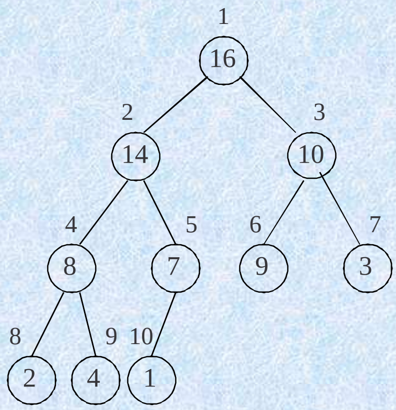
3.3 Ordenação por Heap (Heapsort)

- **Heap descendente (max heap** ou **arvore descendente parcialmente ordenada)** de tamanho *n* é um array que pode ser visto como uma arvore binária quase completa de *n* nós tal que a chave de cada nó seja menor ou igual à chave de seu pai. Cada nó da arvore corresponde um elemento do array.
- A raiz da arvore é *A*[1].

- Dado um elemento *i* no array a posição de seu pai, filho da esquerda e filho da direita podem ser calculados da seguinte forma:

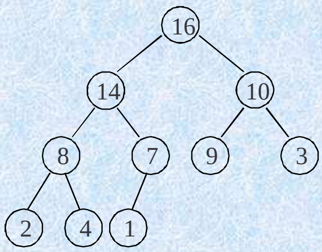
int parent (int i){
return (floor(i/2));
}
	int left (int i){
	return (2*i);
	}
		int right (int i){
		return (2*i+1);
		}



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

- Uma importante propriedade que caracteriza um Heap é que para todo nó *i* diferente da raiz

  A[parent(i)] >= A[i]
- Outra propriedade importante de um Heap é que todo caminhamento em profundidade na arvore gera uma seqüência ordenada de elementos.



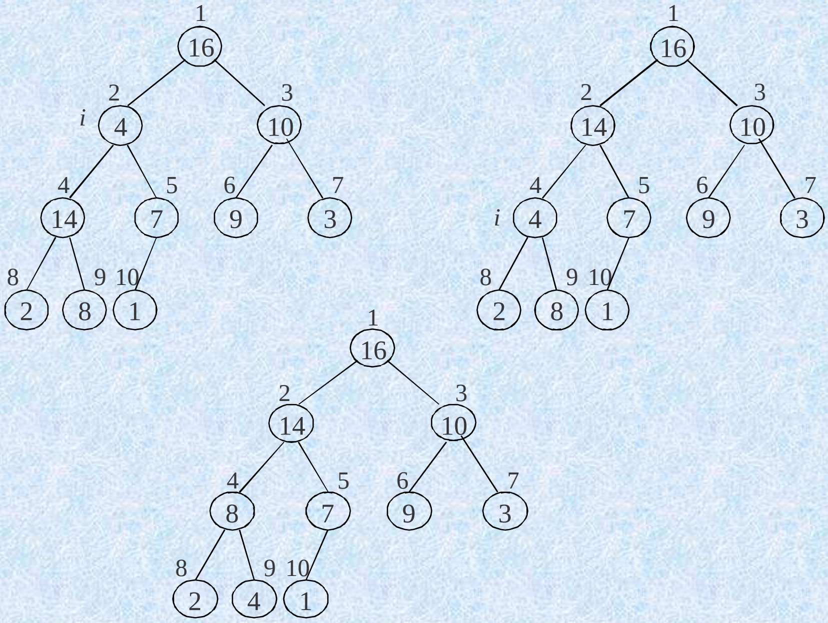
- O algoritmo de ordenação HEAPSORT utiliza três funções:
  - HEAPIFY
  - BUILDHEAP
  - HEAPSORT

