

INFDEV036A - Algorithms

Lesson Unit 3

G. Costantini, F. Di Giacomo

costg@hr.nl, giacf@hr.nl - Office H4.206

A few important reminders

- ▶ Check your progress in the dashboard of GO
 - ▶ Every set of questions should be done multiple times, until all dots are green (without any red in between)
- ▶ Make sure that you implement what we see in the lectures
 - ▶ As specified with more details in the “homework” slides
 - ▶ Remember to test your code with many possible inputs!!!
- ▶ Ask when you have doubts / don't understand



Today

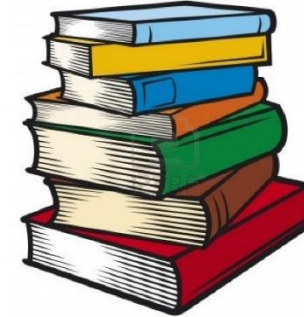


LIST

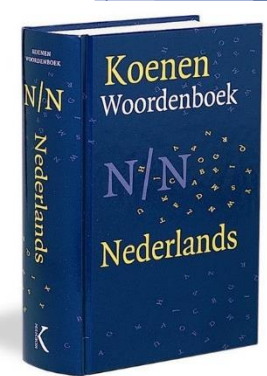


QUEUE

- ▶ ~~Why is my code slow?~~
 - ▶ ~~Empirical and complexity analysis~~
- ▶ ~~How do I order my data?~~
 - ▶ ~~Sorting algorithms~~
- ▶ How do I structure my data?
 - ▶ Linear, tabular, recursive data structures
- ▶ How do I represent relationship networks?
 - ▶ Graphs



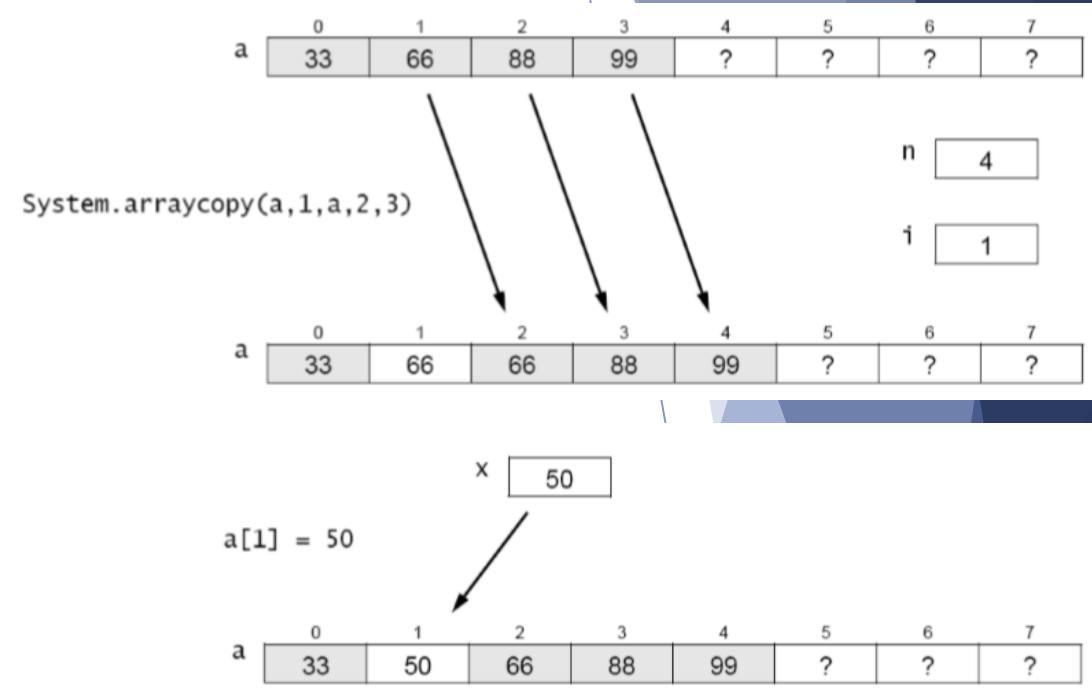
STACK



HASH TABLE

Why arrays are not enough?

- ▶ Arrays are good for...
 - ▶ Sequential access (cache)
- ▶ But not for...
 - ▶ Algorithmic stuff on dynamic data
- ▶ Why?
 - ▶ In an unsorted array, *searching* is slow
 - ▶ Linear search instead of binary search
 - ▶ But to maintain an array sorted, *inserting* & *deleting* elements is slow
 - ▶ Need to shift all elements bigger than the one to insert/delete
 - ▶ Possible resize needed

