

# Module-3

## ■ Arrays and Linked Structures:

- The Array Data Structure
- Operations on Arrays
- Two-Dimensional Arrays (Grids)
- Linked Structures
- Operations on Singly Linked Structures
- Variations on a Link

## ■ Searching Algorithms:

- Search for the Minimum
- Sequential Search of a List
- Binary Search of a Sorted List

## ■ Sorting Algorithms:

- Selection Sort
- Bubble Sort
- Insertion Sort

# The Array Data Structure

- The terms **data structure** and **concrete data type** refer to the internal representation of a collection's data.
- The two data structures most often used to implement collections in programming languages are **arrays** and **linked structures**. These two types of structures take different approaches **to storing and accessing data** in the computer's memory.
- These approaches in turn lead to different **space/time trade-offs** in the algorithms that manipulate the collections.
- **Arrays** are a fundamental **data structure**, and an important part of most programming languages.
- In Python, they are **containers** which are **able to store more than one item at the same time**.
- Specifically, they are an **ordered collection of elements** with every value being of the **same data type**. That is the most important thing to remember about Python arrays - the fact that **they can only hold a sequence of multiple items that are of the same type**.

# What's the Difference between Python Lists and Python Arrays?

- Lists are one of the most common data structures in Python, and a core part of the language.
- Lists and arrays behave similarly. Just like arrays, lists are an ordered sequence of elements.
- They are also mutable and not fixed in size, which means they can grow and shrink throughout the life of the program. Items can be added and removed, making them very flexible to work with.
- However, lists and arrays are not the same thing.
- Lists store items that are of various data types. This means that a list can contain integers, floating point numbers, strings, or any other Python data type, at the same time.
- That is not the case with arrays. Arrays store only items that are of the same single data type. There are arrays that contain only integers, or only floating point numbers, or only any other Python data type you want to use.
- The array.array type is just a thin wrapper on C arrays which provides space-efficient storage of basic C-style data types.
- If you need to allocate an array that you know will not change, then arrays can be faster and use less memory than lists.

Source: <https://www.freecodecamp.org/news/python-array-tutorial-define-index-methods/>

## When to Use Python Arrays

- Lists are built into the Python programming language, whereas arrays aren't.
- Arrays are not a built-in data structure, and therefore need to be imported via the array module in order to be used.
- Arrays of the array module are a thin wrapper over C arrays, and are useful when you want to work with homogeneous data.
- They are also more compact and take up less memory and space which makes them more size efficient compared to lists.
- If you want to perform mathematical calculations, then you should use NumPy arrays by importing the NumPy package.
- Besides that, you should just use Python arrays when you really need to, as lists work in a similar way and are more flexible to work with.

# How to Use Arrays in Python

- In order to create Python arrays, you will first have to import the array module which contains all the necessary functions.
- There are three ways you can import the array module:
- By using import array at the top of the file. This includes the module array. You would then go on to create an array using array.array().

## Example:

- `import array`
- `array.array()`

- Instead of having to type array.array() all the time, you could use import array as arr at the top of the file, instead of import array alone. You would then create an array by typing arr.array(). The arr acts as an alias name, with the array constructor then immediately following it.

## Example:

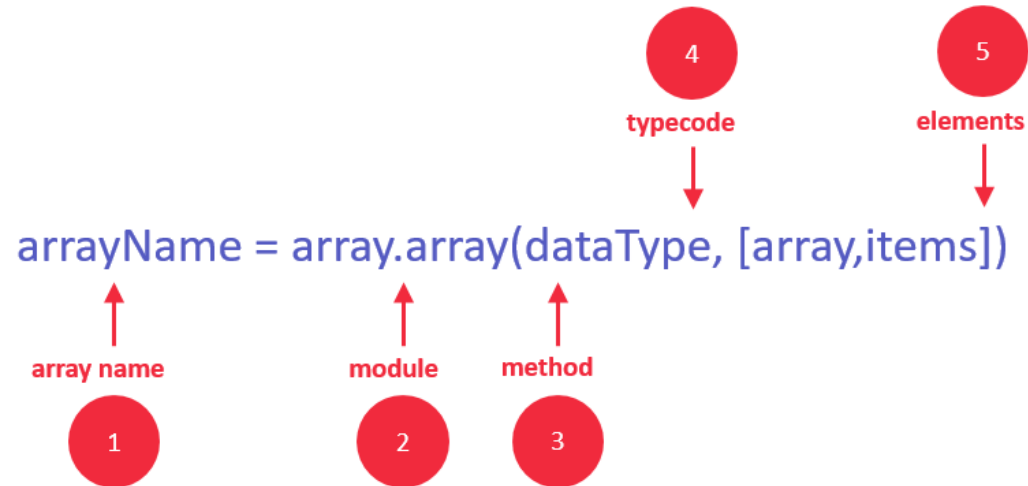
- `import array as arr`
- `arr.array()`

- Lastly, you could also use from array import \*, with \* importing all the functionalities available. You would then create an array by writing the array() constructor alone.

## Example:

- `from array import *`
- `array()`

# Array Syntax



- You can declare an array in Python while initializing it using the following syntax.

***arrayName = array.array(type code for data type, [array,items])***

- **Identifier** : specify a name like usually, you do for variables
- **Module** : Python has a special module for creating array in Python, called “array” – you must import it before using it
- **Method** : the array module has a method for initializing the array. It takes two arguments, type code, and elements.
- **Type Code** : specify the data type using the type codes available (see list below)
- **Elements** : specify the array elements within the square brackets, for example [130,450,103]

# Data Type

```
1 import array as arr
```

```
1 ex=arr.array('i',[1,2,3,4,5,6])
```

Type code	C Type	Python Type	Minimum size in bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	wchar_t	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

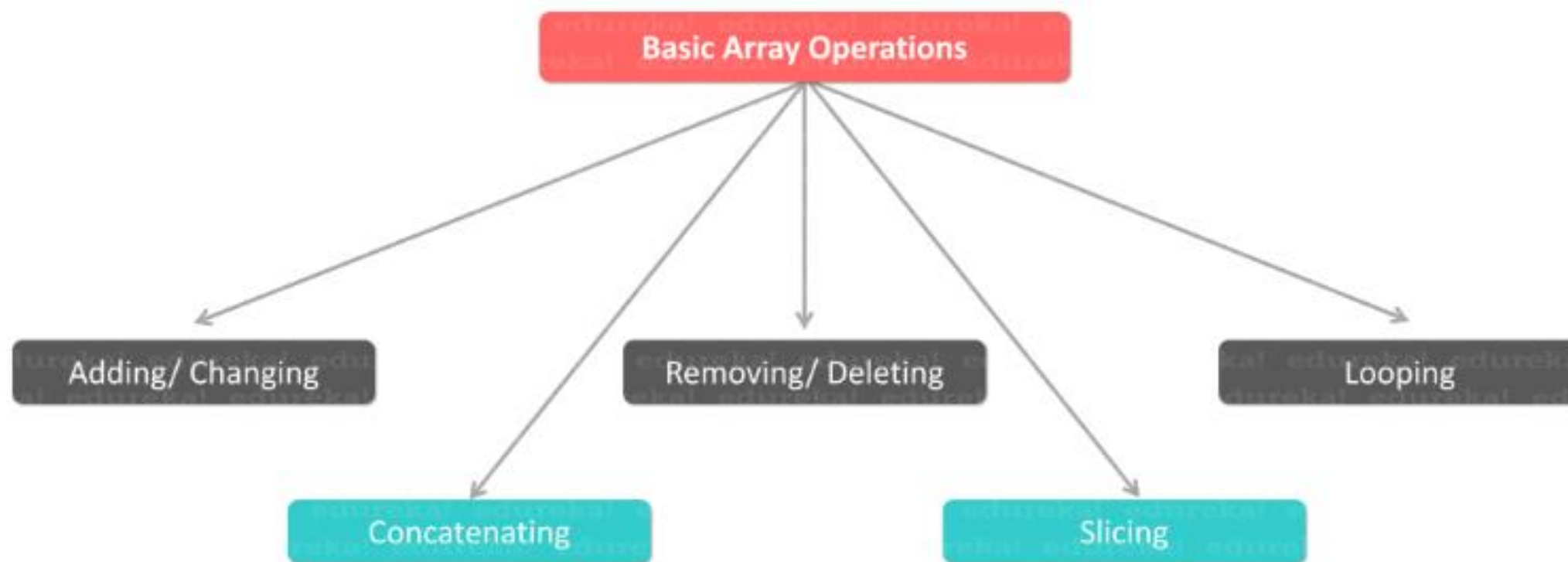
...

