

# M5. List ADT and Linked List DS

Instructor: Manikandan Narayanan

Weeks 4-5

CS2700 (PDS) Moodle: <https://courses.iitm.ac.in/course/view.php?id=4892>

# Acknowledgment of Sources

- Slides based on content from related
  - Courses:
    - IITM – Profs. **Rupesh**/Krishna(S)/Prashanth/Kartik’s PDS (Thy/Lab) offerings (slides, quizzes, notes, lab assignments, etc. for instance from Rupesh’s Jul 2019 offering - [www.cse.iitm.ac.in/~rupesh/teaching/pds/jul19/](http://www.cse.iitm.ac.in/~rupesh/teaching/pds/jul19/) )
    - *Most slides are based on Rupesh Nasre’s slides – we thank him and acknowledge by marking [RN] in the bottom right of these slides.*
    - *Concept of “**current position**” in a list can be realized inside the List class itself, or using a separate class called Iterator as explained in the link below.*
      - <https://opensa-server.cs.vt.edu/OpenDSA/Books/CS2/html/ListADT.html>
  - Books:
    - **Main textbook:** “*Data Structures and Algorithm Analysis in C++*” by **Weiss** (content, figures, slides, exercises/questions, etc.). – cited as [WeissBook]
    - Additional/optional book: “*Practice of Programming*” by *Kernighan and Pike* (style of programming, programming exercises/questions, etc.) – cited as [KPBook]

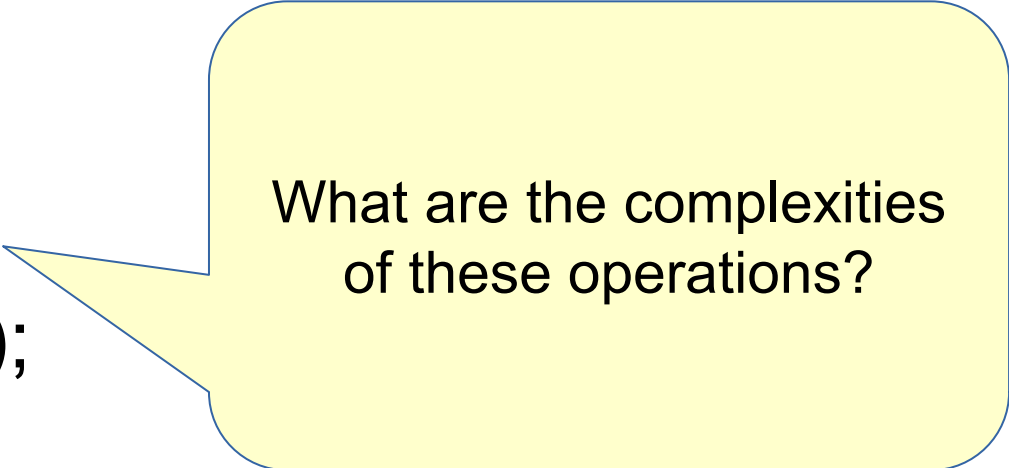
# Outline for Module M5

- M5 List
  - **M5.1 List ADT – implemented via Array DS vs. Linked List DS**
  - M5.2 (Singly) Linked List DS
  - M5.3 Doubly Linked List DS
  - M5.4 Some Linked List Applications/Exercises

# List as an ADT (aka Sequence ADT)

**List:** ordered collection of items (of the same primitive or complex data type) accessible by a **Node-based location** (with **next** operation)  
(contrast with **arrays** that are accessible via **integer-based index**)

```
class List {  
public:  
    List();  
    void insert(Element e);  
    bool find(Element e);  
    void remove(Element e);  
    void print();  
    int size();  
};
```



What are the complexities of these operations?

