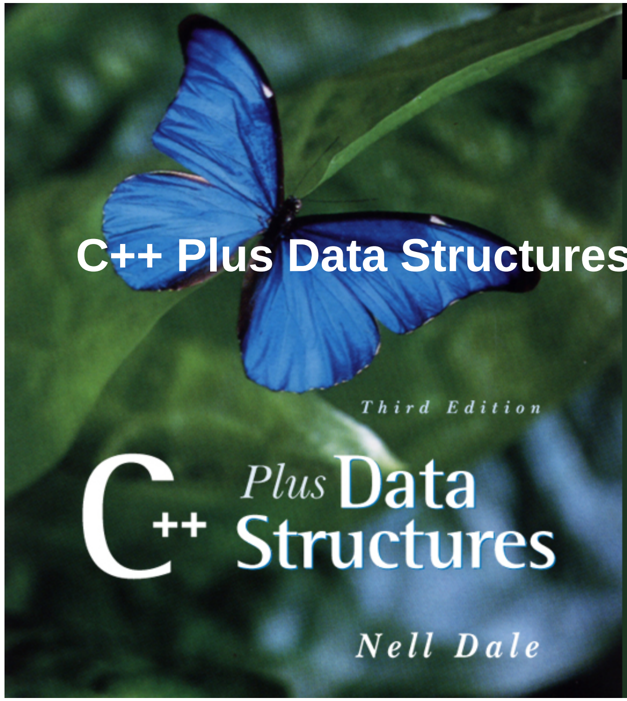


C++ Plus Data Structures

Chapter

10
Sorting and Searching Algorithms

Sorting means . . .

- The values stored in an array have keys of a type for which the relational operators are defined. (We also assume unique keys.)
- Sorting rearranges the elements into either ascending or descending order within the array. (We'll use ascending order.)

Straight Selection Sort

values	
[0]	36
[1]	24
[2]	10
[3]	6
[4]	12

Divides the array into two parts: already sorted, and not yet sorted.

On each pass, finds the smallest of the unsorted elements, and swaps it into its correct place, thereby increasing the number of sorted elements by one.

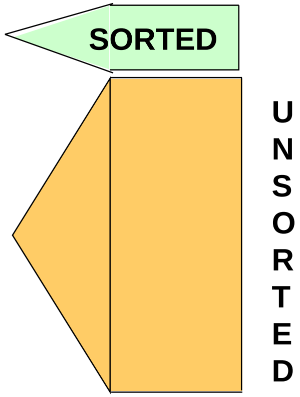
Selection Sort: Pass One

values	
[0]	36
[1]	24
[2]	10
[3]	6
[4]	12

UNSORTED

Selection Sort: End Pass One

values	[0]	6
	[1]	24
	[2]	10
	[3]	36
	[4]	12

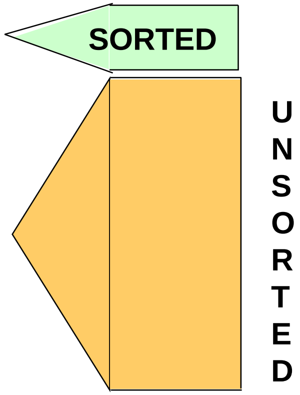


SORTED

UNSORTED

Selection Sort: Pass Two

values	[0]	6
	[1]	24
	[2]	10
	[3]	36
	[4]	12

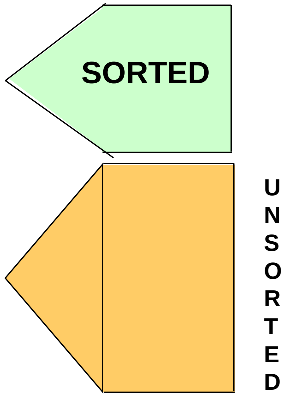


SORTED

UNSORTED

Selection Sort: End Pass Two

values	[0]	6
	[1]	10
	[2]	24
	[3]	36
	[4]	12

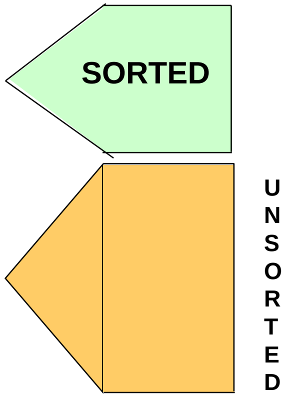


SORTED

UNSORTED

Selection Sort: Pass Three

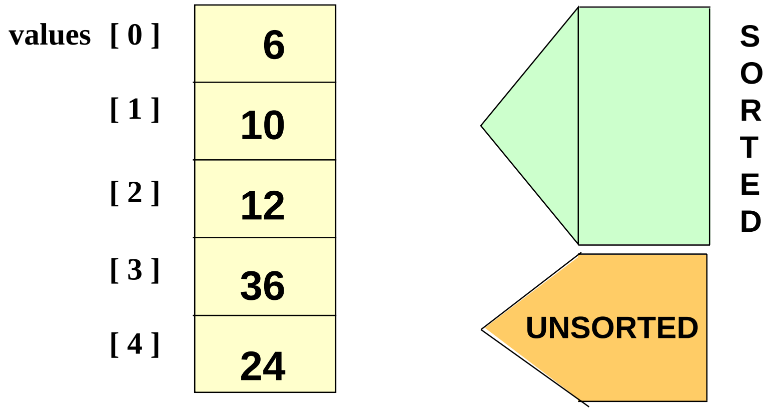
values	[0]	6
	[1]	10
	[2]	24
	[3]	36
	[4]	12



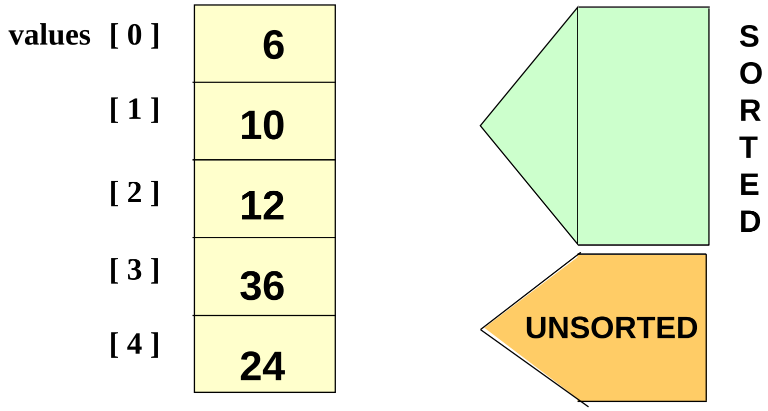
SORTED

UNSORTED

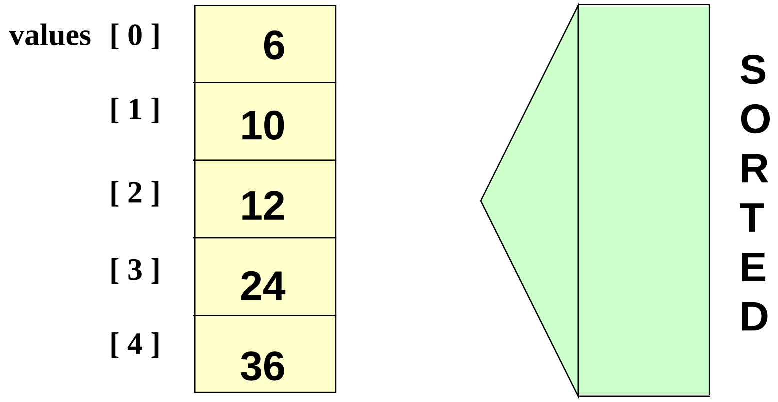
Selection Sort: End Pass Three



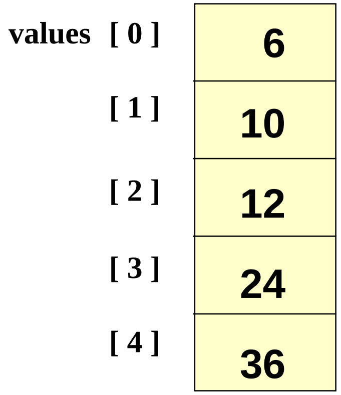
Selection Sort: Pass Four



Selection Sort: End Pass Four



Selection Sort:
How many comparisons?



