Linked Structures

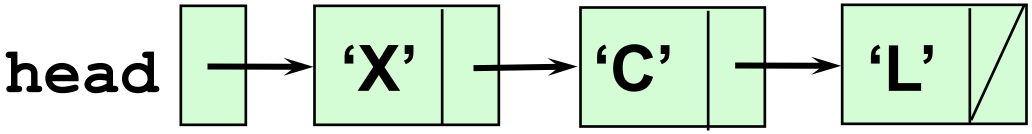
Primer

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

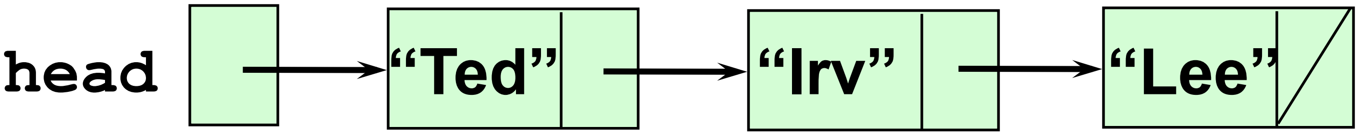
A Linked List

- 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



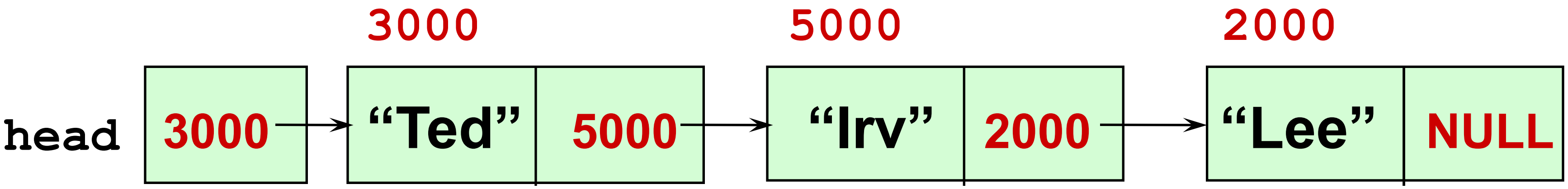
Dynamic Linked List

- 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



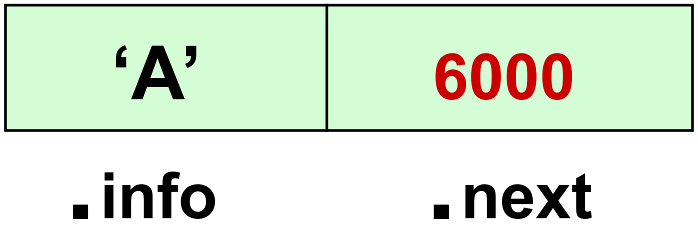
Declarations for a Dynamic Linked List

```
// Type declarations

struct NodeType
{
     char info;
     NodeType* next;
}

typedef  NodeType*  NodePtr;

// Variable DECLARATIONS
NodePtr  head;
NodePtr  ptr;
```



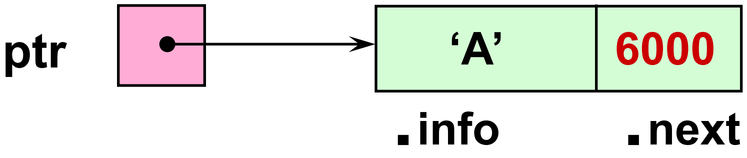
‘A’

6000

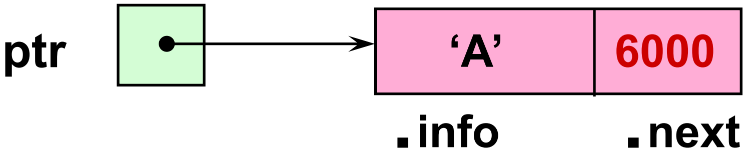
.info

.next

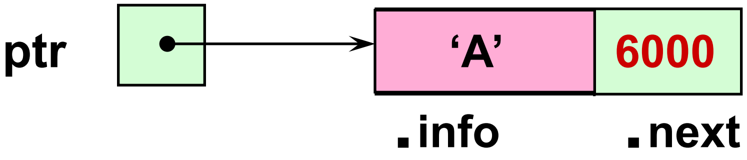
Pointer Dereferencing and Member Selection



