

## Lecture 5

- List
  - ADT
    - List, IteratedList, IndexedList
    - SortedList
  - Singly Linked List
  - Doubly Linked List

Lect. PhD. Lupsa Dana  
Babes - Bolyai University  
Computer Science and Mathematics Faculty  
2021 - 2022

Some slides borrowed from: Lect. PhD. Onet-Marian Zsuzsanna

## Previously, in Lecture 4

- Containers
  - Bag
  - Set
  - Map
  - Multimap
  - Stack
  - Queue
  - Deque
  - PriorityQueue

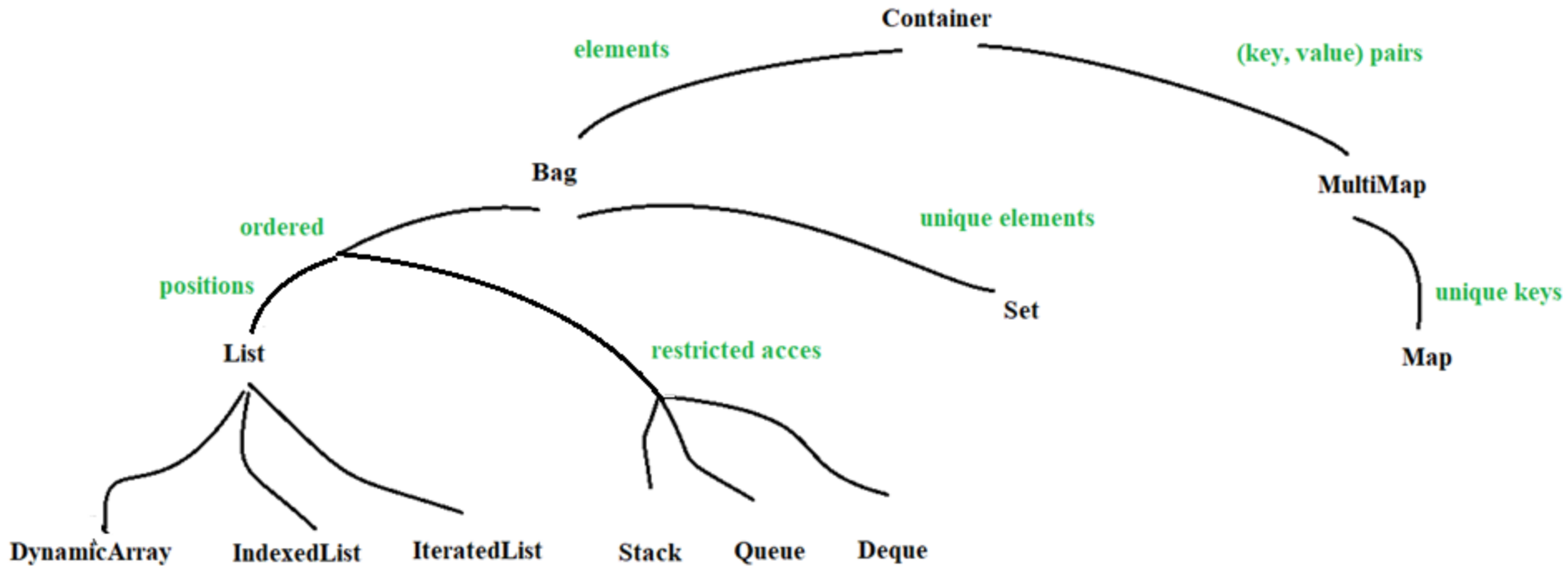


# Previously, in Lecture 4

## Overview

Unsorted containers:

usually used for efficient  
retrieval of values based on keys



# List (or linear list)

Specific:

- elements are arranged in a strict linear order

A list can be seen as a sequence of elements  $\langle l_1, l_2, \dots, l_n \rangle$ , where there is an order of the elements, and each element has a position inside the list.

A List is a container which is either empty or

- it has a unique first element
- it has a unique last element
- for every element (except for the last) there is a unique successor element
- for every element (except for the first) there is a unique predecessor element

Terminology:

**Sequence containers**  
**List**

(C++ STL)  
(Java.util)

# List. Positions

Positions are used to keep information about the order of the elements.

