CS 304 Lecture 9

Sorting and searching

Xiwei Wang, Ph.D.
Assistant Professor
Department of Computer Science
Northeastern Illinois University
Chicago, Illinois, 60625

8 November 2016

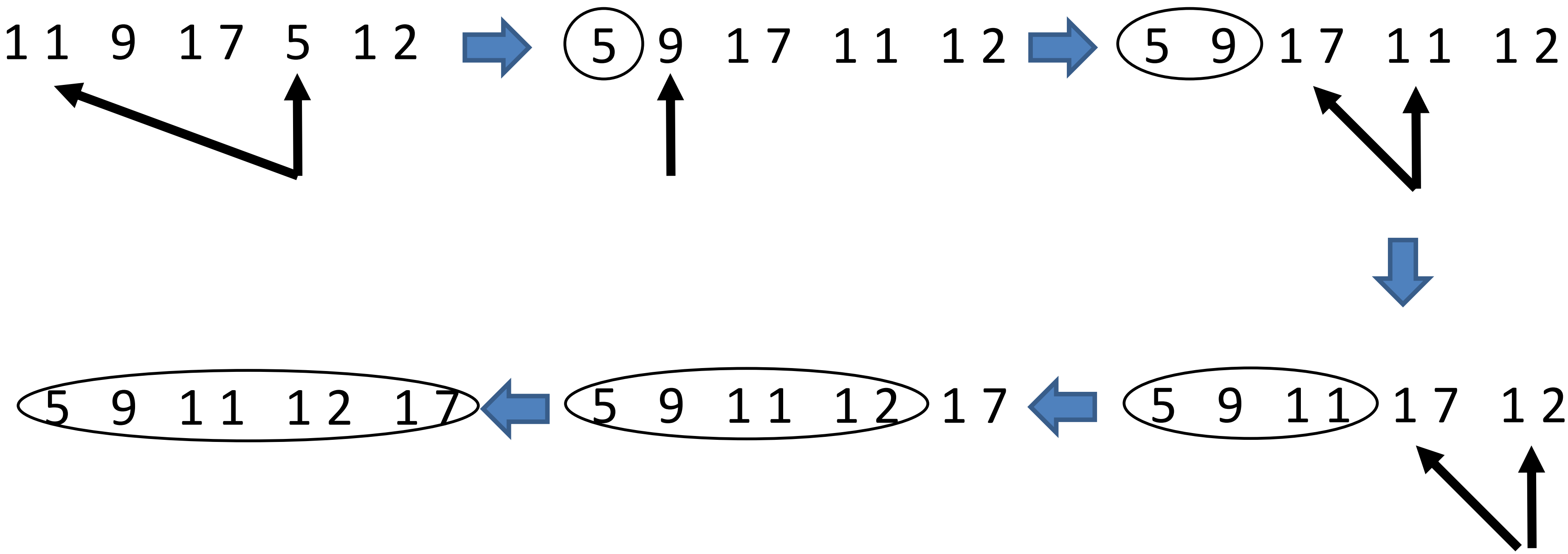
Sorting

- One of the most common tasks in data processing is sorting.
  - A sorting algorithm rearranges the elements of a sequence.
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Quick sort
  - Heap sort

use an unsophisticated brute force approach
not very efficient
easy to understand and to implement

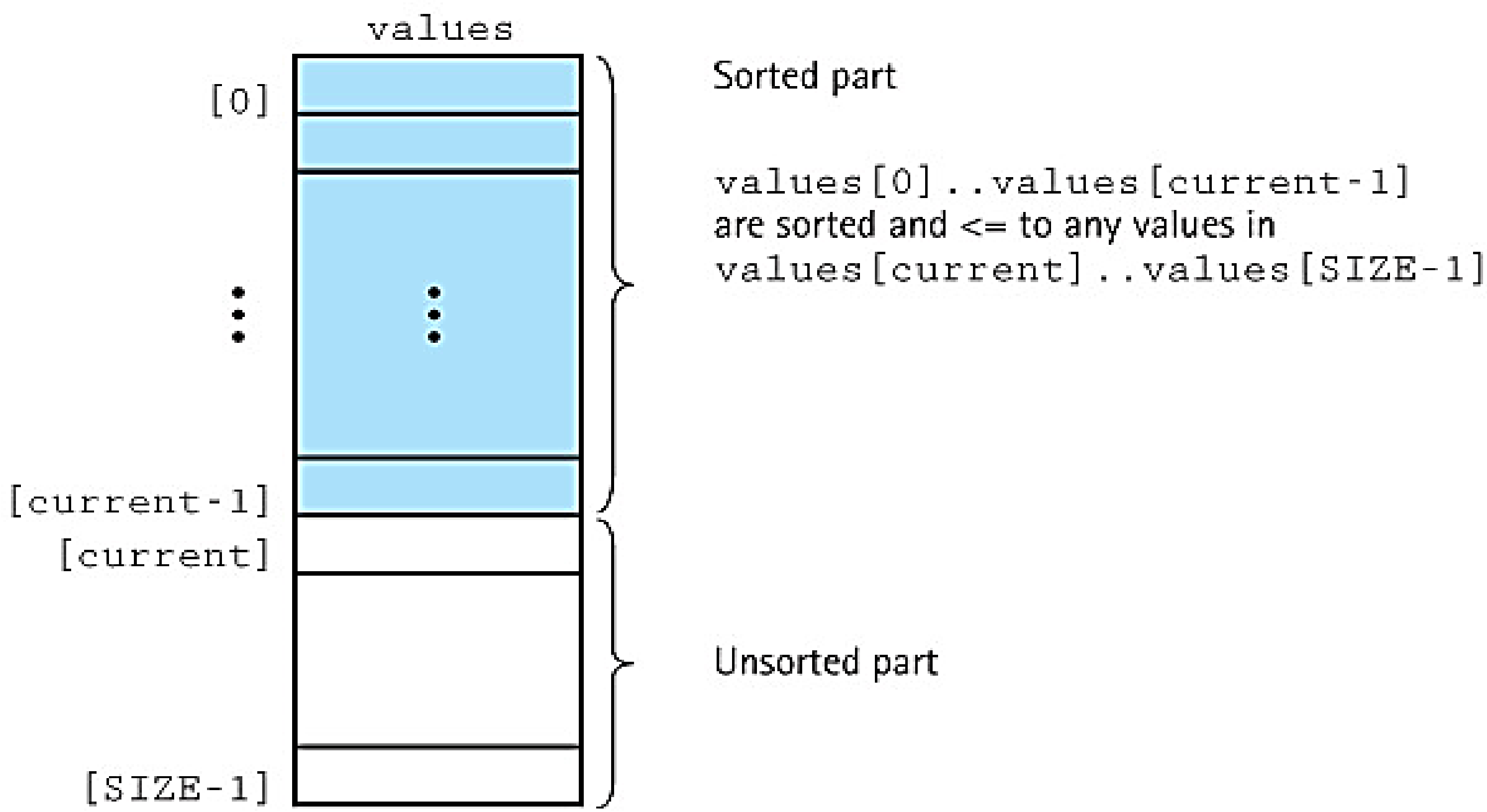
Selection sort

- The selection sort algorithm sorts a sequence by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.



Selection sort

```
SelectionSort
for current going from 0 to SIZE - 2
  Find the index in the array of the smallest unsorted element
  Swap the current element with the smallest unsorted one
```



Selection sort

```
static int minIndex(int startIndex, int endIndex)
// Returns the index of the smallest value in
// values[startIndex]..values[endIndex].
{
  int indexOfMin = startIndex;
  for (int index = startIndex + 1; index <= endIndex; index++)
    if (values[index] < values[indexOfMin])
      indexOfMin = index;
  return indexOfMin;
}

static void selectionSort()
// Sorts the values array using the selection sort algorithm.
{
  int endIndex = SIZE - 1;
  for (int current = 0; current < endIndex; current++)
    swap(current, minIndex(current, endIndex));
}
```

Selection sort

- We describe the number of comparisons as a function of the number of elements in the array. We assume the size of the array is N.
- The `minIndex` method is called N - 1 times.
- Within `minIndex`, the number of comparisons varies:
  - in the first call there are N - 1 comparisons;
  - in the next call there are N - 2 comparisons;
  - and so on, until in the last call, when there is only 1 comparison.
- The total number of comparisons is

$$(N-1) + (N-2) + (N-3) + \ldots + 1$$
$$= N(N-1)/2 = 1/2N^2 - 1/2N$$

- The selection sort algorithm is $O(N^2)$.

