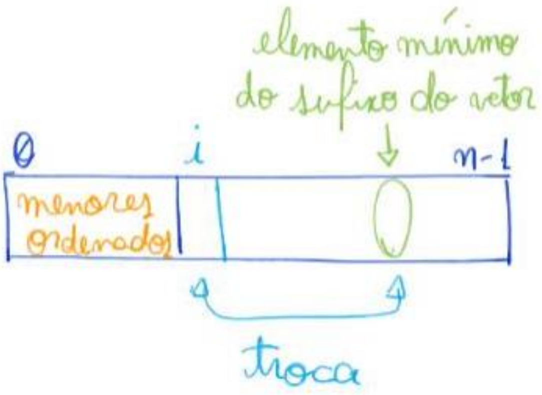
AED1 - Aula 22
Ordenação por seleção eficiente (heapSort),
construção de heap em tempo linear (heapify)

Nesta aula vamos estudar uma aplicação do heap de máximo,
- que estudamos na aula sobre filas de prioridade.

Na maioria das aplicações do Heap,
- percebemos que ele pode nos ajudar
  - a resolver um problema e/ou melhorar um algoritmo,
    - quando nosso algoritmo realiza sucessivas requisições
      - pelo elemento máximo (ou mínimo) de um conjunto.
- Isto pode acontecer em inúmeras situações,
  - como quando temos que decidir o próximo evento a ocorrer,
    - sendo que cada evento tem uma importância
      - ou um tempo associado.
- Neste caso, manter os eventos organizados em um heap
  - nos permite decidir qual é o próximo evento com grande eficiência.

A aplicação do heap que veremos em seguida,
- envolve o problema da ordenação,
  - no qual temos um vetor v de tamanho n
- e queremos colocar seus elementos em ordem crescente.

Começaremos relembrando 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;
    }
}
```

Eficiência de tempo:

- O(n^2), pois são realizadas O(n) buscas pelo mínimo do sufixo do vetor,
    - cada uma levando tempo linear, i.e., O(n).

Como este algoritmo realiza sucessivas buscas

- pelo menor elemento de um conjunto,
    - é um candidato natural a ser melhorado usando um Heap.

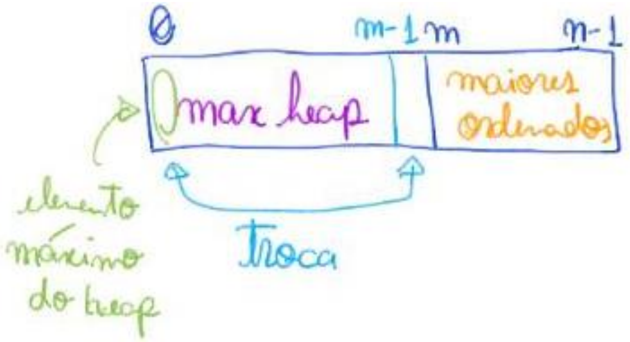
HeapSort

Da união da ideia do selectionSort com a estrutura de dados heap

- surge o algoritmo heapSort, cuja ideia é:
    - Colocar os elementos do vetor em um heap,
    - em cada iteração extrair um elemento do heap,
    - e colocá-lo na posição correta no vetor ordenado.
- Como cada extração do heap leva tempo O(log n),
    - e são necessárias O(n) extrações,
- esse algoritmo deve levar tempo O(n log n) para ordenar o vetor.

Entrando um pouco mais nos detalhes técnicos desse algoritmo,

- 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.



- 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;
    for (i = 1; i < n; i++) // construindo o heap em tempo O(n lg n)
        sobeHeap(v, i);
    for (m = n; m > 0; m--)
    {
        troca(&v[0], &v[m - 1]); // colocando o máximo no final
        desceHeap(v, m - 1, 0);  // restaurando o Heap
    }
}
```

- Exemplo de uso do heapsort1:

```
    printf("Ordenando com heapSort1\n");
    heapSort1(v, n);
```

Corretude e invariante do heapSort1:

- Os invariantes principais, que valem no início do segundo laço são
    - v[0 .. n - 1] é uma permutação do vetor original,
    - v[m .. n - 1] está ordenado em ordem crescente,
    - v[0 .. m - 1] é um heap de máximo,
    - v[0 .. m - 1] <= v[m .. n - 1].
- Note que esses invariantes implicam a ordenação do vetor na última iteração.

