List Manipulation

Lecture 12

Department of Computer Science and Engineering, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

- In several applications, we need to sort data or search for some item of interest in the given data.
- Searching is more efficient when data is sorted.
- The process of arranging data in ascending/descending order is called **sorting**.
- The attribute or property of data that forms the basis of sorting is called the **key**.

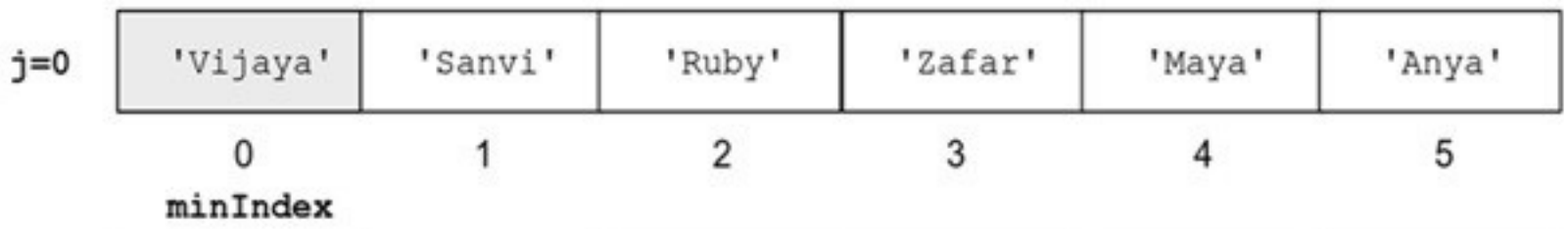
Selection Sort

- To begin with, we find the lexicographically smallest value in the list and interchange this entry in the list with the first value in the list.
- Next, we find the entry with the smallest value out of the remaining entries, and interchange it with the second value in the list, and proceed in this manner.
- If there are n values, only n-1 values need to be placed in order because when (n-1) values have been put in order, then the nth value would automatically be in order.
- This technique of sorting is called **Selection sort**.

Selection sort (Contd.)

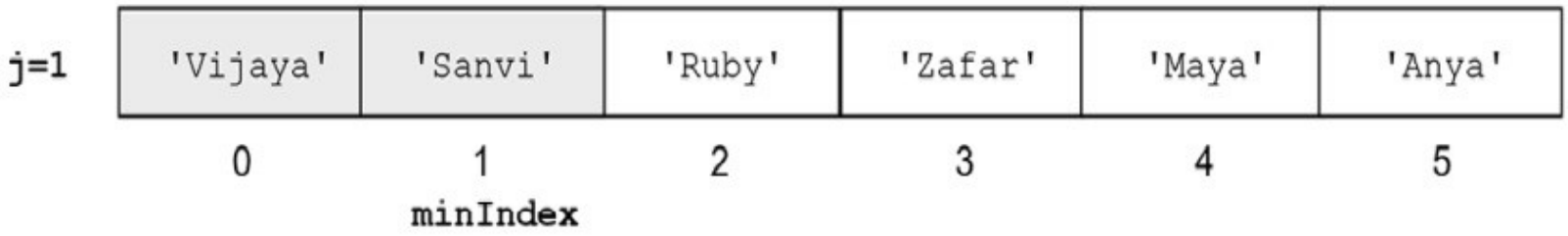
In the first iteration (i=0)

- We find the smallest name in the list.
- To do this, we begin with the name at index j=0.
- We keep track of the index of the smallest name examined so far using the variable *minIndex*.
- When j=0, *minIndex* is equal to 0

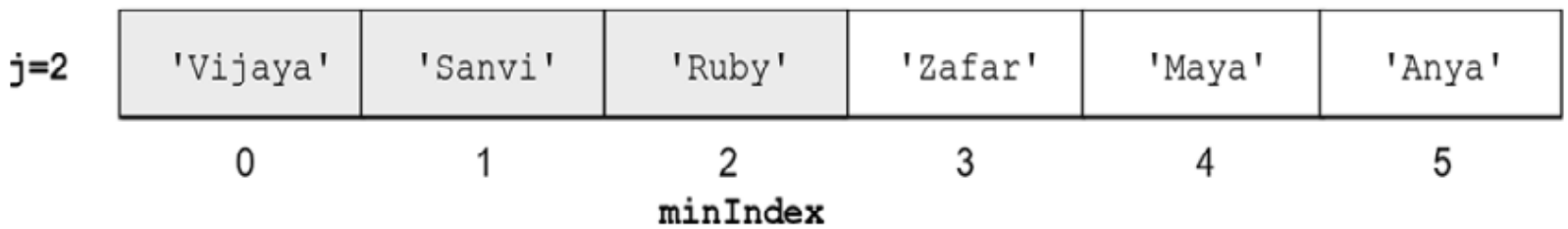


Selection sort (Contd.)