- Every element from a list has a unique position in the list.
- The position of an element:
  - identifies the element from the list
  - “determines” the position of the successor and predecessor element (if they exist).
- A position  $p$  will be considered valid if it denotes the position of an actual element from the list
  - For an invalid position we will use the following notation:  $\perp$  (in pseudocode)

Many operations of ADT List have the next parameters:

- i. element (of type TElem)
- ii. elements' positions (of type TPosition)

See: Lecture04\_ADList

# List

ADTList: Uniform formal approach

Position – give the position of elements in list

- open to many possible instantiations of Position

*Specialized* ADTs List are obtained by instantiation of the type TPosition:

*IndexedList*

- TPosition is an Integer.
- There are less operations in the interface of the IndexedList.
  - Operations first, last, next, previous, valid do not exist.



e.g.  
Python,  
Java

...

*IteratedList*

- TPosition is an Iterator.
  - In case of an IteratedList the operations that take as parameter a position use an Iterator (and the position is the current element from the Iterator)
  - There are less operations
    - Operations valid, next, previous no longer exist in the interface of the List (they are operations for the Iterator).
- ... *add, remove ?*



e.g.  
STL C++

# SortedList

SortedList is a List in which the elements are memorized in an order given by a relation.

The interface of the ADT SortedList is (very) similar to that of List

Some differences:

- The init function takes as parameter a relation that is going to be used to order the elements
- We no longer have several add operations (addToBeginning, addToEnd, addToPosition), we have one single add operation, which takes as parameter only the element to be added (and adds it to the position where it should go based on the relation)
- We no longer have a setElement operation (might violate ordering)

# Linked Lists

## Basic idea:

Order is determined by position information stored with each element

## **Node:**

- elements are stored in nodes
- each node knows the position of the next (/previous) node in the list

Information to store in nodes (choices)

- one link → singly linked list

$$\text{SLLNode} = \{(e; n) \mid e:\text{TElem} \text{ and } n : \text{TPosition}\}$$

*next, link*

- two links → doubly linked list

$$\text{DLLNode} = \{(e; n, p) \mid e:\text{TElem} \text{ and } n : \text{TPosition} \quad \text{next}(\text{DLLNode}) \\ \text{and } p : \text{TPosition} \quad \text{prev}(\text{DLLNode})\}$$

