04-630
Data Structures and Algorithms for Engineers

Lecture 8: Constrainers and Dictionaries II

Agenda

Containers and Dictionaries

- Containers
- Dictionaries
- List ADT
  - Array implementation
  - Linked list implementation

Preliminaries:
Linked Lists Using Pointers

Why Pointer-Based Implementation?

- Linked lists are used to avoid excessive data movement with insertions and deletions
- Elements are not necessarily stored in contiguous memory locations
- Makes efficient use of memory space
  - Allocate space when needed
  - Deallocate space when finished & return it to the free store
- Failure to deallocate space will cause memory leakage

Why Pointer-Based Implementation?

Some guidelines when writing programs that dynamically allocate memory

- Use malloc or new to create data-structures of the appropriate size
- Remember to avoid memory leakage by always using free and delete to deallocate dynamically-created data-structures
- Check every call to **malloc** or **new** to see if it returned NULL (i.e., check if the allocation was unsuccessful)

Why Pointer-Based Implementation?

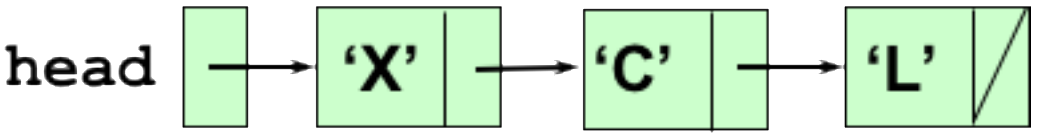
Some guidelines when writing programs that dynamically allocate memory

- You must expect free or delete to alter the contents of the memory that was freed or deleted
- Never access a data structure after it has been freed or deleted
- If malloc fails in a non-interactive program, make that a fatal error
- In an interactive program, it is better to abort the current command and return to the command reader loop

A Linked List

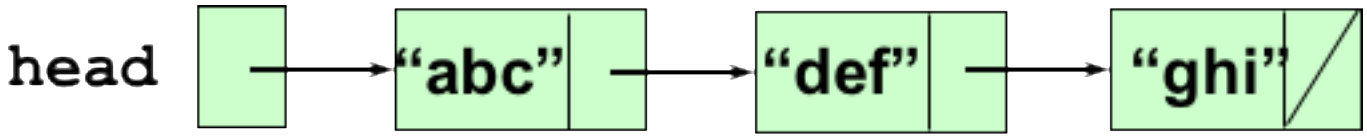
- A linked list is a list in which the order of the components is determined by an explicit link member in each node

- The nodes are **`structs`**
  - each node contains a component member and also a link member that gives the location of the next node in the list



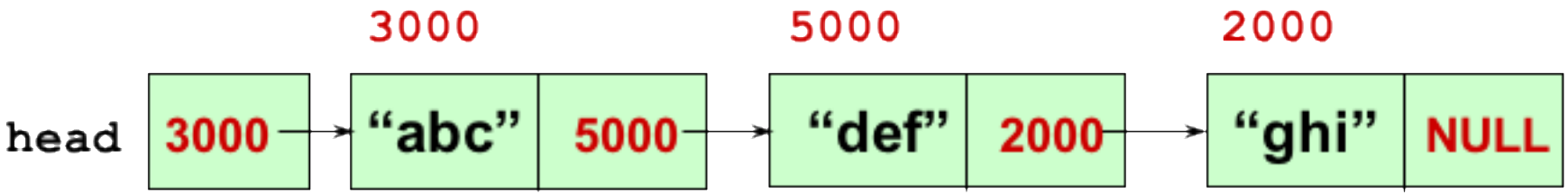
Pointer-Based (Dynamic) Linked List

A pointer-based linked list is a dynamic linked list where nodes are linked together by pointers, and an external pointer (or head pointer) points to the first node in the list



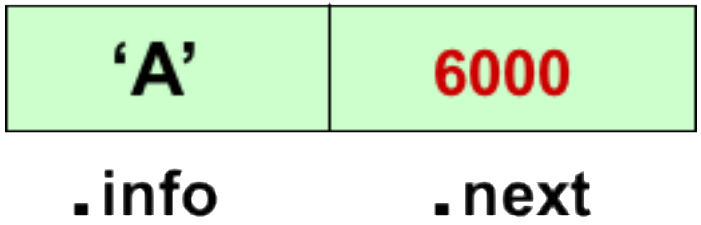
Nodes can be located anywhere in memory

The link member holds the memory address of
(or a reference to) of the next node in the list

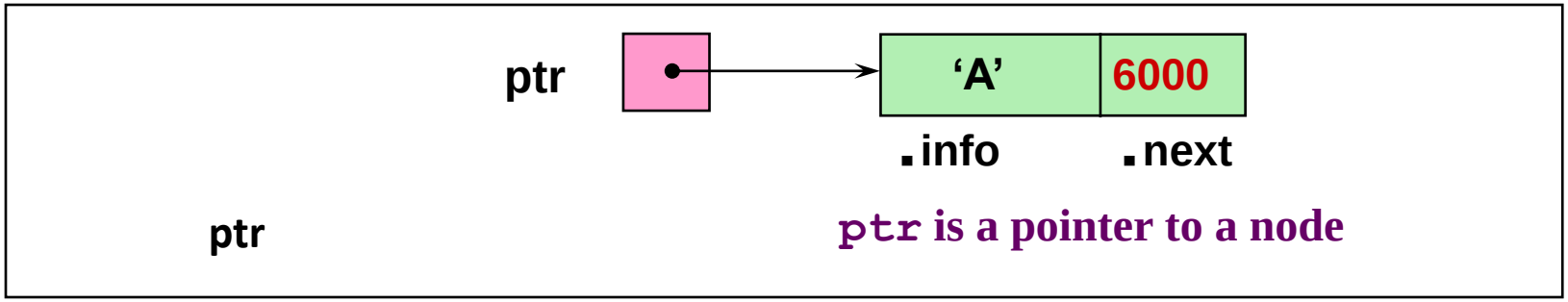


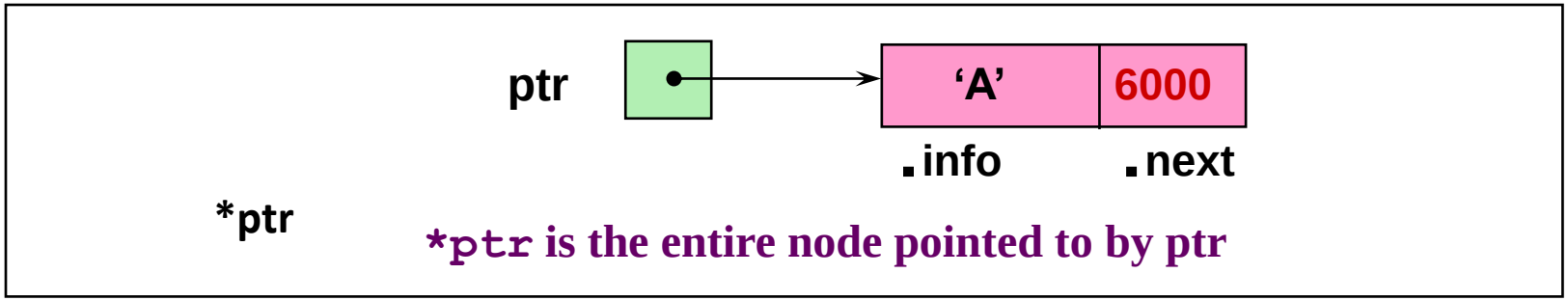
Declarations for a Dynamic Linked List

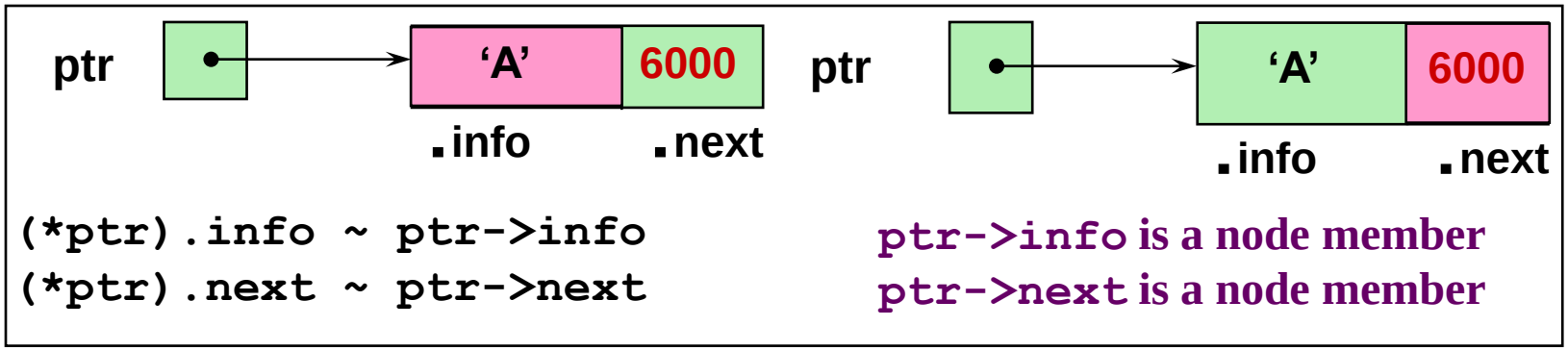
```
// Type DECLARATIONS

struct NodeType  {
   char         info;
   NodeType*    next;
}

typedef  NodeType*  NodePtr;

// Variable DECLARATIONS

NodePtr  head;
NodePtr  ptr;
```



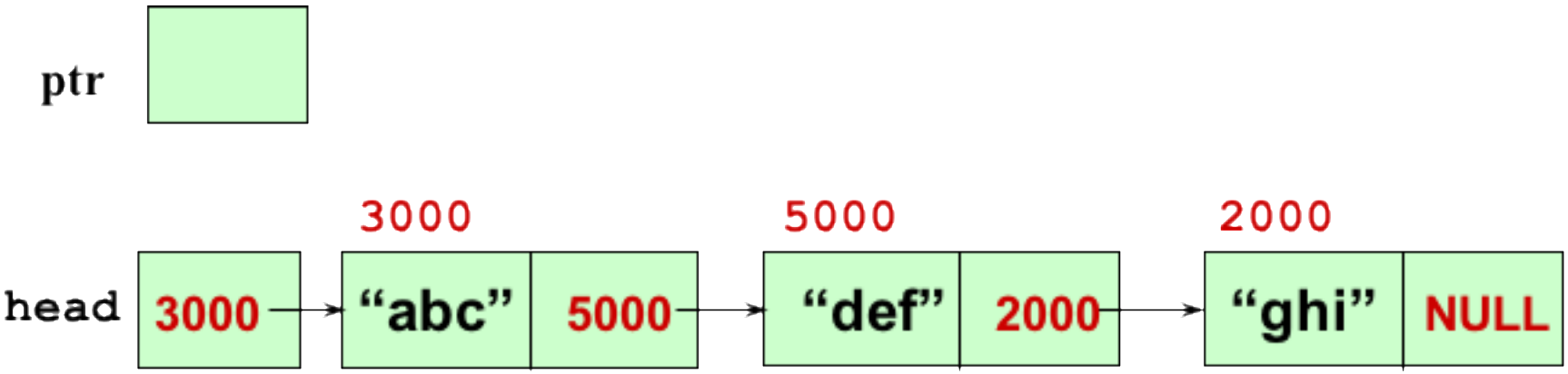
Pointer Dereferencing and Member Selection





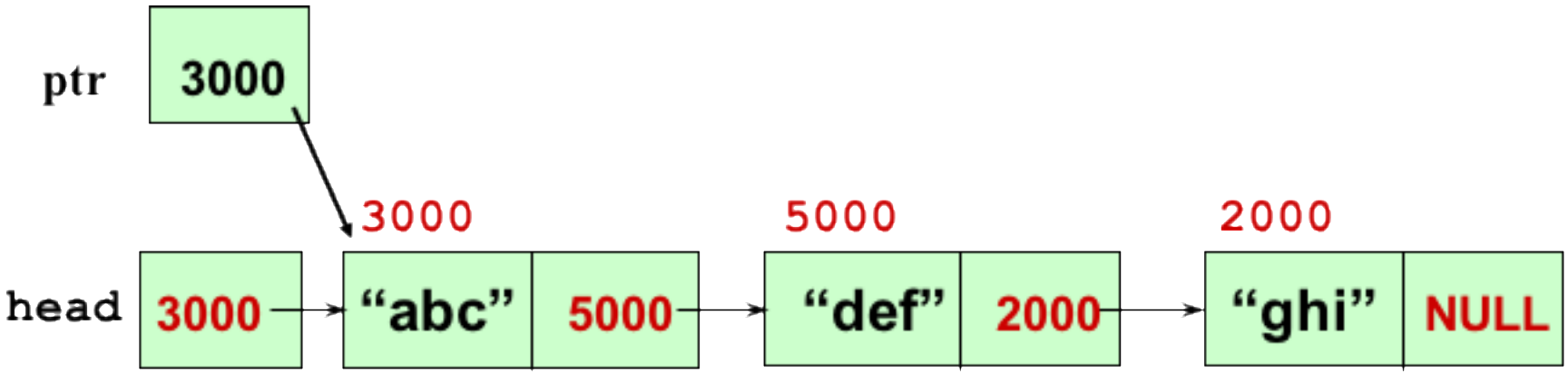


Traversing a Linked List



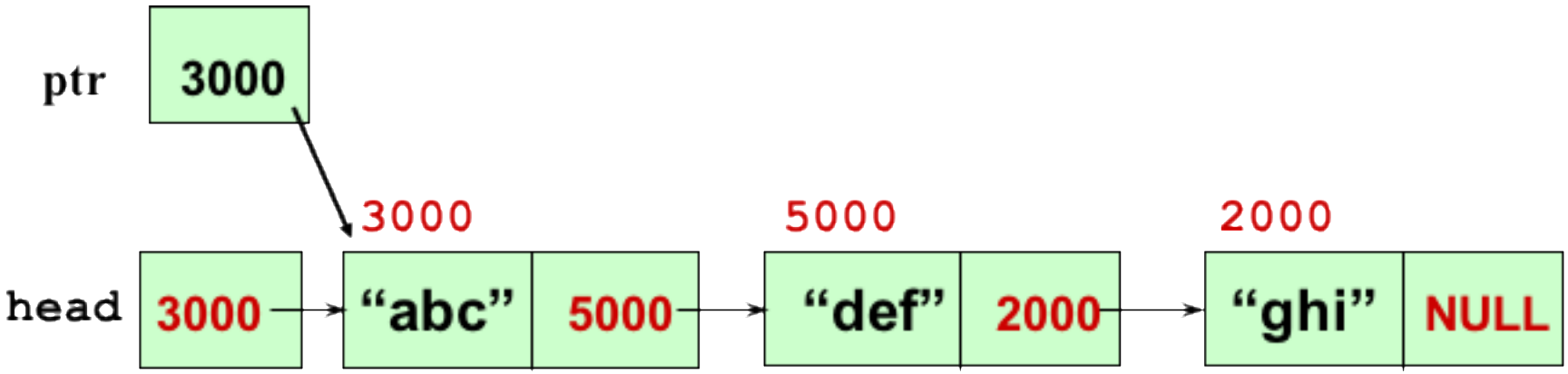

```
//PRE:  head points to a dynamic linked list

ptr  =  head ;
while (ptr != NULL)  {
     cout  <<  ptr->info ;
          // Or, do something else with node *ptr
     ptr  =  ptr->next ;
}
```