# Summary

- An array is a common type of data structure wherein all elements must be of the same data type.
- Python programming, an array, can be handled by the “array” module.
- Python arrays are used when you need to use many variables which are of the same type.
- In Python, array elements are accessed via indices.
- Array elements can be inserted using an `array.insert(i,x)` syntax.
- In Python, arrays are mutable.
- In Python, a developer can use `pop()` method to pop and element from Python array.
- Python array can be converted to Unicode. To fulfill this need, the array must be a type ‘u’; otherwise, you will get “ValueError”.
- Python arrays are different from lists.
- You can access any array item by using its index.
- The array module of Python has separate functions for performing array operations.



# Basic Array Operations



## Two-Dimensional Arrays (Grids)

- The arrays studied so far can represent only simple sequences of items and are called one-dimensional arrays.
- For many applications, two-dimensional arrays or grids are more useful. A table of numbers, for instance, can be implemented as a two-dimensional array.
- Suppose this grid is named `grid`. To access an item in `grid`, you use two subscripts to specify its row and column positions, remembering that indexes start at 0:

`x = grid[2][3]`    # Set `x` to 23, the value in (row 2, column 3)

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	0	1	2	3	4
Row 1	10	11	12	13	14
Row 2	20	21	22	23	24
Row 3	30	31	32	33	34

**Figure 4-5** A two-dimensional array or grid with four rows and five columns

## Processing a Grid

- For instance, the following code segment computes the **sum of all the numbers** in the variable grid.
- The outer loop iterates four times and moves down the rows.
- Each time through the outer loop, the inner loop iterates five times and moves across the columns in a different row.
- Because the methods getHeight and getWidth are used instead of the numbers 4 and 5, this code will work for a grid of any dimensions.

```
sum = 0
for row in range(grid.getHeight()):           # Go through rows
    for column in range(grid.getWidth()):      # Go through columns
        sum +=grid[row][column]
```

## Creating and Initializing a Grid

- Let's assume that there exists a Grid class for two-dimensional arrays.
- To create a Grid object, you can run the Grid constructor with three arguments: its height, its width, and an initial fill value.
- The next session instantiates Grid with 4 rows, 5 columns, and a fill value of 0. Then it prints the resulting object:

```
>>> from grid import Grid
>>> grid = Grid(4, 5, 0)
>>> print(grid)
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

- After a grid has been created, you can reset its cells to any values. The following code segment traverses the grid to reset its cells to the values shown in Figure 4-5:

```
# Go through rows
for row in range(grid.getHeight()):
    # Go through columns
    for column in range(grid.getWidth()):
        grid[row][column] = int(str(row) + str(column))
```

# Linked Structures

- After arrays, linked structures are probably the most commonly used **data structures in programs**. Like an array, a **linked structure is a concrete data type** that implements **many types of collections**, including lists.
- Recall that **array items must be stored in contiguous memory**. This means that the **logical sequence of items in the array is tightly coupled to a physical sequence of cells in memory**.
- By contrast, a **linked structure decouples the logical sequence of items** in the structure from **any ordering in memory**. That is, the **cell for a given item in a linked structure** can be found **anywhere in memory** as long as the computer can follow a **link to its address or location**.
- This kind of memory representation scheme is called **noncontiguous memory**.

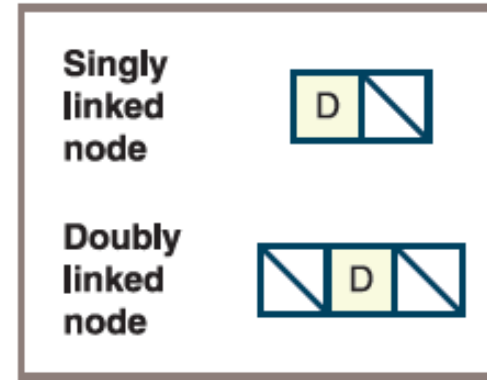
# Non-contiguous Memory and Nodes

- Linked list abstract data type that **stores collections of elements** where each element **points to the next**

- The basic unit of representation in a linked structure is a **node**.

- A **singly linked node** contains the following components or fields:

- *A data item*
- *A link to the next node in the structure*



- In addition to these components, a **doubly linked node** contains a **link to the previous node in the structure**.
- The way in which memory is allocated for linked structures is also quite unlike that of arrays and has two important consequences for insertion and removal operations:
  - *Once an insertion or removal point has been found, the insertion or removal can take place **with no shifting of data items in memory**.*
  - *The linked structure can be resized during each insertion or removal with no extra memory cost and no copying of data items.*

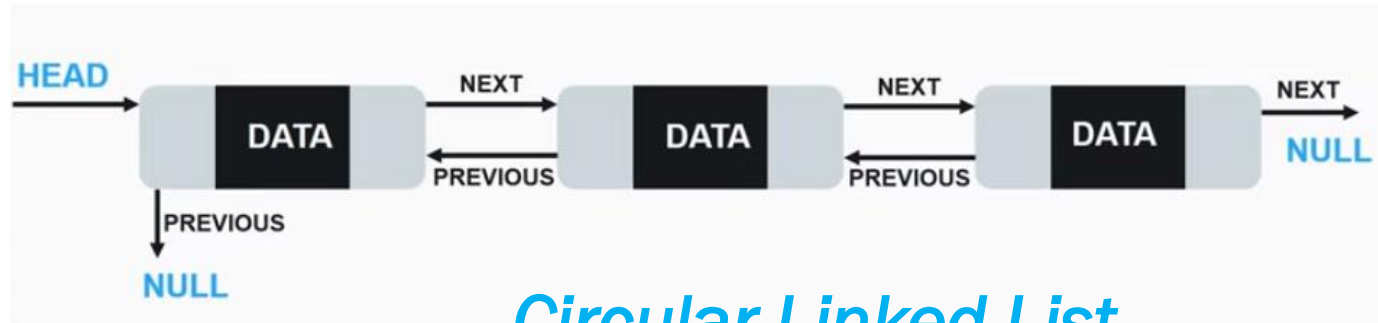
## Singly Linked List

- A **singly linked node** contains the fields : A data item and a link to the next node in the structure



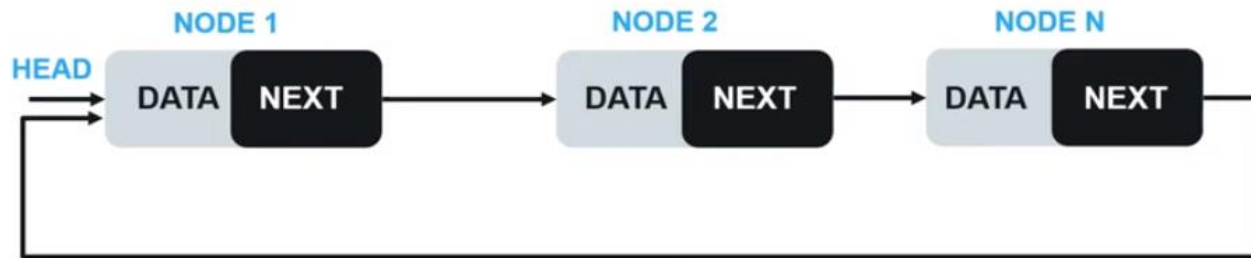
## Doubly Linked List

- Doubly linked list has an additional reference attribute pointing to the previous node



## Circular Linked List

- The last node of Circular Linked list points to the head instead of null



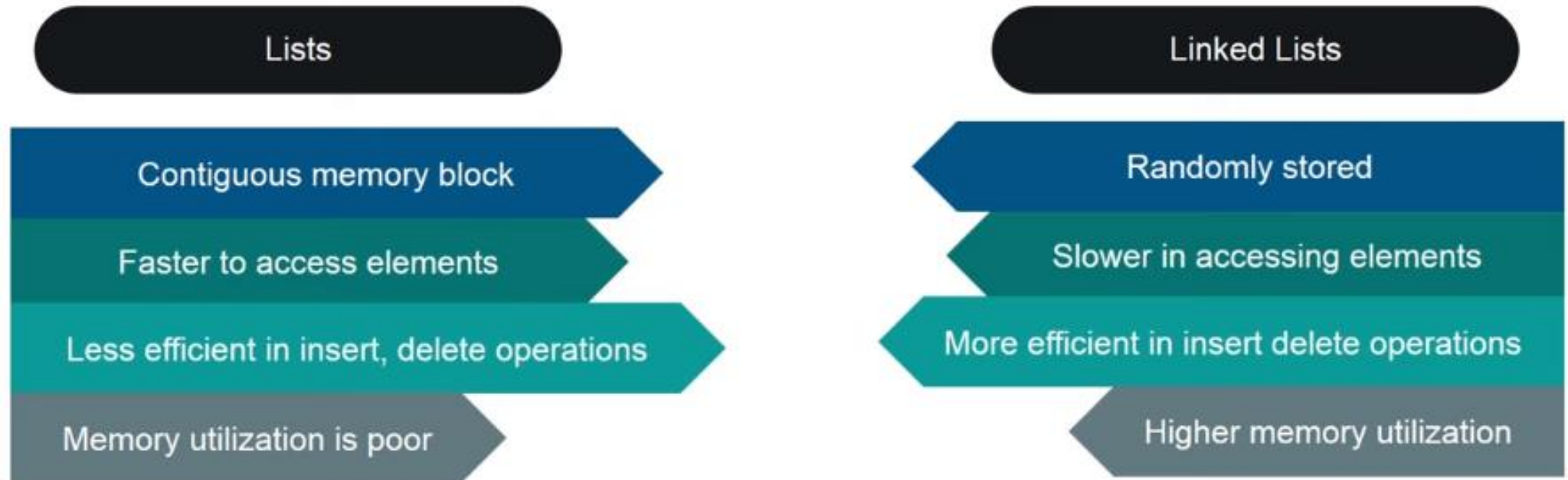
# Linked List

- Python programmers **set up nodes and linked structures** by using **references to objects**.
- In Python, **any variable can refer to anything**, including the **value None**, which can mean an empty link.
- Thus, a Python programmer defines a **singly linked node by defining an object that contains two fields**:
  - *a reference to a data item and*
  - *a reference to another node.*
- Python provides **dynamic allocation of non-contiguous memory** for each **new node object**, as well as automatic return of this memory to the system (garbage collection) when the object no longer can be referenced by the application.



# *Lists vs Linked Lists*

- Linked list provides an implementation of Queues and Stacks in Python




Source : Eduraka

## Example

`stock_prices = [298,305,320,301,292]`

`stock_prices`



0x00500	298
0x00504	305
0x00508	320
0x0050A	301
0x0050F	292

`stock_prices.insert(1, 284)`

298
284
305
320
301
292

