# Linked List

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Data Structures (CS 223)

## 1 Intuition Behind Linked List

Let us first look at the main bottleneck of arrays: they cannot be re-sized. The best we can do is create a completely different array of the desired size and copy elements from the original array as required. Obviously, this is an expensive operation – to insert or delete a single element, we have to copy the entire array.

That brings us to the question: *what impedes us from re-sizing an array*? The reason is that an array occupies a contiguous block in memory. By the time we have created the array and carried out operations on it, the next address in memory may have already been used by the compiler/operating system for some other purpose. Therefore, we cannot get the next memory slot, messing up the contiguous property.

However, imagine that if there is an empty slot in memory, not necessarily contiguous, and somehow we are able to ascertain that this is the next place (after the end of the array) we should be looking at. In other words, if we are able to maintain a link from the last cell of the array to this empty slot in memory, then we are done. That is simply the idea behind linked list.[1]

## 2 Definition

A linked list is a collection of nodes, where each node has primarily two components:

- a *value* stored at the node
- a *next* link that points to the next node on the list – a node has only one incoming link (from the previous node) and one outgoing link (to the next node).

Immediately we see that re-sizing is not a problem any more – simply create a new node (or delete an existing node) and modify next pointers appropriately – all constant time operations, PROVIDED we know the position where we want to insert or delete.

However, since the linked list nodes are randomly scattered in the memory, we have a disadvantage – we CANNOT directly access an arbitrary node of the linked list. Because, unlike arrays, the memory is not contiguous.[2] To access any node, we have to start traversing from the start of the linked list and iterate through each node one-by-one until we reach our desired target node.

Clearly, the first thing that we need in order to access any node in the linked list is to locate where the linked list starts. To this end, we maintain a pointer (C++) or reference (JAVA) to

[1] Do not construe this as we are actually storing a link from the end of an array to a memory location. Arrays and linked lists are different data structures. This is simply to explain the concept of linkage.

[2] Remember that the name of the array is in some sense a pointer (in C++) or reference (in JAVA), which basically tells us where is the first cell of the array located in memory. Since every cell is equally sized and contiguous, we can jump to the address of an arbitrary cell in $O(1)$ time, making direct access to an array cell possible.

the first element in the linked list, appropriately called *head*. As a convention, we also store a pointer/reference to the last node on the linked list, called *tail*.

**Summarizing**, a linked list is a collection of nodes, where each node has a *value* and a *next* link pointing to the next node on the linked list. Additionally we store information about the start (resp. end) of the linked list using *head* (resp. *tail)*.

**As a side remark**, since we cannot directly jump to a particular node on the list, we CANNOT binary search a linked list (efficiently, i.e., in $O(\log n)$ time), or for that matter EFFICIENTLY simulate any algorithm that requires direct access to memory locations.

## 3 Implementation

We can visualize the linked list nodes simply as (C++/JAVA) objects of a class **ListNode**, which contains two members – an integer *value* and a pointer/reference *next* of type ListNode.

The **LinkedList** class contains two pointers/references, each of type ListNode – one for the start and one for the end of the linked list, called respectively *head* and *tail*. Additionally, we maintain an integer variable *size* for keeping track of the size of the linked list, i.e., the current number of nodes on the linked list.

We are going to study the following operations of the linked list:

- $insertAtFront(int\ val)$: adds a new node with value *val* at the front of the linked list.
- $insertAtEnd(int\ val)$: adds a new node with value *val* at the end of the linked list.
- $deleteHead()$: removes the head, i.e., the first node of the linked list.
- $getNodeAt(int\ pos)$: Returns the node at position *pos* of the linked list. Returns null if invalid *pos* is provided. We are using a base-0 indexing i.e., the first node is at $pos = 0$, the second at $pos = 1$, and so on.
- $getSize()$: returns the size, i.e., the number of nodes on the linked list.
- $insertAfter(ListNode\ argNode, int\ val)$: adds a new node with value *val* immediately after $argNode$ on the linked list. We are assuming that $argNode$ exists on the linked list.
- $deleteAfter(ListNode\ argNode)$: removes the node immediately after $argNode$ on the linked list. We are assuming that $argNode$ exists on the linked list.