Traversing a Linked List



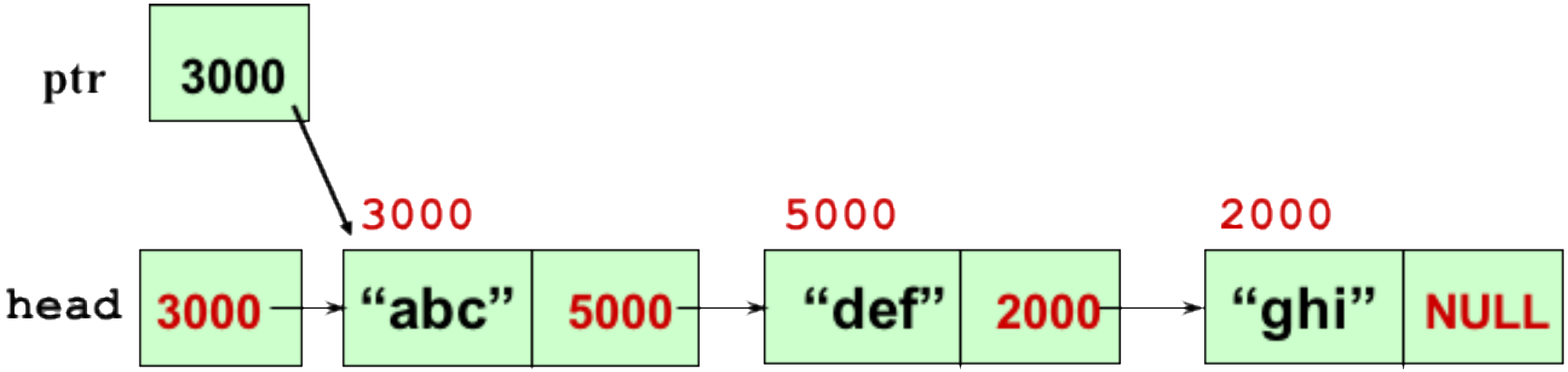

```
//PRE:  head points to a dynamic linked list

ptr  =  head ;
while (ptr != NULL)  {
    cout  <<  ptr->info ;
          // Or, do something else with node *ptr
    ptr  =  ptr->next ;
}
```

Traversing a Linked List



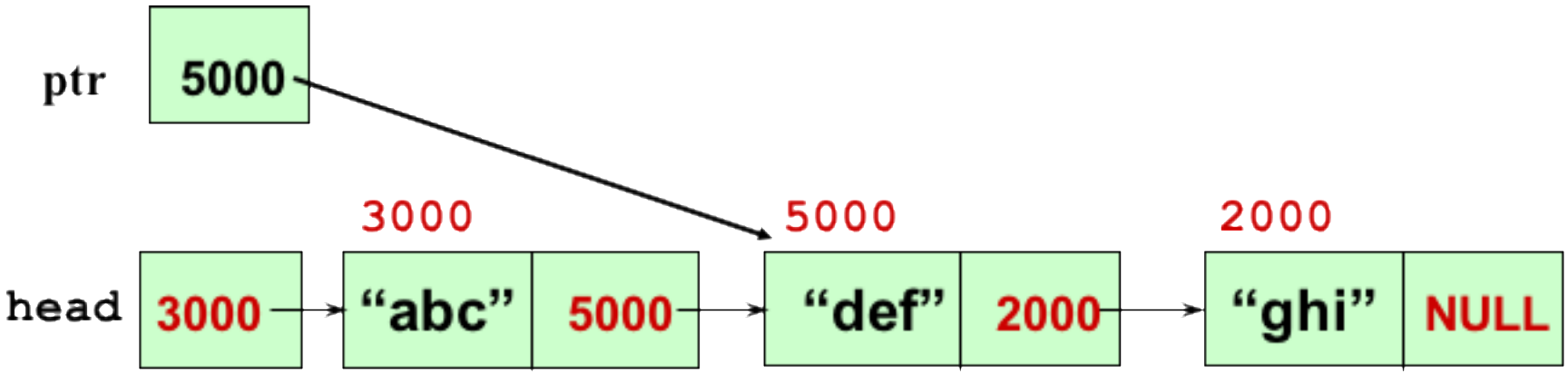

```
//PRE:  head points to a dynamic linked list

ptr  =  head ;
while (ptr != NULL)  {
     cout  <<  ptr->info ;
          // Or, do something else with node *ptr
     ptr  =  ptr->next ;
}
```

Traversing a Linked List



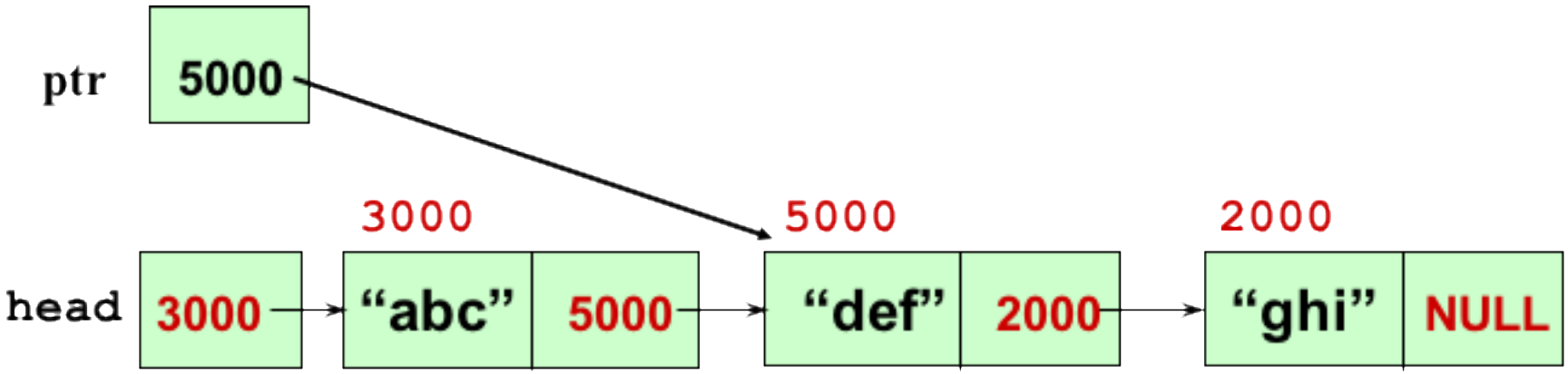

```
//PRE:  head points to a dynamic linked list

ptr  =  head ;
while (ptr != NULL)  {
    cout  <<  ptr->info ;
          // Or, do something else with node *ptr
    ptr  =  ptr->next ;
}
```

Traversing a Linked List



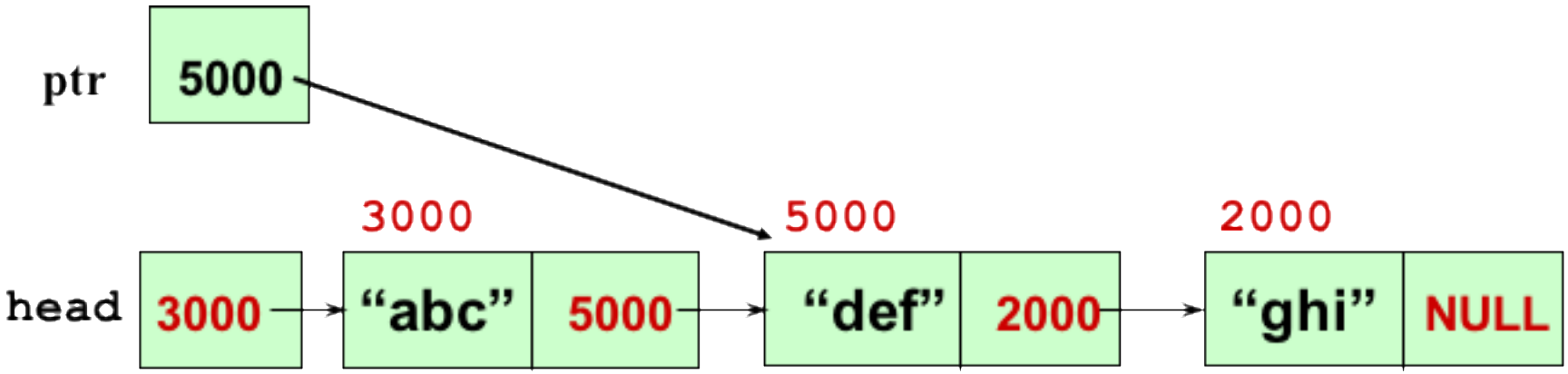

```
//PRE:  head points to a dynamic linked list

ptr  =  head ;
while (ptr != NULL)  {
     cout  <<  ptr->info ;
          // Or, do something else with node *ptr
     ptr  =  ptr->next ;
}
```

Traversing a Linked List



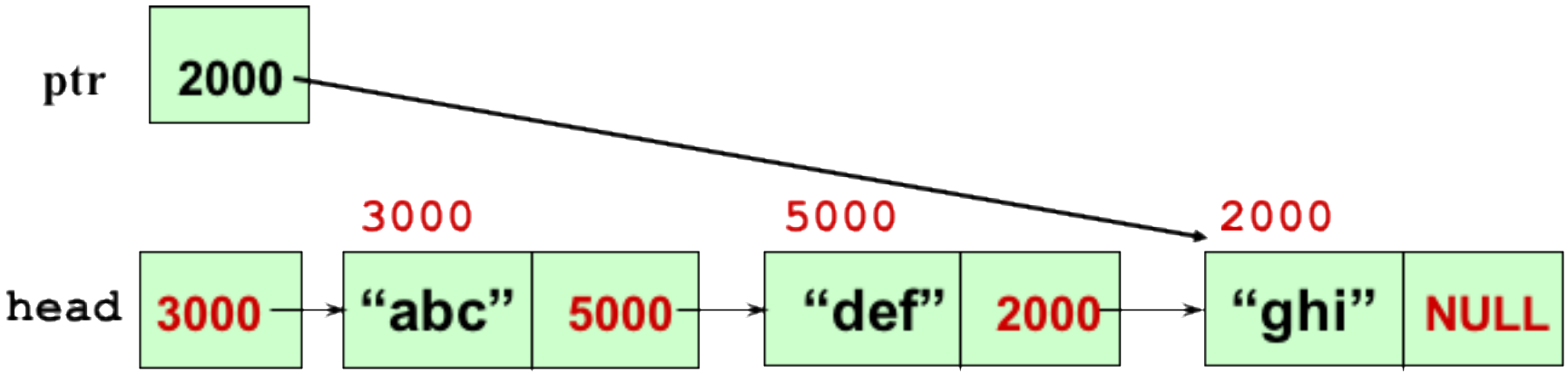

```
//PRE:  head points to a dynamic linked list

ptr  =  head ;
while (ptr != NULL)  {
     cout  <<  ptr->info ;
          // Or, do something else with node *ptr
     ptr  =  ptr->next ;
}
```

Traversing a Linked List



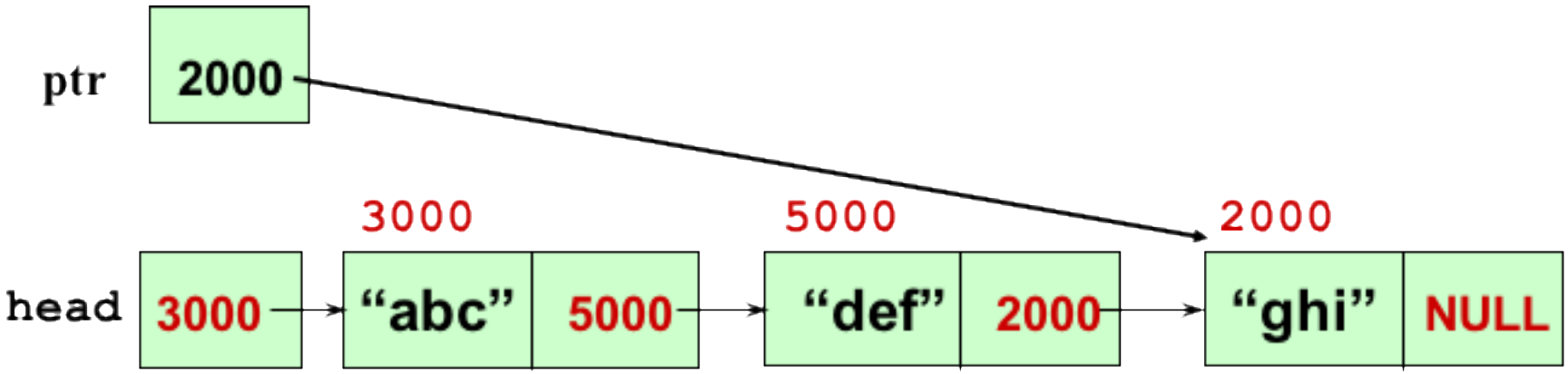

```
//PRE:  head points to a dynamic linked list

ptr  =  head ;
while (ptr != NULL)  {
    cout  <<  ptr->info ;
          // Or, do something else with node *ptr
    ptr  =  ptr->next ;
}
```

Traversing a Linked List



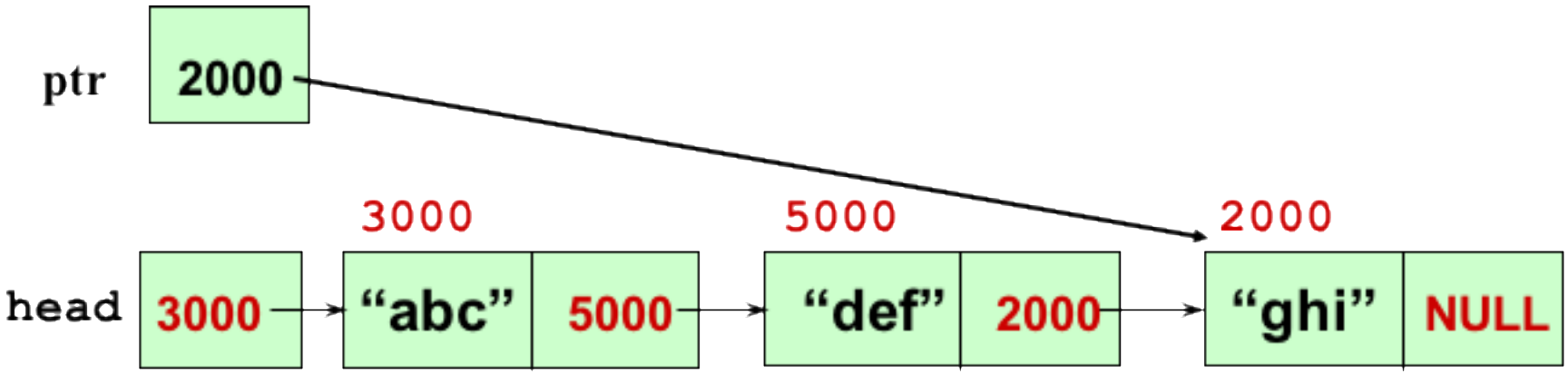

```
//PRE:  head points to a dynamic linked list

ptr  =  head ;
while (ptr != NULL)  {
     cout  <<  ptr->info ;
          // Or, do something else with node *ptr
     ptr  =  ptr->next ;
}
```

Traversing a Linked List



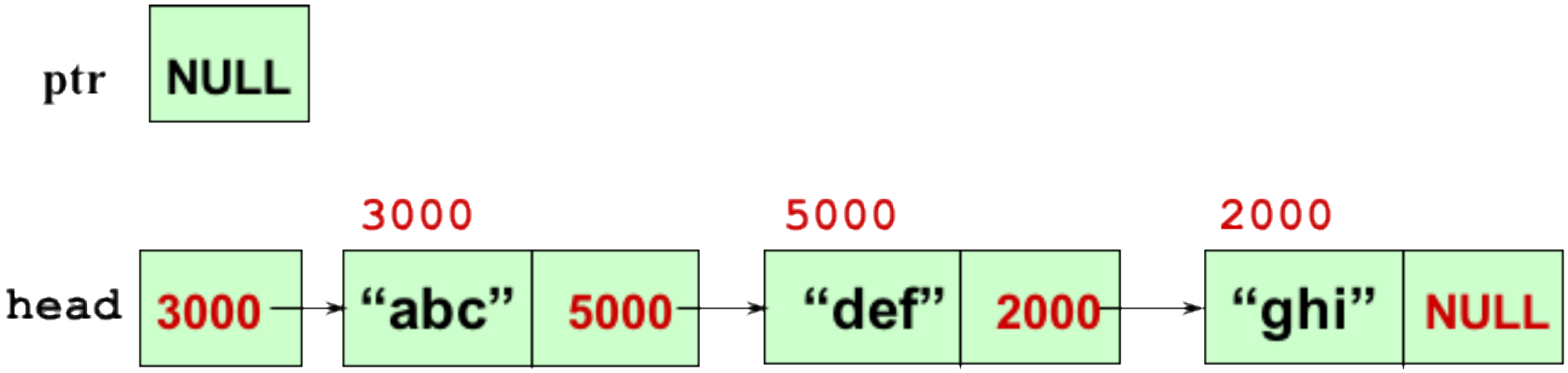

```
//PRE:  head points to a dynamic linked list

ptr  =  head ;
while (ptr != NULL)  {
     cout  <<  ptr->info ;
          // Or, do something else with node *ptr
     ptr  =  ptr->next ;
}
```

