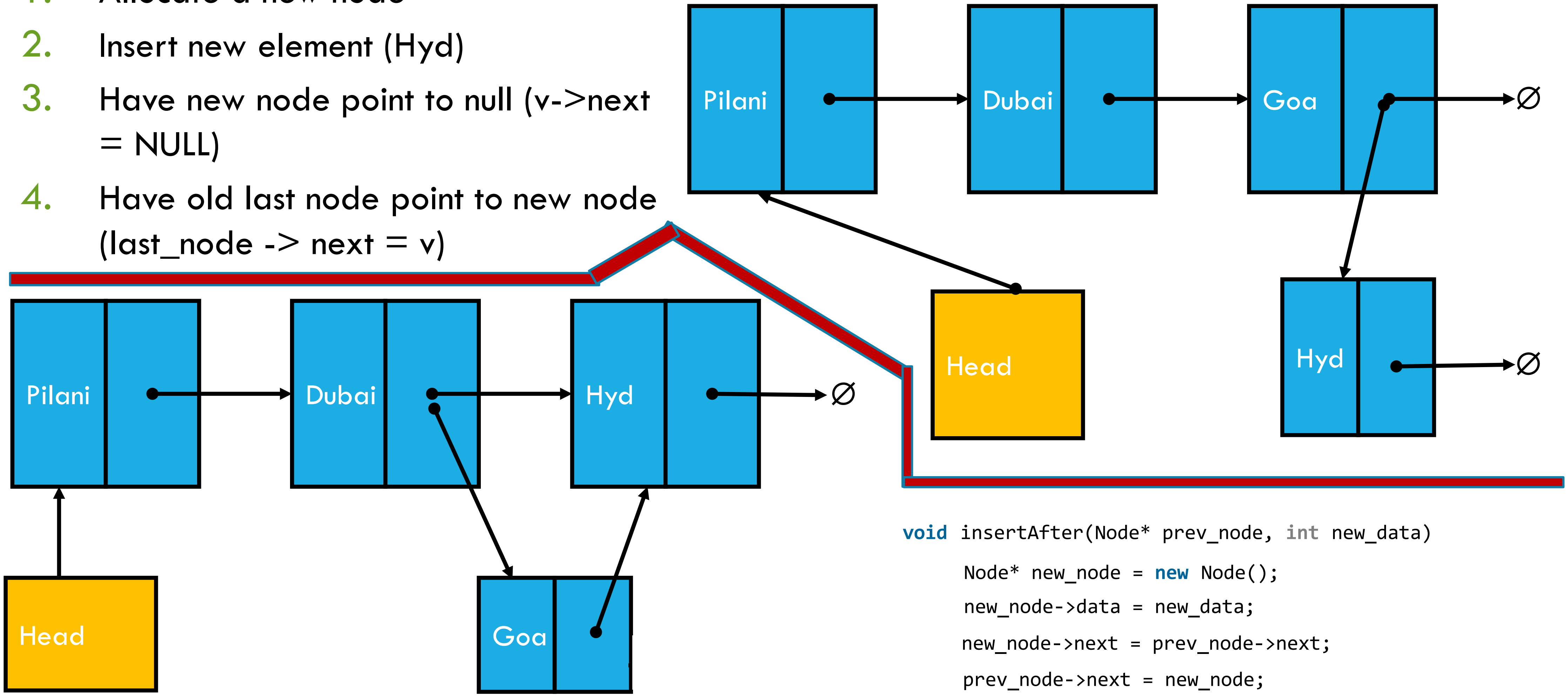
BITS F232: FOUNDATIONS OF DATA STRUCTURES & ALGORITHMS (1ST SEMESTER 2023-24) LINKED LISTS CONTINUED…

Chittaranjan Hota, PhD
Sr. Professor of Computer Sc.
BITS-Pilani Hyderabad Campus
hota[AT]hyderabad.bits-pilani.ac.in

INSERTING AT THE TAIL & INSIDE A LINKED LIST

1. Allocate a new node
2. Insert new element (Hyd)
3. Have new node point to null (v->next = NULL)
4. Have old last node point to new node (last_node -> next = v)

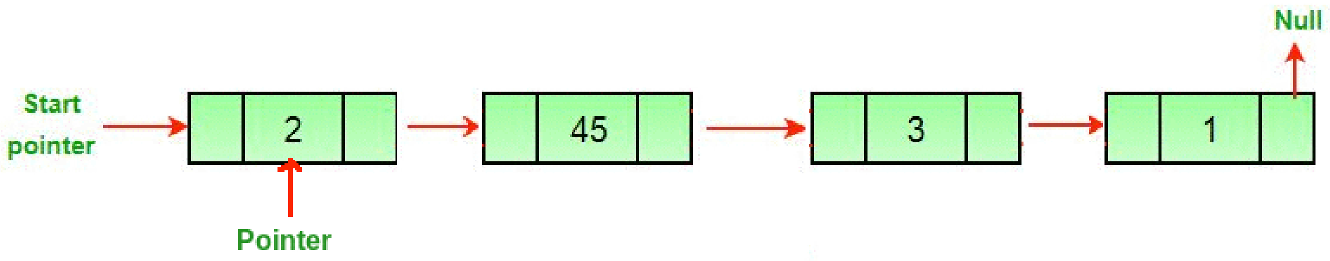


```
void insertAfter(Node* prev_node, int new_data)
      Node* new_node = new Node();
      new_node->data = new_data;
      new_node->next = prev_node->next;
      prev_node->next = new_node;
```