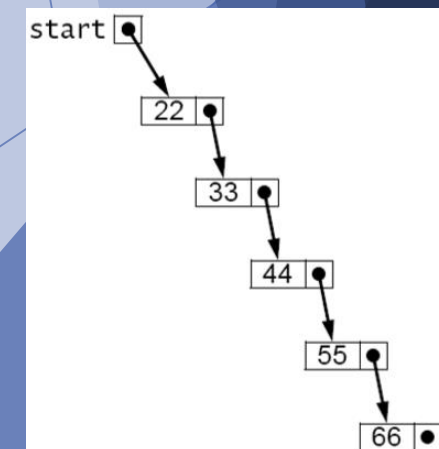
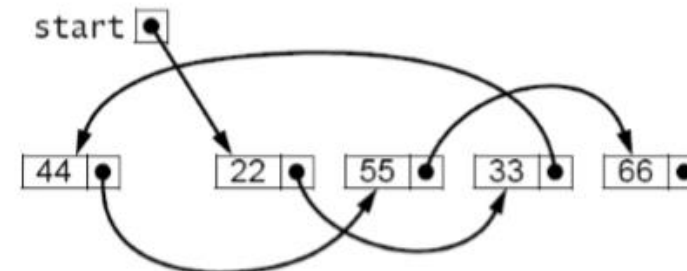
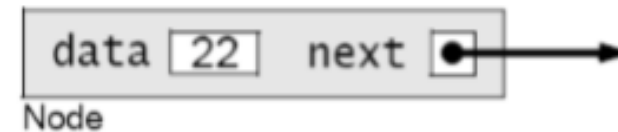




Linked lists

Linked list

- ▶ Simple and flexible representation
- ▶ Objects are arranged in linear order
 - ▶ Order is maintained through the use of *references* inside elements
- ▶ Each element (*node*) of a list is made by
 - ▶ Its value
 - ▶ A reference to the next element of the list
- ▶ A *list* is then defined by
 - ▶ The starting element
 - ▶ All other elements can be reached from there



Linked list operations: SEARCH

- ▶ Given a value k and a list L ...
 - ▶ finds the first element with value k in the list L by a simple linear search
 - ▶ if no object with value k appears, the procedure returns NIL

```
LIST-SEARCH(L,k)
  p = L.start
  while p ≠ NIL and p.data ≠ k
    p = p.next
  return p
```

- ▶ Complexity (worst case)?
 - ▶ $O(n)$ since it may have to search the entire list

Linked list operations: SEARCH

```
LIST-SEARCH(L,k)
```

```
  p = L.start
```

```
  while p ≠ NIL and p.data ≠ k
```

```
    p = p.next
```

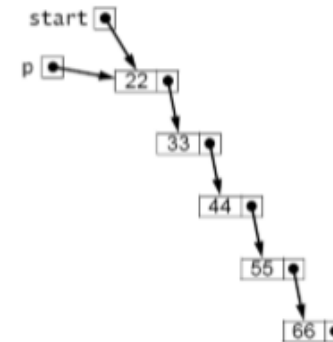
```
  return p
```

► Example: looking for $k = 44$

► First iteration: p is the start node (containing 22)

► Second iteration: p is the second node (containing 33)

► Third (and last) iteration: p is the third node (containing $k = 44$)



Linked list operations: INSERT

- ▶ Given a value k and a (sorted) list $L...$
 - ▶ finds the right position in the list for k through a simple linear search

looking for the position

```
LIST-INSERT(L,k)
  p = L.start
  while p.next ≠ NIL and p.next.data ≤ k
    p = p.next
```

Linked list operations: INSERT

- ▶ Given a value k and a (sorted) list L ...
 - ▶ finds the right position in the list for k through a simple linear search
 - ▶ inserts a new element with value k in such position

looking for the position {

```
LIST-INSERT(L,k)
  p = L.start
  while p.next ≠ NIL and p.next.data ≤ k
    p = p.next
  p.next = new Node(k, p.next)
  return
```

insertion of the new node ←

Linked list operations: INSERT

► Example: inserting 50

looking for the position

insertion of the new node

```
LIST-INSERT(L,k)
```

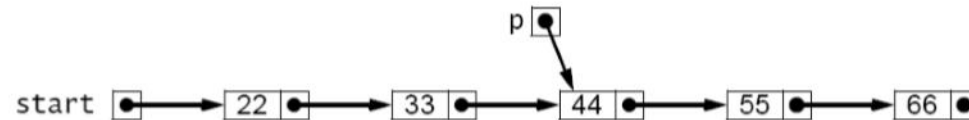
```
  p = L.start
```

```
  while p.next ≠ NIL and p.next.data ≤ k
```

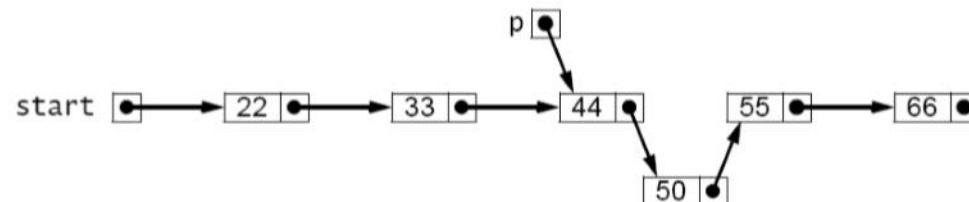
```
    p = p.next
```

```
  p.next = new Node(k, p.next)
```

```
  return
```

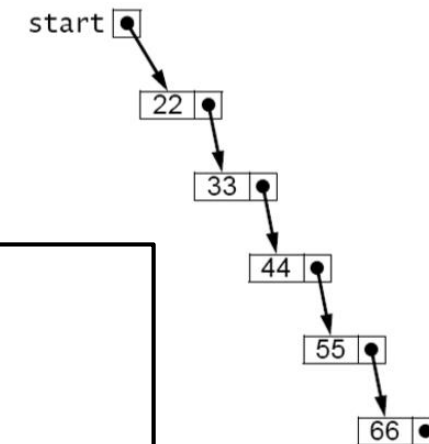


```
p.next = new Node(50, p.next)
```



Linked list operations: INSERT

- What if we tried to insert 20 in the previous example?



