

MC-202

Filas de Prioridade e Heap

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

Universidade Estadual de Campinas

Atualizado em: 2023-10-02 10:00

Fila de Prioridades

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

- Mas veremos uma implementação muito melhor
- Que também armazena em um vetor...

A função troca

Várias vezes trocaremos dois elementos de posição

Para tanto, vamos usar a seguinte função:

```
1 void troca(int *a, int *b) {  
2     int t = *a;  
3     *a = *b;  
4     *b = t;  
5 }
```

Ou seja, `troca(&v[i], &v[j])` troca os valores de `v[i]` e `v[j]`

Outra opção é colocar diretamente no código da função

- não precisa chamar outra função
- um pouco mais rápido
- código um pouco mais longo e difícil de entender

Fila de Prioridade (usando vetores) — TAD

```
1 typedef struct {
2     char nome[20];
3     int chave;
4 } Item;
5
6 typedef struct {
7     Item *v; // vetor de Items alocado dinamicamente
8     int n, tamanho; // n: qtde de elementos, tamanho: qtde alocada
9 } FP;
10
11 typedef FP * p_fp;
12
13 p_fp criar_filaprio(int tam);
14
15 void insere(p_fp fprio, Item item);
16
17 Item extrai_maximo(p_fp fprio);
18
19 int vazia(p_fp fprio);
20
21 int cheia(p_fp fprio);
```

Operações Básicas

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

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

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

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

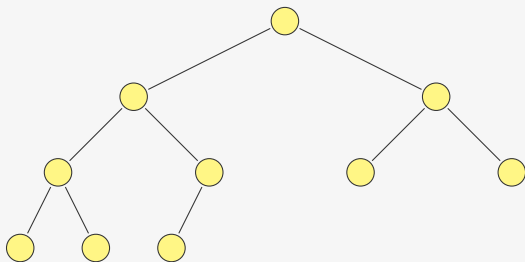
- Se mantiver o vetor ordenado, os tempos se invertem

Árvores Binárias Completas

Uma árvore binária é dita **completa** se:

- Todos os níveis exceto o último estão cheios
- Os nós do último nível estão o mais à esquerda possível

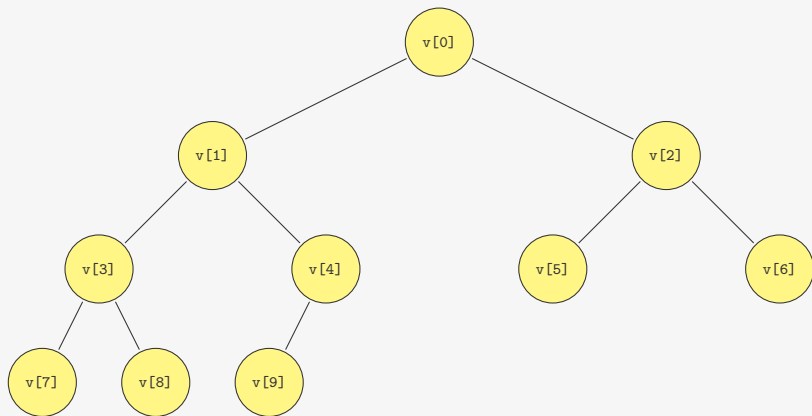
Exemplo:



Quantos níveis tem uma árvore binária completa com n nós?

