Data Structures
Fall 2018
Lists

Luis de Marcos Ortega

luis.demarcos@uah.es

Contents

- Definitions
- List ADT
- Implementation of lists
- Doubly linked lists
- Circular lists
- Ordered lists

Bibliography

- Chapter 2 of:
  - A.V. AHO., J.E. HOPCROFT., J.D. ULLMAN. 1987. “Data Structures and Algorithms.” Addison-Wesley.

Lists

- List – Sequence of elements
  - Generic and flexible data structure. It can grow and shrink on demand.
  - Elements can be accessed, inserted or deleted at any position within a list.
  - Can also be concatenated together or split into sublists.
  - Lists arise routinely in applications such as information retrieval, programming language translation and simulation.

Lists

- Two special cases of lists
  - Stacks – Elements are inserted and deleted at one end only
  - Queues – Elements are inserted at one end and deleted at the other end

Definition

- A *list* is sequence of zero or more elements of a given type (which we generally call *elementtype*):

$$a_1, a_2, \ldots a_n$$

- where n≥0.
- The number of elements *n* is the *lenght* of the list
- Assuming *n*≥1, a_1 is the *first* element and a_n is the *last* element
- If *n*=0, we have an *empty list* (a list with no elements)

Definition

- Elements in a list can be linearly ordered according to their position on the list.
- a_i *precedes* a_{i+1} for i=1,2,…,n-1
- a_i *follows* a_{i-1} for i=2,3,…,n
- Element a_i is at position *i*.
- Convenient to postulate the existence of a position following the last element on a list.
  - END(L) – The position following n on a n-element list.
  - END(L) has a distance from the position of the list that varies as list grows and shrinks, while all other positions do not necessarily.

Alternative Definition

- Recursive definiton: A list is a set of elements of the same type that
  - it is either empty (and it is call an empty list)
  - or it has a first element (x) called head followed by a list (l) called the rest.
  - Notation → List = x:l
- Some programming languages (e.g. Prolog or lisp) include implementations of this type.
  - The assignment `[X|Y]=[1,2,3,4]` in Prolog
  - implies `X=1` and `Y=[2,3,4]`

Specification

- To form an ADT from the notion of list we must define a set of operations
- No set of operations is suitable for all implementations
- We shall give one representative set of operations

Basic Specification

```
spec LIST[ITEM]
     genres list, item, position
     operations
          insert:item position list->list
          delete:position list->list
          locate:item list->position
          retrieve:position list->item
          next:position list->item
          previous:position list->item
          makenull:list->list
          empty:list->bool
endspec
```

An Extended Specification

```
spec LIST[ITEM]
      uses natural
      genres list, item, position
      operations
            ... {all previous ops}
            _[_]:list position->item {alt retrieve}
            _++_:list list->list {concatenate}

            end:list->postion
            first:list->item
            rest:list->list
            length:list->natural
            last:list->item
            modify:position list item->list
            onlist:item list->boolean
endspec
```

Example

```
proc purge(var L:list)
{removes duplicate elements from list L}
      p, q: position {integers}
      p:=first(L)
      while p≠END(L)
            q:=next(p,L)
            while q≠END(L)
                  if retrieve(p,L)=retrieve(q,L)
                  then delete(q,L)
                  else q:=next(q,L)
            p:=next(p,L)
      endwhile
endproc
```

