CENTRO DE CIÊNCIA E TECNOLOGIA
LABORATÓRIO DE CIÊNCIAS MATEMÁTICAS
UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE

Ordenação por Seleção: Selection Sort HeapSort

Disciplina: Estrutura de Dados II

Prof. Fermín Alfredo Tang Montané

Curso: Ciência da Computação

Princípio de ordenação por Seleção

Descrição

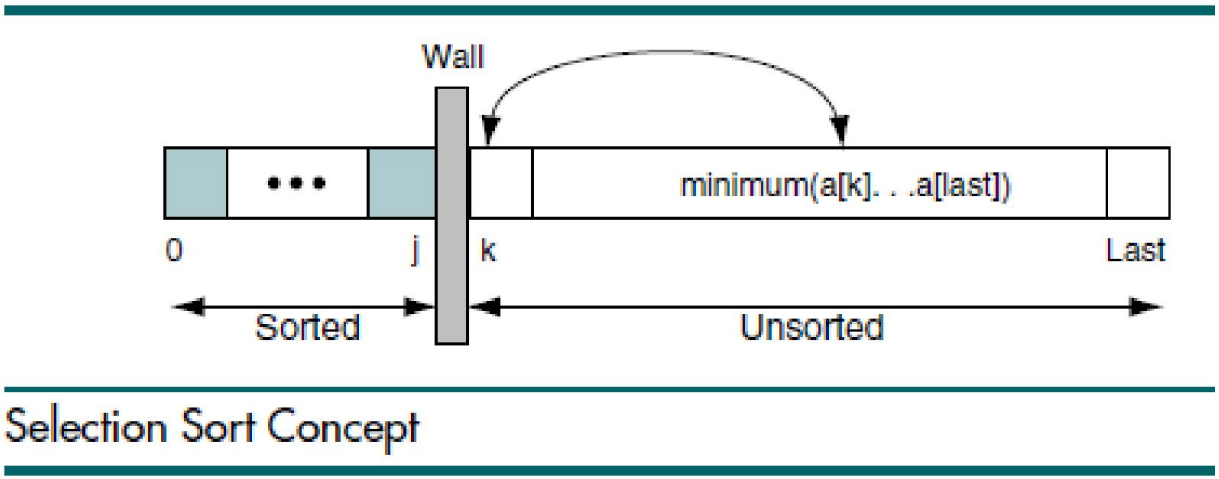
- O princípio de ordenação por seleção é utilizado por alguns métodos de ordenação. Este princípio pode ser considerado como um dos mais intuitivos dentre os diversos princípios adotados nos métodos de ordenação.
- O princípio consiste no seguinte: Dada uma lista de elementos a serem ordenados, simplesmente selecionamos o menor elemento e o colocamos em uma lista ordenada. Este processo é repetido até que todos os elementos estejam ordenados.
- Os métodos que utilizam o princípio de seleção são: o método de ordenação por seleção (*selection sort*) e o método *heapsort*.

Ordenação por Seleção

Descrição

- No método de ordenação por seleção, em qualquer instante, a lista de elementos se encontra dividida em duas: uma **lista ordenada** e outra **lista não-ordenada**, que estão separadas por uma parede (*wall*) imaginária.
- Seleciona-se o **menor elemento** da sublista não-ordenada e troca-se esse elemento com o primeiro da lista não-ordenada. Após a seleção e a troca, a parede entre as duas sublistas se move uma posição, incrementando o número de elementos ordenados e diminuindo o número de elementos não-ordenados.

- Cada vez que um elemento é movido da sublista não-ordenada para a lista ordenada completa-se uma iteração.
- Em uma lista com n elementos, são necessárias, $n-1$ iterações.

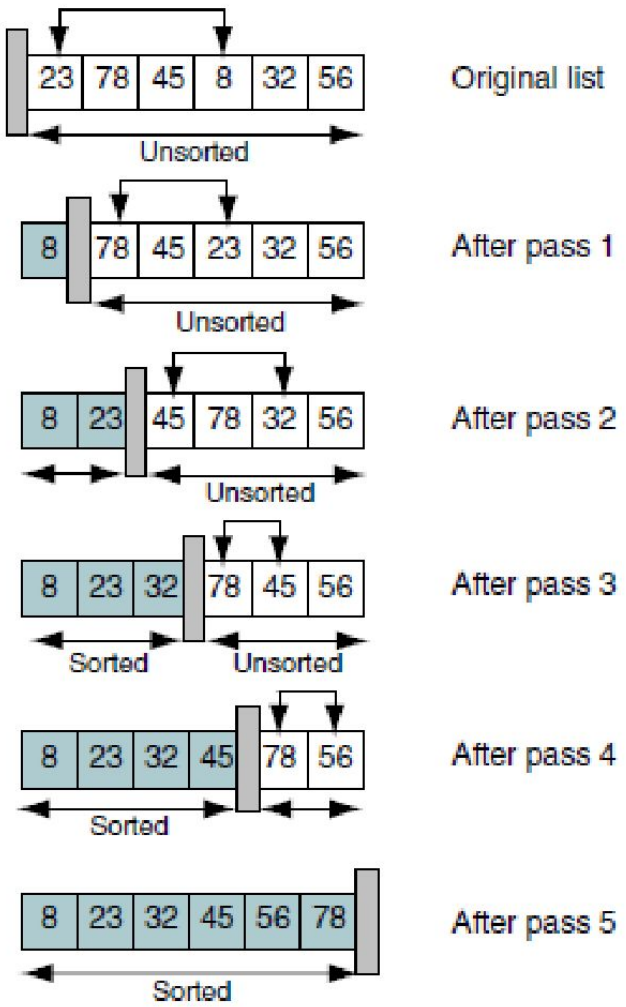


Selection Sort Concept

Ordenação por Seleção

Exemplo

- O exemplo ilustra uma lista com seis elementos que será ordenada pelo método de seleção.
- A lista é ordenada em cinco passos, um a menos que o número de elementos na lista.
- Em cada passo, mostra-se:
  - A seleção e troca do menor elemento;
  - Como se move a parede que divide as listas de elementos ordenados e não- ordenados.