# List ADT using Array DS

```
class List {  
public:  
    List();  
    void insert(Element e);  
    bool find(Element e);  
    void remove(Element e);  
    void print();  
    int size();  
};
```

4	2	7	2	9			
---	---	---	---	---	--	--	--

## Design decisions

- Size of the array?
- Maintain size separately or use a sentinel?
- On overflow: error or realloc?
- On underflow: error message or exit or silent?
- Printing order?
- Duplicates allowed?
- For duplicates, what does remove do? [RN]

# List ADT using Array DS

```
class List {  
public:  
    List();  
    void insert(Element e);  
    bool find(Element e);  
    void remove(Element e);  
    void print();  
    int size();  
};
```

4	2	7	2	9			
---	---	---	---	---	--	--	--

**With certain design decisions:**

**$O(1)$**

**$O(N)$**

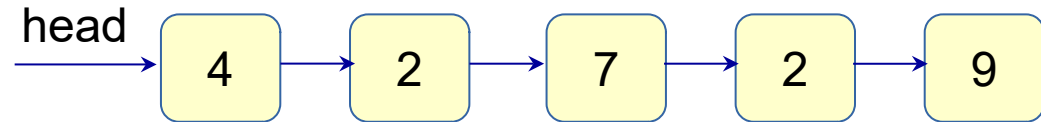
**$O(N)$**

**$O(N)$**

**$O(1)$**

# List ADT using Linked List DS

```
class List {  
public:  
    List();  
    void insert(Element e);  
    bool find(Element e);  
    void remove(Element e);  
    void print();  
    int size();  
};
```



**$O(N)$  without tail pointer, else  $O(1)$**

**$O(N)$**

**$O(N)$**

**$O(N)$**

**$O(1)$**

If the complexities of array-based versus linked-list-based implementations are the same, **why use linked lists?**

# Arrays versus Linked Lists

- Need to copy the existing array on reallocation.
- Removal of  $i^{\text{th}}$  element needs element-shifting from  $i+1$  to end.
- Same with insertion.
- Array concatenation is linear time.
- Only a link needs to be established ( $O(1)$ ).
- Removal of an element using pointers can be done in  $O(1)$ .
- Same with insertion.
- List concatenation is  $O(1)$ .



# A note on “current position” in List ADT

Concept of “current (Node-based location or) position” in List ADT is a key difference from integer-based index in Array ADT. Current position in a List ADT supports:

- **Iteration:** Current position can be set to first Node’s location in the list, advanced (via next opn.) to the next Node’s location, and checked against a final sentinel Node’s location in list.
- **Operations:** In certain List ADTs, insert and remove operations can be defined based on current position in a list.
- **Current position can be realized:** inside the List class itself, or using a separate class called Iterator as explained in this link: <https://opensa-server.cs.vt.edu/OpenDSA/Books/CS2/html/ListADT.html>

In this course, we will implement the “current position” using a separate variable of type “Node\*” (so, it is closer to the Iterator interface).

```
//Ex. impl. of current position in this course
ListNode *cur; //head=lst.head;
for (cur=head; cur!=NULL; cur=cur->next) {  cout << cur->value;  }
```

```
//Above code similar to iterator interface/impl. shown below:
ListIterator cur;
for (cur=lst.begin(); cur!=lst.end(); cur++) {  cout << cur-
>getValue();  }
```

# Outline for Module M5

- M5 List
  - M5.1 List ADT – implemented via Array DS vs. Linked List DS
  - **M5.2 (Singly) Linked List DS**
  - M5.3 Doubly Linked List DS
  - M5.4 Some Linked List Applications/Exercises

# Linked List Implementation

- **Source:**

sl\_code\_templateNode.cpp – 1<sup>st</sup> trial

sl\_code\_**templateElement**.cpp – 2<sup>nd</sup> trial; **Recommended code**

# Practice with pointers

Declare the following:

- Pointer to integer
- Pointer to pointer to integer
- Pointer to array of integers
- Array of pointers to integers
- Access the value in the second node of a linked list pointed to by head.

What does this do?