LIST-INSERT(L,k)

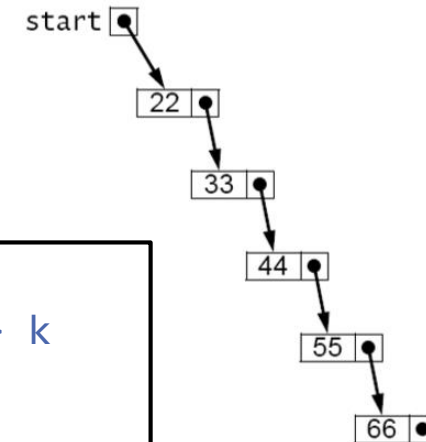
```
p = L.start
while p.next ≠ NIL and p.next.data ≤ k
    p = p.next
p.next = new Node(k, p.next)
return
```

looking for the position

insertion of the new node

Linked list operations: INSERT

- ▶ What if we tried to insert 20 in the previous example?
 - ▶ Special case: insertion AT THE FRONT of the list
 - ▶ If the element to insert is smaller than the starting one



insertion at the front
(if needed)

looking for the position

insertion of the new node

```
LIST-INSERT(L,k)
  if L.start == NIL or L.start.data > k
    L.start = new Node(k, L.start)
    return

  p = L.start
  while p.next ≠ NIL and p.next.data ≤ k
    p = p.next
  p.next = new Node(k, p.next)
  return
```

Linked list operations: INSERT

► Example: Inserting 20

insertion at the front
(if needed)

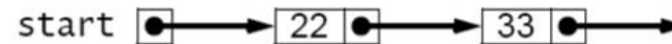
looking for the position

insertion of the new node

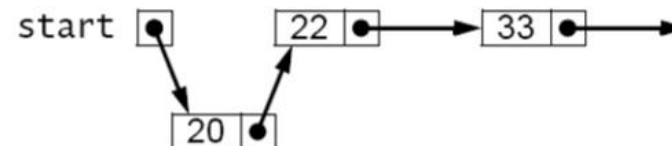
```
LIST-INSERT(L,k)
```

```
  if L.start == NIL or L.start.data > k  
    L.start = new Node(k, L.start)  
  return
```

```
  x = L.start  
  while x.next ≠ NIL and x.next.data ≤ k  
    x = x.next  
  x.next = new Node(k, x.next)  
  return
```

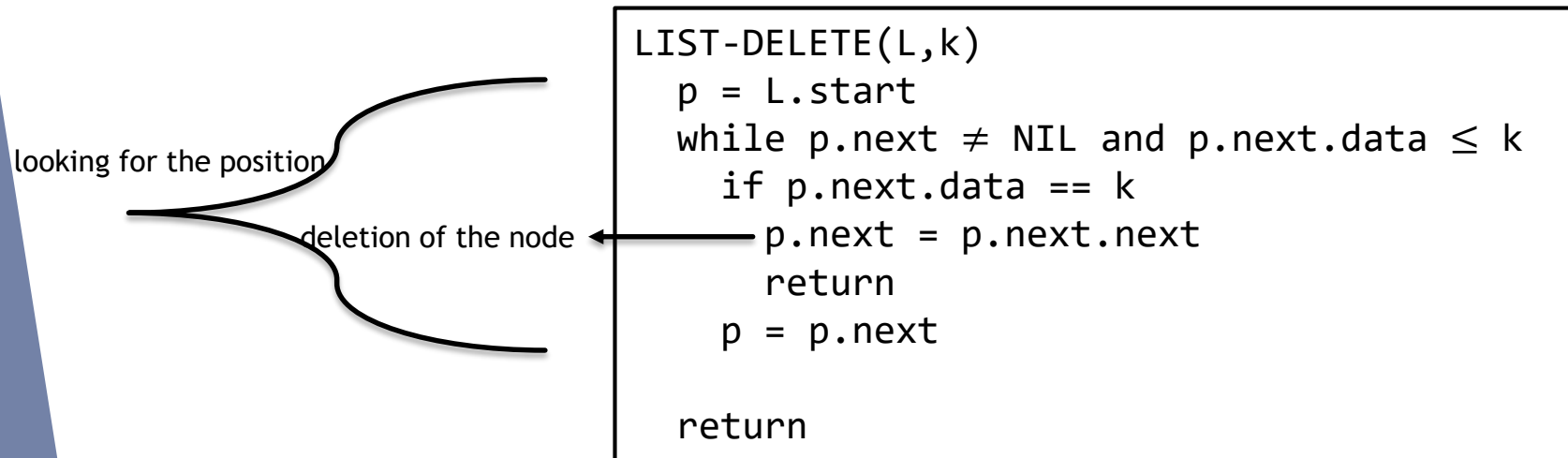


insert(start,20)