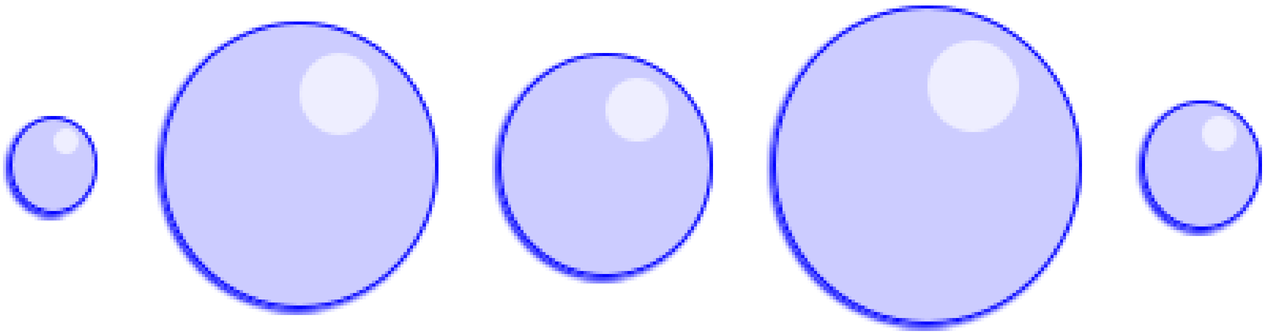
Selection sort

- Number of comparisons required to sort arrays of different sizes using selection sort:

Number of Elements	Number of Comparisons
10	45
20	190
100	4,950
1,000	499,500
10,000	49,995,000

Bubble sort

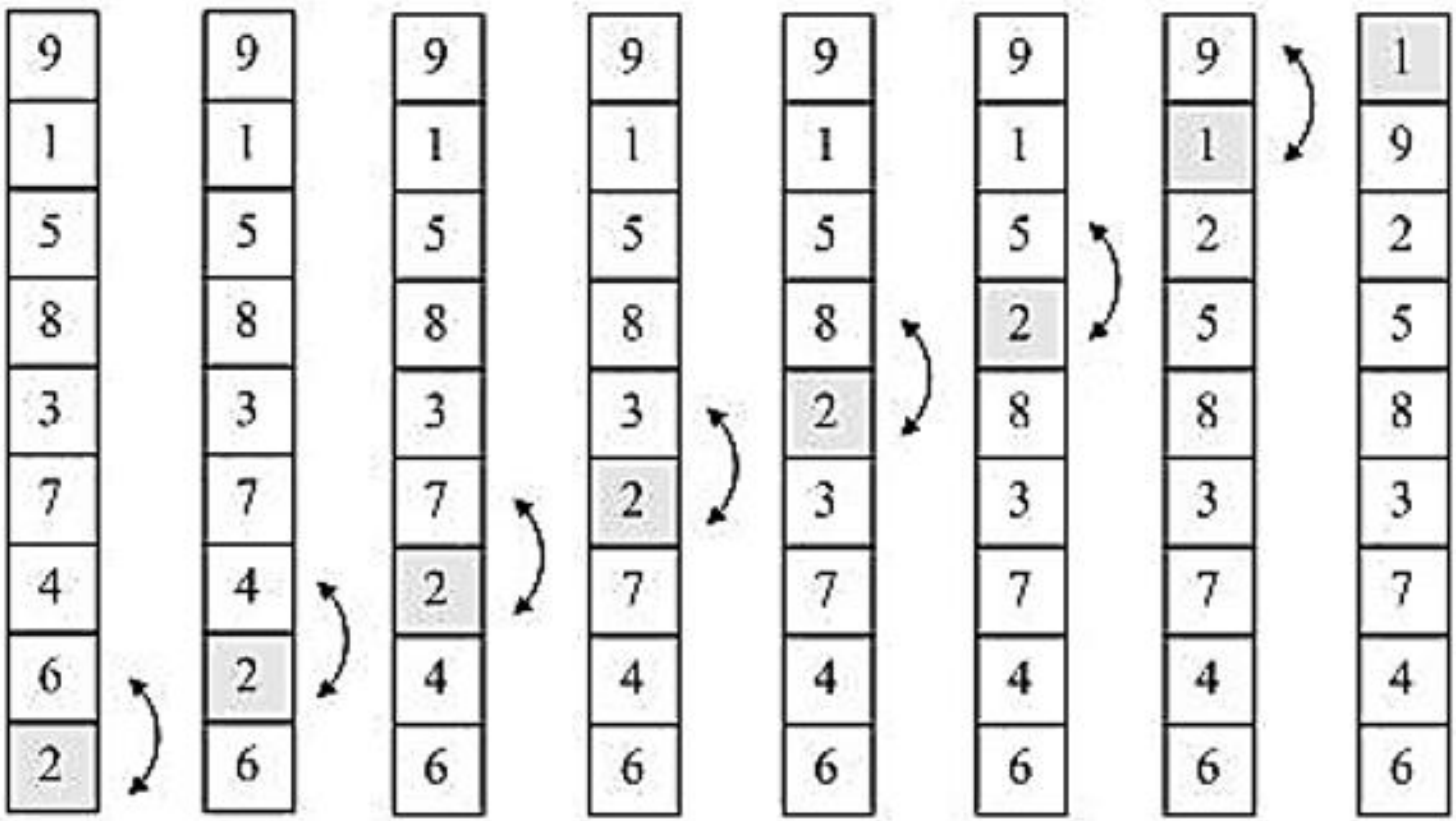
- With this approach the smaller data values "bubble up" to the front of the array …
- Each iteration puts the smallest unsorted element into its correct place, but it also makes changes in the locations of the other elements in the array.



Bubble sort

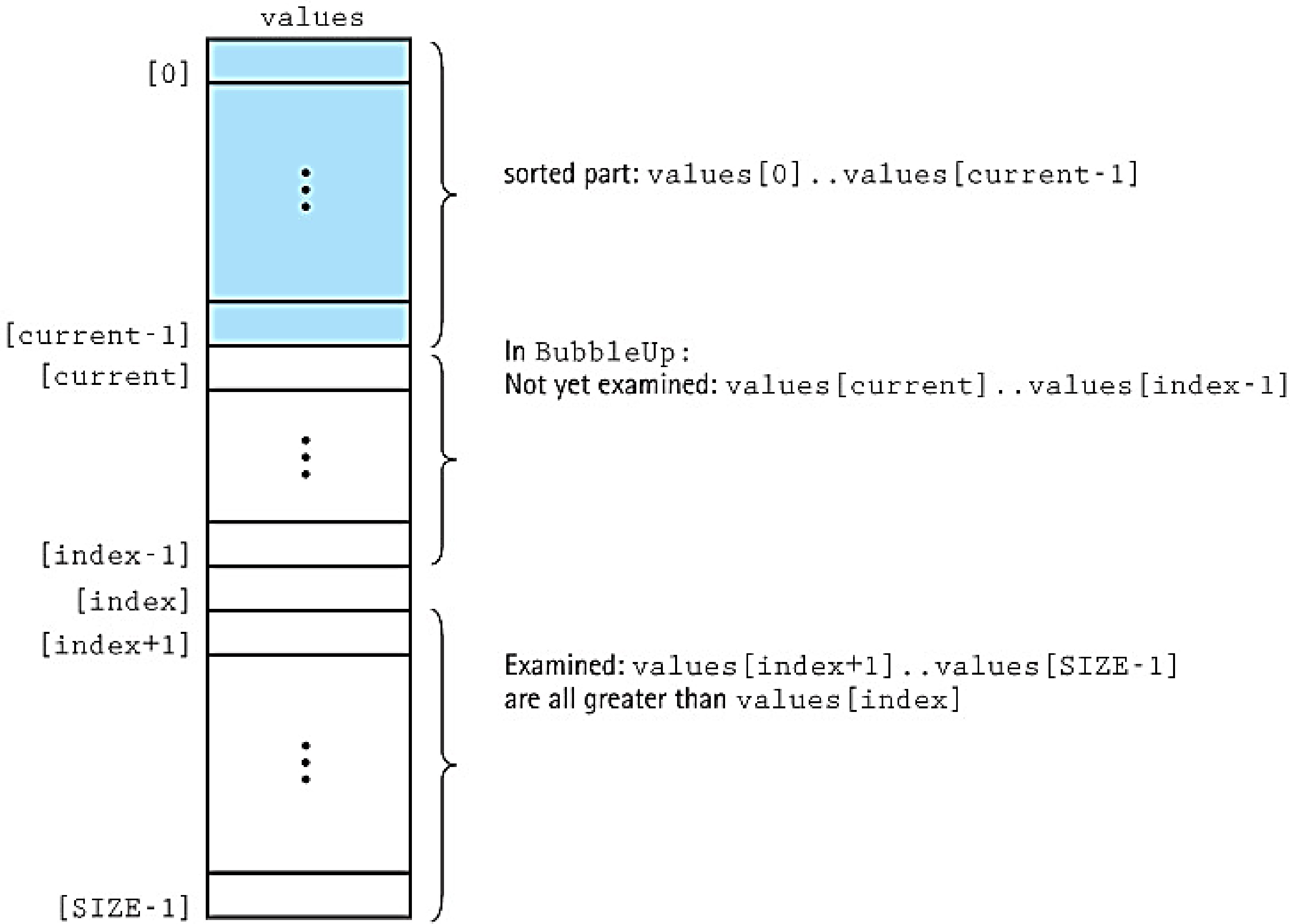
```
BubbleSort
Set current to the index of first element in the array
while more elements in unsorted part of array
    "Bubble up" the smallest element in the unsorted
        part, causing intermediate swaps as needed
     Shrink the unsorted part of the array by
        incrementing current

bubbleUp(startIndex, endIndex)
for index going from endIndex DOWNTO startIndex + 1
    if values[index] < values[index - 1]
        Swap the value at index with the value at index - 1
```