Selection Sort Example

Ordenação por Seleção

Algoritmo

- O algoritmo de ordenação por seleção segue a ideia descrita anteriormente.

```
Algorithm selectionSort (list, last)
Sorts list array by selecting smallest element in
unsorted portion of array and exchanging it with element
at the beginning of the unsorted list.
   Pre  list must contain at least one item
        last contains index to last element in the list
   Post list has been rearranged smallest to largest
 1 set current to 0
 2 loop (until last element sorted)
   1   set smallest to current
   2   set walker   to current + 1
   3   loop (walker <= last)
       1   if (walker key < smallest key)
           1   set smallest to walker
       2   increment walker
   4   end loop
       Smallest selected: exchange with current element.
   5   exchange (current, smallest)
   6   increment current
 3 end loop
end selectionSort
```

Enquanto a parede não chegar no final

Procura o menor elemento na lista não-ordenada

posição da parede

posição do menor elemento

Troca

move a parede

Ordenação por Seleção

Implementação

- A implementação segue quase fielmente o algoritmo descrito.

```
1  /* ================= selectionSort ==================
2     Sorts list [1…last] by selecting smallest element in
3     unsorted portion of array and exchanging it with
4     element at beginning of the unsorted list.
5        Pre  list must contain at least one item
6             last contains index to last list element
7        Post list has been sorted smallest to largest
8  */
9  void selectionSort  (int list[ ], int last)
10 {
11 // Local Declarations
12    int smallest;
13    int holdData;
14
15 // Statements
16    for (int current = 0; current < last; current++)
17        {
18         smallest = current;
19         for (int walker = current + 1;
20                  walker <= last;
21                  walker++)
22             if (list[ walker ] < list[ smallest ])
23                smallest = walker;
24
25         // Smallest selected: exchange with current
26         holdData        = list[ current ];
27         list[current]   = list[ smallest ];
28         list[smallest]  = holdData;
29        } // for current
30    return;
31 } // selectionSort
```

Enquanto a lista ordenada não tiver todos os elementos

Dependendo da versão da linguagen C utilizada. Pode ser necessário declarar algumas variáveis fora dos loops

Procura o menor elemento na lista não-ordenada

Troca o menor elemento com o primeiro da lista não-ordenada

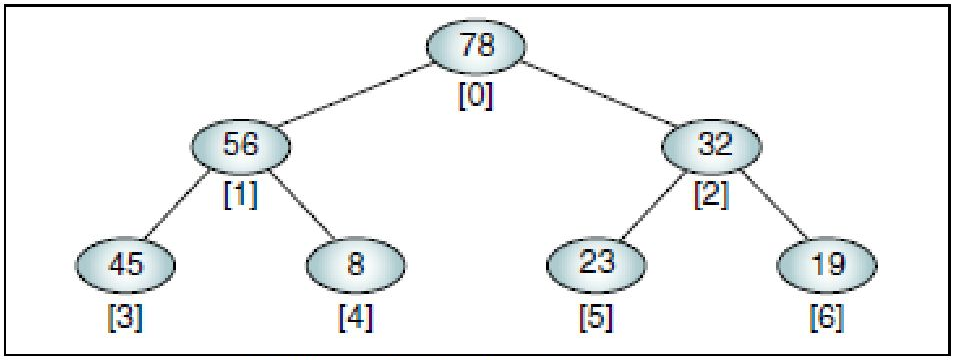
move a parede

posição do menor elemento

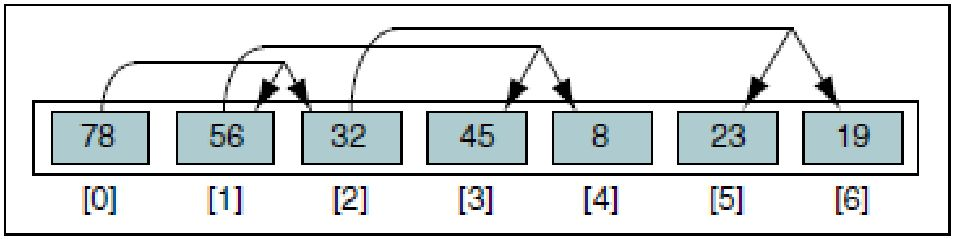
HeapSort
Descrição

- Apresenta-se um método de ordenação com base em *heaps*.
- Na aula sobre *heaps*, definimos um *heap* como uma estrutura de árvore na qual a raiz contém o maior elemento da estrutura. Cada nó é maior que seus filhos.

- Além disso, um *heap* é uma árvore quase completa e por isso, ele pode ser representado usando *arrays*.
- Sendo que os nós filhos de um nó da árvore ocupam posições pré-definidas no *array*.



(a) Heap in its tree form



(b) Heap in its array form

Heap Representations

HeapSort
Descrição

- O algoritmo *heapsort* pode ser visto como uma versão melhorada do algoritmo de ordenação por seleção.
- O algoritmo de ordenação por seleção busca na lista de elementos não-ordenada pelo menor elemento. Encontrar o menor elemento em uma lista de n elementos requer $(n-1)$ comparações. Este processo de busca torna o método lento.
- O algoritmo de ordenação *heapsort*, também seleciona um elemento de uma lista de elementos não-ordenada, seleciona porém, **o maior elemento**.
- Como o *heap* é uma estrutura de árvore, que é uma estrutura hierárquica, não é preciso buscar na lista não-ordenada inteira. O maior elemento já se encontra na raiz da árvore, e poderá ser acessado rápidamente.
- Após remover esse elemento, o *heap* poderá encontrar o segundo maior elemento, percorrendo alguns ramos da árvore e reorganizando árvore mediante um *reheap down*. Esta capacidade de reorganizar a árvore e achar o maior elemento em cada iteração torna este método muito mais rápido que o método de ordenação por seleção.
- O esforço deste processo de reorganização será de O(log n).

