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

Ordenações baseadas em trocas: Bubblesort Quicksort

Disciplina: Estrutura de Dados II

Prof. Fermín Alfredo Tang Montané

Curso: Ciência da Computação

Ordenações baseadas em Trocas

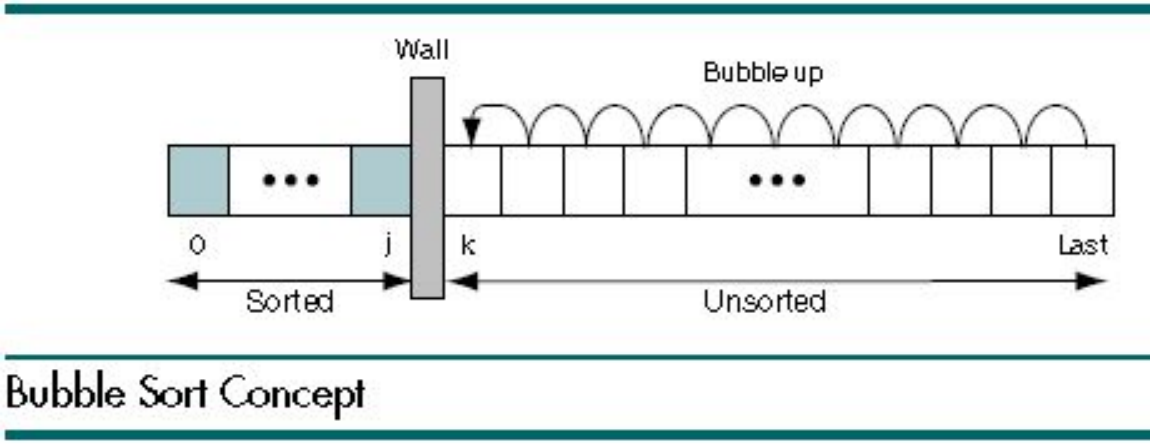
- Na categoria de algoritmos de ordenação baseadas em trocas (***exchange sorts***), colocamos o algoritmo mais conhecido e ensinado na ciência de computação, o algoritmo de bolha, *Bubblesort,* e o algoritmo de propósito geral mais eficiente, *Quicksort*.
- Nesta classe de algoritmo, os elementos que encontram fora de ordem são trocados entre si até que a lista esteja ordenada.
- Embora quase todos os algoritmos de ordenação utilizem algum tipo de troca, os algoritmos desta categoria, utilizam trocas de maneira mais ampla.

O algoritmo *Bubblesort*

Descrição

- No algoritmo *Bubblesort*, considera-se que a lista de elementos se encontra dividida em duas partes. Uma sublista ordenada e outra não ordenada. Uma parede imaginária (*wall*) divide ambas listas.
- Em cada passo, o menor elemento da lista não ordenada é movido para a lista ordenada, mediante movimentos de bolha, que na verdade consistem na **troca de elementos**. Para seguir a convenção podemos dizer que o menor elemento é borbulhado para frente (*Bubble up*).
- Após mover o menor elemento para a lista ordenada, a parede se move uma posição à direita incrementando o número de elementos ordenados e diminuindo o número de elementos não ordenados.

- Um passo ou iteração do algoritmo é completado quando um elemento é movido da lista não ordenada para a lista ordenada.

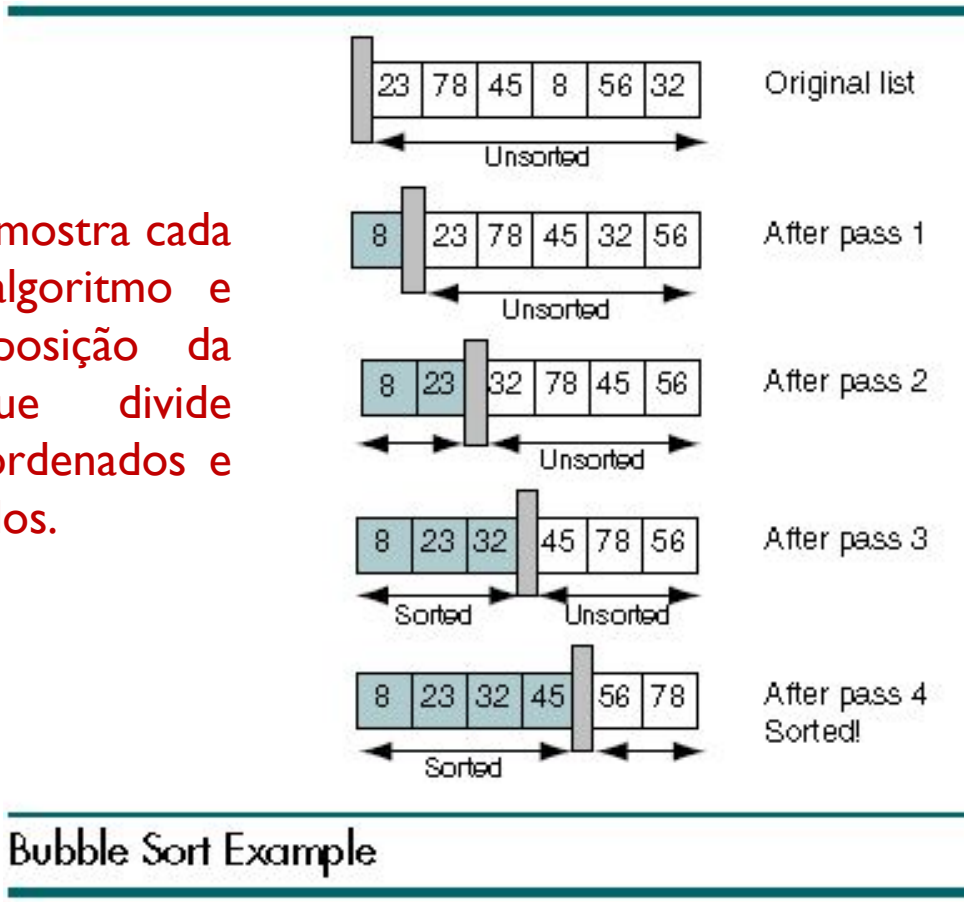


Bubble Sort Concept

O algoritmo *Bubblesort*

Exemplo

- Dada uma lista de n elementos, o bubblesort requert $(n-1)$ passos completar a ordenação.