Traversing a Linked List



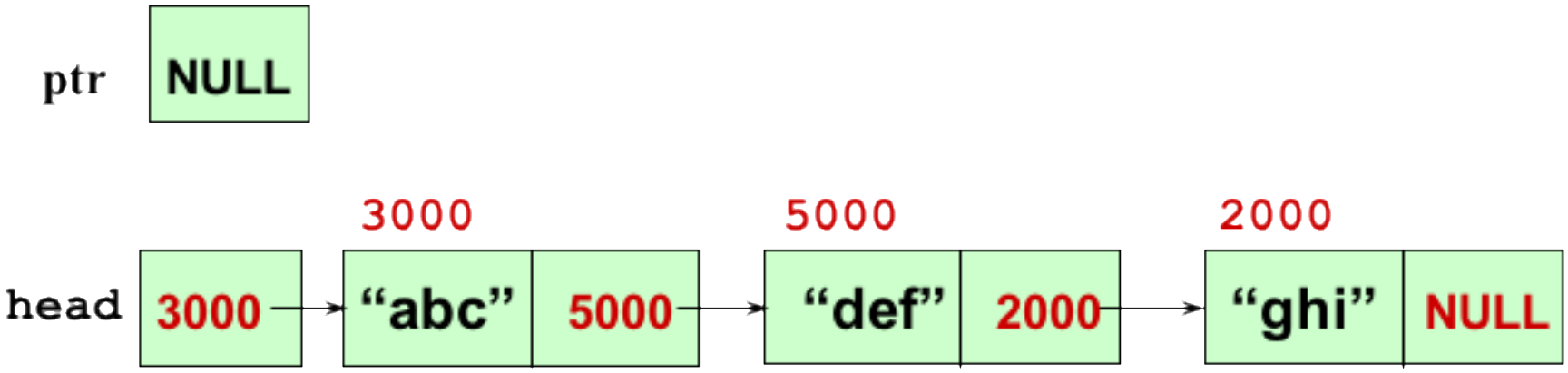

```
//PRE:  head points to a dynamic linked list

ptr  =  head ;
while (ptr != NULL)  {
    cout  <<  ptr->info ;
          // Or, do something else with node *ptr
    ptr  =  ptr->next ;
}
```

Traversing a 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 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`

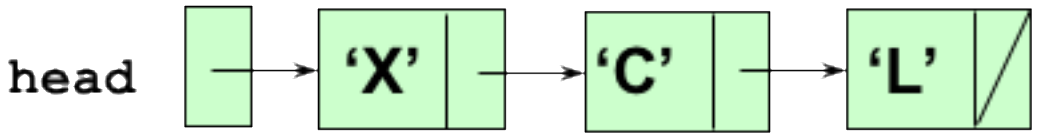
- If memory is available in an area called the free store (or heap), operator `new` allocates the requested object, and returns a pointer to the memory allocated.

- The dynamically allocated object exists until the `delete` operator destroys it.

Inserting a Node at the Front of a List

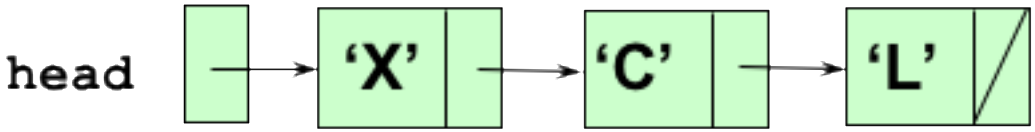
item 'B'

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



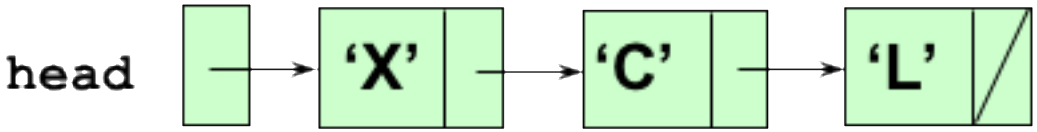
Inserting a Node at the Front of a List

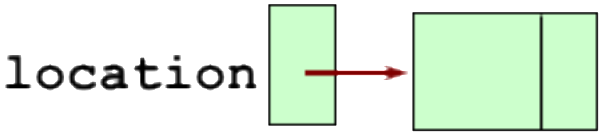

```
char     item = 'B';
NodePtr  location;
location = new  NodeType;
location->info = item;
location->next = head;
head = location;
```



Inserting a Node at the Front of a List

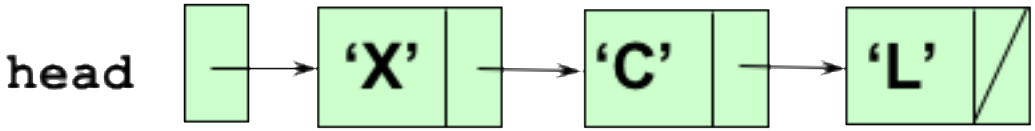

```
char      item = 'B';
NodePtr   location;
location = new  NodeType;
location->info = item;
location->next= head;
head = location;
```

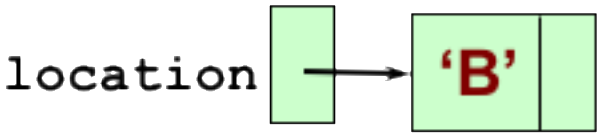




Inserting a Node at the Front of a List


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

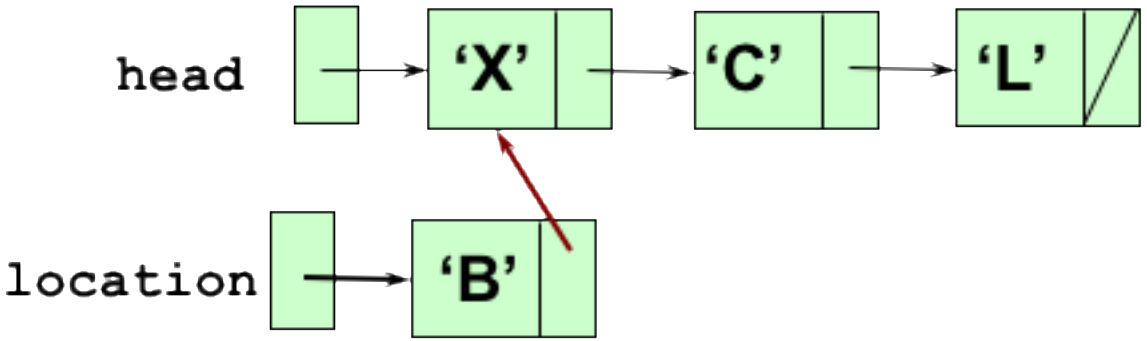




Inserting a Node at the Front of a List

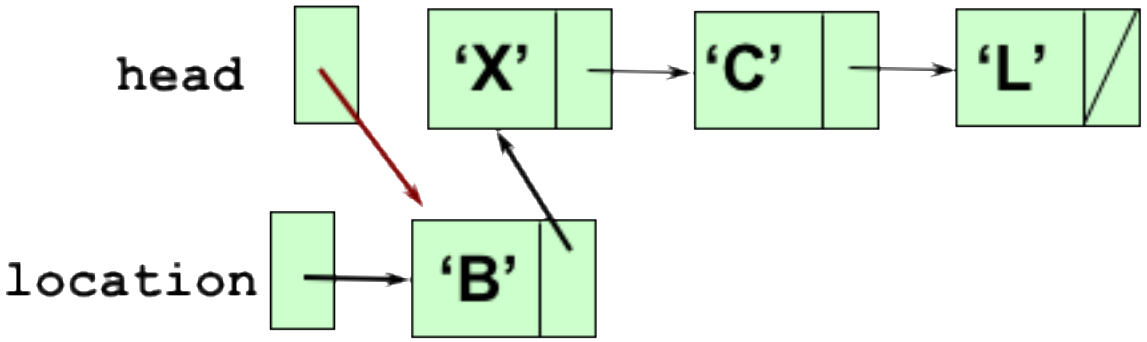
item 'B'

```
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`

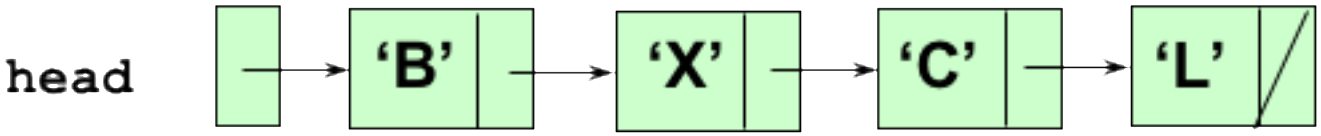
The object currently pointed to by the pointer is deallocated, and the pointer is considered undefined.

The object's memory is returned to the free store.

Deleting the first node from a list

item

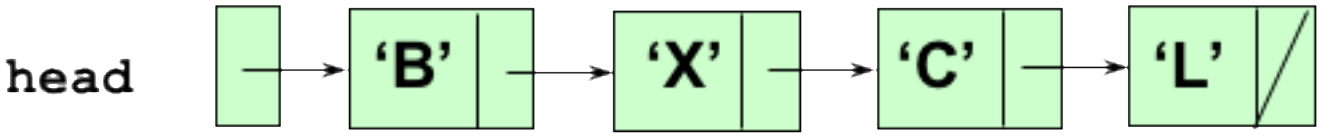
```
NodePtr  tempPtr;

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

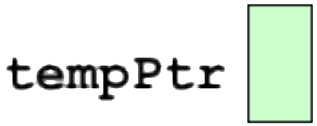


tempPtr

Deleting the first node from a list


```
NodeType *   tempPtr;

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

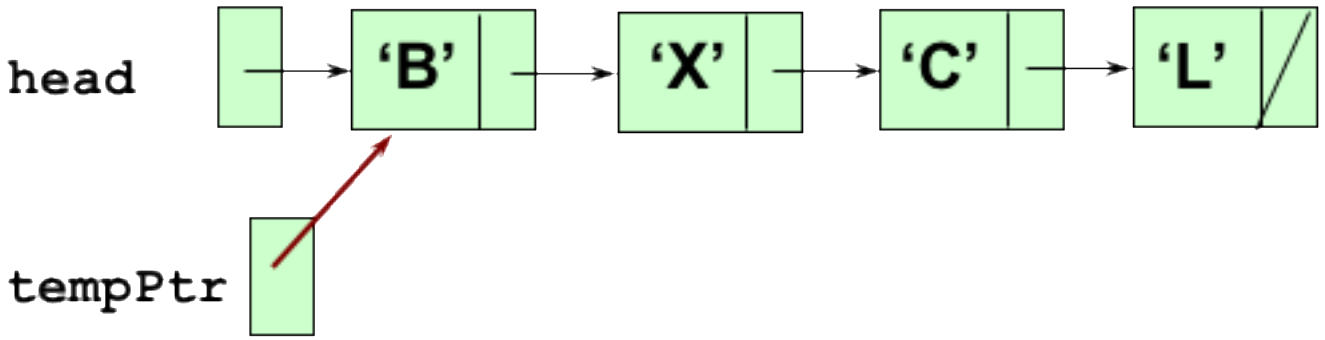




Deleting the first node from a list

item 'B'

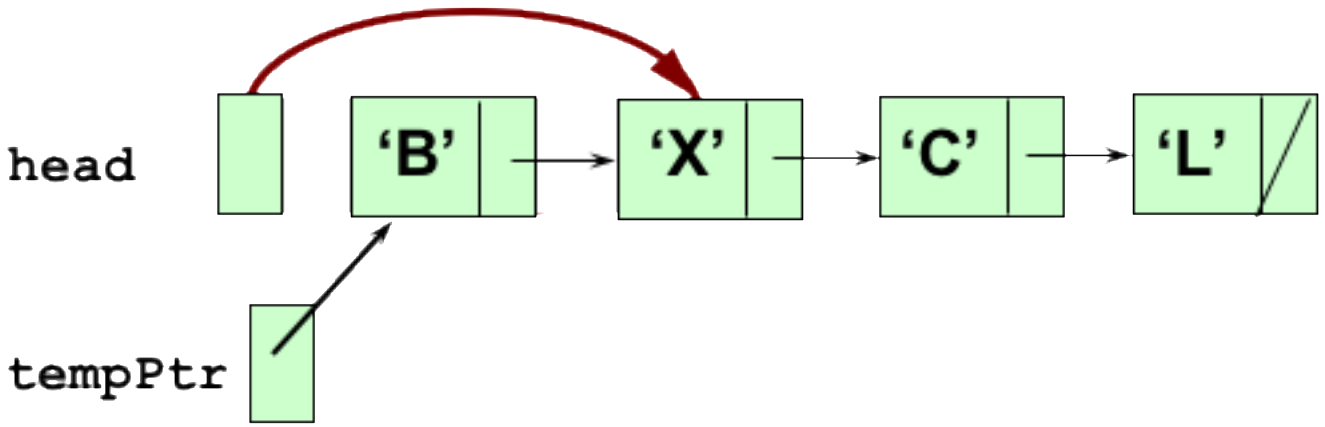
```
NodeType *  tempPtr;

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



Deleting the first node from a list

item 'B'

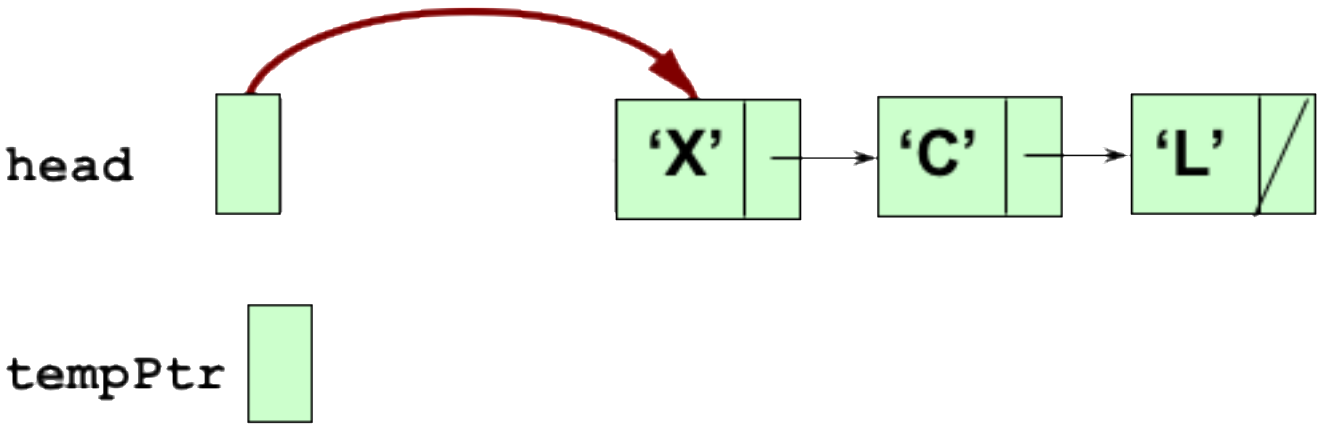
```
NodeType *  tempPtr;

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



Deleting the first node from a list

item 'B'

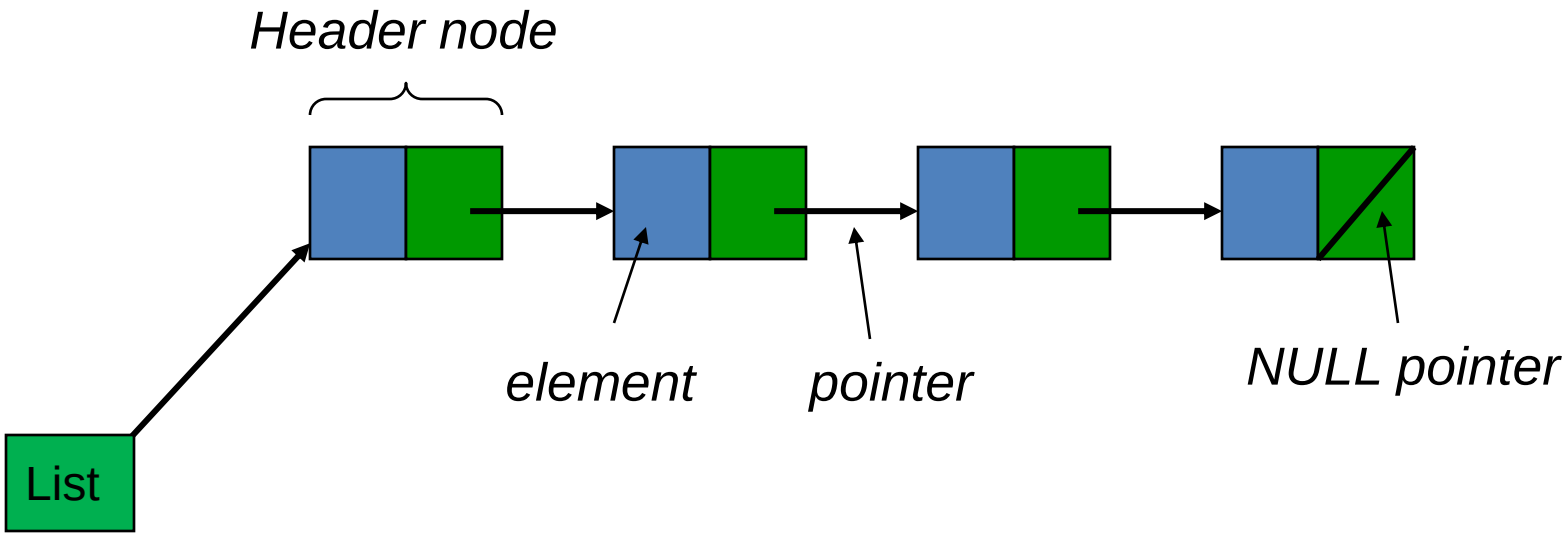
```
NodeType *  tempPtr;