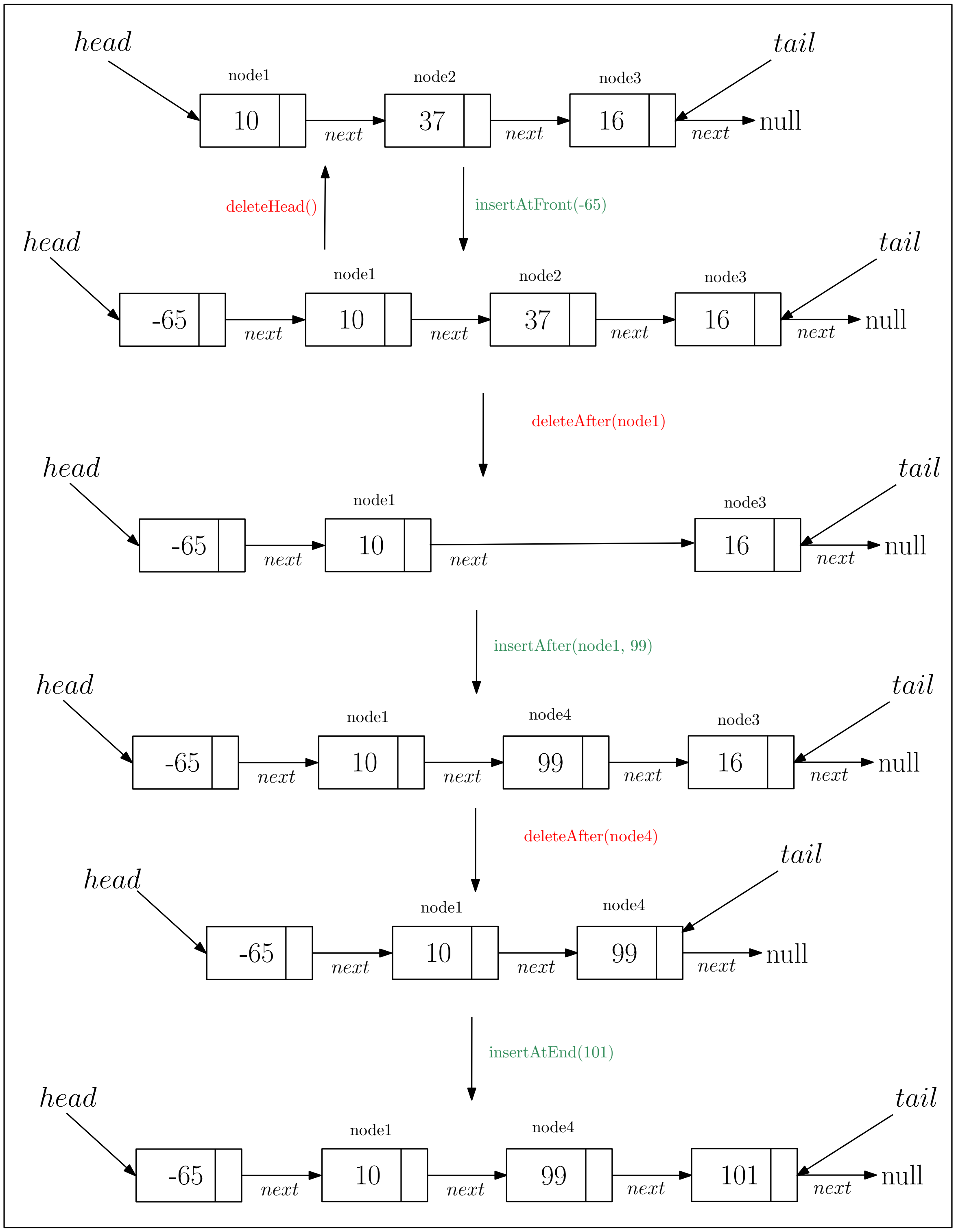
See next page for an illustration of some of the above functions.