- O exemplo mostra cada passo do algoritmo e ilustra a posição da parede que divide elementos ordenados e não ordenados.

No primeiro passo, começa-se pelo elemento 32, que é comparado com 56. Como 32 é menor que 56, os elementos são trocados e descemos uma posição. Agora compara-se 32 e 8. Como 32 não é menor que 8, os elementos são trocados. Descemos outra posição, comparamos 8 e 45, como estão fora de ordem trocamos e descemos mais uma posição. Agora 8 é comparado com 78 e ambos elementos trocados. Finalmente, 8 é comparado com 23 e trocados.

O algoritmo *Bubblesort*

Algoritmo

- O algoritmo de ordenação é bastante simples. Em cada passo, o menor elemento é borbulhado para começo da lista não ordenada.
- Neste processo, elementos adjacentes que se encontrem fora de ordem são trocados, permitindo a ordenação parcial dos dados.
- Ao finalizar cada passo, a lista ordenada cresce em um elemento enquanto a lista não ordenada diminui em um elemento.
- O algoritmo continua realizando um novo passo até que a lista não ordenada fique sem elementos.

O algoritmo *Bubblesort*

Algoritmo

- O algoritmo é descrito a seguir.

```
Algorithm  bubbleSort (list, last)
Sort an array using bubble sort. Adjacent
elements are compared and exchanged until list is
completely ordered.
   Pre  list must contain at least one item
        last contains index to last element in the list
   Post list has been rearranged in sequence low to high
1  set current to 0
2  set sorted  to false
3  loop (current <= last AND sorted false)
      Each iteration is one sort pass.
   1  set walker to last
   2  set sorted to true
   3  loop (walker > current)
      1  if (walker data < walker - 1 data)
            Any exchange means list is not sorted.
         1  set sorted to false
         2  exchange (list, walker, walker - 1)
      2  end if
      3  decrement walker
   4  end loop
   5  increment current
4  end loop
end bubbleSort
```

Enquanto a parede não chegar no final

posição do elemento a ser comparado

Enquanto walker maior que a parede

Se dados fora de ordem

posição da parede

Não está ordenado

Troca

decresce walker

move a parede

O algoritmo *Bubblesort*

Algoritmo

- A versão apresentada do algoritmo *bubblesort,* apresenta algumas diferenças com relação a outras versões.
- Primeiro, ele possui uma ligeira melhoria. Se nenhuma troca é realizada em um certo passo, os elementos já estão na ordem certa, ou seja, ordenados e o algoritmo termina.
- Segundo, as comparações e trocas acontecem de trás para frente, para manter uma coerência com os outros algoritmos básicos de inserção e seleção, no sentido de que todos eles funcionem da mesma maneira. Historicamente, o algoritmo *bubblesort* original foi concebido para realizar as comparações e trocas da frente para trás, colocando o maior elemento no fim da lista.

Bubblesort

Implementação

- A implementação segue quase fielmente o algoritmo descrito.

Dependendo da versão da linguagen C utilizada. Pode ser necessário definir e/ou inicializar algumas variáveis fora dos loops.

```
1  /* ================== bubbleSort ====================
2     Sort list using bubble sort. Adjacent elements are
3     compared and exchanged until list is ordered.
4        Pre  list must contain at least one item
5             last contains index to last list element
6        Post list has been sorted in ascending sequence
7  */
8  void bubbleSort (int list [], int last)
9  {
10 // Local Definitions
11    int temp;
12
13 // Statements
14    // Each iteration is one sort pass
15    for (int current = 0, sorted = 0;
16             current <= last && !sorted;
17             current++)
18          for (int walker = last, sorted = 1;
19                walker > current;
20                walker--)
21           if (list[ walker ]  < list[ walker - 1 ])
22              // Any exchange means list is not sorted
23               {
24                sorted = 0;
25                temp              = list[walker];
26                list[walker]      = list[walker - 1];
27                list[walker - 1] = temp;
28               } // if
29    return;
30 } // bubbleSort
```

Quicksort
Descrição

- No algoritmo *bubblesort,* elementos consecutivos são comparados e em alguns casos trocados em cada passo do algoritmo, o que significa que muitas trocas podem ser necessárias para mover um elemento até a sua posição correta.
- O algoritmo *quicksort,* foi desenvolvido por C.A.R. Hoare em 1962, como um algoritmo que realiza trocas entre elementos que se encontram geralmente distantes, e que realiza um número muito menor de trocas para posicionar os elementos na sua posição certa.

Quicksort
Algoritmo

- Em cada iteração, o algoritmo *quicksort* seleciona um elemento, conhecido como **pivô**, e reorganiza a lista de elementos em três partes:
  - i) os elementos com valor menor que o pivô (partição esquerda);
  - ii) o próprio elemento pivô;
  - iii) os elementos com valor maior ou igual que o pivô (partição direita).
- De maneira que o elemento pivô seja posicionado na posição correta dentro da lista de elementos. Os elementos menores sejam colocados em posições inferiores à posição do pivô (a esquerda dele) e os elementos maiores ou iguais sejam colocados em posições superiores à posição do pivô (a direita dele).
- A ordenação continua aplicando o mesmo procedimento na partição esquerda e depois na partição direita. Sendo que este algoritmo pode ser implementado de maneira recursiva.

