

AED2 - Aula 12

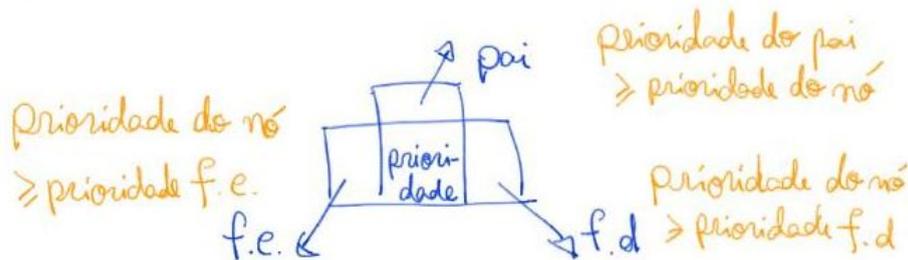
Heap e ordenação por seleção eficiente (heapsort)

Heap é uma estrutura de dados eficiente para

- implementar o tipo abstrato de dados chamado Fila de Prioridades.
- Uma fila de prioridades deve suportar as operações de:
 - inserção de um elemento com um certo valor de prioridade,
 - edição da prioridade de um elemento (operação menos comum),
 - remoção do elemento com maior (ou menor) prioridade.
 - Esta propriedade não atende maior e menor simultaneamente.
 - Por isso temos filas de prioridade (e heaps)
 - de máximo e de mínimo.
 - vamos focar nas versões de máximo.

Um heap de máximo é uma árvore binária

- que respeita a propriedade do heap de máximo, i.e.,
 - o valor da prioridade de um nó é \geq que a prioridade de seus filhos.



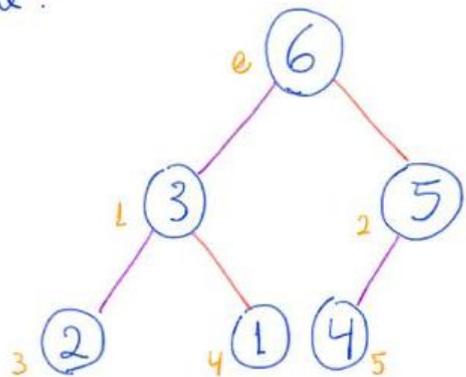
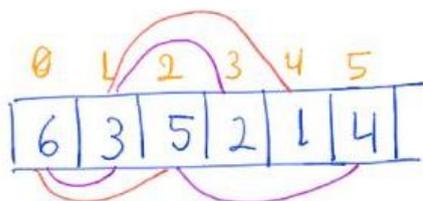
- Note que essa propriedade difere da propriedade de busca em árvores.

Embora corresponda a uma árvore binária,

- o heap é tradicionalmente implementado em um vetor,
 - como mostra o seguinte exemplo

- Exemplo de heap na visão de:

- árvore binária
- linearizado em vetor



- Em tal implementação o vetor é preenchido da esquerda para a direita,
 - enquanto os nós da árvore são lidos de cima para baixo e

- em cada nível, também da esquerda para a direita.
- Isso é possível pois o heap é uma árvore binária quase completa.

Numa árvore binária completa

- cada nível p tem 2^p nós.
- Lembrando que a raiz fica no nível 0
 - e que o nível aumenta cada vez que
 - vamos de um nó para seu filho esquerdo ou direito
- observe que, numa árvore binária completa
 - todo nível tem o máximo de nós possíveis.

Numa árvore binária quase completa

- cada nível i tem 2^i nós,
 - com a possível exceção do último nível.
 - Se for esse o caso, no último nível as posições dos nós estão ocupadas da esquerda para a direita, sem espaços vazios.