- `(((*ptr).next)).val = x;` Is it same as
- `ptr->next->val?`

# Pitfalls

- `ptr = head->next;`      *// segfault. Check if head is NULL.*
- `{ ... Node *ptr = &node1; return; }` *// local variable node1.*
- `Node *ptr = malloc(sizeof(Node*));` *//insufficient memory.*

- Should be:

```
Node *ptr = (Node *)malloc(sizeof(Node));  
If (ptr == NULL) { cerr << "Error allocating memory.\n"; exit(-1); }
```

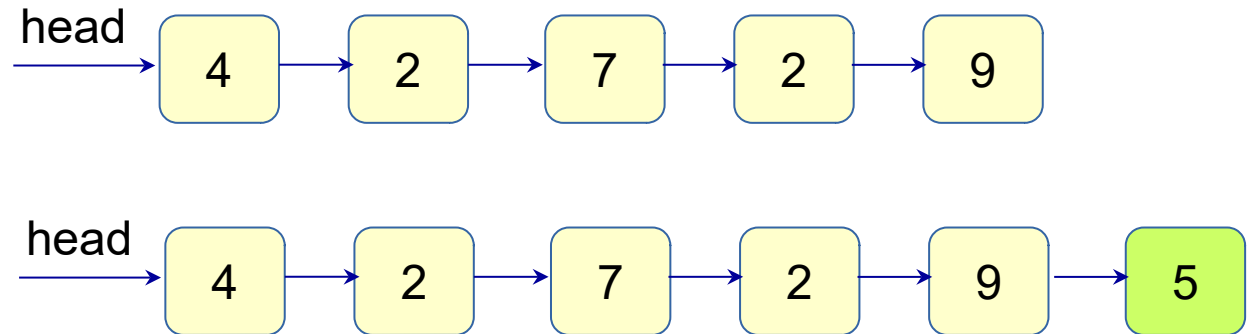
- Wrong deleteList program

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
```

```
    delete ptr;      // invalid memory on delete/free
```

# List insert

- insert(5)



## Setup node:

```
Node *newptr = new Node();  
newptr->val = 5;  
newptr->next = NULL;
```

## End case:

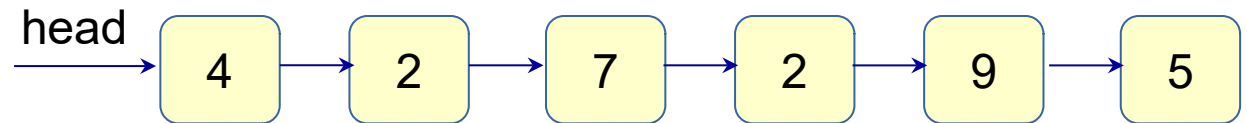
```
if (head == NULL) head = newptr;
```

## Regular case:

```
for (Node *ptr = head; ptr->next != NULL; ptr = ptr->next)  
    ;  
ptr->next = newptr;
```

# List print

- print()



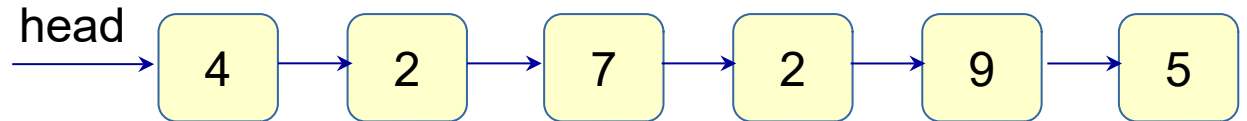
**Output:** 4 2 7 2 9 5

For each element in the list  
Print the element

