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

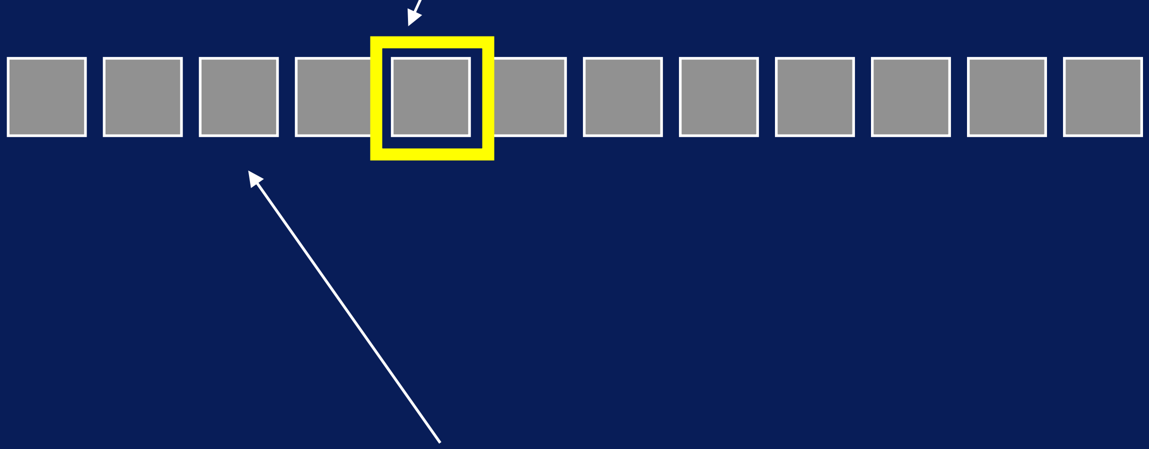
Lists

- 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
  - The position of a_i is i

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*: → L :

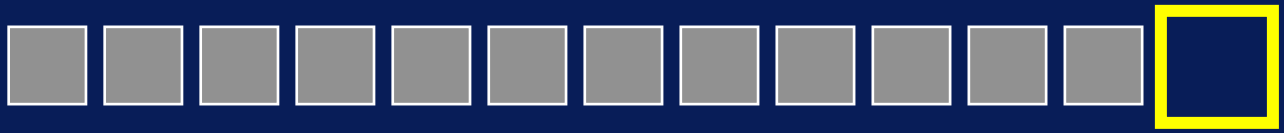
  The function value of *Declare(L)* is an empty list

  - alternative syntax: *LIST L*

LIST Operations

- *End*: L → W :

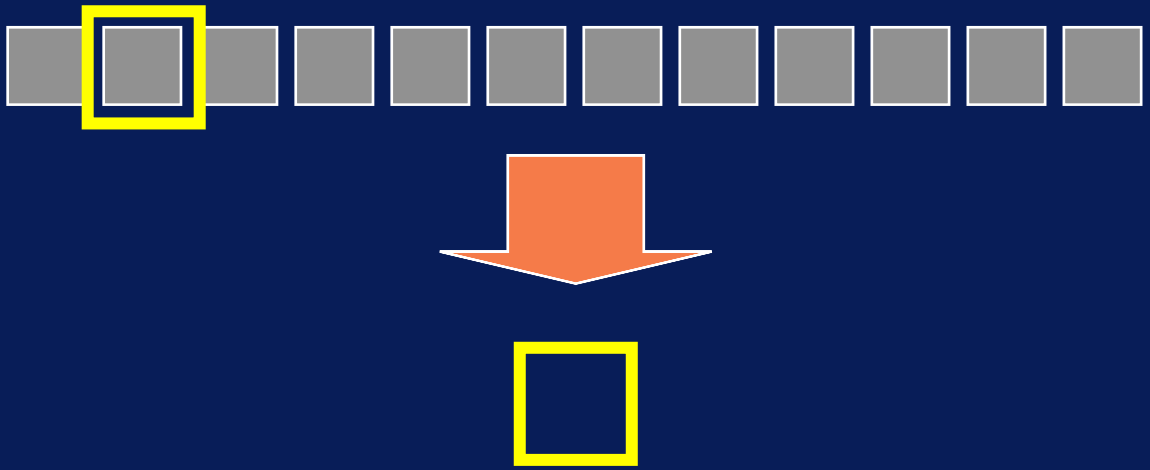
  The function *End(L)* returns the position <u>after</u> the last element in the list
  (i.e. the value of the function is the window position after the last element in the list)



