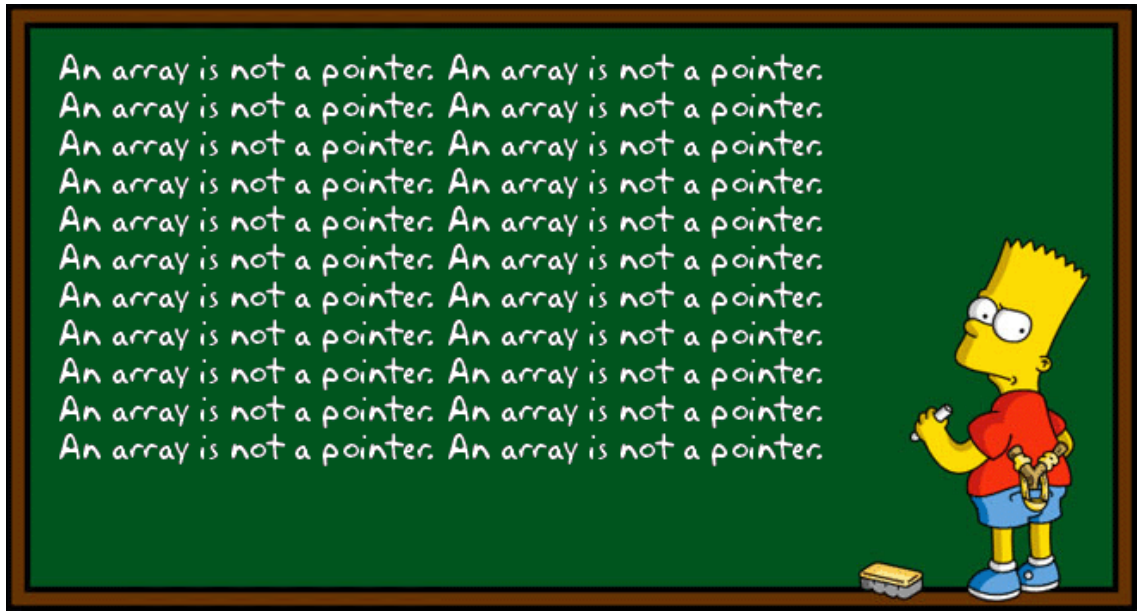
LINEAR DATA STRUCTURES: VECTORS AND LINKED LIST

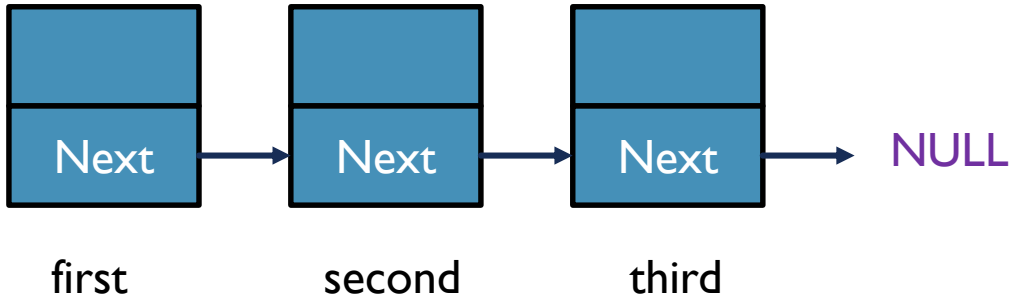
ALGORITHMS AND DATA STRUCTURES

LINEAR DATA STRUCTURES: VECTORS AND LINKED LISTS

Content

- Vectors (Dynamic Arrays)
  - Structure
  - Resizing and pushing
- Lists
  - Creation of structure
  - Append (attaching to the end of the list)
  - Push (attaching to the first element
  - Insert
    - Insert after
    - Insert before
  - Deleting the node
- Lists Algorithms
  - Loop Detection
  - Merge point detection





VECTORS

LINEAR STRUCTURES: VECTORS AND LINKED LIST

VECTORS

Structure

- Vectors represent a dynamic arrays
- Can be resized and reallocated

```
vector<double> age(4);

age[0] = 32.0;
age[1] = 43.2;
age[2] = 54.5;
age[3] = 64.1;
```

```
class vector{

    int sz;
    double* elem;

public:

    vector(int s):sz(s), // init sz
        elem(new double[s]) // init elem
    {
        for (int i = 0; i < s; i++)
            elem[i] = 0;

    }
    int size()const{return sz;} // current size

};
```

VECTOR IMPLEMENTATION

```
template<class T>
class Vector
{
public:
   typedef T * iterator;
   Vector();
   Vector(unsigned int size);
   Vector(unsigned int size, const T & initial);
   Vector(const Vector<T> & v);           // copy constructor
   ~Vector();
   unsigned int capacity() const;         // return capacity of vector (in elements)
   unsigned int size() const;             // return the number of elements in the vector
   bool empty() const;
   iterator begin();                      // return an iterator pointing to the first element
   iterator end();                        // return an iterator pointing to one past the last element
   T & front();                           // return a reference to the first element
   T & back();                            // return a reference to the last element
   void push_back(const T & value);       // add a new element
   void pop_back();                       // remove the last element
   void reserve(unsigned int capacity);   // adjust capacity
   void resize(unsigned int size);        // adjust size
   T & operator[](unsigned int index);    // return reference to numbered element
   Vector<T> & operator=(const Vector<T> &);
private:
   unsigned int my_size;
   unsigned int my_capacity;
   T * buffer;
};
```

VECTOR IMPLEMENTATION

```
template<class T>//
Vector<T>::Vector()
{
    my_capacity = 0;
    my_size = 0;
    buffer = 0;
}
template<class T>
Vector<T>::Vector(const Vector<T> & v)
{
    my_size = v.my_size;
    my_capacity = v.my_capacity;
    buffer = new T[my_size];
    for (int i = 0; i < my_size; i++)
        buffer[i] = v.buffer[i];
}
template<class T>//
Vector<T>::Vector(unsigned int size)
{
    my_capacity = size;
    my_size = size;
    buffer = new T[size];
}
```

```
template<class T>//
Vector<T>::Vector(unsigned int size, const T & initial)
{
    my_size;
    my_capacity = size;
    buffer = new T [size];
    for (int i = 0; i < size; i++)
        buffer[i] = initial;
}
template<class T>//
Vector<T> & Vector<T>::operator = (const Vector<T> & v)
{
    delete[ ] buffer;
    my_size = v.my_size;
    my_capacity = v.my_capacity;
    buffer = new T [my_size];
    for (int i = 0; i < my_size; i++)
        buffer[i] = v.buffer[i];
    return *this;
}
template<class T>//
typename Vector<T>::iterator Vector<T>::begin()
{
    return buffer;
}
template<class T>//
typename Vector<T>::iterator Vector<T>::end()
{
    return buffer + size();
}
```

