

MC-202 — Unidade 13

Filas de Prioridade e HeapSort

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

Universidade Estadual de Campinas

1º semestre/2018

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     int max = r;
3     for (j = r; 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     int max = r;
3     for (j = r; 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 $r - l$ 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 $r - l$ 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 $r - l$ 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 $r - l$ 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 $r - l$ 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 $r - l$ 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 $r - l$ 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 $r - l$ 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

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

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 $r - l$ 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Tal algoritmo levaria tempo:

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

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

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

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

Tal algoritmo levaria tempo:

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

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 typedef struct {
2     char nome[20];
3     int chave;
4 } Item;
```

Fila de Prioridade (usando vetores) - TAD

```
1 typedef struct {
2     char nome[20];
3     int chave;
4 } Item;
5
6 typedef struct {
7     Item *v;
8     int n, tamanho;
9 } FP;
```

Fila de Prioridade (usando vetores) - TAD

```
1 typedef struct {
2     char nome[20];
3     int chave;
4 } Item;
5
6 typedef struct {
7     Item *v;
8     int n, tamanho;
9 } FP;
10
11 typedef FP * p_fp
```

Fila de Prioridade (usando vetores) - TAD

```
1 typedef struct {
2     char nome[20];
3     int chave;
4 } Item;
5
6 typedef struct {
7     Item *v;
8     int n, tamanho;
9 } FP;
10
11 typedef FP * p_fp
12
13 p_fp criar_filaprio(int tam);
14 void insere(p_fp fprio, Item item);
15 Item extrai_maximo(p_fp fprio);
16 int vazia(p_fp fprio);
17 int cheia(p_fp fprio);
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {  
2   p_fp fprio = malloc(sizeof(FP));
```


Operações Básicas

```
1 p_fp criar_filaprio(int tam) {  
2     p_fp fprio = malloc(sizeof(FP));  
3     fprio->v = malloc(tam * sizeof(Item));
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2   p_fp fprio = malloc(sizeof(FP));
3   fprio->v = malloc(tam * sizeof(Item));
4   fprio->n = 0;
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2   p_fp fprio = malloc(sizeof(FP));
3   fprio->v = malloc(tam * sizeof(Item));
4   fprio->n = 0;
5   fprio->tamanho = tam;
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2   p_fp fprio = malloc(sizeof(FP));
3   fprio->v = malloc(tam * sizeof(Item));
4   fprio->n = 0;
5   fprio->tamanho = tam;
6   return fprio;
7 }
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2     p_fp fprio = malloc(sizeof(FP));
3     fprio->v = malloc(tam * sizeof(Item));
4     fprio->n = 0;
5     fprio->tamanho = tam;
6     return fprio;
7 }
```

```
1 void insere(p_fp fprio, Item item) {
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2   p_fp fprio = malloc(sizeof(FP));
3   fprio->v = malloc(tam * sizeof(Item));
4   fprio->n = 0;
5   fprio->tamanho = tam;
6   return fprio;
7 }
```

```
1 void insere(p_fp fprio, Item item) {
2   fprio->v[fprio->n] = item;
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2   p_fp fprio = malloc(sizeof(FP));
3   fprio->v = malloc(tam * sizeof(Item));
4   fprio->n = 0;
5   fprio->tamanho = tam;
6   return fprio;
7 }
```

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

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2   p_fp fprio = malloc(sizeof(FP));
3   fprio->v = malloc(tam * sizeof(Item));
4   fprio->n = 0;
5   fprio->tamanho = tam;
6   return fprio;
7 }
```

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

```
1 Item extrai_maximo(p_fp fprio) {
```


Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2   p_fp fprio = malloc(sizeof(FP));
3   fprio->v = malloc(tam * sizeof(Item));
4   fprio->n = 0;
5   fprio->tamanho = tam;
6   return fprio;
7 }
```

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

```
1 Item extrai_maximo(p_fp fprio) {
2   int j, max = 0;
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2   p_fp fprio = malloc(sizeof(FP));
3   fprio->v = malloc(tam * sizeof(Item));
4   fprio->n = 0;
5   fprio->tamanho = tam;
6   return fprio;
7 }
```

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

```
1 Item extrai_maximo(p_fp fprio) {
2   int j, max = 0;
3   for (j = 1; j < fprio->n; j++)
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2     p_fp fprio = malloc(sizeof(FP));
3     fprio->v = malloc(tam * sizeof(Item));
4     fprio->n = 0;
5     fprio->tamanho = tam;
6     return fprio;
7 }

1 void insere(p_fp fprio, Item item) {
2     fprio->v[fprio->n] = item;
3     fprio->n++;
4 }

1 Item extrai_maximo(p_fp fprio) {
2     int j, max = 0;
3     for (j = 1; j < fprio->n; j++)
4         if (fprio->v[max].chave < fprio->v[j].chave)
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2     p_fp fprio = malloc(sizeof(FP));
3     fprio->v = malloc(tam * sizeof(Item));
4     fprio->n = 0;
5     fprio->tamanho = tam;
6     return fprio;
7 }

1 void insere(p_fp fprio, Item item) {
2     fprio->v[fprio->n] = item;
3     fprio->n++;
4 }

1 Item extrai_maximo(p_fp fprio) {
2     int j, max = 0;
3     for (j = 1; j < fprio->n; j++)
4         if (fprio->v[max].chave < fprio->v[j].chave)
5             max = j;
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2     p_fp fprio = malloc(sizeof(FP));
3     fprio->v = malloc(tam * sizeof(Item));
4     fprio->n = 0;
5     fprio->tamanho = tam;
6     return fprio;
7 }

1 void insere(p_fp fprio, Item item) {
2     fprio->v[fprio->n] = item;
3     fprio->n++;
4 }

1 Item extrai_maximo(p_fp fprio) {
2     int j, max = 0;
3     for (j = 1; j < fprio->n; j++)
4         if (fprio->v[max].chave < fprio->v[j].chave)
5             max = j;
6     troca(&(fprio->v[max]), &(fprio->v[fprio->n-1]));
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2     p_fp fprio = malloc(sizeof(FP));
3     fprio->v = malloc(tam * sizeof(Item));
4     fprio->n = 0;
5     fprio->tamanho = tam;
6     return fprio;
7 }

1 void insere(p_fp fprio, Item item) {
2     fprio->v[fprio->n] = item;
3     fprio->n++;
4 }

1 Item extrai_maximo(p_fp fprio) {
2     int j, max = 0;
3     for (j = 1; j < fprio->n; j++)
4         if (fprio->v[max].chave < fprio->v[j].chave)
5             max = j;
6     troca(&(fprio->v[max]), &(fprio->v[fprio->n-1]));
7     fprio->n--;
```

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2     p_fp fprio = malloc(sizeof(FP));
3     fprio->v = malloc(tam * sizeof(Item));
4     fprio->n = 0;
5     fprio->tamanho = tam;
6     return fprio;
7 }

1 void insere(p_fp fprio, Item item) {
2     fprio->v[fprio->n] = item;
3     fprio->n++;
4 }

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

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2   p_fp fprio = malloc(sizeof(FP));
3   fprio->v = malloc(tam * sizeof(Item));
4   fprio->n = 0;
5   fprio->tamanho = tam;
6   return fprio;
7 }

1 void insere(p_fp fprio, Item item) {
2   fprio->v[fprio->n] = item;
3   fprio->n++;
4 }

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

Inserer em $O(1)$, extrair o máximo em $O(n)$

Operações Básicas

