Estruturas de Dados

Fila de Prioridade e Heap

Aula 09

Prof. Felipe A. Louza

Roteiro

Roteiro

Fila de Prioridade

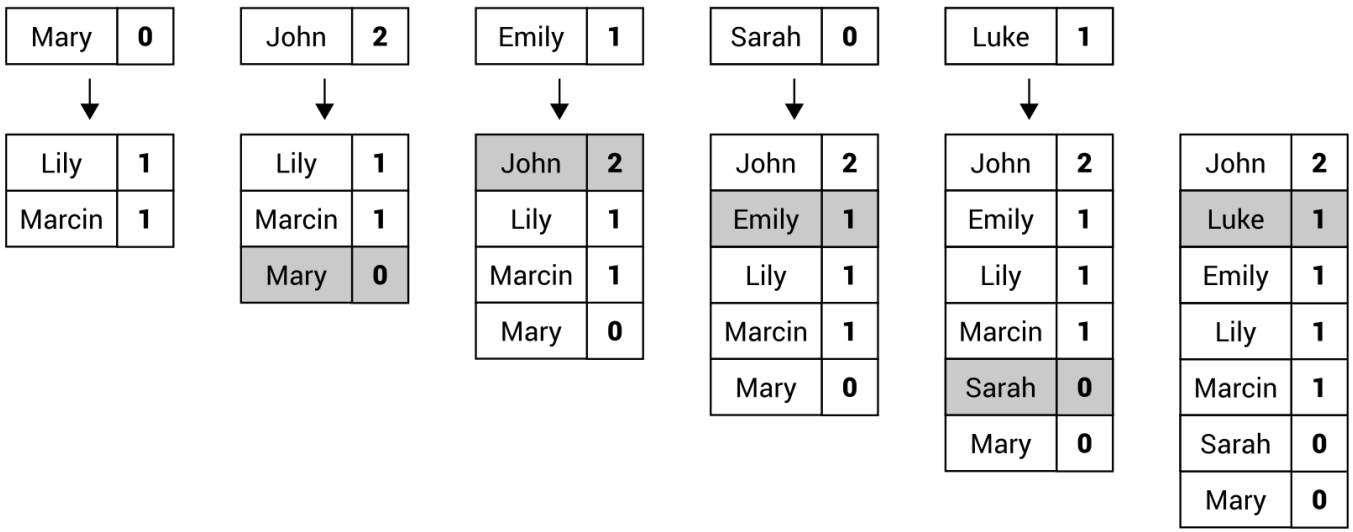
Uma fila de prioridades é uma estrutura de dados com duas operações básicas:

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

Não podemos acessar elementos do "meio" da fila

Fila de Prioridade

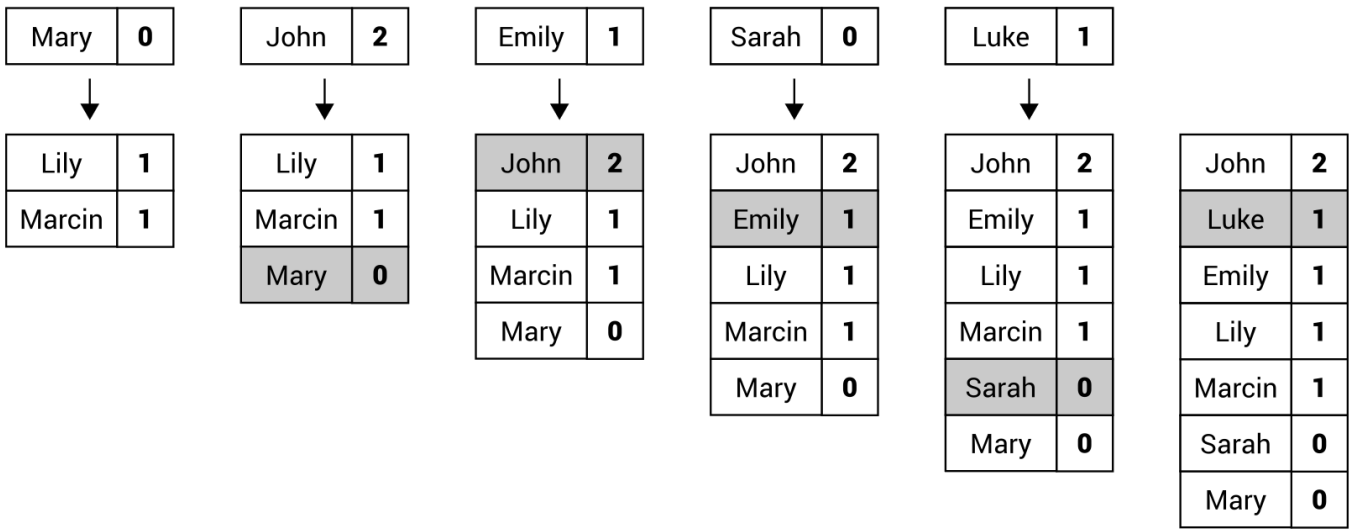
Um exemplo de fila de prioridade (valor, chave):



Fonte: C# Data Structures and Algorithms, Marcin Jamro.

Fila de Prioridade

Um exemplo de fila de prioridade (valor, chave):



Vamos ver duas implementações para filas de prioridades:

1. Utilizando um vetor;
2. Utilizando uma árvore binária

Fonte: C# Data Structures and Algorithms, Marcin Jamro.

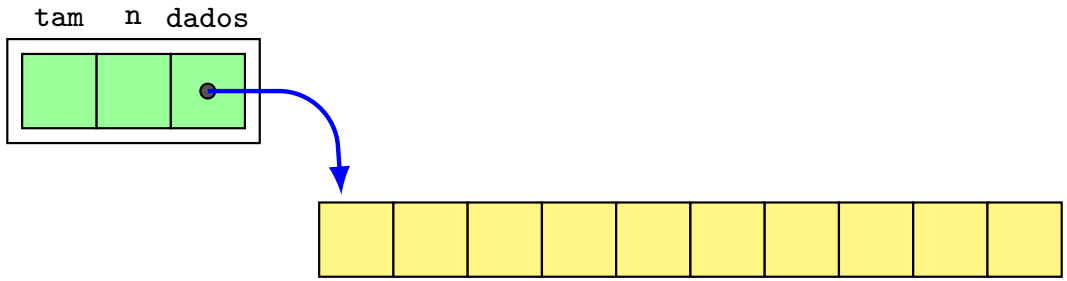
TAD - Interface (implementação com vetor)

pq_vetor.h

```
1  #ifndef PQ_VETOR_H
2  #define PQ_VETOR_H
3
4  //Dados
5  typedef struct {
6    char nome[20];
7    int chave;
8  } t_item;
9
10 typedef struct {
11   t_item *dados;
12   int tam, n;
13 } PQ;
```

```
15 //Funções
16 PQ*  pq_criar(int tam);
17 void pq_destruir(PQ **p);
18
19 void pq_adicionar(PQ *p, t_item x);
20 t_item pq_extrai_maximo(PQ *p);
21
22 int pq_vazia(PQ* p);
23 int pq_cheia(PQ* p);
24
25 #endif
```

Fila de Prioridade - Criar e Destruir



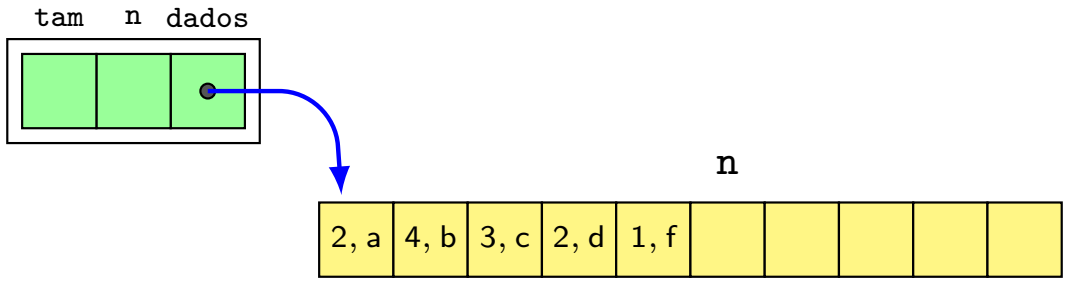
pq_vetor.c

```
7  PQ* pq_criar(int tam){
8    PQ* p = (PQ*) malloc(sizeof(PQ));
9    p->dados = (t_item*) malloc(tam * sizeof(t_item));
10   p->n = 0;
11   p->tam = tam;
12 return p;
13 }
```

```
15 void pq_destruir(PQ **p) {
16   free((*p)->dados);
17   free(*p);
18   *p = NULL;
19 }
```

Código do cliente: `PQ *Fila = pq_criar(10);`

Fila de Prioridade - Vazia ou Cheia



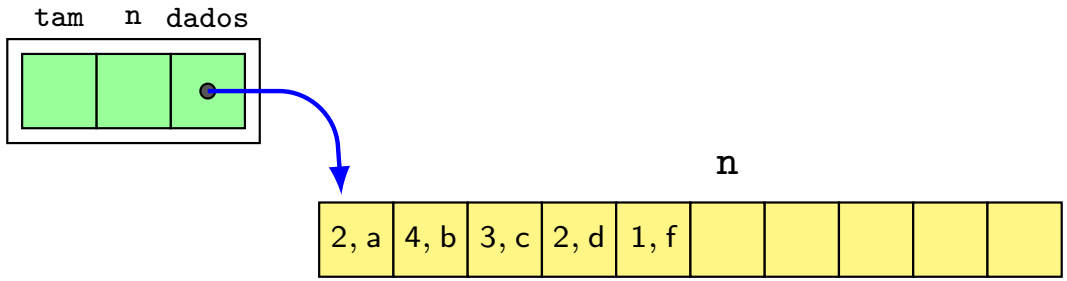
pq_vetor.c

```
43 int pq_vazia(PQ *p) {
44   return p->n == 0;
45 }
```

```
47 int pq_cheia(PQ *p) {
48   return p->n == p->tam;
49 }
```

Verificamos os valores na `struct`.

Fila de Prioridade - Inserir um valor

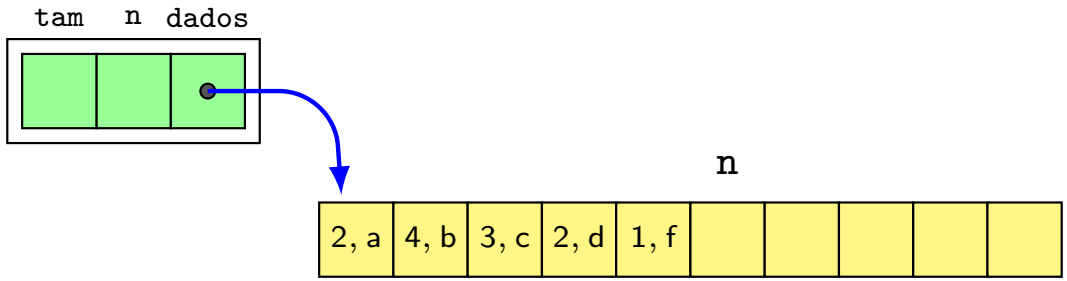


pq_vetor.c

```
21 void pq_adicionar(PQ *p, t_item item) {
22   if(!pq_cheia(p))
23     p->dados[p->n] = item;
24   p->n++;
25 }
```

Insere no final em $O(1)$, extrai o máximo em $O(n)$

Fila de Prioridade - Extrair o Máximo

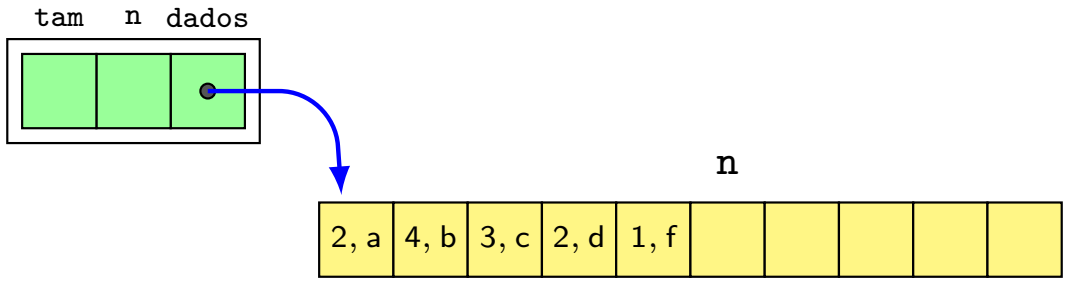


pq_vetor.c

```
27 t_item pq_extrai_maximo(PQ *p) {
28   int j, max = 0;
29   for (j = 1; j < p->n; j++)
30     if (p->dados[max].chave < p->dados[j].chave) max = j;
31   swap(&(p->dados[max]), &(p->dados[p->n-1]));
32   p->n--;
33 return p->dados[p->n];
34 }
```

Extrai o máximo em $O(n)$, insere em $O(1)$

Fila de Prioridade - Extrair o Máximo



pq_vetor.c

```
27 t_item pq_extrai_maximo(PQ *p) {
28   int j, max = 0;
29   for (j = 1; j < p->n; j++)
30     if (p->dados[max].chave < p->dados[j].chave) max = j;
31   swap(&(p->dados[max]), &(p->dados[p->n-1]));
32   p->n--;
33 return p->dados[p->n];
34 }
```

Extrai o máximo em $O(n)$, insere em $O(1)$

- Se mantiver o vetor ordenado, os tempos se invertem

Cliente

exemplo1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "pq_vetor.h"

int main() {
  int n, i;
  scanf("%d", &n);  //valor de n
  PQ *Fila = pq_criar(n);
  for (i = 0; i < n; i++) {
    t_item item;
    scanf("%d %s", &item.chave, item.nome); //chave nome
    pq_adicionar(Fila, item);
  }
  printf("Fila:\n");
  while(!pq_vazia(Fila)) {
    t_item item = pq_extrai_maximo(Fila);
    printf("%d %s\n", item.chave, item.nome);
  }
  pq_destruir(&Fila);
return 0;
}
```

Makefile

Vamos usar o Makefile para compilar:

```
exemplo1: exemplo1.c pq_vetor.o
  gcc $^ -o $@
```

Vamos executar:

```
$ ./exemplo1
5
2 a
4 b
3 c
2 d
1 f
Fila:
4 b
3 c
2 a
2 d
1 f
```

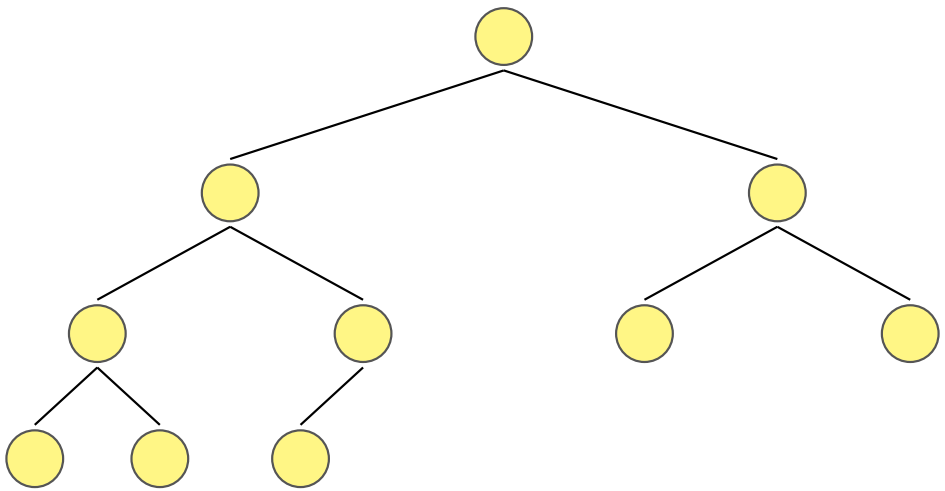
Roteiro

Árvores Binárias Completas

Uma árvore binária é quase-completa se:

- Todos os seus níveis estão **preenchidos**, exceto talvez pelas folhas à direita do ultimo nível

Exemplo:

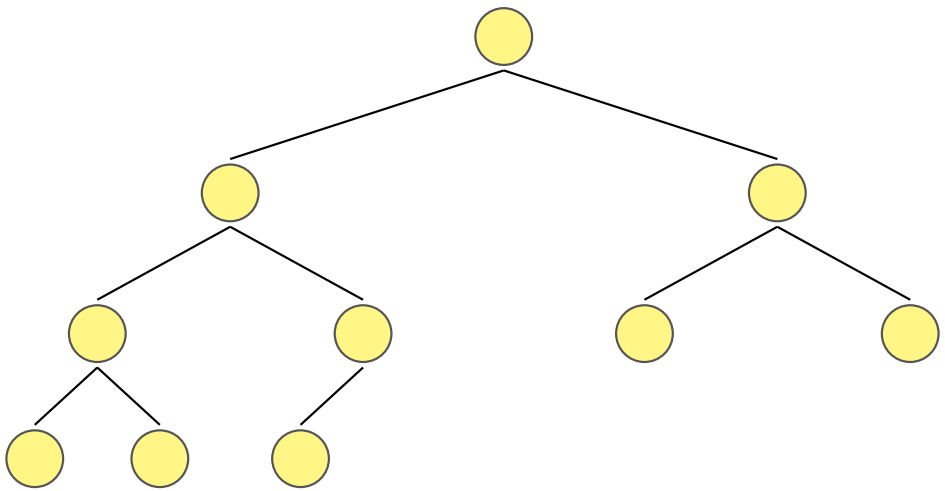


Árvores Binárias Completas

Uma árvore binária é quase-completa se:

- Todos os seus níveis estão **preenchidos**, exceto talvez pelas folhas à direita do ultimo nível

Exemplo:



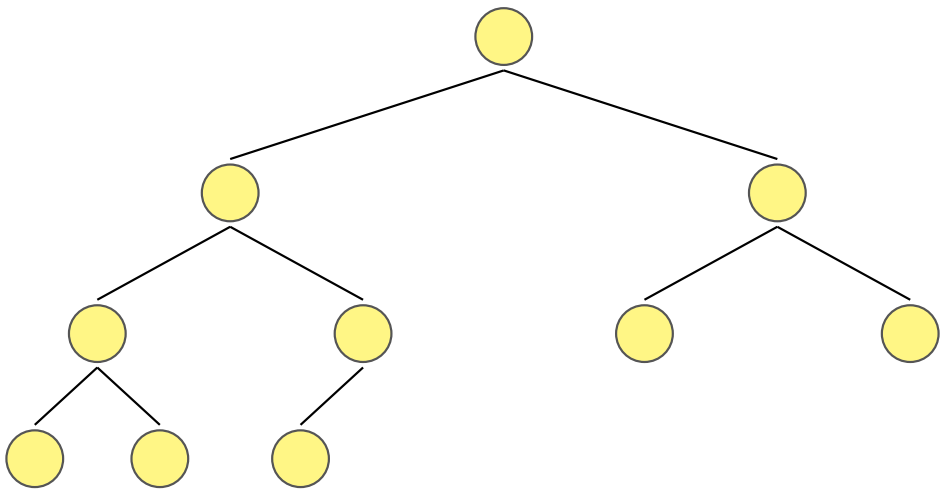
Quantos níveis tem uma árvore binária quase-completa com n nós?

Árvores Binárias Completas

Uma árvore binária é quase-completa se:

- Todos os seus níveis estão **preenchidos**, exceto talvez pelas folhas à direita do ultimo nível

Exemplo:



Quantos níveis tem uma árvore binária quase-completa com n nós?

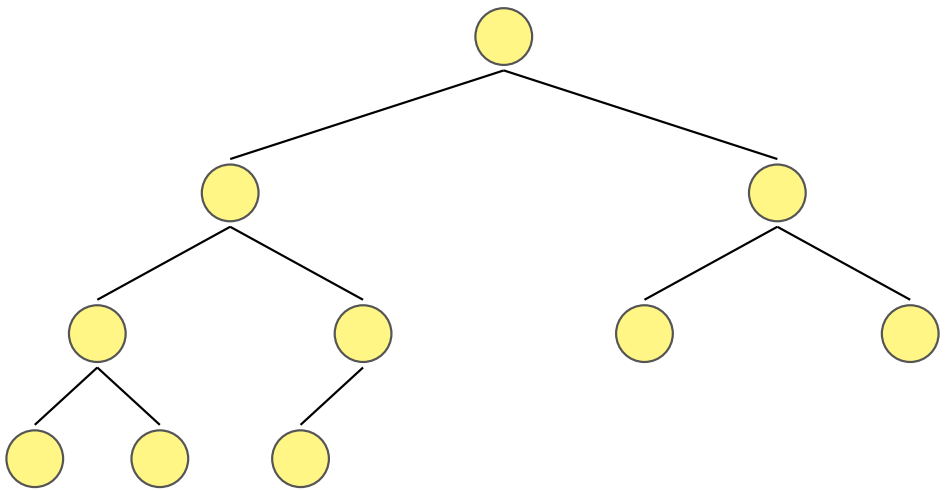
- $\lceil \lg(n+1) \rceil = \mathrm{O}(\lg n)$ níveis

Árvores Binárias Completas

Uma árvore binária é quase-completa se:

- Todos os seus níveis estão **preenchidos**, exceto talvez pelas folhas à direita do ultimo nível

Exemplo:



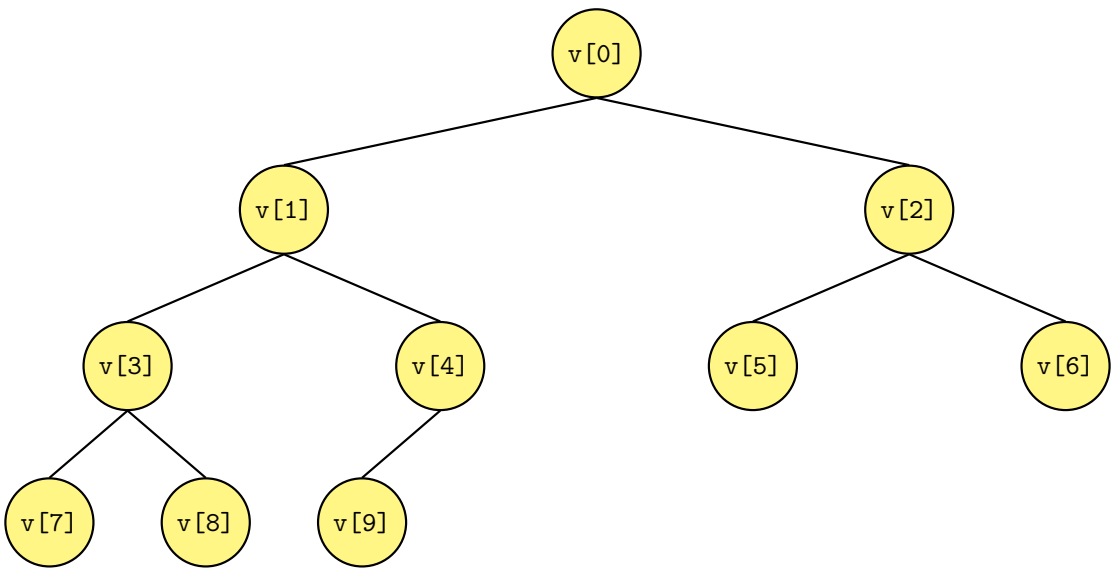
Quantos níveis tem uma árvore binária quase-completa com n nós?

- $\lceil \lg(n+1) \rceil = \mathrm{O}(\lg n)$ níveis
- Exemplo: com $n = 10$, $\lceil \lg(10+1) \rceil = \lceil 3,459 \rceil = 4$ níveis

Árvores Binárias Completas e Vetores

Podemos representar tais árvores usando vetores

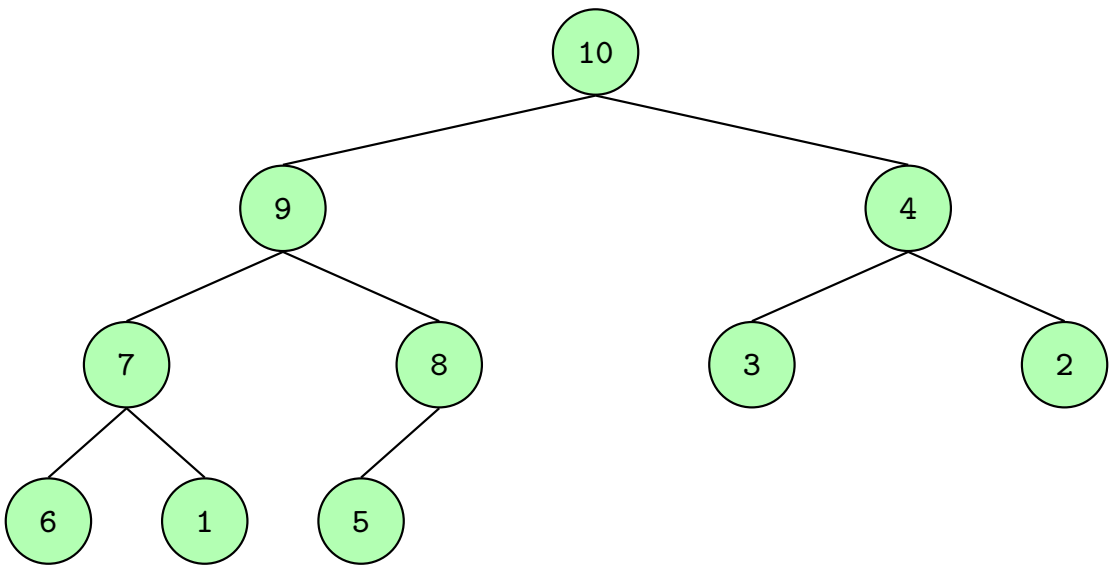
- Isso é, não precisamos de ponteiros
- Representação mais eficiente



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]`



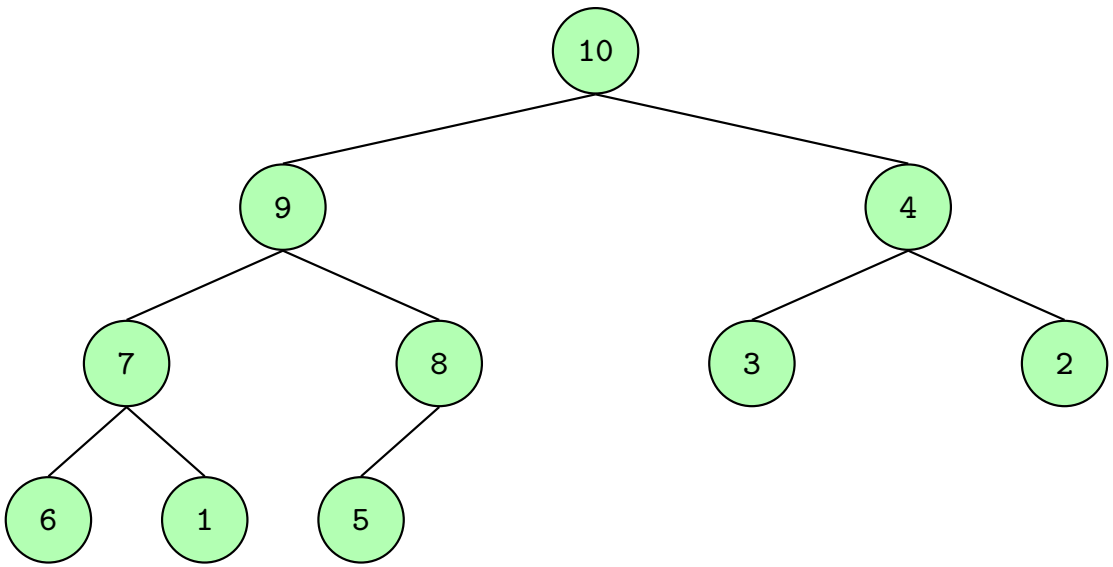
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	4	7	8	3	2	6	1	5	?	?	?	?	?	?

Roteiro

Max-Heap

Vamos definir uma ED chamada de Max-Heap (ou Heap de máximo):

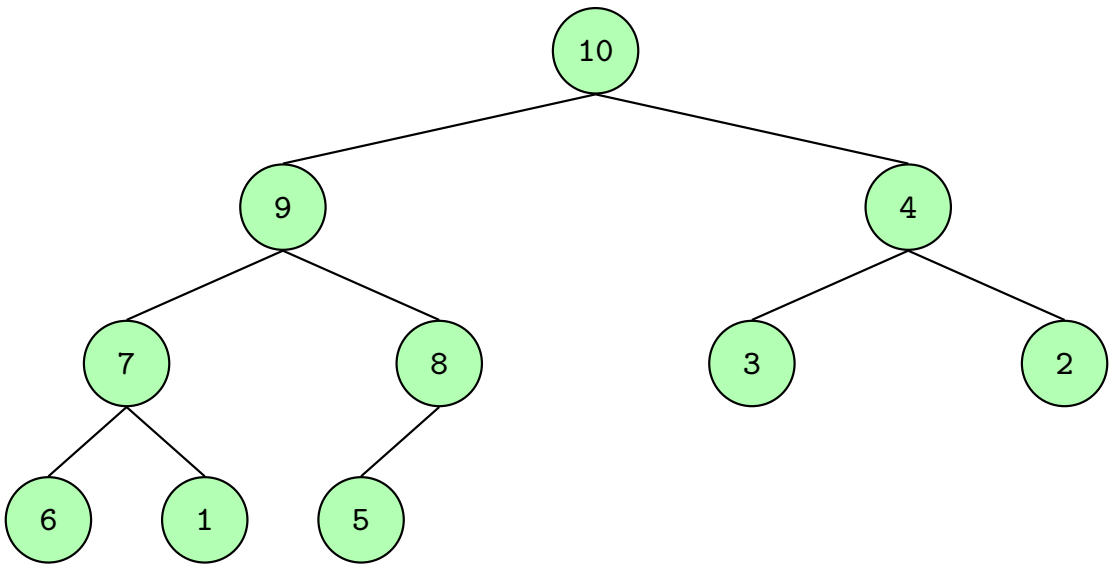
- Os filhos são menores ou iguais ao pai



Max-Heap

Vamos definir uma ED chamada de Max-Heap (ou Heap de máximo):

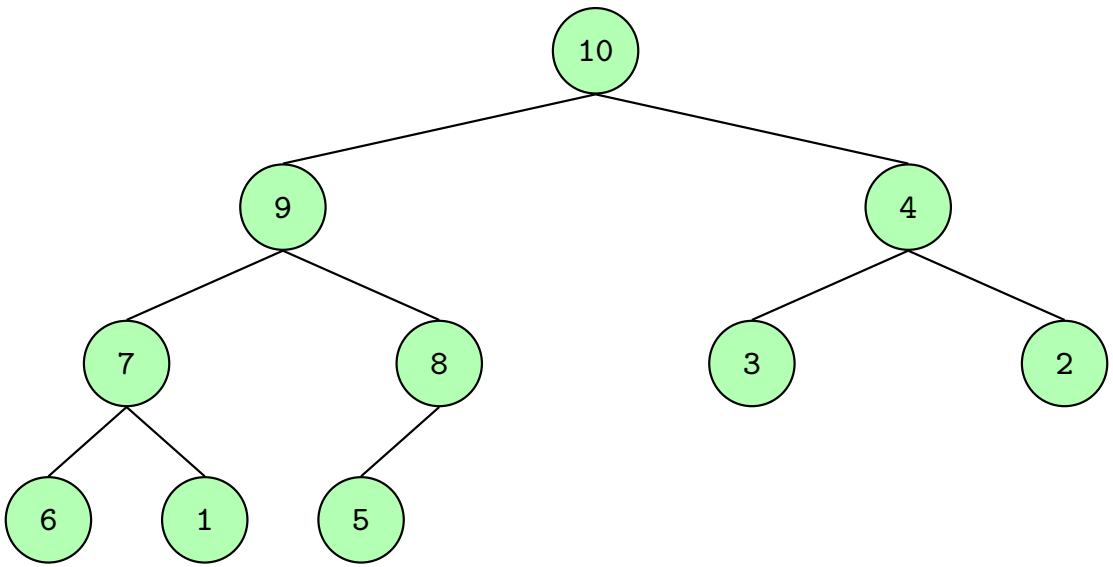
- Os filhos são menores ou iguais ao pai
- Com isso, a raiz contém o valor com maior chave



Max-Heap

Vamos definir uma ED chamada de Max-Heap (ou Heap de máximo):

- Os filhos são menores ou iguais ao pai
- Com isso, a raiz contém o valor com maior chave

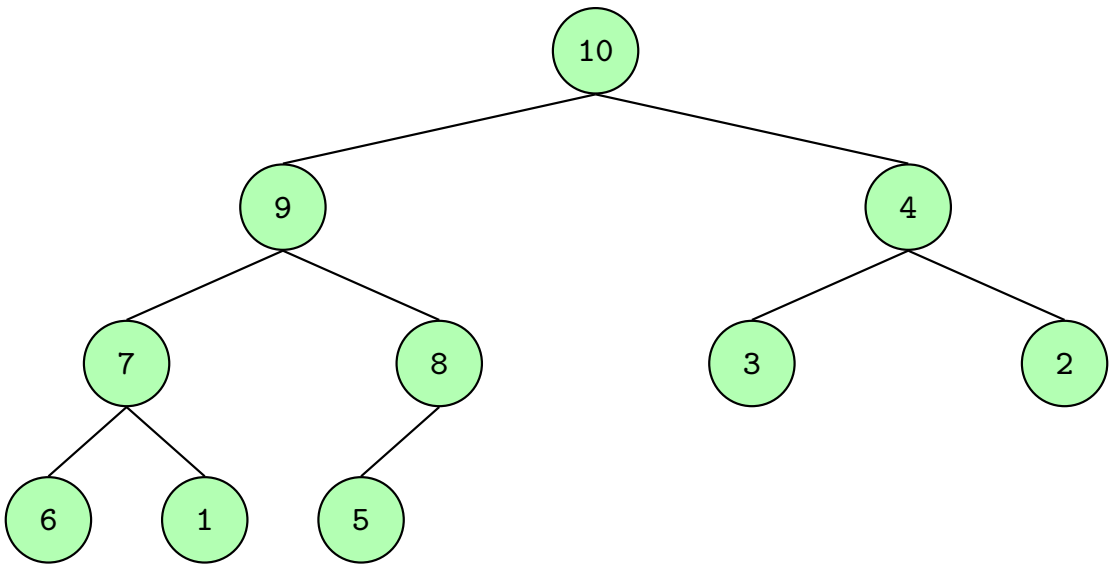


Note que **não** é uma ABB!

Max-Heap

Vamos definir uma ED chamada de Max-Heap (ou Heap de máximo):

- Os filhos são menores ou iguais ao pai
- Com isso, a raiz contém o valor com maior chave



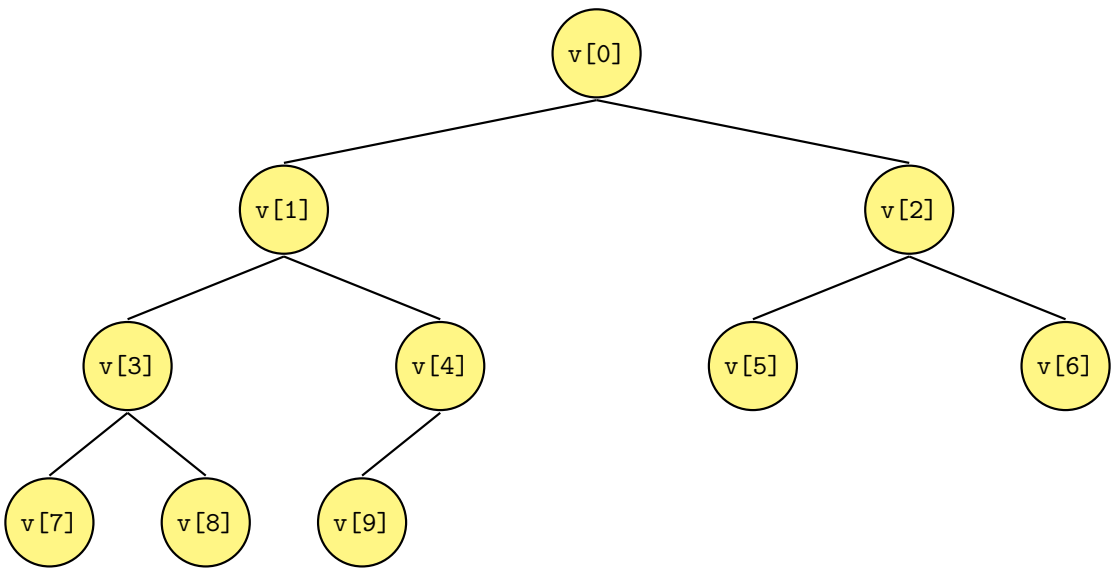
Note que **não** é uma ABB!

- Os dados estão bem menos estruturados (estamos interessados apenas no máximo)

Max-Heap

Uma propriedade importante:

- Um Heap é uma árvore binária quase completa

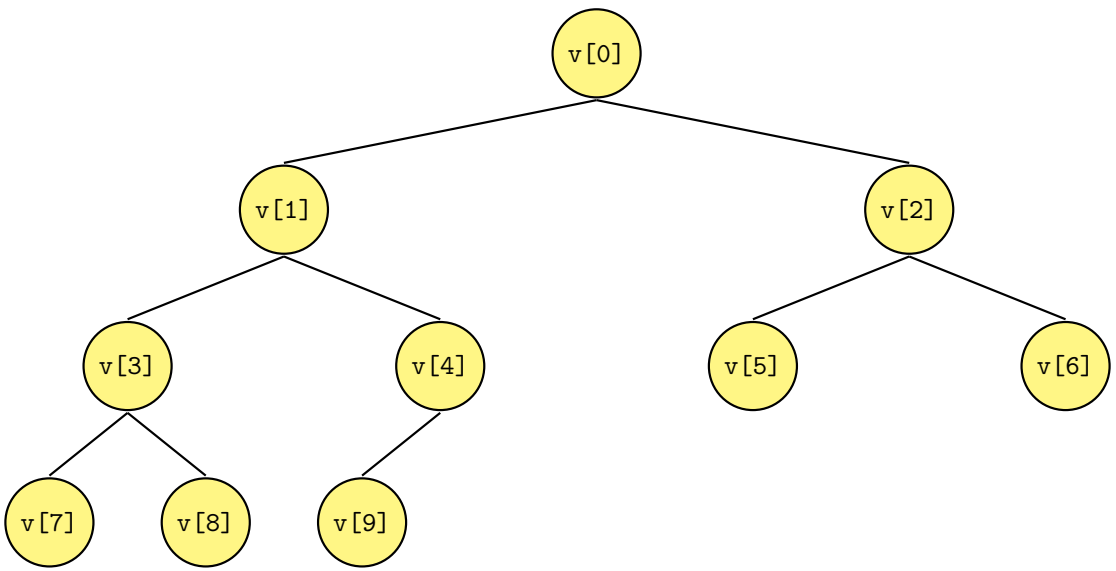


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	4	7	8	3	2	6	1	5	?	?	?	?	?	?

Max-Heap

Uma propriedade importante:

- Um Heap é uma árvore binária quase completa
- Vamos representá-lo usando vetores

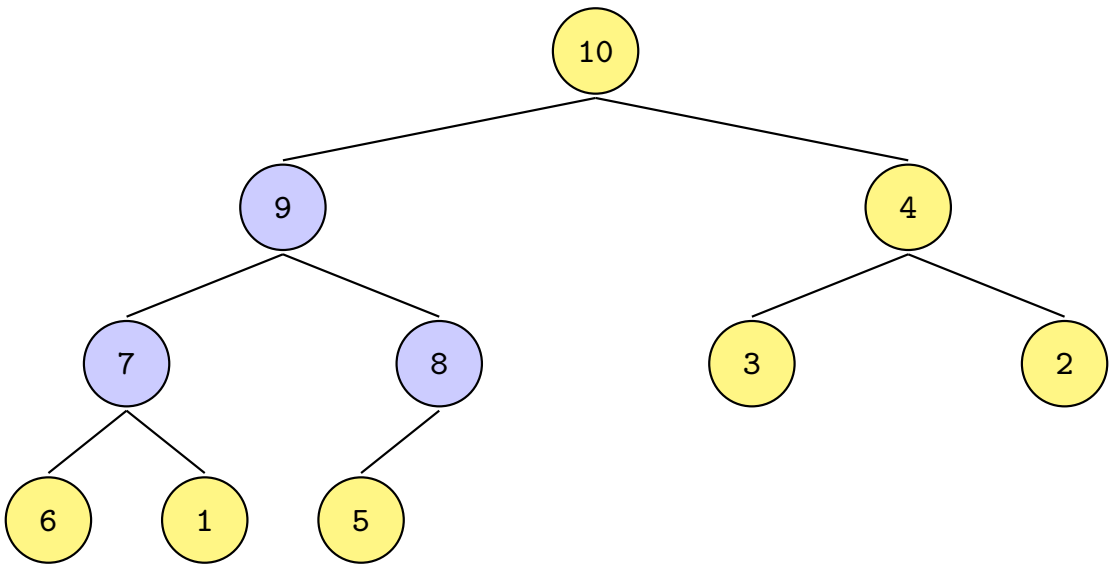


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	4	7	8	3	2	6	1	5	?	?	?	?	?	?

Max-Heap

pq_heap.c

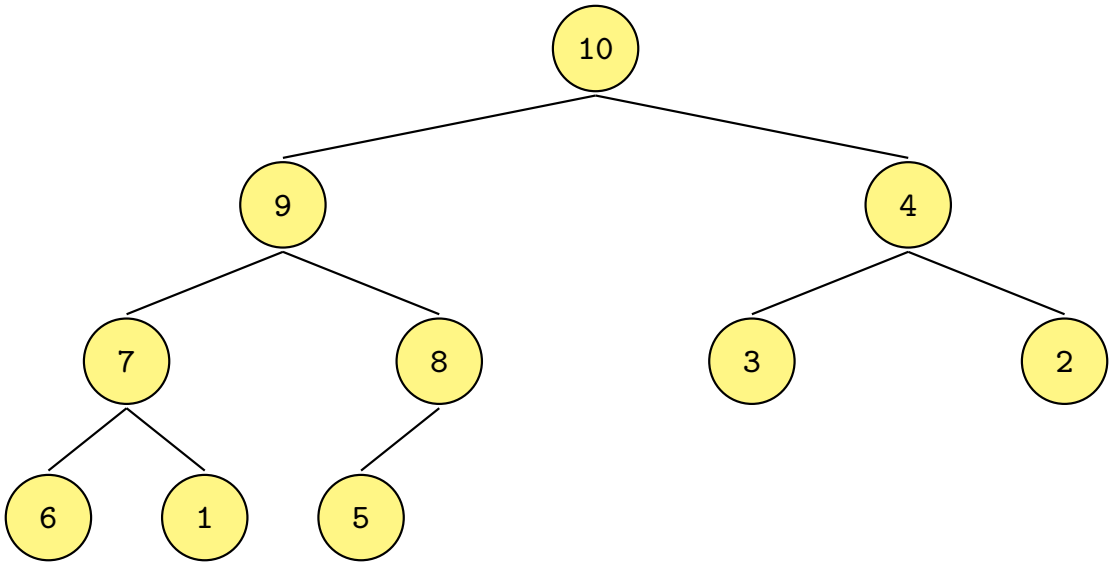
```
5 #define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
6 #define F_DIR(i) (2*i+2) /*Filho direito de i*/
7 #define PAI(i) ((i-1)/2)
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	4	7	8	3	2	6	1	5	?	?	?	?	?	?