VECTOR IMPLEMENTATION

```
template<class T>//
T& Vector<T>::Vector<T>::front()
{
    return buffer[0];
}
template<class T>//
T& Vector<T>::Vector<T>::back()
{
    return buffer[size - 1];
}
template<class T>
void Vector<T>::push_back(const T & v)
{
    if (my_size >= my_capacity)
         reserve(my_capacity +5);
    buffer [my_size++] = v;
}
template<class T>//
void Vector<T>::pop_back()
{
    my_size--;
}
```

```
template<class T>//
void Vector<T>::reserve(unsigned int capacity)
{
    if(buffer == 0)
    {
        my_size = 0;
        my_capacity = 0;
    }
    if (capacity <= my_capacity)
return;
    T * new_buffer = new T [capacity];
    assert(new_buffer);
    copy (buffer, buffer + my_size, new_buffer);
    my_capacity = capacity;
    delete[] buffer;
    buffer = new_buffer;

}
template<class T>//
unsigned int Vector<T>::size()const
{
    return my_size;
}
template<class T>//
void Vector<T>::resize(unsigned int size)
{
    reserve(size);
    my_size = size;
}
```

VECTOR IMPLEMENTATION

```
template<class T>//
T& Vector<T>::operator[](unsigned int index)
{
    return buffer[index];
}

template<class T>//
unsigned int Vector<T>::capacity()const
{
    return my_capacity;
}

template<class T>//
Vector<T>::~Vector()
{
    delete[]buffer;
}
```

LINKED LIST

LINEAR STRUCTURES: VECTORS AND LINKED LIST

LINKED LIST

```
struct Node
{
  int data;
  Node *next;
};
```

Integer data holder

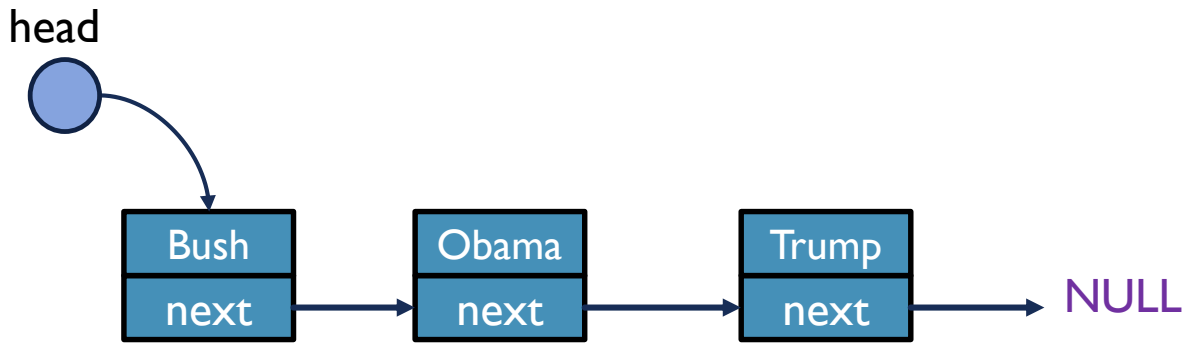
```
template<typename T>
struct Node
{
  T data;
  Node *next;
};
```

General case

Building Block

Additional pointer

- If a node contains a data member that is a pointer to another node, then many nodes can be strung together using only one variable to access the entire sequence of nodes. Such a sequence of nodes is the most frequently used implementation of a linked list
- If a node has a link only to its successor in this sequence, the list is called a singly linked list
- Data could be any type

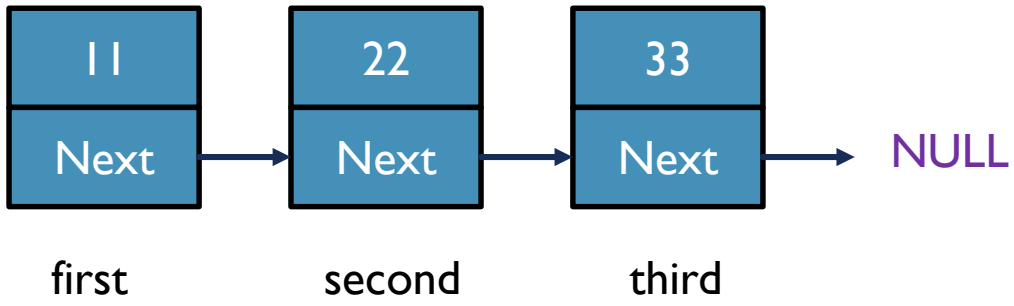


Linked List

NODE AS BUILDING BLOCKS

```
struct Node
{
  int data;
  Node *next;
};

void printList(Node *n)
{
  while (n != NULL)
  {
     cout<<n->data<<" ";
     n = n->next;
  }
}
```

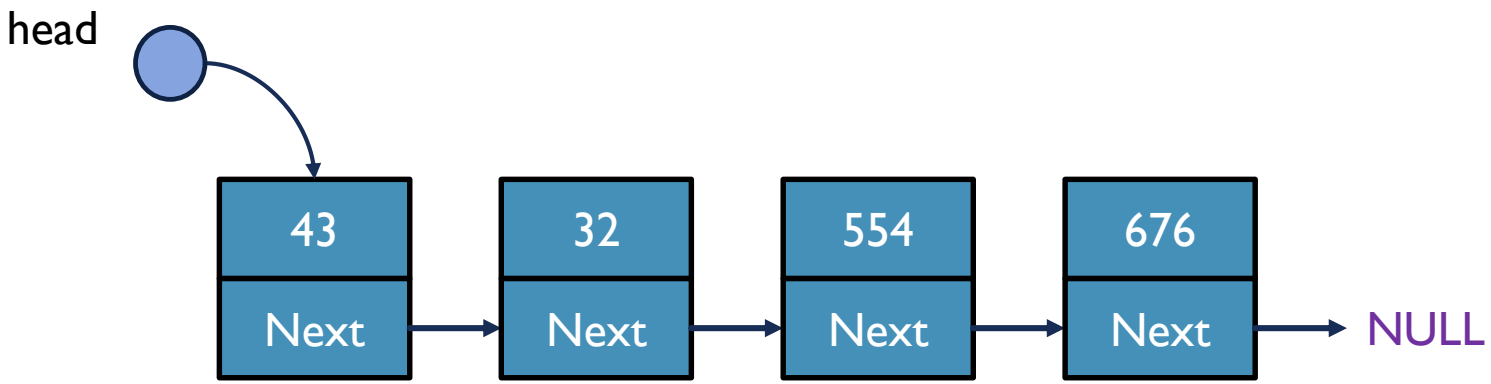
```
int main(){

  Node* first = new Node();
  Node* second = new Node();
  Node* third = new Node();

  first->data = 11;
  first->next = second;

  second->data = 22;
  second->next = third;

  third->data = 33;
  third->next = NULL;

  Node* head = first;
  printList(head);
  system("pause");
}
```