Bubble sort



`bubbleUp(0, 8)`

Bubble sort


values
[0]
[current-1]
[current]
[index-1]
[index]
[index+1]
[SIZE-1]
sorted part: values[0]..values[current-1]
In BubbleUp:
Not yet examined: values[current]..values[index-1]
Examined: values[index+1]..values[SIZE-1]
are all greater than values[index]

Bubble sort

```
static void bubbleUp(int startIndex, int endIndex)
// Switches adjacent pairs that are out of order
// between values[startIndex]..values[endIndex]
// beginning at values[endIndex].
{
  for (int index = endIndex; index > startIndex; index--)
    if (values[index] < values[index - 1])
      swap(index, index - 1);
}

static void bubbleSort()
// Sorts the values array using the bubble sort algorithm.
{
  int current = 0;
  while (current < SIZE - 1)
  {
    bubbleUp(current, SIZE - 1);
    current++;
  }
}
```

Bubble sort

- The comparisons are in **`bubbleUp`**, which is called $N - 1$ times.
- There are $N - 1$ comparisons the first time, $N - 2$ comparisons the second time, and so on.
- Therefore, **`bubbleSort`** and **`selectionSort`** require the same amount of work in terms of the number of comparisons.
- The bubble sort algorithm is O(N^2).

Insertion sort

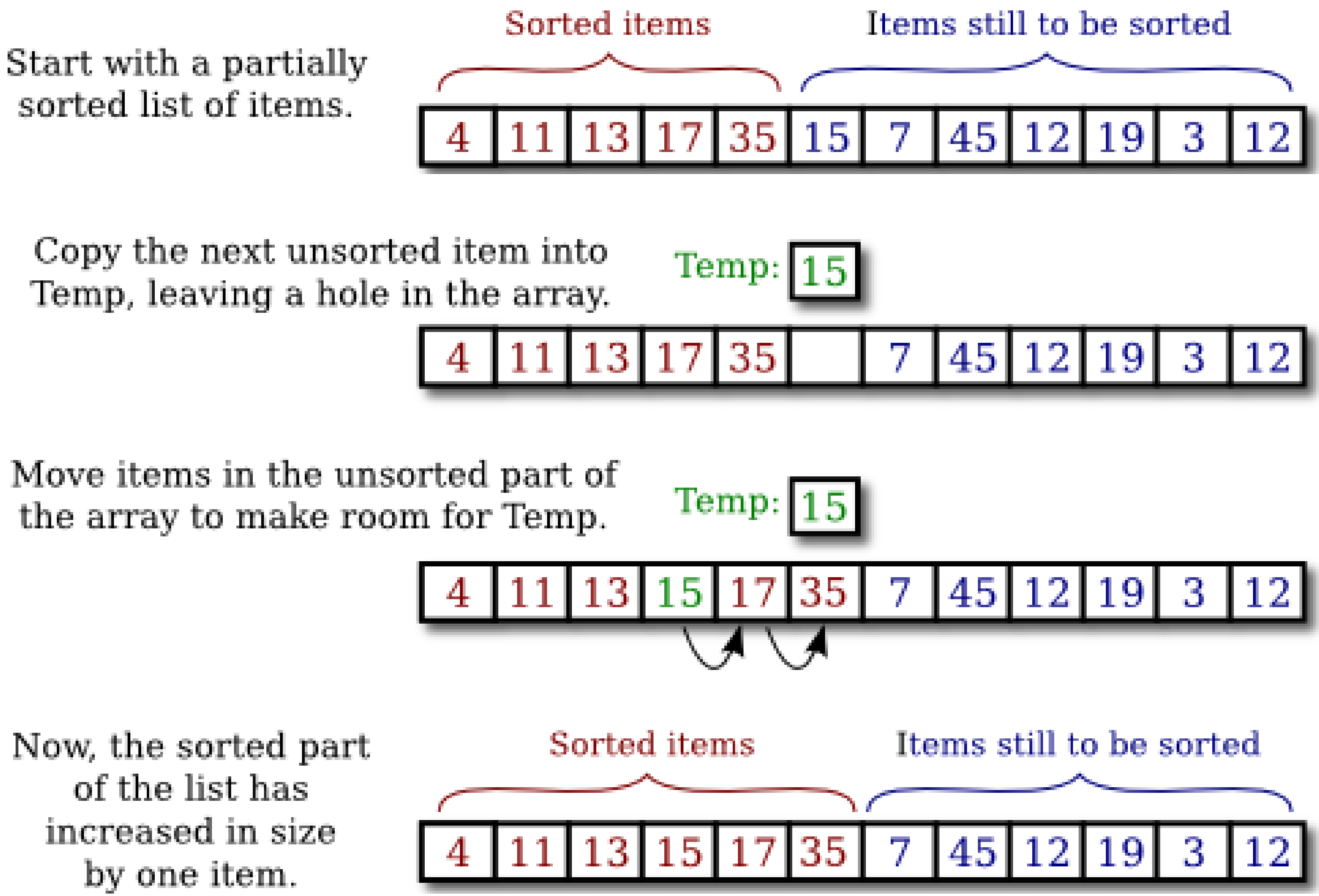
- The insertion sort algorithm moves elements one at a time into the correct position.
- We divide our array into a sorted part and an unsorted part.
  - Initially, the sorted portion contains only one element: the first element in the array.
  - Next we take the second element in the array and put it into its correct place in the sorted part; that is, **`values[0]`** and **`values[1]`** are in order with respect to each other.
  - Next the value in **`values[2]`** is put into its proper place, so **`values[0]`**..**`values[2]`** are in order with respect to each other.
  - This process continues until all the elements have been sorted.

https://www.youtube.com/watch?v=ROalU379l3U

Insertion sort

```
insertionSort
for count going from 1 through SIZE - 1
    insertElement(0, count)


InsertElement(startIndex, endIndex)
Set finished to false
Set current to endIndex
Set moreToSearch to true
while moreToSearch AND NOT finished
    if values[current] < values[current - 1]
        swap(values[current], values[current - 1])
        Decrement current
        Set moreToSearch to (current does not equal startIndex)
    else
        Set finished to true
```

Insertion sort



Insertion sort

```
static void insertElement(int startIndex, int endIndex)
// Upon completion, values[0]..values[endIndex] are sorted.
{
  boolean finished = false;
  int current = endIndex;
  boolean moreToSearch = true;
  while (moreToSearch && !finished) {
    if (values[current] < values[current - 1])  {
      swap(current, current - 1);
      current--;
      moreToSearch = (current != startIndex);
    }
    else
      finished = true;
  }
}

static void insertionSort()
{
  for (int count = 1; count < SIZE; count++)
    insertElement(0, count);
}
```

Insertion sort

- The general case for this algorithm mirrors the **`selectionSort`** and the **`bubbleSort`**, so the general case is O(N^2) .
- But **`insertionSort`** has a "best" case: the data are already sorted in ascending order.
  - **`insertElement`** is called N times, but only one comparison is made each time and no swaps are necessary.
- The maximum number of comparisons is made only when the elements in the array are in reverse order.

Merge sort

- $O(N^2)$ sorts are very time consuming for sorting large arrays.
  - Doubling the size of the array causes a fourfold increase in the time required for sorting it.
  - We will hope that there are more sophisticated sorting algorithms that we can choose to dramatically improve the performance of the sorting process.
  - There are several sorting methods that work better when N is large.
  - Merge sort is an $O(N \log_2 N)$ sorting algorithm.

Merge sort

- Conceptually, a merge sort works as follows
  - 1. Divide the unsorted array into N subarrays, each containing 1 element (remember that an array of 1 element is considered sorted).
  - 2. Repeatedly merge subarrays to produce new subarrays until there is only 1 subarray remaining. This will be the sorted array.

5	9	10	12	17	1	8	11	20	32

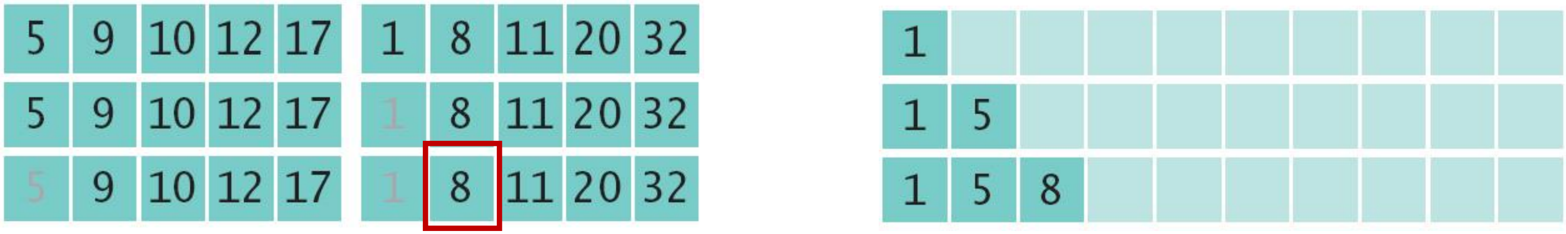
We will simply hope that the first half of the array is already perfectly sorted, and the second half is too.

