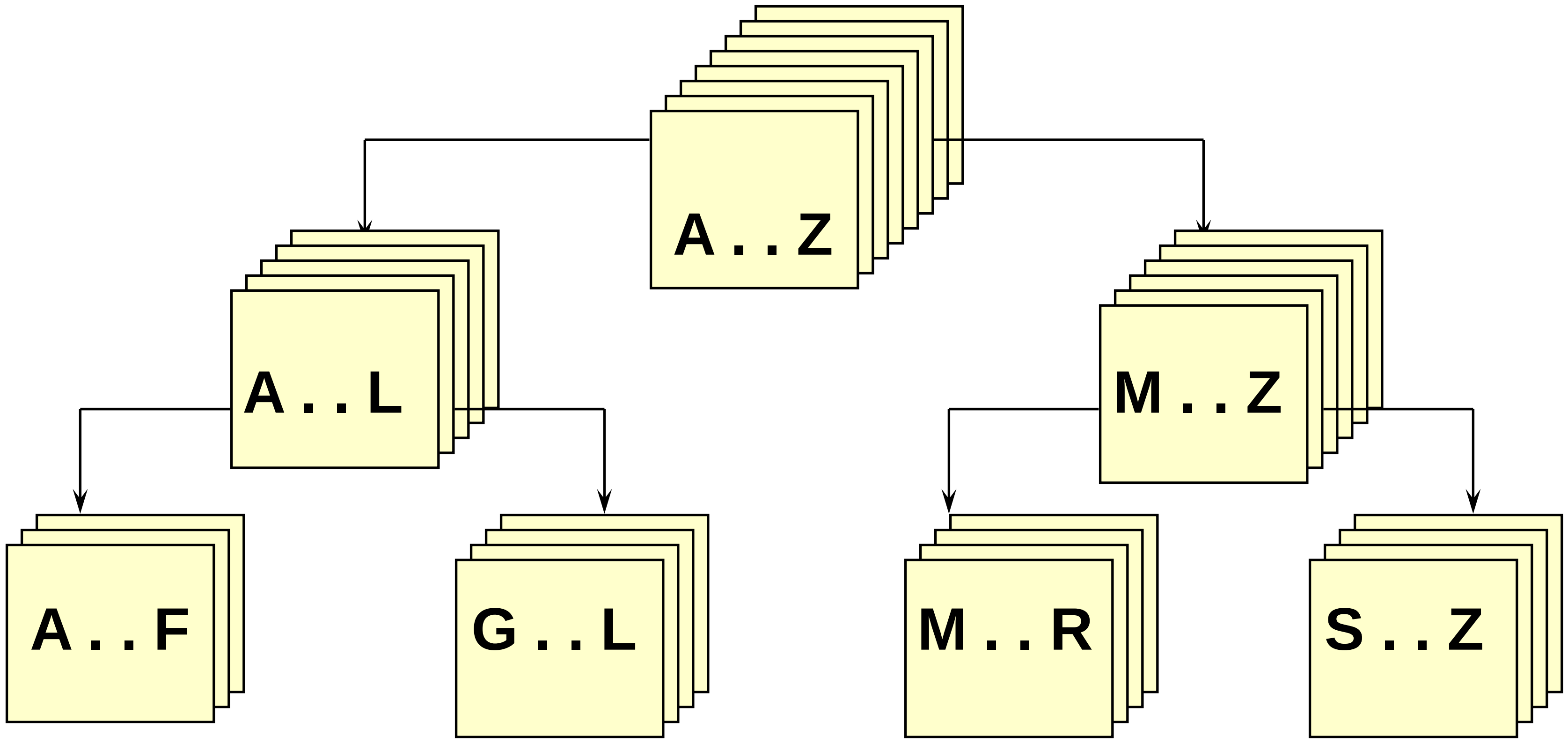
Using quick sort algorithm



```
// Recursive quick sort algorithm

template <class  ItemType >
void  QuickSort  ( ItemType  values[ ] ,  int  first ,
   int  last )

//  Pre:   first <= last
//  Post: Sorts array values[ first .  . last ] into
   ascending order
{
   if  ( first < last )                    //  general case
   {
      int  splitPoint ;
       Split ( values, first, last, splitPoint ) ;
       // values [first]..values[splitPoint - 1] <= splitVal
      // values  [splitPoint] = splitVal
      // values [splitPoint + 1]..values[last] > splitVal
       QuickSort(values,  first,  splitPoint - 1);
       QuickSort(values,  splitPoint + 1,  last);
   }
} ;
```