4 compares for values[0]

3 compares for values[1]

2 compares for values[2]

1 compare for values[3]

= 4 + 3 + 2 + 1

For selection sort in general

- The number of comparisons when the array contains N elements is

Sum = (N-1) + (N-2) + . . . + 2 + 1

Notice that . . .

Sum = (N-1) + (N-2) + . . . + 2 + 1

+ Sum = 1 + 2 + . . . + (N-2) + (N-1)

$$2^{*}\ \text{Sum} = \underbrace{N + N + \ldots + N + N}$$

2 * Sum = N * (N-1)

$$\boxed{\text{Sum} = \frac{N * (N-1)}{2}}$$

For selection sort in general

- The number of comparisons when the array contains N elements is

Sum = (N-1) + (N-2) + . . . + 2 + 1

Sum = N * (N-1) /2

Sum = .5 N^2 - .5 N

Sum = O(N^2)

```
template <class  ItemType >
int  MinIndex(ItemType values [ ], int  start, int end)
//  Post: Function value = index of the smallest value
//  in values [start]  . . values [end].
{
   int  indexOfMin = start ;

   for(int index = start + 1 ; index <= end ; index++)
     if  (values[ index] < values [indexOfMin])
         indexOfMin = index ;

   return   indexOfMin;

}
```

```
template <class  ItemType >
void  SelectionSort (ItemType values[ ],
  int  numValues )
// Post: Sorts array values[0 . . numValues-1 ]
// into ascending order by key
{
  int  endIndex = numValues - 1 ;
  for (int current = 0 ; current < endIndex;
    current++)
    Swap (values[current],
      values[MinIndex(values,current, endIndex)]);
}
```

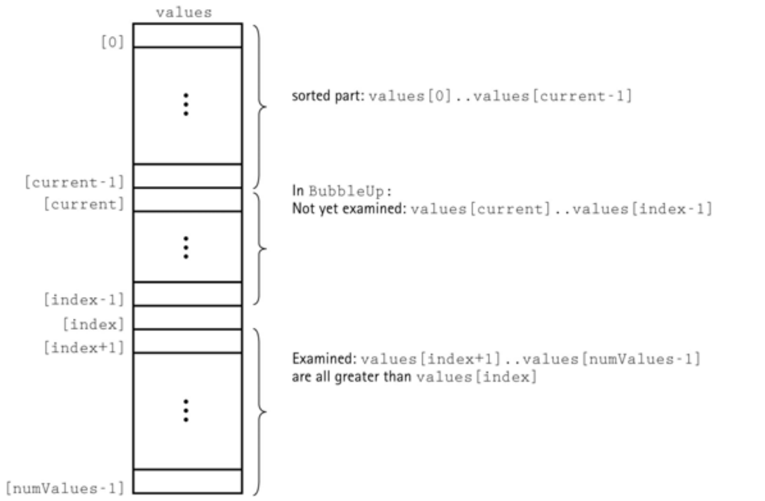
Bubble Sort

values	
[0]	36
[1]	24
[2]	10
[3]	6
[4]	12

Compares neighboring pairs of array elements, starting with the last array element, and swaps neighbors whenever they are not in correct order.

On each pass, this causes the smallest element to "bubble up" to its correct place in the array.

Snapshot of BubbleSort


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

Code for BubbleSort

```
template<class ItemType>
void BubbleSort(ItemType values[],
  int numValues)
{
  int current = 0;
  while (current < numValues - 1)
  {
    BubbleUp(values, current, numValues-1);
    current++;
  }
}
```

Code for BubbleUp

```
template<class ItemType>
void BubbleUp(ItemType values[],
  int startIndex, int endIndex)
// Post: Adjacent pairs that are out of
//   order have been switched between
//   values[startIndex]..values[endIndex]
//   beginning at values[endIndex].

{
  for (int index = endIndex;
    index > startIndex; index--)
    if (values[index] < values[index-1])
      Swap(values[index], values[index-1]);
}
```

Observations on BubbleSort

This algorithm is *always* O(N^2).

There can be a large number of intermediate swaps.

Can this algorithm be improved?

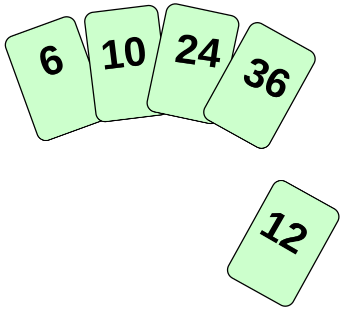
Insertion Sort

values	
[0]	36
[1]	24
[2]	10
[3]	6
[4]	12

One by one, each as yet unsorted array element is inserted into its proper place with respect to the already sorted elements.

On each pass, this causes the number of already sorted elements to increase by one.

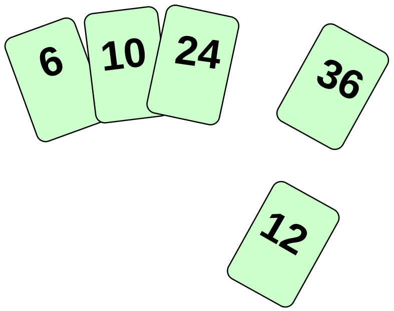
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

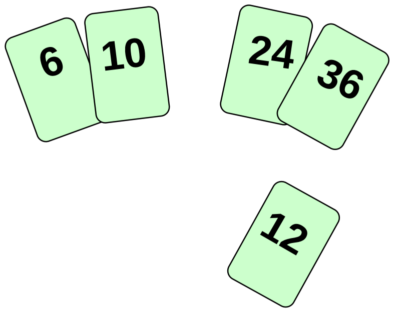
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

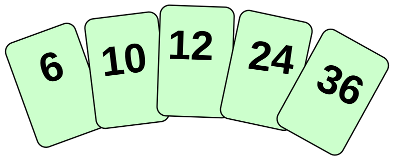
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

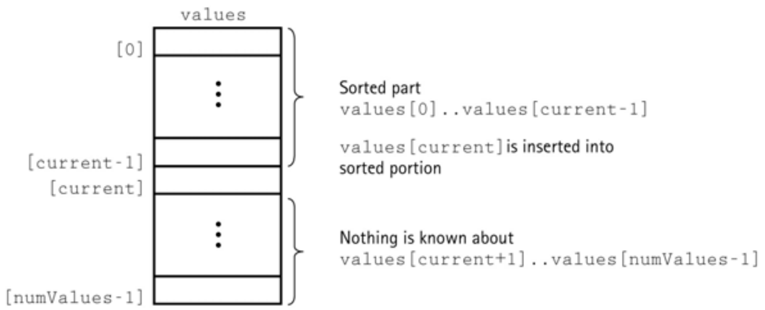
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