- Compare value at index j=1 with the value at index *minIndex*. Since 'Sanvi' is smaller than 'Vijaya', *minIndex* will be set equal to 1.



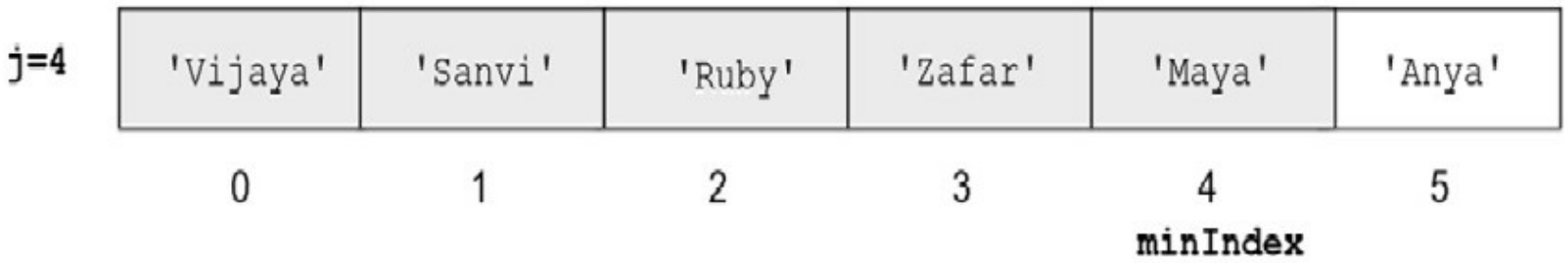
- Now we compare the next value which is at index j=2 with the value at *minIndex*.
- Since 'Ruby' is smaller than 'Sanvi', we will set *minIndex* to 2.



Selection sort (Contd.)

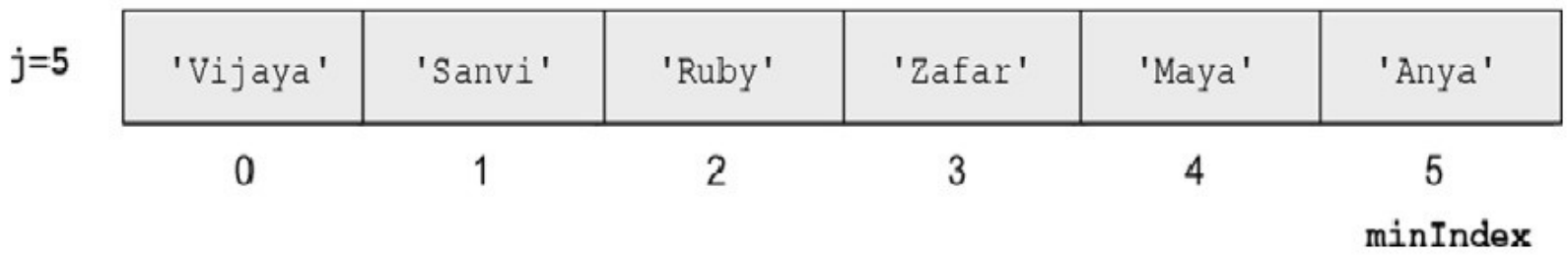
- When j=3, 'Zafar' will be compared with 'Ruby' (value at *minIndex* 2).
- Since 'Zafar' is greater than 'Ruby', the value of *minIndex* will remain unchanged.

- When j=4, 'Maya' will be compared with 'Ruby' (value at *minIndex* 2), and *minIndex* will be set to 4 since 'Maya' is smaller than 'Ruby'.

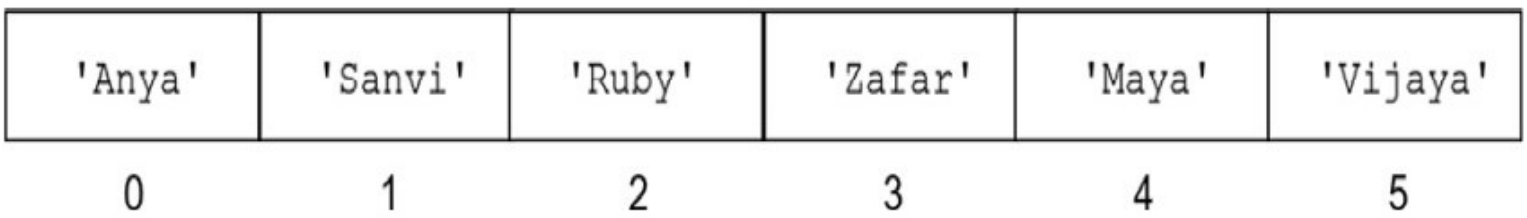


Selection Sort (Contd.)

