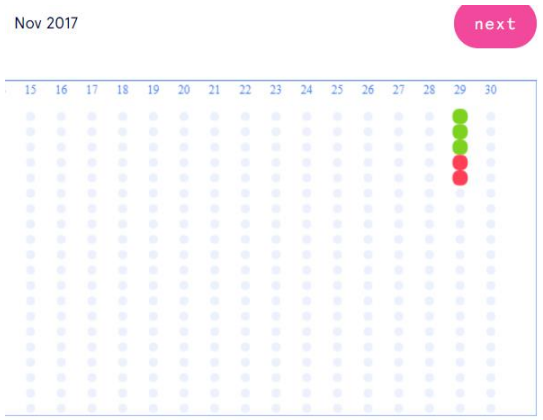
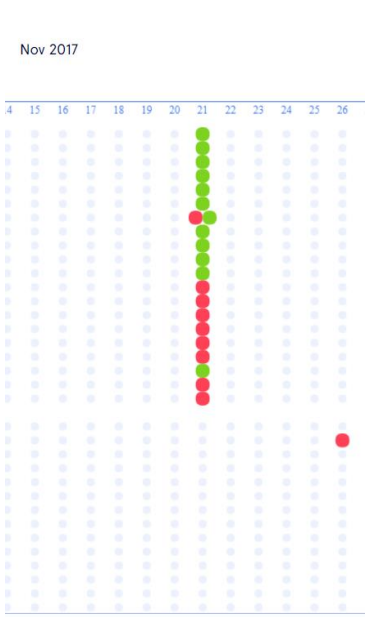
INFDEV036A - Algorithms Lesson Unit 3

G. Costantini, F. Di Giacomo

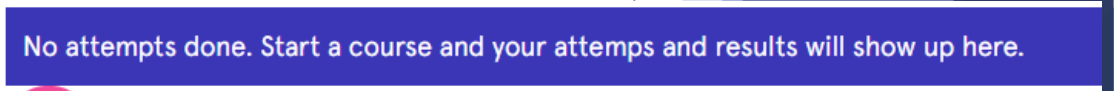
costg@hr.nl, giacf@hr.nl – Office H4.206

A few important reminders

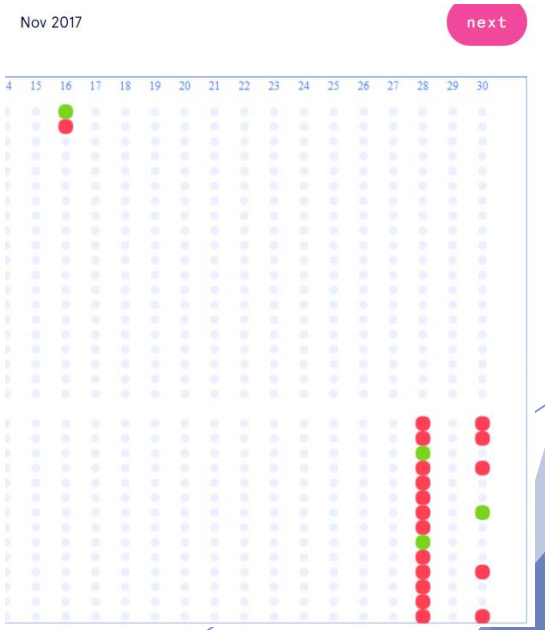


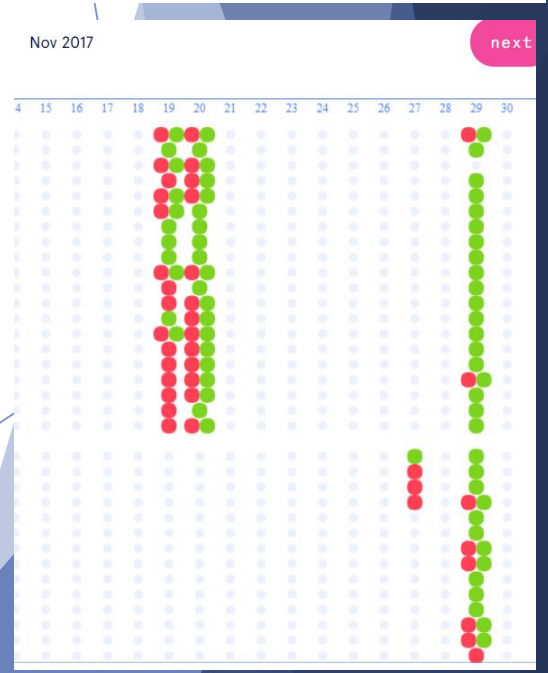


- Check your progress in the dashboard of GO
  - Every set of questions should be done multiple times, until all dots are green (without any red in between)



- 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!!!





