

# EE2331 Data Structures and Algorithms

Linked Lists

# Linear List

- Each element in the list has a **unique predecessor (previous) and successor (next)**.
- **Unordered/Random** list
  - There is **no ordering** of the data.
- **Ordered** list
  - The data are arranged according to a key. A **key** is one or more fields within a structure that is used to **identify** the data or otherwise control its use.
- **General** list
  - Data can be **inserted and deleted anywhere** and there are no restrictions on the operations that can be used to process the list.
- **Restricted** list
  - **Insertion, deletion and processing** of data are **restricted to specific locations**, e.g. **the two ends of the list**. **Stack** and **Queue** are examples of restricted list.

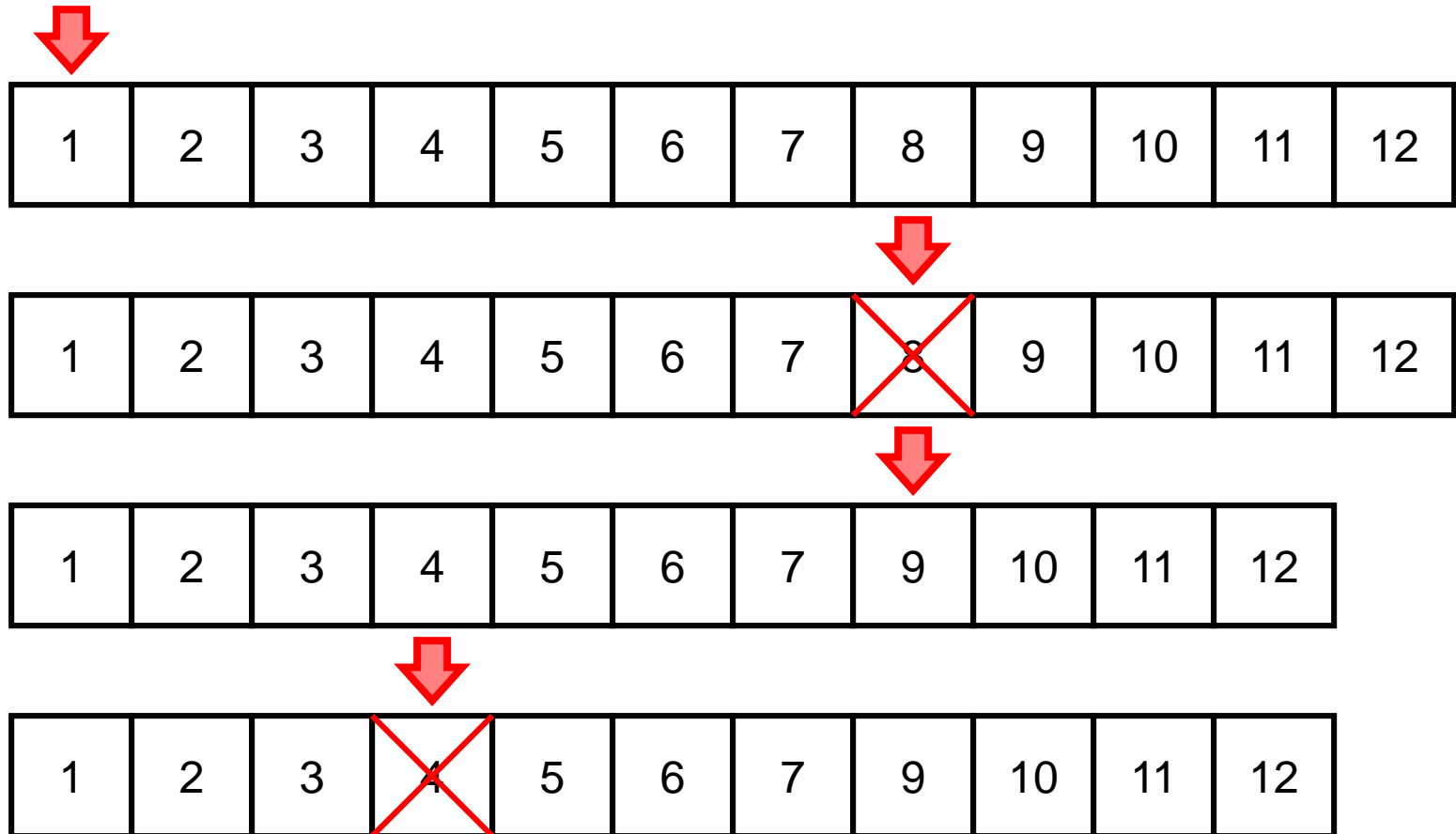
# Josephus Problem



- People are standing in a circle waiting to be executed. Counting begins at a specified point in the circle and proceeds around the circle in a specified direction. After a specified number of people are skipped, the next person is **executed**. The procedure is repeated with the remaining people, starting with the next person, going in the same direction and skipping the same number of people, until only one person remains, and is **freed**.
- The problem — given the number of people ( $n$ ), starting point, direction, and number to be skipped ( $k$ ) — is to choose the position in the initial circle to **avoid execution** (i.e. **guessing who is the survivor**).

