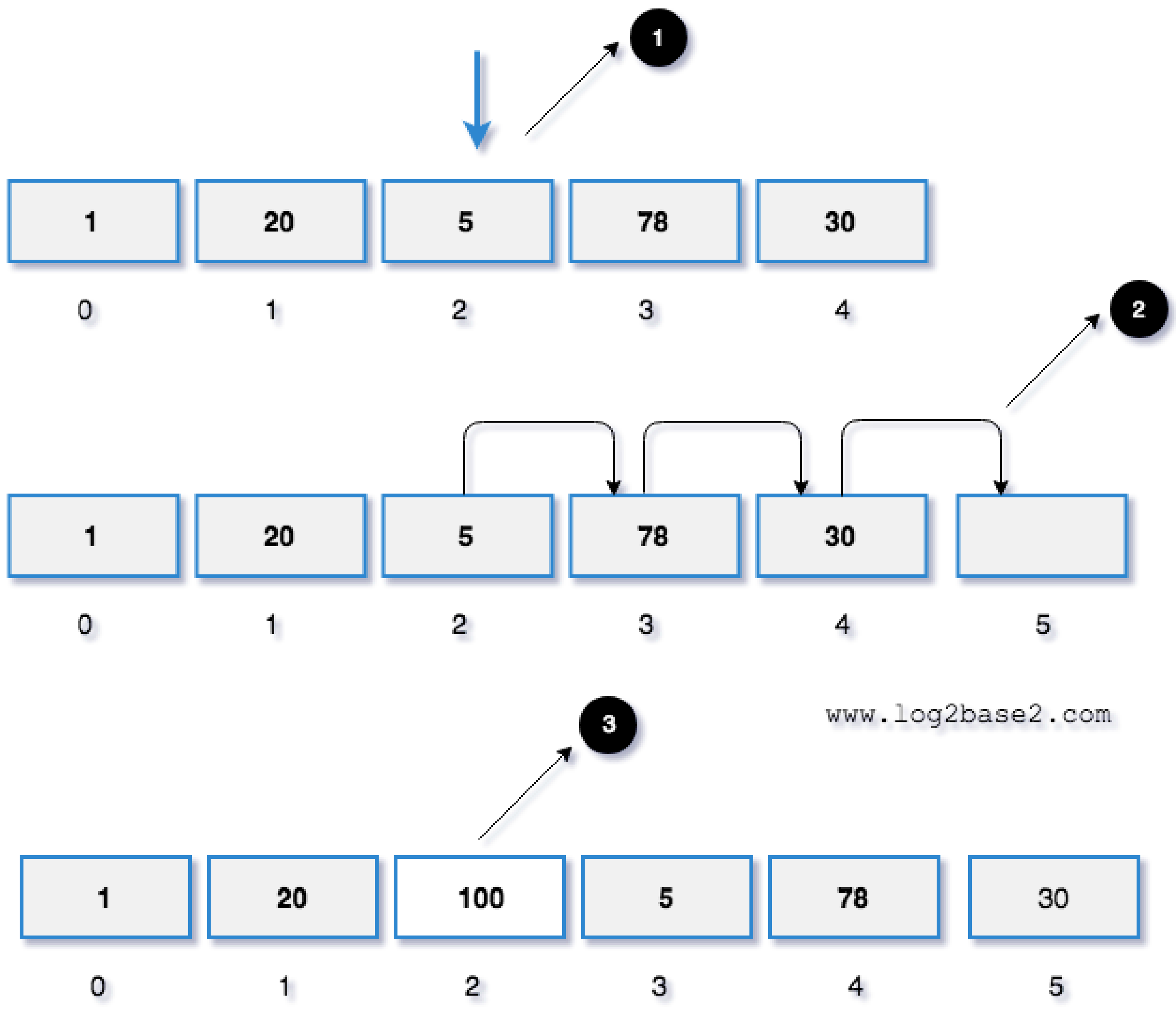
Inserting Elements in an array

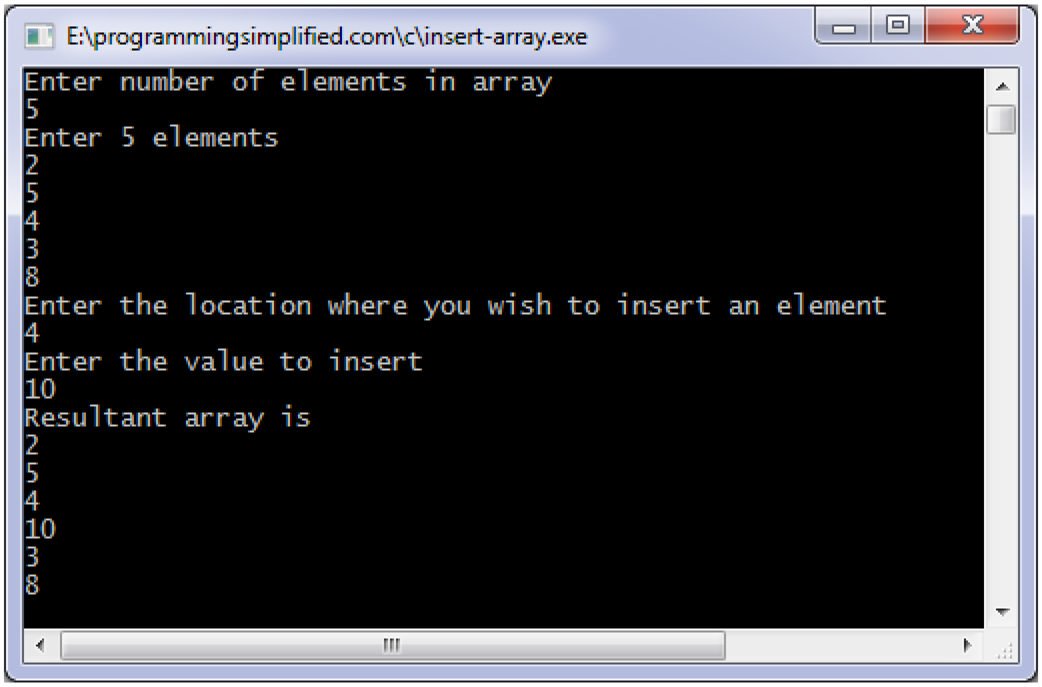
Algorithm

```
1. Get the element value which needs to be inserted.
2. Get the position value.
3. Check whether the position value is valid or not.
4. If it is valid,
     Shift all the elements from the last index to position index by 1 position to the right.
     insert the new element in arr[position]
5. Otherwise,
     Invalid Position
```



Program to insert an element

```
#include <stdio.h>

int main()
{
   int array[100], position, c, n, value;

   printf("Enter number of elements in array\n");
   scanf("%d", &n);

   printf("Enter %d elements\n", n);

   for (c = 0; c < n; c++)
      scanf("%d", &array[c]);

   printf("Enter the location where you wish to insert an 
element\n");
   scanf("%d", &position);

   printf("Enter the value to insert\n");
   scanf("%d", &value);

   for (c = n - 1; c >= position - 1; c--)
      array[c+1] = array[c];

   array[position-1] = value;

   printf("Resultant array is\n");

   for (c = 0; c <= n; c++)
      printf("%d\n", array[c]);

   return 0;
}
```



```
Enter number of elements in array
5
Enter 5 elements
2
5
4
3
8
Enter the location where you wish to insert an element
4
Enter the value to insert
10
Resultant array is
2
5
4
10
3
8
```

Deleting an element from an array

Algorithm

1. Find the given element in the given array and note the index.

2. If the element found,

    Shift all the elements from index + 1 by 1 position to the left.

    Reduce the array size by 1.