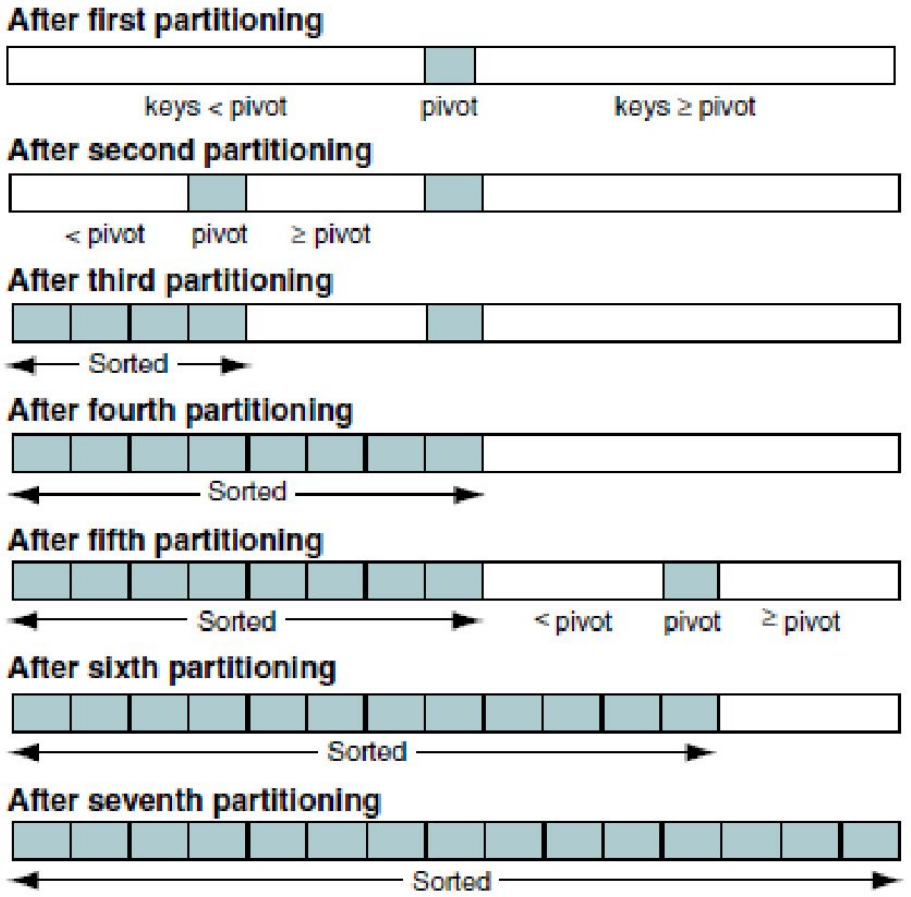
Quicksort
Processamento das partições

- A figura ilustra a ordem de processamento das partições geradas pelo algoritmo *Quicksort*.

O primeiro particionamento gera uma partição esquerda e outra direita. E posiciona o primeiro pivô na posição certa.

O segundo particionamento divide a partição esquerda e posiciona o segundo pivô.

Após o terceiro e quarto particionamentos, a primeira partição esquerda fica ordenada. O trabalho se concentra na primeira partição direita.



Quick Sort Partitions

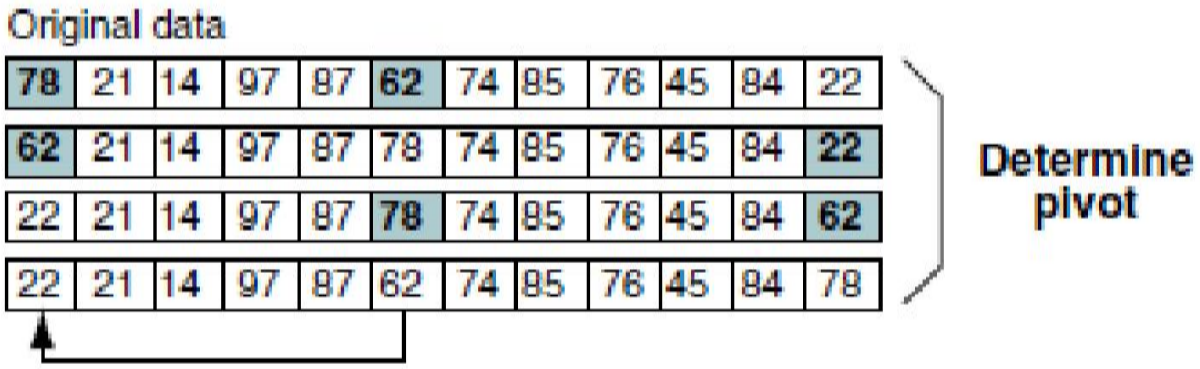
Quicksort
Variantes

- O algoritmo original de Hoare escolhia como elemento pivô o primeiro elemento na lista (na esquerda da lista).
- Em 1969, R.C. Singleton introduziu uma melhoria propondo calcular o pivô como o elemento médio de três elementos: o elemento na esquerda, na direita e no meio da lista. Uma vez que este elemento era identificado este seria trocado com primeiro elemento na lista.
- Uma segunda melhoria foi proposta por Knuth, que sugeriu que quando a partição fosse suficientemente pequena seria muito melhor utilizar o algoritmo de ordenação por inserção para ordenar esta partição. O tamanho sugerido para a partição foi de 16.

