Data Structures

Data Structures

- Lists
- Stacks (special type of list)
- Queues (another type of list)
- Trees
  - General introduction
  - Binary Trees
  - Binary Search Trees (BST)
- Use *Abstract Data Types* (ADT)

Abstract Data Types

- ADTs are an old concept
  - Specify the complete set of values which a variable of this *type* may assume
  - Specify completely the set of all possible operations which can be applied to values of this *type*

Abstract Data Types

- It's worth noting that object-oriented programming gives us a way of combining (or **encapsulating**) both of these specifications in one logical definition
  - **Class** definition
  - **Objects** are instantiated classes
- Actually, object-oriented programming provides much more than this (e.g. inheritance and polymorphism)

Lists

Lists

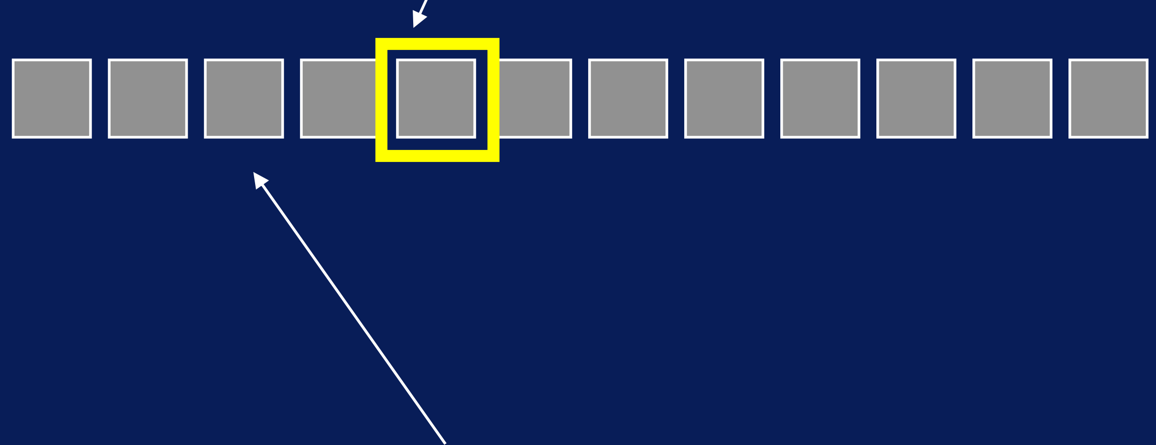
- A list is an ordered sequence of zero or more elements of a given type

  $a_1, a_2, a_3, \ldots a_n$

  - a_i is of type *elementtype*
  - a_i precedes a_{i+1}
  - a_{i+1} succeeds or follows a_i
  - If n=0 the list is empty: a null list

Lists

List element w
(w is of type windowtype:
w could be, but is not necessarily,
the integer sequence position of
the element in the list)



Element of type elementtype

LIST: An ADT specification of a list type

- Let L denote all possible values of type LIST (*i.e.* lists of elements of type *elementtype)*
- Let E denote all possible values of type *elementtype*
- Let B denote the set of Boolean values *true* and *false*
- Let *W* denote the set of values of type *windowtype*

LIST Operations

- *Syntax of ADT Definition*:

  Operation:

  What_You_Pass_It → What_It_Returns :

LIST Operations

- *Declare*: $\rightarrow$ L :

  The function value of *Declare(L)* is an empty list

  – alternative syntax: *LIST L*

LIST Operations

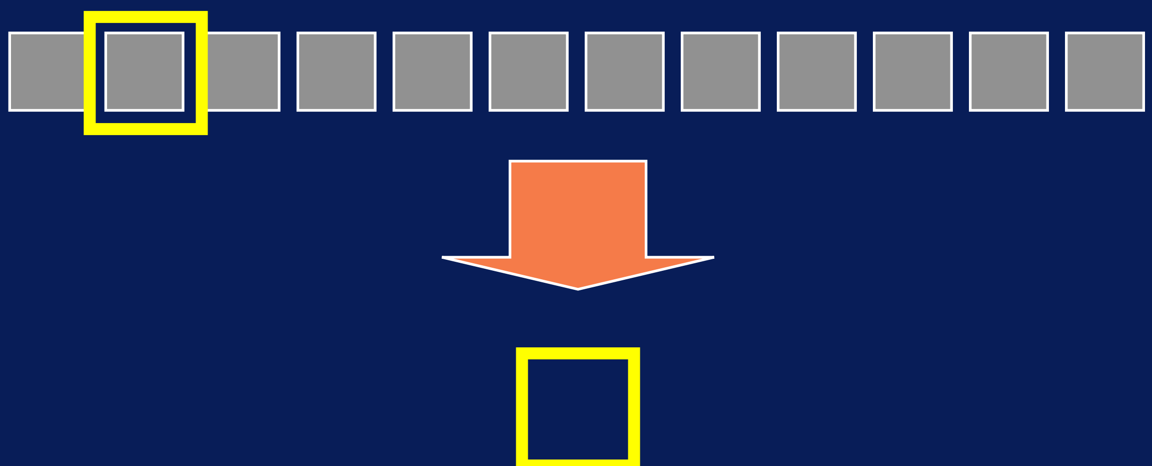
- *End*: L $\rightarrow$ W :

  The function *End(L)* returns the position <u>after</u> the last element in the list
  (i.e. the value of the function is the window position after the last element in the list)

LIST Operations

- *Empty*: L $\rightarrow$ L x W :

  The function *Empty* causes the list to be emptied and it returns position *End(L)*



LIST Operations

- *IsEmpty*: L $\rightarrow$ B :

  The function value *IsEmpty*(*L*) is *true* if *L* is empty; otherwise it is *false*

LIST Operations

- *First*: L $\rightarrow$ W :

  The function value *First*(*L*) is the window position of the first element in the list;

  if the list is empty, it has the value *End(L)*

LIST Operations

- *Next*: $L \times W \rightarrow W$:

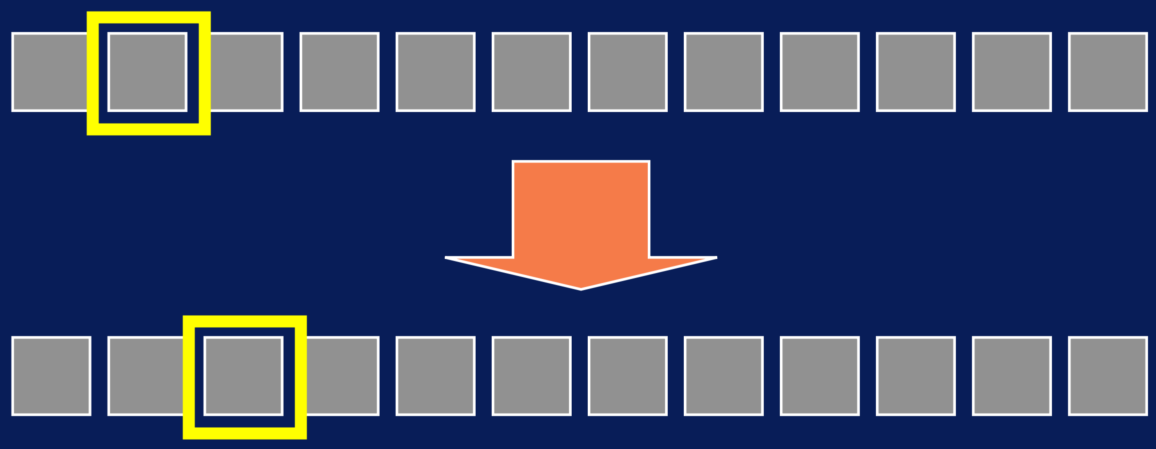
  The function value *Next*(*w*,*L*) is the window position of the next successive element in the list;

  if we are already at the end of the list then the value of *Next*(*w,L*) is *End(L);*

  if the value of *w* is *End(L)*, then the operation is undefined.