- Finally, when j=5, 'Anya' is found to be smaller than 'Maya' (value at *minIndex* 4).
- Thus, after scanning the entire list, minIndex will have value 5.



- Now, we need to exchange this smallest name entry 'Anya' at index 5 (*minIndex*) with the name at index i=0.
- Modified data **at the end of first iteration** looks like:



Selection Sort Code

```
'''Selection Sort'''
def selectionSort(lst):
    for i in range(0,len(lst)-1):
        minIndex=i
        for j in range(i+1,len(lst)):
            if lst[j]<lst[minIndex]:
                    minIndex=j
        if minIndex!=i:
            lst[minIndex],lst[i]=lst[i],lst[minIndex]
def main():
    lst=eval(input('Enter a list :'))
    print('Sorted list :')
    selectionSort(lst)
    print(lst)
if __name__=='__main__':
    main()
```

Selection Sort (Contd.)

- Since a list (like other objects) is passed by reference in Python, and the function only modifies the components of the list, **there is no need to return the sorted list.**
- The function selectionSort works equally well on numeric and string data.

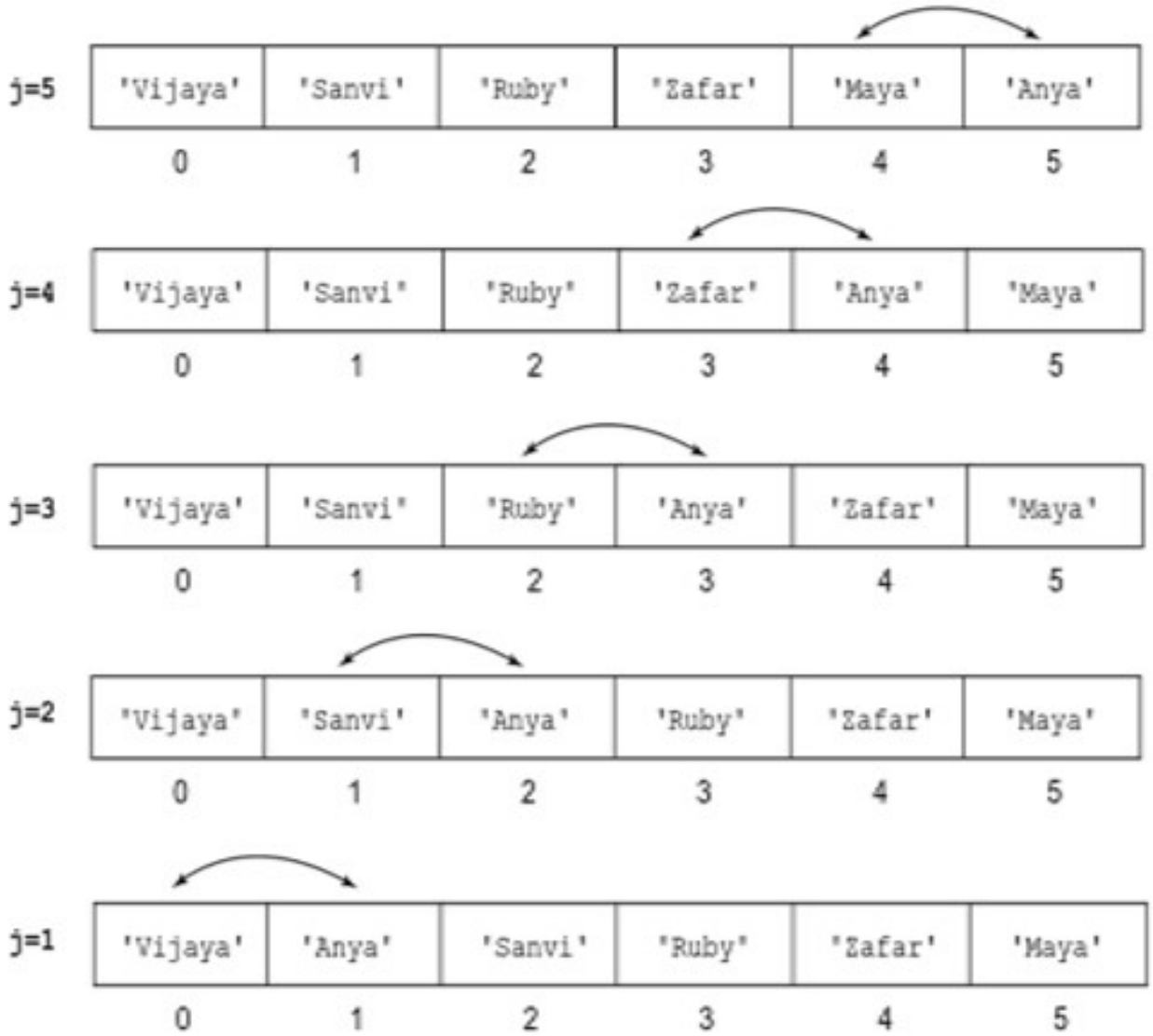
Bubble Sort

- In each pass, we compare values in adjacent positions and interchange them, if they are out of order.
- To begin with, we have a list of n values.
- The nth value and (n-1)th value are compared and interchanged if the nth value is smaller than (n-1)th value, then (n-1)th value and (n-2)th value are compared and interchanged if found out of order, and so on.