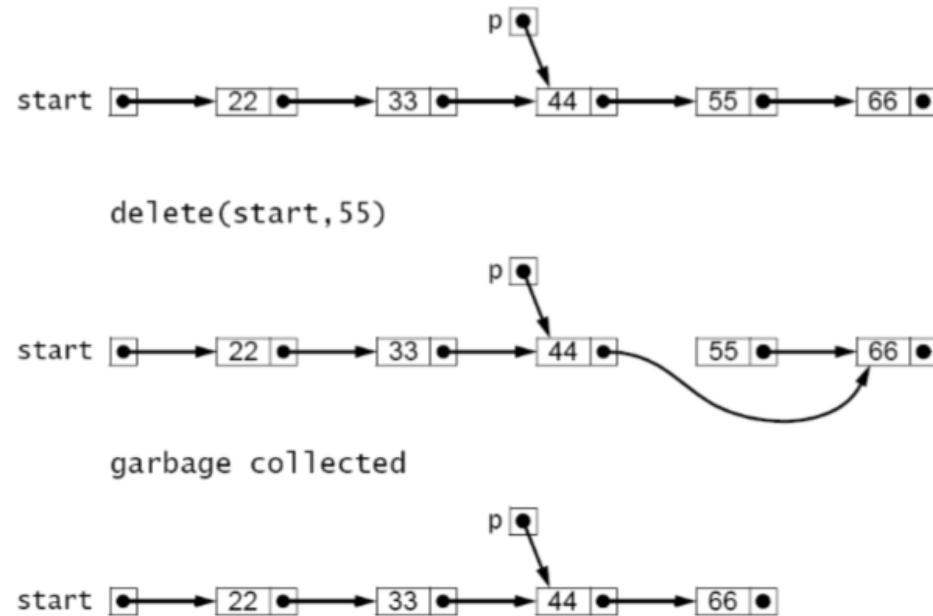
Linked list operations: DELETE

- ▶ Given a value k and a (sorted) list L ...
 - ▶ finds the first occurrence of the value k in the list through a simple linear search
 - ▶ deletes such element (if it exists!)



Linked list operations: DELETE

► Example: deleting 55



```
LIST-DELETE(L,k)
```

```
  p = L.start
```

```
  while p.next ≠ NIL and p.next.data ≤ k
```

```
    if p.next.data == k
```

```
      p.next = p.next.next
```

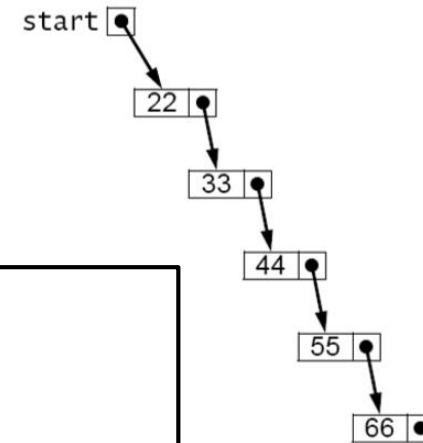
```
      return
```

```
    p = p.next
```

```
  return
```

Linked list operations: DELETE

- And if we wanted to delete 22?



LIST-DELETE(L,k)

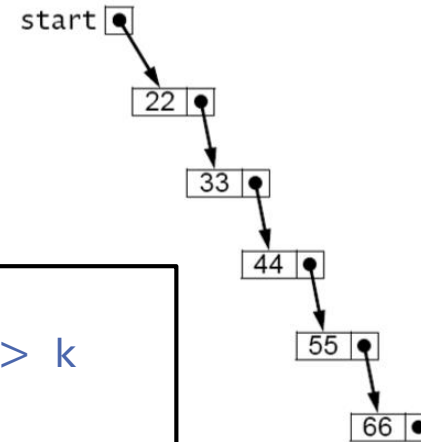
```
x = L.start
while x.next ≠ NIL and x.next.data ≤ k
    if x.next.data == k
        x.next = x.next.next
        return
    x = x.next
return
```

looking for the position

deletion of the node

Linked list operations: DELETE

- ▶ And if we wanted to delete 22?
 - ▶ Special case: deleting *the first element* of the list



```
LIST-DELETE(L,k)
  if L.start == NIL or L.start.data > k
    return
  else if L.start.data == k
    L.start = L.start.next
    return
  x = L.start
  while x.next != NIL and x.next.data ≤ k
    if x.next.data == k
      x.next = x.next.next
      return
    x = x.next
  return
```

element not in the list

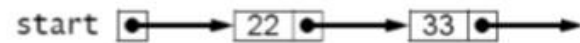
deleting the first element

looking for the position

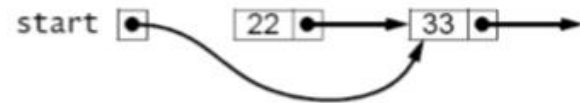
deletion of the node

Linked list operations: DELETE

► Example: deleting 22



delete(start,22)



garbage collected



```
LIST-DELETE(L,k)
```

```
  if L.start == NIL or L.start.data > k  
    return
```

```
  else if L.start.data == k  
    L.start = L.start.next  
    return
```

```
  x = L.start
```

```
  while x.next ≠ NIL and x.next.data ≤ k
```

```
    if x.next.data == k
```

```
      x.next = x.next.next
```

```
      return
```

