

# C++ Object-Oriented Prog.

## Unit 4: Objects and Lists

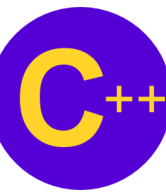
CHAPTER 16: LINKED STRUCTURES

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

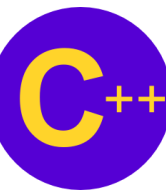
# Overview



# Chapter 16 Topics

---

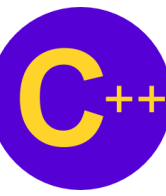
- Meaning of a Linked List
- Meaning of a Dynamic Linked List
- Traversal, Insertion and Deletion of Elements in a Dynamic Linked List
- Specification of a Dynamic Linked Sorted List
- Insertion and Deletion of Elements in a Dynamic Linked Sorted List



# What is a List?

---

- A **list** is a varying-length, linear collection of homogeneous elements
- **Linear** means that each list element (except the first) has a unique predecessor and each element (except the last) has a unique successor

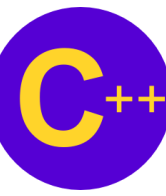


# To implement the List ADT

---

The programmer must

- 1) choose a concrete data representation for the list, and
- 2) implement the list operations



Recall:

## 4 Basic Kinds of ADT Operations

---

**Constructors** -- create a new instance (object) of an ADT

**Transformers** -- change the state of one or more of the data values **of an instance**

**Observers** -- allow client to observe the state of one or more of the data values of an instance without changing them

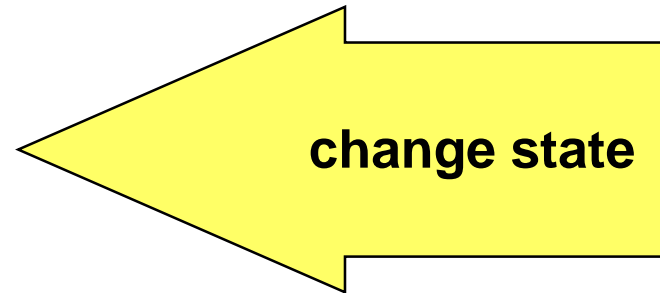
**Iterators** -- allow client to access the data values in sequence

# List Operations

---

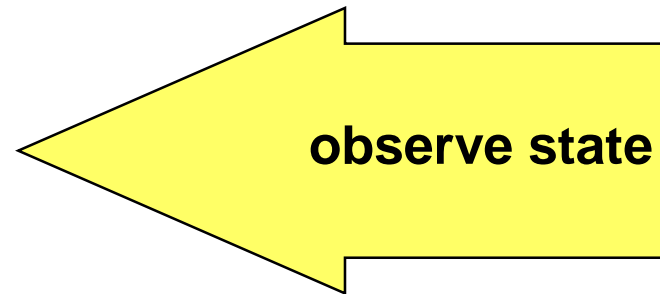
## Transformers

- Insert
- Delete
- Sort



## Observers

- IsEmpty
- IsFull
- Length
- IsPresent

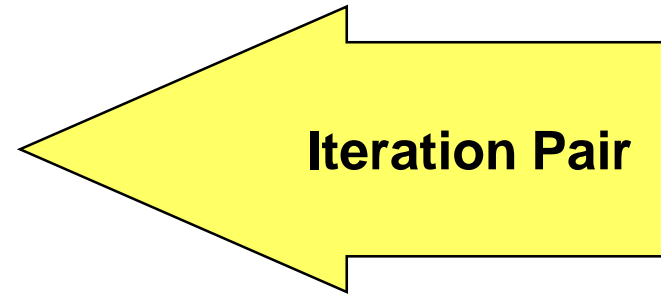


# ADT List Operations

---

## Iterator

- Reset
- GetNextItem



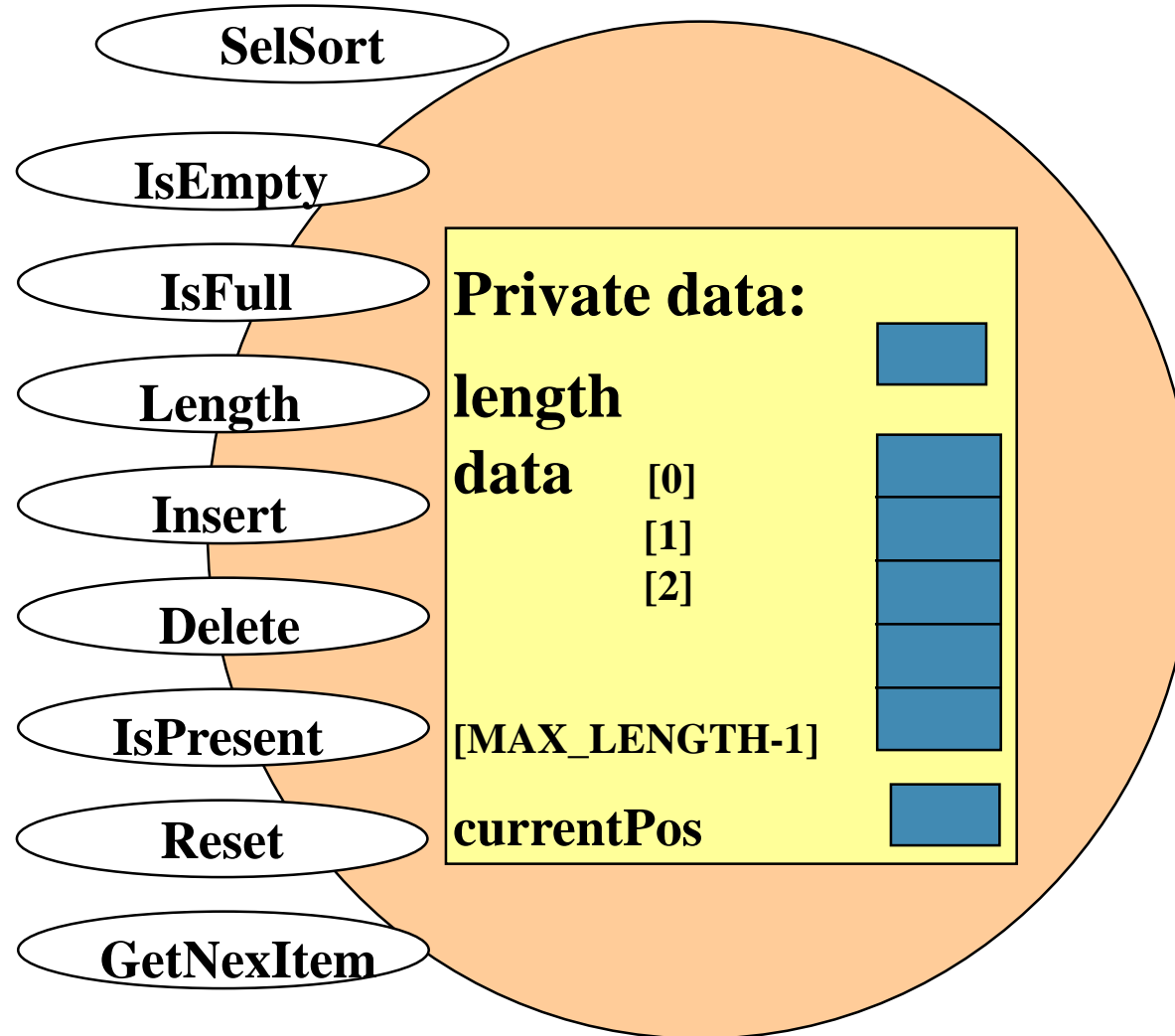
Reset prepares for the iteration

GetNextItem returns the next item in sequence

No transformer can be called between calls to GetNextItem (*Why?*)



# Array-based class List



```

// Specification file array-based list ("list.h")
const int MAX_LENGTH = 50;
typedef int ItemType;

class List // Declares a class data type
{
public: // Public member functions

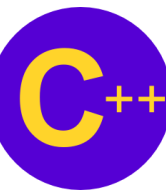
    List(); // constructor
    bool IsEmpty () const;
    bool IsFull () const;
    int Length () const; // Returns length of list
    void Insert (ItemType item);
    void Delete (ItemType item);
    bool IsPresent(ItemType item) const;
    void SelSort ();
    void Reset ();
    ItemType GetNextItem ();

private: // Private data members
    int length; // Number of values currently stored
    ItemType data[MAX_LENGTH];
    int CurrentPos; // Used in iteration
};

```

LECTURE 2

# Implementations of List

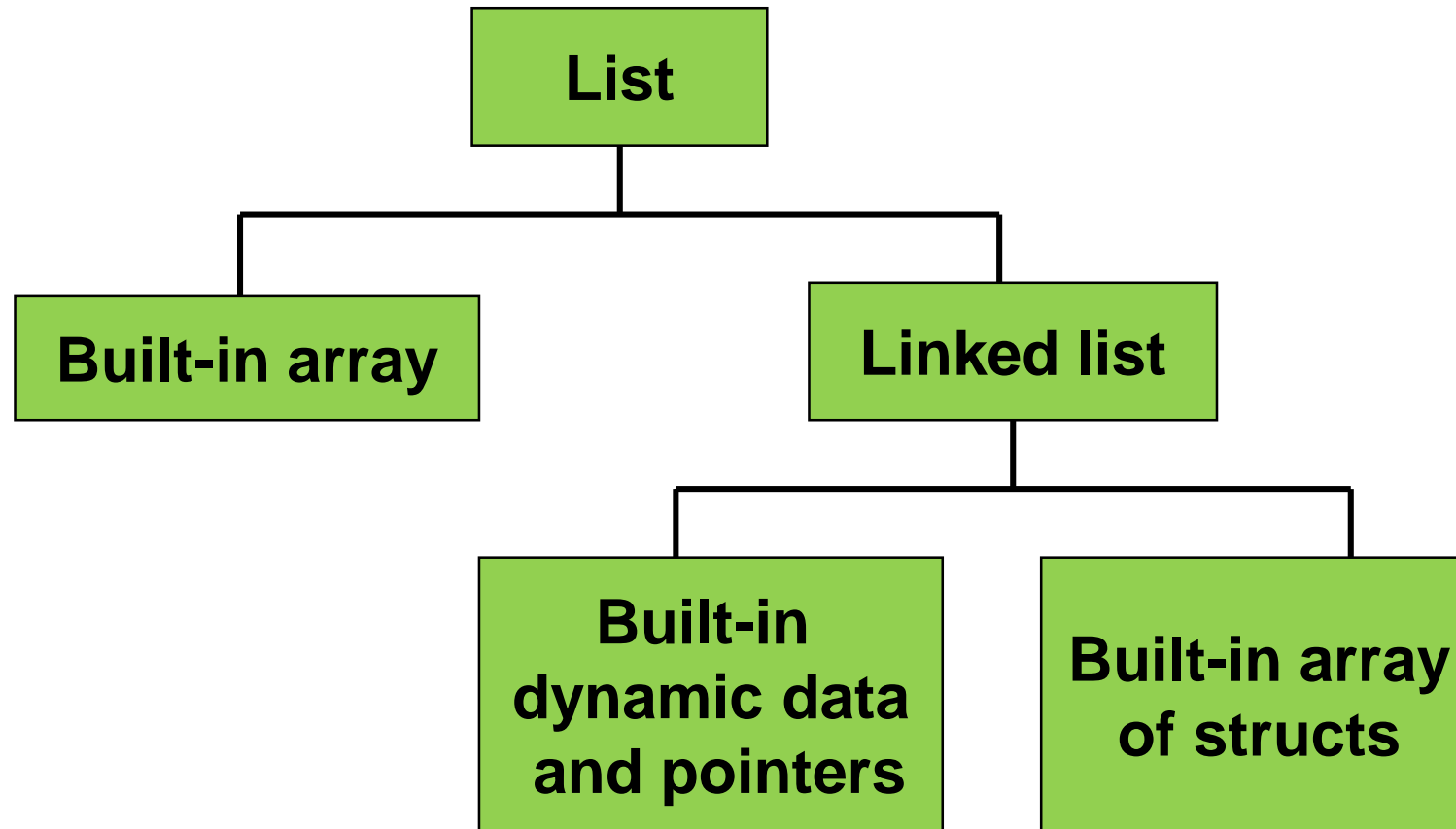


# Implementation Structures

---

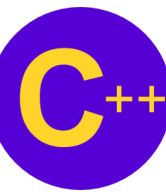
- Use a **built-in array** stored in contiguous memory locations, implementing operations Insert and Delete by moving list items around in the array, as needed
- Use a **linked list** in which items are not necessarily stored in contiguous memory locations
- A linked list avoids excessive data movement from insertions and deletions

# Implementation Possibilities for a List ADT



LECTURE 3

# Nodes

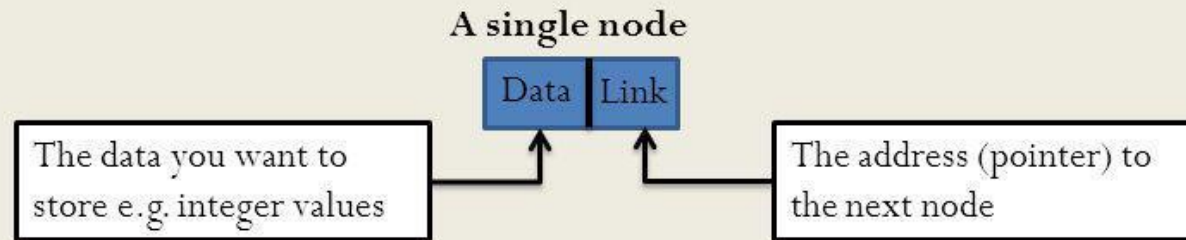


# Node Types

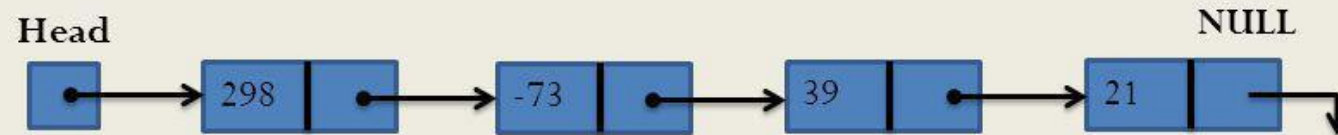
---

- Link Node (Next Node)
- Double Link Node (Next-Prev Node)
- Tree Node (Left-Right Node)
- Matrix Node (Quadruple Node – up/down, left/right node)
- Link Node to Linked List (Adjacent List or Hash Table Entry)  
(List Node the List can be one of the four node types above)