Eficiência de tempo do heapSort1:

- O algoritmo executa da ordem de n lg n operações, i.e., O(n log n),
    - pois tanto o primeiro quanto o segundo laços executam O(n) vezes
- e em cada iteração invocam uma operação do heap
    - que leva tempo O(log n).

Construção de heap em tempo linear (heapify)

Heapify é uma operação auxiliar interessante na manipulação de heaps,

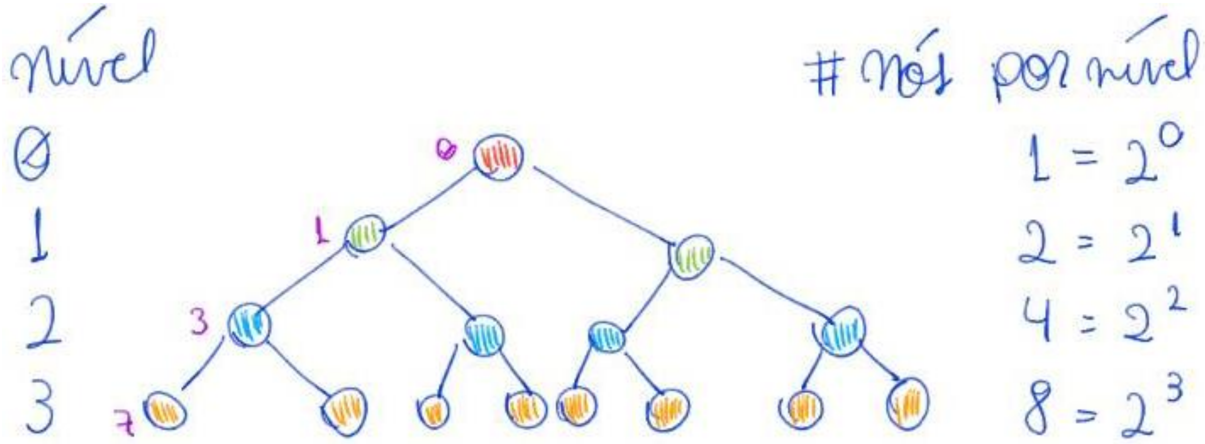
- que transforma um vetor de tamanho m em um heap
  - usando a função desceHeap
- e gastando apenas tempo linear, i.e., O(m).

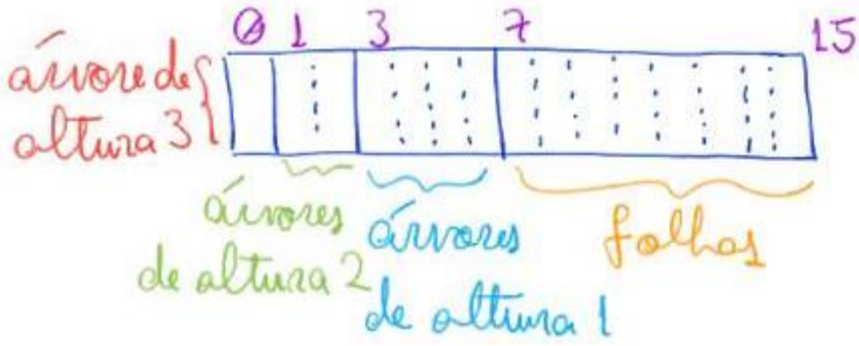
Código da heapify:

```
    printf("Heapify: criando um max heap mandando todos descerem da direita pra esquerda\n");
    for (i = m / 2; i >= 0; i--)
        desceHeap(v, m, i);
```

Análise de corretude:

- Observe que esta função está construindo o Heap de baixo para cima,
  - de modo que uma chamada de desceHeap no índice i
    - faz a árvore binária enraizada em i
      - se transformar em um heap.