LIST Operations

Next(*w*,*L*)



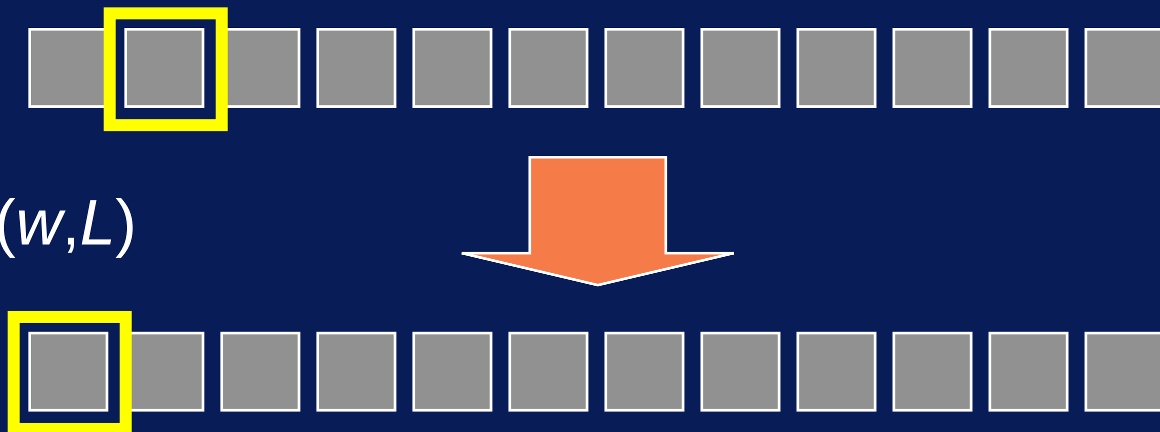
LIST Operations

- *Previous*: L × W → W :

  The function value *Previous*(w, *L*) is the window position of the previous element in the list;

  if we are already at the beginning of the list (*w*=*First*(*L*)), then the value is undefined

LIST Operations



Previous(*w*,*L*)

LIST Operations

- *Last*: L → W :

  The function value *Last*(*L*) is the window position of the last element in the list;

  if the list is empty, it has the value *End(L)*

LIST Operations

- *Insert*: $E \times L \times W \rightarrow L \times W$:

  Insert(e, w, L)
  Insert an element *e* at position *w* in the list *L*, moving elements at *w* and following positions to the next higher position

  $a_1, a_2, \ldots a_n \rightarrow a_1, a_2, \ldots, a_{w-1}, e, a_w, \ldots, a_n$

LIST Operations

If w = *End(L)* then

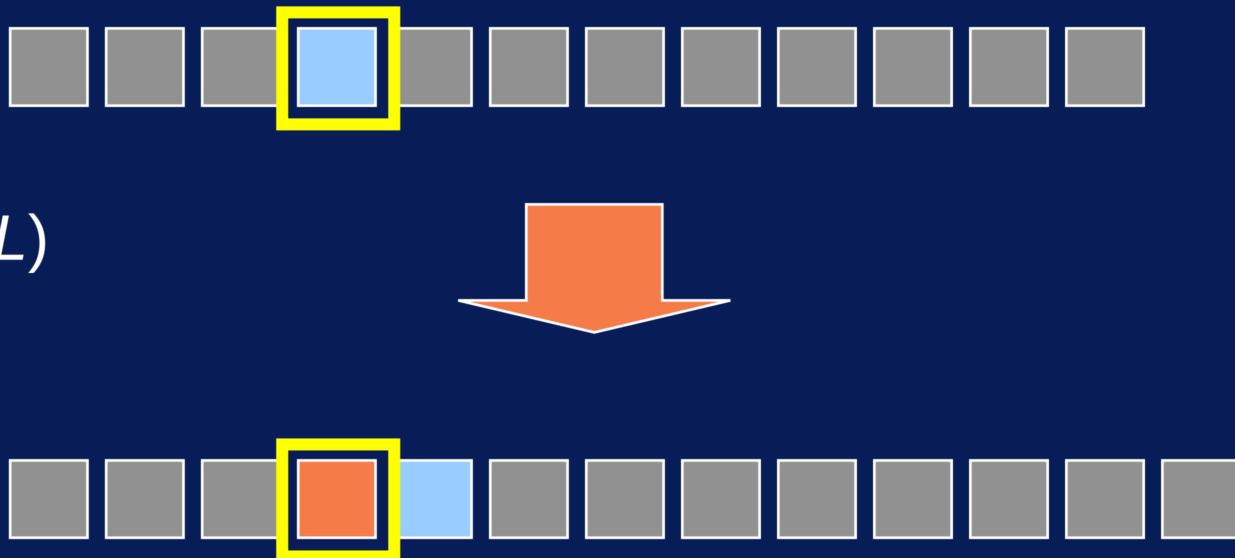
$a_1, a_2, \ldots a_n \rightarrow a_1, a_2, \ldots, a_n, e$

The window *w* is moved over the new element *e*

The function value is the list with the element inserted

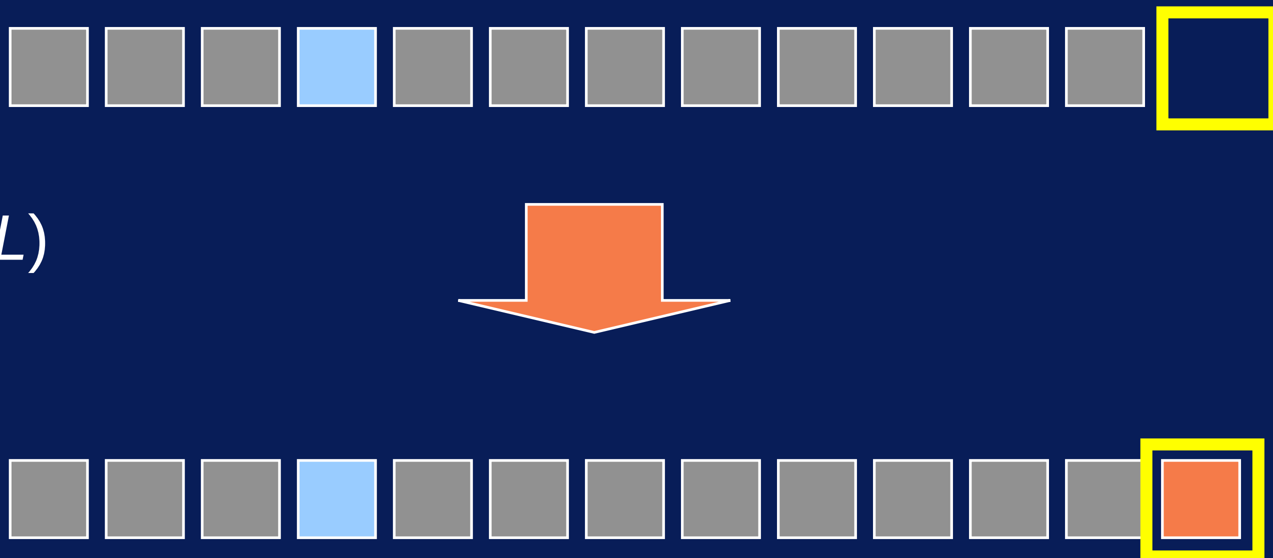
LIST Operations

Insert(*e*,*w*,*L*)