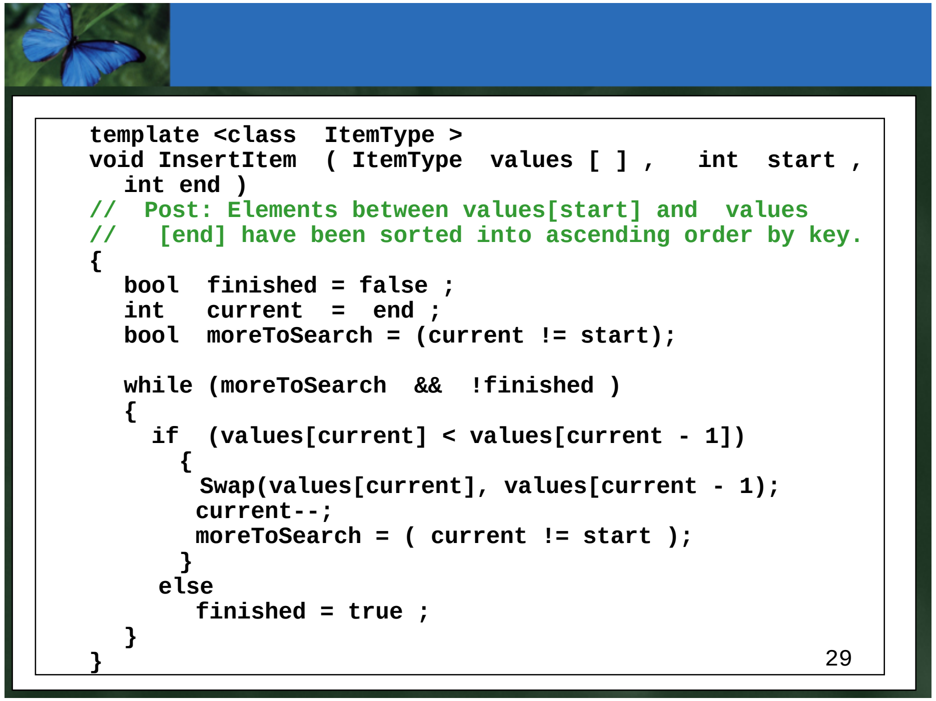
A Snapshot of the Insertion Sort Algorithm



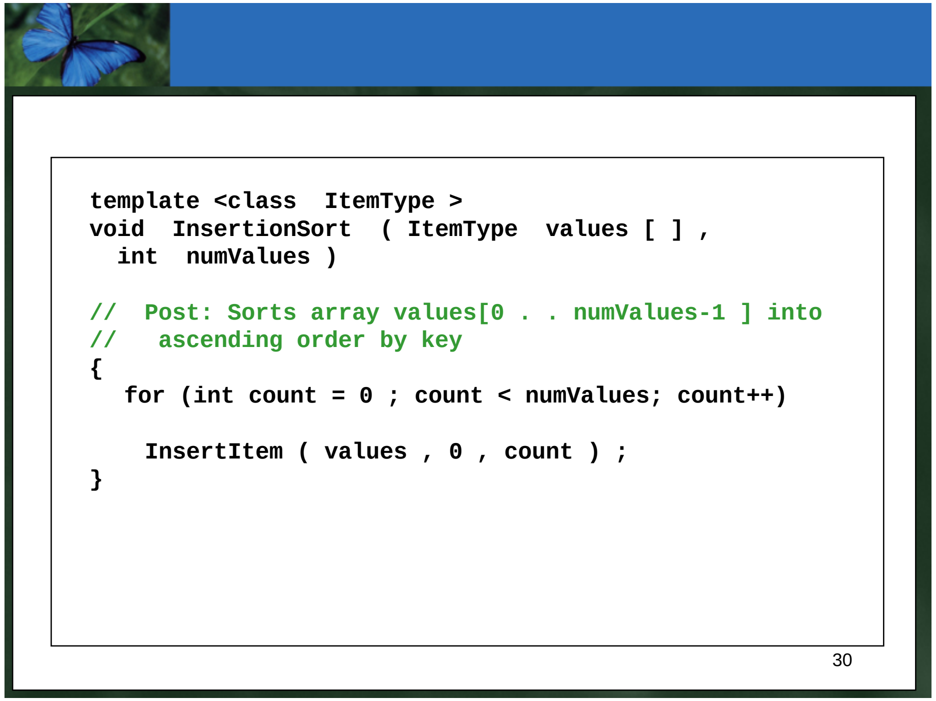
Sorted part
values[0]..values[current-1]

values[current] is inserted into sorted portion

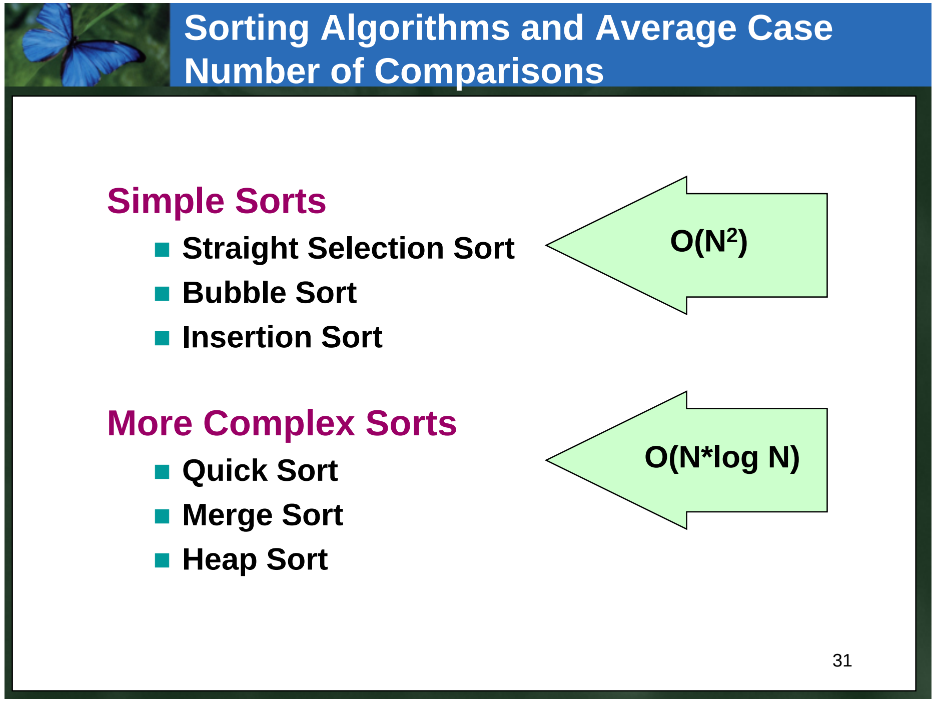
Nothing is known about
values[current+1]..values[numValues-1]



```
template <class  ItemType >
void InsertItem  ( ItemType  values [ ] ,   int  start ,
   int end )
//  Post: Elements between values[start] and  values
//   [end] have been sorted into ascending order by key.
{
   bool  finished = false ;
   int   current  =  end ;
   bool  moreToSearch = (current != start);

   while (moreToSearch  &&  !finished )
   {
     if  (values[current] < values[current - 1])
       {
         Swap(values[current], values[current - 1]);
         current--;
         moreToSearch = ( current != start );
       }
      else
         finished = true ;
   }
}
```



```
template <class  ItemType >
void  InsertionSort  ( ItemType  values [ ] ,
  int  numValues )

//  Post: Sorts array values[0 . . numValues-1 ] into
//   ascending order by key
{
   for (int count = 0 ; count < numValues; count++)

     InsertItem ( values , 0 , count ) ;
}
```



Sorting Algorithms and Average Case Number of Comparisons

Simple Sorts

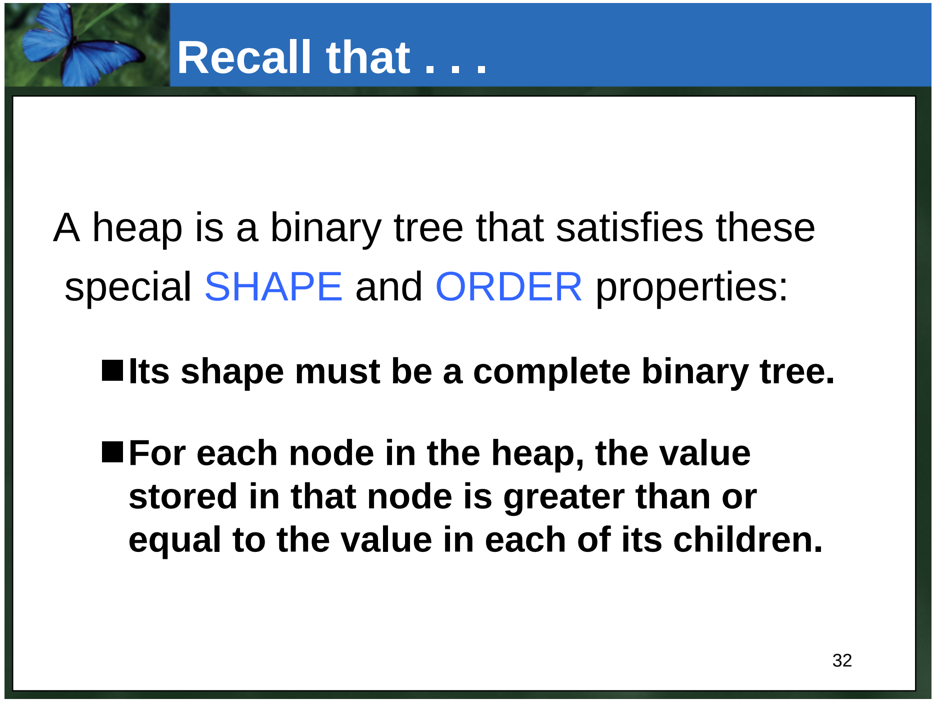
- Straight Selection Sort
- Bubble Sort
- Insertion Sort