Heap - Inserir um valor

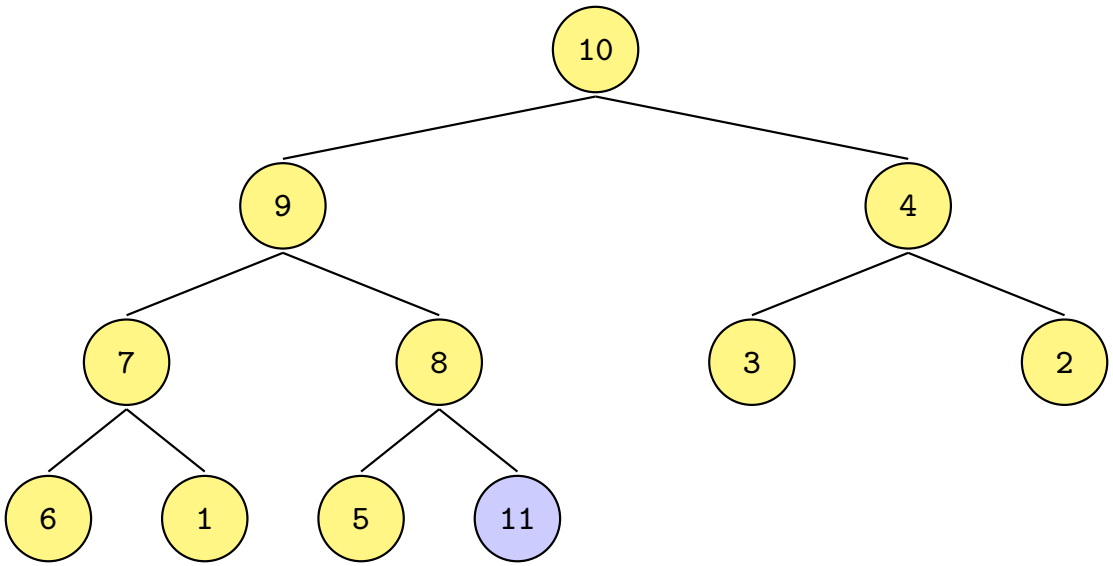
Como inserir no Heap?



Basta inserir na 1ª folha `NULL` e ir subindo no Heap, trocando com o pai se necessário

Heap - Inserir um valor

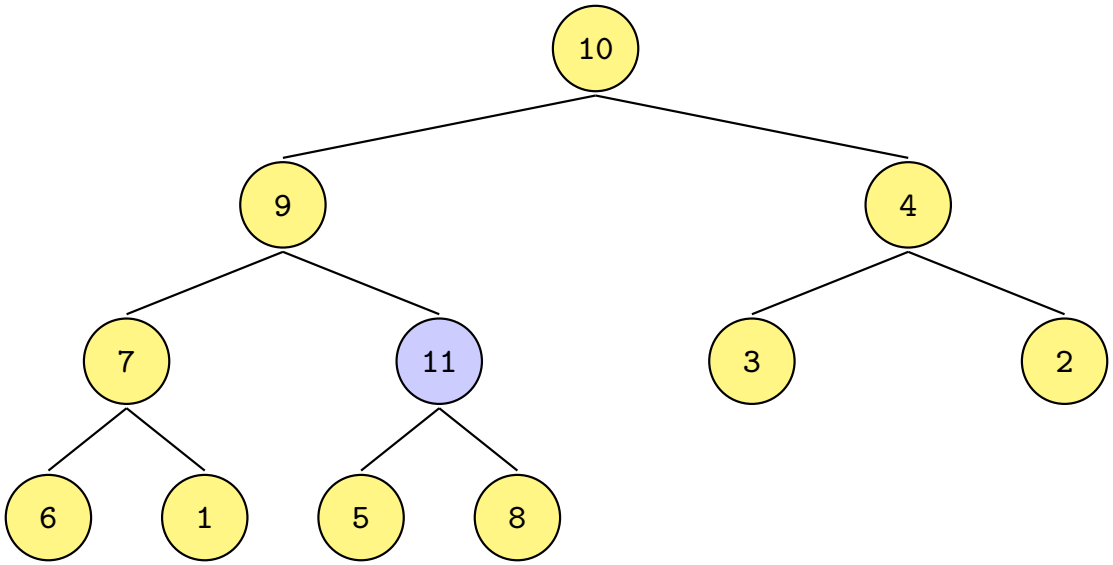
Como inserir no Heap?



Basta inserir na 1ª folha `NULL` e ir subindo no Heap, trocando com o pai se necessário

Heap - Inserir um valor

Como inserir no Heap?

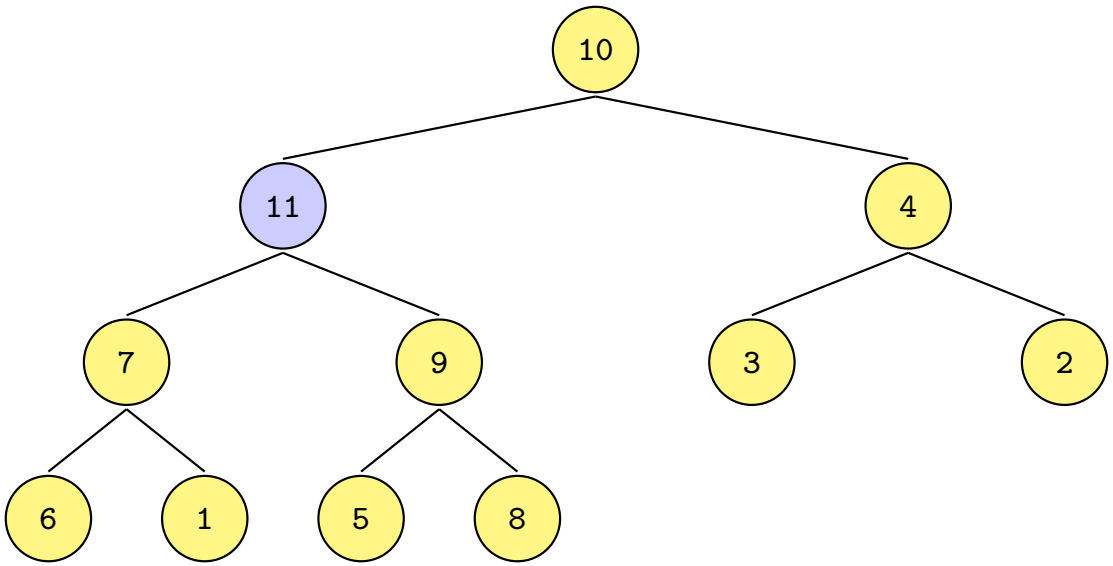


Basta inserir na 1ª folha `NULL` e ir subindo no Heap, trocando com o pai se necessário

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

Heap - Inserir um valor

Como inserir no Heap?

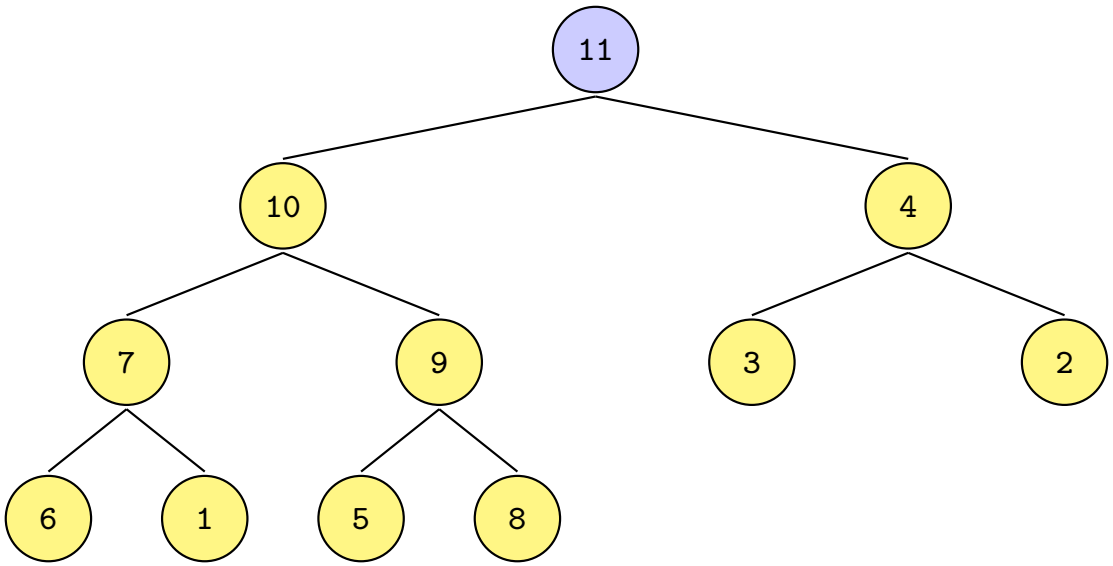


Basta inserir na 1ª folha `NULL` e ir subindo no Heap, trocando com o pai se necessário

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

Heap - Inserir um valor

Como inserir no Heap?

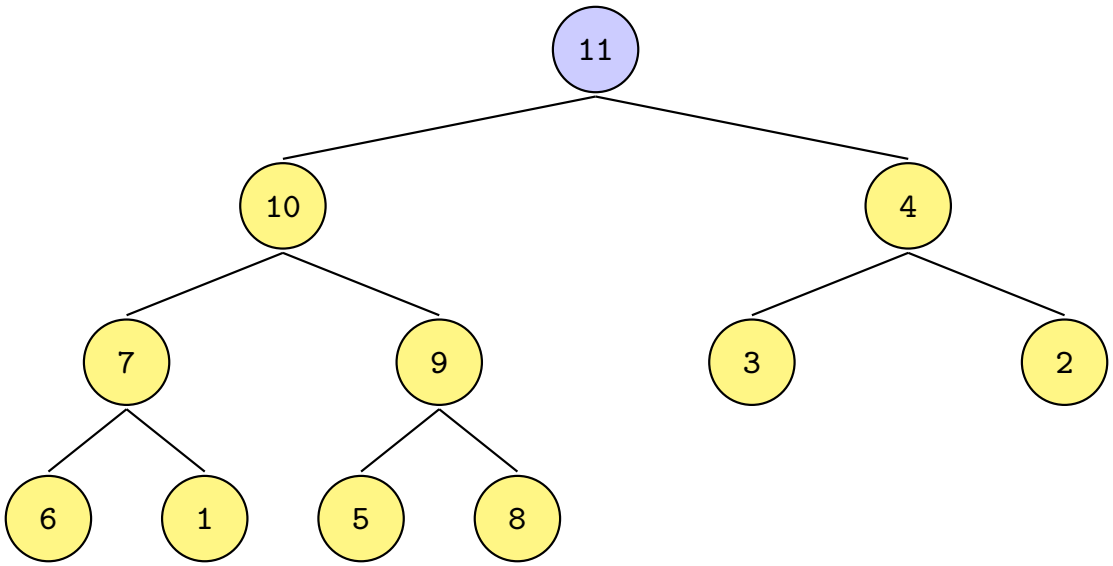


Basta inserir na 1ª folha `NULL` e ir subindo no Heap, trocando com o pai se necessário

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

Heap - Inserir um valor

Como inserir no Heap?