# Linked List in Pictures

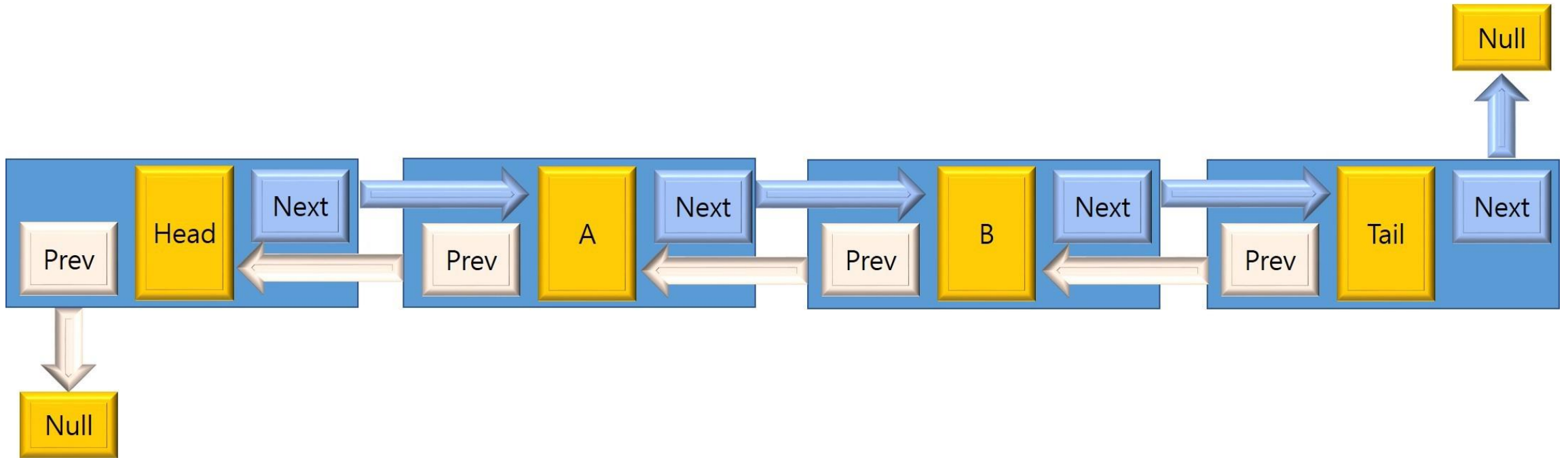


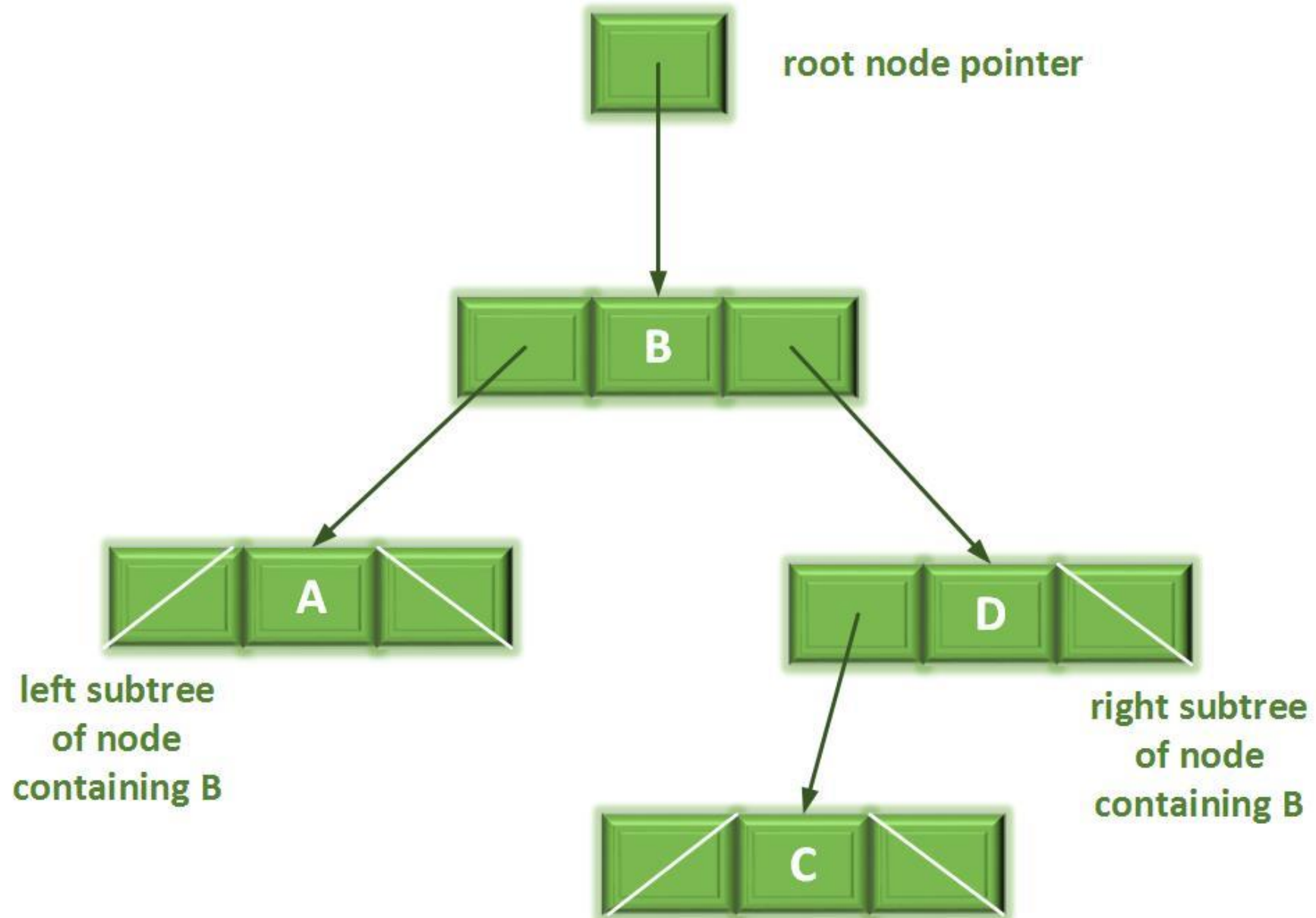
## Example

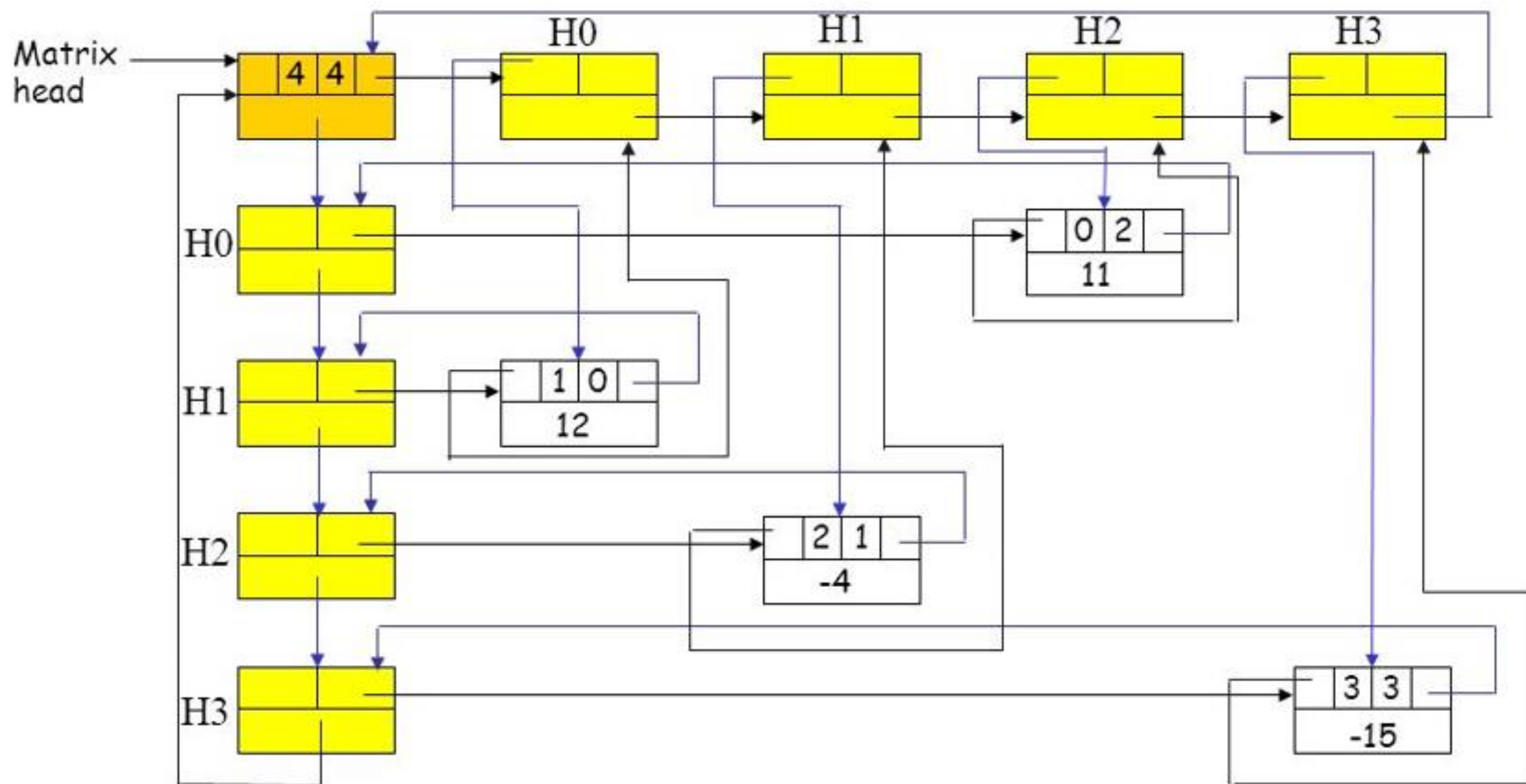


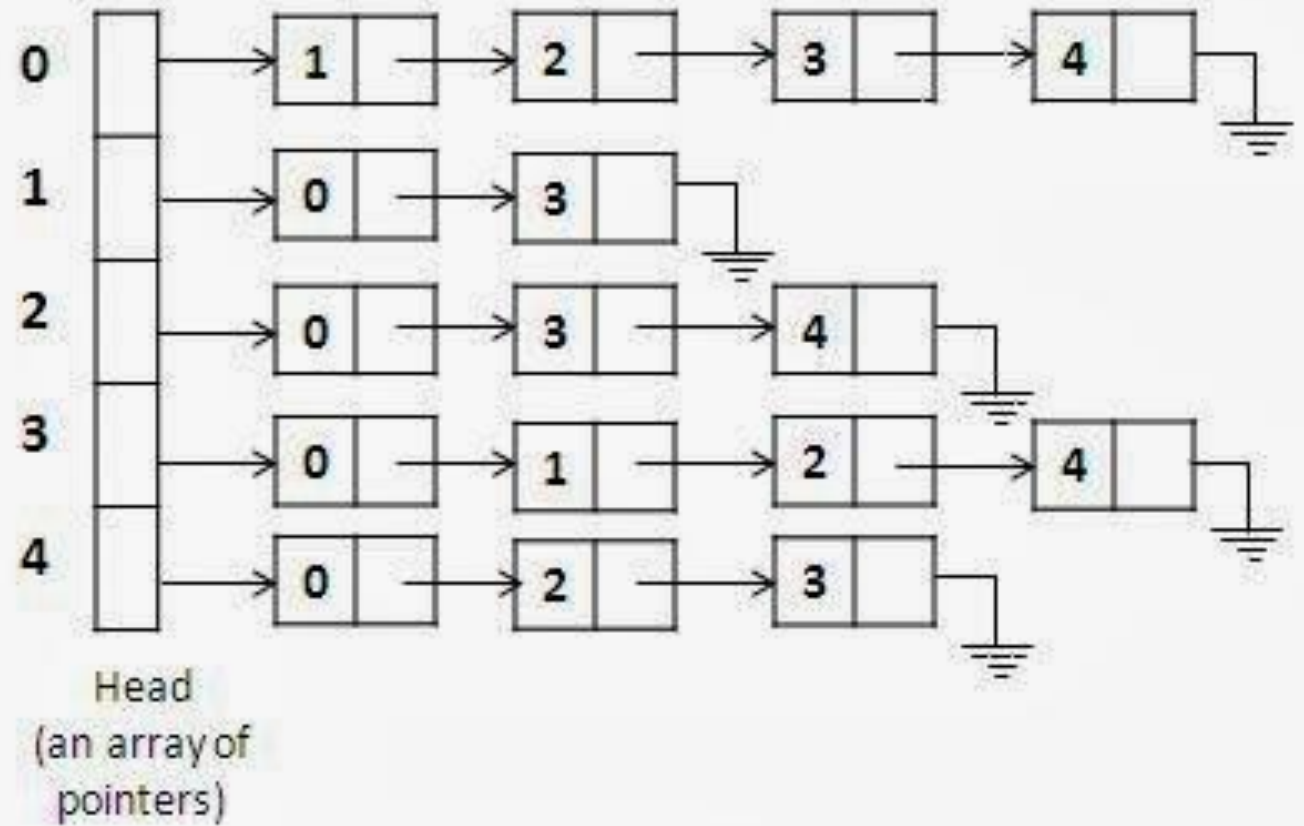
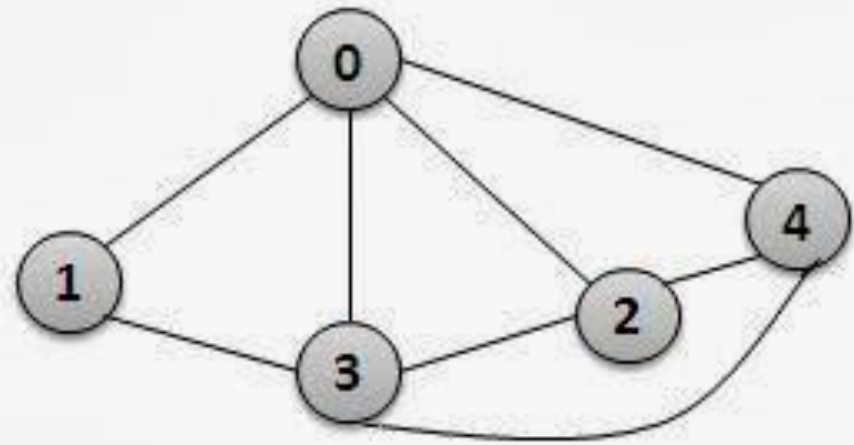
**Linked List:** A list of items, called nodes, in which the order of the nodes is determined by the address, called the link, stored in each node.







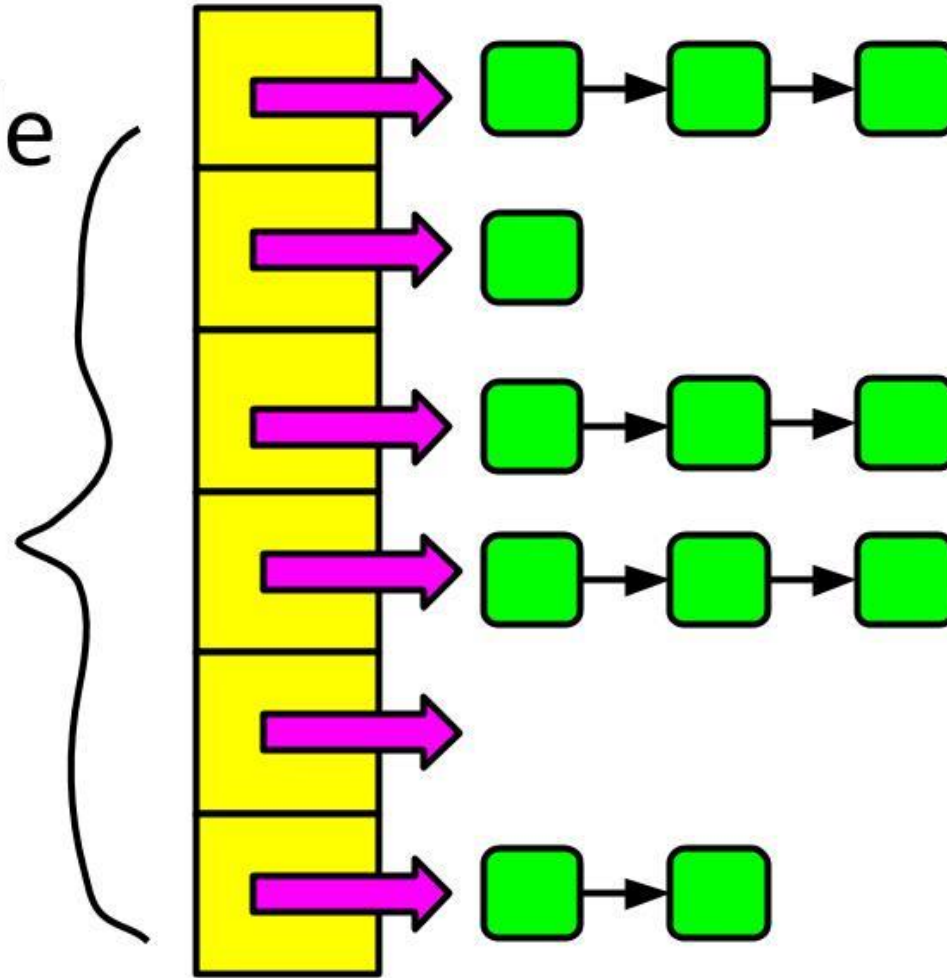


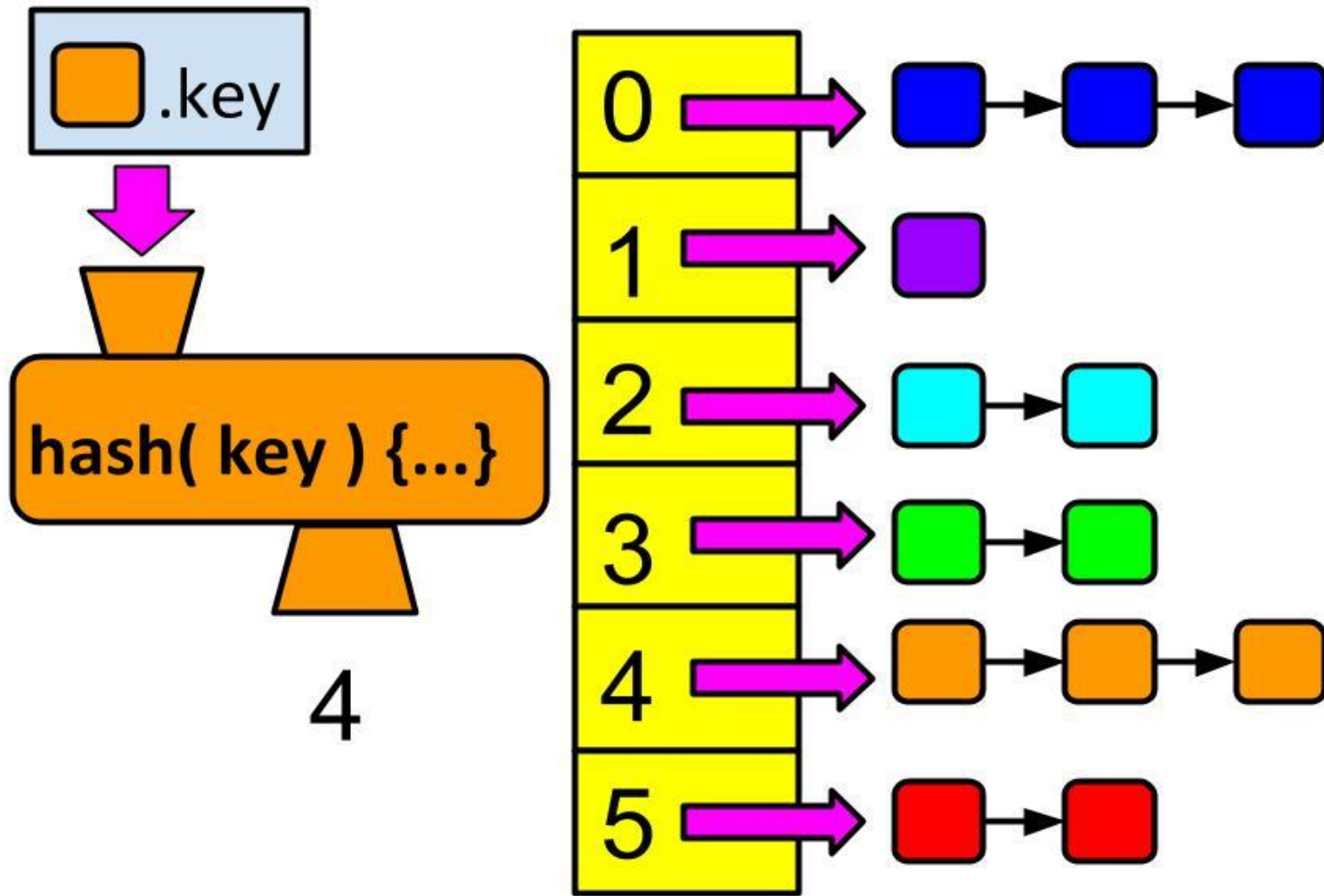


**Adjacency List Representation of Graph**

Hash Table

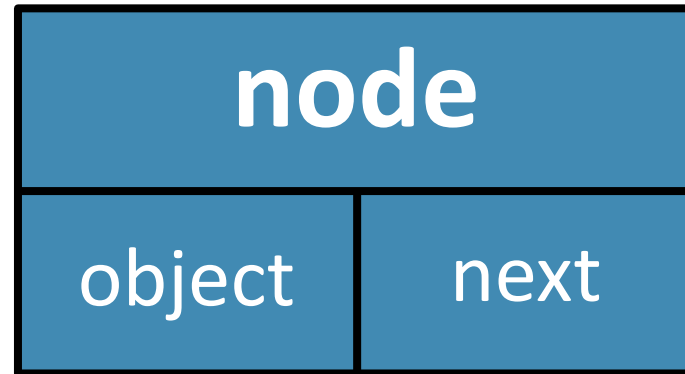
Array of  
Linked Lists





LECTURE 2

# Single Linked Node



**public data:**

`node<T>* next;`

**private data:**

`T object;`

**public method:**

`node();`

`node(T t);`

`get();`

`T`  
`void` `setNext(node<T> *n);`

`node<T>* getNext();`

`bool hasNext();`

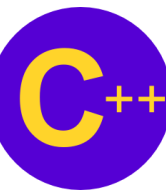
`string to_string();`



```

1  #define MAIN
2  #ifndef NODE_H
3  #define NODE_H
4  #include "to_string.h"
5  using namespace std;
6  template <typename T>
7  class node{
8      public:
9          node<T> *next;
10         node(): object(NULL), next(NULL){}
11         node(T t){ next = NULL; object = t; }
12         void setNext(node<T> *n){ next = n; }
13         node<T> *getNext(){ return next; }
14         bool hasNext(){
15             if (getNext() != NULL) return true;
16             else return false;
17         }
18         string to_string(){ return st::to_string(object); }
19         T get(){ return object; }
20     private:
21         T object;
22 };
23 #endif

```



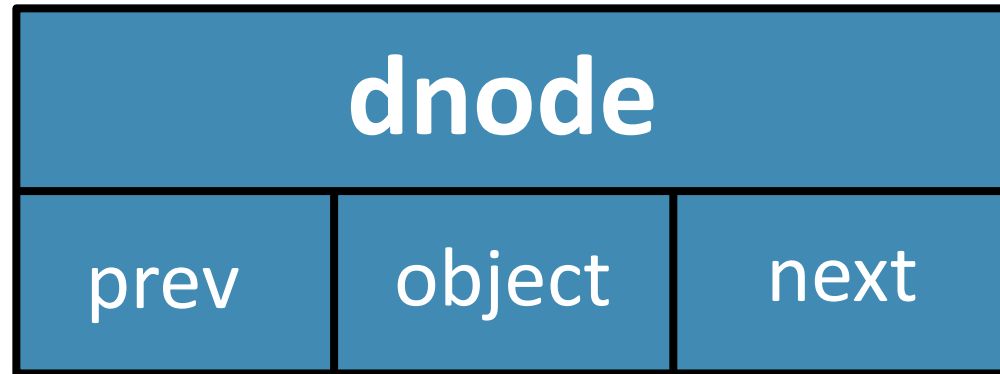
Demo Program: node.cpp

---

Go Notepad++!!!

LECTURE 2

# Double Linked Node



### public data:

```
node<T>* next;
node<T>* prev;
```

### private data:

```
T object;
```

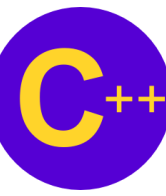
### public method:

```
node();
node(T t);
T get();
void setNext(node<T> *n);
node<T>* getNext();
bool hasNext();
void setPrev(node<T> *n);
node<T>* getPrev();
bool hasPrev();
string to_string();
```

```

1  #define MAIN
2  #ifndef DNODE_H
3  #define DNODE_H
4  #include "to_string.h"
5  using namespace std;
6  template <typename T>
7  class dnode{
8      public:
9          dnode<T> *next;
10         dnode<T> *prev;
11         dnode(): next(NULL), prev(NULL) {}
12         dnode(T t){ next=NULL; prev=NULL; object = t; }
13         void setNext(dnode<T> *n){ next = n; }
14         void setPrev(dnode<T> *p){ prev = p; }
15         dnode<T> *getNext(){ return next; }
16         dnode<T> *getPrev(){ return prev; }
17         bool hasNext(){
18             if (getNext() != NULL) return true;
19             else return false;
20         }
21         bool hasPrev(){
22             if (getPrev() != NULL) return true;
23             else return false;
24         }
25         string to_string(){ return st::to_string(object); }
26         T get(){ return object; }
27     private:
28         T object;
29 };
30 #endif

```



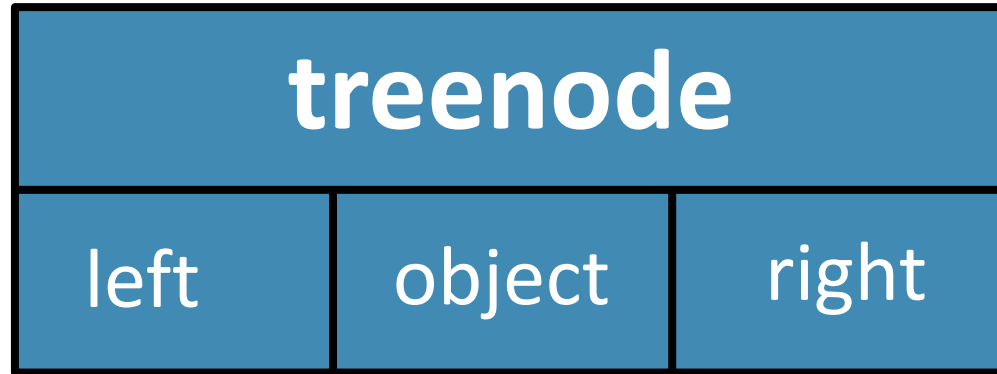
Demo Program: dnode.cpp

---

Go Notepad++!!!

LECTURE 2

# Tree Node



### public data:

```
node<T>* left;
node<T>* right;
```

### private data:

```
T object;
```

### public method:

```
T
void node();
node<T>* node(T t);
bool get();
void setLeft(node<T> *n);
node<T>* getLeft();
bool hasLeft();
void setRight(node<T> *n);
node<T>* getRight();
bool hasRight();
string to_string();

string preorder(treenode<T> *top);
string inorder(treenode<T> *top);
string postorder(treenode<T> *top);
```

#### Note:

If not the tree traversal methods,  
dnode is the same as treenode.



```

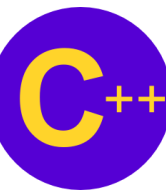
1  #define MAIN
2  #ifndef TREENODE_H
3  #define TREENODE_H
4  #include "to_string.h"
5  using namespace std;
6  template <typename T>
7  class treenode{
8  public:
9      treenode<T> *left;
10     treenode<T> *right;
11     treenode(): left(NULL), right(NULL) {}
12     treenode(T t){ left=NULL; right=NULL; object = t; }
13     void setLeft(treenode<T> *le){ left = le; }
14     void setRight(treenode<T> *r){ right = r; }
15     treenode<T> *getLeft(){ return left; }
16     treenode<T> *getRight(){ return right; }
17     bool hasLeft(){
18         if (getLeft() != NULL) return true;
19         else return false;
20     }
21     bool hasRight(){
22         if (getRight() != NULL) return true;
23         else return false;
24     }
25     string preorder(treenode<T> *top){
26         string rtn = "";
27         if (top != NULL ) {
28             rtn = rtn + top->to_string() + " ";
29             rtn = rtn + top->preorder( top->left );
30             rtn = rtn + top->preorder( top->right );
31         }
32         return rtn;
33     }

```

```

34     string inorder(treenode<T> *top){
35         string rtn = "";
36         if (top != NULL ) {
37             rtn = rtn + top->inorder( top->left );
38             rtn = rtn + top->to_string() + " ";
39             rtn = rtn + top->inorder( top->right );
40         }
41         return rtn;
42     }
43     string postorder(treenode<T> *top){
44         string rtn = "";
45         if (top != NULL ) {
46             rtn = rtn + top->postorder( top->left );
47             rtn = rtn + top->postorder( top->right );
48             rtn = rtn + top->to_string() + " ";
49         }
50         return rtn;
51     }
52     string to_string(){ return st::to_string(object); }
53     T get(){ return object; }
54 private:
55     T object;
56 };
57 #endif
58
59
60
61
62
63
64
65
66
67

```



Demo Program: treenode.cpp

---

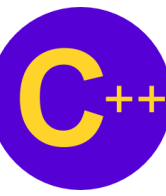
Go Notepad++!!!

LECTURE 3

# Linked List

# Array Representation of a Linked List

	component	link
head	Node[0]	58 -1
2	Node[1]	
	Node[2]	4 5
	Node[3]	
	Node[4]	46 0
	Node[5]	16 7
	Node[6]	
	Node[7]	39 4



# Data Structure of Array Based Linked List

---

```
struct NodeType
```

```
{
```

```
    int component;
```

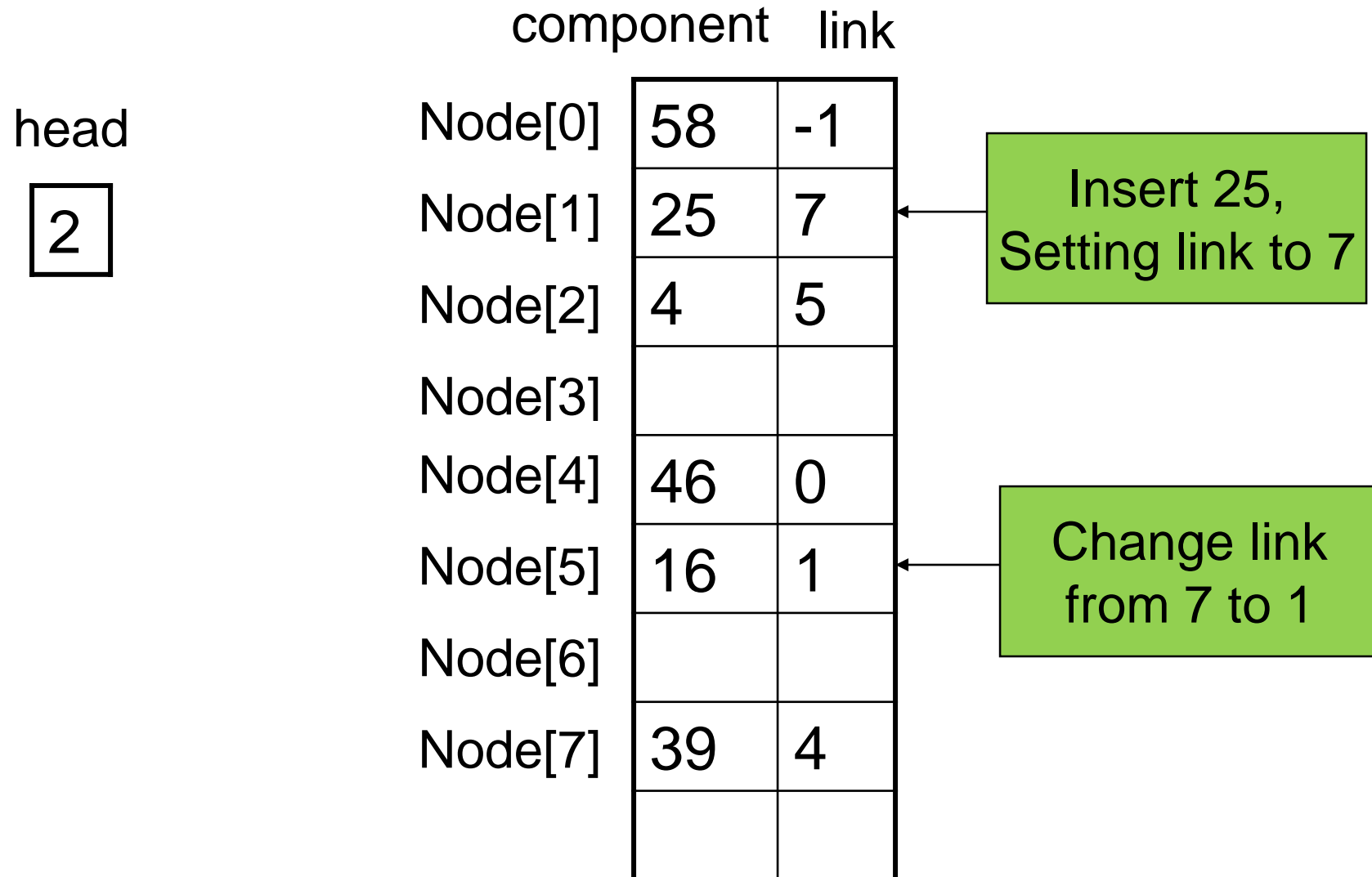
```
    int link;
```

```
};
```

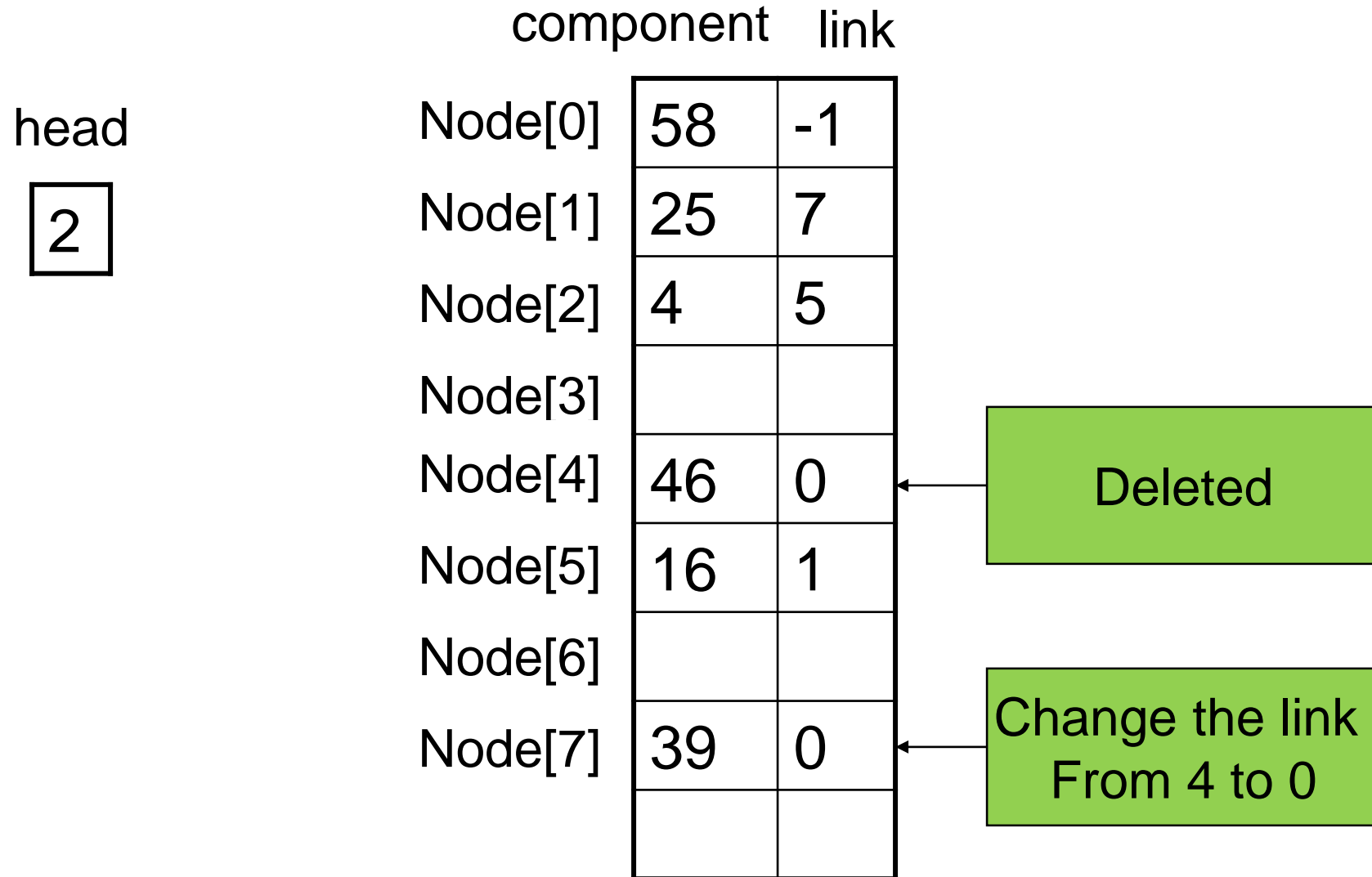
```
NodeType node[1000];    // Max. 1000 nodes
```

```
int head;
```