O(N^2)

More Complex Sorts

- Quick Sort
- Merge Sort
- Heap Sort

O(N*log N)



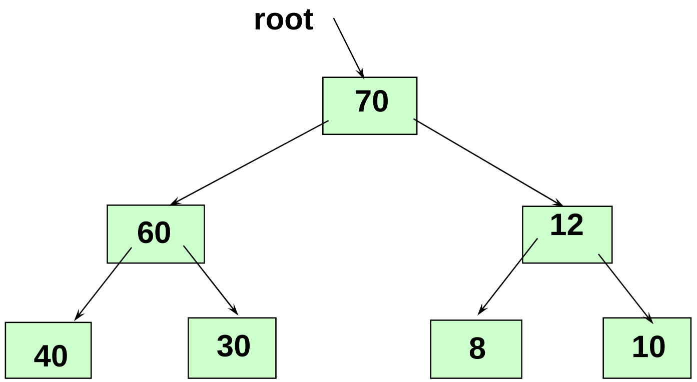
Recall that . . .

A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

- **Its shape must be a complete binary tree.**
- **For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.**

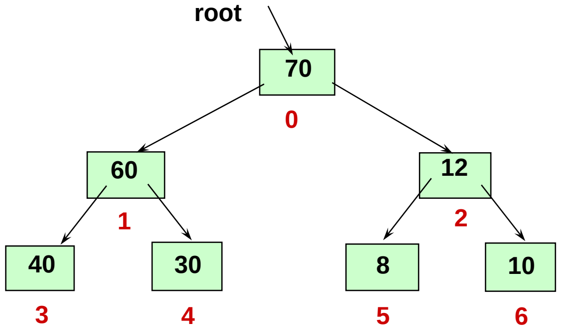
The largest element in a heap

is always found in the root node



The heap can be stored in an array

	values
[0]	70
[1]	60
[2]	12
[3]	40
[4]	30
[5]	8
[6]	10



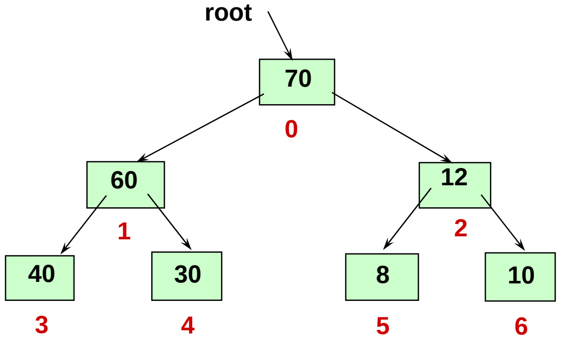
Heap Sort Approach

First, make the unsorted array into a heap by satisfying the order property. Then repeat the steps below until there are no more unsorted elements.

- Take the root (maximum) element off the heap by swapping it into its correct place in the array at the end of the unsorted elements.
- Reheap the remaining unsorted elements. (This puts the next-largest element into the root position).

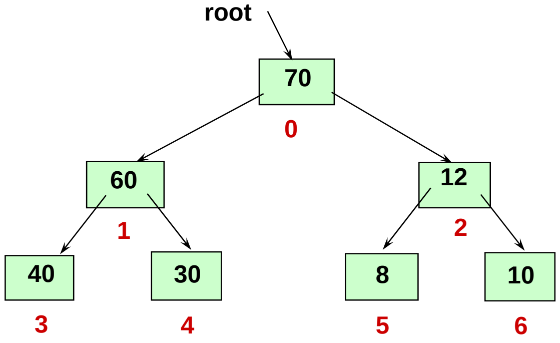
After creating the original heap

	values
[0]	70
[1]	60
[2]	12
[3]	40
[4]	30
[5]	8
[6]	10



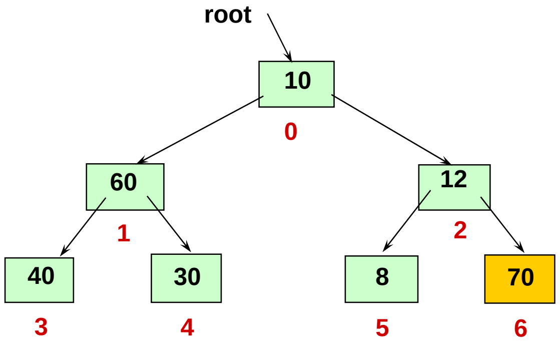
Swap root element into last place in unsorted array

	values
[0]	70
[1]	60
[2]	12
[3]	40
[4]	30
[5]	8
[6]	10



After swapping root element into it place

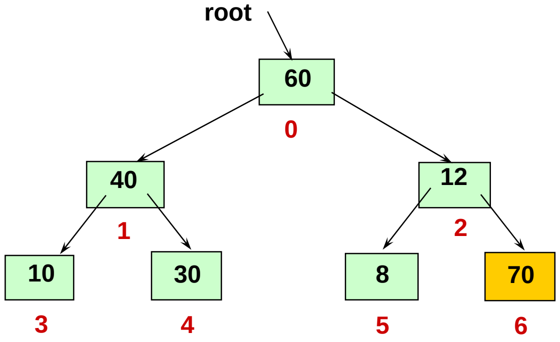
	values
[0]	10
[1]	60
[2]	12
[3]	40
[4]	30
[5]	8
[6]	70



NO NEED TO CONSIDER AGAIN

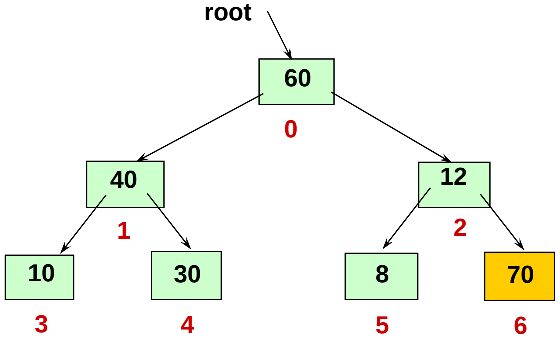
After reheaping remaining unsorted elements

	values
[0]	60
[1]	40
[2]	12
[3]	10
[4]	30
[5]	8
[6]	70



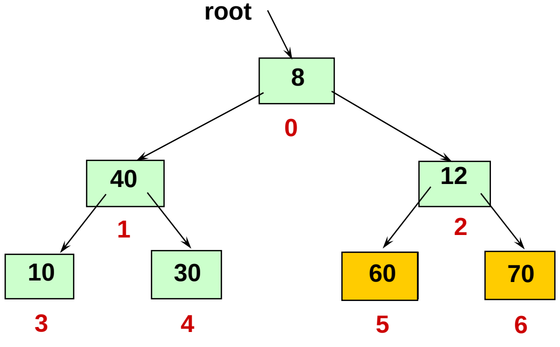
Swap root element into last place in unsorted array

	values
[0]	60
[1]	40
[2]	12
[3]	10
[4]	30
[5]	8
[6]	70



After swapping root element into its place

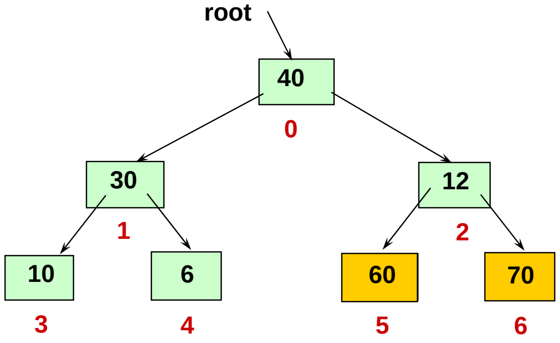
	values
[0]	8
[1]	40
[2]	12
[3]	10
[4]	30
[5]	60
[6]	70