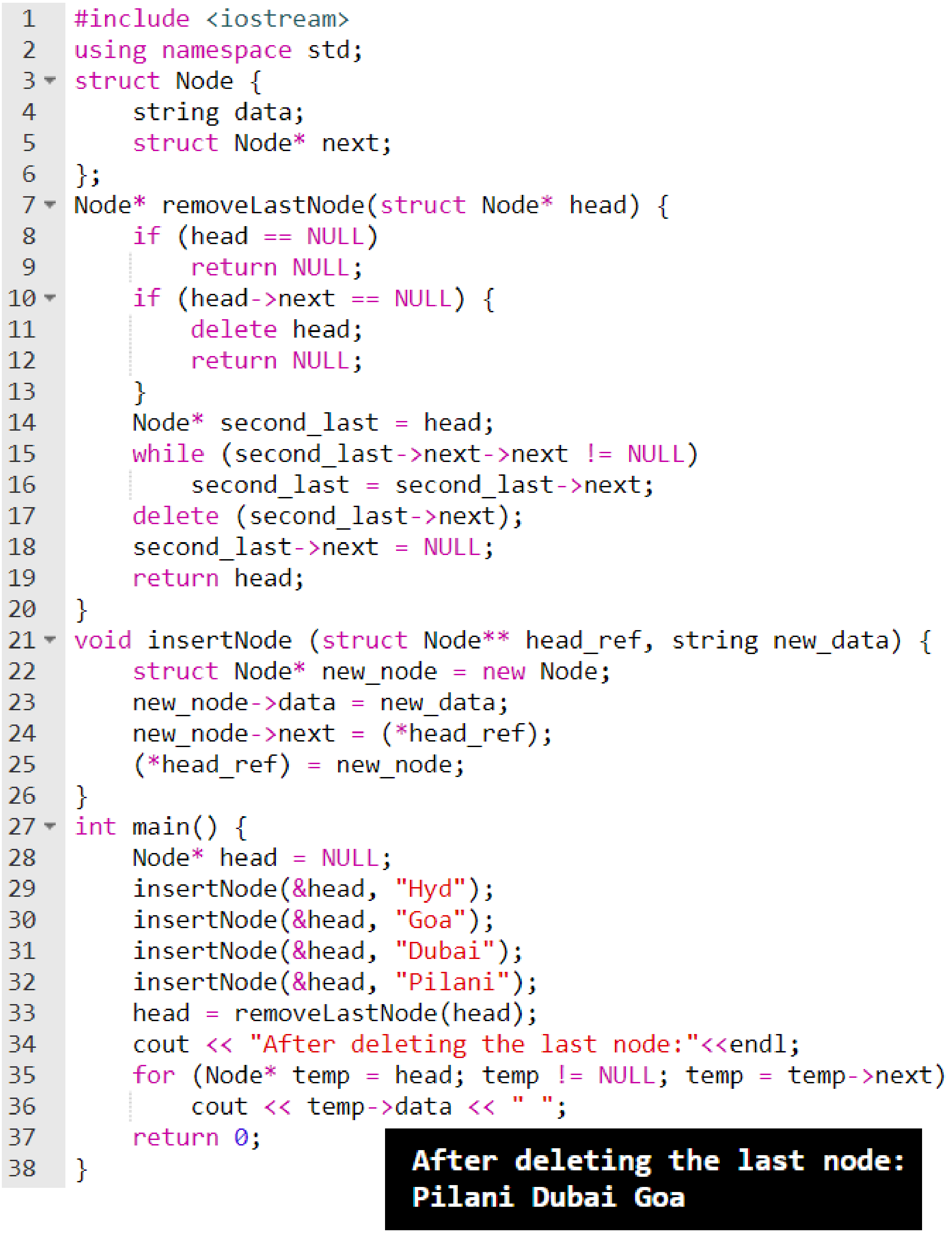
DELETING THE LAST NODE

Algorithm:

```
1. If (headNode == null)  //if the first node is null
       then return null
2. If (headNode.next == null) //if there is only one node
       then free head and return null

3. while secondLast.next.next != null //traverse till secondLast
       secondLast = secondLast.nextNode

4. Delete last node and set the pointer of secondLast to null.
```



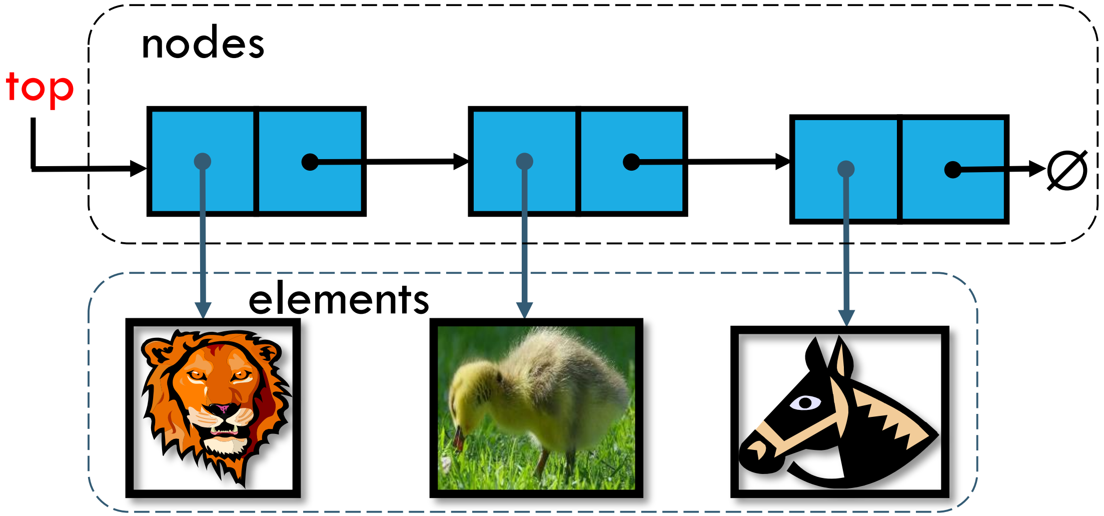
Img. Source: https://www.geeksforgeeks.org/