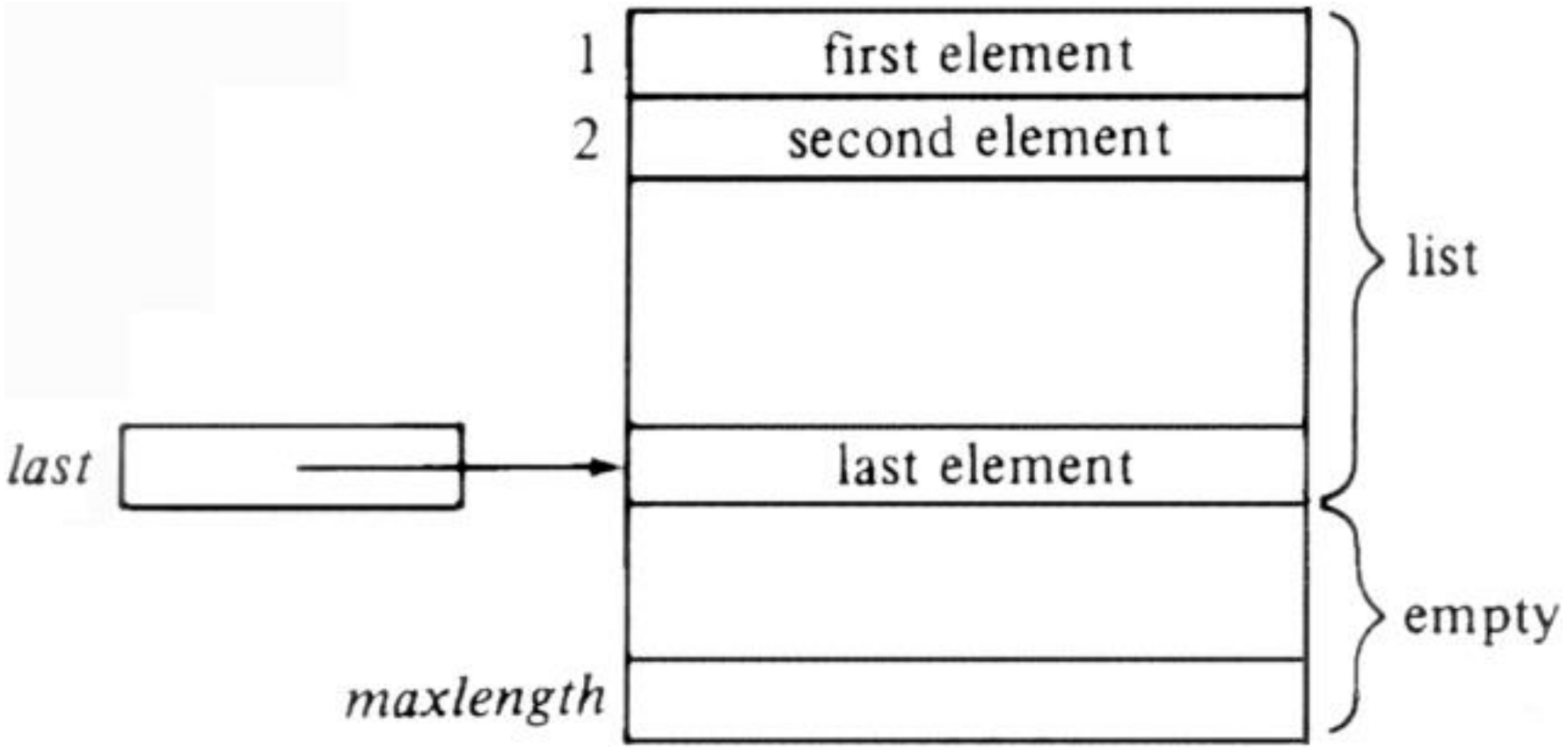
Implementations of Lists

- Array implementation
- Pointer implementation
  - Linked list or dynamic list
- Cursor implementation

Array Implementation

- The elements are stored in contiguous cells of an array
- The list is easily traversed and new elements can be appended readily to the tail of the list
- Inserting an element into the middle of the list, however, requires shifting all following elements one place over in the array to make room for the new element.
- Similarly, deleting any element except the last also requires shifting elements to close up the gap.