Quicksort

Exemplo – Calculo do Pivô

- Para calcular o pivô, identifica-se os elementos na esquerda, na direita e no meio da lista (o primeiro, o último e o meio).
- Depois são realizadas três comparações e três possíveis trocas de maneira a ordenar os três elementos e localizar **o elemento do meio** que será o pivô.
- Finalmente, o pivô é colocado na primeira posição (na esquerda da lista).



Compara-se o primeiro elemento e do meio. Eles são trocados para ficarem em ordem.

Compara-se o primeiro elemento e o último. Eles são trocados para ficarem em ordem.

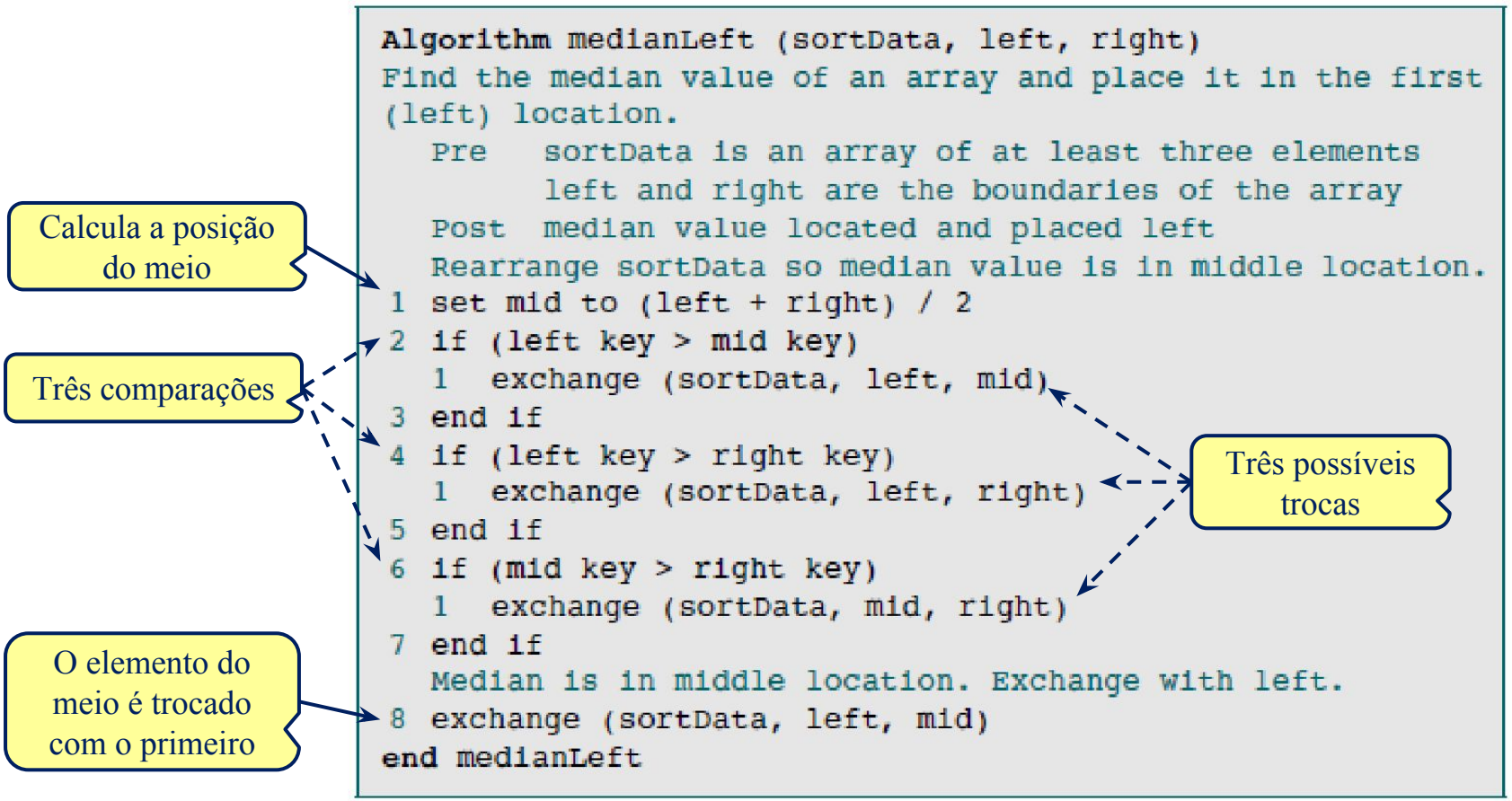
Compara-se o elemento do meio e o último. Eles são trocados para ficarem em ordem.

Com os três elementos ordenados, escolhe-se o elemento do meio como pivô. Trocado como elemento da primeira posição.

Quicksort

Algoritmo – Calculo do Pivô

- O algoritmo segue a ideia apresentada no exemplo.




```
Algorithm medianLeft (sortData, left, right)
Find the median value of an array and place it in the first
(left) location.
   Pre   sortData is an array of at least three elements
         left and right are the boundaries of the array
   Post  median value located and placed left
   Rearrange sortData so median value is in middle location.
 1 set mid to (left + right) / 2
 2 if (left key > mid key)
   1  exchange (sortData, left, mid)
 3 end if
 4 if (left key > right key)
   1  exchange (sortData, left, right)
 5 end if
 6 if (mid key > right key)
   1  exchange (sortData, mid, right)
 7 end if
   Median is in middle location. Exchange with left.
 8 exchange (sortData, left, mid)
end medianLeft
```

Quicksort

Implementação – Calculo do Pivô (Parte1)