# Insert a New Node into a Linked List

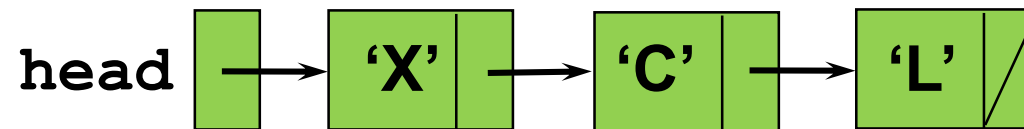


# Delete a Node from a Linked List



# A Linked List

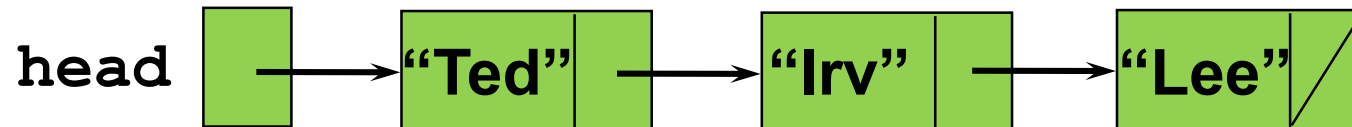
- A **linked list** is a list in which the order of the components is determined by an explicit link member in each node
- Each node is a `struct` containing a data member and a link member that gives the location of the next node in the list

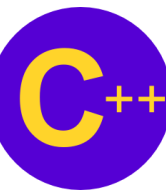




# Dynamic Linked List

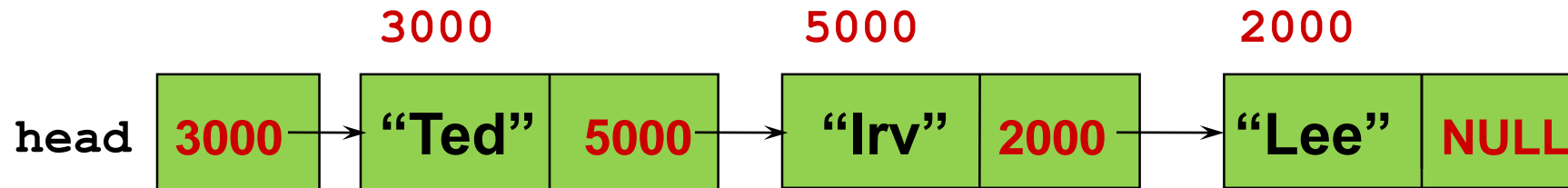
- A **dynamic linked list** is one in which the nodes are linked together by pointers and an external pointer (or head pointer) points to the first node in the list





# Nodes can be located anywhere in memory

- The link member holds the memory address of the next node in the list



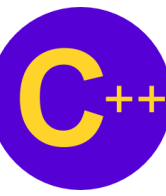
```
#define MAIN
#ifndef LINKEDLIST_H
#define LINKEDLIST_H
#include "node.h"
#include "to_string.h"
using namespace std;
template <typename T>
class linkedlist{
public:
    // data
    int length;
    node<T> *head, *tail;

    // constructor
    linkedlist(){ head=NULL; tail=NULL; length=0; }

    // methods
    int size();
    bool isempty();
    int indexOf(T obj);
    T get(int idx);
    string to_string();
    void set(int idx, T v);
    void add(T v);
    void add_front(T v);
    void insert(int idx, T v);
    node<T> *remove();
    node<T> *remove_front();
    void append(linkedlist<T> *alist);
};
#endif
```

LECTURE 4

# Dynamic Linked List



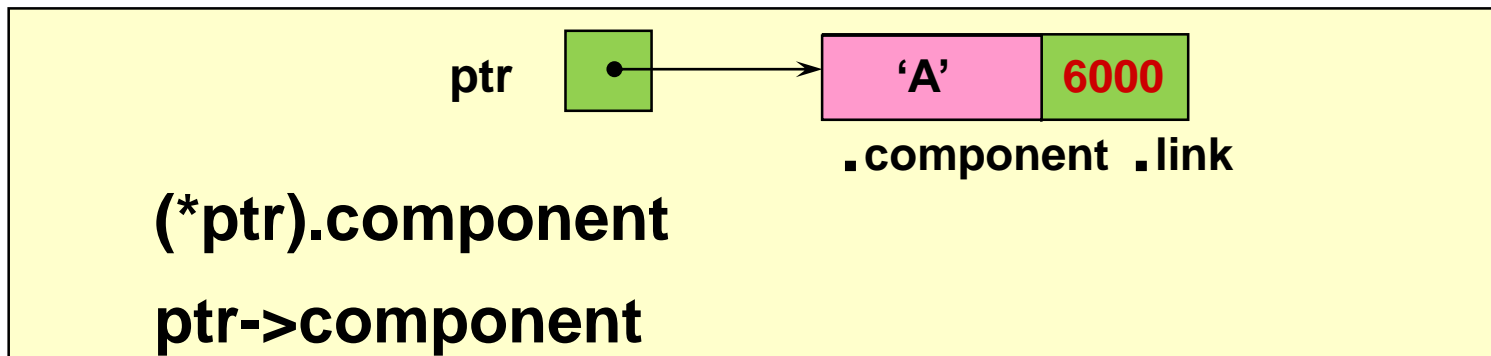
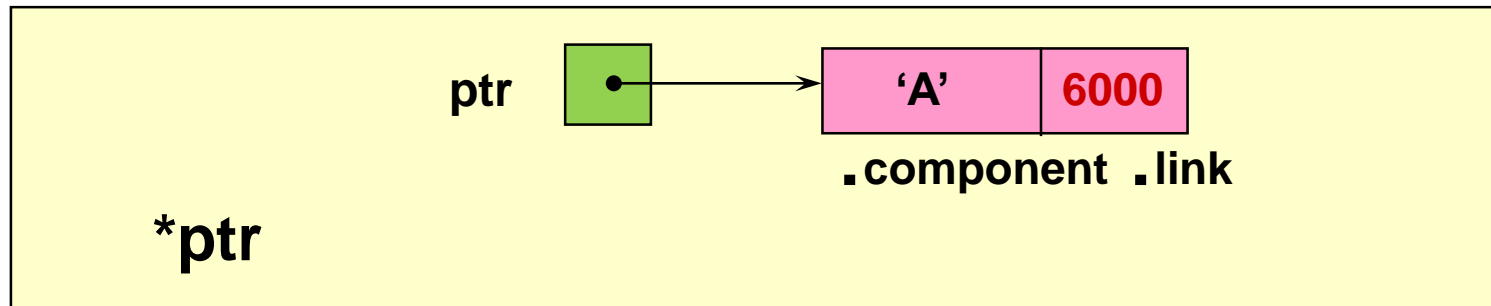
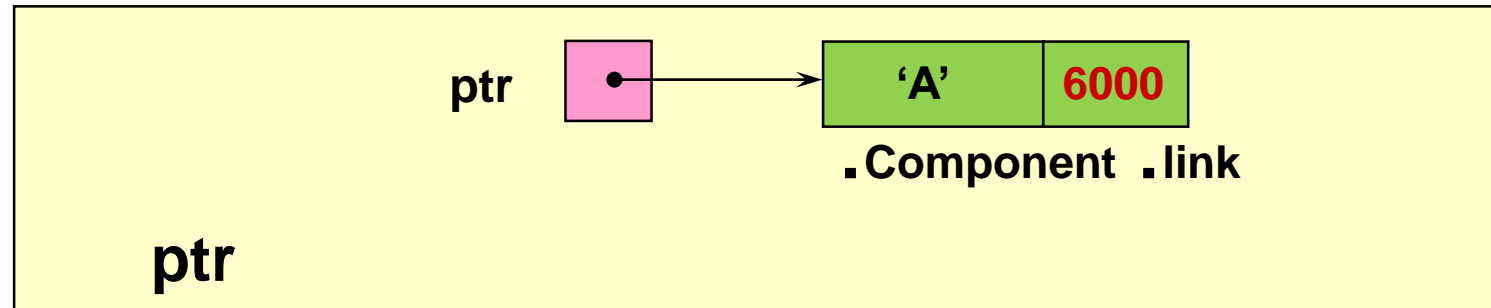
# Declarations for a Dynamic Linked List

```
// Type declarations
struct NodeType {
    char component;
    NodeType* link;
}
typedef NodeType* NodePtr;
// Variable DECLARATIONS
NodePtr head;
NodePtr ptr;
```



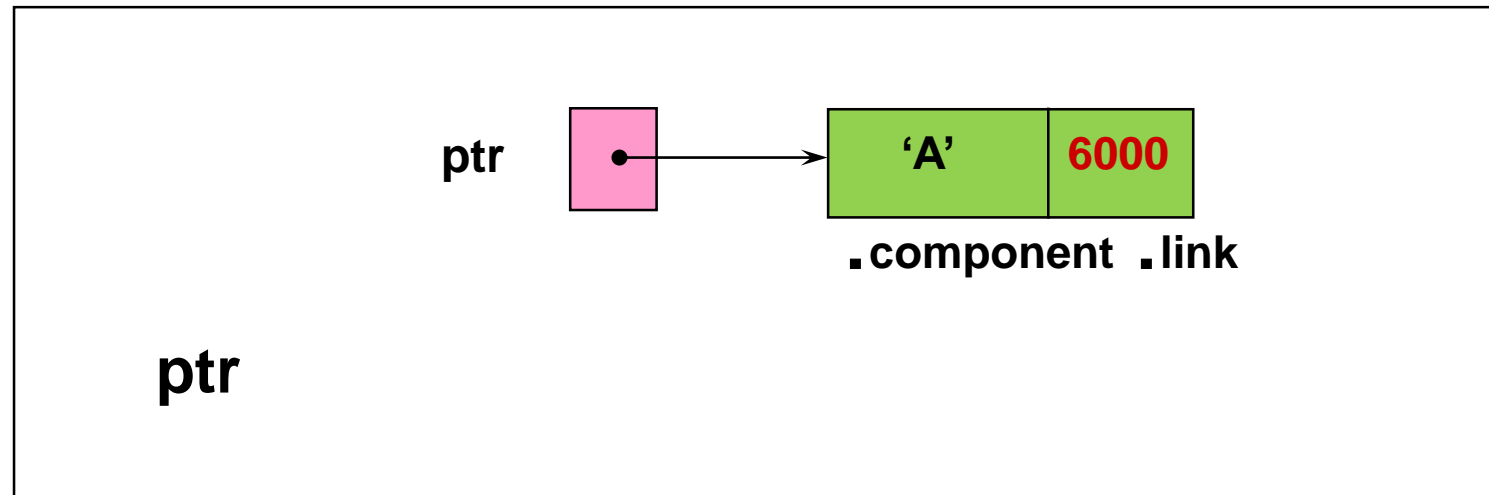
**.component    .link**

# Pointer Dereferencing and Member Selection



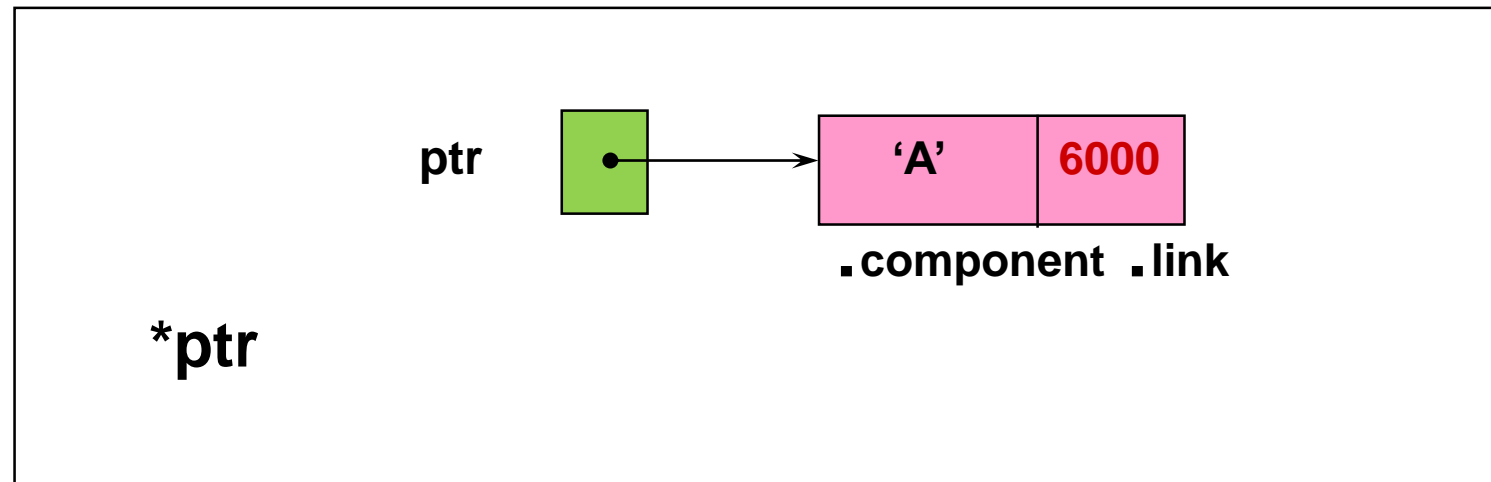
`ptr` is a pointer to a node

---



`*ptr` is the entire node pointed to by `ptr`

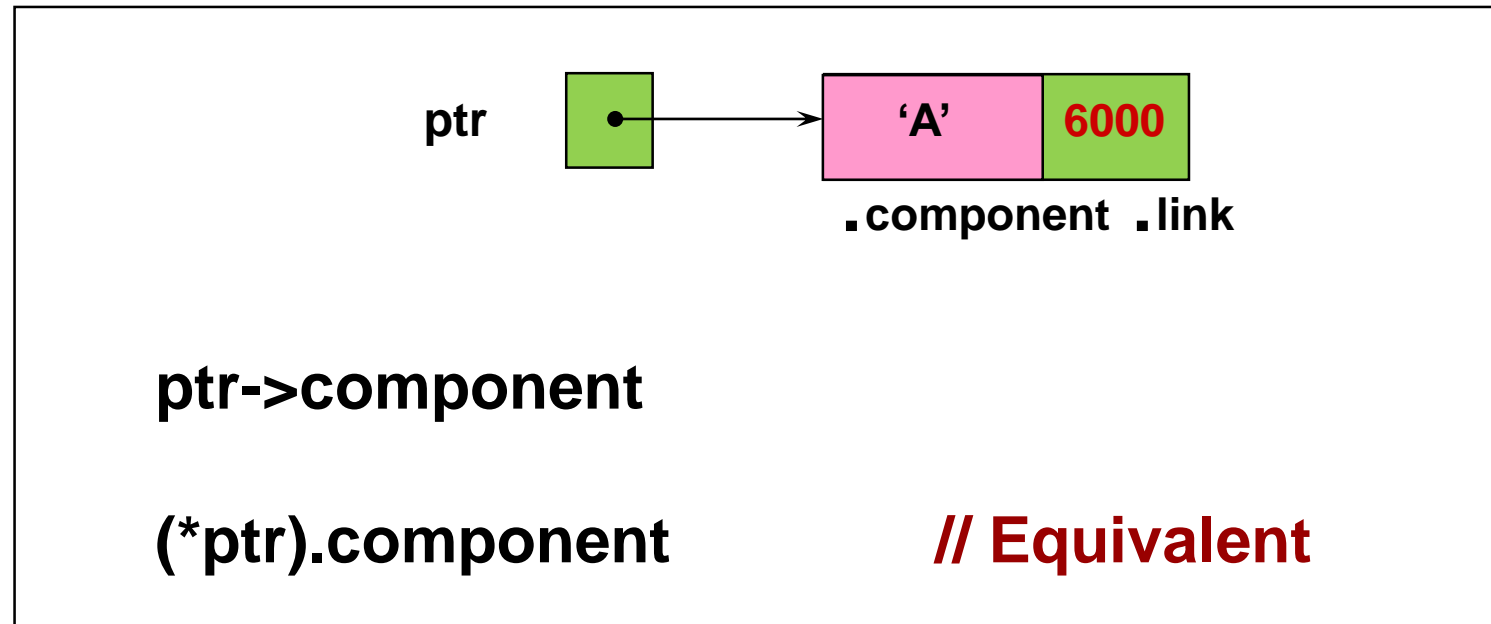
---

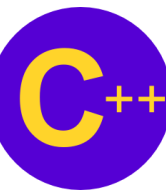




`ptr->component` is a node member

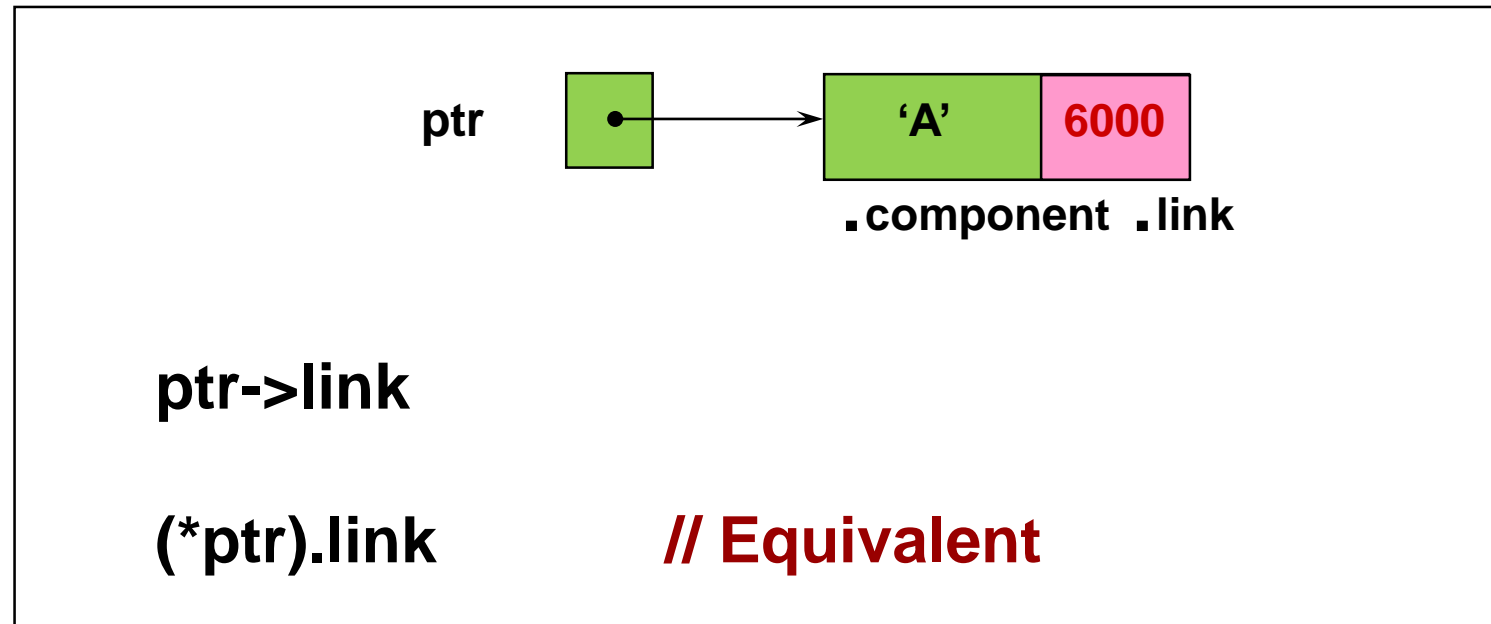
---



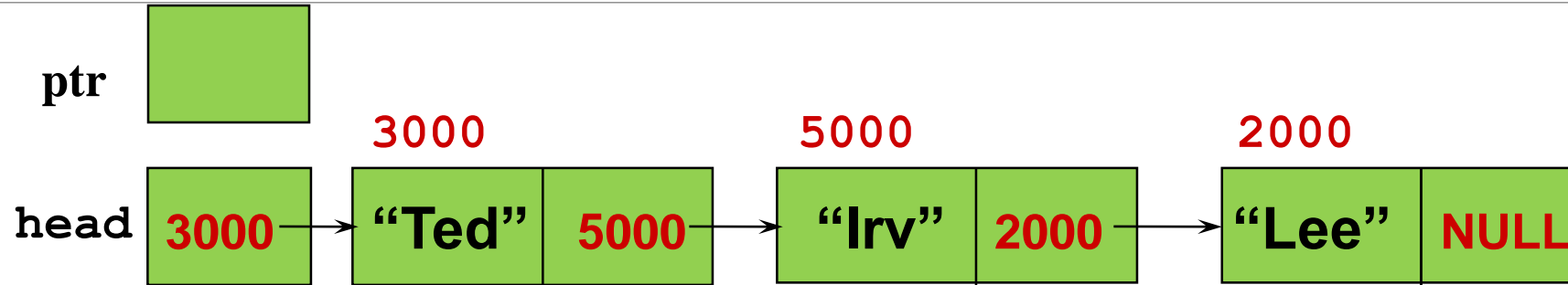


# `ptr->link` is a node member

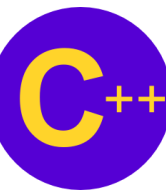
---



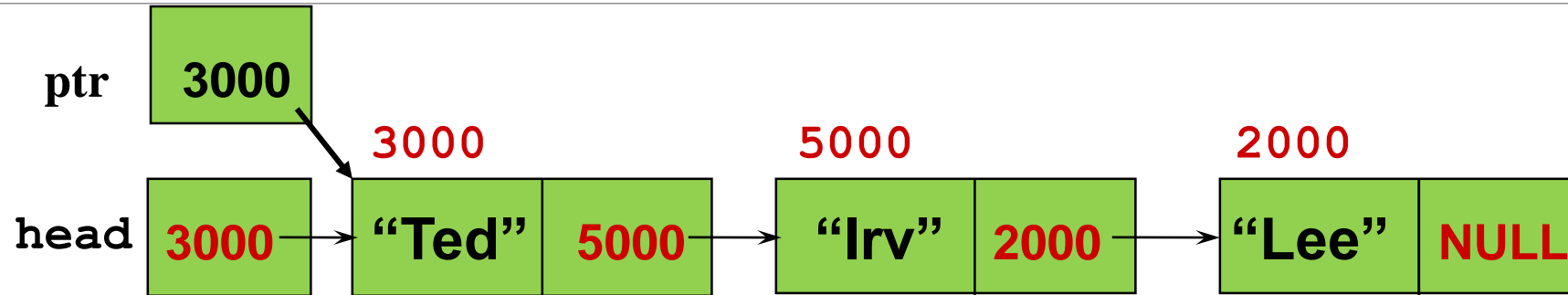
# Traversing a Dynamic Linked List



```
// Pre: head points to a dynamic linked list
ptr = head;
while (ptr != NULL)
{
    cout << ptr->component
    // Or, do something else with node *ptr
    ptr = ptr->link;
}
```