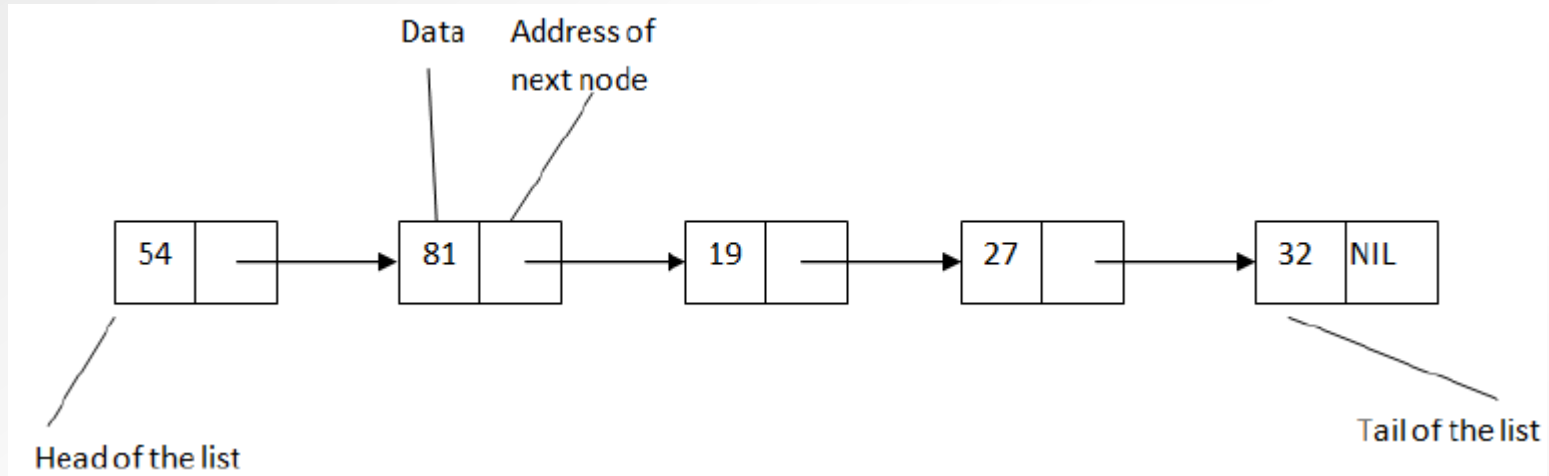
*prev*



# Linked Lists

- A linked list is a linear data structure, where the order of the elements is determined not by indexes, but by an information (pointer) which is placed in each element.
- A linked list is a structure that consists of nodes (sometimes called links) and each node contains, besides the data (that we store in the linked list), a pointer to the address of the next node (and possibly a pointer to the address of the previous node).
- The nodes of a linked list are not necessarily adjacent in the memory, this is why we need to keep the address of the successor in each node.
- Elements from a linked list are accessed based on the pointers stored in the nodes.
- We can directly access only the first element (and maybe the last one) of the list

# Singly Linked Lists - SLL



- In a SLL each node from the list contains the data and the address of the next node.
- The first node of the list is called head of the list and the last node is called tail of the list.
- The tail of the list contains the special value NIL as the address of the next node (which does not exist).
- If the head of the SLL is NIL, the list is considered empty.

# Singly Linked Lists - Representation

- using dynamic memory allocation
  - for each node individually

Terminology (*often*) : dynamic linked storage

SLLNode:

info: TElem *//the actual information*

next:  $\uparrow$  SLLNode *//address of the next node*

SLL:

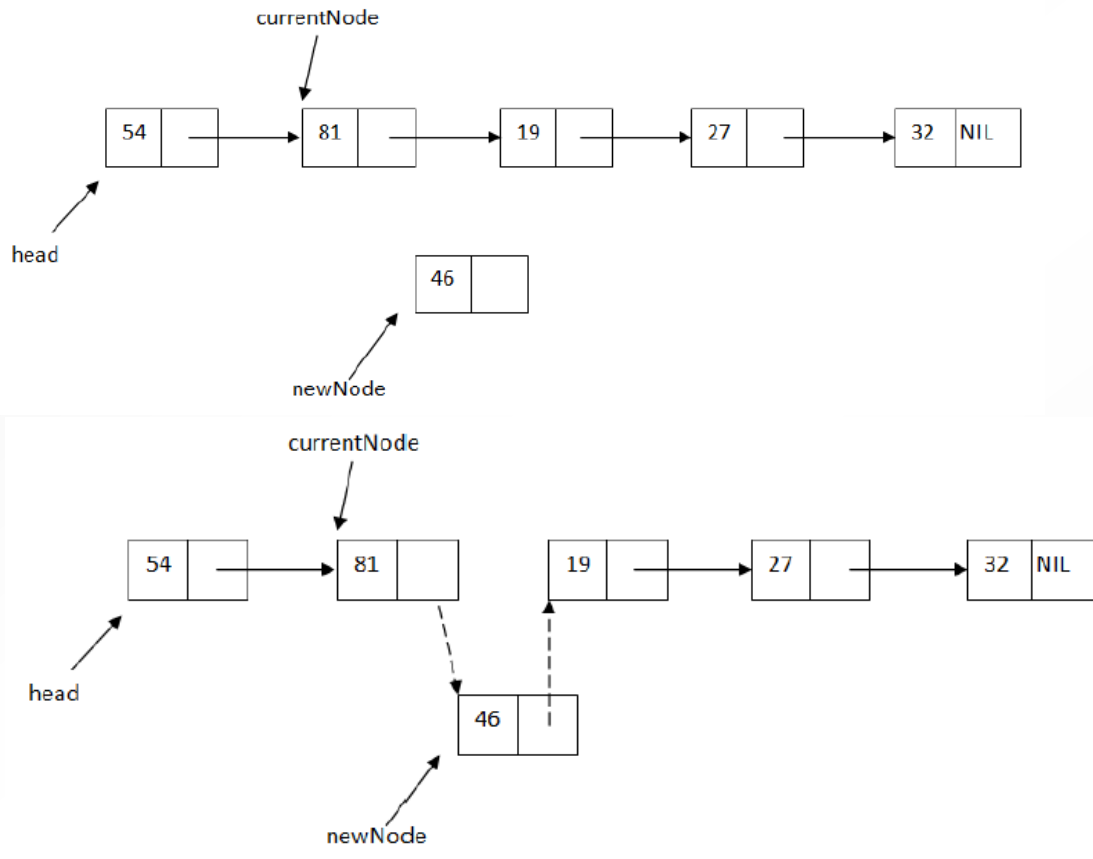
head:  $\uparrow$  SLLNode *//address of the first node*

- represented over an array (We will discuss it later.)