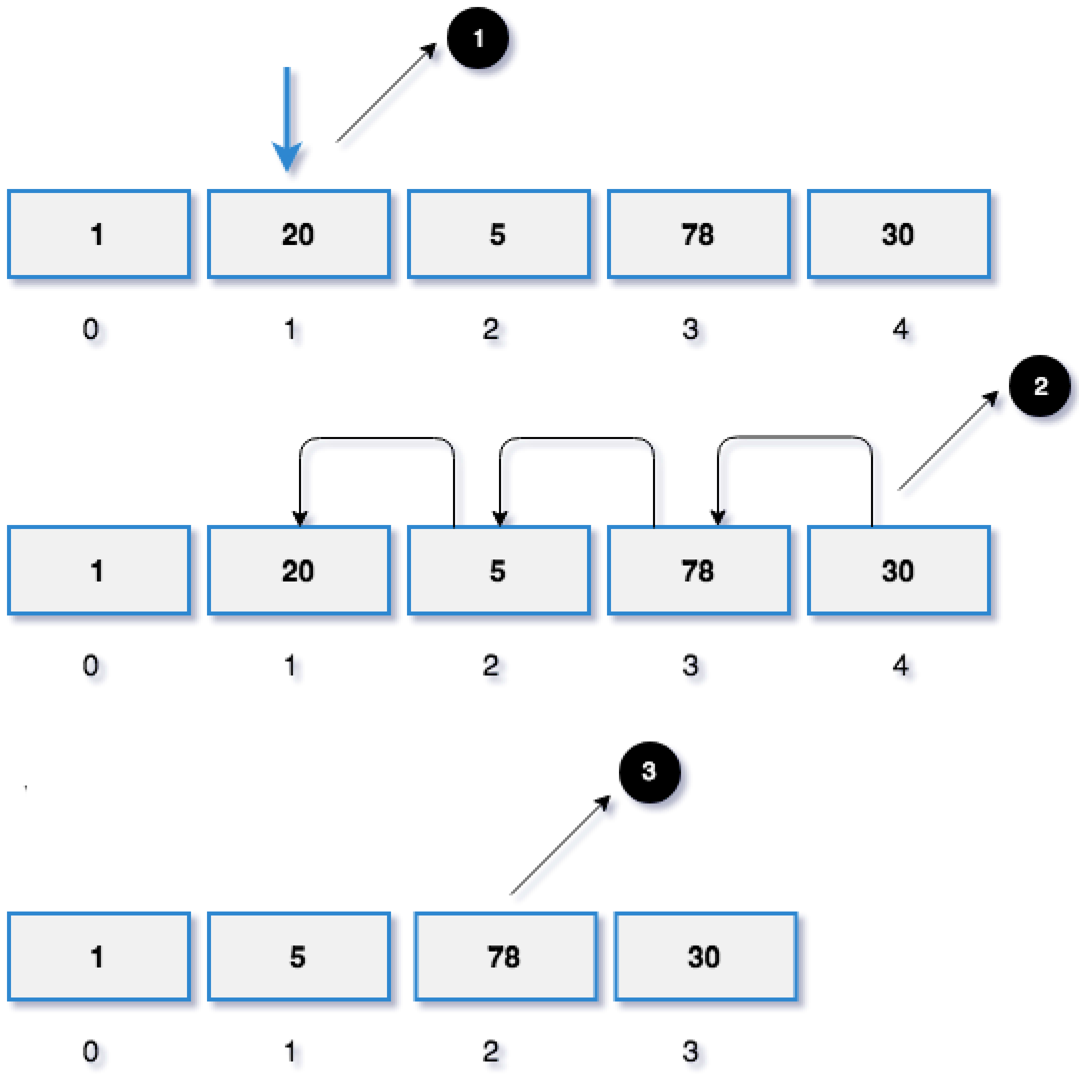
3. Otherwise, print "Element Not Found"

Visual Representation

Let's take an array of 5 elements.

1, 20, 5, 78, 30.

If we remove element 20 from the array, the execution will be,



1. We need to remove the element 20 (index 1) from the array.

2. Shift all the elements from index + 1 (index 2 to 4) by 1 position to the left.

arr[2] (value 5) will be placed in arr[1].

arr[3] (value 78) will be placed in arr[2].

arr[4] (value 30) will be placed in arr[3].