# Traversing a Dynamic Linked List



```
// Pre: head points to a dynamic linked list
```

```
ptr = head;
```

```
while (ptr != NULL)
```

```
{
```

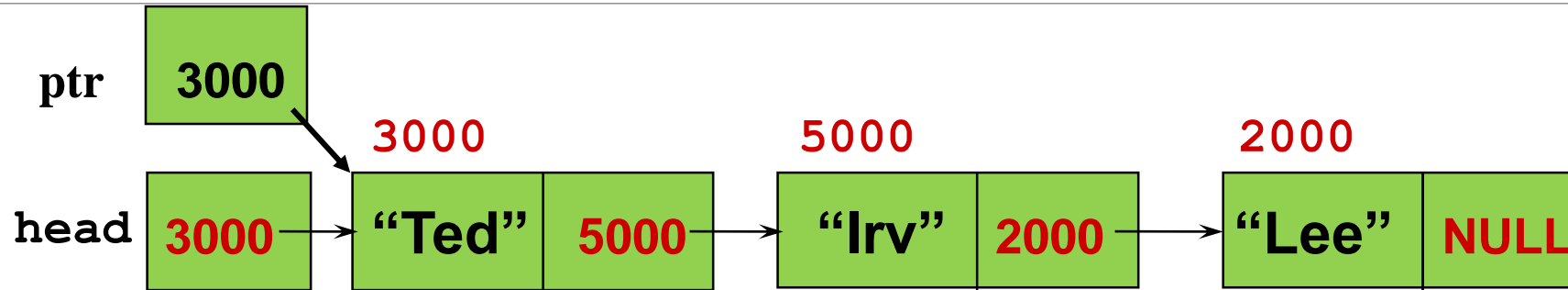
```
    cout << ptr->component;
```

```
    // Or, do something else with node *ptr
```

```
    ptr = ptr->link;
```

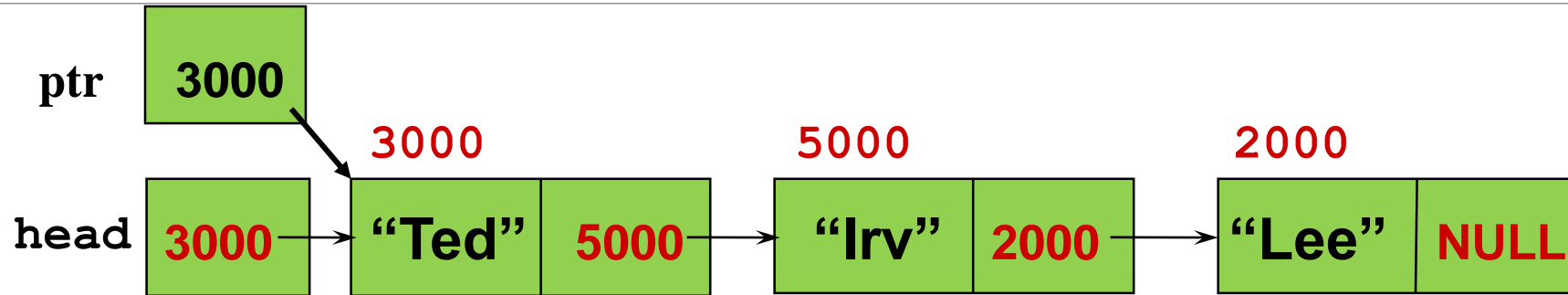
```
}
```

# Traversing a Dynamic Linked List



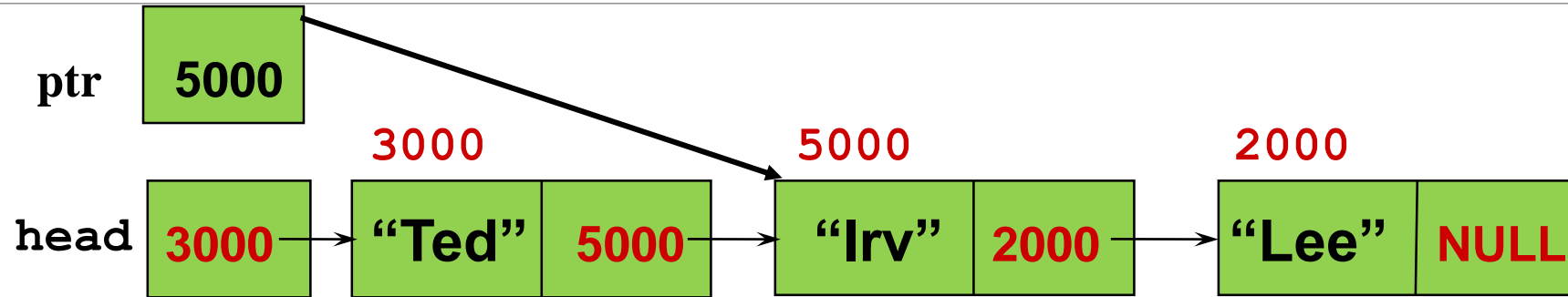
```
// Pre: head points to a dynamic linked list
ptr = head;
while (ptr != NULL)
{
    cout << ptr->component;
    // Or, do something else with node *ptr
    ptr = ptr->link;
}
```

# Traversing a Dynamic Linked List

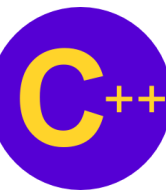


```
// Pre: head points to a dynamic linked list
ptr = head;
while (ptr != NULL)
{
    cout << ptr->component;
    // Or, do something else with node *ptr
    ptr = ptr->link;
}
```

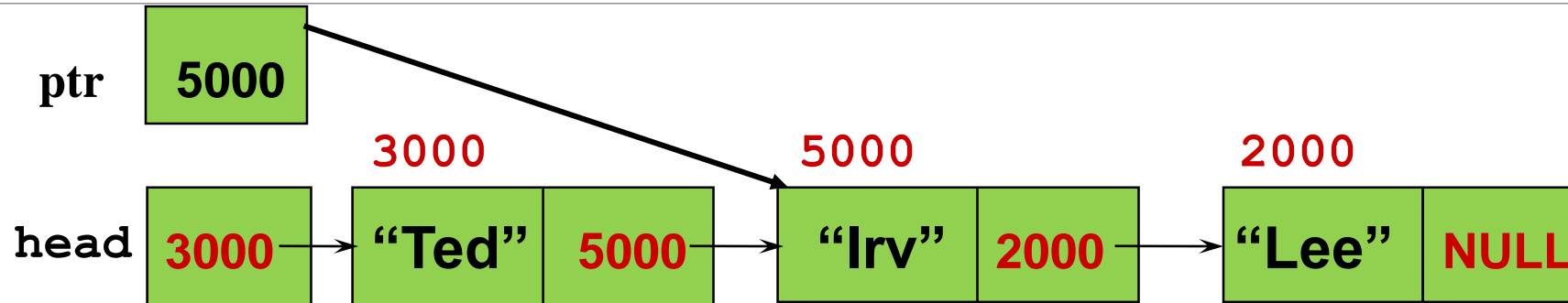
# Traversing a Dynamic Linked List



```
// Pre: head points to a dynamic linked list
ptr = head;
while (ptr != NULL)
{
    cout << ptr->component;
    // Or, do something else with node *ptr
    ptr = ptr->link;
}
```



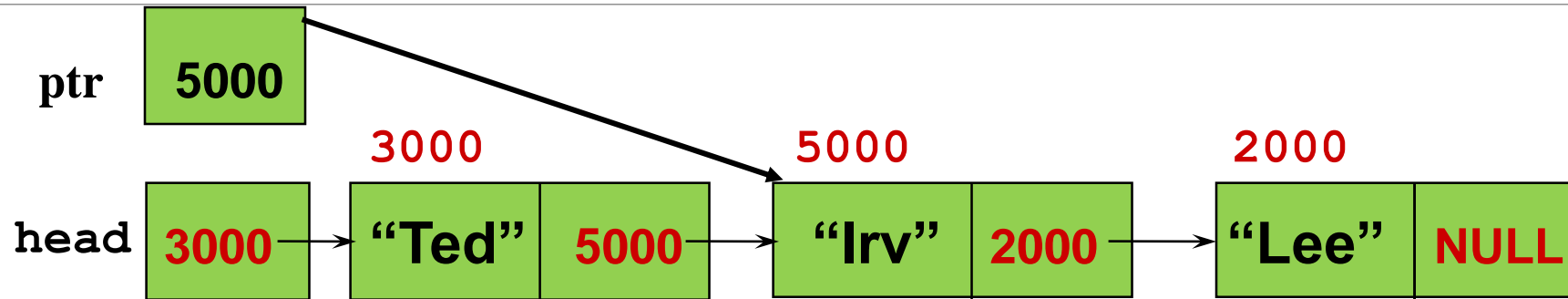
# Traversing a Dynamic Linked List



```
// Pre: head points to a dynamic linked list
ptr = head;
while (ptr != NULL)
{
    cout << ptr->component;
    // Or, do something else with node *ptr
    ptr = ptr->link;
}
```

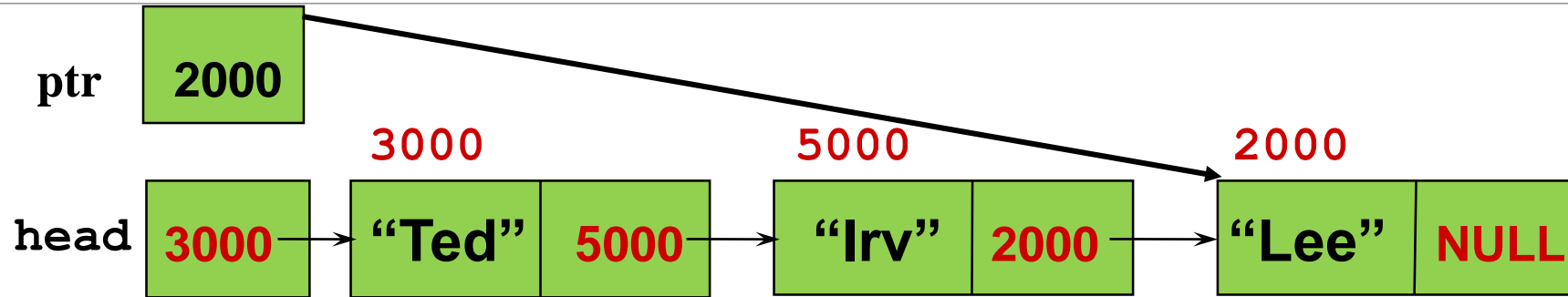


# Traversing a Dynamic Linked List

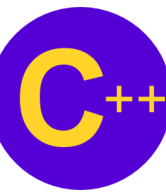


```
// Pre: head points to a dynamic linked list
ptr = head;
while (ptr != NULL)
{
    cout << ptr->component;
    // Or, do something else with node *ptr
    ptr = ptr->link;
}
```

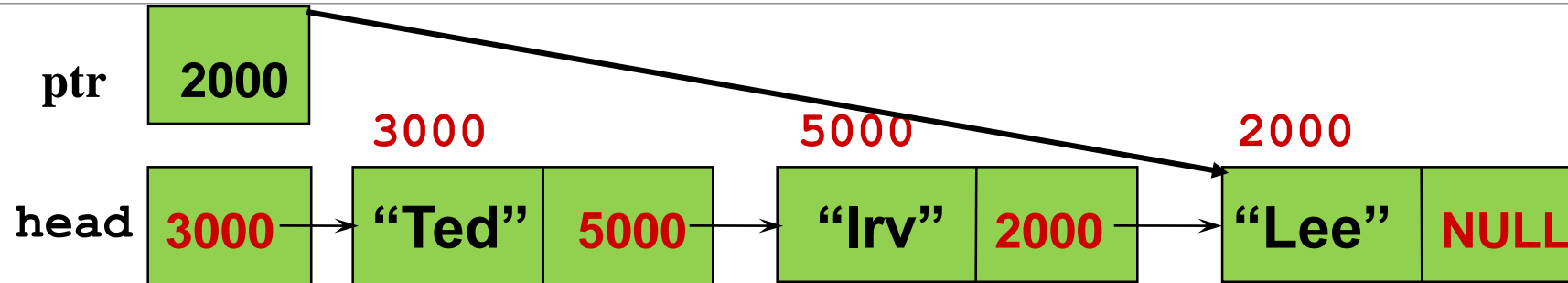
# Traversing a Dynamic Linked List



```
// Pre: head points to a dynamic linked list
ptr = head;
while (ptr != NULL)
{
    cout << ptr->component;
    // Or, do something else with node *ptr
    ptr = ptr->link;
}
```

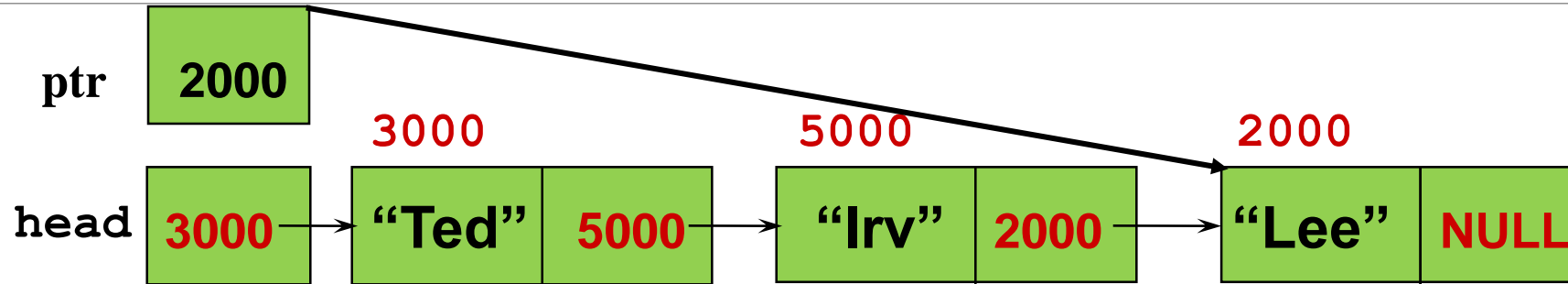


# Traversing a Dynamic Linked List

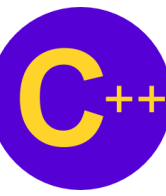


```
// Pre: head points to a dynamic linked list
ptr = head;
while (ptr != NULL)
{
    cout << ptr->component;
    // Or, do something else with node *ptr
    ptr = ptr->link;
}
```

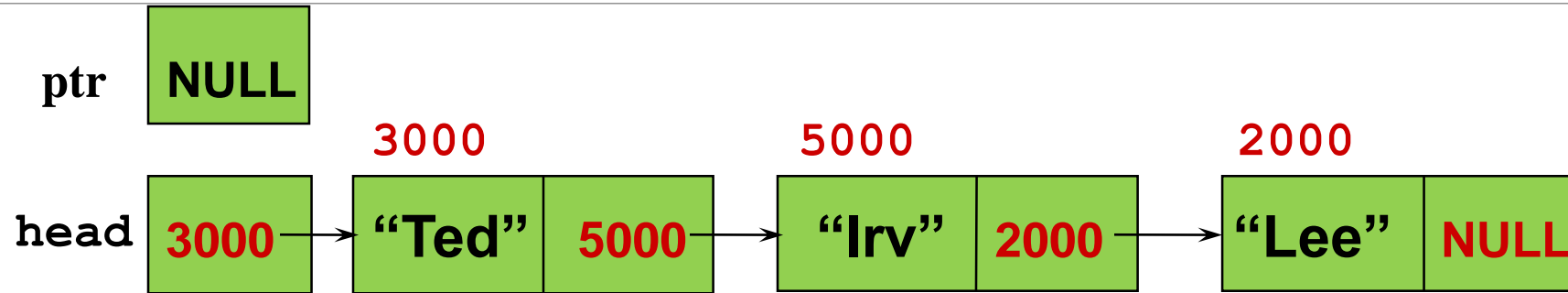
# Traversing a Dynamic Linked List



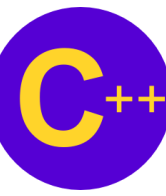
```
// Pre: head points to a dynamic linked list
ptr = head;
while (ptr != NULL)
{
    cout << ptr->component;
    // Or, do something else with node *ptr
    ptr = ptr->link;
}
```



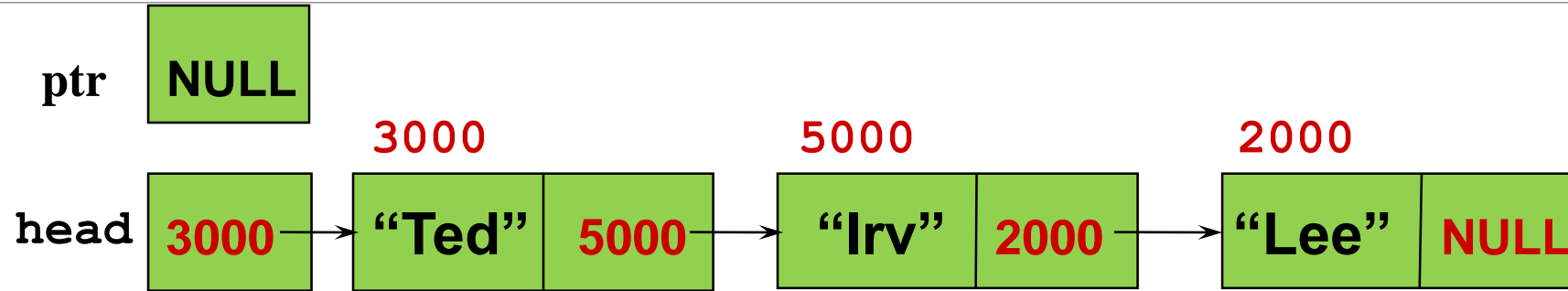
# Traversing a Dynamic Linked List



```
// Pre: head points to a dynamic linked list
ptr = head;
while (ptr != NULL)
{
    cout << ptr->component;
    // Or, do something else with node *ptr
    ptr = ptr->link;
}
```



# Traversing a Dynamic Linked List



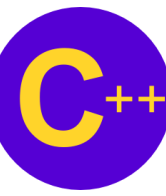
```
// Pre: head points to a dynamic linked list
ptr = head;
while (ptr != NULL)
{
    cout << ptr->component;
    // Or, do something else with node *ptr
    ptr = ptr->link;
}
```

```
template <typename T>
string linkedlist<T>::to_string(){
    string str("");
    node<T> *p = head;
    str += "[";
    int count = 0;
    while (p!= NULL){
        if (count ==0) str += st::to_string(head->get());
        else str += ", " + st::to_string(p->get());
        count++;
        p=p->next;
    }
    str += "]";
    return str;
}
```

## LECTURE 4

# new Operator for Dynamic Linked List





# Using Operator **new**

---

## Recall

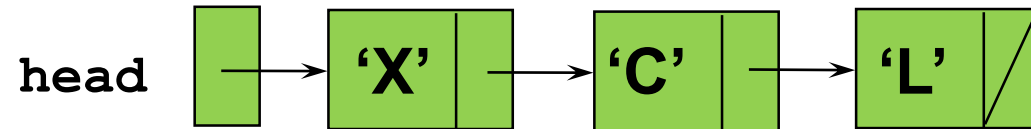
- If memory is available in the free store (or heap), operator new allocates the requested object and returns a pointer to the memory allocated
- The dynamically allocated object exists until the delete operator destroys it

# Inserting a Node at the Front of a List

item

**'B'**

```
char    item = 'B';  
  
NodePtr location;  
location = new  NodeType;  
location->component = item;  
location->link = head;  
head = location;
```



# Inserting a Node at the Front of a List

item

**'B'**

```
char    item = 'B' ;
```

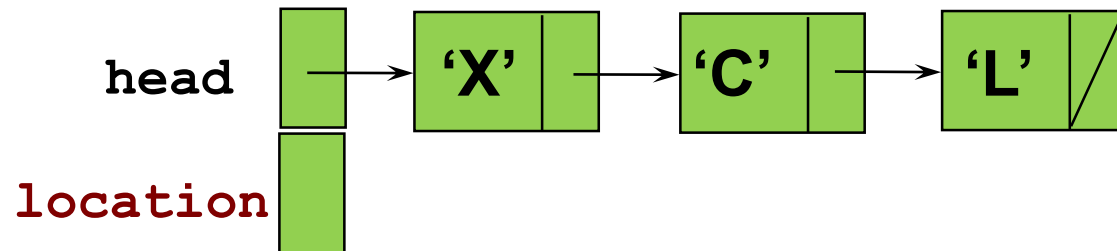
```
NodePtr location;
```

```
location = new NodeType;
```

```
location -> component = item;
```

```
location -> link = head;
```

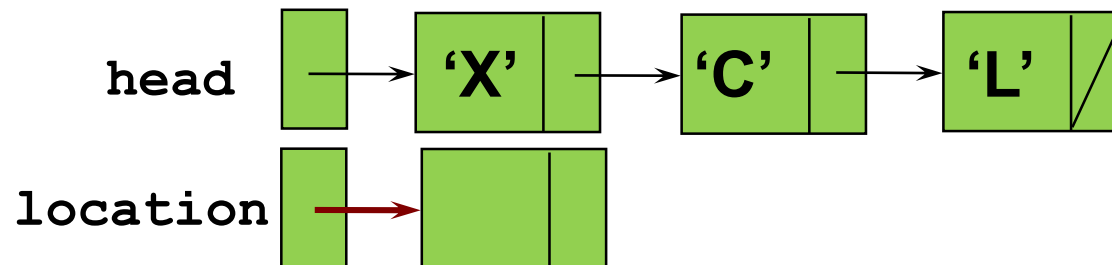
```
head = location;
```



# Inserting a Node at the Front of a List

item **'B'**

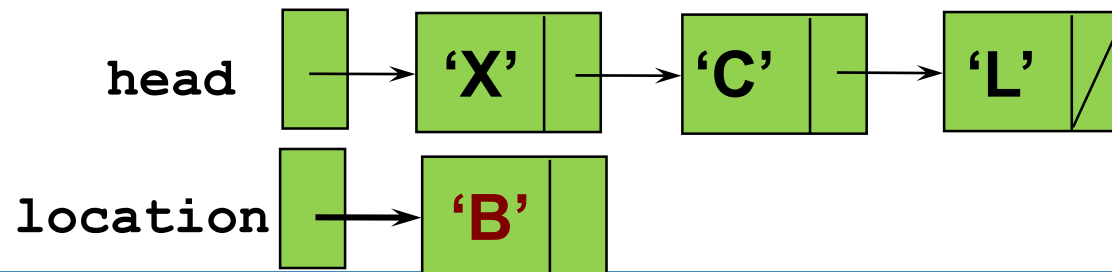
```
char    item = 'B';  
NodePtr location;  
location = new NodeType;  
location -> component = item;  
location -> link = head;  
head = location;
```