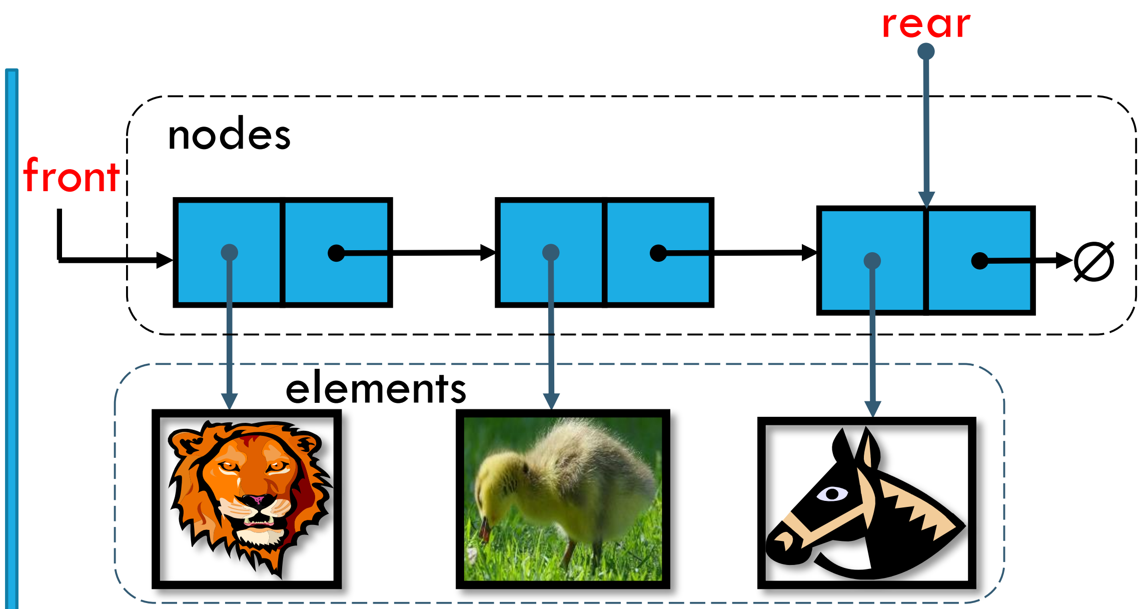


```cpp
#include <iostream>
using namespace std;
struct Node {
    string data;
    struct Node* next;
};
Node* removeLastNode(struct Node* head) {
    if (head == NULL)
        return NULL;
    if (head->next == NULL) {
        delete head;
        return NULL;
    }
    Node* second_last = head;
    while (second_last->next->next != NULL)
        second_last = second_last->next;
    delete (second_last->next);
    second_last->next = NULL;
    return head;
}
void insertNode (struct Node** head_ref, string new_data) {
    struct Node* new_node = new Node;
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}
int main() {
    Node* head = NULL;
    insertNode(&head, "Hyd");
    insertNode(&head, "Goa");
    insertNode(&head, "Dubai");
    insertNode(&head, "Pilani");
    head = removeLastNode(head);
    cout << "After deleting the last node:"<<endl;
    for (Node* temp = head; temp != NULL; temp = temp->next)
        cout << temp->data << " ";
    return 0;
}
```

```
After deleting the last node:
Pilani Dubai Goa
```

STACK & QUEUE AS SINGLY LINKED LISTS



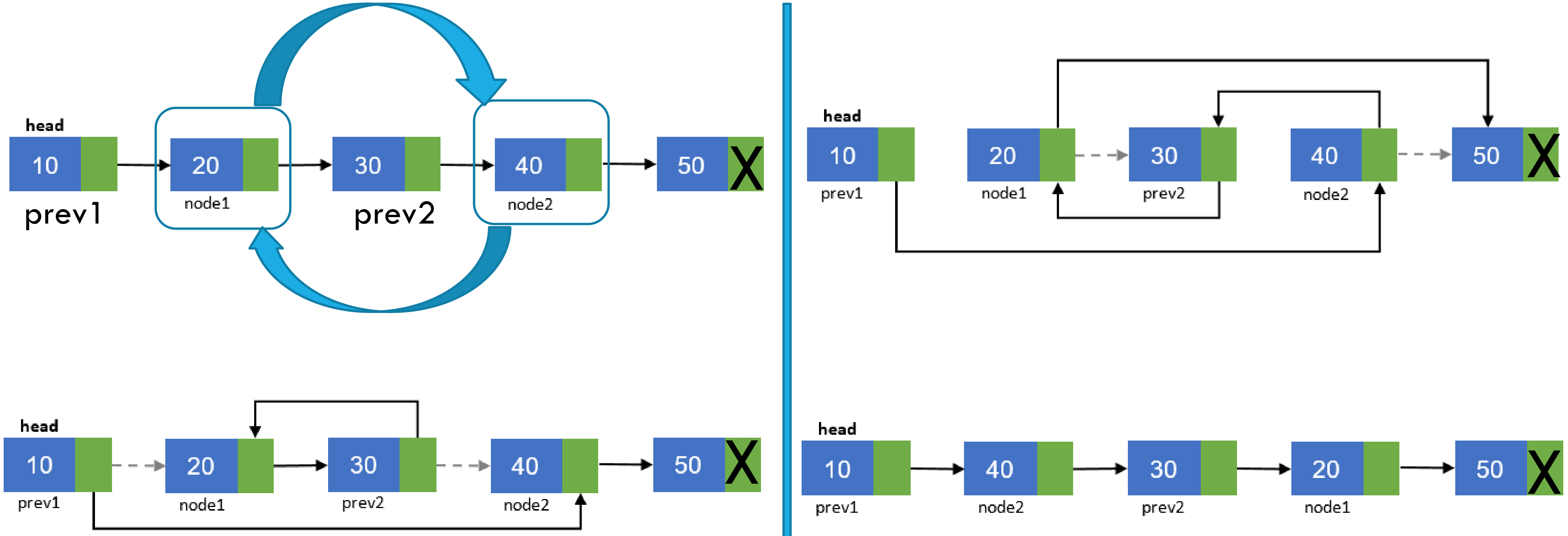


Stack: We can implement stack as linked list. Top element is stored as first element of the linked list.

