MC-202 — Unidade 13
Filas de Prioridade e HeapSort

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

Universidade Estadual de Campinas

2º 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...

```
int selection_invertido(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = i;
    for (j = i-1; j >= l; j--)
      if (v[j] > v[max])
        max = j;
    troca(&v[i],&v[max]);
  }
}
```

Rescrevendo...

Usamos uma função que acha o elemento máximo do vetor

```c
int extrai_maximo(int *v, int l, int r) {
  max = r;
  for (j = r; j >= l; j--)
    if (v[j] > v[max])
      max = j;
  return max;
}
```

Rescrevendo...

Usamos uma função que acha o elemento máximo do vetor

```
int extrai_maximo(int *v, int l, int r) {
  max = r;
  for (j = r; j >= l; j--)
    if (v[j] > v[max])
      max = j;
  return max;
}
```

E reescrevemos o SelectionSort

```
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

Tempo do SelectionSort

```
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

Tempo do SelectionSort

```
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

O tempo do `selection_invertido_v2` é:

Tempo do SelectionSort

```
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

O tempo do `selection_invertido_v2` é:

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

Tempo do SelectionSort

```
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

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

```
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

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

```
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

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

```c
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

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

```c
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

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

```
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

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

```
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

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

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

Tempo do SelectionSort

```c
int selection_invertido_v2(int *v, int l, int r) {
  int i, j, max;
  for (i = r; i > l; i--) {
    max = extrai_maximo(l, i);
    troca(&v[i],&v[max]);
  }
}
```

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

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

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

```c
typedef struct {
  char nome[20];
  int chave;
} Item;
```

Fila de Prioridade (usando vetores) - TAD

```c
typedef struct {
  char nome[20];
  int chave;
} Item;

typedef struct {
  Item *v;
  int n, tamanho;
} FP;
```

Fila de Prioridade (usando vetores) - TAD

```c
typedef struct {
  char nome[20];
  int chave;
} Item;

typedef struct {
  Item *v;
  int n, tamanho;
} FP;

typedef FP * p_fp
```

Fila de Prioridade (usando vetores) - TAD

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

typedef struct {
  Item *v;
  int n, tamanho;
} FP;

typedef FP * p_fp

p_fp criar_filaprio(int tam);
void insere(p_fp fprio, Item item);
Item extrai_maximo(p_fp fprio);
int vazia(p_fp fprio);
int cheia(p_fp fprio);
```

Operações Básicas

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

Operações Básicas

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

Operações Básicas

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

Operações Básicas

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

Operações Básicas

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

Operações Básicas

```
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

Operações Básicas

```c
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

Operações Básicas

```
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

Operações Básicas

```c
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

Operações Básicas

```c
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

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

Operações Básicas

```c
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

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

Operações Básicas

```
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

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

Operações Básicas

```c
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

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

Operações Básicas

```c
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

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

Operações Básicas

```c
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

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

Operações Básicas

```c
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

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

Operações Básicas

```
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

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

Operações Básicas

```c
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

```c
Item extrai_maximo(p_fp fprio) {
  int j, max = 0;
  for (j = 1; j < fprio->n; j++)
    if (fprio->v[max].chave < fprio->v[j].chave)
      max = j;
  troca(&(fprio->v[max]), &(fprio->v[fprio->n-1]));
  fprio->n--;
  return fprio->v[fprio->n];
}
```

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

Operações Básicas

```c
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

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

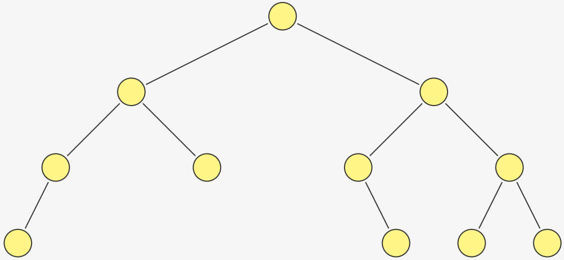
```c
Item extrai_maximo(p_fp fprio) {
  int j, max = 0;
  for (j = 1; j < fprio->n; j++)
    if (fprio->v[max].chave < fprio->v[j].chave)
      max = j;
  troca(&(fprio->v[max]), &(fprio->v[fprio->n-1]));
  fprio->n--;
  return fprio->v[fprio->n];
}
```

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

- Se mantiver o vetor ordenado, os tempos se invertem

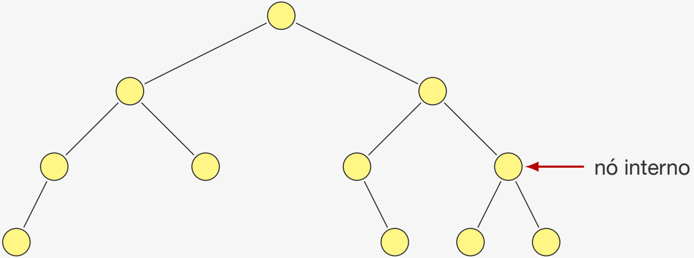
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



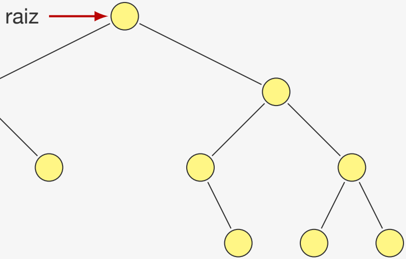
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



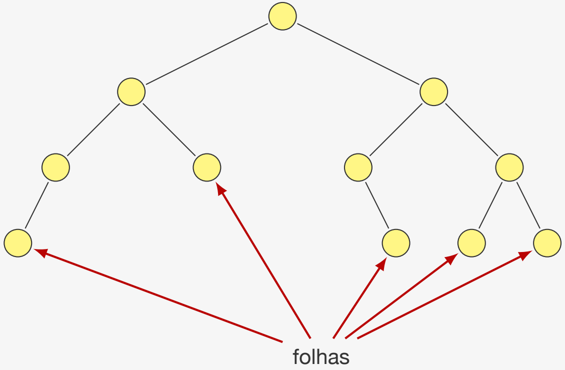
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



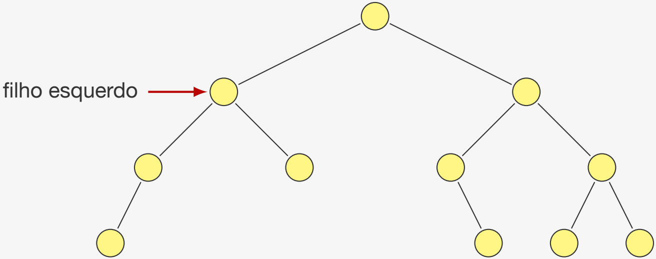
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



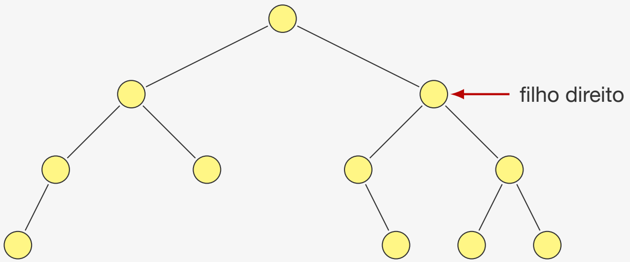
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



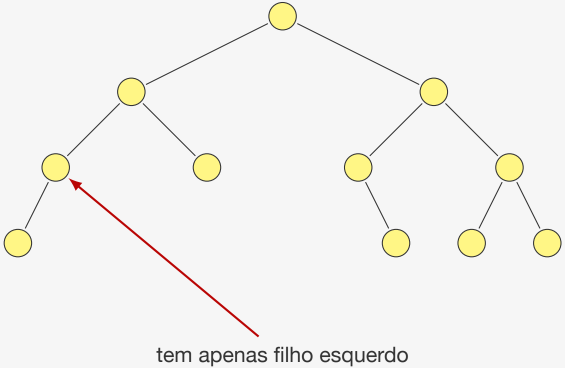
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