- Now it is an easy matter to merge the two sorted sequences into a sorted sequence, simply by taking a new element from either the first or the second subarray and choosing the smaller of the elements each time.

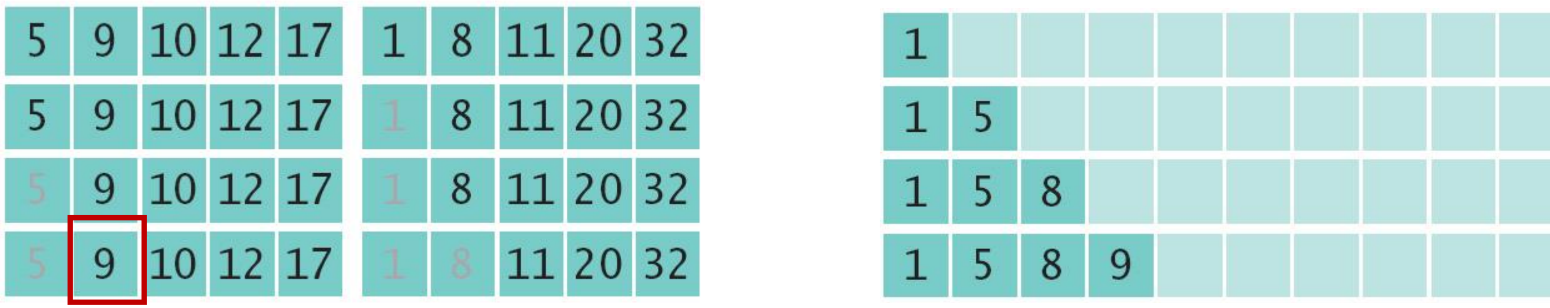
Merge sort

Merge sort

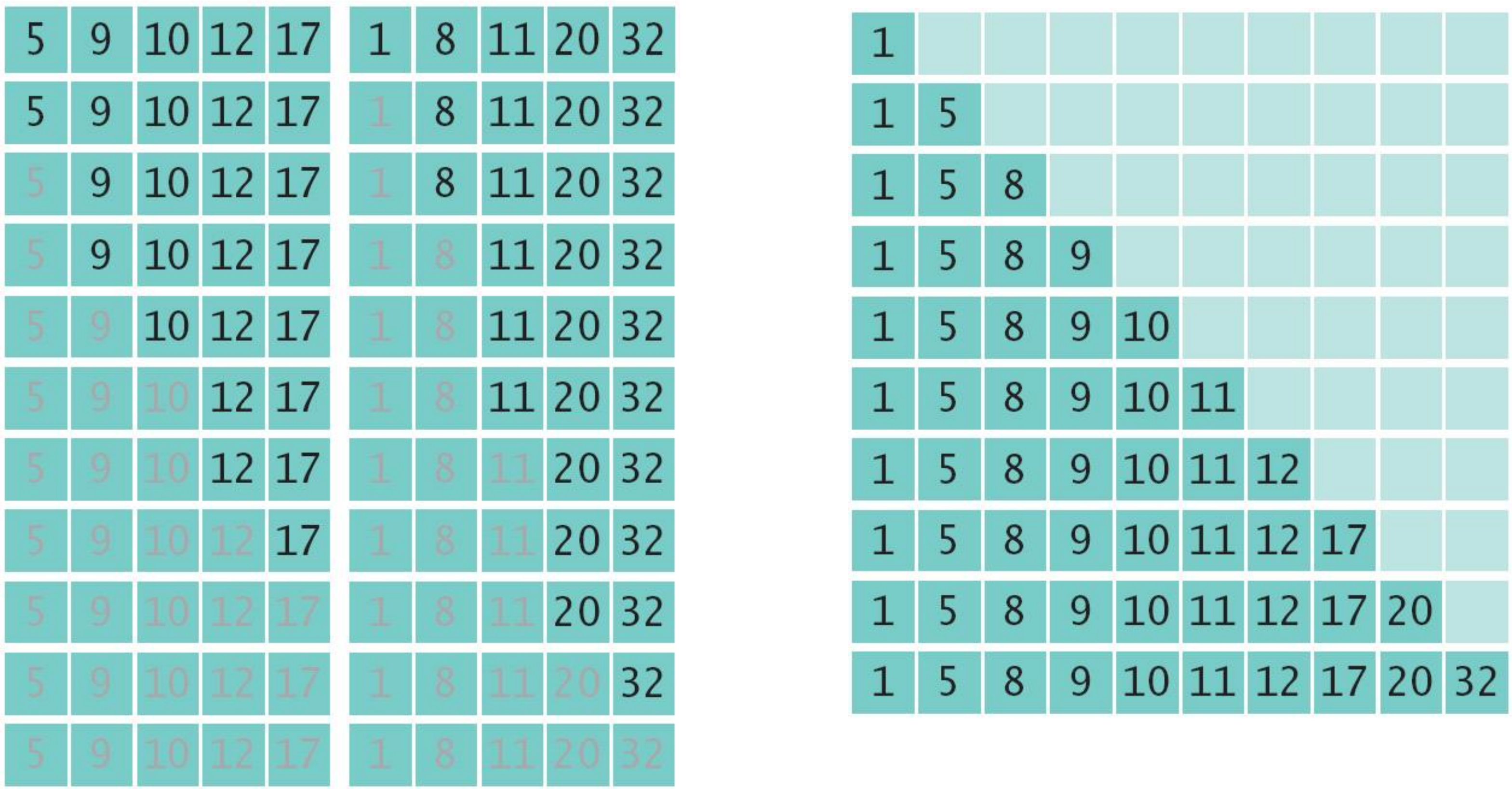
Merge sort


5 9 10 12 17 1 8 11 20 32
5 9 10 12 17 1 8 11 20 32
5 9 10 12 17 1 8 11 20 32
1
1 5
1 5 8

Merge sort


5 9 10 12 17 1 8 11 20 32
5 9 10 12 17 1 8 11 20 32
5 9 10 12 17 1 8 11 20 32
5 9 10 12 17 1 8 11 20 32
1
1 5
1 5 8
1 5 8 9

Merge sort


5 9 10 12 17 1 8 11 20 32
1
1 5
1 5 8
1 5 8 9
1 5 8 9 10
1 5 8 9 10 11
1 5 8 9 10 11 12
1 5 8 9 10 11 12 17
1 5 8 9 10 11 12 17 20
1 5 8 9 10 11 12 17 20 32

Merge sort

```
mergeSort
Cut the array in half
Sort the left half
Sort the right half
Merge the two sorted halves into one sorted array
```

Because **mergeSort** is itself a sorting algorithm, we might as well use it to sort the two halves.

We can make **mergeSort** a recursive method and let it call itself to sort each of the two subarrays:

```
mergeSort—Recursive
Cut the array in half
mergeSort the left half
mergeSort the right half
Merge the two sorted halves into one sorted array
```

Merge sort

Method `mergeSort(first, last)`

Definition: Sorts the array elements in ascending order.

Size: last - first + 1

Base Case: If size less than 2, do nothing.

General Case: Cut the array in half.

`mergeSort` the left half.

`mergeSort` the right half.

Merge the sorted halves into one sorted array.