NO NEED TO CONSIDER AGAIN

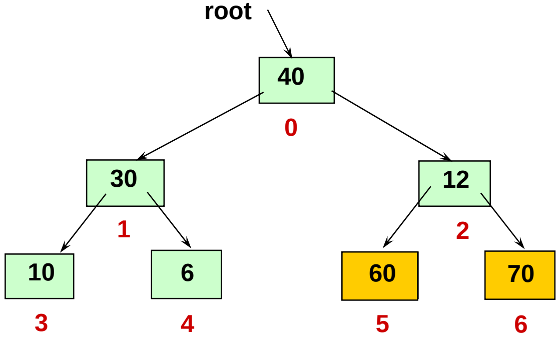
After reheaping remaining unsorted elements

	values
[0]	40
[1]	30
[2]	12
[3]	10
[4]	6
[5]	60
[6]	70



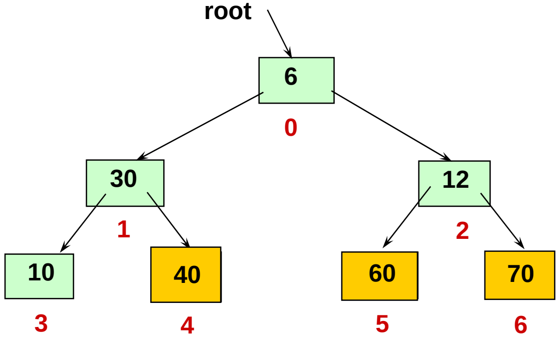
Swap root element into last place in unsorted array

	values
[0]	40
[1]	30
[2]	12
[3]	10
[4]	6
[5]	60
[6]	70



After swapping root element into its place

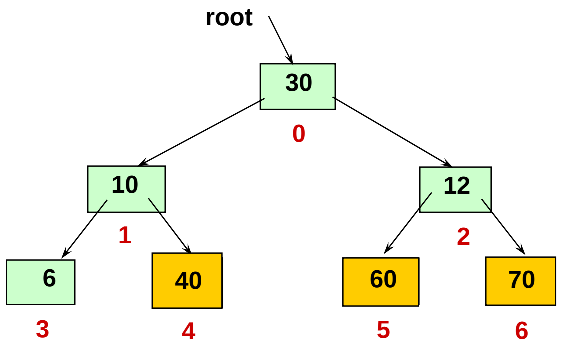
	values
[0]	6
[1]	30
[2]	12
[3]	10
[4]	40
[5]	60
[6]	70



NO NEED TO CONSIDER AGAIN

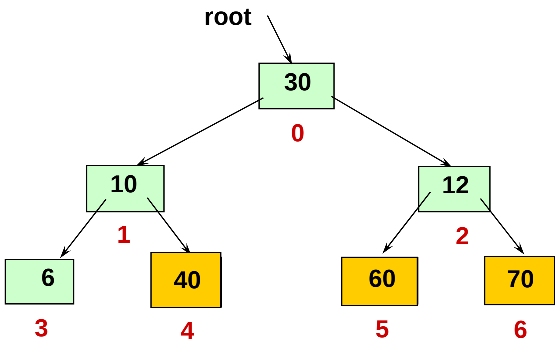
After reheaping remaining unsorted elements

	values
[0]	30
[1]	10
[2]	12
[3]	6
[4]	40
[5]	60
[6]	70



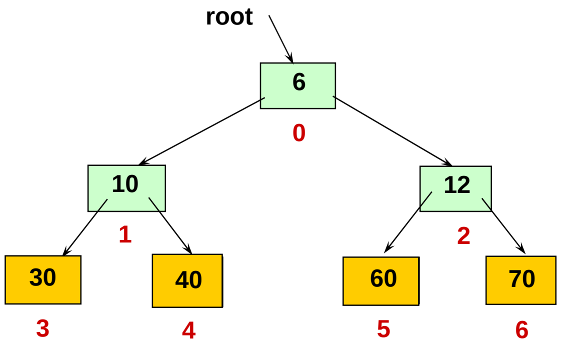
Swap root element into last place in unsorted array

	values
[0]	30
[1]	10
[2]	12
[3]	6
[4]	40
[5]	60
[6]	70



After swapping root element into its place

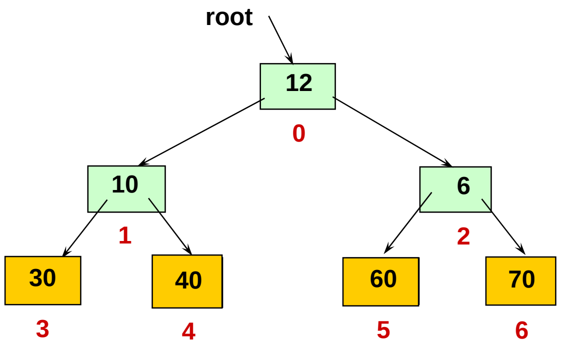
	values
[0]	6
[1]	10
[2]	12
[3]	30
[4]	40
[5]	60
[6]	70