LIST Operations

Insert(*e*,*w*,*L*)



LIST Operations

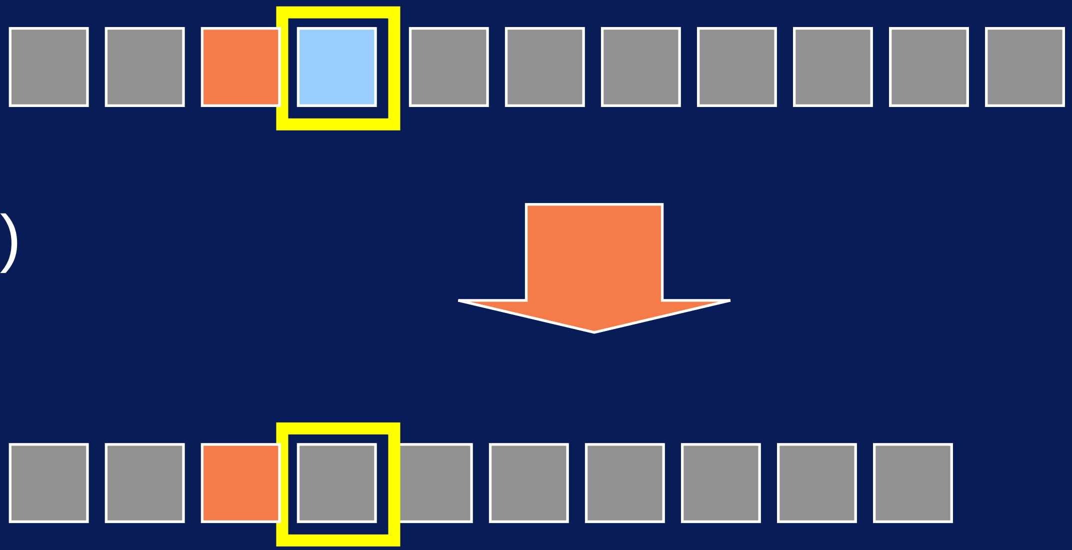
- *Delete*: L × W → L × W :

  Delete(w, L)
  Delete the element at position *w* in the list *L*

  $a_1, a_2, \ldots\ a_n \rightarrow a_1, a_2, \ldots, a_{w-1}, a_{w+1}, \ldots, a_n$

  - If *w* = *End(L)* then the operation is undefined
  - The function value is the list with the element deleted

LIST Operations



Delete(*w*,*L*)

LIST Operations

- *Examine*: $L \times W \rightarrow E$:

  The function value *Examine*(*w*, *L*) is the value of the element at position *w* in the list;

  if we are already at the end of the list (*i.e.* w = *End(L)),* then the value is undefined

LIST Operations

- *Declare(L)* *returns listtype*
- *End(L)* *returns windowtype*
- *Empty(L)* *returns windowtype*
- *IsEmpty(L)* *returns Boolean*
- *First(L)* *returns windowtype*
- *Next(w,L)* *returns windowtype*
- *Previous(w,L)* *returns windowtype*
- *Last(L)* *returns windowtype*

LIST Operations

- *Insert(e,w,L)* *returns listtype*
- *Delete(w,L)* *returns listtype*
- *Examine(w,L)* *returns elementtype*

LIST Operations

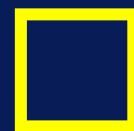
- *Example of List manipulation*

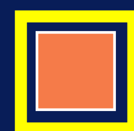
w = End(L)

empty list

LIST Operations

- *Example of List manipulation*

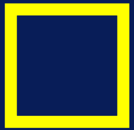
w = End(L) 

Insert(e,w, L) 

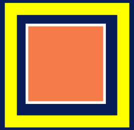
LIST Operations

- *Example of List manipulation*

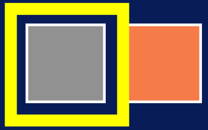
w = End(L)



Insert(e,w, L)



Insert(e,w, L)



LIST Operations

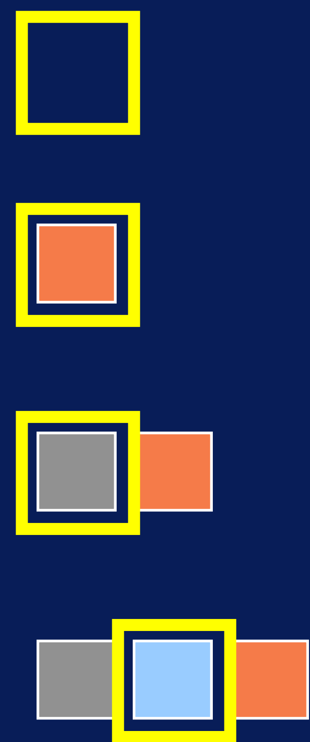
- *Example of List manipulation*

w = End(L)

Insert(e,w, L)

Insert(e,w, L)

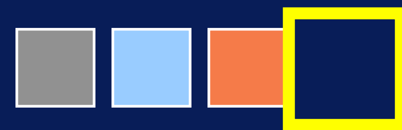
Insert(e,Last(L), L)



LIST Operations

- *Example of List manipulation*

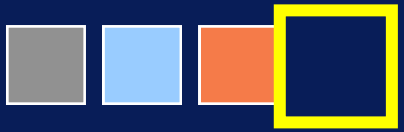
w = Next(Last(L),L)



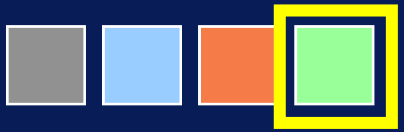
LIST Operations

- *Example of List manipulation*

w = Next(Last(L),L)



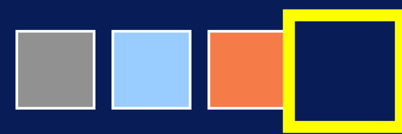
Insert(e,w,L)



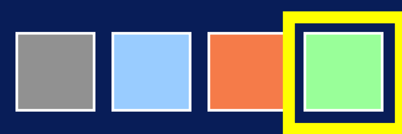
LIST Operations

- *Example of List manipulation*

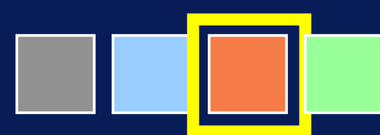
w = Next(Last(L),L)



Insert(e,w,L)



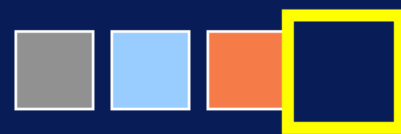
w = Previous(w,L)



