Data Structures and Algorithms

(**CS210A**)

Semester I – **2014-15**

Lecture 7:

Data structures:

- **Modeling** versus **Implementation**
- Abstract data type "**List**" and its implementation

Proof of correctness of algorithm: **Examples**

Data Structure

Definition: A collection of data elements *arranged* and *connected* in a way which can facilitate efficient executions of a (possibly long) sequence of operations.

Two steps process for designing a Data Structure

Step 1: Mathematical Modeling

A Formal description of the possible operations of a data structure. Operations can be classified into two categories:

Query Operations: Retrieving some information from the data structure

Update operations: Making a change in the data structure

Outcome of Mathematical Modeling: an **Abstract Data Type**

Step 2: Implementation

Explore the ways of organizing the data that facilitates performing each operation efficiently using the existing tools available.

Since we don't specify here the way how each operation of the data structure will be implemented

MODELING OF LIST

OUTCOME WILL BE:

ABSTRACT DATA TYPE “LIST”

Mathematical Modeling of a List

What is common in the following examples ?

- List of Roll numbers passing a course.
- List of Criminal cases pending in High Court.
- List of Rooms reserved in a hotel.
- List of Students getting award in IITK convocation 2015.

Inference: List is a **sequence** of elements.

L: $a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$

$\Uparrow$ ith element of list **L**

Query Operations on a List

- **IsEmpty(L)**: determine if **L** is an empty list.
- **Search(x,L)**: determine if **x** appears in list **L**.
- **Successor(p,L)**:

  return the element of list **L** which succeeds/follows the element at location **p**.

  The type of this parameter will depend on the implementation

  Example:

  If **L** is a_1, a_2, …, a_{i-1}, a_i, a_{i+1}, …, a_n and **p** is location of element a_i,

  then **Successor(p,L)** returns a_{i+1}.
- **Predecessor(p,L)**:

  Return the element of list **L** which precedes (appears before) the element at location **p**.

Other possible operations:

- **First(L)**: return the first element of list **L**.
- **Enumerate(L)**: Enumerate/print all elements of list **L** in the order they appear.

Update Operations on a List

- **CreateEmptyList**(**L**): Create an empty list.
- **Insert**(**x,p,L**): Insert **x** at a given location **p** in list **L**.

  Example: If **L** is $a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$ and **p** is location of element a_i, then after **Insert**(**x,p,L**), **L** becomes

  $$a_1, \ldots, a_{i-1}, \mathbf{x}, a_i, a_{i+1}, \ldots, a_n$$

- **Delete**(**p,L**): Delete element at location **p** in **L**

  Example: If **L** is $a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$ and **p** is location of element a_i, then after **Delete**(**p,L**), **L** becomes

  $$a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$$

- **MakeListEmpty(L):** Make the List **L** empty.

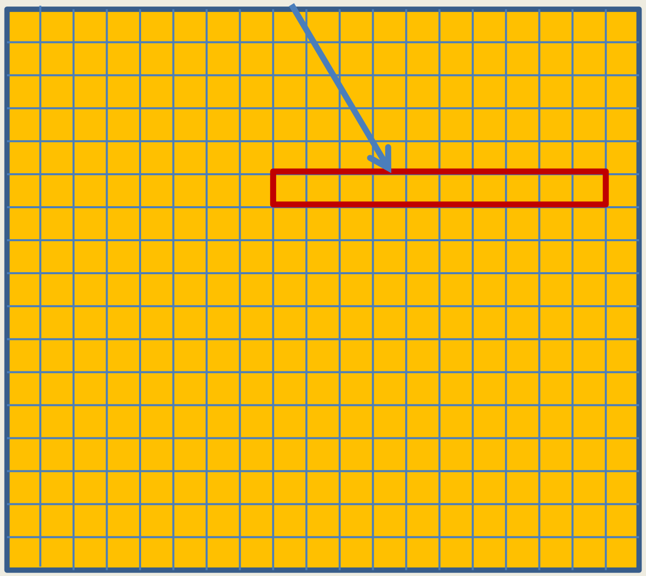
IMPLEMENTATION OF ABSTRACT DATA TYPE “LIST”

Array based Implementation

- RAM allows **O**(1) time to access any memory location.
- Array is a <u>contiguous</u> chunk of memory kept in RAM.
- For an array **A**[] storing n words,

  the address of element **A**[i] =

  "start address of array **A**" + i



This feature supports **O**(1) time to access **A**[i] for any i

Array based Implementation

- Store the elements of List in array **A** such that **A**[i] denotes (i+1)th element of the list at each stage (since index starts from 0).