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



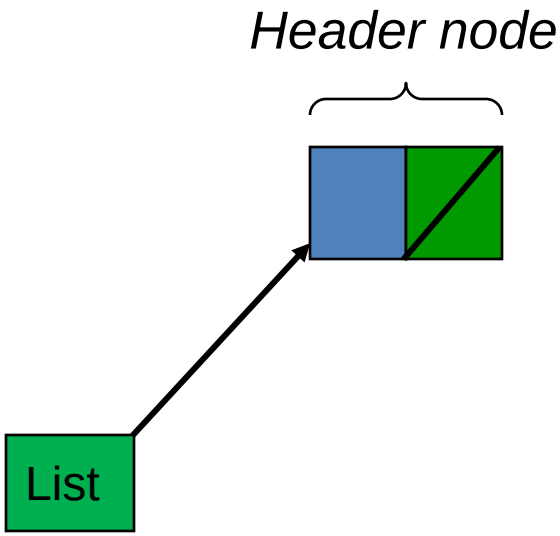
End of Preliminaries:

Linked Lists Using Pointers

LIST: Linked-List Implementation

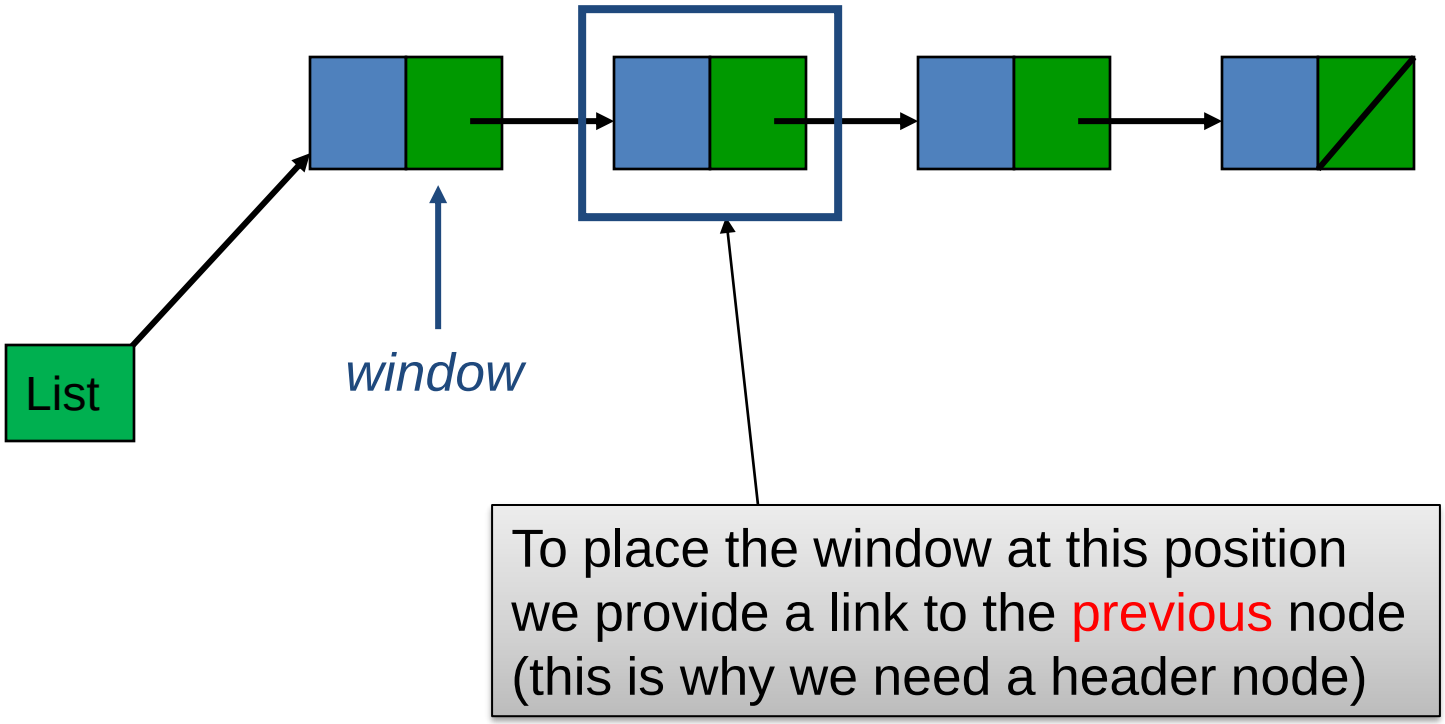


LIST: Linked-List Implementation

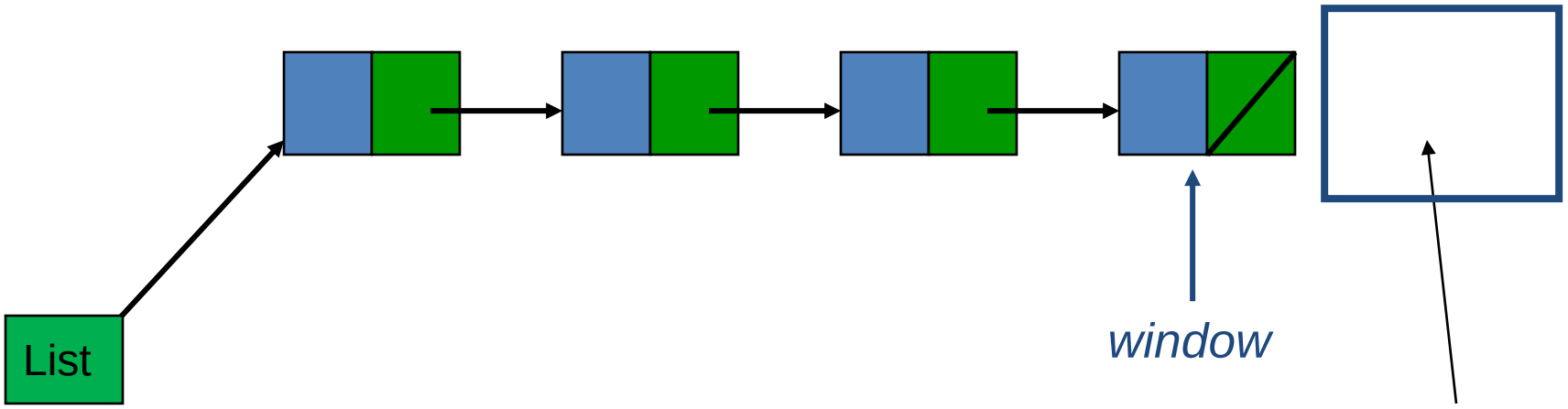


An empty list!!!

LIST: Linked-List Implementation

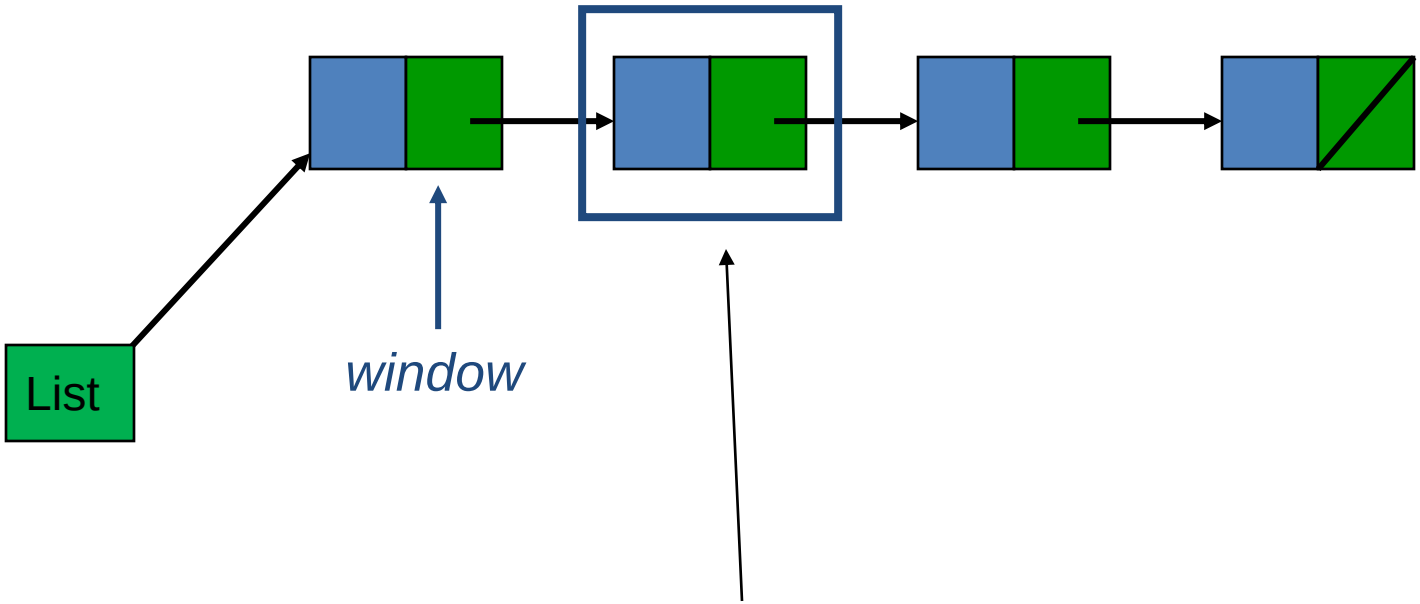


LIST: Linked-List Implementation



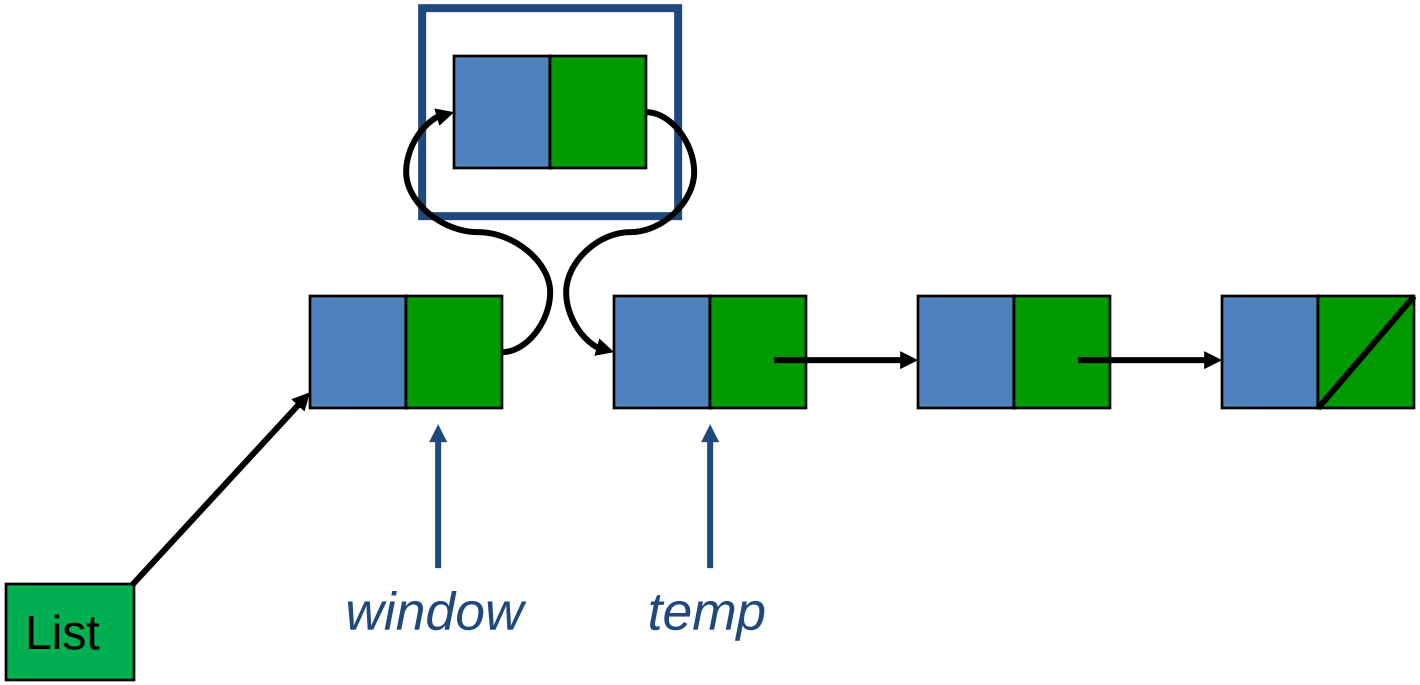
To place the window at end of the list we provide a link to the last node

LIST: Linked-List Implementation



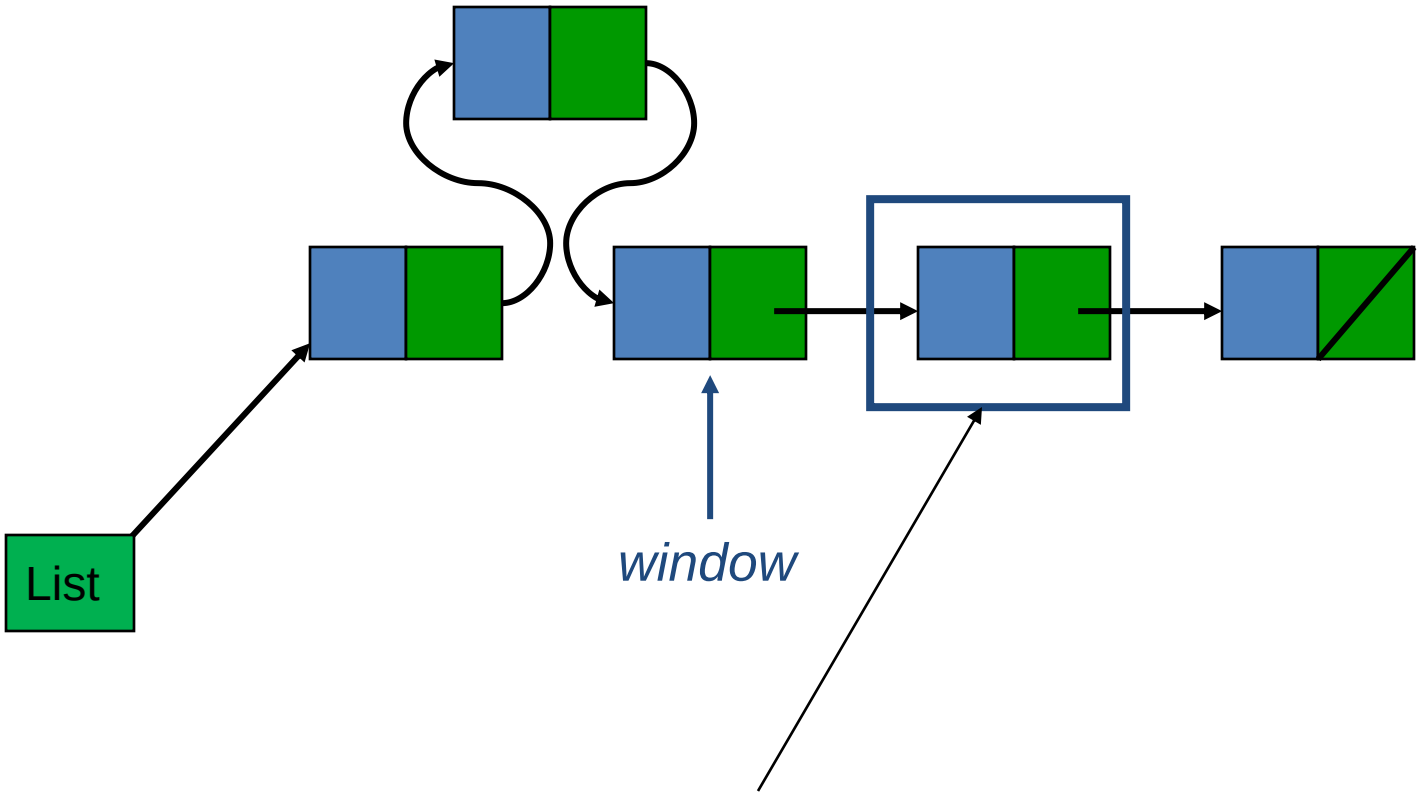
To insert a node at this window position
we create the node and re-arrange the links