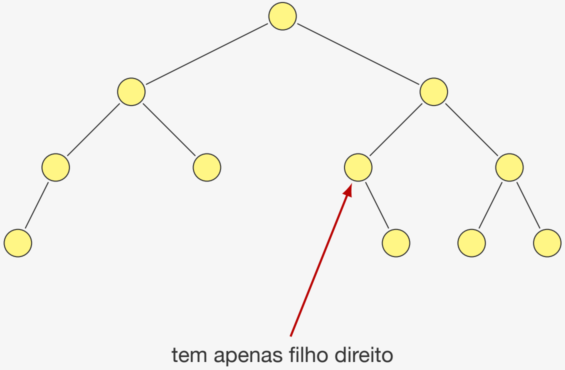
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



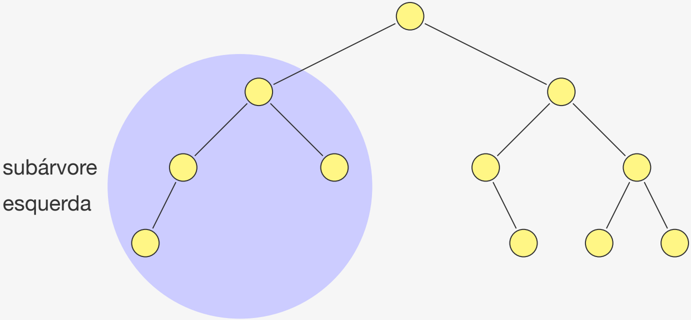
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



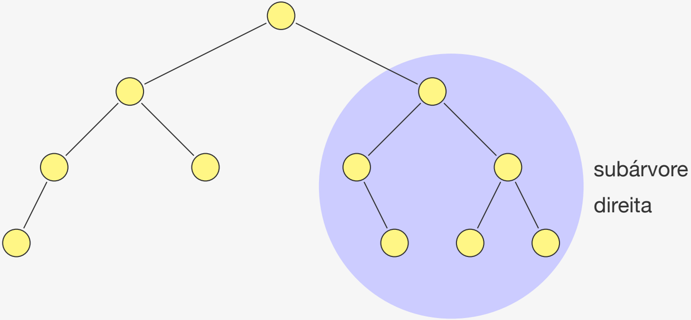
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



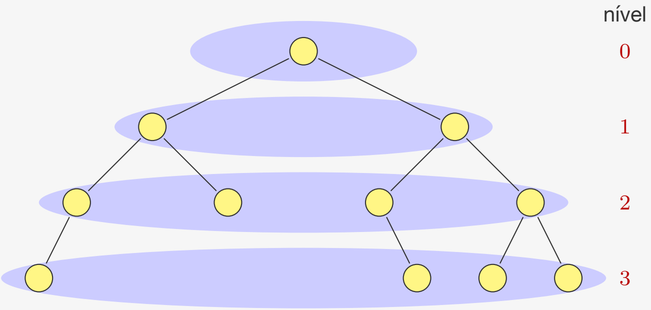
Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:



Interlúdio - Árvores Binárias

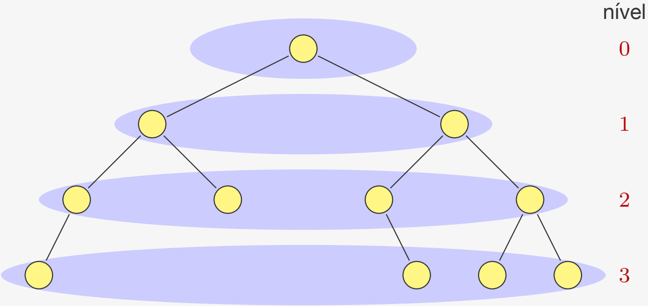
Exemplo de uma árvore binária:



Uma árvore binária é:

Interlúdio - Árvores Binárias

Exemplo de uma árvore binária:

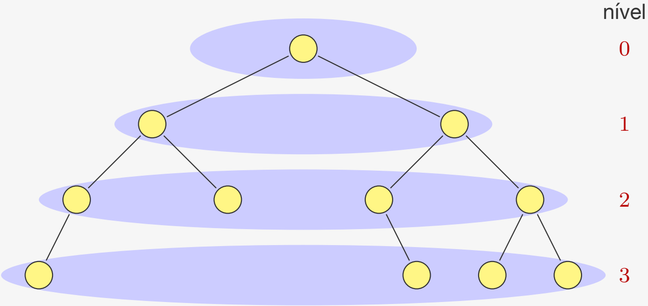


Uma árvore binária é:

- 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:

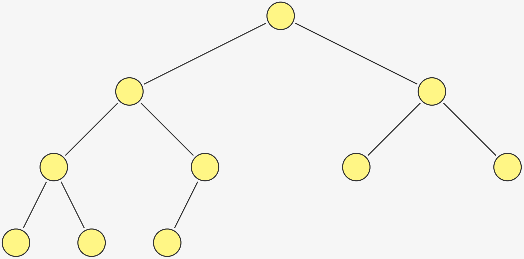
- Todos os níveis exceto o último estão cheios
- No último nível, todos os nós estão o mais a esquerda possível

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:

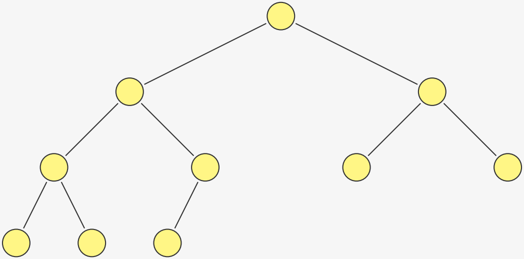


Árvores Binárias Completas

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:



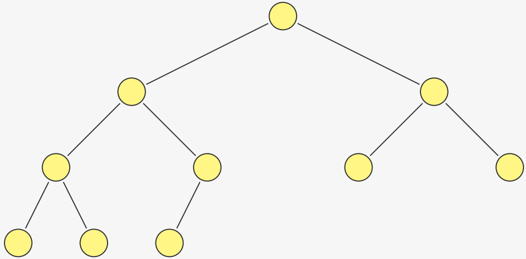
Uma árvore binária completa de n nós tem quantos níveis?

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

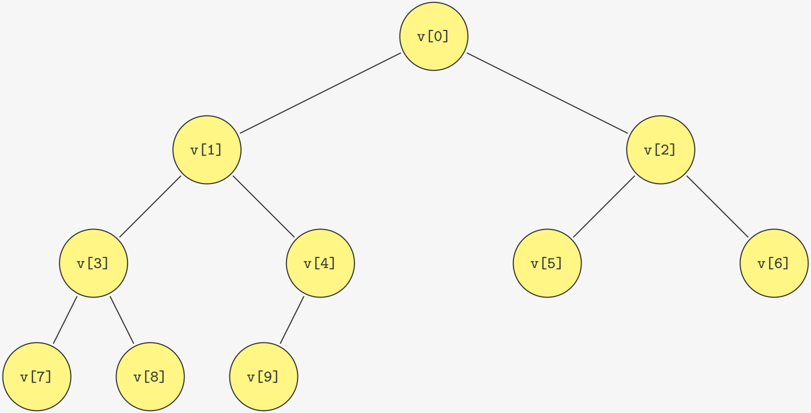
Exemplo:



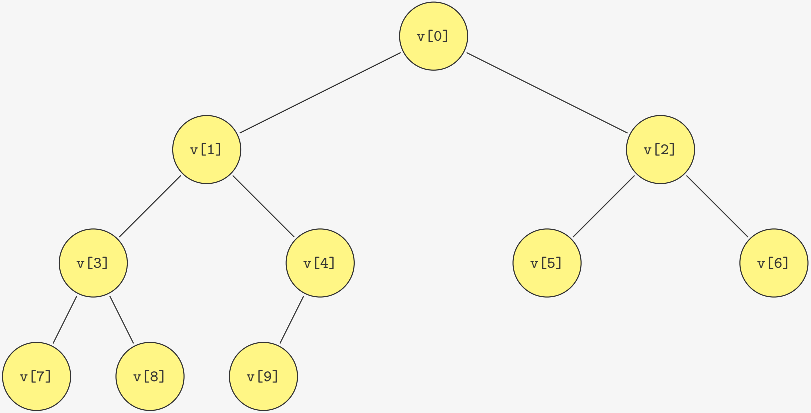
Uma árvore binária completa de n nós tem quantos níveis?

