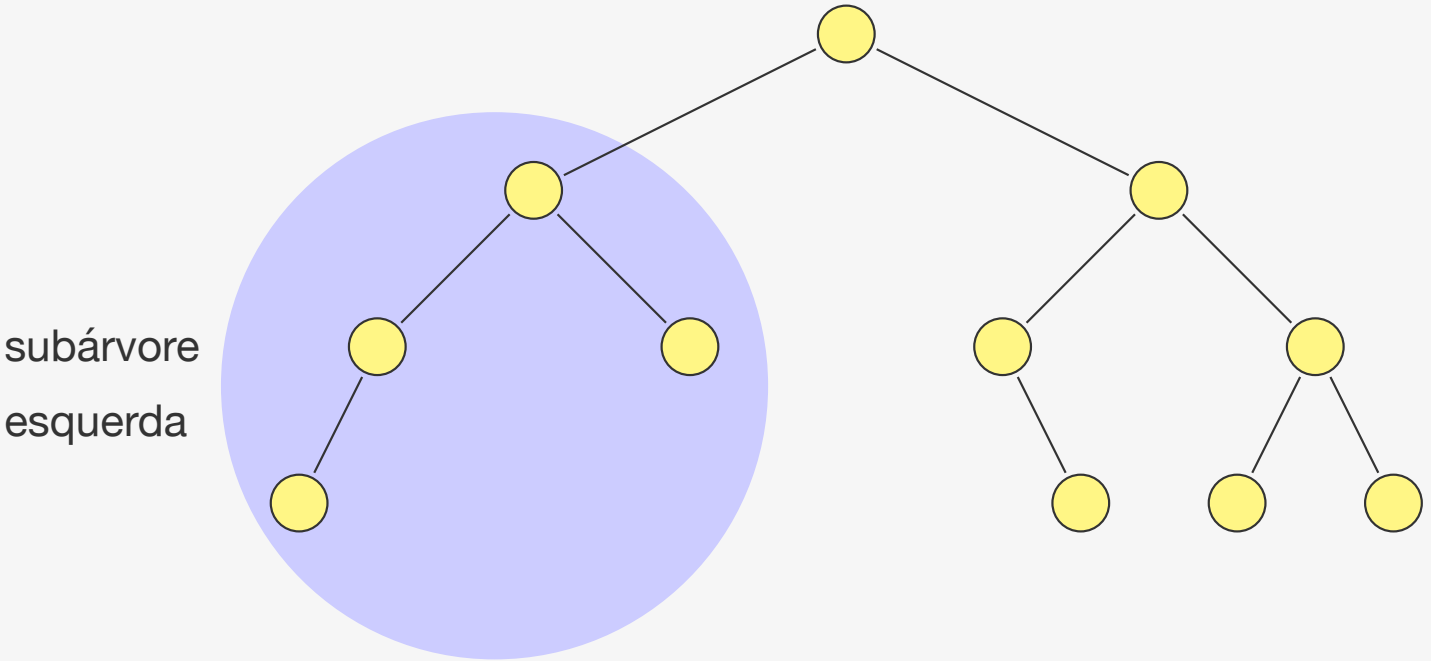
# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



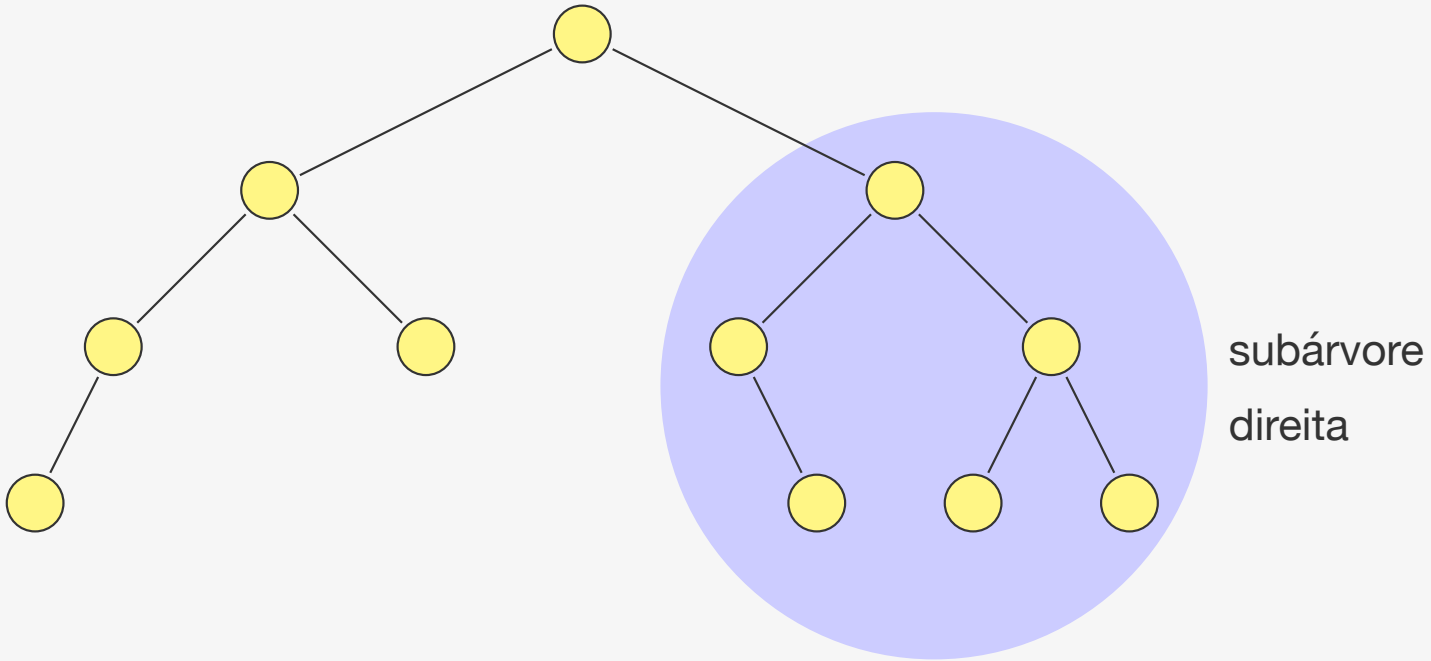
# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



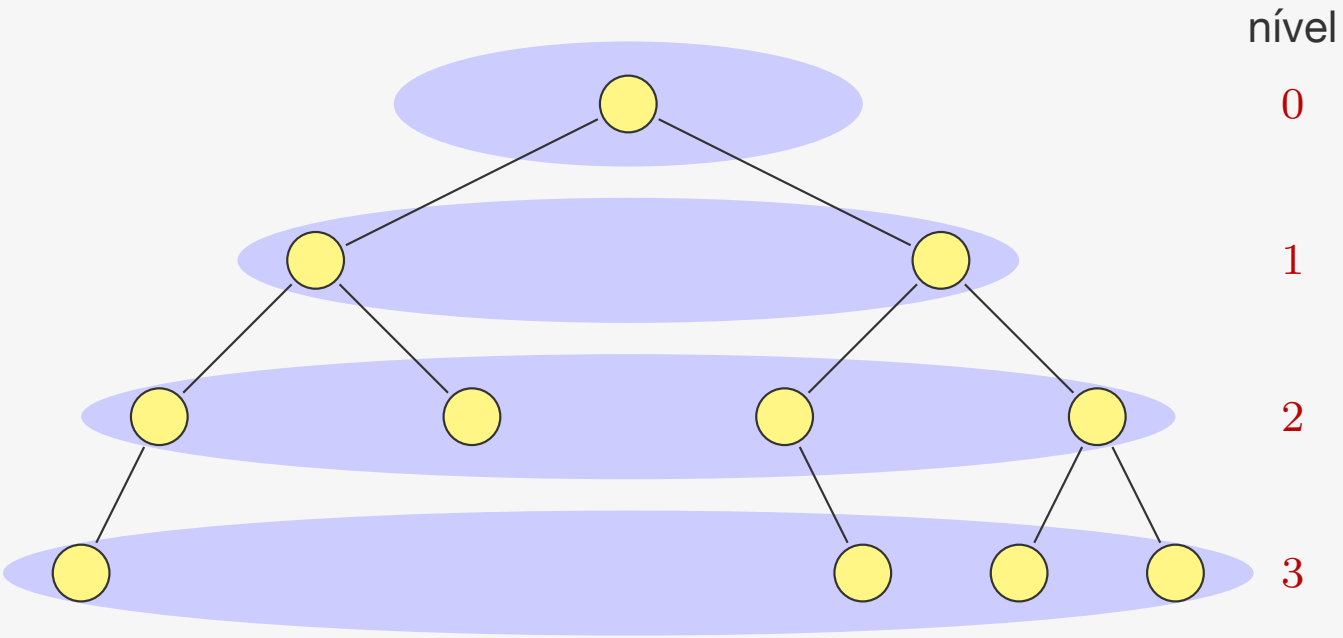
# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



# Interlúdio - Árvores Binárias

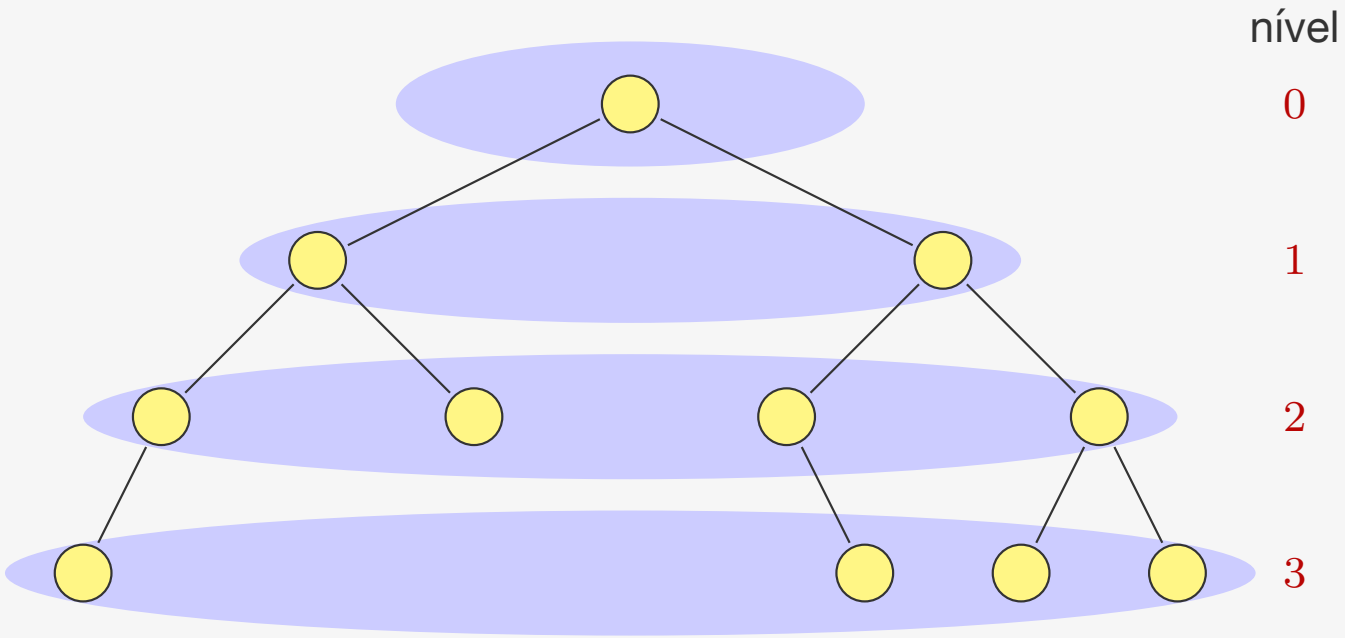
Exemplo de uma árvore binária:




Uma árvore binária é:

# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:

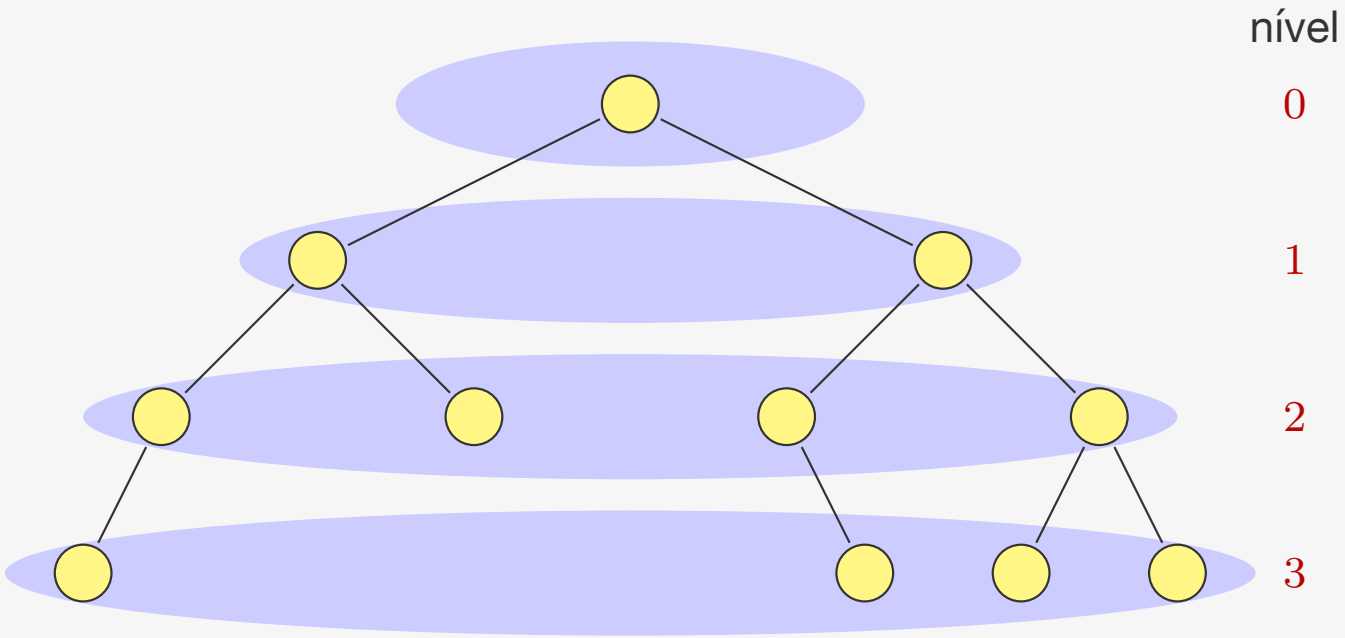



Uma árvore binária é:

- Ou o conjunto vazio

# Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:




Uma árvore binária é:

- Ou o conjunto vazio
- Ou um nó conectado a duas árvores binárias

# Árvores Binárias Completas

Uma árvore binária é dita completa se:

# Árvores Binárias Completas

Uma árvore binária é dita completa se:

- Todos os níveis exceto o último estão cheios

# Árvores Binárias Completas

Uma árvore binária é dita completa se:

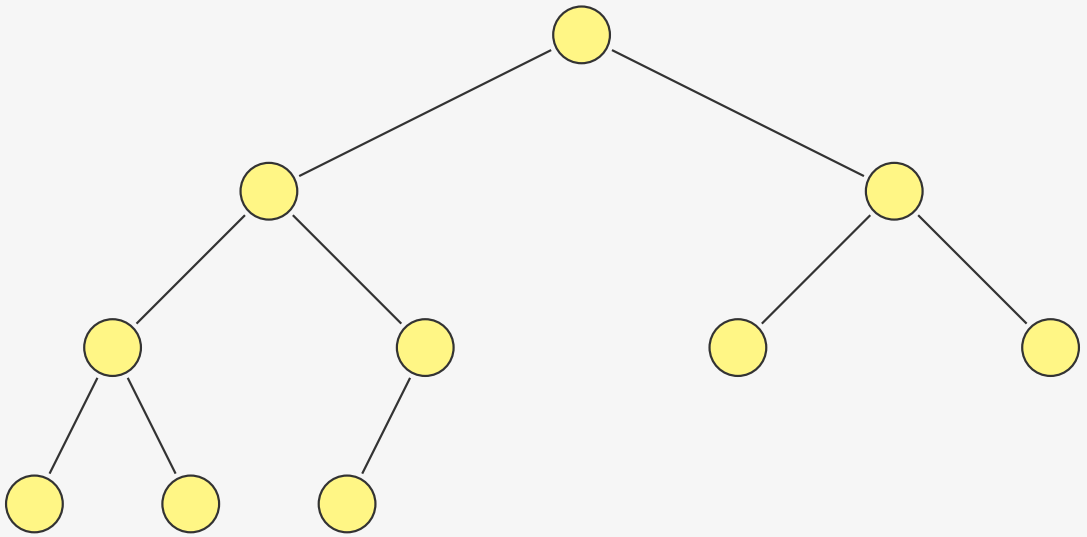
- Todos os níveis exceto o último estão cheios
- No último nível, todos os nós estão o mais a esquerda possível

# Árvores Binárias Completas

Uma árvore binária é dita completa se:

- Todos os níveis exceto o último estão cheios
- No último nível, todos os nós estão o mais a esquerda possível

Exemplo:

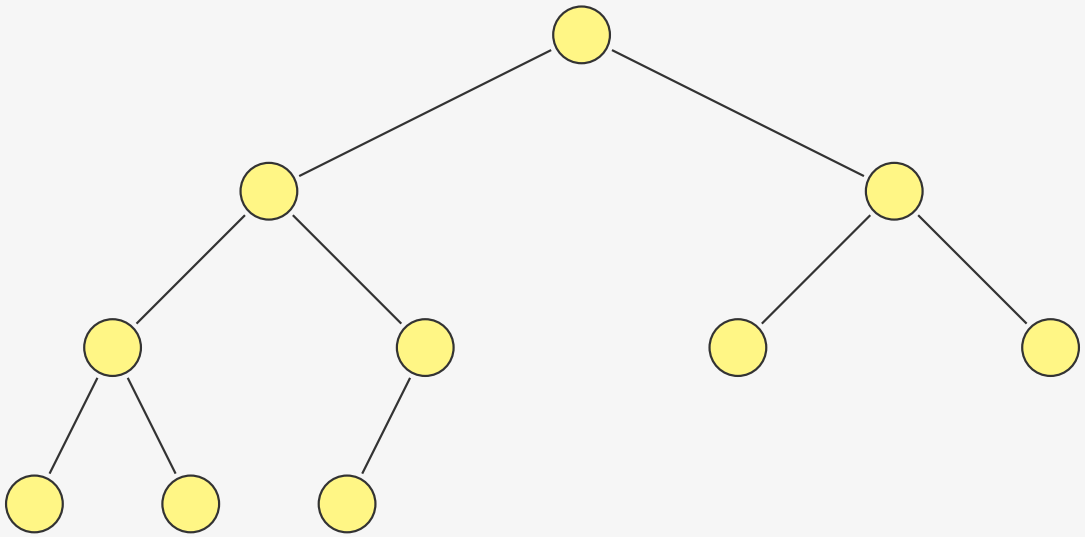


# Árvores Binárias Completas

Uma árvore binária é dita completa se:

- Todos os níveis exceto o último estão cheios
- No último nível, todos os nós estão o mais a esquerda possível

Exemplo:



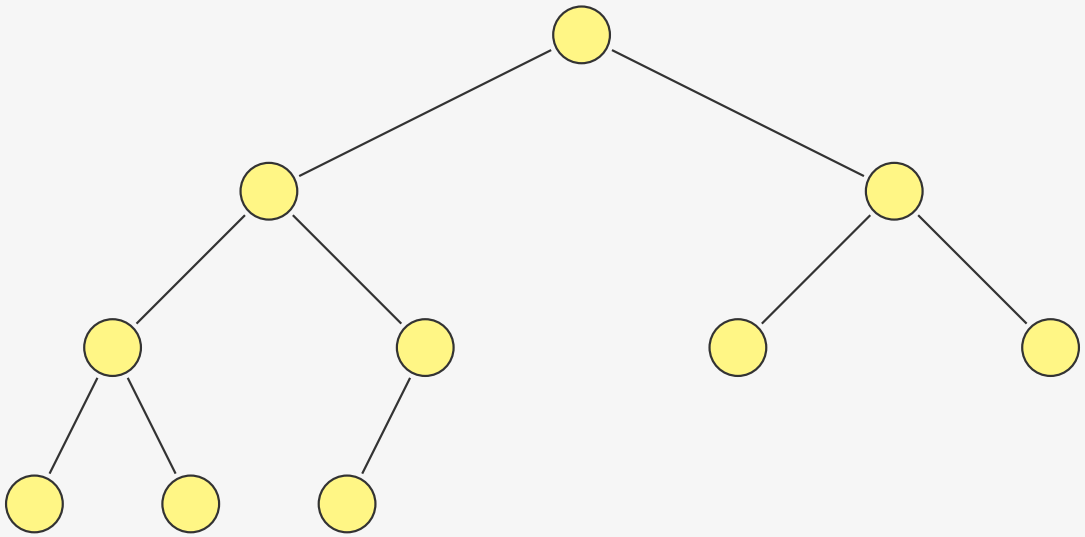
Uma árvore binária completa de $n$ nós tem quantos níveis?