LIST Operations

- *Empty*: L → L x W :

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



LIST Operations

- *IsEmpty*: L → B :

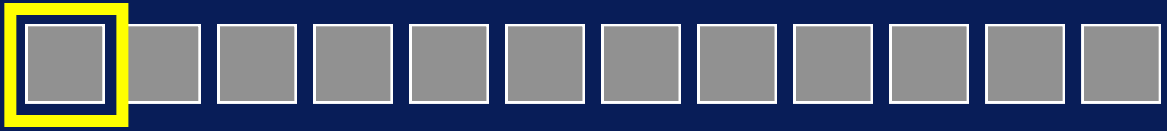
  The function value *IsEmpty*(*L*) is *true* if *L* is empty; otherwise it is *false*

LIST Operations

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

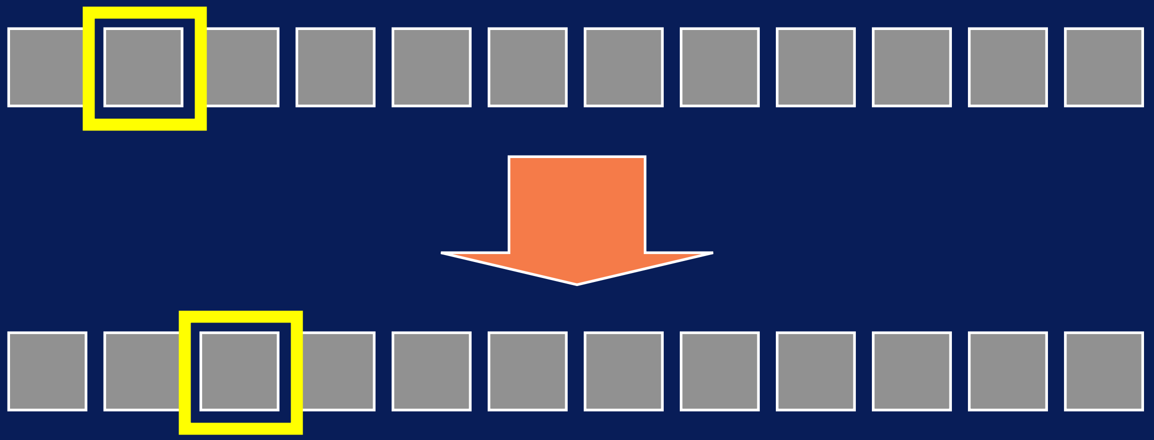
  The function value *Next*(*w*,*L*) is the window position of the next successive element in the list;

  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*)



