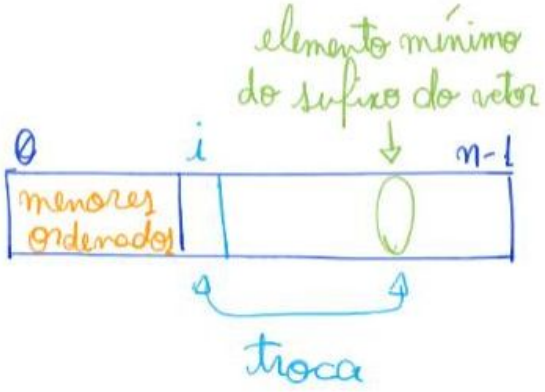
AED1 - Aula 24
Ordenação por seleção eficiente (heapsort), heapify

Agora que nosso heap de máximo está funcionando,
- vamos estudar uma aplicação do Heap.

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,
    - sendo provavelmente a mais direta aquela em que
        - temos de decidir qual 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 fundamental 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;
    }
}
```

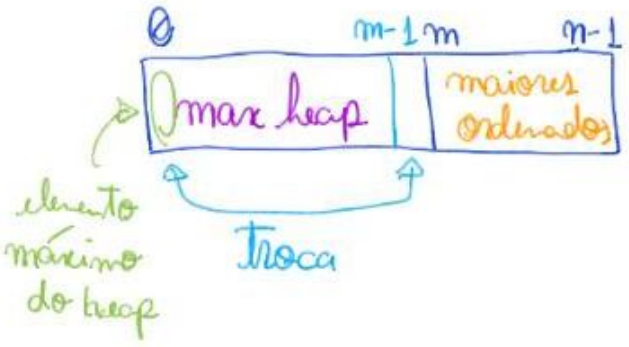
Como este algoritmo realiza

- 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, em que:
    - 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++) // construindo o Heap
```

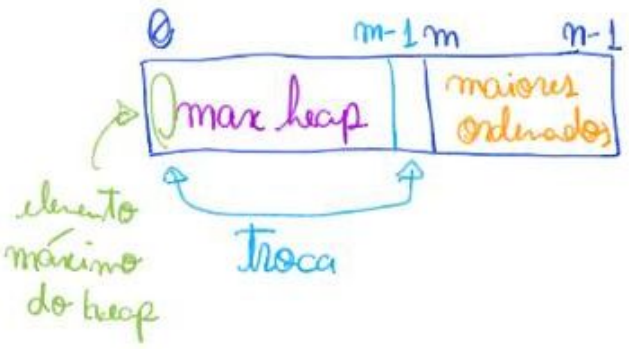
```
        sobeHeap(v, i);
    for (m = n - 1; m > 0; m--)
    {
        troca(&v[0], &v[m]); // colocando o máximo no final
        desceHeap(v, m, 0); // restaurando o Heap
    }
}
```

- 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[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].



Eficiência de tempo da heapsort1:

- no pior caso 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).

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).