Bubble Sort (Contd.)

- In the first pass, the smallest value will move to the front of the list at index 0; on subsequent passes, it will be ignored.
- In the second iteration, we need to sort the remaining list of n-1 values excluding the value at index 0.
- After n-1 iterations, the list will be completely sorted, and the algorithm will halt.
- We start at index 5 and keep on comparing adjacent locations in order to move the smallest name to the beginning of the list.

The steps performed in first iteration are as follows:



Bubble Sort Code

```
1  def bubbleSort(lst):
2      n=len(lst)-1
3      for i in range(0,n):
4          for j in range(n,i,-1):
5              if lst[j]<lst[j-1]:
6                  lst[j],lst[j-1]=lst[j-1],lst[j]
7  def main():
8      lst=eval(input('Enter the list :'))
9      print('Sorted list :')
10     bubbleSort(lst)
11     print(lst)
12 if __name__=='__main__':
13     main()
```

Bubble Sort (Modified)

- If not even a single pair of data is swapped in an iteration, we conclude that the list has already been sorted.
- The control should break out of the loop, instead of unnecessarily proceeding with the remaining iterations.
- So, we modify the Bubble sort code to include this possibility and reduce the time.

Bubble Sort Code

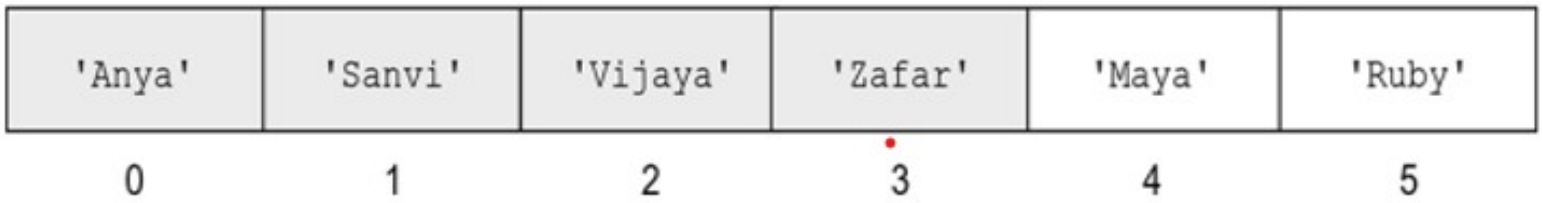
```
1  def bubbleSort(lst):
2      n=len(lst)-1
3      for i in range(0,n):
4          swap=False
5          for j in range(n,i,-1):
6              if lst[j]<lst[j-1]:
7                  swap=True
8                  lst[j],lst[j-1]=lst[j-1],lst[j]
9          if swap==False:
10             break
11 def main():
12     lst=eval(input('Enter the list :'))
13     print('Sorted list :')
14     bubbleSort(lst)
15     print(lst)
16 if __name__=='__main__':
17     main()
```

Insertion Sort

- The list is logically divided into two parts.
- The left part is the sorted part, the right part is the unsorted part comprising the elements yet to be arranged in sorted order.
- In each iteration, we increase the length of the sorted part by one in the following manner:
  - Insert the first element from the unsorted part into the sorted part at the correct position.
- To find the correct position of the value to be inserted, we compare it with values in the sorted part (starting from the rightmost value in the sorted part) and shift each value to the right by one position, until the correct position is found.

Insertion Sort (Contd.)