LIST: Linked-List Implementation



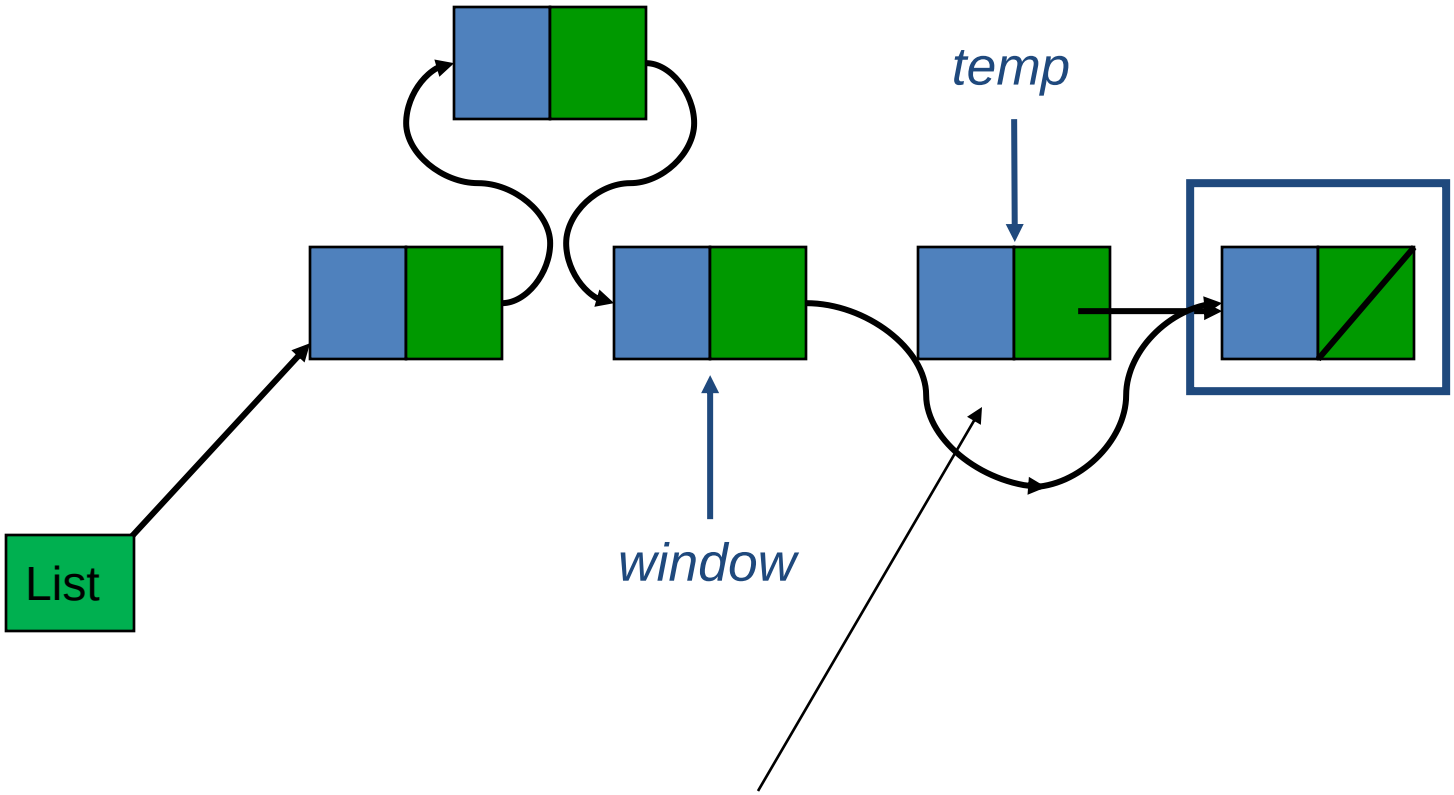
To insert a node at this window position
we create the node and re-arrange the links

LIST: Linked-List Implementation



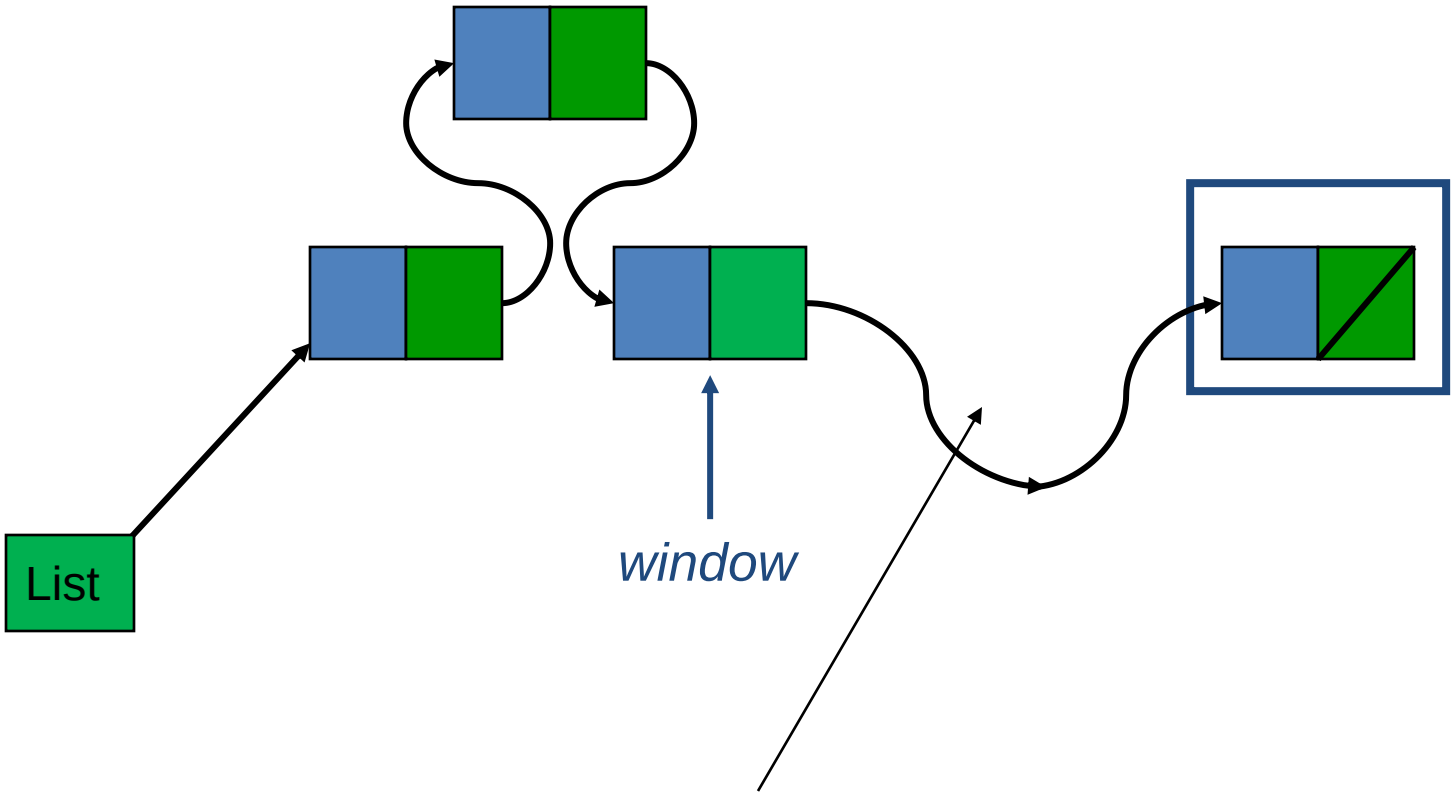
To delete a node at this window position
we re-arrange the links and free the node

LIST: Linked-List Implementation



To delete a node at this window position
we re-arrange the links and free the node

LIST: Linked-List Implementation



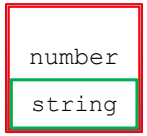
To delete a node at this window position
we re-arrange the links and free the node

LIST: Linked-List Implementation

- type elementtype
- type LIST
- type Boolean
- type windowtype

LIST: Linked-List Implementation

```
/* linked-list implementation of LIST ADT */

#include <stdio.h>
#include <math.h>
#include <string.h>

#define FALSE 0
#define TRUE 1

typedef struct {
          int number;
          char *string;
        } ELEMENT_TYPE;
```



LIST: Linked-List Implementation

```
typedef struct node *NODE_TYPE;
```

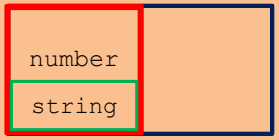
```
typedef struct node {
            ELEMENT_TYPE element;
            NODE_TYPE next;
          } NODE;
```


```
typedef NODE_TYPE LIST_TYPE;
typedef NODE_TYPE WINDOW_TYPE;
```

LIST: Linked-List Implementation

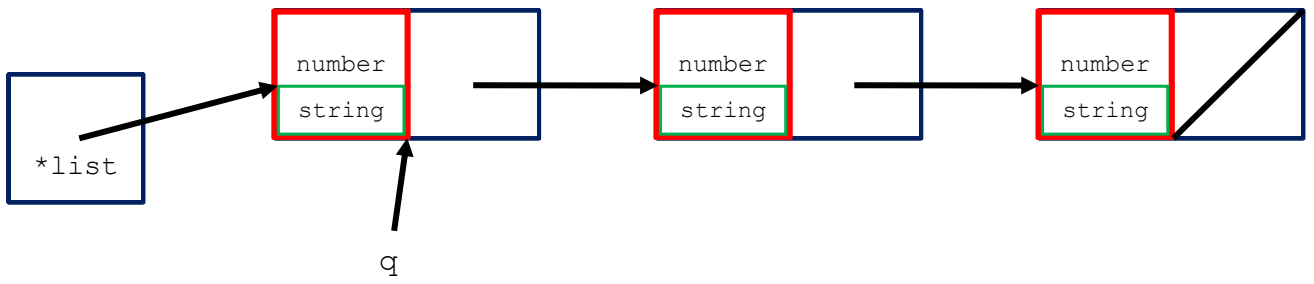
```
typedef struct node *NODE_TYPE;
```

```
/* alternative approach ...                              */
/* but need to use sizeof(struct node) in malloc()*/

struct node {
          ELEMENT_TYPE element;
          NODE_TYPE next;
      };

typedef NODE_TYPE LIST_TYPE;
typedef NODE_TYPE WINDOW_TYPE;
```



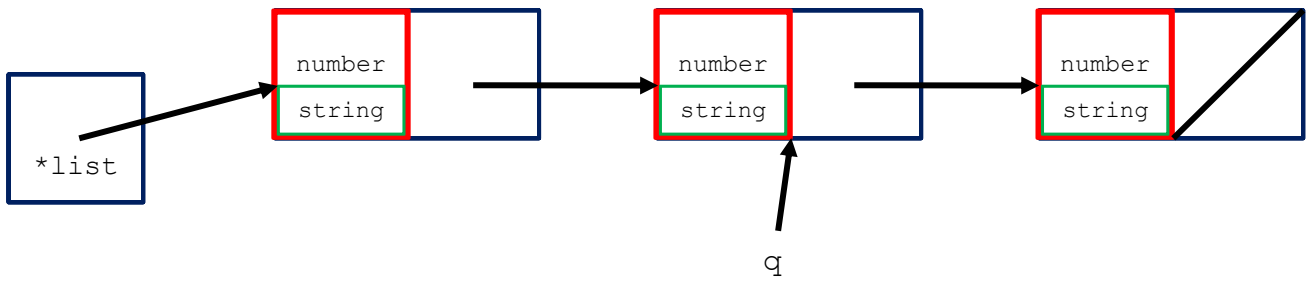
LIST: Linked-List Implementation

```
/*** position following last element in a list ***/

WINDOW_TYPE end(LIST_TYPE *list) {
   WINDOW_TYPE q;
   q = *list;
   if (q == NULL) {
      error("non-existent list");
   }
   else {
      while (q->next != NULL) {
         q = q->next;
      }
   }
   return(q);
}
```



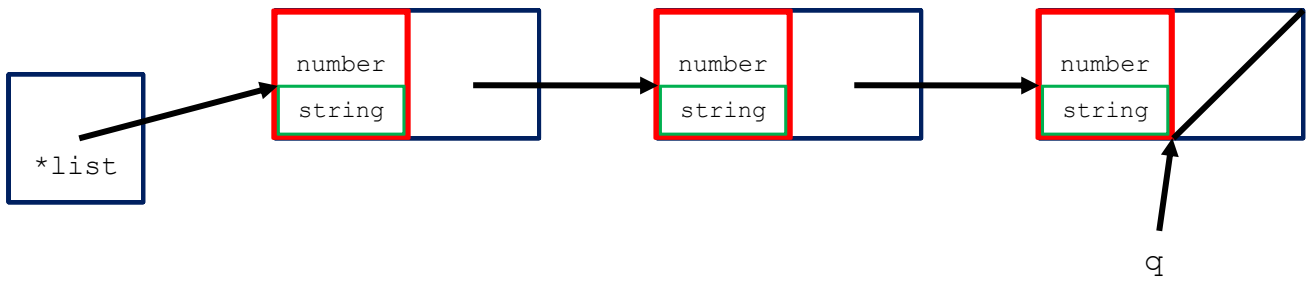
LIST: Linked-List Implementation

```
/*** position following last element in a list ***/

WINDOW_TYPE end(LIST_TYPE *list) {
   WINDOW_TYPE q;
   q = *list;
   if (q == NULL) {
      error("non-existent list");
   }
   else {
      while (q->next != NULL) {
         q = q->next;
      }
   }
   return(q);
}
```



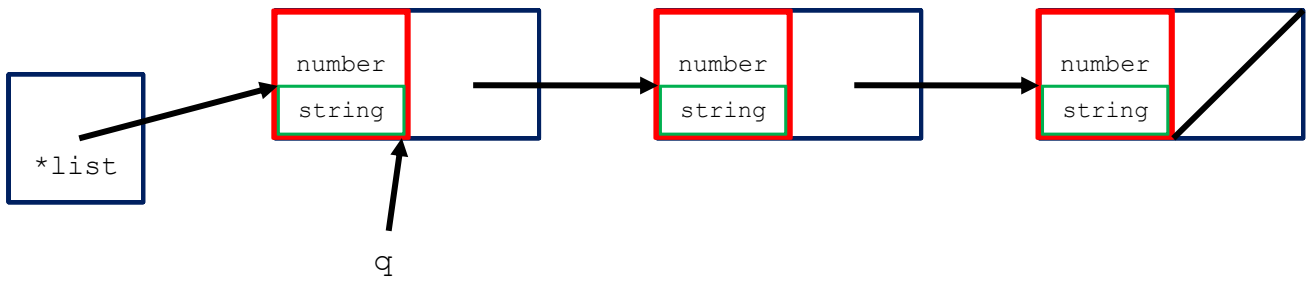
LIST: Linked-List Implementation

```
/*** position following last element in a list ***/

WINDOW_TYPE end(LIST_TYPE *list) {
   WINDOW_TYPE q;
   q = *list;
   if (q == NULL) {
      error("non-existent list");
   }
   else {
      while (q->next != NULL) {
         q = q->next;
      }
   }
   return(q);
}
```