3/21/2022      - *space management in the array !!*

# SLL – Operations

e.g.: Insert after a node



# SLL – Operations. Insert

e.g.: Insert after a node

```
subalgorithm insertAfter(sll, currentNode, elem) is:  
  //pre: sll is a SLL; currentNode is an SLLNode from sll;  
  //elem is a TElem  
  //post: a node with elem will be inserted after node currentNode  
  newNode ← allocate() //allocate a new SLLNode  
  [newNode].info ← elem  
  [newNode].next ← [currentNode].next  
  [currentNode].next ← newNode  
end-subalgorithm
```

**Complexity: ?**

# SLL – Operations. Insert

## Think about:

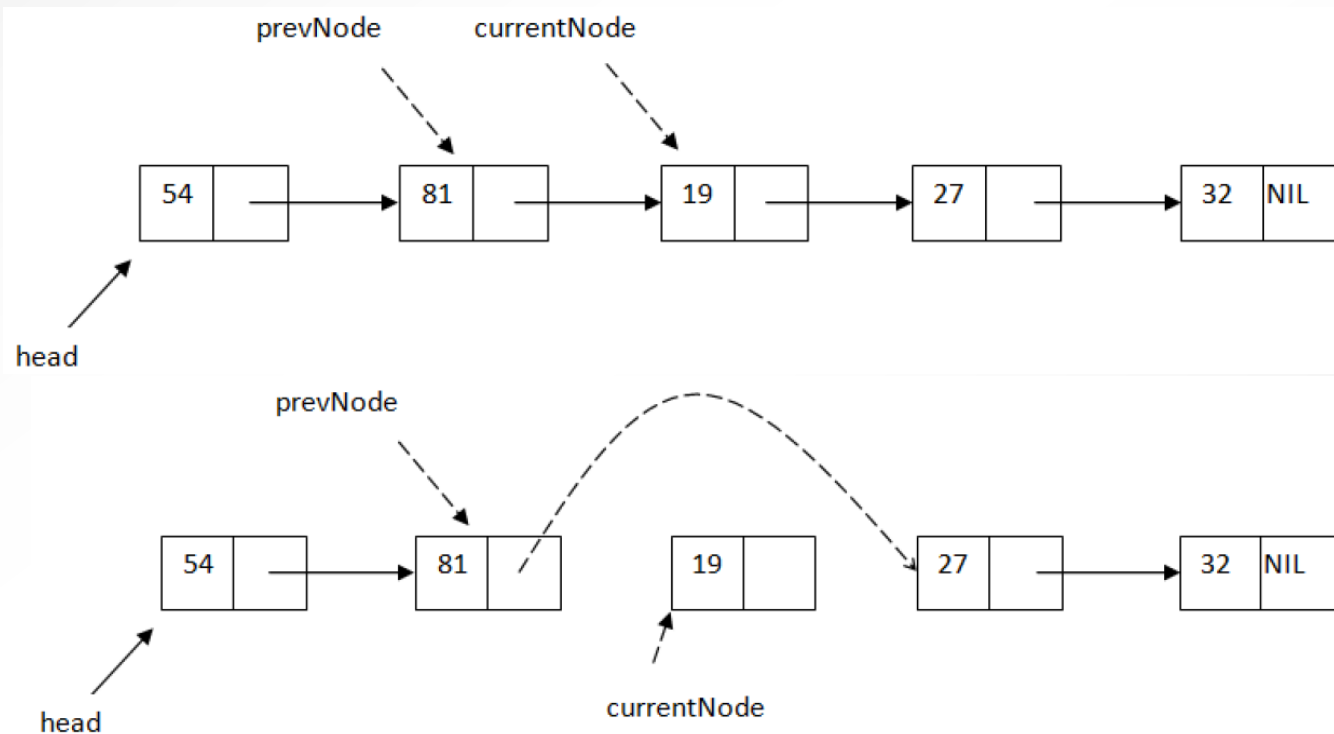
(specification of the operation, pseudocode, complexity)

