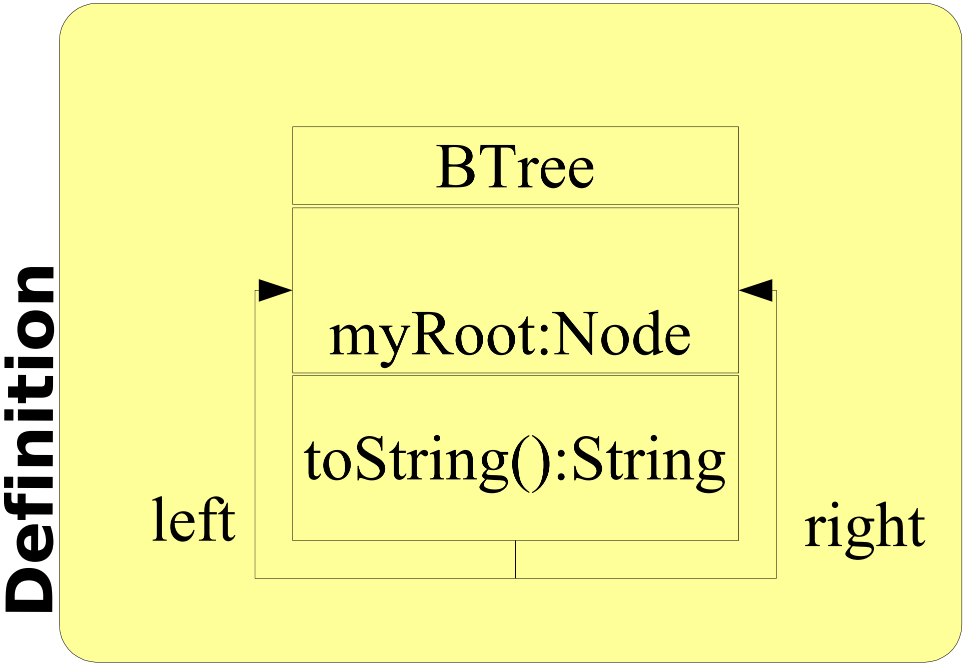
Recursion

- Recursive definition
  - A recursive definition is one that uses the concept or thing that is being defined as part of of the definition.
    - defining something at least partially in terms of itself
  - e.g.
    - a directory is a part of a drive that can hold files *and* other directories.
    - an ancestor is a parent or an ancestor of a parent

Recursion (cont.)

- Recursive definition
  - A recursive definition is one that uses the concept or thing that is being defined as part of of the definition.
    - defining something at least partially in terms of itself



Definition

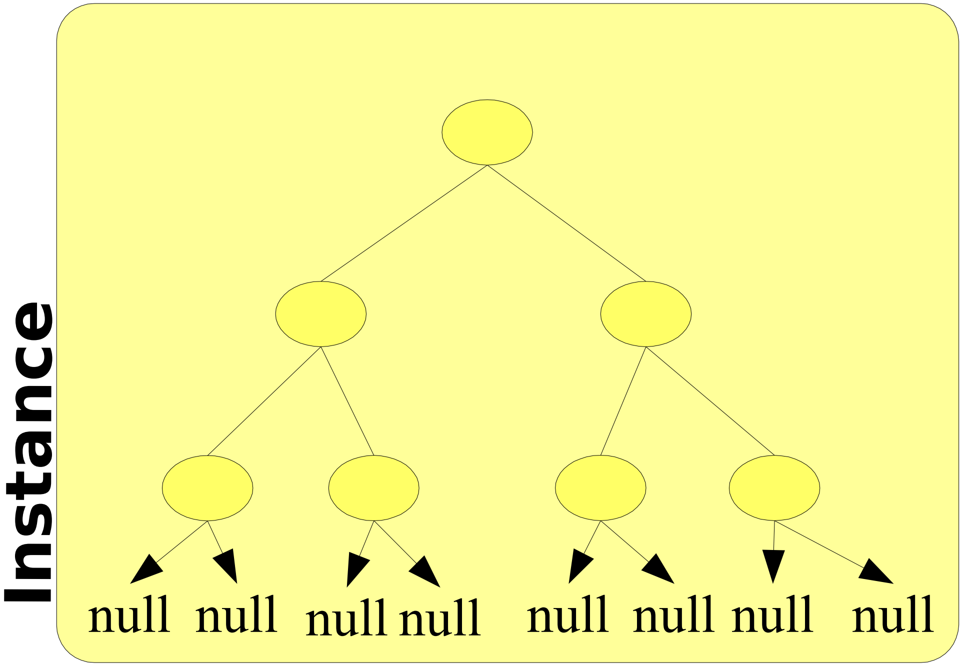
BTree

myRoot:Node

toString():String

left

right



Instance

Recursion (cont.)