- A implementação segue fielmente o algoritmo descrito anteriormente.
- Três comparações, três possíveis trocas para ordenar os três elementos.
- Uma troca final para localizar o pivô a primeira posição.

```
1  /* ================== medianLeft ==================
2     Find the median value of an array,
3     sortData[left..right], and place it in the
4     location sortData[left].
5        Pre  sortData is array of at least three elements
6             left and right are boundaries of array
7        Post median value placed at sortData[left]
8  */
9  void medianLeft (int sortData[], int left, int right)
10 {
11 // Local Definitions
12    int mid;
13    int hold;
14
15 // Statements
16    // Rearrange sortData so median is middle location
17    mid = (left + right) / 2;
18    if (sortData[left] > sortData[mid])
19       {
20        hold            = sortData[left];
21        sortData[left]  = sortData[mid];
22        sortData[mid]   = hold;
23       } // if
24    if (sortData[left] > sortData[right])
25       {
26        hold            = sortData[left];
27        sortData[left]  = sortData[right];
28        sortData[right] = hold;
29       } // if
```

Quicksort

Implementação – Calculo do Pivô (Parte2)

- (Continuação...)

```
30    if (sortData[mid]  > sortData[right])
31       {
32        hold            = sortData[mid];
33        sortData[mid]   = sortData[right];
34        sortData[right] = hold;
35       } // if
36
37    // Median is in middle. Exchange with left
38    hold           = sortData[left];
39    sortData[left] = sortData[mid];
40    sortData[mid]  = hold;
41
42    return;
43 }  // medianLeft
```

Quicksort

Exemplo – Ordenação de uma partição

- O exemplo ilustra o processo de reordenação da lista de elementos com base no pivô. Este processo divide a lista de elementos em três partes.

O algoritmo percorre a lista de elementos em duas direções:

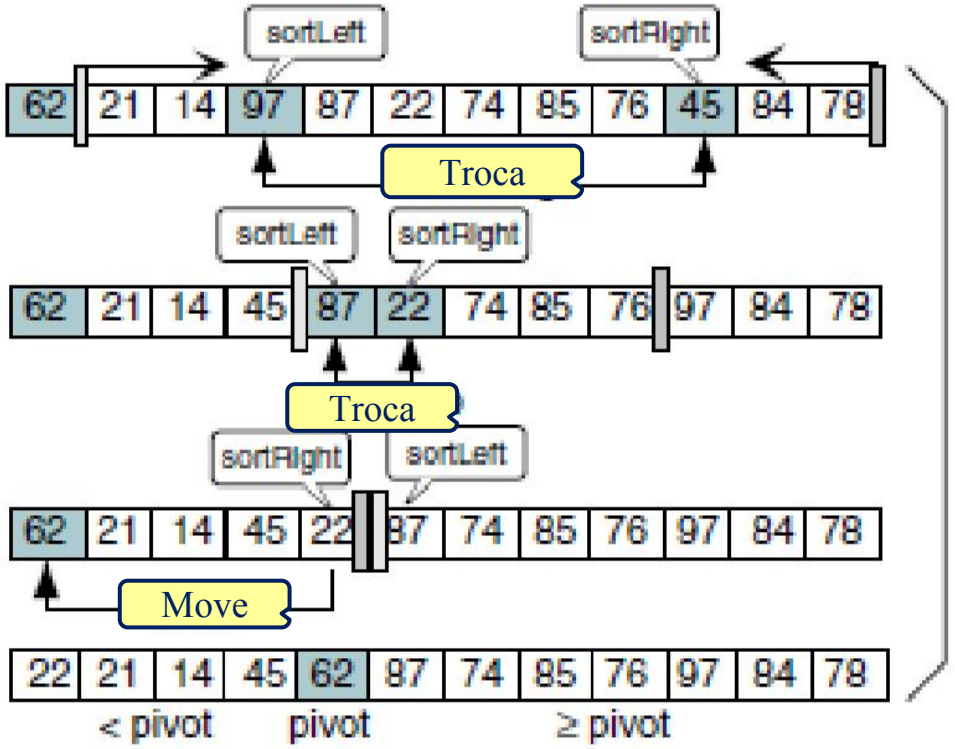
i) De esquerda para direita;

ii) De direita para esquerda.

O primeiro percurso aceita elementos menores que o pivô e termina assim que achar um elemento maior.

O segundo percurso aceita elementos maiores ou iguais que o pivô e termina assim que achar um elemento menor.

Ao concluir os dois percursos as posições de parada indicam dois elementos fora de ordem que são então trocados.

