Sorting & Searching

- Insertion Sort
- Bubble Sort
- Selection Sort
- Quick Sort
- Merge Sort
- Heap Sort
- Linear Search
- Binary Search

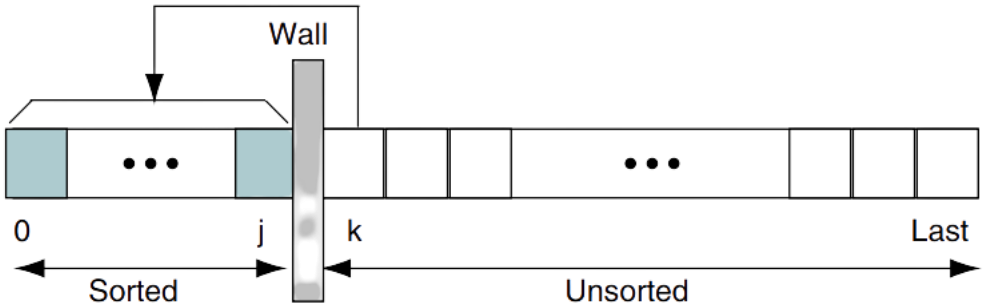
Prof. Trupesh Patel's
NOTES

Sorting is an algorithm that arranges the elements of a list in a certain order [either ascending or descending].
Searching is the process of finding an item with specified properties from a collection of items.

♣ Insertion Sort

❖ Concept:

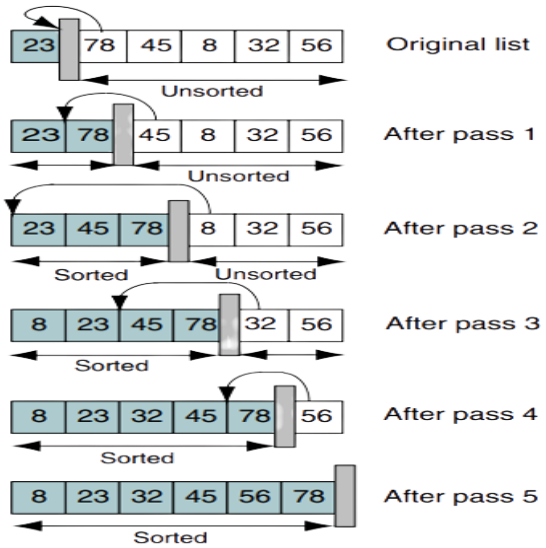
In the straight insertion sort, the list is divided into two parts: sorted and unsorted. In each pass the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place. If we have a list of n elements, it will take at most n – 1 passes to sort the data.



❖ Algorithm:

```
insertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end insertionSort
```

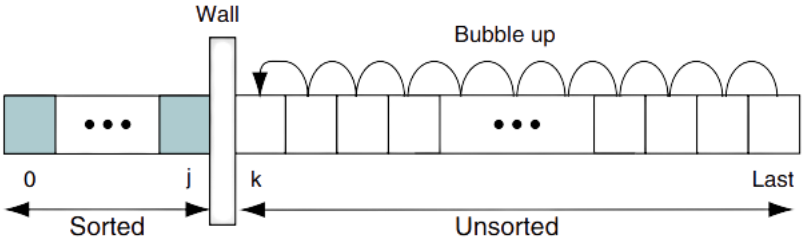
❖ Example:



♣ Bubble Sort:

❖ Concept:

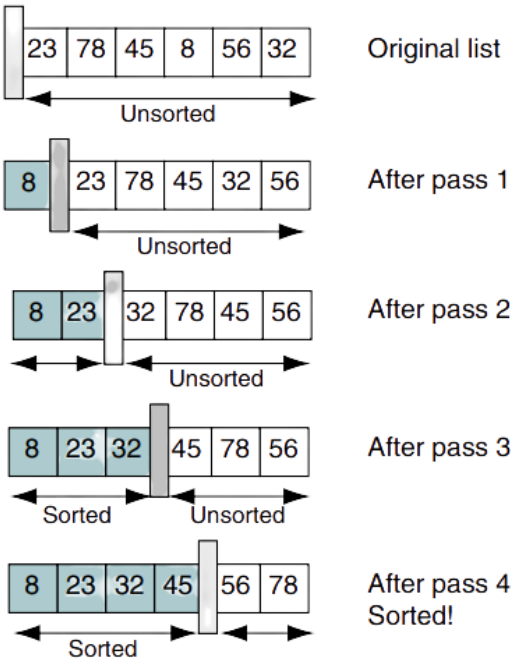
In the bubble sort, the list at any moment is divided into two sublists: sorted and unsorted. The smallest element is bubbled from the unsorted sublist and moved to the sorted sublist. After moving the smallest to the sorted list, the wall moves one element to the right, increasing the number of sorted elements and decreasing the number of unsorted ones. Each time an element moves from the unsorted sublist to the sorted sublist, one sort pass is completed. Given a list of n elements, the bubble sort requires up to n – 1 passes to sort the data.



❖ Algorithm:

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

❖ Example:



♣ Selection Sort

❖ Concept:

In the selection sort, the list at any moment is divided into two sublists, sorted and unsorted, which are divided by an imaginary wall.

We select the smallest element from the unsorted sublist and exchange it with the element at the beginning of the unsorted data.

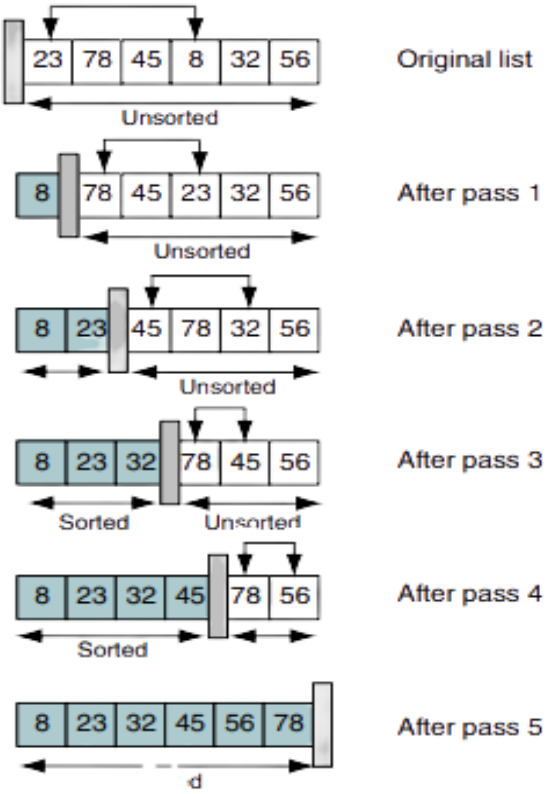
After each selection and exchange, the wall between the two sublists moves one element, increasing the number of sorted elements and decreasing the number of unsorted ones.

Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed one sort pass. If we have a list of n elements, therefore, we need n – 1 passes to completely rearrange the data.

♣ Algorithm:

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

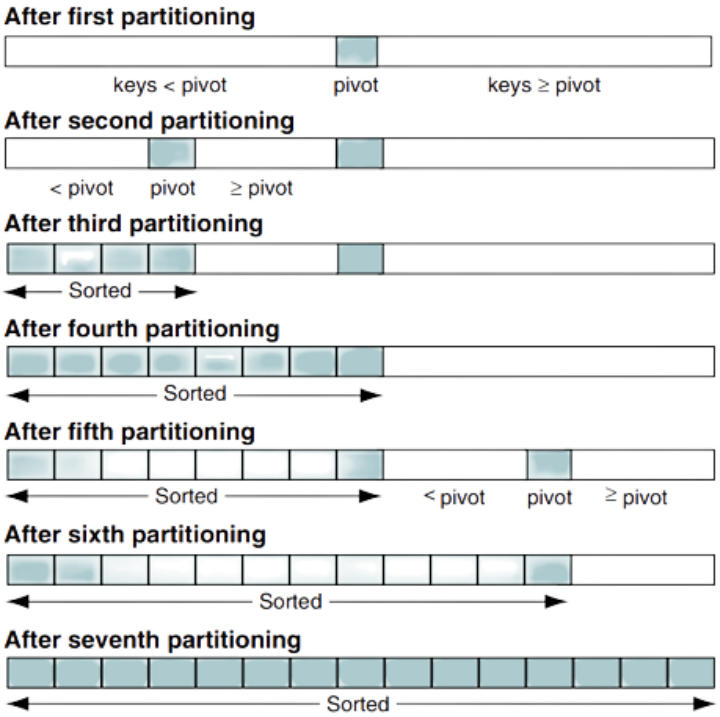
❖ Example:



♣ Quick Sort

❖ Concept:

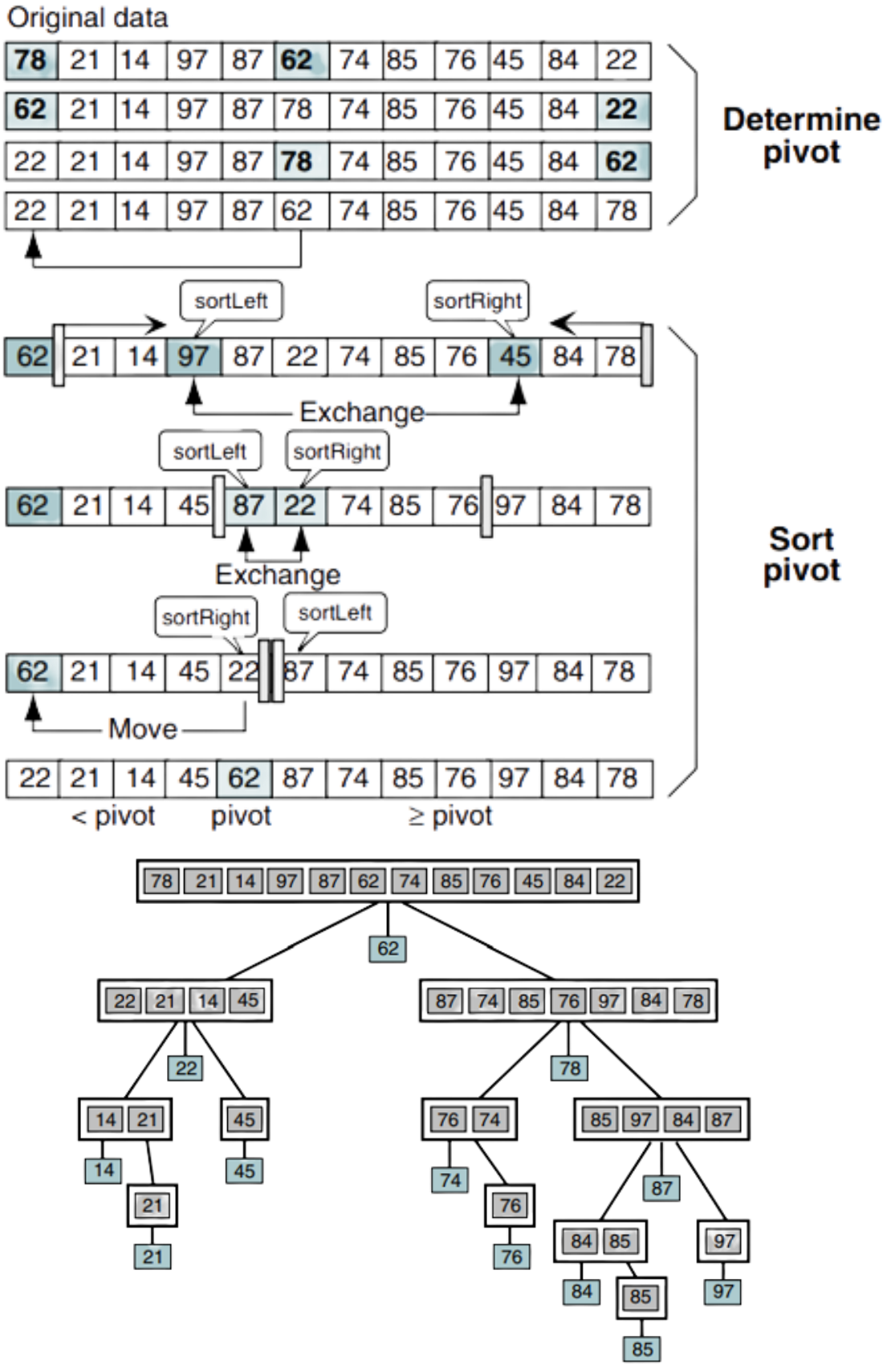
Each iteration of the quick sort selects an element, known as pivot, and divides the list into three groups: a partition of elements whose keys are less than the pivot's key, the pivot element that is placed in its ultimately correct location in the list, and a partition of elements greater than or equal to the pivot's key. The sorting then continues by quick sorting the left partition followed by quick sorting the right partition.