# The Josephus Problem

■ If  $k = 7$ ,  $n = 12$



# Array Implementation

- A simple approach is by writing a program to simulate the counting-out game. But what data structure should be use?
- With a list using array implementation
  - Array has the advantage of random access (i.e. direct access to any position)
  - However, the insert and delete operation may involve substantial data movement
  - Another disadvantage of representing a list using an array is that the maximum length of the list needs to be determined a priori

# Linked List & Node Structure

- A sequence of nodes (elements), each containing arbitrary **data** and **links** (pointers) pointing to the next and/or previous nodes

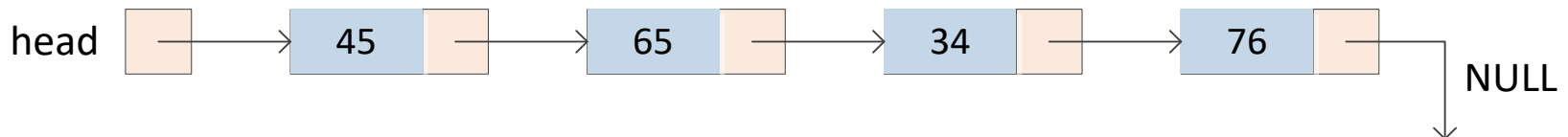


- It contains pointer(s) of the same type
  - Recursive data structure (self-referential datatype)
- A list is formed by **linking nodes** together in a **sequential manner**
- In typical C++ implementations, we shall define the node using **struct**, and the linked list is defined as a **class**.

```
struct node {  
    int info;           // data  
    node* link;         // another commonly used var name is "next"  
};  
  
node* head;             // head pointer pointing to first node
```

# Linked List Example

- In the C++ terminology, a linked list is classified as a **container**. In Java, it is called a **collection**.
- The address of the first node in the list is stored in a separate pointer variable usually called **head**, **first**, or **list**.
- The **null pointer** (NULL, physical value **0** in C/C++) is used to denote the end of the list, or not a valid address.
- Example: a linked list of 4 integers.



# **Common Operations on Linked List**



# Find Length & Find Node

- To find the **length** of the linked list

```
int len = 0;
node *cur = head;           //traverse the list using cur

while (cur != NULL) {       // cur points to a valid node
    len++;
    cur = cur->link;         //move to the next node
}
```

- To **search** an element x from the beginning of the list

```
node *cur = head;

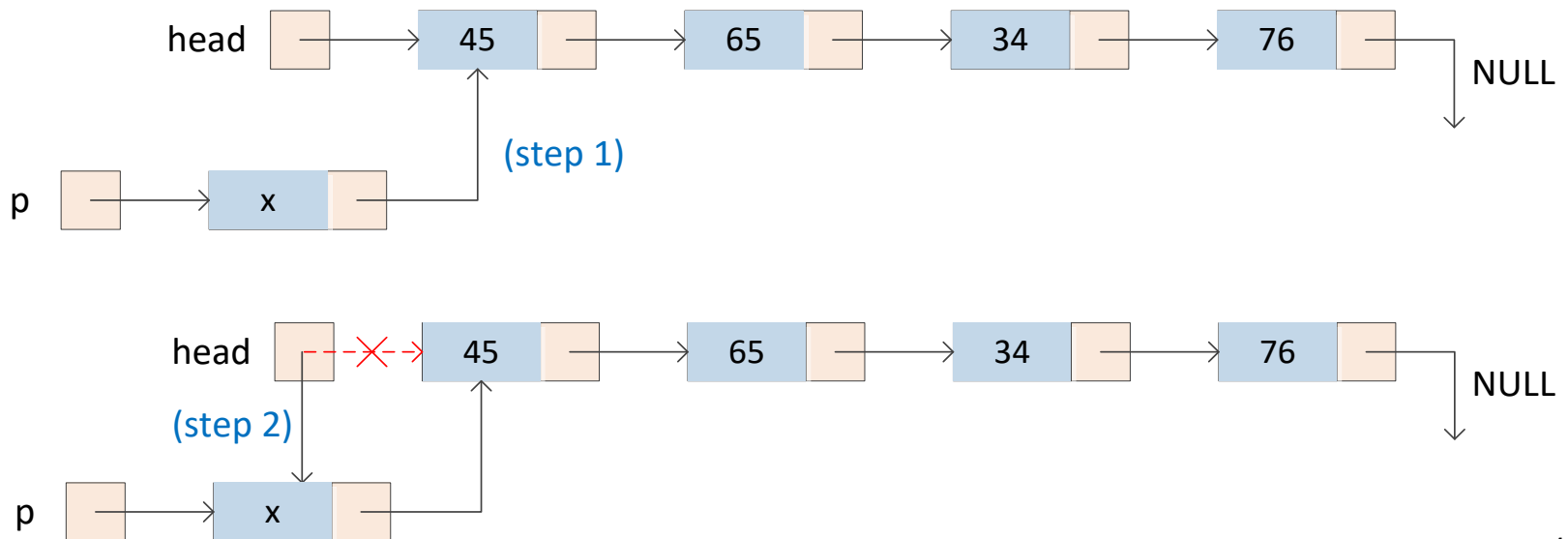
while (cur != NULL && cur->info != x) {           // search x
    cur = cur->link;
}
// cur now points to target x or is NULL (x not exist)
```