# Inserting a Node at the Front of a List

item **'B'**

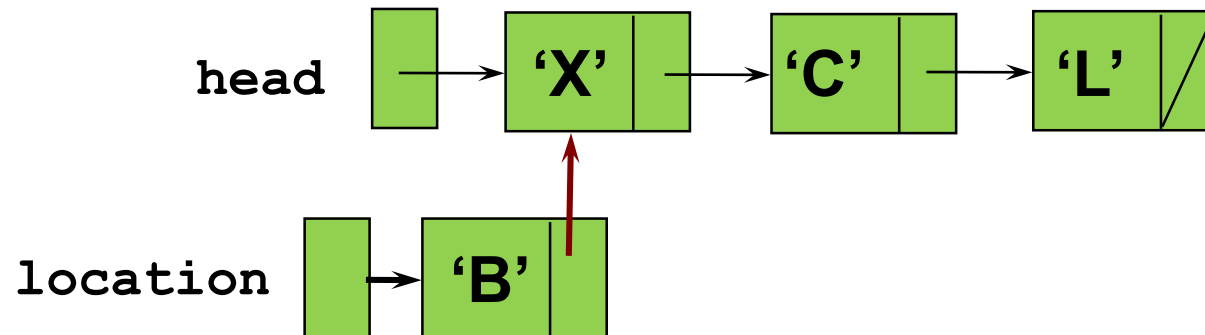
```
char    item = 'B';  
NodePtr location;  
location = new NodeType;  
location -> component = item;  
location -> link = head;  
head = location;
```



# Inserting a Node at the Front of a List

item **'B'**

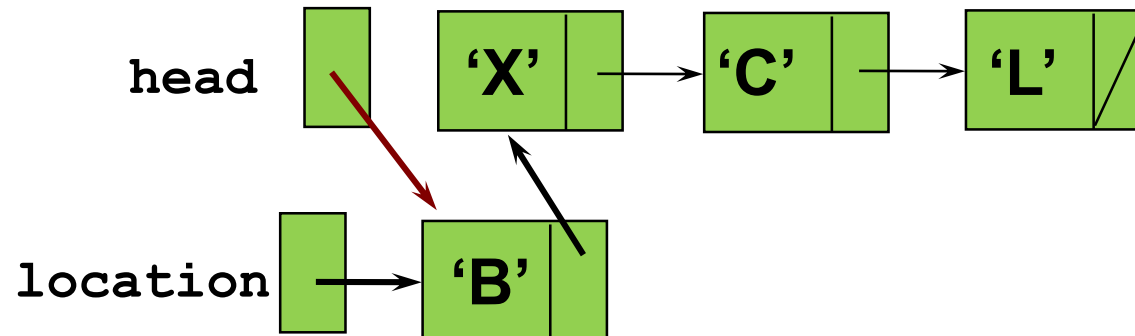
```
char    item = 'B';  
NodePtr location;  
location = new NodeType;  
location -> component = item;  
location -> link = head;  
head = location;
```



# Inserting a Node at the Front of a List

item **'B'**

```
char    item = 'B';  
NodePtr location;  
location = new NodeType;  
location -> component = item;  
location -> link = head;  
head = location;
```



```
template <typename T>
void linkedlist<T>::add(T v){
    length++;
    node<T> *n = new node<T>(v);
    if (head == NULL){
        head = n;
        head->next = NULL;
        tail = n;
        return;
    }

    node<T> *p = (node<T> *) head;
    node<T> *q = NULL;
    while (p!= NULL){
        q = p;
        p=p->next;
    }
    tail = n;
    q->next = n;
    n->next = NULL;
}
```

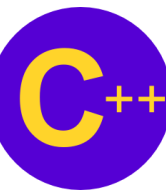
```
template <typename T>
void linkedlist<T>::add_front(T v){
    length++;
    node<T> *n = new node<T>(v);
    if (head == NULL){
        head = n;
        head->next = NULL;
        tail = n;
        return;
    }

    node<T> *p = (node<T> *) head;
    head = n;
    n->next = p;
}
```



## LECTURE 4

# delete Operator for Dynamic Linked List



# Using Operator `delete`

---

- When you use the operator `delete`
- The object currently pointed to by the pointer is deallocated and the pointer is considered undefined
- The object's memory is returned to the free store

# Deleting the First Node from the List

item 

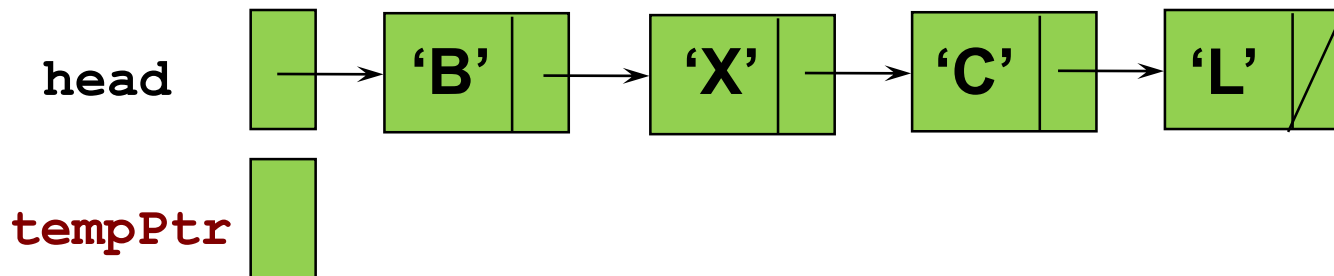
```
NodePtr tempPtr;
```

```
item = head->component;
```

```
tempPtr = head;
```

```
head = head->link;
```

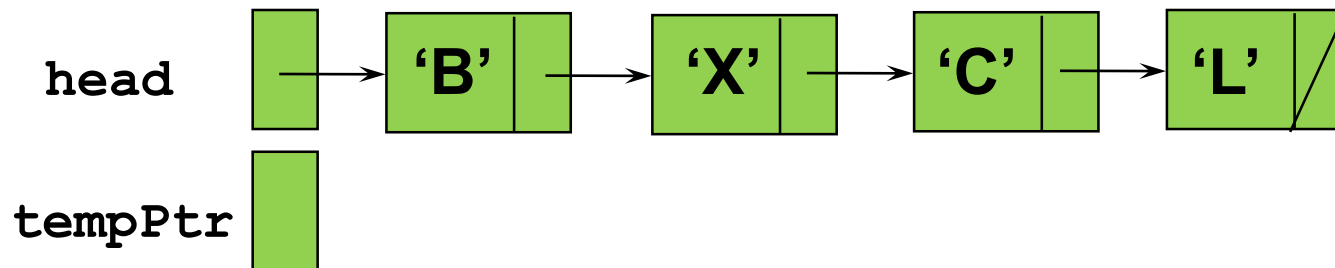
```
delete tempPtr;
```



# Deleting the First Node from the List

item **'B'**

```
NodeType * tempPtr;  
item = head->component;  
tempPtr = head;  
head = head->link;  
delete tempPtr;
```

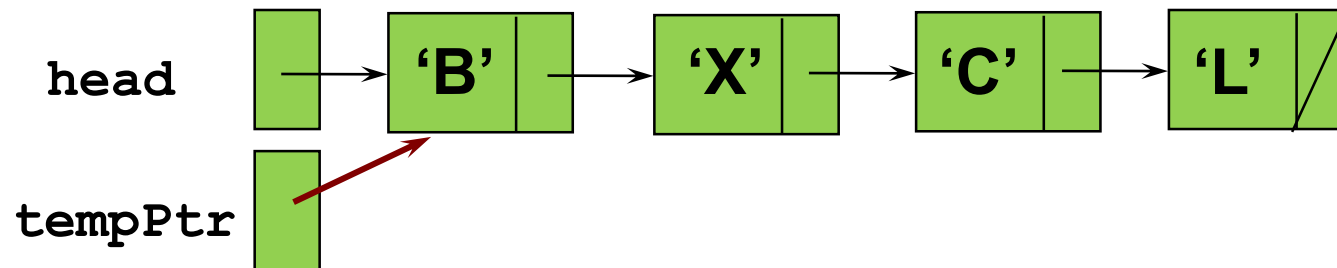


# Deleting the First Node from the List

item

**'B'**

```
NodeType * tempPtr;  
item = head->component;  
tempPtr = head;  
head = head->link;  
delete tempPtr;
```

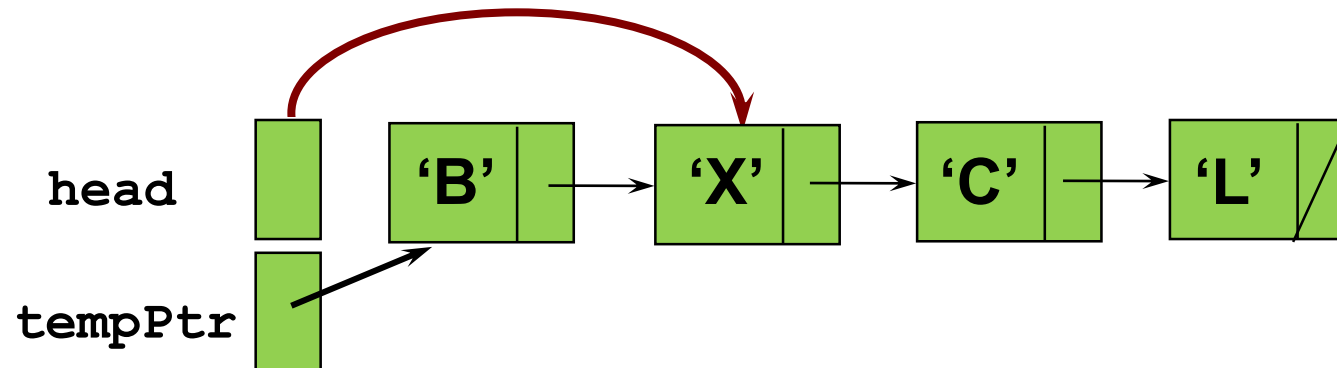


# Deleting the First Node from the List

item

**'B'**

```
NodeType * tempPtr;  
item = head->component;  
tempPtr = head;  
head = head->link;  
delete tempPtr;
```

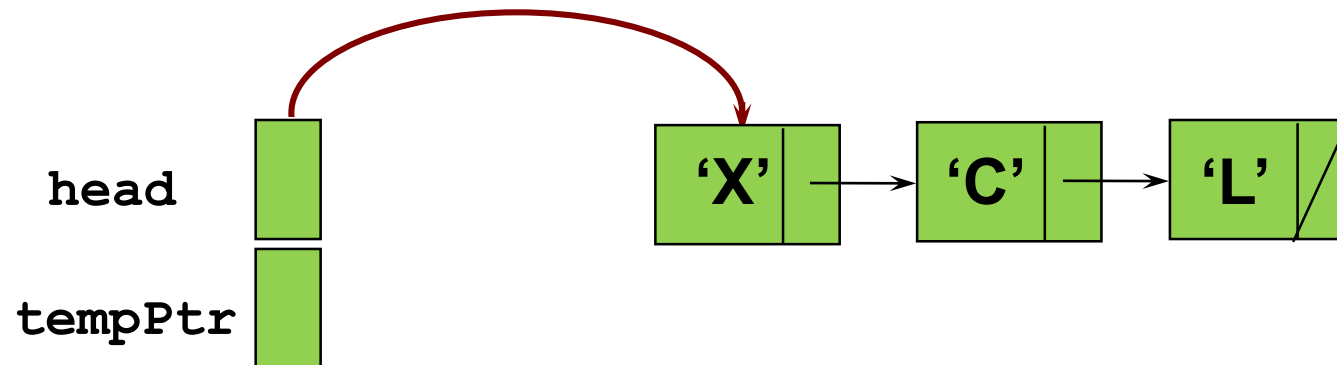


# Deleting the First Node from the List

item

**'B'**

```
NodeType * tempPtr;  
item = head->component;  
tempPtr = head;  
head = head->link;  
delete tempPtr;
```



```

template <typename T>
node<T>* linkedlist<T>::remove(){
    if (head == NULL){ // zero element
        return NULL;
    }

    length--;
    node<T> *p = (node<T> *) head;
    node<T> *q = NULL;
    node<T> *r = NULL;
    while (p!= NULL){
        r = q;
        q = p;
        p=p->next;
    }

    if (r==NULL){ // only one element
        head = NULL;
        tail = NULL;
        return q;
    }

    r->next = NULL;
    tail = r;
    return q;
}

```

```

template <typename T>
node<T>* linkedlist<T>::remove_front(){
    if (head == NULL){ // zero element
        return NULL;
    }

    length--;
    if (head->next == NULL){
        node<T> * q = head;
        head = NULL;
        tail = NULL;
        return q;
    }

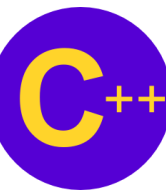
    node<T> * q = head;
    head = head->next;
    return q;
}

```



LECTURE 4

# Sorted List



# What is a Sorted List?

---

- A **sorted list** is a variable-length, linear collection of homogeneous elements, ordered according to the value of one or more data members
- The transformer operations must maintain the ordering
- In addition to Insert and Delete, let's add two new operations to our list

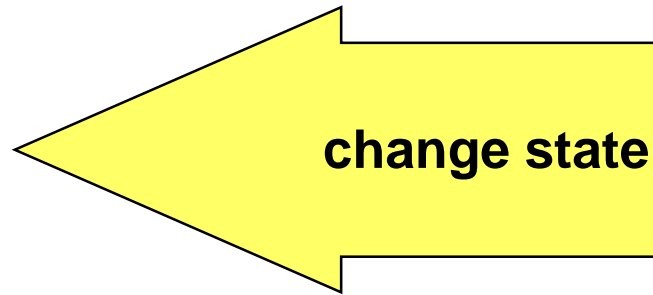
***InsertAsFirst and RemoveFirst***

# ADT HybridList Operations

---

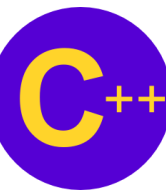
## Transformers

- InsertAsFirst
- Insert
- RemoveFirst
- Delete



## Same observers and iterators as ADT List

Since we have two insertion and two deletion operations, let's call this a Hybrid List



# struct NodeType

```
// Specification file sorted list ("slist2.h")

typedef int ItemType; // Type of each component is
                      // a simple type or a string

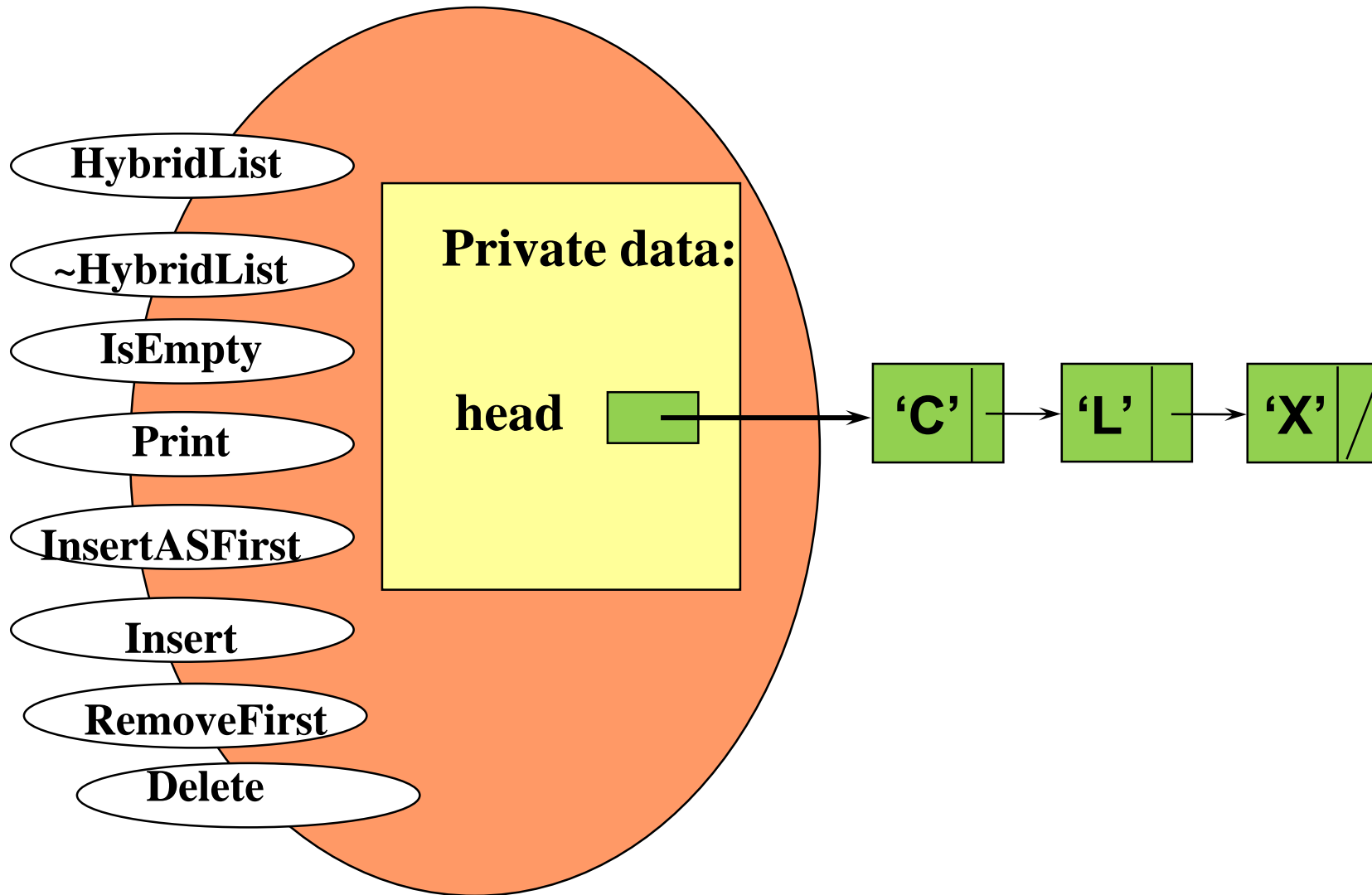
struct NodeType
{
    ItemType component; // Pointer to person's name
    NodeType* link;     // Link to next node in list
};

typedef NodeType* NodePtr;
```

```
// Specification file  hybrid sorted list("slist2.h")
class HybridList
{
public:
    bool IsEmpty () const;
    void InsertAsFirst (/* in */ ItemType item);
    void Insert (/* in */ ItemType item);
    void RemoveFirst(/* out */ ItemType& item);
    void Delete (/* in */ ItemType item);
    void Print () const;
    HybridList ();    // Constructor
    ~HybridList ();   // Destructor
    HybridList (const HybridList& otherList);
                    // Copy-constructor

private:
    NodeType* head;
};
```

# class HybridList



## // IMPLEMENTATION DYNAMIC-LINKED SORTED LIST (slist2.cpp)

HybridList :: HybridList ( )      **// Constructor**

*// Post:    head == NULL*

```
{  
    head = NULL ;  
}
```

HybridList :: ~ HybridList ( )      **// Destructor**

*// Post: All linked nodes deallocated*

```
{  
    ItemType temp ;  
  
                                     // keep deleting top node  
  
    while ( !IsEmpty ( ) )  
        RemoveFirst ( temp );  
}
```

```

void HybridList::InsertAsFirst(/* in */ ItemType item)

// Pre: item is assigned && components in ascending order
// Post: New node containing item is the first item in the list
//
//          && components in ascending order
{
    NodePtr newNodePtr = new NodeType;

    newNodePtr -> component = item;
    newNodePtr -> link = head;
    head = newNodePtr;
}

Void HybridList::Print() const

// Post: All values within nodes have been printed
{
    NodePtr currPtr = head; // Loop control pointer
    while (currPtr != NULL)
    {
        cout << currPtr->component << endl;
        currPtr = currPtr->link;
    }
}

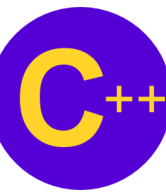
```



```
void HybridList::RemoveFirst (  
    /* out */ ItemType& item)  
  
    // Pre: list is not empty && components in ascending order  
    // Post: item == element of first list node @ entry  
    // && node containing item is no longer in list  
    // && list components in ascending order  
{  
  
    NodePtr tempPtr = head;  
  
    // Obtain item and advance head  
    item = head->component;  
    head = head->link;  
    delete tempPtr;  
  
}
```

LECTURE 4

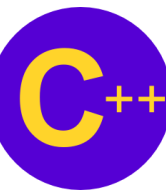
# Insertion Algorithm for Sorted List



# Insert Algorithm

---

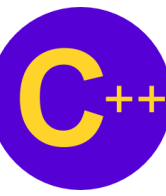
- What will be the algorithm to Insert an item into its proper place in a sorted linked list?
- That is, for a linked list whose elements are maintained in ascending order?



