

## Topic 17

# Sorting

## Using ADTs to Implement Sorting Algorithms

13-1

## Objectives

- Examine several sorting algorithms that can be implemented using collections:
  - *Insertion Sort*
  - *Selection Sort*
  - *Quick Sort*
- Analyse the time complexity of these algorithms

13-2

## Sorting Problem

- Suppose we have an unordered list of objects that we wish to have sorted into ascending order
- We will discuss the implementation of several sort methods with a header of the form:

```
public void someSort( UnorderedList list)
// precondition: list holds a sequence of objects in
//               some random order
// postcondition: list contains the same objects,
//               now sorted into ascending order
```

13-3

## Comparing Sorts

- We will compare the following sorts:
  - *Insertion Sort* using stacks
  - *Selection Sort* using queues
  - *Quick Sort*
- Assume that there are **n** items to be sorted into ascending order

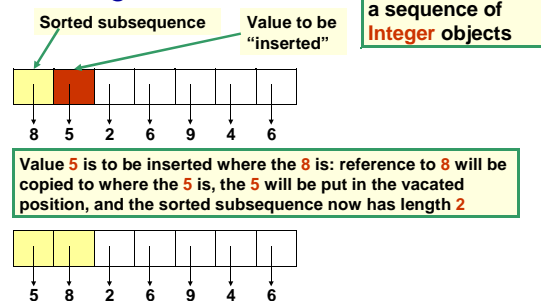
13-4

## Insertion Sort

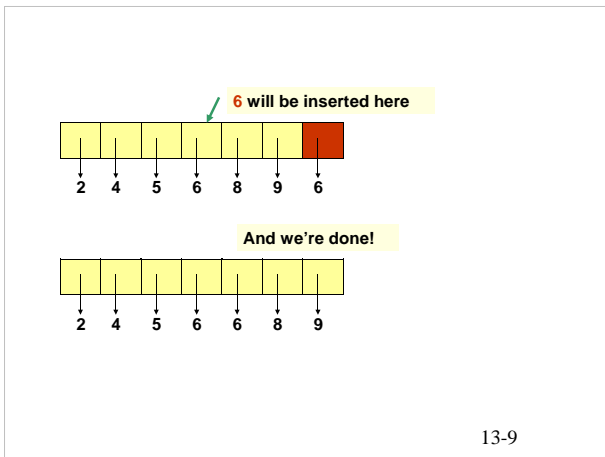
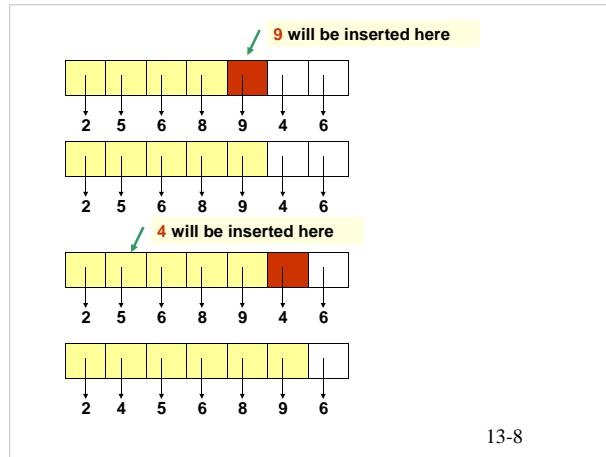
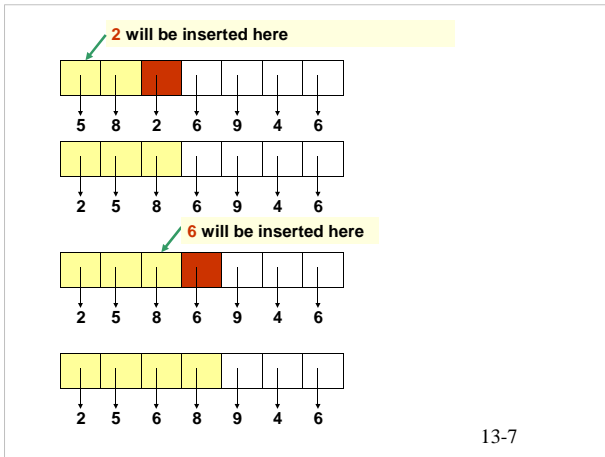
- *Insertion Sort* orders a sequence of values by repetitively inserting a particular value into a **sorted subset** of the sequence
- More specifically:
  - Consider the first item to be a **sorted subsequence** of length **1**
  - Insert the second item into the **sorted subsequence**, now of length **2**
  - Repeat the process, always inserting the **first** item from the **unsorted portion** into the **sorted subsequence**, until the entire sequence is in order