Essa definição possibilita a implementação de um heap com m elementos

- em um vetor v que começa em 0 e vai até $m - 1$.
- Para tanto, dado um elemento na posição i ,
 - é essencial saber quem é pai, filho esquerdo e filho direito de i .
- Como numa árvore binária o número de nós dobra a cada novo nível,
 - a posição do pai de i é aproximadamente metade de i ,
 - e a posição dos filhos é aproximadamente o dobro de i .
- Mais precisamente, determinamos as posições usando as seguintes fórmulas

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

```
#define FILHO_ESQ(i) (2 * i + 1)
```

```
#define FILHO_DIR(i) (2 * i + 2)
```

- Traduzindo a propriedade do heap de máximo para a implementação em vetor temos
 - $v[\text{PAI}(i)] = v[(i - 1) / 2] \geq v[i]$
 - $v[i] \geq v[2 * i + 1] = v[\text{FILHO_ESQ}(i)]$
 - $v[i] \geq v[2 * i + 2] = v[\text{FILHO_DIR}(i)]$
- Observe que, o nó raiz, que não tem pai, fica na posição 0.
- Além disso, se $\text{FILHO_ESQ}(i)$ ou $\text{FILHO_DIR}(i)$ forem $\geq m$,
 - então i não tem filho esquerdo ou direito, respectivamente.
- Note que os nós da segunda metade do vetor não tem filhos, já que
 - para $i \geq m / 2$ temos $\text{FILHO_ESQ}(i) = 2 * i + 1 \geq 2 * m / 2 + 1 \geq m$.
- De fato, em um heap (e em toda árvore binária quase completa),
 - o número de folhas (nós sem filhos) é pelo menos metade do total.

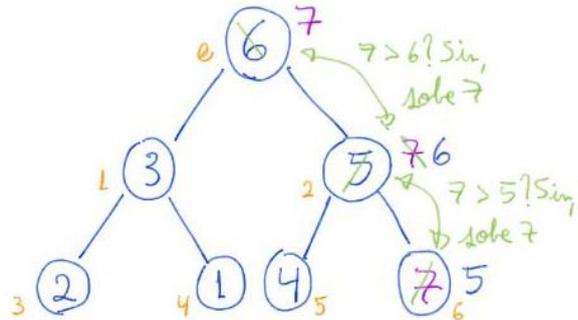
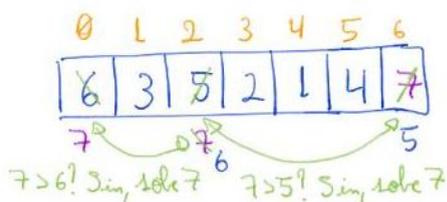
Lembrando que o nível p de uma árvore binária quase-completa

- tem 2^p nós, exceto talvez se p for o último nível,
 - temos que tais nós são $2^p - 1, 2^p, 2^p + 1, \dots, 2^{(p+1)} - 2$
- Dessa observação conseguimos determinar que
 - o nível de um nó i é $\text{piso}(\lg(i+1))$
- Demonstração:
 - seja p o nível do nó i . Então
 - $2^p - 1 \leq i \leq 2^{(p+1)} - 2$
 - $2^p \leq i + 1 \leq 2^{(p+1)} - 1$
 - $2^p \leq i + 1 < 2^{(p+1)}$
 - $p \leq \lg(i+1) < p + 1$
 - Como p é inteiro, $\text{piso}(\lg(i+1)) = p$
- Como o último nó é $m - 1$, conseguimos determinar que o último nível é
 - $\text{piso}(\lg(m - 1 + 1)) = \text{piso}(\lg m)$
 - e o total de níveis é $\text{piso}(\lg m) + 1$,
 - já que o primeiro nível é 0.

Agora vamos estudar as duas funções mais importantes para manutenção do heap.

- A primeira é a sobe heap,
 - que veremos aplicada à seu uso mais comum,
 - a inserção de um novo elemento.

- Exemplo de inserção no heap usando função sobe heap



Código da sobeHeap:

```
void sobeHeap(int v[], int m)
{
    int f = m;
    while (f > 0 && v[PAI(f)] < v[f])
    {
        troca(&v[f], &v[PAI(f)]);
        f = PAI(f);
    }
}
```

- Exemplos de uso da sobeHeap:

```
printf("Testando sobeHeap com elemento da ultima posicao\n");
```

```

sobeHeap(v, m - 1);
printf("Criando um max heap mandando todos subirem da esquerda pra direita\n");
for (i = 1; i < m; i++)
    sobeHeap(v, i);

```

Corretude e invariante da sobeHeap:

- o invariante principal que vale no início de cada iteração é
 - todo elemento em $v[0 .. m]$ respeita a propriedade do heap,
 - exceto, possivelmente, pelo elemento f .
 - Isto é, $v[i] \leq v[\text{PAI}(i)] = v[(i - 1) / 2]$ vale para todo $i \neq f$.

