

# **TITLE:** Python Programming: Problem Solving, Packages and Libraries

## **Edition**

---

Lecture PPT 17. Linear List Manipulation, Stacks and Queues

# Chapter 17. Linear List Manipulation, Stacks and Queues

## LEARNING OBJECTIVES

*After studying this chapter, you will be able to:*

- LO 1** Understand the linear versus nonlinear data structures.
- LO 2** Apply insertion in a sorted list using two different methods of the bisect module namely `bisect()` and `insort()`.
- LO 3** Use an algorithm for manual insertion of an item in a sorted list.
- LO 4** Implement some algorithms for deleting an item in a list manually .
- LO 5** Understand the linear search algorithm and binary search algorithm.
- LO 6** Implement binary search by recursion.
- LO 7** Implement three algorithms for sorting, namely: selection sort, bubble sort and insertion sort.

# Data Structures

A data structure is a group of data which is processed together. Data structures are of two types: linear and nonlinear. Both the data structures are explained in detail in below.

Properties	Linear data structures	Nonlinear data structures
<b>1</b>	Data elements are stored one after another	Data elements are not stored one after another
<b>2</b>	Elements form a sequence (Example: array, list, que)	Elements do not form a sequence (Example: Tree, Hashed tree, etc.)
<b>3</b>	If you traverse from one element, you can strictly reach only one other element	In traversal, from one element, you may reach more than one other element
<b>Advantages</b>	If you search one item, it is very easy to find the next item.	Use memory efficiently since free contiguous memory not a requirement. Length of data items being stored need not be known prior to allocation
<b>Disadvantages</b>	Size of array should be known prior to allocation. Requires contiguous memory	Overhead of link to next item.

# Inserting element in a sorted list

- A list is a linear data structure, so if you try to insert an item somewhere in the list, you have to change the index of some of the elements.
- For example, if you have a sorted list say [2,4,6,8], and if you want to put a 5 in it and maintain the sorted order then it will become [2,4,5,6,8].
- So the index of all the items after the inserted item will change.
- Of course if you append, i.e., add an item at the end, it will not change the index of the rest of the items, but it will increase the number of items by one.
- So if you want to insert an item somewhere in a sorted list and also maintain the sort order, then you may have to modify the indexes of some of the items of the list. (Of course if the inserted item is larger than all the items in the sorted list, then you don't have to modify index of any of the items).
- Similarly, if the inserted item is smaller than all the items in the list then the index of all the items will have to be incremented by 1. Python provides in-built module for inserting an item in a sorted list.

# Inserting an element in a sorted list using bisect module -- 1

- Using the bisect module, you can insert an item into a sorted list while “maintaining its sort order”.
- So using bisect module to insert while maintaining the sort order is much more efficient than randomly inserting an item and then sorting all the items after each insertion.
- The `insert(sequence, item)` method of bisect module inserts item into the sequence, keeping it sorted.
- Here, two methods of the bisect module may be used .
- First is the `bisect()` method of the bisect module and it returns the index where the element is to be inserted in the list.
- The second is the `insert()` method of the bisect module which returns a sorted list.
- To sort the list you need only the `insert()` method.
- But if you need to know the index where the item was inserted in the list, then you need the `bisect()` method.
- The `bisect()` method does not modify the list. It simply tells at what index, the item will be inserted (if it is inserted).

# Inserting an element in a sorted list using bisect module -- 2

The following sample script shows use of

- `bisect.bisect()`
- `bisect.insort()`

Note that to insert an item in a sorted list you don't need `bisect.bisect()`. You need only to get the index where the item is to be inserted.

```
1 >>> import bisect
2 >>> L = [1,3,5,7,9]
3 >>> bisect.bisect(L, 6) # Returns index where item ie 6 to be inserted
4 3
5 >>> bisect.insort(L,6) # item 6 inserted so list remains in ascending order
6 >>> L
7 [1, 3, 5, 6, 7, 9]
8 >>>
```

# Inserting an item in a sorted list manually -- 1

This method applies when you have been given a sorted list of numbers (ascending/descending) and you are asked to “insert” a number so that the sort does not get disturbed. The steps in the implementation are:

1	Take the number and a list which is in ascending order
2	Check whether the number is smaller or equal to the smallest number in the list. If yes add the number at the beginning of the list using the '+' operator. Exit
3	Check whether the number is larger or equal to the largest number in the list. If yes then add the number to the end of the list. Exit
3	You have checked that the number is larger than the smallest number in the list but smaller than largest number in the list. So the number must be inserted somewhere in the list
4	Write a function to find the index where this number should be inserted. This is done by comparing the number with each number in the list until that number (of the list) is found which is equal or larger than the number being added. Take the index of this number in the list
5	Now break up the list in two parts at the index found in step 4
6	Now convert the number to be added into a list using the square brackets and call it tempL2. Suppose the lower part of the broken list is tempL1, the number to be added is converted into tempL2 and the upper part of the list is tempL3. Then the final list will be tempL1 + tempL2 + tempL3

# Inserting an item in a sorted list manually -- 2

The following figure shows an example of a list in ascending order namely [1, 2, 3, 4, 5]. Then three cases are shown, i.e., adding a 0, a 7 and a 2.5 to the list.