Algorithm:

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex - 1)
    quickSort(array, pivotIndex, rightmostIndex)
partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex - 1
  for i <- leftmostIndex + 1 to rightmostIndex
  if element[i] < pivotElement
    swap element[i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

❖ Example:


Original data
78 21 14 97 87 62 74 85 76 45 84 22
62 21 14 97 87 78 74 85 76 45 84 22
22 21 14 97 87 78 74 85 76 45 84 62
22 21 14 97 87 62 74 85 76 45 84 78
Determine pivot
sortLeft
sortRight
62 21 14 97 87 22 74 85 76 45 84 78
Exchange
sortLeft
sortRight
62 21 14 45 87 22 74 85 76 97 84 78
Exchange
sortRight
sortLeft
62 21 14 45 22 87 74 85 76 97 84 78
Move
22 21 14 45 62 87 74 85 76 97 84 78
Sort pivot
< pivot
pivot
≥ pivot
78 21 14 97 87 62 74 85 76 45 84 22
62
22 21 14 45
87 74 85 76 97 84 78
22
78
14 21
45
76 74
85 97 84 87
14
45
74
76
87
21
84 85
97
21
76
84
85
97
85

♣ Merge Sort

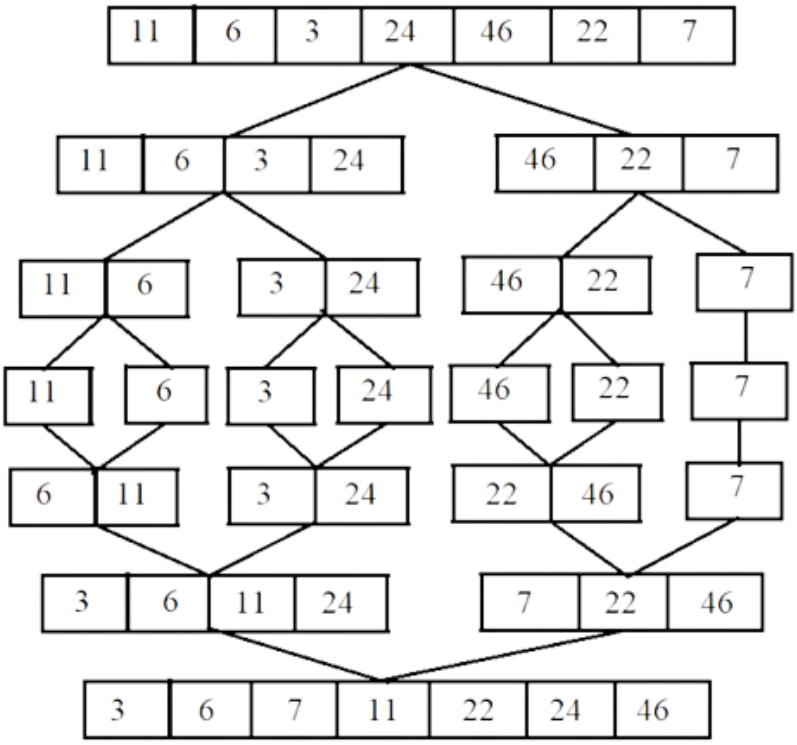
❖ Concept:

Merge sort is an example of the divide and conquer strategy. Merging is the process of combining two sorted files to make one bigger sorted file. Merge sort divides the list into two parts; then each part is conquered individually. Merge sort starts with the small subfiles and finishes with the largest one. As a result, it doesn't need stack. This algorithm is stable.

♣ Algorithm:

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```

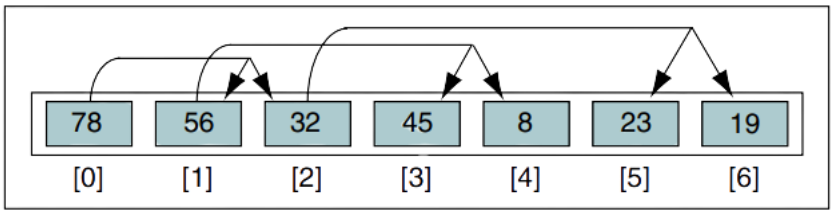
❖ Example:



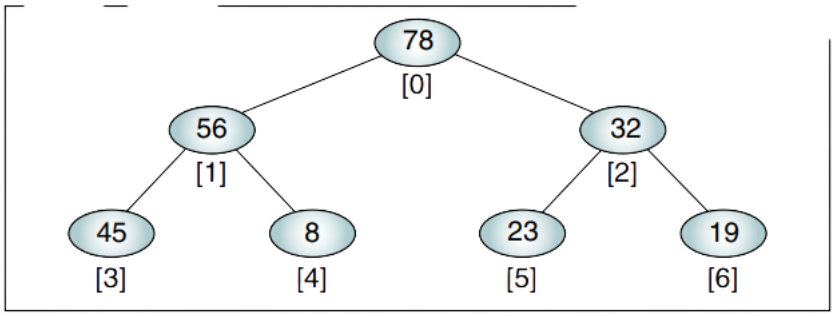
♣ Heap Sort

♣ Concept:

a heap is a tree structure in which the root contains the largest element in the tree. Heap sort begins by turning the array to be sorted into a heap. The array is turned into a heap only once for each sort. We then exchange the root, which is the largest element in the heap, with the last element in the unsorted list. This exchange results in the largest element's being added to the beginning of the sorted list. We then reheap down to reconstruct the heap and exchange again. The reheap and exchange process continues until the entire list is sorted.



Heap in Array



Heap in Tree

❖ Algorithm:

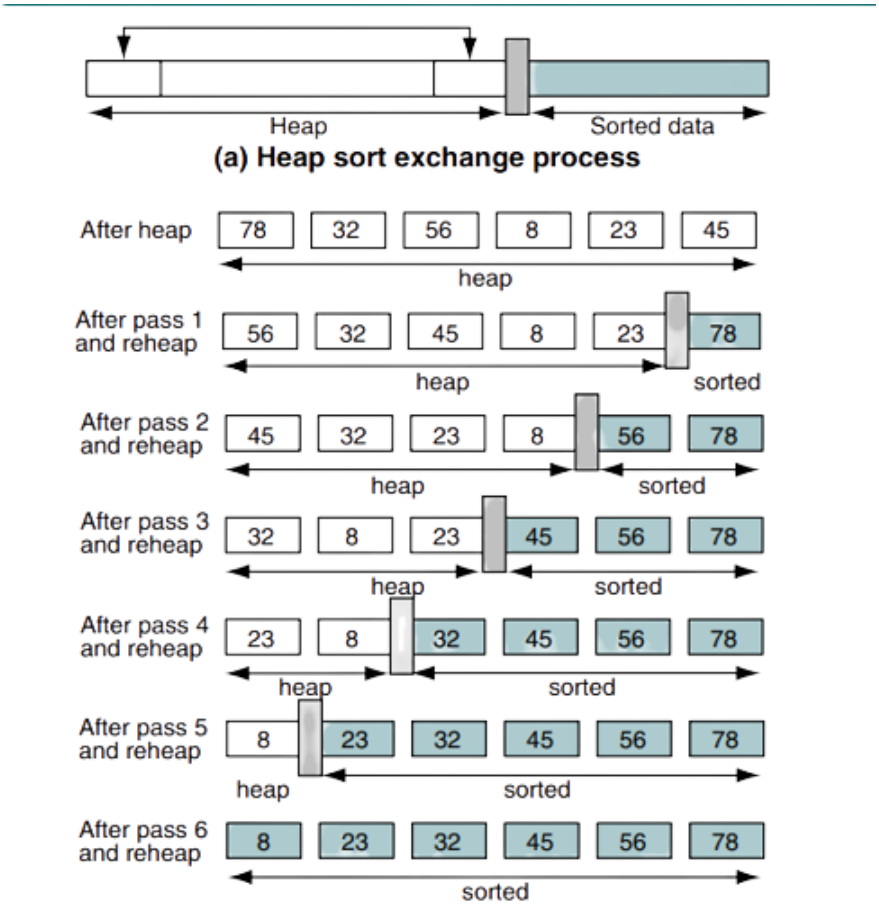
```
heapify(array)
    Root = array[0]
    Largest = largest( array[0] , array [2*0 + 1]. array[2*0+2])
    if(Root != Largest)
          Swap(Root, Largest)
```

```
void heapify(int arr[], int n, int i) {
  // Find largest among root, left child and right child
  int largest = i;
  int left = 2 * i + 1;
  int right = 2 * i + 2;

  if (left < n && arr[left] > arr[largest])
    largest = left;

  if (right < n && arr[right] > arr[largest])
```

```
    largest = right;

    // Swap and continue heapifying if root is not largest
    if (largest != i) {
      swap(&arr[i], &arr[largest]);
      heapify(arr, n, largest);
  }
}
```

```
// Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
      heapify(arr, n, i);
```

❖ Example:



♣ Linear Search

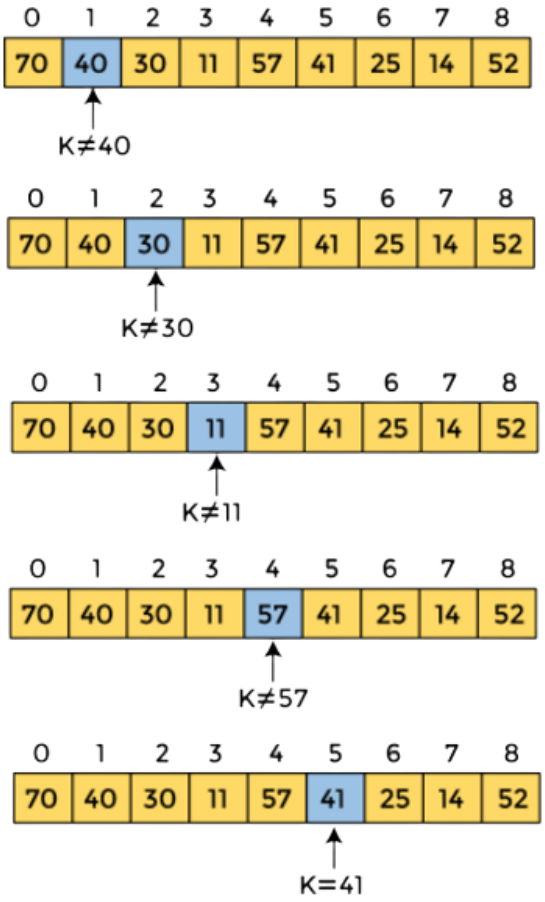
❖ Concept:

linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found. The efficiency of the sequential search is O(n).

❖ Algorithm:

```
Linear_Search
    for i=0 to last index of A:
        if A[i] equals key:
        return i
return -1
```

❖ Example:



♣ Binary Search

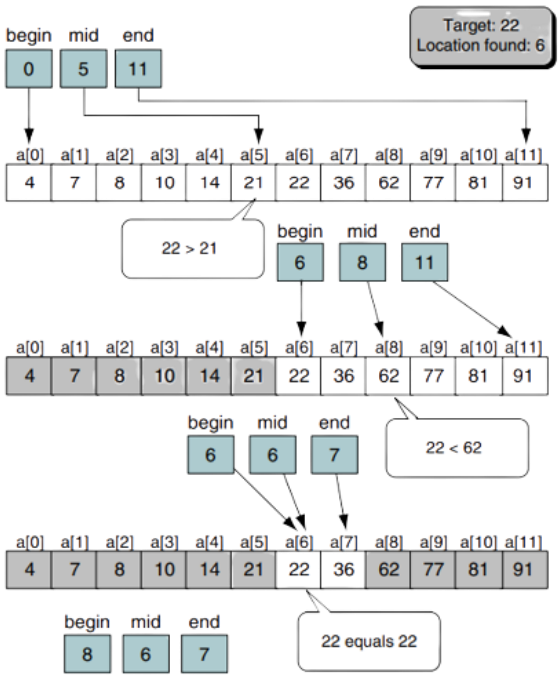
❖ Concept:

we should use a binary search whenever the list starts to become large. we should use a binary search whenever the list starts to become large. The binary search starts by testing the data in the element at the middle of the array to determine if the target is in the first or the second half of the list. If it is in the first half, we do not need to check the second half. If it is in the second half, we do not need to test the first half. In other words, we eliminate half the list from further consideration with just one comparison. We repeat this process, eliminating half of the remaining list with each test, until we find the target or determine that it is not in the list. To find the middle of the list, we need three variables: one to identify the beginning of the list, one to identify the middle of the list, and one to identify the end of the list.

❖ Algorithm:

```
do until the pointers low and high meet each other.

    mid = (low + high)/2

    if (x == arr[mid])

        return mid

    else if (x > arr[mid]) // x is on the right side

        low = mid + 1

    else                        // x is on the left side

       high = mid - 1
```

❖ Example:



We traces the binary search for a target of 22 in a sorted array. We descriptively call our three indexes begin, mid, and end. Given begin as 0 and end as 11, we can calculate mid as follows: At index location 5, we discover that the target is greater than the list value (22 > 21). We can therefore eliminate the array locations 0 through 5. (Note that mid is automatically eliminated.) To narrow our search, we set mid + 1 to begin and repeat the search. The next loop calculates mid with the new value for begin (6) and determines that the midpoint is now 8.

mid = [(6 + 11) / 2] = [17 / 2] = 8

When we test the target to the value at mid a second time, we discover that the target is less than the list value (22 < 62). This time we adjust the end of the list by setting end to mid - 1 and recalculating mid.

mid = [(6 + 7) / 2] = 13 / 2 = 6

This step effectively eliminates elements 8 through 11 from consideration. We have now arrived at index location 6, whose value matches our target.