Lab 07: Implementation of Linked List and its operations

What is a Link List?

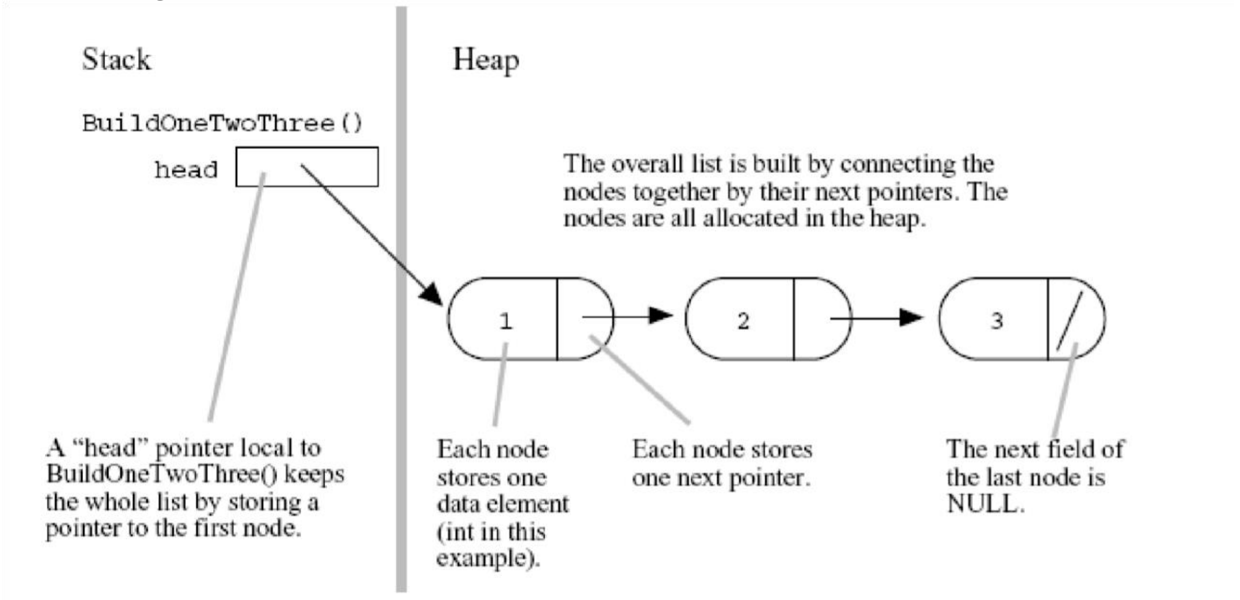
One of the most common data structures used in C is the singly linked list. The singly linked list is a convenient way to store an unbounded array, that is create an array where one doesn't know in advance how large the array will need to be. The disadvantage of the linked list is that data can only be accessed sequentially and not in random order. To read the hundredth element of a linked list, you must read the 99 elements that precede it. A linked list usually consists of a **root node**, allocated on the stack (an ordinary C variable) and one or more records, allocated on the heap (by calling malloc). The record has two parts, the **link field** to the subsequent record and the **data field[s]** containing the information you want stored in the linked list. The end of the linked list is indicated by setting the link field to NULL.

What Linked Lists Look Like

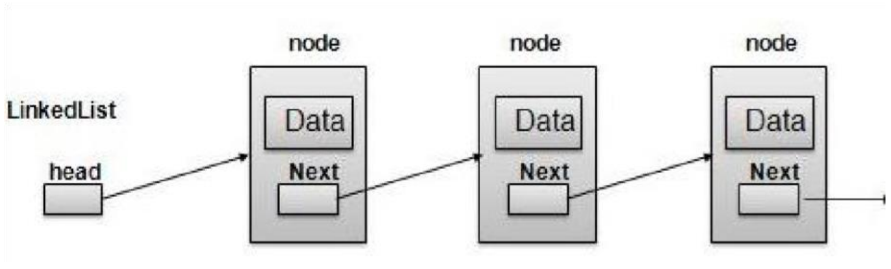
An array allocates memory for all its elements lumped together as one block of memory. In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets is overall structure by using pointers to connect all its nodes together like the links in a chain.

Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field which a pointer is used to link one node to the next node. The front of the list is a pointer to the first node. Here is what a list containing the numbers 1, 2, and 3 might look like...