3 Cases	Steps		
	1	2	3
1. Adding a number small or equal to the smallest in the list	Add 0 to [1, 2, 3, 4, 5]	[0] + [1, 2, 3, 4, 5]	[0, 1, 2, 3, 4, 5]
2. Adding a number larger than or equal to the largest number in the list	Add 7 to [1, 2, 3, 4, 5]	[1, 2, 3, 4, 5] + [7]	[1, 2, 3, 4, 5, 7]
3. Adding a number which is larger than the smallest number of the list but smaller than largest number of the list	Add 2.5 to [1, 2, 3, 4, 5]	[1, 2] + [2.5] + [3, 4, 5]	[1, 2, 2.5, 3, 4, 5]

**FIGURE 17.1** Three different numbers 0, 7 and 2.5 added to a “sorted” list



# Deleting an item whose “index” is given from a list (sorted or unsorted)

- The figure from the book takes a list [10, 20, 30, 40, 50, 60, 70, 80].
- It then removes the item at index 3, which is 40.
- On removing item at index 3, all subsequent items have to be “shifted left” by 1, which is done by copying the item to the cell to its left.
- Finally the last item becomes duplicate, so it is deleted

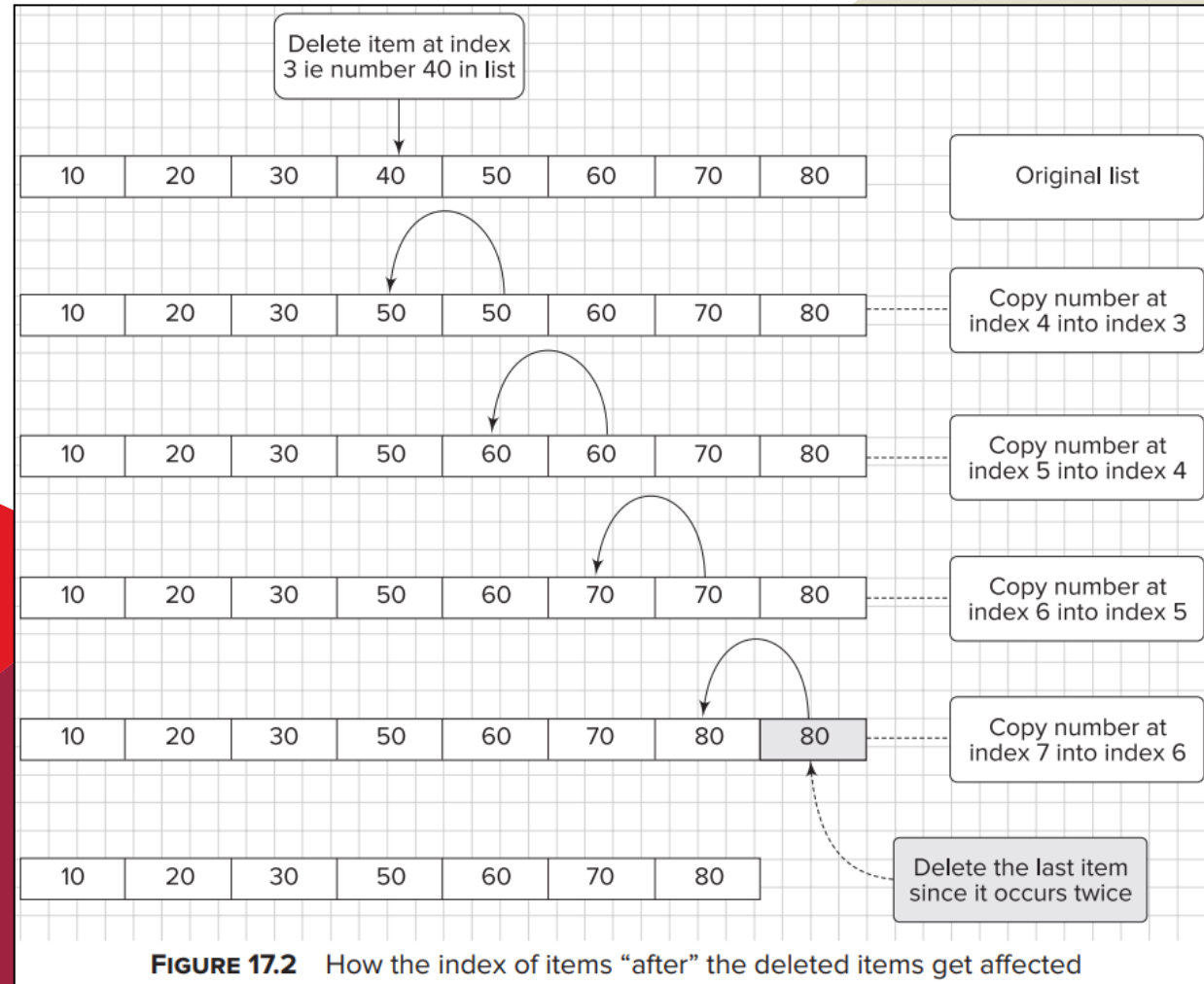
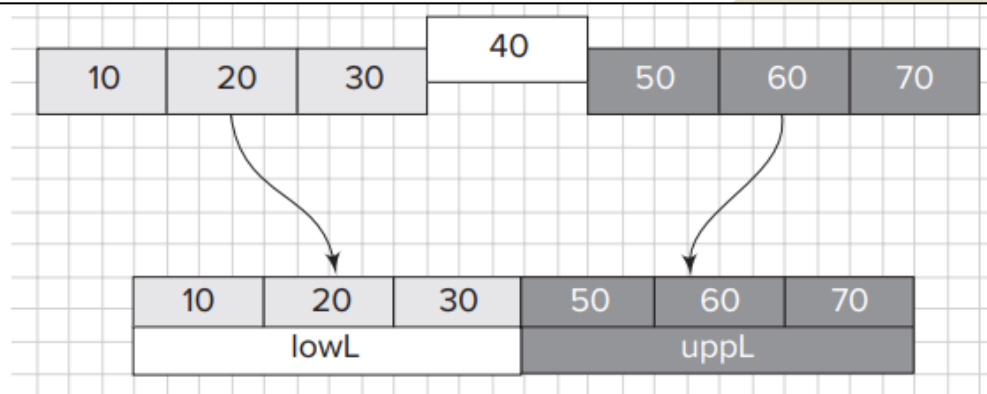