```
1 p_fp criar_filaprio(int tam) {
2   p_fp fprio = malloc(sizeof(FP));
3   fprio->v = malloc(tam * sizeof(Item));
4   fprio->n = 0;
5   fprio->tamanho = tam;
6   return fprio;
7 }
```

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

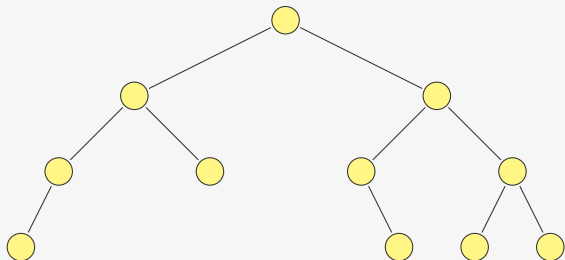
```
1 Item extrai_maximo(p_fp fprio) {
2   int j, max = 0;
3   for (j = 1; j < fprio->n; j++)
4     if (fprio->v[max].chave < fprio->v[j].chave)
5       max = j;
6   troca(&(fprio->v[max]), &(fprio->v[fprio->n-1]));
7   fprio->n--;
8   return fprio->v[fprio->n];
9 }
```

Insere em $O(1)$, extrai o máximo em $O(n)$

- Se mantiver o vetor ordenado, os tempos se invertem

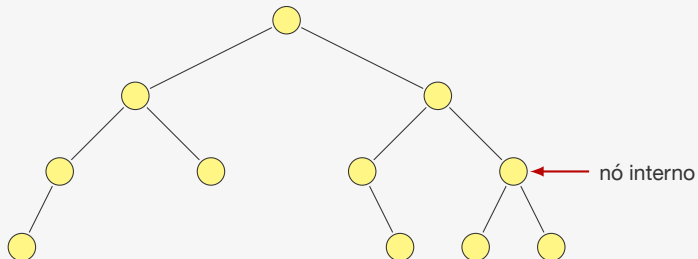
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



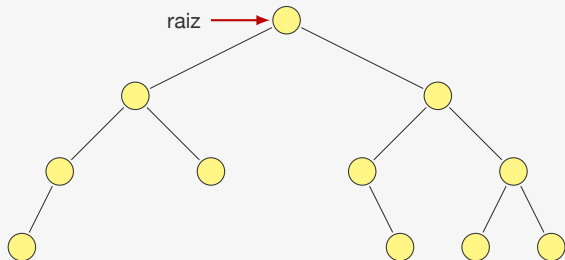
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



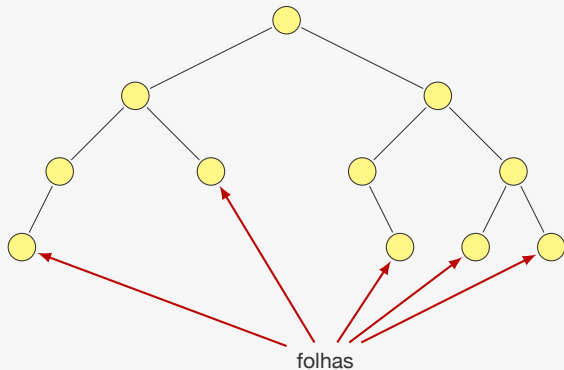
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



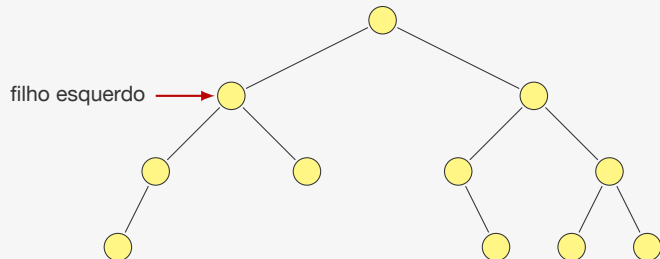
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



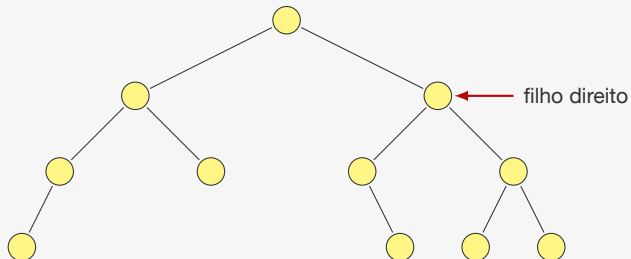
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



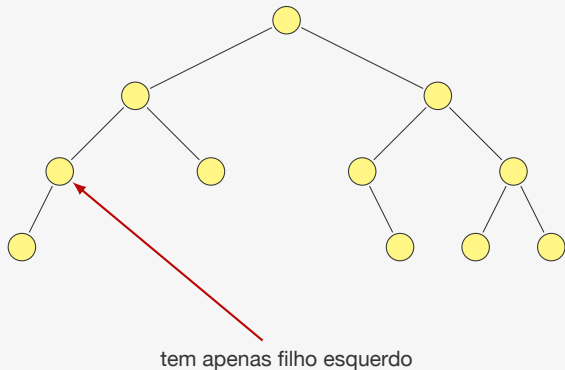
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



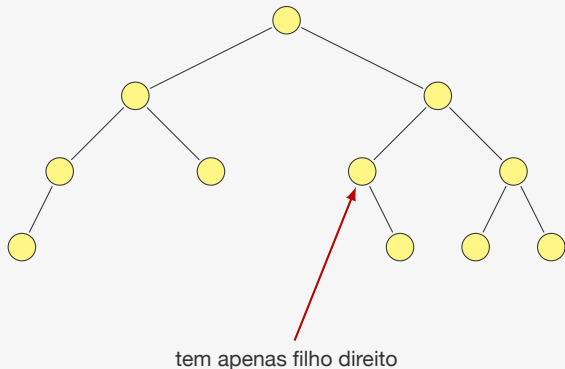
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



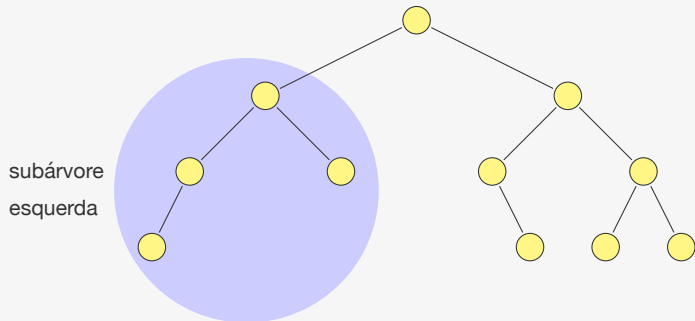
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