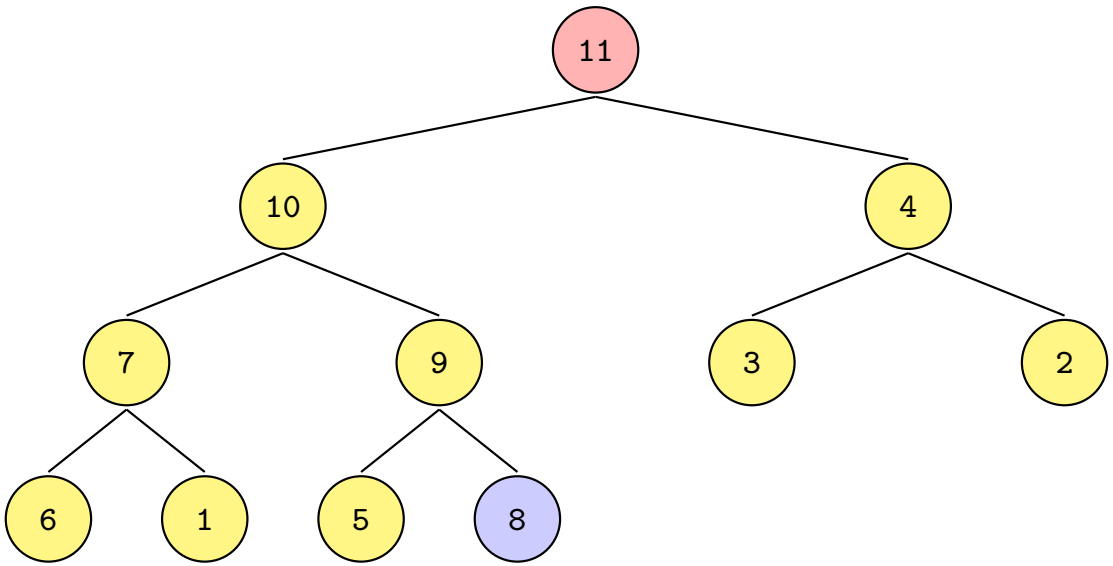


Basta inserir na 1ª folha `NULL` e ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, **não** precisamos mexer nele
- Custo computacional: $O(\lg n)$

Heap - Extraindo o Máximo

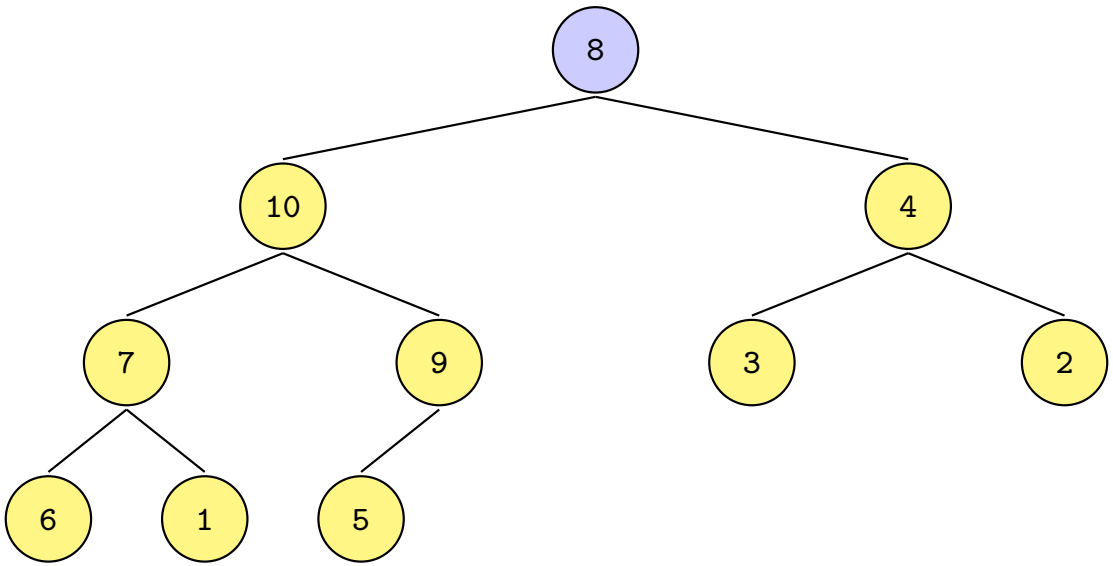
Como extrair o máximo (maior prioridade) da Heap?



- Acessamos a **raiz**, e trocamos com o último elemento do heap

Heap - Extraindo o Máximo

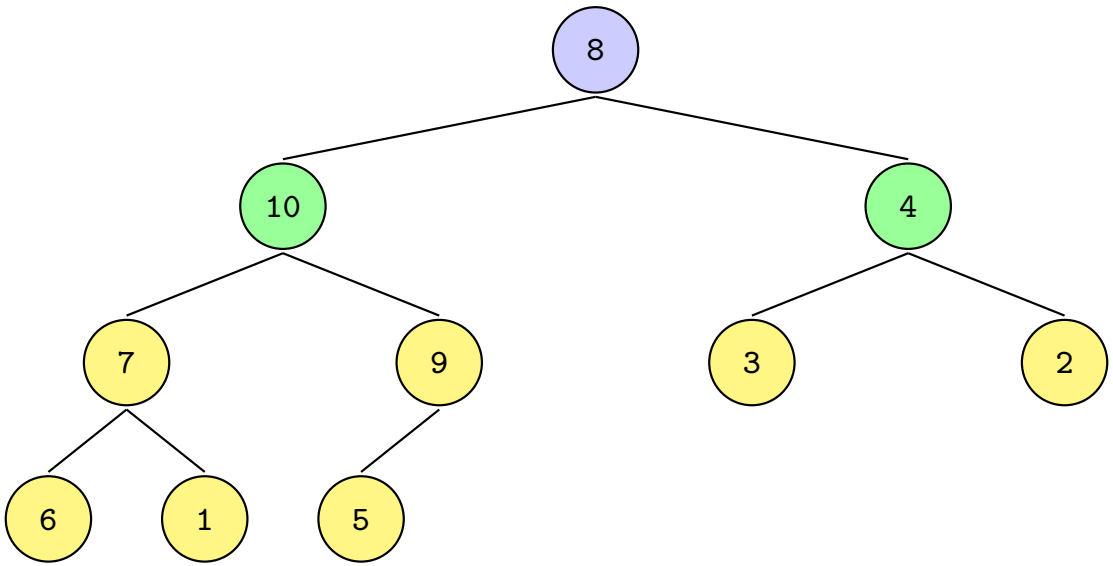
Como extrair o máximo (maior prioridade) da Heap?



- Acessamos a **raiz**, e trocamos com o último elemento do heap

Heap - Extraindo o Máximo

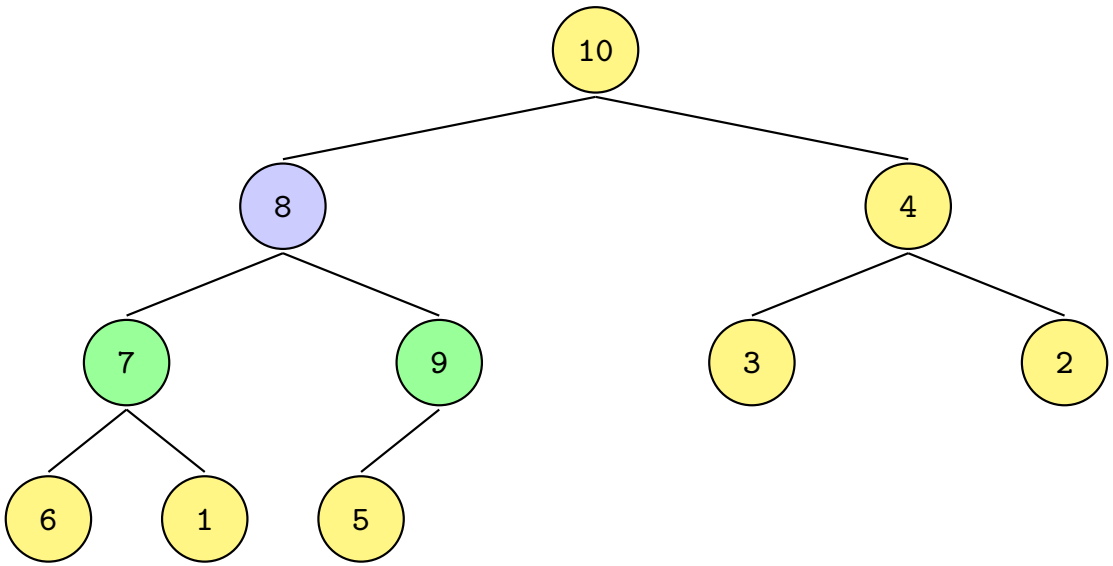
Como extrair o máximo (maior prioridade) da Heap?



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

Heap - Extraindo o Máximo

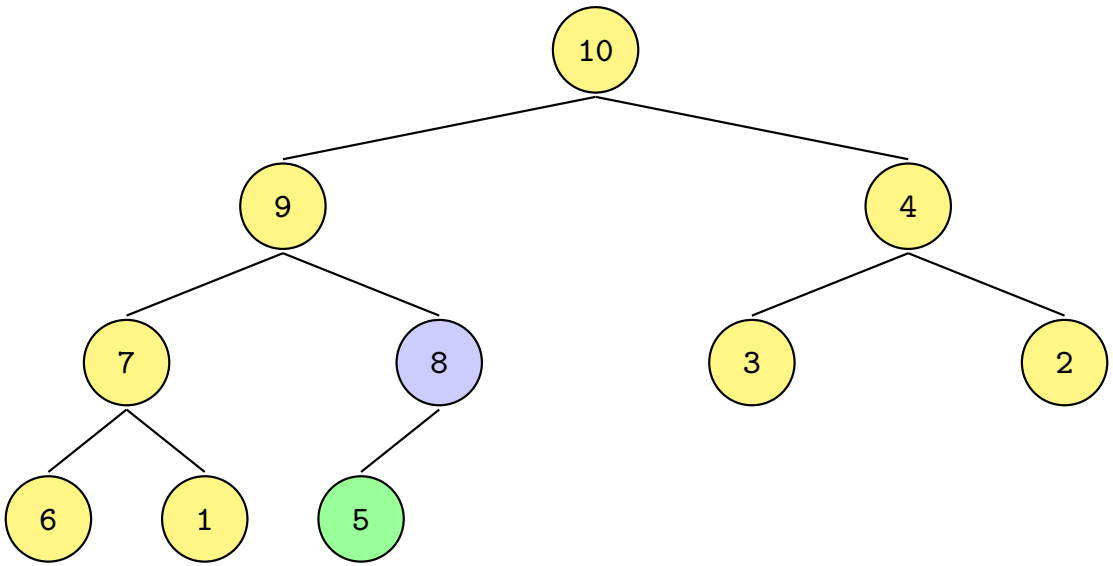
Como extrair o máximo (maior prioridade) da Heap?



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

Heap - Extraindo o Máximo

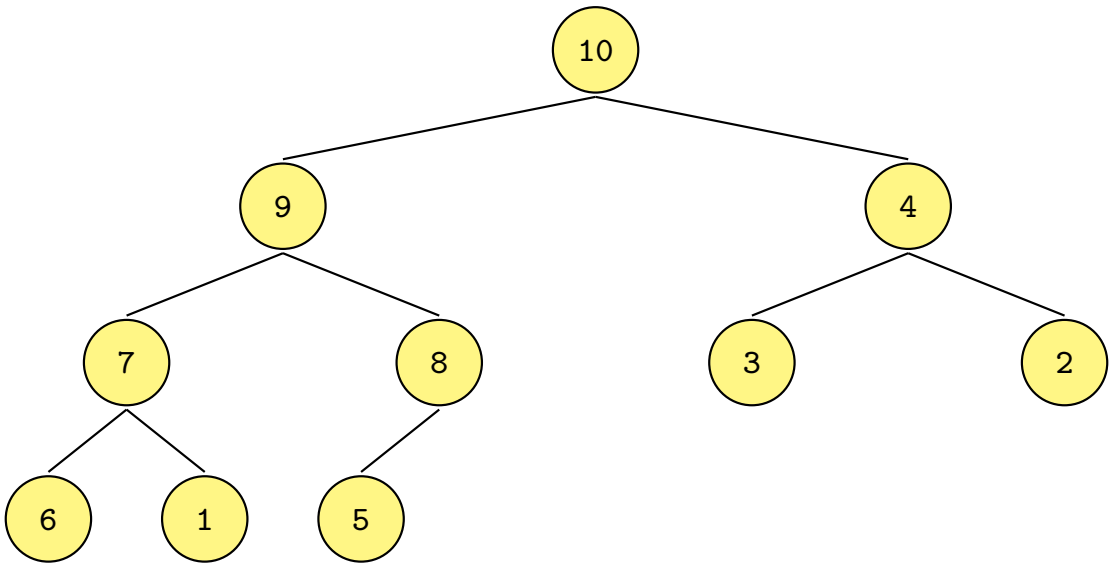
Como extrair o máximo (maior prioridade) da Heap?



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

Heap - Extraindo o Máximo

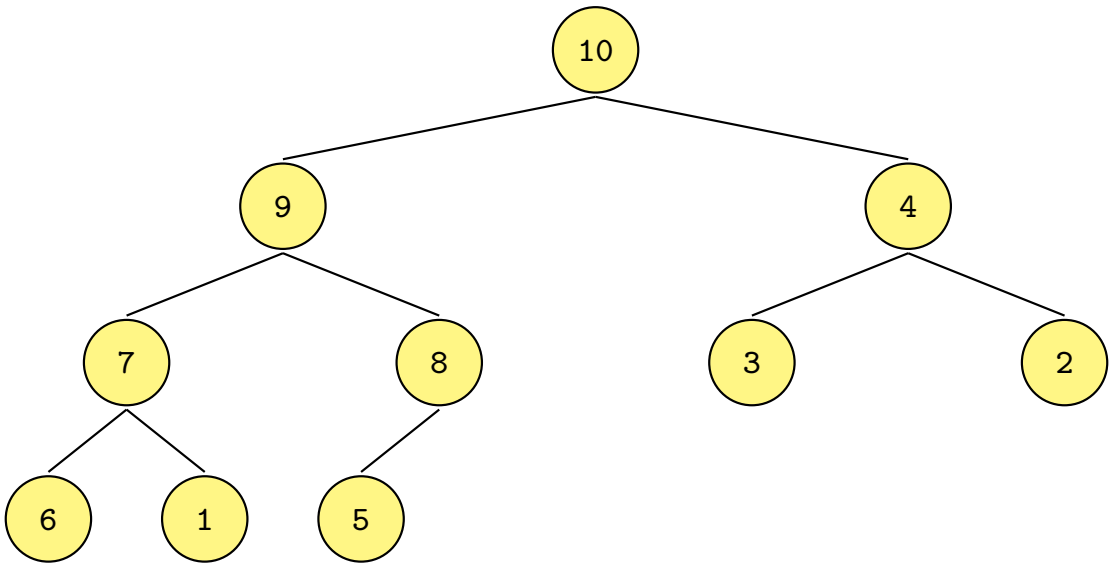
Como extrair o máximo (maior prioridade) da Heap?



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

Heap - Extraindo o Máximo

Como extrair o máximo (maior prioridade) da Heap?