- Insert into the head of a SLL
- Use only one insert operation to insert into the head of a SLL or after a position
- Insert an element on a given indexed position
- Insert an element before a given node
  - Can it be done in  $\Theta(1)$ ? Is there a cost to be paid?
- Design SLList with **fast addLast?**

# SLL – Operations. Delete

**e.g.**: Delete a given element

- Move with the two pointers until currentNode is the node we want to delete.
- Delete currentNode by jumping over it



# SLL – Operations. Delete

e.g.: Delete a given element

```
function deleteElement(sll, elem) is:  
  //pre: sll is a SLL, elem is a TElem  
  //post: returns true if elem was removed, false otherwise  
  currentNode ← sll.head  
  prevNode ← NIL  
  while currentNode ≠ NIL and [currentNode].info ≠ elem execute  
    prevNode ← currentNode  
    currentNode ← [currentNode].next  
  end-while  
  if currentNode ≠ NIL then  
    if prevNode = NIL then //we delete the head  
      sll.head ← [sll.head].next  
    else  
      [prevNode].next ← [currentNode].next  
    end-if  
    free(currentNode)  
    deleteElement ← True  
  else  
    deleteElement ← False  
  end-if  
end-function
```

**Complexity:  $O(n)$**



# SLL - Iterator

Read-only, uni-directional iterator:

- In case of a SLL, the current element from the iterator is actually a node of the list.

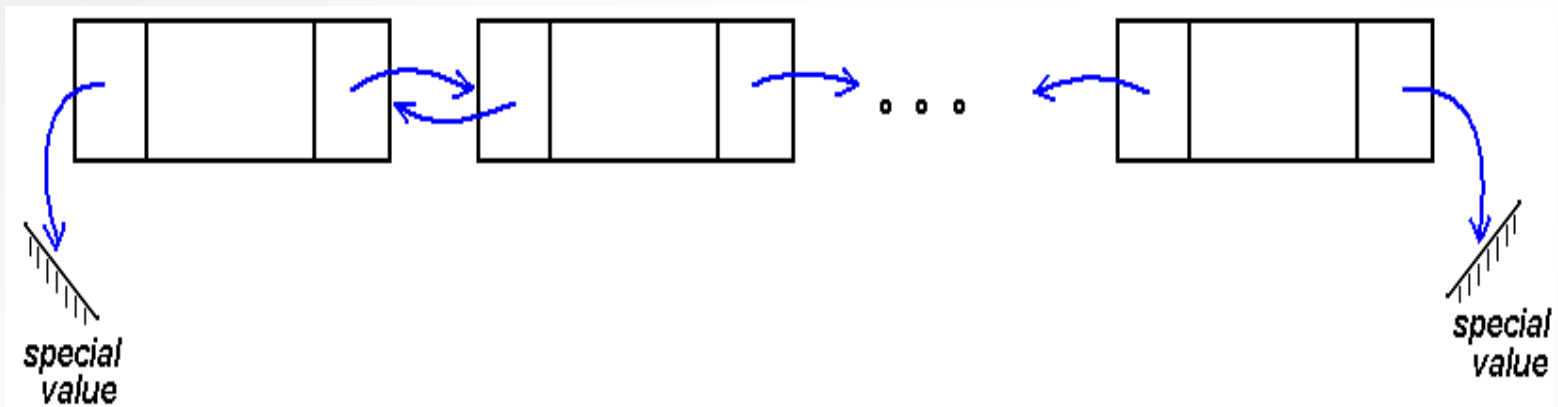
SLLIterator:

list: SLL

currentElement: ↑ SLLNode

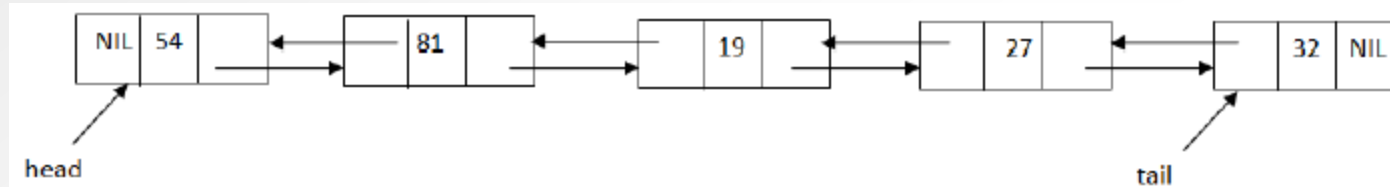
- What would be the complexity of the operations?
- 
- How could we define a bi-directional iterator for a SLL? What would be the complexity of the previous operation?
  - How could we define a bi-directional iterator for a SLL if we know that the previous operation will never be called twice consecutively (two consecutive calls for the previous operation will always be divided by at least one call to the next operation)? What would be the complexity of the operations?