Before call to function Split

splitVal = 9

GOAL: place splitVal in its proper position with all values less than or equal to splitVal on its left and all larger values on its right

9	*20*	*6*	*18*	*14*	*3*	*60*	*11*

values[first] **[last]**

After call to function Split

splitVal = 9

smaller values in left part

larger values in right part

6	*3*	*9*	*18*	*14*	*20*	*60*	*11*

values[first] **[last]**

splitVal in correct position

Quick Sort of N elements: How many comparisons?

N	**For first call, when each of N elements is compared to the split value**
2 * N/2	**For the next pair of calls, when N/2 elements in each "half" of the original array are compared to their own split values.**
4 * N/4	**For the four calls when N/4 elements in each "quarter" of original array are compared to their own split values.**

.

.

. **HOW MANY SPLITS CAN OCCUR?**

Quick Sort of N elements: How many splits can occur?

It depends on the order of the original array elements!

If each split divides the subarray approximately in half, there will be only $\log_2 N$ splits, and QuickSort is $O(N*\log_2 N)$.

But, if the original array was sorted to begin with, the recursive calls will split up the array into parts of unequal length, with one part empty, and the other part containing all the rest of the array except for split value itself. In this case, there can be as many as N-1 splits, and QuickSort is $O(N^2)$.

Before call to function Split

splitVal = 9

GOAL: place splitVal in its proper position with all values less than or equal to splitVal on its left and all larger values on its right