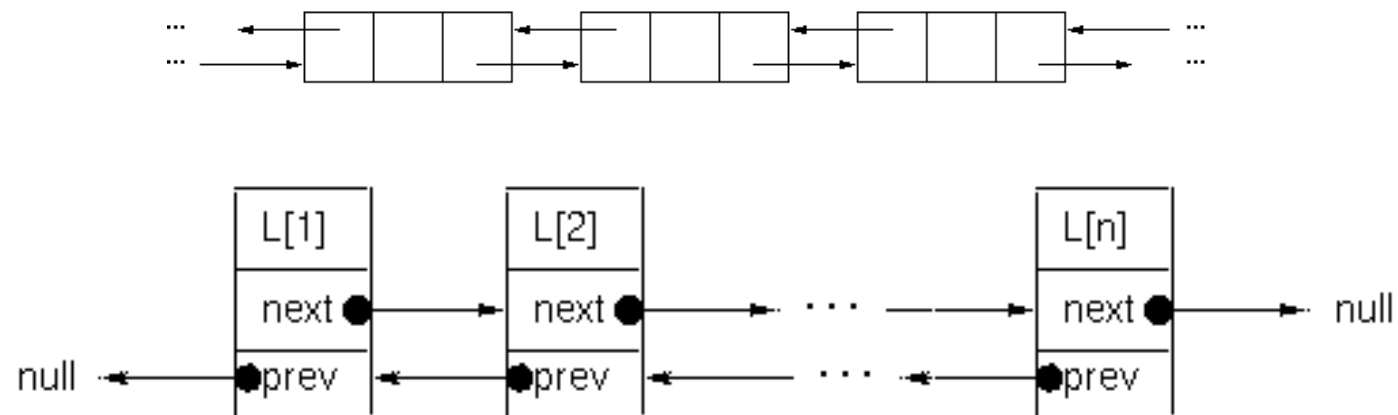
```
      x = x.next
```

```
  return
```

Doubly linked list

- ▶ What if we want to move both forward and backward?
 - ▶ Add another reference to the node: the previous element in the list