HeapSort
Descrição

- O algoritmo *heapsort* começa transformando o array a ser ordenado em um *heap*. Este passo é realizado apenas uma vez. Para isso utiliza-se o algoritmo *reheap up*.
- Após a construção do *heap,* troca-se o elemento raiz, que corresponde ao maior elemento da lista não-ordenada, com o último elemento desta lista. Com isso, o maior elemento será adicionado ao inicio da lista ordenada. A lista ordenada aumentará em um elemento e a lista não-ordenada (ou *heap*) diminuirá em um elemento.
- O elementos restantes no *heap* são reordenados usando o algoritmo *reheap down* para reconstruir o *heap*, e escolher e trocar o segundo maior elemento.
- Assim, em cada iteração, troca-se o maior elemento (elemento raiz) com o último da lista não-ordenada, diminui-se o tamanho do *heap* e aplica-se o algoritmo *reheap down*.

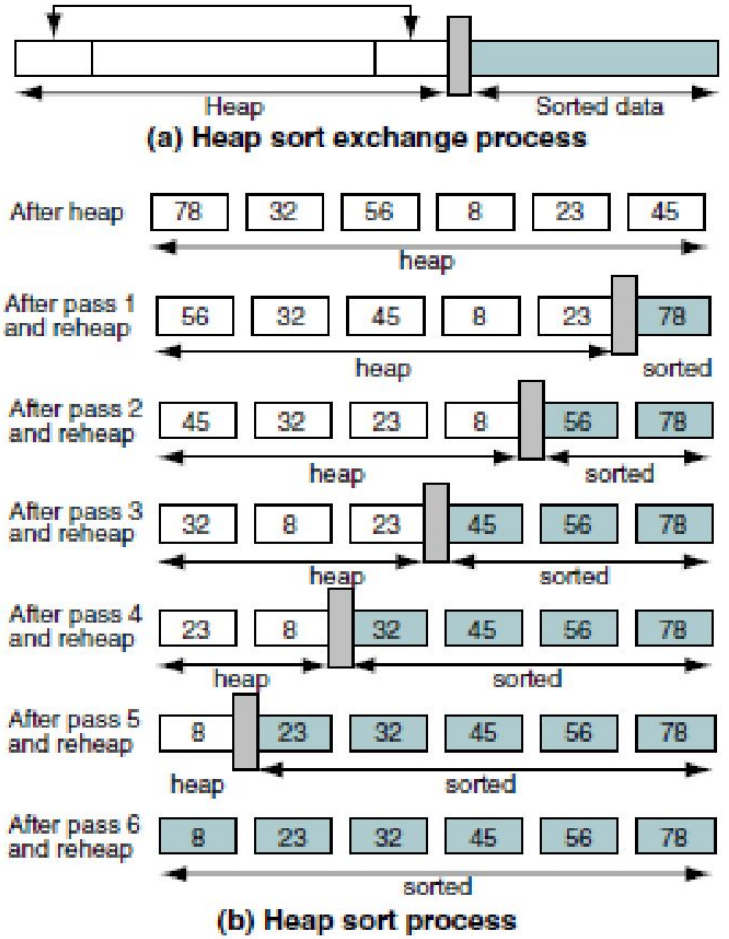
HeapSort
Exemplo

- O exemplo ilustra o passos do algoritmo *heapsort* em um *array* com seis elementos.
- Em cada iteração, o elementos que fazem parte do *heap*, são elementos não-ordenados. Já, os elementos do array além do *heap* são elementos ordenados.

Heap inicial

Troca da raiz com o último do *heap*. *Reheap down*.

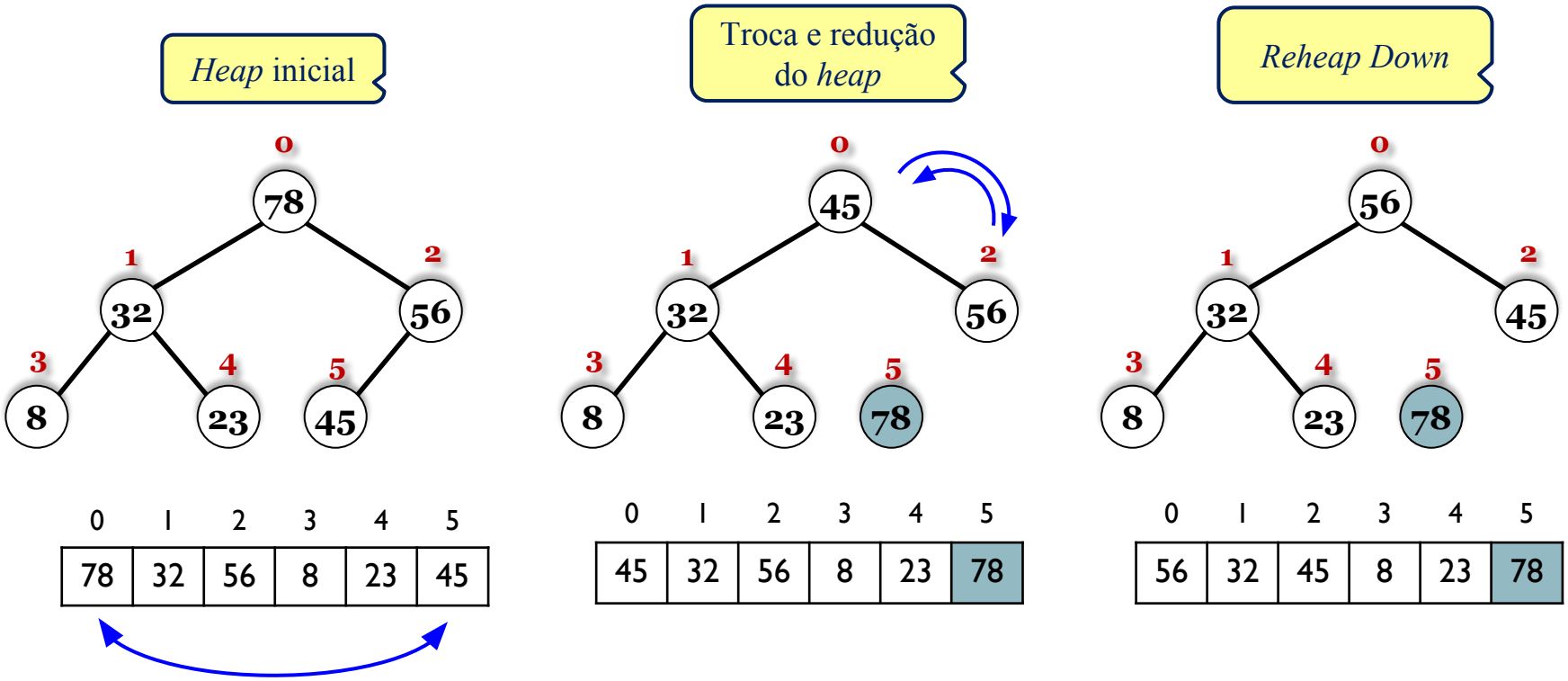
- Observe que o tamanho do *heap* vai diminuindo e a lista de elementos ordenados vai aumentando.