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

Árvores Binárias Completas e Vetores

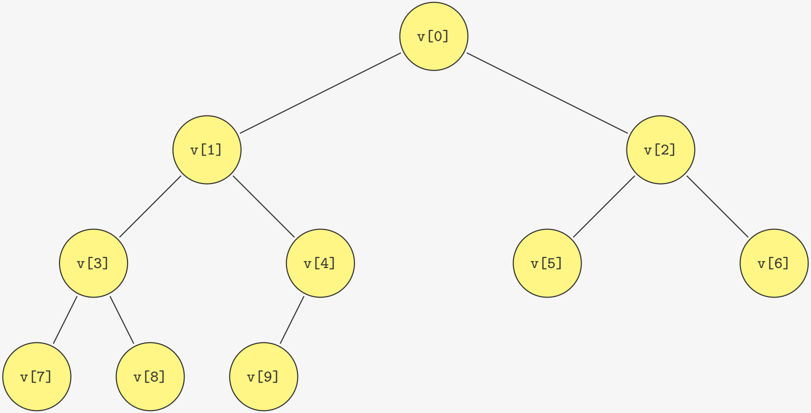


Árvores Binárias Completas e Vetores



Em relação a `v[i]`:

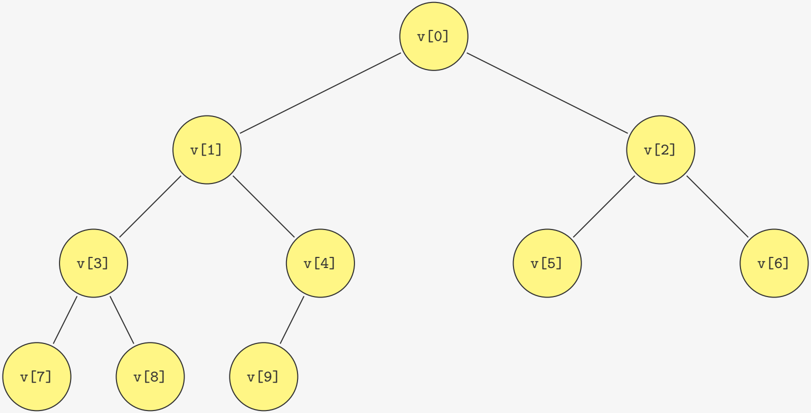
Árvores Binárias Completas e Vetores



Em relação a `v[i]`:

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

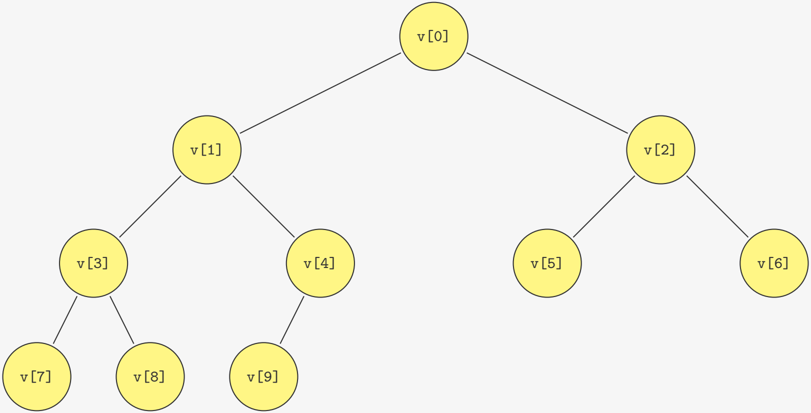
Árvores Binárias Completas e Vetores



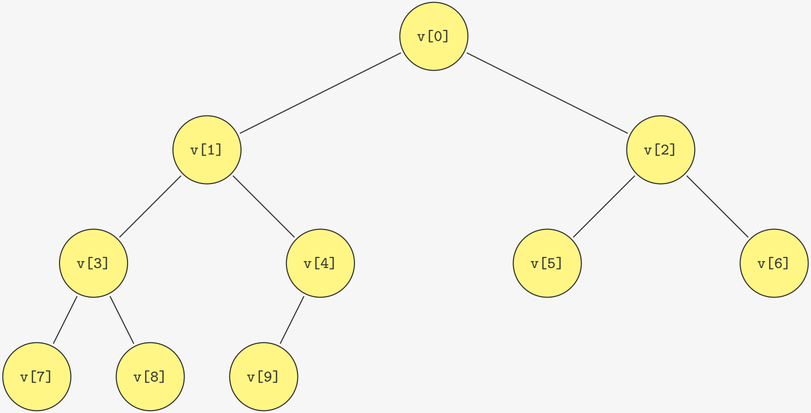
Em relação a `v[i]`:

- 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

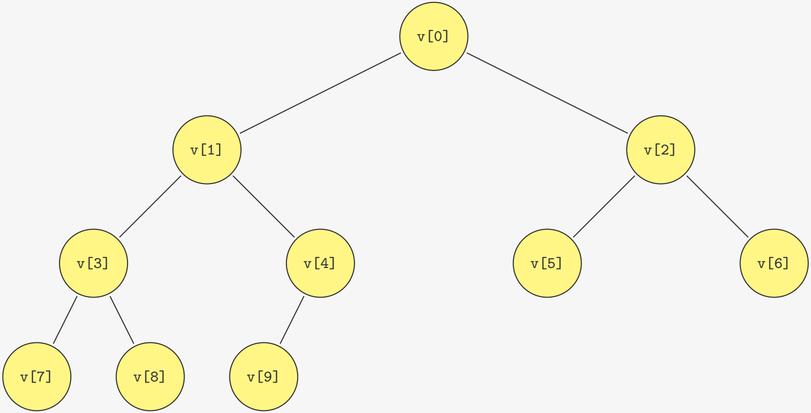


Max-Heap



Em um Heap (de máximo):

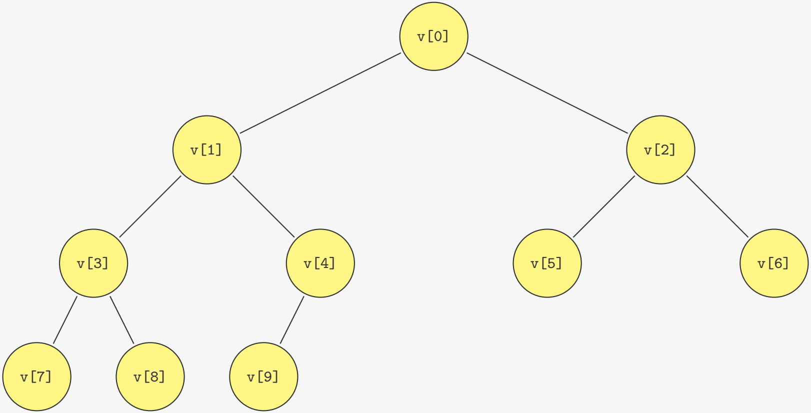
Max-Heap



Em um Heap (de máximo):