13-5

## Insertion Sort Algorithm



13-6



### Insertion Sort using Stacks

**Approach to the problem:**

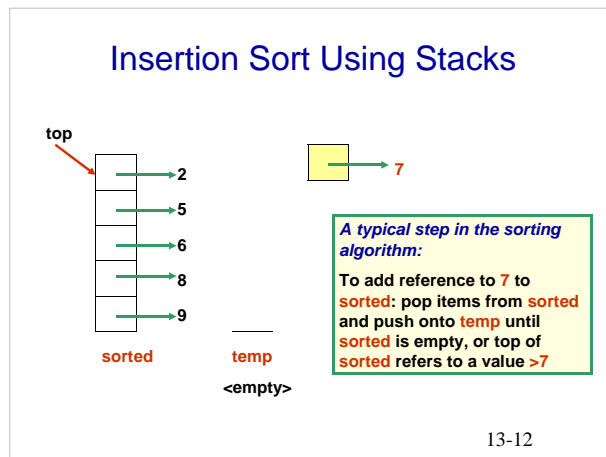
- Use two temporary stacks **sorted** and **temp**, both of which are originally empty
- The contents of **sorted** will always be in order, with the **smallest** item on the top of the stack
  - This will be the "sorted subsequence"
- **temp** will temporarily hold items that need to be "shifted" out in order to insert the new item in the proper place in **sorted**

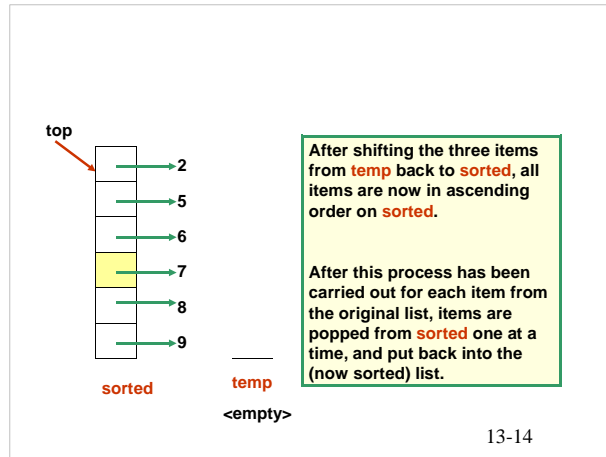
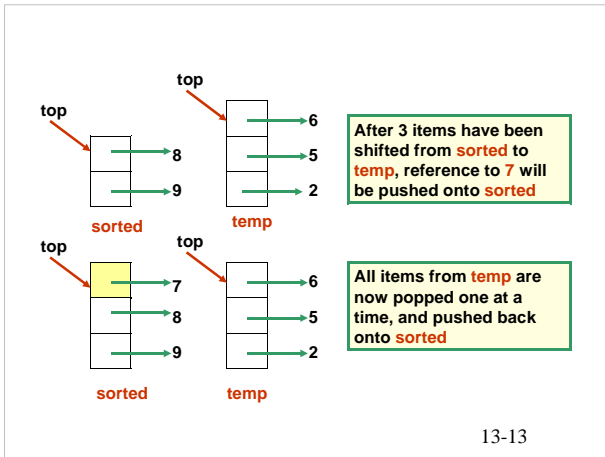
13-10