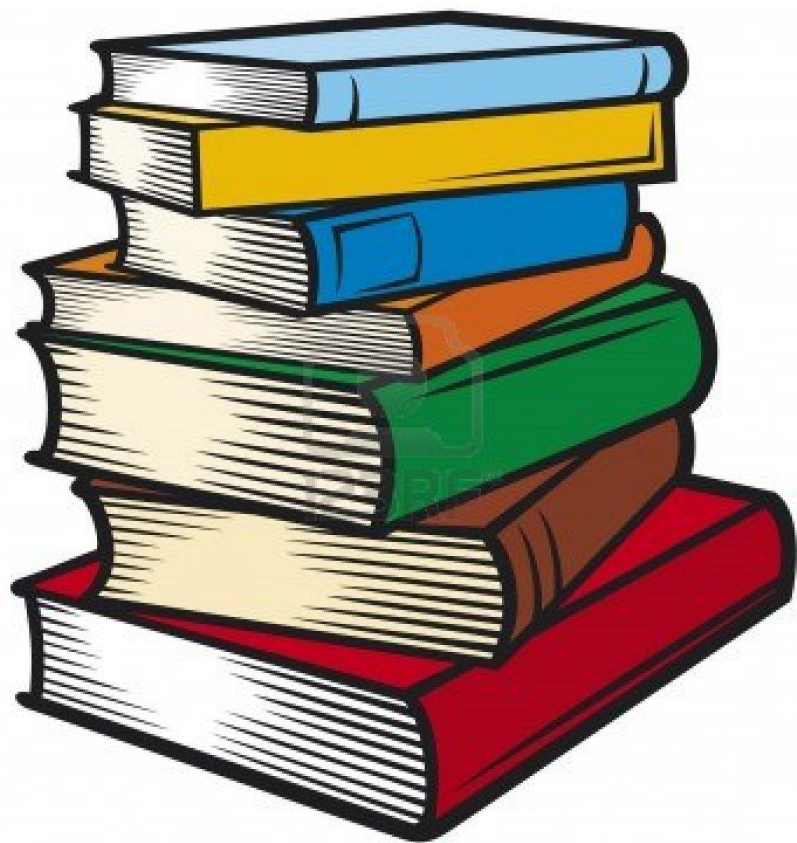
A Doubly-Linked List



Implementation exercises

- ▶ Write the code of all previous algorithms *and* to...
 - ▶ Insert a new value after/before a certain node in a doubly linked list
 - ▶ function `insertAfter<T>(List<T> list, Node<T> node, T newValue)`
 - ▶ function `insertBefore<T>(List<T> list, Node<T> node, T newValue)`
 - ▶ Insert a new value at the beginning and end of a doubly linked list
 - ▶ function `insertBeginning<T>(List<T> list, T newValue)`
 - ▶ function `insertLast<T>(List<T> list, T newValue)`
 - ▶ Delete a certain value in a doubly linked list
 - ▶ function `remove<T>(List<T> list, T value)`

```
DoublyLinkedListNode {
    prev // A reference to the previous node
    next // A reference to the next node
    data // Data or a reference to data
}
DoublyLinkedList {
    DoublyLinkedListNode firstNode // points to first node of list
    DoublyLinkedListNode lastNode // points to last node of list
}
```



Stack

Stack - Definition

- ▶ Collection implementing the LIFO protocol

- ▶ LIFO = Last In First Out
- ▶ Only accessible object: last one inserted



- ▶ Operations allowed

- ▶ Adding an element onto the top of the stack (**PUSH**)
- ▶ Accessing the current element on the top of the stack (**PEEK**)
- ▶ Removing the current element on the top of the stack (**POP**)

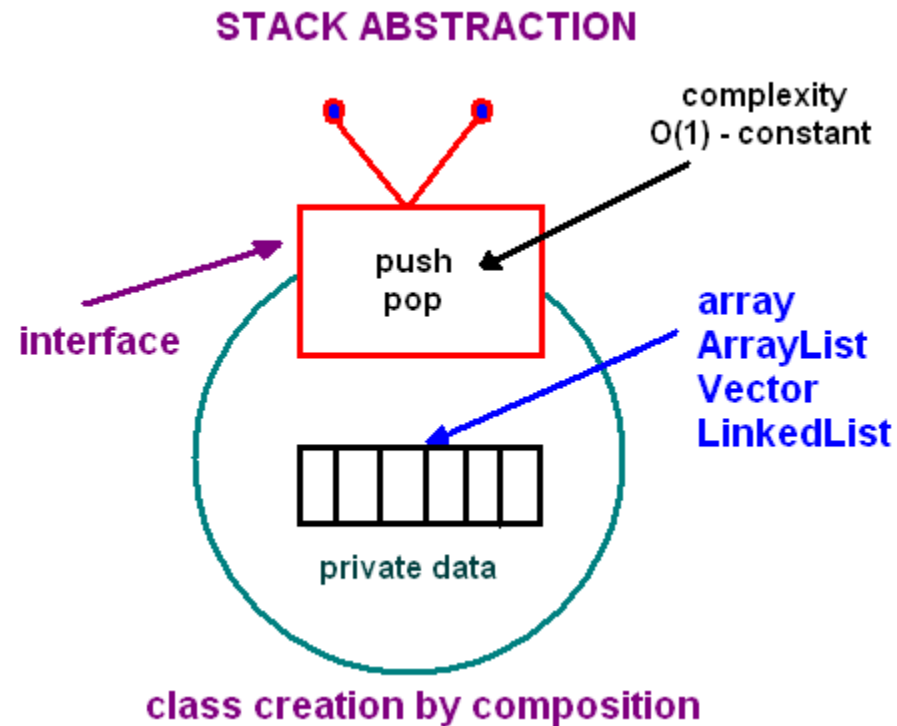
Stack - Implementation

- ▶ Built on top of other data structures
 - ▶ array, linked list, ...
- ▶ However, it implements always the same functionality
 - ▶ defined by the following interface

```
public interface StackInterface<T>
{
    void push(T e);
    T pop();
    T peek();
    boolean isEmpty();
}
```

Stack - Implementation

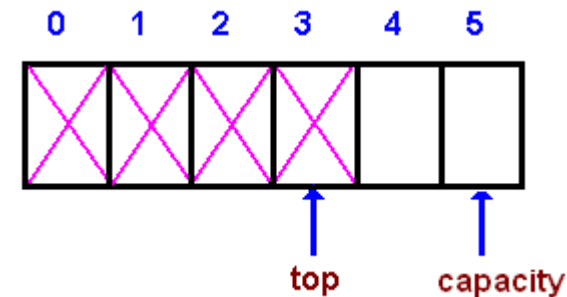
- Built on top of other data structures, but implementing always the same functionality