Array Implementation



Array Implementation

```
const maxlength = 100 {some suitable constant}

list = record
     elements[1..maxlength] of elementtype;
     last: integer
endrecord

position = integer
```

Array Implementation

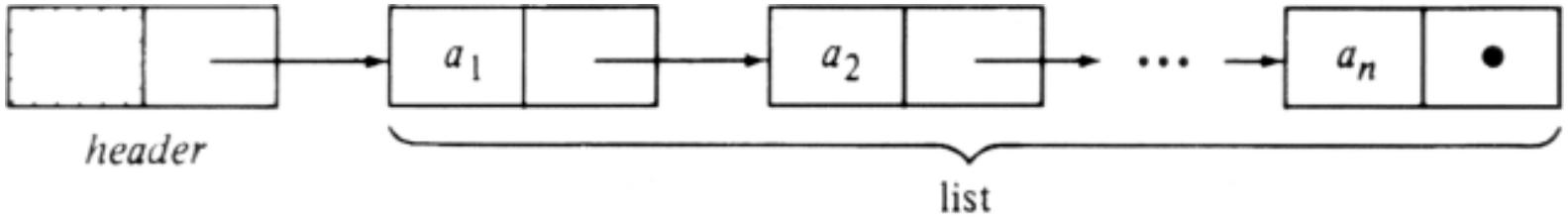
- Running time of operations
  - insert – O(n)
  - delete – O(n)
  - locate – O(n)
  - retrieve – O(1)
  - next & previous – O(1)
  - makenull – O(1)
  - empty – O(1)

Pointer Implementation

- Singly-linked cells
- Pointers to link successive list elements
- This implementation frees us from using contiguous memory for storing a list and hence from shifting elements to make room for new elements or to close up gaps created by deleted elements
- However, one price we pay is extra space for pointers

Pointer Implementation

- the cell holding a_i has a pointer to the cell holding a_{i+1}
- The cell holding a_n has a **null** pointer.
- There is also a *header* cell that points to the cell holding a_1; the header holds no element.
- In the case of an empty list, the header's pointer is **null**, and there are no other cells.



Pointer Implementation

```
celltype = record
    element: elementtype
    next: ^celltype
endrecord
list: ^celltype;

position = ^celltype
```

Pointer Implementation

- Running time of operations
  - insert – O(1)
  - delete – O(1)
  - locate – O(n)
  - retrieve – O(1)
  - next is O(1) but previous is O(n)
    - To make previous O(1) → doubly-linked list
  - makenull – O(1) (O(n) to dipose every element)
  - empty – O(1)