- Acessamos a **raiz**, e trocamos com o último elemento do heap
- Descemos no heap arrumando
  - Trocamos o pai com o **maior dos** dois filhos (se necessário)
- Custo computacional: $O(\lg n)$

TAD - Interface (implementação com heap)

pq_heap.h

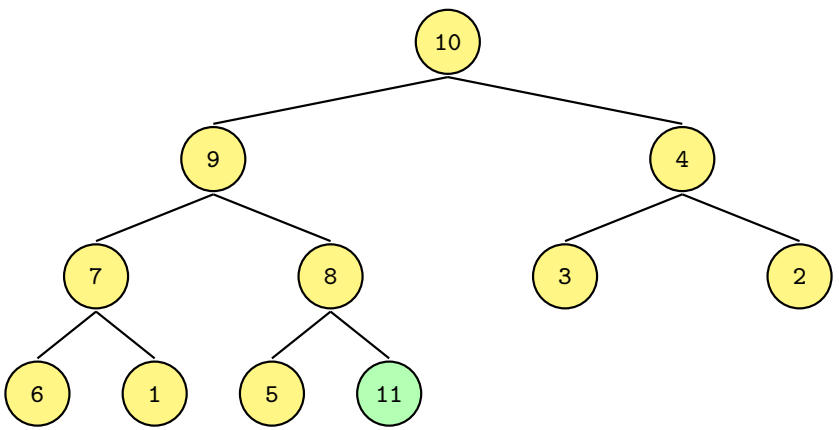
```
1  #ifndef PQ_HEAP_H
2  #define PQ_HEAP_H
3
4  //Dados
5  typedef struct {
6    char nome[20];
7    int chave;
8  } t_item;
9
10 typedef struct {
11   t_item *dados;
12   int tam, n;
13 } PQ;
```

```
15 //Funções
16 PQ*  pq_criar(int tam);
17 void pq_destruir(PQ **p);
18
19 void pq_adicionar(PQ *p, t_item x);
20 t_item pq_extrai_maximo(PQ *p);
21
22 int pq_vazia(PQ* p);
23 int pq_cheia(PQ* p);
24
25 #endif
```

Heap - Inserir um valor

pq_heap.c

```
45 void pq_adicionar(PQ *p, t_item item) {
46   p->dados[p->n] = item;
47   p->n++;
48   sobe_no_heap(p, p->n - 1);
49 }
```

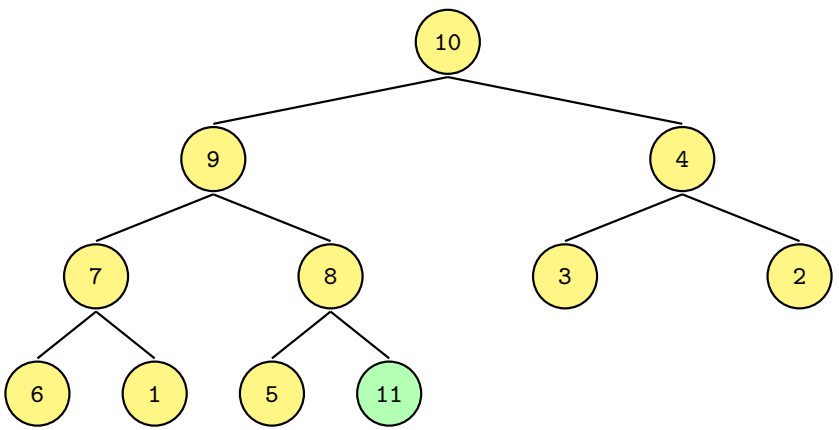


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	4	7	8	3	2	6	1	5	?	?	?	?	?	?

Heap - Inserir um valor

pq_heap.c

```
45 void pq_adicionar(PQ *p, t_item item) {
46   p->dados[p->n] = item;
47   p->n++;
48   sobe_no_heap(p, p->n - 1);
49 }
```

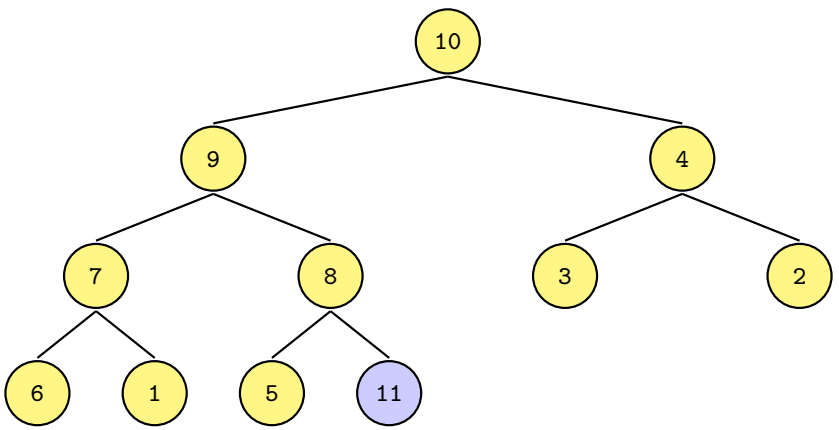


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	4	7	8	3	2	6	1	5	11	?	?	?	?	?

Heap - Inserir um valor

pq_heap.c

```
25 void sobe_no_heap(PQ *p, int pos) {
26   if (pos > 0 && p->dados[PAI(pos)].chave < p->dados[pos].chave) {
27     swap(&p->dados[pos], &p->dados[PAI(pos)]);
28     sobe_no_heap(p, PAI(pos));
29   }
30 }
```

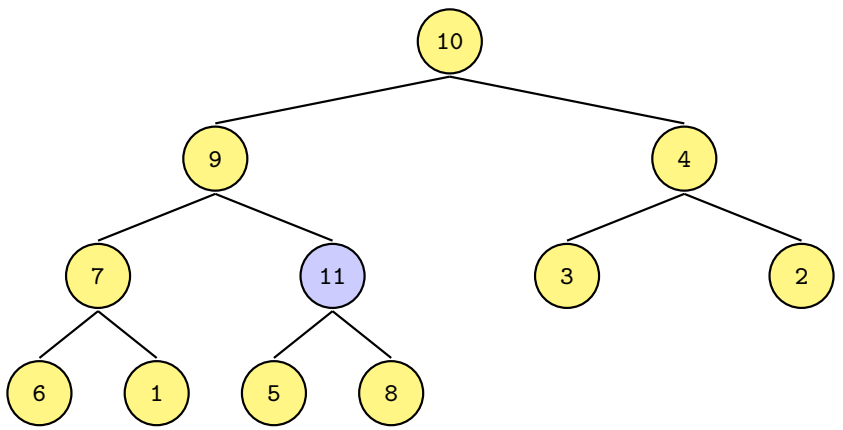


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	4	7	8	3	2	6	1	5	11	?	?	?	?	?

Heap - Inserir um valor

pq_heap.c

```
25 void sobe_no_heap(PQ *p, int pos) {
26   if (pos > 0 && p->dados[PAI(pos)].chave < p->dados[pos].chave) {
27     swap(&p->dados[pos], &p->dados[PAI(pos)]);
28     sobe_no_heap(p, PAI(pos));
29   }
30 }
```

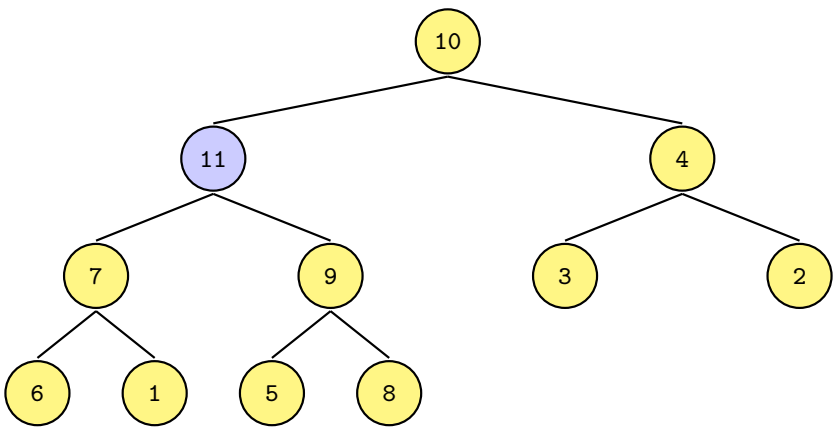


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	4	7	11	3	2	6	1	5	8	?	?	?	?	?

Heap - Inserir um valor

pq_heap.c

```
25 void sobe_no_heap(PQ *p, int pos) {
26   if (pos > 0 && p->dados[PAI(pos)].chave < p->dados[pos].chave) {
27     swap(&p->dados[pos], &p->dados[PAI(pos)]);
28     sobe_no_heap(p, PAI(pos));
29   }
30 }
```

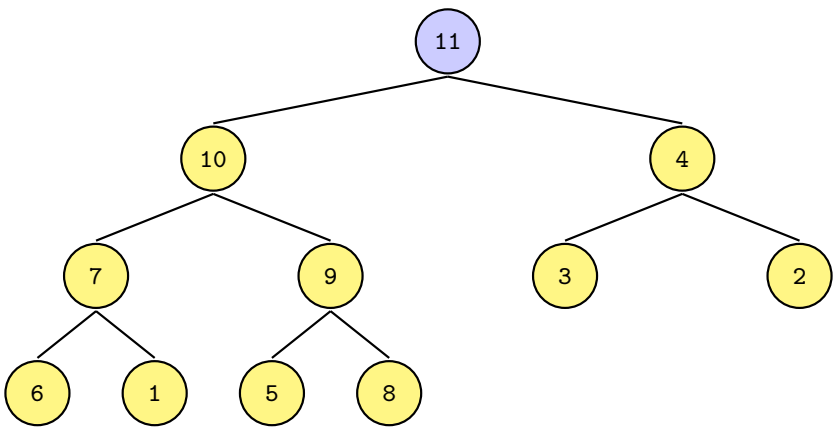


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	11	4	7	9	3	2	6	1	5	8	?	?	?	?	?

Heap - Inserir um valor

pq_heap.c

```
25 void sobe_no_heap(PQ *p, int pos) {
26   if (pos > 0 && p->dados[PAI(pos)].chave < p->dados[pos].chave) {
27     swap(&p->dados[pos], &p->dados[PAI(pos)]);
28     sobe_no_heap(p, PAI(pos));
29   }
30 }
```

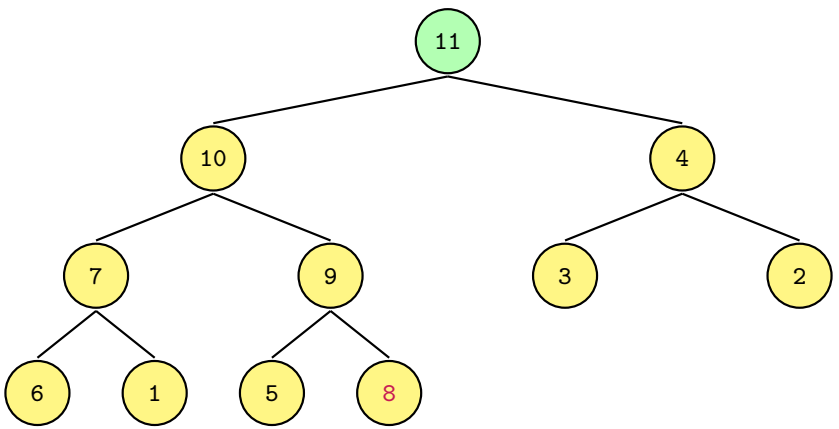


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	10	4	7	9	3	2	6	1	5	8	?	?	?	?	?

Heap - Extraindo o Máximo

pq_heap.c

```
51 t_item pq_extrai_maximo(PQ *p) {
52   t_item item = p->dados[0];
53   swap(&p->dados[0], &p->dados[p->n - 1]);
54   p->n--;
55   desce_no_heap(p, 0);
56 return item;
57 }
```

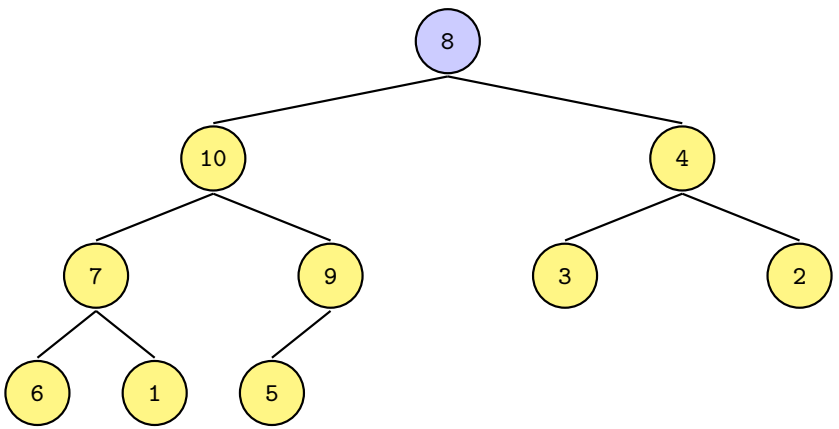


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	10	4	7	9	3	2	6	1	5	8	?	?	?	?	?

Heap - Extraindo o Máximo

pq_heap.c

```
51 t_item pq_extrai_maximo(PQ *p) {
52   t_item item = p->dados[0];
53   swap(&p->dados[0], &p->dados[p->n - 1]);
54   p->n--;
55   desce_no_heap(p, 0);
56 return item;
57 }
```

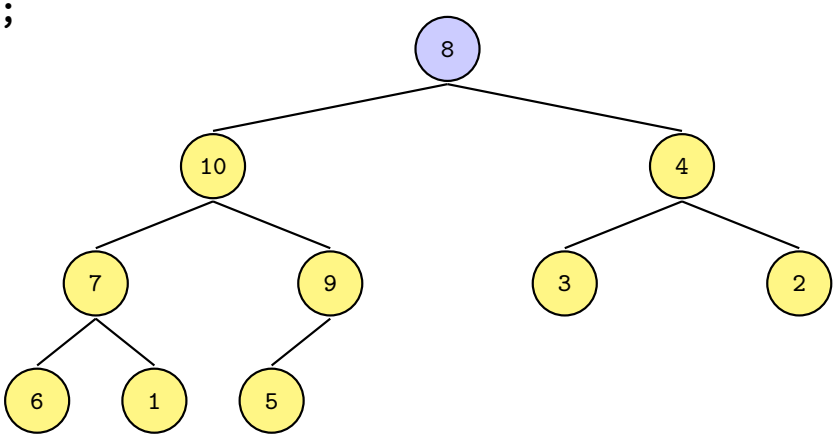


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	10	4	7	9	3	2	6	1	5	?	?	?	?	?	?

Heap - Extraindo o Máximo

pq_heap.c

```
32 void desce_no_heap(PQ *p, int pos) {
33   if (F_ESQ(pos) < p->n){
34     int maior_filho = F_ESQ(pos);
35     if (F_DIR(pos) < p->n &&
36         p->dados[F_ESQ(pos)].chave < p->dados[F_DIR(pos)].chave)
37       maior_filho = F_DIR(pos);
38     if (p->dados[pos].chave < p->dados[maior_filho].chave) {
39       swap(&p->dados[pos], &p->dados[maior_filho]);
40       desce_no_heap(p, maior_filho);
41     }
42   }
43 }
```

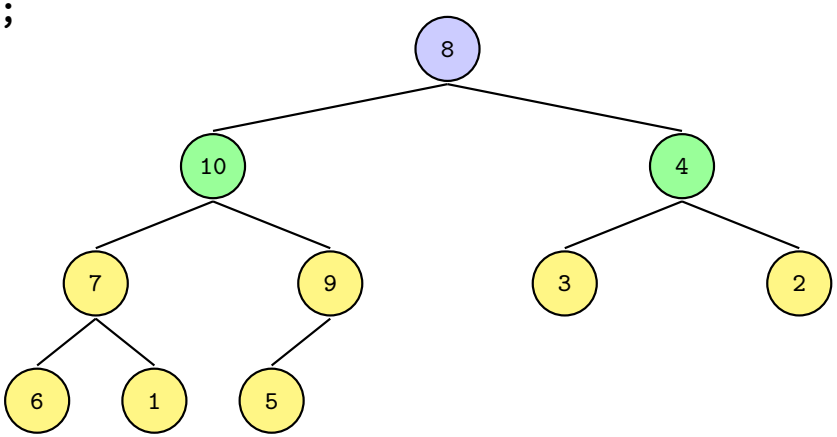


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	10	4	7	9	3	2	6	1	5	?	?	?	?	?	?

Heap - Extraindo o Máximo

pq_heap.c

```
32 void desce_no_heap(PQ *p, int pos) {
33   if (F_ESQ(pos) < p->n){
34     int maior_filho = F_ESQ(pos);
35     if (F_DIR(pos) < p->n &&
36         p->dados[F_ESQ(pos)].chave < p->dados[F_DIR(pos)].chave)
37       maior_filho = F_DIR(pos);
38     if (p->dados[pos].chave < p->dados[maior_filho].chave) {
39       swap(&p->dados[pos], &p->dados[maior_filho]);
40       desce_no_heap(p, maior_filho);
41     }
42   }
43 }
```

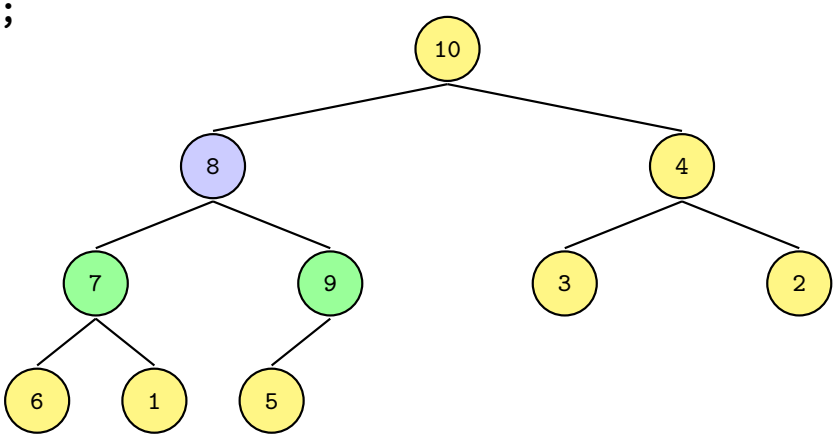


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	10	4	7	9	3	2	6	1	5	?	?	?	?	?	?

Heap - Extraindo o Máximo

pq_heap.c

```
32 void desce_no_heap(PQ *p, int pos) {
33   if (F_ESQ(pos) < p->n){
34     int maior_filho = F_ESQ(pos);
35     if (F_DIR(pos) < p->n &&
36         p->dados[F_ESQ(pos)].chave < p->dados[F_DIR(pos)].chave)
37       maior_filho = F_DIR(pos);
38     if (p->dados[pos].chave < p->dados[maior_filho].chave) {
39       swap(&p->dados[pos], &p->dados[maior_filho]);
40       desce_no_heap(p, maior_filho);
41     }
42   }
43 }
```

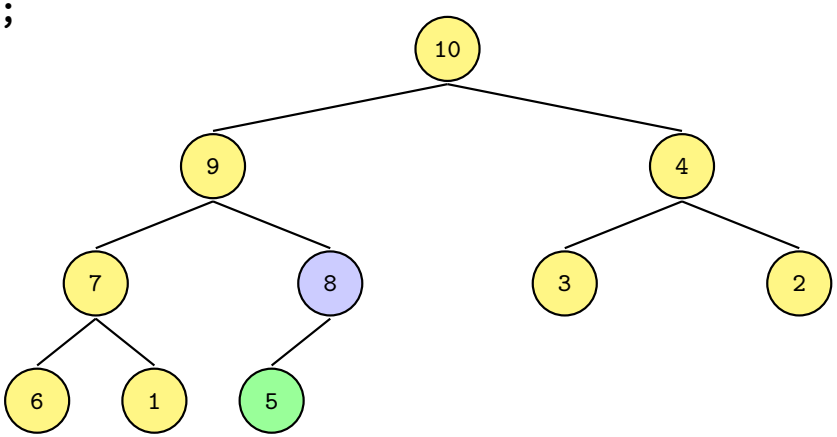


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	8	4	7	9	3	2	6	1	5	?	?	?	?	?	?