- Recursion as a programming technique
  - A recursive subroutine is one that calls itself, either directly or indirectly
    - a subroutine calls itself directly means that its definition contains a subroutine call statement that calls the subroutine that is being defined.
    - a subroutine calls itself indirectly means that it calls a second subroutine which in turn calls the first subroutine

```java
public int fact(int n){
  if (n < 0) {
    System.out.println("Error:NO negatives");
    return 0;
  }else if( n == 0 || n == 1) {
    return 1;
  }else{
    return (n * fact(n-1));
  }
}
```

Recursion (cont.)

- Recursion as a programming technique
  - A recursive subroutine is one that calls itself, either directly or indirectly
    - a subroutine calls itself directly means that its definition contains a subroutine call statement that calls the

```
public String toString(){
   if (left == null && right==null) {
     return myRoot.toString();
   }
   else if (left != null && right == null){
     return new String(left.toString()+myRoot.toString());
   }else if (left == null && right != null){
     return new String(myRoot.toSTring()+right.toString());
   }else {
     return new String(left.toString()+
                       myRoot.toString()+
                       right.toString());
   }
 }
```

This is not a recursive call. It refers to *toString()* inside *Node*

Recursive methods

```
public String toString(){
   if (left == null && right==null) {
     return myRoot.toString();
   }
   else if (left != null && right == null){
     return new String(left.toString()+myRoot.toString());
   }else if (left == null && right != null){
     return new String(myRoot.toSTring()+right.toString());
   }else {
     return new String(left.toString()+
                       myRoot.toString()+
                       right.toString());
   }
 }
```

base case

- Base Case
  - a case that is handled directly instead of calling the method definition again!
    - in a binary tree, this is when a node has no children.

Recursive methods

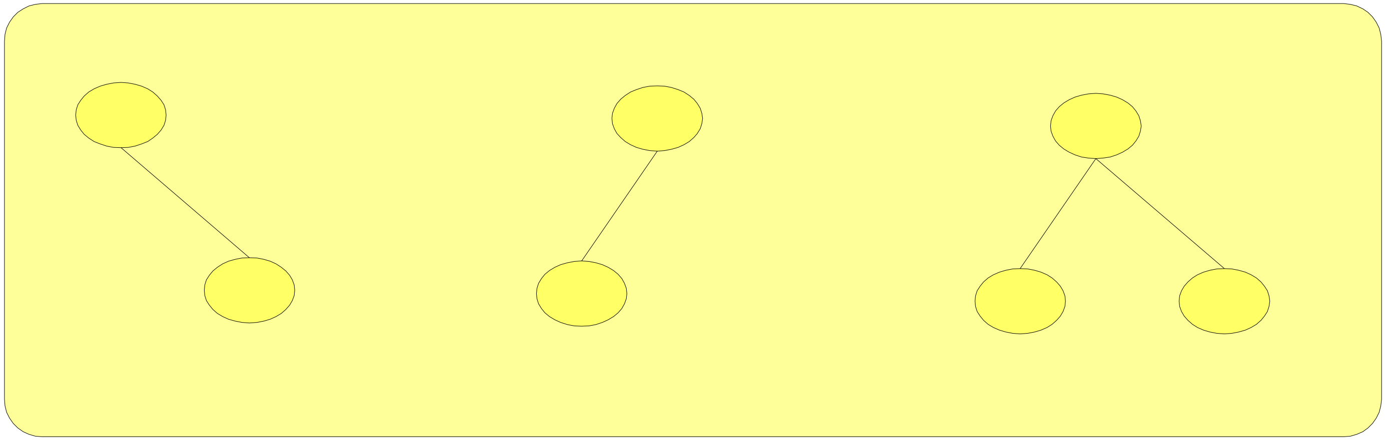
```
public String toString(){
   if (left == null && right==null) {
     return myRoot.toString();
   }
   else if (left != null && right == null){
     return new String(left.toString()+myRoot.toString());
   }else if (left == null && right != null){
     return new String(myRoot.toSTring()+right.toString());
   }else {
     return new String(left.toString()+
                       myRoot.toString()+
                       right.toString());
   }
 }
```

recursive cases

- Recursive cases
  - calls the method again but on a different instance and possibly different arguments
    - node with only one child and node with 2 children.