Heap Sort Process

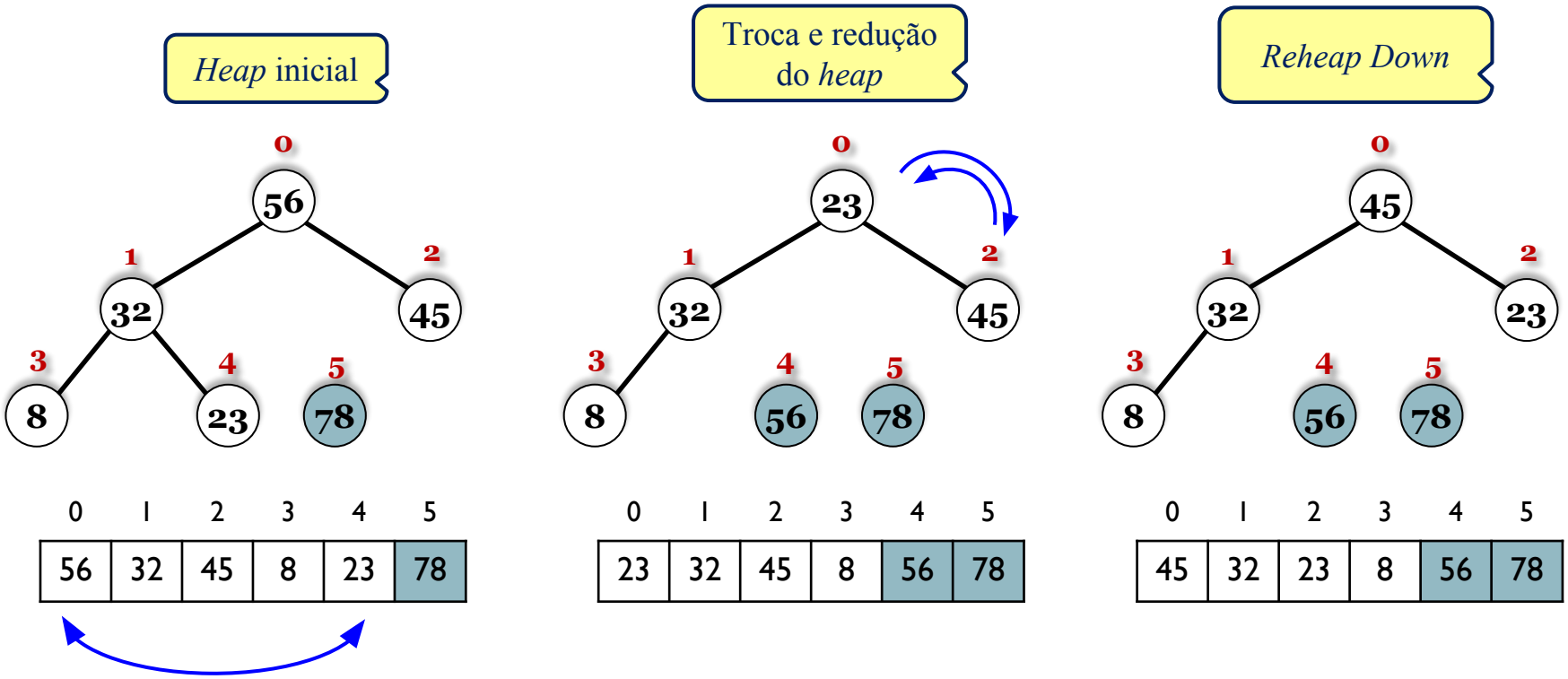
HeapSort
Exemplo

- Detalhamento de cada iteração, incluindo a representação da árvore, a troca da raiz e o último elemento do *heap*, a redução de tamanho do *heap* e a reconstrução do *heap* mediante o *reheap down*.