FIGURE 17.2 How the index of items “after” the deleted items get affected

# A second way of deleting an item:

You can delete an item by splitting the list into two lists, i.e., a sub-list containing the lower elements and an upper sub-list containing the upper elements and then joining the two sub-lists using the concatenation, i.e., '+' operator.

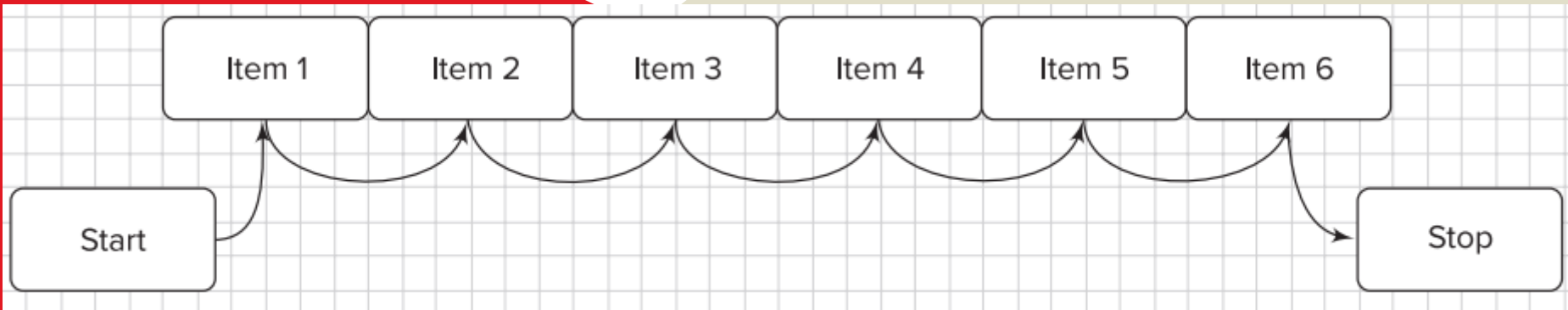


**FIGURE 17.3** Deleting an item by first splitting the list, removing the item to be deleted and then “concatenating” the two sub-lists

The book has script which implements the above logic.

# Linear search

- In a search you have an item being searched which you may call “pattern” and you have some “collection” of items.
- In a linear search you compare the “pattern” to each item in the container until you succeed or all items in the container are compared. In common Python scripts, this container is generally a list.
- Figure 17.4 below gives a “visual representation” of the process of linear search.