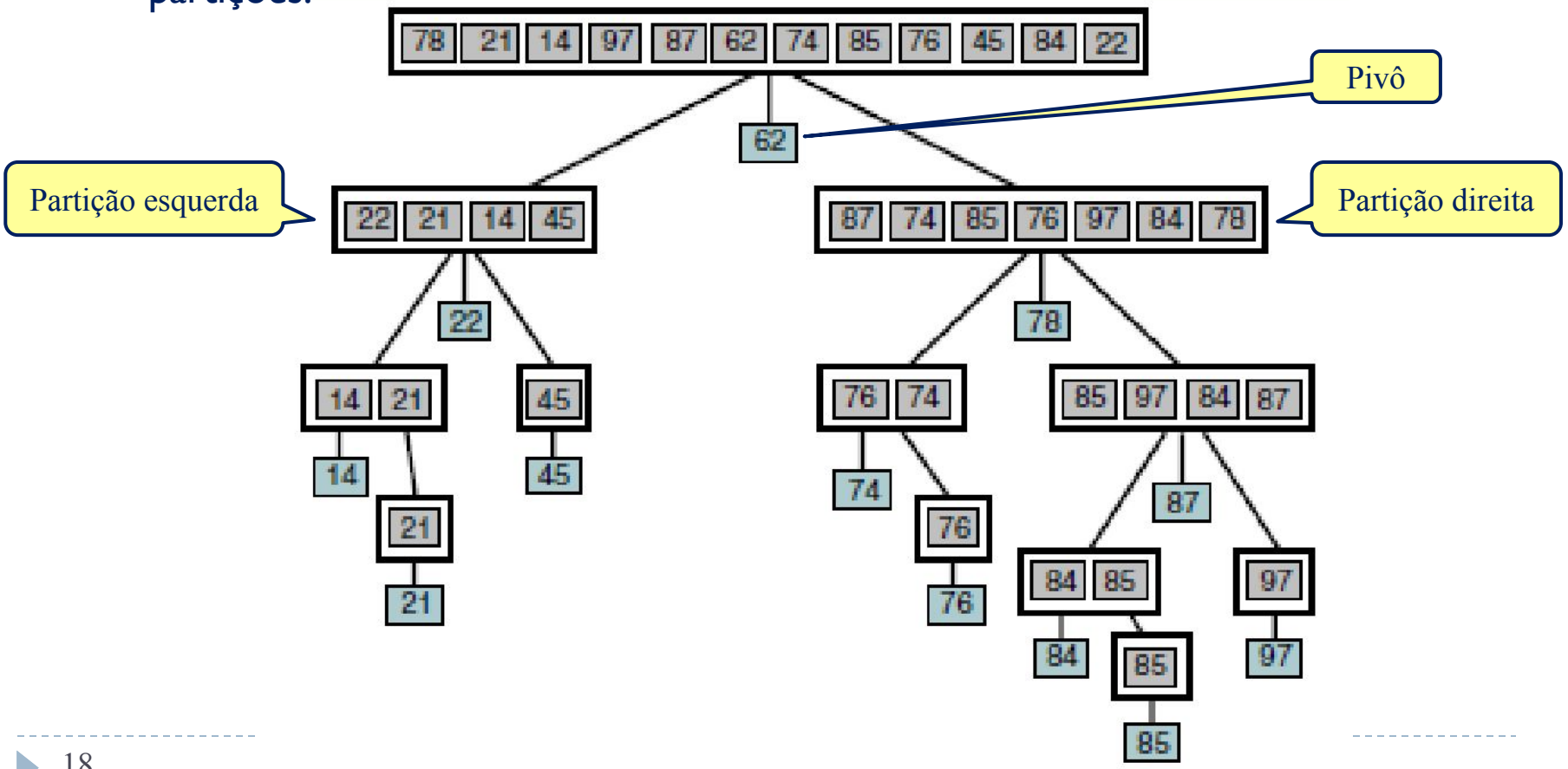


O processo se repete até que as posições de percurso esquerdo e direito se cruzarem.

Quicksort

Exemplo – Particionamento

- O exemplo ilustra o processo de particionamento de uma lista de elementos com base em um pivô. Cada lista é dividida em uma partição esquerda, o pivô e uma partição direita. O processo termina quando não existem mais partições.



Quicksort

Algoritmo – Quicksort (Primeira Parte)

- O algoritmo Quicksort segue as ideias de particionamento e ordenação de partição apresentadas nos exemplos.
- O algoritmo aplica estes princípios para listas suficientemente grandes, em listas pequenas aplica o algoritmo de ordenação por Inserção.

```
Algorithm quickSort (list, left, right)
An array, list, is sorted using recursion.
   Pre  list is an array of data to be sorted
        left and right identify the first and last
        elements of the list, respectively
   Post list is sorted
1  if ((right - left) > minSize)
      Quick sort
   1  medianLeft (list, left, right)
   2  set pivot     to left element
   3  set sortLeft  to left + 1
   4  set sortRight to right
   5  loop (sortLeft    <= sortRight)
         Find key on left that belongs on right
      1  loop (sortLeft key < pivot key)
         1  increment sortLeft
      2  end loop
         Find key on right that belongs on left
      3  loop (sortRight key >= pivot key)
         1  decrement sortRight
      4  end loop
      5  if (sortLeft <= sortRight)
         1  exchange(list, sortLeft, sortRight)
         2  increment sortLeft
         3  decrement sortRight
      6  end if
   6  end loop
```

Se o tamanho da partição for maior que o mínimo

Posições de inicio de percursos

Percurso de esquerda a direita

Percurso de direita a esquerda

Troca elementos fora de ordem

Enquanto não houver cruzamento de posições de percurso.

Quicksort

Algoritmo – Quicksort (Segunda Parte)

- Algoritmo Quicksort (Continuação...)

Copia o último elemento da primeira partição sobre o pivô.

Copia o pivô sobre o último elemento da primeira partição.

Chamadas recursivas nas partições esquerda e direita

```
      Prepare for next pass
   7  move sortLeft - 1 element to left element
   8  move pivot element to sortLeft - 1 element
   9  if  (left < sortRight)
   1  quickSort (list, left, sortRight - 1)
  10  end if
  11  if (sortLeft < right)
   1  quickSort (list, sortLeft, right)
  12  end if
 2 else
   1  insertionSort (list, left, right)
 3 end if
end quickSort
```

Se a partição for pequena ordena por inserção

Quicksort

Implementação – Quicksort (Primeira Parte)