Comparison of Methods

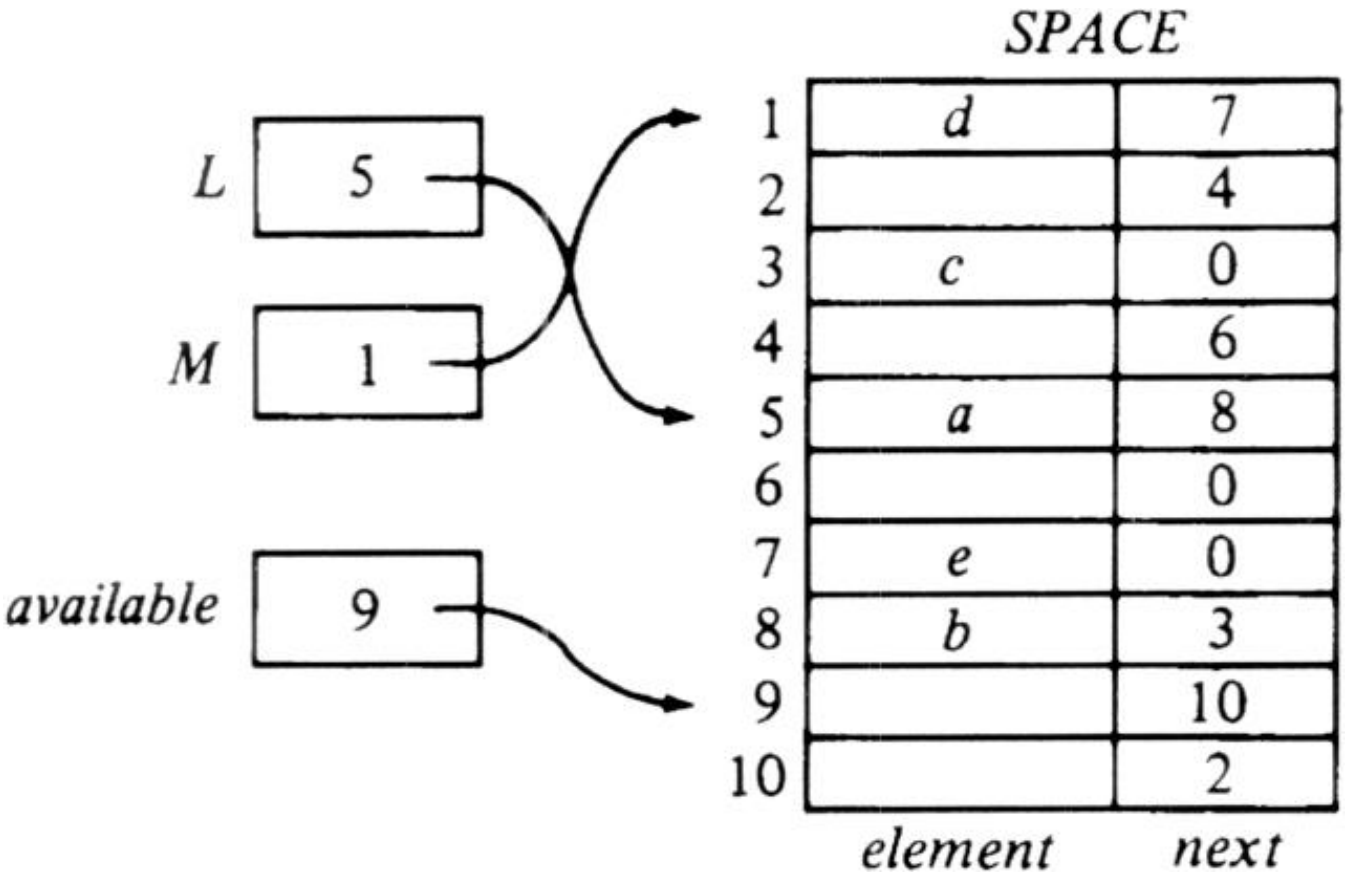
- Array implementation or pointer implementation??
  - depends on which operations we intend to perform, or on which are performed most frequently.
  - Other times, the decision rests on how large the list is likely to get.

Comparison of Methods

- Principal issues to consider:
  - The array implementation requires us to specify the maximum size of a list at compile time.
    - Possible waste of space.
  - INSERT and DELETE take a constant number of steps for a linked list, but require time proportional to the number of following elements with an array
  - PREVIOUS require constant time with the array implementation, but time proportional to the length of the list if pointers are used

Cursor Implementation

- Example

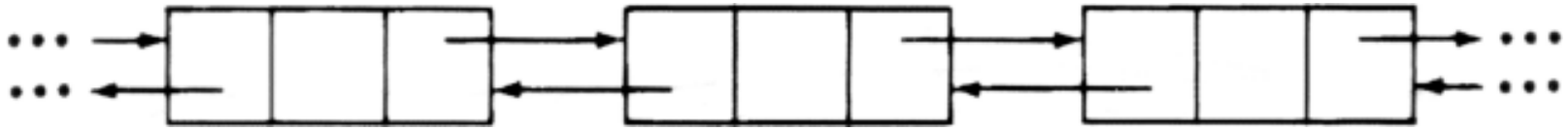


Cursor Implementation

- Two lists, *L* = *a*, *b*, *c* and *M* = *d*, *e*, sharing the array *SPACE*, with *maxlength* = 10.
- Notice that all the cells of the array that are not on either list are linked on another list called *available*.
- This list is necessary so we can obtain an empty cell when we want to insert into some list, and so we can have a place to put deleted cells for later reuse.