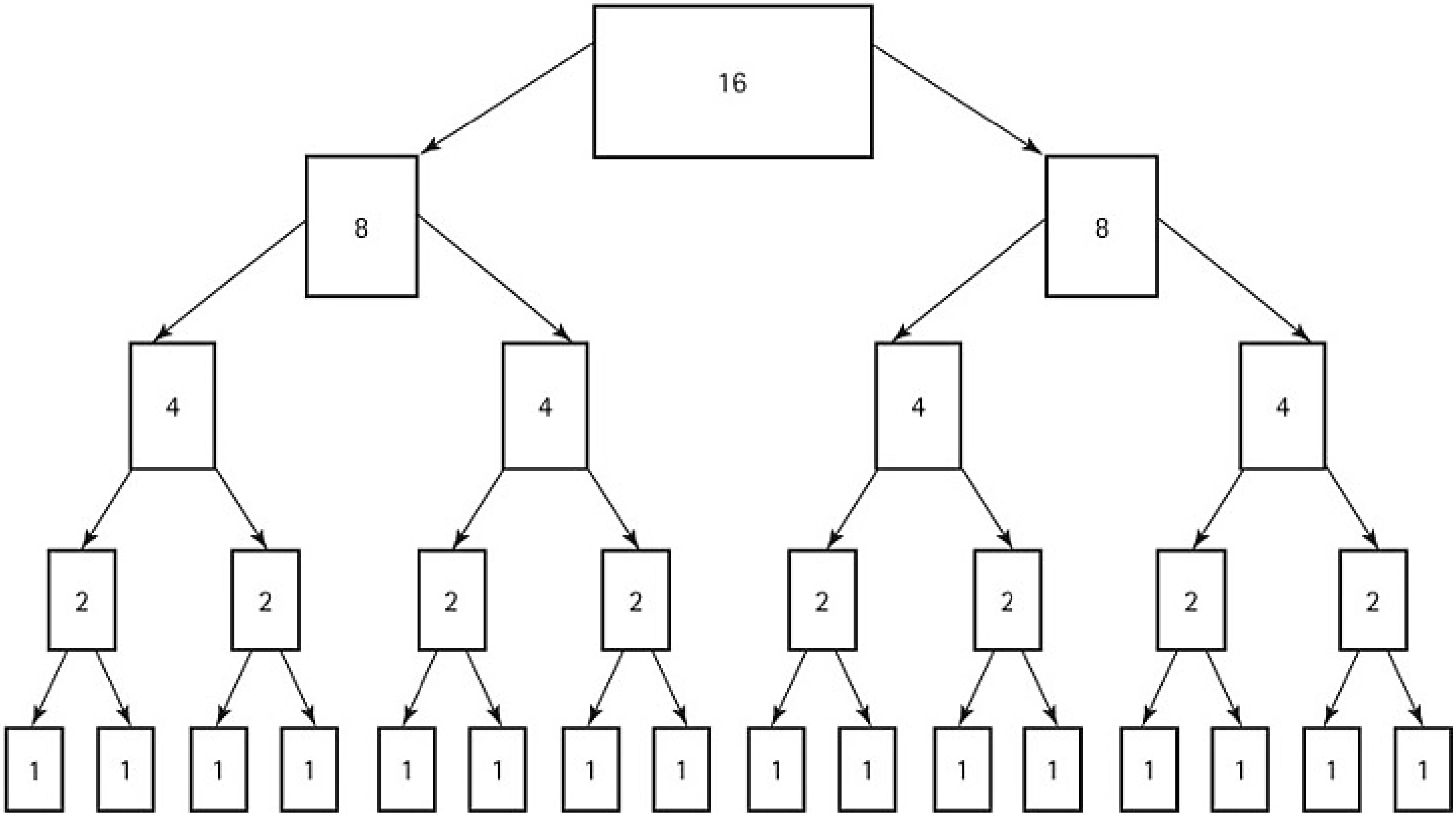
Merge sort

```
merge(leftFirst, leftLast, rightFirst, rightLast)

(uses a local array, tempArray)

Set index to leftFirst
while more elements in left half AND more elements in right half
    if values[leftFirst] < values[rightFirst]
        Set tempArray[index] to values[leftFirst]
        Increment leftFirst
    else
        Set tempArray[index] to values[rightFirst]
        Increment rightFirst
    Increment index
Copy any remaining elements from left half to tempArray
Copy any remaining elements from right half to tempArray
Copy the sorted elements from tempArray back into values
```

Merge sort

```
static void mergeSort(int first, int last)
// Sorts the values array using the merge sort algorithm.
{
  if (first < last)
  {
    int middle = (first + last) / 2;
    mergeSort(first, middle);
    mergeSort(middle + 1, last);
    merge(first, middle, middle + 1, last);
  }
}
```

Merge sort



Merge sort

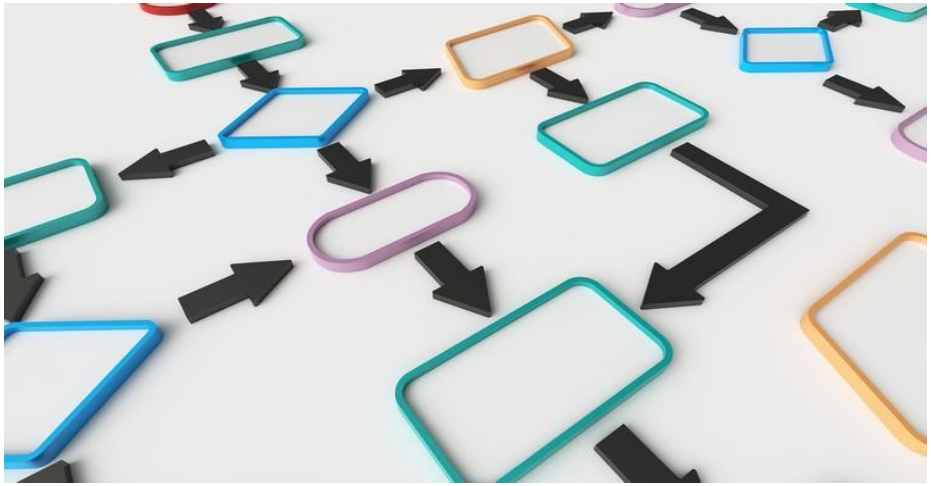
- The total work needed to divide the array in half, over and over again until we reach subarrays of size 1, is O(N).
- It takes O(N) total steps to perform merging at each "level" of merging.
- The number of levels of merging is equal to the number of times we can split the original array in half.
  - If the original array is size N, we have $\log_2 N$ levels.
- Because we have $\log_2 N$ levels, and we require O(N) steps at each level, the total cost of the merge operation is: O($N \log_2 N$).
- Because the splitting phase was only O(N), we conclude that Merge Sort algorithm is O($N \log_2 N$).

Comparing N^2 and $N\log_2 N$

N	$\log_2 N$	N^2	$N\log_2 N$
32	5	1,024	160
64	6	4.096	384
128	7	16,384	896
256	8	65,536	2,048
512	9	262,144	4,608
1024	10	1,048,576	10,240
2048	11	4,194,304	22,528
4096	12	16,777,216	49,152

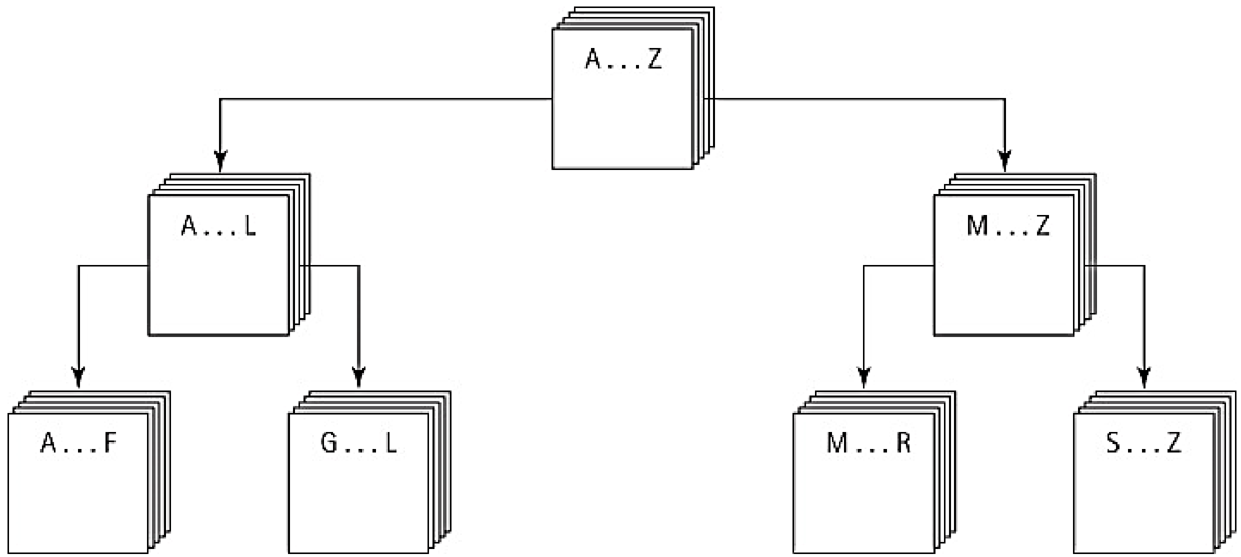
Drawback of merge sort

- A disadvantage of **`mergeSort`** is that it requires an auxiliary array that is as large as the original array to be sorted.
- If the array is large and space is a critical factor, this sort may not be an appropriate choice.
- Next we discuss an $O(N\log_2 N)$ sort that moves elements around in the original array and does not need an auxiliary array.



Quick sort

- A divide-and-conquer algorithm and inherently recursive.
- At each stage the part of the array being sorted is divided into two "piles", with everything in the left pile less than everything in the right pile.
    - The same approach is used to sort each of the smaller piles (a smaller case).
    - This process goes on until the small piles do not need to be further divided (the base case).



Quick sort

Method `quickSort(first, last)`

Definition: Sorts the elements in sub array **`values[first]..values[last]`**.

Size: **`last`** - **`first`** + 1

Base Case: If size less than 2, do nothing.

General Case: Split the array according to splitting value.

`quickSort` the elements <= splitting value.

`quickSort` the elements > splitting value.