Queue: We can implement a queue as a linked list. Front element is stored as first element of the linked list, and rear element is stored as the last element.

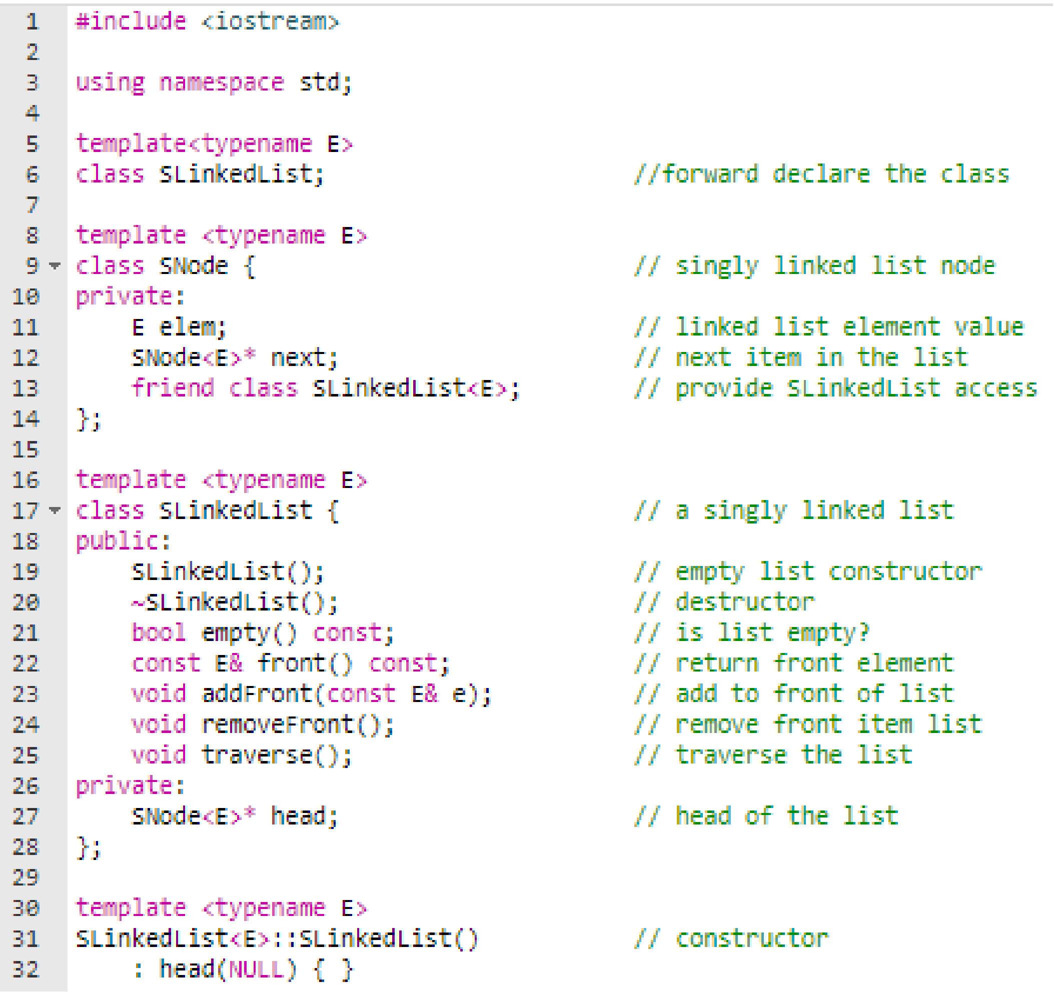
Implementation in later chapters…

SWAPPING TWO NODES IN A LINKED LIST

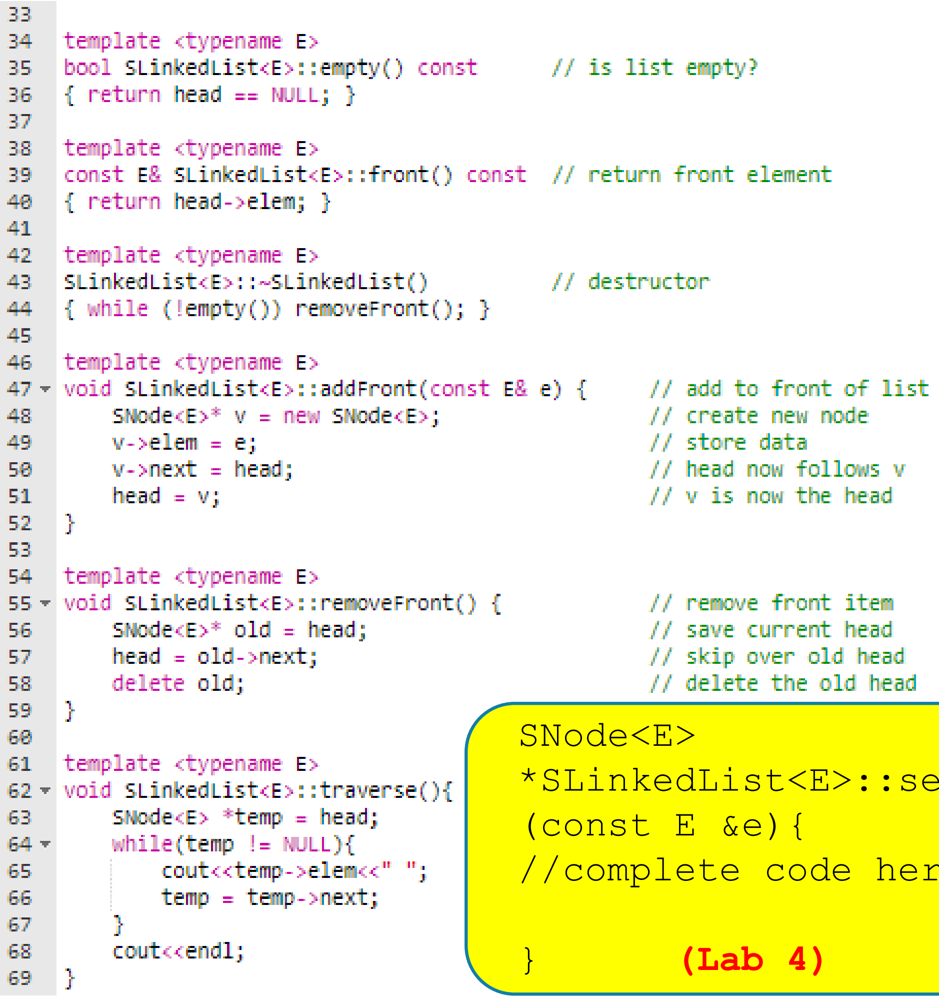


Lab 4 next week

GENERIC SINGLY LINKED LISTS: USING TEMPLATES



```cpp
#include <iostream>

using namespace std;

template<typename E>
class SLinkedList;                          //forward declare the class

template <typename E>
class SNode {                               // singly linked list node
private:
    E elem;                                 // linked list element value
    SNode<E>* next;                         // next item in the list
    friend class SLinkedList<E>;            // provide SLinkedList access
};

template <typename E>
class SLinkedList {                         // a singly linked list
public:
    SLinkedList();                          // empty list constructor
    ~SLinkedList();                         // destructor
    bool empty() const;                     // is list empty?
    const E& front() const;                 // return front element
    void addFront(const E& e);              // add to front of list
    void removeFront();                     // remove front item list
    void traverse();                        // traverse the list
private:
    SNode<E>* head;                         // head of the list
};

template <typename E>
SLinkedList<E>::SLinkedList()               // constructor
    : head(NULL) { }
```

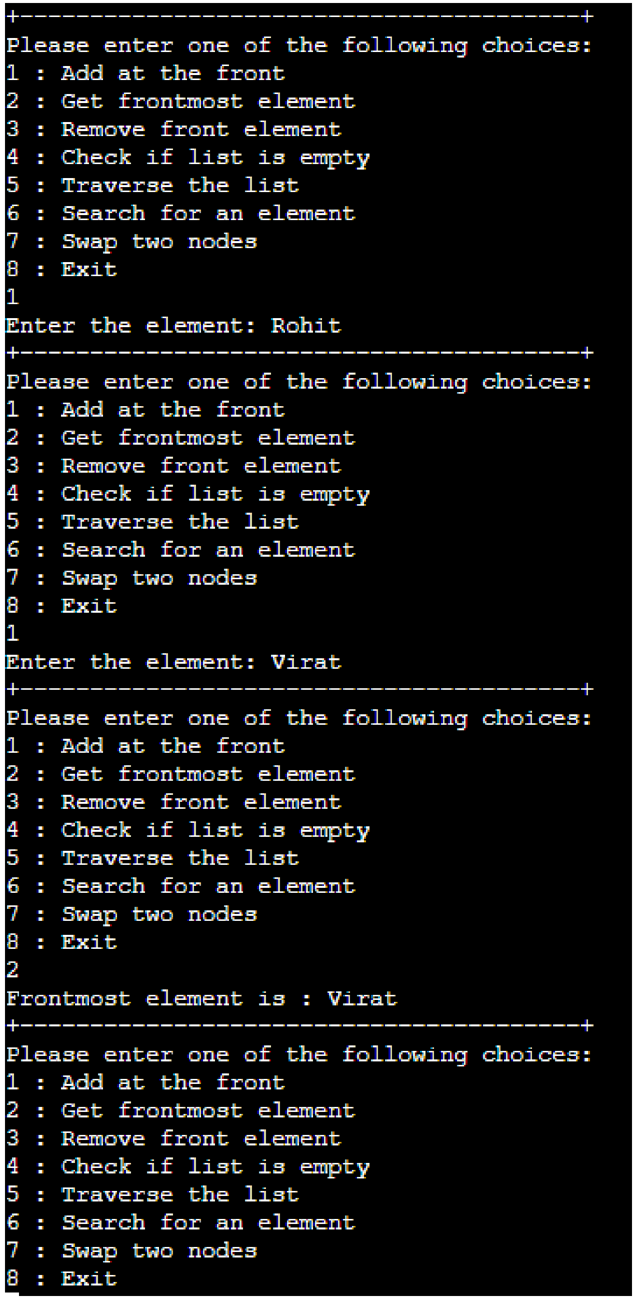


```cpp

template <typename E>
bool SLinkedList<E>::empty() const       // is list empty?
{ return head == NULL; }

template <typename E>
const E& SLinkedList<E>::front() const   // return front element
{ return head->elem; }

template <typename E>
SLinkedList<E>::~SLinkedList()           // destructor
{ while (!empty()) removeFront(); }

template <typename E>
void SLinkedList<E>::addFront(const E& e) {     // add to front of list
    SNode<E>* v = new SNode<E>;                 // create new node
    v->elem = e;                                // store data
    v->next = head;                             // head now follows v
    head = v;                                   // v is now the head
}

template <typename E>
void SLinkedList<E>::removeFront() {            // remove front item
    SNode<E>* old = head;                       // save current head
    head = old->next;                           // skip over old head
    delete old;                                 // delete the old head
}

template <typename E>
void SLinkedList<E>::traverse(){
    SNode<E> *temp = head;
    while(temp != NULL){
        cout<<temp->elem<<" ";
        temp = temp->next;
    }
    cout<<endl;
}
```