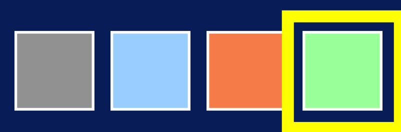
LIST Operations

- *Example of List manipulation*

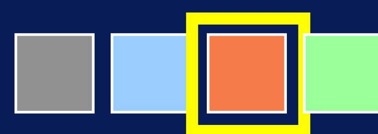
w = Next(Last(L),L)



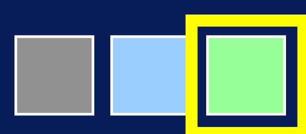
Insert(e,w,L)



w = Previous(w,L)



Delete(w,L)



ADT Specification

- The key idea is that we have not specified how the lists are to be implemented, merely their values and the operations of which they can be operands
- This ‘old’ idea of data abstraction is one of the key features of object-oriented programming
- C++ is a particular implementation of this object oriented methodology

ADT Implementation

- Of course, we still have to implement this ADT specification
- The choice of implementation will depend on the requirements of the application

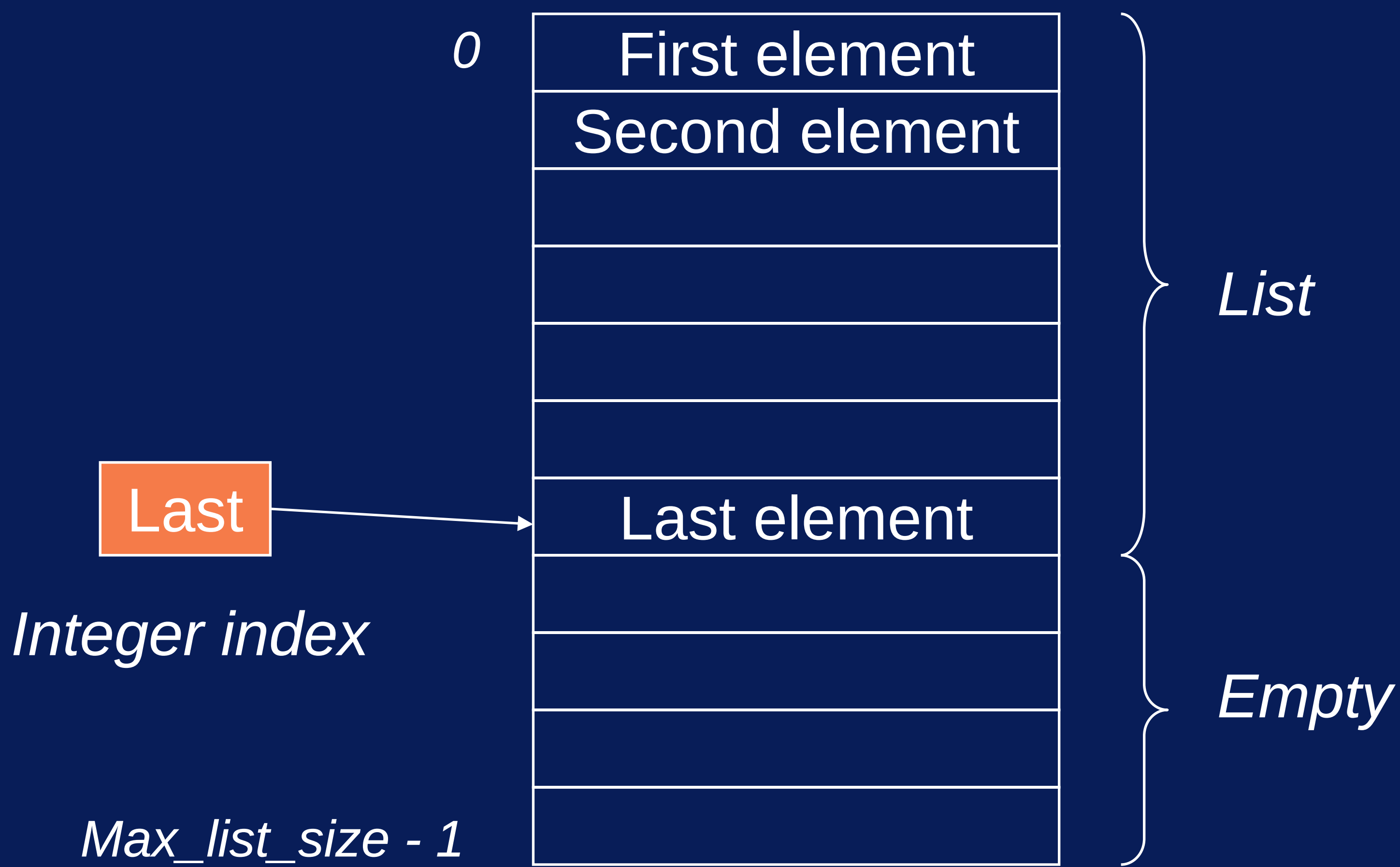
ADT Implementation

- We will look at two implementations
  - Array implementation
    - » uses a static data-structure
    - » reasonable if we know in advance the maximum number of elements in the list
  - Pointer implementation
    - » Also known as a linked-list implementation
    - » uses dynamic data-structure
    - » best if we don’t know in advance the number of elments in the list (or if it varies significantly)
    - » overhead in space: the pointer fields

LIST: Array Implementation

- We will do this in two steps:
  – *the implementation (or representation) of the four constituents datatypes of the ADT:*
    » *list*
    » *elementtype*
    » *Boolean*
    » *windowtype*
  – *the implementation of each of the ADT operations*

LIST: Array Implementation



LIST: Array Implementation

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

LIST: Array Implementation

```
/* array implementation of LIST ADT */

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

#define MAX_LIST_SIZE 100
#define FALSE 0
#define TRUE 1

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

LIST: Array Implementation

```
typedef struct {
           int last;
           ELEMENT_TYPE a[MAX_LIST_SIZE];
        } LIST_TYPE;

typedef int WINDOW_TYPE;

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

WINDOW_TYPE end(LIST_TYPE *list) {
   return(list->last+1);
}
```

LIST: Array Implementation

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

WINDOW_TYPE empty(LIST_TYPE *list) {
   list->last = -1;
   return(end(list));
}

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

int is_empty(LIST_TYPE *list) {
   if (list->last == -1)
      return(TRUE);
   else
      return(FALSE)
```

LIST: Array Implementation

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

WINDOW_TYPE first(LIST_TYPE *list) {
   if (is_empty(list) == FALSE) {
      return(0);
   else
      return(end(list));
}
```

LIST: Array 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+1);
   }
}
```

LIST: Array Implementation

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

WINDOW_TYPE previous(WINDOW_TYPE w, LIST_TYPE *list) {
   if (w != first(list)) {
      return(w-1);
   else {
      error(“can’t find previous before first element of
  list”);
      return(w);
   }
}
```

LIST: Array Implementation

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

WINDOW_TYPE last(LIST_TYPE *list) {
return(list->last);
}
```

LIST: Array Implementation

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

LIST_TYPE *insert(ELEMENT_TYPE e, WINDOW_TYPE w,
                  LIST_TYPE *list) {
   int i;
   if (list->last >= MAX_LIST_SIZE-1) {
      error(“Can’t insert - list is full”);
   }
   else if ((w > list->last + 1) ¦¦ (w < 0)) {
      error(“Position does not exist”);
   }
   else {
      /* insert it … shift all after w to the right */
```

LIST: Array Implementation