ptr



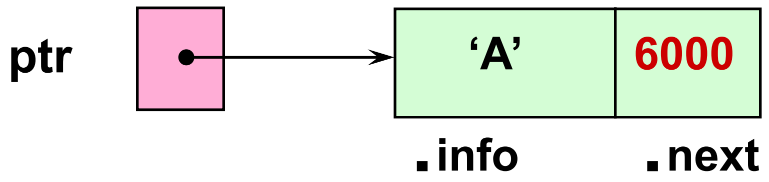
*ptr



(*ptr).info

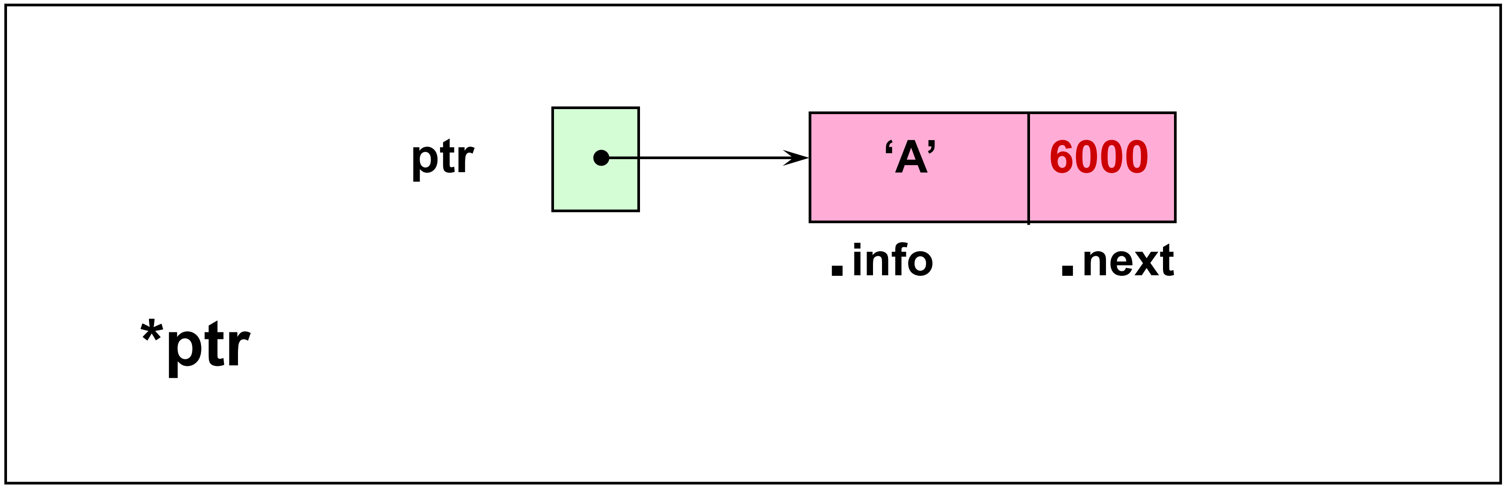
ptr->info

`ptr` is a pointer to a node

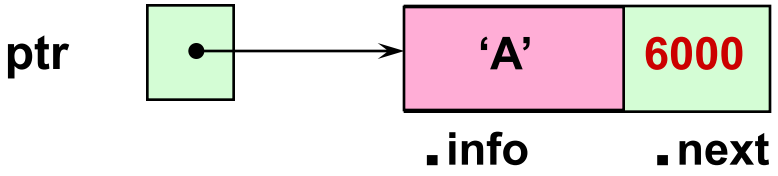


ptr

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



`ptr->info`
is a node member



ptr

'A'