```
stock_prices = []
```

```
stock_prices.append(298)
```

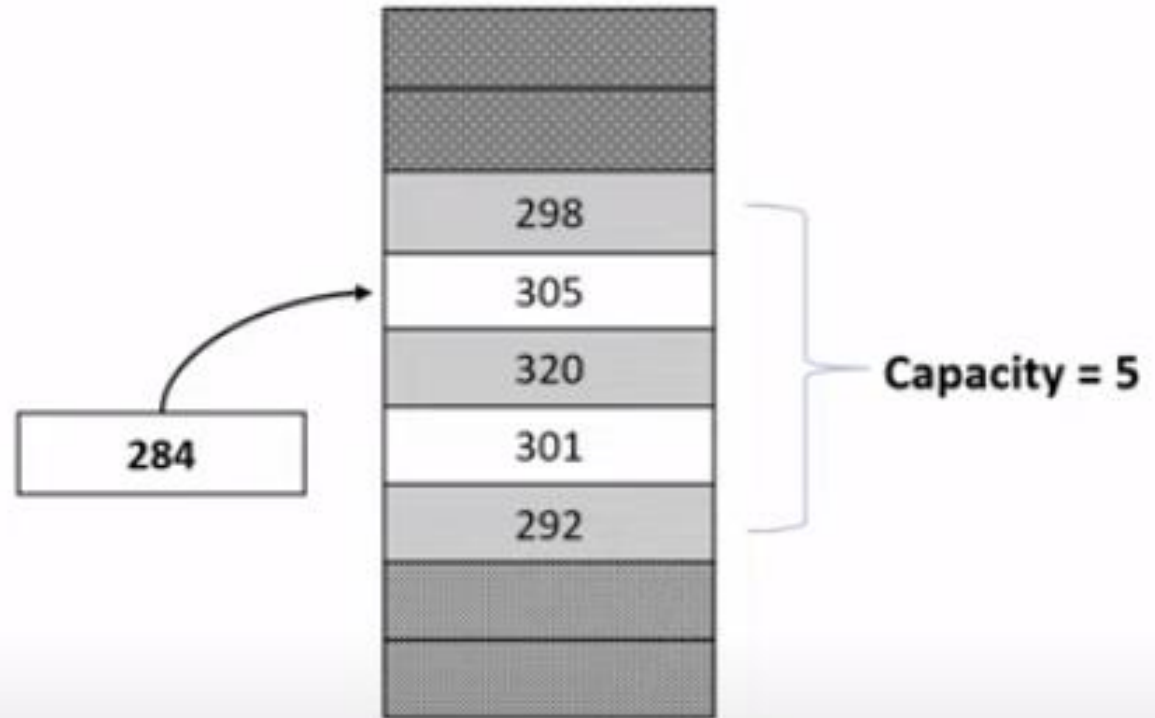
```
stock_prices.append(305)
```

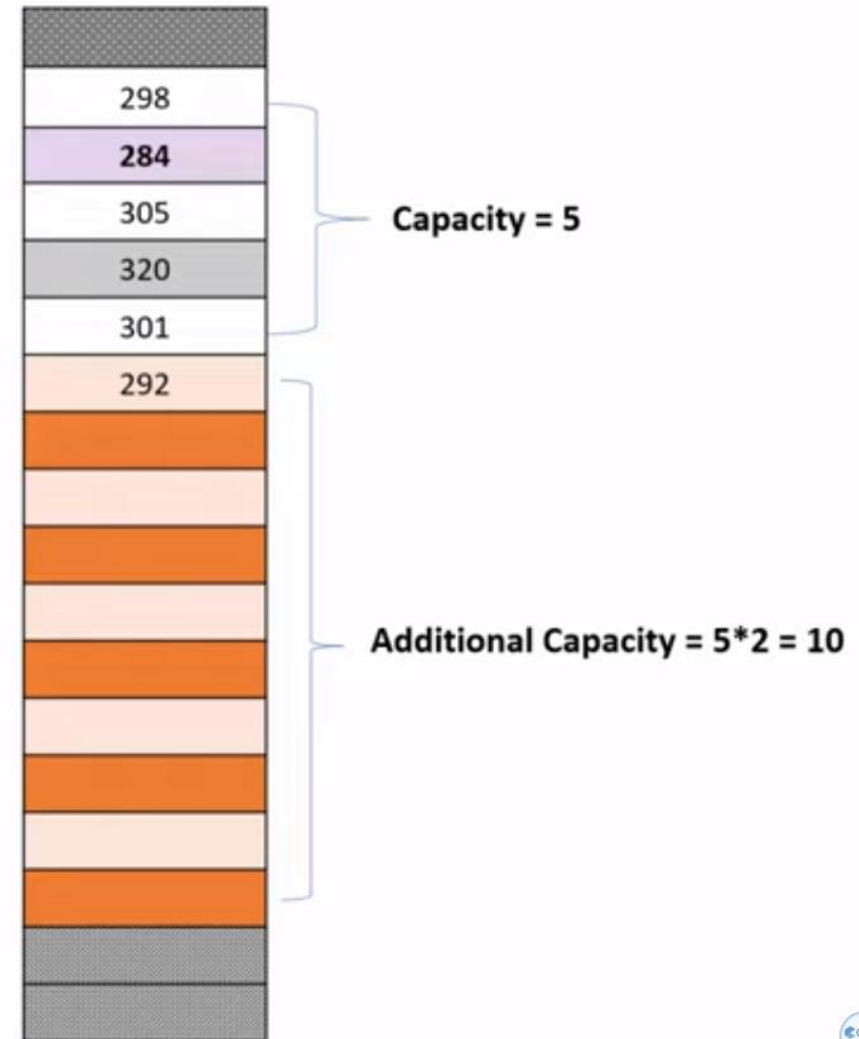
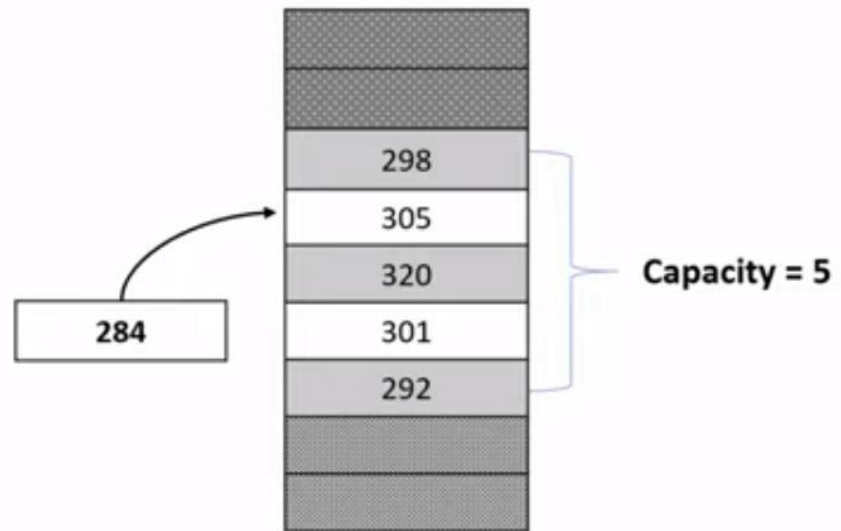
```
stock_prices.append(320)
```