# Árvores Binárias Completas

Uma árvore binária é dita completa se:

- Todos os níveis exceto o último estão cheios
- No último nível, todos os nós estão o mais a esquerda possível

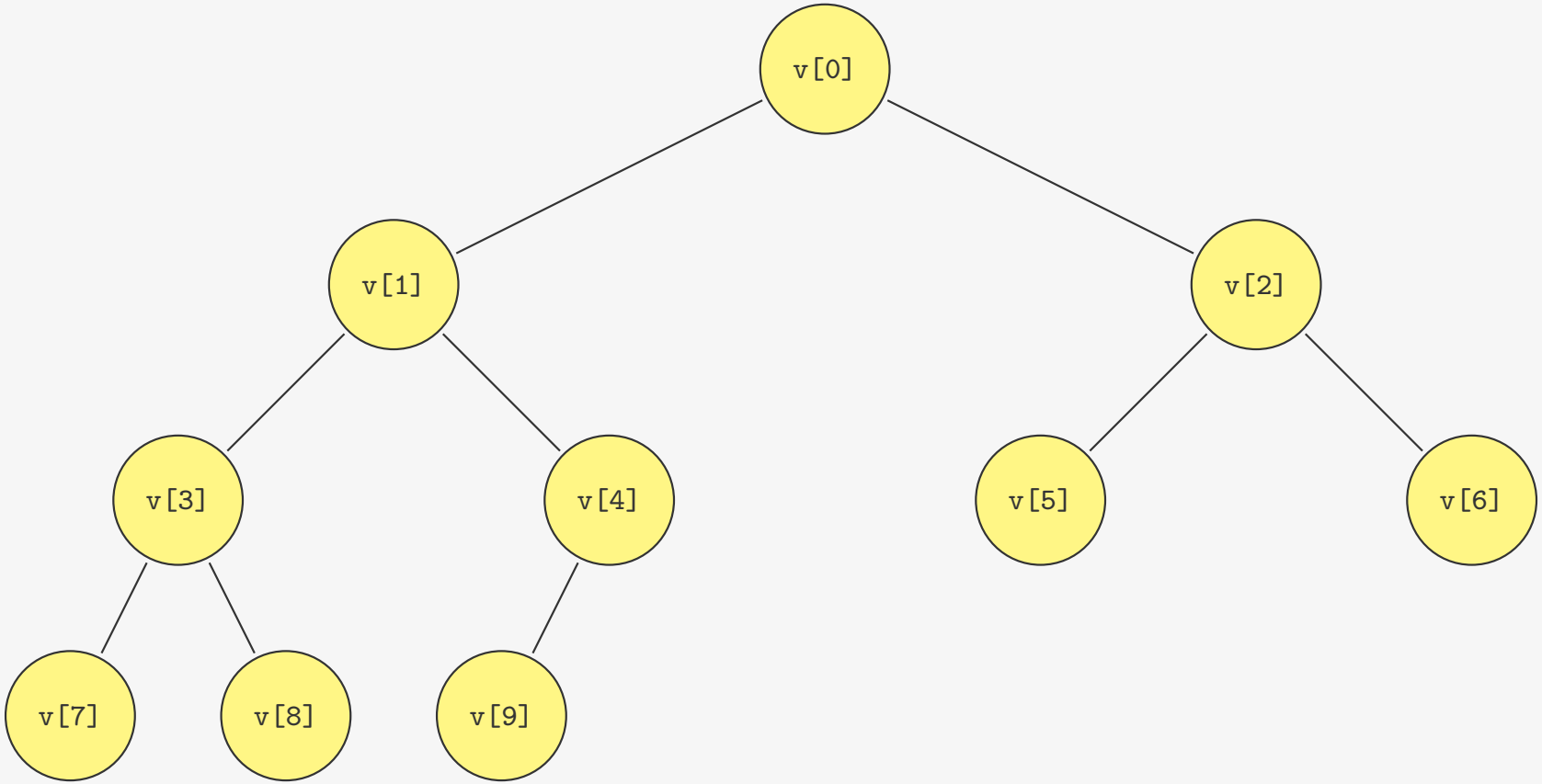
Exemplo:



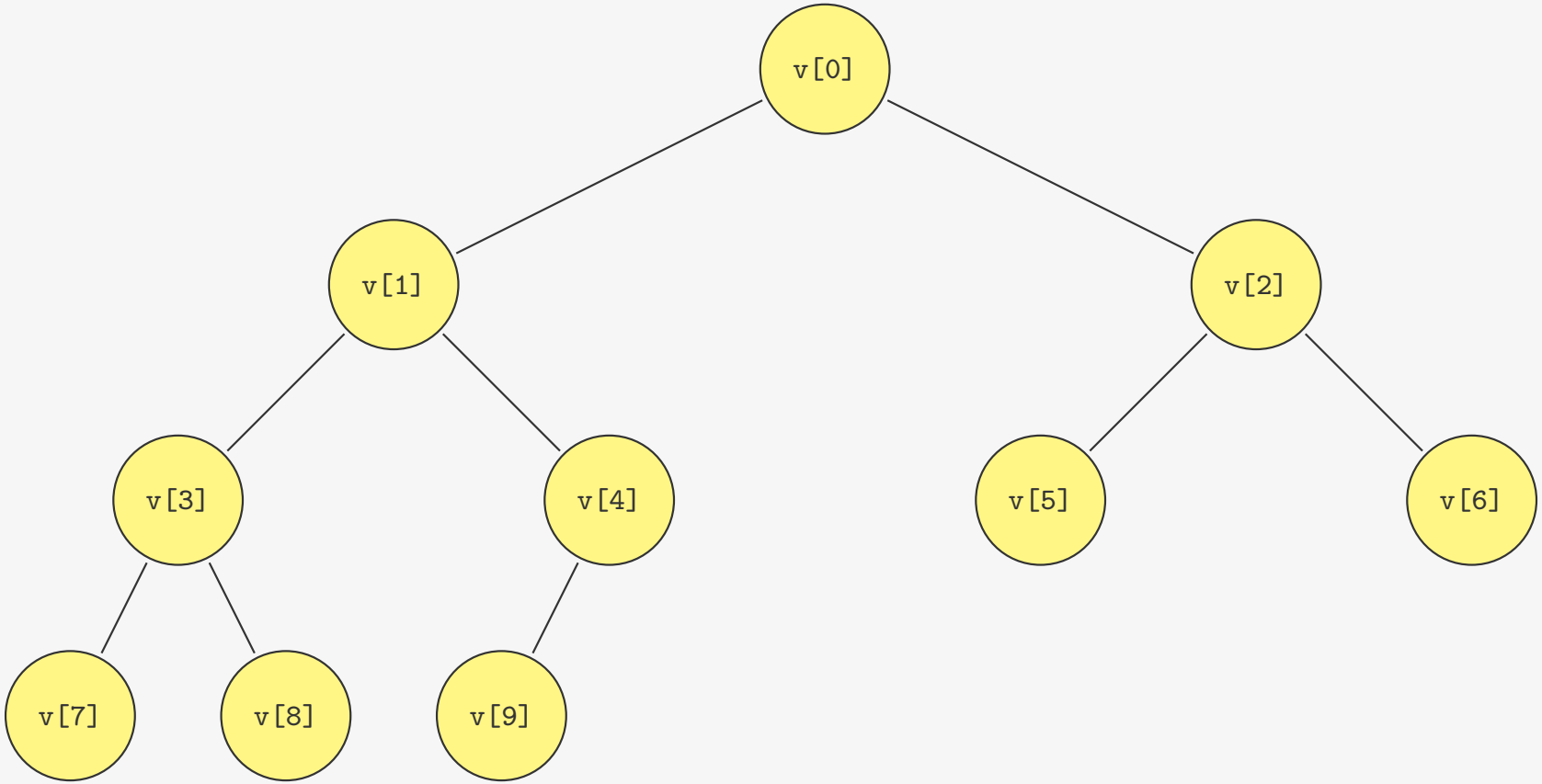
Uma árvore binária completa de $n$ nós tem quantos níveis?

- $\lceil \lg(n+1) \rceil = \mathrm{O}(\lg n)$ níveis

# Árvores Binárias Completas e Vetores

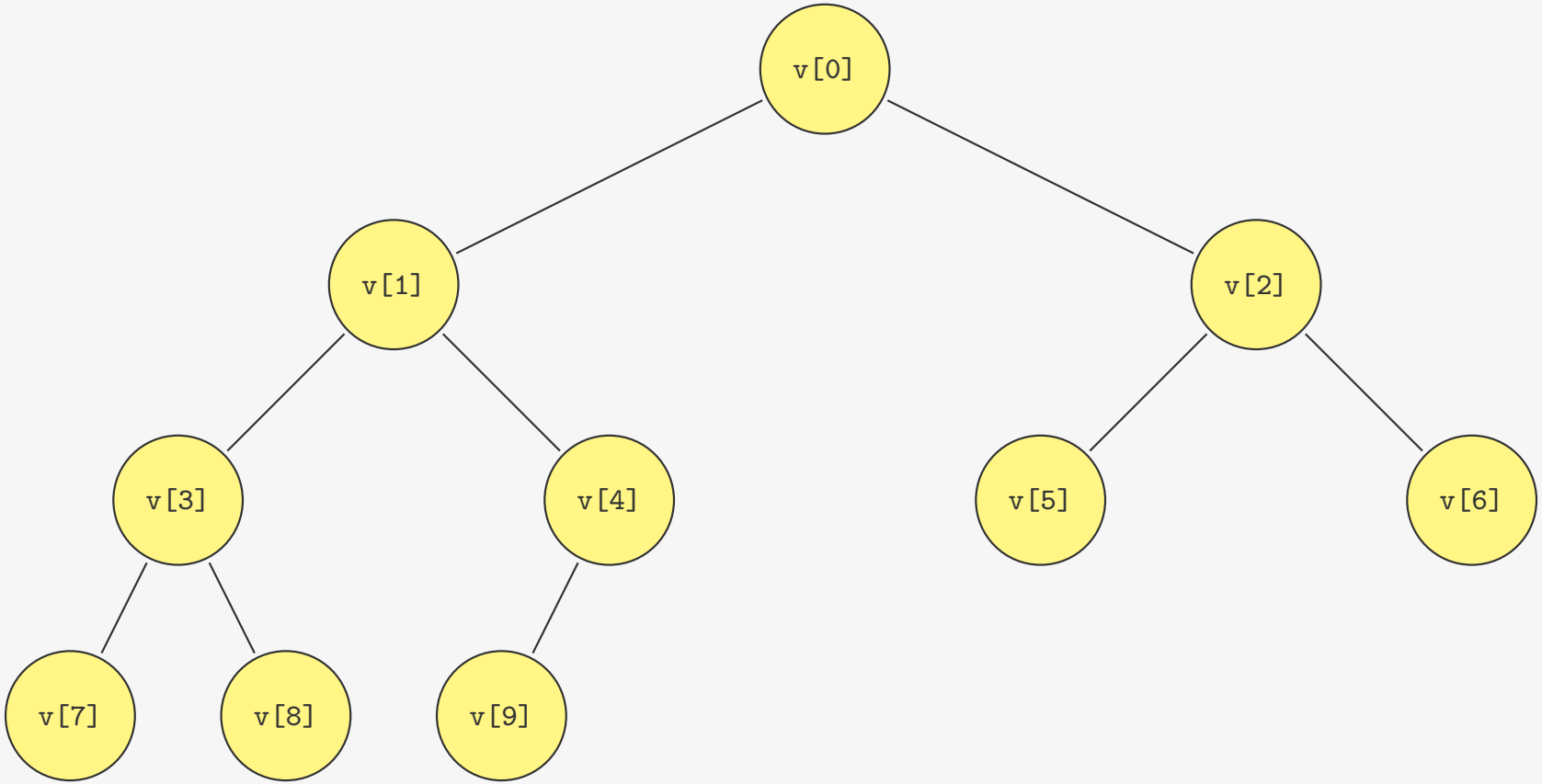


# Árvores Binárias Completas e Vetores




Em relação a `v[i]`:

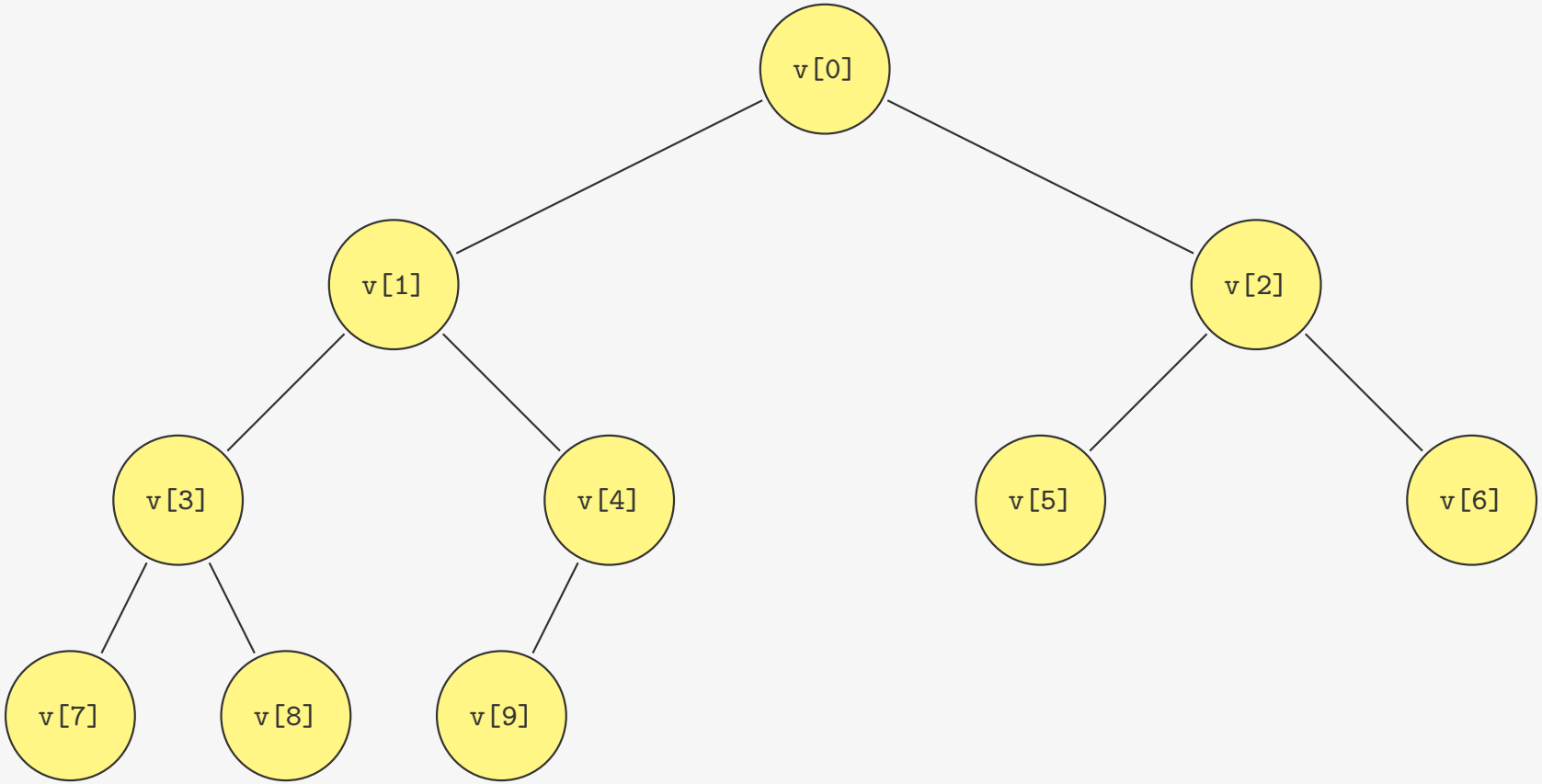
# Árvores Binárias Completas e Vetores




Em relação a `v[i]`:

- o filho esquerdo é `v[2*i+1]` e o filho direito é `v[2*i+2]`

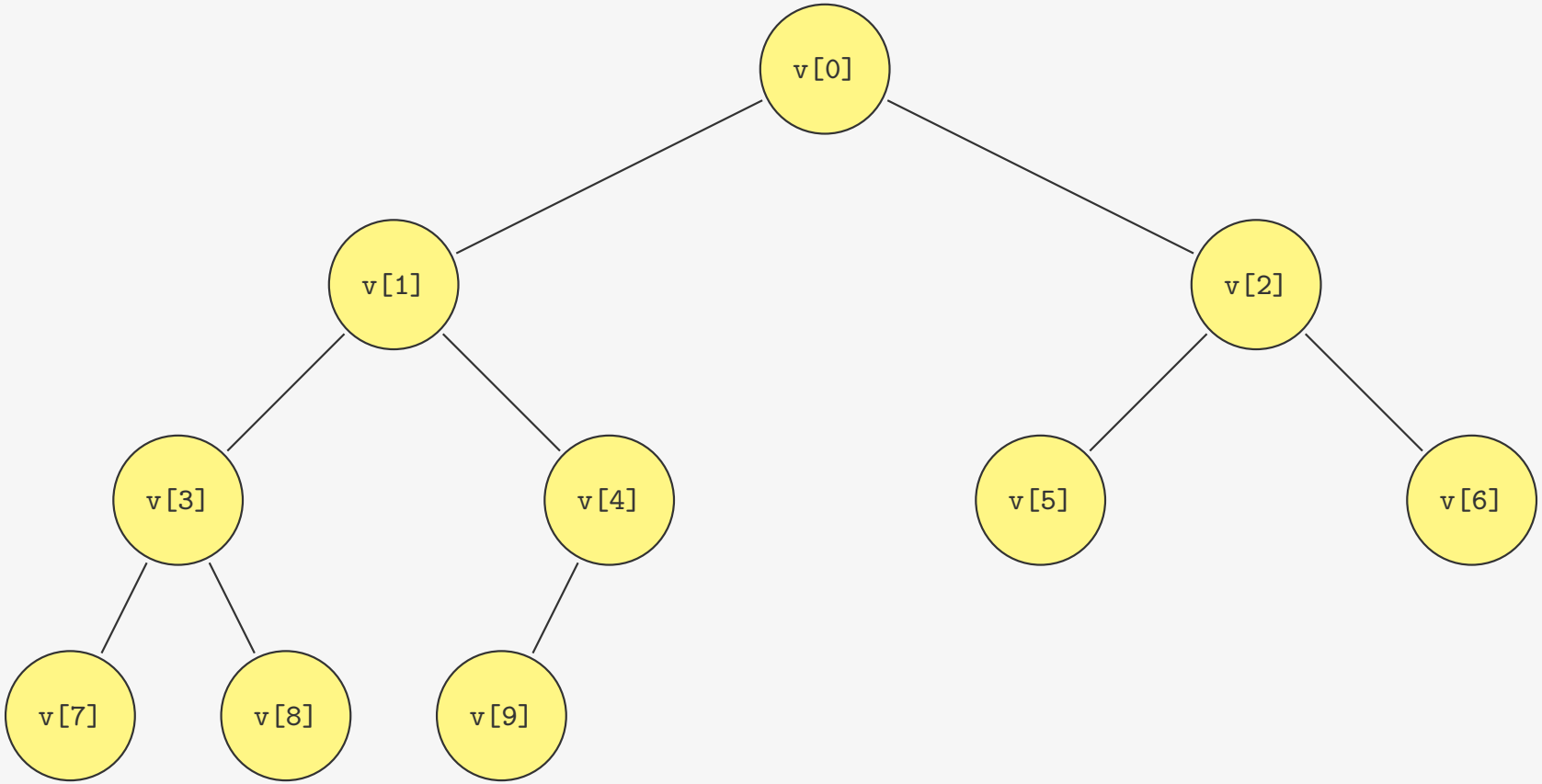
# Árvores Binárias Completas e Vetores



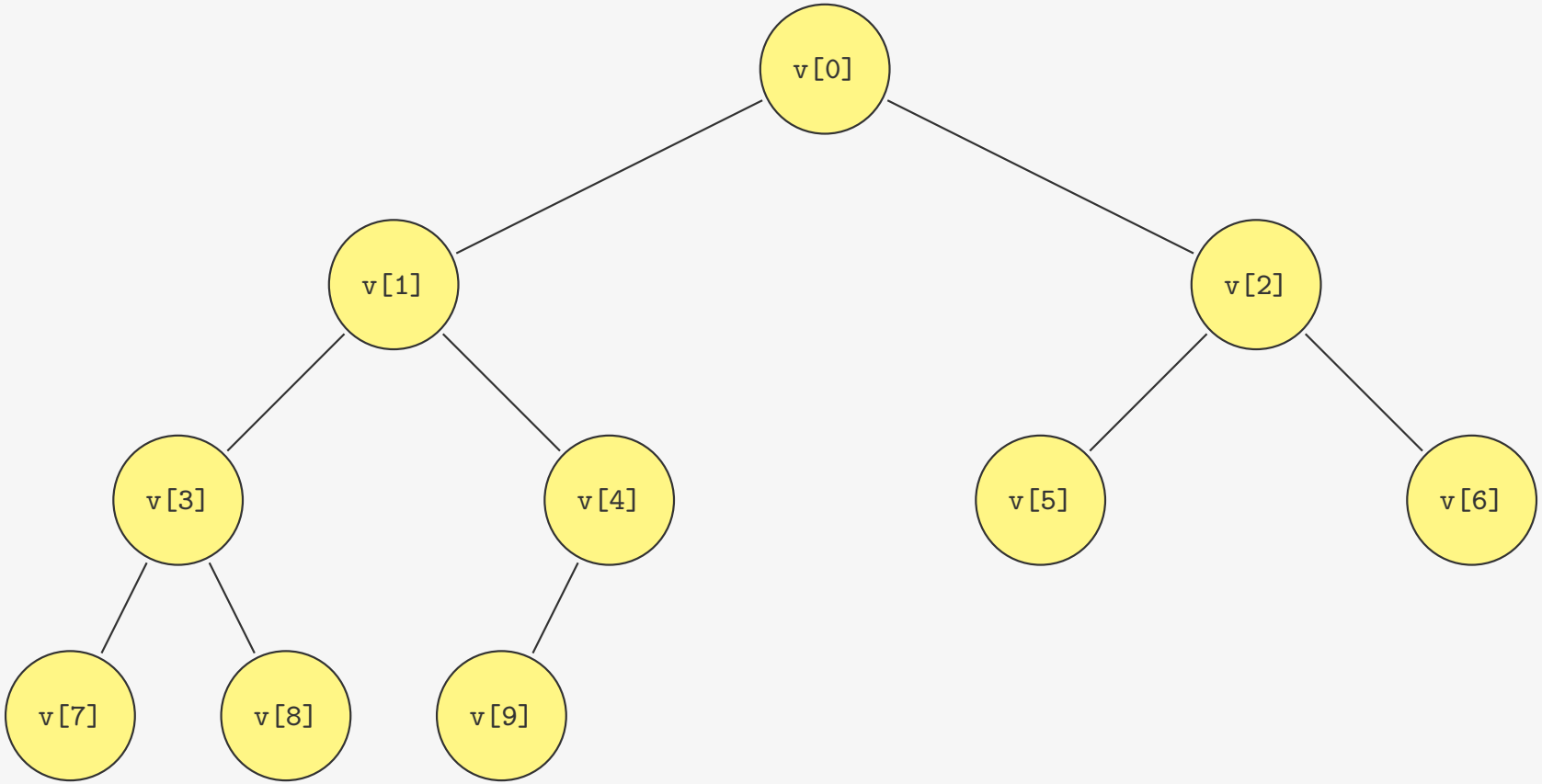

Em relação a `v[i]`:

- o filho esquerdo é `v[2*i+1]` e o filho direito é `v[2*i+2]`
- o pai é `v[(i-1)/2]`

# Max-Heap

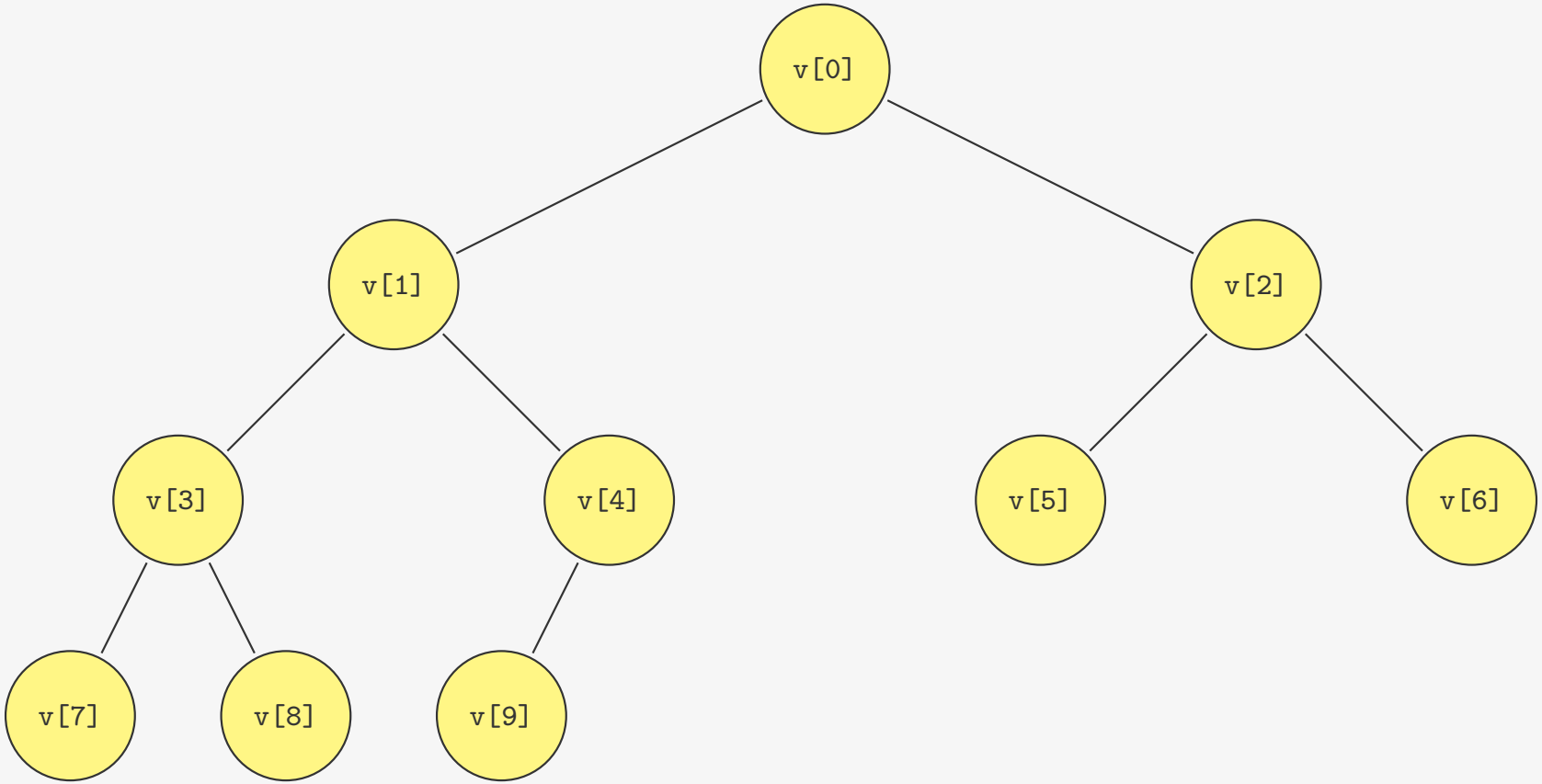


# Max-Heap




Em um Heap (de máximo):

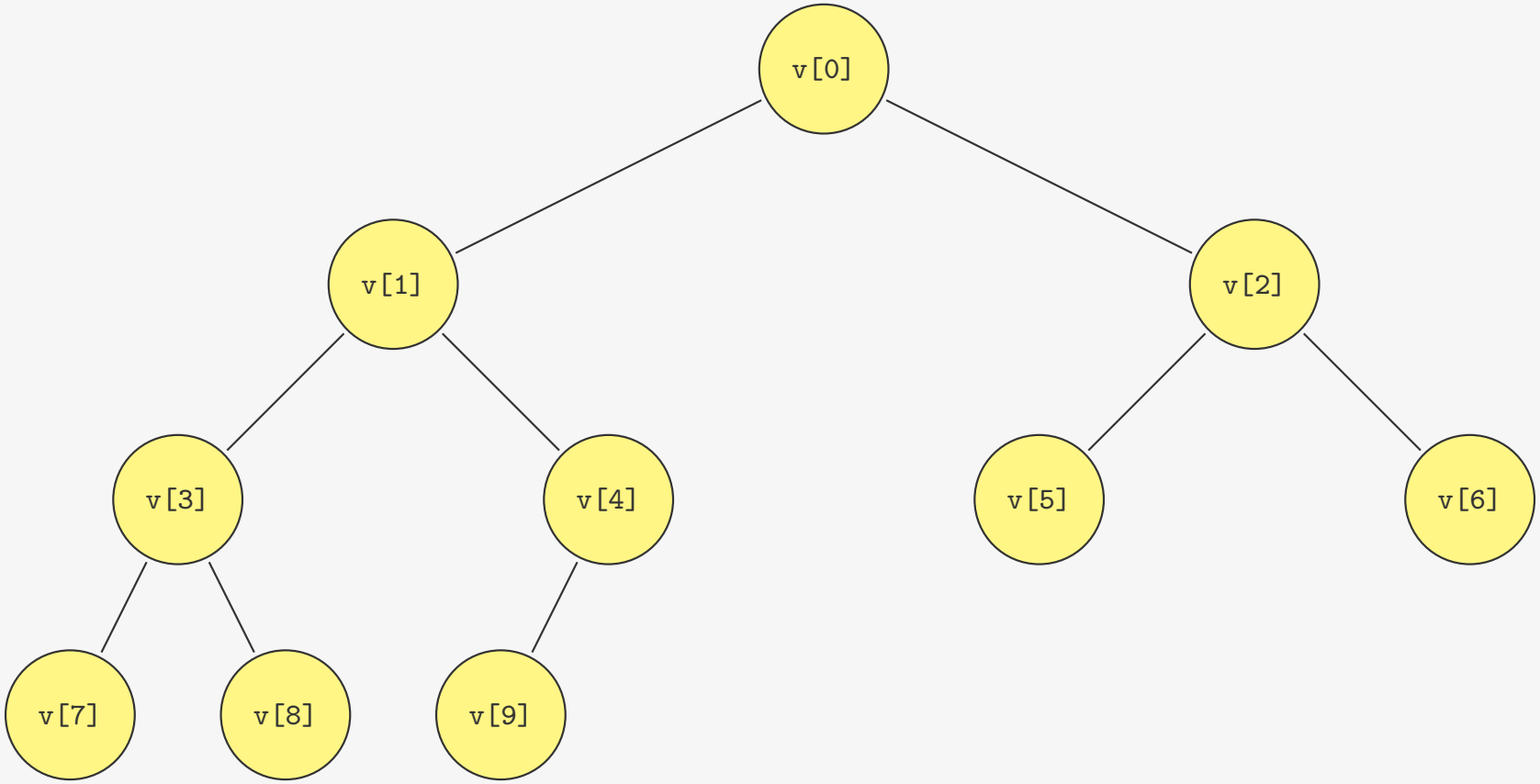
# Max-Heap




Em um Heap (de máximo):

- Os filhos são menores ou iguais ao pai

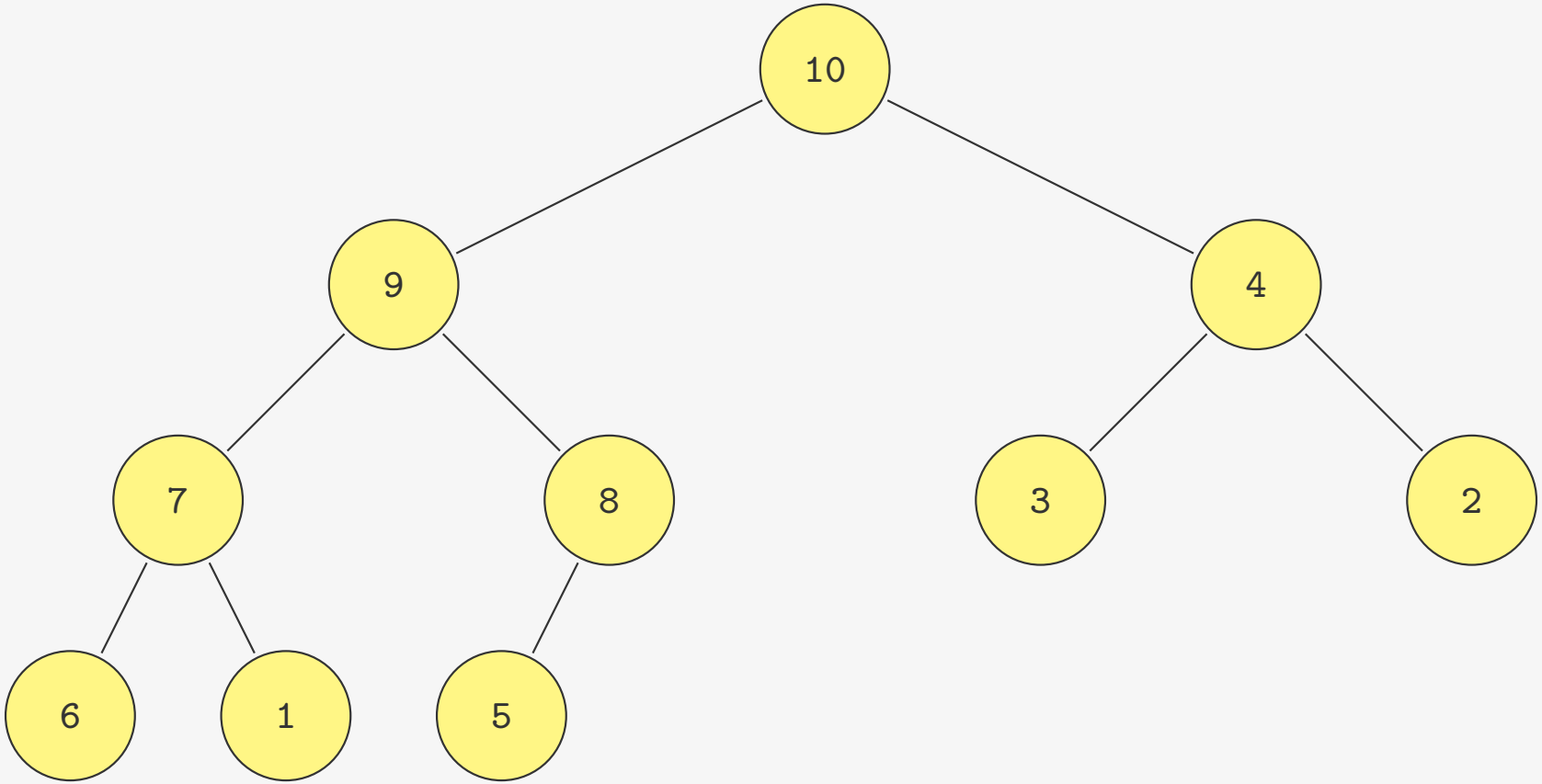
# Max-Heap




Em um Heap (de máximo):