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

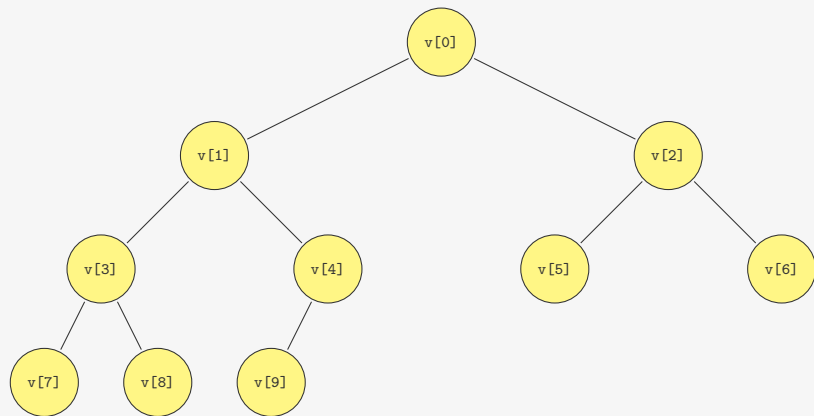
Árvores Binárias Completas e Vetores



Podemos representar tais árvores usando vetores

- Isso é, não precisamos de ponteiros

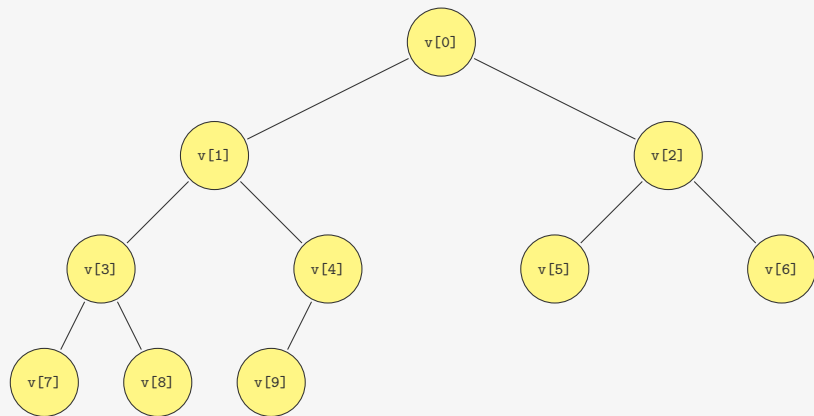
Árvores Binárias Completas e Vetores



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 $v[(i-1)/2]$

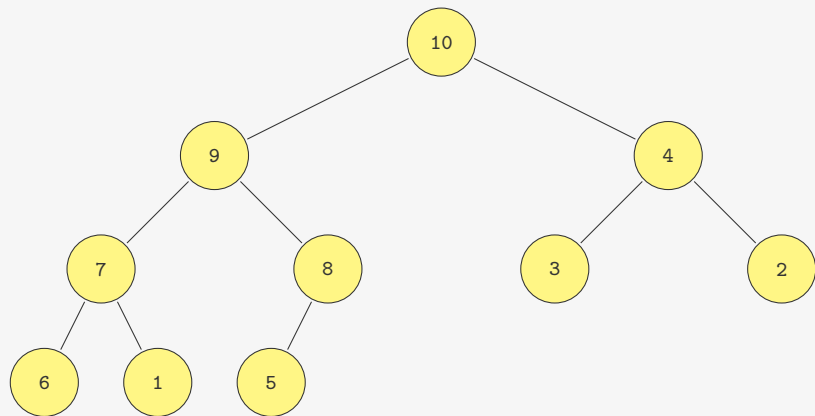
Max-Heap



Em um Heap (de máximo):

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

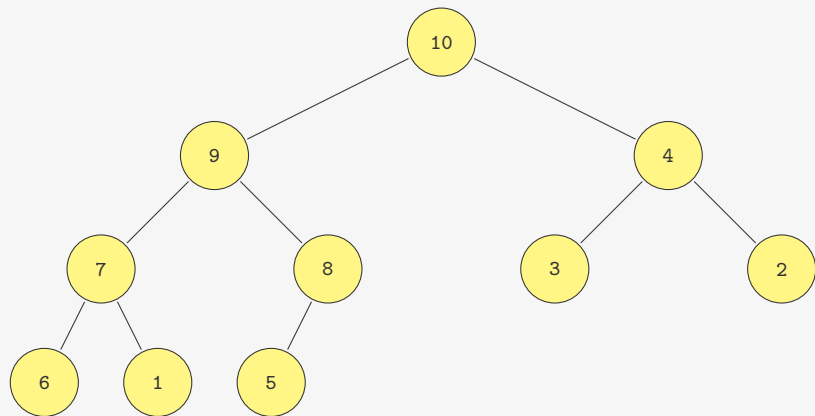
Max-Heap



Em um Heap (de máximo):

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

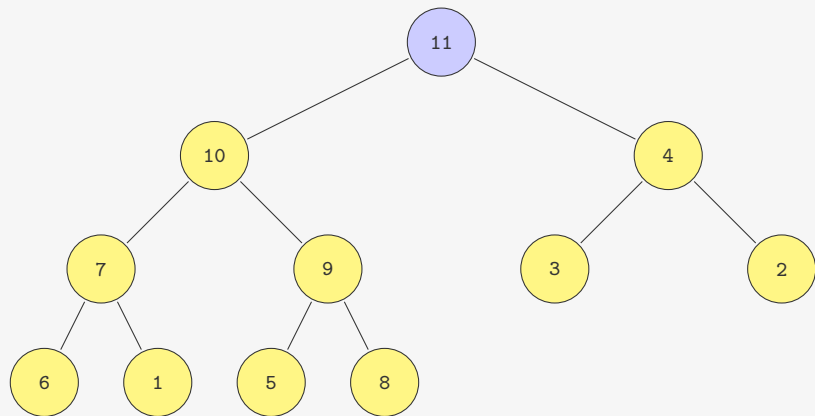
Max-Heap



Note que **não** é uma árvore binária de busca!

- E os dados estão bem menos estruturados
- pois estamos interessados apenas no 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

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

Tempo de **insere**:

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

Extraindo o Máximo

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

Extraindo o Máximo

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

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

Mudando a prioridade de um item

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

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

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

Tempo: $O(\lg n)$

- mas precisamos saber a posição do item no heap
- e percorrer o heap para achar o item leva $O(n)$
 - dá para fazer melhor?

Posição do item no heap

Se os itens tiverem um campo `id` com valores de `0` a `n-1`

- Criamos um vetor de `n` posições
- Como parte da `struct` do heap
- Que armazena a posição do item no heap
- Em $O(1)$ encontramos a posição do item no heap

Como modificar os algoritmos para atualizar esse vetor?

- Toda vez que fizer uma troca, troque também as posições

E se os itens não tiverem esse campo `id`?

- Atribua `ids` aos elementos você mesmo
- Use uma estrutura de dados para encontrar o `id` rapidamente
- Ex: ABBs ou Tabela de Hashing (veremos no futuro)

Exercício

Crie versões iterativas de `desce_no_heap` e `sobe_no_heap`