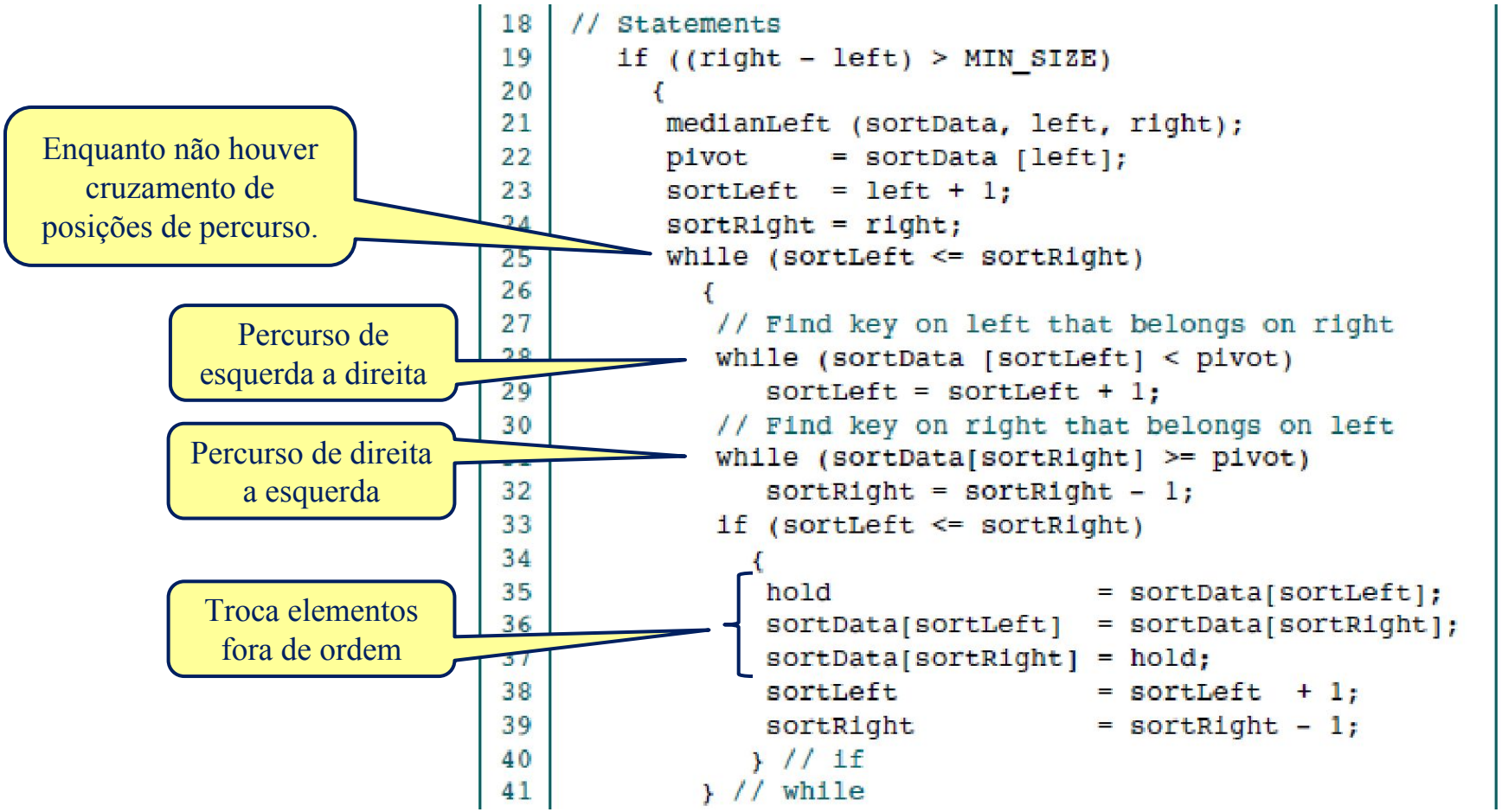
- A implementação segue fielmente o algoritmo descrito anteriormente.
- Aqui temos as definições de variáveis para as posições de percurso esquerdo e direito, o pivô e uma variavél auxiliar para realizar trocas.

```
1  /* ==================== quickSort ====================
2     Array, sortData[left..right] sorted using recursion.
3        Pre  sortData is an array of data to be sorted
4             left identifies first element of sortData
5             right identifies last element of sortData
6        Post sortData array is sorted
7  */
8  void quickSort (int sortData[ ], int left, int right)
9  {
10 #define MIN_SIZE 16
11
12 // Local Definitions
13 int sortLeft;
14 int sortRight;
15 int pivot;
16 int hold;
17
```

Quicksort

Implementação – Quicksort (Segunda Parte)

- Implementação Quicksort (Continuação...)