```
stock_prices.append(301)
```

```
stock_prices.append(292)
```

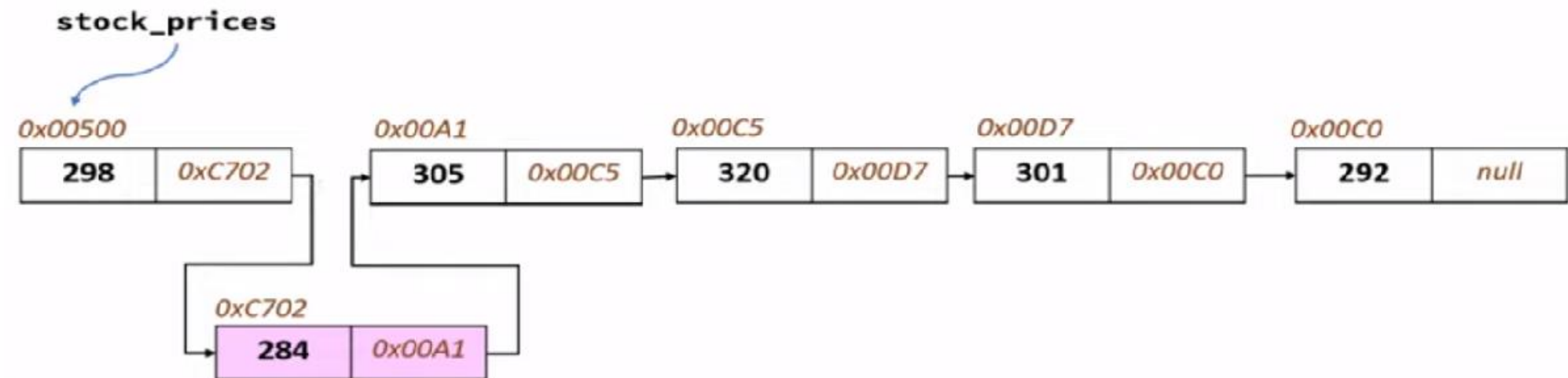
```
stock_prices.insert(1,284)
```





# Example

- Linked list has two main benefits over an array,
  - *You don't need to preallocate space*
  - *Insertion is easier*



# Implement a Linked List

- Deque from collections module in Python is a linked list implementation to get a Stack and Queue data type in python

Queue follows FIFO Approach



Stack follows LIFO Approach



# Custom Linked List

- We can create a custom linked list. Provide it fulfils that properties of a linked list completely



## Single Linked List



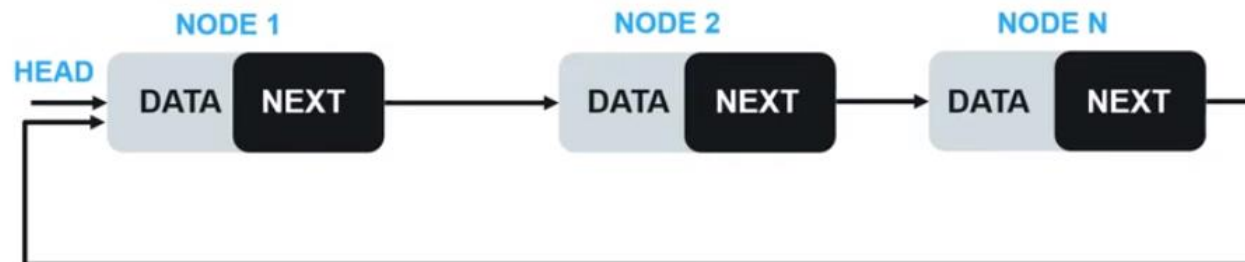
## Doubly Linked List

- Doubly linked list has an additional reference attribute pointing to the previous node



## Circular Linked List

- The last node of Circular Linked list points to the head instead of null





# A Circular Linked Structure and Doubly Linked Structures

- A circular linked structure contains a link from the last node back to the first node in the structure.
- There is always at least one node in this implementation. This node, the dummy header node, contains no data but serves as a marker for the beginning and the end of the linked structure.
- Initially, in an empty linked structure, the head variable points to the dummy header node, and the dummy header node's next pointer points back to the dummy header node itself.
- A doubly linked structure has the advantages of a singly linked structure.
- In addition, it allows the user to do the following:
  - *Move left, to the previous node, from a given node.*
  - *Move immediately to the last node.*

# Summary

- The main advantage of the singly linked structure over the array is not time performance but memory performance.
- Resizing an array, when this must occur, is linear in time and memory.
- Resizing a linked structure, which occurs upon each insertion or removal, is constant in time and memory. Moreover, no memory ever goes to waste in a linked structure.
- The physical size of the structure never exceeds the logical size.
- Linked structures do have an extra memory cost in that a singly linked structure must use  $n$  cells of memory for the pointers. This cost increases for doubly linked structures, whose nodes have two links.

## comparison

	Array	Linked List
Indexing	$O(1)$	$O(n)$
Insert/Delete Element At Start	$O(n)$	$O(1)$
Insert/Delete Element At End	$O(1)$ - amortized	$O(n)$
Insert Element in Middle	$O(n)$	$O(n)$

# Search Algorithms

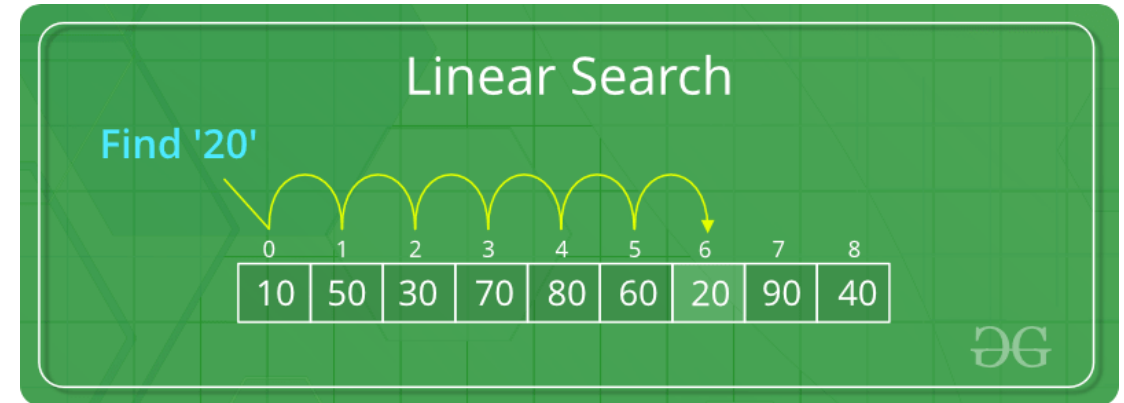
## Search for the Minimum

- Python's `min` function returns the minimum or smallest item in a list.
- To study the complexity of this algorithm, you'll develop an alternative version that returns the index of the minimum item.
- The algorithm begins by treating the first position as that of the minimum item.
- It then searches to the right for an item that is smaller and, if it is found, resets the position of the minimum item to the current position.
- When the algorithm reaches the end of the list, it returns the position of the minimum item.
- Thus, the algorithm must make  $n - 1$  comparisons for a list of size  $n$ .

```
def indexOfMin(lyst):  
    """Returns the index of the minimum item."""  
    minIndex = 0  
    currentIndex = 1  
    while currentIndex < len(lyst):  
        if lyst[currentIndex] < lyst[minIndex]:  
            minIndex = currentIndex  
        currentIndex += 1  
    return minIndex
```

# Sequential or Linear Search of a List

- Python's `in` operator is implemented as a method named `__contains__` in the list class. This method searches for a particular item (called the **target item**) within a list of arbitrarily arranged items.
- In such a list, the only way to search for a target item is to **begin with the item at the first position** and **compare it to the target**.
- If the **items are equal**, the **method returns True**. Otherwise, the **method moves on to the next position** and **compares its item with the target**.
- If the **method arrives at the last position** and **still cannot find the target**, it **returns False**. This kind of search is called a **sequential search** or a **linear search**.
- A more useful sequential search function would return the index of a target if it's found, or -1 otherwise.