6000

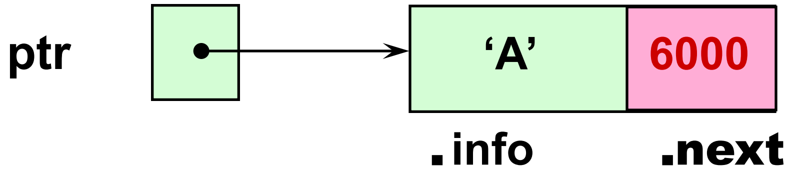
.info

.next

```
ptr->info

(*ptr).info            // Equivalent
```

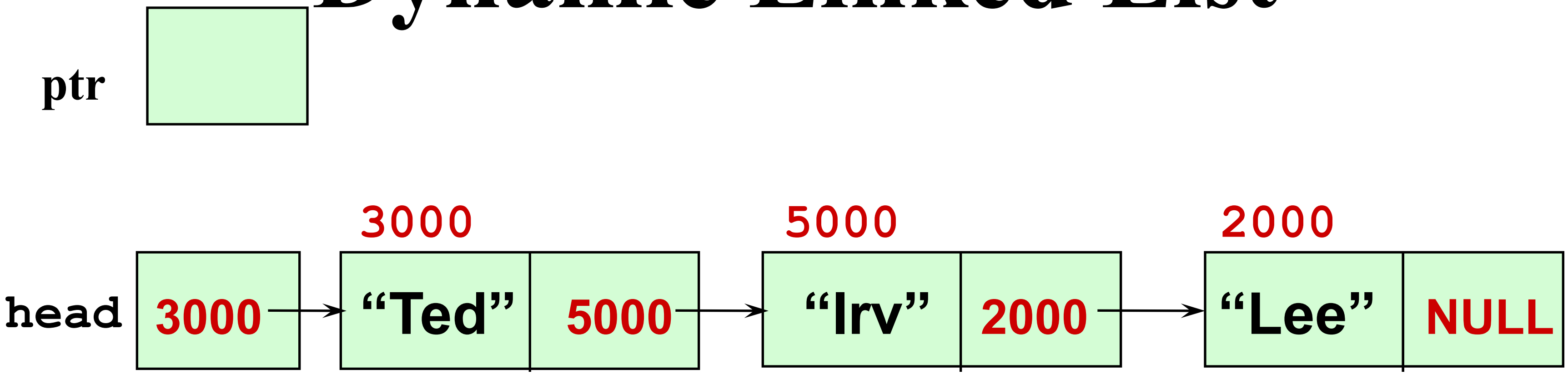
`ptr->link`
is a node member



ptr->next

(*ptr).next // Equivalent

Traversing a Dynamic Linked List



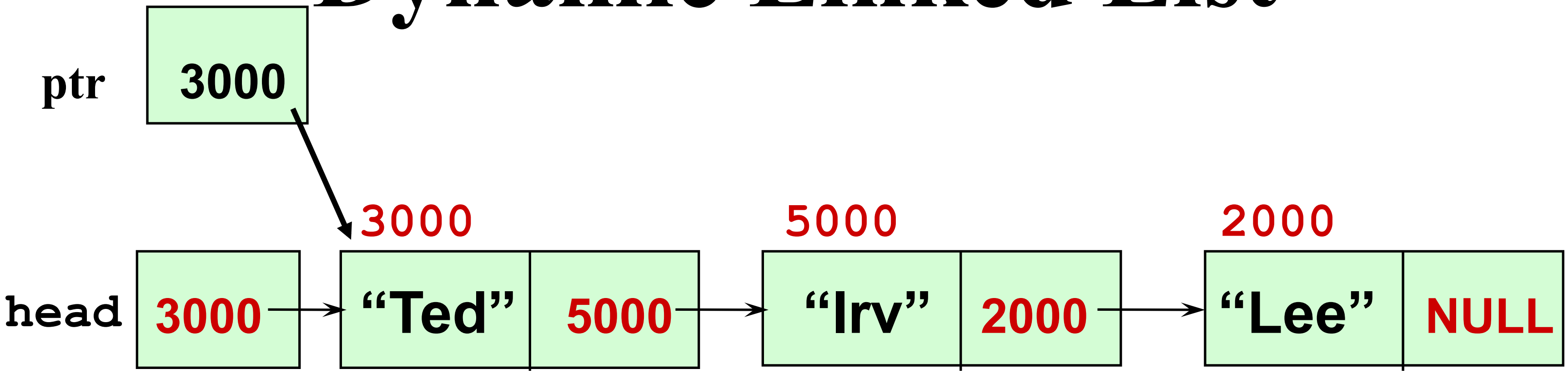
ptr

3000 5000 2000

head 3000 → "Ted" 5000 → "Irv" 2000 → "Lee" NULL

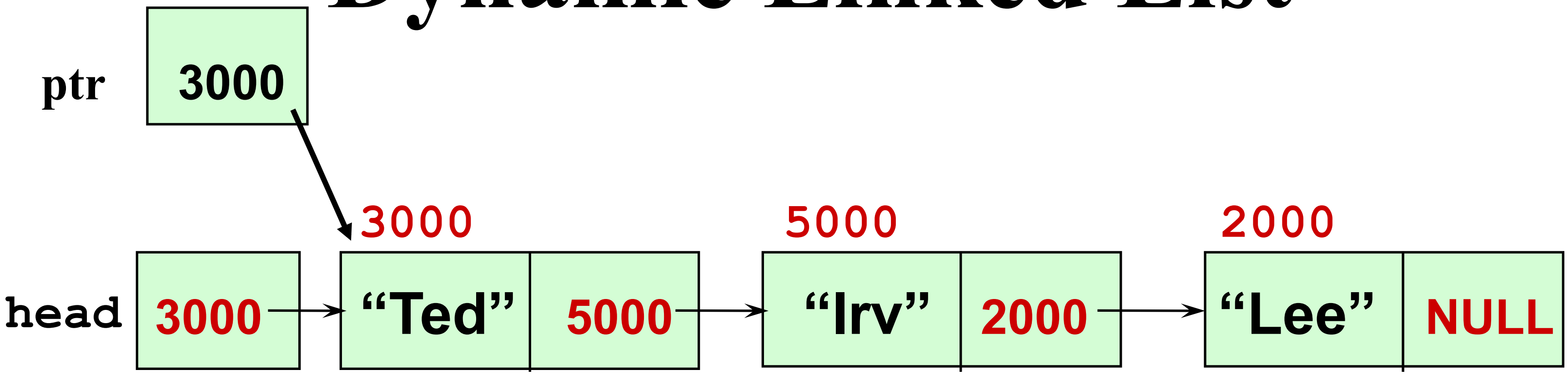
```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
    // Or, do something else with node *ptr
     ptr  =  ptr->next;
}
```

Traversing a Dynamic Linked List



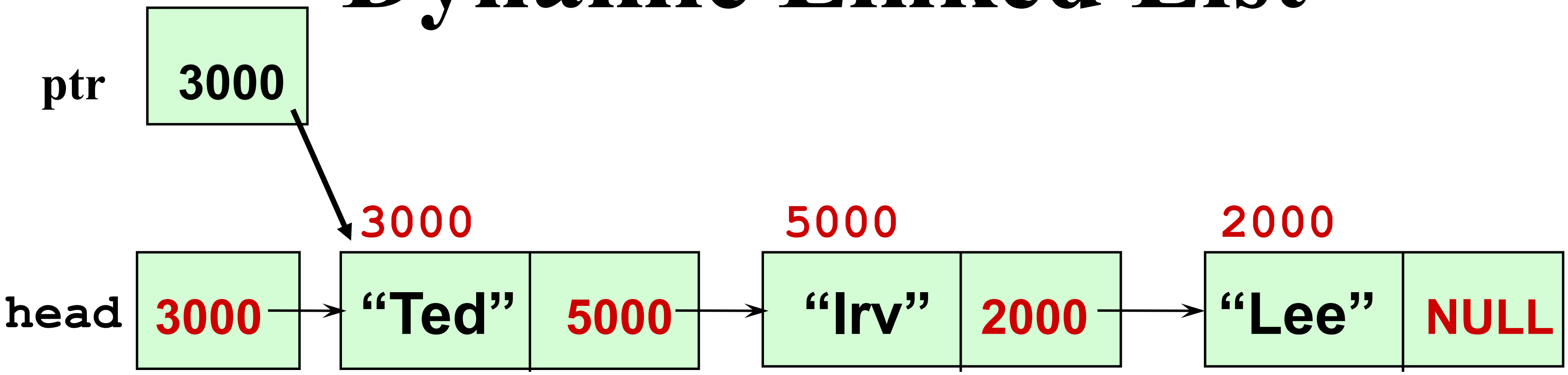
```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
    cout  <<  ptr->info;
   // Or, do something else with node *ptr
   ptr  =  ptr->next;
}
```

Traversing a Dynamic Linked List



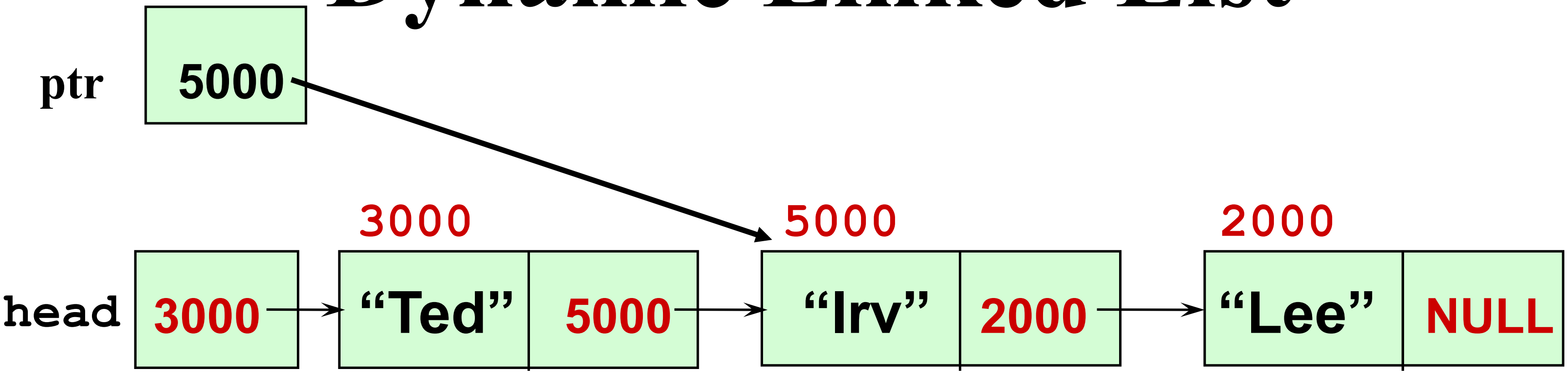
```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
    cout  <<  ptr->info;
    // Or, do something else with node *ptr
    ptr  =  ptr->next;
}
```

Traversing a Dynamic Linked List



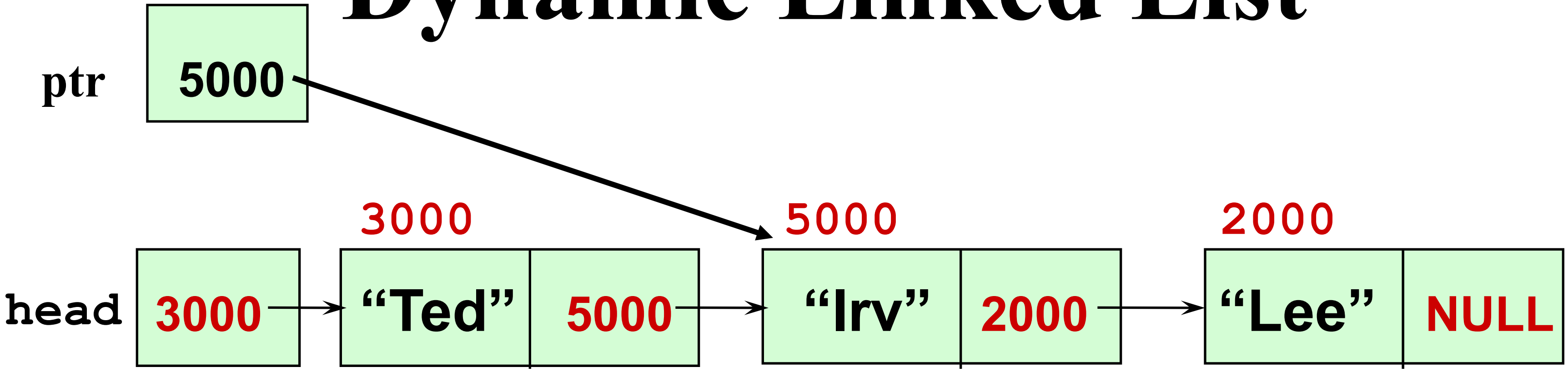
```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
     // Or, do something else with node *ptr
     ptr  =  ptr->next;
}
```