# Insert Node (at front)

- Insert a new element x at the front of the list

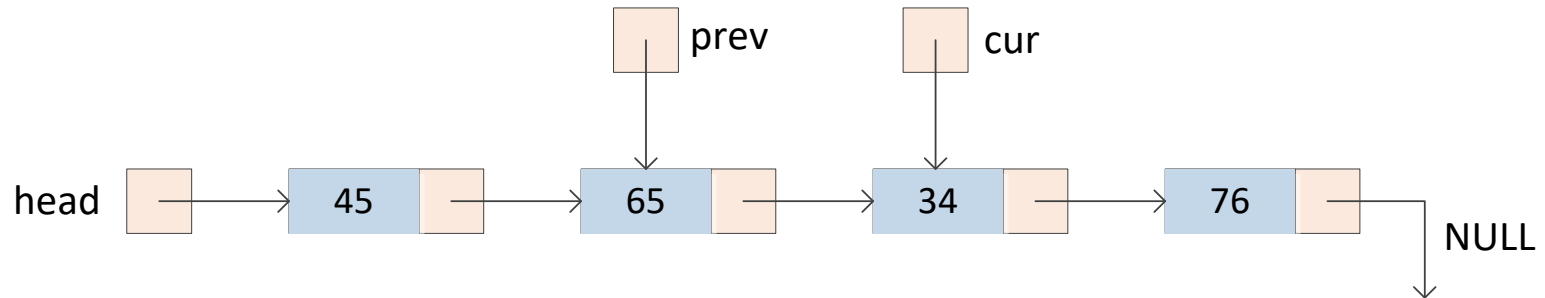
```
node *p = new node;           // create the node dynamically for storing x
p->info = x;

p->link = head;                // step 1
head = p;                      // step 2
```