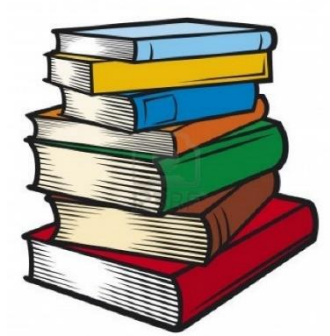
- Ask when you have doubts / don’t understand

Today

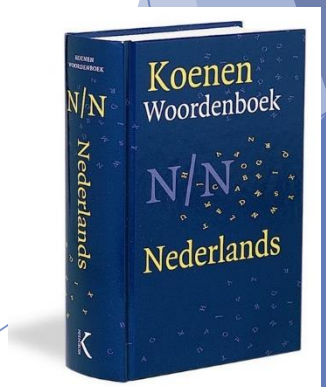
LIST

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**



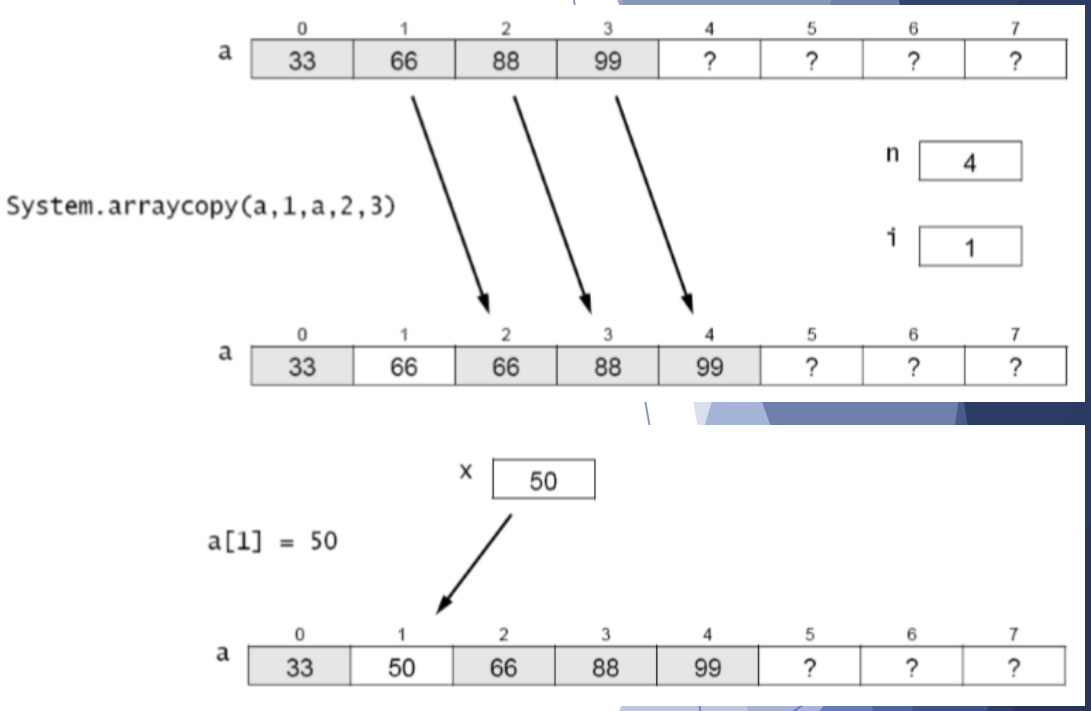
STACK



HASH TABLE

Why arrays are not enough?