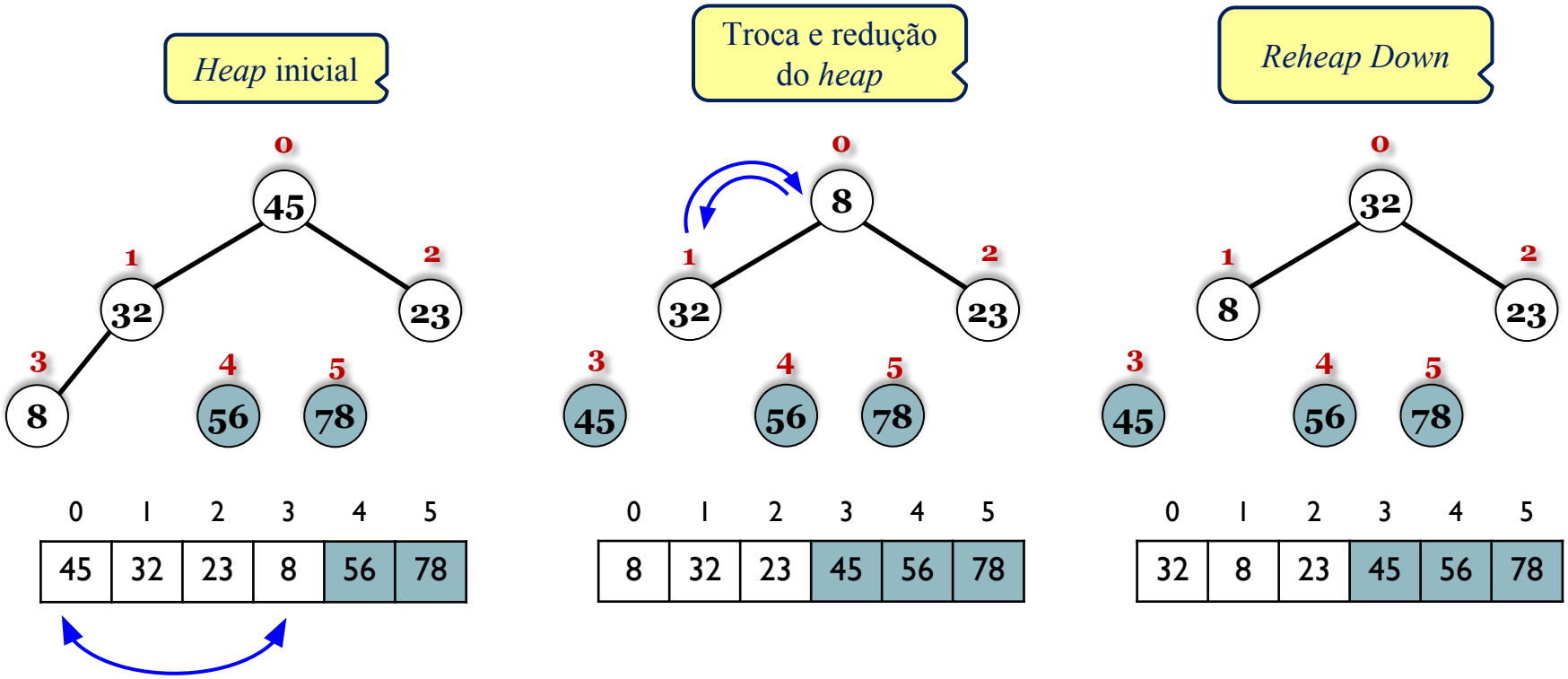
HeapSort
Exemplo

- Detalhamento de cada iteração, incluindo a representação da árvore, a troca da raiz e o último elemento do *heap*, a redução de tamanho do *heap* e a reconstrução do *heap* mediante o *reheap down*.