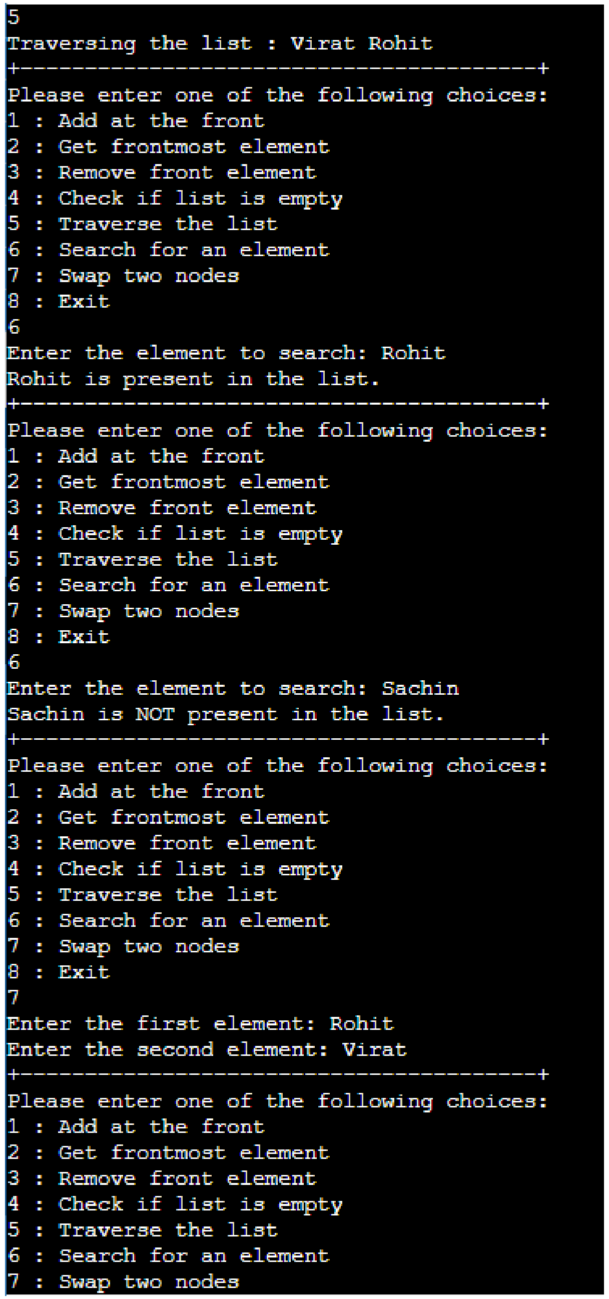
```
SNode<E>
*SLinkedList<E>::search
(const E &e){
//complete code here

}
```

(Lab 4)

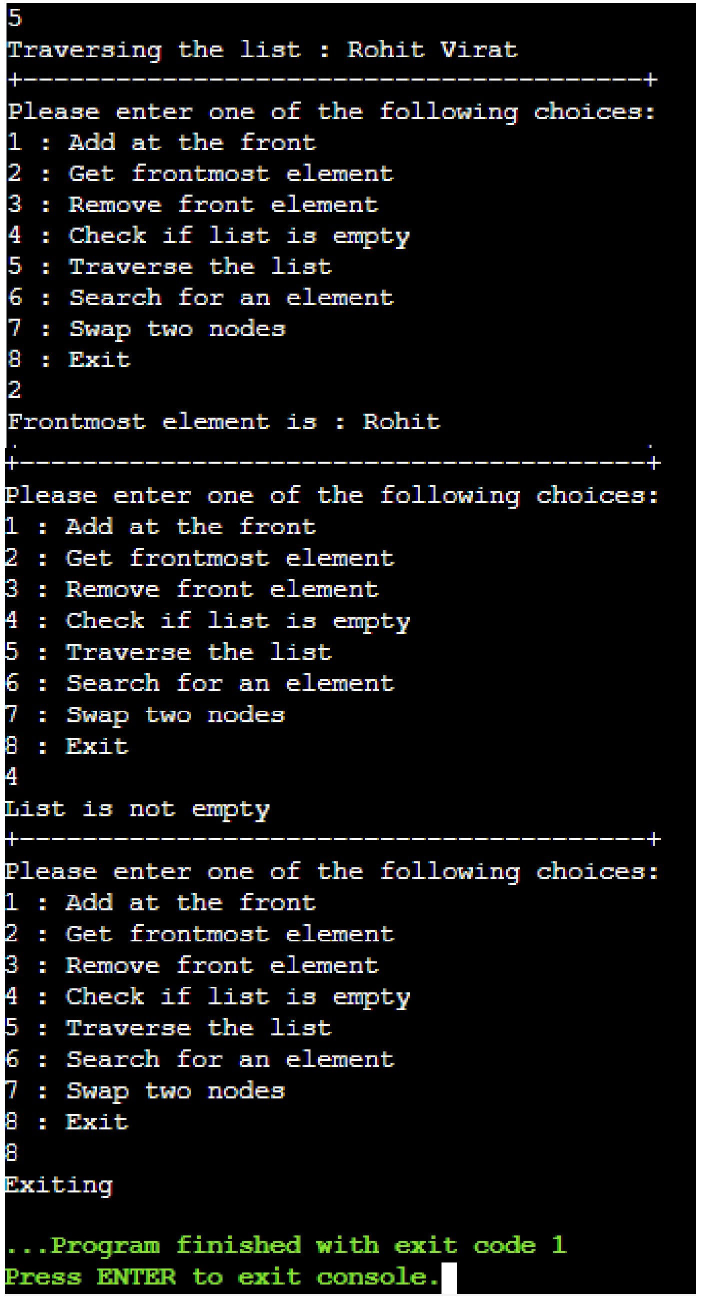
LAB 4



```
+----------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
1
Enter the element: Rohit
+----------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
1
Enter the element: Virat
+----------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
2
Frontmost element is : Virat
+----------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
```



```
5
Traversing the list : Virat Rohit
+----------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
6
Enter the element to search: Rohit
Rohit is present in the list.
+----------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
6
Enter the element to search: Sachin
Sachin is NOT present in the list.
+----------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
7
Enter the first element: Rohit
Enter the second element: Virat
+----------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
```



```
5
Traversing the list : Rohit Virat
+----------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
2
Frontmost element is : Rohit
+----------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
4
List is not empty
+----------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
8
Exiting

...Program finished with exit code 1
Press ENTER to exit console.
```