LINKED LIST APPENDING

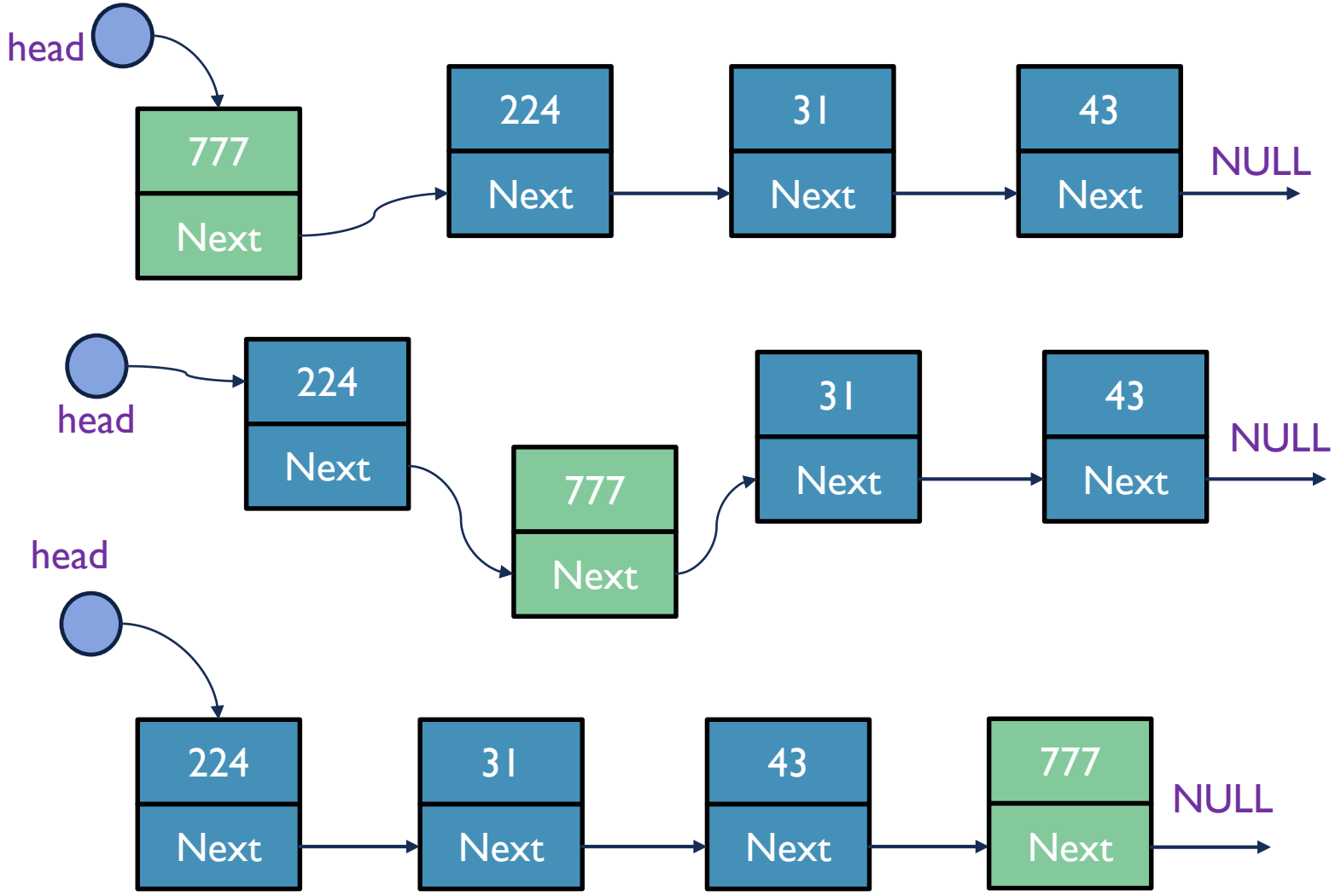
```
void append(Node** head, int added_data)
{

    Node* new_node = new Node();
    new_node->data  = added_data;
    new_node->next = NULL;

if (*head == NULL){
       *head = new_node;
       return;
    }

    Node *tail = *head; // initial

    while (tail->next != NULL)
        tail = tail->next;

    tail->next = new_node;
    return;
}
```

```
int main(){
  Node* h = NULL;
  append(&h, 43);
  append(&h, 32);
  append(&h, 554);
  append(&h, 676);

  printList(h);
  system("pause");
}
```