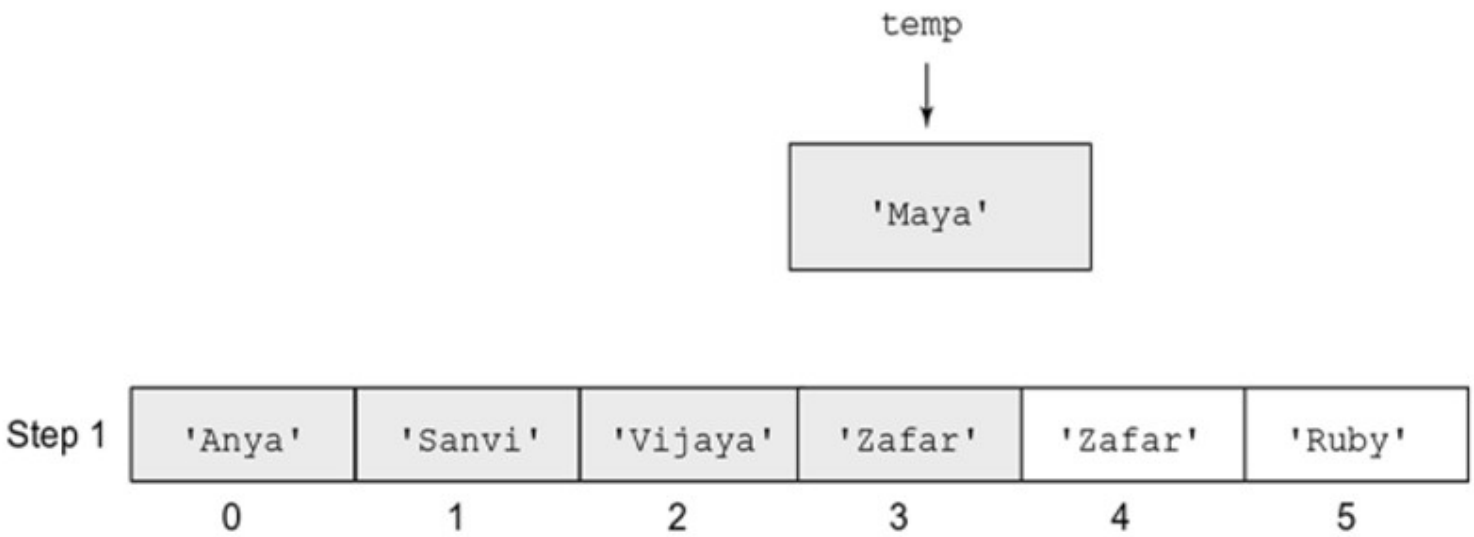
- Examine the list below:



- In this example, the left sorted part (grey portion) comprises elements at indices 0, 1, 2, 3
- The right unsorted part (white portion) comprises the elements at indices 4 and 5.

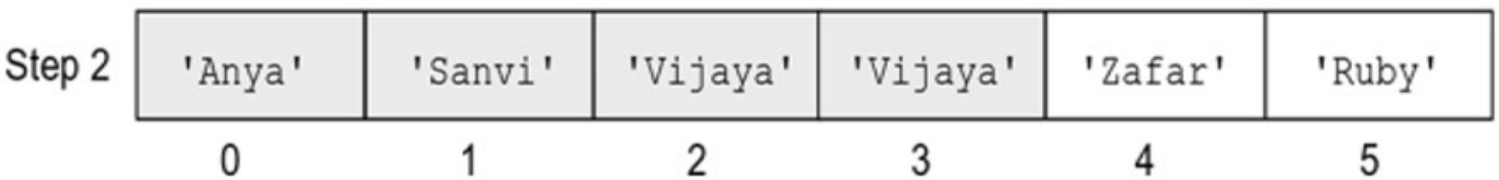
Insertion Sort (Contd.)

- We need to insert element *lst[4]* at the correct position.
- As 'Maya' is less than 'Zafar', we shift 'Zafar' one position right, but before we do so, we must save 'Maya' in a temporary variable, say, temp.
- The modified list is



Insertion Sort (Contd.)

- Next, we compare the value of the variable temp, i.e. 'Maya' with *lst[2]*, i.e., 'Vijaya'.
- As 'Maya' is less than 'Vijaya', we shift 'Vijaya' one position right.
- The modified list is

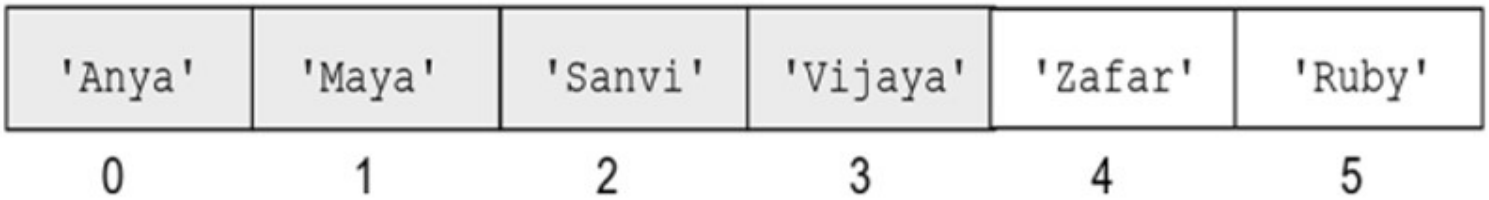


Insertion Sort (Contd.)