- Os filhos são menores ou iguais ao pai

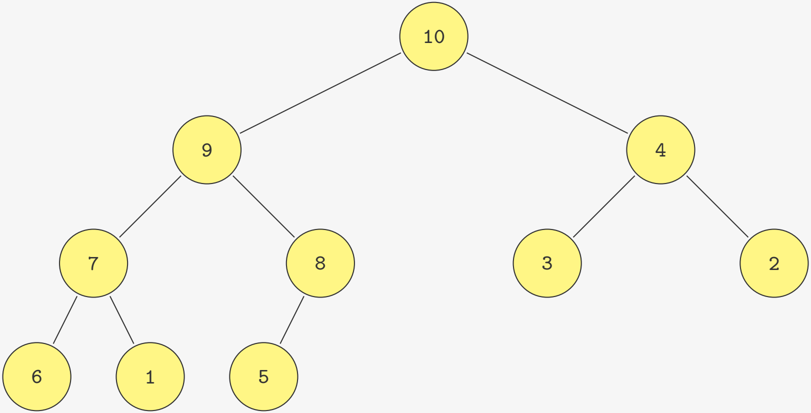
Max-Heap



Em um Heap (de máximo):

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

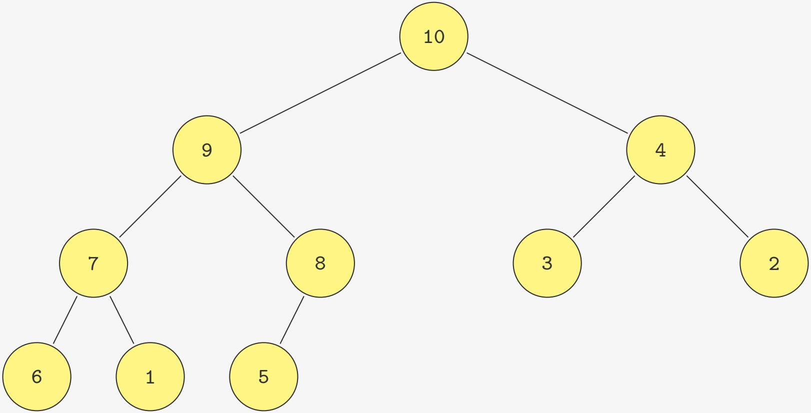
Max-Heap



Em um Heap (de máximo):

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

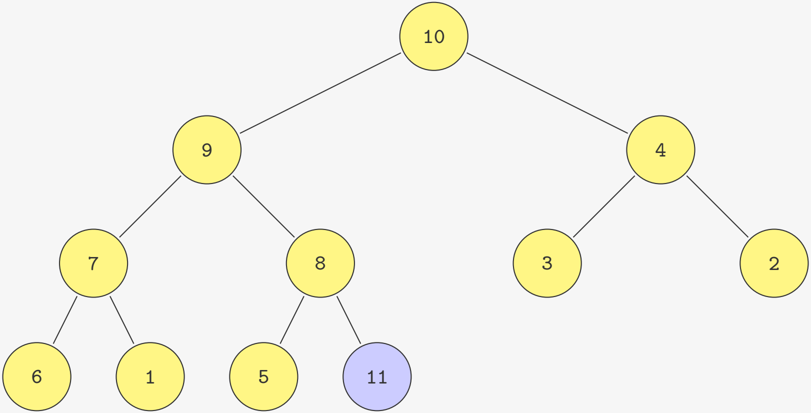
Inserindo no Heap



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

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

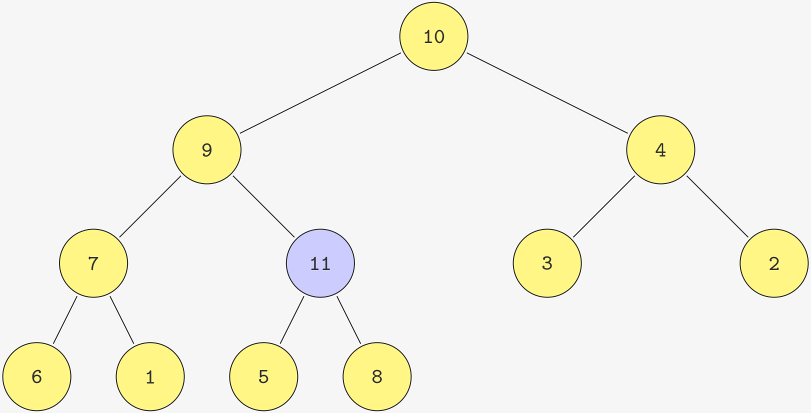
Inserindo no Heap



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

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

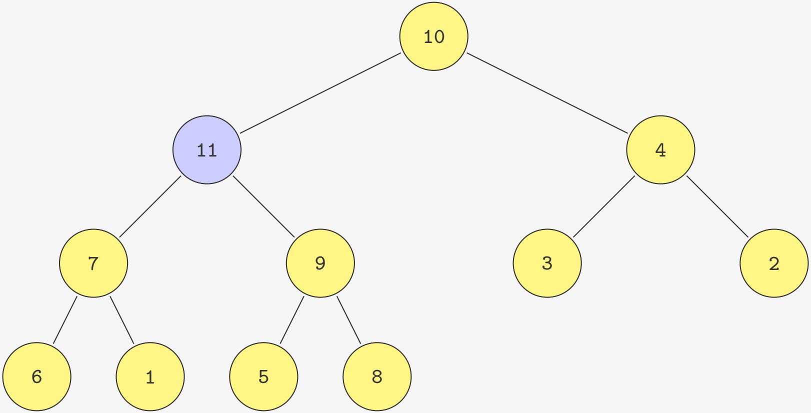
Inserindo no Heap



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

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

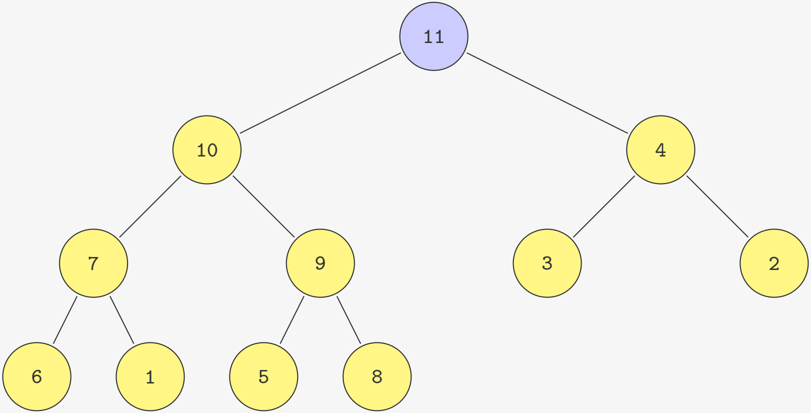
Inserindo no Heap



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

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

Inserindo no Heap

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

Inserindo no Heap

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

Inserindo no Heap

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

Inserindo no Heap

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

#define PAI(i) ((i-1)/2)
```

Inserindo no Heap

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

#define PAI(i) ((i-1)/2)

void sobe_no_heap(p_fp fprio, int k) {
```

Inserindo no Heap

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

#define PAI(i) ((i-1)/2)

void sobe_no_heap(p_fp fprio, int k) {
  if (k > 0 && fprio->v[PAI(k)].chave < fprio->v[k].chave) {
```

Inserindo no Heap

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

#define PAI(i) ((i-1)/2)

void sobe_no_heap(p_fp fprio, int k) {
  if (k > 0 && fprio->v[PAI(k)].chave < fprio->v[k].chave) {
    troca(&fprio->v[k], &fprio->v[PAI(k)]);
```