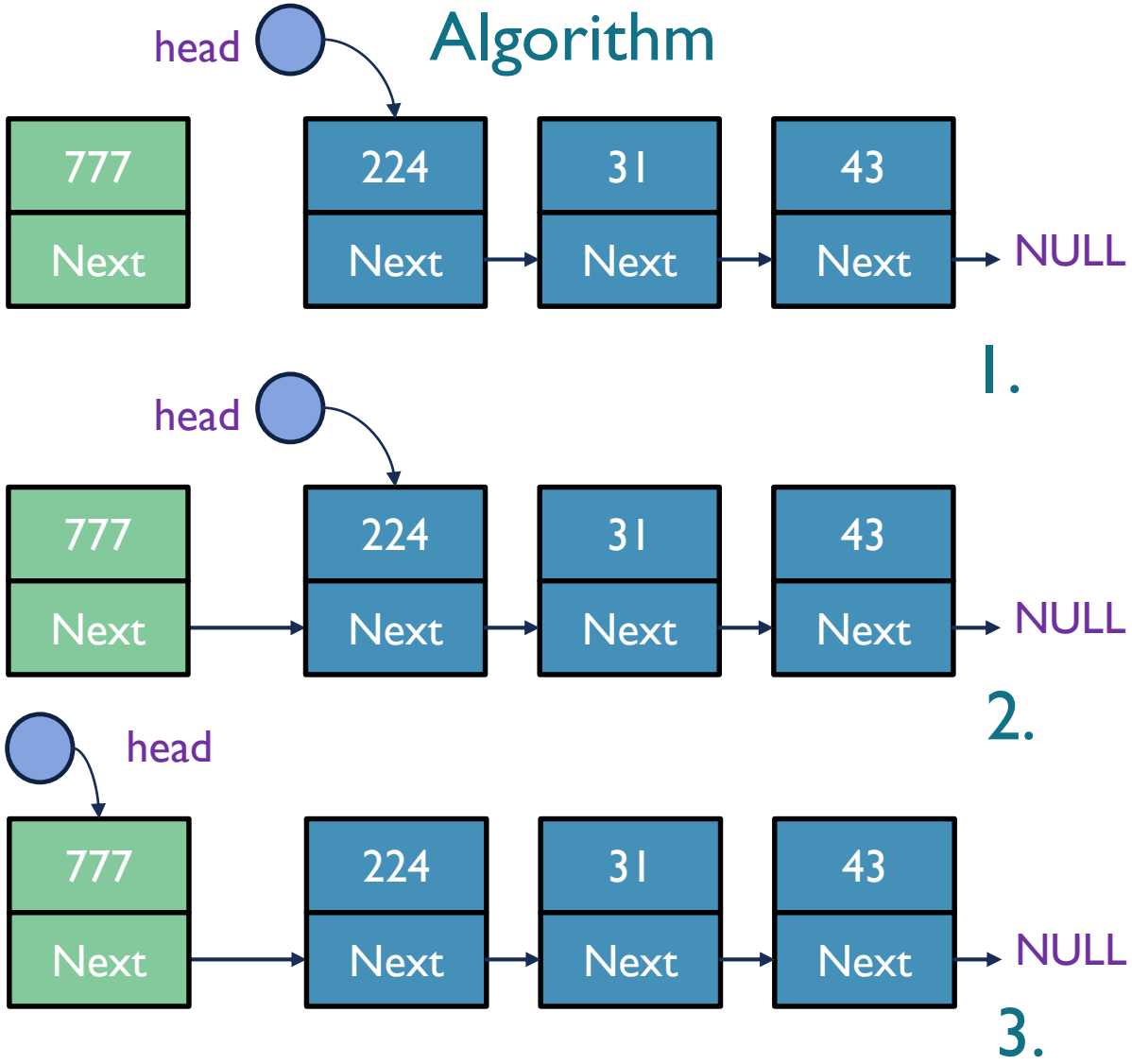


INSERTING A NODE

- inserting to the front
  - push( )
- inserting after given node
  - insert_after( )
- Inserting at the end of the list
  - append( )



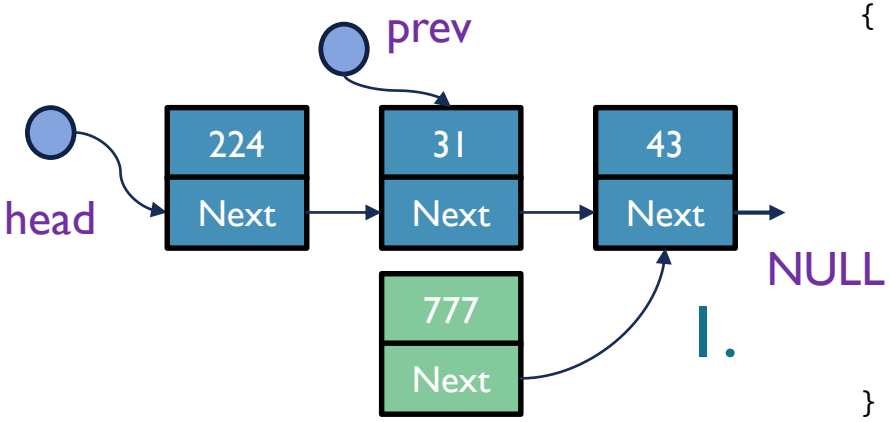
INSERTING TO THE FRONT PUSH METHOD

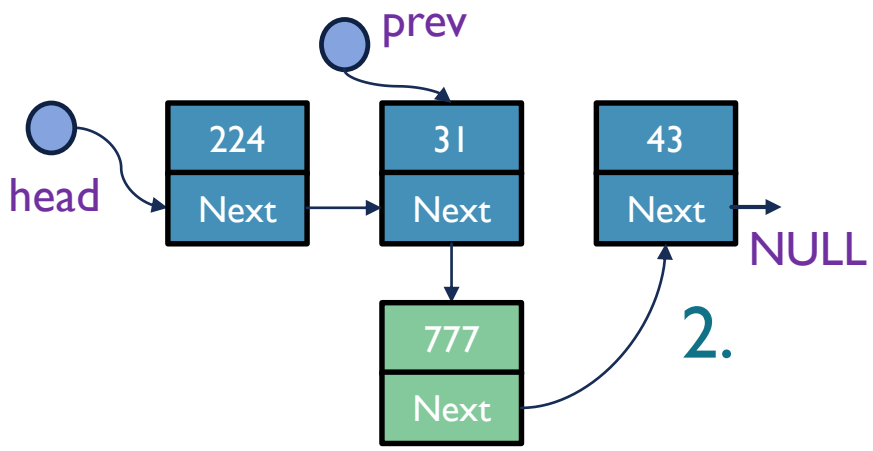



```
void push(Node** head, int new_data)
{

    Node* new_node = new Node;

    new_node->data  = new_data;
    new_node->next = *head;

    *head  = new_node;
}
```

INSERT AFTER METHODS

Algorithm






```
void insertAfter(Node* prev_node, int new_data)
{
    if (prev_node == NULL)
    {
       return;
    }

    Node* new_node = new Node();
    new_node->data  = new_data;

    new_node->next = prev_node->next;
    prev_node->next = new_node;
}
```