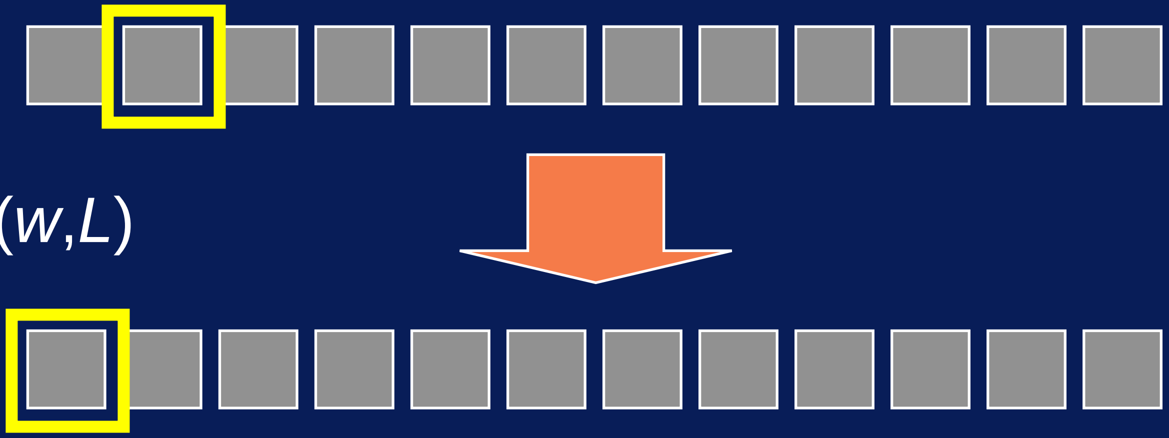
LIST Operations

- *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



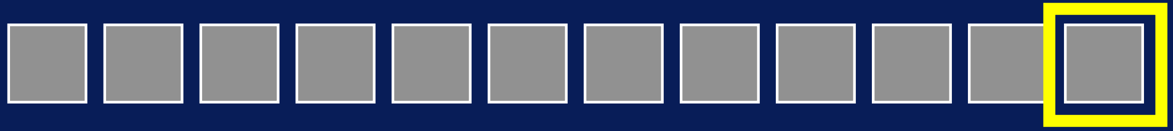
Previous(w,L)

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 × L × W → L × 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

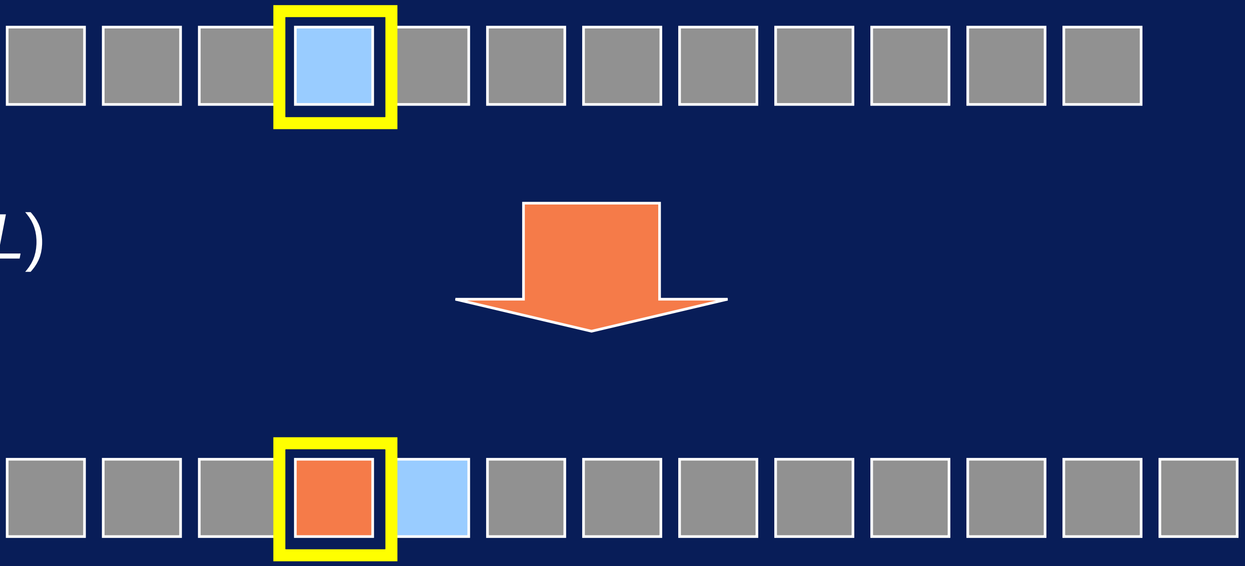
$a_1, a_2, \ldots a_n \rightarrow a_1, a_2, \ldots, a_n, e$