Inserindo no Heap

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

#define PAI(i) ((i-1)/2)

void sobe_no_heap(p_fp fprio, int k) {
  if (k > 0 && fprio->v[PAI(k)].chave < fprio->v[k].chave) {
    troca(&fprio->v[k], &fprio->v[PAI(k)]);
    sobe_no_heap(fprio, PAI(k));
  }
}
```

Inserindo no Heap

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

#define PAI(i) ((i-1)/2)

void sobe_no_heap(p_fp fprio, int k) {
  if (k > 0 && fprio->v[PAI(k)].chave < fprio->v[k].chave) {
    troca(&fprio->v[k], &fprio->v[PAI(k)]);
    sobe_no_heap(fprio, PAI(k));
  }
}
```

Tempo de `insere`:

Inserindo no Heap

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

#define PAI(i) ((i-1)/2)

void sobe_no_heap(p_fp fprio, int k) {
  if (k > 0 && fprio->v[PAI(k)].chave < fprio->v[k].chave) {
    troca(&fprio->v[k], &fprio->v[PAI(k)]);
    sobe_no_heap(fprio, PAI(k));
  }
}
```

Tempo de `insere`:

- No máximo subimos até a raiz

Inserindo no Heap

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

#define PAI(i) ((i-1)/2)

void sobe_no_heap(p_fp fprio, int k) {
  if (k > 0 && fprio->v[PAI(k)].chave < fprio->v[k].chave) {
    troca(&fprio->v[k], &fprio->v[PAI(k)]);
    sobe_no_heap(fprio, PAI(k));
  }
}
```

Tempo de `insere`:

- No máximo subimos até a raiz

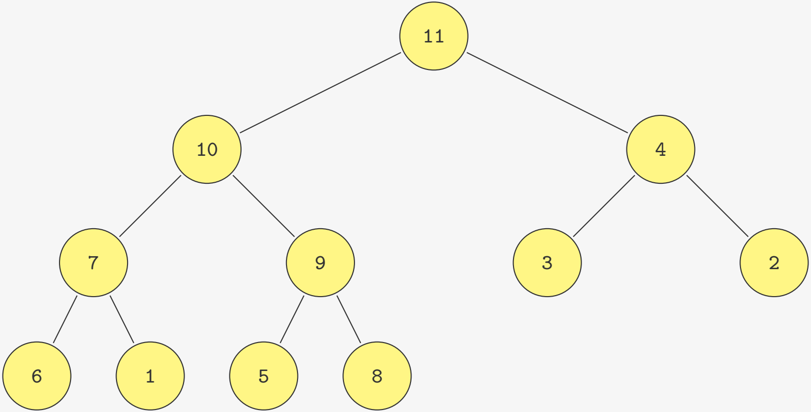
Inserindo no Heap

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

#define PAI(i) ((i-1)/2)

void sobe_no_heap(p_fp fprio, int k) {
  if (k > 0 && fprio->v[PAI(k)].chave < fprio->v[k].chave) {
    troca(&fprio->v[k], &fprio->v[PAI(k)]);
    sobe_no_heap(fprio, PAI(k));
  }
}
```

Tempo de `insere`:

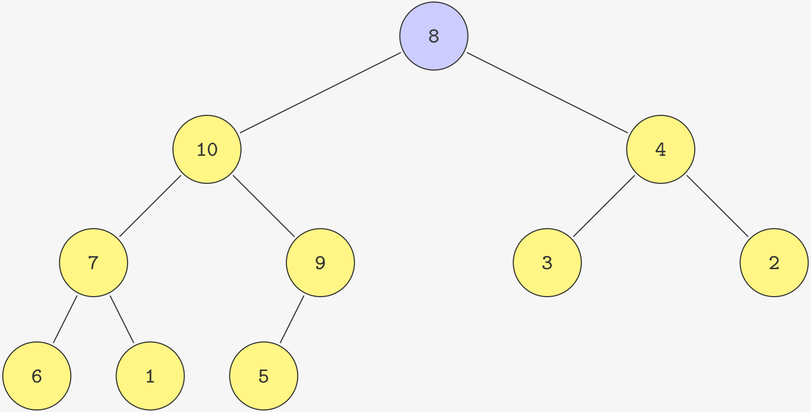
- No máximo subimos até a raiz
- Ou seja, $O(\lg n)$

Extraindo o Máximo



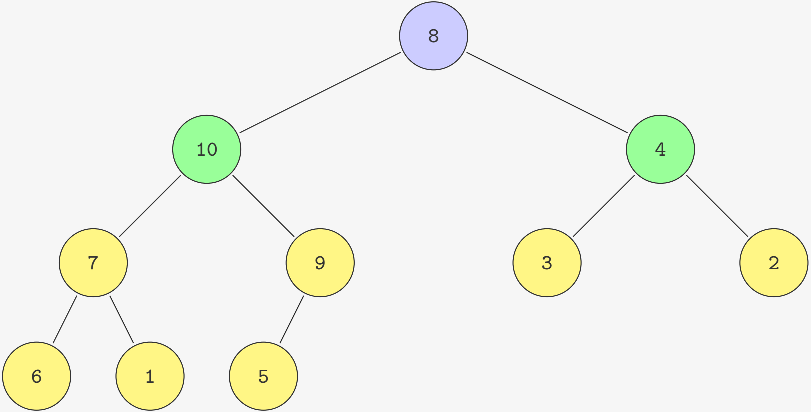
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

Extraindo o Máximo



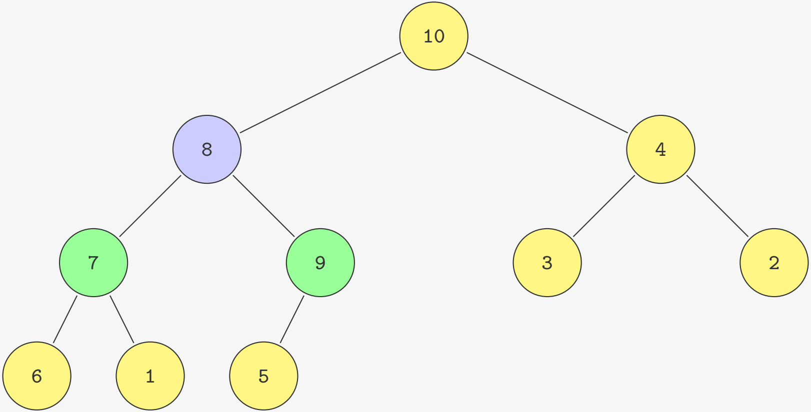
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

Extraindo o Máximo



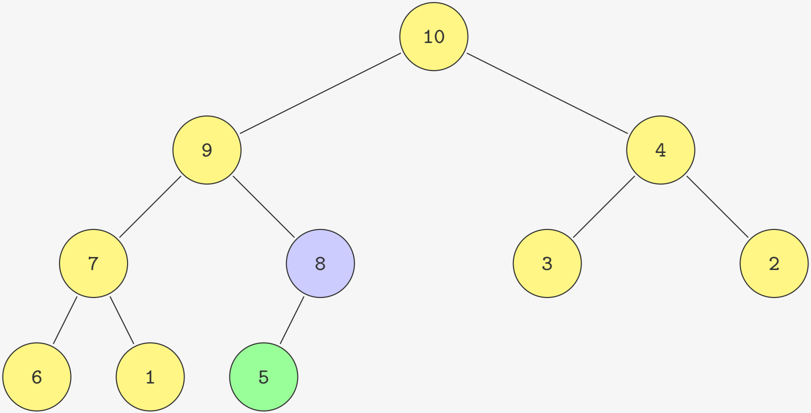
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

Extraindo o Máximo



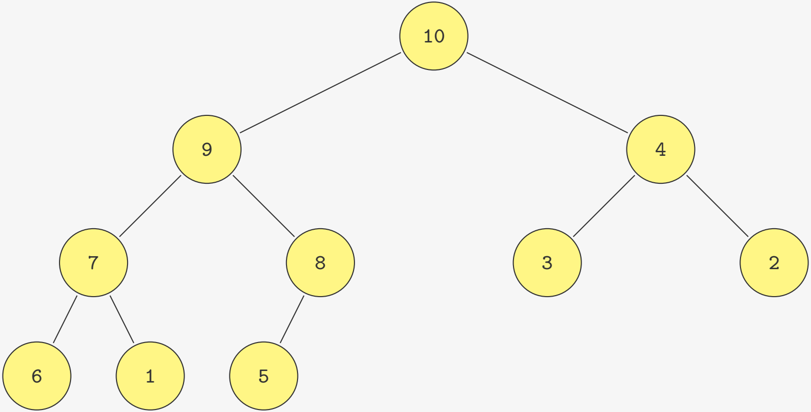
- Trocamos a raiz com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o maior dos dois filhos (se necessário)

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

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

Extraindo o Máximo

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

Extraindo o Máximo

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

Extraindo o Máximo

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

Extraindo o Máximo

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

Extraindo o Máximo

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

Extraindo o Máximo

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

Extraindo o Máximo