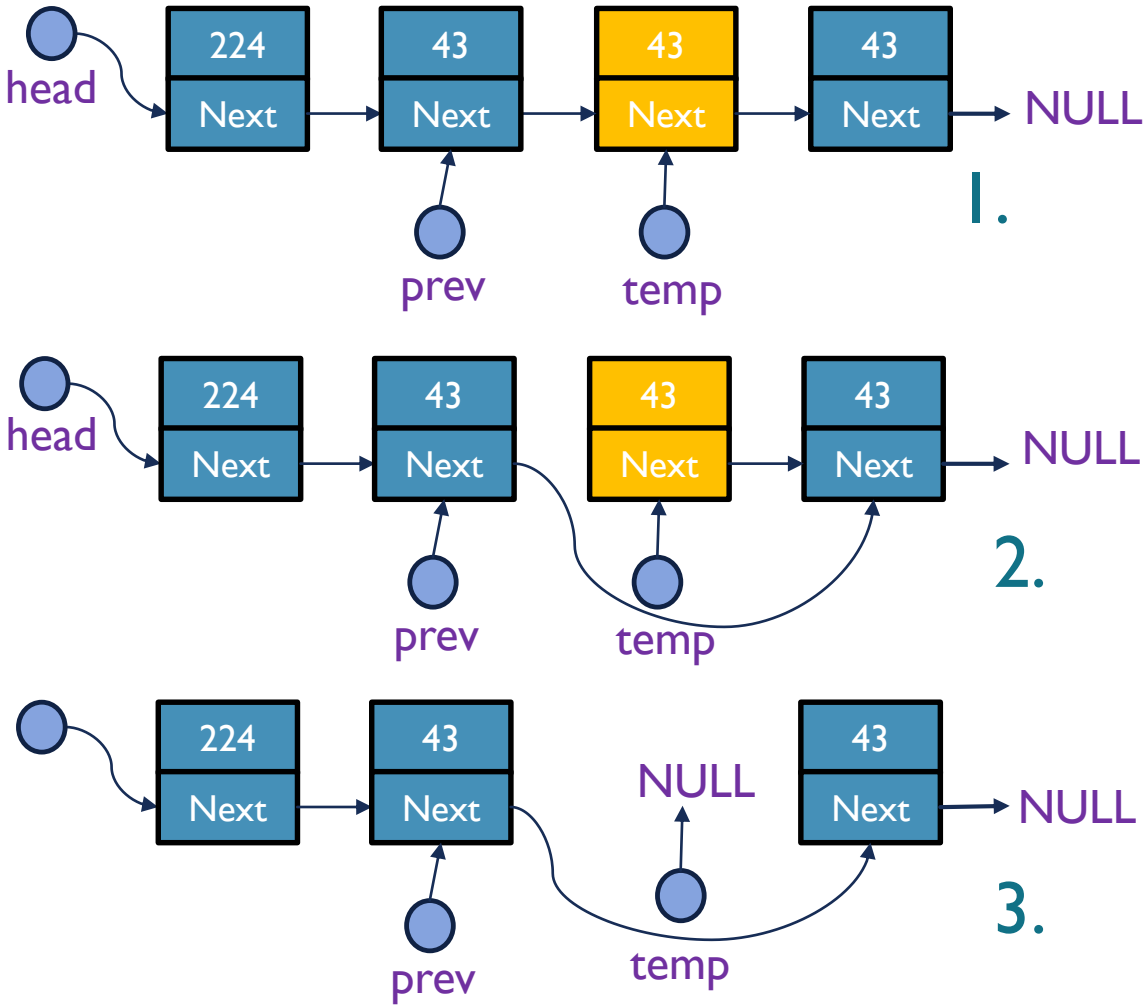
After prev pointer

```
void insert_after_pos(Node* head, int pos, int new_data)
{
    if (head == NULL)
    {
       return;
    }

    Node* new_node = new Node();
    new_node->data  = new_data;

Node* pos_before = head;
while (pos > 1)
{
pos_before = pos_before->next;
pos--;
}

    new_node->next = pos_before->next;
    pos_before->next = new_node;
}
```

After counter

DELETING THE NODE

Algorithm

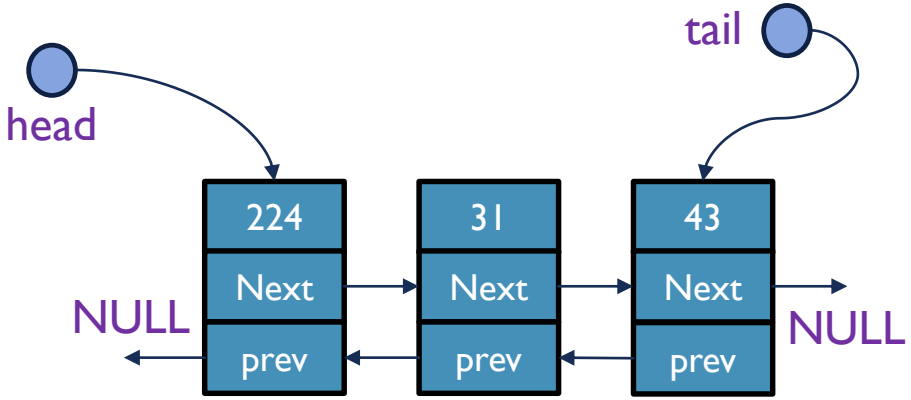



```
void deleteNode(Node **head_ref, int key)
{
    Node *temp = *head_ref, *prev;
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next;
        delete temp;
        return;
    }
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL)
          return;
    prev->next = temp->next;
    delete temp;
}
```

ALTERNATIVE LINKED LISTS

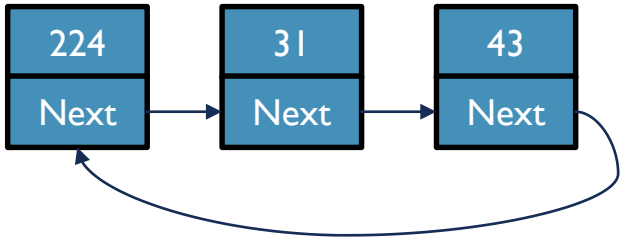
Doubly Linked List

- Cyclic linked lists
- Bidirectional Linked list (Doubly linked list)
- Linked list with additional pointers




```
template<typename T>
struct Node
{
  T data;
  Node *next;
  Node * prev;
};
```

Additional pointer



Must keep track on number

```
template<typename T>
struct Node
{
  T data;
  Node *next;
  unsigned int elements;
};
```