```
    for (i=list->last; i>= w; i--) {
       list->a[i+1] = list->a[i];
    }

    list->a[w] = e;
    list->last = list->last + 1;

    return(list);
  }
}
```

LIST: Array Implementation

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

LIST_TYPE *delete(WINDOW_TYPE w, LIST_TYPE *list) {
   int i;
   if ((w > list->last) ¦¦ (w < 0)) {
      error(“Position does not exist”);
   }
   else {
      /* delete it … shift all after w to the left */
      list->last = list->last - 1;
      for (i=w; i <= list->last; i++) {
         list->a[i] = list->a[i+1];
      }
      return(list);
```

LIST: Array Implementation

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

ELEMENT_TYPE retrieve(WINDOW_TYPE w, LIST_TYPE *list) {
   if ( (w < 0)) ¦¦  (w > list->last)) {

      /* list is empty */

      error("Position does not exist");
   }
   else {
      return(list->a[w]);
   }
}
```

LIST: Array 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: Array Implementation

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

/*** 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: Array Implementation

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

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

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

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

LIST: Array 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: Array Implementation

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

}
```

LIST: Array Implementation

- Key points:
  - *we have implemented all list manipulation operations with dedicated access functions*
  - *we never directly access the data-structure when using it but we always use the access functions*
  - *Why?*

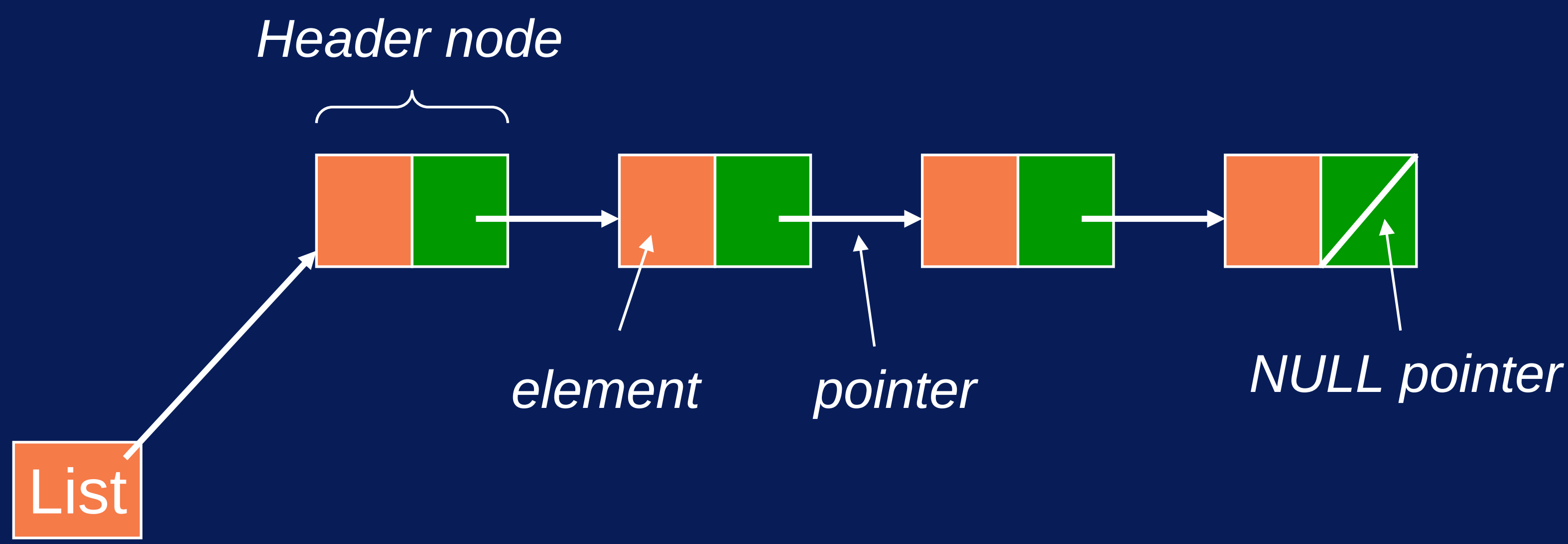
LIST: Array Implementation

- Key points:
  - *greater security: localized control and more resilient software maintenance*
  - *data hiding: the implementation of the data-structure is hidden from the user and so we can change the implementation and the user will never know*

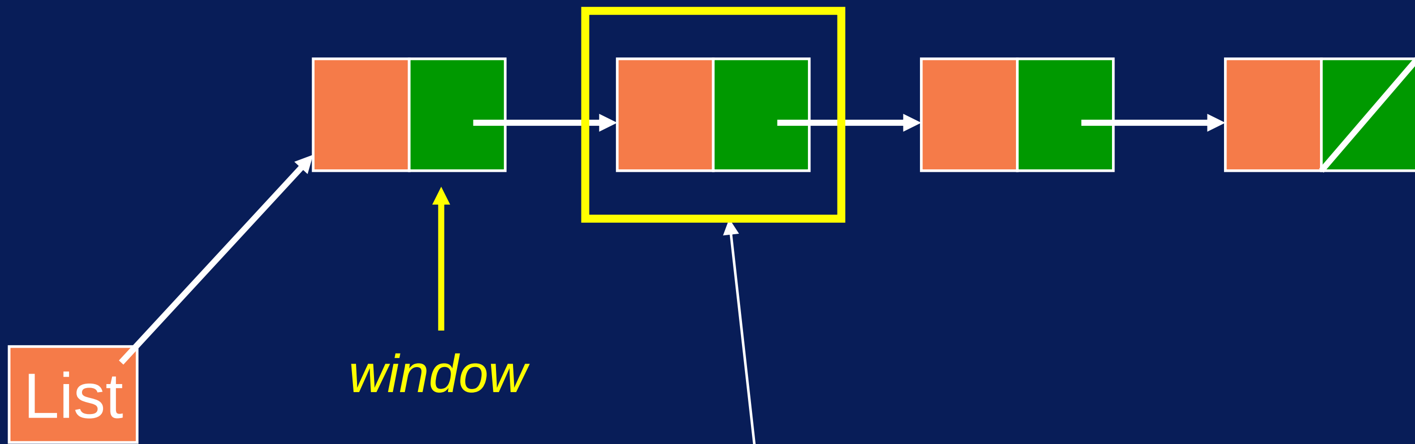
LIST: Array Implementation

- Possible problems with the implementation:
  - *have to shift elements when inserting and deleting (i.e. insert and delete are O(n))*
  - *have to specify the maximum size of the list at compile time*