# Doubly-Linked list



- can access elements if we know one of the two ends
- *but*: keep Positions of both ends  
reason: access in both direction

# Doubly-Linked List. Representation



- using dynamic memory allocation
  - for each node individually

DLLNode:

info: TElem  
next: ↑ DLLNode  
prev: ↑ DLLNode

DLL:

head: ↑ DLLNode  
tail: ↑ DLLNode

- represented over an array
  - over an array (We will discuss it later.)

# Doubly-Linked List. Operations

- Inserting/deleting a new element at a given position similar as as in case of a SLL.
- The main difference is that we need to set more links (we have the prev links as well) and we have to check whether we modify the tail of the list.

# Doubly-Linked List. Example: operation insertLast

- Inserting a new element at the end of a DLL is simple, because we have the tail of the list

```
subalgorithm insertLast(dll, elem) is:  
  //pre: dll is a DLL, elem is TElem  
  //post: elem is added to the end of dll  
  newNode ← allocate() //allocate a new DLLNode  
  [newNode].info ← elem  
  [newNode].next ← NIL  
  [newNode].prev ← dll.tail  
  if dll.head = NIL then //the list is empty  
    dll.head ← newNode  
    dll.tail ← newNode  
  else  
    [dll.tail].next ← newNode  
    dll.tail ← newNode  
  end-if  
end-subalgorithm
```

**Complexity:**  $\Theta(1)$

# DLL. Operation deleteElement

e.g.: Delete a given element

```
function deleteElement(dll, elem) is:
  currentNode ← dll.head
  while currentNode ≠ NIL and [currentNode].info ≠ elem execute
    currentNode ← [currentNode].next
  end-while
  deletedNode ← currentNode
  if currentNode ≠ NIL then
    if currentNode = dll.head then
      if currentNode = dll.tail then
        dll.head ← NIL ; dll.tail ← NIL
      else
        dll.head ← [dll.head].next ; [dll.head].prev ← NIL
      end-if
    else if currentNode = dll.tail then
      dll.tail ← [dll.tail].prev ; [dll.tail].next ← NIL
    else
      [[currentNode].next].prev ← [currentNode].prev
      [[currentNode].prev].next ← [currentNode].next
    end-if
    free(currentNode)
    deleteElement ← True
  else
    deleteElement ← False
  end-if
end-function
```

**Complexity:  $O(n)$**

# DLL vs. SLL

Double-linked lists ...

- require more space per node ... than SLL
- sequential access in both directions (easier to manipulate)

Which operations are more expensive ?

- elementary operations ?
- one can insert or delete a node in a constant number of operations given only that node's address, compared with singly-linked lists, which require the previous node's address in order to insert or delete

**Specific iterators**

- SLL - forward iterator
- DLL - bidirectional iterator

# Linked List vs. Array

- **extra-storage & linked-list**

- ? total overhead, overhead per element

- allocate memory separately for each new element - slow

- array: operation resize** - an expensive operation

- **operations**

- new items can be added or deleted anywhere in the list, managing the links between them elements, without high costs (constant time complexity);

- elements are never moved (important if copying an element takes a lot of time).

- access to the element based on index is difficult (complexity of linear time)

## **When to use linked-list?**

- if you have elements that are frequently added and deleted
- when the dimension of an elements is significant

*As with most choices in computer programming and design, no method is well suited to all circumstances. A linked list data structure might work well in one case, but cause problems in another.*



# Think about:

1. Consider ADT IndexedList and list traversal by using:

a) by using the iterator

```
while (valid(it)) execute
    e ← getElement(it)
    @process e
    next(it)
endwhile
```

b) by using indexing

```
for i ← 1 , n execute
    e ← getElement(l, i)
    @process e
endwhile
```

Analyze traversal time complexity, when:

- i) List is represented over an array
- ii) List is represented over a linked list

2. Consider ADT SortedList and search operation.

Which is the total time complexity that we can achieve

by choosing an appropriate algorithm for the next 3 representations ?