```
def sequentialSearch(target, lyst):  
    """Returns the position of the target item if found,  
    or -1 otherwise."""  
    position = 0  
    while position < len(lyst):  
        if target == lyst[position]:  
            return position  
        position += 1  
    return -1
```

# Binary Search of a Sorted List

- A sequential search is necessary for data that are **not arranged in any particular order**. When searching sorted data, you can use a binary search.
- Assume that the items in the **list are sorted in ascending order** (as they are in a phone book).
- The search algorithm goes directly to the **middle position in the list** and **compares the item at that position to the target**.
- **If there is a match, the algorithm returns the position**. Otherwise, if the target is **less than the current item**, the algorithm searches the portion of the list **before the middle position**.
- If the target is **greater than the current item**, the algorithm searches the portion of the list **after the middle position**.
- The **search process stops when the target is found**, or the current beginning position is greater than the current ending position.

# Example

Item to be searched = 23

Step 1 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

$a[mid] = 13$   
 $13 < 23$   
 $beg = mid + 1 = 5$   
 $end = 8$   
 $mid = (beg + end)/2 = 13 / 2 = 6$

Step 2 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

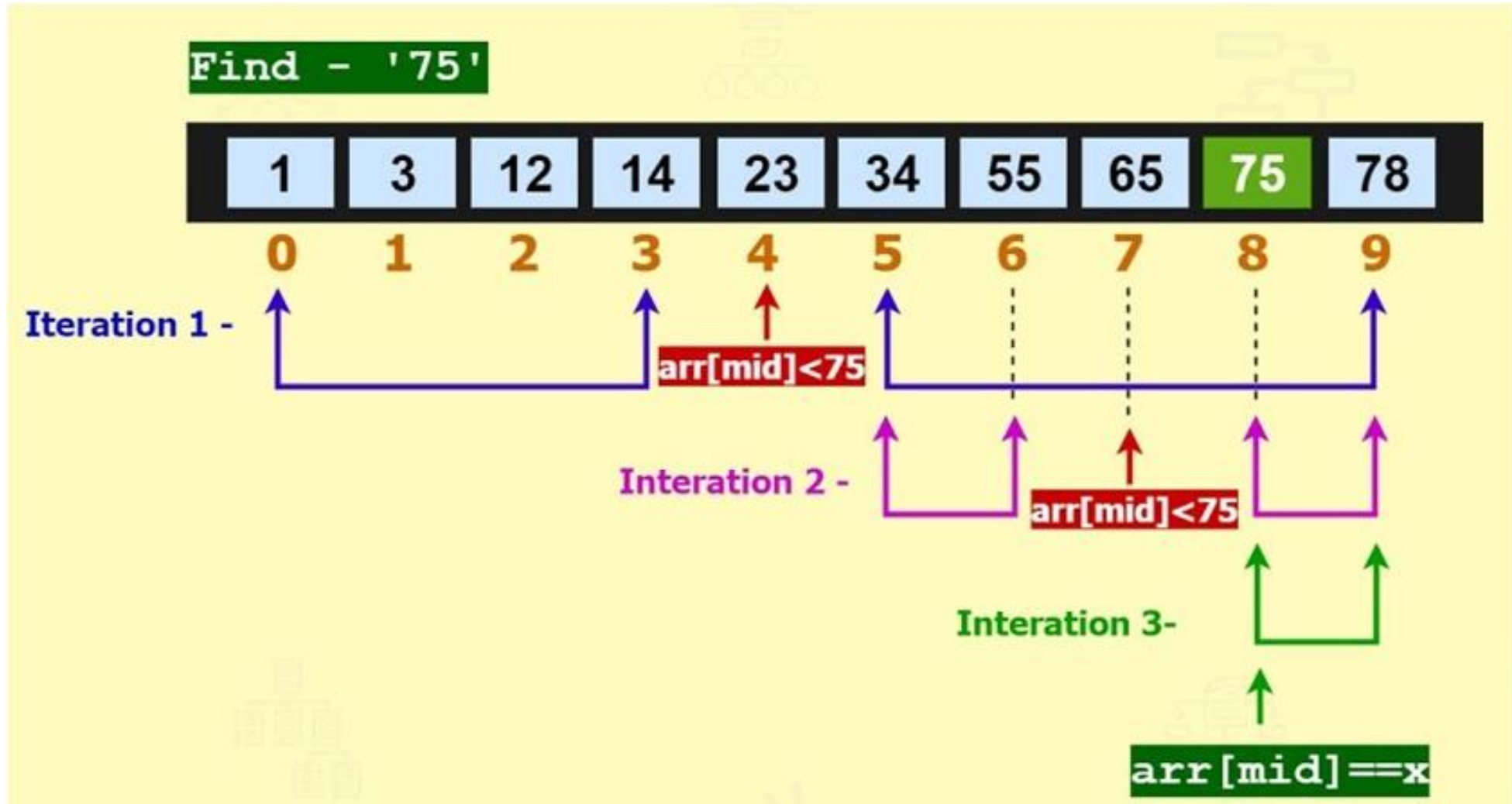
$a[mid] = 20$   
 $20 < 23$   
 $beg = mid + 1 = 7$   
 $end = 8$   
 $mid = (beg + end)/2 = 15 / 2 = 7$

Step 3 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

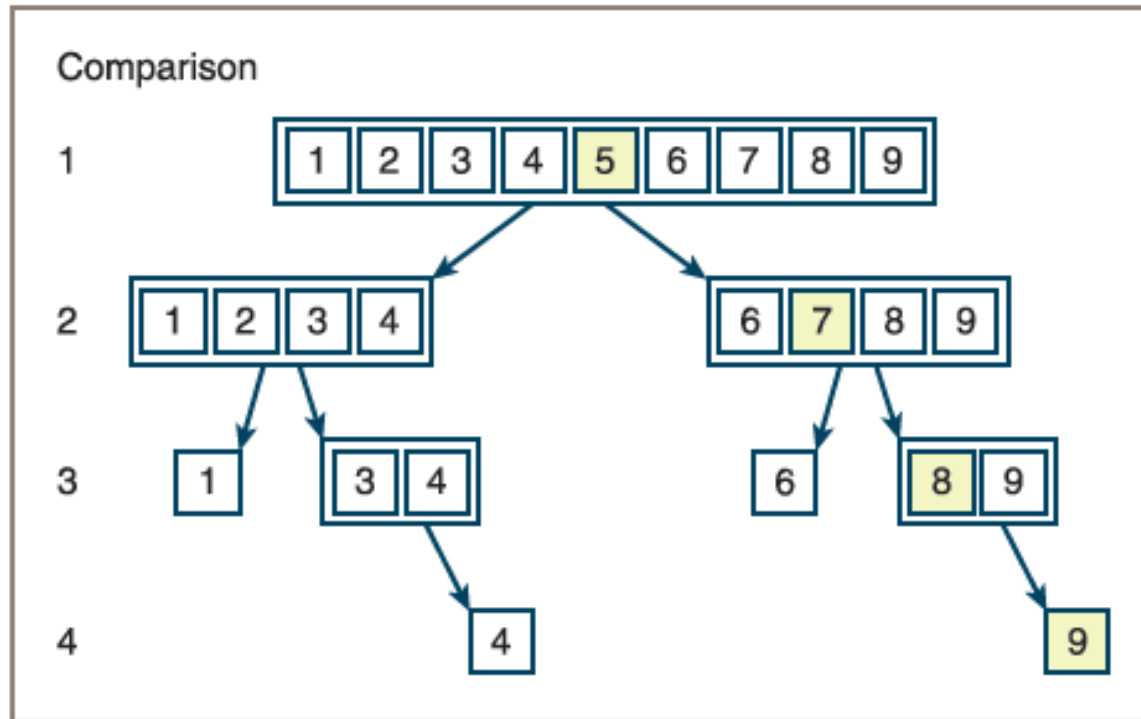
$a[mid] = 23$   
 $23 = 23$   
 $loc = mid$

# Example





# Binary Search



Here is the code for the binary search function:

```
def binarySearch(target, sortedLyst):
    left = 0
    right = len(sortedLyst) - 1
    while left <= right:
        midpoint = (left + right) // 2
        if target == sortedLyst[midpoint]:
            return midpoint
        elif target < sortedLyst[midpoint]:
            right = midpoint - 1
        else:
            left = midpoint + 1
    return -1
```

- Binary search is certainly more efficient than sequential search. However, the kind of search algorithm you choose depends on the organization of the data in the list.
- There is an additional overall cost to a binary search, having to do with keeping the list in sorted order.

# Basic Sort Algorithms

- Each of the Python sort functions that are developed operates on a list of integers and uses a swap function to exchange the positions of two items in the list. Here is the code for that function:

```
def swap(lyst, i, j):  
    """Exchanges the items at positions i and j."""  
    # You could say lyst[i], lyst[j] = lyst[j], lyst[i]  
    # but the following code shows what is really going on  
    temp = lyst[i]  
    lyst[i] = lyst[j]  
    lyst[j] = temp
```

# Selection Sort

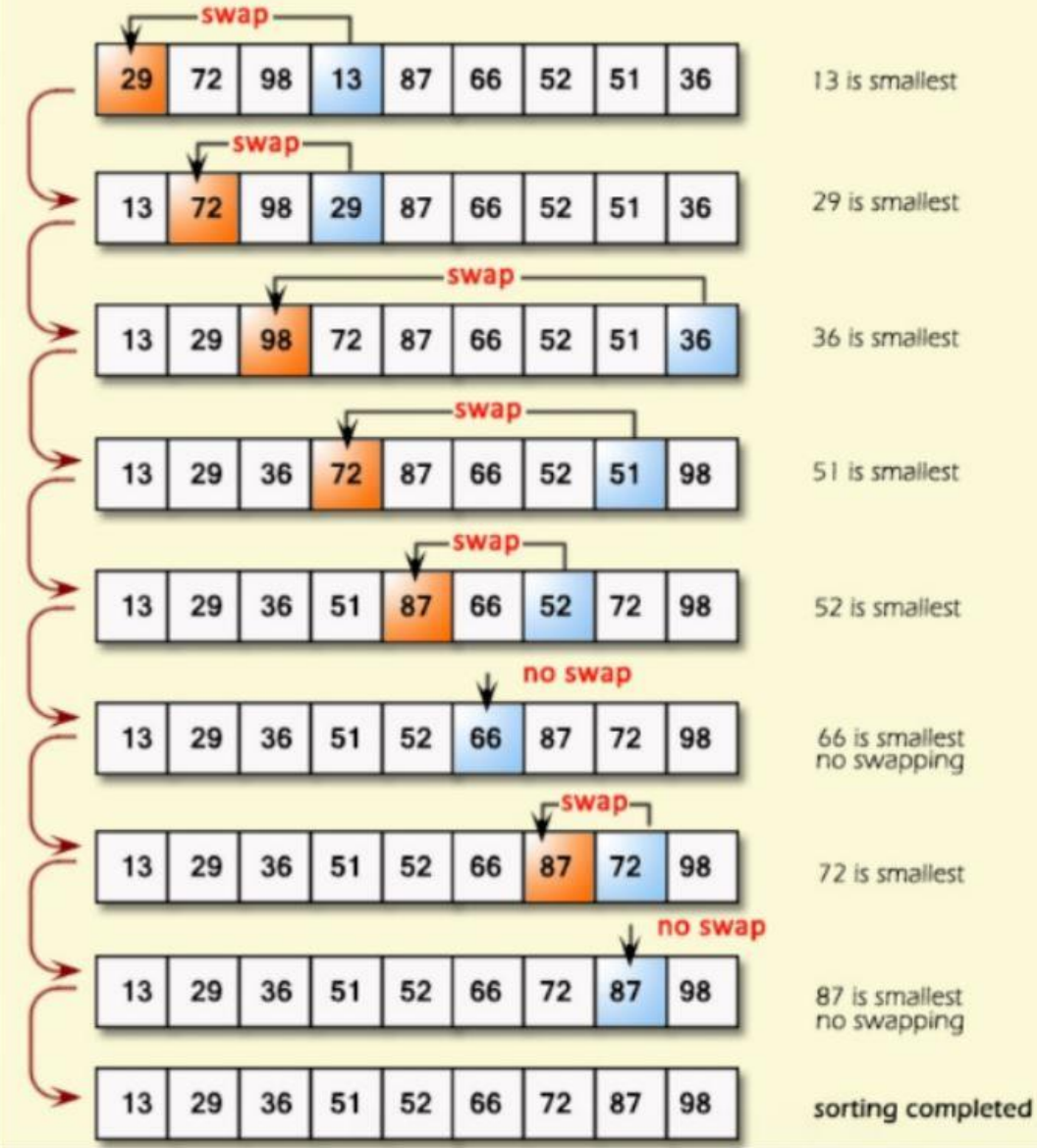
- Search the entire list for the position of the smallest item.
- If that position does not equal the first position, the algorithm swaps the items at those positions.
- The algorithm then returns to the second position and repeats this process, swapping the smallest item with the item at the second position, if necessary.
- When the algorithm reaches the last position in the overall process, the list is sorted.
- The algorithm is called **selection sort** because each pass through the main loop selects a single item to be moved.
- Figure 3-8 shows the states of a list of five items after each search and swap pass of a selection sort. c
- The two items just swapped on each pass have asterisks next to them, and the sorted portion of the list is shaded.

UNSORTED LIST	AFTER 1st PASS	AFTER 2nd PASS	AFTER 3rd PASS	AFTER 4th PASS
5	1*	1	1	1
3	3	2*	2	2
1	5*	5	3*	3
2	2	3*	5*	4*
4	4	4	4	5*

**Figure 3-8** A trace of data during a selection sort

```
def selectionSort(lyst):
    i = 0
    while i < len(lyst) - 1:           # Do n - 1 searches
        minIndex = i                 # for the smallest
        j = i + 1
        while j < len(lyst):         # Start a search
            if lyst[j] < lyst[minIndex]:
                minIndex = j
            j += 1
        if minIndex != i:            # Exchange if needed
            swap(lyst, minIndex, i)
        i += 1
```

## Selection Sort



# Bubble Sort

- Its strategy is to start at the beginning of the list and compare pairs of data items as it moves down to the end.
- Each time the items in the pair are out of order, the algorithm swaps them. This process has the effect of bubbling the largest items to the end of the list.
- The algorithm then repeats the process from the beginning of the list and goes to the next-to-last item, and so on, until it begins with the last item.
- At that point, the list is sorted.
- Figure 3-9 shows a trace of the bubbling process through a list of five items.
- This process makes four passes through a nested loop to bubble the largest item down to the end of the list.
- Once again, the items just swapped are marked with asterisks, and the sorted portion is shaded.

UNSORTED LIST	AFTER 1st PASS	AFTER 2nd PASS	AFTER 3rd PASS	AFTER 4th PASS
5	4*	4	4	4
4	5*	2*	2	2
2	2	5*	1*	1
1	1	1	5*	3*
3	3	3	3	5*

**Figure 3-9** A trace of data during a bubble sort