Stack - Indexed implementation

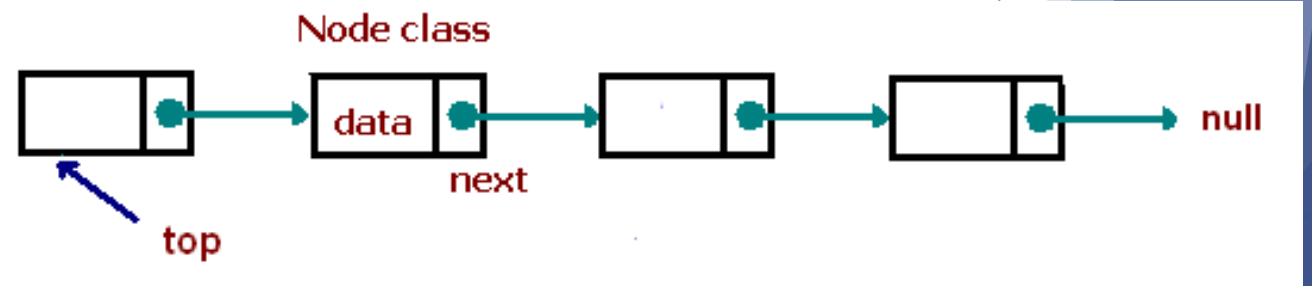
Fields of the implementation

- ▶ Array A of a default size
- ▶ Variable top (reference to the top element)
- ▶ Variable $capacity$ (last index of the array)
- ▶ Stack empty $\Leftrightarrow top = -1$
- ▶ Stack full $\Leftrightarrow top = capacity$
 - ▶ Static implementation \rightarrow adding another element throws exception
 - ▶ Dynamic implementation \rightarrow double the size of the stack



Stack - Linked implementation

- ▶ Best (in efficiency) dynamic stack implementation
 - ▶ Be careful at the special case of empty stack
- ▶ Top?
 - ▶ starting element of the list
- ▶ Access (peek)?
 - ▶ Read the content of the top
- ▶ Push?
 - ▶ Create a new node and add it at the beginning of the list
- ▶ Pop?
 - ▶ Move the beginning of the list at the second element



Stack - Exercise

- What is the content of the stack after executing all following operations?

```
Push(3)
Push(5)
Push(1)
Pop()
Pop()
Push(8)
Pop()
Pop()
Push(2)
Push(4)
Pop()
Push(7)
```

Answer:

```
{ 7, 2 }
```

(note: the top of the stack is written at the left of the sequence)



Queue

Queue - Definition

- ▶ Collection implementing the FIFO protocol
 - ▶ FIFO = First In First Out
 - ▶ Only accessible object: first one inserted
 - ▶ In the stack it's the opposite (last one inserted)
- ▶ Operations allowed
 - ▶ Adding an element to the back of the queue (**ENQUEUE**)
 - ▶ Accessing the current element at the front of the queue (**PEEK**)
 - ▶ Removing the current element at the front of the queue (**DEQUEUE**)



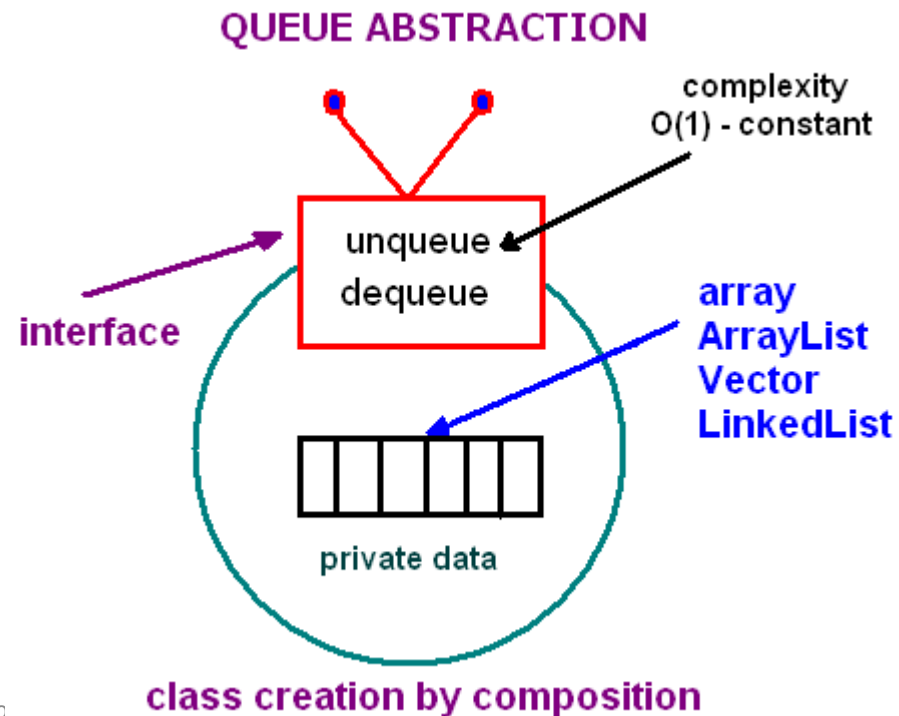
Queue - Implementation

- ▶ Built on top of other data structures
 - ▶ array, linked list, ...
- ▶ However, it implements always the same functionality
 - ▶ defined by the following interface

```
public interface QueueInterface<T>
{
    void enqueue(T e);
    T peek();
    T dequeue();
    boolean isEmpty();
}
```

Queue - Implementation

- Built on top of other data structures, but implementing always the same functionality