- Next, we compare the value of the variable temp, i.e. 'Maya' with *lst[1]*, i.e. 'Sanvi'.
- As 'Maya' is less than 'Sanvi', we shift 'Sanvi' one position right.
- The modified list is

Insertion Sort (Contd.)

- Next, we compare, 'Maya' with *lst[0]*, i.e. 'Anya'.
- As 'Maya' is greater than 'Anya', we do not need to shift 'Anya', and 'Maya' can now be placed at index 1.
- The modified list is



- Now 'Maya' has been placed at its correct position.

Insertion Sort Code

```
1  def insertionSort(lst):
2      for i in range(1,len(lst)):
3          temp=lst[i]
4          j=i-1
5          while j>=0 and lst[j]>temp:
6              lst[j+1]=lst[j]
7              j=j-1
8          lst[j+1]=temp
9  def main():
10     lst=eval(input('Enter a list :'))
11     print('Sorted list :')
12     insertionSort(lst)
13     print(lst)
14
15 if __name__=='__main__':
16     main()
```

Searching

- We will see how to find out whether a data value appears in a list.
- For example, given a list of names of students in a class, we wish to find whether a particular name appears in the list.
- We'll discuss two methods of searching here, **Linear search** and **Binary search**

Linear search

- In this method, we scan the list from the beginning till the required data value is found or the list is exhausted.
- This searching technique is known as **linear search** as the search process is sequential.

Linear search Code

```
1  def linearSearch(lst,searchValue):
2      for i in range(0,len(lst)):
3          if lst[i]==searchValue:
4              return i
5      return None
6  def main():
7      lst=eval(input('Enter a list :'))
8      searchVal=eval(input('Enter the value to be searched :'))
9      searchResult=linearSearch(lst,searchVal)
10     print(searchVal,'found at index',searchResult)
11 if __name__=='__main__':
12     main()
```

Binary search

- If the list to be searched is already sorted, we can use a faster method, called **binary search**.
- First, we examine the middle name of the list, if it matches the name that we are looking for, we stop.
- If we do not find the name at the middle position, we will only have to search on the left or right side of the middle name depending on whether the name at the middle position is greater or smaller than the name we are looking for.
- To see how fast binary search works, we only have to observe that every time search fails, the search interval is reduced to half. Thus, in the worst case, we would require no more than (log N) + 1 steps

Binary search Code

```
1  def binarySearch(lst,searchValue):
2      low,high=0,len(lst)-1
3      while low<=high:
4          mid=int((low+high)/2)
5          if lst[mid]==searchValue:
6              return mid
7          elif searchValue<lst[mid]:
8              high=mid-1
9          else:
10             low=mid+1
11     return None
12 def isSorted(lst):
13     for i in range(1,len(lst)):
14         if lst[i]<lst[i-1]:
15             return False
16     return True
```