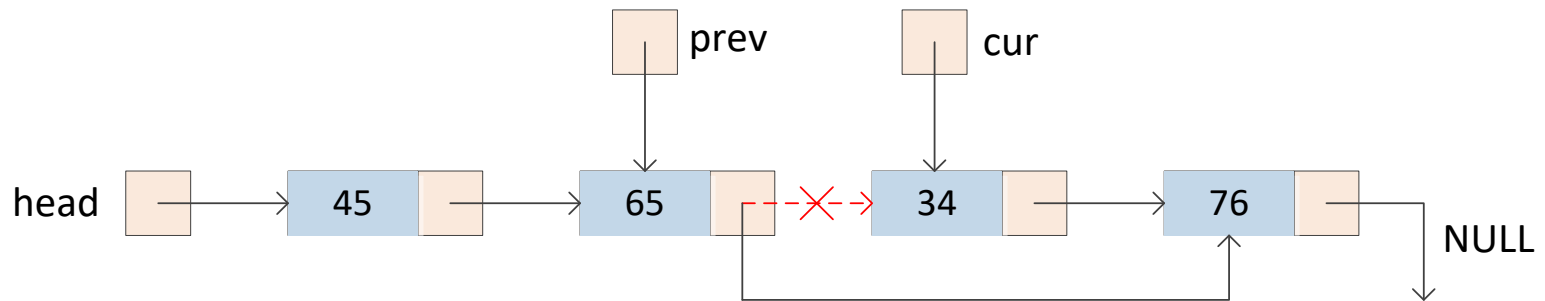


# Remove Node

1. Locate the node storing the value  $x$ , e.g.  $x = 34$

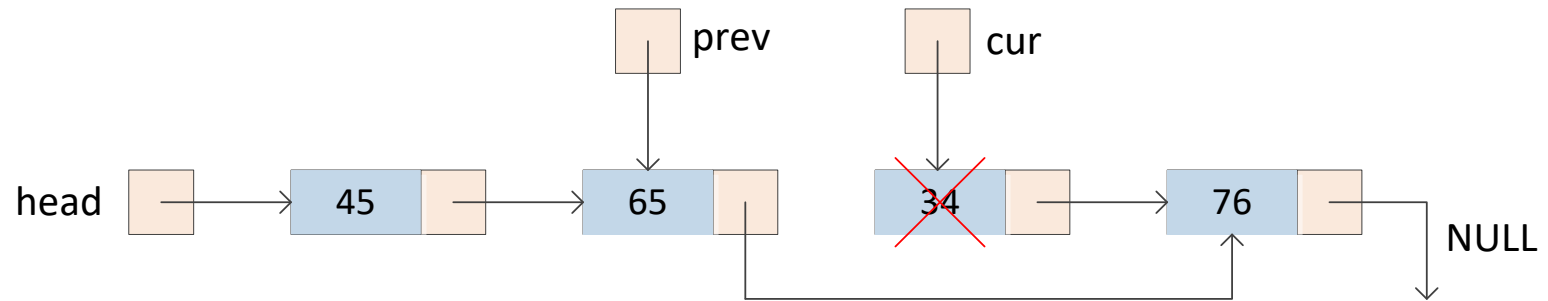


2. Update the links

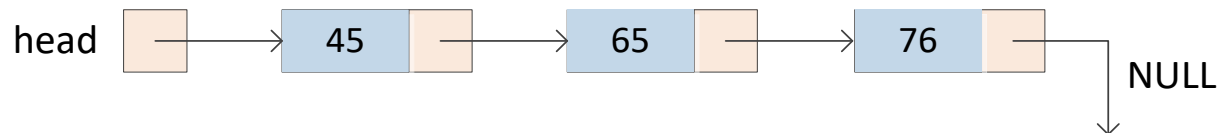


# Remove Node (cont.)

## 3. Physically delete the node



## 4. Structure of the list after removing the element 34



# Remove Node (cont.)

- Remove the element x (1<sup>st</sup> instance) from the linked list
- To remove a node from the linked list, we need to know the reference to its predecessor.

```
node *cur = head;
node *prev = NULL;           // prev points to the predecessor of cur

while (cur != NULL && cur->info != x) {           // search x
    prev = cur;
    cur = cur->link;
}

// if cur == NULL, x is not found in the linked list
if (cur != NULL) {           // cur->info == x
    if (prev != NULL)        // why checking this?
        prev->link = cur->link; // skip cur node
    else
        head = cur->link;    // cur is the first node in the list
    delete cur;             // x is the first node
                             // free the storage of the removed node
}
```

# Insert Node

- Insert a new element x into an **ordered** list

```
node *p = new node;
p->info = x;

if (head == NULL || x <= head->info) {
    p->link = head;           //insert at front
    head = p;
} else {                     //head != NULL && x > head->info
    node *prev = head;       // x to be inserted between prev and cur
    node *cur = head->link;   // i.e. prev->info < x <= cur->info

    while (cur != NULL && x > cur->info) {    // search position
        prev = cur;
        cur = cur->link;
    }

    // x not exist OR x <= cur->info, so insert node p after node prev
    p->link = prev->link;
    prev->link = p;
}
```

# Remark

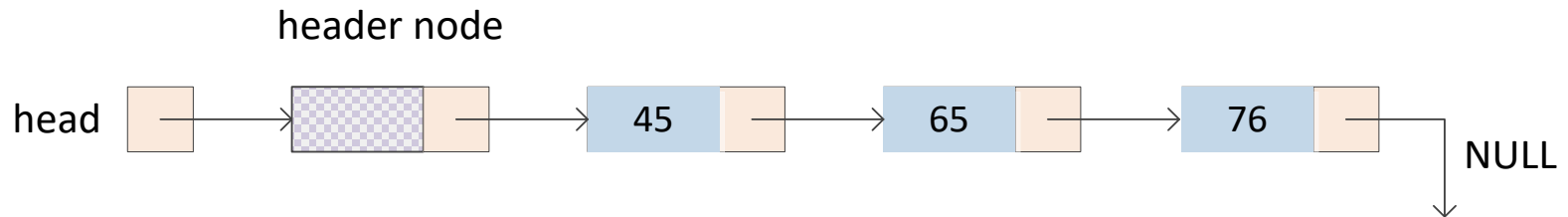
- Because you **can't go backward** in a **singly linked** list, for removing/inserting node, you usually need to use a **pair of pointers** – **predecessor** and **current**, to keep track of the previous node and update its link.
- For a linked list of size  $n$ , you generally should test your algorithm against the two **boundary cases**:  **$n=0$**  and  **$n=1$**  in addition to the general case.
- **Common Problems**
  - **Null-pointer exception** is a common error in programs that manipulate linked list.
  - A pointer **must be properly initialized or tested for not equal to NULL** before you can use it to access a data member (dereferencing) or the next node.
  - **Broken list** due to deletion of nodes
  - Losing reference to some nodes (**memory leak**)