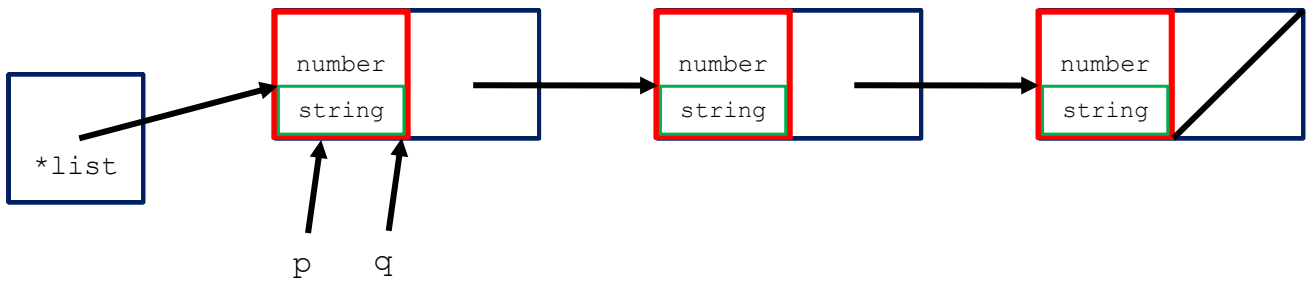
LIST: Linked-List Implementation

```
/*** empty a list ***/

WINDOW_TYPE empty(LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (*list != NULL) {
      /* list exists: delete all nodes including header */
      q = *list;
      while (q->next != NULL) {
         p = q;
         q = q->next;
         free(p);
      }
      free(q)
   }
```



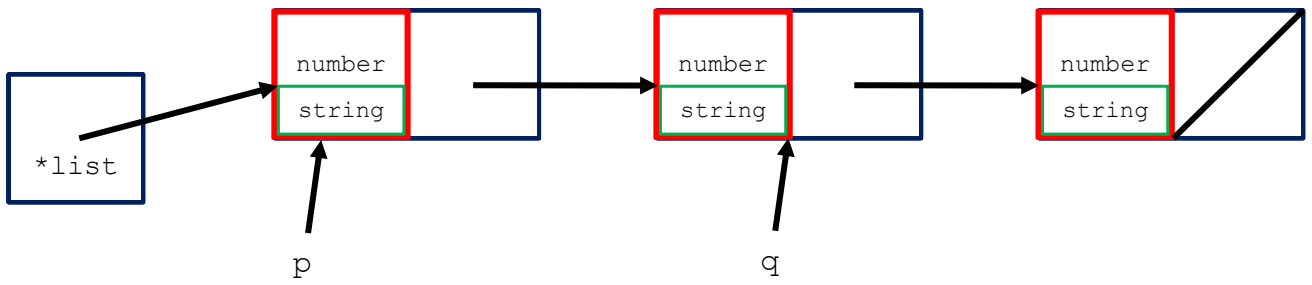
LIST: Linked-List Implementation

```
/*** empty a list ***/

WINDOW_TYPE empty(LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (*list != NULL) {
      /* list exists: delete all nodes including header */
      q = *list;
      while (q->next != NULL) {
         p = q;//temporary hold before deletion
         q = q->next;//advance to next item before deleting
         free(p);
      }
      free(q);
   }
```



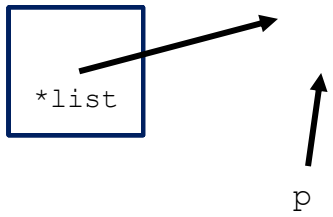
LIST: Linked-List Implementation

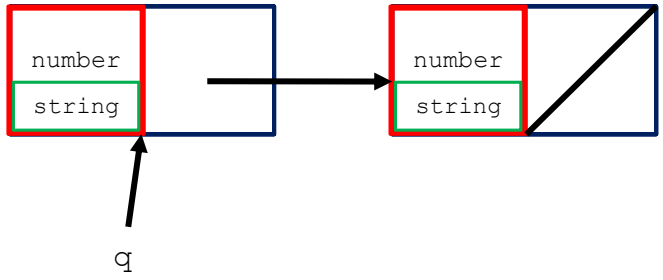
```
/*** empty a list ***/

WINDOW_TYPE empty(LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (*list != NULL) {
      /* list exists: delete all nodes including header */
      q = *list;
      while (q->next != NULL) {
         p = q;
         q = q->next;
         free(p);
      }
      free(q)
   }
```



LIST: Linked-List Implementation

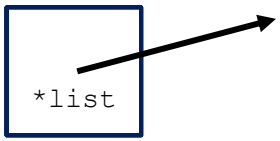
```
/*** empty a list ***/

WINDOW_TYPE empty(LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (*list != NULL) {
      /* list exists: delete all nodes including header */
      q = *list;
      while (q->next != NULL) {
         p = q;
         q = q->next;
         free(p);
      }
      free(q)
   }
```

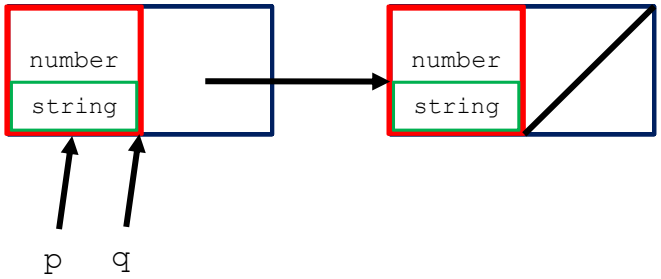




LIST: Linked-List Implementation

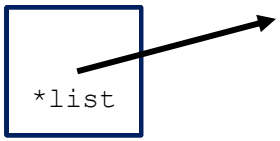
```
/*** empty a list ***/

WINDOW_TYPE empty(LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (*list != NULL) {
      /* list exists: delete all nodes including header */
      q = *list;
      while (q->next != NULL) {
         p = q;
         q = q->next;
         free(p);
      }
      free(q)
   }
```

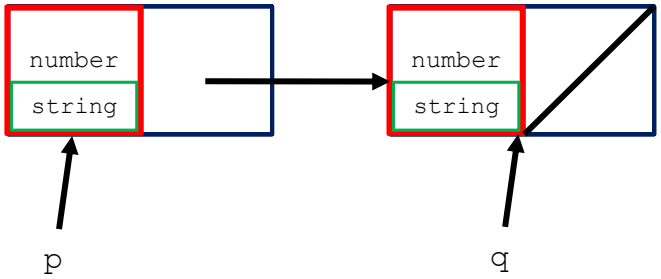




LIST: Linked-List Implementation

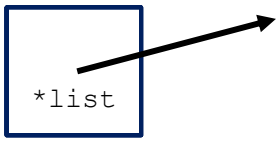
```
/*** empty a list ***/

WINDOW_TYPE empty(LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (*list != NULL) {
      /* list exists: delete all nodes including header */
      q = *list;
      while (q->next != NULL) {
         p = q;
         q = q->next;
         free(p);
      }
      free(q)
   }
```

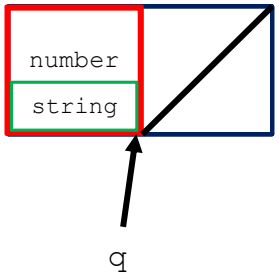




LIST: Linked-List Implementation

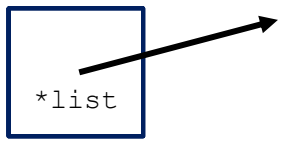
```
/*** empty a list ***/

WINDOW_TYPE empty(LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (*list != NULL) {
      /* list exists: delete all nodes including header */
      q = *list;
      while (q->next != NULL) {
         p = q;
         q = q->next;
         free(p);
      }
      free(q)
   }
```





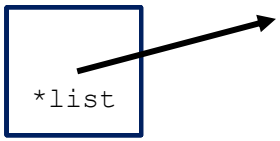
LIST: Linked-List Implementation

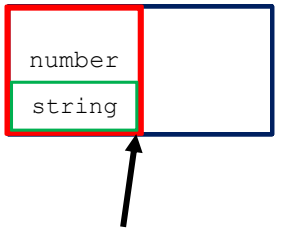
```
/*** empty a list ***/

WINDOW_TYPE empty(LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (*list != NULL) {
      /* list exists: delete all nodes including header */
      q = *list;
      while (q->next != NULL) {
         p = q;
         q = q->next;
         free(p);
      }
      free(q)
   }
```



LIST: Linked-List Implementation

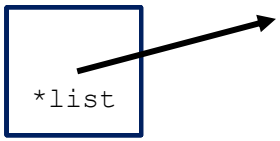
```
   /* now, create a new empty one with a header node */

   if ((q = (NODE_TYPE) malloc(sizeof(NODE))) == NULL)
      error("function empty: unable to allocate memory");
   else {
      q->next = NULL;
      *list = q;
   }
   return(end(list));
}
```

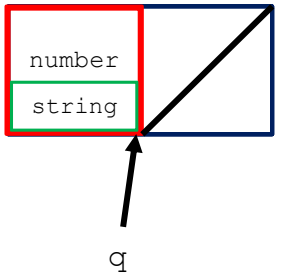




LIST: Linked-List Implementation

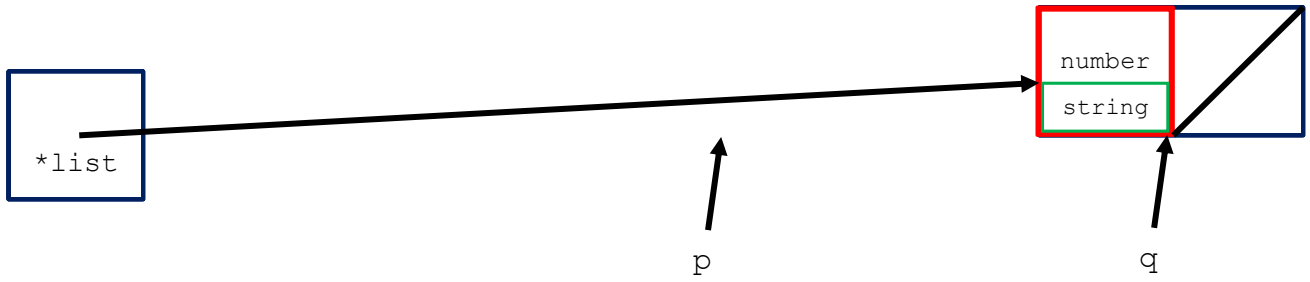
```
   /* now, create a new empty one with a header node */

   if ((q = (NODE_TYPE) malloc(sizeof(NODE))) == NULL)
      error("function empty: unable to allocate memory");
   else {
      q->next = NULL;
      *list = q;
   }
   return(end(list));
}
```





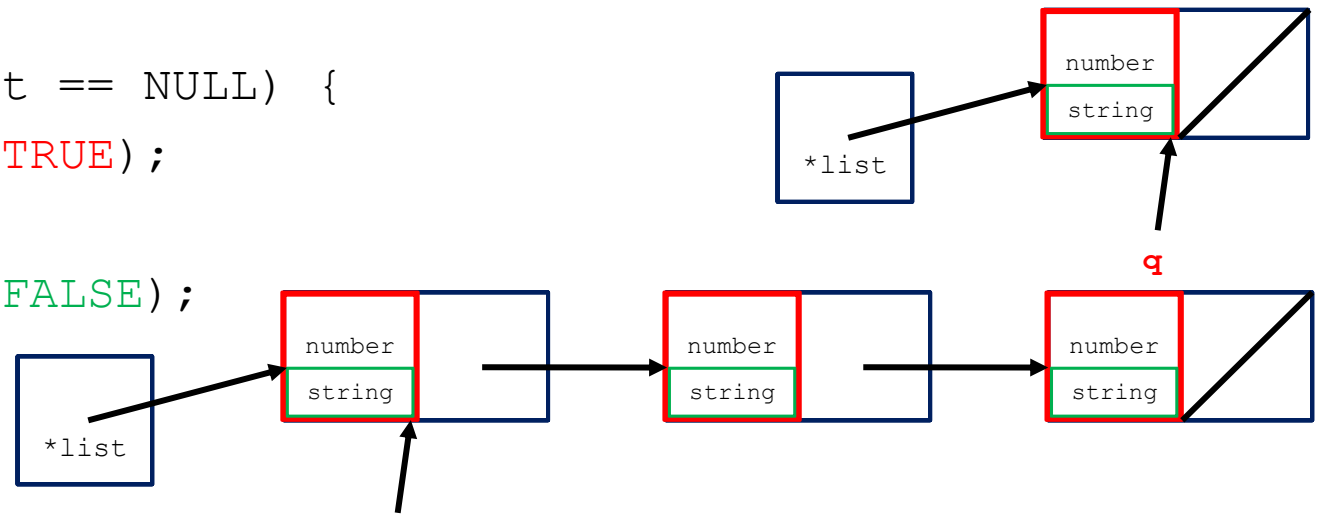
LIST: Linked-List Implementation

```
   /* now, create a new empty one with a header node */

   if ((q = (NODE_TYPE) malloc(sizeof(NODE))) == NULL)
      error("function empty: unable to allocate memory");
   else {
      q->next = NULL;
      *list = q;
   }
   return(end(list));
}
```



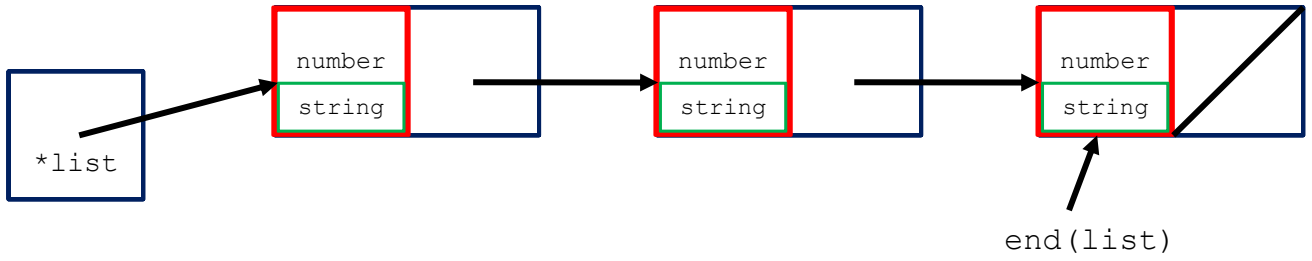
LIST: Linked-List Implementation

```
/*** test to see if a list is empty ***/

int is_empty(LIST_TYPE *list) {
    WINDOW_TYPE q;
    q = *list;
    if (q == NULL) {
        error("non-existent list");
    }
    else {
        if (q->next == NULL) {
            return(TRUE);
        else
            return(FALSE);
    }
}
```



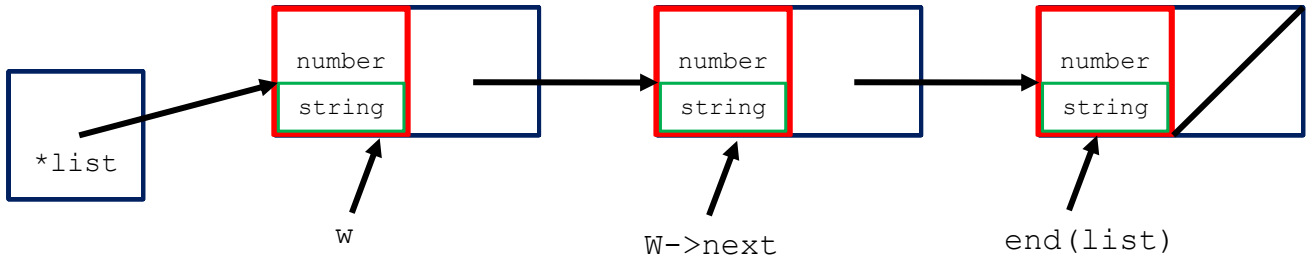
LIST: Linked-List Implementation

```
/*** position at first element in a list ***/

WINDOW_TYPE first(LIST_TYPE *list) {
   if (is_empty(list) == FALSE) {
      return(*list);
   else
      return(end(list));
}
```



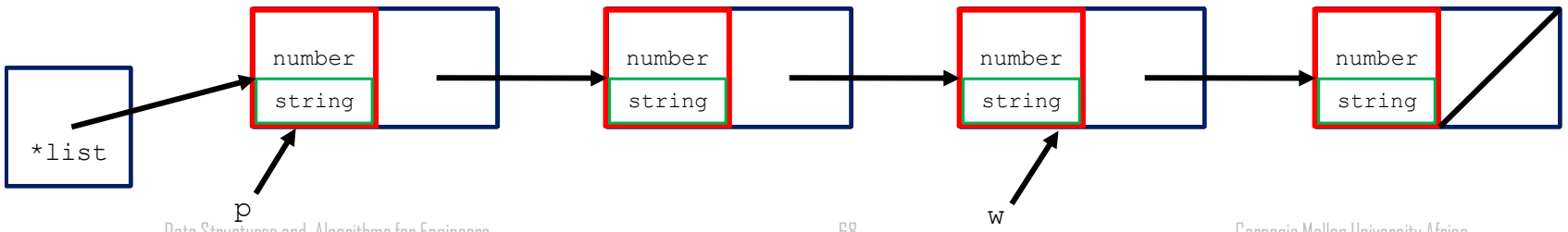
LIST: Linked-List Implementation

```
/*** position at next element in a list ***/

WINDOW_TYPE next(WINDOW_TYPE w, LIST_TYPE *list) {
   if (w == last(list)) {
      return(end(list));
   }
   else if (w == end(list)) {
      error("can't find next after end of list");
   }
   else {
      return(w->next);
   }
}
```