9	*20*	*26*	*18*	*14*	*53*	*60*	*11*

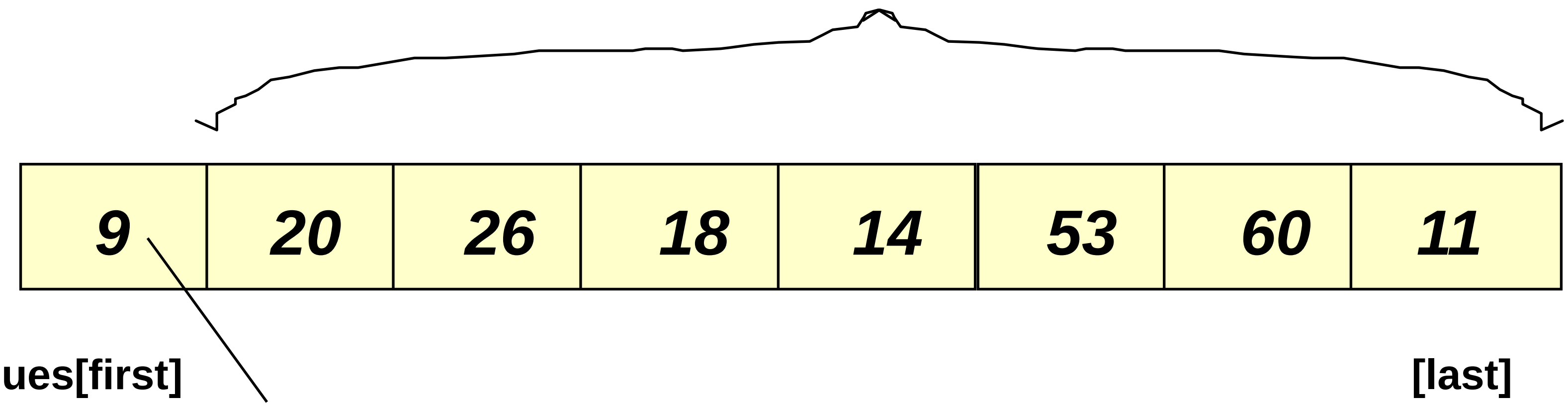
values[first] **[last]**

After call to function Split

splitVal = 9

no smaller values
empty left part

larger values
in right part with N-1 elements



values[first] **[last]**

splitVal in correct position

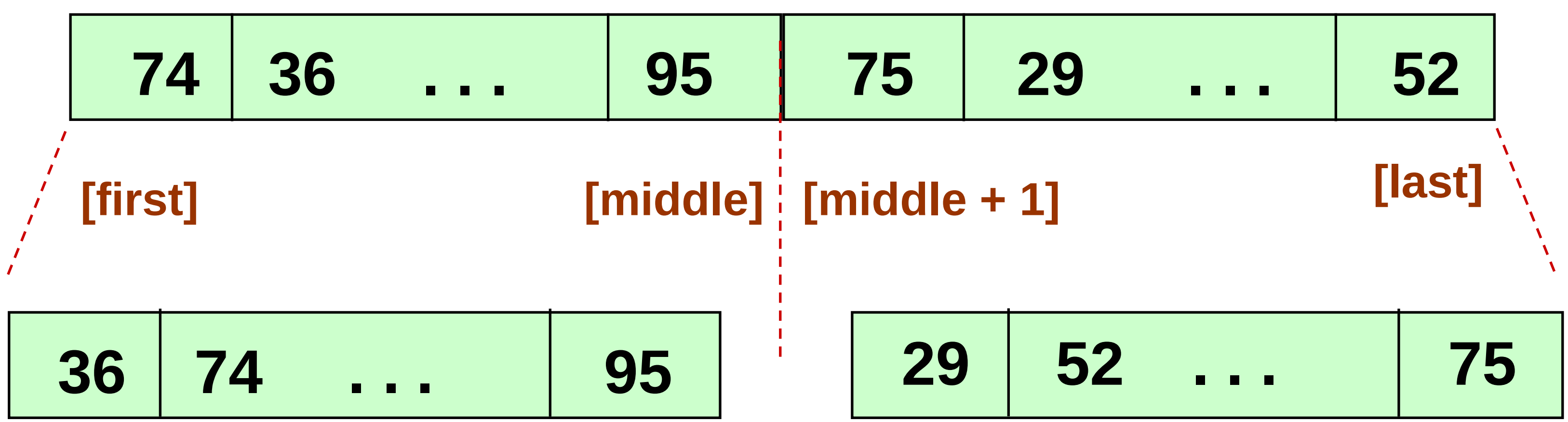
Merge Sort Algorithm

Cut the array in half.

Sort the left half.

Sort the right half.

Merge the two sorted halves into one sorted array.