```
template <typename Node>
void List<Node>::print()
{
    for (Node *ptr = head; ptr != NULL; ptr = ptr->next)
    {
        ptr->print(); //(*ptr).print()
    }
    cout << endl;
}
```

# List find

- find(9)



For each element in the list

    If the element is same as that to be searched

        Found the element

Element not present

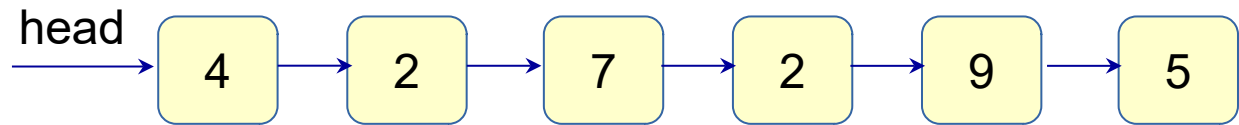
```
for (Node *ptr = head; ptr != NULL; ptr = ptr->next)
    if (ptr->val == val) return true;
return false;
```



# List remove

- remove(2)
- remove(5)
- remove(4)

We want to remove all occurrences of the value.



## Special case:

```
if (head == NULL) return false;
```

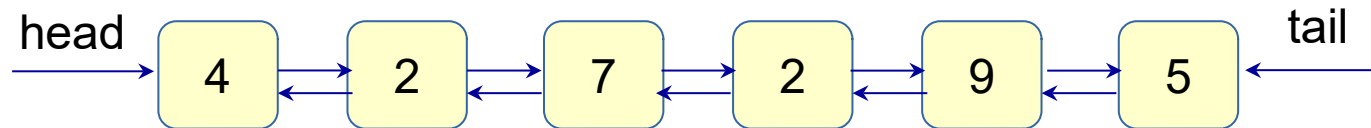
## General case:

```
Node *previous = NULL;
for (Node *ptr = head; ptr;) {
    if (ptr->val == val) {
        Node *toberemoved = ptr;
        if (previous) {
            previous->next = ptr->next;
        } else head = ptr->next;
        ptr = ptr->next;
        delete toberemoved;
        removed = true;
    } else {
        previous = ptr;
        ptr = ptr->next;
    }
}
```

# Outline for Module M5

- M5 List
  - M5.1 List ADT – implemented via Array DS vs. Linked List DS
  - M5.2 (Singly) Linked List DS
  - **M5.3 Doubly Linked List DS**
  - **M5.4 Some Linked List Applications/Exercises**

# Doubly Linked List



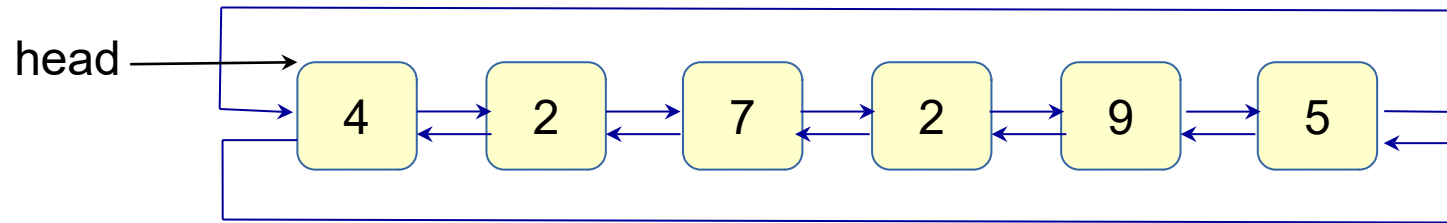
- Links in both the directions.
- Node structure contains two pointers: next and previous.
- Deletion now becomes simpler.
- Two pointers: head and tail maintain list ends.
- **Classwork:** Write a function to remove a node.

# For the Road: From “How to solve it by computer?” by RG Dromey

**Problem:** *Change find() to find\_and\_optimize() such that Most Frequently Accessed (MFA) elements are moved closer to the beginning of the list!*

- 7.4.4 In some applications, particular information needs to be retrieved from a list with a much higher frequency than other information. Under these conditions the best way to arrange the list is to put the most frequently retrieved item at the head of the list, the second most frequently retrieved item in the second position and so on. The appropriate order is often not known in advance. To make a list progress towards the appropriate order whenever an item is retrieved it can be exchanged with its predecessor. Under this scheme the most frequently retrieved items eventually migrate to the front of the list. Design algorithms that search and maintain such a linked list.
- 7.4.5 Another way of performing the task described in 7.4.4 is to always move an item to the head of the list after it is retrieved. Design and implement algorithms that search and maintain a linked list in this form.

# Circular Doubly Linked List



- Last element points to the first, and first element's previous is the last node.
- Node structure continues to contain two pointers: next and previous.
- Tail pointer is **not** required.
- A singly linked list can also be circular.
- **Classwork:** Write a function to print all the node values in a CDLL.

# Standard idioms need to be replaced by something more appropriate for circular LLs!