- i) Array
- ii) SLList
- iii) DLList

# Linked Lists

- **Circularly-linked list**

the first and final nodes are linked together

- **Sentinel nodes**

node with default value for *info* : sentinel

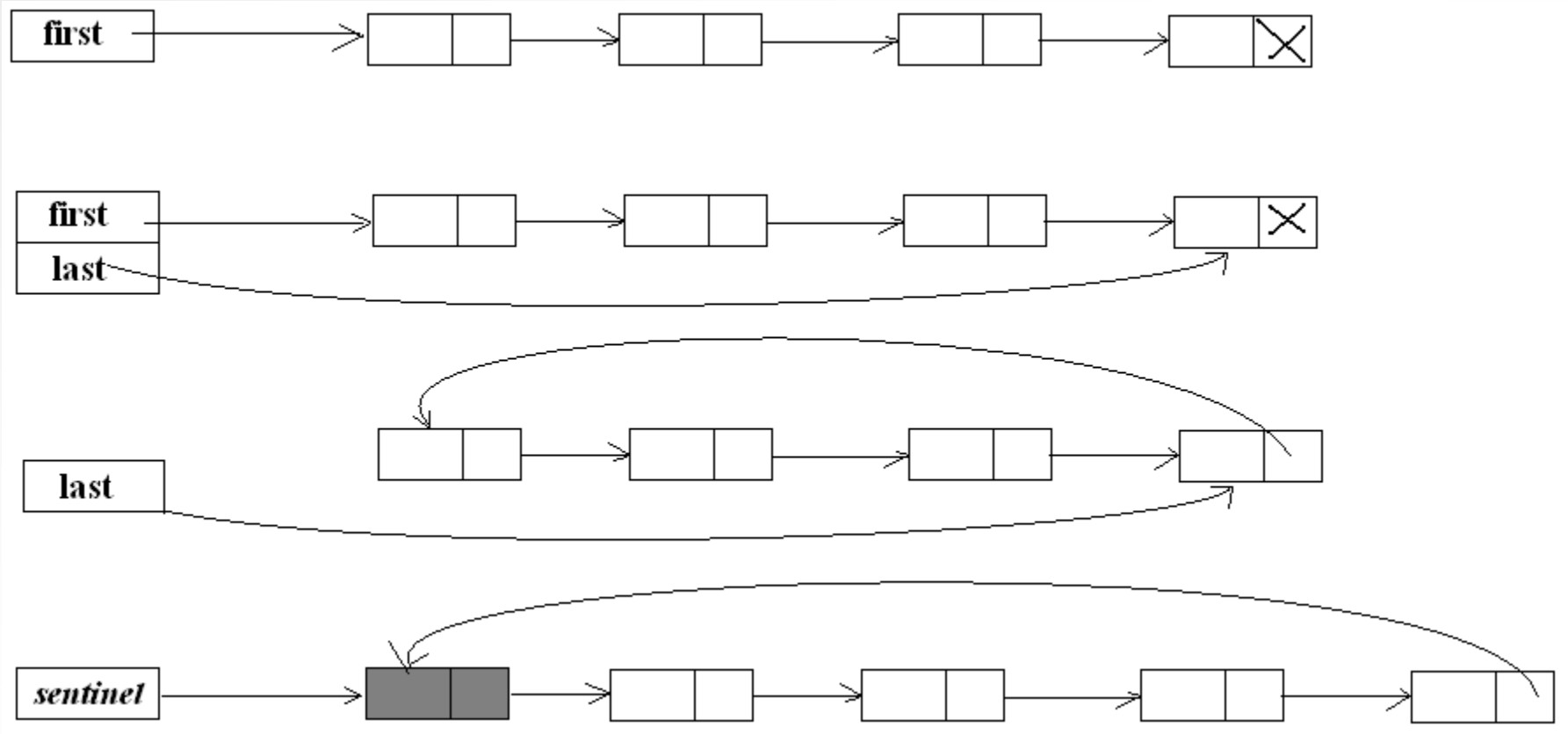
→ avoid special value  $\perp$  for position