Refer to the code at the end of the notes (both Java and C++ have been provided) on how to implement some of these functions.

There should not be any need for trying to memorize the code/algorithm. If you understand the structure of linked list, then adding/removing a node is simply modifying the next pointers/references appropriately. Also, understanding the sequence of operations is important – switching the order of operations may lead to incorrect functionality in certain cases.

## 4 Complexity

Note that $insertAtFront$, $insertAtEnd$, and $deleteHead$, all involve a constant number of operations; hence, they have a complexity of $O(1)$ in the worst case. However, $getNodeAt$ has a worst case complexity of $O(N)$, where $N$ is the size of the linked list (because we may have to traverse the entire list in the worst case). The operations $insertAfter$ and $deleteAfter$ are $O(1)$ time operations in the worst case, provided you already know the node you are going to insert or delete after (i.e., we already know $argNode$). Of course if we have to find $argNode$ (using the $getNodeAt$ function), then the worst case complexity is $O(N)$.



## 5 Java Code

```
public class LinkedList {

    class ListNode {
        protected int value;
        protected ListNode next;

        public ListNode(int val) {
            value = val;
            next = null;
        }
    }

    protected ListNode head, tail;
    protected int size;

    public LinkedList() {
        head = tail = null;
        size = 0;
    }

    ListNode insertAtFront(int value) {
        ListNode newNode = new ListNode(value);
        if (size == 0) {
            head = tail = newNode;
        } else {
            newNode.next = head;
            head = newNode;
        }
        size++;
        return newNode;
    }

    ListNode insertAtEnd(int value) {
        ListNode newNode = new ListNode(value);
        if (size == 0) {
            head = tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
        size++;
        return newNode;
    }

    void deleteHead() {
        if (0 == size) {
            System.out.println("Cannot delete from an empty list.");
        } else if (1 == size) {
            head = tail = null;
            size--;
        } else {
            size--;
            ListNode tmp = head;
            head = head.next;
            tmp.next = null;
            tmp = null;
        }
    }

    public ListNode getNodeAt(int pos) {
        if (pos < 0 || pos >= size || 0 == size) {
            System.out.println("No such position exists.");
            return null;
        } else {
            ListNode tmp = head;
            for (int i = 0; i < pos; i++)
                tmp = tmp.next;
            return tmp;
        }
    }

    public int getSize() {
        return size;
    }
}
```

## 6 C++ Code

```
#include <iostream>
using namespace std;


class ListNode {

public:
    int value;
    ListNode *next;

    ListNode(int val) {
        value = val;
        next = NULL;
    }
};

class LinkedList {

protected:
    ListNode *head, *tail;
    int size;

public:
    LinkedList() {
        head = tail = NULL;
        size = 0;
    }

    ListNode *insertAtFront(int value) {
        ListNode *newNode = new ListNode(value);
        if (size == 0) {
            head = tail = newNode;
        } else {
            newNode->next = head;
            head = newNode;
        }
        size++;
        return newNode;
    }

    ListNode *insertAtEnd(int value) {
        ListNode *newNode = new ListNode(value);
        if (size == 0) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
        size++;
        return newNode;
    }

    void deleteHead() {
        if (0 == size) {
            cout << "Cannot delete from an empty list." << endl;
        } else if (1 == size) {
            size--;
            delete head;
        } else {
            size--;
            ListNode *tmp = head;
            head = head->next;
            delete tmp;
        }
    }

    ListNode *getNodeAt(int pos) {
        if (pos < 0 || pos >= size) {
            cout << "No such position exists." << endl;
            return NULL;
        } else {
            ListNode *tmp = head;
            for (int i = 0; i < pos; i++)
                tmp = tmp->next;
            return tmp;
        }
    }

    int getSize() {
        return size;
    }
};
```