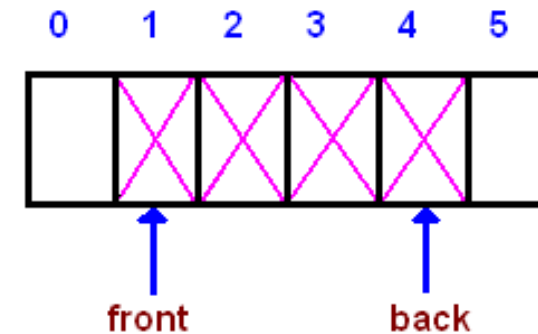
Queue - Indexed implementation

Fields

- ▶ Array A
- ▶ Variable $front$ (reference to the front of the queue)
- ▶ Variable $back$ (reference to the back of the queue)

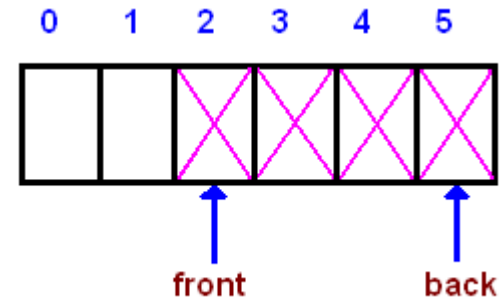
The queue moves in the array from left to right

- ▶ Inserting a new item (enqueue) → increase the back index
- ▶ Removing an item (dequeue) → increase the front index

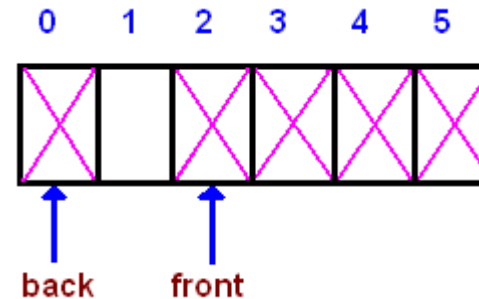


Queue - Indexed implementation

- ▶ What happens when *back* reaches the end of the array?



- ▶ We can use the free space before the front index to store new items
 - ▶ *Wrap around queue or Circular queue*

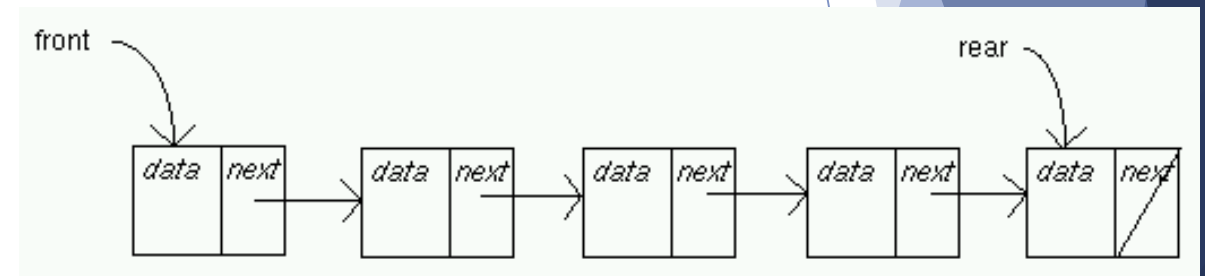


Queue - Indexed implementation

- ▶ And what happens when *back* reaches *front*?
 - ▶ The queue is completely full
 - ▶ Two choices to handle this situation (as with the stack)
 - ▶ Throw exception
 - ▶ Double the array size

Queue - Linked implementation

- ▶ Almost the same as the stack linked implementation
 - ▶ Here we maintain also a pointer to the last element
- ▶ Front → starting element of the list
- ▶ Rear → last element of the list
- ▶ Enqueue
 - ▶ Create a new node and add it at the end of the list
- ▶ Dequeue
 - ▶ Move the beginning of the list at the second element



Queue - Exercise

- What is the content of the queue after executing all following operations?

```
Enqueue(3)
Enqueue(5)
Enqueue(1)
Dequeue()
Dequeue()
Enqueue(8)
Dequeue()
Enqueue(2)
Enqueue(4)
Dequeue()
Enqueue(7)
```

Answer: { 2, 4, 7 }

(note: the front of the queue is written at the left of the sequence)

Lists, stacks, queues in .NET

- ▶ [http://msdn.microsoft.com/en-US/library/ms379570\(v=vs.80\).aspx](http://msdn.microsoft.com/en-US/library/ms379570(v=vs.80).aspx)
- ▶ [http://msdn.microsoft.com/en-us/library/ms379571\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/ms379571(v=vs.80).aspx)
- ▶ <http://www.dotnetperls.com/list>
- ▶ <http://www.dotnetperls.com/stack>
- ▶ <http://www.dotnetperls.com/queue>

Homework

- ▶ Study the slides
- ▶ Answer the *MC questions* on GrandeOmega
- ▶ **Implement**
 - ▶ *SortedList* $< T >$ (with the associated insert/search/delete operations) and *DoublyLinkedList* $< T >$ (operations specified in the slide “implementation exercises”)
 - ▶ *Queue* $< T >$
 - ▶ *Stack* $< T >$

See you next week 😊