# **Other Variants of Linked List**

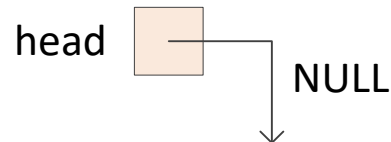


# Linked List with Header Node

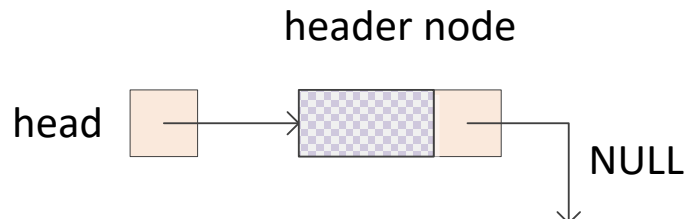
- The data field of the header node is **NOT** used to store valid data. Some **metadata** may be stored in the **header node**.



- List does not exist (or not yet created):



- List is empty:



# Why Header Node?

- Header node is guaranteed to **exist at all times**.
- Header node makes all list nodes intrinsically the same – **having a predecessor**.
- Having a sentinel zeroth node simplifies a lot of operations you might want to perform on a linked list - for example, a lot of operations no longer need to explicitly check for an empty list.

```
// Simplified version: Insert a new element x into an ordered list
node *p = new node;
p->info = x;

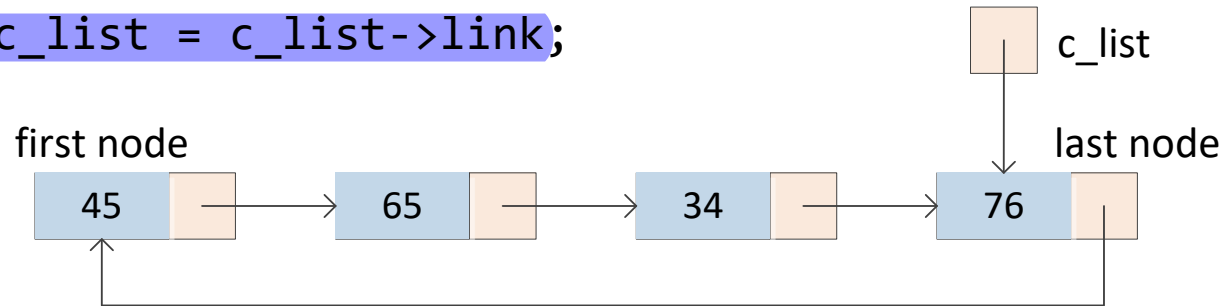
node *prev = head;           // point to the dummy header
node *cur = head->link;      // point to 1st node

// while (x > cur->info && cur != NULL)           // Is it the same?
while (cur != NULL && x > cur->info) {           // search position
    prev = cur;
    cur = cur->link;
}
p->link = prev->link;
prev->link = p;
```

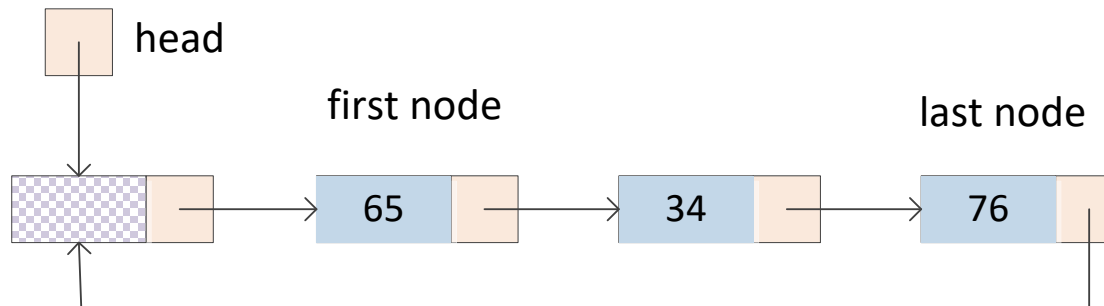
# Circularly Linked List

- The link of the last node points back to the first node.
- When the pointer **c\_list** reaches the last node in the list, it can re-visit the first node in one step:

```
c_list = c_list->link;
```