3. Finally, the new array.

Implementation Algorithm

```
1. Set index value as -1 initially. i.e. index = -1

2. Get key value from the user which needs to be deleted.

3. Search and store the index of a given key

[index will be -1, if the given key is not present in the array]

4. if index not equal to -1

    Shift all the elements from index + 1 by 1 position to the left.

5. else

    print "Element Not Found"
```

Program for deleting an element from an array

```
#include <stdio.h>

int main()
{
   int array[100], position, c, n;

   printf("Enter number of elements in array\n");
   scanf("%d", &n);

   printf("Enter %d elements\n", n);

   for (c = 0; c < n; c++)
      scanf("%d", &array[c]);

   printf("Enter the location where you wish to delete 
element\n");
   scanf("%d", &position);

   if (position >= n+1)
      printf("Deletion not possible.\n");
   else
   {
      for (c = position - 1; c < n - 1; c++)
         array[c] = array[c+1];

      printf("Resultant array:\n");

      for (c = 0; c < n - 1; c++)
         printf("%d\n", array[c]);
   }

   return 0;
}
```

Searching for an element from an array

Algorithm

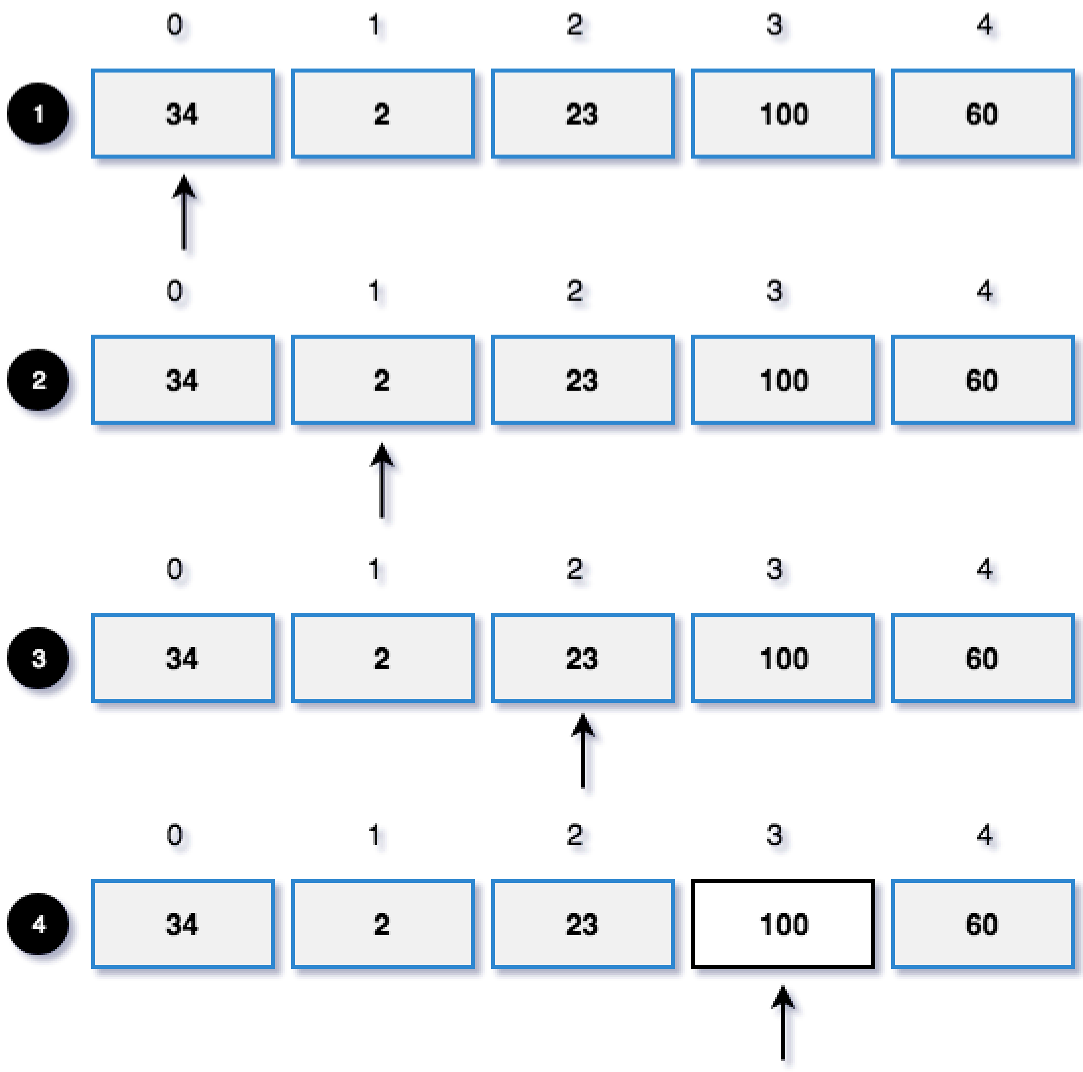
```
1. Iterate the array using the loop.
2. Check whether the given key present in the array i.e. arr[i] == key.
3. If yes,
     print "Search Found".
4. Else
     print "Search Not Found".
```