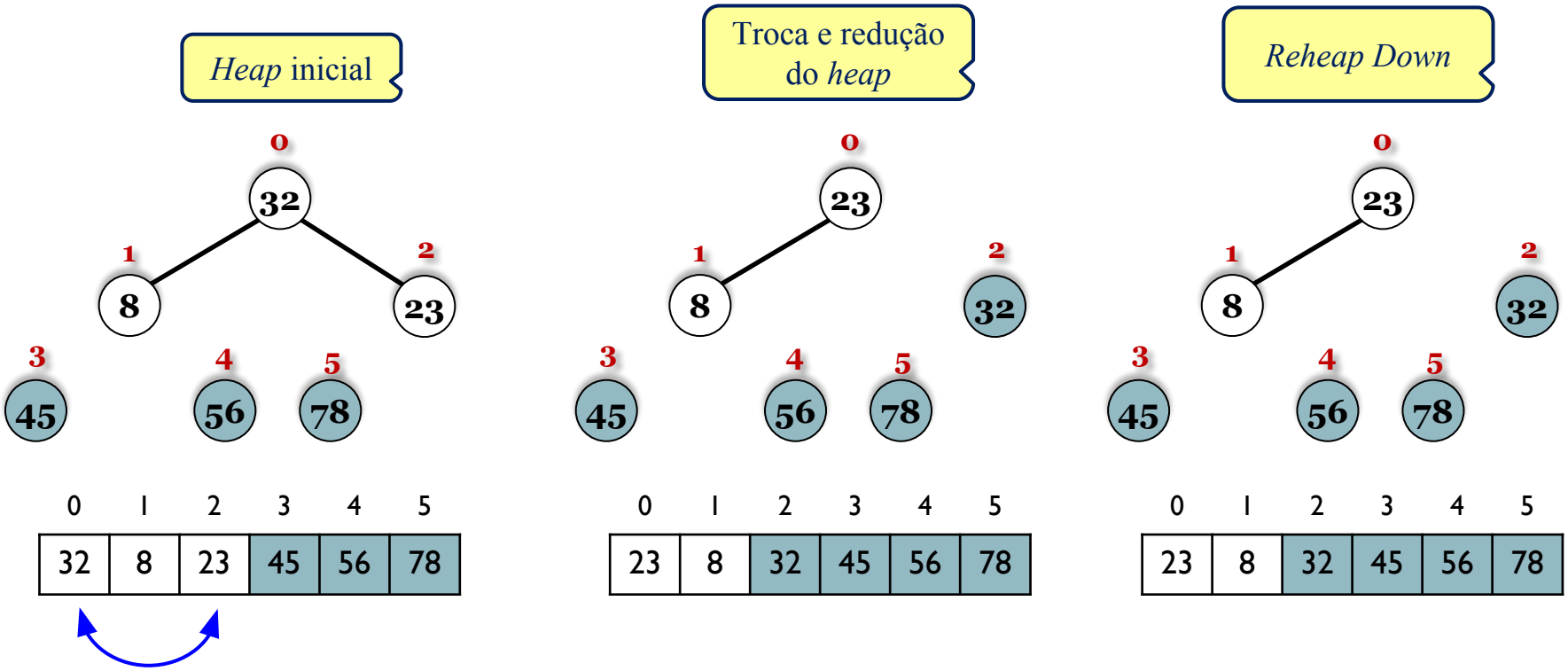
HeapSort
Exemplo

- Detalhamento de cada iteração, incluindo a representação da árvore, a troca da raiz e o último elemento do *heap*, a redução de tamanho do *heap* e a reconstrução do *heap* mediante o *reheap down*.



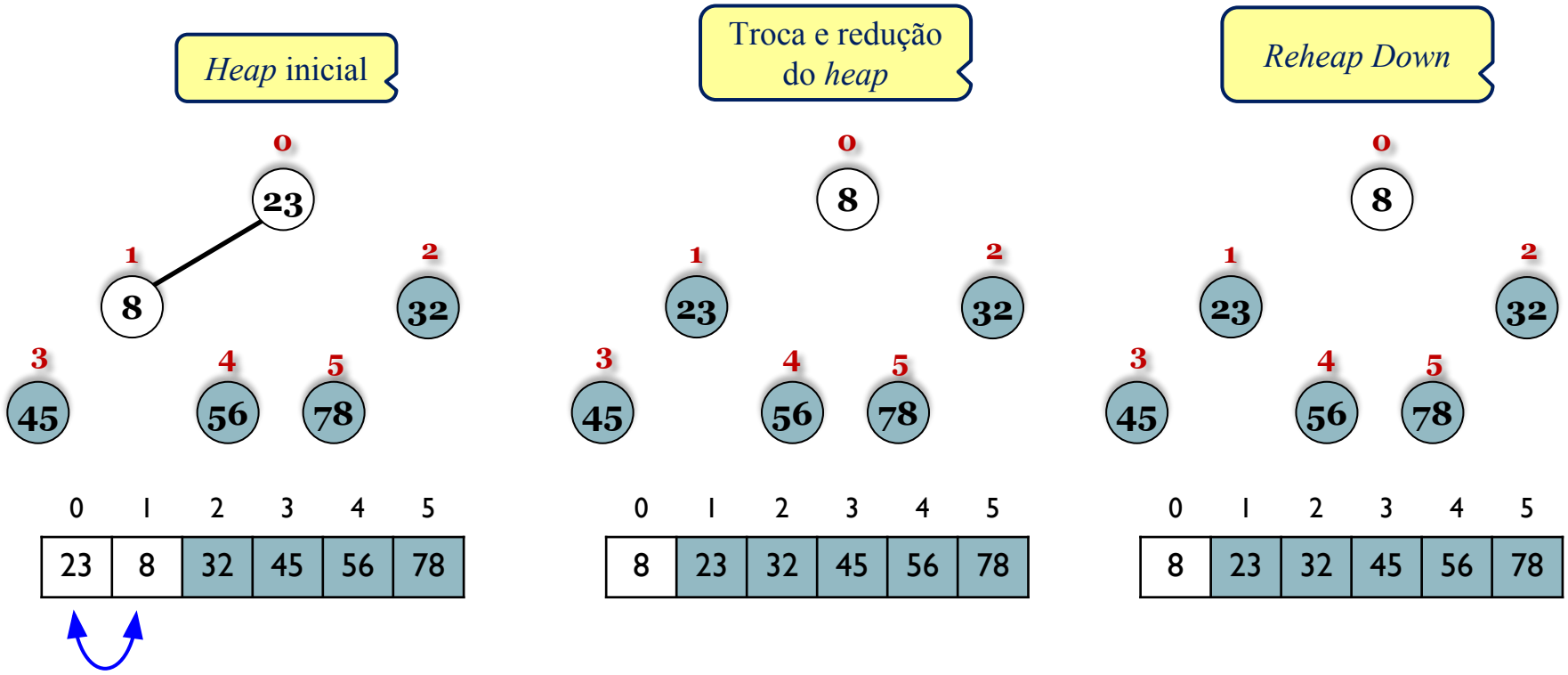
HeapSort
Exemplo

- Detalhamento de cada iteração, incluindo a representação da árvore, a troca da raiz e o último elemento do *heap*, a redução de tamanho do *heap* e a reconstrução do *heap* mediante o *reheap down*.