# Insert algorithm for HybridList

---

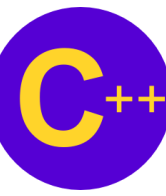
- Find proper position for the new element in the sorted list using **two pointers prevPtr and currPtr**, where prevPtr trails behind currPtr
- Obtain a new node and place item in it
- Insert the new node by adjusting pointers



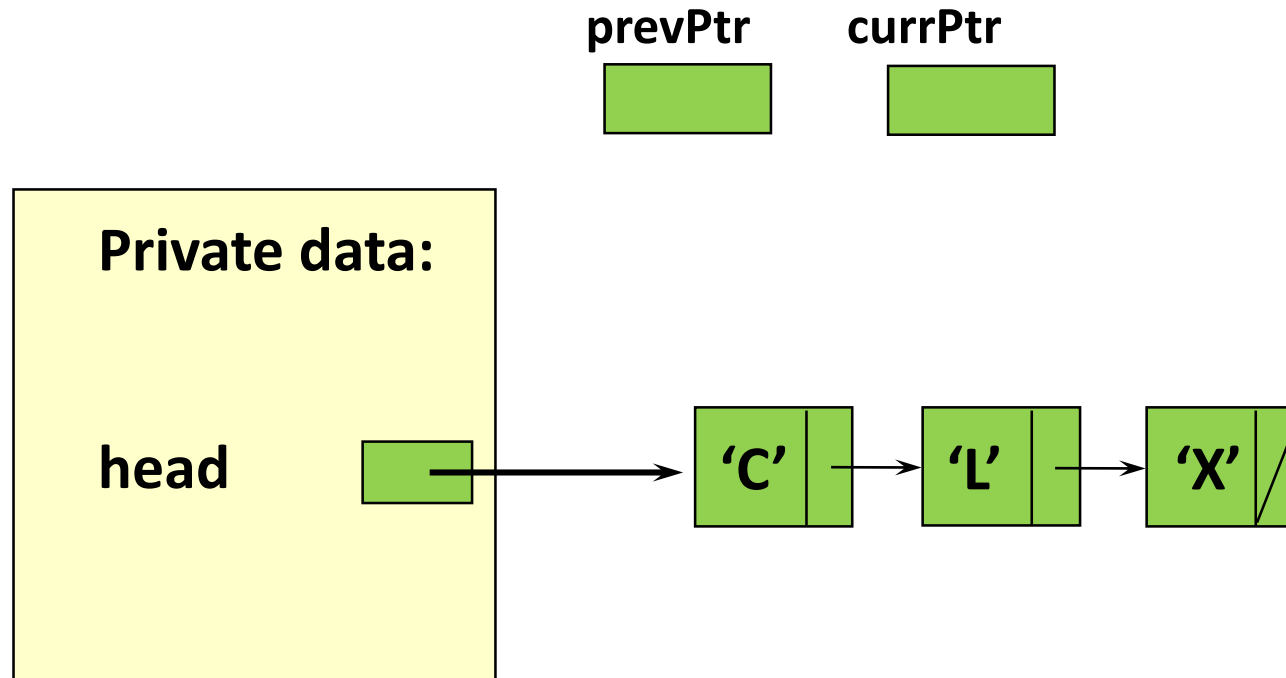
# Implementing HybridList Member Function Insert

```
// Dynamic linked list implementation ("slist2.cpp")

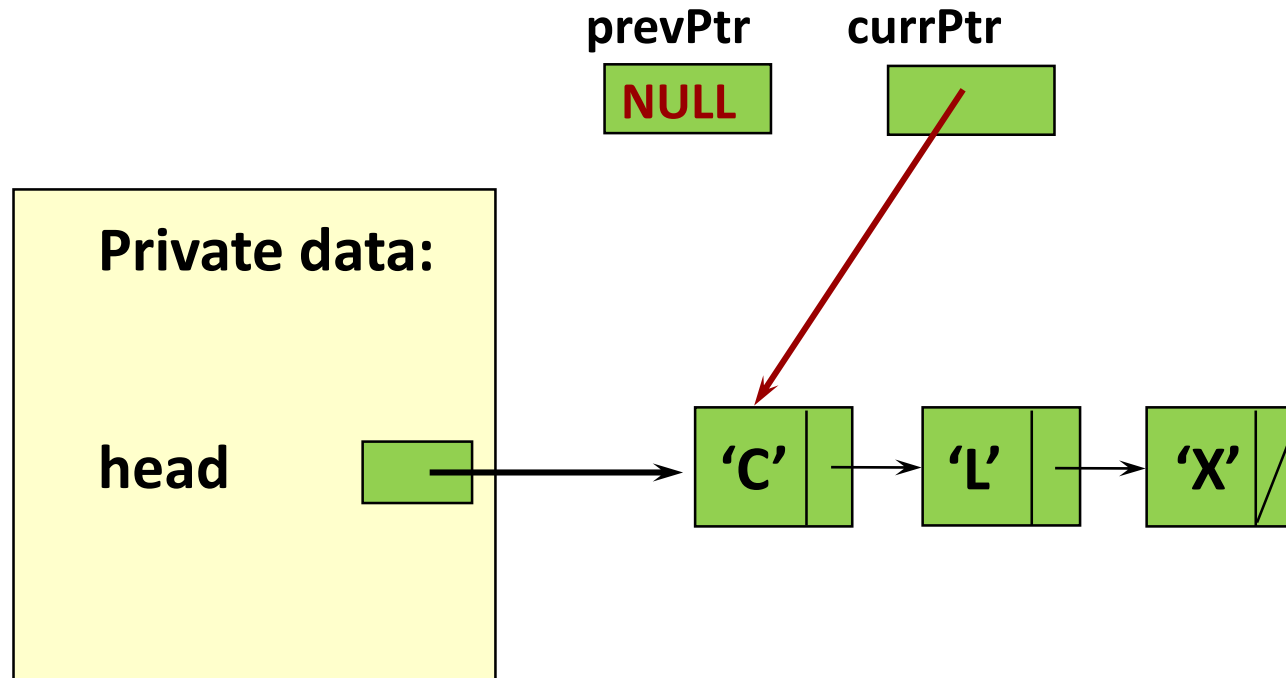
void HybridList::Insert (/* in */ ItemType item)
// PRE:
//      item is assigned && components in ascending order
// POST:
//      item is in List && components in ascending order
{
    .
    .
    .
}
```



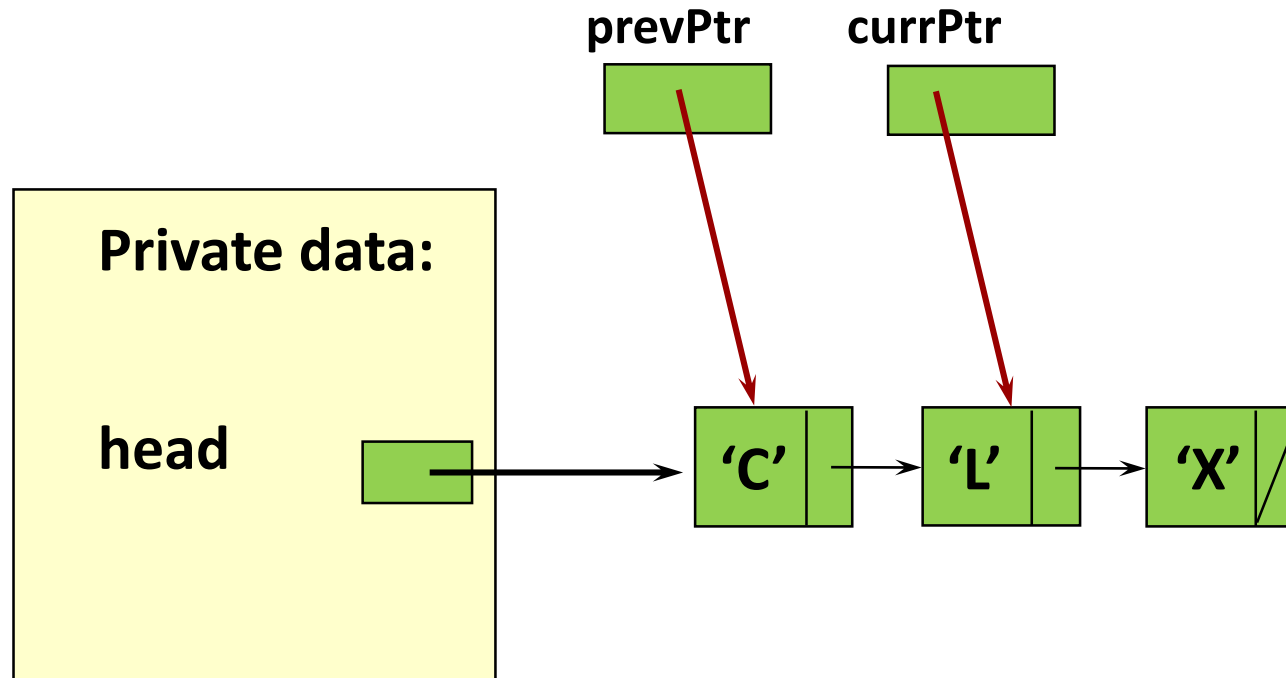
# Inserting 'S' into a List



# Finding Proper Position for 'S'

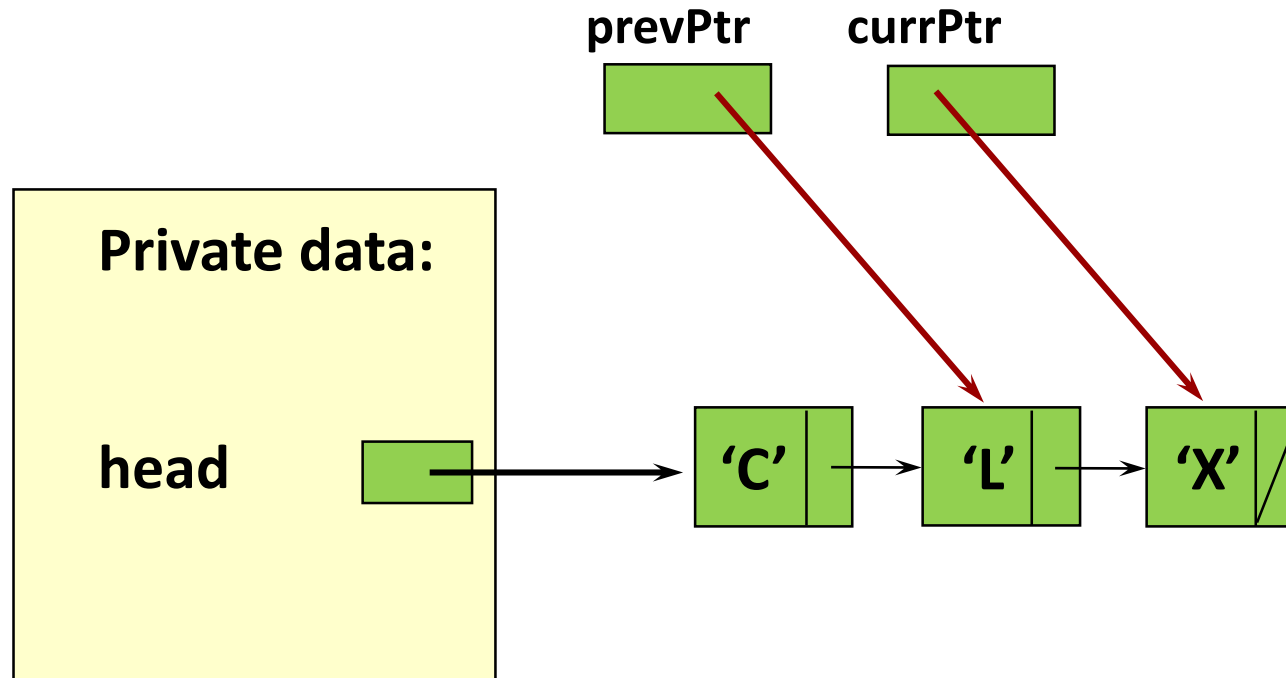


# Finding Proper Position for 'S'

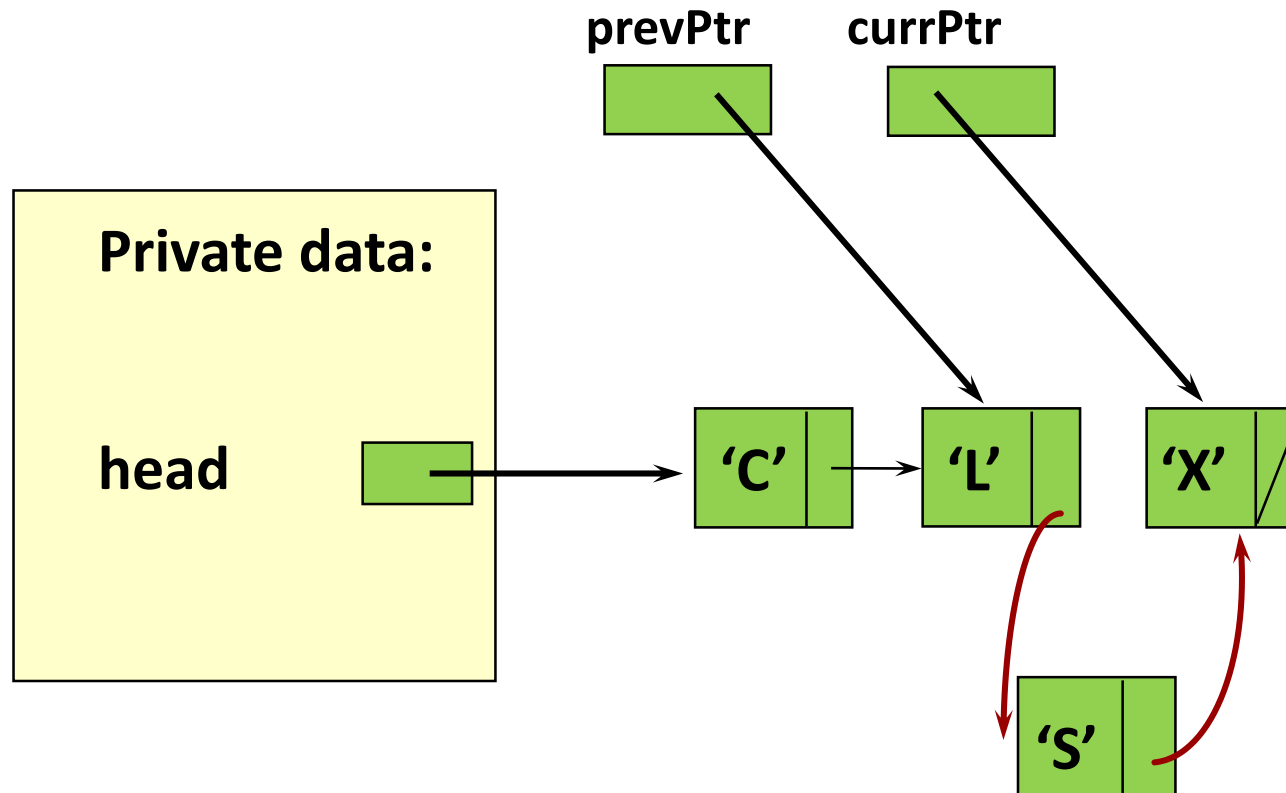




# Finding Proper Position for 'S'



# Inserting 'S' into Proper Position



```

void HybridList::Insert(/* in */ ItemType item)
// Pre: item is assigned && components in ascending order
// Post: new node containing item is in its proper place
//      && components in ascending order
{
    NodePtr currPtr;
    NodePtr prevPtr;
    NodePtr location;
    location = new NodeType;
    location->component = item;
    prevPtr = NULL;
    currPtr = head;
    while (currPtr != NULL && item > currPtr->component)
    {
        prevPtr = currPtr;           // Advance both pointers
        currPtr = currPtr->link;
    }
    location->link = currPtr; // Insert new node here
    if (prevPtr == NULL)
        head = location;
    else
        prevPtr->link = location;
}

```

```

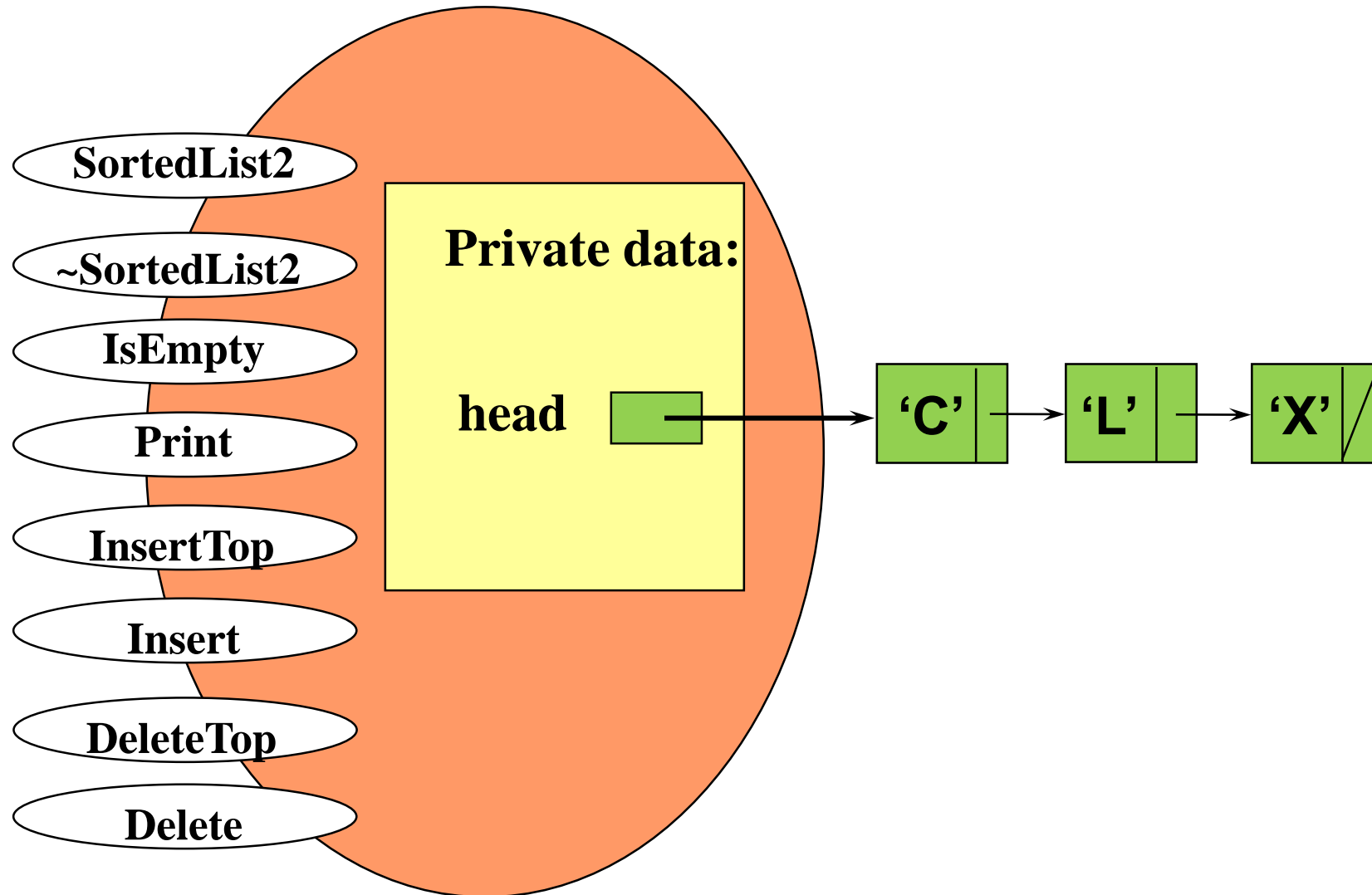
template <typename T>
void linkedlist<T>::insert(int idx, T v){
    //cout << idx << "-"<< v << endl;
    if (head == NULL){
        length++;
        add(v);
        return;
    }
    if (idx<0 || idx>length){
        throw "index out of bound";
    }

    if (idx==0) { add_front(v); return; }
    if (idx==length) { add(v); return; }

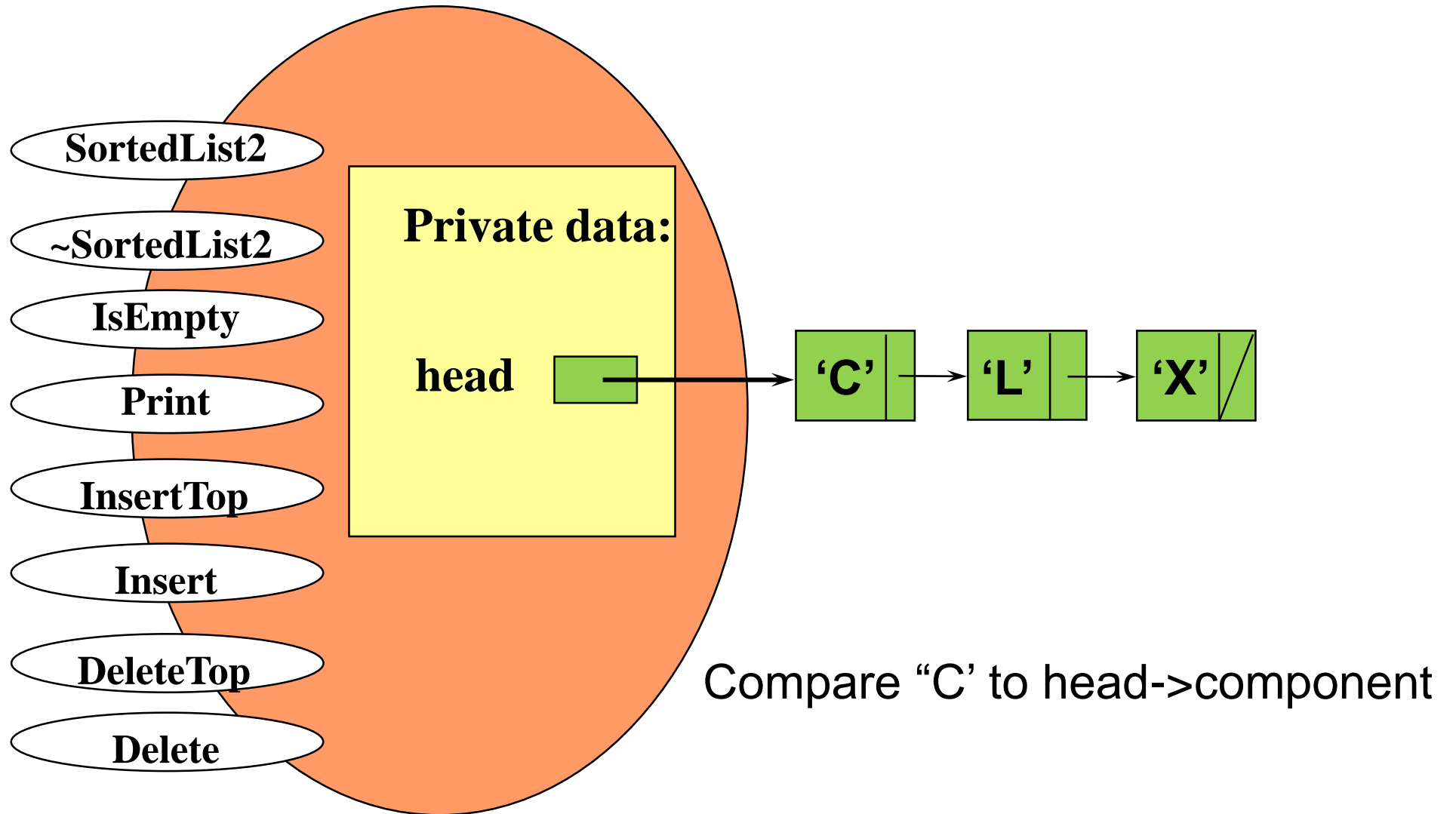
    length++;
    node<T>*p = head;
    node<T>*q = NULL;
    for (int i=0; i<=idx; i++){
        if (i==idx) {
            //cout << i << " " << q->get() << " " << p->get() << endl;
            node<T> *n = new node<T>(v);
            q->next = n;
            n->next = p;
        }
        q = p;
        p = p->next;
    }
}