Search Found

Let's take an array of 5 elements.

34, 2, 23, 100, 60.

If we search element 100, the execution will be,



1. 34 != 100. Move to the next element.

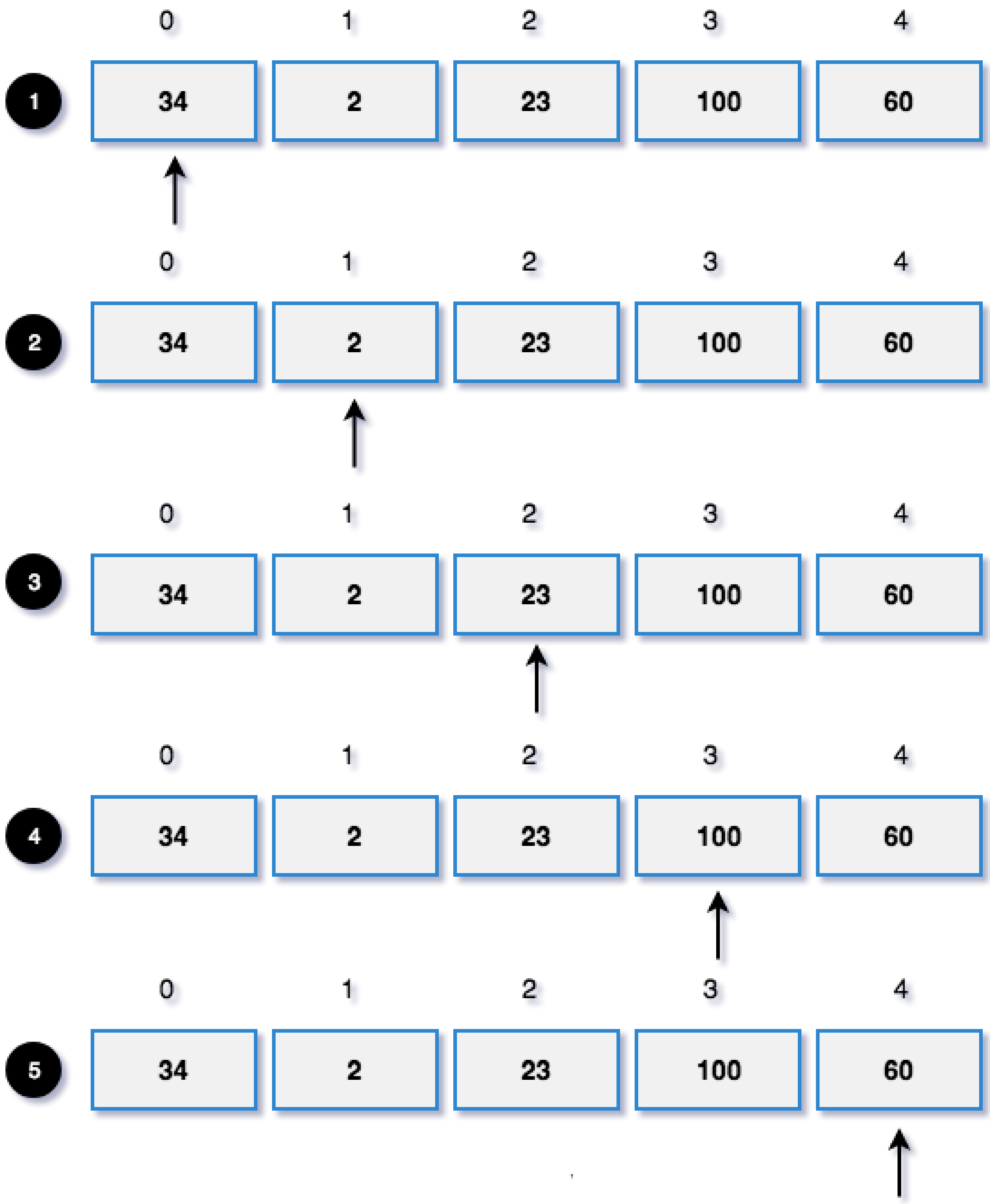
2. 2 != 100. Move to the next element.