**FIGURE 17.4** A “visual representation” of the process of linear search

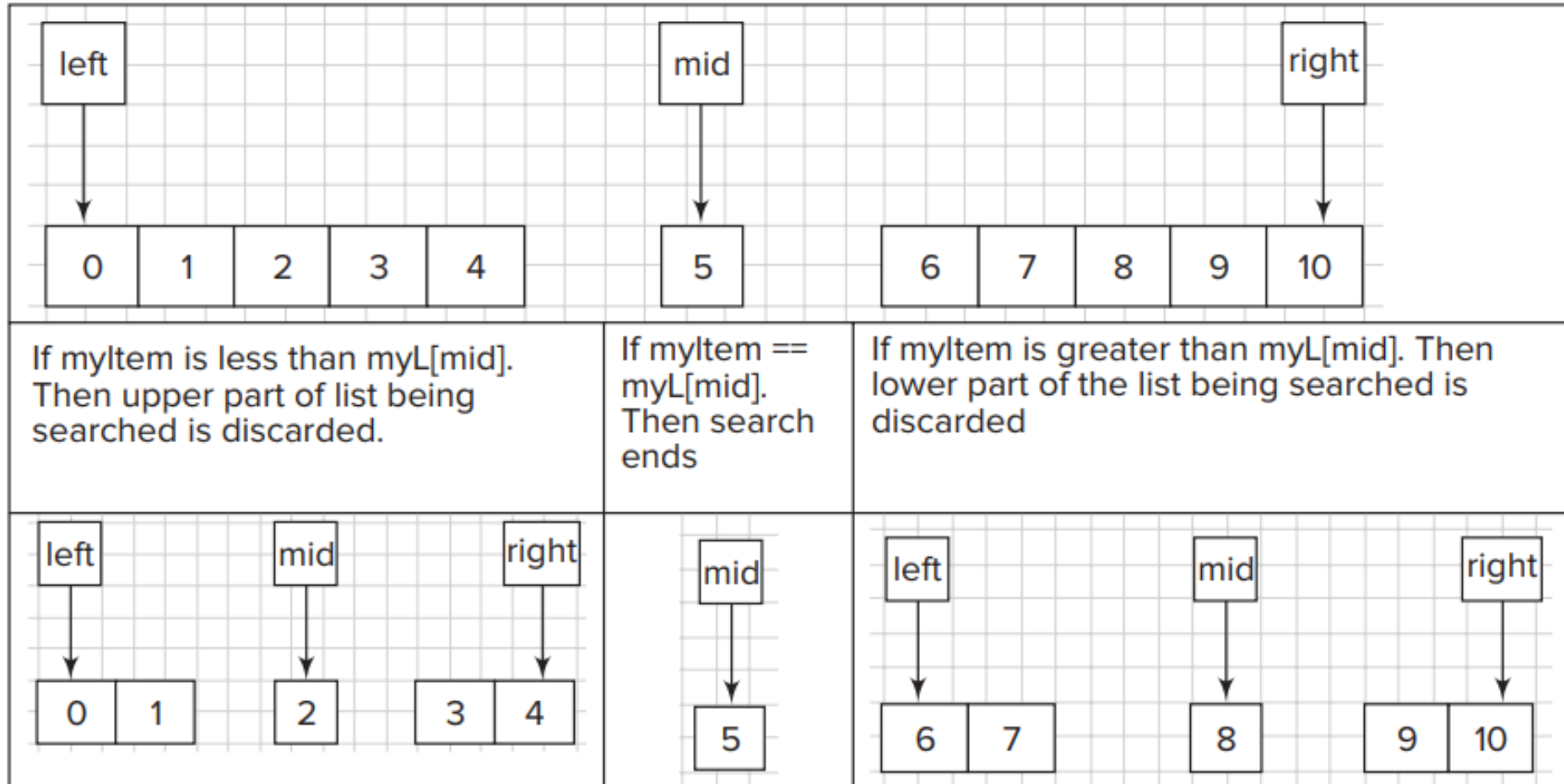
# Binary search -- 1

**This algorithm is applicable only to a sorted list.**

	Steps	Explanation
1	<pre>left ← 0, right ← len(myL) - 1 mid ← int((left + right)/2)</pre>	left represents the lower index of the part of the list being searched while right represents the upper index of the part of list being searched. mid represents the integer of the middle of the list. Int of middle means that if the sum is odd, then after dividing by 2, the result is converted to integer value after removing the decimal part (something like floor function).
2	<pre>myL[mid] == myItem</pre>	Each time, the item being searched is compared to the item in the middle of the list being searched.
3	<pre>if myL[mid] &lt; myItem:     left = mid +1</pre>	If myItem being searched is bigger than the item in middle of list, the lower half of the list is discarded.
4	<pre>else:     right = mid - 1</pre>	If myItem is less than the item in the middle of list, then upper half of the list being searched is discarded

# Binary search -- 2

The following figure gives a visual representation of how binary search works



**FIGURE 17.5** “binary search” on a sorted list

# Binary search -- 3

How the algorithm works on the list

**L = [21, 32, 33, 44, 56, 57, 68, 79, 81, 92, 100, 101]**

while looking for **92**, is as follows:

Index →	0	1	2	3	4	5	6	7	8	9	10	11
Start	L					M						R
	21	32	33	44	56	57	68	79	81	92	100	101
End of Pass1							L		M			R
	21	32	33	44	56	57	68	79	81	92	100	101
End of Pass2										L	M	R
	21	32	33	44	56	57	68	79	81	92	100	101
End of Pass3										L/R	M	
	21	32	33	44	56	57	68	79	81	92	100	101
End of Pass4										L/R/ M		
	21	32	33	44	56	57	68	79	81	92	100	101

# Binary search -- 4

**How the algorithm works on the list**

**L = [21, 32, 33, 44, 56, 57, 68, 79, 81, 92, 100, 101]**  
**while looking for 92, is as follows:**

Index →	0	1	2	3	4	5	6	7	8	9	10	11
Start	L					M						R
	21	32	33	44	56	57	68	79	81	92	100	101
End of Pass1							L		M			R
	21	32	33	44	56	57	68	79	81	92	100	101
End of Pass2										L	M	R
	21	32	33	44	56	57	68	79	81	92	100	101
End of Pass3										L/R	M	
	21	32	33	44	56	57	68	79	81	92	100	101
End of Pass4										L/R/M		
	21	32	33	44	56	57	68	79	81	92	100	101

**The script implementing binary search is given in the book**

# Binary search (using recursion)

You can search for an item in a list sorted in ascending order by using recursion also.

The steps are as follows:

- Check if the item being searched (say  $n$ ) is smaller than the smallest or bigger than the biggest item in the sorted list.
- Compare item being searched (i.e.,  $n$ ) to the middle item in the sorted list. If it matches, terminate with index of middle item.
- If item  $n$  doesn't match with the middle item, see whether the item is smaller or bigger than the middle item.
- If  $n$  is smaller than the middle item, discard the upper half of the list being searched and search in lower half.
- If  $n$  is bigger than the middle item, discard the lower half of the list and search in the upper half.
- Keep on recursively calling the function until item at right end of list is bigger or equal to the item at left end of the sub-list being searched.

**[Script implementing binary search through recursion is given in the book]**



# Selection sorting -- 1

**Sorting means ordering a list of values.**

**There are three important sorting algorithms to be covered. They are (1) selection sort, (2) bubble sort and (3) insertion sort.**

**Selection sort works as follows:**

- Find the smallest item in an array and exchange it with the item at the first place (i.e., at index 0)**
- Find the next smallest item in the array and exchange it with item at second place (index 1)**
- The process is continued till the entire list is sorted.**

# Selection sorting -- 2

Take a list [5, 4, 2, 3, 7, 9, 10] to explain the concept “visually”. The “current item” in the list which is “being compared” is marked with curly brackets, i.e., { and } as follows:

1	{5}[4, 2, 3, 7, 9, 10]	Item at index 0, i.e., {5} compared to item at index 1, i.e., 4 and exchanged
2	{4}[5, 2, 3, 7, 9, 10]	Item at index 0, i.e., {4} compared to item at index 2, i.e., 2 and exchanged
3	{2}[5, 4, 3, 7, 9, 10]	Now item at index 0 is the smallest
4	[2]{5}[4, 3, 7, 9, 10]	Item at index 1, i.e., {5} is taken up
5	[2]{4}[5, 3, 7, 9, 10]	Item at index 1, i.e., {5} is exchanged with item at index 2, i.e., 4.
6	[2]{3}[5, 4, 7, 9, 10]	Item at index 1, i.e., {4} is exchanged with item at index 3, i.e., 3
7	[2, 3]{5}[4, 7, 9, 10]	Item at index 2, i.e., 5 is taken up
8	[2, 3]{4}[5, 7, 9, 10]	Item at index 2 is exchanged with item at index 3, i.e., 5
9	[2, 3, 4]{5}[7, 9, 10]	No further exchange
10	[2, 3, 4, 5]{7}[9, 10]	No further exchange
11	[2, 3, 4, 5, 7]{9}[10]	No further exchange
12	[2, 3, 4, 5, 7, 9, 10]	Final list

# Bubble sort

In “bubble sort” you make “multiple passes” over a list. In each pass, you compare adjacent items and exchange those that are out of order.

This will become clear from the following example. Here curly brackets {} denote the item which is being compared and round brackets () represent the items to which it is being compared.