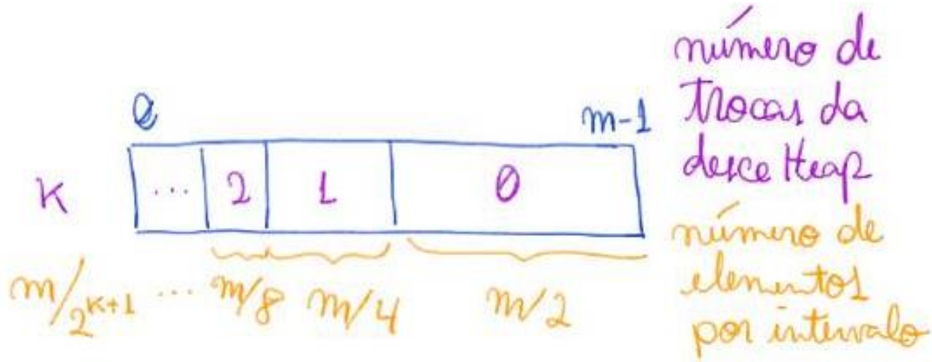




- Note que, isso só funciona porque
  - como as chamadas vão da direita para a esquerda,
- as árvores binárias correspondentes aos filhos de i
  - já são heaps válidos quando mandamos descer i.

Análise de eficiência de tempo:

- A princípio, pode parecer que essa função leva tempo O(m lg m),
    - já que o laço realiza O(m) chamadas à função desceHeap,
        - que leva tempo O(lg m).
- Vamos fazer uma análise mais cuidadosa. Note que
    - para os m/2 últimos elementos do vetor
        - nenhuma troca é realizada,
    - para os próximos m/4
        - desceHeap fará no máximo 1 troca,
    - e para os próximos m/8
        - desceHeap fará no máximo 2 trocas.



- Em geral, teremos m/2^(k+1) elementos realizando k trocas
    - para k entre 0 e lg(m) - 1.
- Assim, o total de trocas é limitado superiormente pelo somatório
    - m/2 * 0 + m/4 * 1 + m/8 * 2 + … + m/2^(k+1) * k + … + 1 * lg m,

total de trocas realizadas pelos desceHeap ≤

$$\leq \frac{m}{2}\cdot 0 + \frac{m}{4}\cdot 1 + \frac{m}{8}\cdot 2 + \frac{m}{16}\cdot 3 + \dots + \frac{m}{2^{k+1}}\, k + \dots + \frac{m}{2^{\lg m}}\cdot(\lg m - 1) =$$

$$= \sum_{i=0}^{\lg m - 1} \frac{m}{2^{i+1}}\cdot i = m \cdot \sum_{i=0}^{\lg m - 1} i/2^{i+1} = \frac{m}{2}\cdot \sum_{i=1}^{\lg m - 1} i/2^{i} = \dots \circledast_2$$

$$\boxed{\sum_{i=1}^{+\infty} i/2^i} = 1/2 + 2/2^2 + 3/2^3 + 4/2^4 + \dots + k/2^k + \dots$$

$$\begin{aligned}
= &\left(1/2^1 + 1/2^2 + 1/2^3 + 1/2^4 + \dots\right) \\
&+ \left(1/2^2 + 1/2^3 + 1/2^4 + \dots\right) \\
&+ \left(1/2^3 + 1/2^4 + \dots\right) \\
&+ \left(1/2^4 + \dots\right) \\
&\vdots
\end{aligned}$$

$$= \sum_{i=1}^{+\infty} 1/2^i + \sum_{i=2}^{+\infty} 1/2^i + \sum_{i=3}^{+\infty} 1/2^i + \sum_{i=4}^{+\infty} 1/2^i + \dots$$

$$= \sum_{j=1}^{+\infty} \sum_{i=j}^{+\infty} 1/2^i = \dots \circledast_1$$

---

a $\sum_{i=j}^{+\infty} 1/2^i = \frac{1}{2^j} + \frac{1}{2^{j+1}} + \frac{1}{2^{j+2}} + \dots$

b $\frac{1}{2} \cdot \sum_{i=j}^{+\infty} 1/2^i = \frac{1}{2^{j+1}} + \frac{1}{2^{j+2}} + \dots$

a $-$ b $\left(1 - \frac{1}{2}\right) \sum_{i=j}^{+\infty} 1/2^i = \left(\frac{1}{2^j} + \cancel{\frac{1}{2^{j+1}}} + \cancel{\frac{1}{2^{j+2}}} + \cancel{\dots}\right) - \left(\cancel{\frac{1}{2^{j+1}}} + \cancel{\frac{1}{2^{j+2}}} + \cancel{\dots}\right)$

$$\frac{1}{2} \sum_{i=j}^{+\infty} 1/2^i = \frac{1}{2^j} \Longrightarrow \boxed{\sum_{i=j}^{+\infty} 1/2^i = \frac{1}{2^{j-1}}}$$