### Insertion Sort Using Stacks Algorithm

- While the list is not empty
  - **Remove the first** item from the list
  - While **sorted** is not empty and the top of **sorted** is smaller than this item, **pop** the top of **sorted** and **push** it onto **temp**
  - **push** the current item onto **sorted**
  - While **temp** is not empty, **pop** its top item and **push** it onto **sorted**
- The list is now empty, and **sorted** contains the items in ascending order from top to bottom
- To restore the list, **pop** the items one at a time from **sorted**, and **add to the rear** of the list

13-11





### Analysis of Insertion Sort Using Stacks

- Roughly:** nested loops on  $n$  items
  - So, what do you think the time complexity might be?
- In more detail:** analyze it in terms of the number of stack and list operations **push**, **pop**, **removeFirst**, **addToRear**

- Each time through the outer while loop, one more item is removed from the list and put into place on sorted:
  - Assume that there are  $k$  items in sorted. **Worst case:** every item has to be popped from sorted and pushed onto temp, so  $k$  pops and  $k$  pushes
  - New item is pushed onto sorted
  - Items in temp are popped and pushed onto sorted, so  $k$  pops and  $k$  pushes
  - So, total number of stack operations is  $4*k + 1$

- The outer while loop is executed  $n$  times, but each time the number of elements in sorted increases by 1, from 0 to  $(n-1)$ 
  - So, the total number of stack operations in the outer while loop, in the worst case, is  $(4*0 + 1) + (4*1 + 1) + \dots + (4*(n-1) + 1) = 2 * n^2 - n$
- Then there are  $n$  additional pops to move the sorted items back onto the list (adding at the rear of the list)
- So, the total number of stack operations is  $2 * n^2$

- Total number of stack operations is  $2 * n^2$
- Total number of list operations is  $2 * n$ 
  - In the outer while loop, removeFirst is done  $n$  times
  - At the end, addToRear is done  $n$  times to get the items back on the list
- Total number of stack and list operations is  $2 * n^2 + 2 * n$
- So, insertion sort using stacks is an  $O(n^2)$  algorithm

## Discussion

- Is there a **best case**?
  - Yes: the items are already sorted, but in reverse order (largest to smallest)
  - What is the time complexity then?
- What is the **worst case**?
  - The items are already sorted, in the correct order!!
  - Why is this the worst case?

13-19

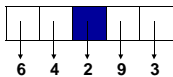
## Selection Sort

- **Selection Sort** orders a sequence of values by repetitively putting a particular value into its **final** position
- More specifically:
  - Find the **smallest value** in the sequence
  - Switch it with the value in the **first position**
  - Find the **next smallest value** in the sequence
  - Switch it with the value in the **second position**
  - Repeat until all values are in their proper places

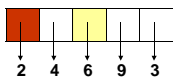
13-20

### Selection Sort Algorithm

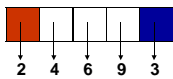
Initially, the **entire** container is the "**unsorted portion**" of the container.  
Sorted portion is coloured **red**.



Find smallest element in unsorted portion of container

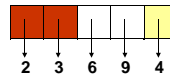


Interchange the smallest element with the one at the front of the unsorted portion

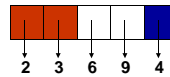


Find smallest element in unsorted portion of container

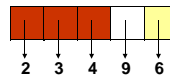
13-21



Interchange the smallest element with the one at the front of the unsorted portion

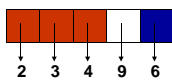


Find smallest element in unsorted portion of container



Interchange the smallest element with the one at the front of the unsorted portion

13-22



Find smallest element in unsorted portion of container



Interchange the smallest element with the one at the front of the unsorted portion

After **n-1** repetitions of this process, the last item has automatically fallen into place

13-23

### Selection Sort Using a Queue

#### Approach to the problem:

- Create a queue **sorted**, originally empty, to hold the items that have been sorted **so far**
- The contents of **sorted** will always be in order, with new items added at the end of the queue

13-24

## Selection Sort Using Queue Algorithm

- While the unordered list **list** is not empty:
  - **remove** the **smallest item** from **list** and **enqueue** it to the end of **sorted**
- The list is now empty, and **sorted** contains the items in ascending order, from front to rear
- To restore the original list, **dequeue** the items one at a time from **sorted**, and **add them to the rear** of **list**