Heap - Extraindo o Máximo

pq_heap.c

```
32 void desce_no_heap(PQ *p, int pos) {
33   if (F_ESQ(pos) < p->n){
34     int maior_filho = F_ESQ(pos);
35     if (F_DIR(pos) < p->n &&
36         p->dados[F_ESQ(pos)].chave < p->dados[F_DIR(pos)].chave)
37       maior_filho = F_DIR(pos);
38     if (p->dados[pos].chave < p->dados[maior_filho].chave) {
39       swap(&p->dados[pos], &p->dados[maior_filho]);
40       desce_no_heap(p, maior_filho);
41     }
42   }
43 }
```

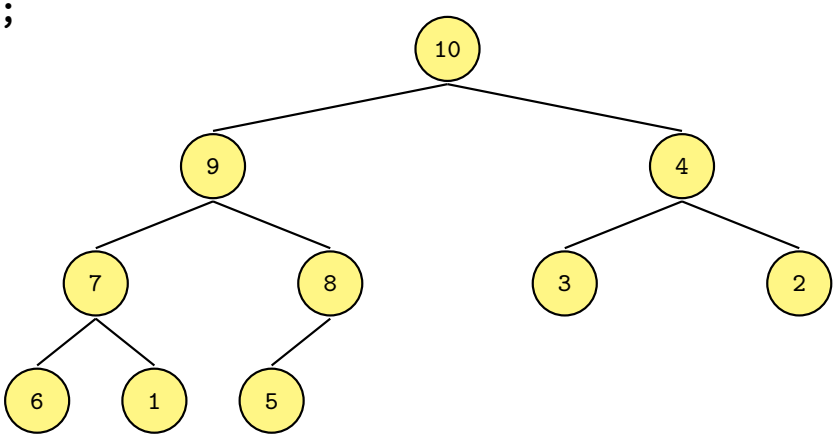


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	4	7	8	3	2	6	1	5	?	?	?	?	?	?

Heap - Extraindo o Máximo

pq_heap.c

```
32 void desce_no_heap(PQ *p, int pos) {
33   if (F_ESQ(pos) < p->n){
34     int maior_filho = F_ESQ(pos);
35     if (F_DIR(pos) < p->n &&
36         p->dados[F_ESQ(pos)].chave < p->dados[F_DIR(pos)].chave)
37       maior_filho = F_DIR(pos);
38     if (p->dados[pos].chave < p->dados[maior_filho].chave) {
39       swap(&p->dados[pos], &p->dados[maior_filho]);
40       desce_no_heap(p, maior_filho);
41     }
42   }
43 }
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	4	7	8	3	2	6	1	5	?	?	?	?	?	?

Fila de Prioridade: Vetores vs. Max-Heap

	Vetores	Max-Heap
Inserção	$\mathrm{O}(1)$	$\mathrm{O}(\lg n)$
Remoção	$\mathrm{O}(n)$	$\mathrm{O}(\lg n)$

Uso do espaço:

- $\mathrm{O}(n)$ nas duas abordagens.

Qual é melhor?

$\log_2(1 \text{ bilhão}) \approx 30$

Fila de Prioridade: Vetores vs. Max-Heap

	Vetores	Max-Heap
Inserção	$\mathrm{O}(1)$	$\mathrm{O}(\lg n)$
Remoção	$\mathrm{O}(n)$	$\mathrm{O}(\lg n)$

Uso do espaço:

- $\mathrm{O}(n)$ nas duas abordagens.

Qual é melhor?

- Max-Heap!!

$\log_2(1 \text{ bilhão}) \approx 30$

Cliente

exemplo2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "pq_heap.h"

int main() {
  int n, i;
  scanf("%d", &n);  //valor de n
  PQ *Fila = pq_criar(n);
  for (i = 0; i < n; i++) {
    t_item item;
    scanf("%d %s", &item.chave, item.nome); //chave nome
    pq_adicionar(Fila, item);
  }
  printf("Fila:\n");
  while(!pq_vazia(Fila)) {
    t_item item = pq_extrai_maximo(Fila);
    printf("%d %s\n", item.chave, item.nome);
  }
  pq_destruir(&Fila);
return 0;
}
```

Makefile

Vamos usar o Makefile para compilar:

```
exemplo2: exemplo2.c pq_heap.o
  gcc $^ -o $@
```

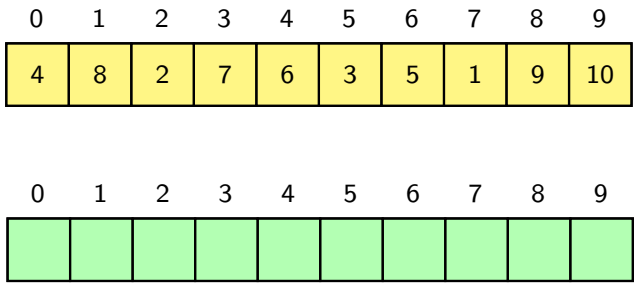
Vamos executar:

```
$ ./exemplo2
5
2 a
4 b
3 c
2 d
1 f
Fila:
4 b
3 c
2 d
1 f
0
```

Roteiro

Ordenação usando Fila de Prioridades

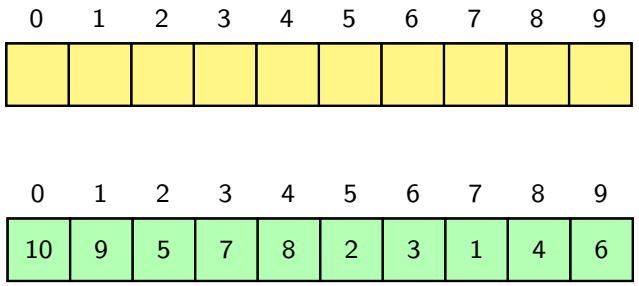
```
44 void heapsort_v1(int *v, int n) {
45   int i;
46   int *novo = (int*) malloc(n*sizeof(int));
47   for (i = 0; i < n; i++)
48     heap_adicionar(novo, i, v[i]);
49   for (i = n-1; i >= 0; i--)
50     v[i] = heap_extrai_maximo(novo, i+1);
51   free(novo);
52 }
```



Tempo: $O(n \lg n)$

Ordenação usando Fila de Prioridades

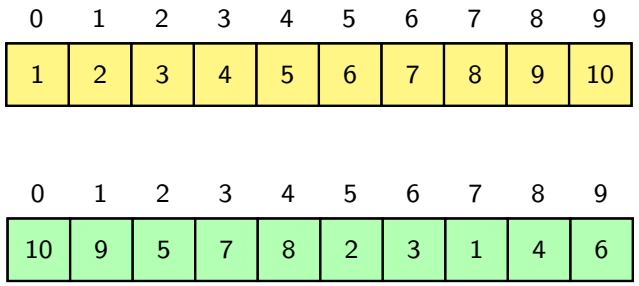
```
44 void heapsort_v1(int *v, int n) {
45   int i;
46   int *novo = (int*) malloc(n*sizeof(int));
47   for (i = 0; i < n; i++)
48     heap_adicionar(novo, i, v[i]);
49   for (i = n-1; i >= 0; i--)
50     v[i] = heap_extrai_maximo(novo, i+1);
51   free(novo);
52 }
```



Tempo: $O(n \lg n)$

Ordenação usando Fila de Prioridades

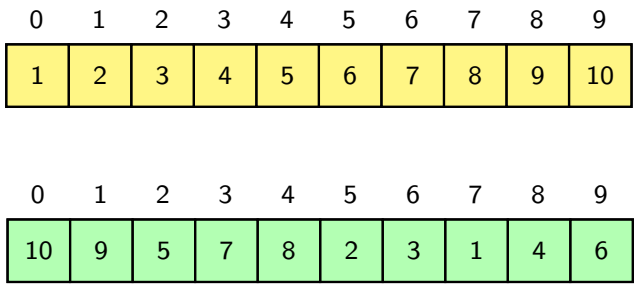
```
44 void heapsort_v1(int *v, int n) {
45   int i;
46   int *novo = (int*) malloc(n*sizeof(int));
47   for (i = 0; i < n; i++)
48     heap_adicionar(novo, i, v[i]);
49   for (i = n-1; i >= 0; i--)
50     v[i] = heap_extrai_maximo(novo, i+1);
51   free(novo);
52 }
```



Tempo: $O(n \lg n)$

Ordenação usando Fila de Prioridades

```
44 void heapsort_v1(int *v, int n) {
45   int i;
46   int *novo = (int*) malloc(n*sizeof(int));
47   for (i = 0; i < n; i++)
48     heap_adicionar(novo, i, v[i]);
49   for (i = n-1; i >= 0; i--)
50     v[i] = heap_extrai_maximo(novo, i+1);
51   free(novo);
52 }
```

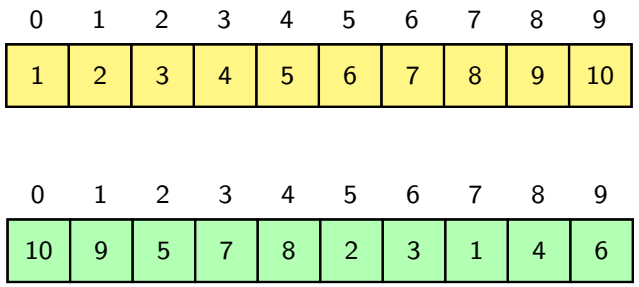


Tempo: $O(n \lg n)$

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

Ordenação usando Fila de Prioridades

```
44 void heapsort_v1(int *v, int n) {
45   int i;
46   int *novo = (int*) malloc(n*sizeof(int));
47   for (i = 0; i < n; i++)
48     heap_adicionar(novo, i, v[i]);
49   for (i = n-1; i >= 0; i--)
50     v[i] = heap_extrai_maximo(novo, i+1);
51   free(novo);
52 }
```



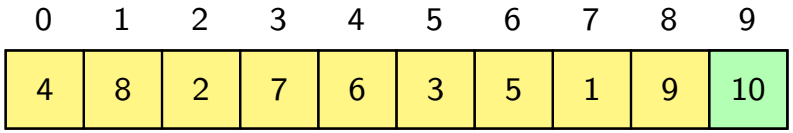
Tempo: $O(n \lg n)$

- Estamos usando espaço adicional, mas não precisamos...
- Podemos transformar um **vetor** em um **heap** rapidamente
  - Mais rápido do que fazer n inserções

Transformando um vetor em um heap

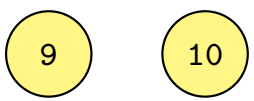
0	1	2	3	4	5	6	7	8	9
4	8	2	7	6	3	5	1	9	10