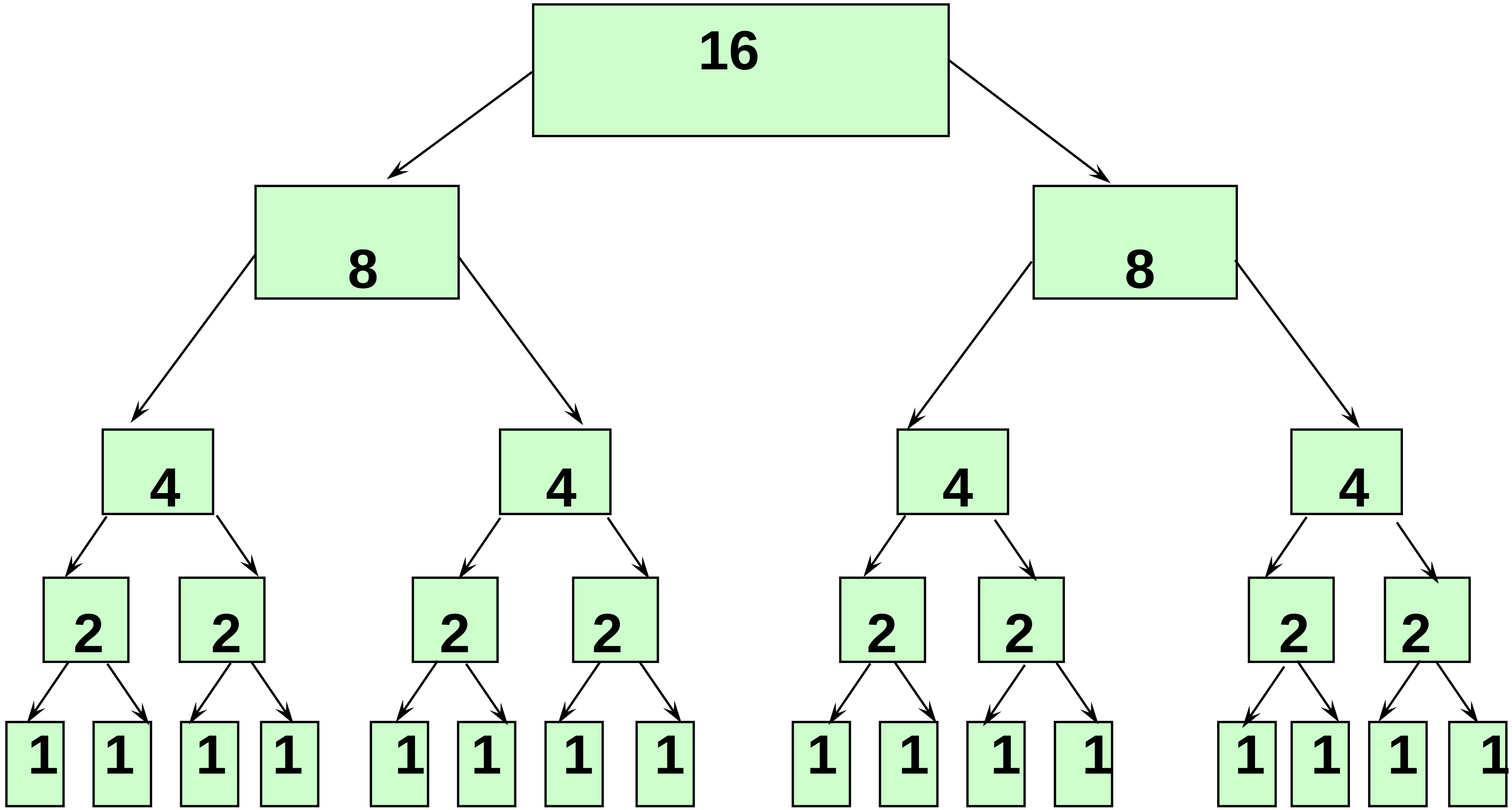
```
// Recursive merge sort algorithm

template <class  ItemType >
void  MergeSort  ( ItemType  values[ ] ,  int  first ,
   int  last )
//  Pre:   first <= last
//  Post: Array values[first..last] sorted into
//     ascending order.
{
   if  ( first < last )                    //  general case
   {
       int  middle = ( first  +  last ) / 2  ;
        MergeSort ( values, first, middle ) ;
        MergeSort( values,  middle + 1, last ) ;

        // now  merge two subarrays
        // values  [ first . . . middle ] with
        // values [ middle + 1,  . . . last ].

        Merge(values,  first, middle, middle + 1, last);
   }
}
```

Using Merge Sort Algorithm with N = 16



Merge Sort of N elements: How many comparisons?

The entire array can be subdivided into halves only $\log_2 N$ times.

Each time it is subdivided, function Merge is called to re-combine the halves. Function Merge uses a temporary array to store the merged elements. Merging is O(N) because it compares each element in the subarrays.

Copying elements back from the temporary array to the values array is also O(N).

MERGE SORT IS $O(N*\log_2 N)$.

Comparison of Sorting Algorithms

	Order of Magnitude		
Sort	Best Case	Average Case	Worst Case
`selectionSort`	$O(N^2)$	$O(N^2)$	$O(N^2)$
`bubbleSort`	$O(N^2)$	$O(N^2)$	$O(N^2)$
`shortBubble`	$O(N)$ (*)	$O(N^2)$	$O(N^2)$
`insertionSort`	$O(N)$ (*)	$O(N^2)$	$O(N^2)$
`mergeSort`	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$
`quickSort`	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N^2)$ (depends on split)
`heapSort`	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$

*Data almost sorted.