  (Assumption: The maximum size of list is known in advance.)
- Keep a integer variable **Length** to denote the number of elements in the list at each stage.

Example: If at any moment of time List is 3,5,1,8,0,2,40,27,44,67, then the array **A** looks like:

A

3	5	1	8	0	2	40	27	44	67						

Length = 10

Question: How to describe location of an element of the list ?

Answer: by the corresponding array index. Location of 5th element of List is 4.

Time Complexity of each List operation using Array based implementation

Operation	Time Complexity per operation
IsEmpty(L)	O(1)
Search(x,L)	O(n)
Successor(i,L)	O(1)
Predecessor(i,L)	O(1)
CreateEmptyList(L)	O(1)
Insert(x,i,L)	O(n)
Delete(i,L)	O(n)
MakeListEmpty(L)	O(1)

n: number of elements in list at present

Arrays are very **rigid**

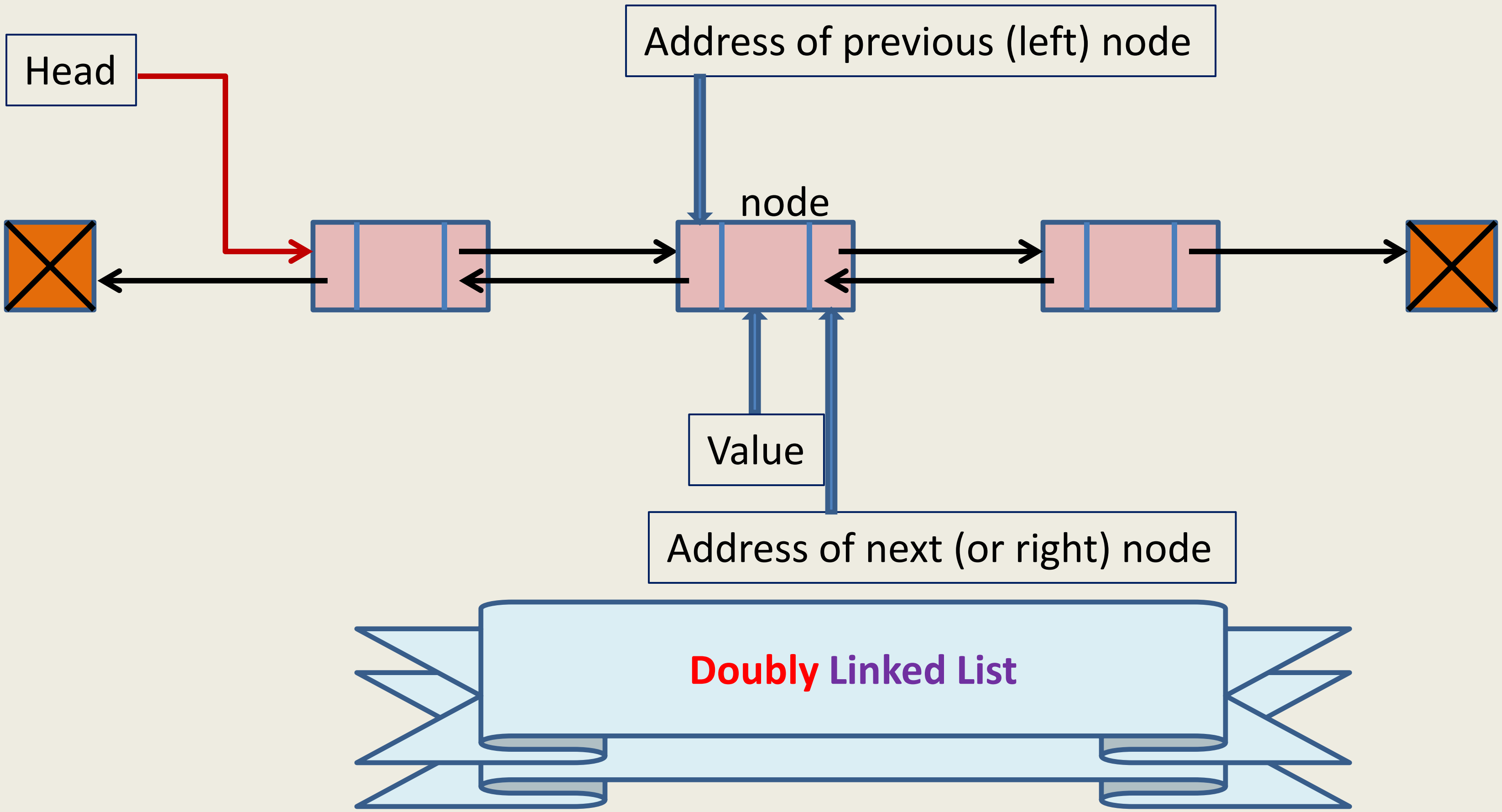
All elements from A[i] to A[n-1] have to be shifted to the **right** by one place.