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

The function value is the list with the element inserted

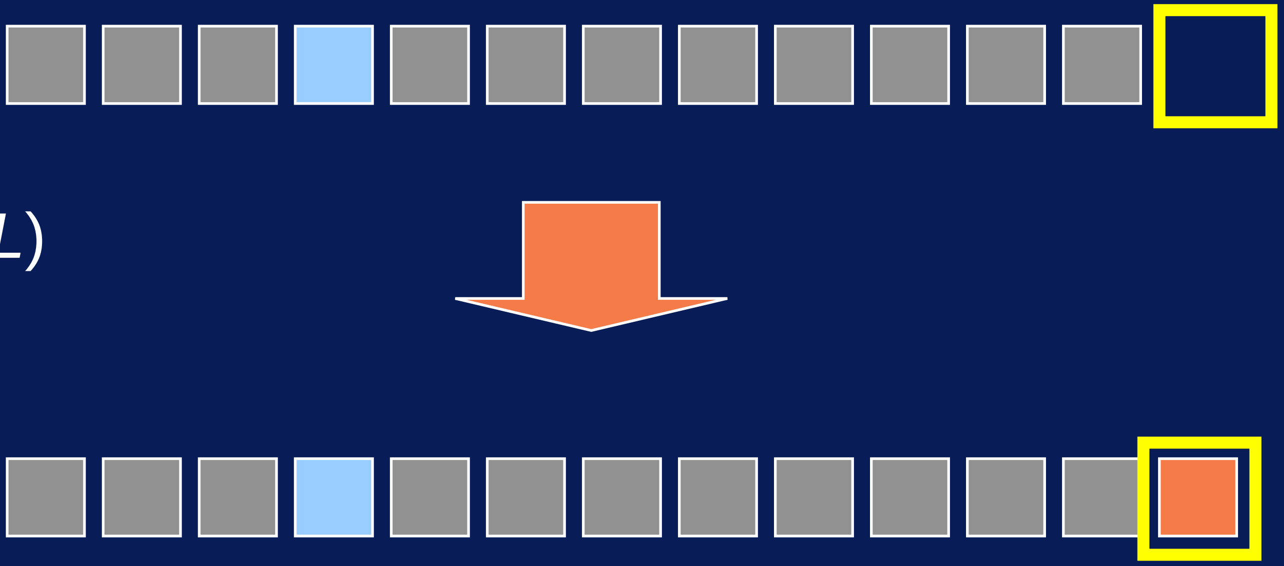
LIST Operations

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



LIST Operations

Insert(e,w,L)



LIST Operations

- *Delete*: L × W → L × W :

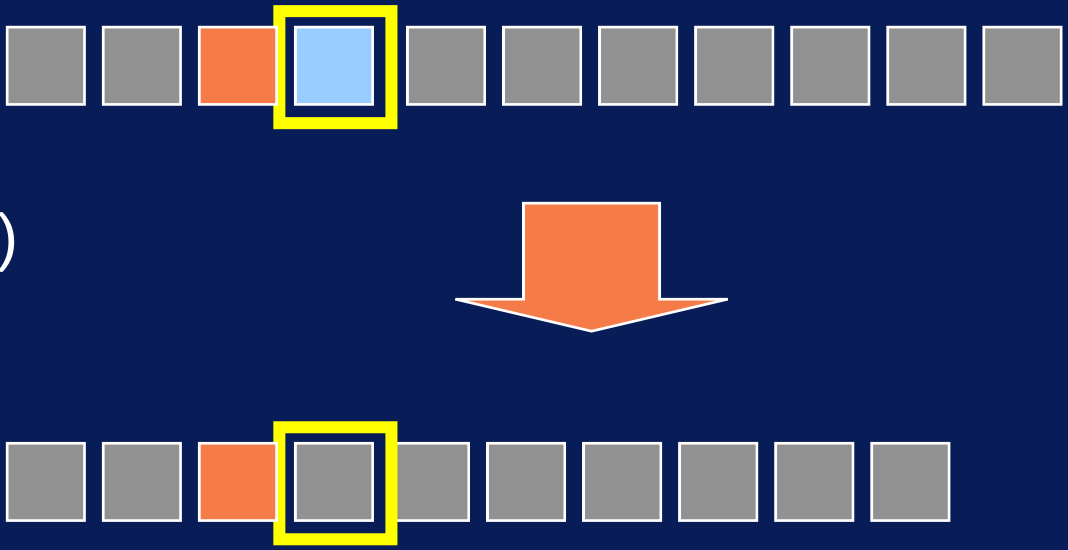
  Delete(w, L)
  Delete the element at position *w* in the list *L*