Must keep track on elements

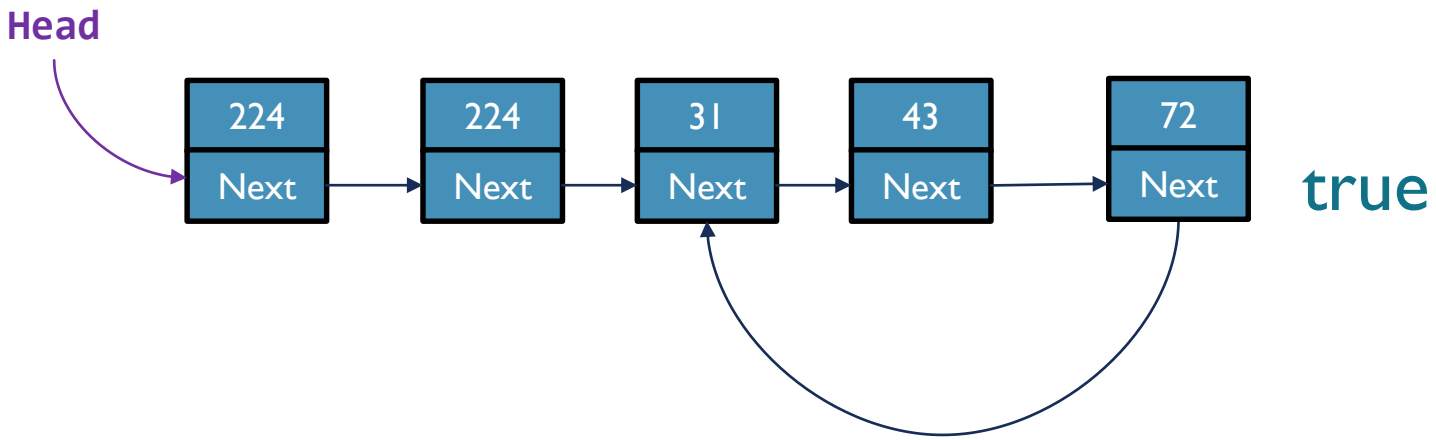
LISTS ALGORITHMS

LINEAR STRUCTURES: VECTORS AND LINKED LIST

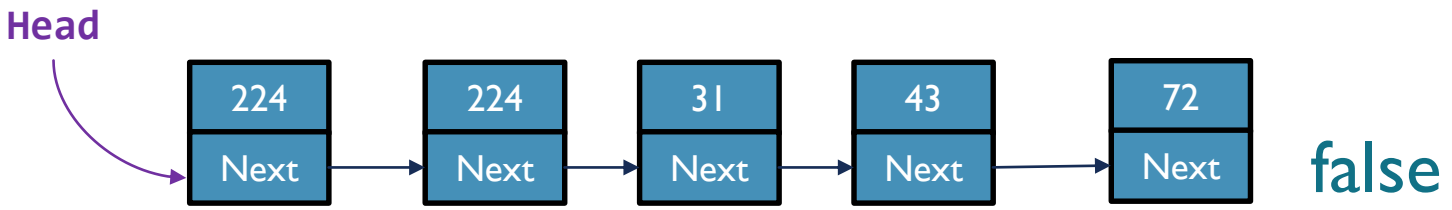
LOOP DETECTION

Task

- Given linked List (head pointer)
- Detect the exiting of the loop



true

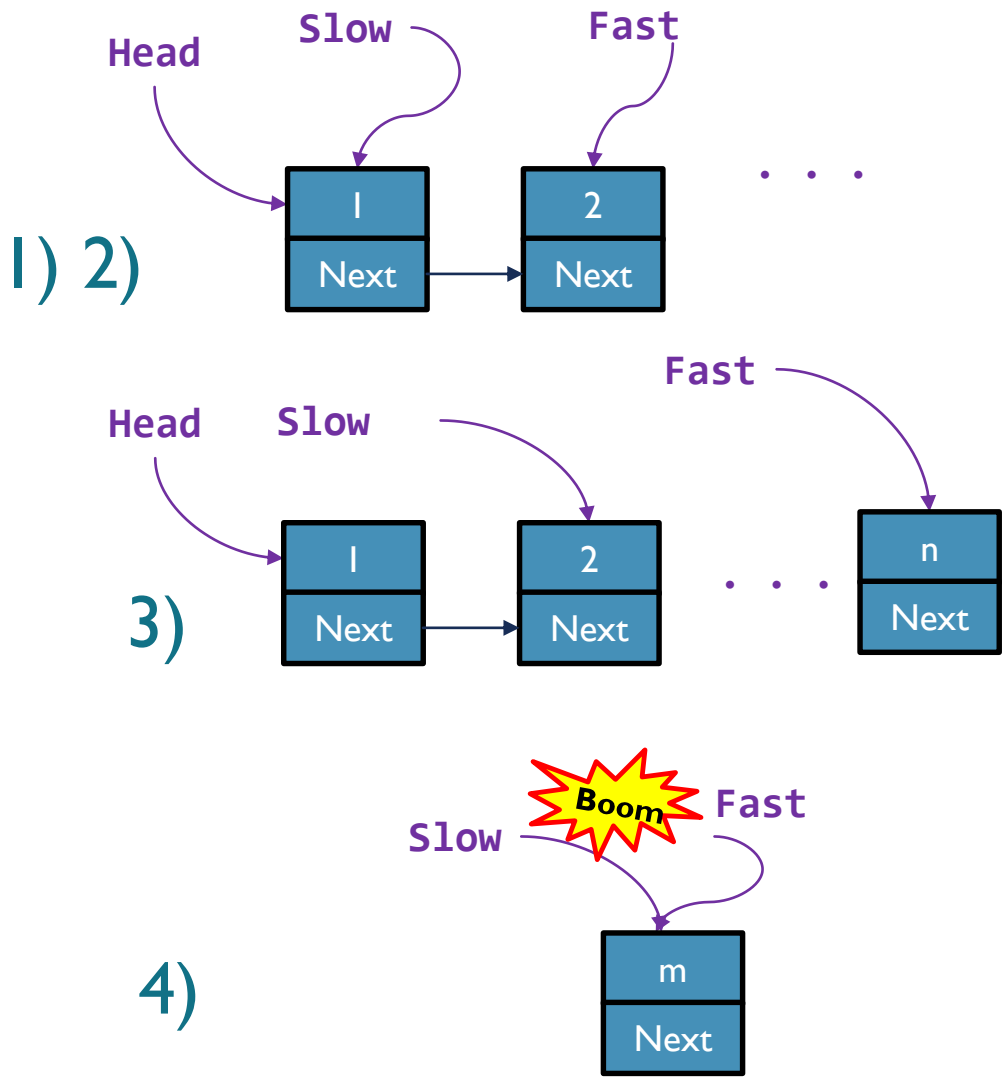


false

LOOP DETECTION

Algorithm

- Create 2 pointers. (1,2)
  - Slow Pointer,
  - Fast Pointer
- Start travers through the list (with different speed) (3)
  - Slow = Slow -> next
  - Fast = Fast ->next -> next
- If Loop exist: (4)
  - Fast and Slow pointers will eventually meet each other



