MC-202
Filas de Prioridade e Heap

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

Universidade Estadual de Campinas

2º semestre/2023

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

- Mas veremos uma implementação muito melhor

A função troca

Várias vezes iremos trocar dois elementos de posição

Para tanto, vamos usar a seguinte função:

```
void troca(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}
```

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

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

typedef struct {
  Item *v; // vetor de Items alocado dinamicamente
  int n, tamanho; // n: qtde de elementos, tamanho: qtde alocada
} 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

```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 $\mathrm{O}(n)$

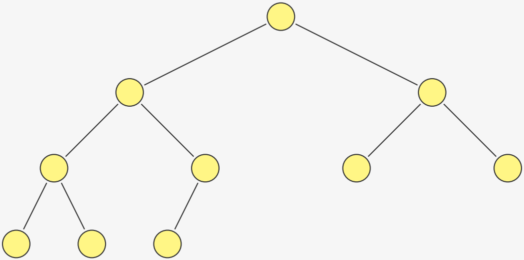
- Se mantiver o vetor ordenado, os tempos se invertem

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

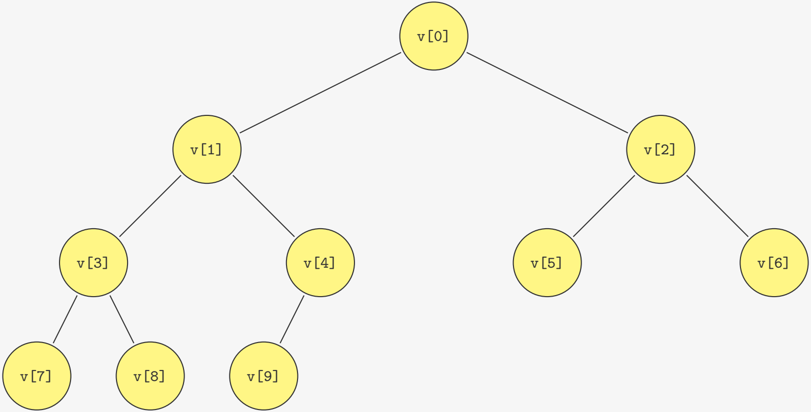
Exemplo:



Quantos níveis tem uma árvore binária completa com n nós?