DOUBLY LINKED LIST

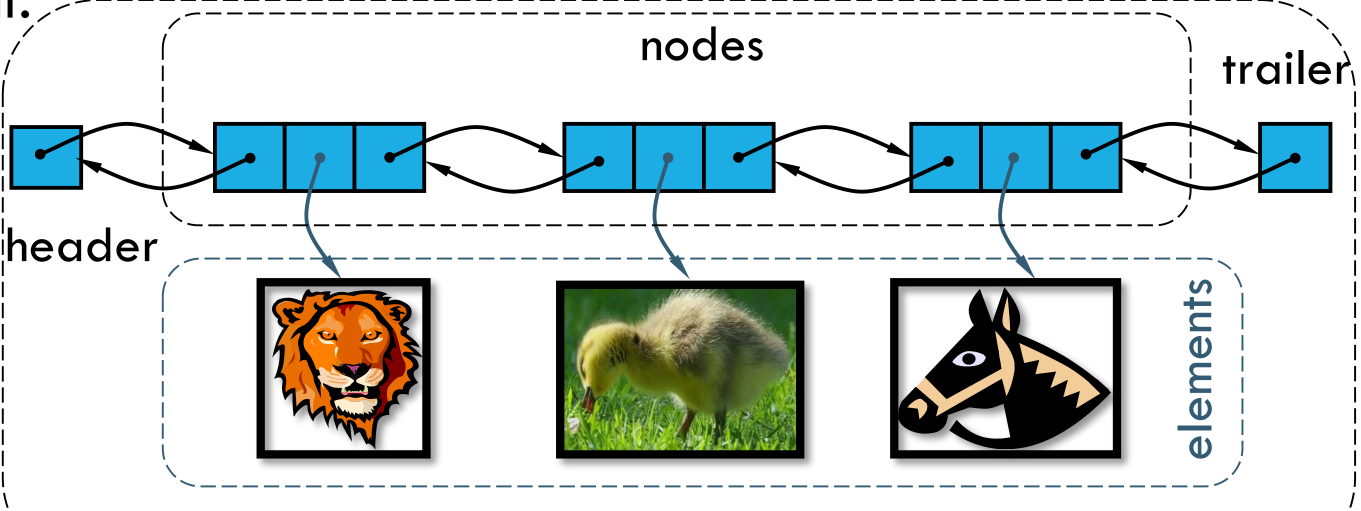
- Deleting the last node in a singly linked list is not efficient. Why? (rather any node other than first one or two)
- What is a doubly linked list?
- Insertions and deletions are more efficient.

Applications:

- Used by browsers for what functionality?
- Used to implement MRU, and LRU caches?
- Undo/ Redo functionality in Word.
- Used to implement hash tables, stacks, binary tree etc.

```
typedef string Elem;
class DNode {
  private: Elem elem;
              DNode* prev;
              DNode* next;
              friend class DLinkedList;
};
```

(Implementation of DLL Node)