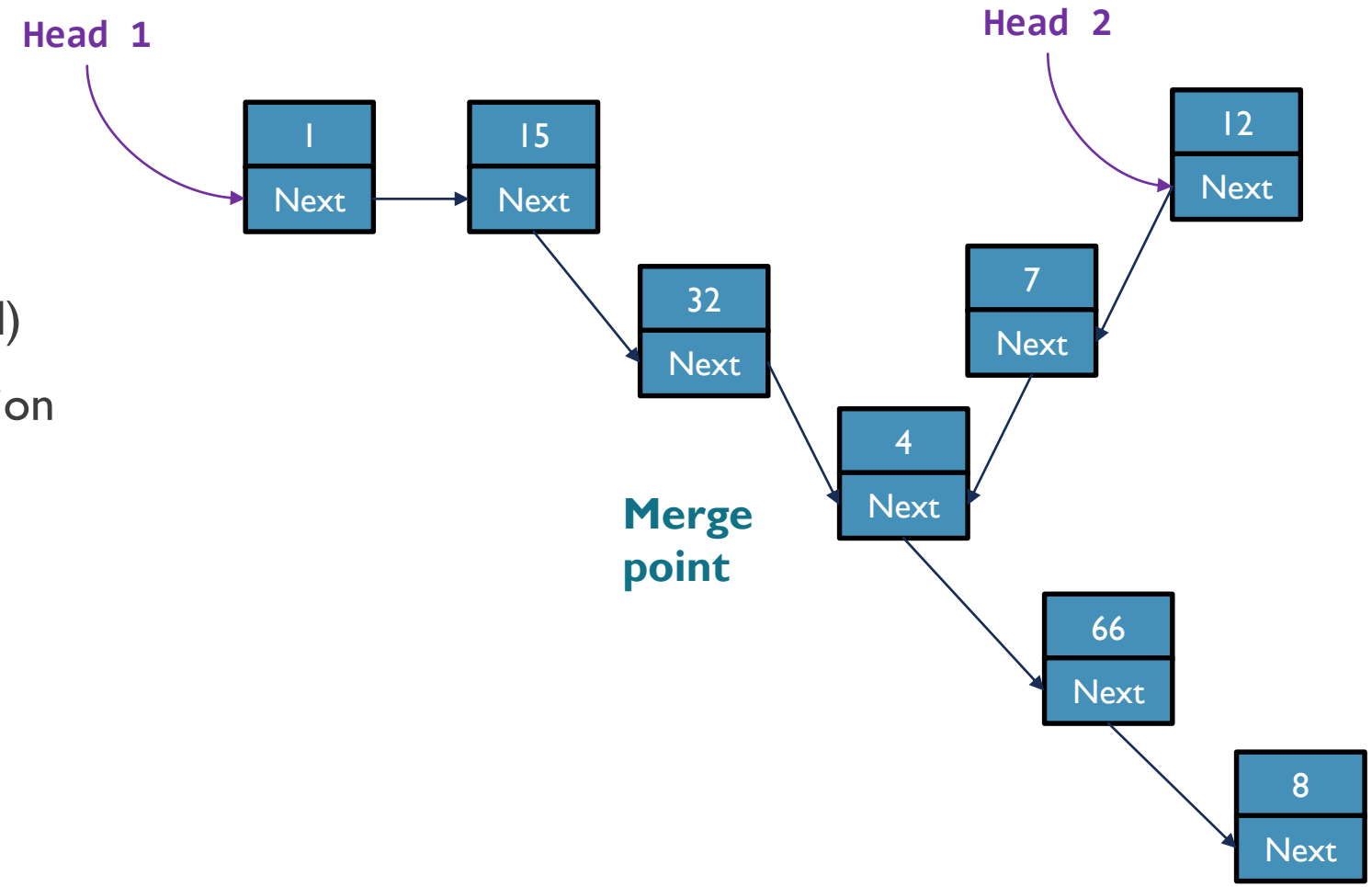
LOOP DETECTION

```
int find_loop(Node *list)
{
    Node  *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next )
    {
        slow_p = slow_p->next;
        fast_p  = fast_p->next->next;
        if (slow_p == fast_p)
        {
           cout<<"Loop Detected"<<endl;
           return true;
        }
    }
    return false;
}
```

MERGE POINT DETECTION

Task

- Given two Linked Lists (the head)
- Find the merge point (aka Insertion point) of two linked lists