1	{6}, [(5), 4, 3, 2, 1]	→	[5]{6}[4, 3, 2, 1]	
2	[5]{6}[(4), 3, 2, 1]	→	[5, 4]{6}[3, 2, 1]	
3	[5, 4]{6}[(3), 2, 1]	→	[5, 4, 3]{6}[2, 1]	
4	[5, 4, 3]{6}[(2), 1]	→	[5, 4, 3, 2]{6}[1]	
5	[5, 4, 3, 2]{6}(1)	→	[5, 4, 3, 2, 1]{6}	Biggest item, i.e., 6 bubbled to end of list
6	{5}[(4), 3, 2, 1][6]	→	[4]{5}[3, 2, 1][6]	
7	[4]{5}[(3), 2, 1][6]	→	[4, 3]{5}[2, 1][6]	
8	[4, 3]{5}[(2), 1][6]	→	[4, 3, 2]{5}[1][6]	
9	[4, 3, 2]{5}(1)[6]	→	[4, 3, 2, 1][5, 6]	Second biggest, i.e., 5 bubbled to second last
10	{4}[(3), 2, 1][5, 6]	→	[3]{4}[2, 1][5, 6]	
11	[3]{4}[(2), 1][5, 6]	→	[3, 2]{4}[1][5, 6]	
12	[3, 2]{4}(1)[5, 6]	→	[3, 2, 1]{4}[5, 6]	
13	{3}[(2)1][4, 5, 6]	→	[2]{3}[1][4, 5, 6]	Third biggest, i.e., 4 bubbled to third last
14	[2]{3}(1)[4, 5, 6]	→	[2, 1]{3}[4, 5, 6]	
15	{2}(1)[3, 4, 5, 6]	→	[1]{2}[3, 4, 5, 6]	Fourth biggest, i.e., 3 bubbled to fourth last
16	[1, 2, 3, 4, 5, 6]			Final sorted list

# Insertion sort -- 1

- The best way to think of insertion sort is as if you deal with cards one by one and you arrange them in ascending order.
- Initially you start with only one card.
- Then you get another card and you sort the two cards. So the cards you are holding in your hand are sorted, but when you get a new card, it may lie somewhere between the cards you are holding so you put it in proper position so that the cards you are holding in your hand are again sorted.
- So each new card is “inserted” at the proper place in the partially sorted cards. The process is repeated.
- Call the card dealt as “key”. So, now the cards are in three parts.
  - The first is the partial sorted list in your hand.
  - The second is the card dealt or the “key”.
  - The third is the partial list of cards to be dealt.
- Suppose you have cards 1,4 and 6 in your hand and you start with card 3. Further suppose that cards 9, 2 and 5 will come later.
- You could represent this as: [1, 4, 6] {3} [9, 2, 5].
- Here [1,4,6 ]and [9,2,5] are used to represent the sorted and unsorted lists and {3} is for the key.
- So now when you insert a 3, the situation becomes [1,3,4,6]{9}[2,5]. Now when you insert a 9, it becomes [1,3,4,6,9]{2}[5]. After you insert the 2, it becomes [1,2,3,4,6,9]{5}. After you insert the 5, it becomes [1,2,3,4,5,6,9].

# Insertion sort -- 2

Following is another example, i.e., list [5, 4, 2, 3, 7, 9, 10].

[5]{4}[2,3,10,9,8,1]	Sorted sub-list is [5], key is {4}, unsorted sub-list is [2,3,10,9,8,1]
[4,5]{2}[3,10,9,8,1]	Sorted sub-list is [4,5], key is {2}, unsorted sub-list is [3,10,9,8,1]
[2,4,5]{3}[10,9,8,1]	Sorted sub-list is [2,4,5], key is {3}, unsorted sub-list is [10,9,8,1]
[2,3,4,5]{10}[9,8,1]	Sorted sub-list is [2,3,4,5], key is {10}, unsorted sub-list is [9,8,1]
[2,3,4,5,10]{9}[8,1]	Sorted sub-list is [2,3,4,5,10], key is {9}, unsorted sub-list is [8,1]
[2,3,4,5,9,10]{8}[1]	Sorted sub-list is [2,3,4,5,9,10], key is {8}, unsorted sub-list is [1]
[2,3,4,5,8,9,10]{1}	Sorted sub-list is [2,3,4,5,8,9,10], key is {1}. No unsorted sub-list
[1,2,3,4,5,8,9,10]	Sorted sub-list is [1,2,3,4,5,8,9,10]

# Comparison of the 3 sorting algorithms

The following list shows use of the three sorting algorithms namely: selection, bubble and insertion sort on the same list L= [90, 78, 20, 46, 54, 1]. This gives a visual representation of how these three algorithms work.