```
//DLL or SLL
```

```
for (Node* cur=head; cur != nullptr; cur = cur->next) {  
    ...  
}
```

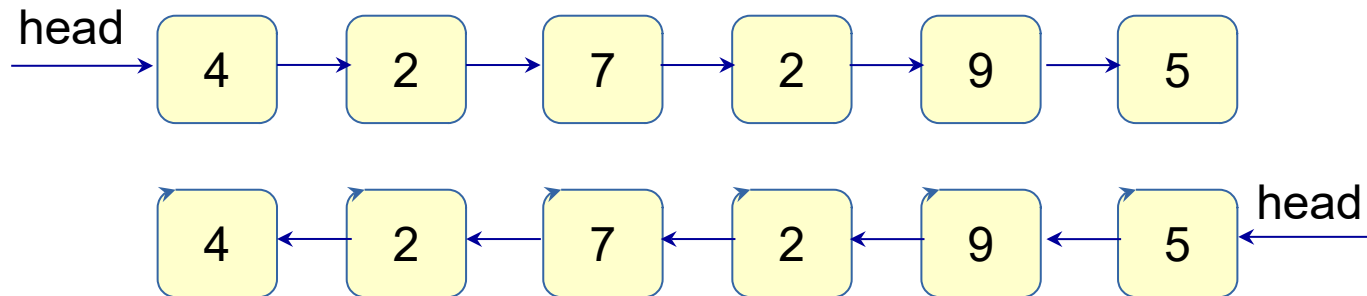
```
//CDLL or CSLL with size field maintained
```

```
for (int i=0, cur=head; i < lst_size; i++, cur = cur->next) {  
    ...  
}
```

```
//CDLL or CSLL without size field
```

```
for (Node *cur=head, isFirstIter=true;  
     (cur != head || isFirstIter) && (head != nullptr);  
     cur = cur->next, isFirstIter = false) {  
    ...  
}
```

# List Reversal



- Given a list (SLL, DLL, CSLL, CDLL), reverse it.
- The traversal from head should result in the opposite order.
- Typically need three pointers: previous, current and next.
- **Classwork**: Complete a list reversal code for SLL (iterative).
- **Classwork**: Complete a recursive list reversal code.

# Recursive Methods

- Sometimes natural to model.
- Sometimes inefficient to implement.
- **Homework**: find an element recursively.
- **Homework**: print a list recursively.
  - How to print in reverse?



# Application: Polynomial ADT

- $F(X) = \sum_{i=0}^N A_i X^i$
- Example:  $x^4 - 4x^3 + 7x - 6$
- **Member functions**
  - Initialize
  - Set a coefficient (for a power)
  - Add polynomials
  - Multiply polynomials
  - ...
- **Implementation**
  - Could be using arrays
  - Could be using linked lists
- **Homework:** Create a struct / class to implement polynomials.
- Are there disadvantages of using arrays?
  - $2x^{1000} - x$
  - What are the design decisions for using lists?

# Application: Polynomial ADT using Array vs. LL DS

```
class Polynomial {  
    //choose one of these two: Array vs. LL  
    int coeff[MaxDegree + 1]; //Array (OR)  
    SLL<Literal> literalsLst; //each literal/monomial has coef and  
    exp  
};  
void Polynomial::initialize(int coeff[ ]) {  
    // Homework: implement this.  
}  
void Polynomial::add(Polynomial p2, Polynomial psum) {  
    // Homework: implement this. (for LL impl., it can help if  
    literals are maintained in sorted order of their exponents)  
}
```