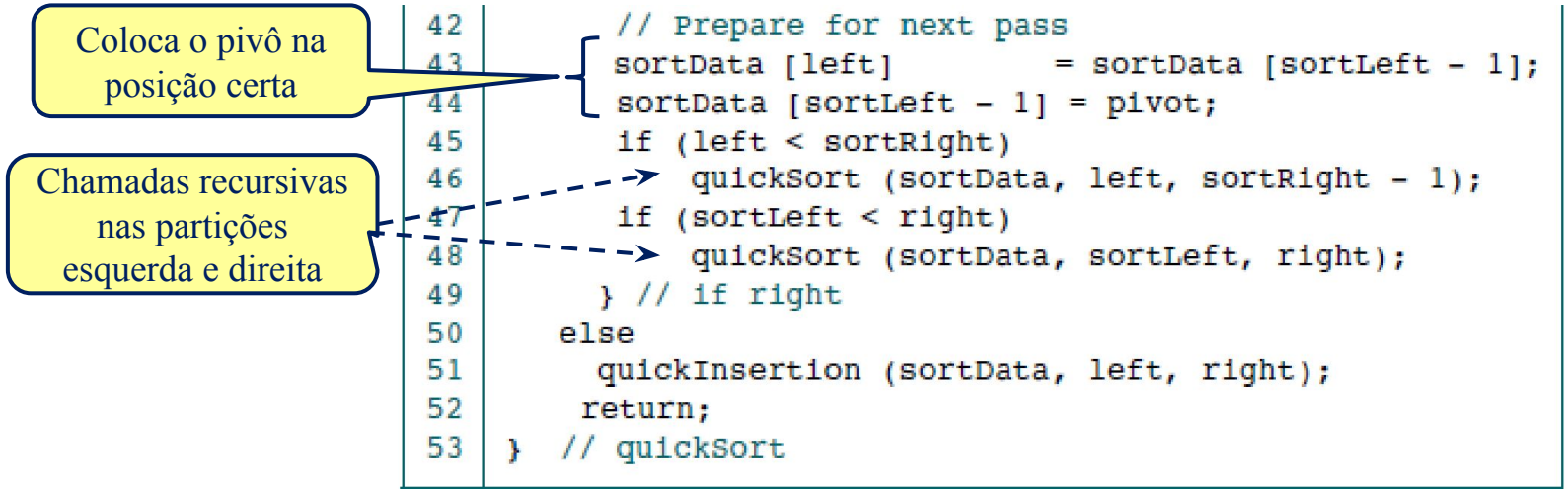



```
18  // Statements
19     if ((right - left) > MIN_SIZE)
20       {
21        medianLeft (sortData, left, right);
22        pivot     = sortData [left];
23        sortLeft  = left + 1;
24        sortRight = right;
25        while (sortLeft <= sortRight)
26          {
27           // Find key on left that belongs on right
28           while (sortData [sortLeft] < pivot)
29              sortLeft = sortLeft + 1;
30           // Find key on right that belongs on left
31           while (sortData[sortRight] >= pivot)
32              sortRight = sortRight - 1;
33           if (sortLeft <= sortRight)
34             {
35              hold                = sortData[sortLeft];
36              sortData[sortLeft]  = sortData[sortRight];
37              sortData[sortRight] = hold;
38              sortLeft            = sortLeft  + 1;
39              sortRight           = sortRight - 1;
40             } // if
41          } // while
```

Quicksort

Implementação – Quicksort (Terceira Parte)

- Implementação Quicksort (Continuação...)




```
42        // Prepare for next pass
43        sortData [left]         = sortData [sortLeft - 1];
44        sortData [sortLeft - 1] = pivot;
45        if (left < sortRight)
46          quickSort (sortData, left, sortRight - 1);
47        if (sortLeft < right)
48          quickSort (sortData, sortLeft, right);
49       } // if right
50     else
51       quickInsertion (sortData, left, right);
52     return;
53   } // quickSort
```

Quicksort

Algoritmo – Ordenação por Inserção Adaptado

- O algoritmo Quicksort tem um melhor desempenho quando utiliza a ordenação por inserção em partições suficientemente pequenas.
- Esta adaptação define o inicio e o fim da partição a ser ordenada.

```
Algorithm quickInsertion (list, first, last)
Sort array list using insertion sort. The list is
divided into sorted and unsorted lists. With each pass, the
first element in the unsorted list is inserted into the
sorted list. This is a special version of the insertion
sort modified for use with quick sort.
   Pre  list must contain at least one element
        first is an index to first element in the list
        last is an index to last element in the list
   Post list has been rearranged
 1 set current to first + 1
 2 loop (until current partition sorted)
   1  move current element to hold
   2  set walker to current - 1
   3  loop (walker >= first AND hold key < walker key)
      1  move walker element one element right
      2  decrement walker
   4  end loop
   5  move hold to walker + 1 element
   6  increment current
 3 end loop
end quickInsertion
```

Enquanto existam elementos a serem inseridos na partição ordenada.

Salva o elemento a inserir em hold.

Busca a posição de inserção.

Move os dados para direita.

Insere hold

Quicksort

Implementação – Inserção Adaptado (Parte1)

- A implementação segue fielmente o algoritmo descrito anteriormente.

```
 1  /* ================ quickInsertion ================
 2     Sort data[1…last] using insertion sort. Data is
 3     divided into sorted and unsorted lists. With each
 4     pass, the first element in unsorted list is inserted
 5     into the sorted list. This is a special version of the
 6     insertion sort modified for use with quick sort.
 7        Pre  data must contain at least one element
 8             first is index to first element in data
 9             last is index to last element in data
10        Post data has been rearranged
11  */
12  void quickInsertion (int data[], int first, int last)
13  {
14  // Local Definitions
15     int hold;
16     int walker;
17
```

Quicksort

Implementação – Inserção Adaptado (Parte2)

- Implementação Inserção Adaptado (Continuação...)

Dependendo da versão da linguagen C utilizada. Pode ser necessário definir a variável fora do loop.

```
18  // Statements
19     for (int current = first + 1;
20               current <= last;
21               current++)
22        {
23         hold    = data[current];
24         walker = current - 1;
25         while (walker >= first
26             && hold < data[walker])
27               {
                  data[walker + 1] = data[walker];
29                walker = walker - 1;
30               } // while
31         data[walker + 1] = hold;
32        } // for
33     return;
34  } // quickInsertion
```

Enquanto existam elementos a serem inseridos na partição ordenada.

Salva o elemento a inserir em hold.

Busca a posição de inserção.

Move os dados para direita

Insere hold

Algoritmos de Ordenação

Comparação

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

n	Number of loops		
	Straight insertion Straight selection Bubble sorts	Shell	Heap and quick
25	625	55	116
100	10,000	316	664
500	250,000	2364	4482
1000	1,000,000	5623	9965
2000	4,000,000	13,374	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.