- Os filhos são menores ou iguais ao pai
- Ou seja, a raiz é o máximo

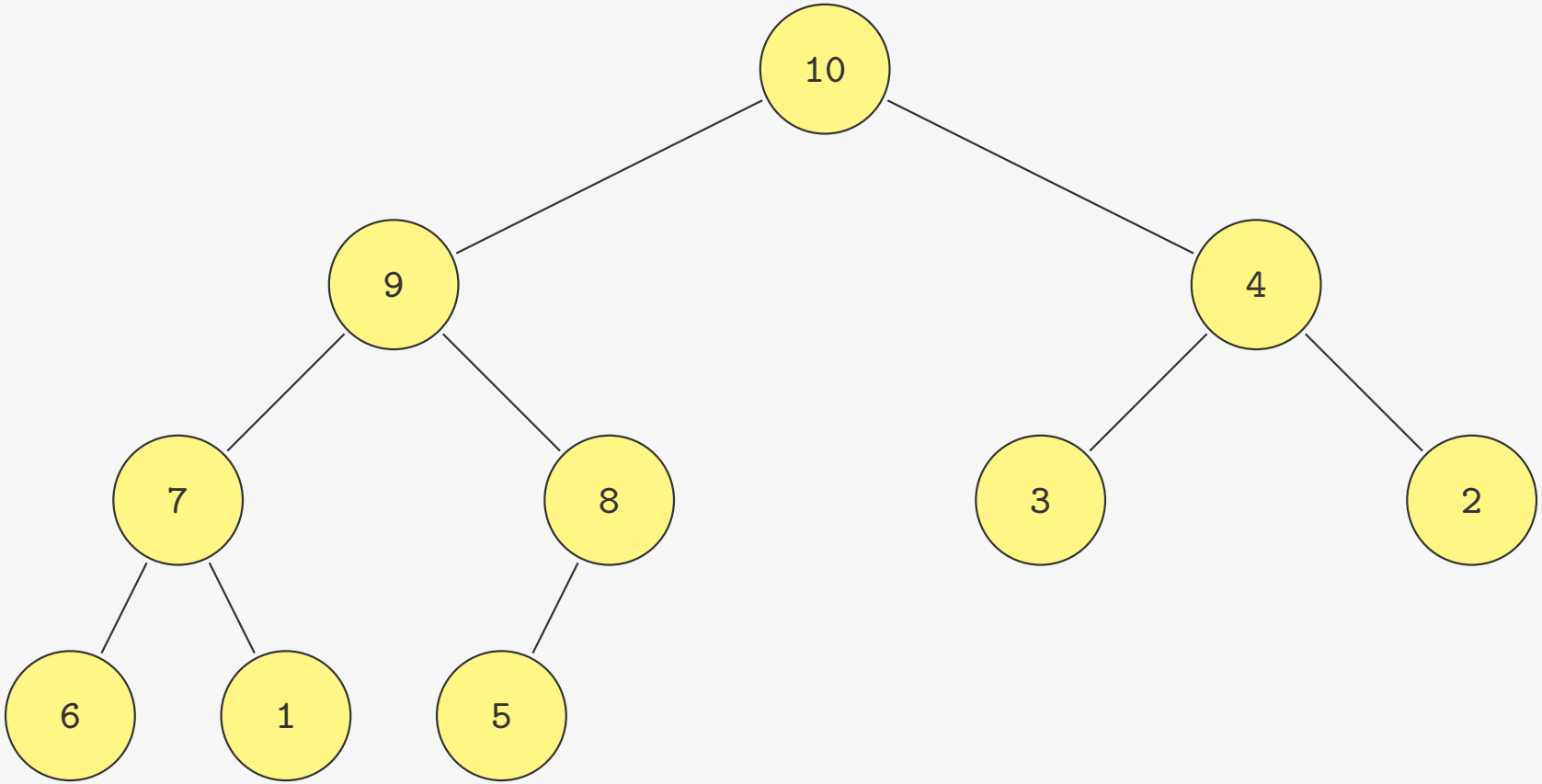
# Max-Heap




Em um Heap (de máximo):

- Os filhos são menores ou iguais ao pai
- Ou seja, a raiz é o máximo

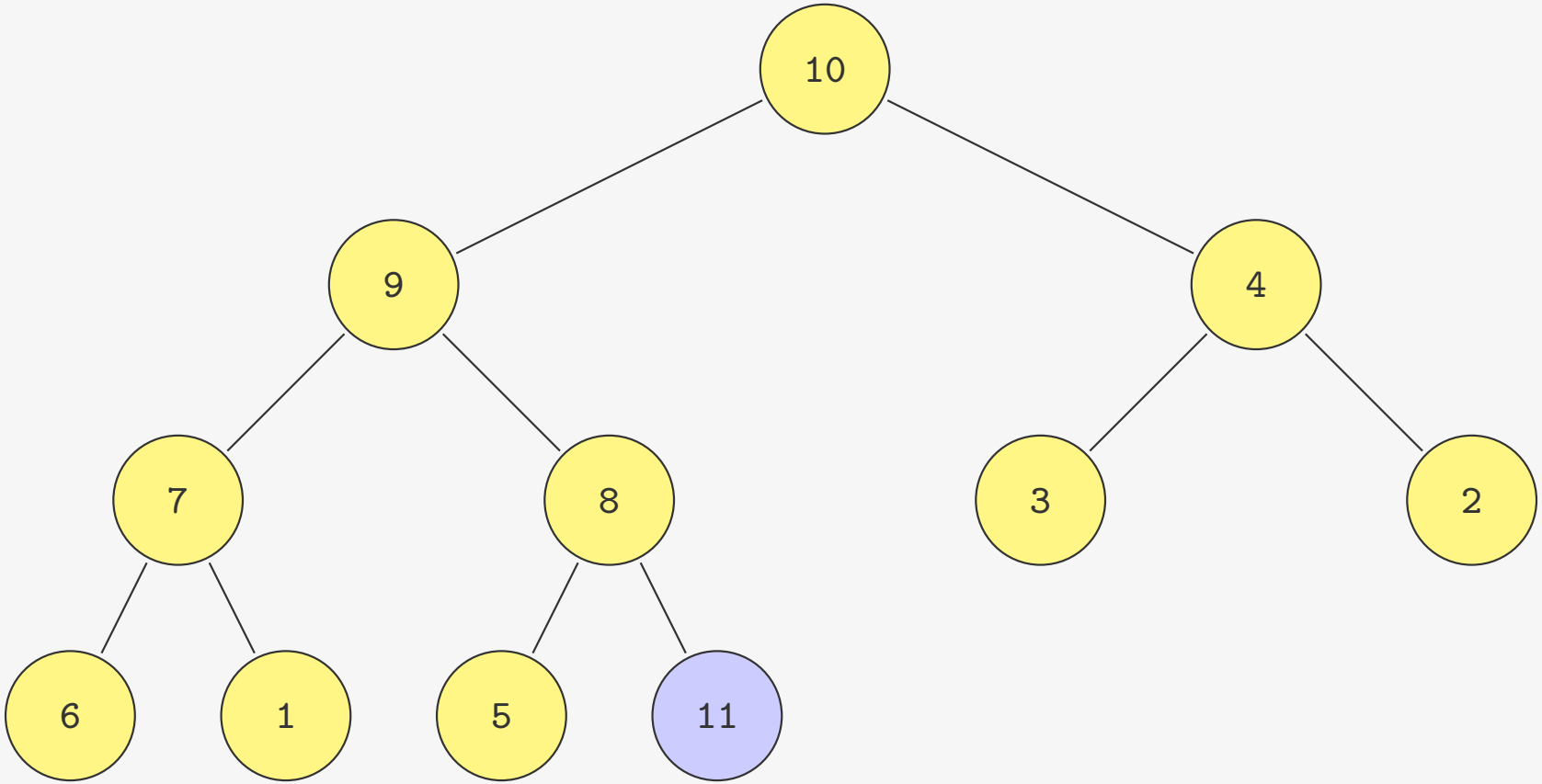
# Inserindo no Heap




Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

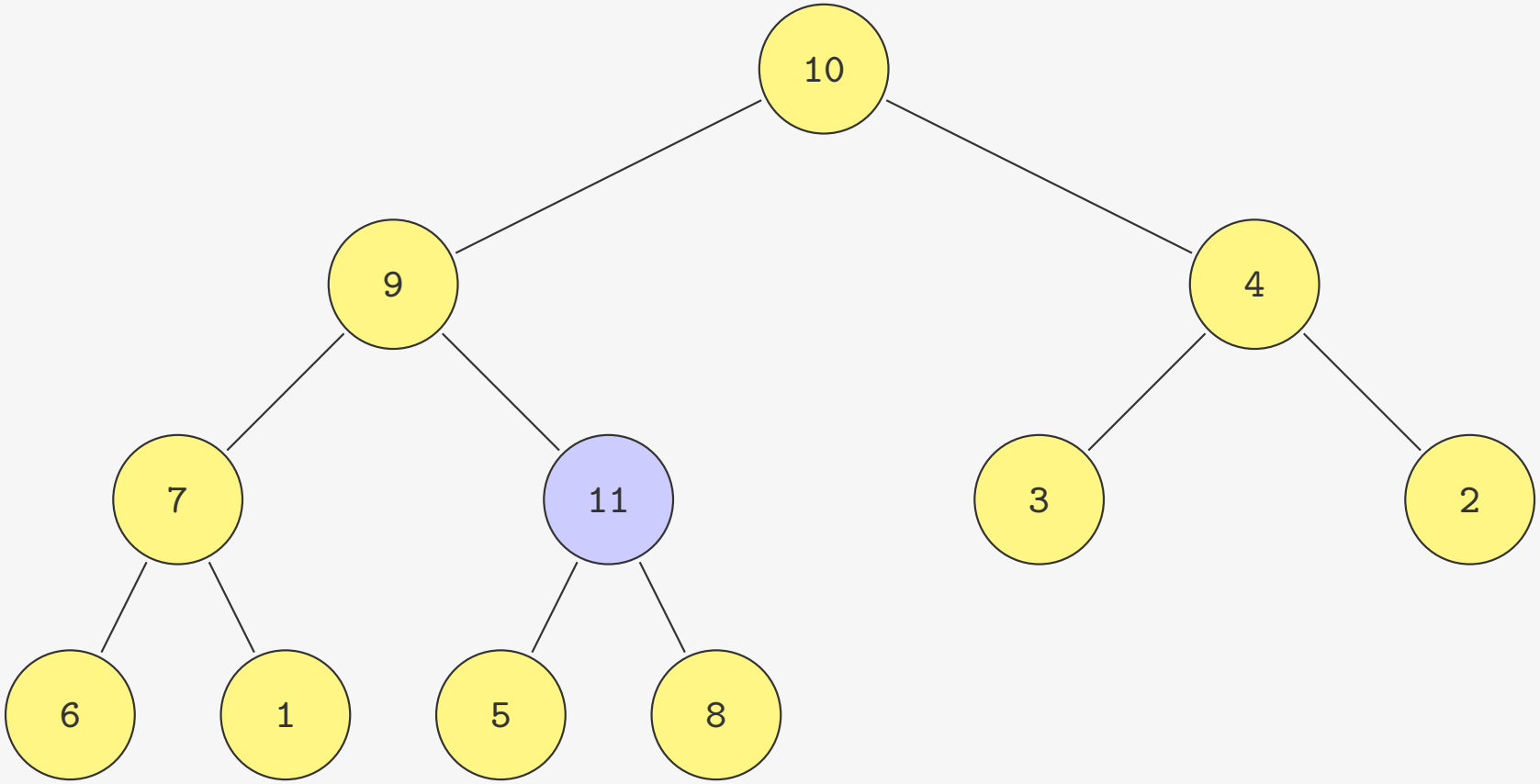
# Inserindo no Heap




Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

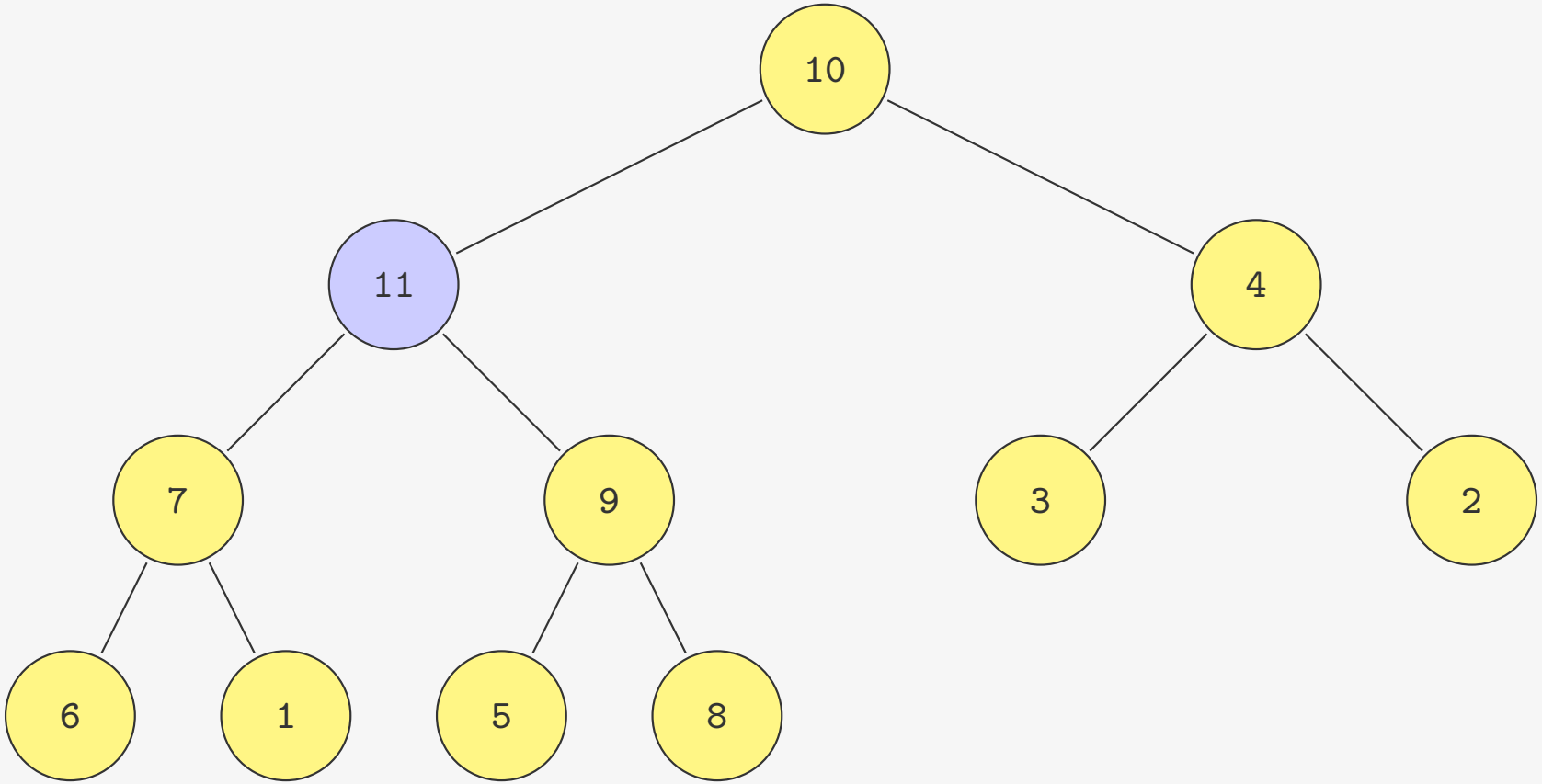
# Inserindo no Heap




Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

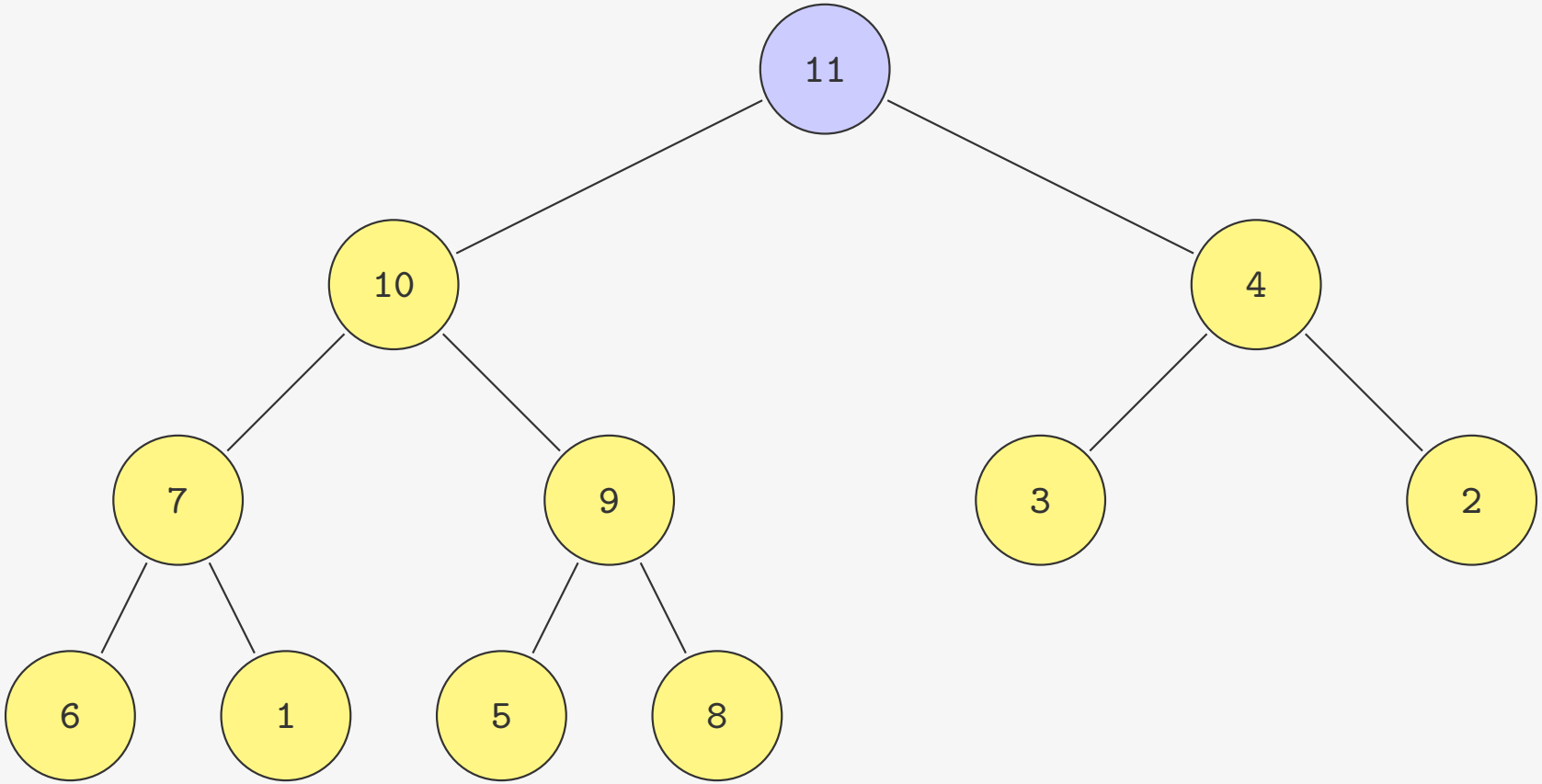
# Inserindo no Heap




Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

# Inserindo no Heap




Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

# Inserindo no Heap

```
 1 void insere(FilaP *fp, Item item) {
 2   fp->v[fp->n] = item;
 3   (fp->n)++;
 4   sobe_no_heap(fp, fp->n - 1);
 5 }
 6
 7 #define PAI(i) ((i-1)/2)
 8
 9 void sobe_no_heap(FilaP *fp, int k) {
10   if (k > 0 && fp->v[PAI(k)].chave < fp->v[k].chave) {
11     troca(&fp->v[k], &fp->v[PAI(k)]);
12     sobe_no_heap(fp, PAI(k));
13   }
14 }
```

# Inserindo no Heap

```
 1 void insere(FilaP *fp, Item item) {
 2   fp->v[fp->n] = item;
 3   (fp->n)++;
 4   sobe_no_heap(fp, fp->n - 1);
 5 }
 6
 7 #define PAI(i) ((i-1)/2)
 8
 9 void sobe_no_heap(FilaP *fp, int k) {
10   if (k > 0 && fp->v[PAI(k)].chave < fp->v[k].chave) {
11     troca(&fp->v[k], &fp->v[PAI(k)]);
12     sobe_no_heap(fp, PAI(k));
13   }
14 }
```

Tempo de `insere`:

# Inserindo no Heap

```
 1 void insere(FilaP *fp, Item item) {
 2   fp->v[fp->n] = item;
 3   (fp->n)++;
 4   sobe_no_heap(fp, fp->n - 1);
 5 }
 6
 7 #define PAI(i) ((i-1)/2)
 8
 9 void sobe_no_heap(FilaP *fp, int k) {
10   if (k > 0 && fp->v[PAI(k)].chave < fp->v[k].chave) {
11     troca(&fp->v[k], &fp->v[PAI(k)]);
12     sobe_no_heap(fp, PAI(k));
13   }
14 }
```

Tempo de `insere`:

- No máximo subimos até a raiz

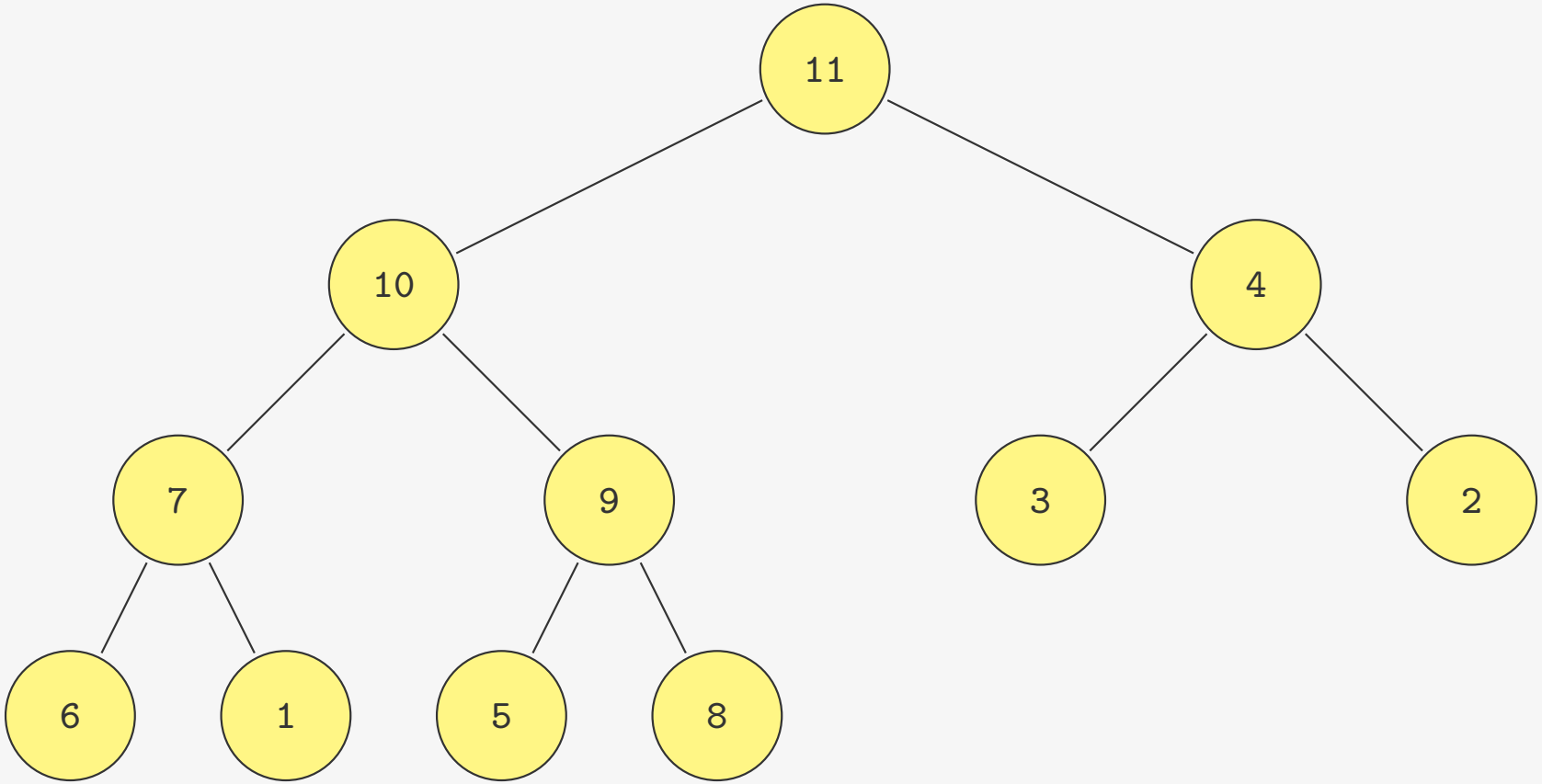
# Inserindo no Heap

```
 1 void insere(FilaP *fp, Item item) {
 2   fp->v[fp->n] = item;
 3   (fp->n)++;
 4   sobe_no_heap(fp, fp->n - 1);
 5 }
 6
 7 #define PAI(i) ((i-1)/2)
 8
 9 void sobe_no_heap(FilaP *fp, int k) {
10   if (k > 0 && fp->v[PAI(k)].chave < fp->v[k].chave) {
11     troca(&fp->v[k], &fp->v[PAI(k)]);
12     sobe_no_heap(fp, PAI(k));
13   }
14 }
```

Tempo de `insere`:

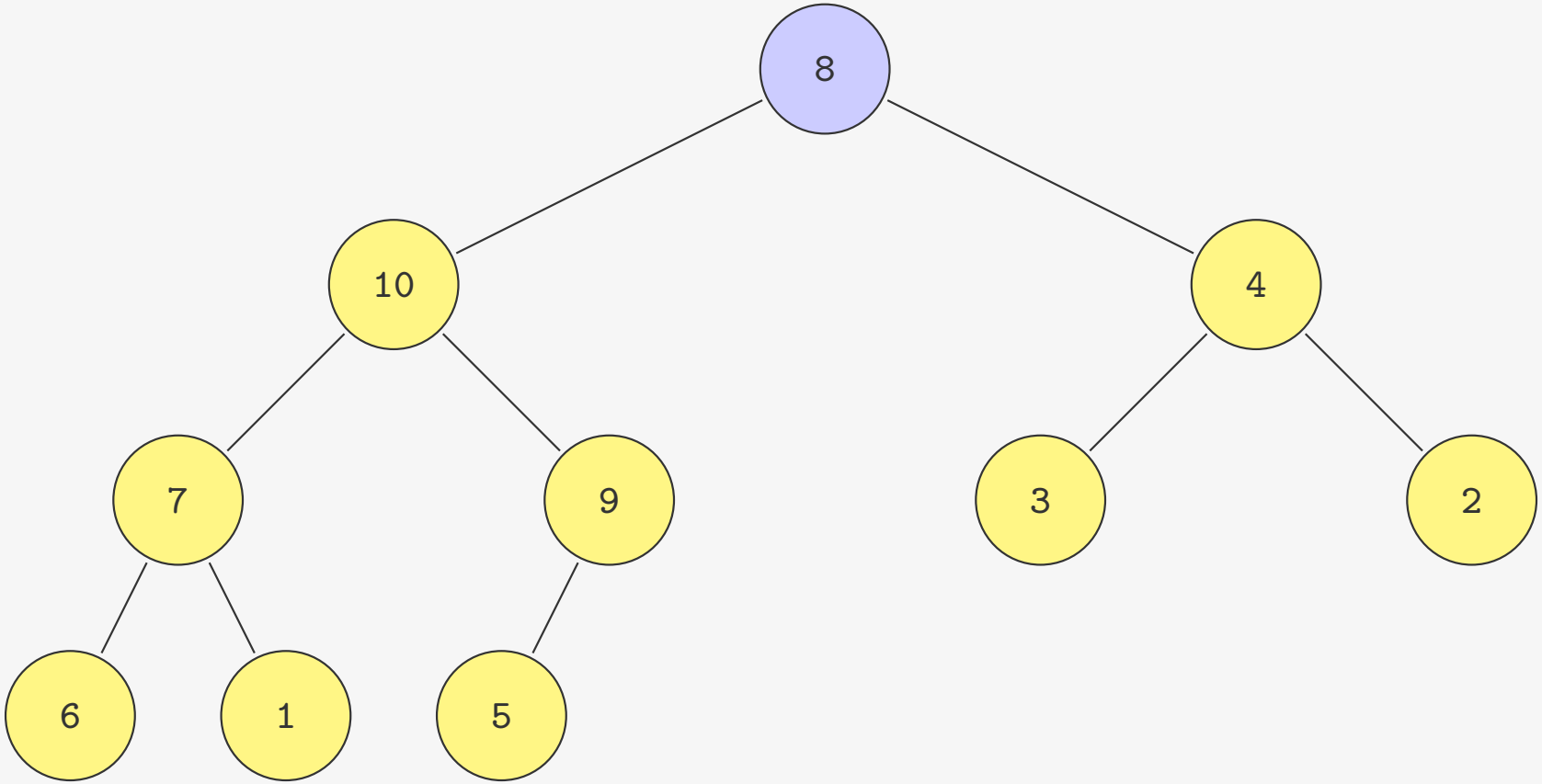
- No máximo subimos até a raiz
- Ou seja, $\mathrm{O}(\lg n)$