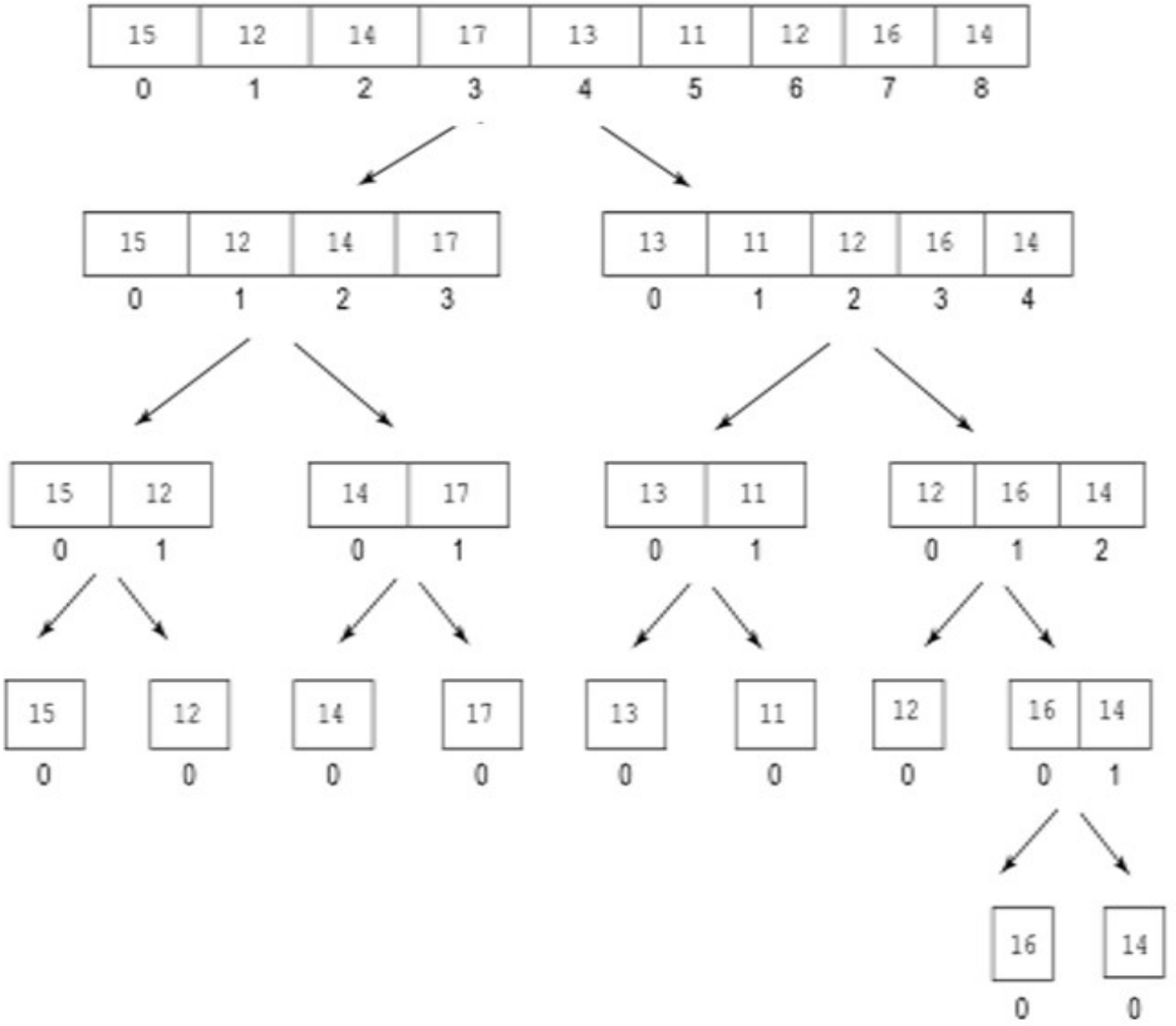
Binary search Code(Contd.)

```
1 def main():
2     lst=eval(input('Enter a sorted list (ascending order) :'))
3     if not(isSorted(lst)):
4         print('Given list is not sorted')
5     else:
6         searchVal=eval(input('Enter the value to be searched :'
      ))
7         print(searchVal,'found at index',binarySearch(lst,
      searchVal))
8 if __name__=='__main__':
9     main()
```

Merge Sort

- The merge sort algorithm is based on the divide and conquer strategy that takes a list as an input and keeps on dividing it into two lists until a list of length 1 is obtained.
- It conquers the sorting problem by merging pairs of length one lists to obtain lists of length two, merging pairs of lists of length two to obtain lists of length four, and so on until the final sorted list of the size of the original list is obtained.

Test Case


15 12 14 17 13 11 12 16 14
0 1 2 3 4 5 6 7 8
15 12 14 17
0 1 2 3
13 11 12 16 14
0 1 2 3 4
15 12
0 1
14 17
0 1
13 11
0 1
12 16 14
0 1 2
15
0
12
0
14
0
17
0
13
0
11
0
12
0
16 14
0 1
16
0
14
0

Merge Sort Code

```
1  def merge(lst1,lst2):
2      sortedList=[]
3      while len(lst1)!=0 and len(lst2)!=0:
4          if lst1[0]<lst2[0]:
5              sortedList.append(lst1[0])
6              lst1.remove(lst1[0])
7          else:
8              sortedList.append(lst2[0])
9              lst2.remove(lst2[0])
10     if len(lst1)==0:
11         sortedList+=lst2
12     else:
13         sortedList+=lst1
14     return sortedList
```

Merge Sort Code(Contd.)

```
1 def mergeSort(lst):
2     if len(lst)==0 or len(lst)==1:
3         return lst
4     else:
5         mid=len(lst)//2
6         lst1=mergeSort(lst[:mid])
7         lst2=mergeSort(lst[mid:])
8         return merge(lst1,lst2)
```