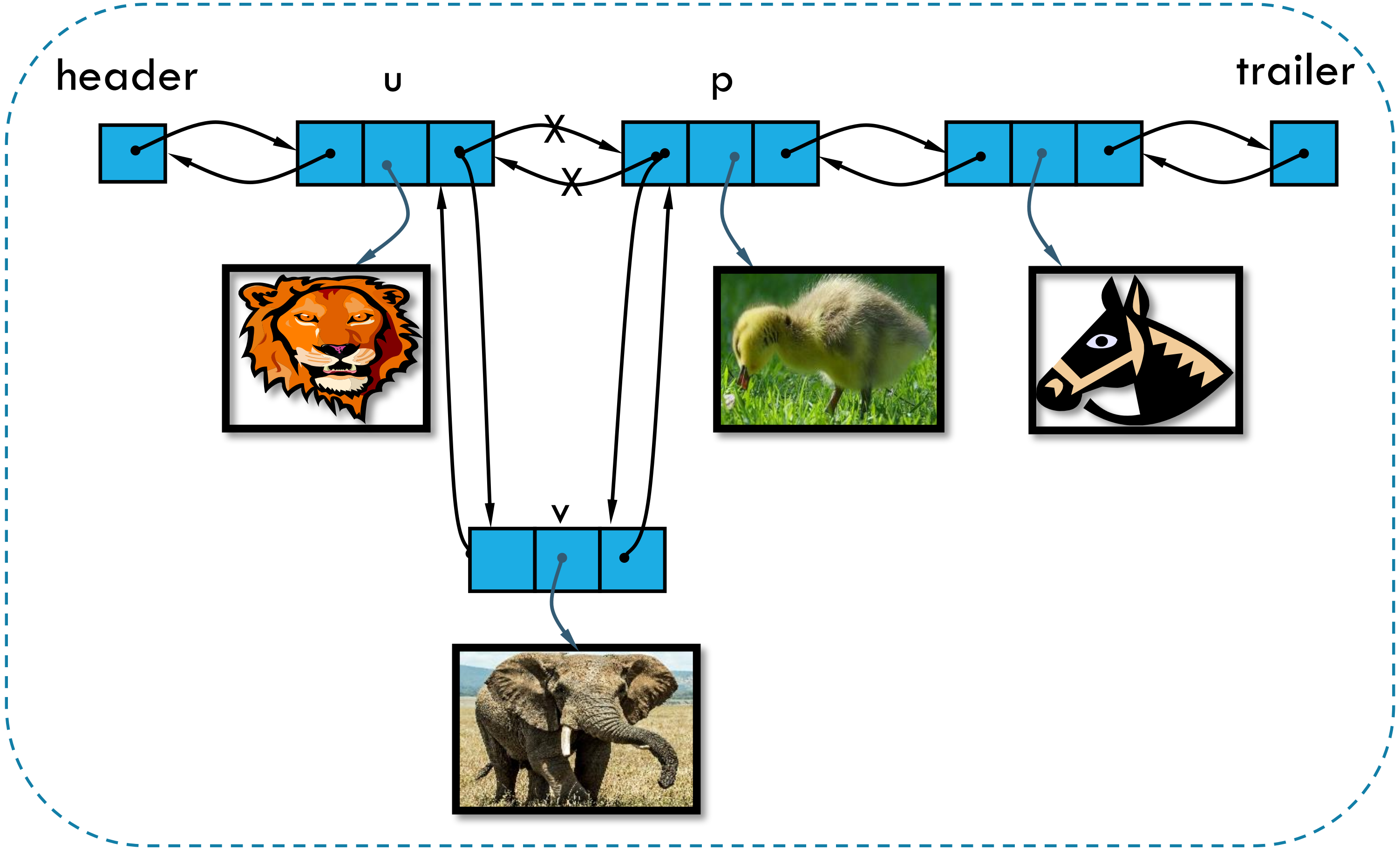


(A doubly linked list with two special nodes (sentinels)

INSERTING INTO DOUBLY-LINKED LIST

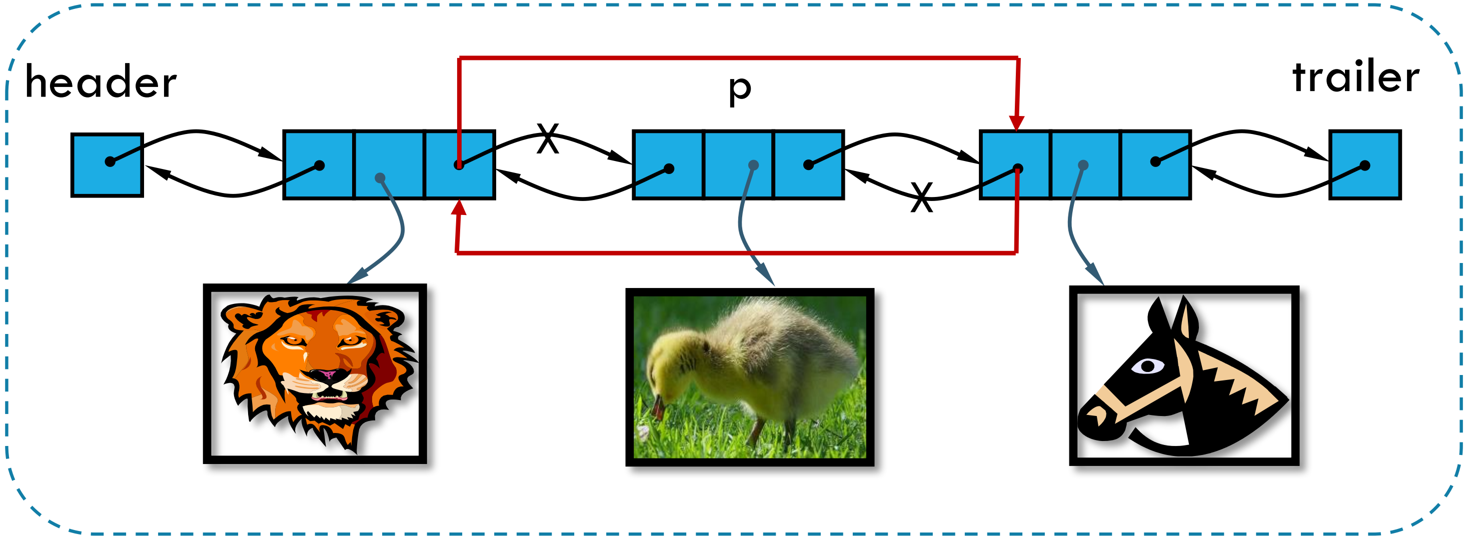
Algorithm insert(p, e): //insert e before p

Let us write the pseudo code in parallel…



REMOVING A NODE IN DOUBLY-LINKED LIST

```
Algorithm remove (p: position ) {
    if (p->previous != nil) // not first
        p->previous->next = ???;
    if (p->next != nil) //not the last
        p->next->previous  = ???;
}
```


header
p
trailer
X
X