$$\dots\circledast_1 = \sum_{j=1}^{+\infty} \sum_{i=j}^{+\infty} 1/2^i = \sum_{j=1}^{+\infty} \frac{1}{2^{j-1}} = 2 \sum_{i=1}^{+\infty} \frac{1}{2^i} = 2 \frac{1}{2^{1-1}} = \boxed{2}$$

$$\otimes_2 = \frac{m}{2} \cdot \sum_{i=1}^{\lg m - 1} i/2^i \leq \frac{m}{2} \sum_{i=1}^{+\infty} i/2^i = \frac{m}{2} \cdot 2 = m$$

- Ou seja, o total de trocas realizadas pelo desceHeap é <= m,
  - portanto o custo total do heapify é O(m).

Agora usaremos esta abordagem de construção do heap,
- para melhorar a eficiência do heapSort.

Código do heapSort2:
```
void heapSort2(int v[], int n)
{
    int i, m;
    for (i = n / 2; i >= 0; i--) // construindo o Heap em tempo O(n)
        desceHeap(v, n, i);
    for (m = n; m > 0; m--)
    {
        troca(&v[0], &v[m - 1]); // colocando o máximo no final
        desceHeap(v, m - 1, 0);  // restaurando o Heap
    }
}
```
- Exemplo de uso do heapSort2:
```
    printf("Ordenando com heapSort2\n");
    heapSort2(v, n);
```

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(log n) operações.
- No entanto, vale destacar que a constante de tempo desse algoritmo
  - é melhor que a do anterior, pois no primeiro laço
    - ele constrói o heap em tempo linear, i.e., O(n).

Quiz1:
- Quanto é n^2 para n = 1000? E n lg n para o mesmo valor de n?
  - Supondo um computador com 1GHz, quanto tempo ele deve levar
    - para ordenar um vetor usando selectionSort? E heapSort?
- Quanto é n^2 para n = 1000000? E n lg n para o mesmo valor de n?
  - Supondo um computador com 1GHz, quanto tempo ele deve levar

            - para ordenar um vetor usando selectionSort? E heapSort?
- Faça testes para verificar a precisão das previsões.

Estabilidade:
- Será que este algoritmo preserva a ordem relativa
    - de elementos que possuem a mesma chave?
- Não, esta ordenação não é estável,
    - por conta de transposições que ocorrem ao manipular o heap.
- Para visualizar, considere a troca que ocorre antes do desceHeap.
    - 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 auxiliares utilizadas
        - tem tamanho constante em relação ao vetor de entrada.
- Inclusive, a eficiência no uso de memória é um diferencial do heapSort
    - em relação a outros métodos de ordenação eficientes em tempo,
        - que veremos no futuro.
- Destaco que, 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,
        - mas neste caso passamos a poder utilizar um heap de mínimo.
- 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.
- Observe que, no algoritmo a seguir simulo um heap de mínimo
    - invertendo o valor das chaves passadas para o heap de máximo,
        - e tomando o cuidado de desinverter os valores destas
            - ao copiá-los de volta ao vetor original.

Código do heapSort3:
```
void heapSort3(int v[], int n)
```

```
{
    int i, m, *w;
    w = mallocSafe(sizeof(int) * n);
    for (i = 0; i < n; i++) // copiando para o vetor do heap
        w[i] = -v[i];
    for (i = 1; i < n; i++) // construindo heap de mínimo em w
        sobeHeap(w, i);
    for (m = n; m > 0; m--)
    {
        v[n - m] = -w[0]; // colocando o mínimo no vetor original
        w[0] = w[m - 1]; // colocando o último na raiz do heap
        desceHeap(w, m - 1, 0); // restaurando o heap
    }
    free(w);
}
```

- Exemplo de uso do heapSort3:

```
    printf("Ordenando com heapSort3\n");
    heapSort3(v, n);
```

Quiz2: Sabendo que o heapSort sempre levará tempo O(n log n),

- mas que ele pode levar um pouco mais ou menos tempo,
    - de acordo com o esforço para construir o heap, responda:
- Qual disposição inicial do vetor leva ao melhor caso do heapSort1?
    - E qual disposição inicial do vetor leva ao melhor caso do heapSort3?

Animação:

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