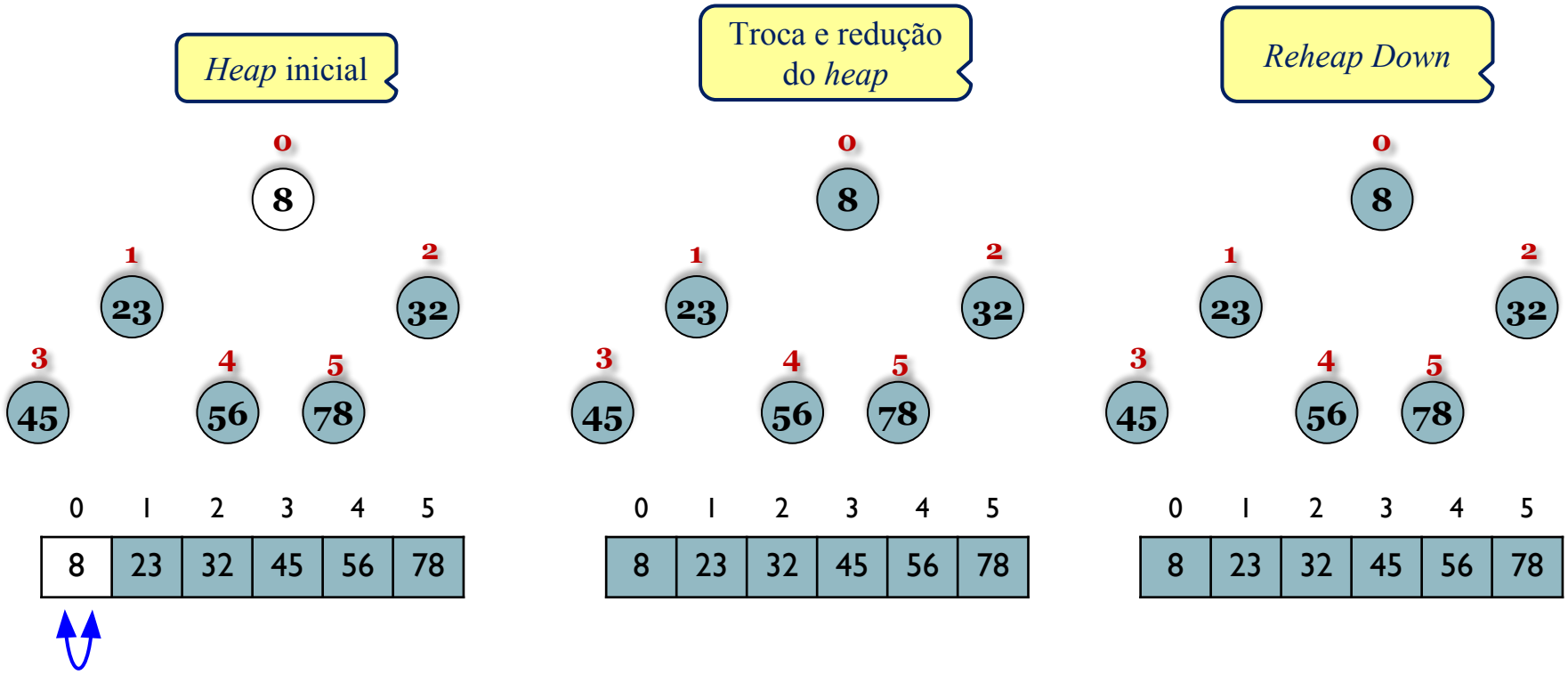
HeapSort
Exemplo

- Detalhamento de cada iteração, incluindo a representação da árvore, a troca da raiz e o último elemento do *heap*, a redução de tamanho do *heap* e a reconstrução do *heap* mediante o *reheap down*.



HeapSort
Exemplo

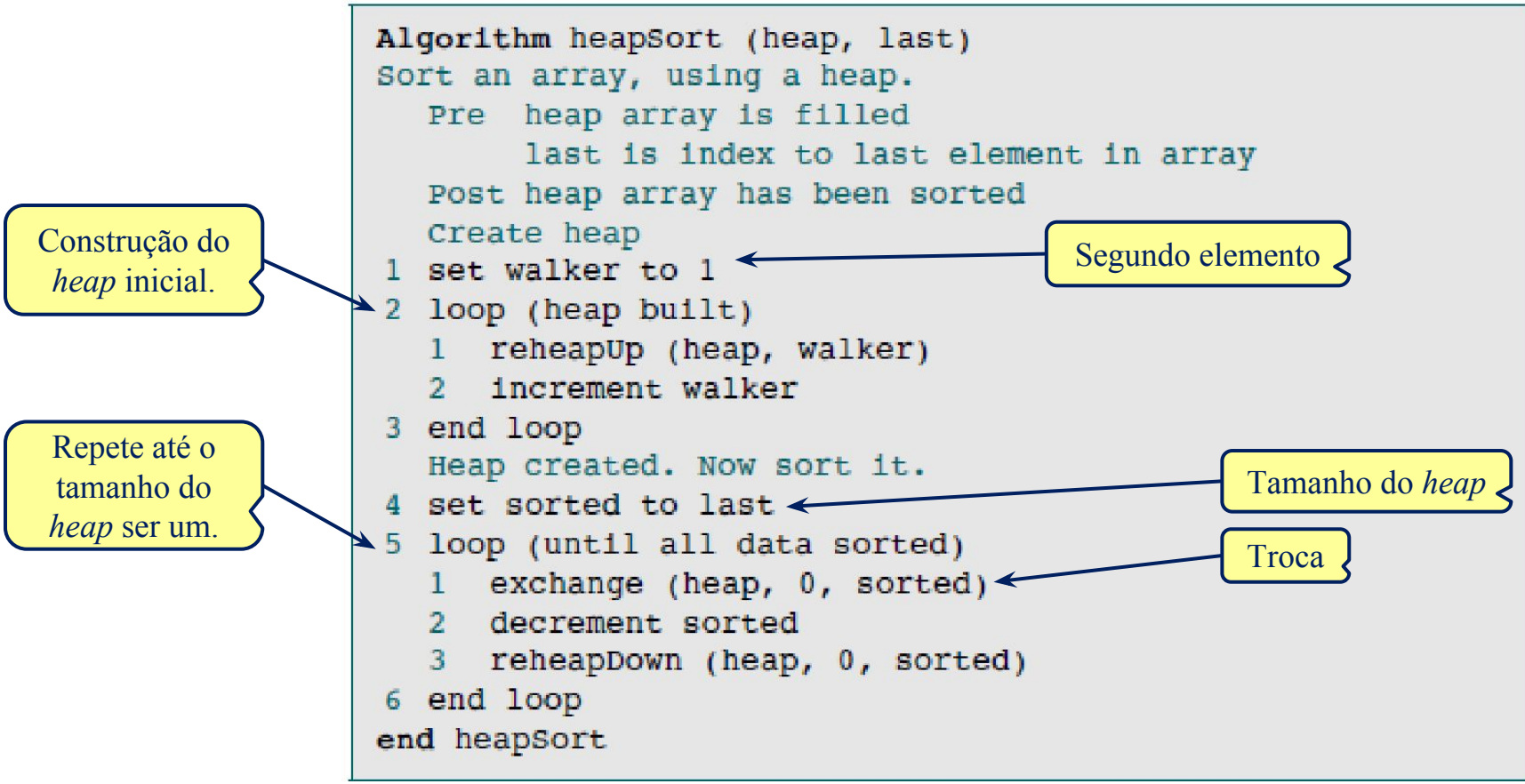
- Detalhamento de cada iteração, incluindo a representação da árvore, a troca da raiz e o último elemento do *heap*, a redução de tamanho do *heap* e a reconstrução do *heap* mediante o *reheap down*.