Traversing a Dynamic Linked List



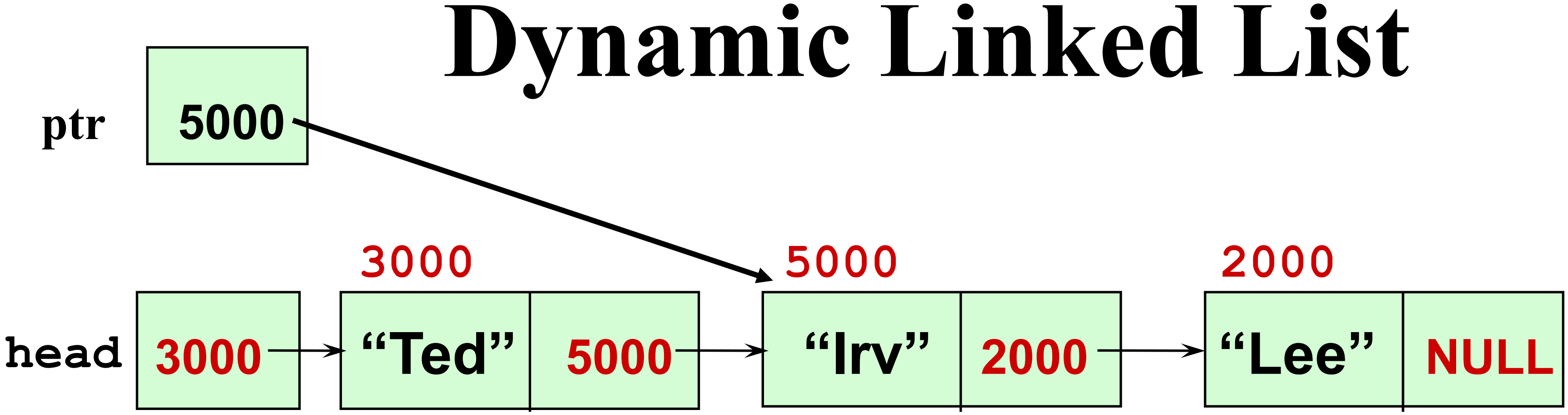
```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
    // Or, do something else with node *ptr
    ptr  =  ptr->next;
}
```

Traversing a Dynamic Linked List



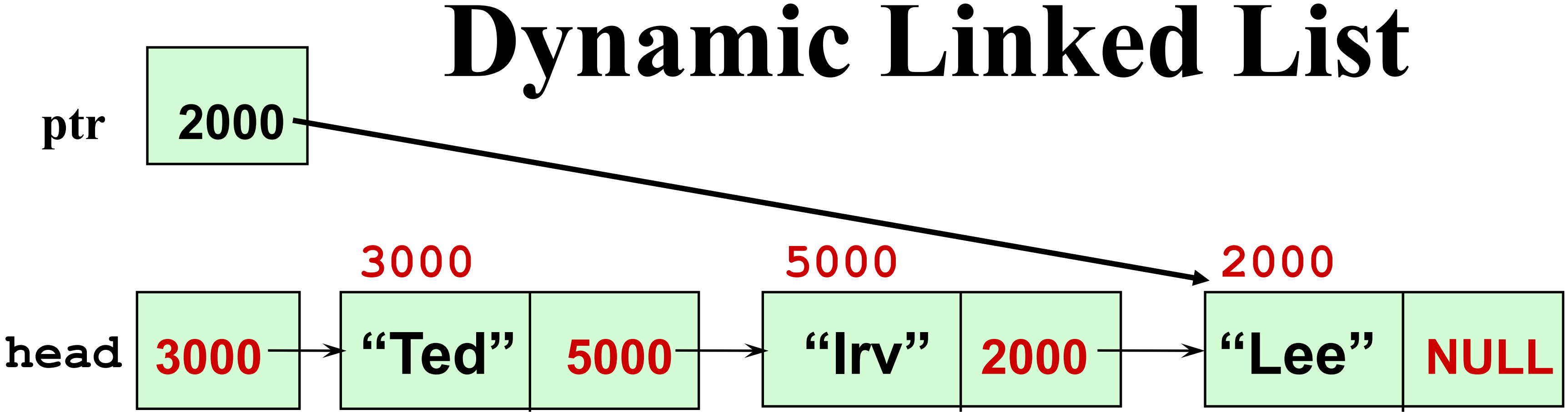
```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
       cout  <<  ptr->info;
      // Or, do something else with node *ptr
      ptr  =  ptr->next;
}
```

Traversing a Dynamic Linked List



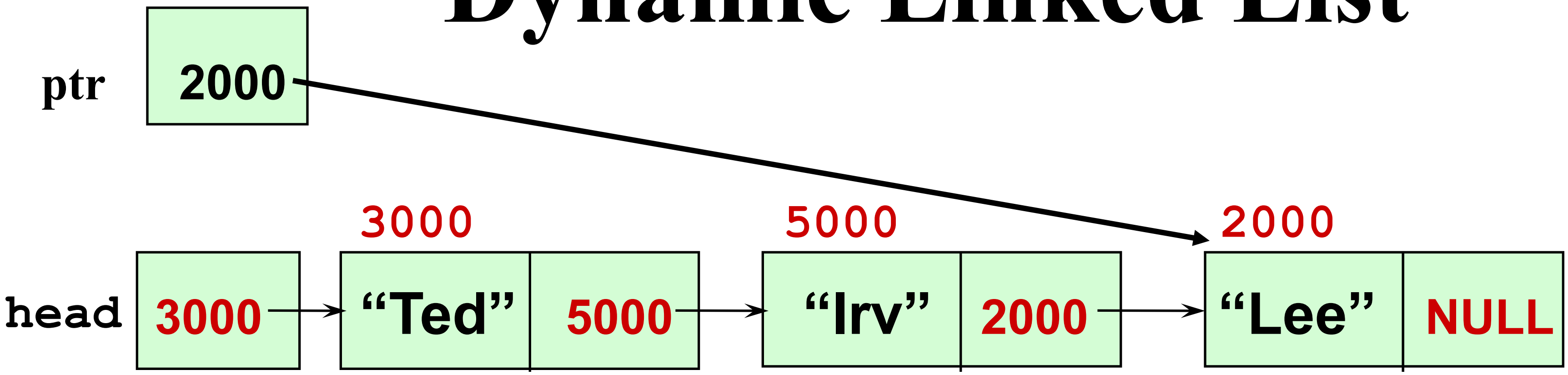
```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
     // Or, do something else with node *ptr
     ptr  =  ptr->next;
}
```