All elements from A[i+1] to A[n-1] have to be shifted to the **left** by one place.

Ho

operation with matching complexity.

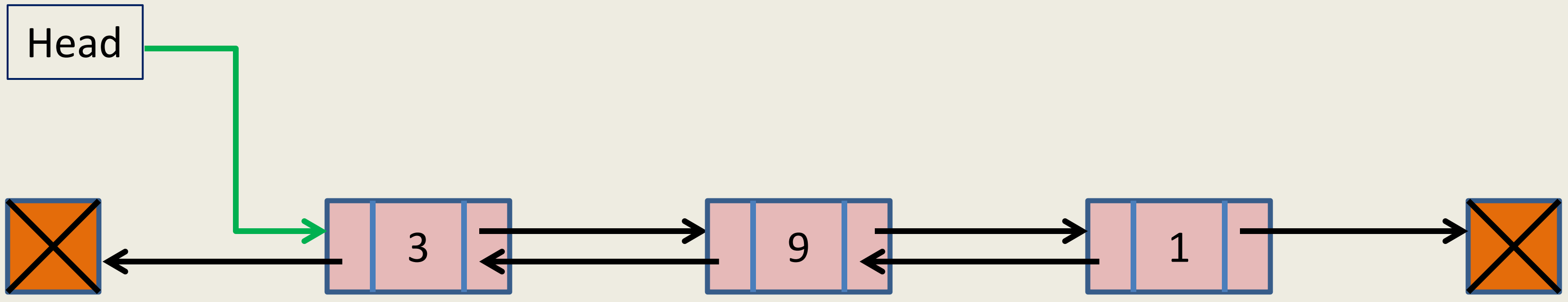
Link based Implementation:



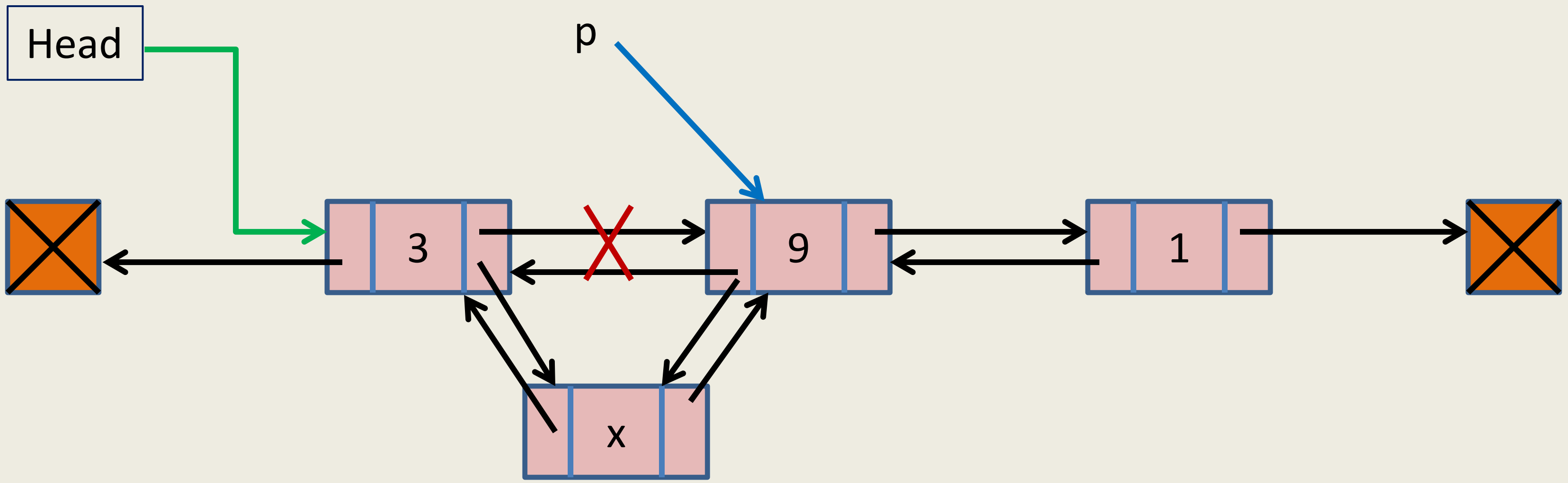
Doubly Linked List based Implementation

- Keep a doubly linked list where elements appear in the order we follow while traversing the list.
- The location of an element is the address of the node containing it.