```

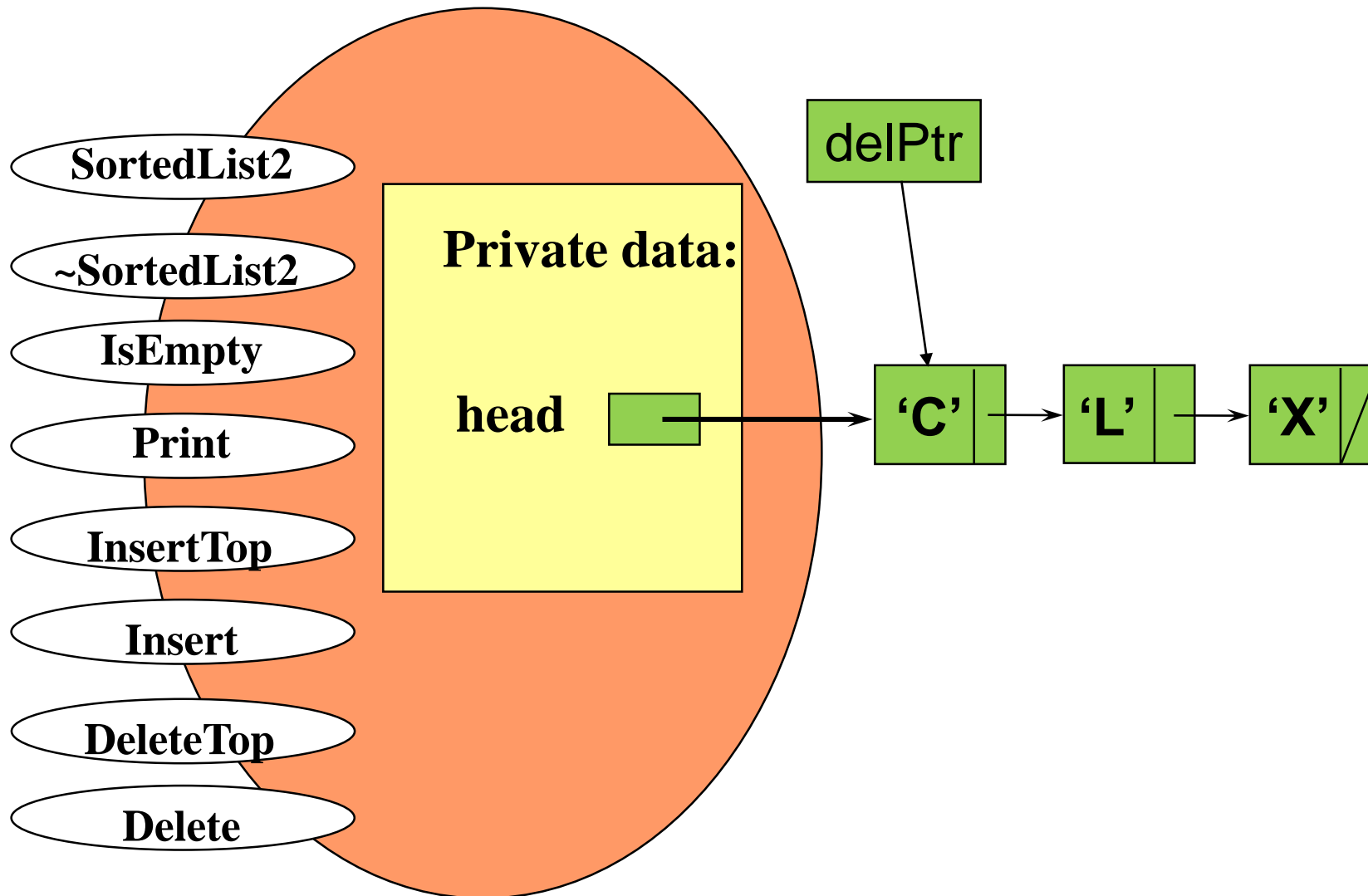
# class SortedList2



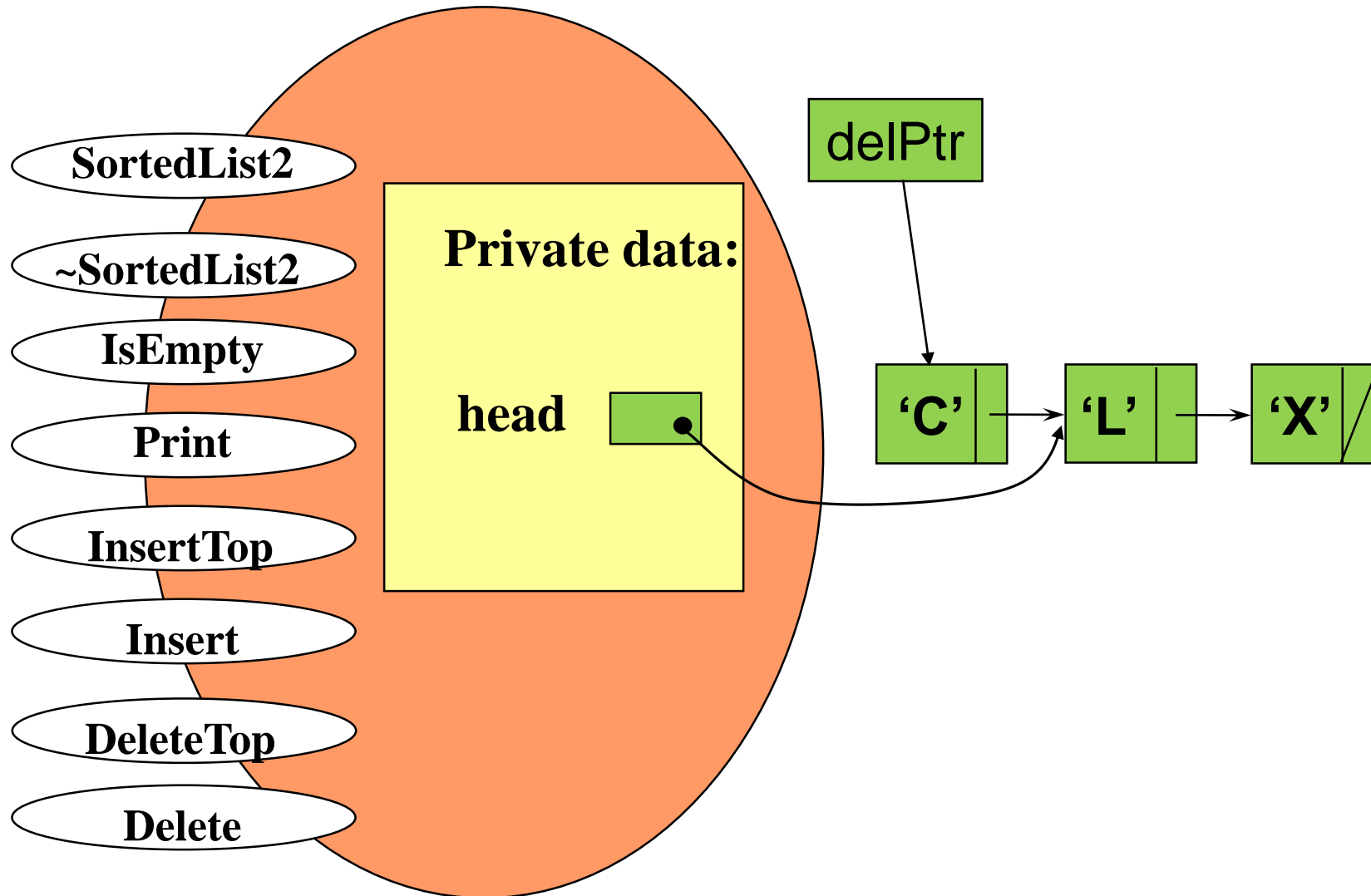
# Deleting 'C' from the List



# Deleting 'C' from the List

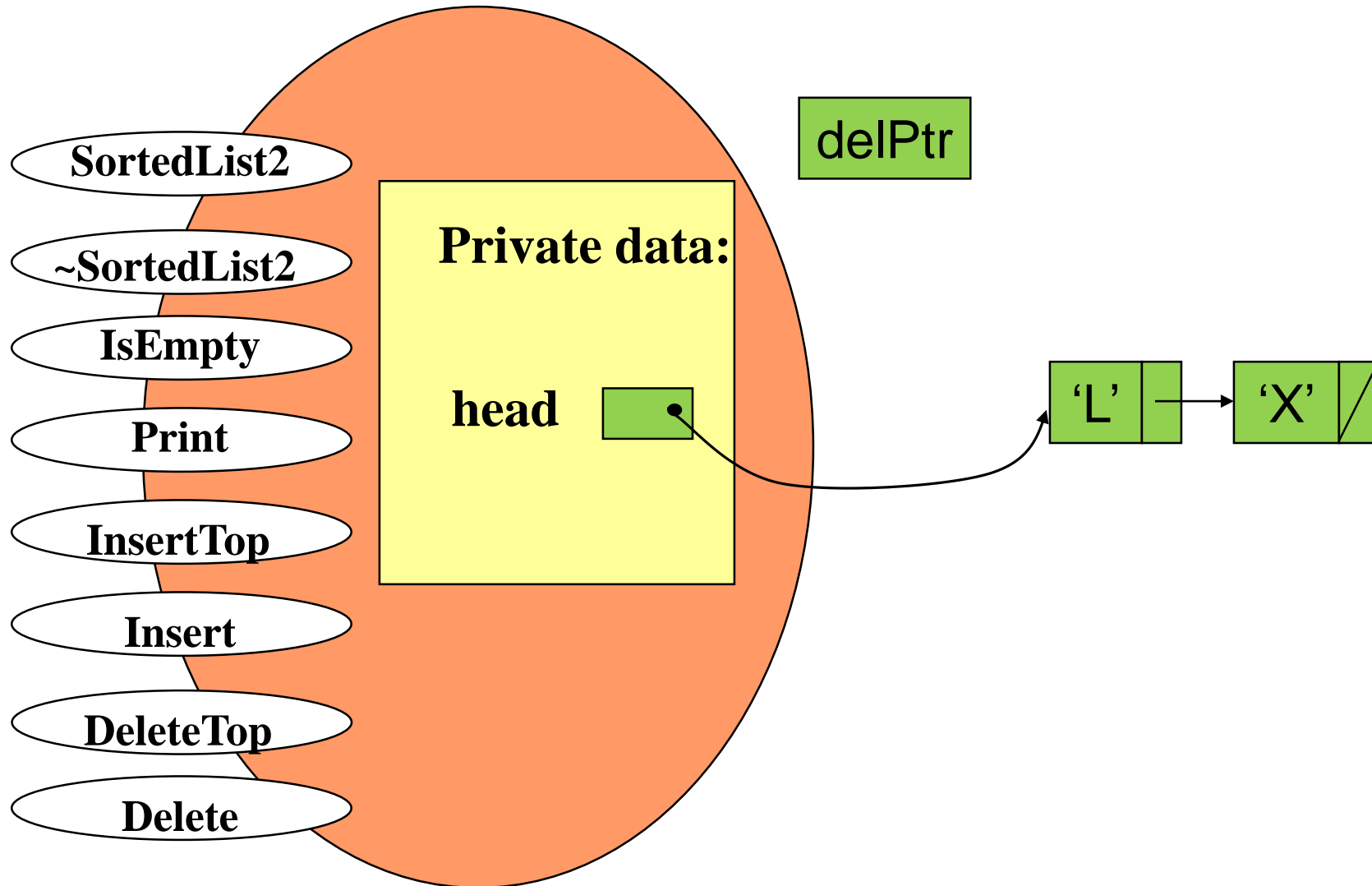


# Deleting 'C' from the List

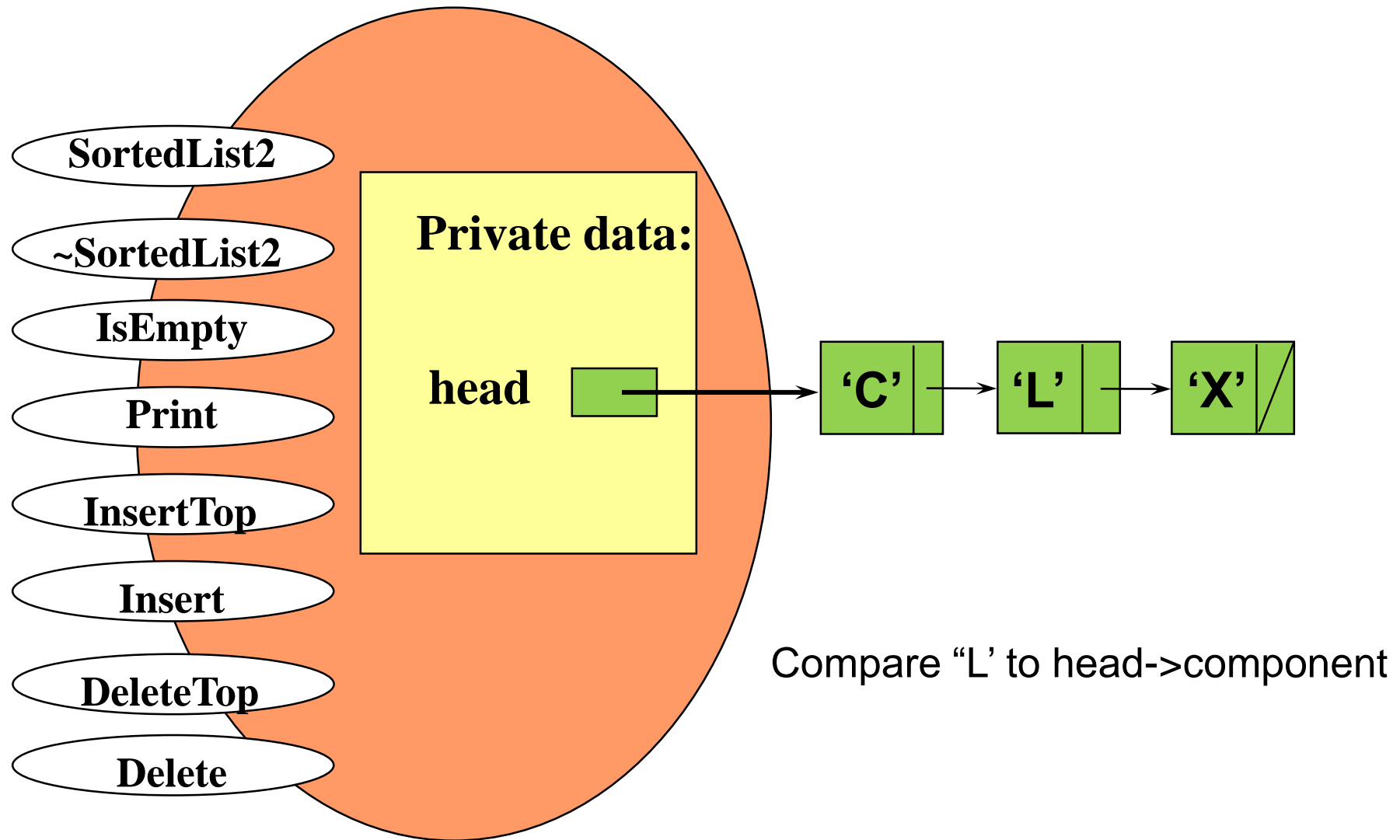




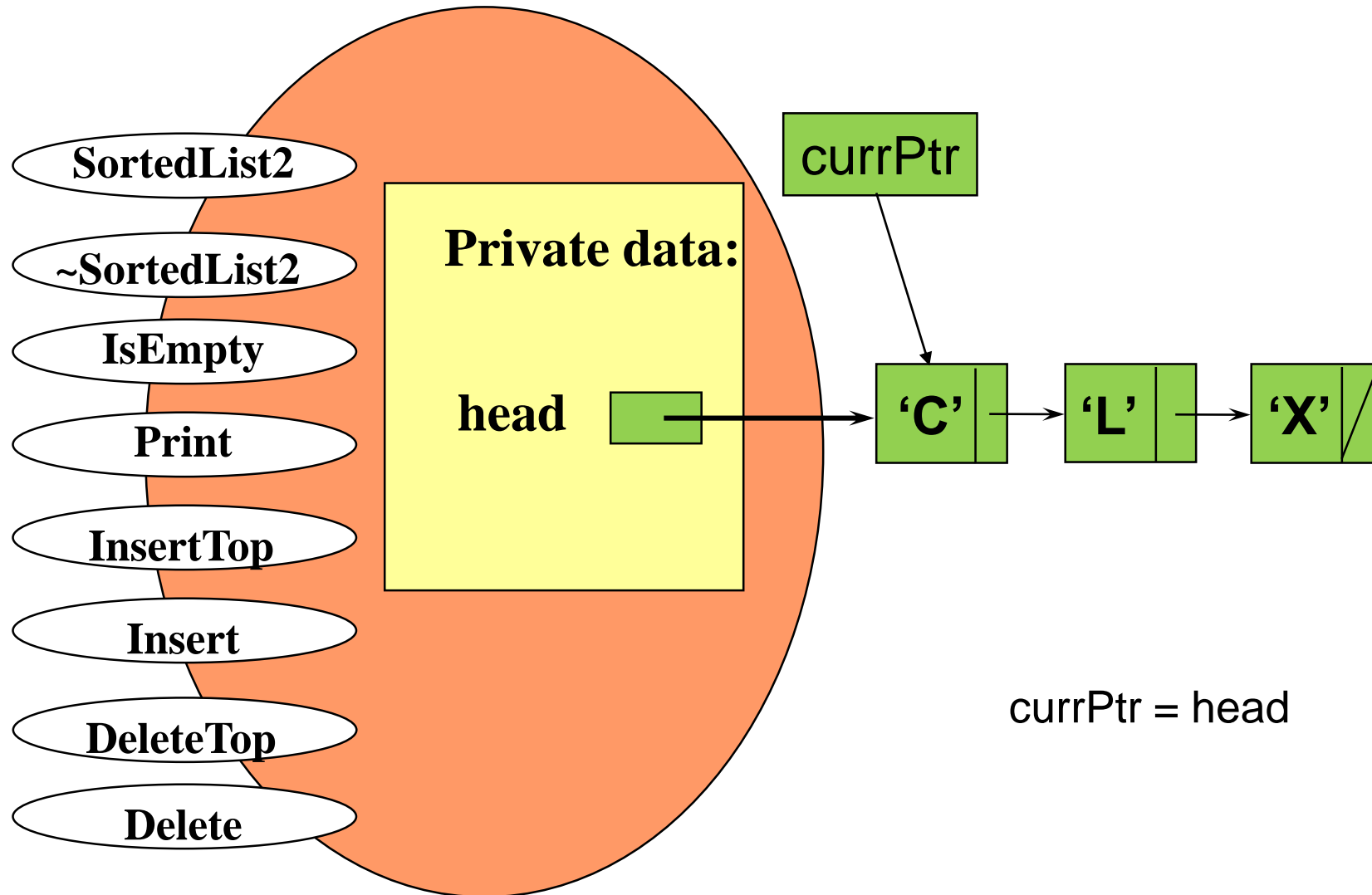
# Deleting 'C' from the List



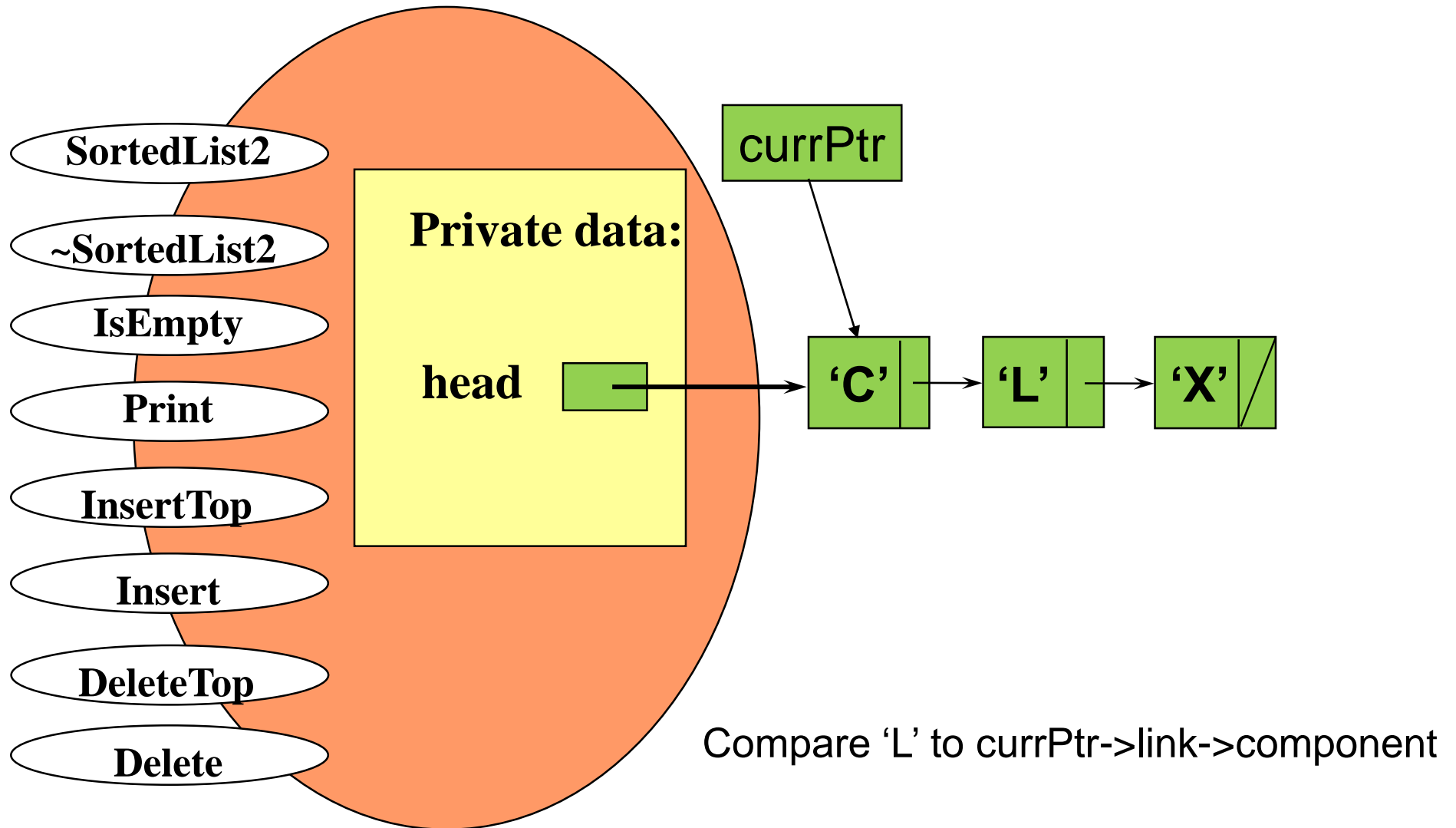
# Deleting 'L' from the List



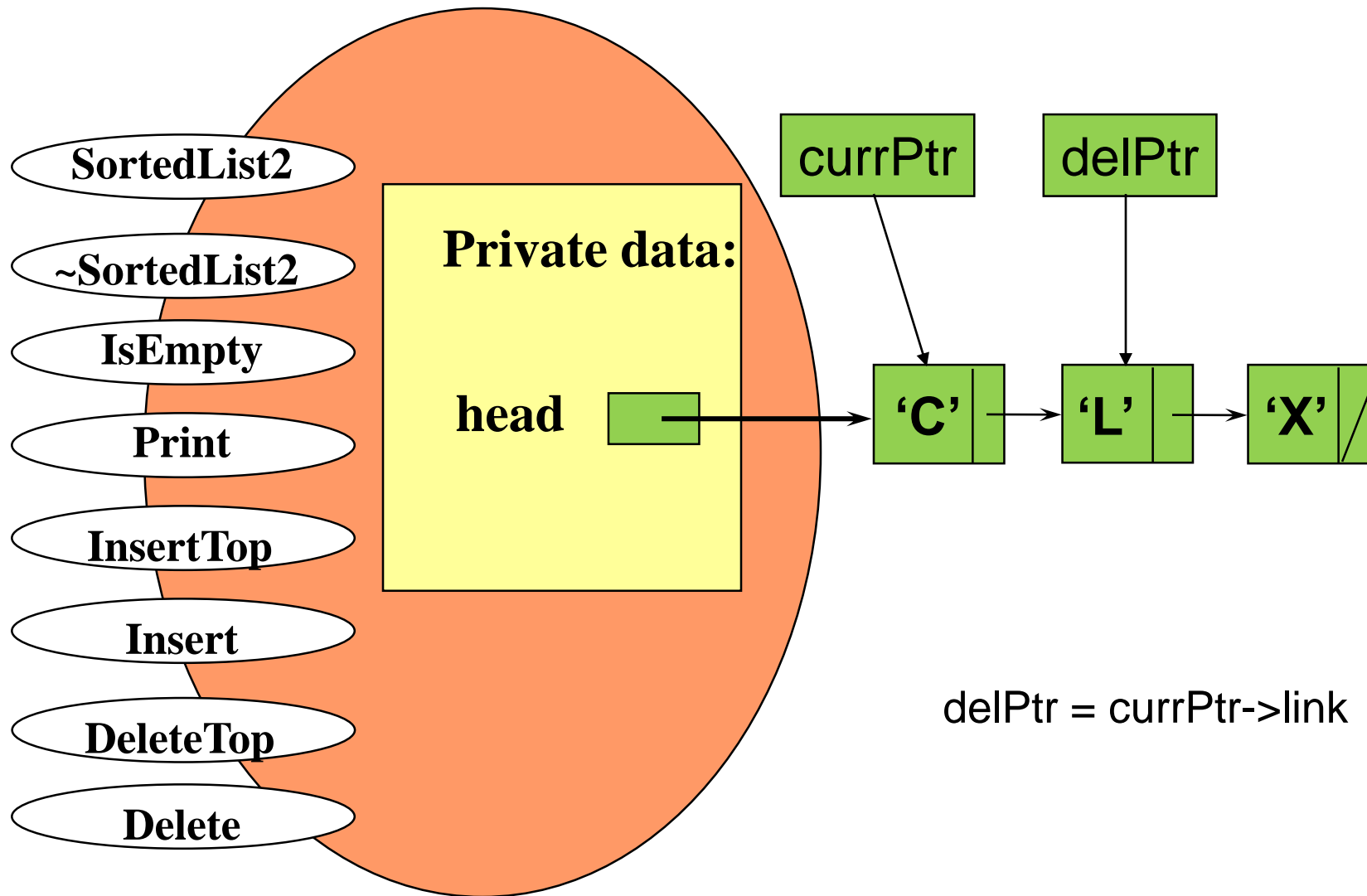
# Deleting 'L' from the List



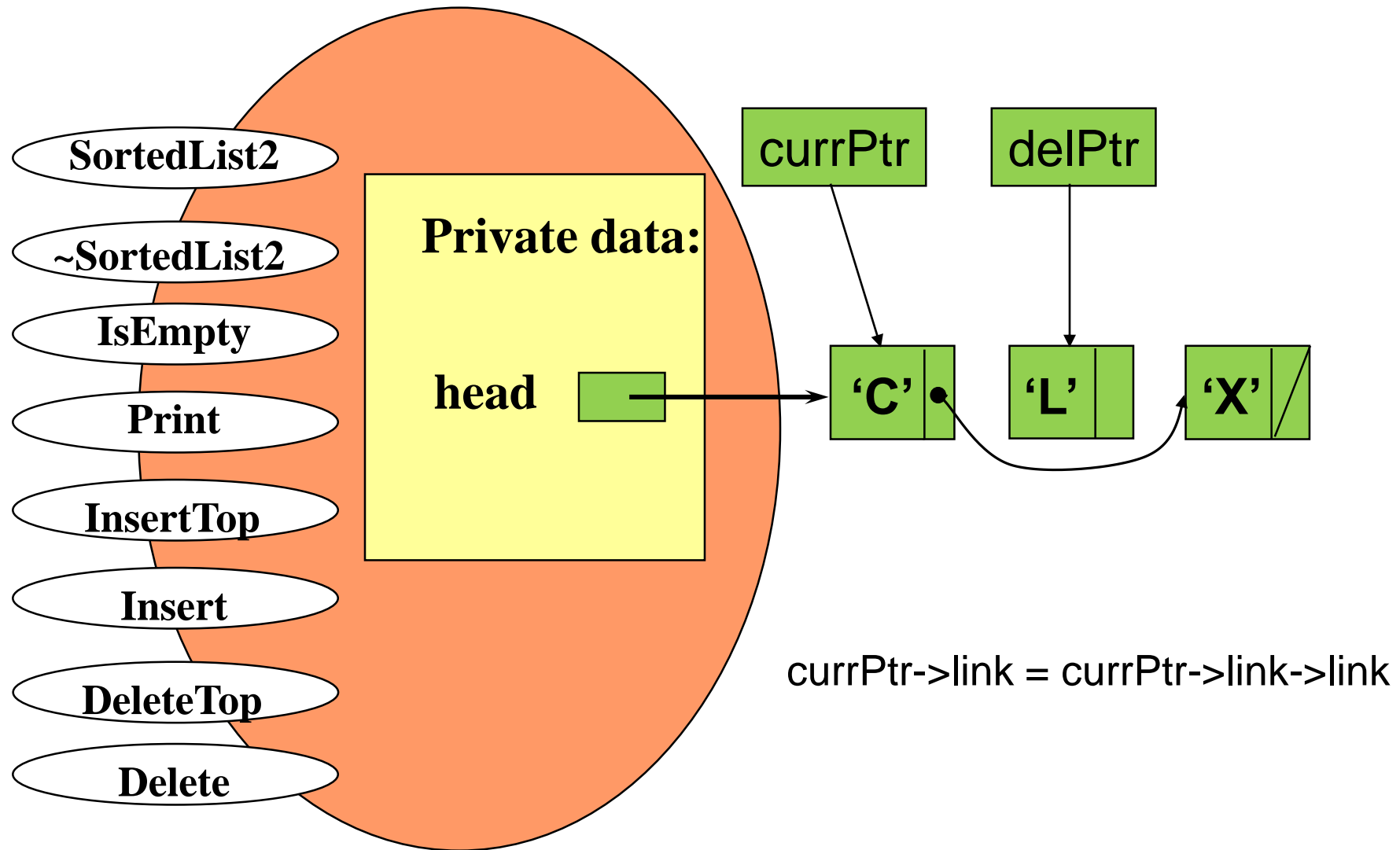
# Deleting 'L' from the List



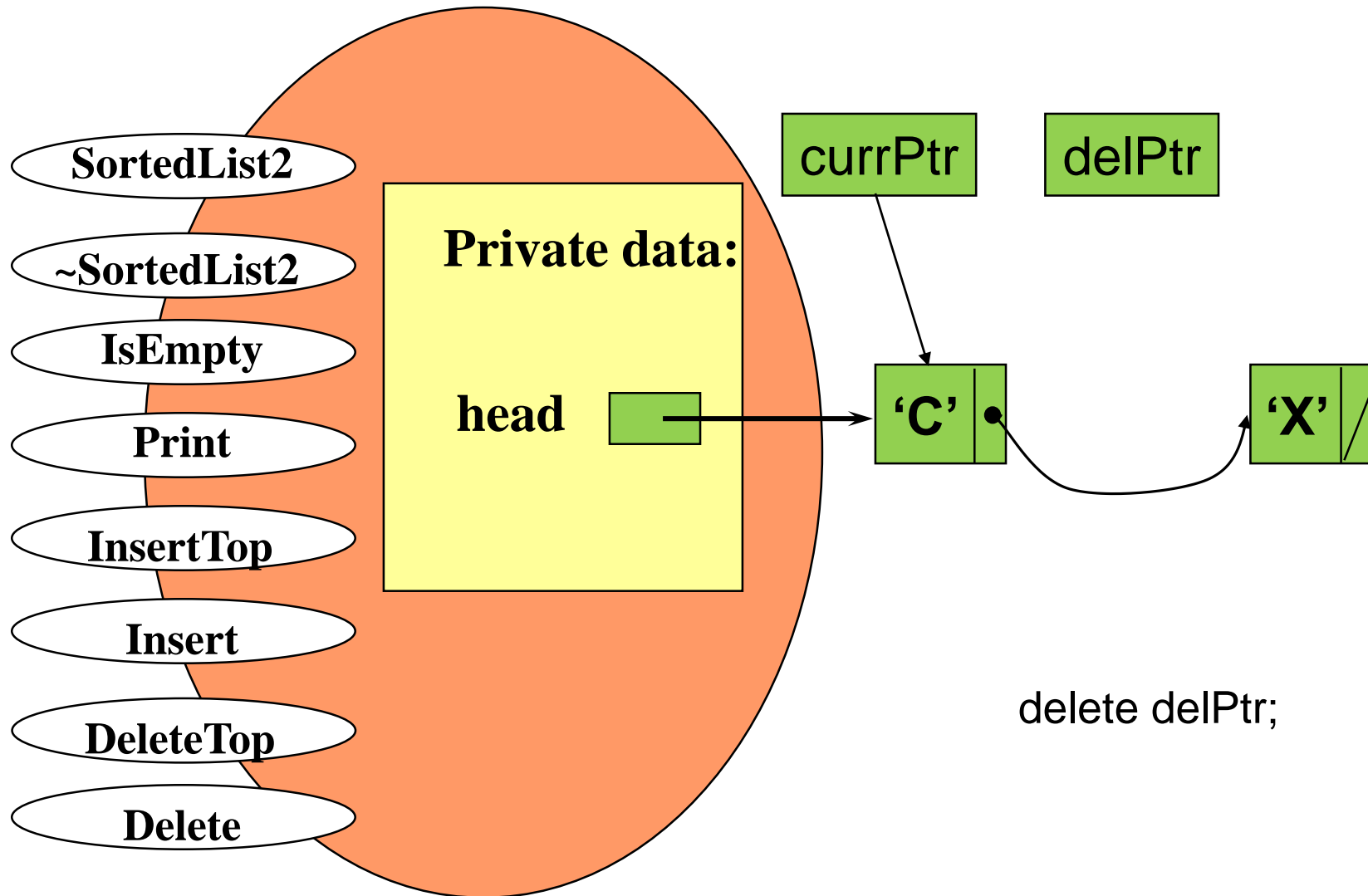
# Deleting 'L' from the List



# Deleting 'L' from the List



# Deleting 'L' from the List



```

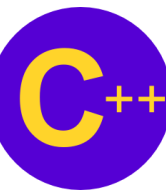
void HybridList::Delete (/* in */ ItemType item)
// Pre: list is not empty && components in ascending order
//           && item == component member of some list node
// Post: item == element of first list node @ entry
//           && node containing first occurrence of item no longer
//           in list   && components in ascending order
{
    NodePtr    delPtr;
    NodePtr    currPtr; // Is item in first node?
    if (item == head->component)
    { // If so, delete first node
        delPtr = head;
        head = head->link;
    }
    else { // Search for item in rest of list
        {
            currPtr = head;
            while (currPtr->link->component != item)
                currPtr = currPtr->link;
            delPtr = currPtr->link;
            currPtr->link = currPtr->link->link;
        }
        delete delPtr;
    }
}

```



## LECTURE 4

# Copy Constructor for Hybrid List



# Copy Constructor

---

- Most difficult algorithm so far
  - If the original is empty, the copy is empty
  - Otherwise, make a copy of the head with pointer to it
  - Loop through original, copying each node and adding it to the copy until you reach the end

**// IMPLEMENTATION DYNAMIC-LINKED SORTED LIST (slist2.cpp)**

HybridList :: HybridList ( const HybridList & otherList ) ;

**// Copy Constructor**

*// Pre: otherList is assigned*

*// Post: create a deep copy of the otherList*

```
{
    if (otherList.head == NULL)
        head = NULL ;
    else
    {
        NodePtr otherPtr = otherList.head, thisPtr;
        head = new NodeType;
        head -> component = otherPtr -> component;
        thisPtr = head;
        otherPtr = otherPtr -> link;
        while (otherPtr != NULL)
        {
            NodePtr tempPtr = new NodeType;
            tempPtr -> component = otherPtr -> component;
            thisPtr -> link = tempPtr;
            thisPtr = tempPtr;
            otherPtr = otherPtr -> link;
        }
        thisPtr -> link = NULL;
    }
}
```

```
template <typename T>
void linkedlist<T>::append(linkedlist<T> *alist){
    if (head == NULL) {
        head = alist->head;
        tail  = alist->tail;
        return;
    }

    tail->next = alist->head;
    tail = alist->tail;
}
```

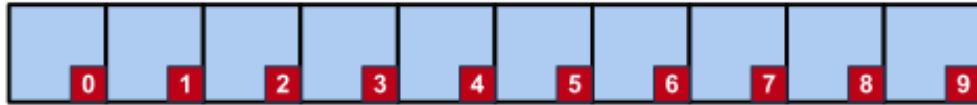
LECTURE 4

# Linked Structure for Data Structure

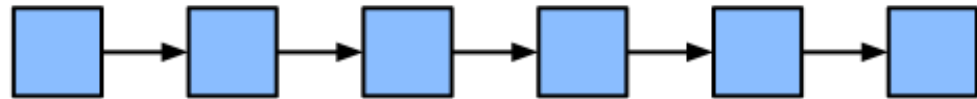
# Linked-List, Array and Heap-Tree

## Array & Linked List

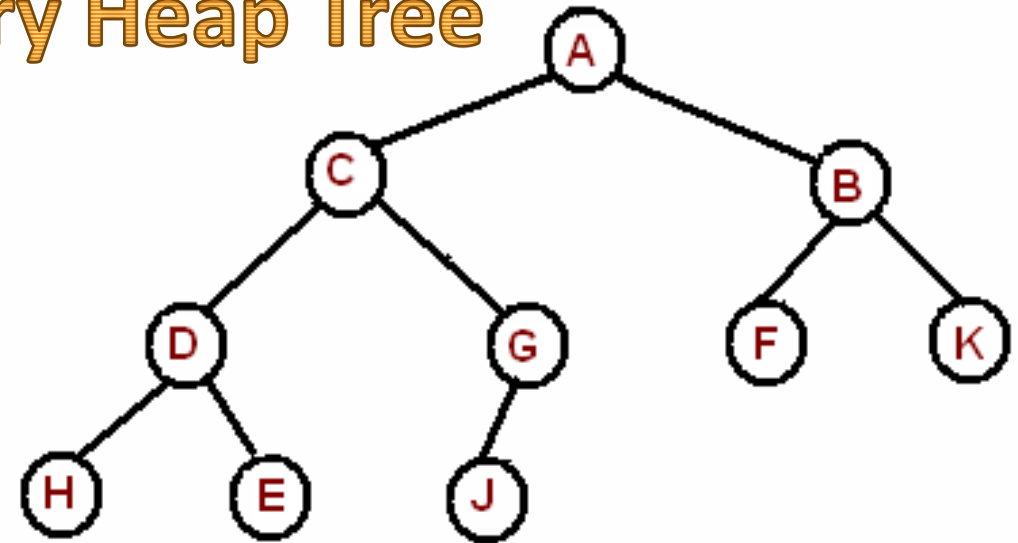
Access  $A[k]$  in  $O(1)$  time!



Access  $L[k]$  in  $O(n)$  time!



## Binary Heap Tree



0	1	2	3	4	5	6	7	8	9	10
	A	C	B	D	G	F	K	H	E	J

Note: Array, Linked List and Binary Heap Tree are used to build all data collections

# Data Structures

## **Basic Abstract Data Type:**

- Array
- Linked List
- Binary Tree (Iterable Heap Tree)

Note: In this course, we covered these topics.

## **Advanced Abstract Data Type:**

- Hashing
- Graph
- Matrix
- Misc
- Advanced Data Structure

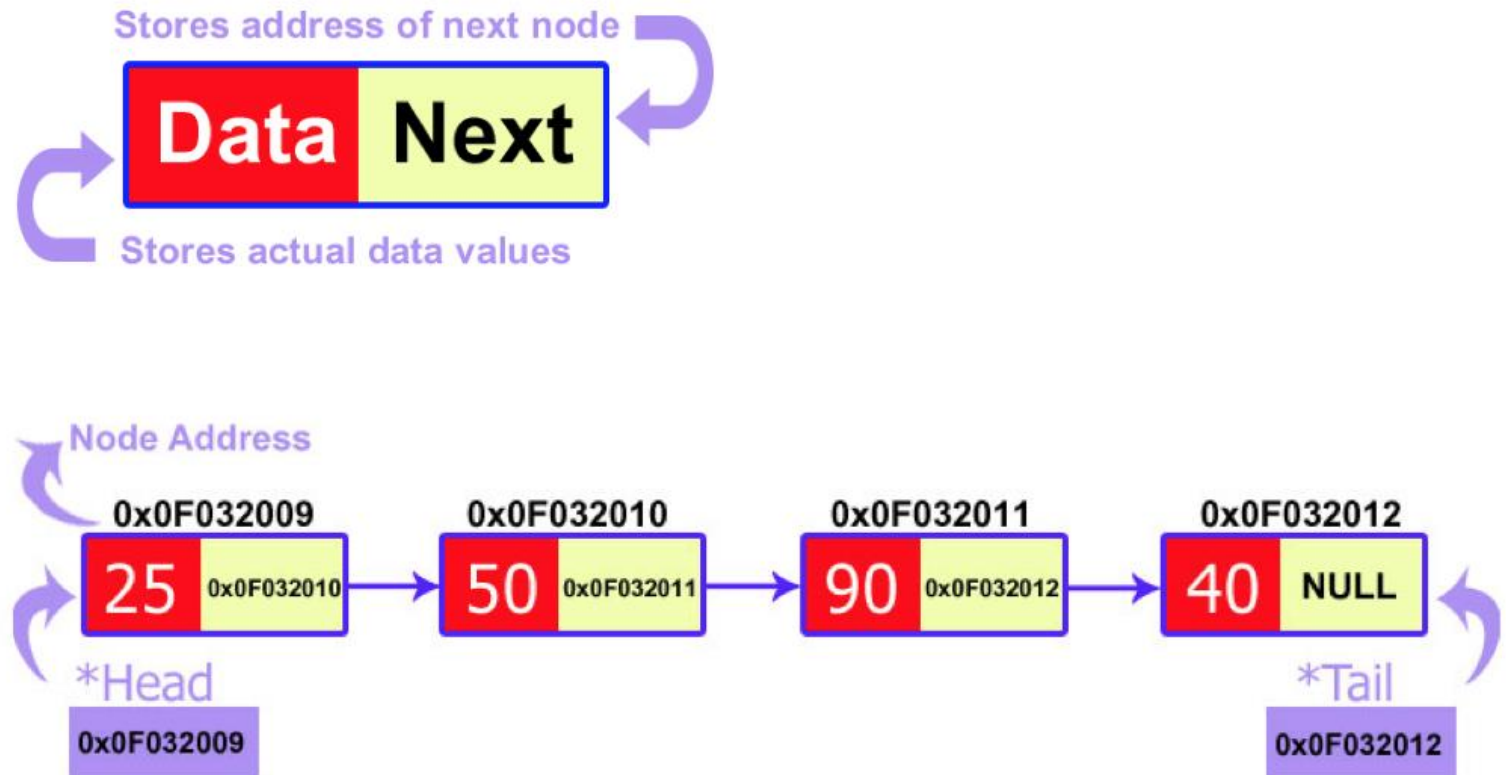
## **Abstract Data Type (Data Collections):**

- Stack
- Queue
- Priority Queue
- Binary Tree
- Binary Search Tree
- Heap
- Set
- Map

## linkedList<T>

+length: int;  
+head: node<T>  
+tail: node<T> \*

+size(void): int  
+isEmpty(): bool  
+indexOf(T obj): int  
+get(int idx): T  
+to\_string(): string  
+set(int idx, T v): void  
+add(T v): void  
+add\_front(T v): void  
+insert(int idx, T v): void  
+remove(): node<T> \*  
+remove\_front(): node<T> \*  
+append(linkedList<T> \*alist): void





```

template<typename T>
int linkedlist<T>::size(){ // list.length
    return length;
}

template<typename T>
bool linkedlist<T>::isempty(){
    return (length==0);
}

template <typename T>