# Extraindo o Máximo



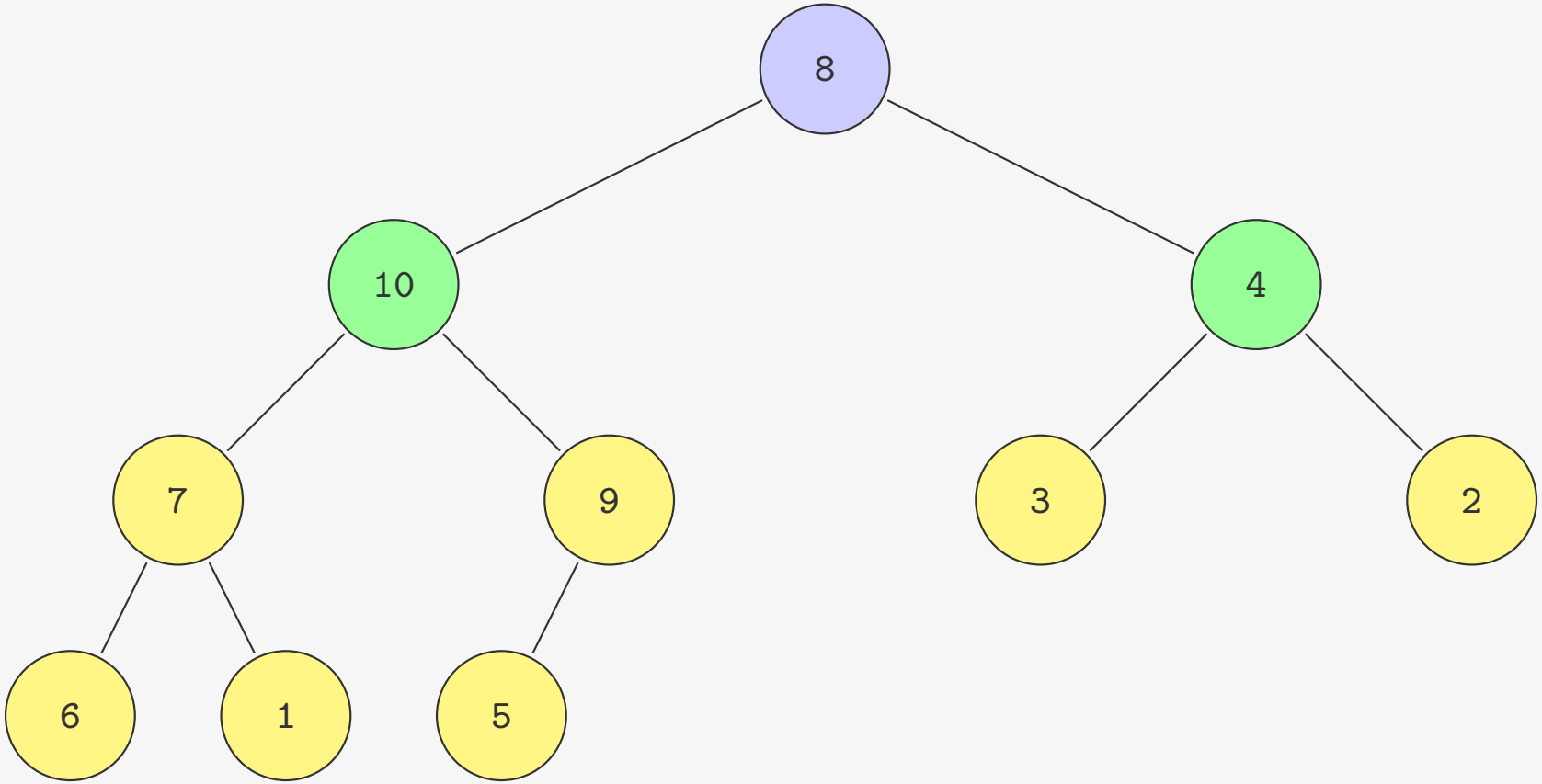

- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

# Extraindo o Máximo



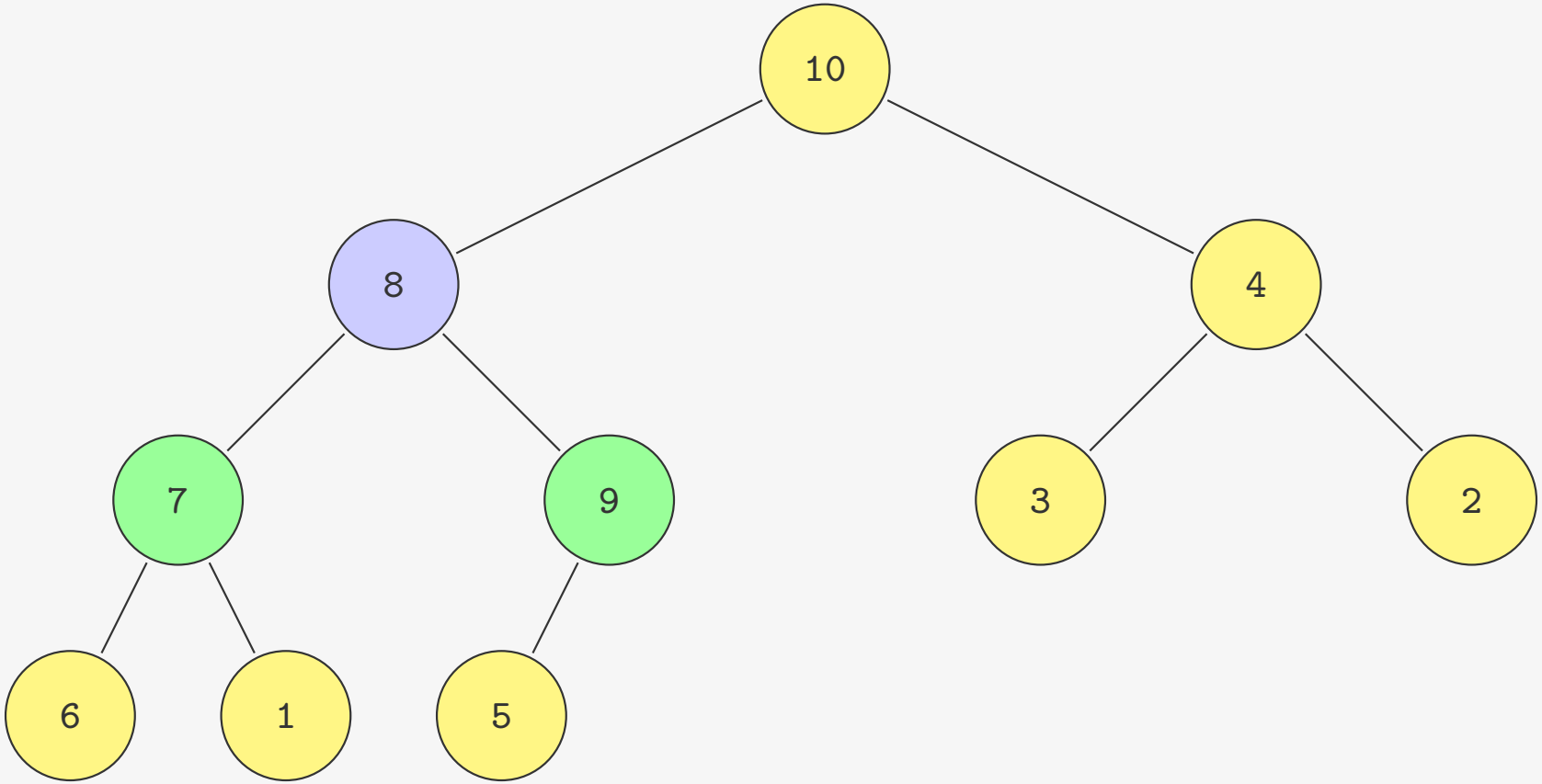

- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

# Extraindo o Máximo



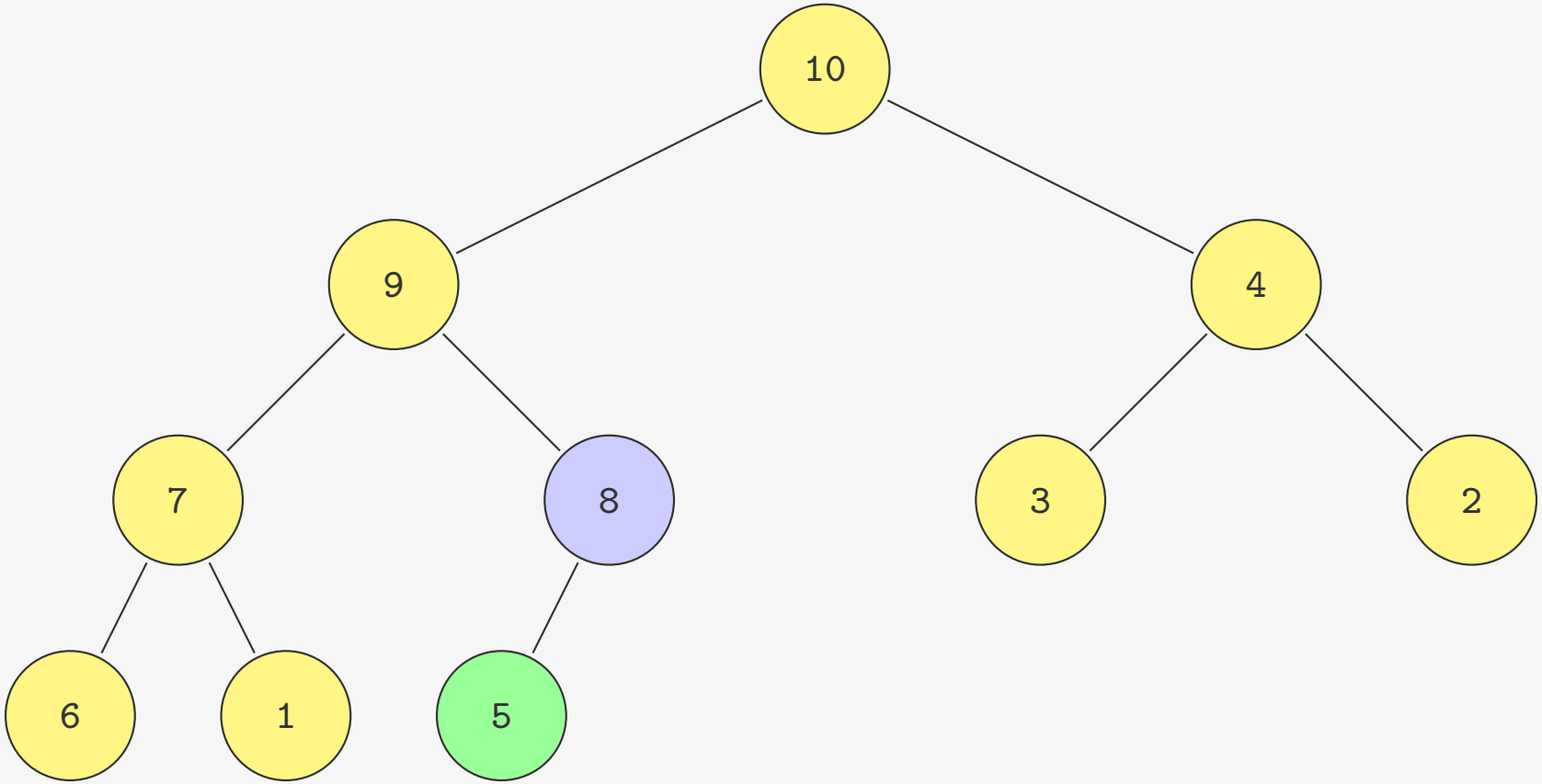

- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

# Extraindo o Máximo



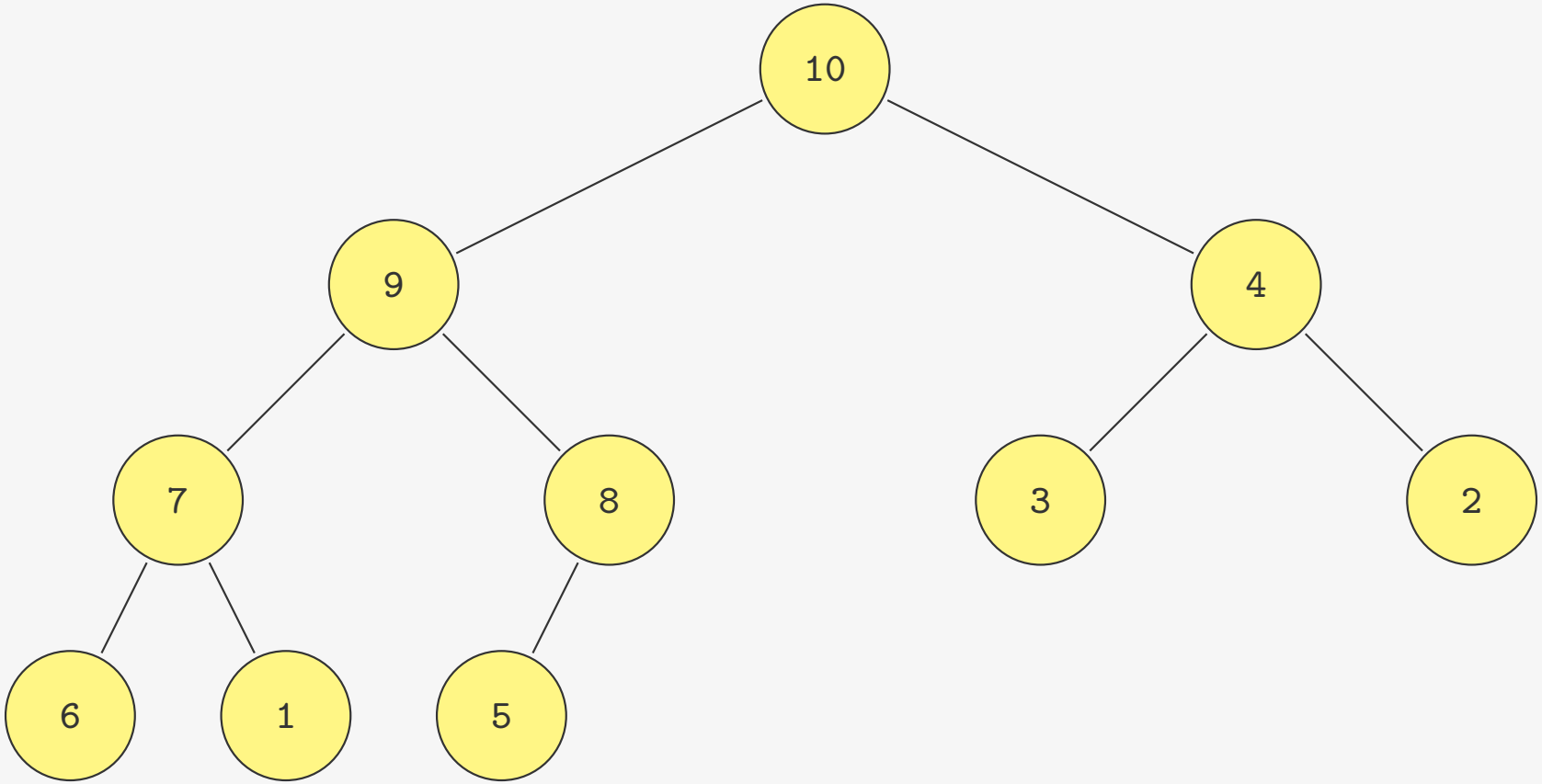

- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

# Extraindo o Máximo




- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

# Extraindo o Máximo




- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

# Extraindo o Máximo

```
 1 Item extrai_maximo(FilaP *fp) {
 2   Item item;
 3   troca(&fp->v[0], &fp->v[fp->n - 1]);
 4   item = fp->v[fp->n - 1];
 5   (fp->n)--;
 6   desce_no_heap(fp, 0);
 7   return item;
 8 }
 9
10 #define FILHO_ESQUERDO(i) (2*i+1)
11
12 void desce_no_heap(FilaP *fp, int k) {
13   int filho = FILHO_ESQUERDO(k);
14   if (filho < fp->n) {
15     if (filho < fp->n - 1 &&
16         fp->v[filho].chave < fp->v[filho+1].chave)
17       filho++;
18     if (fp->v[k].chave < fp->v[filho].chave) {
19       troca(&fp->v[k], &fp->v[filho]);
20       desce_no_heap(fp, filho);
21     }
22   }
23 }
```

# Extraindo o Máximo

```
1  Item extrai_maximo(FilaP *fp) {
2    Item item;
3    troca(&fp->v[0], &fp->v[fp->n - 1]);
4    item = fp->v[fp->n - 1];
5    (fp->n)--;
6    desce_no_heap(fp, 0);
7    return item;
8  }
9
10 #define FILHO_ESQUERDO(i) (2*i+1)
11
12 void desce_no_heap(FilaP *fp, int k) {
13   int filho = FILHO_ESQUERDO(k);
14   if (filho < fp->n) {
15     if (filho < fp->n - 1 &&
16         fp->v[filho].chave < fp->v[filho+1].chave)
17       filho++;
18     if (fp->v[k].chave < fp->v[filho].chave) {
19       troca(&fp->v[k], &fp->v[filho]);
20       desce_no_heap(fp, filho);
21     }
22   }
23 }
```

Tempo de `extrai_maximo`: $\mathrm{O}(\lg n)$

# Mudando a prioridade de um item

Com o que vimos, é fácil mudar a prioridade de um item

# Mudando a prioridade de um item

Com o que vimos, é fácil mudar a prioridade de um item

- Se a prioridade aumentar, precisamos subir arrumando

# Mudando a prioridade de um item

Com o que vimos, é fácil mudar a prioridade de um item

- Se a prioridade aumentar, precisamos subir arrumando
- Se a prioridade diminuir, precisamos descer arrumando

# Mudando a prioridade de um item

Com o que vimos, é fácil mudar a prioridade de um item

- Se a prioridade aumentar, precisamos subir arrumando
- Se a prioridade diminuir, precisamos descer arrumando

```
1 void muda_prioridade(FilaP *fp, int k, int valor) {
2   if (fp->v[k].chave < valor) {
3     fp->v[k].chave = valor;
4     sobe_no_heap(fp, k);
5   } else {
6     fp->v[k].chave = valor;
7     desce_no_heap(fp, k);
8   }
9 }
```

# Mudando a prioridade de um item

Com o que vimos, é fácil mudar a prioridade de um item

- Se a prioridade aumentar, precisamos subir arrumando
- Se a prioridade diminuir, precisamos descer arrumando

```
1 void muda_prioridade(FilaP *fp, int k, int valor) {
2   if (fp->v[k].chave < valor) {
3     fp->v[k].chave = valor;
4     sobe_no_heap(fp, k);
5   } else {
6     fp->v[k].chave = valor;
7     desce_no_heap(fp, k);
8   }
9 }
```

Tempo: $\mathrm{O}(\lg n)$

# Ordenação usando Fila de Prioridades

```
1 void fpsort(Item *v, int l, int r) {
2   int i;
3   FilaP fp;
4   inicializa(&fp, r-l+1);
5   for (i = l; i < r; i++)
6     insere(&fp, v[i]);
7   for (i = r; i >= l; i--)
8     v[i] = extrai_maximo(&fp);
9 }
```

# Ordenação usando Fila de Prioridades

```
1 void fpsort(Item *v, int l, int r) {
2   int i;
3   FilaP fp;
4   inicializa(&fp, r-l+1);
5   for (i = l; i < r; i++)
6     insere(&fp, v[i]);
7   for (i = r; i >= l; i--)
8     v[i] = extrai_maximo(&fp);
9 }
```

Tempo: $\mathrm{O}(n \lg n)$

# Ordenação usando Fila de Prioridades

```
1 void fpsort(Item *v, int l, int r) {
2   int i;
3   FilaP fp;
4   inicializa(&fp, r-l+1);
5   for (i = l; i < r; i++)
6     insere(&fp, v[i]);
7   for (i = r; i >= l; i--)
8     v[i] = extrai_maximo(&fp);
9 }
```

Tempo: $\mathrm{O}(n \lg n)$

- Estamos usando espaço adicional, mas não precisamos...

# Ordenação usando Fila de Prioridades

```
1 void fpsort(Item *v, int l, int r) {
2   int i;
3   FilaP fp;
4   inicializa(&fp, r-l+1);
5   for (i = l; i < r; i++)
6     insere(&fp, v[i]);
7   for (i = r; i >= l; i--)
8     v[i] = extrai_maximo(&fp);
9 }
```

Tempo: $\mathrm{O}(n \lg n)$

- Estamos usando espaço adicional, mas não precisamos...
- Perdemos tempo para copiar do vetor para o heap

# Ordenação usando Fila de Prioridades

```
1 void fpsort(Item *v, int l, int r) {
2   int i;
3   FilaP fp;
4   inicializa(&fp, r-l+1);
5   for (i = l; i < r; i++)
6     insere(&fp, v[i]);
7   for (i = r; i >= l; i--)
8     v[i] = extrai_maximo(&fp);
9 }
```

Tempo: $\mathrm{O}(n \lg n)$

- Estamos usando espaço adicional, mas não precisamos...
- Perdemos tempo para copiar do vetor para o heap
- Podemos transformar um vetor em um heap rapidamente

# Ordenação usando Fila de Prioridades

```
1 void fpsort(Item *v, int l, int r) {
2   int i;
3   FilaP fp;
4   inicializa(&fp, r-l+1);
5   for (i = l; i < r; i++)
6     insere(&fp, v[i]);
7   for (i = r; i >= l; i--)
8     v[i] = extrai_maximo(&fp);
9 }
```

Tempo: $\mathrm{O}(n \lg n)$

- Estamos usando espaço adicional, mas não precisamos...
- Perdemos tempo para copiar do vetor para o heap
- Podemos transformar um vetor em um heap rapidamente
  - Mais rápido do que fazer $n$ inserções

# Transformando um vetor em um heap

# Transformando um vetor em um heap

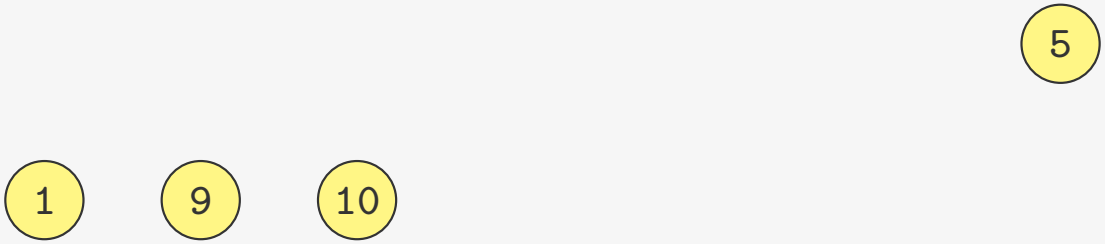
| 4 | 8 | 2 | 7 | 6 | 3 | 5 | 1 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

# Transformando um vetor em um heap

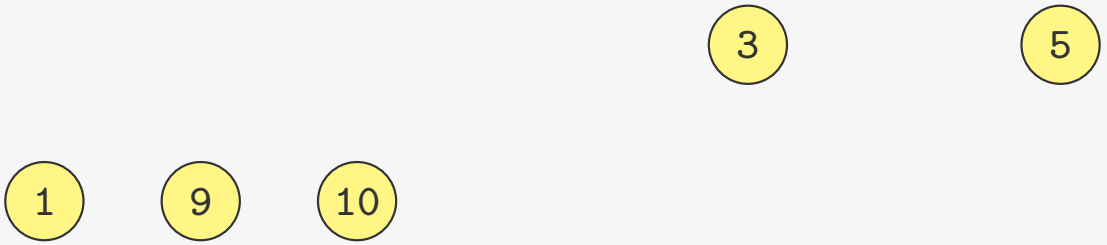

# Transformando um vetor em um heap

| 4 | 8 | 2 | 7 | 6 | 3 | 5 | 1 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

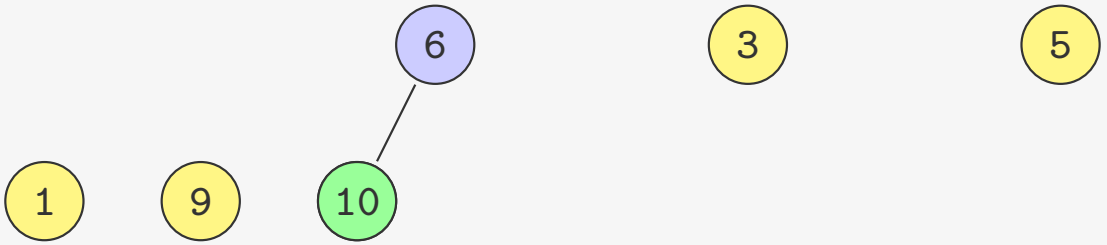
# Transformando um vetor em um heap

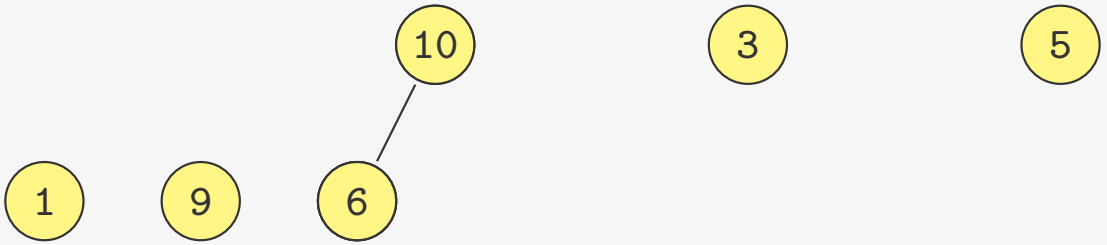
# Transformando um vetor em um heap

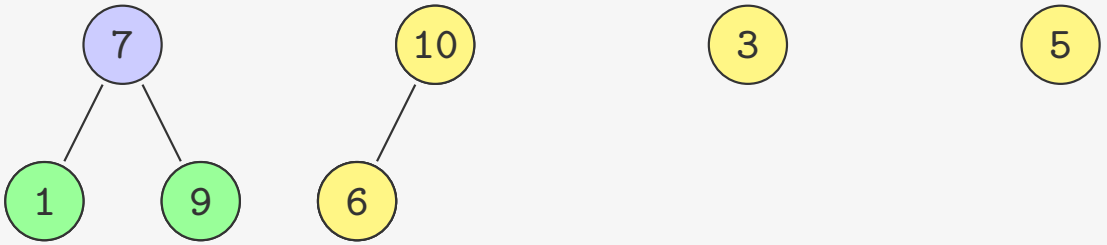
# Transformando um vetor em um heap

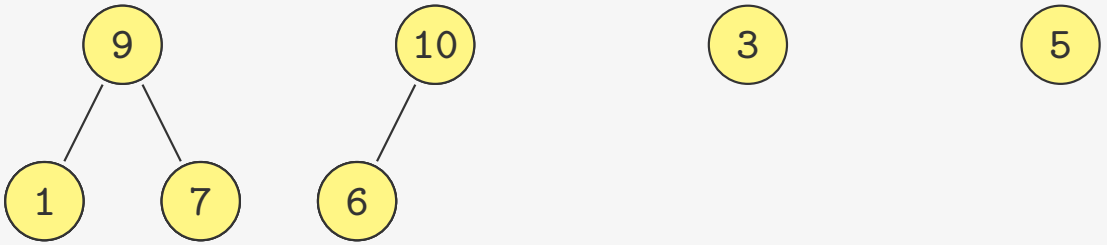
# Transformando um vetor em um heap

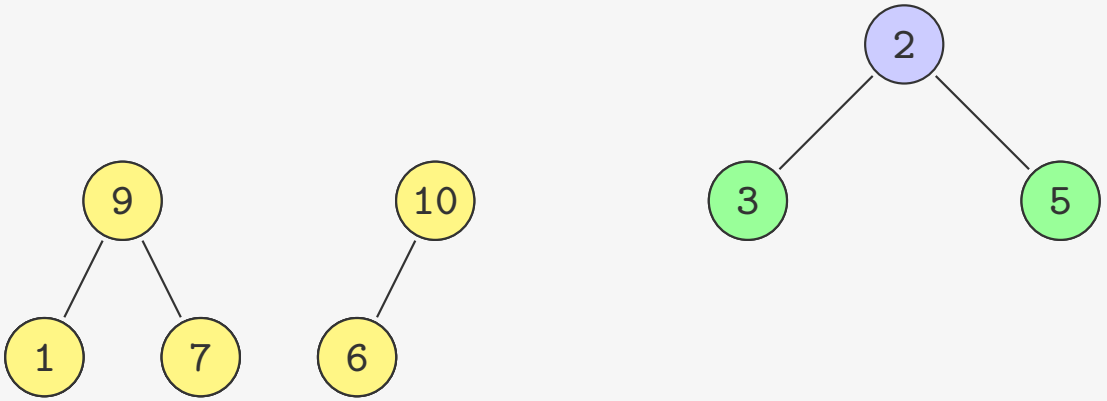
# Transformando um vetor em um heap

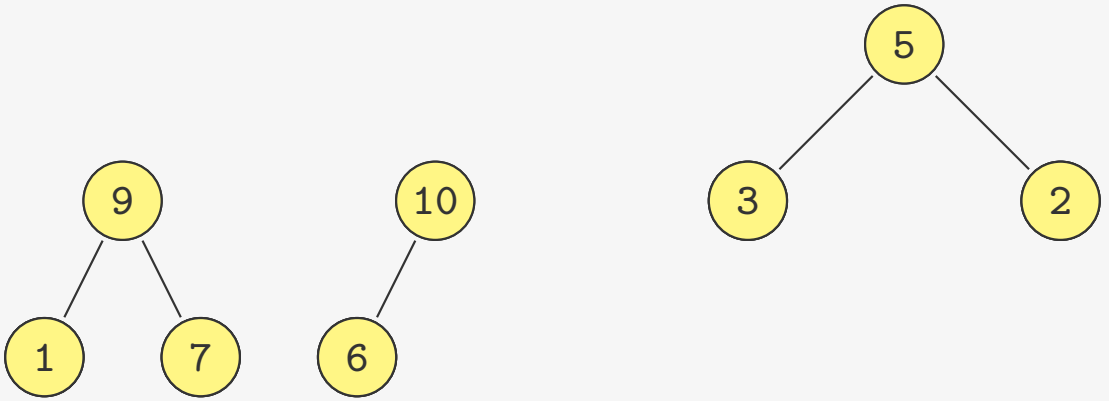
# Transformando um vetor em um heap

# Transformando um vetor em um heap

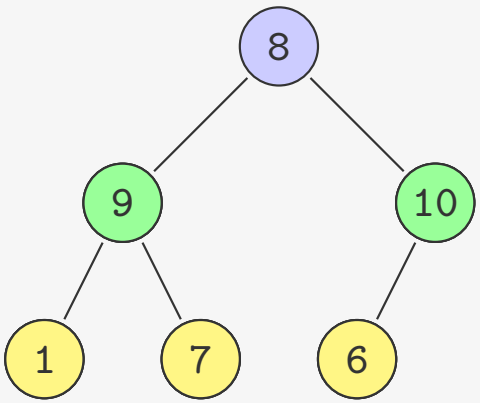



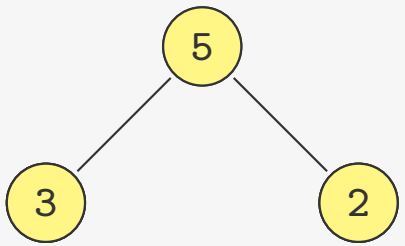
# Transformando um vetor em um heap

# Transformando um vetor em um heap

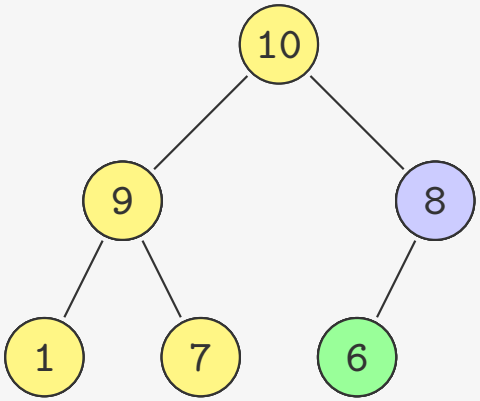
| 4 | 8 | 2 | 7 | 6 | 3 | 5 | 1 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