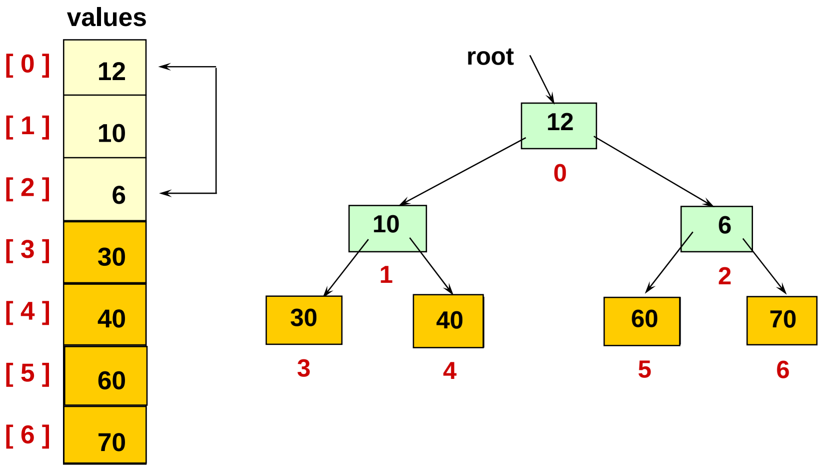
NO NEED TO CONSIDER AGAIN

After reheaping remaining unsorted elements

	values
[0]	12
[1]	10
[2]	6
[3]	30
[4]	40
[5]	60
[6]	70

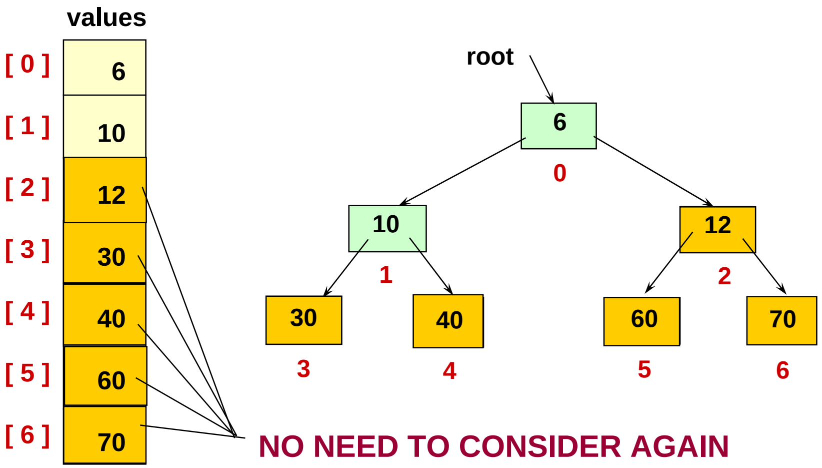


Swap root element into last place in unsorted array



	values
[0]	12
[1]	10
[2]	6
[3]	30
[4]	40
[5]	60
[6]	70

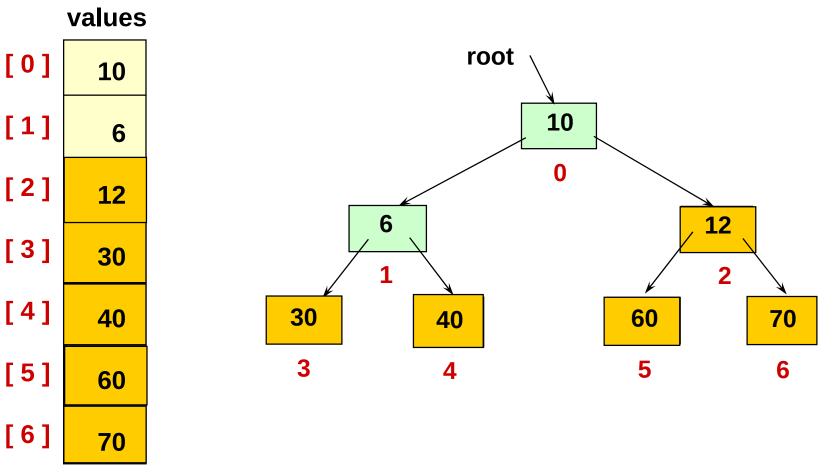
After swapping root element into its place



	values
[0]	6
[1]	10
[2]	12
[3]	30
[4]	40
[5]	60
[6]	70

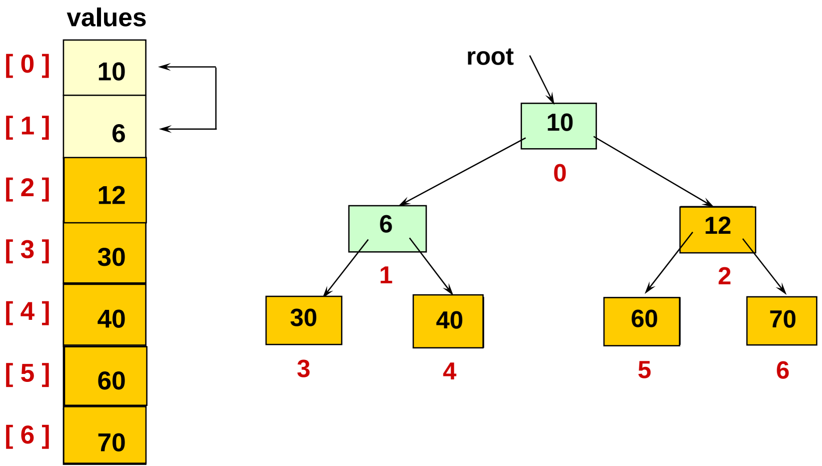
NO NEED TO CONSIDER AGAIN

After reheaping remaining unsorted elements



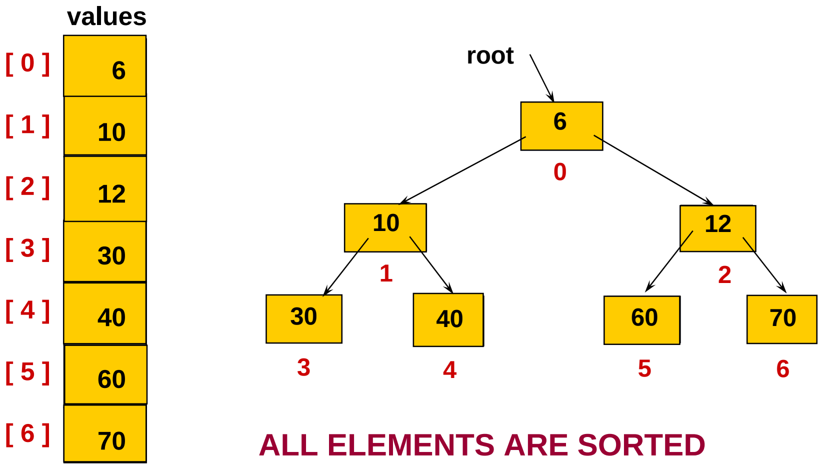
	values
[0]	10
[1]	6
[2]	12
[3]	30
[4]	40
[5]	60
[6]	70

Swap root element into last place in unsorted array



	values
[0]	10
[1]	6
[2]	12
[3]	30
[4]	40
[5]	60
[6]	70

After swapping root element into its place



ALL ELEMENTS ARE SORTED

```
template <class  ItemType >
void  HeapSort  ( ItemType  values [ ] ,   int
   numValues )
//  Post: Sorts array values[ 0 . . numValues-1 ] into
//   ascending order by key
{
   int  index ;

   // Convert array  values[0..numValues-1] into a heap
   for  (index = numValues/2 - 1;  index >= 0;  index--)
      ReheapDown ( values , index , numValues - 1 ) ;

   //  Sort the array.
   for (index = numValues - 1;  index >= 1;  index--)
   {
      Swap (values [0] , values[index]);
      ReheapDown (values , 0 , index - 1);
   }
}
```

ReheapDown

```
template< class  ItemType >
void  ReheapDown ( ItemType  values [ ],  int  root,
  int  bottom )

//  Pre:  root is the index of a node that may violate the
//    heap order property
//  Post:  Heap order property is restored between root and
//    bottom

{
    int  maxChild ;
    int  rightChild ;
    int  leftChild ;

    leftChild  =  root * 2 + 1 ;
    rightChild  =  root * 2 + 2 ;
```

```
   if (leftChild  <=  bottom)    // ReheapDown continued
   {
     if  (leftChild  ==  bottom)
       maxChild  = leftChild;
     else
     {
       if (values[leftChild] <=  values [rightChild])
         maxChild  =  rightChild ;
       else
          maxChild  =  leftChild ;
     }
     if  (values[ root ] < values[maxChild])
     {
       Swap (values[root], values[maxChild]);
       ReheapDown ( maxChild, bottom  ;
     }
   }
}
```

Building Heap From Unsorted Array

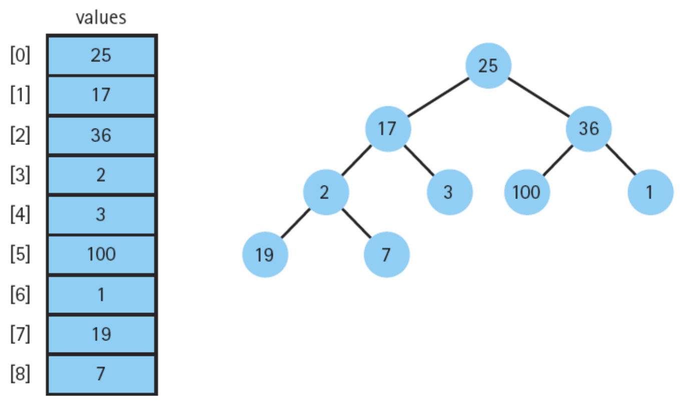
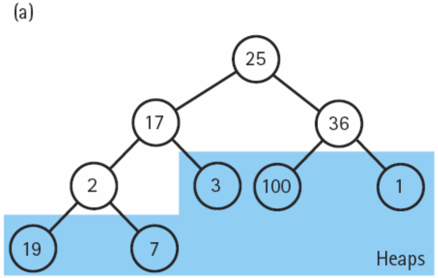