Doubly Linked Lists

- In a number of applications we may wish to traverse a list both forwards and backwards efficiently.
- Or, given an element, we may wish to determine the preceding and following elements quickly.
- In such situations we might wish to give each cell on a list a pointer to both the next and previous cells on the list, as suggested by the doubly-linked list.



Doubly Linked Lists

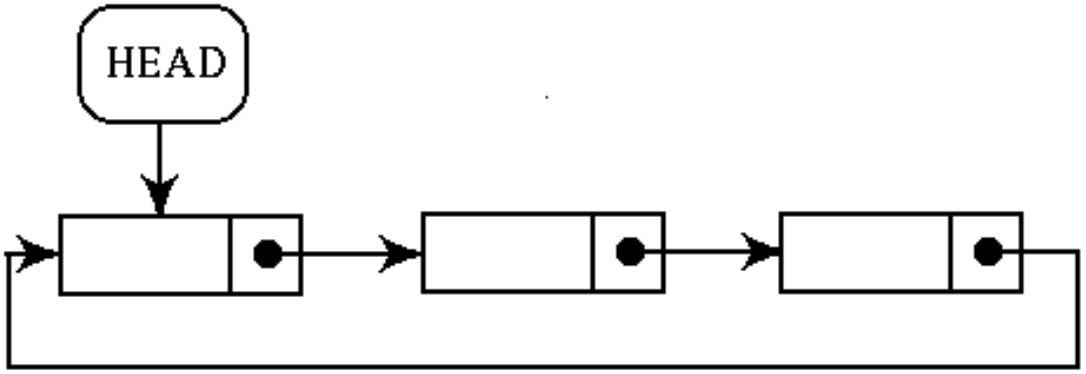
- Price we pay for these features
  - presence of an additional pointer in each cell
  - somewhat lengthier procedures for some of the basic list operations

```
celltype = record
      element: elementtype
      next, previous: ^celltype
endrecord
list: ^celltype;

position = ^celltype
```

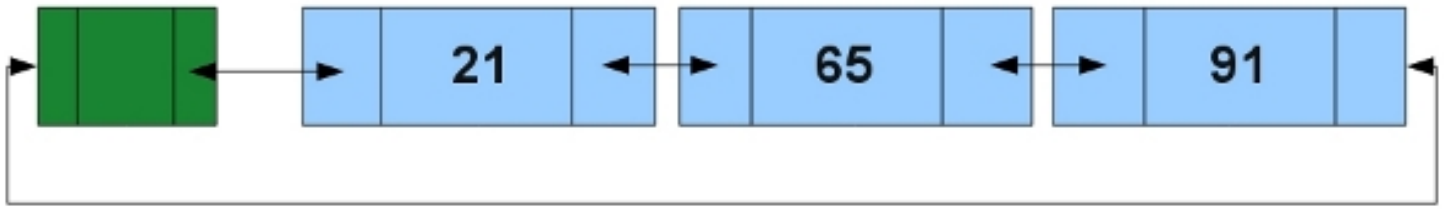
Circular List

- A less common convention is to make the last node of a list point to the first node of the list
- in that case the list is said to be circular or circularly linked; otherwise it is said to be open or linear