Traversing a Dynamic Linked List



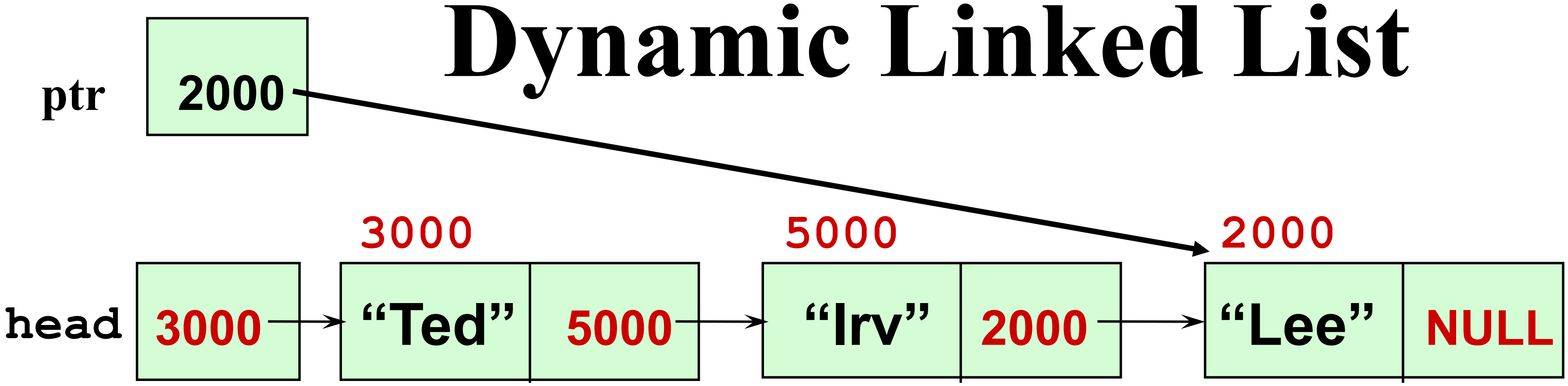
```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
     // Or, do something else with node *ptr
     ptr  =  ptr->next;
}
```

Traversing a Dynamic Linked List



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

Traversing a Dynamic Linked List



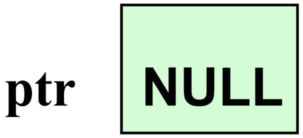
ptr 2000

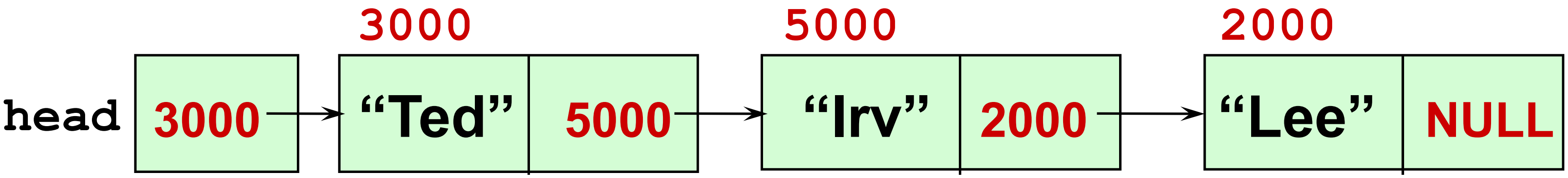
3000 5000 2000

head 3000 “Ted” 5000 “Irv” 2000 “Lee” NULL

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

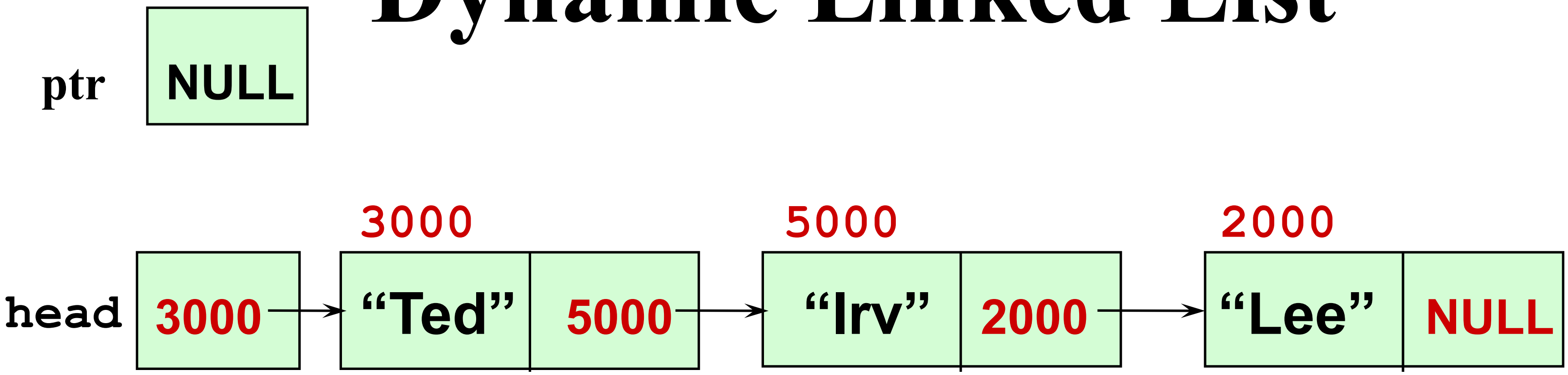
Traversing a Dynamic Linked List





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

Traversing a Dynamic Linked List

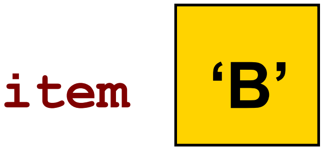


```
// Pre:  head points to a dynamic linked list
ptr  =  head;
while (ptr != NULL)
{
     cout  <<  ptr->info;
    // Or, do something else with node *ptr
    ptr  =  ptr->next;
}
```