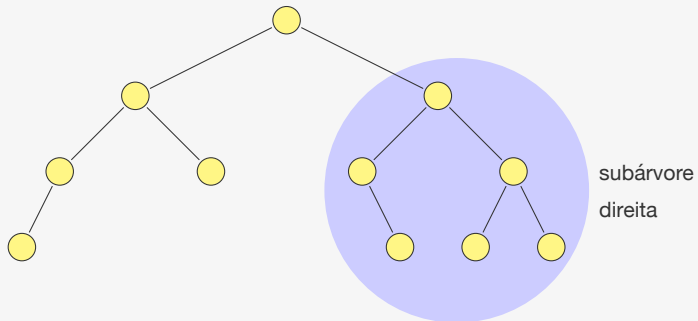
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



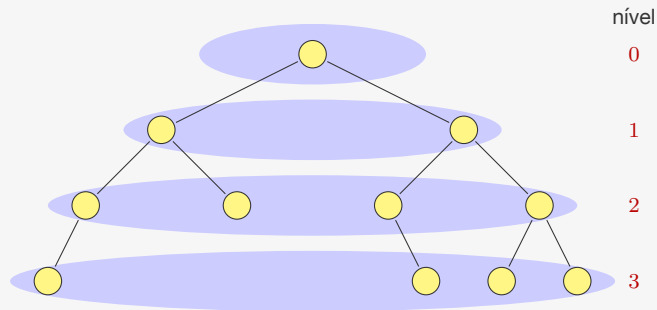
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



Interlúdio - Árvores Binárias

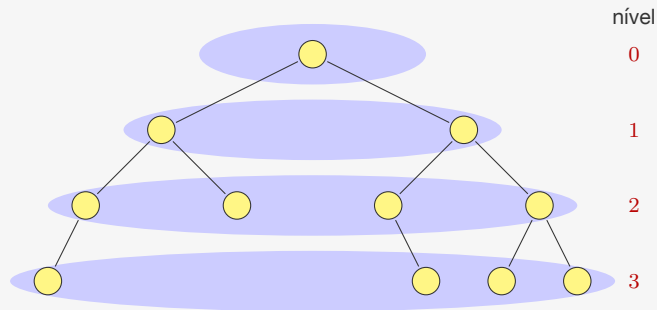
Exemplo de uma árvore binária:



Uma árvore binária é:

Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:

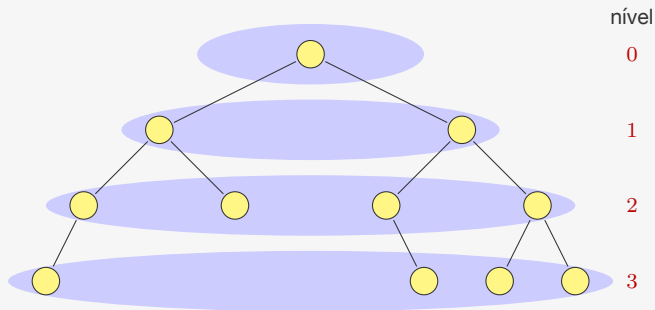


Uma árvore binária é:

- 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:

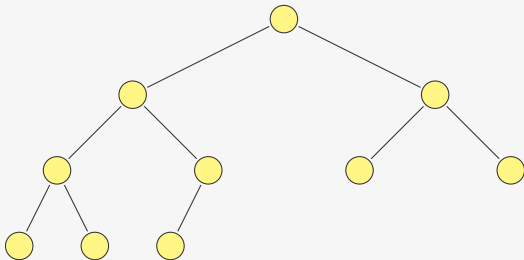
- Todos os níveis exceto o último estão cheios
- No último nível, todos os nós estão o mais a esquerda possível

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:

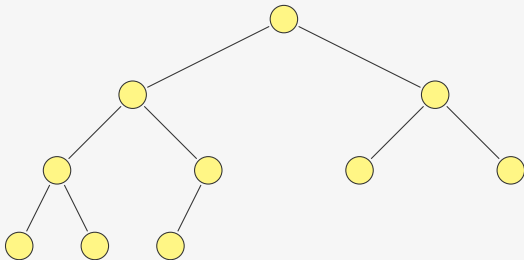


Árvores Binárias Completas

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:



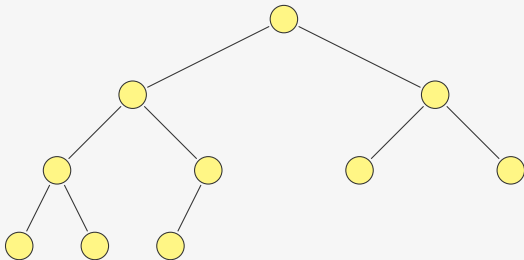
Uma árvore binária completa de n nós tem quantos níveis?

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

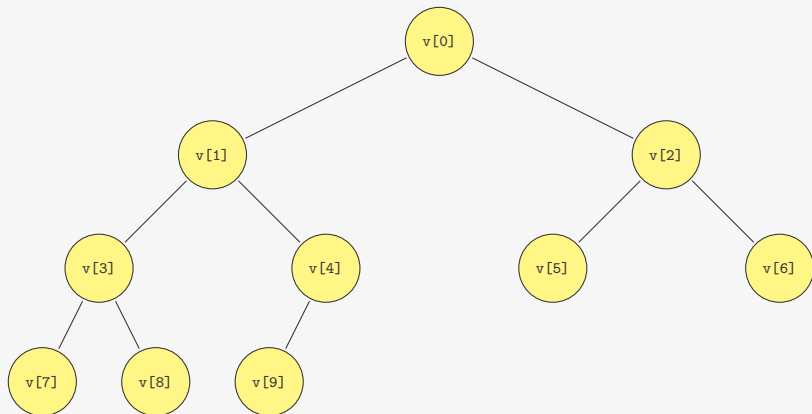
Exemplo:



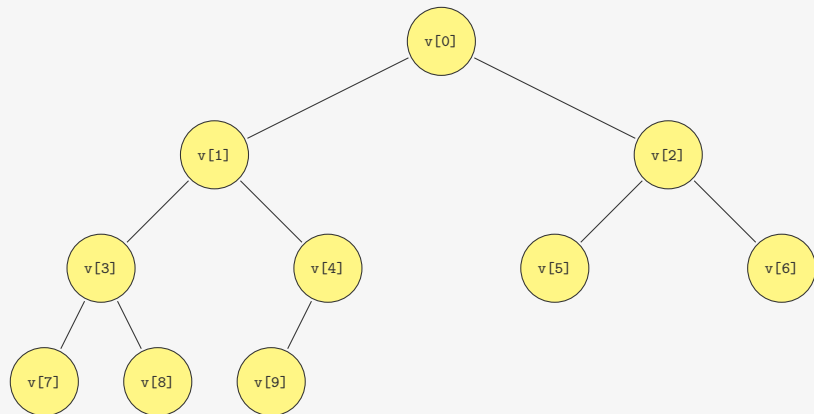
Uma árvore binária completa de n nós tem quantos níveis?

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

Árvores Binárias Completas e Vetores

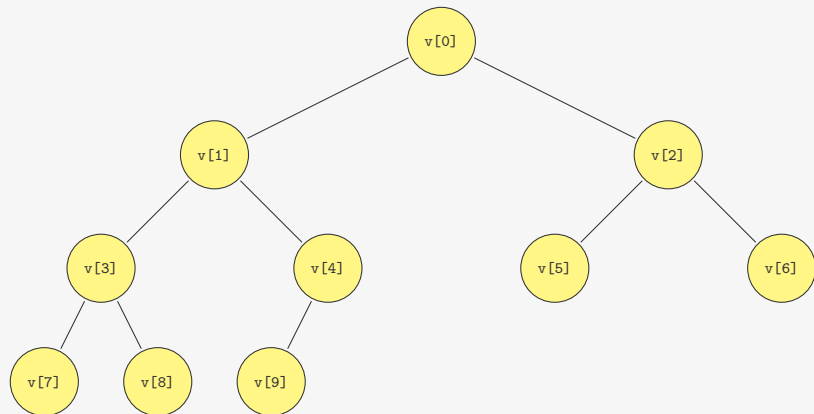


Árvores Binárias Completas e Vetores



Em relação a $v[i]$:

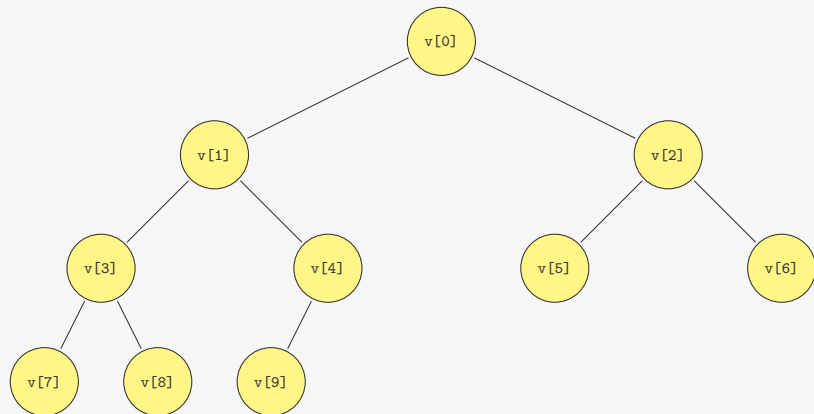
Árvores Binárias Completas e Vetores