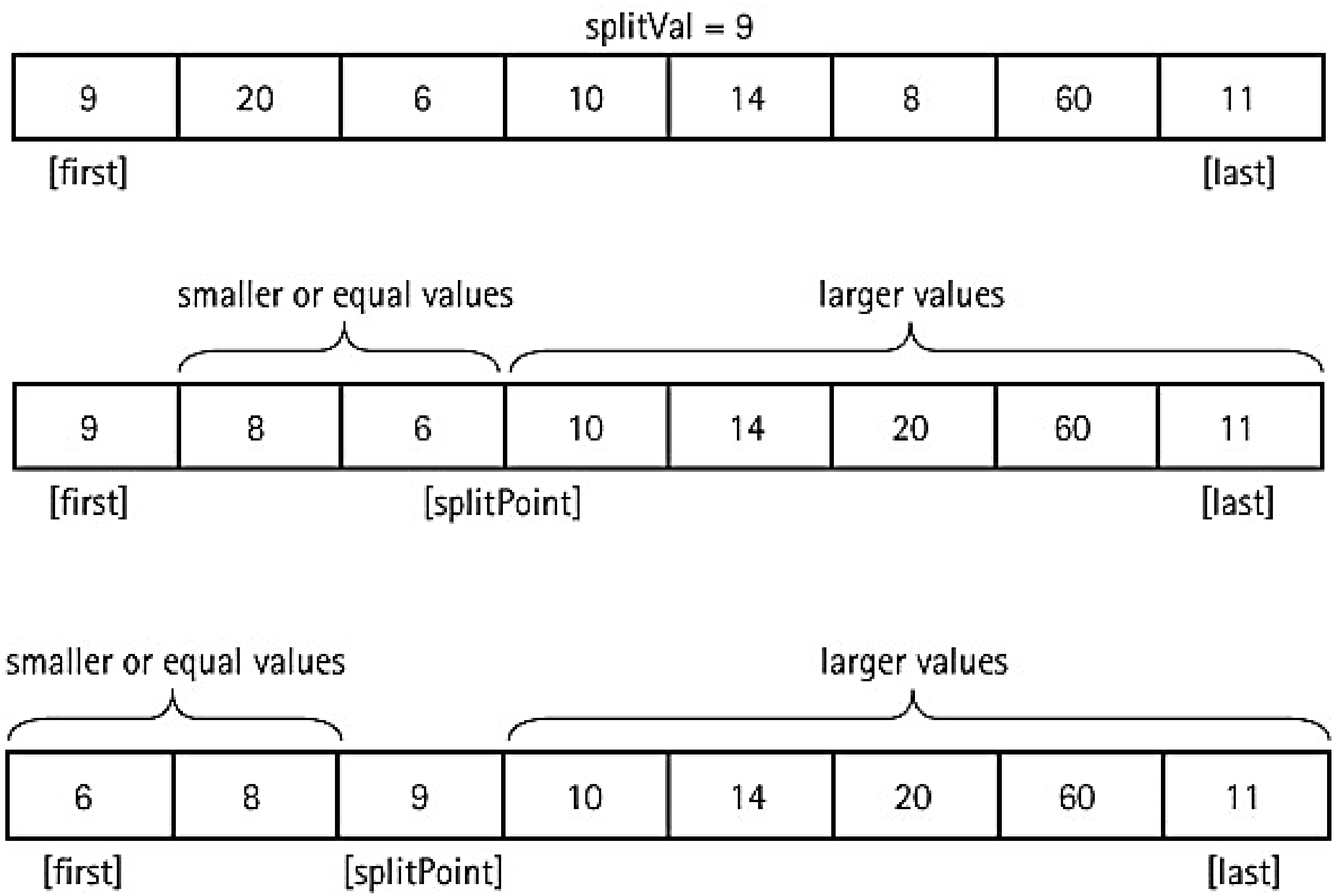
Quick sort

```
quickSort
if there is more than one element in values[first]..values[last]
    Select splitVal
    Split the array so that
        values[first]..values[splitPoint – 1] <= splitVal
        values[splitPoint] = splitVal
        values[splitPoint + 1]..values[last] > splitVal
    quickSort the left sub array
    quickSort the right sub array
```

- The algorithm depends on the selection of a "splitting value" (usually known as "*pivot*"), called **`splitVal`**, that is used to divide the array into two sub arrays.
- How do we select **`splitVal`**?
    - One simple solution is to use the value in **`values[first]`** as the splitting value.

Quick sort

- Quick sort steps



Quick sort

```
static void quickSort(int first, int last)
{
  if (first < last)
  {
    int splitPoint;

    splitPoint = split(first, last);
    // values[first]..values[splitPoint - 1] <= splitVal
    // values[splitPoint] = splitVal
    // values[splitPoint+1]..values[last] > splitVal

    quickSort(first, splitPoint - 1);
    quickSort(splitPoint + 1, last);
  }
}
```

Quick sort

- The `split` operation

(a) Initialization. Note that `splitVal = values[first]` = 9.

9	20	6	10	14	8	60	11
[saveF] [first]							[last]

(b) Increment `first` until `values[first]>splitVal`

9	20	6	10	14	8	60	11
[saveF]	[first]						[last]

(c) Decrement `last` until `values[last]<= splitVal`

9	20	6	10	14	8	60	11
[saveF]	[first]				[last]		

Quick sort

- The `split` operation

(d) Swap `values[first]` and `values[last]`; move `first` and `last` toward each other

9	8	6	10	14	20	60	11
[saveF]		[first]		[last]			

(e) Increment `first` until `values[first]>splitVal` or `first>last`.
Decrement `last` until `values[last]<= splitVal` or `first>last`

9	8	6	10	14	20	60	11
[saveF]		[last]	[first]				

(f) `first>last` so no swap occurs within the loop.
swap `values[saveF]` and `values[last]`

6	8	9	10	14	20	60	11
[saveF]		[last] (splitPoint)					

Quick sort

- On the first call, every element in the array is compared to the splitting value (the "pivot"), so the work done is O(N).
- The array is divided into two sub arrays (not necessarily halves)
- Each of these pieces is then divided in two, and so on.
- If each piece is split approximately in half, there are O($\log_2 N$) levels of splits. At each level, we make O(N) comparisons.
- So quick sort is an O($N \log_2 N$) algorithm. It is especially quick for large collections of *random data*.

Drawback of quick sort

- Quick Sort isn't always quicker.
  - There are $\log_2 N$ levels of splits if each split divides the segment of the array approximately in half.
  - If the splits are very lopsided, and the subsequent recursive calls to **`quickSort`** also result in lopsided splits, we can end up with a sort that is $O(N^2)$.
- What about space requirements?
  - There can be many levels of recursion "saved" on the system stack at any time.
  - On average, the algorithm requires $O(\log_2 N)$ extra space to hold this information and in the worst case requires $O(N)$ extra space, the same as merge sort.

More sorting considerations

- To thoroughly test our sorting methods we should
  - vary the size of the array
  - vary the original order of the array
    - Random order
    - Reverse order
    - Almost sorted
    - All identical elements
- When N is small the simple sorts may be more efficient than the "fast" sorts because they require less overhead.
- Stability of a sorting algorithm
  - **Stable sort** - A sorting algorithm that preserves the order of duplicates.
  - **`quickSort`** and **`heapSort`** are inherently unstable.

Searching

- Searching for an element in a sequence is an extremely common task.
- As with sorting, the right choice of algorithms can make a big difference.
  - Linear search
  - Binary search
  - Hashing

Linear search

- A linear search begins with the first element in the list, searches for the desired element by examining each subsequent element's key until either the search is successful or the list is exhausted.
  - Based on the number of comparisons this search is O(N).
  - In the worst case we have to make N key comparisons.
  - On the average, assuming that there is an equal probability of searching for any element in the list, we make $N/2$ comparisons for a successful search.

High-probability ordering

- Sometimes certain list elements are in much greater demand than others. We can then improve the search:
    - Put the most-often-desired elements at the beginning of the list.
    - Using this scheme, we are more likely to make a hit in the first few tries, and rarely do we have to search the whole list.
- If the elements in the list are not static or if we cannot predict their relative demand, we can
    - move each element accessed to the front of the list;
    - as an element is found, it is swapped with the element that precedes it.
- Lists in which the relative positions of the elements are changed in an attempt to improve search efficiency are called *self-organizing* or *self-adjusting lists*.

Binary search

- If we know that the values in an array were already sorted, we will probably not use linear search.
- Consider this array:

We are looking for 123.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
14	43	76	100	115	290	400	511

Is 123 in the first half ? You should compare 123 with element [3]. Since 123 is greater than 100, if 123 is in the array, it must be in the second half.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
14	43	76	100	115	290	400	511

Consider the last index in the lower half of the array from index 4 to 7. That's index 5. Since 123 is less than 290 so it must be in the left half of this subarray (or not in the array at all)

Binary search

- If we know that the values in an array were already sorted, we will probably not use linear search.
- Consider this array:

We are looking for 123.