LIST: Linked-List Implementation

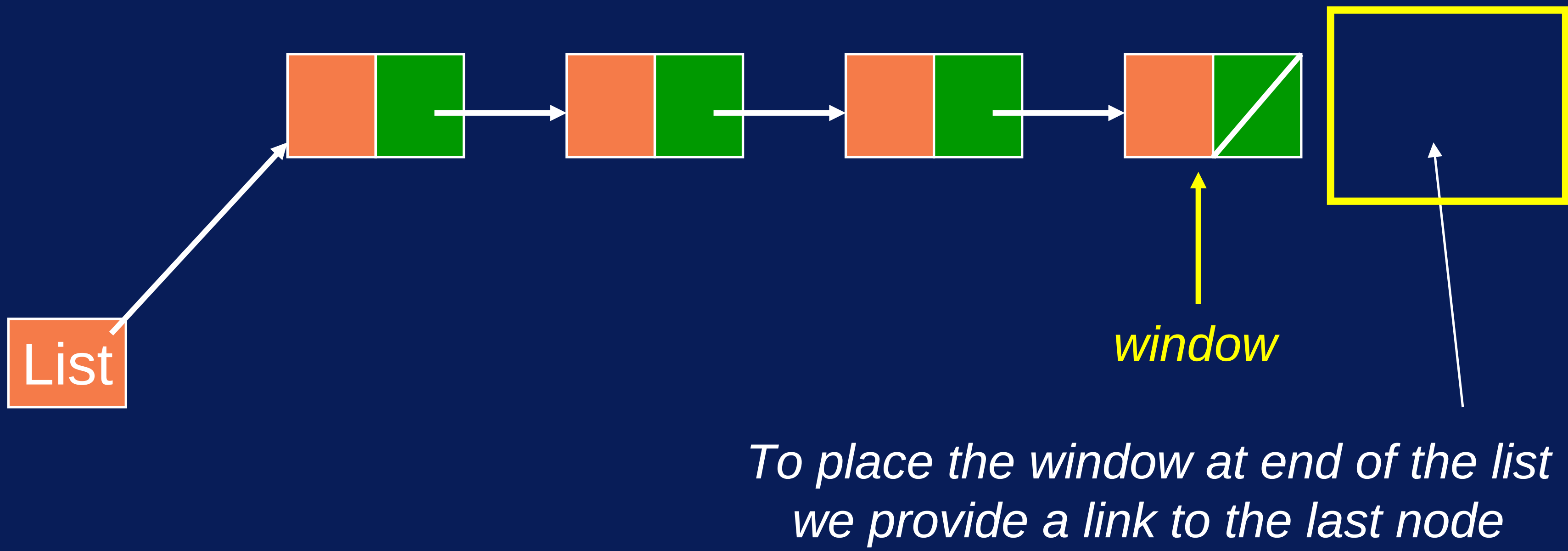


LIST: Linked-List Implementation

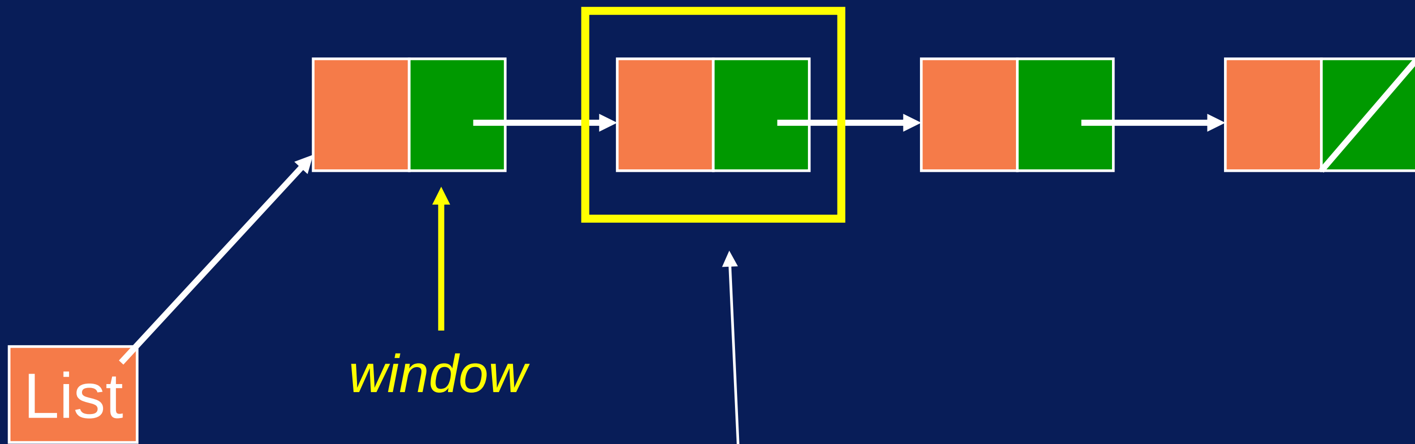


To place the window at this position
we provide a link to the previous node
(this is why we need a header node)