- Arrays are good for…
  - Sequential access (cache)
- But not for…
  - Algorithmic stuff on dynamic data



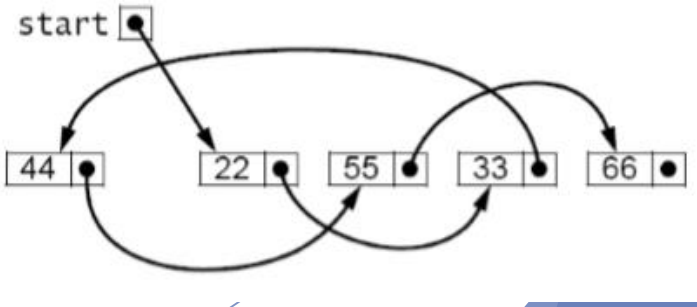
- Why?
  - In an unsorted array, *searching* is slow
    - Linear search instead of binary search
  - But to maintain an array sorted, *inserting* & *deleting* elements is slow
    - Need to shift all elements bigger than the one to insert/delete
    - Possible resize needed

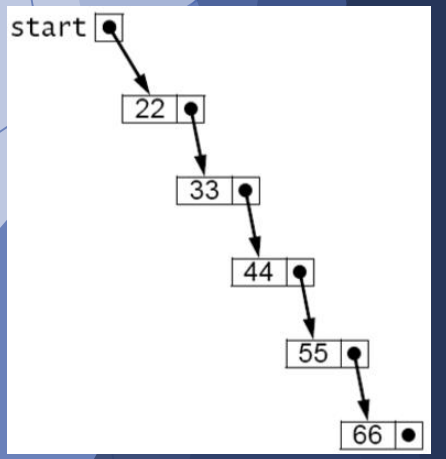
Linked lists

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

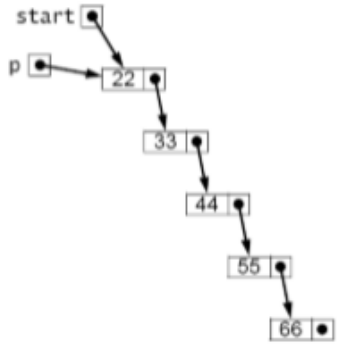
```
LIST-SEARCH(L,k)
  p = L.start
  while p ≠ NIL and p.data ≠ k
    p = p.next
  return p
```

- 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

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 L.start
```

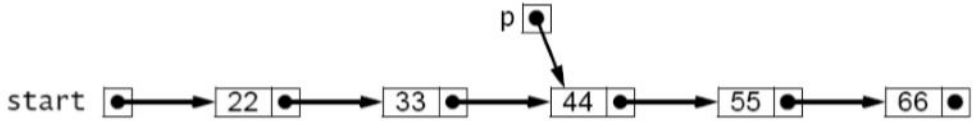
Linked list operations: INSERT

- Example: inserting 50

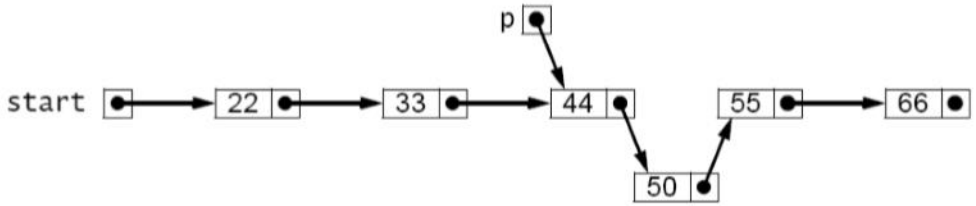
looking for the position

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 L.start
```

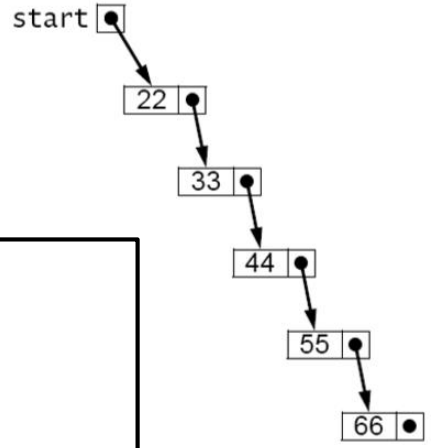


p.next = new Node(50,p.next)