Em relação a $v[i]$:

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

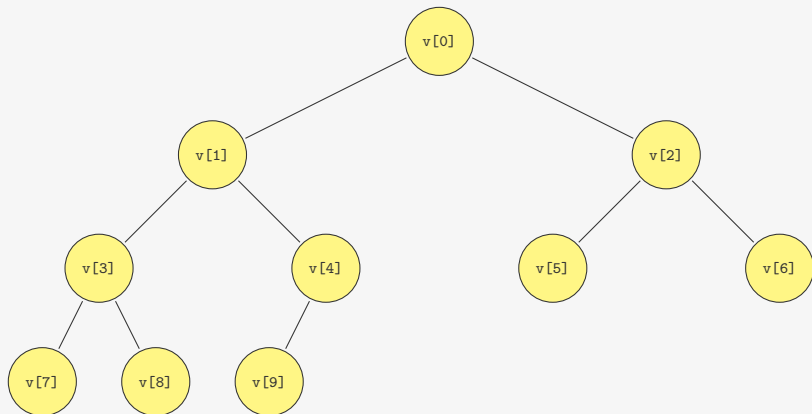
Árvores Binárias Completas e Vetores



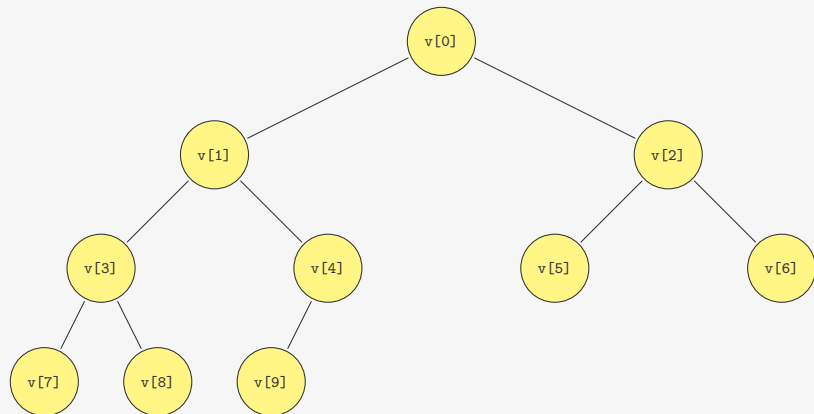
Em relação a $v[i]$:

- 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

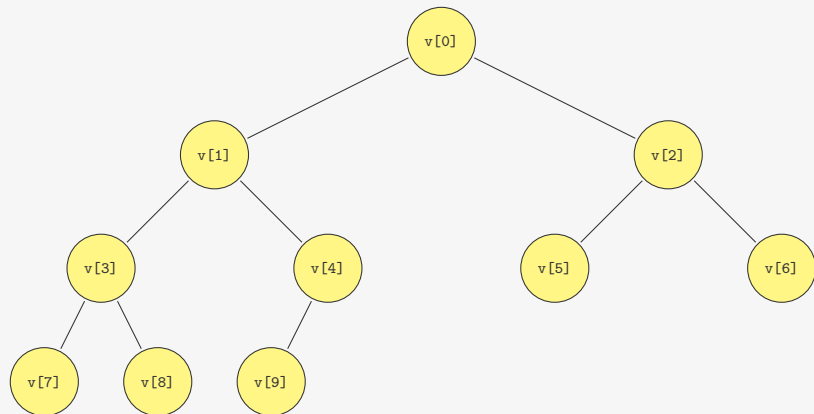


Max-Heap



Em um Heap (de máximo):

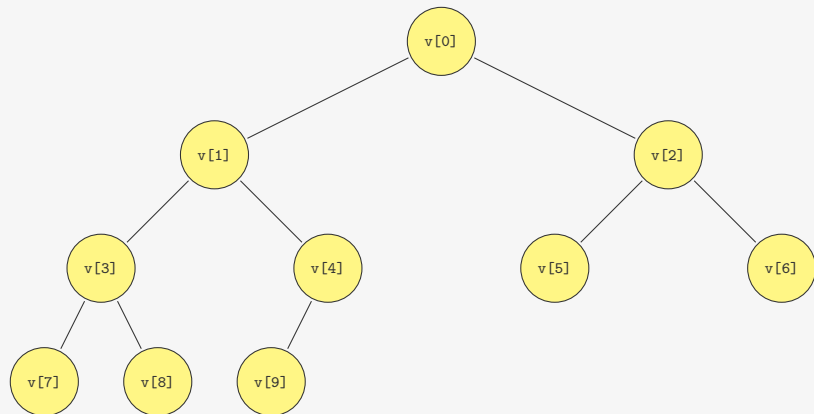
Max-Heap



Em um Heap (de máximo):

- Os filhos são menores ou iguais ao pai

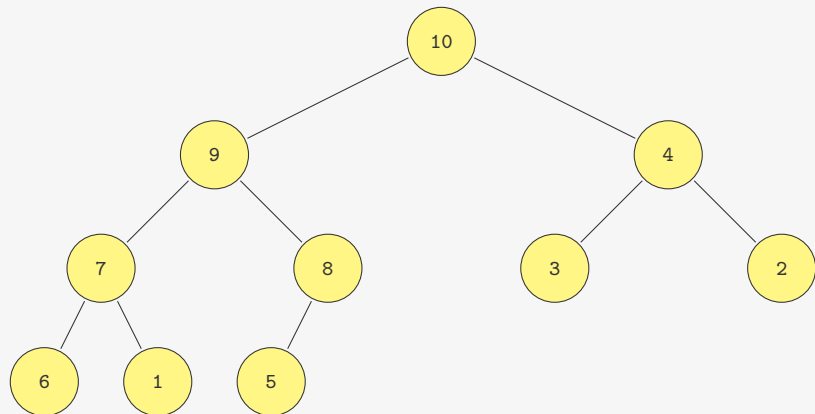
Max-Heap



Em um Heap (de máximo):

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

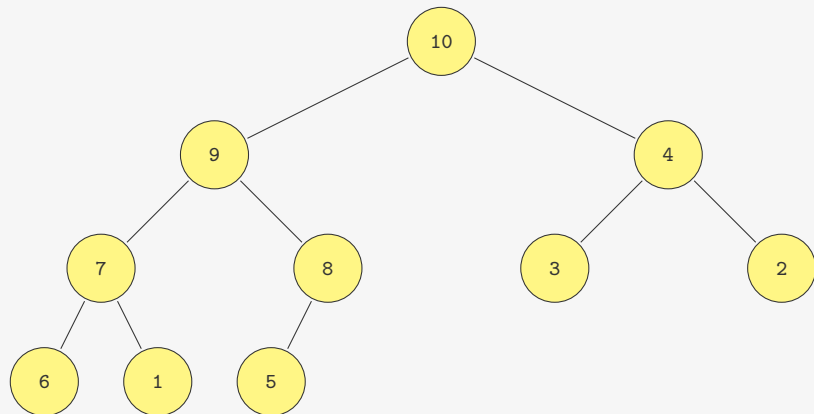
Max-Heap



Em um Heap (de máximo):

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

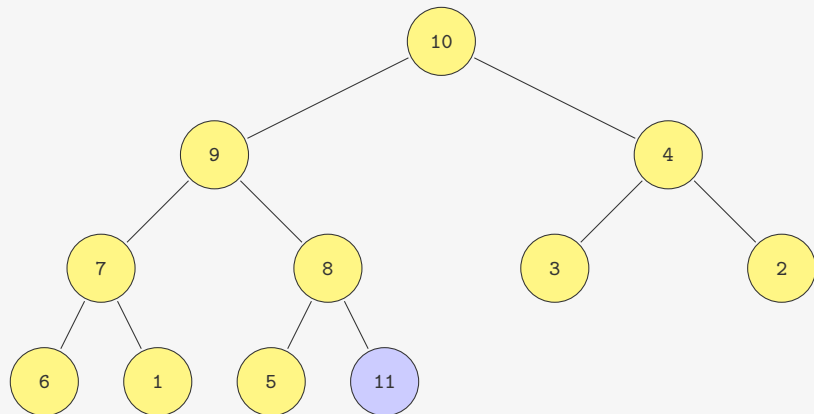
Inserindo no Heap



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

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

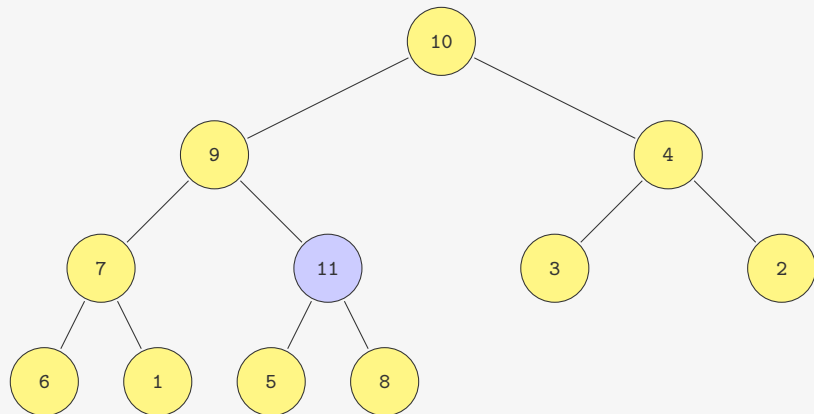
Inserindo no Heap



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

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

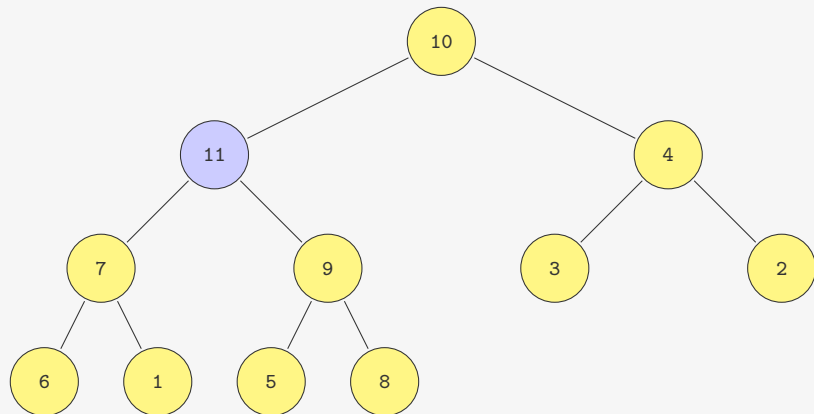
Inserindo no Heap



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

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

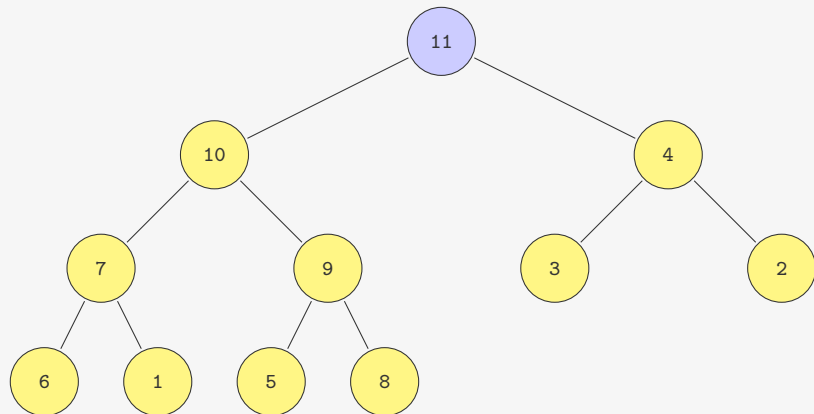
Inserindo no Heap



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(p_fp fprio, Item item) {
```