Figure 10.12 *An unsorted array and its tree*

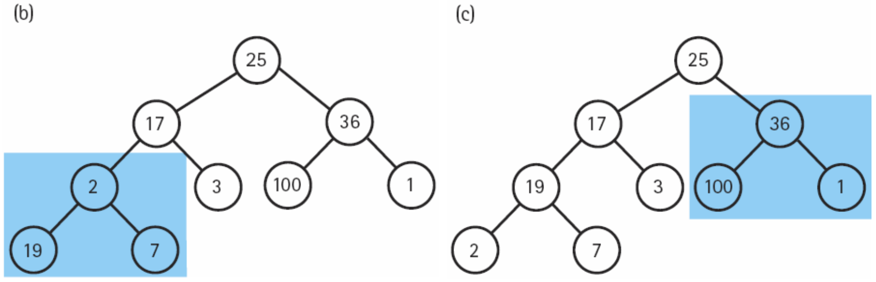
Building Heap From Unsorted Array (cont'd)

- Leaf nodes are already heaps



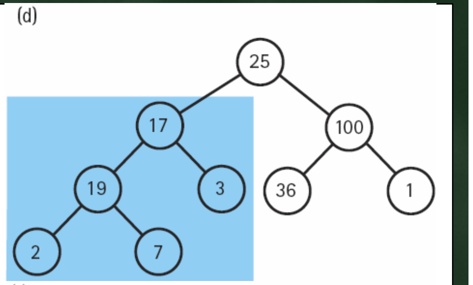
Building Heap From Unsorted Array (cont'd)

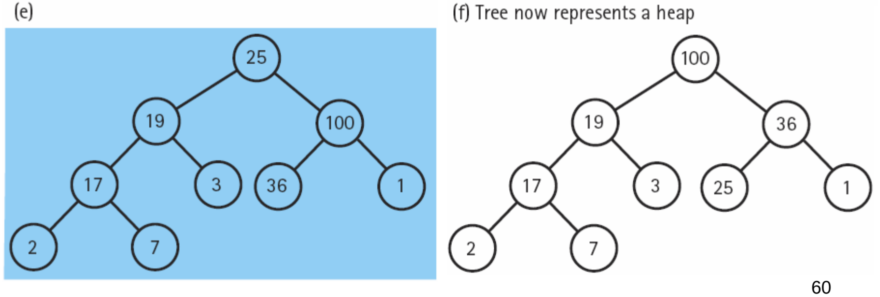
- The subtrees rooted at first nonleaf nodes are almost heaps



Building Heap From Unsorted Array (cont'd)

- Move up a level in the tree and continue reheaping until we reach the root node

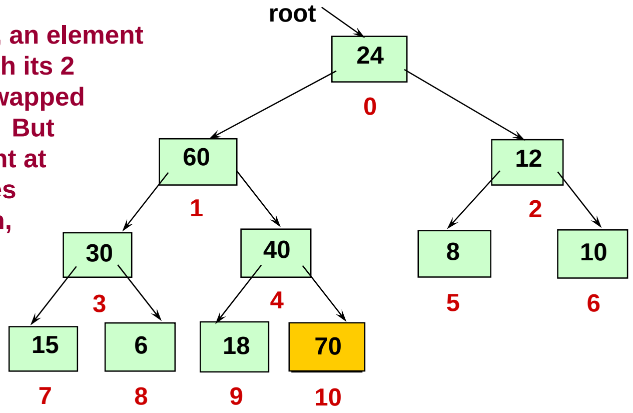




(f) Tree now represents a heap

Heap Sort:
How many comparisons?

In reheap down, an element is compared with its 2 children (and swapped with the larger). But only one element at each level makes this comparison, and a complete binary tree with N nodes has only O($\log_2$N) levels.



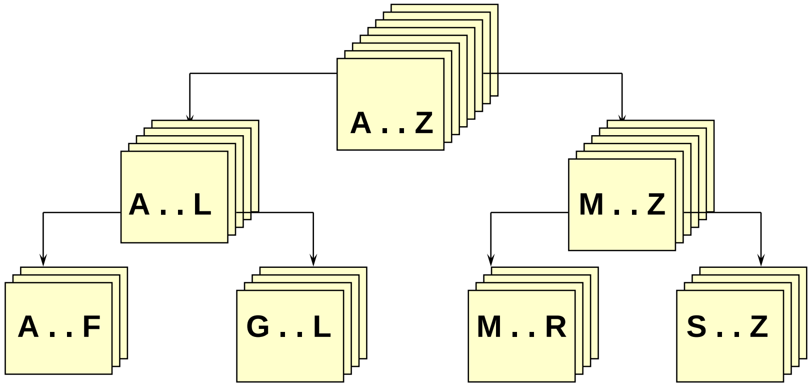
Heap Sort of N elements:
How many comparisons?

(N/2) * O(log N) compares to create original heap

(N-1) * O(log N) compares for the sorting loop

= O (N * log N) compares total

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

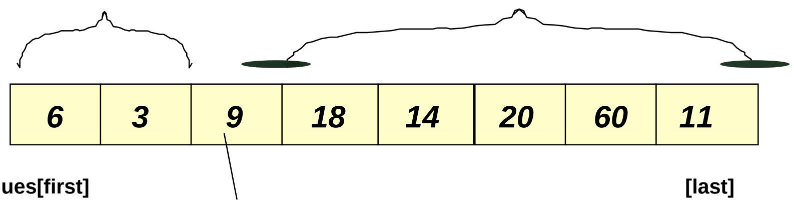
GOAL: place splitVal in its proper position with
all values less than or equal to splitVal on its left
and all larger values on its right

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!

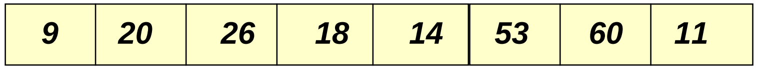
If each split divides the subarray approximately in half, there will be only $\log_2 N$ splits, and QuickSort is $O(N*\log_2 N)$.

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



values[first]

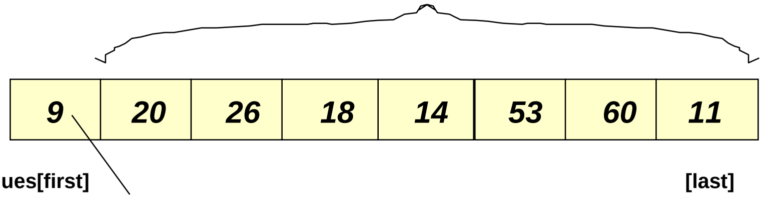
[last]

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

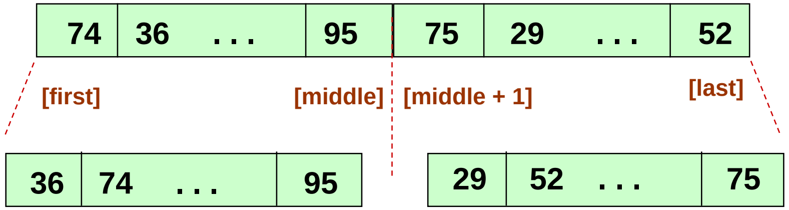
Merge Sort Algorithm

Cut the array in half.

Sort the left half.

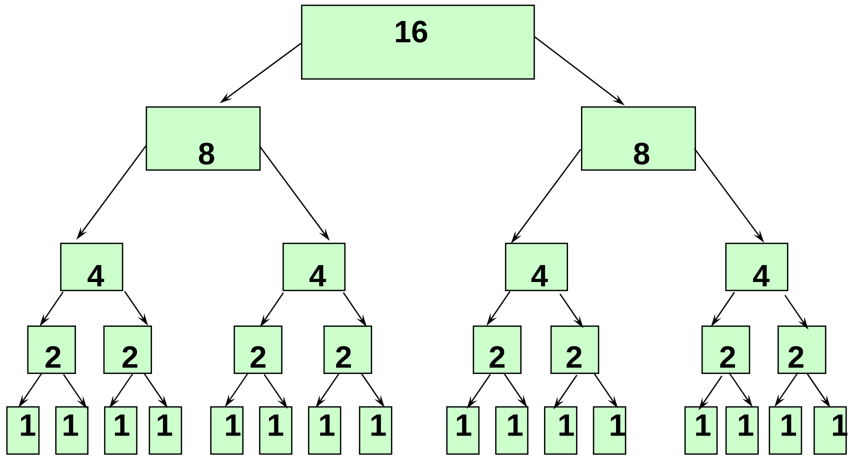
Sort the right half.