Example: List 3,9,1 appears as

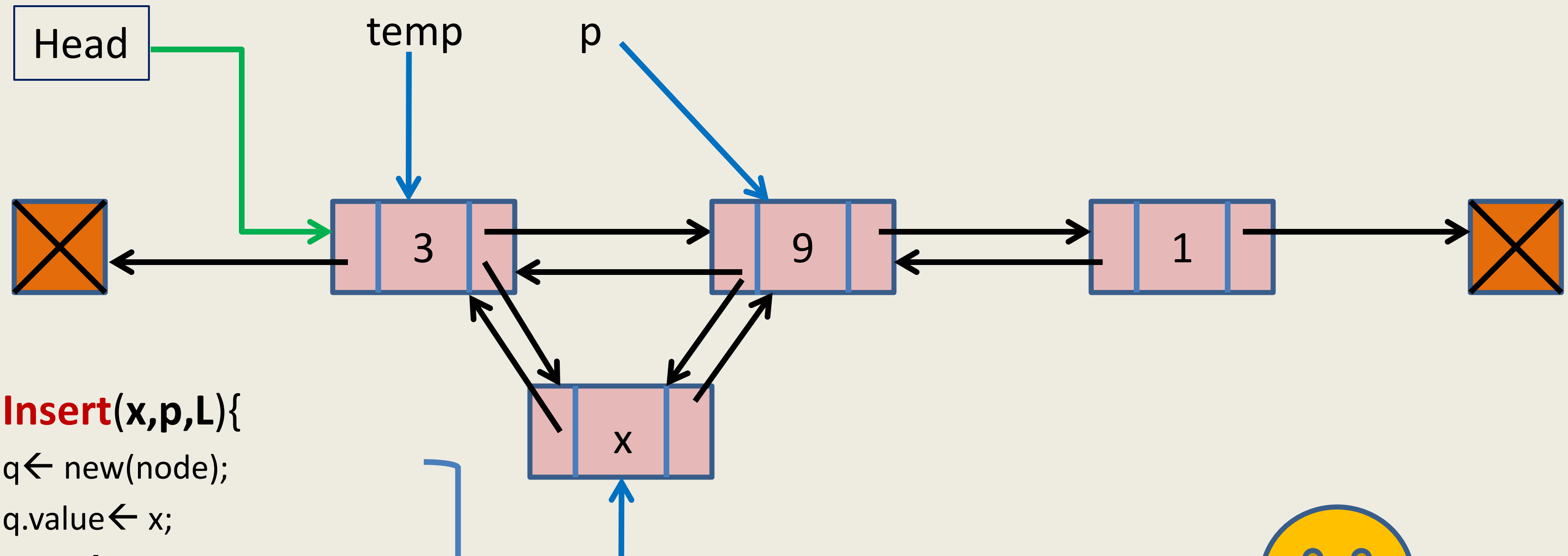


How to perform **Insert**(**x,p,L**) ?



How is it done actually ?

How to perform **Insert**(**x,p,L**) ?



Insert(**x,p,L**){

q← new(node);

q.value← x;

temp← p.left;

q.right ← p;

p.left← q;

q.left ← temp;

temp.right← q;

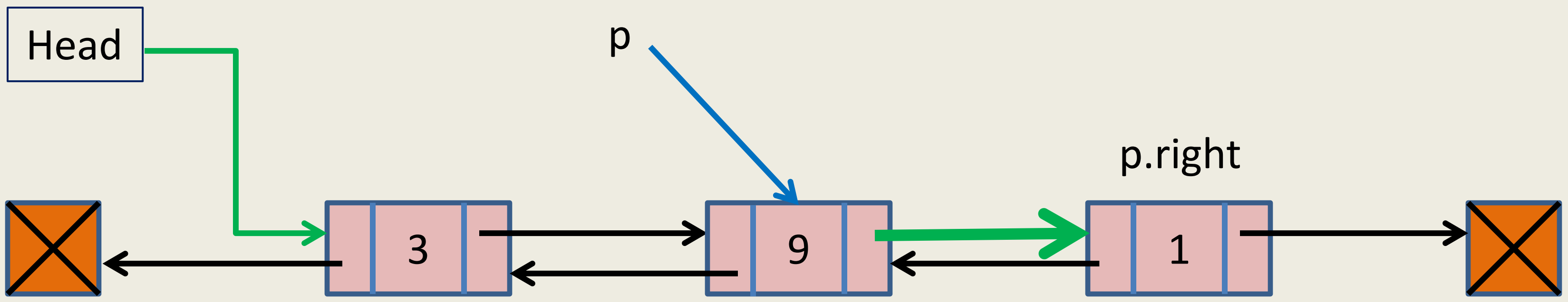
}

O(1) time

Lists are very **flexible**

Homework: How to perform **Delete**(**x,p,L**) ?