3. 23 != 100. Move to the next element.

4. 100 == 100. Search Found.

Search Not Found

If we search element 1000, the execution will be,



1. 34   != 1000. Move to the next element.

2. 2   != 1000. Move to the next element.

3. 23   != 1000. Move to the next element.

4. 100 != 1000. Move to the next element.

5. 60   != 1000. Search Not Found.

Algorithm for searching for an element in an array

1. Input size and elements from user. Store it in some variable say `size` and `arr`.
2. Input number to search from user in some variable say `toSearch`.
3. Define a flag variable as `found = 0`. I have initialized `found` with 0, which means initially I have assumed that searched element does not exists in array.
4. Run loop from 0 to `size`. Loop structure should look like `for(i=0; i<size; i++)`.
5. Inside loop check if current array element is equal to searched number or not. Which is `if(arr[i] == toSearch)` then set `found = 1` flag and terminate from loop. Since element is found no need to continue further.
6. Outside loop `if(found == 1)` then element is found otherwise not.

```c
#include <stdio.h>

#define MAX_SIZE 100  // Maximum array size

int main()
{
    int arr[MAX_SIZE];
    int size, i, toSearch, found;

    /* Input size of array */
    printf("Enter size of array: ");
    scanf("%d", &size);

    /* Input elements of array */
    printf("Enter elements in array: ");
    for(i=0; i<size; i++)
    {
        scanf("%d", &arr[i]);
    }

    printf("\nEnter element to search: ");
    scanf("%d", &toSearch);

    /* Assume that element does not exists in array */
    found = 0;

    for(i=0; i<size; i++)
    {
        /*
         * If element is found in array then raise found flag
         * and terminate from loop.
         */
        if(arr[i] == toSearch)
        {
            found = 1;
            break;
        }
    }

    /*
     * If element is not found in array
     */
    if(found == 1)
    {
        printf("\n%d is found at position %d", toSearch, i + 1);
    }
    else
    {
        printf("\n%d is not found in the array", toSearch);
    }

    return 0;
}
```

List

A list or sequence is an abstract data type that represents a finite number of ordered values, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a tuple or finite sequence; the (potentially) infinite analog of a list is a stream. Lists are a basic example of containers, as they contain other values. If the same value occurs multiple times, each occurrence is considered a distinct item.

Operations

Implementation of the list data structure may provide some of the following operations

- A constructor for creating an empty list;
- an operation for testing whether or not a list is empty;
- an operation for prepending an entity to a list
- an operation for appending an entity to a list
- an operation for determining the first component (or the "head") of a list
- an operation for referring to the list consisting of all the components of a list except for its first (this is called the "tail" of the list.)
- an operation for accessing the element at a given index.