int linkedlist<T>::indexOf(T obj){
    node<T> *p = head;

    int i = 0;
    int idx = -1;
    bool found = false;

    while (p != NULL && !found){
        if (p->get() == obj){
            idx = i;
        }
        i++;
        p = p->next;
    }
    return idx;
}

```

```

T linkedlist<T>::get(int idx){
    T rtn;
    node<T>*p = head;
    if (head == NULL){
        throw "empty list";
    }
    if (idx<0 || idx>=length){
        throw "index out of bound";
    }
    for (int i=0; i<=idx; i++){
        if (i==idx) rtn = p->get();
        p = p->next;
    }
    return rtn;
}

template <typename T>
void linkedlist<T>::set(int idx, T v){
    node<T>*p = head;
    if (head == NULL){
        throw "empty list";
    }
    if (idx<0 || idx>=length){
        throw "index out of bound";
    }
    for (int i=0; i<=idx; i++){
        if (i==idx) p->set(v);
        p = p->next;
    }
}

```

```

template <typename T>
void linkedlist<T>::add(T v){
    length++;
    node<T> *n = new node<T>(v);
    if (head == NULL){
        head = n;
        head->next = NULL;
        tail = n;
        return;
    }

    node<T> *p = (node<T> *) head;
    node<T> *q = NULL;
    while (p!= NULL){
        q = p;
        p=p->next;
    }
    tail = n;
    q->next = n;
    n->next = NULL;
}

```

```

template <typename T>
void linkedlist<T>::add_front(T v){
    length++;
    node<T> *n = new node<T>(v);
    if (head == NULL){
        head = n;
        head->next = NULL;
        tail = n;
        return;
    }

    node<T> *p = (node<T> *) head;
    head = n;
    n->next = p;
}

```

```

template <typename T>
void linkedlist<T>::insert(int idx, T v){
    //cout << idx << "-" << v << endl;
    if (head == NULL){
        length++;
        add(v);
        return;
    }
    if (idx<0 || idx>length){
        throw "index out of bound";
    }

    if (idx==0) { add_front(v); return; }
    if (idx==length) { add(v); return; }

    length++;
    node<T> *p = head;
    node<T> *q = NULL;
    for (int i=0; i<=idx; i++){
        if (i==idx) {
            //cout << i << " " << q->get() << " " << p->get() << endl;
            node<T> *n = new node<T>(v);
            q->next = n;
            n->next = p;
        }
        q = p;
        p = p->next;
    }
}

```

```

template <typename T>
node<T>* linkedlist<T>::remove(){
    if (head == NULL){ // zero element
        return NULL;
    }

    length--;
    node<T> *p = (node<T> *) head;
    node<T> *q = NULL;
    node<T> *r = NULL;
    while (p!= NULL){
        r = q;
        q = p;
        p=p->next;
    }

    if (r==NULL){ // only one element
        head = NULL;
        tail = NULL;
        return q;
    }

    r->next = NULL;
    tail = r;
    return q;
}

```

```

template <typename T>
node<T>* linkedlist<T>::remove_front(){
    if (head == NULL){ // zero element
        return NULL;
    }

    length--;
    if (head->next == NULL){
        node<T>* q = head;
        head = NULL;
        tail = NULL;
        return q;
    }

    node<T>* q = head;
    head = head->next;
    return q;
}

```

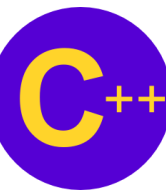
```

template <typename T>
string linkedlist<T>::to_string(){
    string str("");
    node<T> *p = head;
    str += "[";
    int count = 0;
    while (p!= NULL){
        if (count == 0) str += st::to_string(head->get());
        else str += ", " + st::to_string(p->get());
        count++;
        p=p->next;
    }
    str += "]";
    return str;
}

template <typename T>
void linkedlist<T>::append(linkedlist<T> *alist){
    if (head == NULL) {
        head = alist->head;
        tail = alist->tail;
        return;
    }

    tail->next = alist->head;
    tail = alist->tail;
}

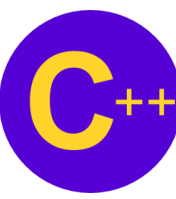
```



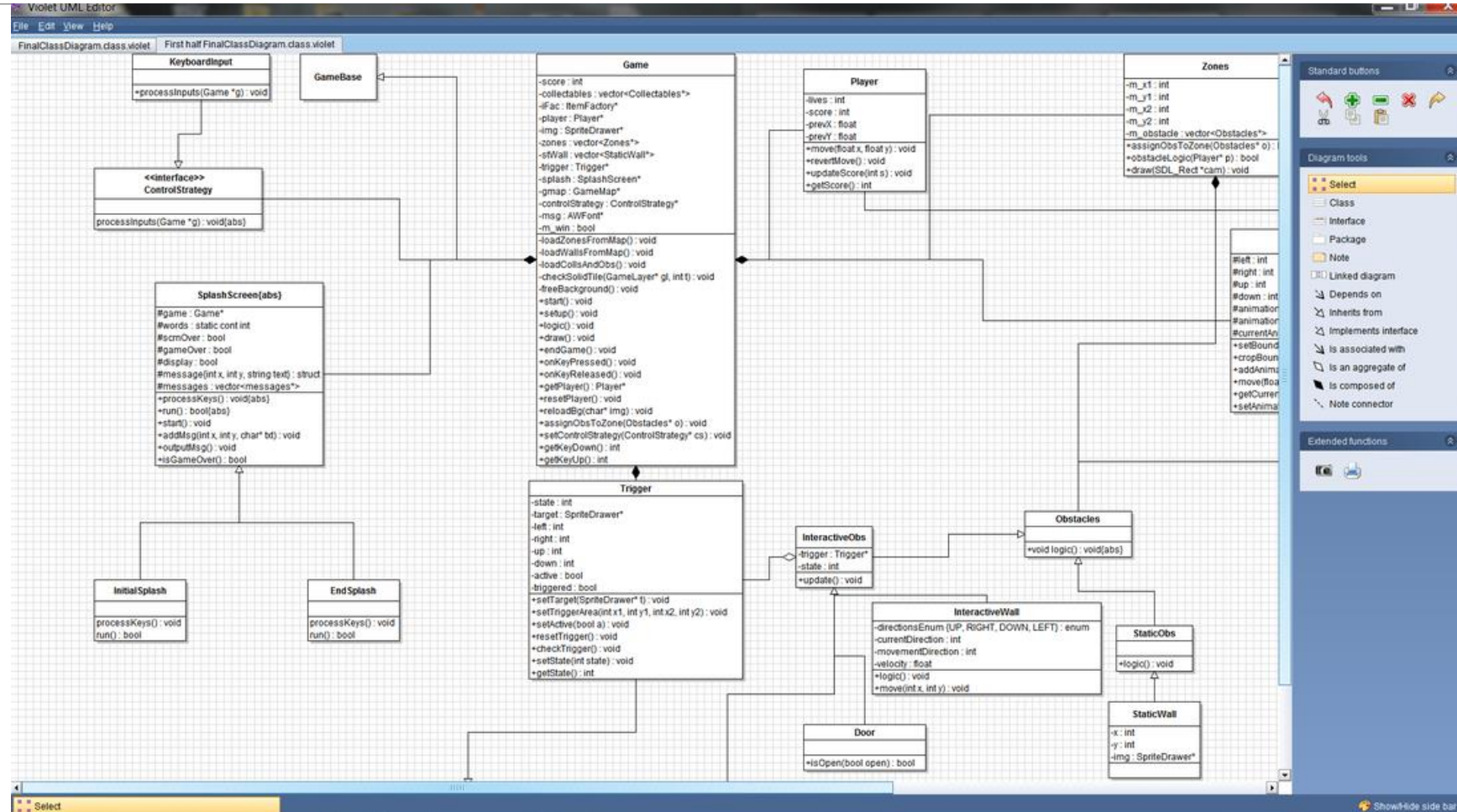
# Demo Program: linkedlist.cpp

---

## Go Notepad++!!!



# Demo Program: [linkedList.class.violet.html](#)





# Violet UML Editor

Get ready for  
Violet 2.0.0!

home

demo

- Home
- What is UML?
- Quick tour
- Features
- Wish List
- Demo
- Download
- Documentation
  - Installation
  - User guide
  - Developer guide
  - Books
- Authors
- Contact us

Violet is a UML editor with these benefits:  
Very easy to learn and use. Draws nice-looking diagrams. Completely free.  
Cross-platform. Violet is intended for developers, students, teachers, and  
authors who need to produce simple UML diagrams quickly



Download now



Run it now!