```c
Item extrai_maximo(p_fp fprio) {
  Item item = fprio->v[0];
  troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
  fprio->n--;
  desce_no_heap(fprio, 0);
  return item;
}

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

```

Extraindo o Máximo

```c
Item extrai_maximo(p_fp fprio) {
  Item item = fprio->v[0];
  troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
  fprio->n--;
  desce_no_heap(fprio, 0);
  return item;
}

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
```

Extraindo o Máximo

```c
Item extrai_maximo(p_fp fprio) {
  Item item = fprio->v[0];
  troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
  fprio->n--;
  desce_no_heap(fprio, 0);
  return item;
}

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
  int maior_filho;
```

Extraindo o Máximo

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

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
  int maior_filho;
  if (F_ESQ(k) < fprio->n) {
```

Extraindo o Máximo

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

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
  int maior_filho;
  if (F_ESQ(k) < fprio->n) {
    maior_filho = F_ESQ(k);
```

Extraindo o Máximo

```c
Item extrai_maximo(p_fp fprio) {
  Item item = fprio->v[0];
  troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
  fprio->n--;
  desce_no_heap(fprio, 0);
  return item;
}

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
  int maior_filho;
  if (F_ESQ(k) < fprio->n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < fprio->n &&
```

Extraindo o Máximo

```c
Item extrai_maximo(p_fp fprio) {
  Item item = fprio->v[0];
  troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
  fprio->n--;
  desce_no_heap(fprio, 0);
  return item;
}

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
  int maior_filho;
  if (F_ESQ(k) < fprio->n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < fprio->n &&
        fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
```

Extraindo o Máximo

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

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
  int maior_filho;
  if (F_ESQ(k) < fprio->n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < fprio->n &&
        fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
      maior_filho = F_DIR(k);
```

Extraindo o Máximo

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

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
  int maior_filho;
  if (F_ESQ(k) < fprio->n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < fprio->n &&
        fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
      maior_filho = F_DIR(k);
    if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
```

Extraindo o Máximo

```c
Item extrai_maximo(p_fp fprio) {
  Item item = fprio->v[0];
  troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
  fprio->n--;
  desce_no_heap(fprio, 0);
  return item;
}

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
  int maior_filho;
  if (F_ESQ(k) < fprio->n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < fprio->n &&
        fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
      maior_filho = F_DIR(k);
    if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
      troca(&fprio->v[k], &fprio->v[maior_filho]);
```

Extraindo o Máximo

```c
Item extrai_maximo(p_fp fprio) {
  Item item = fprio->v[0];
  troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
  fprio->n--;
  desce_no_heap(fprio, 0);
  return item;
}

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
  int maior_filho;
  if (F_ESQ(k) < fprio->n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < fprio->n &&
        fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
      maior_filho = F_DIR(k);
    if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
      troca(&fprio->v[k], &fprio->v[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}
```

Extraindo o Máximo

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

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
  int maior_filho;
  if (F_ESQ(k) < fprio->n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < fprio->n &&
        fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
      maior_filho = F_DIR(k);
    if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
      troca(&fprio->v[k], &fprio->v[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}
```

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

```c
void muda_prioridade(p_fp fprio, int k, int valor) {
  if (fprio->v[k].chave < valor) {
    fprio->v[k].chave = valor;
    sobe_no_heap(fprio, k);
  } else {
    fprio->v[k].chave = valor;
    desce_no_heap(fprio, k);
  }
}
```

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

```c
void muda_prioridade(p_fp fprio, int k, int valor) {
  if (fprio->v[k].chave < valor) {
    fprio->v[k].chave = valor;
    sobe_no_heap(fprio, k);
  } else {
    fprio->v[k].chave = valor;
    desce_no_heap(fprio, k);
  }
}
```

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

Ordenação usando Fila de Prioridades

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

Ordenação usando Fila de Prioridades

```c
void fpsort(Item *v, int l, int r) {
  int i;
  p_fp fprio = criar_fprio(r-l+1);
  for (i = l; i <= r; i++)
    insere(fprio, v[i]);
  for (i = r; i >= l; i--)
    v[i] = extrai_maximo(fprio);
}
```

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

Ordenação usando Fila de Prioridades

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

Tempo: $O(n \lg n)$

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

Ordenação usando Fila de Prioridades

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

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

```c
void fpsort(Item *v, int l, int r) {
  int i;
  p_fp fprio = criar_fprio(r-l+1);
  for (i = l; i <= r; i++)
    insere(fprio, v[i]);
  for (i = r; i >= l; i--)
    v[i] = extrai_maximo(fprio);
}
```

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

```c
void fpsort(Item *v, int l, int r) {
  int i;
  p_fp fprio = criar_fprio(r-l+1);
  for (i = l; i <= r; i++)
    insere(fprio, v[i]);
  for (i = r; i >= l; i--)
    v[i] = extrai_maximo(fprio);
}
```

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

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

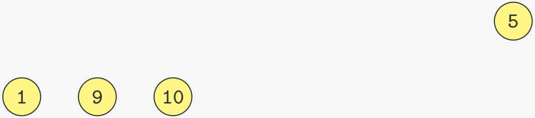
4	8	2	7	6	3	5	1	9	10

Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10

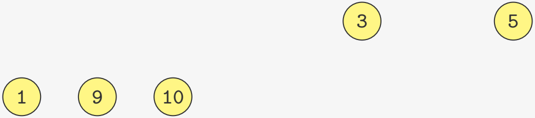
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



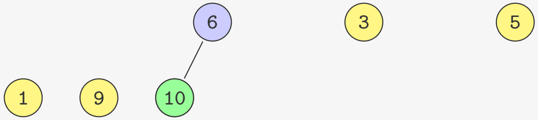
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



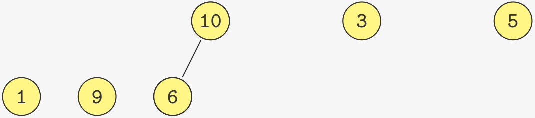
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



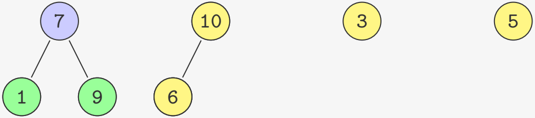
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



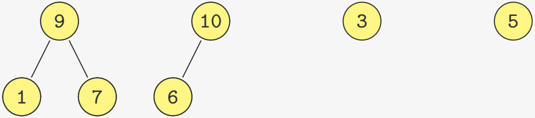
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



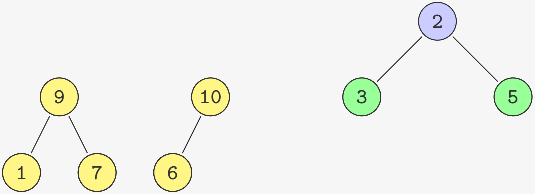
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