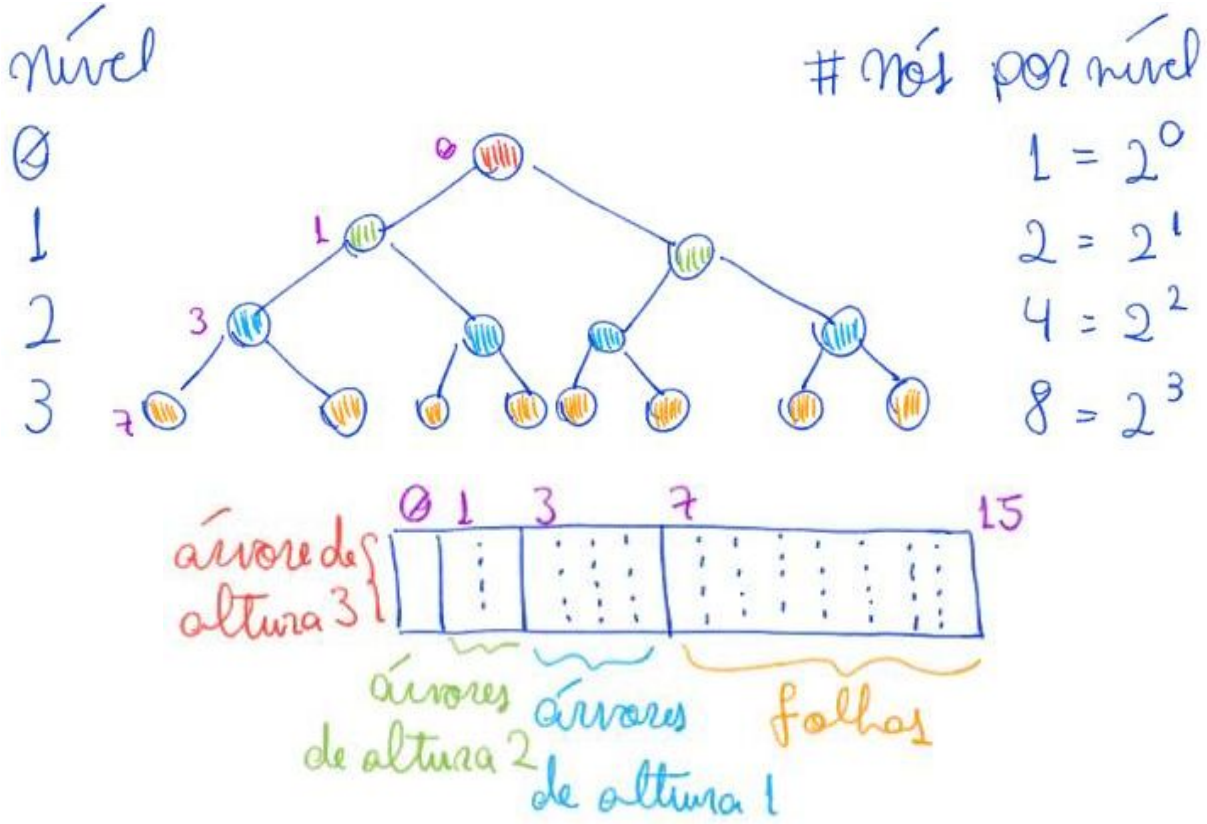
Código da Heapify:

```
    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.

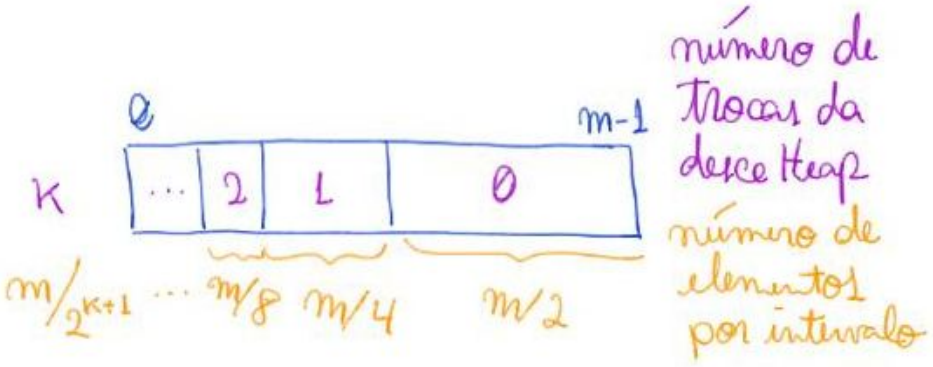


- Note que, isso só funciona porque
    - como as chamadas vão da direita para a esquerda,
        - as “árvores binárias” correspondentes aos filhos de i
            - já são heaps válidos quando mandamos descer i.

Análise de eficiência:

- 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 dado pelo somatório
    - m/2 * 0 + m/4 * 1 + m/8 * 2 + … + m/2^(k+1) * k + … + 1 * lg m,
        - cujo valor é <= uma constante * m, i.e., 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 = n;
   for (i = n / 2; i >= 0; i--) // construindo o Heap em tempo O(n)
       desceHeap(v, n, i);
   for (m = n - 1; m > 0; m--)
   {
       troca(&v[0], &v[m]); // colocando o máximo no final
       desceHeap(v, m, 0);  // restaurando o Heap
   }
}
```

- Exemplo de uso do heapsort2:

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

Eficiência de tempo da heapsort2:

- no pior caso 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 constrói o Heap em tempo linear, i.e., O(n).

Estabilidade:

- Será que este algoritmo preserva a ordem relativa

        - de elementos que tem 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 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.
- 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
            - ao copiá-los de volta ao vetor original.

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[i] = -v[i];
   for (i = 1; i < n; i++) // construindo o Heap de mínimo em vetor auxiliar
       sobeHeap(w, i);
```

```
    for (m = n - 1; m >= 0; m--)
    {
        v[n - 1 - m] = -w[0]; // colocando o mínimo no início do vetor original
        w[0] = w[m];          // colocando o último na raiz do Heap
        desceHeap(w, m, 0);   // restaurando o Heap
    }
    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