Transformando um vetor em um heap

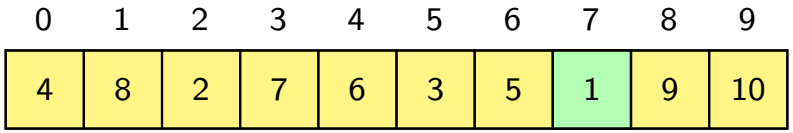


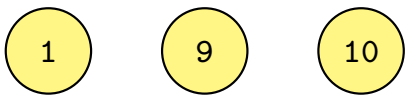
Transformando um vetor em um heap

0	1	2	3	4	5	6	7	8	9
4	8	2	7	6	3	5	1	9	10

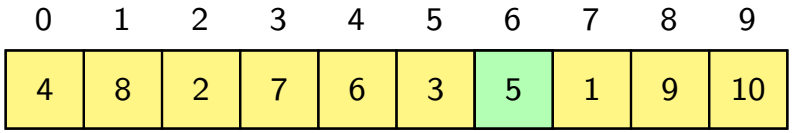


Transformando um vetor em um heap

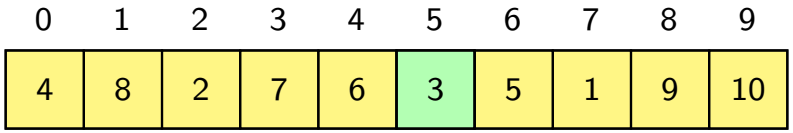




Transformando um vetor em um heap

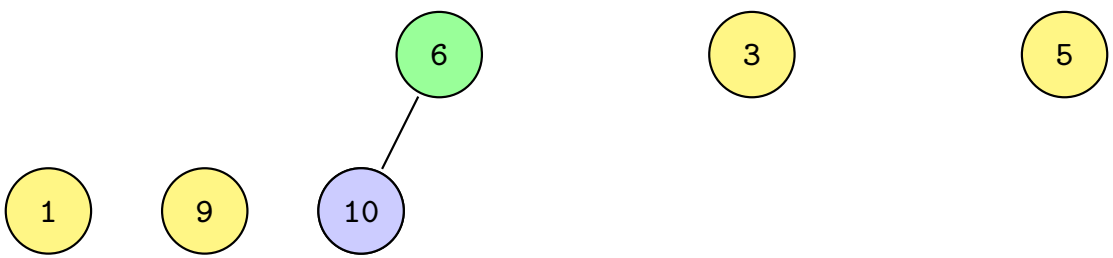


Transformando um vetor em um heap



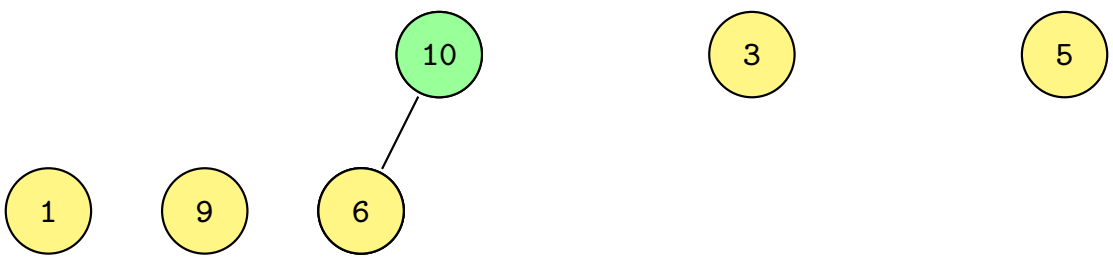
Transformando um vetor em um heap

0	1	2	3	4	5	6	7	8	9
4	8	2	7	6	3	5	1	9	**10**



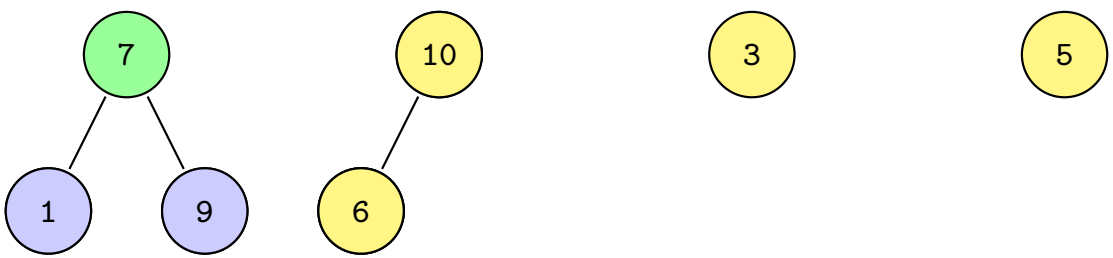
Transformando um vetor em um heap

0	1	2	3	4	5	6	7	8	9
4	8	2	7	10	3	5	1	9	6



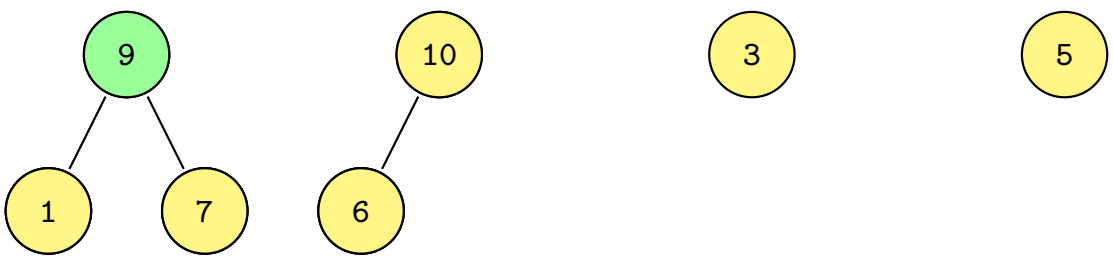
Transformando um vetor em um heap

0	1	2	3	4	5	6	7	8	9
4	8	2	7	10	3	5	1	9	6



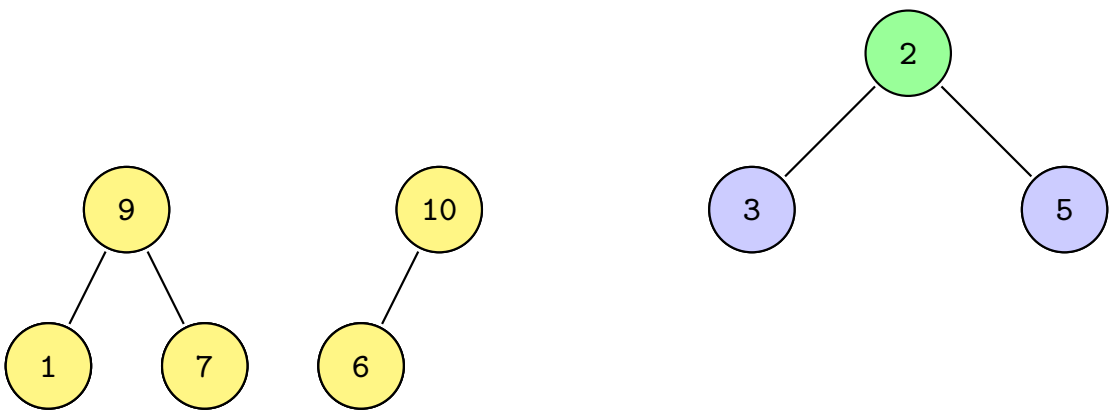
Transformando um vetor em um heap

0	1	2	3	4	5	6	7	8	9
4	8	2	9	10	3	5	1	7	6



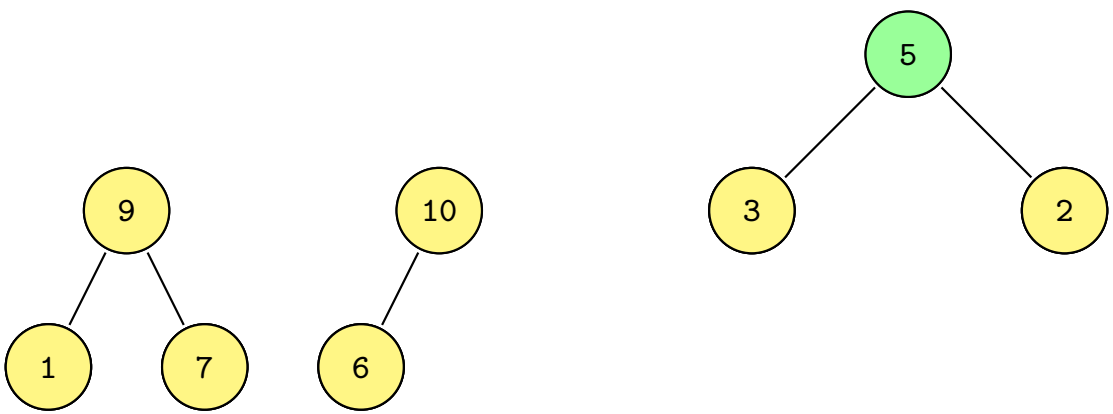
Transformando um vetor em um heap

0	1	2	3	4	5	6	7	8	9
4	8	2	9	10	**3**	**5**	1	7	6



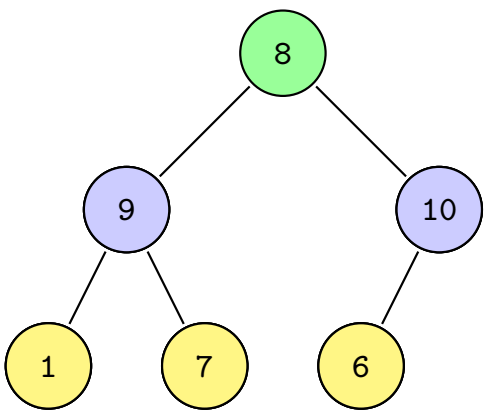
Transformando um vetor em um heap

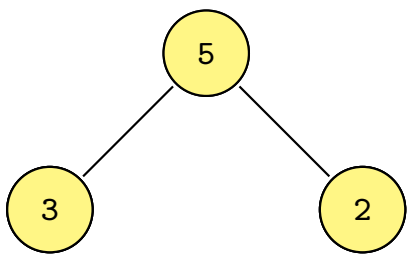
0	1	2	3	4	5	6	7	8	9
4	8	5	9	10	2	5	1	7	6



Transformando um vetor em um heap

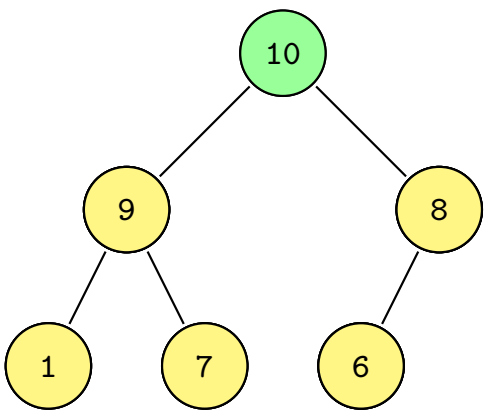
0	1	2	3	4	5	6	7	8	9
4	8	5	**9**	**10**	2	5	1	7	6

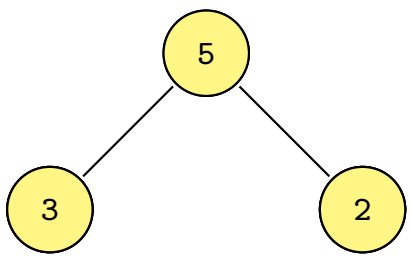




Transformando um vetor em um heap

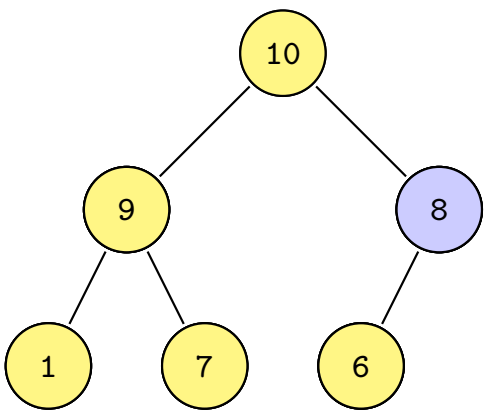
0	1	2	3	4	5	6	7	8	9
4	10	5	9	8	2	5	1	7	6

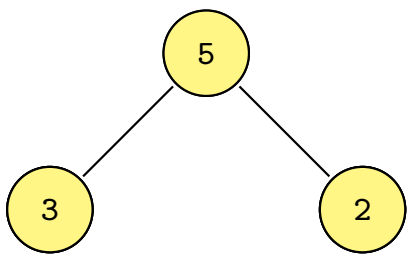




Transformando um vetor em um heap

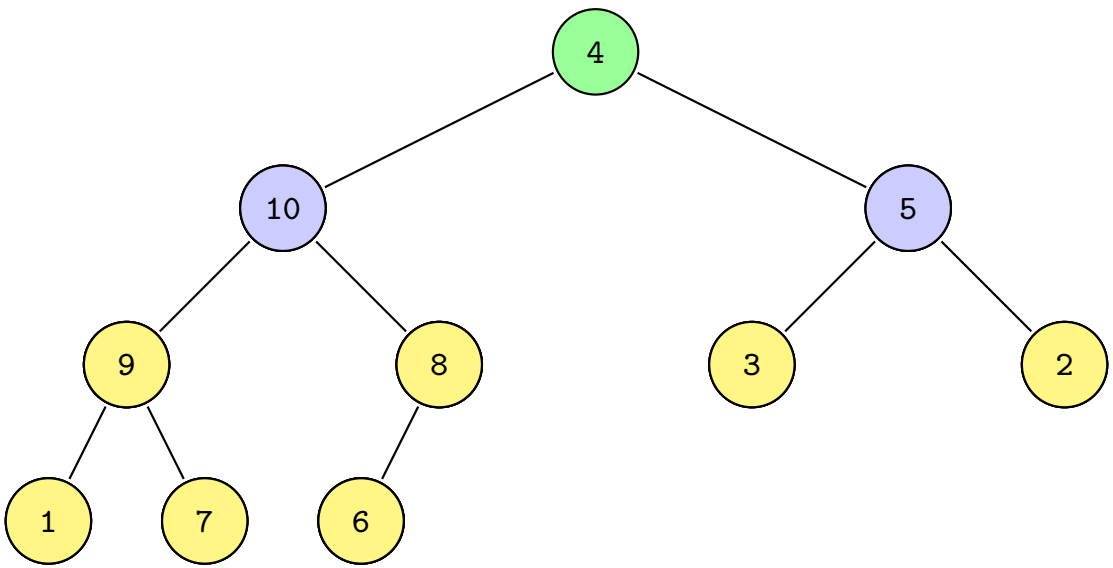
0	1	2	3	4	5	6	7	8	9
4	10	5	9	**8**	2	5	1	7	**6**