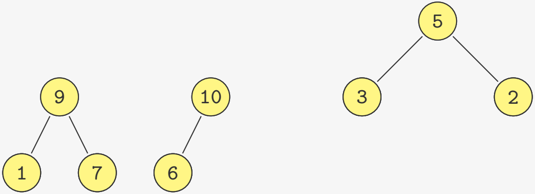
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



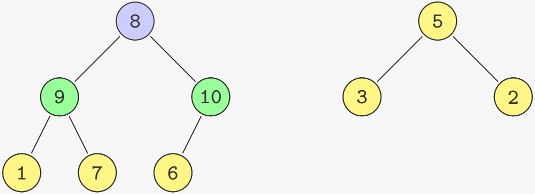
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



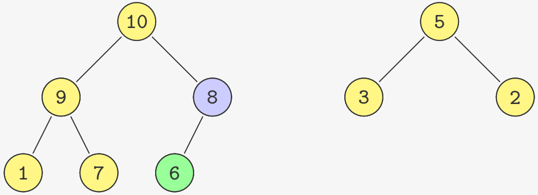
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



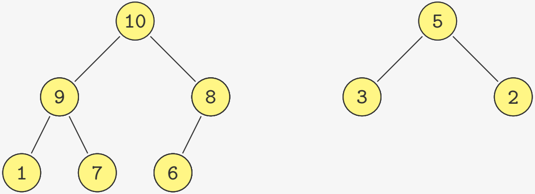
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



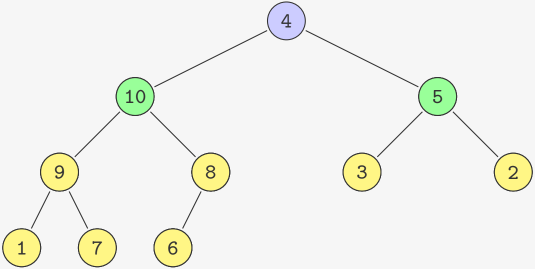
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10

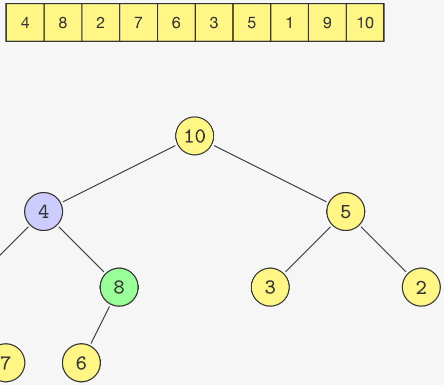


Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10

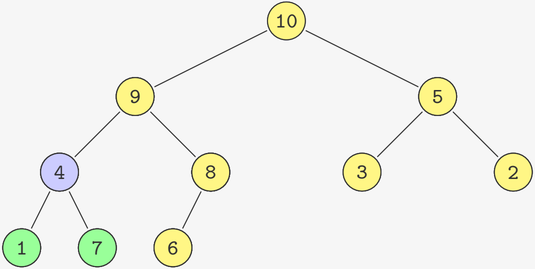


Transformando um vetor em um heap



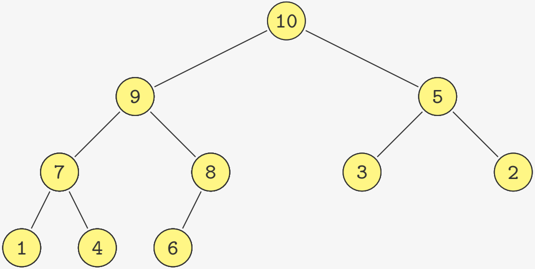
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



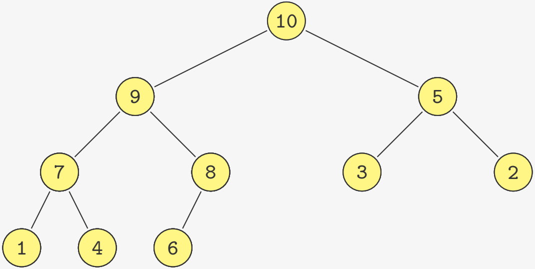
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10



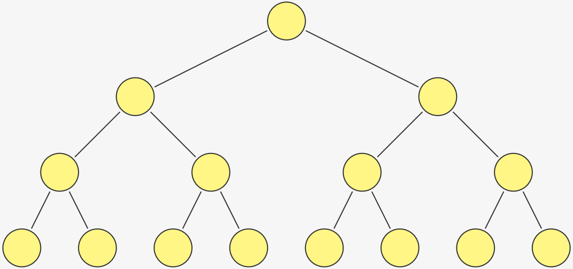
Transformando um vetor em um heap

4	8	2	7	6	3	5	1	9	10

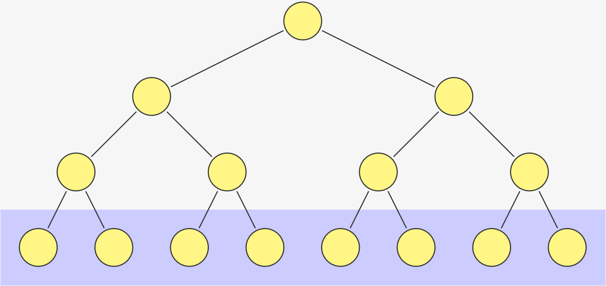


Quanto tempo demora?

Tempo da construção para $n = 2^k$

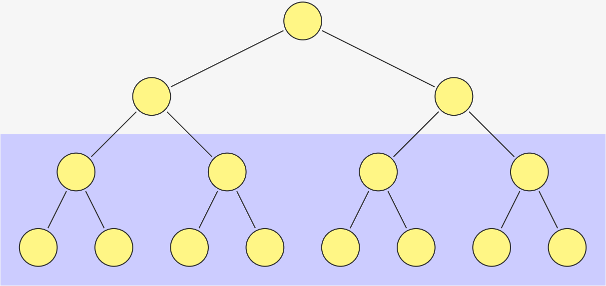


Tempo da construção para $n = 2^k$



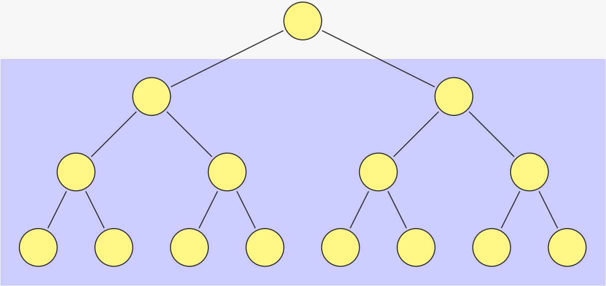
- Temos $n/2 = 2^{k-1}$ heaps de altura 1

Tempo da construção para $n = 2^k$



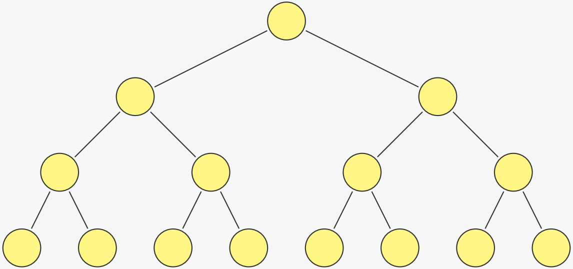
- Temos $n/2 = 2^{k-1}$ heaps de altura 1
- Temos $n/4 = 2^{k-2}$ heaps de altura 2

Tempo da construção para $n = 2^k$



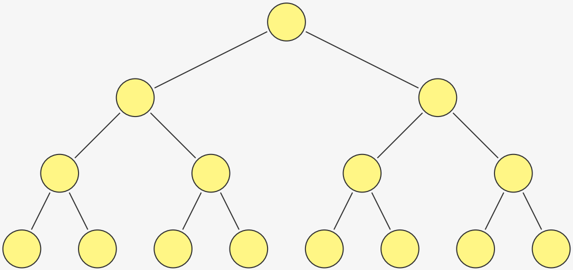
- Temos $n/2 = 2^{k-1}$ heaps de altura 1
- Temos $n/4 = 2^{k-2}$ heaps de altura 2
- Temos $n/2^{h+1} = 2^{k-h-1}$ heaps de altura h