Testing

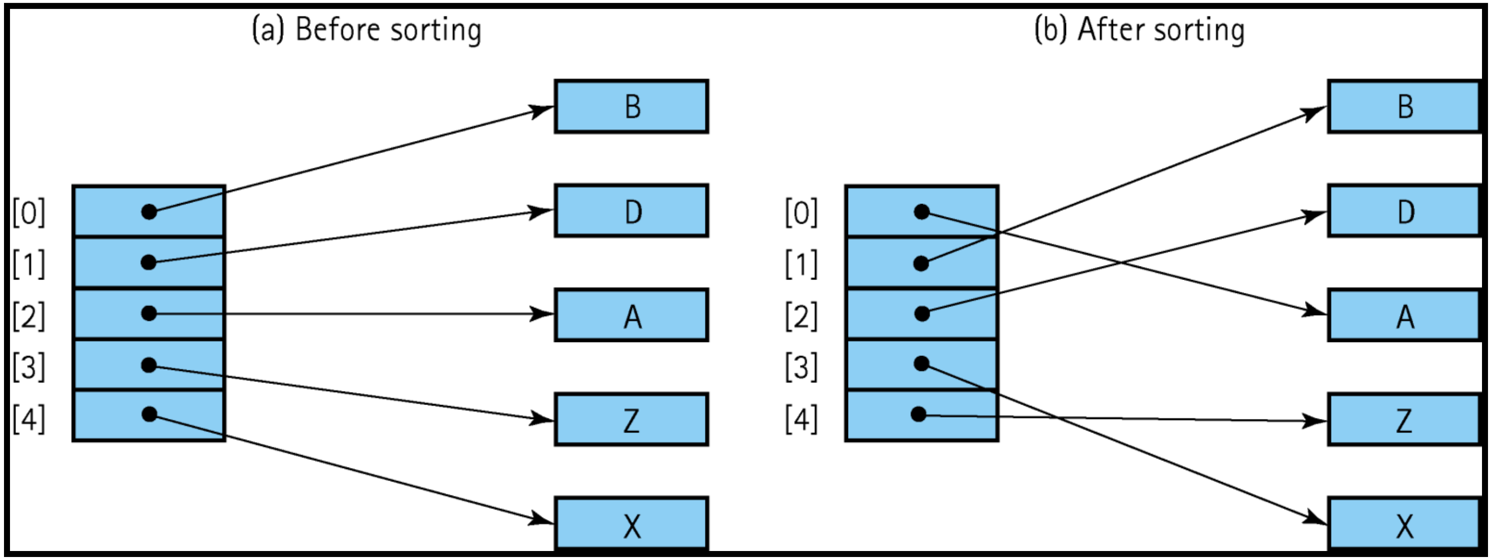
To thoroughly test our sorting methods we should vary the size of the array they are sorting

Vary the original order of the array-test

- Reverse order
- Almost sorted
- All identical elements

Sorting Objects

When sorting an array of objects we are manipulating references to the object, and not the objects themselves



Stability

Stable Sort: A sorting algorithm that preserves the order of duplicates

Of the sorts that we have discussed in this book, only `heapSort` and `quickSort` are inherently unstable

Searching

Linear (or Sequential) Searching

Beginning with the first element in the list, we search for the desired element by examining each subsequent item's key

High-Probability Ordering

Put the most-often-desired elements at the beginning of the list

Self-organizing or *self-adjusting* lists

Key Ordering

Stop searching before the list is exhausted if the element does not exist

Function `BinarySearch( )`

`BinarySearch` takes sorted array `info`, and two subscripts, `fromLoc` and `toLoc`, and `item` as arguments. It returns false if `item` is not found in the elements `info[fromLoc…toLoc]`. Otherwise, it returns true.