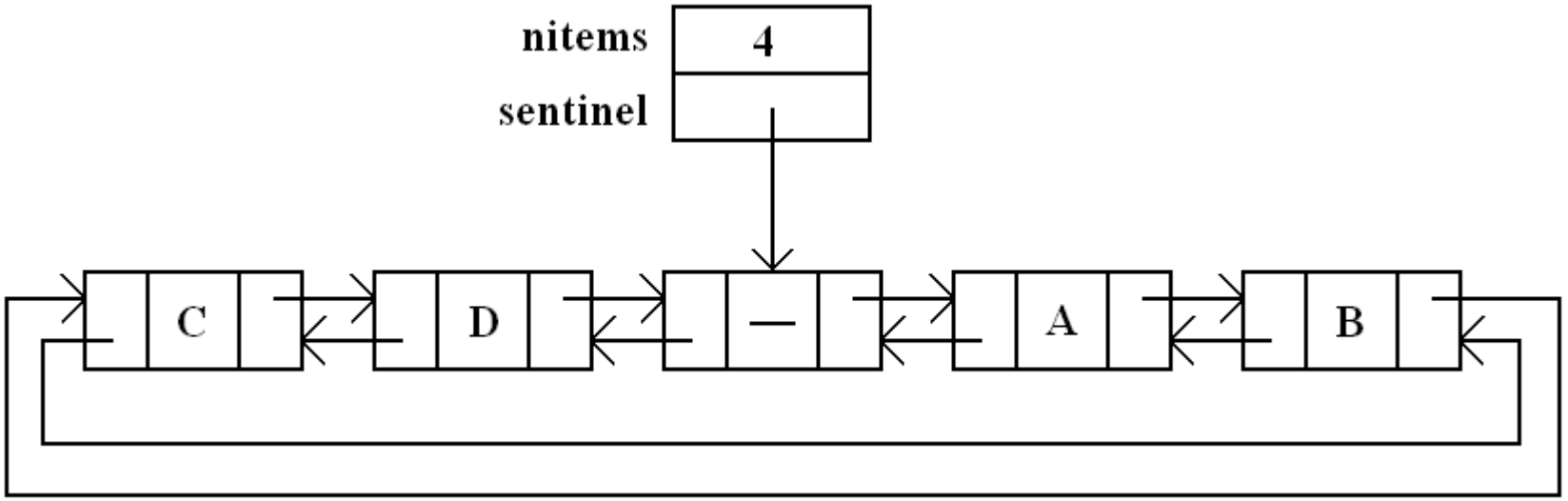
Sentinel Nodes

- In some implementations, an extra **sentinel** or **dummy** node may be added before the first data record and/or after the last one.
- This convention simplifies and accelerates some list operations, by ensuring that all links can be safely dereferenced and that every list (even one that contains no data elements) always has a "first" and "last" node.



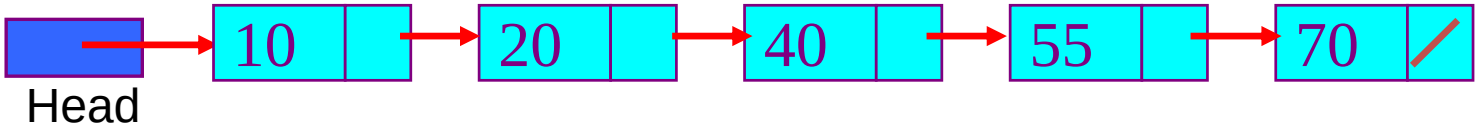
Sentinel Nodes

- Dummy nodes can be particulary useful on circular doubly lists since insert and delete ops will be done on the same way for every node.
  - No need to check for first or last element
  - Thus, less code is necessary
- The pointer to the dummy can be used to store additional information
  - e.g. the number of items → Op lenght is now O(1)