Eficiência da sobeHeap:

- número de operações é $O(\lg m)$,
 - pois no início $f = m$ e em cada iteração f é dividido por 2.

Código da insereHeap:

```

int insereHeap(int v[], int m, int x)
{
    v[m] = x;
    sobeHeap(v, m);
    return m + 1;
}

```

- Exemplos de uso da insereHeap:

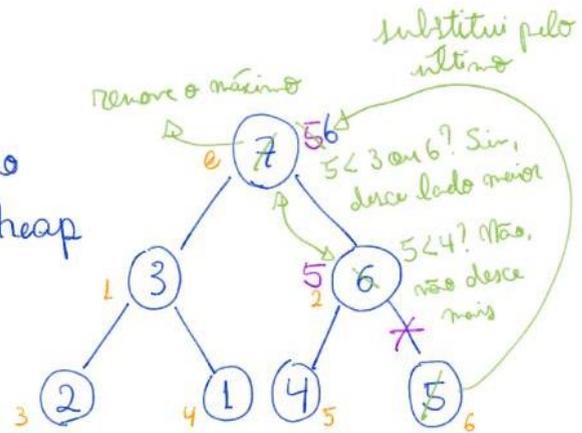
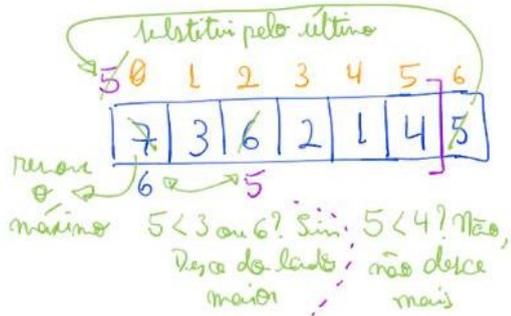
```

printf("Inserindo novo elemento no max heap\n");
m = insereHeap(v, m, 999);
printf("Criando novo max heap usando insereHeap - ordem direta\n");
m = 0;
for (i = 0; i < n; i++)
    m = insereHeap(v, m, i);
printf("Criando novo max heap usando insereHeap - ordem inversa\n");
m = 0;
for (i = 0; i < n; i++)
    m = insereHeap(v, m, n - i - 1);

```

- A segunda função mais importante na manutenção do heap é a desce heap,
 - que veremos aplicada à seu uso mais comum,
 - a remoção de um novo elemento.

- Exemplo de remoção do máximo de um heap seguida de restauração do heap usando a função `desceHeap`



Código da `desceHeap`:

```
void desceHeap(int v[], int m, int k)
{
    int p = k, f;
    while (FILHO_ESQ(p) < m && (v[FILHO_ESQ(p)] > v[p] || (FILHO_DIR(p) < m &&
v[FILHO_DIR(p)] > v[p]))
    {
        f = FILHO_ESQ(p);
        if (FILHO_DIR(p) < m && v[FILHO_DIR(p)] > v[f])
            f = FILHO_DIR(p);
        troca(&v[p], &v[f]);
        p = f;
    }
}
```

• Exemplos de uso da `desceHeap`:

```
printf("Testando desceHeap com elemento da primeira posicao\n");
v[0] = 0;
desceHeap(v, m, 0);
printf("Criando um max heap mandando todos descerem da direita pra esquerda\n");
for (i = m - 1; i >= 0; i--)
    desceHeap(v, m, i);
```

Corretude e invariante da `desceHeap`:

- o invariante principal que vale no início de cada iteração é
 - todo elemento em $v[0 .. m - 1]$ respeita a propriedade do heap,
 - exceto, possivelmente, pelo elemento p .
 - Isto é, $v[i] \geq v[\text{FILHO_ESQ}(i)] = v[2 * i + 1]$
 - e $v[i] \geq v[\text{FILHO_DIR}(i)] = v[2 * i + 2]$ vale para todo $i \neq p$.

Eficiência da `desceHeap`:

- número de operações é $O(\lg m)$,

- pois em cada iteração descemos um nível na árvore do heap
 - e o maior nível é $\text{piso}(\lg m)$.