HeapSort

Algoritmo *Heapsort*

- O algoritmo segue a ideia apresentada no exemplo.




```
Algorithm heapSort (heap, last)
Sort an array, using a heap.
   Pre  heap array is filled
        last is index to last element in array
   Post heap array has been sorted
   Create heap
 1 set walker to 1
 2 loop (heap built)
   1   reheapUp (heap, walker)
   2   increment walker
 3 end loop
   Heap created. Now sort it.
 4 set sorted to last
 5 loop (until all data sorted)
   1   exchange (heap, 0, sorted)
   2   decrement sorted
   3   reheapDown (heap, 0, sorted)
 6 end loop
end heapSort
```

HeapSort

Implementação – Algoritmo *Heapsort*

- A implementação segue fielmente o algoritmo descrito anteriormente.

```
1  /* ==================== heapSort ====================
2     Sort an array, [list0 .. last], using a heap.
3        Pre  list must contain at least one item
4             last contains index to last element in list
5        Post list has been sorted smallest to largest
6  */
7  void heapSort (int  list[ ], int  last)
8  {
9  // Local Definitions
10    int sorted;
11    int holdData;
12
13 // Statements
14    // Create Heap
15    for (int walker = 1; walker <= last; walker++)
16       reheapUp (list, walker);
17
18    // Heap created. Now sort it.
19    sorted = last;
20    while (sorted > 0)
21       {
22        holdData     = list[0];
23        list[0]      = list[sorted];
24        list[sorted] = holdData;
25        sorted--;
26        reheapDown (list, 0, sorted);
27       } // while
28    return;
29 }  // heapSort
30
```

Construção do *heap* inicial.

Repete até o tamanho do *heap* ser um.

Tamanho do *heap*

Troca

Reconstrução do *heap*

HeapSort

Implementação – *Reheap Up*

- A implementação do algoritmo *reheap up*.

```
31 /* ==================== reheapUp ====================
32    Reestablishes heap by moving data in child up to
33    correct location heap array.
34       Pre  heap is array containing an invalid heap
35            newNode is index location to new data in heap
36       Post newNode has been inserted into heap
37 */
38 void reheapUp (int* heap, int newNode)
39 {
40 // Local Declarations
41    int parent;
42    int hold;
43
44 // Statements
45    // if not at root of heap
46    if (newNode)
47       {
48        parent = (newNode - 1)/ 2;
49        if ( heap[newNode] > heap[parent] )
50           {
51            // child is greater than parent
52            hold          = heap[parent];
53            heap[parent]  = heap[newNode];
54            heap[newNode] = hold;
55            reheapUp (heap, parent);
56           } // if heap[]
57       } // if newNode
58    return;
59 }  // reheapUp
60
```

Se não é raiz

Nó pai

Troca

Chamada recursiva

HeapSort

Implementação – *Reheap Down* (Parte1)

- A implementação do algoritmo *reheap down*.

```
61  /* ==================== reheapDown ====================
62     Reestablishes heap by moving data in root down to its
63     correct location in the heap.
64        Pre  heap is an array of data
65             root is root of heap or subheap
66             last is an index to the last element in heap
67        Post heap has been restored
68  */
69  void reheapDown   (int* heap, int root, int last)
70  {
71  // Local Declarations
72     int hold;
73     int leftKey;
74     int rightKey;
75     int largeChildKey;
76     int largeChildIndex;
77
```

HeapSort

Implementação – *Reheap Down* (Parte2)

- A implementação do algoritmo *reheap down*.

```
78  // Statements
79     if ((root * 2 + 1) <= last)
80         // There is at least one child
81        {
82         leftKey   = heap[root * 2 + 1];
83         if ((root * 2 + 2) <= last)
84            rightKey  = heap[root * 2 + 2];
85         else
86            rightKey  = -1;
87
88         // Determine which child is larger
89         if (leftKey > rightKey)
90            {
91             largeChildKey   = leftKey;
92             largeChildIndex = root * 2 + 1;
93            } // if leftKey
94         else
95            {
96             largeChildKey   = rightKey;
97             largeChildIndex = root * 2 + 2;
98            } // else
99         // Test if root > larger subtree
100        if (heap[root] < heap[largeChildIndex])
101            {
102             // parent < children
103             hold       = heap[root];
104             heap[root] = heap[largeChildIndex];
105             heap[largeChildIndex] = hold;
106             reheapDown (heap, largeChildIndex, last);
107            } // if root <
108    } // if root
109    return;
110 } // reheapDown
```

Determina o maior filho

Troca

Chamada recursiva

Ordenação por Seleção vs *HeapSort*

Comparação

- A tabela compara o número passos realizados pelos algoritmos de ordenação estudados, desde a perspectiva do número de iterações.

n	Number of loops	
	Straight selection	Heap
25	625	116
100	10,000	664
500	250,000	4482
1000	1,000,000	9965
2000	4,000,000	10,965

Referências

- Gilberg, R.F. e Forouzan, B. A. Data Structures_A Pseudocode Approach with C. Capítulo 12. Sorting. Segunda Edição. Editora Cengage, Thomson Learning, 2005.