Inserindo no Heap

```
1 void insere(p_fp fprio, Item item) {  
2     fprio->v[fprio->n] = item;
```

Inserindo no Heap

```
1 void insere(p_fp fprio, Item item) {  
2     fprio->v[fprio->n] = item;  
3     fprio->n++;  
}
```

Inserindo no Heap

```
1 void insere(p_fp fprio, Item item) {  
2     fprio->v[fprio->n] = item;  
3     fprio->n++;  
4     sobe_no_heap(fprio, fprio->n - 1);  
5 }
```

Inserindo no Heap

```
1 void insere(p_fp fprio, Item item) {
2     fprio->v[fprio->n] = item;
3     fprio->n++;
4     sobe_no_heap(fprio, fprio->n - 1);
5 }
6
7 #define PAI(i) ((i-1)/2)
```

Inserindo no Heap

```
1 void insere(p_fp fprio, Item item) {
2     fprio->v[fprio->n] = item;
3     fprio->n++;
4     sobe_no_heap(fprio, fprio->n - 1);
5 }
6
7 #define PAI(i) ((i-1)/2)
8
9 void sobe_no_heap(p_fp fprio, int k) {
```


Inserindo no Heap

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

Inserindo no Heap

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

Inserindo no Heap

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

Inserindo no Heap

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

Tempo de **insere**:

Inserindo no Heap

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

Tempo de **insere**:

- No máximo subimos até a raiz

Inserindo no Heap

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

Tempo de **insere**:

- No máximo subimos até a raiz

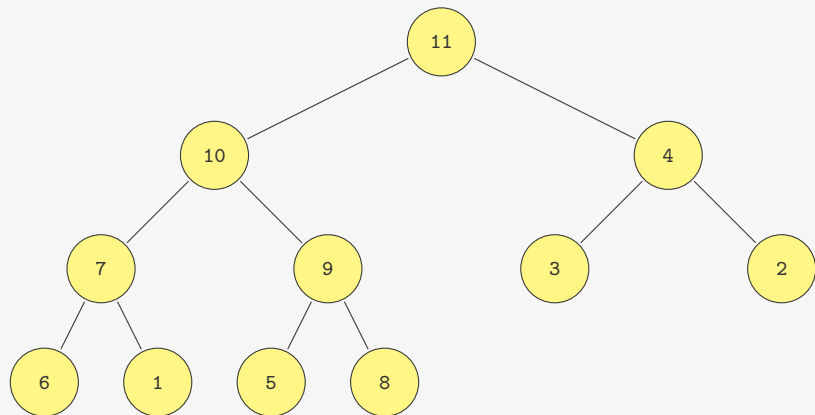
Inserindo no Heap

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

Tempo de **insere**:

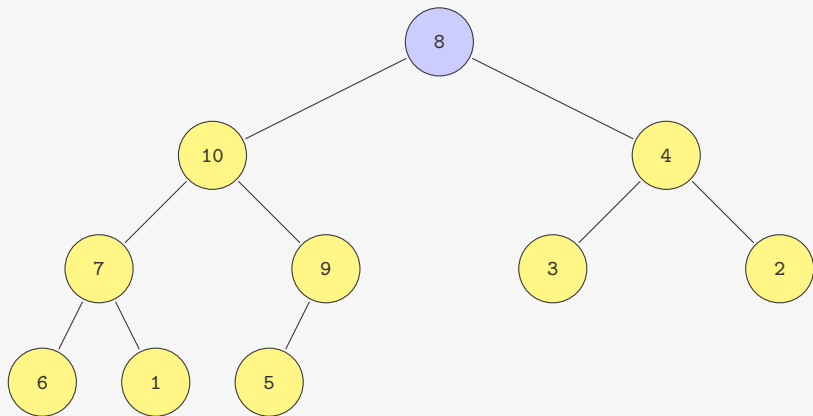
- No máximo subimos até a raiz
- Ou seja, $O(\lg n)$