LIST: Linked-List Implementation

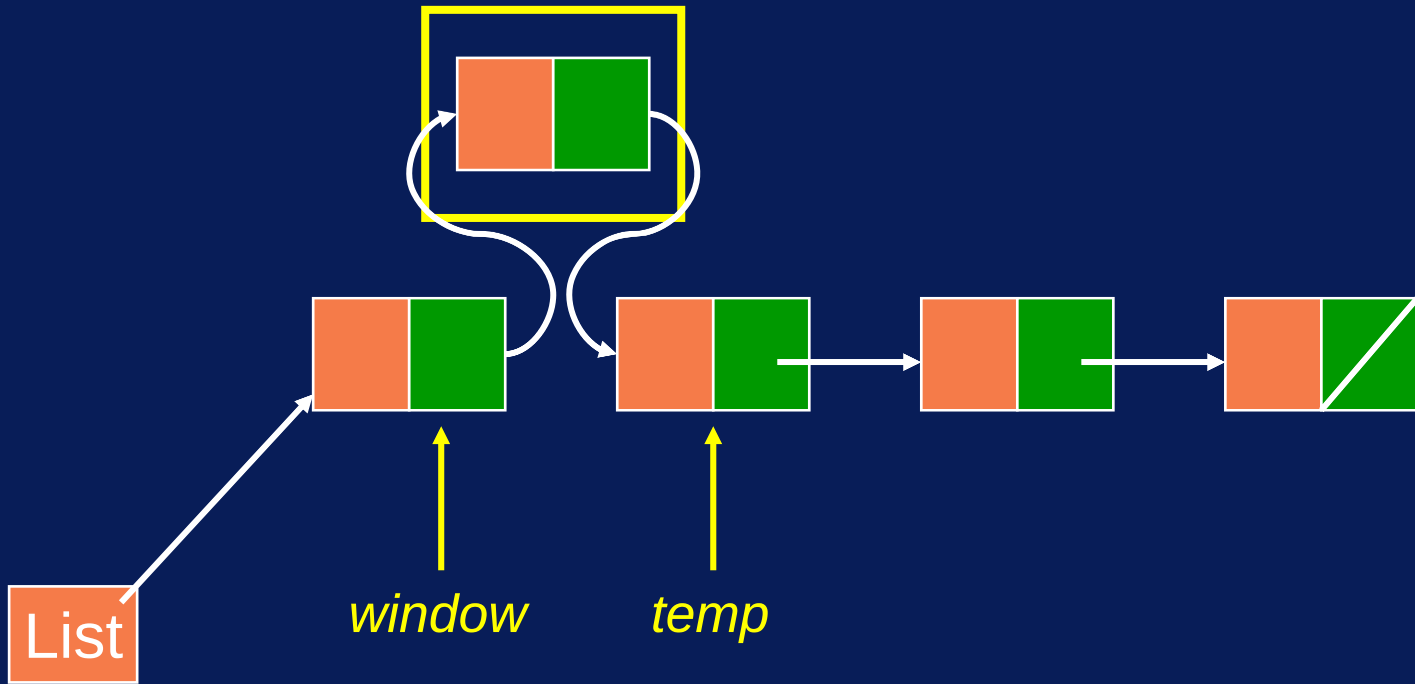


LIST: Linked-List Implementation



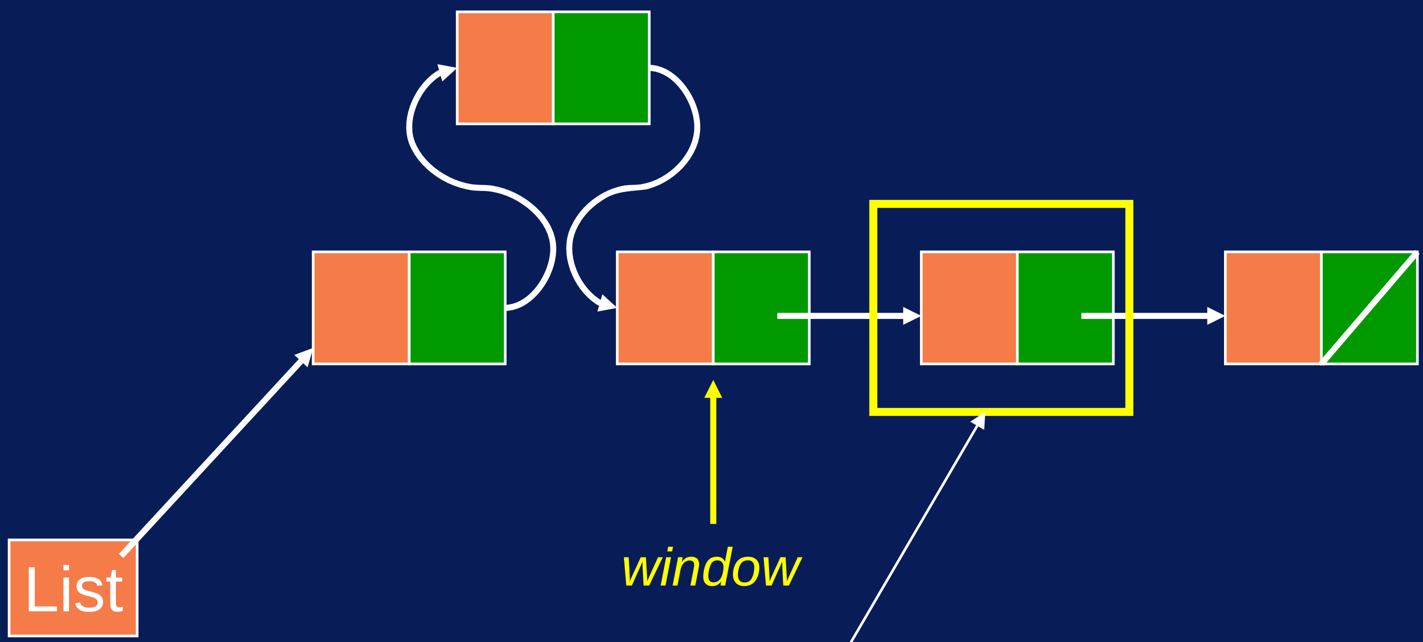
To insert a node at this window position we create the node and re-arrange the links

LIST: Linked-List Implementation



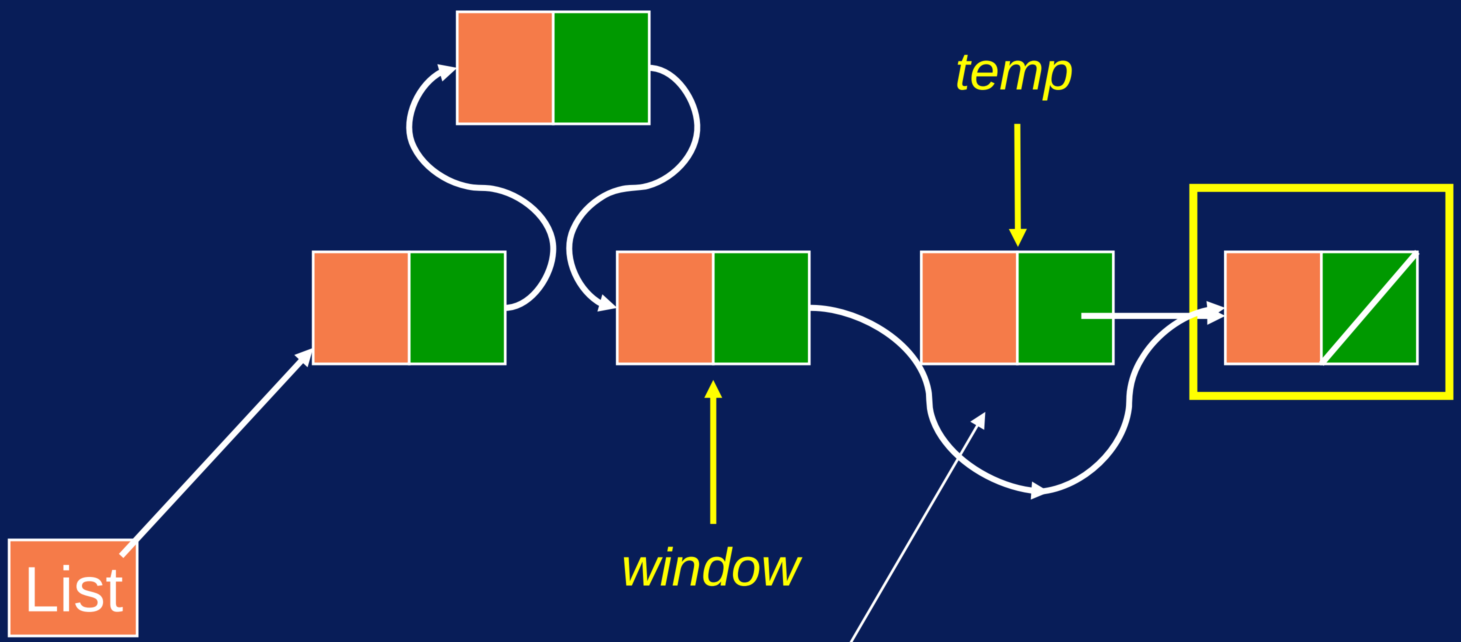
To insert a node at this window position
we create the node and re-arrange the links