Using Operator `new`

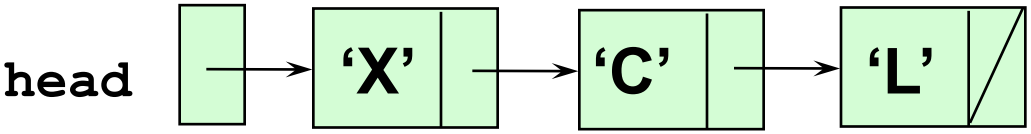
Recall

- **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**

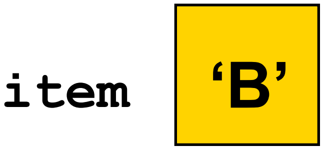

item
‘B’

Inserting a Node at the Front of a List

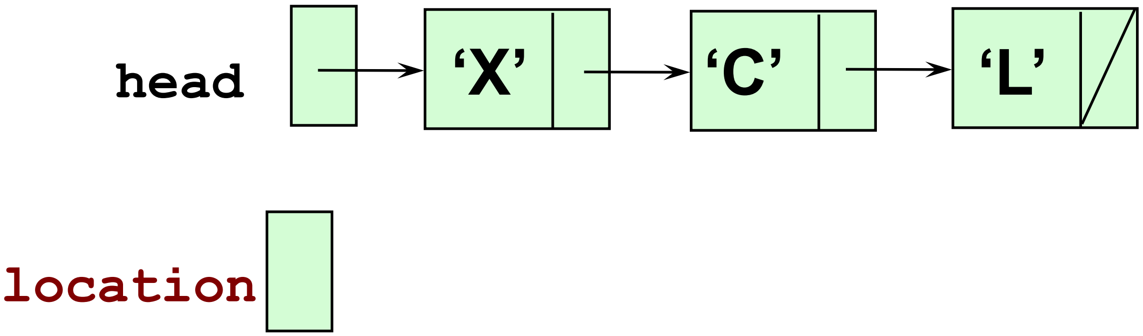
```
char       item = ‘B’;
NodePtr    location;
location = new   NodeType;
location->info = item;
location->next = head;
head = location;
```


head
‘X’
‘C’
‘L’

Inserting a Node at the Front of a List

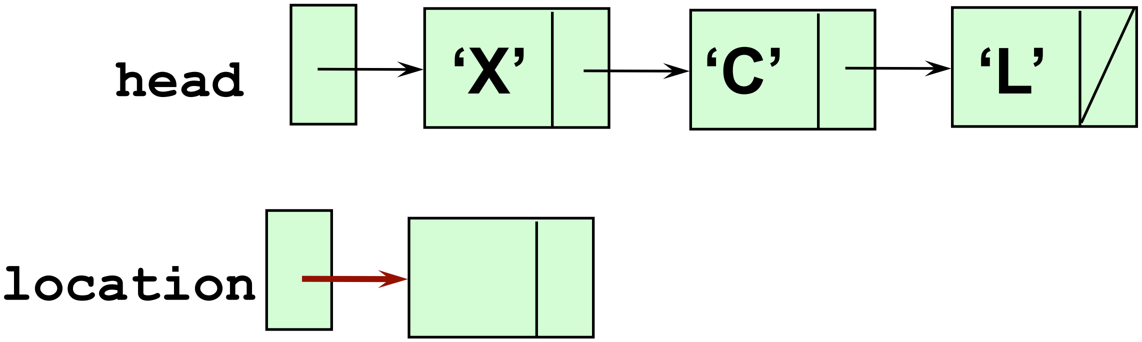


```
    char      item = ‘B’;
    NodePtr   location;
    location = new  NodeType;
    location->info = item;
    location->next = head;
    head = location;
```



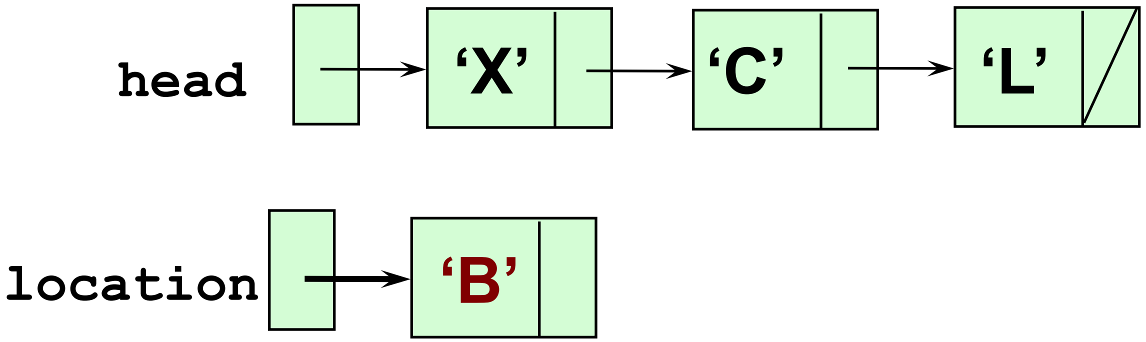
Inserting a Node at the Front of a List

```
char      item = ‘B’;
NodePtr   location;
location = new  NodeType;
location->info = item;
location->next = head;
head = location;
```



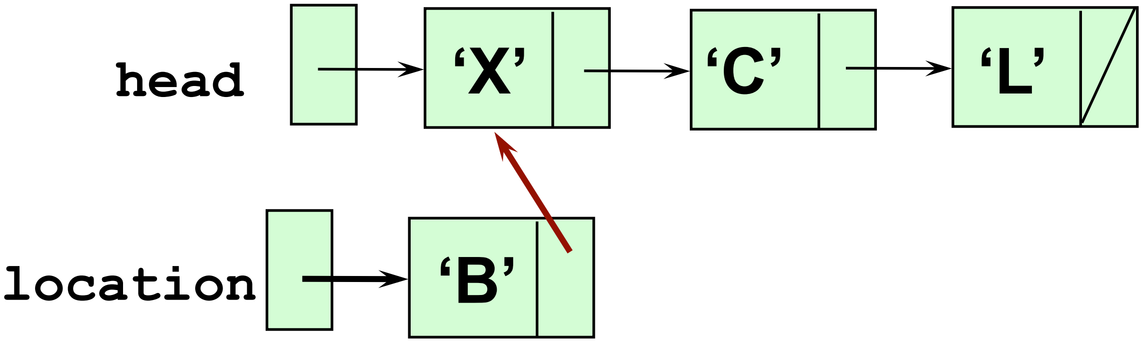
Inserting a Node at the Front of a List

```
char       item = 'B';
NodePtr    location;
location = new  NodeType;
location->info = item;
location->next = head;
head = location;
```