Counting the number of nodes

- Create a method (*NumberOfNodes()*) that counts the number of nodes in a binary tree.
- Recipe
  - what is the base case ?
    - when the whole tree is made up of one node !
  - what is the recursive case
    - any tree that has more than one node



Counting the number of nodes (cont.)

- Recipe
  - what is the base case ?
    - when the whole tree is made up of one node !
  - what is the recursive case
    - any tree that has more than one node

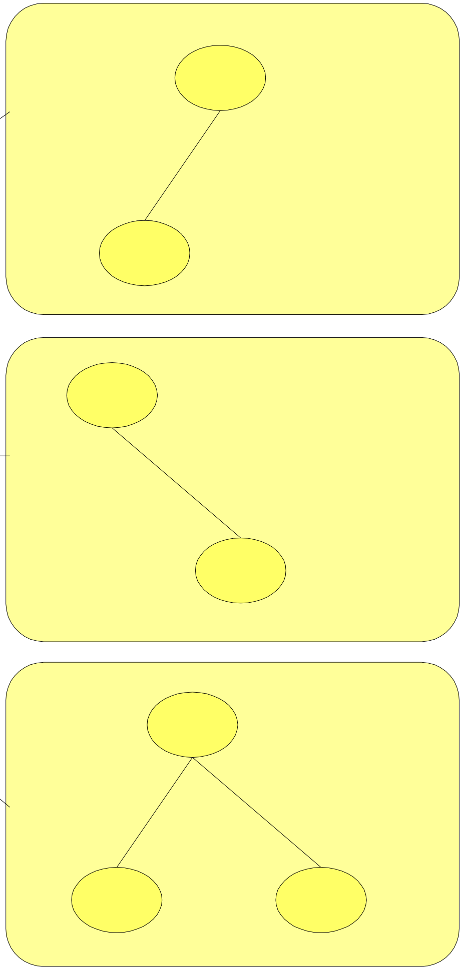
Base Case

```
public int numberOfNodes(){
  int count = 0 ;
  if (left == null && right==null) {
    count++;
    return count;
  }
}
```

Counting the number of nodes (cont.)

```
public int numberOfNodes(){
  int count = 0 ;
  if (left == null && right==null) {
    count++;
    return count;
  }else if (left != null && right == null){
    count++;
    return count += left.numberOfNodes();
  }else if (left == null && right != null){
    count++;
    return count += right.numberOfNodes();
  }else {
    count++;
    count += left.numberOfNodes();
    return  count += right.numberOfNodes();
  }

}
```



Complexity

- Evaluating execution of programs
  - time taken to complete computation
  - space required to complete computation
- Time and space depend on the programs input !
  - Worst case analysis
  - Average case analysis
  - Best case analysis
- Primitive operations do not all take the same amount of time to complete.
  - assume that all take exactly one unit of time to complete.

Searching for an element

Searching involves determining if an element is a member of the collection.

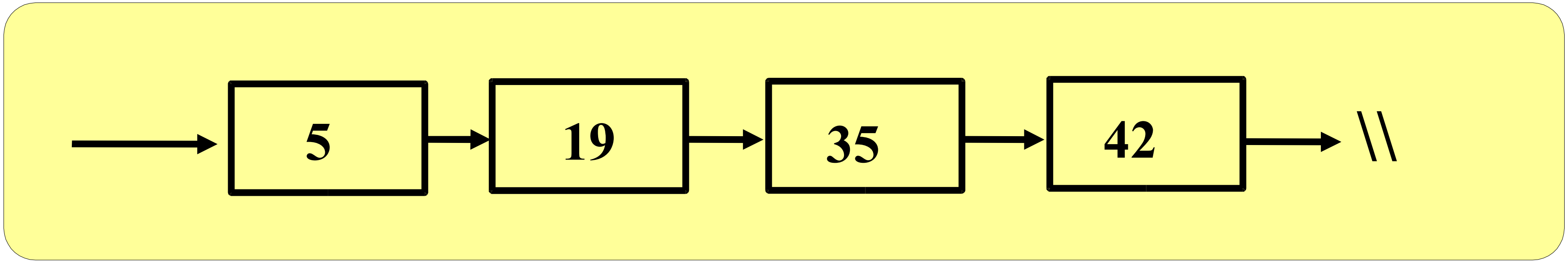
- Simple/Linear Search:
  - If there is no ordering in the data structure
  - If the ordering is not applicable
- Binary Search:
  - If the data is ordered or sorted
  - Requires non-linear access to the elements

Simple/Linear Search

- Best Case
  - The element you are looking for is the first one in the collection.
- Worst Case
  - The element you are looking for is the last one in the collection
  - The element is not in the collection.
- Average Case
  - its not the first and not the last, somewhere in the middle.