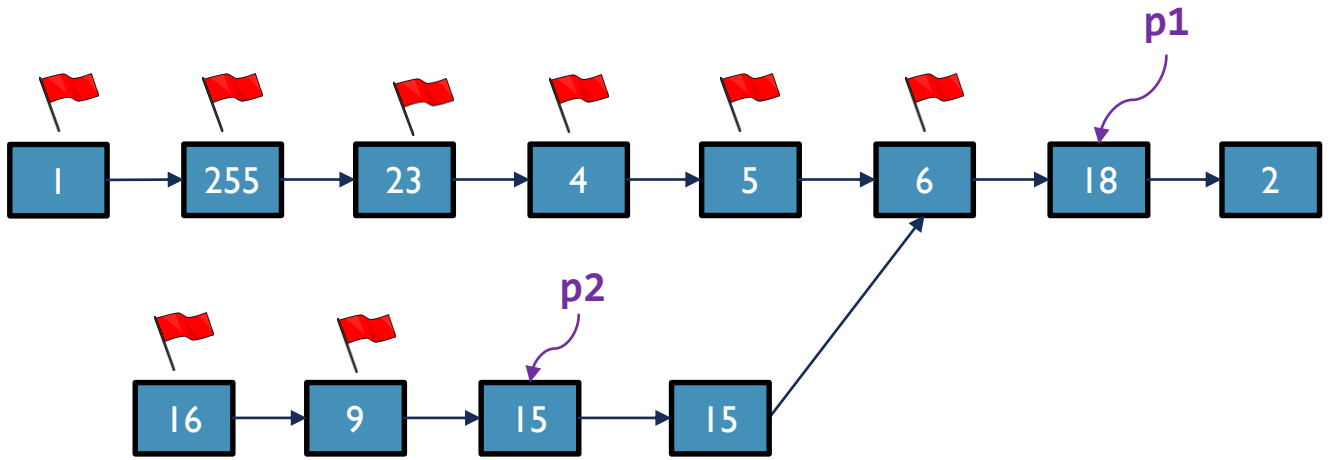
MERGE POINT DETECTION

Algorithm 1

- Brute Force Approach $O(n^2)$
  - Create two nested loops
  - Check if the node in the inner node is the same as the node in the outer loop

Algorithm 2

- Additional information in the node $O(n+m)$
  - Modify the current Linked List structure by adding a flag.
  - Traverse both heads through the loop and change the state of the flag inside the nodes.
  - If one of heads detects that flag inside the traversed node has been changed – this node is an insertions node.



MERGE POINT DETECTION

Algorithm 3

- Couting the node $O(n+m)$
  - Create the first and the second pointer from head of both nodes
  - Count the number of nodes while traversing in both lists
  - Check the absolute difference between both list number **num = *abs(Count1 – Count2)***
  - traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes
  - traverse both the lists in parallel till we come across a common node. (Note that getting a common node is done by comparing the address of the nodes)

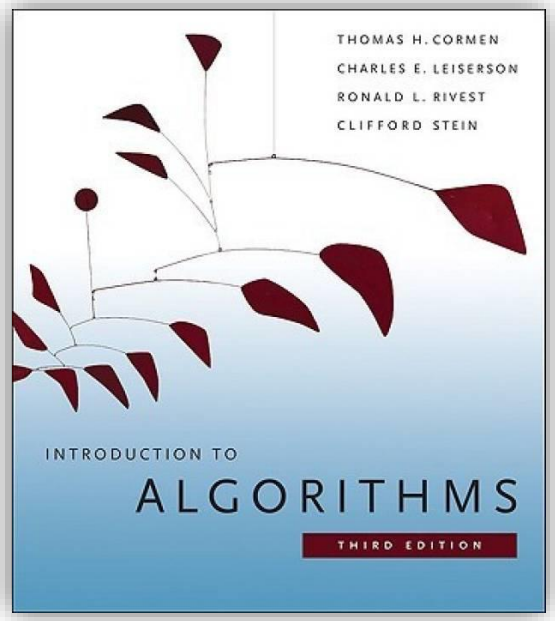
MERGE POINT DETECTION

```
int count(struct Node* head)
{
  struct Node* current = head;
  int count = 0;

  while (current != NULL)
  {
    count++;
    current = current->next;
  }

  return count;
}
```

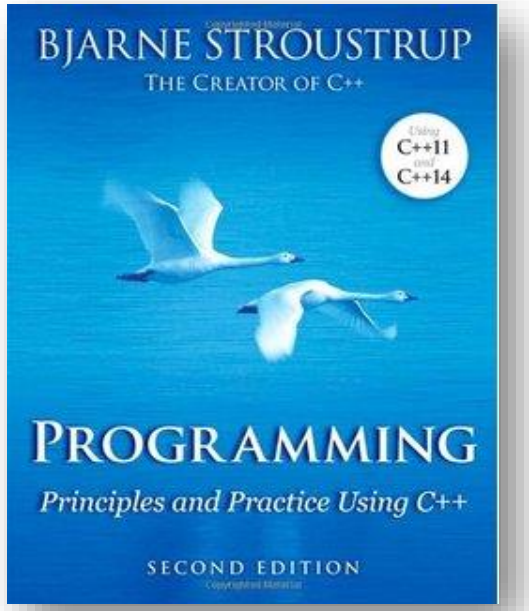
```
int merge_point_util(int d, Node* head1, Node* head2)
{
  int i;
  Node* current1 = head1;
  Node* current2 = head2;

  for(i = 0; i < d; i++)
  {
    if(current1 == NULL)
    {  return -1; }
    current1 = current1->next;
  }

  while(current1 !=  NULL && current2 != NULL)
  {
    if(current1 == current2)
      return current1->data;
    current1= current1->next;
    current2= current2->next;
  }

  return -1;
}
```

```
int Merge_point( Node* head1, Node* head2)
{
  int c1 = count(head1);
  int c2 = count(head2);
  int d;

  if(c1 > c2)
  {
    d = c1 - c2;
    return merge_point_util(d, head1, head2);
  }
  else
  {
    d = c2 - c1;
    return merge_point_util(d, head2, head1);
  }
}
```

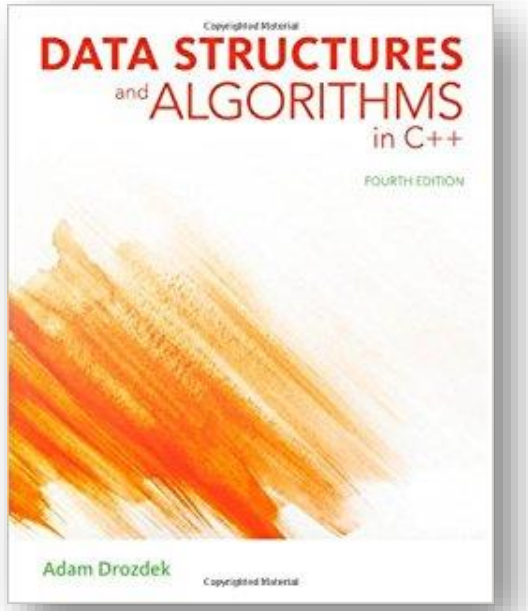
LITERATURE



Thomas H. Cormen
Introduction to Algorithms
Chapter III: Data structures
Page 236 (Linked Lists)



Bjarne Stroustrup
Principles and practice using C++
Chapter 17: vectors and free store
Page 569.



Adam Drozdek
Data structures and Algorithms in C++
Chapter 3 Linked List
Page 75