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

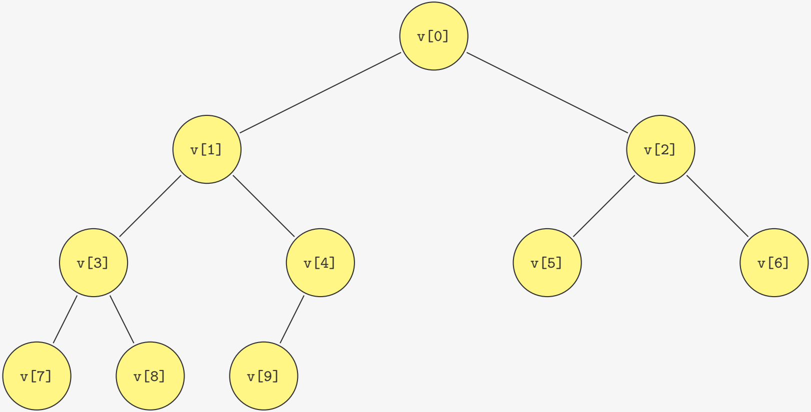
Árvores Binárias Completas e Vetores



Podemos representar tais árvores usando vetores

- Isso é, não precisamos de ponteiros

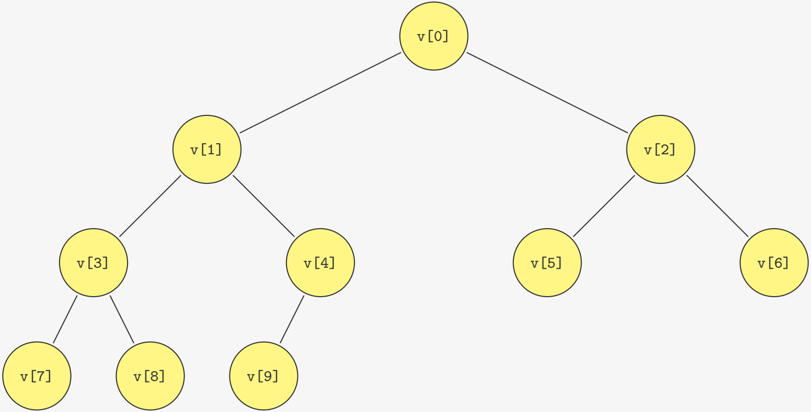
Árvores Binárias Completas e Vetores



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]`

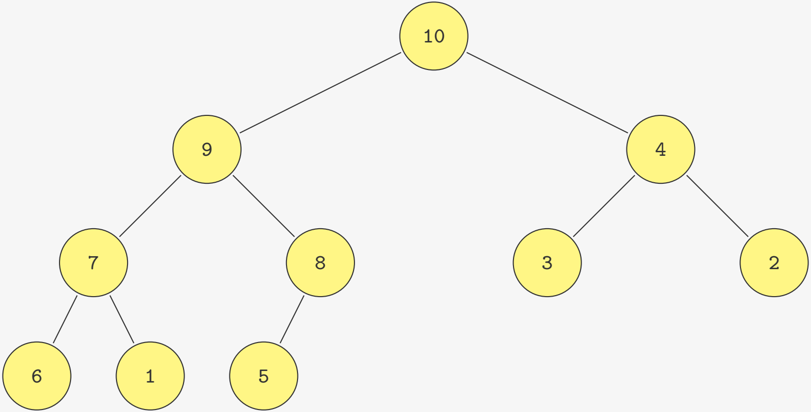
Max-Heap



Em um Heap (de máximo):

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

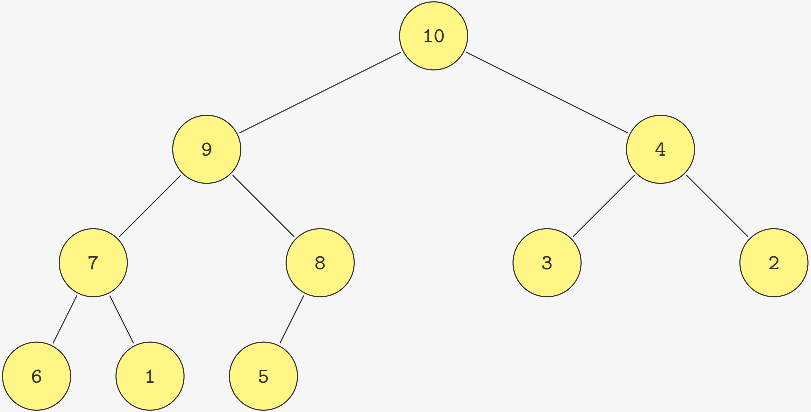
Max-Heap



Em um Heap (de máximo):

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

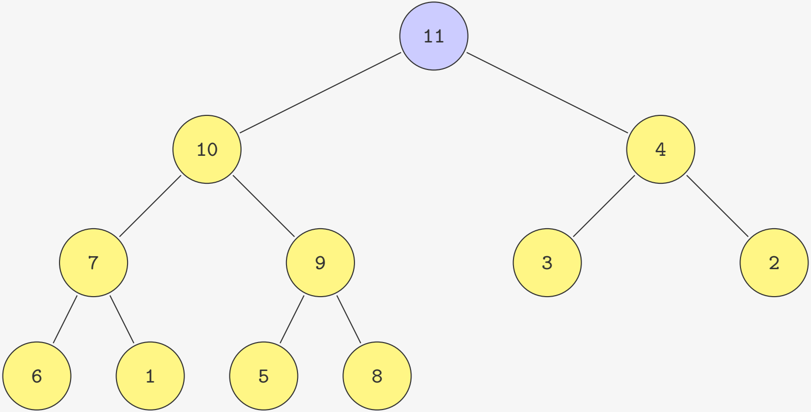
Max-Heap



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

```
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

```
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

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

- mas precisamos saber a posição do item no heap
- e percorrer o heap para achar o item leva $\mathrm{O}(n)$
  - 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`