`BinarySearch` is $O(\log_2 N)$.

```
found = BinarySearch(info, 25, 0, 14 );
```

item fromLoc toLoc

indexes	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
info	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28
									16	18	20	22	24	26	28
													24	26	28
													24		

NOTE: denotes element examined

```
template<class  ItemType>
bool  BinarySearch(ItemType  info[ ], ItemType  item,
                   int   fromLoc ,    int  toLoc )
   // Pre:  info [ fromLoc . . toLoc ] sorted in ascending order
   // Post: Function value =  ( item  in info[fromLoc .. toLoc])
{
   int  mid ;
   if  ( fromLoc > toLoc ) //  base case -- not found
      return  false ;
   else
   {
     mid = ( fromLoc + toLoc ) / 2 ;
     if ( info[mid] == item )      // base case-- found at mid
        return  true  ;
     else
       if ( item < info[mid])     //  search lower half
          return BinarySearch( info, item, fromLoc, mid-1 );
     else                         // search upper half
        return  BinarySearch( info, item, mid + 1, toLoc );
   }

}
```

Hashing

is a means used to order and access elements in a list quickly -- the goal is O(1) time -- by using a function of the key value to identify its location in the list.

The function of the key value is called a hash function.

FOR EXAMPLE . . .

Using a hash function

	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

HandyParts company makes no more than 100 different parts. But the parts all have four digit numbers.

This hash function can be used to store and retrieve parts in an array.

Hash(key) = partNum % 100

Placing Elements in the Array

	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Use the hash function

Hash(key) = partNum % 100

to place the element with

part number 5502 in the

array.

Placing Elements in the Array

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Next place part number 6702 in the array.

Hash(key) = partNum % 100

6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS

the condition resulting when two or more keys produce the same hash location

How to Resolve the Collision?

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

One way is by linear probing. This uses the rehash function

(HashValue + 1) % 100

repeatedly until an empty location is found for part number 6702.

Resolving the Collision

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Still looking for a place for 6702 using the function

(HashValue + 1) % 100

Collision Resolved

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Part 6702 can be placed at the location with index 4.

Collision Resolved

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	
[4]	7803
. . .	6702 . . .
[97]	Empty
[98]	2298
[99]	3699

Part 6702 is placed at the location with index 4.

Where would the part with number 4598 be placed using linear probing?

Deletion with Linear Probing

Order of Insertion:

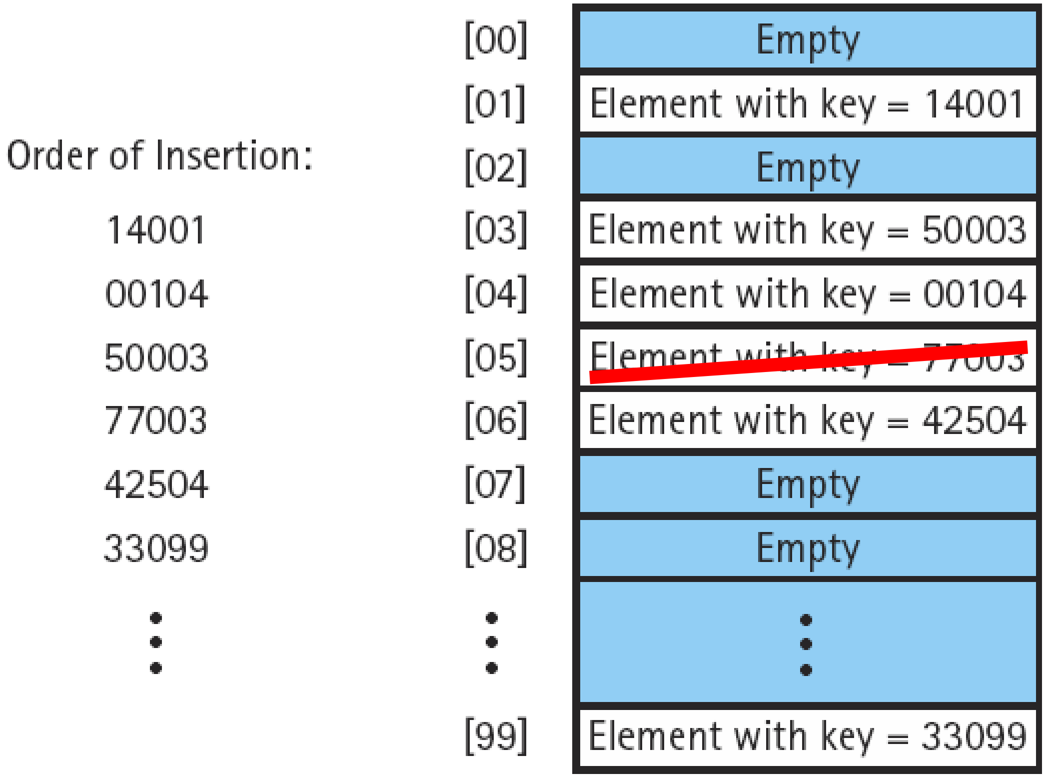
14001
00104
50003
77003
42504
33099
⋮

[00]	Empty
[01]	Element with key = 14001
[02]	Empty
[03]	Element with key = 50003
[04]	Element with key = 00104
[05]	Element with key = 77003
[06]	Element with key = 42504
[07]	Empty
[08]	Empty
⋮	⋮
[99]	Element with key = 33099

What happens if we perform

- **first, delete the element with 77003**
- **then, search for the element with 42504**

Deletion with Linear Probing



set this slot to ***Deleted*** rather than *Empty*

We cannot find the element with 42504 if we set the deleted slot to *Empty*

Resolving Collisions: Rehashing

Resolving a collision by computing a new hash location from a hash function that manipulates the original location rather than the element's key

Linear probing

(*HashValue* + 1) % 100

(*HashValue* + *constant*) % *array-size*

quadratic probing

(*HashValue* $\pm$ I^2) % *array-size*

random probing

(*HashValue* + *random-number*) % *array-size*

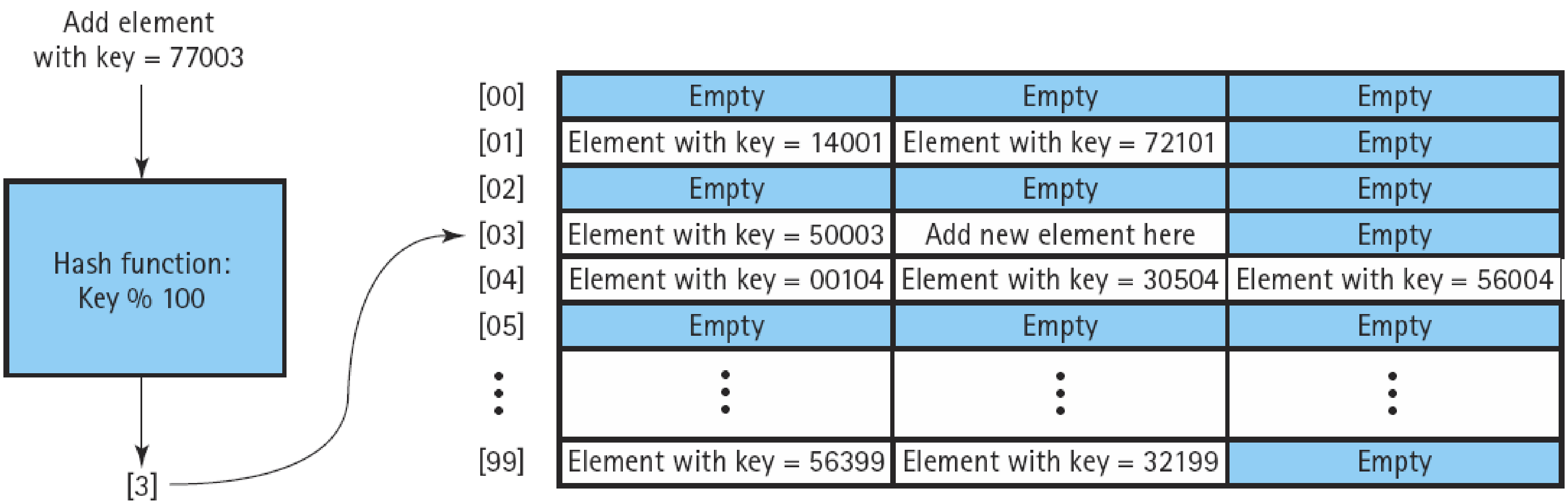
Resolving Collisions: Buckets and Chaining