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 $c \cdot h$

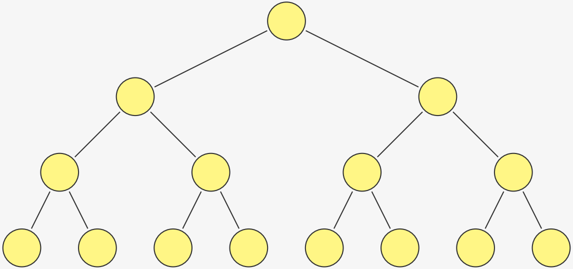
Tempo da construção para $n = 2^k$



- 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

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

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

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

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

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

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

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

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

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

Ou seja,

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

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

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

Ou seja,

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

Heapsort

```
void desce_no_heap(int *heap, int n, int k) {
  int maior_filho;
  if (F_ESQ(k) < n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < n &&
        heap[F_ESQ(k)] < heap[F_DIR(k)])
      maior_filho = F_DIR(k);
    if (heap[k] < heap[maior_filho]) {
      troca(&heap[k], &heap[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}
```

Heapsort

```c
void desce_no_heap(int *heap, int n, int k) {
  int maior_filho;
  if (F_ESQ(k) < n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < n &&
        heap[F_ESQ(k)] < heap[F_DIR(k)])
      maior_filho = F_DIR(k);
    if (heap[k] < heap[maior_filho]) {
      troca(&heap[k], &heap[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}

void heapsort(int *v, int l, int r) {
```

Heapsort

```
void desce_no_heap(int *heap, int n, int k) {
  int maior_filho;
  if (F_ESQ(k) < n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < n &&
        heap[F_ESQ(k)] < heap[F_DIR(k)])
      maior_filho = F_DIR(k);
    if (heap[k] < heap[maior_filho]) {
      troca(&heap[k], &heap[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}

void heapsort(int *v, int l, int r) {
  int k, n = r-l+1;
```

Heapsort

```c
void desce_no_heap(int *heap, int n, int k) {
  int maior_filho;
  if (F_ESQ(k) < n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < n &&
        heap[F_ESQ(k)] < heap[F_DIR(k)])
      maior_filho = F_DIR(k);
    if (heap[k] < heap[maior_filho]) {
      troca(&heap[k], &heap[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}

void heapsort(int *v, int l, int r) {
  int k, n = r-l+1;
  int *heap = &v[l];
```

Heapsort

```c
void desce_no_heap(int *heap, int n, int k) {
  int maior_filho;
  if (F_ESQ(k) < n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < n &&
        heap[F_ESQ(k)] < heap[F_DIR(k)])
      maior_filho = F_DIR(k);
    if (heap[k] < heap[maior_filho]) {
      troca(&heap[k], &heap[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}

void heapsort(int *v, int l, int r) {
  int k, n = r-l+1;
  int *heap = &v[l];
  for (k = n/2; k >= 1; k--) /* transforma em heap */
```

Heapsort

```
void desce_no_heap(int *heap, int n, int k) {
  int maior_filho;
  if (F_ESQ(k) < n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < n &&
        heap[F_ESQ(k)] < heap[F_DIR(k)])
      maior_filho = F_DIR(k);
    if (heap[k] < heap[maior_filho]) {
      troca(&heap[k], &heap[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}

void heapsort(int *v, int l, int r) {
  int k, n = r-l+1;
  int *heap = &v[l];
  for (k = n/2; k >= 1; k--) /* transforma em heap */
    desce_no_heap(heap, n, k);
```

Heapsort

```c
void desce_no_heap(int *heap, int n, int k) {
  int maior_filho;
  if (F_ESQ(k) < n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < n &&
        heap[F_ESQ(k)] < heap[F_DIR(k)])
      maior_filho = F_DIR(k);
    if (heap[k] < heap[maior_filho]) {
      troca(&heap[k], &heap[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}

void heapsort(int *v, int l, int r) {
  int k, n = r-l+1;
  int *heap = &v[l];
  for (k = n/2; k >= 1; k--) /* transforma em heap */
    desce_no_heap(heap, n, k);
  while (n > 1) { /* extrai o máximo */
```

Heapsort

```c
void desce_no_heap(int *heap, int n, int k) {
  int maior_filho;
  if (F_ESQ(k) < n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < n &&
        heap[F_ESQ(k)] < heap[F_DIR(k)])
      maior_filho = F_DIR(k);
    if (heap[k] < heap[maior_filho]) {
      troca(&heap[k], &heap[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}

void heapsort(int *v, int l, int r) {
  int k, n = r-l+1;
  int *heap = &v[l];
  for (k = n/2; k >= 1; k--) /* transforma em heap */
    desce_no_heap(heap, n, k);
  while (n > 1) { /* extrai o máximo */
    troca(&heap[0], &heap[fprio->n - 1]);
```

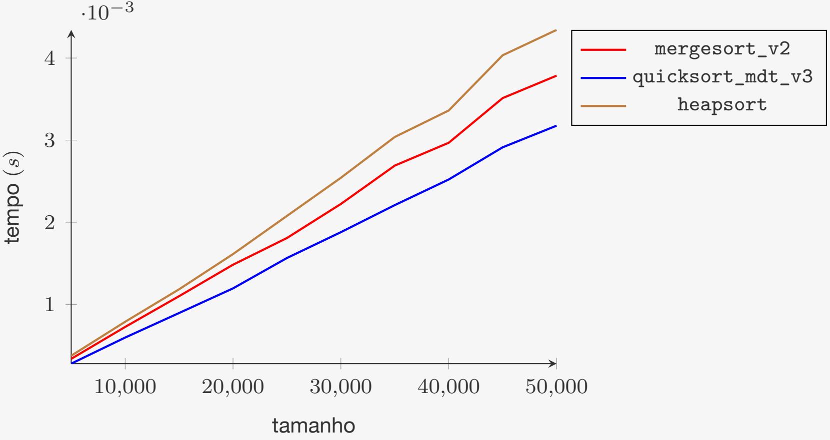
Heapsort

```
void desce_no_heap(int *heap, int n, int k) {
  int maior_filho;
  if (F_ESQ(k) < n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < n &&
        heap[F_ESQ(k)] < heap[F_DIR(k)])
      maior_filho = F_DIR(k);
    if (heap[k] < heap[maior_filho]) {
      troca(&heap[k], &heap[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}

void heapsort(int *v, int l, int r) {
  int k, n = r-l+1;
  int *heap = &v[l];
  for (k = n/2; k >= 1; k--) /* transforma em heap */
    desce_no_heap(heap, n, k);
  while (n > 1) { /* extrai o máximo */
    troca(&heap[0], &heap[fprio->n - 1]);
    n--;
```

Heapsort

```c
void desce_no_heap(int *heap, int n, int k) {
  int maior_filho;
  if (F_ESQ(k) < n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < n &&
        heap[F_ESQ(k)] < heap[F_DIR(k)])
      maior_filho = F_DIR(k);
    if (heap[k] < heap[maior_filho]) {
      troca(&heap[k], &heap[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}

void heapsort(int *v, int l, int r) {
  int k, n = r-l+1;
  int *heap = &v[l];
  for (k = n/2; k >= 1; k--) /* transforma em heap */
    desce_no_heap(heap, n, k);
  while (n > 1) { /* extrai o máximo */
    troca(&heap[0], &heap[fprio->n - 1]);
    n--;
    desce_no_heap(heap, n, 0);
  }
}
```

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).