Simple/Linear Search (example)

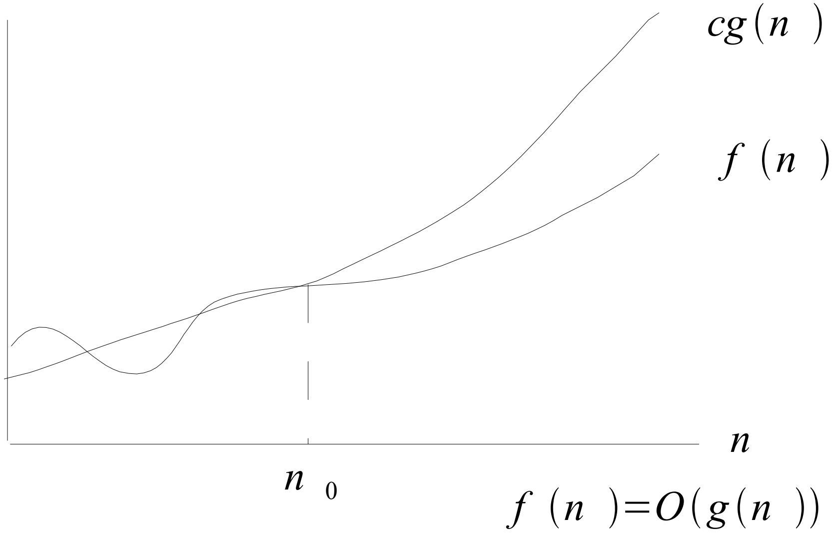
- Assume that we have a linked list that contains unordered integers. Is 10 in the list?



- It will take:
  - – 4 comparisons
  - – 4 advance operations
  - – total = 2 x 4.
- How much will it take if the list had 100 and '10' was not inclcued. 1000 elements ?

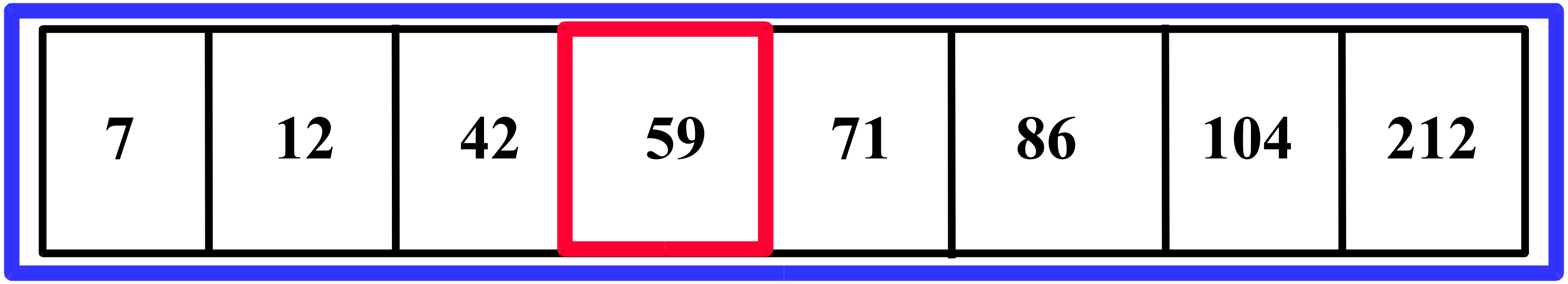
Simple/Linear Search (example)

- For any list of size n
  - total = kn for some k. Written as $O(n)$.
- The O() notation
  - Upper Bound.
  - $O(g(n)) = \{ f(n)$: there exists a positive constant c and n_0 such that $0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \}$



Binary Search

- We may perform binary search on
  - Sorted arrays
  - Full and balanced binary search trees
- Tosses out ½ the elements at each comparison.