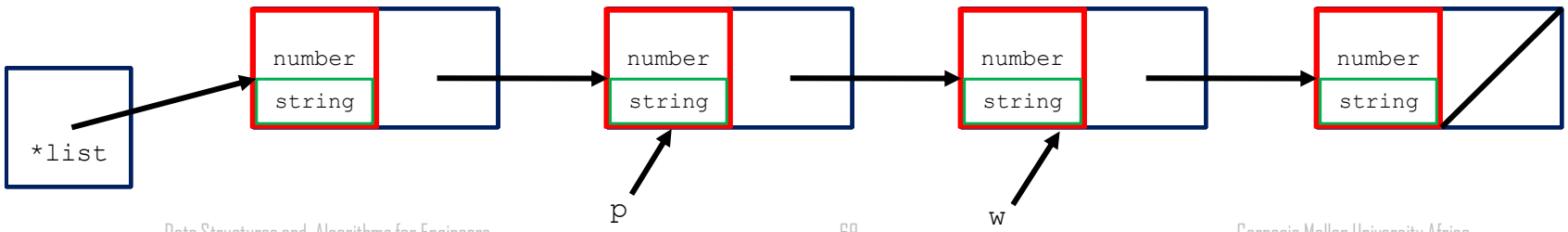
LIST: Linked-List Implementation

```
/*** position at previous element in a list ***/

WINDOW_TYPE previous(WINDOW_TYPE w, LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (w != first(list)) {
      p = first(list);
      while (p->next != w) {
         p = p->next;
         if (p == NULL) break; /* trap this to ensure   */
      }                        /* we don't dereference  */
      if (p != NULL)           /* a null pointer in the */
         return(p);            /* while condition       */
```



LIST: Linked-List Implementation

```
/*** position at previous element in a list ***/

WINDOW_TYPE previous(WINDOW_TYPE w, LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (w != first(list)) {
      p = first(list);
      while (p->next != w) {
         p = p->next;
         if (p == NULL) break; /* trap this to ensure   */
      }                        /* we don't dereference  */
      if (p != NULL)           /* a null pointer in the */
         return(p);            /* while condition       */
```



LIST: Linked-List Implementation

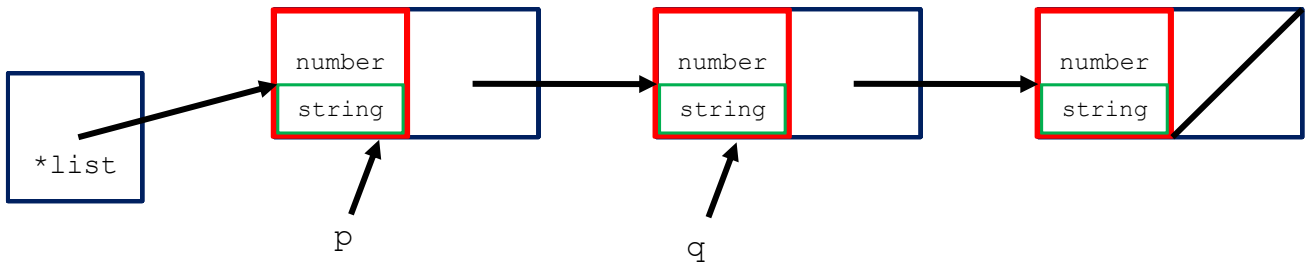
```
        else {
            error("can't find previous to a non-existent node");
        }
    }
    else {
        error("can't find previous before first element of list");
        return(w);
    }
}
```

LIST: Linked-List Implementation

```
/*** position at last element in a list ***/

WINDOW_TYPE last(LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (*list == NULL) {
      error("non-existent list");
   }
   else {
      /* list exists: find last node */
```

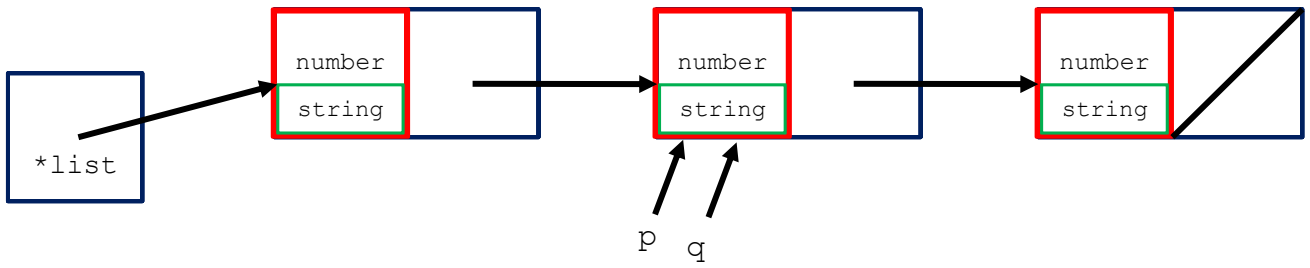
LIST: Linked-List Implementation

```
    /* list exists: find last node */

    if (is_empty(list)) {
       p = end(list);
    }
    else {
       p = *list;
       q = p->next;
       while (q->next != NULL) {
          p = q;
          q = q->next;
       }
    }
    return(p);
  }
}
```



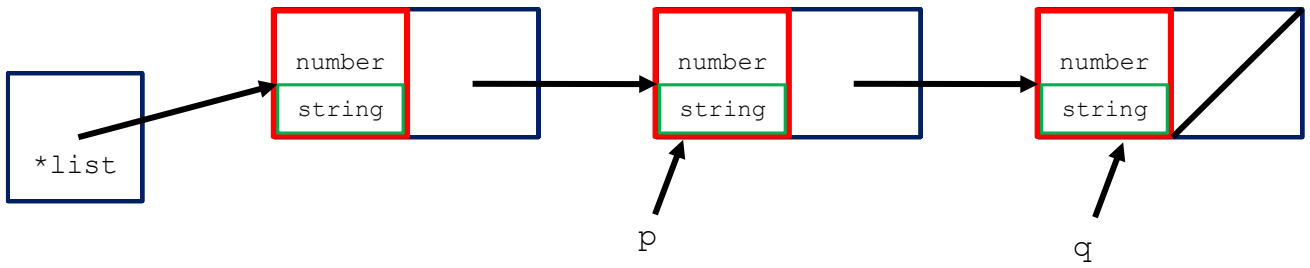
LIST: Linked-List Implementation

```
    /* list exists: find last node */

    if (is_empty(list)) {
       p = end(list);
    }
    else {
       p = *list;
       q = p->next;
       while (q->next != NULL) {
          p = q;
          q = q->next;
       }
    }
    return(p);
  }
}
```



LIST: Linked-List Implementation

```
    /* list exists: find last node */

    if (is_empty(list)) {
       p = end(list);
    }
    else {
       p = *list;
       q = p->next;
       while (q->next != NULL) {
          p = q;
          q = q->next;
       }
    }
    return(p);
  }
}
```

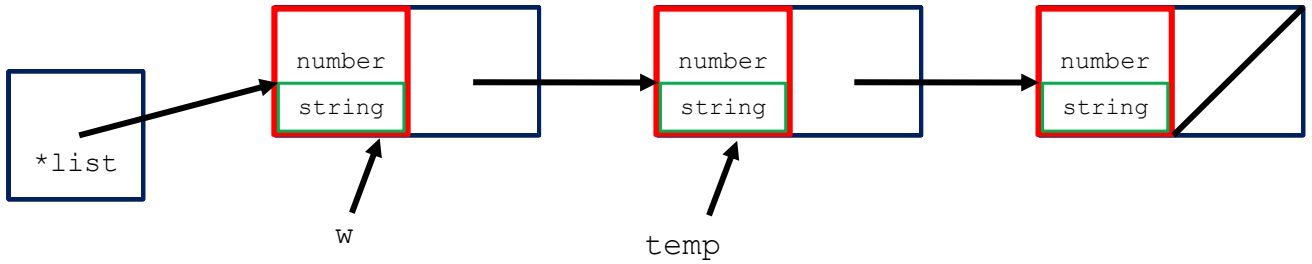


LIST: Linked-List Implementation

```
/*** insert an element in a list ***/

LIST_TYPE *insert(ELEMENT_TYPE e, WINDOW_TYPE w,
                  LIST_TYPE *list) {
   WINDOW_TYPE temp;
   if (*list == NULL) {
      error("cannot insert in a non-existent list");
   }
```

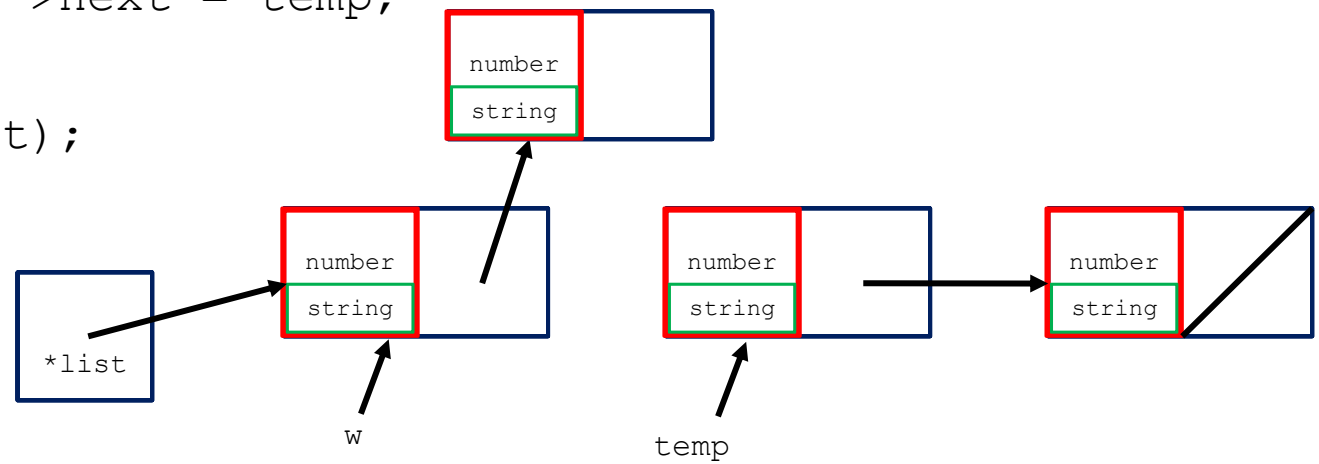
LIST: Linked-List Implementation

```
    else {
        /* insert it after w */
        temp = w->next;
        if ((w->next = (NODE_TYPE) malloc(sizeof(NODE))) == NULL)
            error("function insert: unable to allocate memory");
        else {
            w->next->element = e;
            w->next->next = temp;
        }
        return(list);
    }
}
```



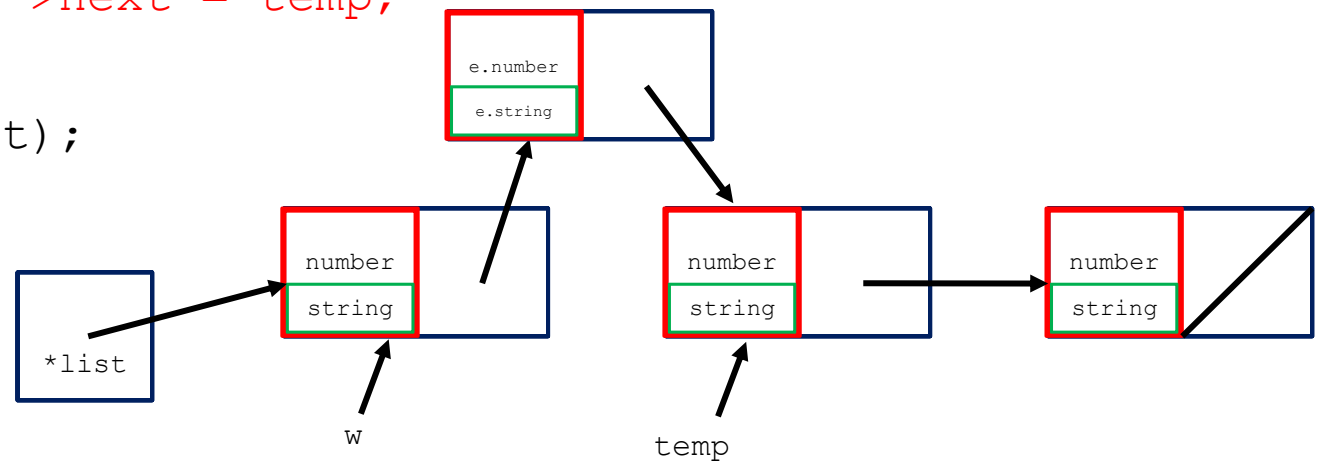
LIST: Linked-List Implementation

```
    else {
        /* insert it after w */
        temp = w->next;
        if ((w->next = (NODE_TYPE) malloc(sizeof(NODE))) = NULL)
            error("function insert: unable to allocate memory");
        else {
            w->next->element = e;
            w->next->next = temp;
        }
        return(list);
    }
}
```



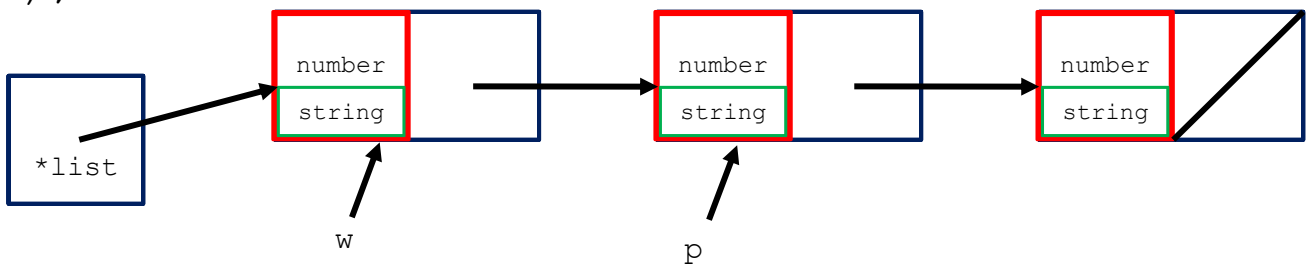
LIST: Linked-List Implementation

```
    else {
        /* insert it after w */
        temp = w->next;
        if ((w->next = (NODE_TYPE) malloc(sizeof(NODE))) = NULL)
            error("function insert: unable to allocate memory");
        else {
            w->next->element = e;
            w->next->next = temp;
        }
        return(list);
    }
}
```



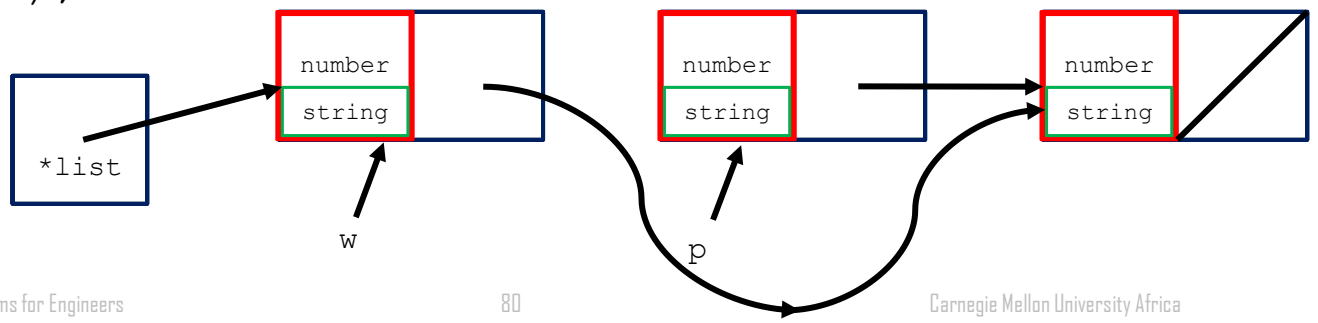
LIST: Linked-List Implementation

```
/*** delete an element from a list ***/

LIST_TYPE *delete(WINDOW_TYPE w, LIST_TYPE *list) {
   WINDOW_TYPE p;
   if (*list == NULL) {
      error("cannot delete from a non-existent list");
   }
   else {
      p = w->next; /* node to be deleted */
      w->next = w->next->next;  /* rearrange the links */
      free(p);      /* delete the node */
      return(list);
   }
}
```