The main idea is to allow multiple element keys to hash to the same location

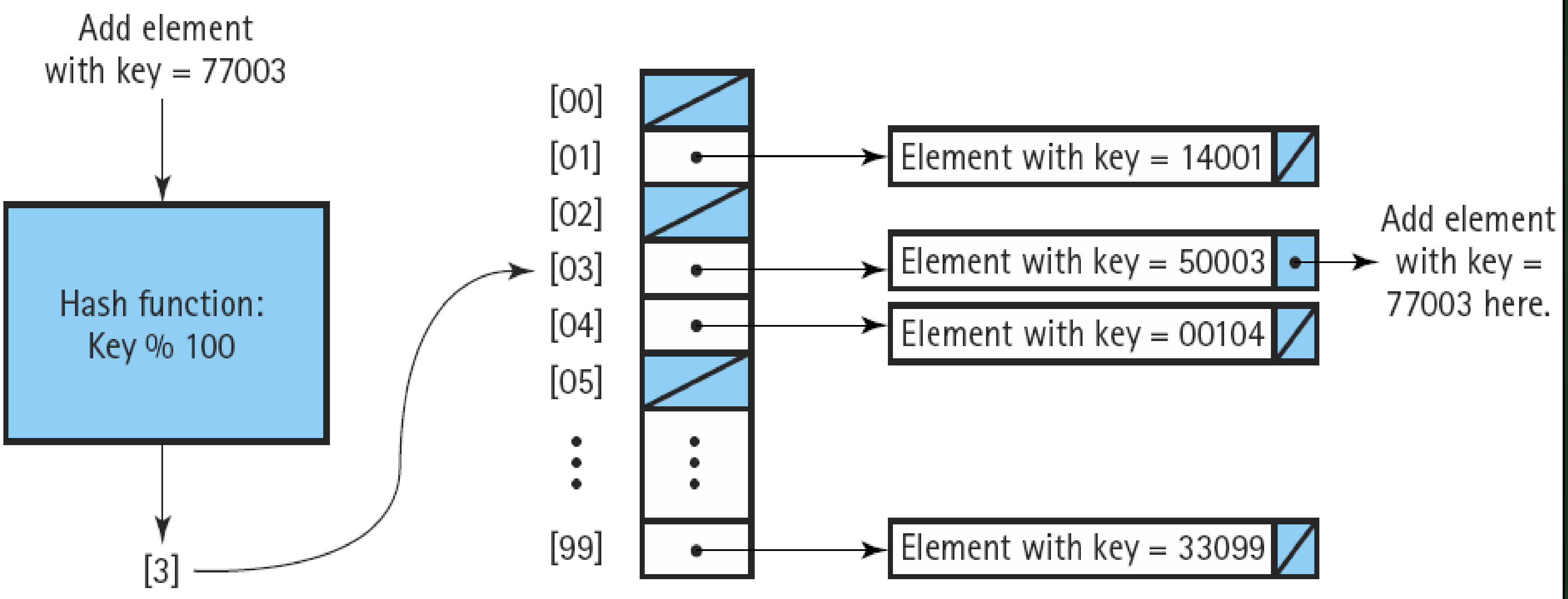
Bucket **A collection of elements associated with a particular hash location**

Chain **A linked list of elements that share the same hash location**

Resolving Collisions: Buckets



Resolving Collisions: Chain



Choosing a Good Hash Functions

- Two ways to minimize collisions are
  - Increase the range of the hash function
  - Distribute elements as uniformly as possible throughout the hash table
- How to choose a good hash function
  - Utilize knowledge about statistical distribution of keys
  - Select appropriate hash functions
    - division method
    - sum of characters
    - folding
    - ...

Radix Sort

Radix sort

- Is *not* a comparison sort

- Uses a radix-length array of queues of records

- Makes use of the values in digit positions in the keys to select the queue into which a record must be enqueued

Original Array

762
124
432
761
800
402
976
100
001
999

Queues After First Pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
800	761	762		124		976			999
100	001	432							
		402							

Array After First Pass

800
100
761
001
762
432
402
124
976
999

Queues After Second Pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
800		124	432			761	976		999
100						762			
001									
402									

Array After Second Pass

800
100
001
402
124
432
761
762
976
999

Queues After Third Pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
001	100			402			761	800	976
	124			432			762		999

Array After Third Pass

001
100
124
402
432
761
762
800
976
999