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

  - 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 × W → 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*

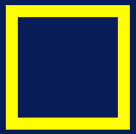
LIST Operations

• *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)



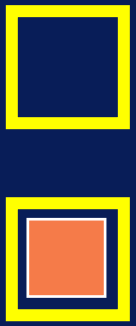
empty list

LIST Operations

- *Example of List manipulation*

w = End(L)

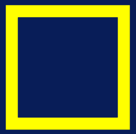
Insert(e,w, L)



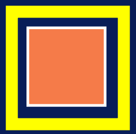
LIST Operations

- *Example of List manipulation*

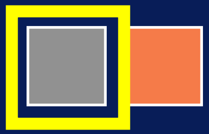
w = End(L)



Insert(e,w, L)



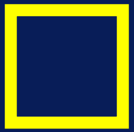
Insert(e,w, L)



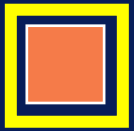
LIST Operations

- *Example of List manipulation*

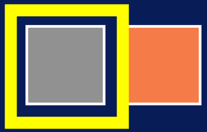
w = End(L)



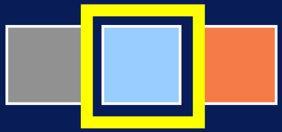
Insert(e,w, L)



Insert(e,w, L)



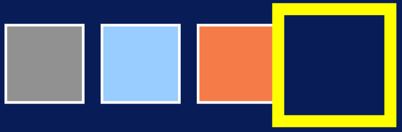
Insert(e,Last(L), L)



LIST Operations

- *Example of List manipulation*

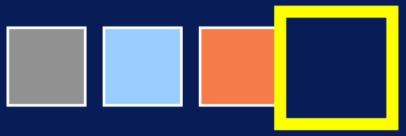
w = Next(Last(L),L)



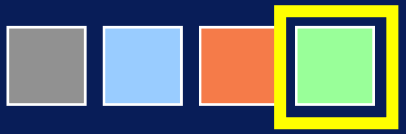
LIST Operations

- *Example of List manipulation*

w = Next(Last(L),L)



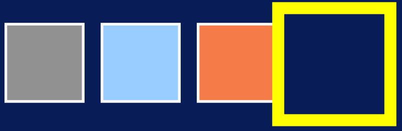
Insert(e,w,L)



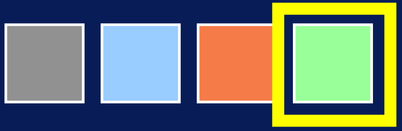
LIST Operations

- *Example of List manipulation*

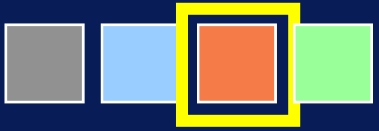
w = Next(Last(L),L)



Insert(e,w,L)



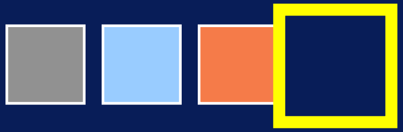
w = Previous(w,L)



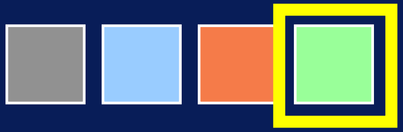
LIST Operations

- *Example of List manipulation*

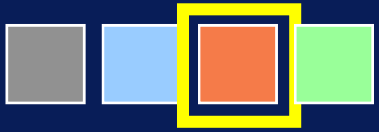
w = Next(Last(L),L)



Insert(e,w,L)



w = Previous(w,L)



Delete(w,L)