- may simplify certain operations

- ensure that next and/or previous nodes exist for any element

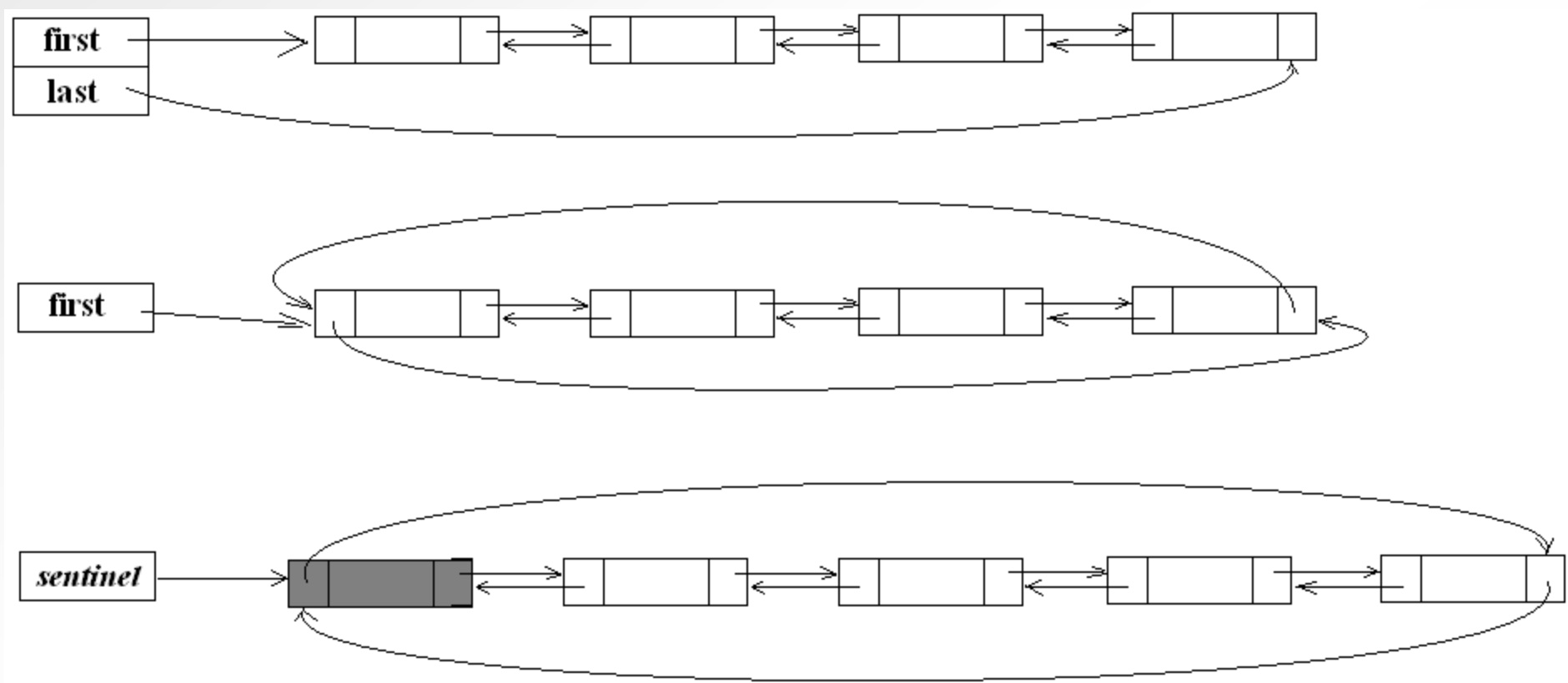
- use extra space

# Singly-linked list (representation choices/variations)



? SLList with fast addLast

# Doubly-linked and circular list variations



# DLL with sentinel. Operation deleteElement

DLL:

sentinel:  $\uparrow$  DLLNode

e.g.: Delete a given element

```
function deleteElement(dll, elem) is:
  currentNode  $\leftarrow$  dll.sentinel.next
  while currentNode  $\neq$  NIL and [currentNode].info  $\neq$  elem execute
    currentNode  $\leftarrow$  [currentNode].next
  end-while
  deletedNode  $\leftarrow$  currentNode
  if currentNode  $\neq$  NIL then
    [[currentNode].next].prev  $\leftarrow$  [currentNode].prev
    [[currentNode].prev].next  $\leftarrow$  [currentNode].next
    free(currentNode)
    deleteElement  $\leftarrow$  True
  else
    deleteElement  $\leftarrow$  False
  end-if
end-function
```

Complexity:  $O(n)$