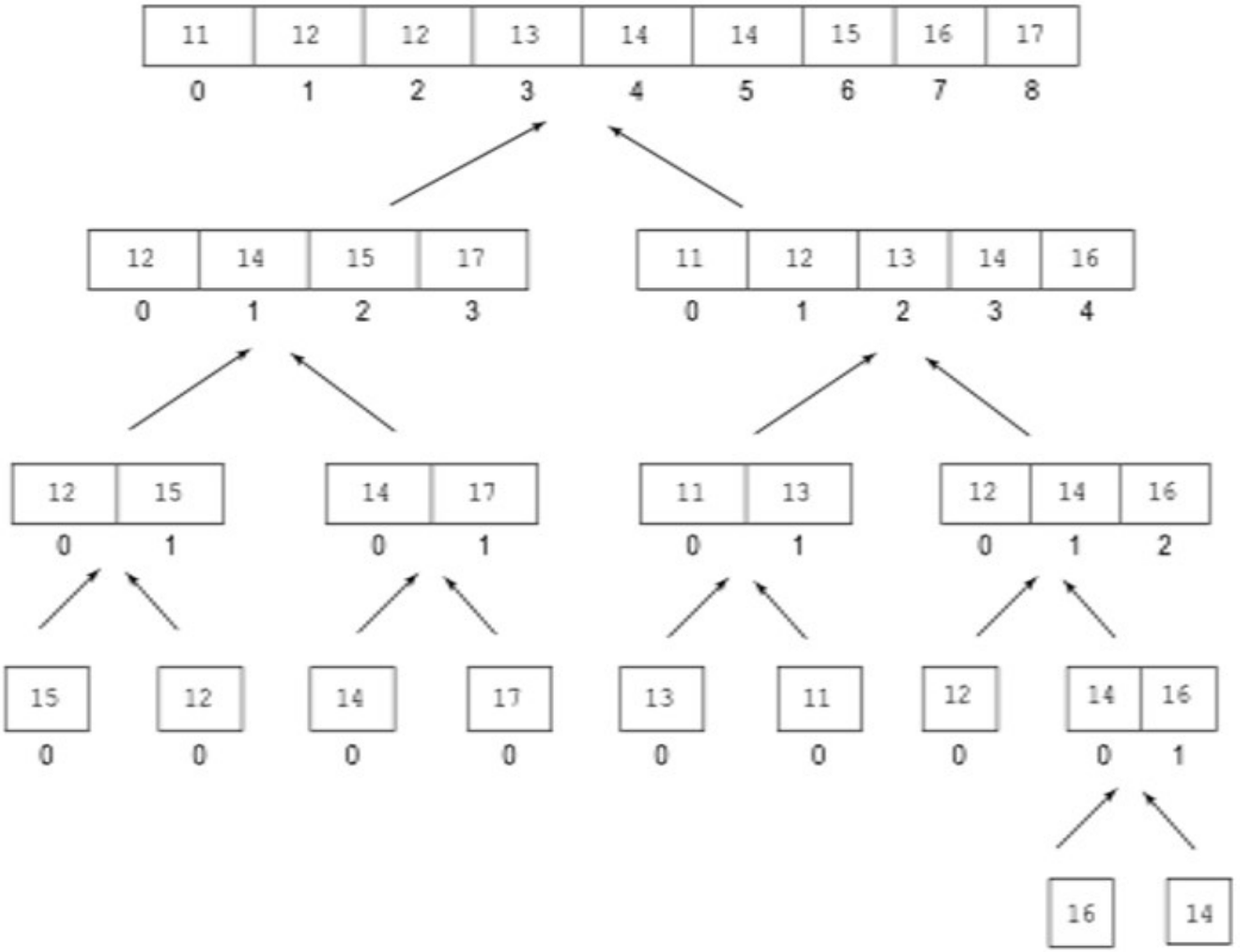
Quick Sort

- **Quick sort** is more efficient in terms of the space requirement as it does not need extra space for sorting.
- It works by choosing a value called **pivot element**, using which we split the list to be sorted in two parts:
  - The first part comprising elements smaller than or equal to the pivot element
  - The second part comprising elements greater than the pivot element
- The two parts are further sorted using quick sort in a similar manner until the partitions of length 1 or 0 are left.

Quick Sort Code

```
1  def partition(lst,start,end):
2      pivotElement=lst[end]
3      i=start-1
4      for j in range(start,end):
5          if lst[j]<=pivotElement:
6              i+=1
7              lst[i],lst[j]=lst[j],lst[i]
8      lst[i+1],lst[end]=lst[end],lst[i+1]
9      return i+1
10 def quickSort(lst,start=0,end=None):
11     if end==None:
12         end=len(lst)-1
13     if start<end:
14         splitPoint=partition(lst,start,end)
15         quickSort(lst,start,splitPoint-1)
16         quickSort(lst,splitPoint+1,end)
17     return lst
```

Test Case


11 12 12 13 14 14 15 16 17
0 1 2 3 4 5 6 7 8
12 14 15 17
0 1 2 3
11 12 13 14 16
0 1 2 3 4
12 15
0 1
14 17
0 1
11 13
0 1
12 14 16
0 1 2
15
0
12
0
14
0
17
0
13
0
11
0
12
0
14 16
0 1
16
14

References

[1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You
Any Questions?