- **HEAPIFY**

A função HEAPIFY tem como entrada um array e um índice *i*. Quando HEAPIFY é chamada ela assume que as sub-árvores left(i) e right(i) já satisfazem a propriedade de Heap, mas A[i] pode ser menor que seus filhos. A função de HEAPIFY é tornar a árvore com raiz em *i* um Heap.

```
/* Suponha heapsize uma variavel global e A um array comecando em 0*/
void HEAPIFY(int A[ ], int heapsize, int i)
{
   int l, r, largest;

   l = left(i);  r = right(i);
   if ((l <= heapsize) && (A[l] > A[i]))
           largest = l;
   else
           largest = i;
   if ((r <= heapsize) && (A[r] > A[largest]))
           largest = r;
   if (largest != i)
   {
         swap(A[i], A[largest]);          /* troca a posicao dos elementos */
         HEAPIFY(A, heapsize, largest);
   }
   return;
}
```

Exemplo de HEAPFY



- **BUILDHEAP**

A função BUILDHEAP é gerar a partir de um array qualquer um Heap utilizando o HEAPIFY.

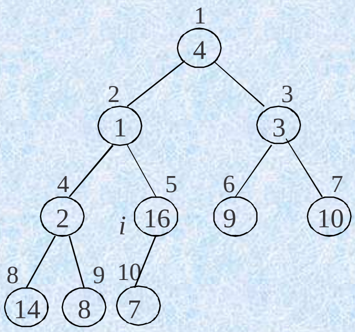
```
void BUILDHEAP (int *A,  int heapsize)
{
   int i;

   for (i = floor(heapsize/2); i > 0; i--)
         HEAPIFY(A, heapsize, i);
}
```

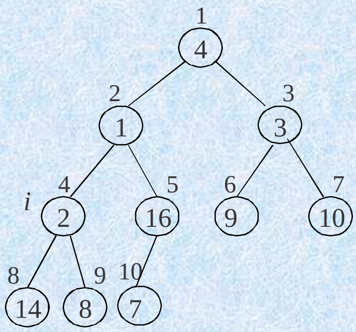
Obs.: os elementos em subvetor A[floor((heapsize/2)+1) .. heapsize] são folhas da arvore, cada um é um heap de um elemento. Portanto, só precisa aplicar HEAPFY para elementos restantes (não folhas).

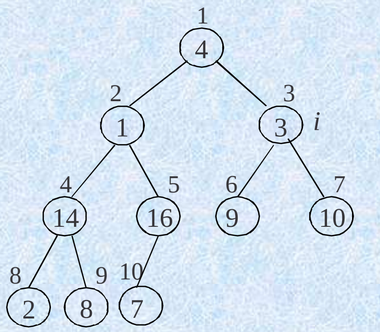
Exemplo:

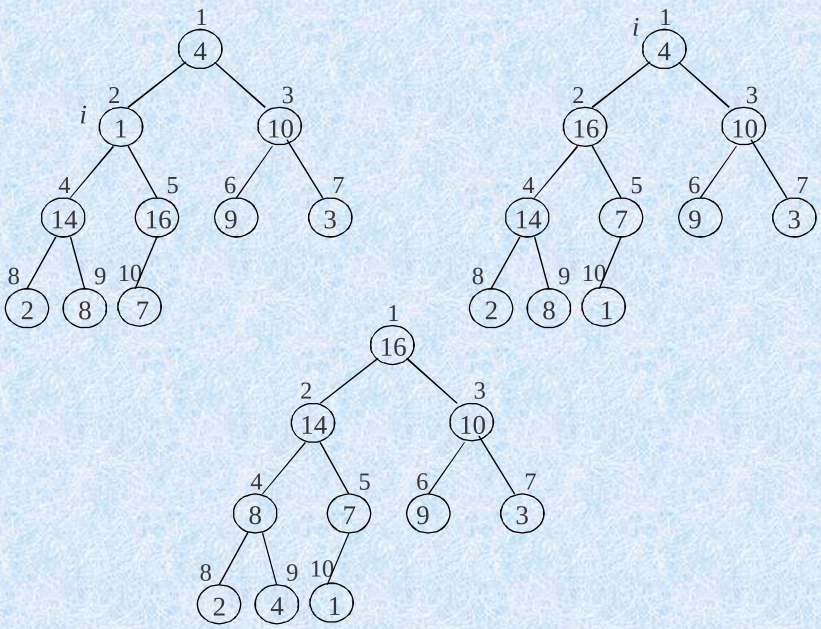
A

4	1	3	2	16	9	10	14	8	7



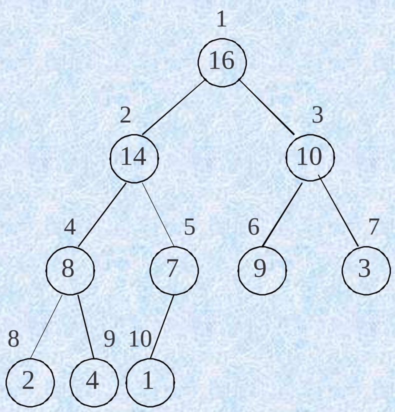


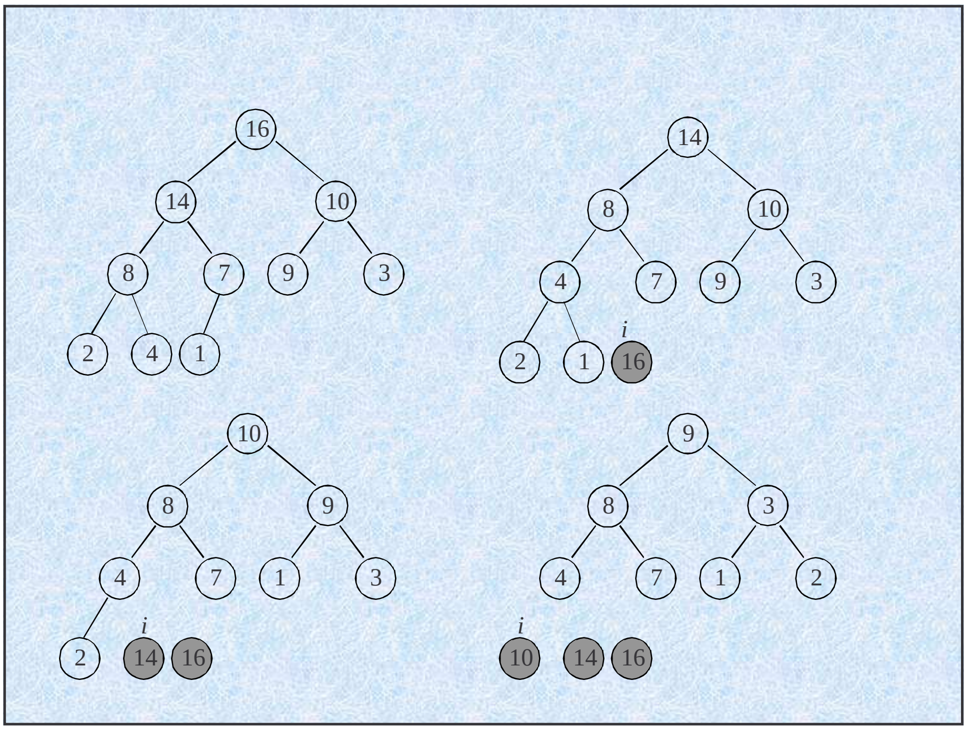


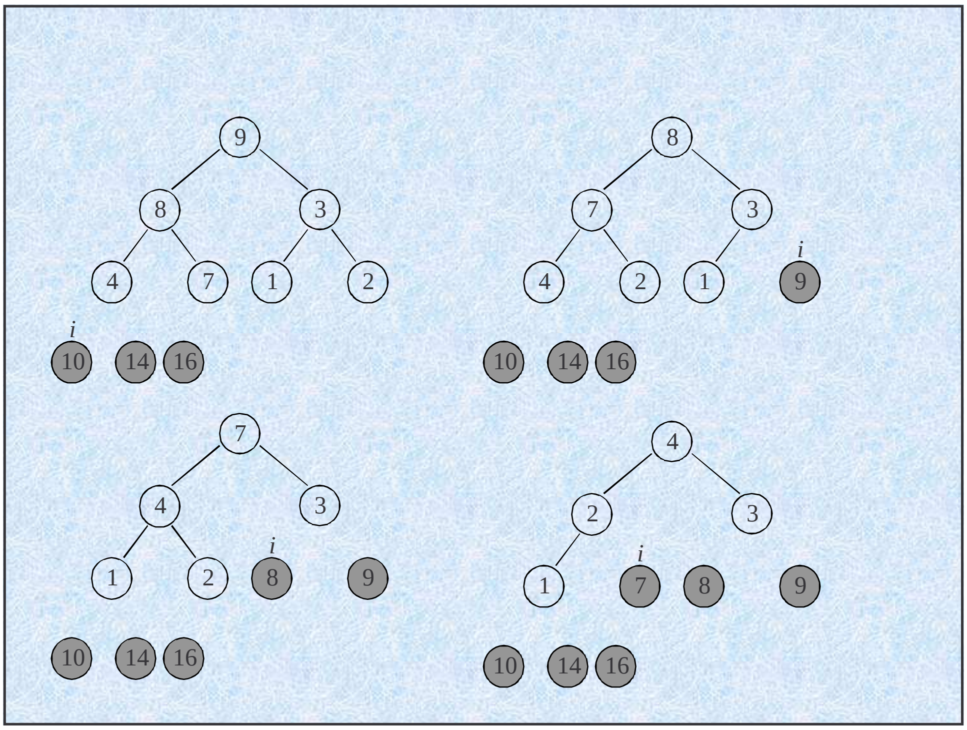


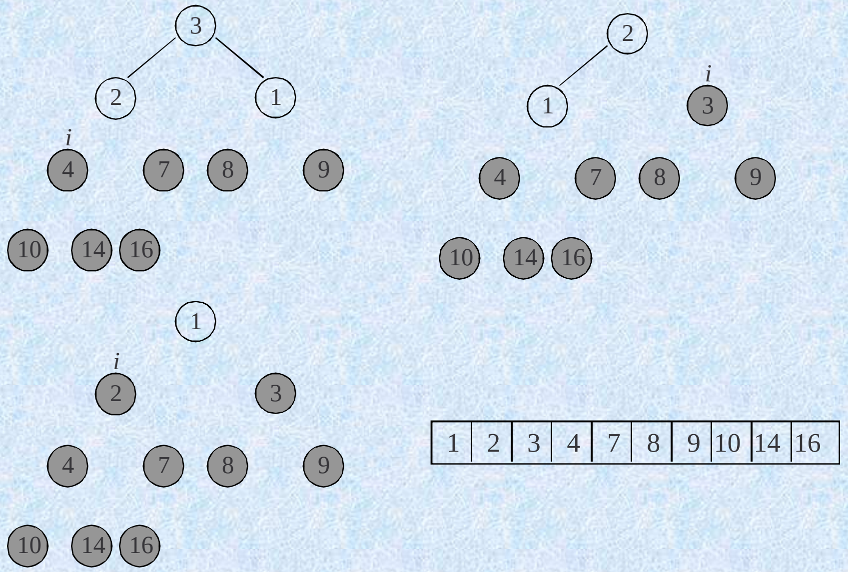
```
void HEAPSORT(int *A, int heapsize)
{
    int i;
    BUILDHEAP(int *A, int heapsize)
    for (i = heapsize; i > 1; i--)
    {
        swap(A[1], A[i]);
        heapsize--;
        HEAPIFY(A, heapsize, 1);
    }
}
```

Exemplo:




16
14
10
8
7
9
3
2
4
1
14
8
10
4
7
9
3
i
2
1
16
10
8
9
4
7
1
3
i
2
14
16
9
8
3
4
7
1
2
i
10
14
16


9
8
3
4
7
1
2
i
10
14
16
8
7
3
i
4
2
1
9
10
14
16
7
4
3
i
1
2
8
9
10
14
16
4
2
3
i
1
7
8
9
10
14
16