Linked list operations: INSERT

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



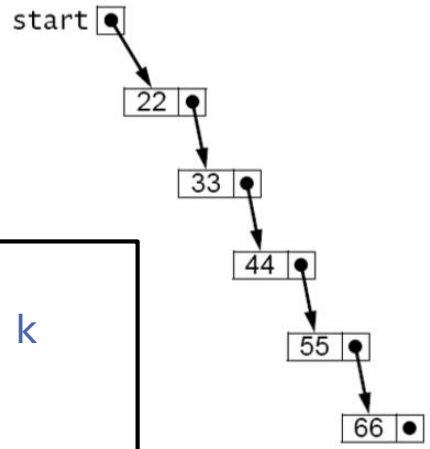

```
LIST-INSERT(L,k)




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

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 L.start

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

Linked list operations: INSERT

- Example:

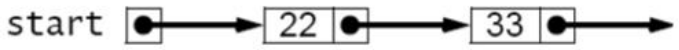
Inserting 20

insertion at the front (if needed)

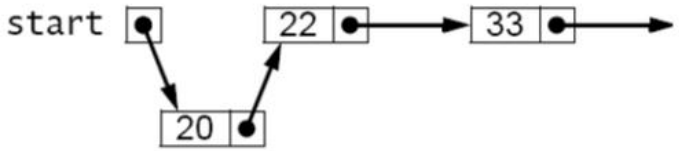
looking for the position

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 L.start

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

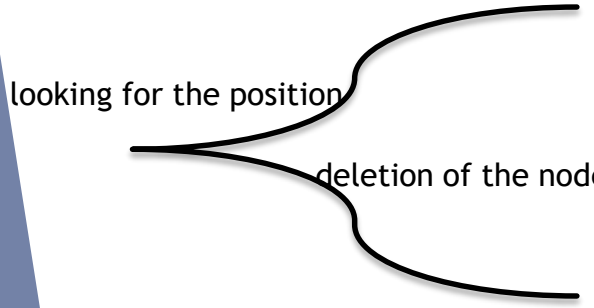


insert(start,20)



Linked list operations: DELETE

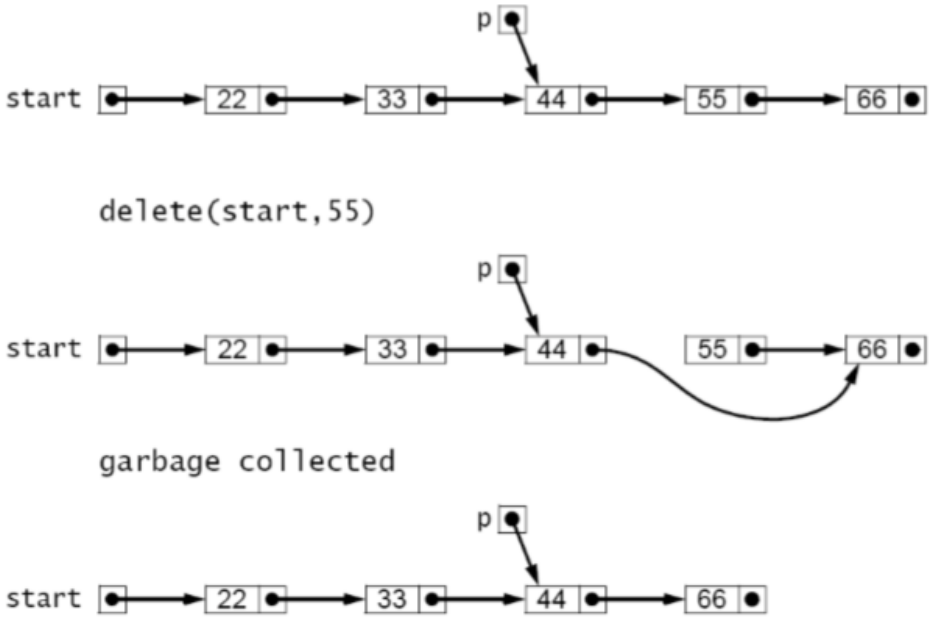
- Given a value k and a (sorted) list L…
  - finds the first occurrence of the value k in the list through a simple linear search
  - deletes such element (if it exists!)




```
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 L.start
    p = p.next

  return L.start
```

Linked list operations: DELETE

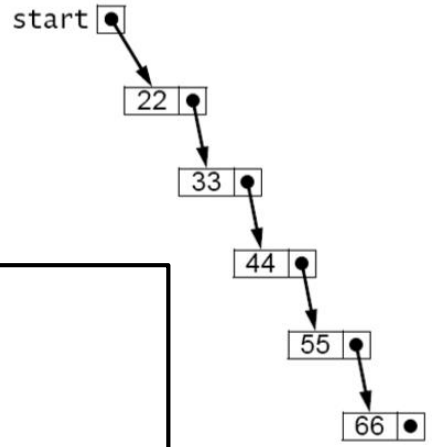
- Example: deleting 55