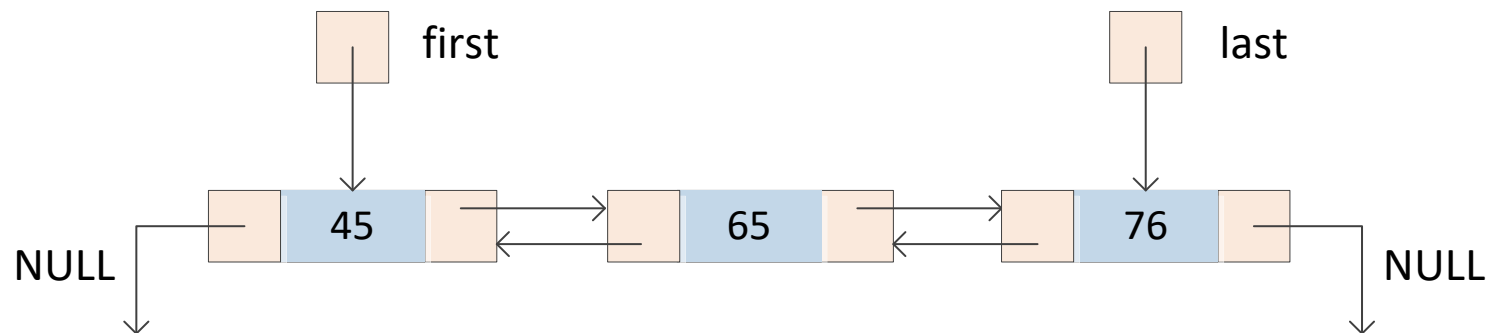
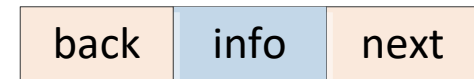
- Circular list with header



# Doubly Linked List

- With a doubly-linked list, we can traverse the list in the forward or backward direction.

```
template<class Type>
struct nodeType {
    Type info;
    nodeType<Type> *next;    //points to successor node
    nodeType<Type> *back;    //points to predecessor node
}
```



# Insert Node on Doubly Linked List

- Insert element x **after** the **current** node in a doubly-linked list

```
nodeType<Type> *p = new nodeType<Type>;  
p->info = x;
```

*// assume current is a pointer to current node*

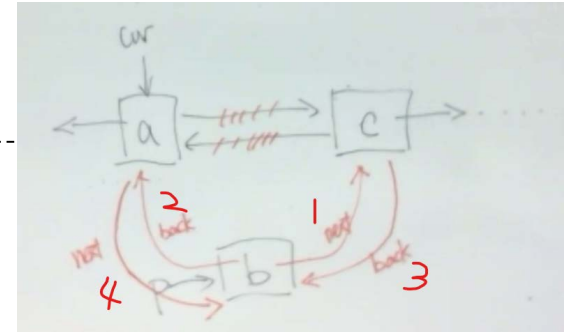
```
p->next = current->next;    //(1)
```

```
p->back = current;         //(2)
```

```
if (current->next != NULL)  //current has a successor
```

```
    current->next->back = p;  //(3)
```

```
current->next = p;          //(4)
```