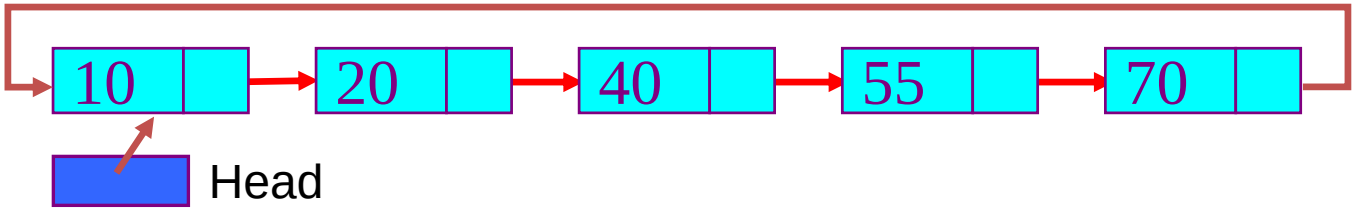


Zoo of Lists

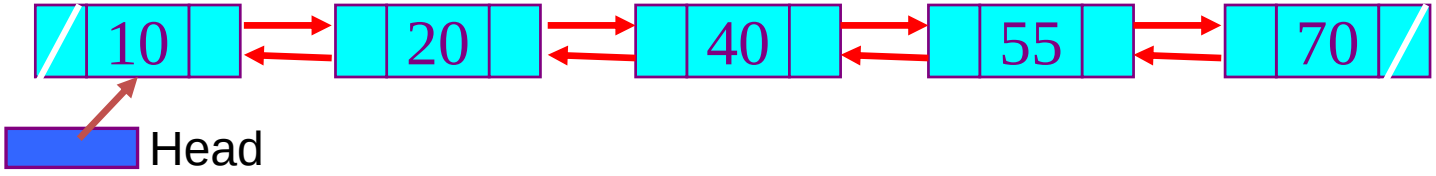
(singly linear) linked list



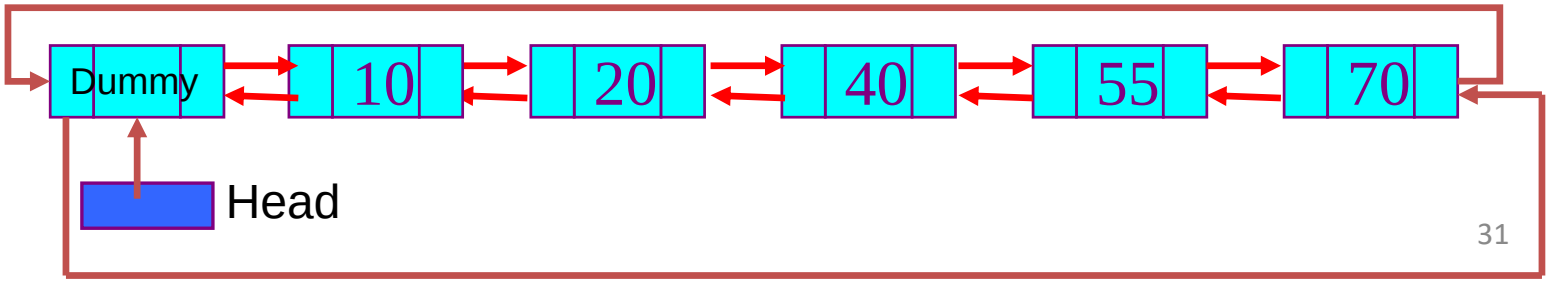
(singly) circular linked list



(linear) doubly linked list



doubly circular linked list with dummy



Ordered Lists

- An ordered list (or sorted list) is a list in which all elements are in a given order at every moment
- One of the values on *elementtype* will be the key value for establishing the order
  - Alphabetical order, numerical order, …
- Operations must be implemented to ensure that order is respected

Ordered Lists

- As for the basic specification, the only operation that needs to be redefined is insert

  insert:item list->list

- An argument for position is no longer required as insertion will take place orderly
- As for the extended specification, more meaningful definitions of modify can also be used

  modify:item list item->list

Ordered Lists

- Running time of operations on an **ordered linked list**:
  - insert – O(n) as I need to seek the insert position
  - delete – O(1) if the position is given as an argument
  - locate – O(n)
  - retrieve – O(1)
  - next is O(1) but previous is O(n)
  - makenull – O(1) (O(n) to dipose every element)
  - empty – O(1)
- insert has a worse running time and all other ops do not improve then, why use ordered lists?

Ordered Lists

- Running time of operations on an **ordered array list**:
  - insert – O(n)
  - delete – O(n)
  - locate – **O(log n)!!!**
  - retrieve – O(1)
  - next & previous – O(1)
  - makenull – O(1)
  - empty – O(1)
- Locate (search) can be speeded up → Binary Search

Lists

Luis de Marcos Ortega

luis.demarcos@uah.es