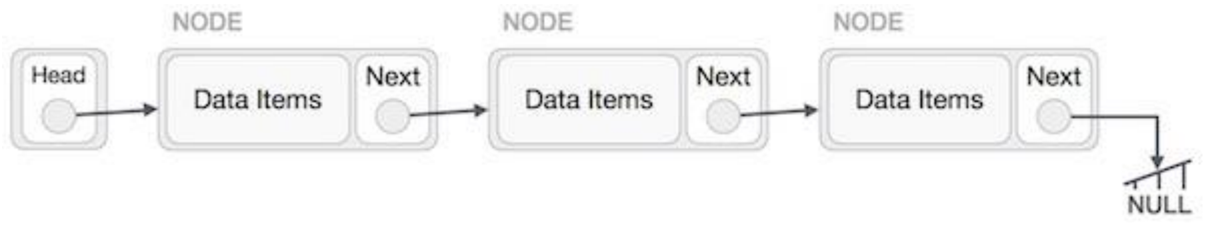
A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.

- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

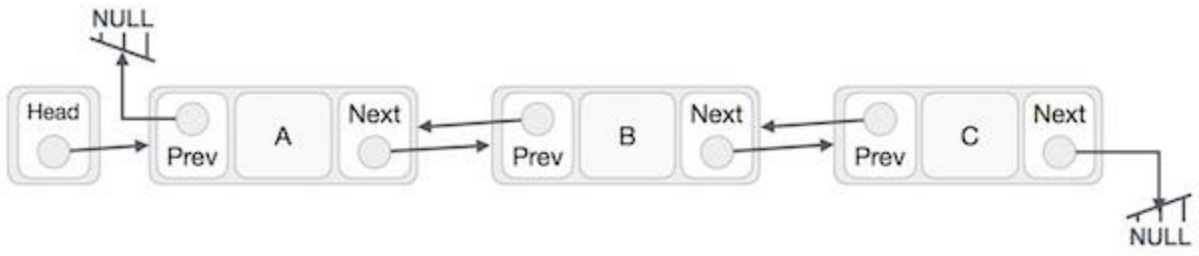
- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations

Following are the basic operations supported by a list.

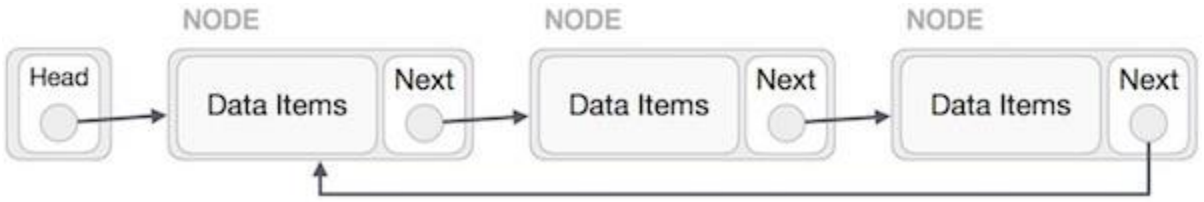
- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Doubly linked list



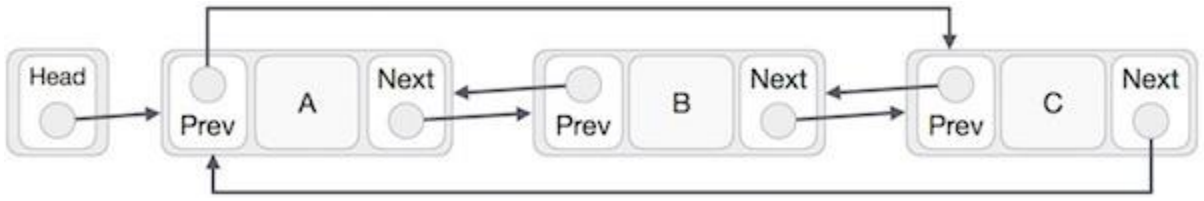
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.


Head
Prev
A
Next
Prev
B
Next
Prev
C
Next