13-25

- Now we will **refine** the step **remove the smallest item from list and enqueue it to the end of sorted**
  - We will call this step **moveSmallest**
  - It will use a second queue **temp** for temporary storage

13-26

## moveSmallest

- **smallestSoFar** = first item in **list** (**removeFirst**)
- While **list** is not empty
  - { **removeFirst item** from **list**
  - If **item** is less than **smallestSoFar**
    - {
    - enqueue smallestSoFar** to end of **temp**
    - smallestSoFar** = **item**
    - }
  - Else **enqueue item** to **temp**
  - }

13-27

## moveSmallest

- **list** will now be empty, **smallestSoFar** contains the smallest item, and **temp** contains all the other items
- **enqueue smallestSoFar** to the end of **sorted**
- **dequeue** the items one at a time from **temp**, and **add them to the rear** of **list** (a **while** loop)

13-28

## Analysis of Selection Sort Using Queue

- Analyze it in terms of the number of queue and list operations **enqueue**, **dequeue**, **removeFirst**, **addToRear**
- Each time **moveSmallest** is called, one more item is moved out of the original list and put into place in the queue **sorted**
- Assume there are **k** items in the list; inside **moveSmallest** we have:
  - **removeFirst** before the first while loop
  - **k-1 removeFirsts** and **k-1 enqueues** inside 1<sup>st</sup> while loop
  - **1 enqueue** (of **smallestSoFar**)
  - **k-1 dequeues** and **k-1 addToRears** inside 2<sup>nd</sup> while loop
  - So, total number of queue and list operations is **4\*k - 2**

13-29

- Now let's look at the whole algorithm:
  - It calls **moveSmallest** **n** times, and the number of elements in **list** ranges from **n** down to **1**
  - So, the number of list and queue operations in the first loop is
$$(4*n-2) + (4*(n-1)-2) + \dots + (4*2-2) + (4*1-2)$$
$$= 4*(n*(n+1)/2) - 2*n$$
$$= 2*n^2$$
  - Number of list and queue operations in the second loop is **2\*n**
  - Total number of list and queue operations is **2\*n<sup>2</sup> + 2\*n**
  - So, Selection Sort is an **O(n<sup>2</sup>)** algorithm

13-30

## Discussion

- Is there a best case?
  - No, we have to step through the entire remainder of the list looking for the next smallest item, no matter what the ordering
- Is there a worst case?
  - No

13-31

## Quick Sort

- **Quick Sort** orders a sequence of values by **partitioning** the list around one element (called the **pivot** or **partition element**), then sorting each partition
- More specifically:
  - Choose one element in the sequence to be the **pivot**
  - Organize the remaining elements into two groups (**partitions**): those **greater than** the **pivot** and those **less than** the **pivot**
  - Then sort each of the partitions (recursively)

13-32

## Quick Sort

- **partition element** or **pivot**:
  - The choice of the **pivot** is arbitrary
  - For efficiency, it would be nice if the pivot divided the sequence roughly in half
    - However, the algorithm will work in any case

13-33

## Quick Sort

### Approach to the problem:

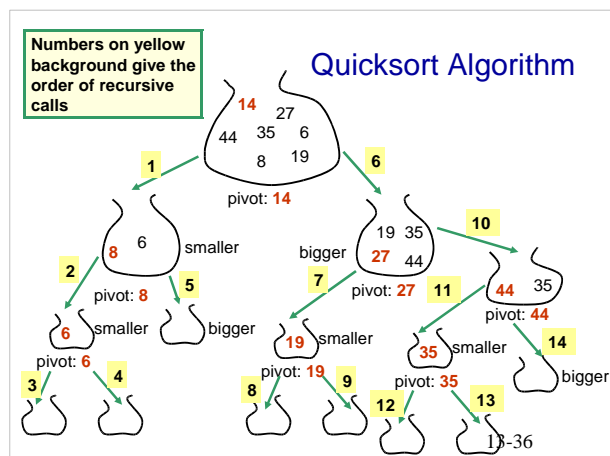
- We put all the items to be sorted into a container (e.g. an array, a collection)
  - We will call the container operations **get** and **put**
- We choose the pivot (partition element) at random from the container
- We use a container **smaller** to hold the items that are smaller than the pivot, and a container **bigger** to hold the items that are larger than the pivot
- We then **recursively** sort the items in the containers **smaller** and **bigger**