Extraindo o Máximo



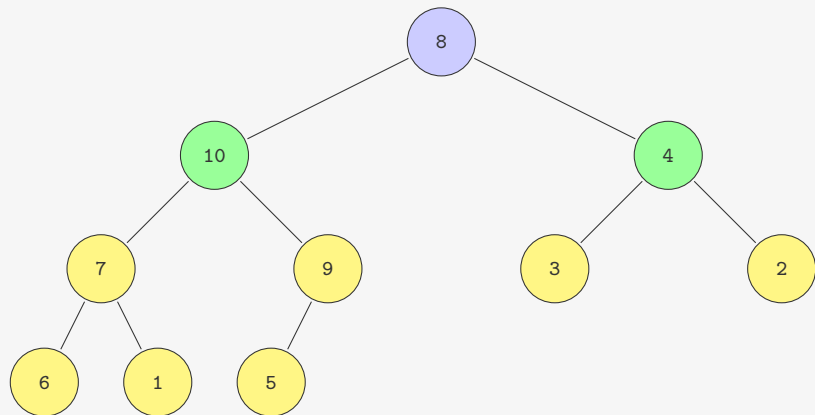
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
 - Trocamos o pai com o maior dos dois filhos (se necessário)

Extraindo o Máximo



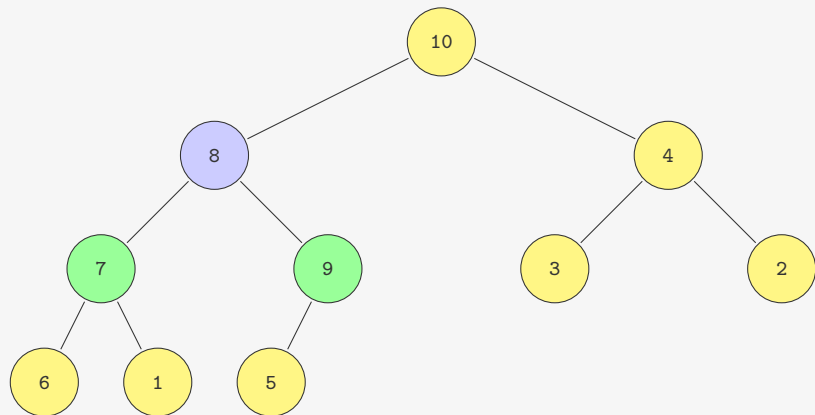
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
 - Trocamos o pai com o maior dos dois filhos (se necessário)

Extraindo o Máximo



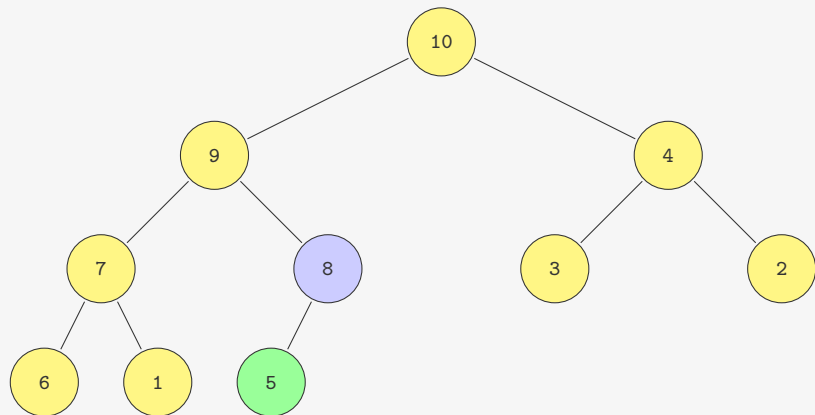
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
 - Trocamos o pai com o maior dos dois filhos (se necessário)

Extraindo o Máximo



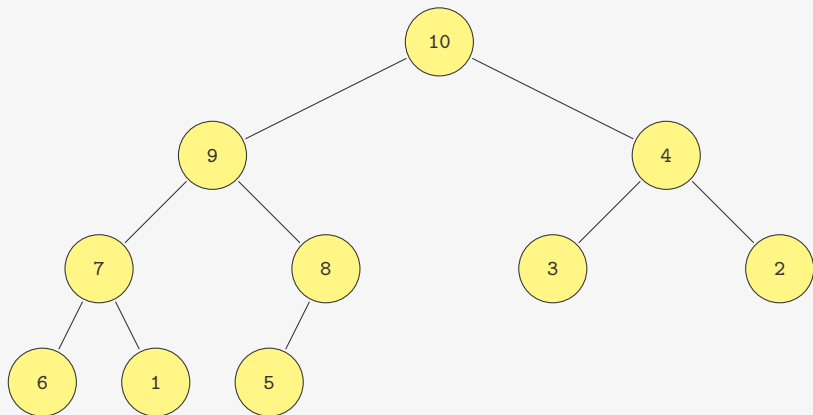
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
 - Trocamos o pai com o maior dos dois filhos (se necessário)

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(p_fp fprio) {
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {  
2     Item item = fprio->v[0];
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {  
2     Item item = fprio->v[0];  
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
```


Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {  
2     Item item = fprio->v[0];  
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);  
4     fprio->n--;
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
13     int maior_filho;
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
13     int maior_filho;
14     if (F_ESQ(k) < fprio->n) {
```


Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
13     int maior_filho;
14     if (F_ESQ(k) < fprio->n) {
15         maior_filho = F_ESQ(k);
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
13     int maior_filho;
14     if (F_ESQ(k) < fprio->n) {
15         maior_filho = F_ESQ(k);
16         if (F_DIR(k) < fprio->n &&
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
13     int maior_filho;
14     if (F_ESQ(k) < fprio->n) {
15         maior_filho = F_ESQ(k);
16         if (F_DIR(k) < fprio->n &&
17             fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
13     int maior_filho;
14     if (F_ESQ(k) < fprio->n) {
15         maior_filho = F_ESQ(k);
16         if (F_DIR(k) < fprio->n &&
17             fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
18             maior_filho = F_DIR(k);
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
13     int maior_filho;
14     if (F_ESQ(k) < fprio->n) {
15         maior_filho = F_ESQ(k);
16         if (F_DIR(k) < fprio->n &&
17             fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
18             maior_filho = F_DIR(k);
19         if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
13     int maior_filho;
14     if (F_ESQ(k) < fprio->n) {
15         maior_filho = F_ESQ(k);
16         if (F_DIR(k) < fprio->n &&
17             fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
18             maior_filho = F_DIR(k);
19         if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
20             troca(&fprio->v[k], &fprio->v[maior_filho]);
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
13     int maior_filho;
14     if (F_ESQ(k) < fprio->n) {
15         maior_filho = F_ESQ(k);
16         if (F_DIR(k) < fprio->n &&
17             fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
18             maior_filho = F_DIR(k);
19         if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
20             troca(&fprio->v[k], &fprio->v[maior_filho]);
21             desce_no_heap(fprio, maior_filho);
22         }
23     }
24 }
```

Extraindo o Máximo

```
1 Item extrai_maximo(p_fp fprio) {
2     Item item = fprio->v[0];
3     troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
4     fprio->n--;
5     desce_no_heap(fprio, 0);
6     return item;
7 }
8
9 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
10 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
11
12 void desce_no_heap(p_fp fprio, int k) {
13     int maior_filho;
14     if (F_ESQ(k) < fprio->n) {
15         maior_filho = F_ESQ(k);
16         if (F_DIR(k) < fprio->n &&
17             fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
18             maior_filho = F_DIR(k);
19         if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
20             troca(&fprio->v[k], &fprio->v[maior_filho]);
21             desce_no_heap(fprio, maior_filho);
22         }
23     }
24 }
```

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

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(p_fp fprio, int k, int valor) {
2   if (fprio->v[k].chave < valor) {
3     fprio->v[k].chave = valor;
4     sobe_no_heap(fprio, k);
5   } else {
6     fprio->v[k].chave = valor;
7     desce_no_heap(fprio, 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(p_fp fprio, int k, int valor) {
2   if (fprio->v[k].chave < valor) {
3     fprio->v[k].chave = valor;
4     sobe_no_heap(fprio, k);
5   } else {
6     fprio->v[k].chave = valor;
7     desce_no_heap(fprio, k);
8   }
9 }
```