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?

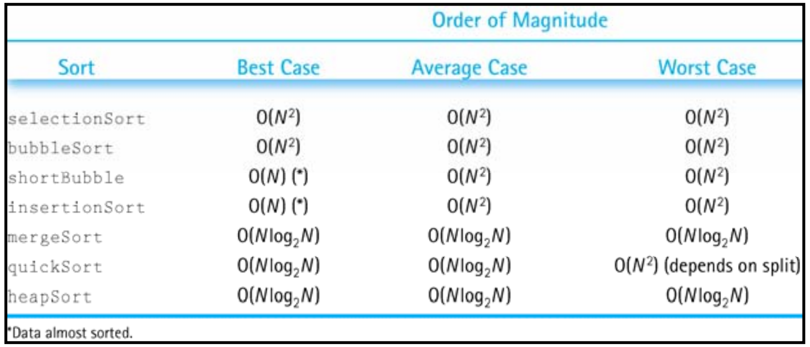
The entire array can be subdivided into halves only $\log_2 N$ times.

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	$O(N\log_2 N)$	$O(N\log_2 N)$	$O(N\log_2 N)$

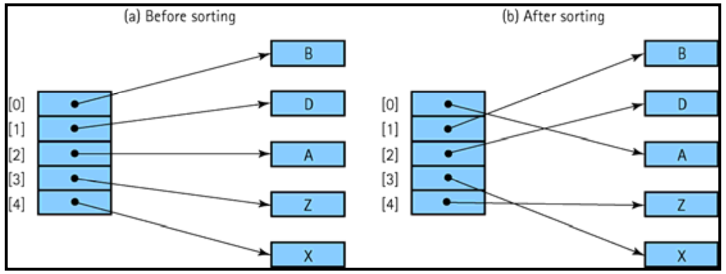
*Data almost sorted.

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

↳ Linear probing

	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.

Open hash

– Chaining

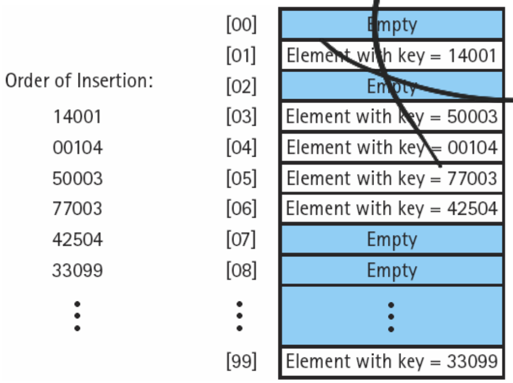
Collision Resolved

	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:

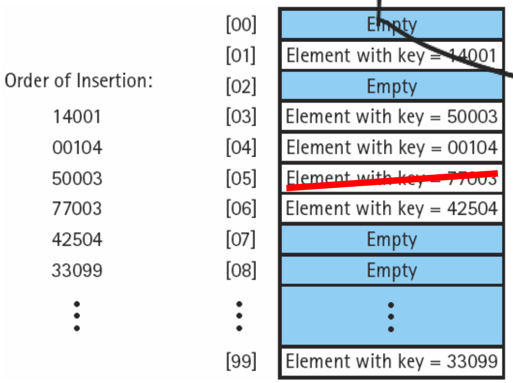
14001
00104
50003
77003
42504
33099
⋮

[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



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

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

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

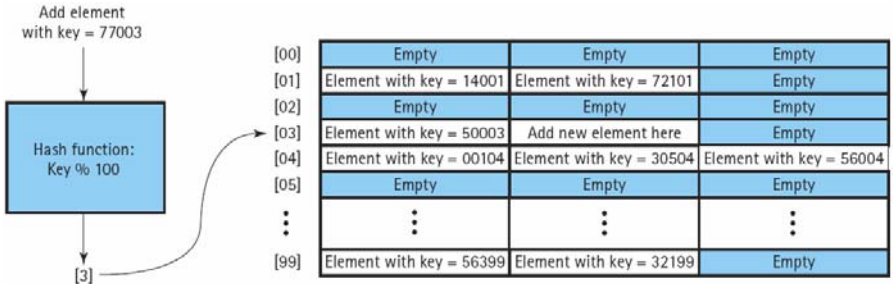
Resolving Collisions: Rehashing

- 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* ± I^2) % *array-size*
- random probing
  - (*HashValue + random-number*) % *array-size*

Resolving Collisions: Buckets and Chaining

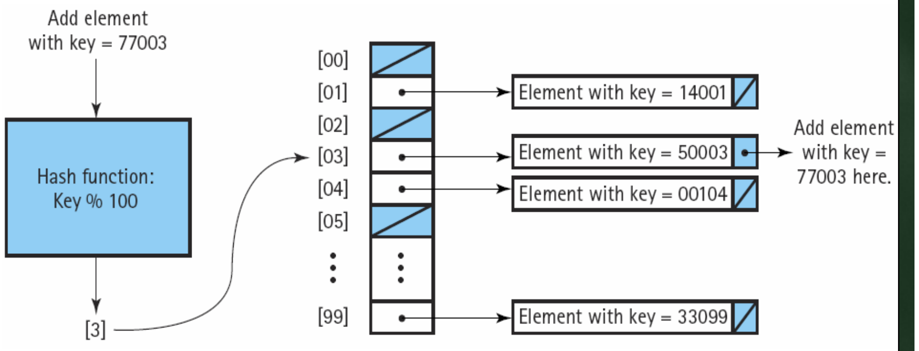
- The main idea is to allow multiple element keys to hash to the same location
- *Bucket* A collection of elements associated with a particular hash location
- *Chain* A linked list of elements that share the same hash location

Resolving Collisions: Buckets



[00]	Empty	Empty	Empty
[01]	Element with key = 14001	Element with key = 72101	Empty
[02]	Empty	Empty	Empty
[03]	Element with key = 50003	Add new element here	Empty
[04]	Element with key = 00104	Element with key = 30504	Element with key = 56004
[05]	Empty	Empty	Empty
⋮	⋮	⋮	⋮
[99]	Element with key = 56399	Element with key = 32199	Empty

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

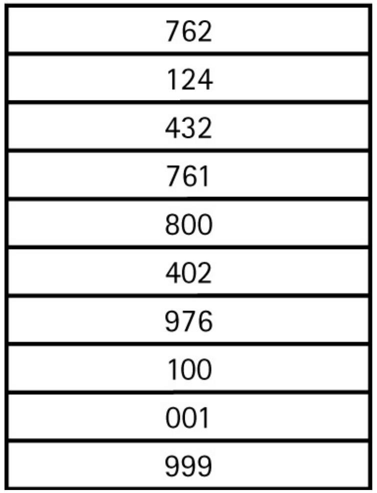
Radix sort

Is *not* a comparison sort

Uses a radix-length array of queues of records

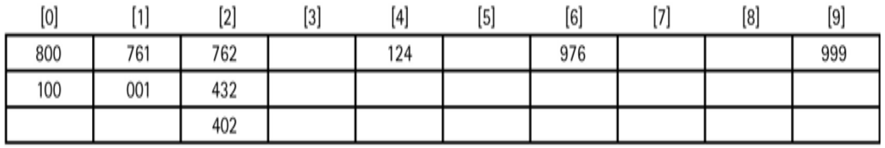
Makes use of the values in digit positions in the keys to select the queue into which a record must be enqueued

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