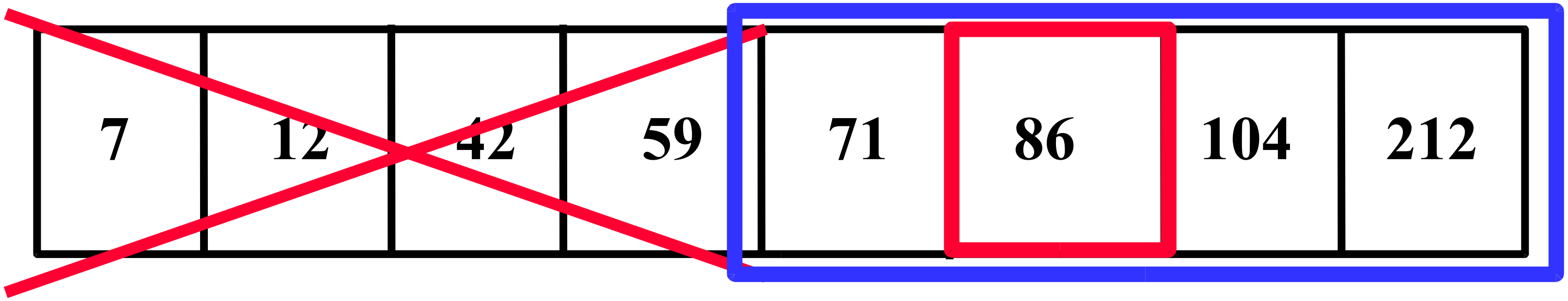


7	**12**	**42**	**59**	**71**	**86**	**104**	**212**

Looking for 89

Binary Search (cont.)

- We may perform binary search on
  - Sorted arrays
  - Full and balanced binary search trees
- Tosses out ½ the elements at each comparison.

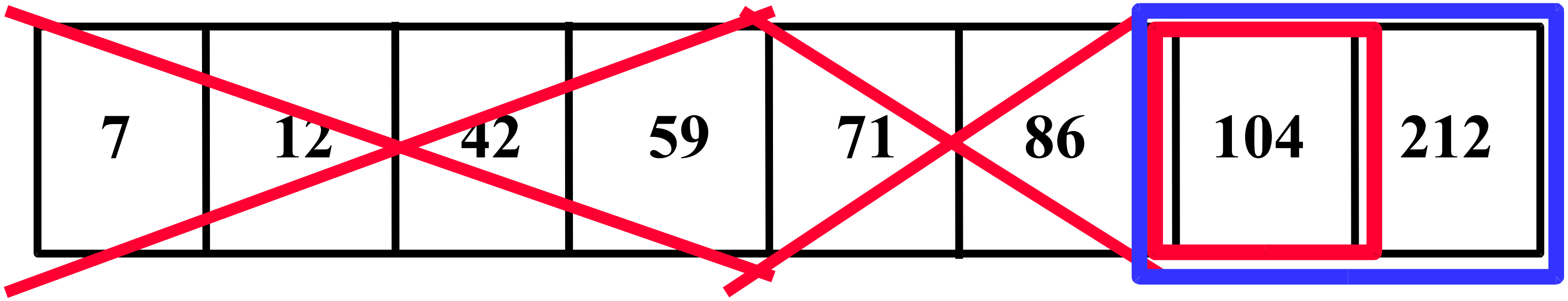


7	12	42	59	71	86	104	212

Looking for 89

Binary Search (cont.)

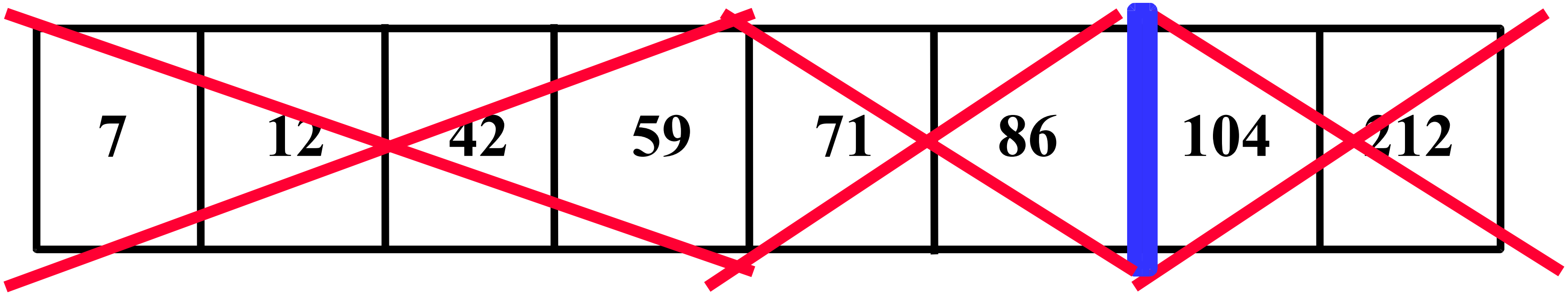
- We may perform binary search on
  - Sorted arrays
  - Full and balanced binary search trees
- Tosses out ½ the elements at each comparison.