1	2	3	4	7	8	9	10	14	16

Eficiência de HEAPSORT

- $T_{HEAPIFY}(n) = O(h)$, onde h é a altura da sub-arvore
- Observe que tempo da HEAPIFY varia com a profundidade do nó na arvore. Observe também que um heap de n elementos tem no máximo $\lceil n/2^{h+1} \rceil$ nós de altura h. O tempo requerido pela HEAPIFY quando chamada sobre um nó de altura h é $O(h)$. Então, o tempo total da BUILDHEAP é

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

- $T_{HEAPSORT}(n) = O(n \log n)$

4 ORDENAÇÃO POR INTERCALAÇÃO (MergeSort)

A idéia principal deste algoritmo é combinar duas listas já ordenadas. O algoritmo quebra um array original em dois outros de tamanhos menores recursivamente até obter arrays de tamanho 1, depois retorna da recursão combinando os resultados.

Algoritmo

```
void MergeSort(int *A, int e, int d)
{
      int q;

      if (e < d)
      {
            q = floor((e+d)/2);
            MergeSort(A, e, q);
            MergeSort(A, q+1, d);
            Merge(A, e, q, d);
      }
}
```

Exemplo MergeSort

[25] [57] [48] [37] [12] [92] [86] [33]

[25 57] [37 48] [12 92] [33 86]

[25 37 48 57] [12 33 86 92]

[12 25 33 37 48 57 86 92]

Algoritmo Merge (intercalação)

Este algoritmo faz fusão de duas listas x[m1],..., x[m2] e x[m2+1],..., x[m3] em array y. As duas sublistas estão em ordens inicialmente.

```
Merge (int x[ ], int m1, int m2, int m3)
{
    int y[ ];
    int apoint, bpoint, cpoint;
    int n1, n2, n3;

    apoint = m1;    bpoint = m2+1;
    for( cpoint = m1; apoint <= m2 && bpoint <= m3; cpoint++)
        if (x[apoint] < x[bpoint])
            y[cpoint] = x[apoint++];
        else
            y[cpoint] = x[bpoint++];
    while (apoint <= m2)
        y[cpoint++] = x[apoint++];
    while (bpoint <= m3)
        y[cpoint++] = x[bpoint++];
}
```

Eficiência de MergeSort

- $T(n) = T(\text{floor}(n/2)) + T(\text{ceiling}(n/2)) + \alpha n = O(n \log n)$
- Exige $O(n)$ espaço adicional para o vetor auxiliar

5 ORDENAÇÃO SEM COMPARAÇÃO

5.1 Ordenação por Contagem (Counting Sort)

• A idéia básica de Counting Sort é determinar, para cada elemento x, o numero de elementos menor que x. Esta informação pode ser usada para colocar o elemento x na posição correta. Por exemplo, se tem 17 elementos menor que x, então, x deve está na posição 17.

• Assume que cada elemento na lista é um numero inteiro entre 1 e k.

• Assume que a entrada é um array A de n elementos. O array B aguarda a saída ordenada e o array C é usada para armazenamento temporário.

```
Counting-Sort (int A[ ], int B[ ], int k, int n)
{
   for (i = 1; i <= k; i++)                                        1)
          C[i] = 0;                                                2)
   for (j = 1; j <= n; j++)                                        3)
          C[A[j]] = C[A[j]] + 1;                                   4)
   /*agora C[i] contem o numero de elementos igual a i. */
   for (i = 2; i <= k; i++)                                        5)
          C[i] = C[i] + C[i-1];                                    6)
   /*agora C[i] contem o numero de elementos menor ou igual a i */
   for (j = n; j >= 1; j--) {                                      7)
          B[C[A[j]]] = A[j];                                       8)
          C[A[j]] = C[A[j]] - 1;                                   9)
   }
}
```

a)

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	2	0	2	3	0	1

b)

	1	2	3	4	5	6
C	2	2	4	7	7	8

c)