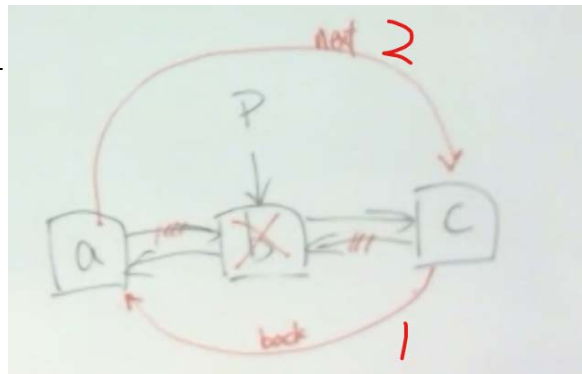
# Remove Node on Doubly Linked List

- Remove the node pointed by **p** in a doubly-linked list

```
if (p->next != NULL)
    p->next->back = p->back;           //(1)

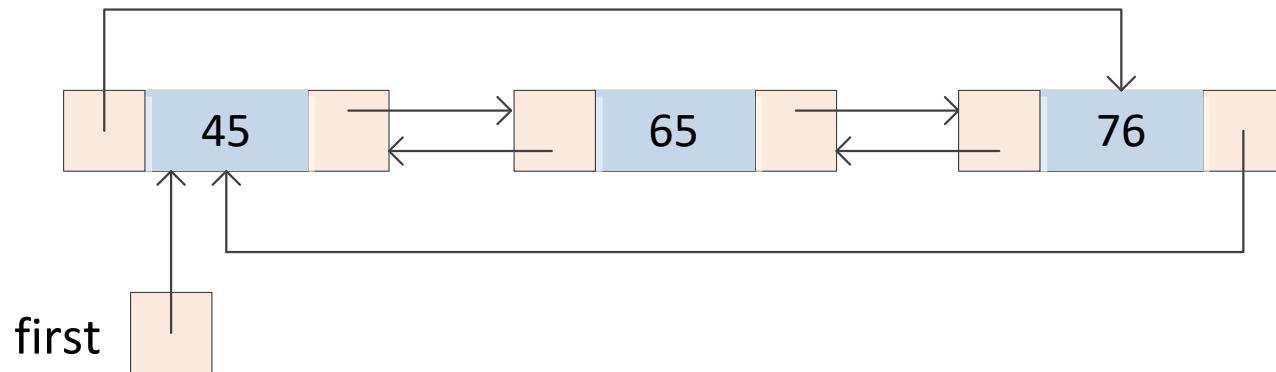
if (p->back != NULL)
    p->back->next = p->next;           //(2)

delete p;
```



# Circular Doubly Linked List

- Circular doubly-linked list is used in the OS for dynamic memory allocation and de-allocation. The algorithm is called **boundary-tag method**.



# Orthogonal List

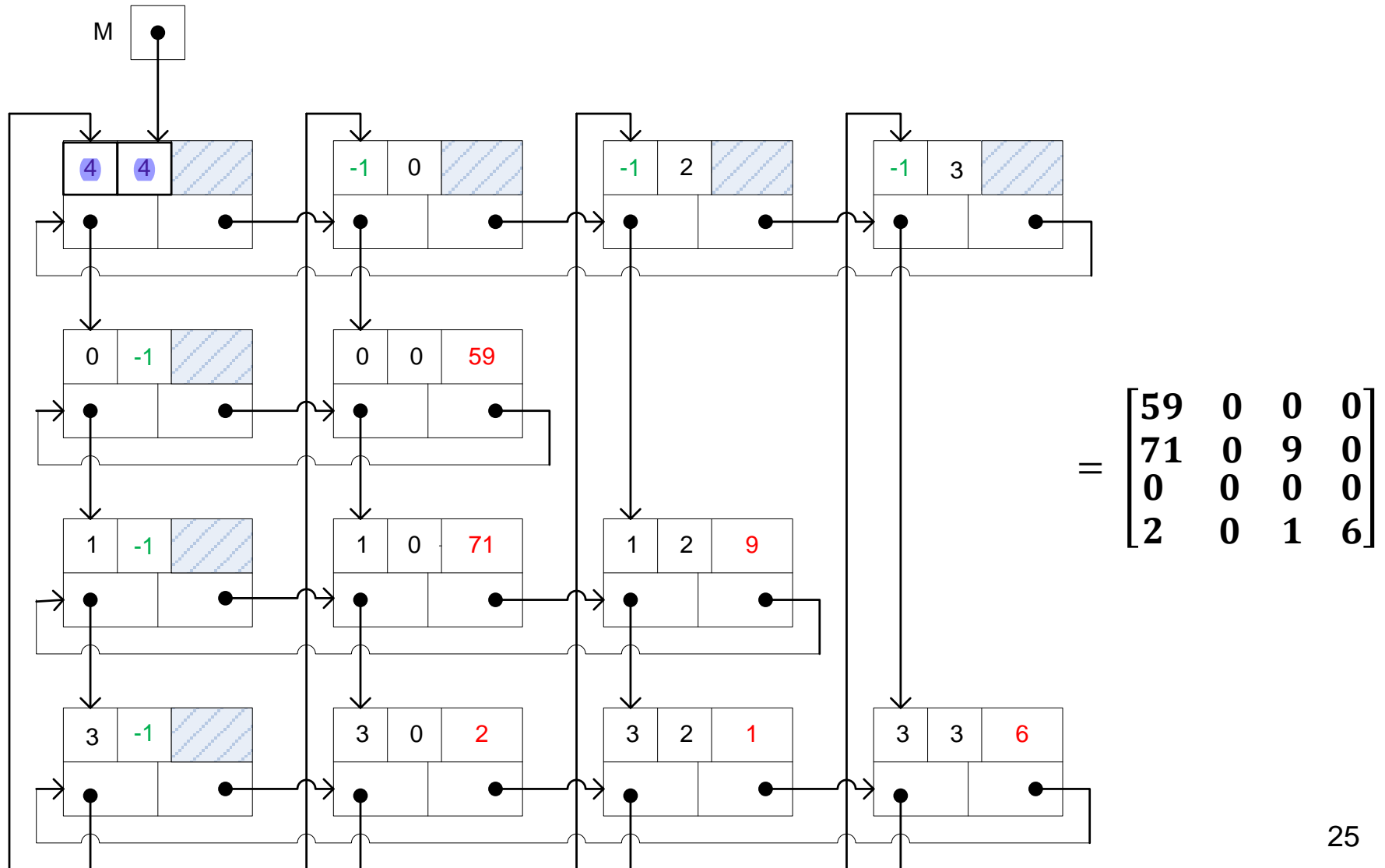
- Linked representation of sparse matrix
- **Non-zero** elements in a row (or column) are connected in a **circular linked list with header**.

```
struct node {  
    int row, col;  
    double value;  
    node *down, *right;  
};  
node *M;           //reference pointer to the sparse matrix
```