Código da `removeHeap`:

```
int removeHeap(int v[], int m, int *x)
{
    *x = v[0];
    troca(&v[0], &v[m - 1]);
    desceHeap(v, m, 0);
    return m - 1;
}
```

- Exemplo de uso do `removeHeap`:

```
m = removeHeap(v, m, &x);
```

Agora que nosso heap de máximo está funcionando,

- podemos voltar a atenção ao problema de ordenar
 - um vetor `v` de tamanho `n`.

Começemos relembando a ideia do `selectionSort`,

- que percorre o vetor da esquerda para a direita
 - e em cada iteração busca o menor elemento do sufixo do vetor
 - colocando este na posição corrente.

Código do `selectionSort`:

```
void selectionSort(int v[], int n)
{
    int i, j, ind_min, aux;
    for (i = 0; i < n - 1; i++)
    {
        ind_min = i;
        for (j = i + 1; j < n; j++)
            if (v[j] < v[ind_min])
                ind_min = j;
        aux = v[i];
        v[i] = v[ind_min];
        v[ind_min] = aux;
    }
}
```

Chama atenção neste algoritmo que ele realiza sucessivas

- buscas pelo menor elemento de um conjunto.
- Mas nós acabamos de estudar uma estrutura de dados que é muito eficiente
 - justamente nesse tipo de busca.

Da união dessas ideias surge o algoritmo `heapSort`.

- Primeiro re-organizamos os elementos do vetor
 - de modo a construir um heap de máximo.
- Então, em cada iteração,
 - extraímos o maior elemento do heap e o colocamos
 - na última posição do vetor corrente.
- É basicamente a ideia do selectionSort com um heap
 - o motivo de usarmos um heap de máximo,
 - e não de mínimo,
 - será explicado em seguida.

Código do heapsort1:

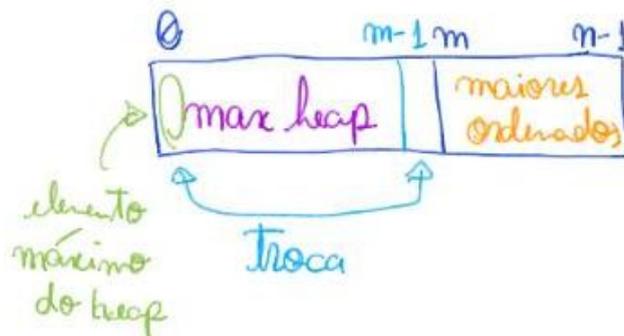
```
void heapsort1(int v[], int n)
{
    int i, m = n;
    for (i = 1; i < n; i++)
        sobeHeap(v, i);
    for (m = n - 1; m > 0; m--)
    {
        troca(&v[0], &v[m]);
        desceHeap(v, m, 0);
    }
}
```

- Exemplo de uso do heapsort1:

```
printf("Ordenando com heapsort1\n");
heapsort1(v, m);
```

Corretude e invariante da heapSort1:

- os invariantes principais que valem no início de cada iteração do segundo laço são
 - $v[m .. n - 1]$ está ordenado em ordem crescente,
 - $v[0 .. m - 1]$ é um heap de máximo,
 - $v[0 .. m - 1] \leq v[m .. n - 1]$



Eficiência de tempo da heapsort1:

- o algoritmo executa da ordem de $n \lg n$ operações, i.e., $O(n \lg n)$

- pois tanto o primeiro quando o segundo laço executam $O(n)$ vezes
 - e em cada iteração invocam uma operação do heap
 - que leva tempo $O(\lg n)$.

Código do heapsort2:

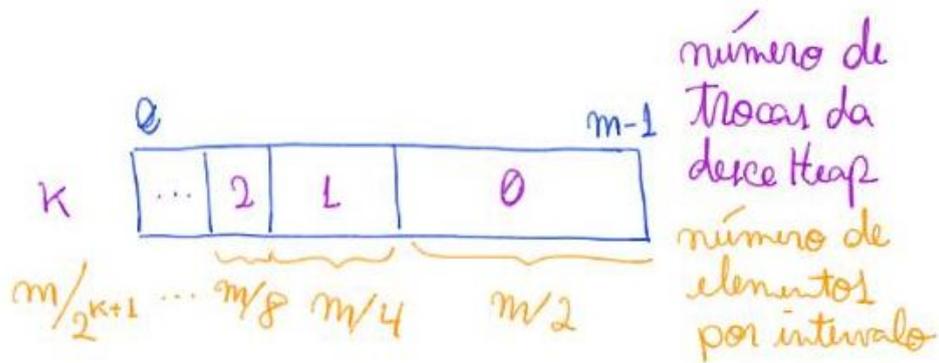
```
void heapsort2(int v[], int n)
{
    int i, m = n;
    for (i = n / 2; i >= 0; i--)
        desceHeap(v, n, i);
    for (m = n - 1; m > 0; m--)
    {
        troca(&v[0], &v[m]);
        desceHeap(v, m, 0);
    }
}
```