How to perform **successor**(**p,L**) ?



Successor(**p,L**){
q← p.right;
return q.value;
}

Time Complexity of each List operation using Doubly Linked List based implementation

Operation	Time Complexity per operation
IsEmpty(L)	O(1)
Search(x,L)	O(n)
Successor(p,L)	O(1)
Predecessor(p,L)	O(1)
CreateEmptyList(L)	O(1)
Insert(x,p,L)	O(1)
Delete(p,L)	O(1)
MakeListEmpty(L)	O(1)

It takes O(1) time if we implement it by setting the **head** pointer of list to NULL. However, if one has to **free** the memory used by the list, then it will require traversal of the entire list and hence O(n) time. You might learn more about it in Operating System course.

Homework: Write **C** Function for each operation with matching complexity.

Doubly Linked List based implementation versus array based implementation of "List"

Operation	Time Complexity per operation for array based implementation	Time Complexity per operation for doubly linked list based implementation
IsEmpty(L)	O(1)	O(1)
Search(x,L)	O(n)	O(n)
Successor(p,L)	O(1)	O(1)
Predecessor(p,L)	O(1)	O(1)
CreateEmptyList(L)	O(1)	O(1)
Insert(x,p,L)	O(n)	O(1)
Delete(p,L)	O(n)	O(1)
MakeListEmpty(L)	O(1)	O(1)

A CONCRETE PROBLEM

Problem

Maintain a telephone directory

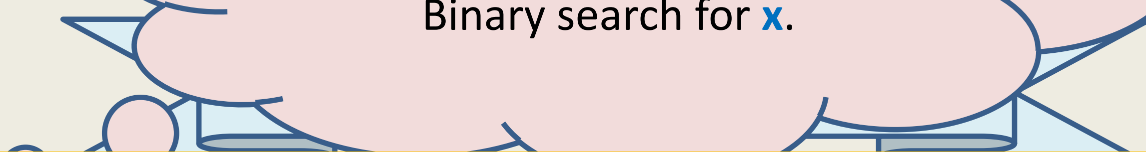
Operations:

- Search the phone # of a person with name **x**
- Insert a new record (name, phone #,…)

Array based solution	Linked list based solution
Log n	**O(n)**
O(n)	**O(1)**

…er it …

Yes. Keep the array sorted according to the **names** and do Binary search for **x**.



We shall together invent such **a novel data structure** in the next class

Important Advice

In this lecture, it was assumed that the students have a basic knowledge of records and singly linked lists from ESC101. In case, you lack this basic knowledge, you are advised to revise the basic concepts of **pointers, records** in C from ESC101. This will also be helpful for some programming assignment in future as well.

In case you need some assistance in these fundamentals, send email to TA **Mr. Piyush Bhardwaj** (piyushb@cse.iitk.ac.in)

PROOF OF CORRECTNESS OF ALGORITHMS

GCD

GCD(a,b)   // $a \geq b$

{

  while (b <> 0)

  { $t \leftarrow b$;

    $b \leftarrow a$ mod b ;

    $a \leftarrow t$

  }

  return a;

}

Lemma (Euclid):

If $n \geq m > 0$, then

gcd(n,m) = **gcd**(m,n mod m)

Proof of correctness of GCD(a,b) :

Let a_i : the value of variable a after ith iteration.

b_i : the value of variable b after ith iteration.

Assertion $P(i)$: **gcd**(a_i, b_i) = **gcd**(a,b)

Theorem : $P(i)$ holds for each iteration $i \geq 0$.

Proof: (By induction on i).

Base case: ($i = 0$) hold trivially.

Induction step:

(Assume $P(j)$ holds, show that $P(j+1)$ holds too)

$P(j)$ ➔ **gcd**(a_j, b_j) = **gcd**(a,b). ----(1)

$(j+1)$ iteration ➔ a_{j+1}= b_j **and** b_{j+1}= a_j mod b_j ---(2)

Using **Euclid**'s Lemma and (2),

gcd(a_j, b_j) = **gcd**(a_{j+1}, b_{j+1}) ------(3).

Using (1) and (3),assertion $P(j+1)$ holds too.

Sum of first n positive integers

```
Sum(n)     // n is  a positive integers ≥ 1
{     sum ← 0;
          i ← 1;
      while (i ≤ n)
      {     sum ← sum + i;
            i ← i + 1;
      }
      return sum;
}
```

Homework:

Write a formal proof to show that **Sum**(n) returns the sum of first n positive integers ?