■ Example:  $M = \begin{bmatrix} 59 & 0 & 0 & 0 \\ 71 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 1 & 6 \end{bmatrix}$



In this example, values of **nRow** and **nCol** of the sparse matrix are stored in the **dummy header** node pointed at by **M**. The header of each circular list stores the **respective row/column number**.



# Get Value from Matrix

```
//Return the value of matrix[i][j]
//Precondition: i < nRow and j < nCol

double getValue(node *m, int i, int j) {
    node *p, *q;
    int found;

    // search vertically
    found = 0;
    p = m->down;
    while (p != m && !found) {
        if (p->row < i)
            p = p->down;
        else if (p->row == i) {
            //found the row
            found = 1;
        } else {
            // row not exist!
            // terminate the loop
            p = m;
        }
    }
}
```

```
// then search horizontally
//p points to header of row i
if (found) {
    q = p->right;
    while (q != p) {
        if (q->col < j)
            q = q->right;
        else if (q->col == j) {
            //value of matrix[i][j]
            return q->value;
        } else {
            // col not exist!
            //terminate the loop
            q = p;
        }
    }
}
// not exist -> 0
return 0; //matrix[i][j] == 0
}
```

# Class Exercise

- Search last node of a list
- Remove last node of a list
- The given singly linked list has a ***dummy header***
  - Input: a pointer to node structure, *list*, which points to the head of a linked list
  - Precondition: *list* is a valid linked list with header

# Search the Last Node


// Output: a pointer  $p$  points to the last node of the list  
node\* searchLastNode(node \*list) {

# Remove the Last Node

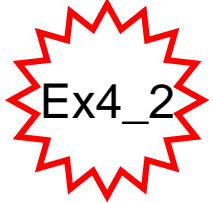
```
// singly linked list with header hode  
void removeLastNode(node *list) {
```

```
}
```

# Linked List C++ Implementation



Ex4\_1



Ex4\_2

- Operations that we would perform on a linked list:
- Initialize the list.
- Clear the list.
- Determine if the list is empty.
- Print the list.
- Find the length of the list.
- Make a copy of the list, e.g. **assignment operator=** and the **copy constructor**.
- Search the list for a given item.
- Insert an item to the list.
  - The requirement of the insert operation depends on the representation invariant or the intended uses of the list.
  - For ordered list, we need to maintain the ordering of list elements.
  - If it is used as a queue, insertion is performed at the rear (end of list).
  - If it is used as a stack, insertion is performed at the front.

# Linked List C++ Implementation

- Remove an item from the list. Similar to the case of insertion.
- Traverse the list (in the application program that uses the linked list object), i.e. retrieve the elements one by one (in some specific order) to carry out the required computation on each node.
  - To implement the traversal, we shall make use of an **iterator**.
  - A linked list is a **container** that holds together a collection of items.
  - An iterator is an object that produces each element of a container, one at a time.
  - The two basic operations on an iterator are the **dereference operator \***, and the **pre-increment operator ++** (advance to the next element).
- There can be other operations on the linked list, e.g. reverse the list, merge two lists, etc.
- We want the linked list class to be **generic** such that it can be used to process different data types.