Looking for 89

Binary Search (cont.)

- We may perform binary search on
  - Sorted arrays
  - Full and balanced binary search trees
- Tosses out ½ the elements at each comparison.



89 not found – 3 comparisons

log(8) = 3

Binary Search (cont.)

- An element can be found by comparing and cutting the work in half.
  - We cut work in ½ each time
  - How many times can we cut in half?
  - $\log_2 N$

- Thus binary search is O(log N).

Insert into unsorted collections

- Inserting an element requires two steps:
  - Find the right location
  - Perform the instructions to insert

- If the collection in question is unsorted, then O(1)
  - insert to the front
  - insert to end (in the case of an array)
  - There is no work to find the right spot and only constant work to actually insert.

Insert into sorted collections

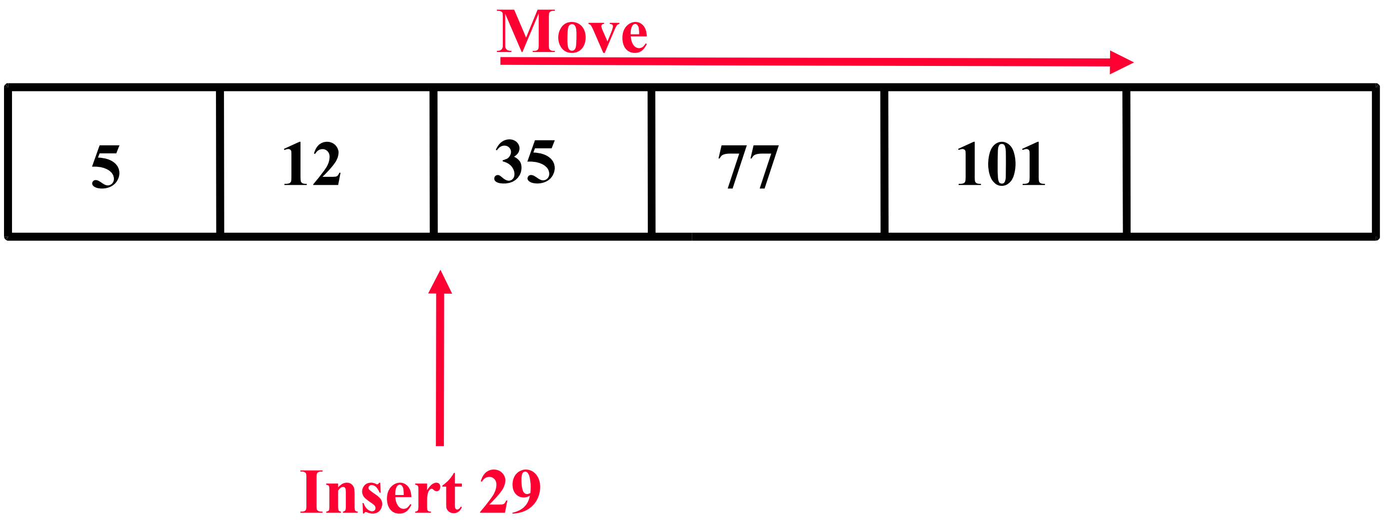
Finding the right spot is O(log N)

- Binary search on the element to insert

Performing the insertion

- Shuffle the existing elements to make room for the new item

Shuffling elements



Move

5	12	35	77	101	

Insert 29

- In the worst case, shuffle takes $O(n)$
  - – adding to the beginning of the list.

Insert into sorted collections

Finding the right spot is O(log N)

- Binary search on the element to insert

Performing the insertion O(N)

- Shuffle the existing elements to make room for the new item

These are sequential steps, add their complexities

- Total = O(log N + N) = O(N)