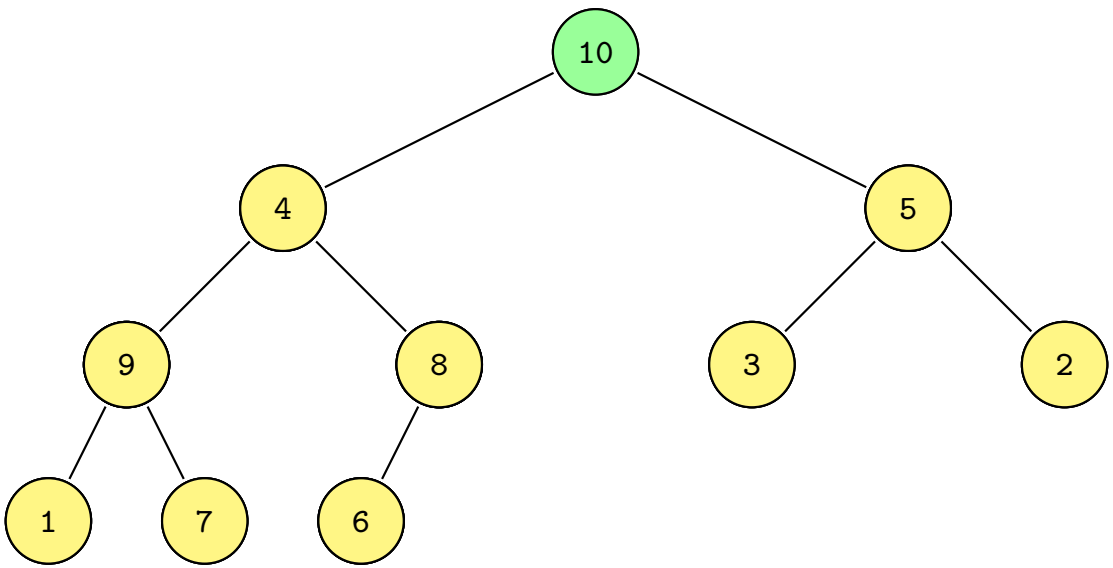
Transformando um vetor em um heap

0	1	2	3	4	5	6	7	8	9
4	**10**	**5**	9	8	2	5	1	7	6

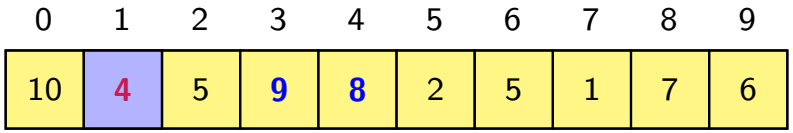


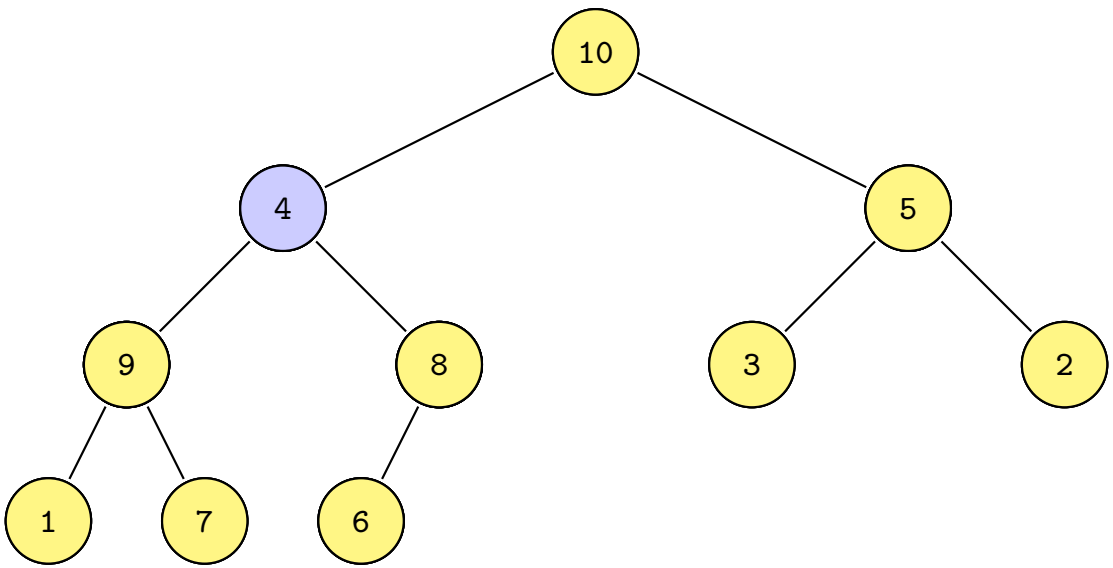
Transformando um vetor em um heap

0	1	2	3	4	5	6	7	8	9
10	4	5	9	8	2	5	1	7	6

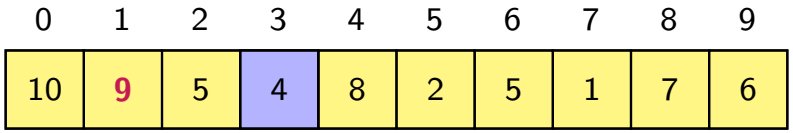


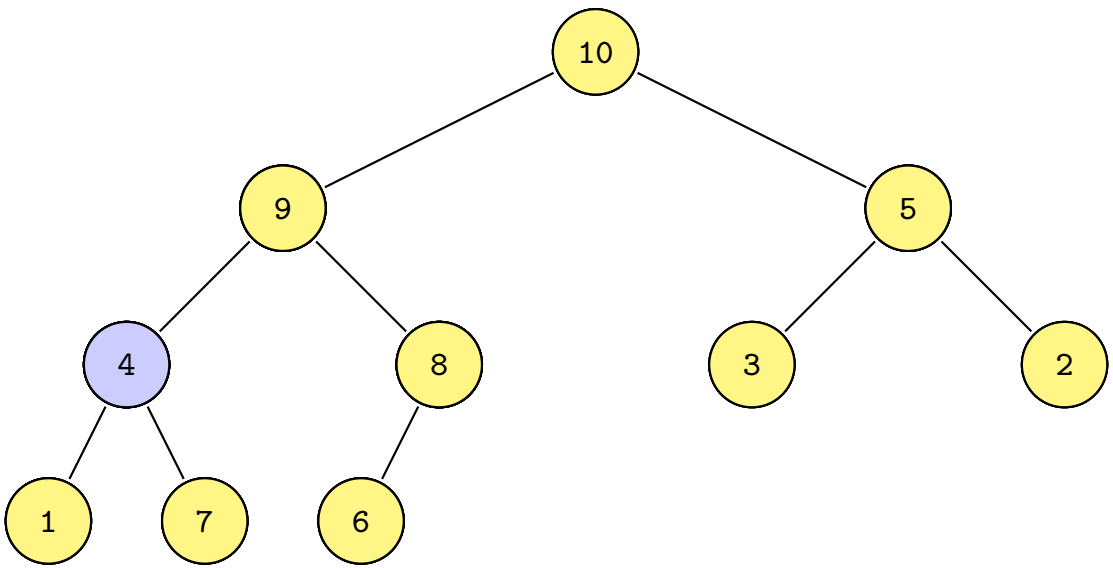
Transformando um vetor em um heap





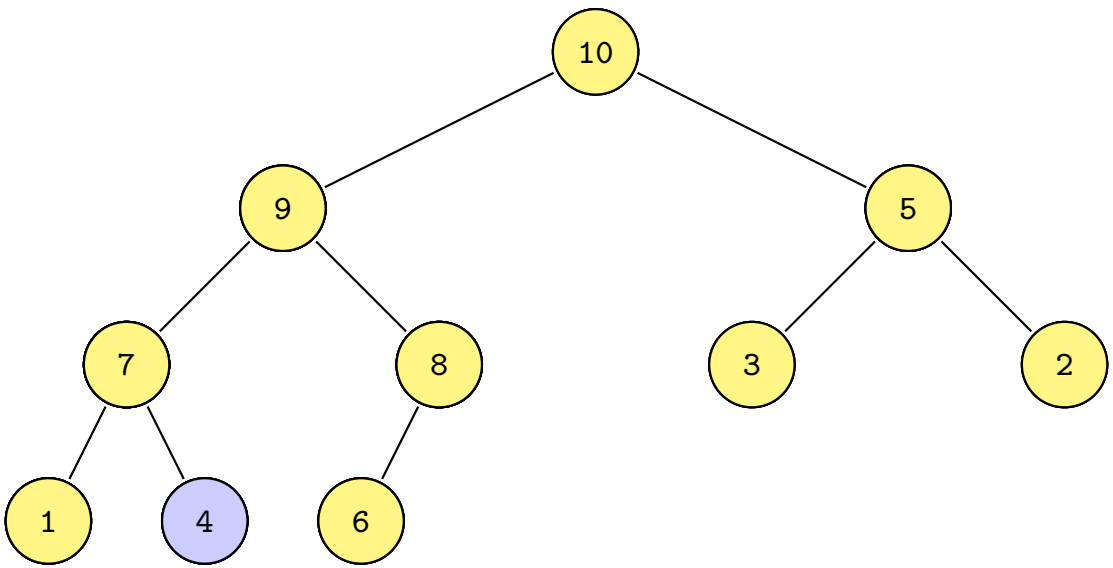
Transformando um vetor em um heap



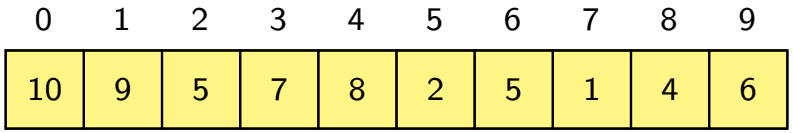


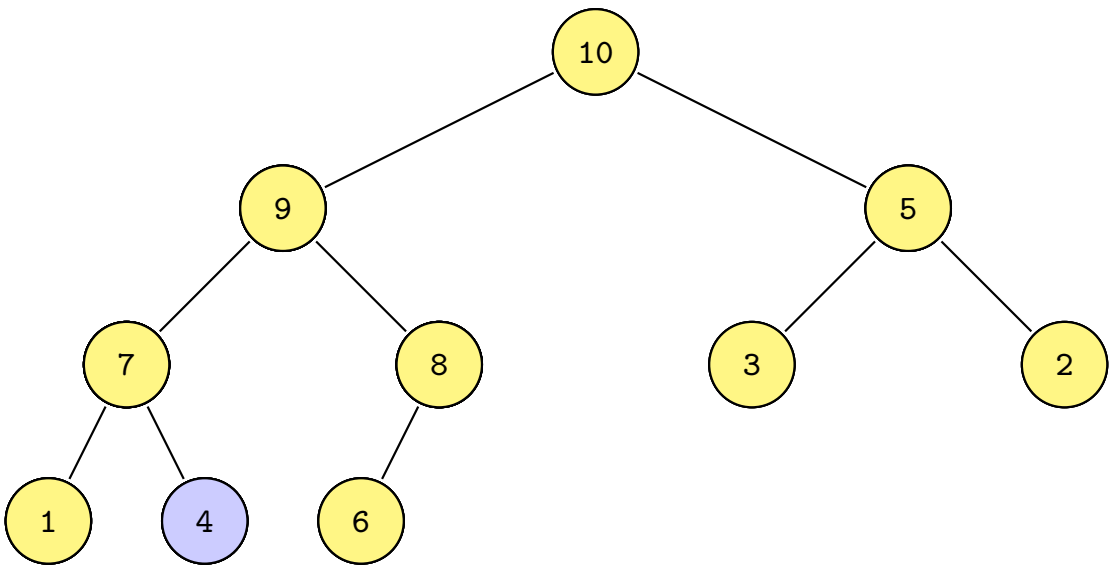
Transformando um vetor em um heap

0	1	2	3	4	5	6	7	8	9
10	9	5	7	8	2	5	1	4	6

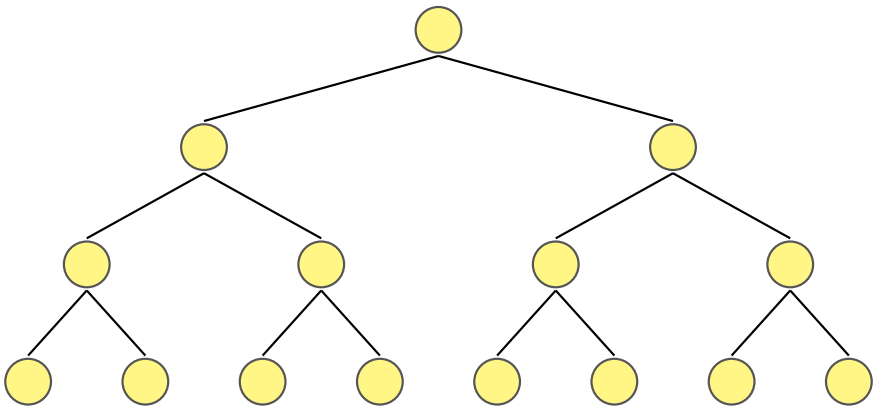


Transformando um vetor em um heap

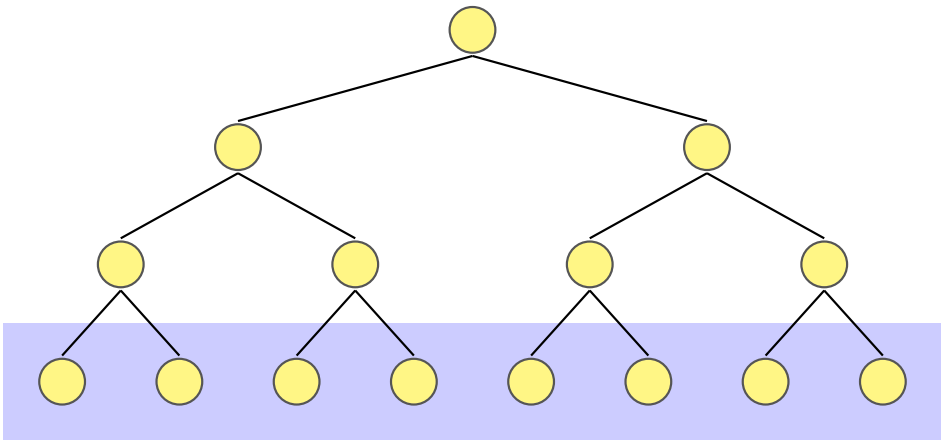




Tempo da construção para $n = 2^k - 1$

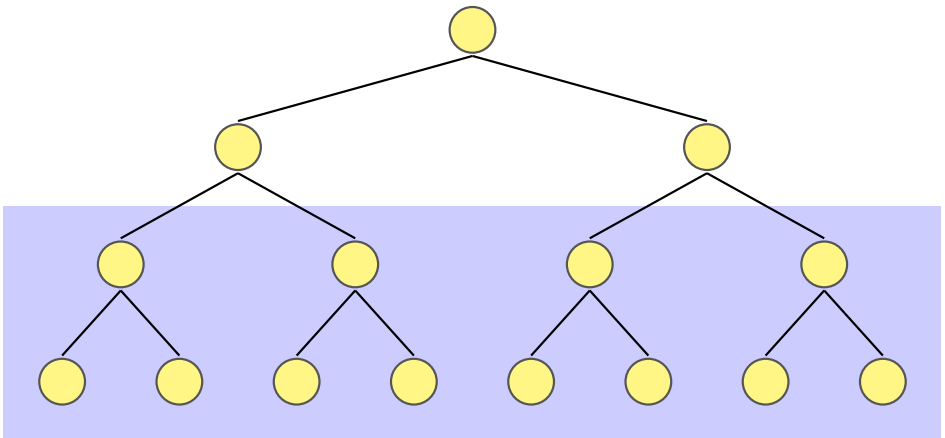


Tempo da construção para $n = 2^k - 1$



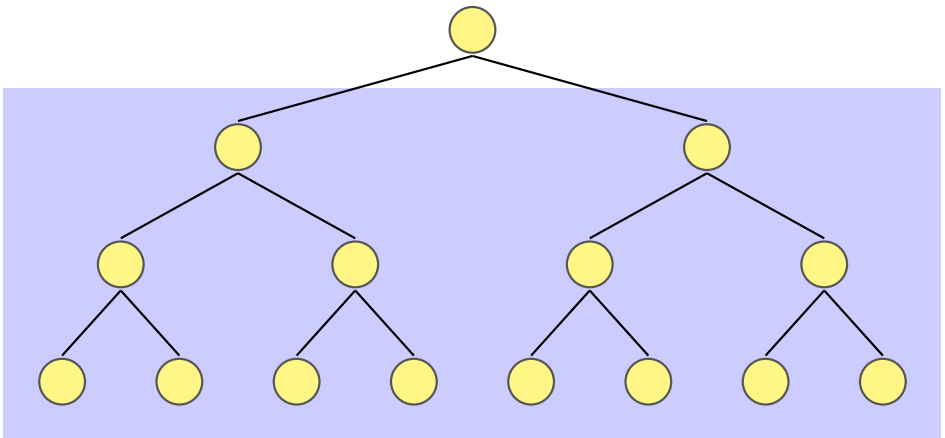
- Temos 2^{k-1} heaps de altura 1 $\approx n/2$

Tempo da construção para $n = 2^k - 1$



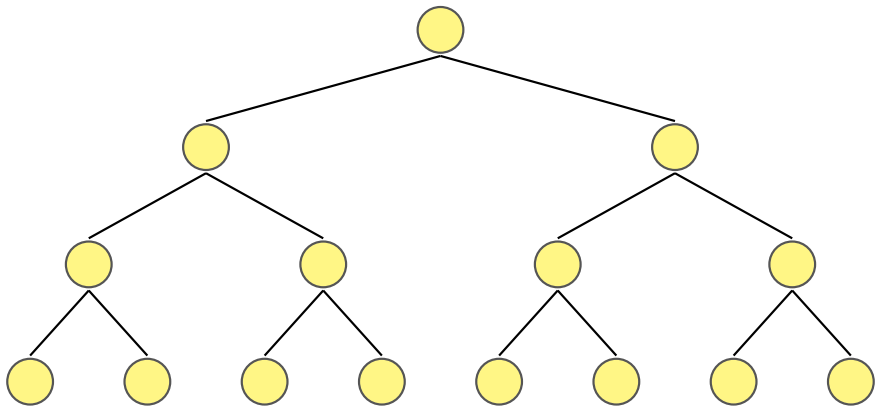
- Temos 2^{k-1} heaps de altura 1 $\approx n/2$
- Temos 2^{k-2} heaps de altura 2 $\approx n/4$

Tempo da construção para $n = 2^k - 1$



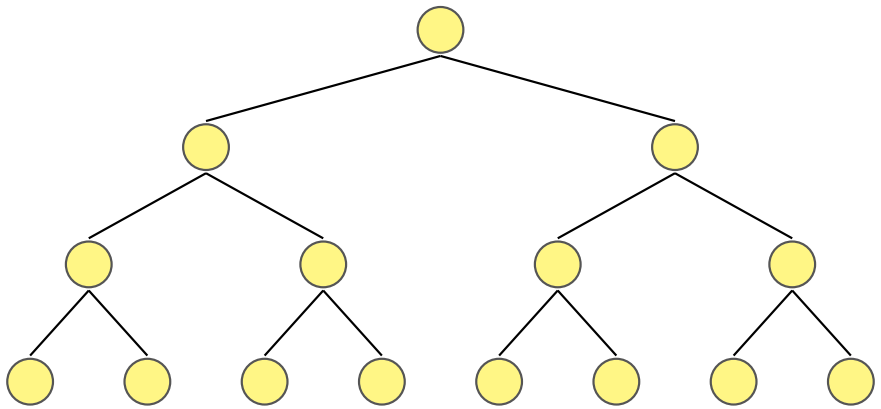
- Temos 2^{k-1} heaps de altura 1 $\approx n/2$
- Temos 2^{k-2} heaps de altura 2 $\approx n/4$
- Temos 2^{k-h} heaps de altura h $\approx n/2^h = 1$

Tempo da construção para $n = 2^k - 1$



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

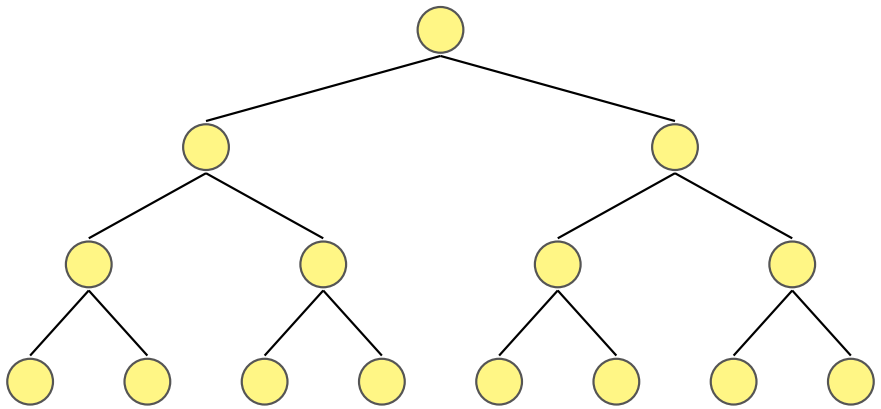
Tempo da construção para $n = 2^k - 1$



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

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

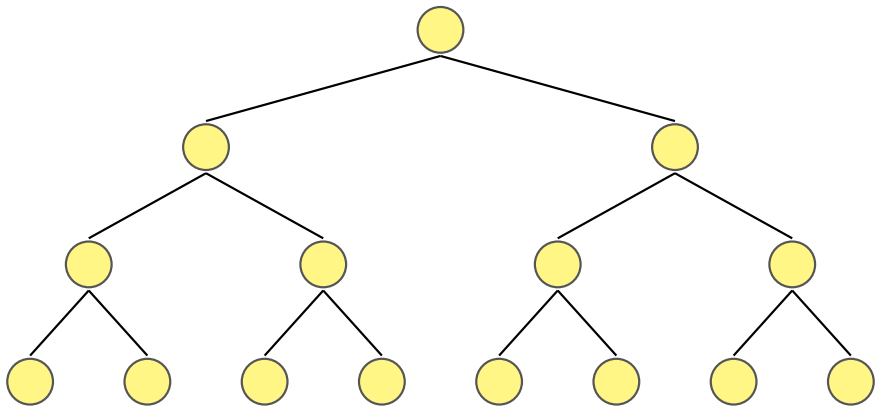
Tempo da construção para $n = 2^k - 1$



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

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

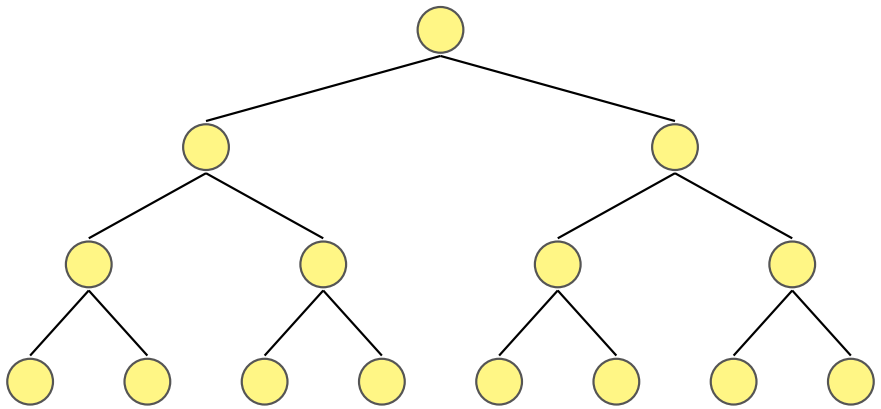
Tempo da construção para $n = 2^k - 1$



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

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

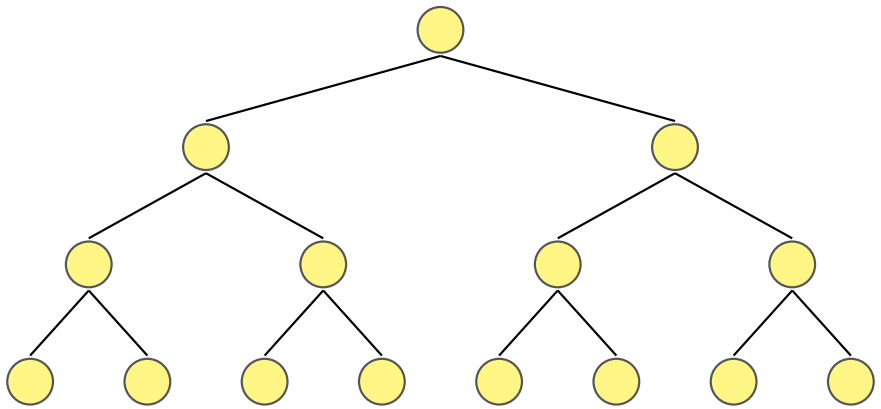
Tempo da construção para $n = 2^k - 1$



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

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

Tempo da construção para $n = 2^k - 1$

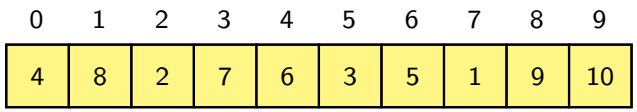


- Temos 2^{k-1} heaps de altura 1 $\approx n/2$
- Temos 2^{k-2} heaps de altura 2 $\approx n/4$
- Temos 2^{k-h} heaps de altura h $\approx n/2^h = 1$
- Cada heap de altura h consome tempo $c \cdot h$

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

Heapsort

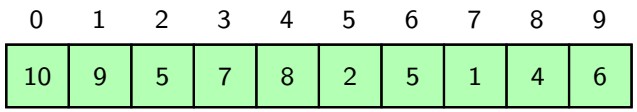
```
54 void heapsort_v2(int *v, int n) {
55   int k;
56   for (k = n/2; k >= 0; k--) /* transforma em heap */
57     desce_no_heap(v, n, k);
58   while (n > 1) { /* extrai o máximo */
59     swap(v[0], v[n - 1]);
60     n--;
61     desce_no_heap(v, n, 0);
62   }
63 }
```



Tempo: $O(n \lg n)$

Heapsort

```
54 void heapsort_v2(int *v, int n) {
55   int k;
56   for (k = n/2; k >= 0; k--) /* transforma em heap */
57     desce_no_heap(v, n, k);
58   while (n > 1) { /* extrai o máximo */
59     swap(v[0], v[n - 1]);
60     n--;
61     desce_no_heap(v, n, 0);
62   }
63 }
```



Tempo: $O(n \lg n)$

- Não usamos espaço adicional
- Tempo de construção do heap: $O(n)$
  - Mais rápido do que fazer n inserções

Cliente

exemplo3.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "heapsort.h"
```

```c
int main() {
  int n, i;
  scanf("%d", &n);  //valor de n
  int *v = (int*) malloc(n*sizeof(int));
  time_t t;
  srand((unsigned) time(&t));
  for (i = 0; i < n; i++)
    v[i] = rand()%n; //random number in [0,n)
  imprime_vetor(v, n);
  heapsort_v2(v, n);
  printf("Vetor ordenado:\n");
  imprime_vetor(v, n);
  free(v);
return 0;
}
```

Makefile

Vamos usar o Makefile para compilar:

```
exemplo3: exemplo3.c heapsort.o
	gcc $^ -o $@
```

Vamos executar:

```
$ ./exemplo3
15
4 4 13 4 13 10 0 13 2 1 6 1 0 10 7
Vetor ordenado:
0 0 1 1 2 4 4 4 6 7 10 10 13 13 13
```

Fim

Dúvidas?

Roteiro

Referências

1. Materiais adaptados dos slides do Prof. Rafael C. S. Schouery, da Universidade Estadual de Campinas.
2. Feofiloff, Paulo. Algoritmos em linguagem C. Elsevier Brasil, 2009.