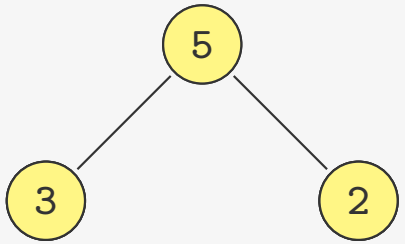


# Transformando um vetor em um heap

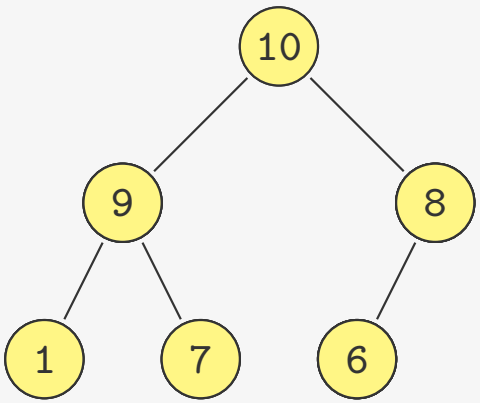
| 4 | 8 | 2 | 7 | 6 | 3 | 5 | 1 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

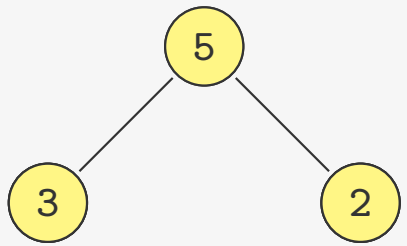


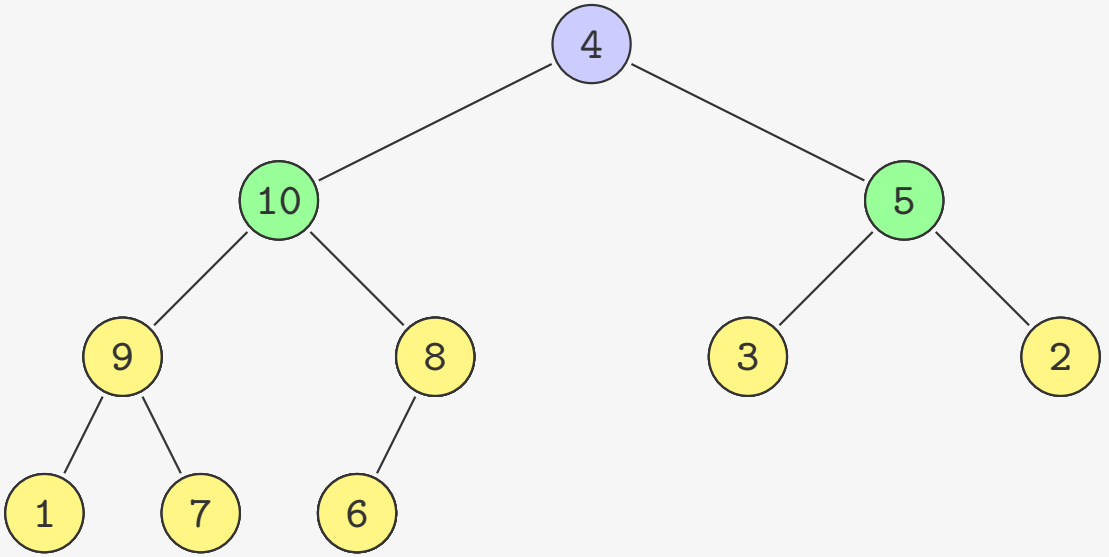
# Transformando um vetor em um heap

| 4 | 8 | 2 | 7 | 6 | 3 | 5 | 1 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|



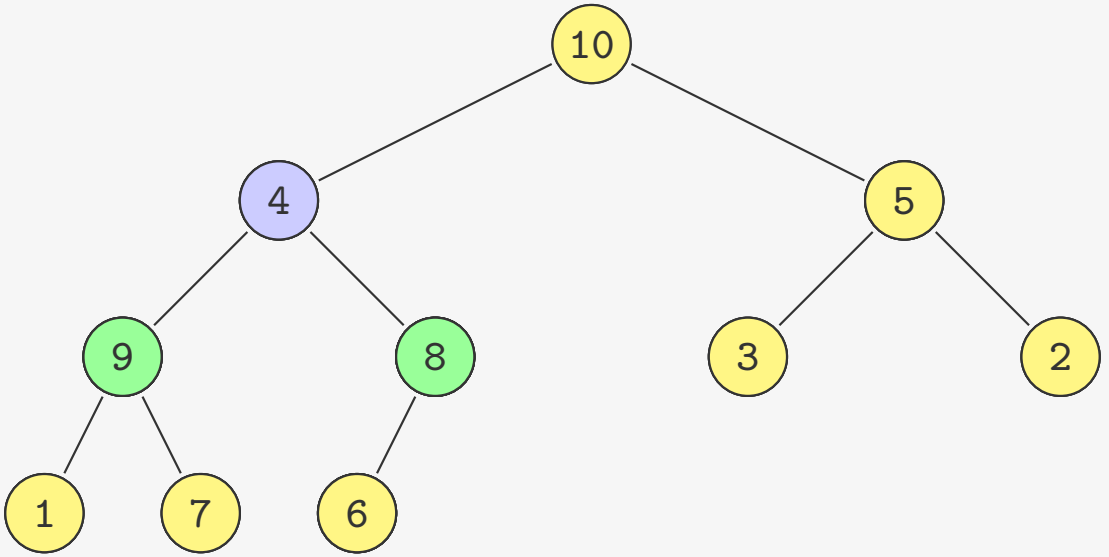



# Transformando um vetor em um heap

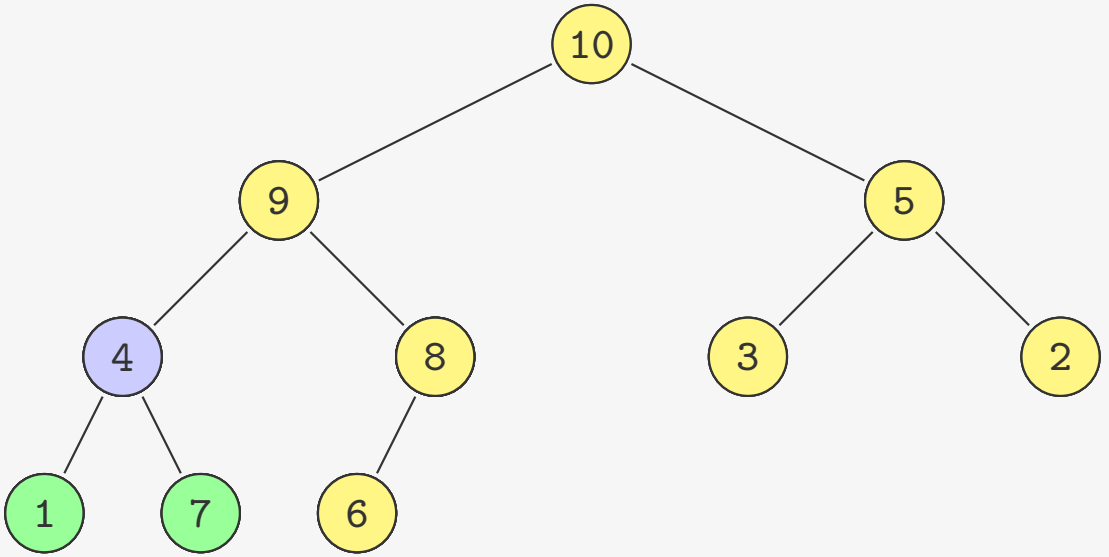



# Transformando um vetor em um heap

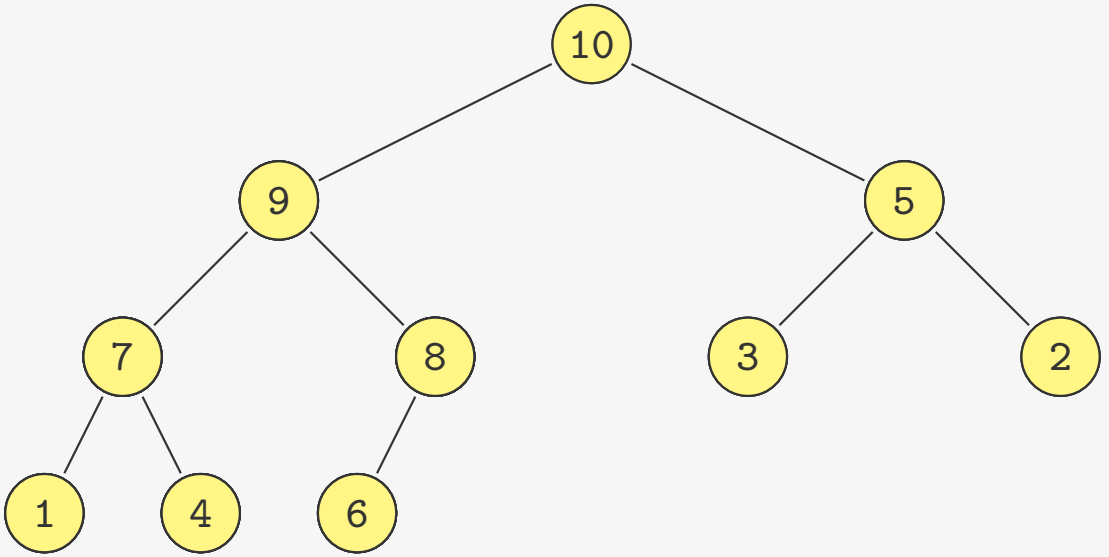
| 4 | 8 | 2 | 7 | 6 | 3 | 5 | 1 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|



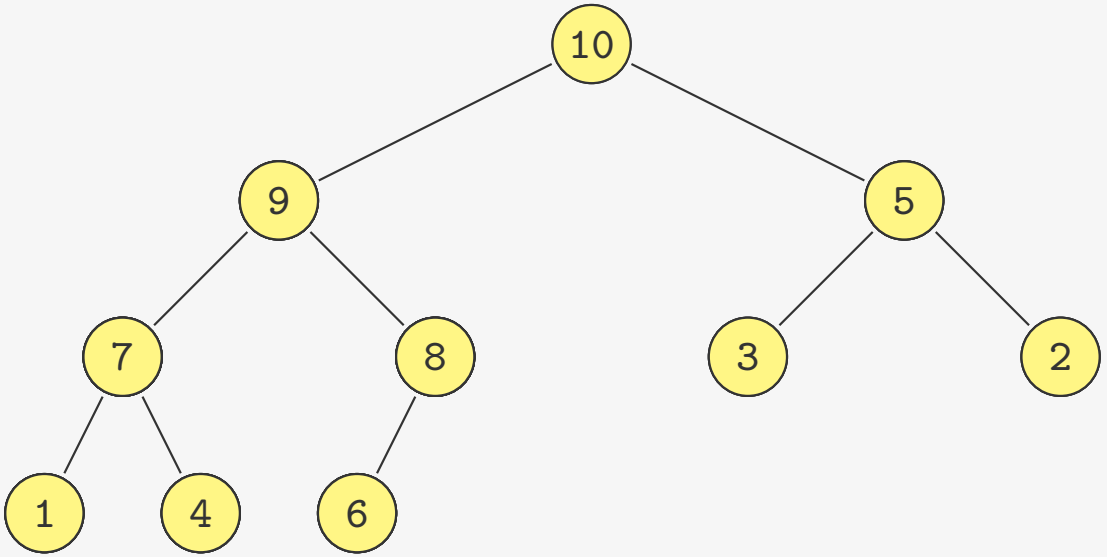
# Transformando um vetor em um heap

# Transformando um vetor em um heap

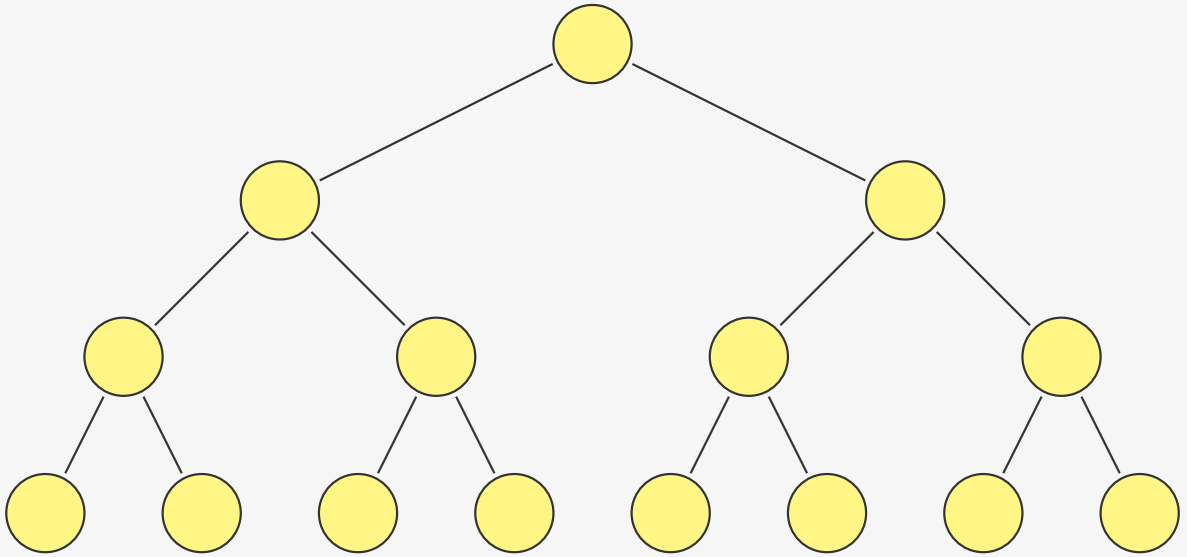



# Transformando um vetor em um heap

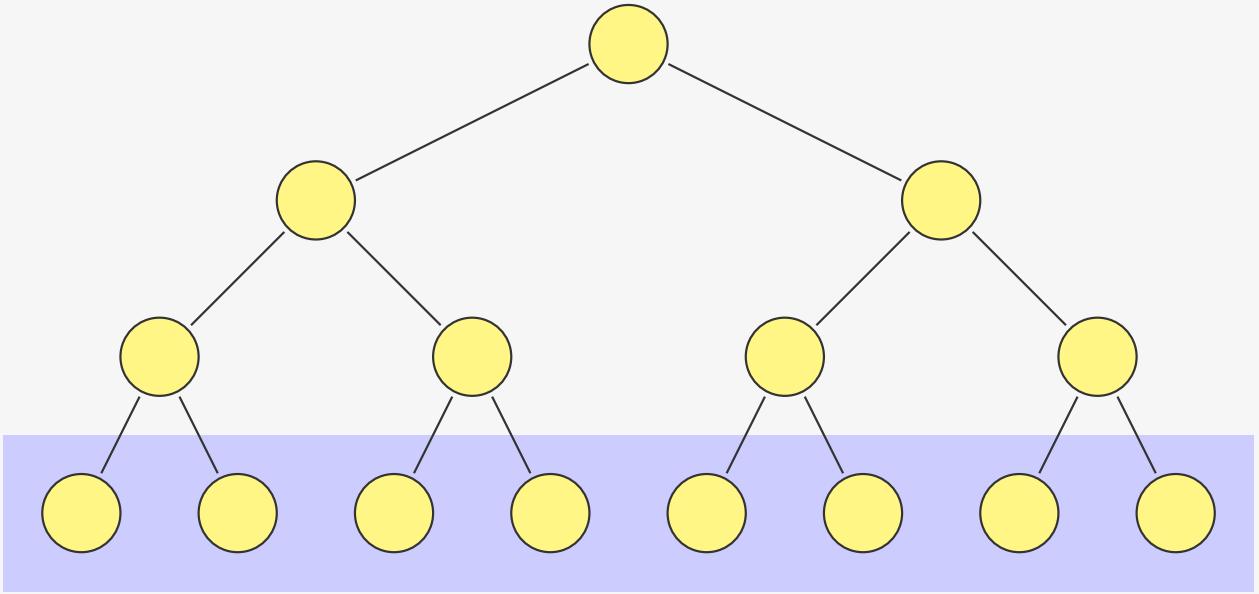




Quanto tempo demora?