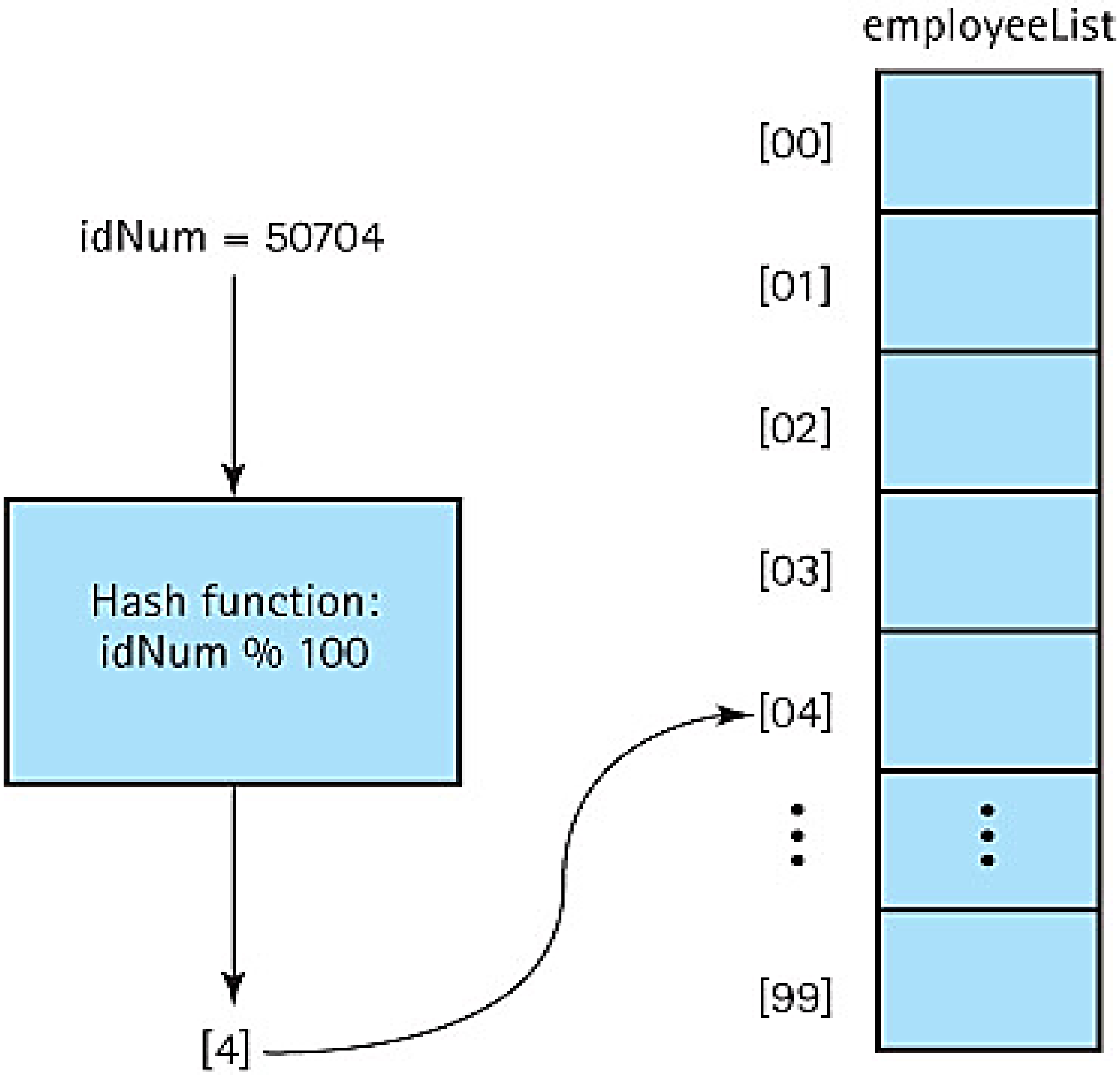
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
14	43	76	100	115	290	400	511

Consider the last index in the lower half of the very short subarray from index 4 to 5. That's at index 4. Since 123 is greater than 115 so you must look at the very, very short array index 5.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
14	43	76	100	115	290	400	511

There's only one element in this subarray and 290 is not 123. Thus, 123 is not found.

Hashing



employeeList
[00]
[01]
[02]
[03]
[04]
[99]
idNum = 50704
Hash function:
idNum % 100
[4]

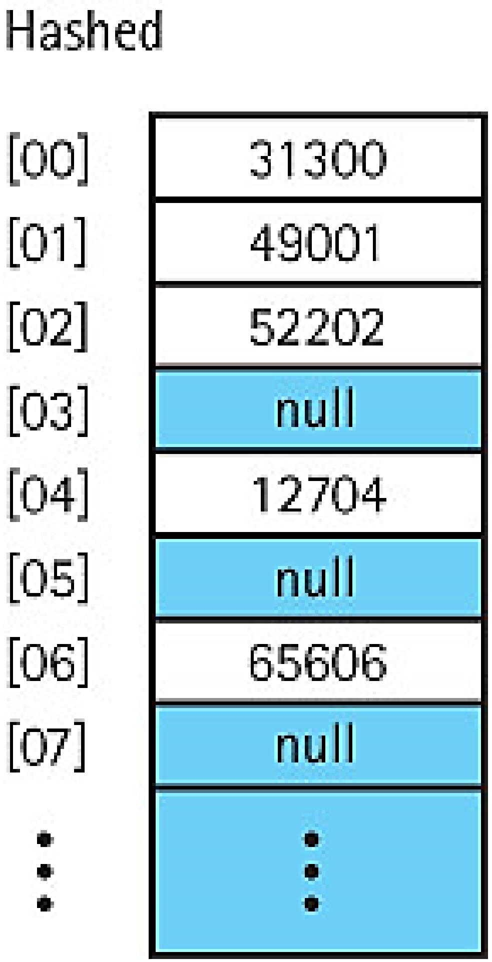
Hashing

- **Hash Function** - A function used to manipulate the key of an element in a list to identify its location in the list.
- **Hashing** - The technique for ordering and accessing elements in a list in a relatively constant amount of time by manipulating the key to identify its location in the list.
- **Hash table** - A data structure used to store and retrieve elements using hashing.

```
public interface Hashable
// Objects of classes that implement this interface can be
// used with lists based on hashing.
{
// A mathematical function used to manipulate the key of
// an element in a list to identify its location in the list.
  int hash();
}
```

Hashing

```
public void add(Hashable element)
// Adds element to this list at position element.hash().
{
  int location;
  location = element.hash();
  list[location] = element;
  numElements++;
}


public Hashable get(Hashable element)
// Returns an element e from this list such
// that e.equals(element).
{
  int location;
  location = element.hash();
  return (Hashable)list[location];
}
```



Hashing

- **Collision** - The condition resulting when two or more keys produce the same hash location.
- **Linear probing** - Resolving a hash collision by sequentially searching a has table beginning at the location returned by the hash function.