Tempo: $O(\lg n)$

Ordenação usando Fila de Prioridades

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

Ordenação usando Fila de Prioridades

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

Tempo: $O(n \lg n)$

Ordenação usando Fila de Prioridades

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

Tempo: $O(n \lg n)$

- 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     p_fp fprio = criar_fprio(r-l+1);
4     for (i = l; i <= r; i++)
5         insere(fprio, v[i]);
6     for (i = r; i >= l; i--)
7         v[i] = extrai_maximo(fprio);
8 }
```

Tempo: $O(n \lg n)$

- 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     p_fp fprio = criar_fprio(r-l+1);
4     for (i = l; i <= r; i++)
5         insere(fprio, v[i]);
6     for (i = r; i >= l; i--)
7         v[i] = extrai_maximo(fprio);
8 }
```

Tempo: $O(n \lg n)$

- 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     p_fp fprio = criar_fprio(r-l+1);
4     for (i = l; i <= r; i++)
5         insere(fprio, v[i]);
6     for (i = r; i >= l; i--)
7         v[i] = extrai_maximo(fprio);
8 }
```

Tempo: $O(n \lg n)$

- 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

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
---	---	---	---	---	---	---	---	---	----

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

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

1

9

10

5

Transformando um vetor em um heap

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

1

9

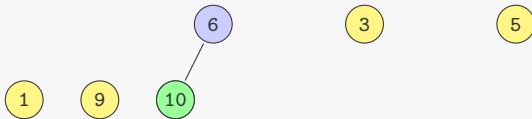
10

3

5

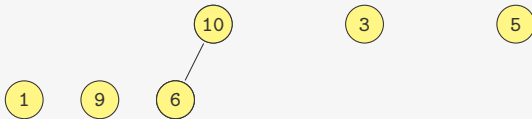
Transformando um vetor em um heap

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



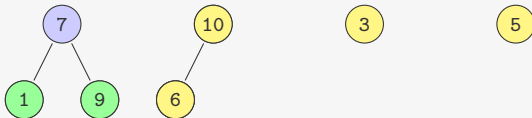
Transformando um vetor em um heap

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



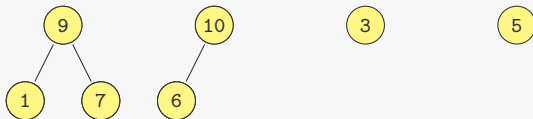
Transformando um vetor em um heap

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



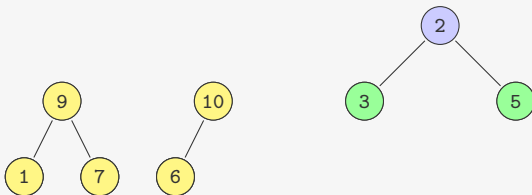
Transformando um vetor em um heap

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



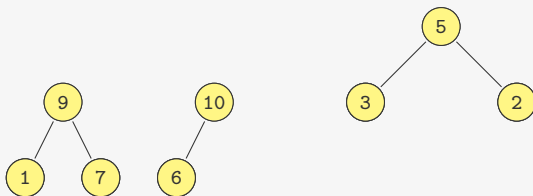
Transformando um vetor em um heap

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



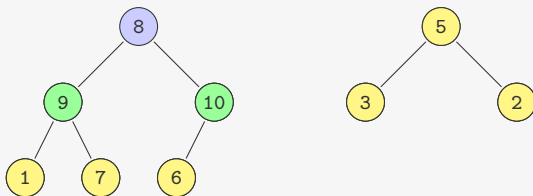
Transformando um vetor em um heap

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



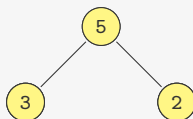
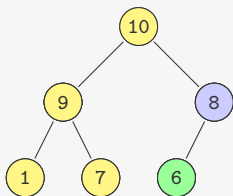
Transformando um vetor em um heap

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



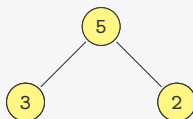
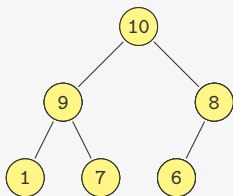
Transformando um vetor em um heap

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



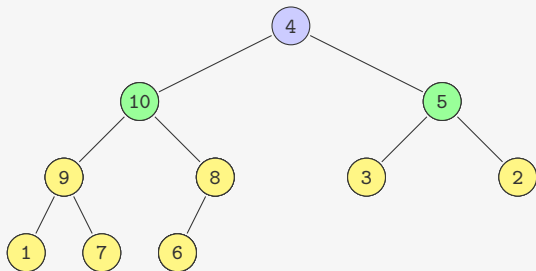
Transformando um vetor em um heap

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



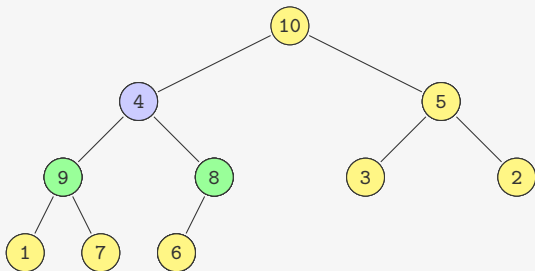
Transformando um vetor em um heap

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



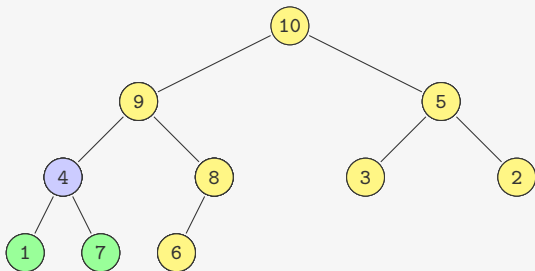
Transformando um vetor em um heap

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



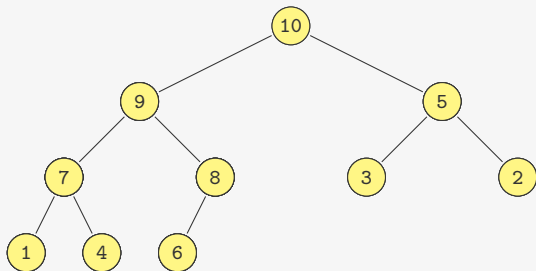
Transformando um vetor em um heap

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



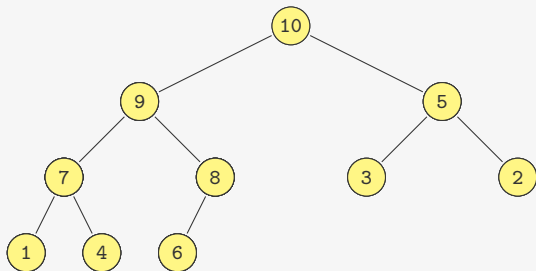
Transformando um vetor em um heap

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



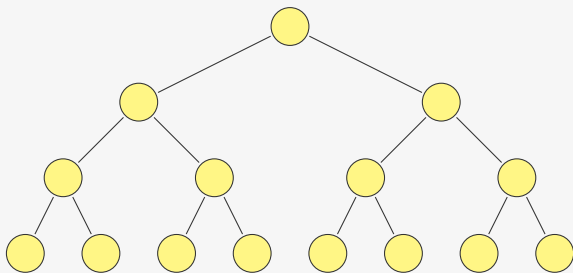
Transformando um vetor em um heap

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

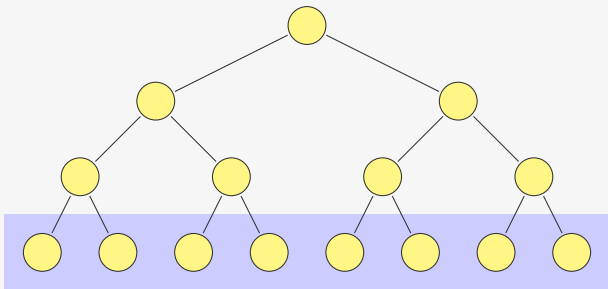


Quanto tempo demora?

Tempo da construção para $n = 2^k$

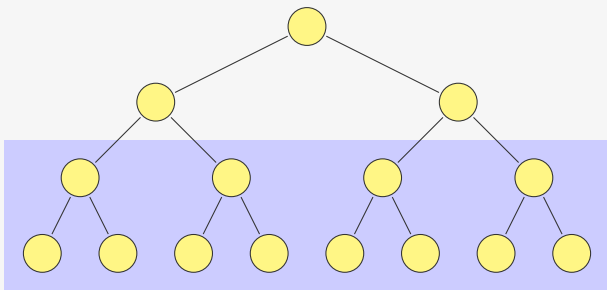


Tempo da construção para $n = 2^k$



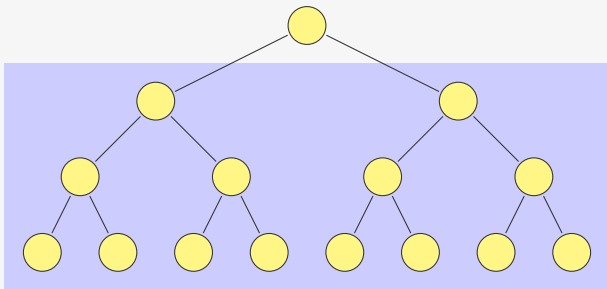
- Temos $n/2 = 2^{k-1}$ heaps de altura 1

Tempo da construção para $n = 2^k$



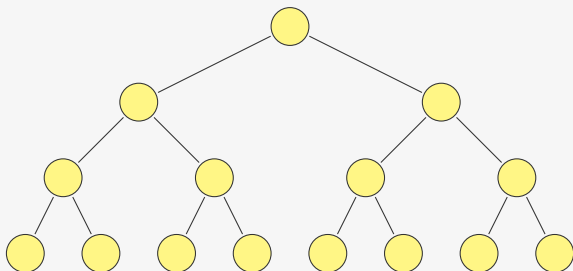
- Temos $n/2 = 2^{k-1}$ heaps de altura 1
- Temos $n/4 = 2^{k-2}$ heaps de altura 2

Tempo da construção para $n = 2^k$



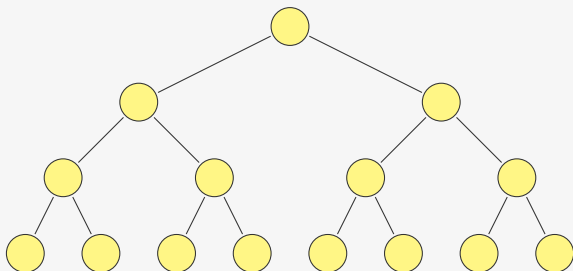
- Temos $n/2 = 2^{k-1}$ heaps de altura 1
- Temos $n/4 = 2^{k-2}$ heaps de altura 2
- Temos $n/2^{h+1} = 2^{k-h-1}$ heaps de altura h

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 $c \cdot h$

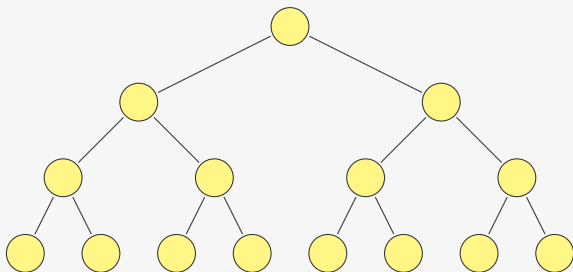
Tempo da construção para $n = 2^k$



- 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Ou seja,

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

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

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

Ou seja,

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

Heapsort