```
def bubbleSort(lyst):  
    n = len(lyst)  
    while n > 1:  
        i = 1  
        while i < n:  
            if lyst[i] < lyst[i - 1]:  
                swap(lyst, i, i - 1)  
            i += 1  
        n -= 1
```

# Do n - 1 bubbles  
# Start each bubble  
# Exchange if needed

# Example

**n = 8**

0	1	2	3	4	5	6	7
6	5	3	1	8	7	2	4
5	6	3	1	8	7	2	4
5	3	6	1	8	7	2	4
5	3	1	6	8	7	2	4
5	3	1	6	8	7	2	4
5	3	1	6	7	8	2	4
5	3	1	6	7	2	8	4

j=1  
j=2  
j=3  
j=4  
j=5  
j=6  
j=7

**n = 7**

0	1	2	3	4	5	6	7
5	3	1	6	7	2	4	8
3	5	1	6	7	2	4	8
3	1	5	6	7	2	4	8
3	1	5	6	7	2	4	8
3	1	5	6	7	2	4	8
3	1	5	6	2	7	4	8

j=1  
j=2  
j=3  
j=4  
j=5  
j=6

**n = 6**

0	1	2	3	4	5	6	7
3	1	5	6	2	4	7	8
1	3	5	6	2	4	7	8
1	3	5	6	2	4	7	8
1	3	5	6	2	4	7	8
1	3	5	2	6	4	7	8

j=1  
j=2  
j=3  
j=4  
j=5

**n = 5**

0	1	2	3	4	5	6	7
1	3	5	2	4	6	7	8
1	3	5	2	4	6	7	8
1	3	5	2	4	6	7	8
1	3	2	5	4	6	7	8

j=1  
j=2  
j=3  
j=4

**n = 4**

0	1	2	3	4	5	6	7
1	3	2	4	5	6	7	8
1	3	2	4	5	6	7	8
1	2	3	4	5	6	7	8

j=1  
j=2  
j=3

**n = 3**

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

j=1  
j=2

**n = 2**

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

j=1

# Insertion Sort

- On the  $i$ th pass through the list, where  $i$  ranges from 1 to  $n - 1$ , the  $i$ th item should be inserted into its proper place among the first  $i$  items in the list.
- After the  $i$ th pass, the first  $i$  items should be in sorted order.
- This process is analogous to the way in which many people organize playing cards in their hands. That is, if you hold the first  $i - 1$  cards in order, you pick the  $i$ th card and compare it to these cards until its proper spot is found.
- As with our other sort algorithms, insertion sort consists of two loops. The outer loop traverses the positions from 1 to  $n - 1$ .
- For each position  $i$  in this loop, you save the item and start the inner loop at position  $i - 1$ . For each position  $j$  in this loop, you move the item to position  $j + 1$  until you find the insertion point for the saved ( $i$ th) item.
- Figure 3-10 shows the states of a list of five items after each pass through the outer loop of an insertion sort. The item to be inserted on the next pass is marked with an arrow; after it is inserted, this item is marked with an asterisk.

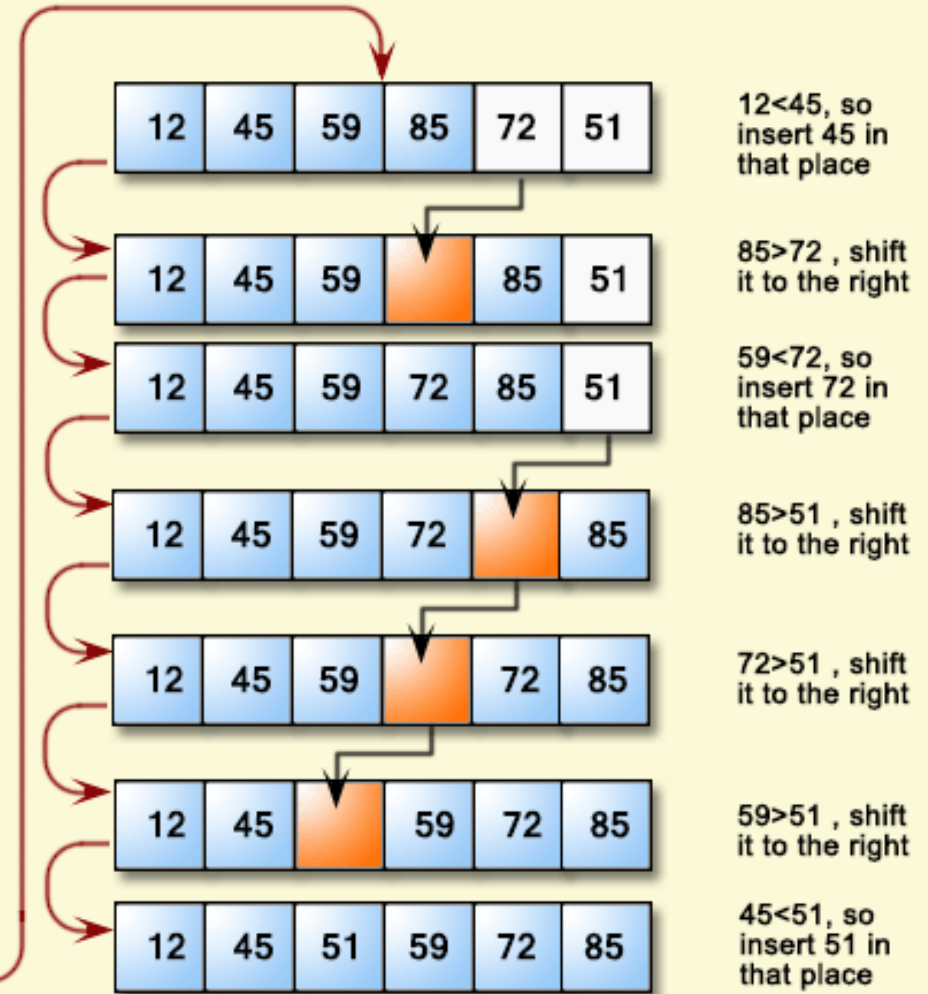
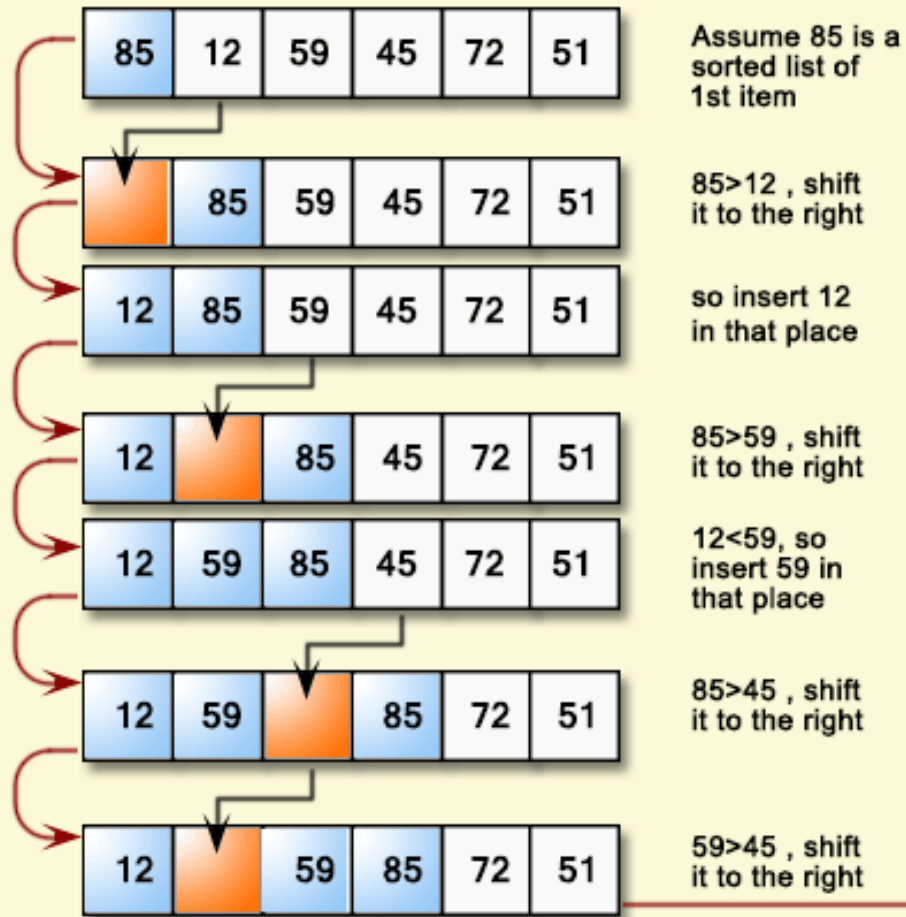
UNSORTED LIST	AFTER 1st PASS	AFTER 2nd PASS	AFTER 3rd PASS	AFTER 4th PASS
2	2	1*	1	1
5 ←	5 (no insertion)	2	2	2
1	1 ←	5	4*	3*
4	4	4 ←	5	4
3	3	3	3 ←	5

**Figure 3-10** A trace of data during insertion sort

```
def insertionSort(lyst):
    i = 1
    while i < len(lyst):
        itemToInsert = lyst[i]
        j = i - 1
        while j >= 0:
            if itemToInsert < lyst[j]:
                lyst[j + 1] = lyst[j]
                j -= 1
            else:
                break
        lyst[j + 1] = itemToInsert
        i += 1
```



# Insertion Sort



© w3resource.com