The Drawing of List {1, 2, 3}


Stack
Heap
BuildOneTwoThree()
head
The overall list is built by connecting the nodes together by their next pointers. The nodes are all allocated in the heap.
1
2
3
A "head" pointer local to BuildOneTwoThree() keeps the whole list by storing a pointer to the first node.
Each node stores one data element (int in this example).
Each node stores one next pointer.
The next field of the last node is NULL.

The beginning of the linked list is stored in a "head" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the head and following the next pointers. Operations towards the front of the list are fast while operations which access node farther down the list take longer the further they are from the front.



Structure

```
struct  node  {
int          data;
node* next; };
```

Linked List Types: Node and Pointer

Node	The type for the nodes which will make up the body of the list. These are allocated in the heap. Each node contains a single client data element and a pointer to the next node in the list. Type: struct node
Node Pointer	The type for pointers to nodes. This will be the type of the head pointer and the next fields inside each node. In C and C++, no separate type declaration is required since the pointer type is just the node type followed by a '*'.

Traversing a Linked List:

```
public void Traverse(singlelist node)
{
    if (node == null)
        node = this;
    System.Console.WriteLine("Traversing :");
    while (node != null)
    {
        System.Console.WriteLine(node.data);
        node = node.next;
    }
}
```

Insertion into a Linked List:

```
public singlelist InsertNext(int value)
{

    singlelist node = new singlelist(value);
    if (this.next == null)
    {
        node.next = null;
        this.next = node;
    }
   else
    {
        singlelist temp = this.next;
        node.next = temp;
        this.next = node;
    }
    return node;
}
```

Deletion from a Linked List

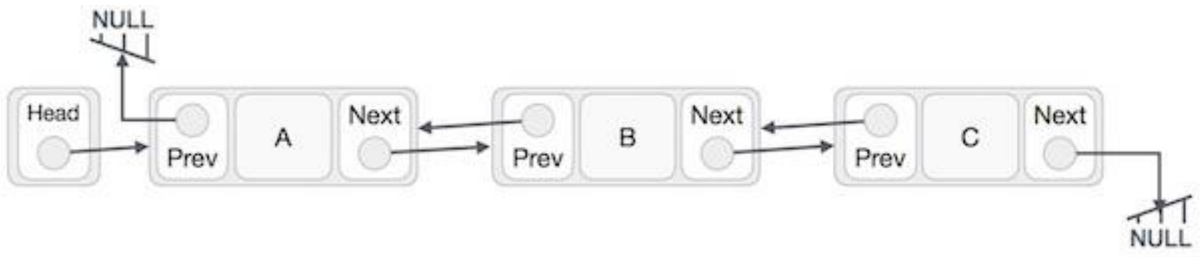
```
public int DeleteNext()
{
    if (next == null)
        return 0;
    singlelist node = this.next;
    this.next = this.next.next;
    node = null;
    return 1;
}
```

Doubly Linked List:

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

Doubly Linked List Representation

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.



Insertion into a Doubly Linked List:

```
public DLinkedList InsertNext(int value)
{
       DLinkedList node = new DLinkedList(value);
       if(this.next == null)
       {

              // Easy to handle
              node.prev = this;
              node.next = null; |
              this.next = node;
       }
       else
       {
              // Insert in the middle
              DLinkedList temp = this.next;
              node.prev = this;
              node.next = temp;
              this.next = node;
              temp.prev = node;
              // temp.next does not have to be changed
       }
       return node;
}
```

Traversing a Doubly Linked List from Front:

```
        public void TraverseFront()
        {
           TraverseFront(this);
        }

        public void TraverseFront(DLinkedList node)
        {
                if(node == null)
                        node = this;
                System.Console.WriteLine("\n\nTraversing in Forward 
  Direction\n\n");

                while(node != null)
                {
                        System.Console.WriteLine(node.data);
                        node = node.next;
                }
        }
```

Traversing a Doubly Linked List from Back:

```
        public void TraverseBack()
        {
           TraverseBack(this);
        }

        public void TraverseBack(DLinkedList node)
        {
                if(node == null)
                        node = this;
                System.Console.WriteLine("\n\nTraversing in Backward 
  Direction\n\n");
                while(node != null)
                {
                        System.Console.WriteLine(node.data);
                        node = node.prev;
                }
        }
```