```
1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, maior_filho);
11        }
12    }
13 }
```

Heapsort

```
1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, maior_filho);
11        }
12    }
13 }
14
15 void heapsort(int *v, int l, int r) {
```

Heapsort

```
1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, maior_filho);
11        }
12    }
13 }
14
15 void heapsort(int *v, int l, int r) {
16     int k, n = r-1+1;
```

Heapsort

```
1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, maior_filho);
11        }
12    }
13 }
14
15 void heapsort(int *v, int l, int r) {
16     int k, n = r-1+1;
17     int *heap = &v[l];
```

Heapsort

```
1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, maior_filho);
11        }
12    }
13 }
14
15 void heapsort(int *v, int l, int r) {
16     int k, n = r-1+1;
17     int *heap = &v[l];
18     for (k = n/2; k >= 1; k--) /* transforma em heap */
```

Heapsort

```
1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, maior_filho);
11        }
12    }
13 }
14
15 void heapsort(int *v, int l, int r) {
16     int k, n = r-1+1;
17     int *heap = &v[l];
18     for (k = n/2; k >= 1; k--) /* transforma em heap */
19         desce_no_heap(heap, n, k);
```

Heapsort

```
1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, maior_filho);
11        }
12    }
13 }
14
15 void heapsort(int *v, int l, int r) {
16     int k, n = r-1+1;
17     int *heap = &v[l];
18     for (k = n/2; k >= 1; k--) /* transforma em heap */
19         desce_no_heap(heap, n, k);
20     while (n > 1) { /* extrai o máximo */
```


Heapsort

```
1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, maior_filho);
11        }
12    }
13 }
14
15 void heapsort(int *v, int l, int r) {
16     int k, n = r-1+1;
17     int *heap = &v[l];
18     for (k = n/2; k >= 1; k--) /* transforma em heap */
19         desce_no_heap(heap, n, k);
20     while (n > 1) { /* extrai o máximo */
21         troca(&heap[0], &heap[fprio->n - 1]);
```

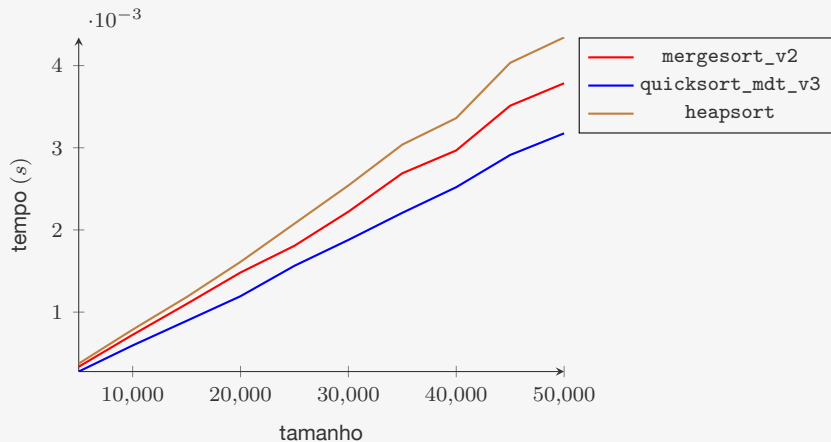
Heapsort

```
1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, maior_filho);
11        }
12    }
13 }
14
15 void heapsort(int *v, int l, int r) {
16     int k, n = r-1+1;
17     int *heap = &v[l];
18     for (k = n/2; k >= 1; k--) /* transforma em heap */
19         desce_no_heap(heap, n, k);
20     while (n > 1) { /* extrai o máximo */
21         troca(&heap[0], &heap[fprio->n - 1]);
22         n--;
```

Heapsort

```
1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, maior_filho);
11        }
12    }
13 }
14
15 void heapsort(int *v, int l, int r) {
16     int k, n = r-1+1;
17     int *heap = &v[l];
18     for (k = n/2; k >= 1; k--) /* transforma em heap */
19         desce_no_heap(heap, n, k);
20     while (n > 1) { /* extrai o máximo */
21         troca(&heap[0], &heap[fprio->n - 1]);
22         n--;
23         desce_no_heap(heap, n, 0);
24     }
25 }
```

Comparação com QuickSort e MergeSort



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

Exercício

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`).