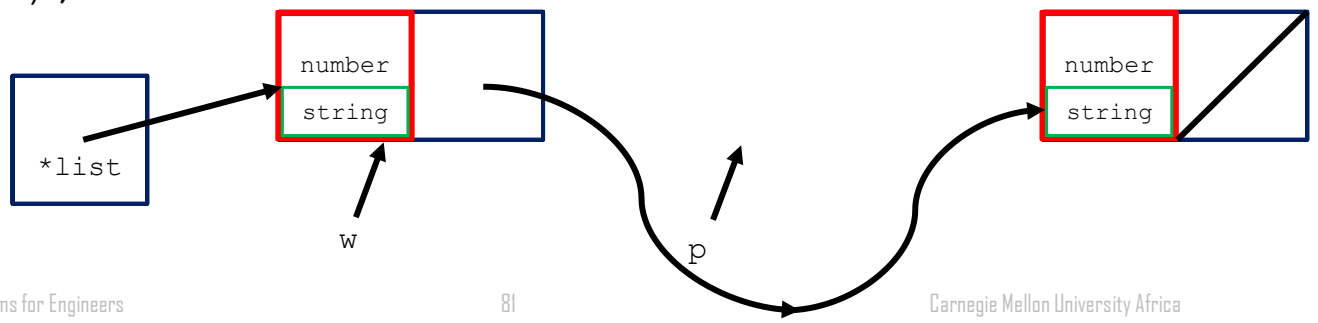
LIST: Linked-List Implementation

```
/*** delete an element from a list ***/

LIST_TYPE *delete(WINDOW_TYPE w, LIST_TYPE *list) {
   WINDOW_TYPE p;
   if (*list == NULL) {
      error("cannot delete from a non-existent list");
   }
   else {
      p = w->next; /* node to be deleted */
      w->next = w->next->next;  /* rearrange the links */
      free(p);      /* delete the node */
      return(list);
   }
}
```



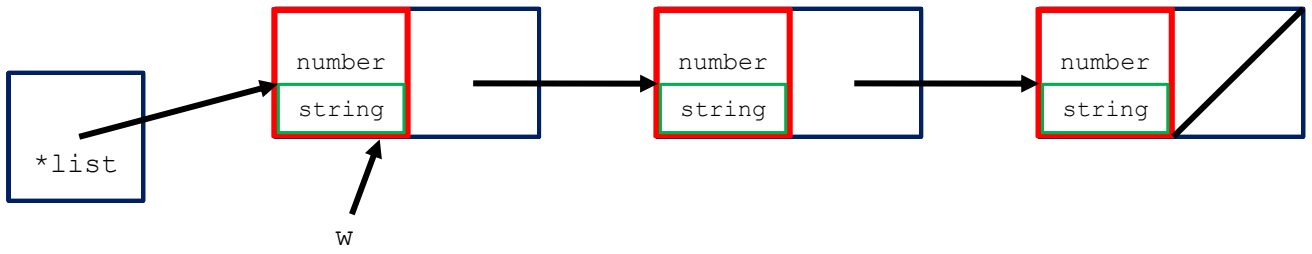
LIST: Linked-List Implementation

```
/*** delete an element from a list ***/

LIST_TYPE *delete(WINDOW_TYPE w, LIST_TYPE *list) {
   WINDOW_TYPE p;
   if (*list == NULL) {
      error("cannot delete from a non-existent list");
   }
   else {
      p = w->next; /* node to be deleted */
      w->next = w->next->next;  /* rearrange the links */
      free(p);      /* delete the node */
      return(list);
   }
}
```



LIST: Linked-List Implementation

```
/*** retrieve an element from a list ***/

ELEMENT_TYPE retrieve(WINDOW_TYPE w, LIST_TYPE *list) {
   WINDOW_TYPE p;

   if (*list == NULL) {
      error("cannot retrieve from a non-existent list");
   }
   else {
      return(w->next->element);
   }
}
```



LIST: Linked-List Implementation

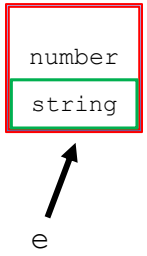
```
/*** print all elements in a list ***/

int print(LIST_TYPE *list) {
   WINDOW_TYPE w;
   ELEMENT_TYPE e;

   printf("Contents of list: \n");
   w = first(list);
   while (w != end(list)) {
      e = retrieve(w, *list);
      printf("%d %s\n", e.number, e.string);
      w = next(w, list);
   }
   printf("---\n");
   return(0);
}
```

LIST: Linked-List Implementation

```
/*** error handler: print message passed as argument and
     take appropriate action                              ***/
int error(char *s); {
   printf("Error: %s\n", s);
   exit(0);
}
```

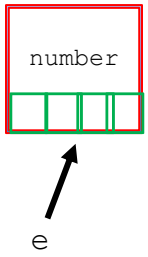
LIST: Linked-List Implementation

```
/*** assign values to an element ***/

int assign_element_values(ELEMENT_TYPE *e, int number, char s[])
   {
    e->string = (char *) malloc(sizeof(char) * (strlen(s)+1));
    strcpy(e->string, s);
    e->number = number;
}
```



LIST: Linked-List Implementation

```
/*** assign values to an element ***/

int assign_element_values(ELEMENT_TYPE *e, int number, char s[])
   {
   e->string = (char *) malloc(sizeof(char) * (strlen(s)+1));
   strcpy(e->string, s);
   e->number = number;
}
```



LIST: Linked-List Implementation

```
/*** initialize the list pointer to make sure     ***/
/*** all subsequent checks on its value are valid ***/

void initialize_list(LIST_TYPE *list) {
  *list = NULL;
}
```

LIST: Linked-List Implementation

```
/*** main driver routine ***/

   WINDOW_TYPE w;
   ELEMENT_TYPE e;
   LIST_TYPE list;
   int i;

   initialize_list(&list);
   empty(&list);
   print(&list);

   assign_element_values(&e, 1, "String A");
   w = first(&list);
   insert(e, w, &list);
   print(&list);
```

LIST: Linked-List Implementation

```
assign_element_values(&e, 2, “String B”);
insert(e, w, &list);
print(&list);

assign_element_values(&e, 3, “String C”);
insert(e, last(&list), &list);
print(&list);

assign_element_values(&e, 4, “String D”);
w = next(last(&list), &list);
insert(e, w, &list);
print(&list);
```

LIST: Linked-List Implementation

```
    w = previous(w, &list);
    delete(w, &list);
    print(&list);

}
```

LIST: Linked-List Implementation

Key points:

- All we changed was the implementation of the data-structure and the access routines
- But by keeping the interface to the access routines the same as before, these changes are transparent to the user
- And we didn't have to make any changes in the main function which was actually manipulating the list

LIST: Linked-List Implementation

Key points:

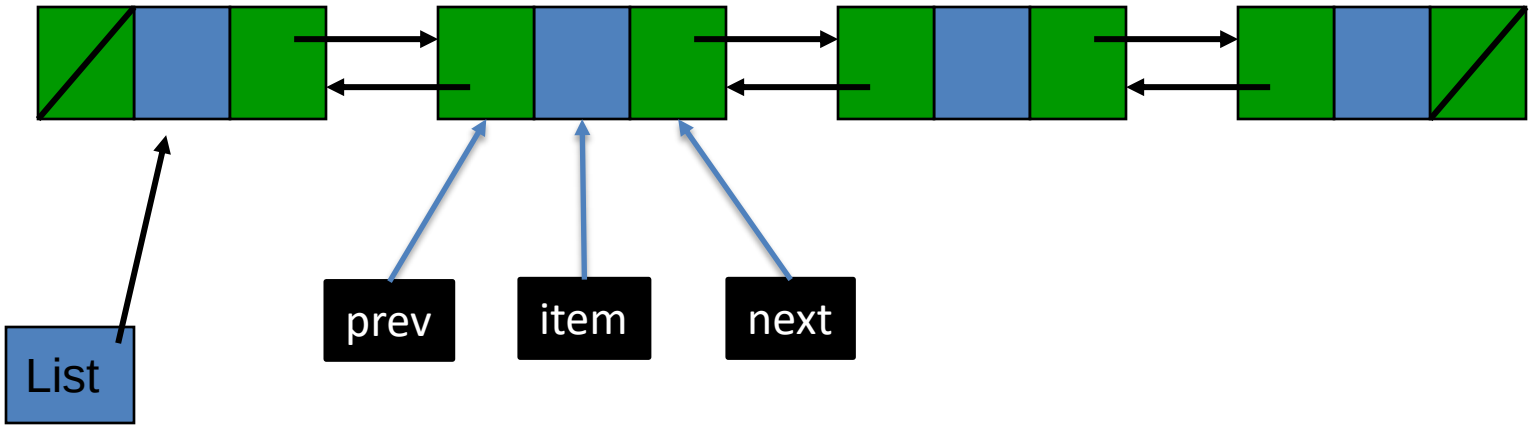
- In a real software system where perhaps hundreds (or thousands) of people are using these list primitives, this transparency is critical
- We couldn't have achieved it if we manipulated the data-structure directly

LIST: Linked-List Implementation

- Possible problems with the implementation:
  - we have to run the length of the list in order to find the end (i.e. end(L) is O(n))
  - there is a (small) overhead in using the pointers
- On the other hand, the list can now grow as large as necessary, without having to predefine the maximum size

LIST: Linked-List Implementation

Each node contains the list item and two pointers- next and previous.

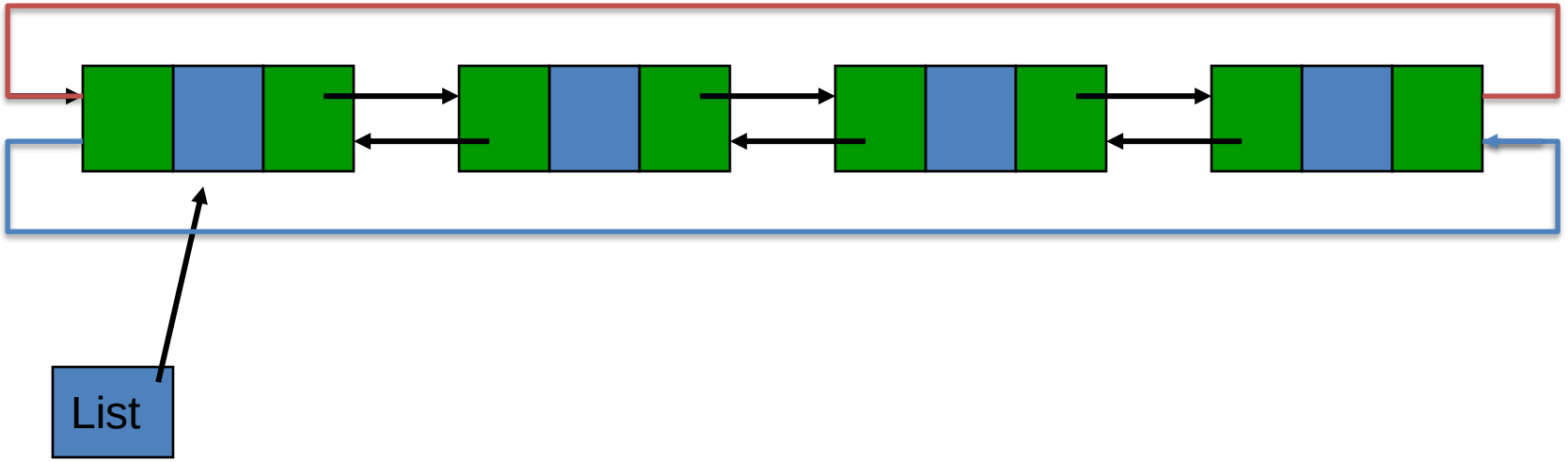


We can also have a doubly-linked list;
this removes the need to have a header node
and make finding the previous node more efficient.

However, we could maintain the Head node as a means of list traversal.

LIST: Linked-List Implementation

Lists can also be circular



First element points at last element and last element points at first element.

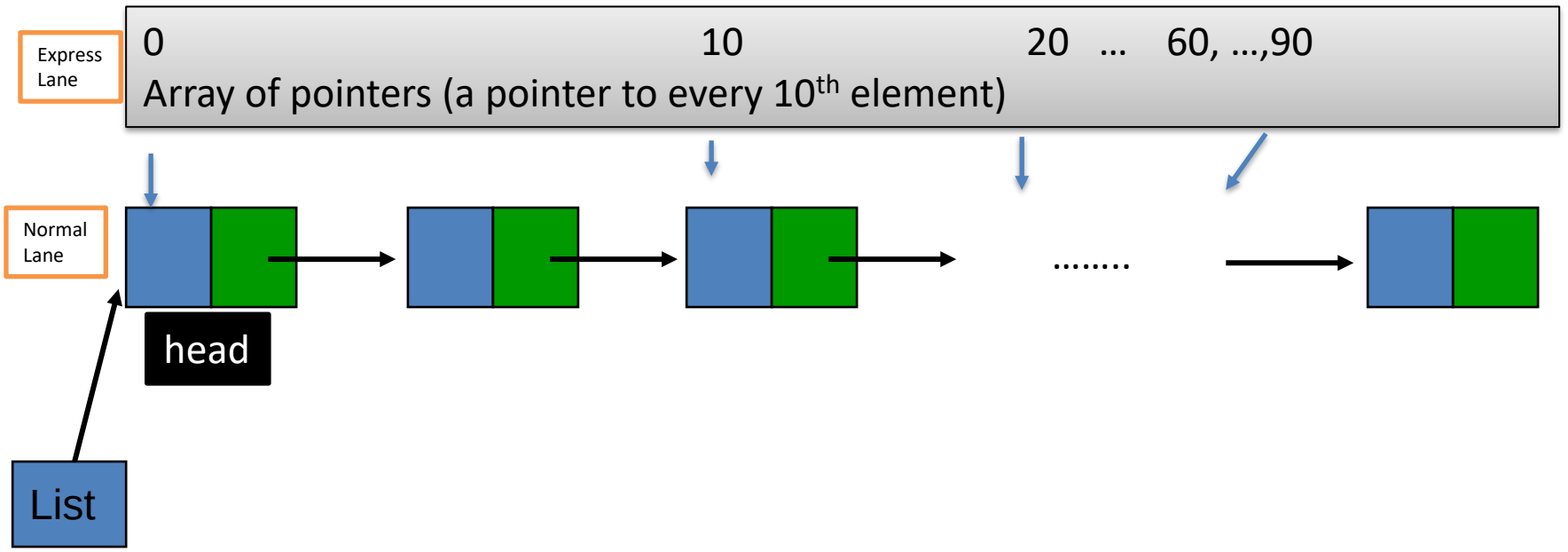
Both ***singly-*** and ***doubly-linked*** lists can be made circular lists.

LIST: Linked-List Implementation

Linked lists can be ***slow for sort or search operations***, because they rely on linear access.

Skip-linked lists maintain pointers to some or all elements.

- Designed to speed up ***sort or search*** operations.
- Consider a list of 100 items



Lists can also be skip-linked lists

Review Exercises

- Implementation of:
  - Doubly-linked lists
  - Circular linked lists
  - Skip linked lists

- Complexity analysis of:
  - Insert, retrieve, and delete operations under different circumstances, e.g. after some node, before some node, at the head, at the back, etc.

Comparison: Linked Lists vs. Arrays

- Relative advantages of linked lists
  - Overflow on linked structures can never occur unless memory is actually full
  - Insertions and deletions are simpler than for contiguous (array) lists
  - With large records, moving pointers is easier and faster than moving the items themselves

Comparison: Linked Lists vs. Arrays

- Relative advantages of arrays
  - Linked structures require extra space for storing pointer fields
  - Linked lists do not allow efficient random access to items(due to linear access)
  - Arrays allow better memory locality and cache performance than random pointer jumping
- Dynamic memory allocation provides us with flexibility on how and where to use limited storage resources

Comparison: Linked Lists vs. Arrays

- Both lists and arrays can be thought of a recursive objects:
  - Lists: chopping off the first element of a linked list leaves a smaller linked list
  - Lists are recursive objects
  - Splitting the first *k* elements off an n element array give two smaller arrays, of size *k* and *n-k*, respectively
  - Arrays are recursive objects
  - This shows us that lists are amenable to efficient (recursive) divide-and-conquer algorithms, such as binary search and quicksort

Practical Application

- Linked lists are great for implementing queues and stacks.

Acknowledgement

- Adopted and Adapted from Material by:
- David Vernon: vernon@cmu.edu ; www.vernon.eu