13-34

## Quick Sort Algorithm

- **Remove** the items to be sorted from the list and **put** them into an empty container
- If the container is not empty
  - **pivot** = one item from the container, chosen at random (**get**)
  - Create containers **smaller** and **bigger**, originally empty
  - While the container is not empty
    - **Get** an **item** from the container
    - If **pivot < item**, **put item** into **bigger** otherwise **put** item into **smaller**
  - Sort **smaller**
  - **Add pivot at rear** of list
  - Sort **bigger**

13-35



## Discussion

- What is the **base case**?
  - If the container is empty, done.
- How is this recursive solution different from the recursive solution to Fibonacci?

13-37

## Analysis of Quick Sort

- We will look at two cases for Quick Sort :
  - **Worst case**
    - When the pivot element is the **largest** or **smallest** item in the container (why is this the worst case?)
  - **Best case**
    - When the pivot element is the **middle** item (why is this the best case?)

13-38

- To simplify things, we will do the analysis only in terms of the number of **get** operations needed to sort an initial container of **n** items, **T(n)**
  - We could also include the **put** operations – but why can we leave them out?
  - In the interest of fairness of comparison with the other sorting methods, we should also consider that the unsorted items started in a list and should end in the list - we will discuss this later

13-39

### Worst Case Analysis:

- Assume that the pivot is the **largest** item in the bag
- If **n** is **0**, there are no operations, so **T(0)=0**
- If **n > 0**, the pivot is removed from the container (**get**) and the remaining **n-1** items are redistributed (**get, put**) into two containers:
  - **smaller** is of size **n-1**
  - **bigger** is of size **0**
- So, the number of **get** operations for this step is **1 + (n-1)**

13-40

- Then we have two recursive calls:
  - Sort **smaller**, which is of size **n-1**
  - Sort **bigger**, which is of size **0**
- So, **T(n) = 1 + (n-1) + T(n-1) + T(0)**
  - But, the number of **get** operations required to sort a container of size **0** is **0**
  - And, the number of **get** operations required to sort a container of size **k** in general is  
**T(k) = 1 + (k-1) + (the number of get operations needed to sort a container of size k-1)**  
**= k + T(k-1)**

13-41

- So, the total number of **get** operations **T(n)** for the sort is

$$\begin{aligned}T(n) &= n + T(n-1) \\ &= n + (n-1) + T(n-2) \\ &= n + (n-1) + (n-2) + \dots + 2 + 1 + T(0) \\ &= n*(n+1)/2 \\ &= n^2/2 + n/2\end{aligned}$$

- So, the **worst case** time complexity of Quick Sort is **O(n<sup>2</sup>)**

13-42

## Discussion

- Let's now also consider the operations needed to get the  $n$  items from the original list into the container:
  - $n$  `removeFirst` operations,
  - $n$  `put` operations
- We also have the operations needed to `get` the  $n$  items from the container at the end, and `add them at the rear` of the list
- Will considering these operations change the time complexity of our Quick Sort?

13-43

## Best Case Analysis

- The **best case** occurs when the pivot element is chosen so that the two new containers are as close as possible to having the same size
- Again, we would consider only the `get` operations
  - For a container of size  $n$  there are  $1 + (n-1) = n$  `get` operations before the recursive calls are encountered
  - **Each** of the **two** recursive calls will sort a container that contains **at most**  $n/2$  items

13-44

- It is beyond the scope of this course to do the analysis, but it turns out that the **best case** time complexity for Quick Sort is  $O(n \log_2 n)$
- And it turns out that the **average** time complexity for Quick Sort is the same

13-45

## Summary

- **Insertion Sort** is  $O(n^2)$
- **Selection Sort** is  $O(n^2)$
- **Quick Sort** is (if we do it right!)  $O(n \log_2 n)$
- Which one would you choose?

13-46