# Tempo da construção para $n = 2^k$

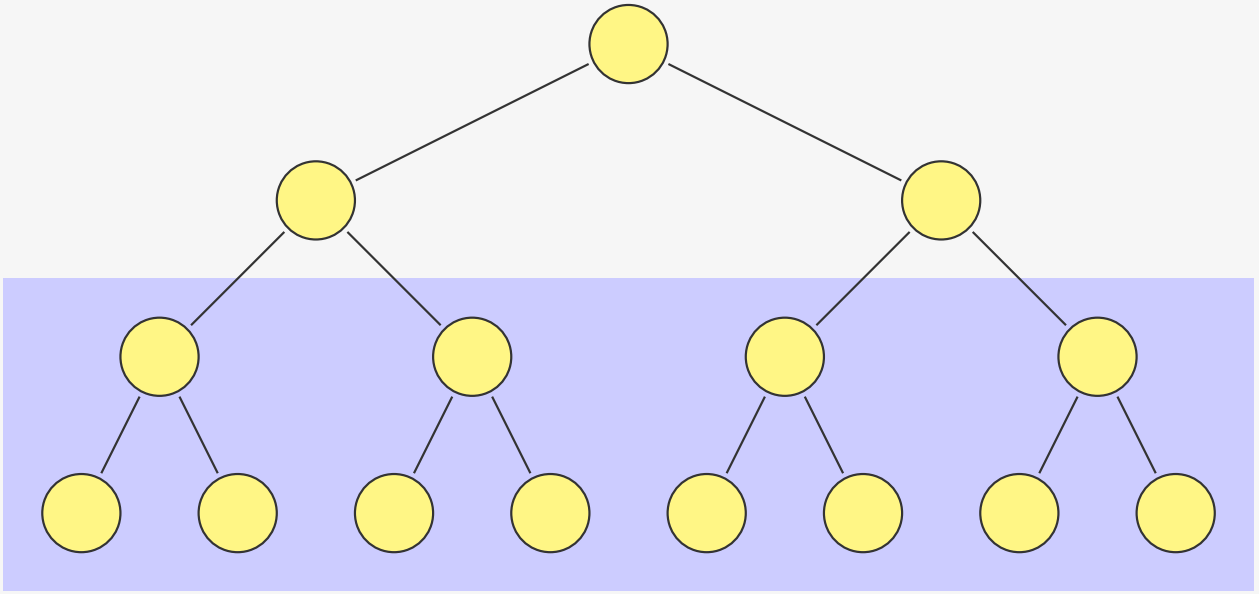


# Tempo da construção para $n = 2^k$



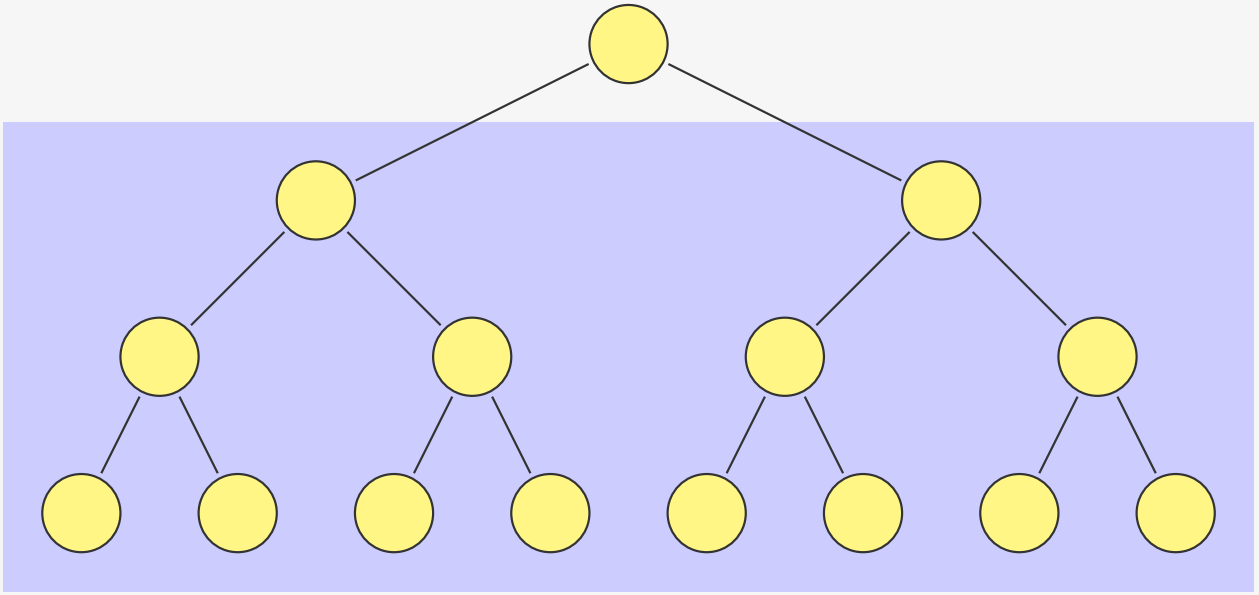
- Temos $n/2 = 2^{k-1}$ heaps de altura $1$

# Tempo da construção para $n = 2^k$



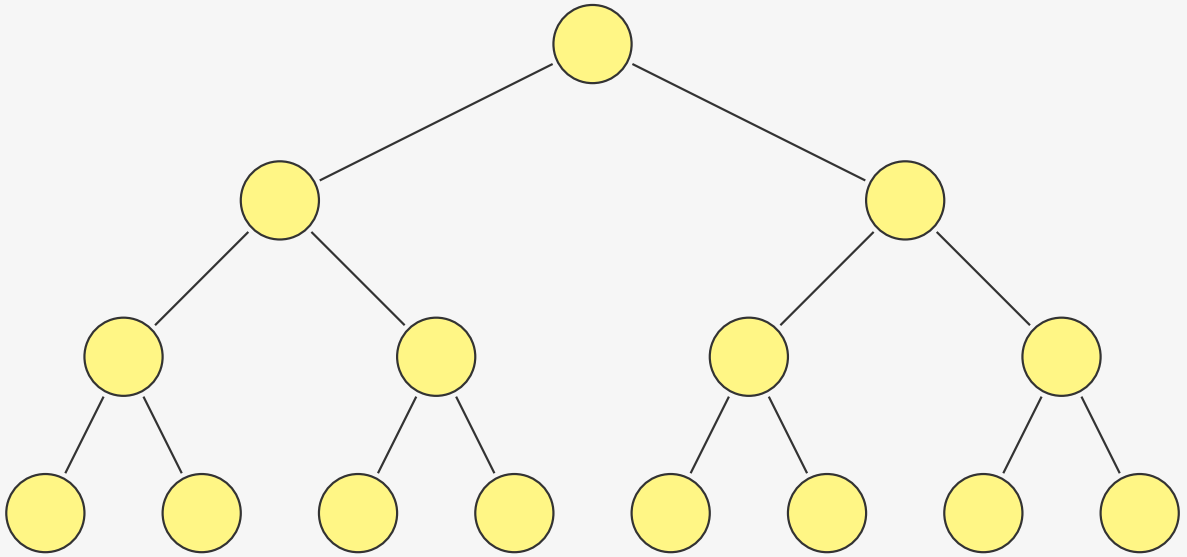
- Temos $n/2 = 2^{k-1}$ heaps de altura $1$
- Temos $n/4 = 2^{k-2}$ heaps de altura $2$

# Tempo da construção para $n = 2^k$



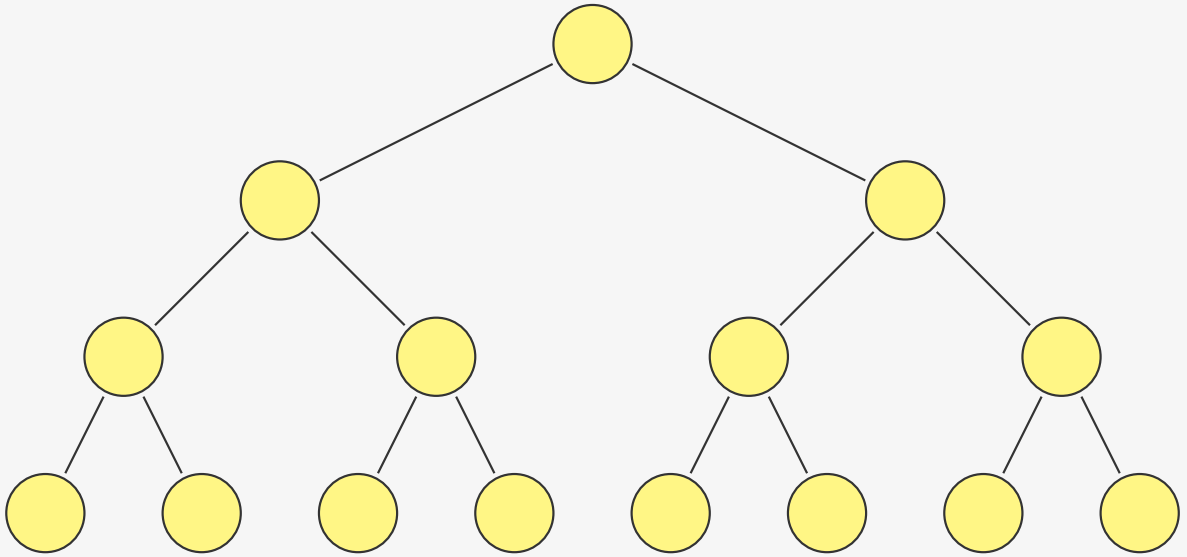
- Temos $n/2 = 2^{k-1}$ heaps de altura $1$
- Temos $n/4 = 2^{k-2}$ heaps de altura $2$
- Temos $n/2^{h+1} = 2^{k-h-1}$ heaps de altura $h$

# Tempo da construção para $n = 2^k$



- Temos $n/2 = 2^{k-1}$ heaps de altura $1$
- Temos $n/4 = 2^{k-2}$ heaps de altura $2$
- Temos $n/2^{h+1} = 2^{k-h-1}$ heaps de altura $h$
- Cada heap de altura $h$ consome tempo $c \cdot h$

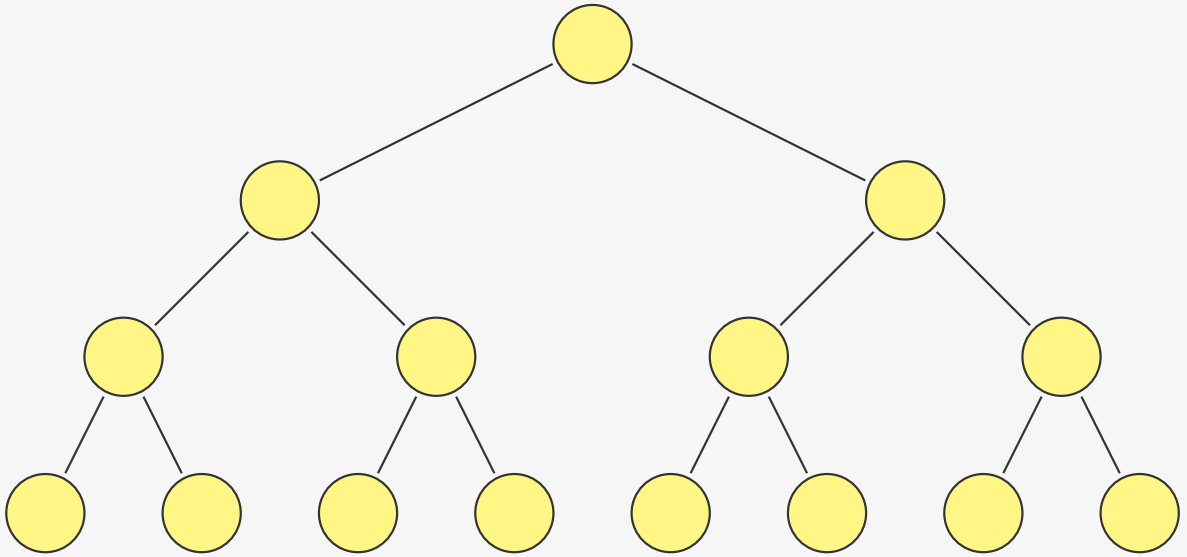
# Tempo da construção para $n = 2^k$



- Temos $n/2 = 2^{k-1}$ heaps de altura $1$
- Temos $n/4 = 2^{k-2}$ heaps de altura $2$
- Temos $n/2^{h+1} = 2^{k-h-1}$ heaps de altura $h$
- Cada heap de altura $h$ consome tempo $c \cdot h$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1}$$

# Tempo da construção para $n = 2^k$



- Temos $n/2 = 2^{k-1}$ heaps de altura $1$
- Temos $n/4 = 2^{k-2}$ heaps de altura $2$
- Temos $n/2^{h+1} = 2^{k-h-1}$ heaps de altura $h$
- Cada heap de altura $h$ consome tempo $c \cdot h$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

# Tempo da construção para $n = 2^k$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

# Tempo da construção para $n = 2^k$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\sum_{h=1}^{k-1} \frac{h}{2^h} =$$

# Tempo da construção para $n = 2^k$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \; & \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \\
& + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \\
& + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \\
& \cdots + \frac{1}{2^{k-1}}
\end{aligned}$$

# Tempo da construção para $n = 2^k$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = & \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} = 1 - \frac{1}{2^{k-1}} \\
& + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \\
& + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} \\
& \cdots + \frac{1}{2^{k-1}}
\end{aligned}$$

# Tempo da construção para $n = 2^k$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= 1 - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{2} - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} & \\
\cdots + \frac{1}{2^{k-1}} &
\end{aligned}$$

# Tempo da construção para $n = 2^k$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= 1 - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{2} - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{4} - \frac{1}{2^{k-1}} \\
\cdots + \frac{1}{2^{k-1}} &
\end{aligned}$$

# Tempo da construção para $n = 2^k$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= 1 - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{2} - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{4} - \frac{1}{2^{k-1}} \\
\cdots + \frac{1}{2^{k-1}} &= \frac{1}{2^{r-1}} - \frac{1}{2^{k-1}}
\end{aligned}$$

# Tempo da construção para $n = 2^k$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= 1 - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{2} - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{4} - \frac{1}{2^{k-1}} \\
\cdots + \frac{1}{2^{k-1}} &= \frac{1}{2^{r-1}} - \frac{1}{2^{k-1}}
\end{aligned}$$

Ou seja,

$$c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

# Tempo da construção para $n = 2^k$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= 1 - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{2} - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{4} - \frac{1}{2^{k-1}} \\
\cdots + \frac{1}{2^{k-1}} &= \frac{1}{2^{r-1}} - \frac{1}{2^{k-1}}
\end{aligned}$$

Ou seja,

$$c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h} \leq c \cdot 2^{k-1} \cdot 2$$

# Tempo da construção para $n = 2^k$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= 1 - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{2} - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{4} - \frac{1}{2^{k-1}} \\
\cdots + \frac{1}{2^{k-1}} &= \frac{1}{2^{r-1}} - \frac{1}{2^{k-1}}
\end{aligned}$$

Ou seja,

$$c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h} \leq c \cdot 2^{k-1} \cdot 2 = \mathrm{O}(2^k)$$

# Tempo da construção para $n = 2^k$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h-1} = c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned}
\sum_{h=1}^{k-1} \frac{h}{2^h} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= 1 - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{2} - \frac{1}{2^{k-1}} \\
+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} &= \frac{1}{4} - \frac{1}{2^{k-1}} \\
\cdots + \frac{1}{2^{k-1}} &= \frac{1}{2^{r-1}} - \frac{1}{2^{k-1}}
\end{aligned}$$

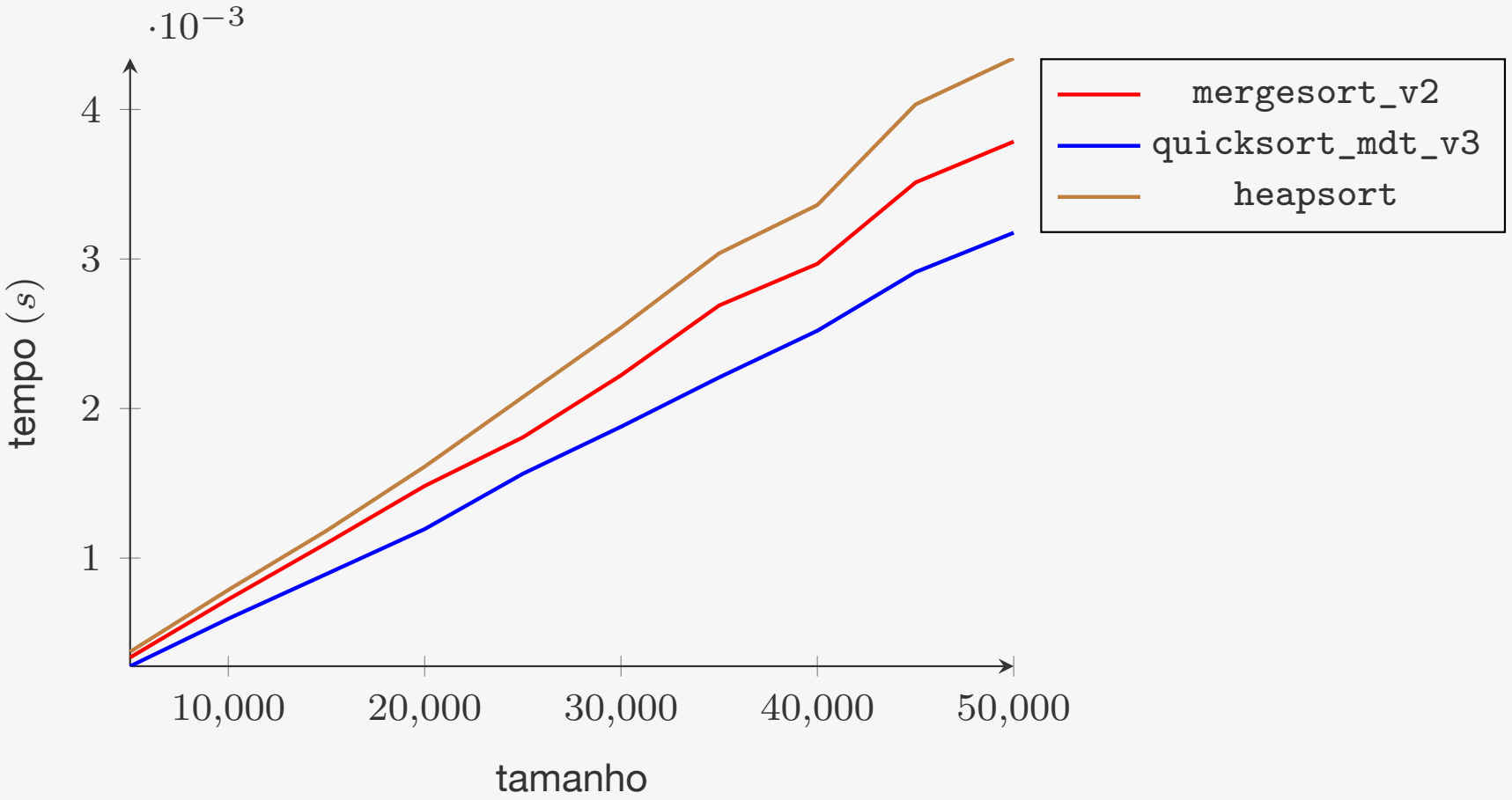
Ou seja,

$$c \cdot 2^{k-1} \sum_{h=1}^{k-1} \frac{h}{2^h} \leq c \cdot 2^{k-1} \cdot 2 = \mathrm{O}(2^k) = \mathrm{O}(n)$$

# Heapsort

```
 1 void heapsort(Item *v, int l, int r) {
 2   int k;
 3   FilaP fp;
 4   fp.v = &v[l];
 5   fp.n = r-l+1;
 6   for (k = fp.n/2; k >= 1; k--) /* transforma em heap */
 7     desce_no_heap(&fp, k);
 8   while (fp.n > 1) { /* extrai o máximo */
 9     troca(&fp.v[0], &fp.v[fp.n - 1]);
10     (fp.n)--;
11     desce_no_heap(&fp, 0);
12   }
13 }
```

# Comparação com QuickSort e MergeSort




- O HeapSort é mais lento que o MergeSort
- Mas não precisa de um vetor auxiliar...

# Exercícios

1. Faça uma implementação de uma lista de prioridades usando uma lista ligada.
2. Reflita: um vetor ordenado de forma decrescente é um heap?
3. Crie versão iterativas de `desce_no_heap` e `sobe_no_heap`
   - Em `sobe_no_heap` trocamos `k` com `PAI(k)`, `PAI(k)` com `PAI(PAI(k))` e assim por diante. Algo similar acontece com `desce_no_heap`. Modifique as versões iterativas das duas funções para diminuir o número de atribuições (como feito no InsertionSort).