- Exemplo de uso do heapsort2:

```
printf("Ordenando com heapsort2\n");
heapsort2(v, m);
```

Eficiência de tempo da heapsort2:

- o algoritmo executa da ordem de $n \lg n$ operações, i.e., $O(n \lg n)$
 - pois no segundo laço ele realiza n extrações do máximo,
 - cada uma seguida por uma operação de desceHeap
 - que realiza da ordem de $O(\lg n)$ operações.
- No entanto, vale destacar que a constante de tempo desse algoritmo
 - é melhor que a do anterior, porque no primeiro laço
 - ele contrói o heap em tempo linear, i.e., $O(n)$.
 - Isso pode parecer estranho, já que o primeiro laço realiza
 - $O(n)$ chamadas à função desceHeap,
 - que leva tempo $O(\lg n)$.
- Vamos fazer uma análise mais cuidadosa. Note que
 - para os $n/2$ últimos elementos do vetor nenhuma troca é realizada,
 - para os próximos $n/4$ desceHeap fará no máximo 1 troca,
 - e para os próximos $n/8$ desceHeap fará no máximo 2 trocas.
- Em geral, teremos $n/2^{(k+1)}$ elementos realizando k trocas
 - para k entre 0 e $\lg n$.
- Assim, o total de trocas é dado pelo somatório
 - $n/2 * 0 + n/4 * 1 + n/8 * 2 + \dots + n/2^{(k+1)} * k + \dots + 1 * \lg n$
 - cujo valor $\leq O(n)$



Estabilidade:

- ordenação não é estável, pois a manipulação do heap destrói a estabilidade
 - para visualizar, considere a troca que ocorre antes do descHeap,
 - nela o último elemento do heap corrente
 - vai para a posição do primeiro,
 - invertendo a posição relativa deste com todos os seus iguais.

Eficiência de espaço:

- ordenação é in place, pois não usa vetor auxiliar,
 - e as únicas variáveis utilizadas
 - tem tamanho constante em relação ao vetor de entrada.
- De fato, usamos um heap de máximo ao invés de um heap de mínimo
 - para que o algoritmo possa ser in-place,
 - já que ao removermos o elemento máximo do heap,
 - ele diminui no final do vetor,
 - e é nessa posição liberada no final que devemos colocar
 - o maior elemento que acabamos de remover.

Curiosidade:

- se construirmos o heap num vetor auxiliar,
 - o algoritmo deixa de ser in place,
- Neste caso, passamos a poder utilizar um heap de mínimo, por exemplo.
- Além disso, seu melhor caso pode mudar,
 - pois quando o vetor original já está em ordem crescente
 - a construção do heap não precisa inverter todos os elementos.
- Destaco que isso é só uma curiosidade, pois
 - a economia de memória é desejável,
 - e a implementação mais eficiente do heapsort é a segunda que vimos.

Código do heapsort3:

```
void heapsort3(int v[], int n)
```

```

{
    int i, m = n, *w;
    w = mallocSafe(sizeof(int) * n);
    for (i = 0; i < n; i++)
        w[n - i - 1] = v[i];
    for (i = 1; i < n; i++)
        sobeHeap(w, i);
    for (m = n - 1; m >= 0; m--)
    {
        v[m] = w[0];
        w[0] = w[m];
        desceHeap(w, m, 0);
    }
    free(w);
}

```

- Exemplo de uso do heapsort3:

```

printf("Ordenando com heapsort3\n");
heapsort3(v, m);

```

Animação:

- Visualization and Comparison of Sorting Algorithms - www.youtube.com/watch?v=ZZuD6iUe3Pc