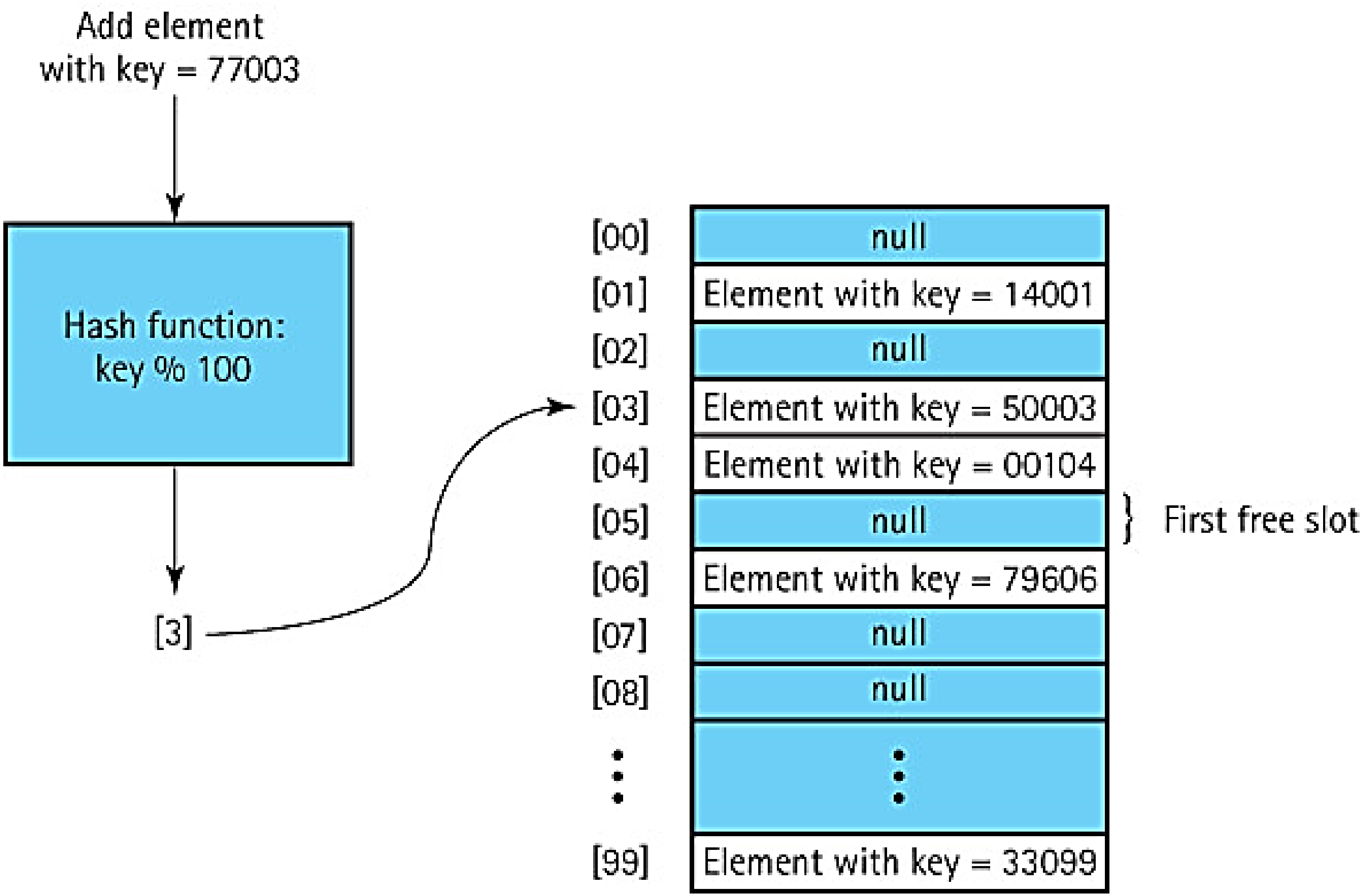
Hashing

```
public static void add(Hashable element)
// Adds element to this list at position element.hash(),
// or the next free array slot.
{
  int location;
  location = element.hash();
  while (list[location] != null)
    location = (location + 1) % list.length;
  list[location] = element;
  numElements++;
}

public static Hashable get(Hashable element)
// Returns an element e from this list such that
e.equals(element).
{
  int location;
  location = element.hash();
  while (!list[location].equals(element))
    location = (location + 1) % list.length;
  return (Hashable)list[location];
}
```

The revised methods

Hashing

- Handling collisions with linear probing:



Hashing

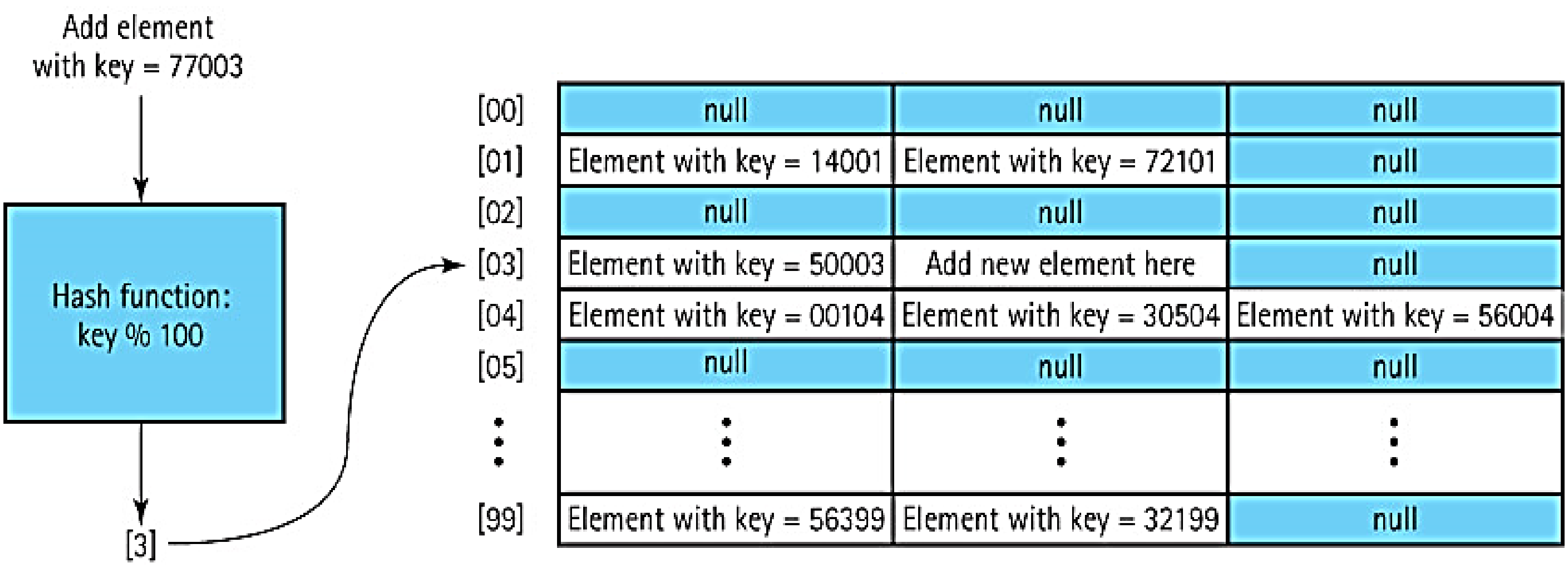
- Removing an element. Here is a possible approach:

```
remove (element)
Set location to element.hash( )
Set list[location] to null
```

- Collisions, however, complicate the matter. We cannot be sure that our element is in location **`element.hash()`**.
  - We must examine every array element, starting with location **`element.hash()`**, until we find the matching element.
  - We cannot stop looking when we reach an empty location, because that location may represent an element that was previously removed.
- This problem illustrates that hash tables, in the forms that we have studied thus far, are not the most effective data structure for implementing lists whose elements may be deleted.

Buckets and chaining

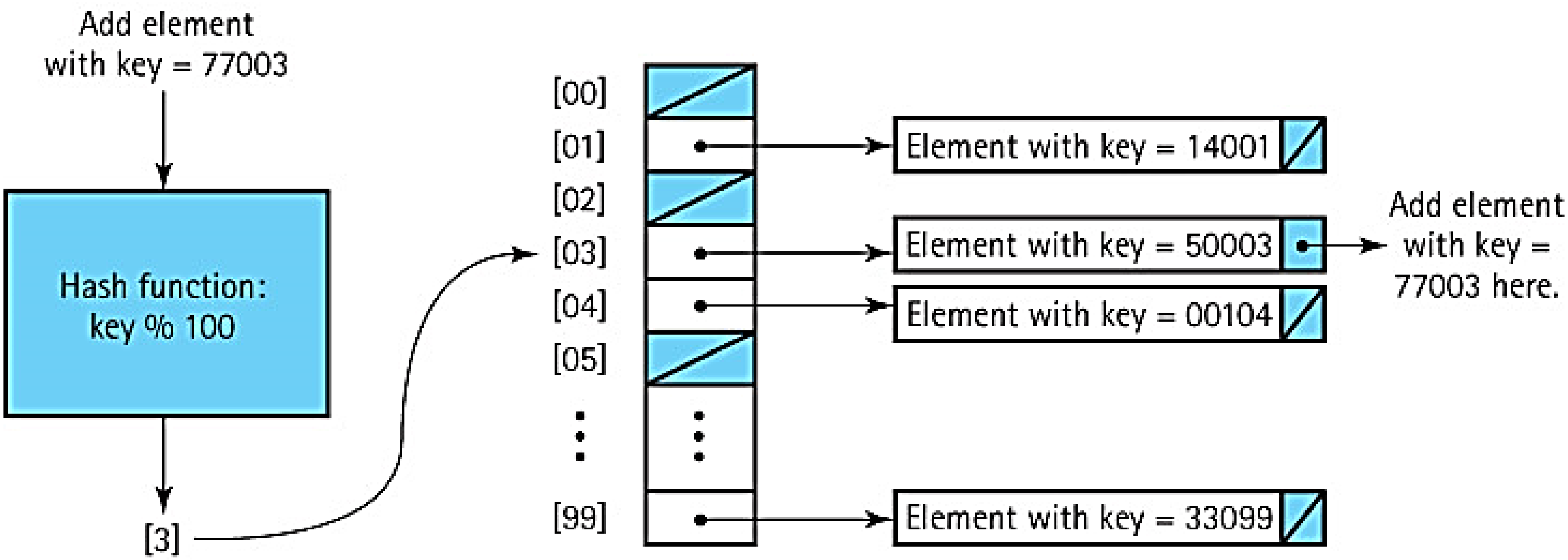
- **Bucket** - A collection of elements associated with a particular hash location.
- **Chain** - A linked list of elements that share the same hash location.



Handling collisions by hashing with buckets

Buckets and chaining

- **Bucket** - A collection of elements associated with a particular hash location.
- **Chain** - A linked list of elements that share the same hash location.



Handling collisions by hashing with chaining

Action items

- Read book chapter 10.