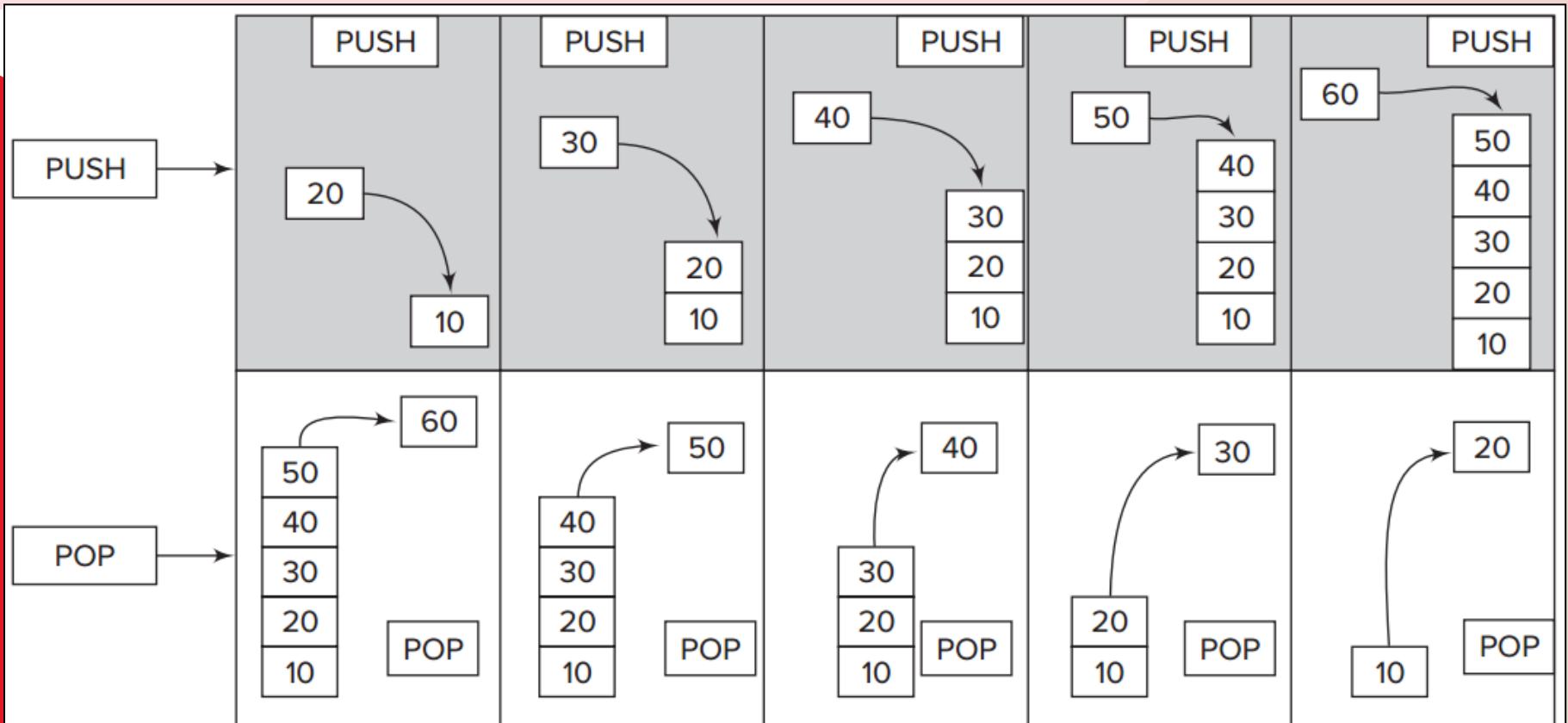
	Selection	Bubble	Insertion
	[{90}, 78, 20, 46, 54, 1]	[{90}, 78, 20, 46, 54, 1]	[90, {78}, 20, 46, 54, 1]
1	{78}[90, 20, 46, 54, 1]	[78]{90}[20, 46, 54, 1]	[78, 90]{20}[46, 54, 1]
2	{20}[90, 78, 46, 54, 1]	[78, 20]{90}[46, 54, 1]	[78]{20}[90][46, 54, 1]
3	{20}[90, 78, 46, 54, 1]	[78, 20, 46]{90}[54, 1]	[20, 78, 90]{46}[54, 1]
4	{20}[90, 78, 46, 54, 1]	[78, 20, 46, 54]{90}[1]	[20, 78]{46}[90, 54, 1]
5	[1]{90}[78, 46, 54, 20]	{78}[20, 46, 54, 1][90]	[20]{46}[78][90, 54, 1]
6	[1]{78}[90, 46, 54, 20]	[20]{78}[46, 54, 1][90]	[20, 46, 78, 90]{54}[1]
7	[1]{46}[90, 78, 54, 20]	[20, 46]{78}[54, 1][90]	[20, 46, 78]{54}[90, 1]
8	[1]{46}[90, 78, 54, 20]	[20, 46, 54]{78}[1][90]	[20, 46]{54}[78, 90, 1]
9	[1, 20]{90}[78, 54, 46]	{20}[46, 54, 1][78, 90]	[20, 46, 54, 78, 90][1]
10	[1, 20]{78}[90, 54, 46]	[20]{46}[54, 1][78, 90]	[20, 46, 54, 78]{1}[90]
11	[1, 20]{54}[90, 78, 46]	[20, 46]{54}[1][78, 90]	[20, 46, 54]{1}[78, 90]
12	[1, 20, 46]{90}[78, 54]	{20}[46, 1][54, 78, 90]	[20, 46]{1}[54, 78, 90]
13	[1, 20, 46]{78}[90, 54]	[20]{46}[1][54, 78, 90]	[20]{1}[46, 54, 78, 90]
14	[1, 20, 46, 54]{90}[78]	{20}[1][46, 54, 78, 90]	[1, 20, 46, 54, 78, 90]
15	[1, 20, 46, 54, 78, 90]	[1, 20, 46, 54, 78, 90]	