	1	2	3	4	5	6	7	8
B							4	

	1	2	3	4	5	6
C	2	2	4	6	7	8

d)

	1	2	3	4	5	6	7	8
B		1					4	

	1	2	3	4	5	6
C	1	2	4	6	7	8

e)

	1	2	3	4	5	6	7	8
B		1				4	4	

	1	2	3	4	5	6
C	1	2	4	5	7	8

f)

A

1	2	3	4	5	6	7	8
	1		3		4	4	

C

1	2	3	4	5	6
1	2	3	5	7	8

g)

B

1	2	3	4	5	6	7	8
1	1		3		4	4	

C

1	2	3	4	5	6
0	2	3	5	7	8

h)

B

1	2	3	4	5	6	7	8
1	1		3	4	4	4	

C

1	2	3	4	5	6
0	2	3	4	7	8

i)

B

1	2	3	4	5	6	7	8
1	1		3	4	4	4	6

C

1	2	3	4	5	6
0	2	3	4	7	7

j)

B

1	2	3	4	5	6	7	8
1	1	3	3	4	4	4	6

C

1	2	3	4	5	6
0	2	2	4	7	7

Eficiência de Counting Sort

- $T_{\text{linha 1-2}}(k) = O(k)$, $T_{\text{linha 3-4}}(n) = O(n)$, $T_{\text{linha 5-6}}(k) = O(k)$ e $T_{\text{linha 7-9}}(n) = O(n)$. Então, $T(n) = O(n+k)$. Quando $k = O(n)$, $T(n) = O(n)$.
- Este algoritmo exige dois arrays auxiliares, um de tamanho *n*, outro de tamanho *k*.

5.2 Ordenação de Raízes (Radix Sort)

- Esta Ordenação baseia-se nos valores dos dígitos nas representações posicionais dos números sendo ordenados.
- Executa as seguintes ações pelo digito menos significativo e terminando com o mais significativo. Pegue cada numero numero na seqüência e posicione-o em uma das dez filas, dependendo do valor do digito sendo processado. Em seguida, restaure cada fila para a seqüência original, começando pela fila de números com um digito 0 e terminando com a fila de números com o digito 9. Quando essas ações tiverem sido executadas para cada digito, a seqüência estará ordenada.

Exemplo:

Lista original: 25 57 48 37 12 92 86 33

Filas baseadas no digito menos significativo

	inicio	final
fila[0]		
fila[1]		
fila[2]	12	92
fila[3]	33	
fila[4]		
fila[5]	25	
fila[6]	86	
fila[7]	57	37
fila[8]	48	
fila[9]		

Exemplo:

Depois da primeira passagem: 12 92 33 25 86 57 37 48

Filas baseadas no digito mais significativo

	inicio	final
fila[0]		
fila[1]	12	
fila[2]	25	
fila[3]	33	37
fila[4]	48	
fila[5]	57	
fila[6]		
fila[7]		
fila[8]	86	
fila[9]	92	

Lista classificada: 12 25 33 37 48 57 86 92

Algoritmo

```
for (k = digito menos significativo; k <= digito mais significativo; k++){
    for (i = 0; i < n; i++){
        y = x[i];
        j = k-ésimo digito de y;
        posiciona y no final da fila[j];
    }
    for (qu = 0; qu < 10; qu++)
        coloca elementos da fila[qu] na próxima posição seqüencial;
}
```

Obs.: os dados originais são armazenados no array *x*

Eficiência de Radix Sort

- Evidentemente, as exigências de tempo para o método de ordenação de raízes dependem da quantidade de dígitos (m) e do numero de elementos na lista (n). Como a repetição mais externa é percorrida m vezes e a repetição mais interna é percorrida n vezes. Então, $T(n) = O(m*n)$.
- Se o numero de dígitos for menor, $T(n) = O(n)$
- Se as chaves forem densas (isto é, se quase todo numero que possa ser uma chave for de fato uma chave), m se aproximara de $\log n$. Neste caso, $T(n) = O(n \log n)$.