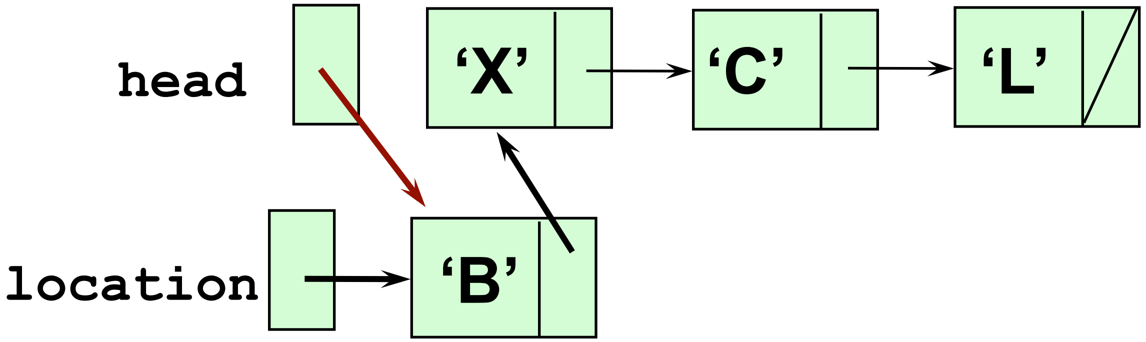
Inserting a Node at the Front of a List

```
char      item = 'B';
NodePtr   location;
location = new  NodeType;
location->info = item;
location->next = head;
head = location;
```



Inserting a Node at the Front of a List

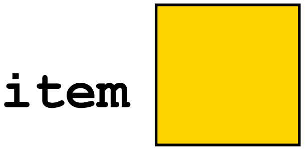
```
    char      item = ‘B’;
    NodePtr   location;
    location = new  NodeType;
    location->info = item;
    location->next = head;
    head = location;
```



Using Operator `delete`

When you use the operator `delete`

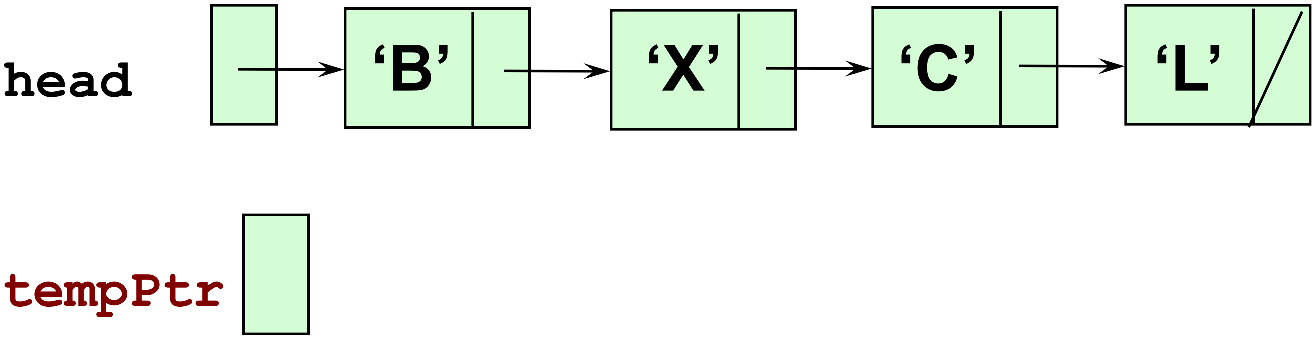
- The object currently pointed to by the pointer is deallocated and the pointer is considered undefined
- The object’s memory is returned to the free store



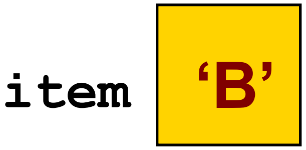
Deleting the First Node from the List

```
NodePtr  tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete  tempPtr;
tempPtr = NULL;
```

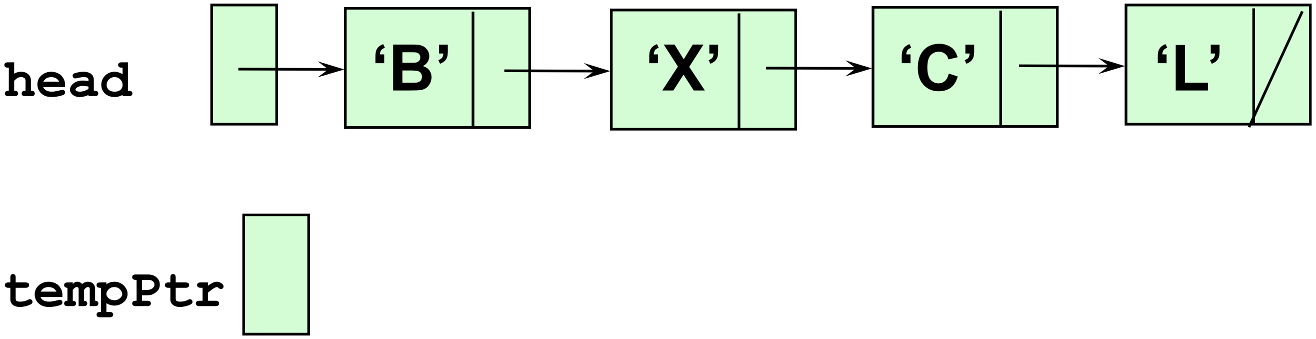


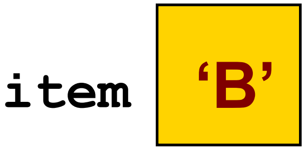
Deleting the First Node from the List



```
NodeType *  tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete  tempPtr;
tempPtr = NULL;
```

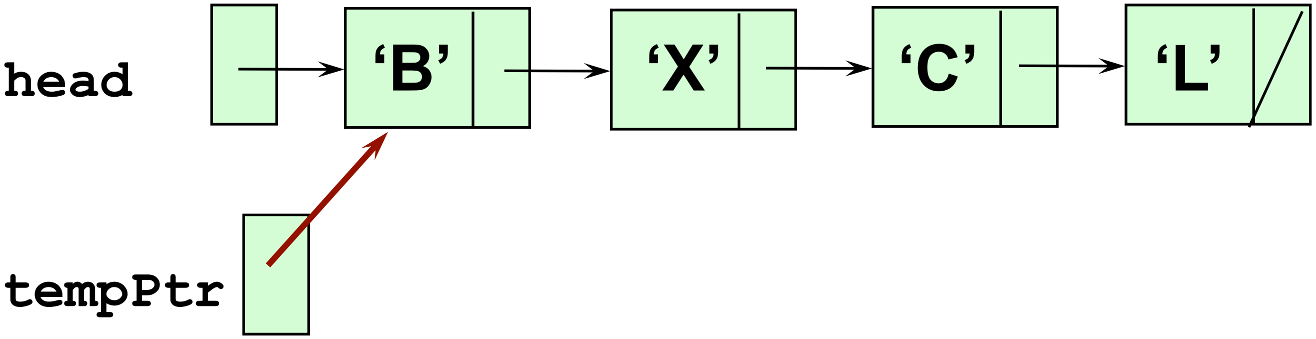




Deleting the First Node from the List

```
NodeType *  tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete  tempPtr;
tempPtr = NULL;
```