# Stacks -- 1

- **Think of a stack as a stack of plates.**
- **You can add a plate to the top and you can remove a plate from the top.**
- **So a stack is a last in first out (LIFO).**
- **A stack in Python is also a “collection” of items.**
- **It has two basic operations, namely:**
  - 1. push, which adds an element to the collection, and**
  - 2. pop, which removes the most recently added but not yet removed element from the collection.**

## Stacks -- 2

The following figure shows how items are pushed and popped from a stack. The figure has 2 rows and each row has 5 columns. The top row shows how “push” works and the bottom row shows how “pop” works



**FIGURE 17.6** Top row of figure shows how “push” works and bottom row shows how “pop” works



# Queue -- 1

**A queue is a container of objects with the following characteristics:**

- The objects are inserted and removed using first-in first-out (FIFO) principle. What is put in first comes out first.**
- The permitted operations are: (1) enQueue and (2) deQueue. The enQueue operation is for inserting an item at the end/ back of the queue The deQueue means operation is for removing the item at beginning/ front of the queue.**
- Stacks and queues different in the way they “remove” items. In a stack you remove the item added last whereas in a queue you remove the item added first.**
- Note that adding elements in both stack and queue are same and in both the items are added at the “tail or end” of the data structure.**

## Queue -- 2

**Some relevant aspects of “operations” on queue data structure are:**

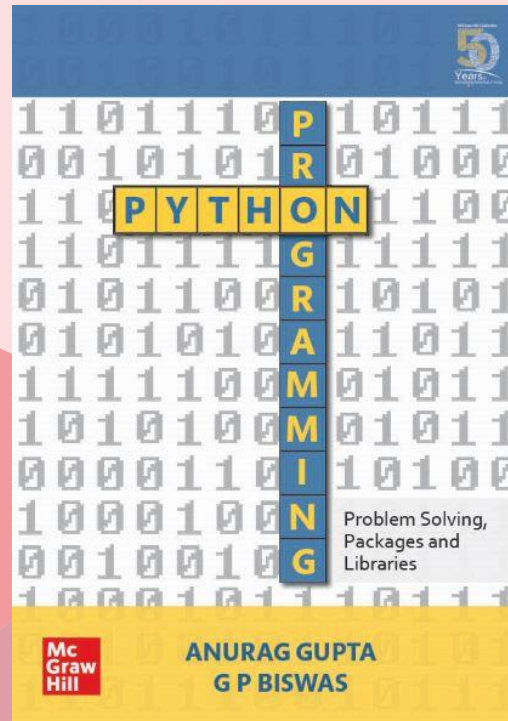
- You can add an element to a queue. This process is called enqueuing or an “insert” operation. It is always done at the “end or tail” of the queue.**
- You can remove an element from a queue. This is called “dequeuing” or a “delete” operation. It is done at the “front or beginning or head” of the queue.**
- Note that the two operations, i.e., enqueueing and dequeuing are done at two opposite ends of the queue. So you need to know the “index” of both the head and tail.**
- Queue is a FIFO operation. So index of tail is used for enqueueing and index of head is used for dequeuing.**

**Note that list objects have two methods: `append()` and `insert()`.**

- For `list.insert(idx, item)`: You can pick where the value will be added to the list. You can only add one value to a list at a time. Each value you insert to a list is considered one element.**
- For `list.append(item)`: You cannot pick where the value will be added to the list (it will be added as the last value).**

# Some common list methods

- `list.append(x)` (equivalent to `push` in Python)
- `my_list.extend(seq)`
- `my_list.insert(i, x)`
- `my_list.remove(x)`
- `my_list.pop([i])`
- `del list[idx]`
- `my_list.index(x)`
- `my_list.count(x)`
- `my_list.sort(cmp=None, key=None, reverse=False)`
- `list.reverse()`






Because learning changes everything.®

# Thank You!

---

**For any queries or feedback contact us at:**

 [support.india@mheducation.com](mailto:support.india@mheducation.com)

 1800-103-5875

 [www.mheducation.co.in](http://www.mheducation.co.in)

in