LIST: Linked-List Implementation



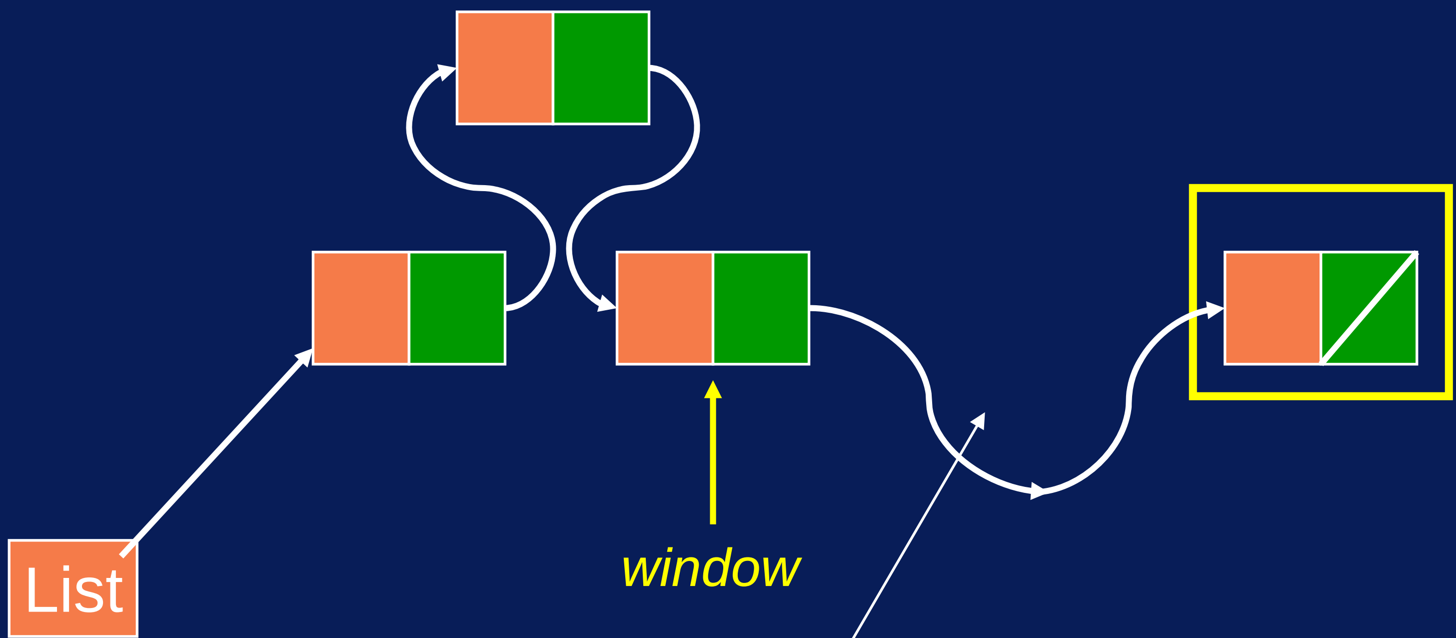
To delete a node at this window position
we re-arrange the links and free the node

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

```
/** 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

```
   /* 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

```
        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

```
/*** 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

```
/*** 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)) {
      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);
}

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

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

LIST: Linked-List Implementation

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

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

   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*

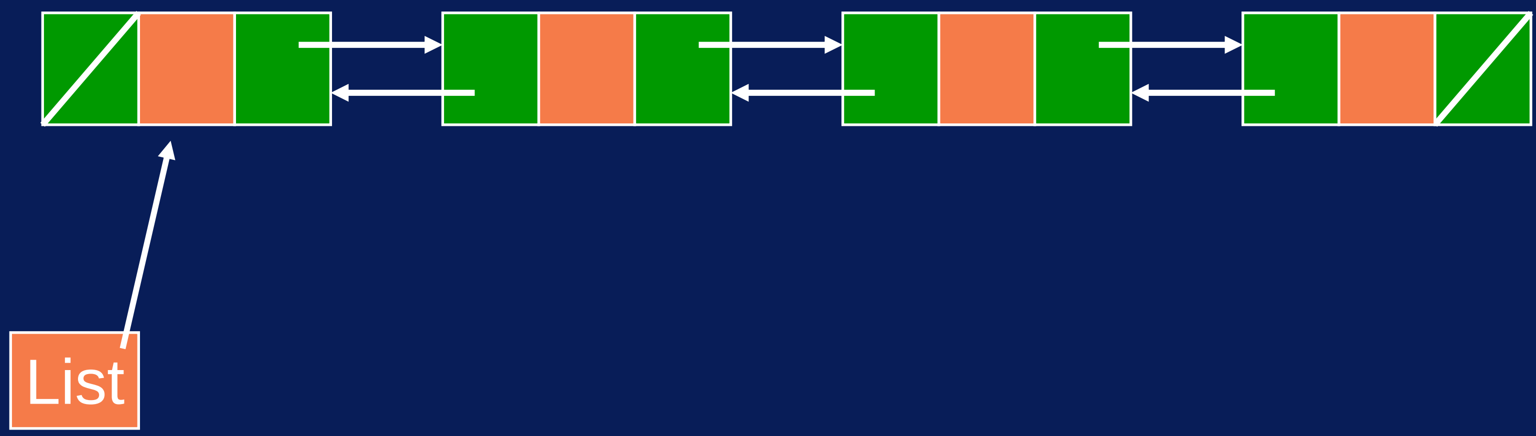
LIST: Linked-List Implementation

- 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

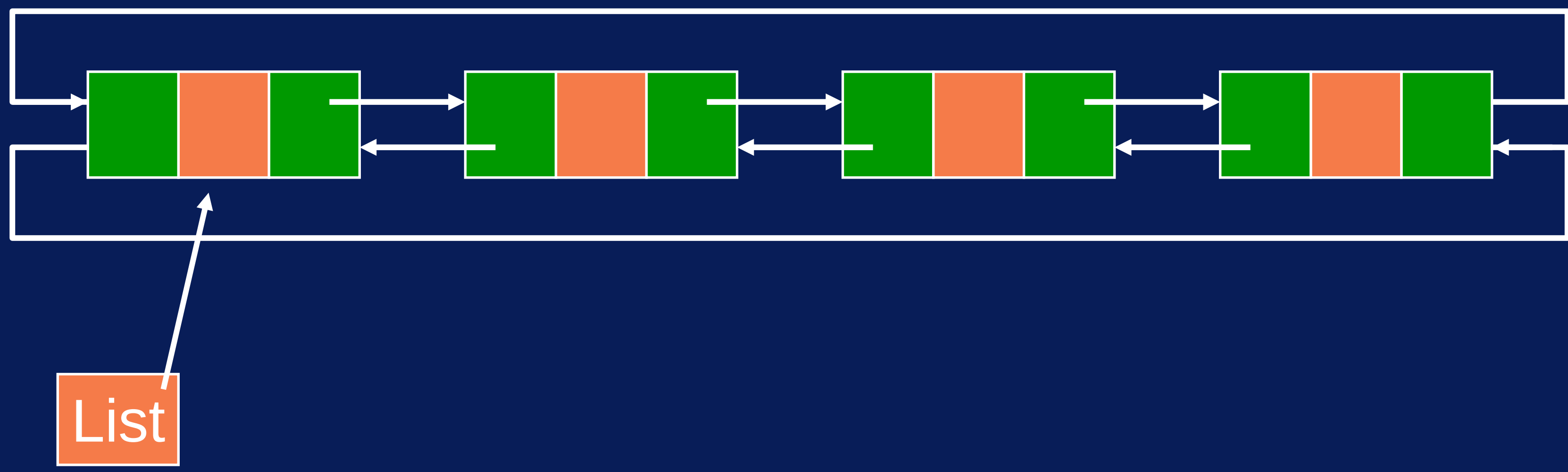
- Possible problems with the implementation:
  - *we have to run the length of the list in order to find the end (i.e. end(L) is O(n))*
  - *there is a (small) overhead in using the pointers*
- *On the other hand, the list can now grow as large as necessary, without having to predefine the maximum size*

LIST: Linked-List Implementation



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

LIST: Linked-List Implementation



Lists can also be circular