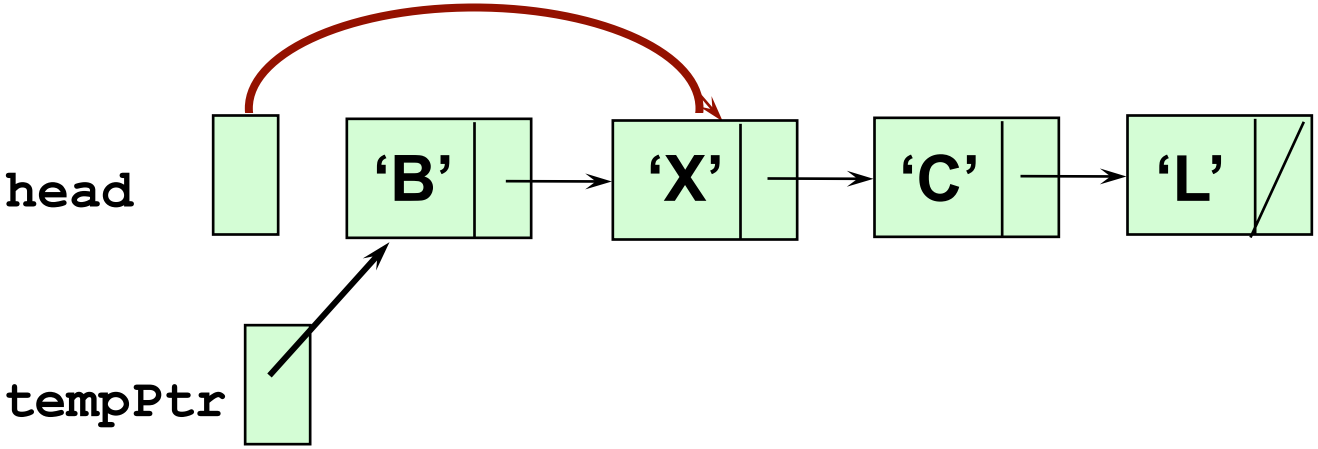


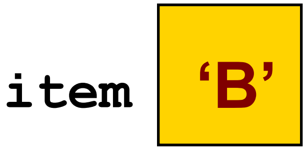
item ‘B’

Deleting the First Node from the List

```
NodeType *  tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete  tempPtr;
```

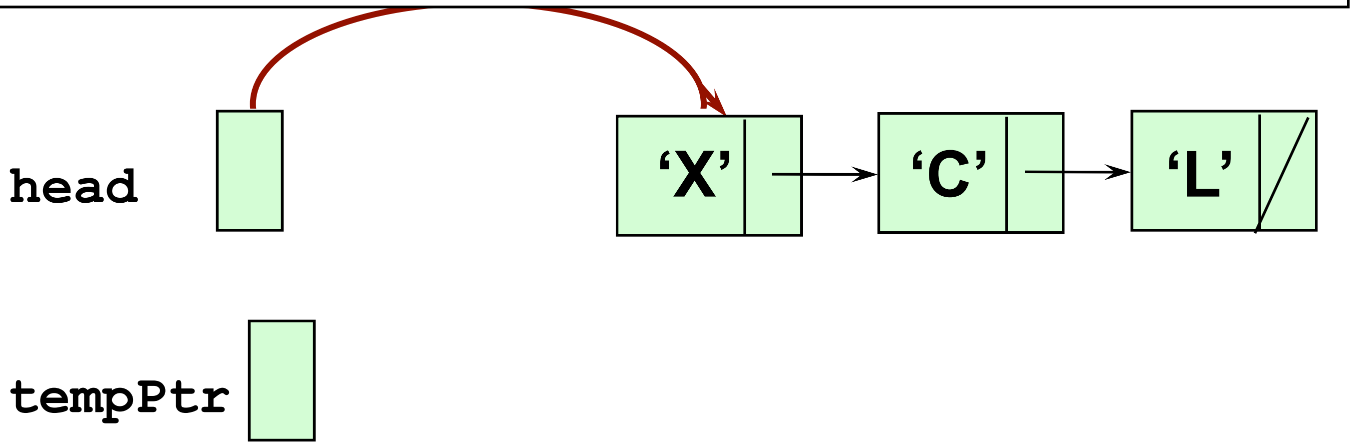


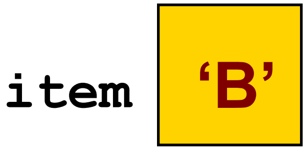


Deleting the First Node from the List

```
NodeType *  tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete  tempPtr;
tempPtr = NULL;
```

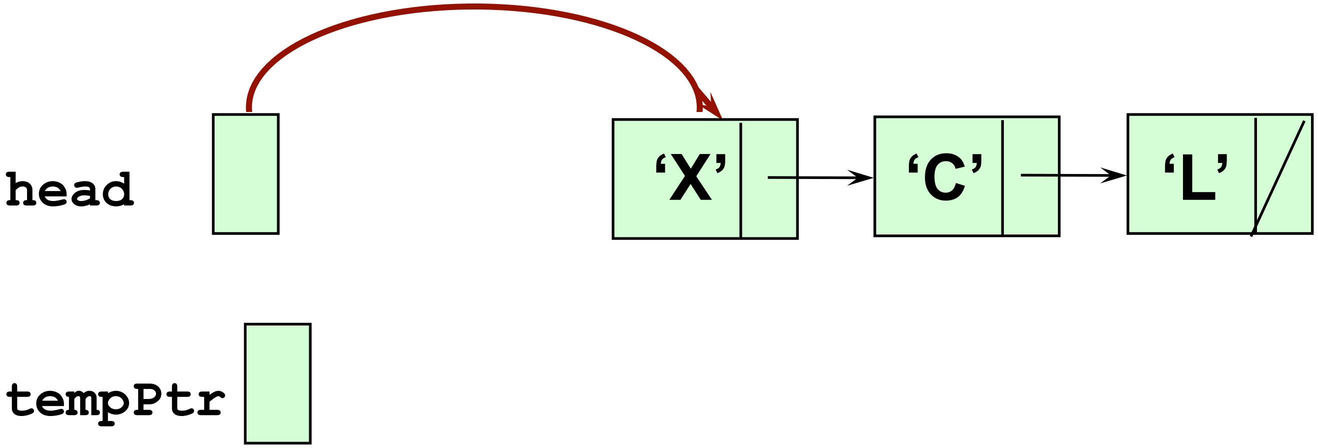




Deleting the First Node from the List

```
NodeType *  tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete  tempPtr;
tempPtr = NULL;
```


head
‘X’
‘C’
‘L’
tempPtr