```
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 L.start
    p = p.next

  return L.start
```

Linked list operations: DELETE

- And if we wanted to delete 22?



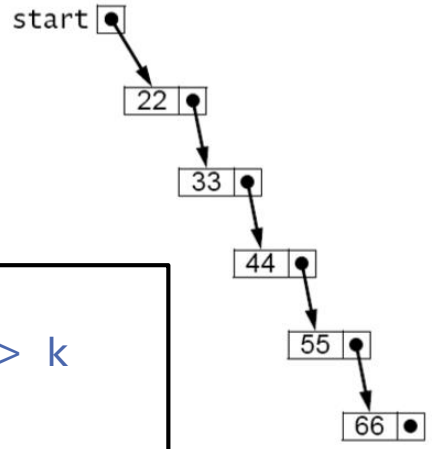

```
LIST-DELETE(L,k)

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

  return L.start
```

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



element not in the list

deleting the first element

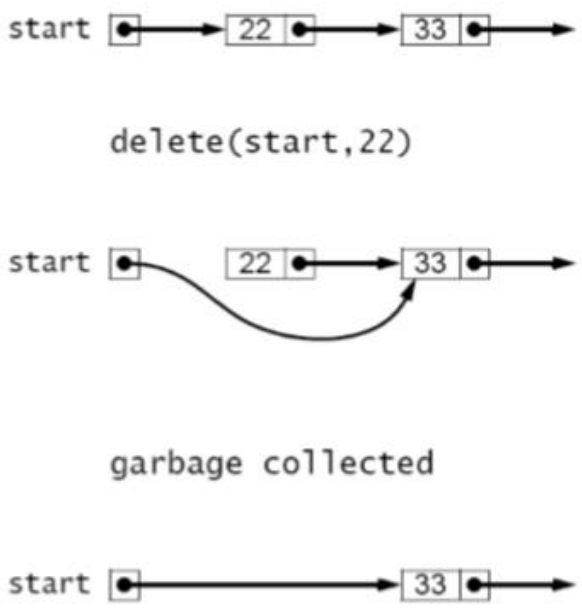
looking for the position

deletion of the node

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

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

  return L.start
```

Linked list operations: DELETE

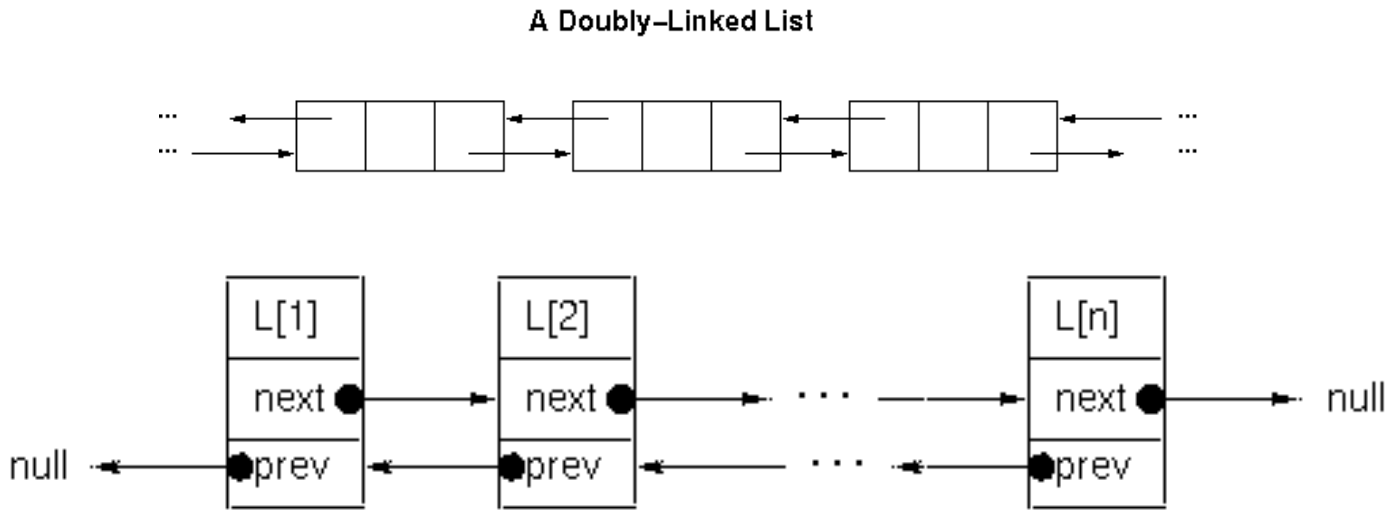
- Example: deleting 22




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

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

  return L.start
```

Doubly linked list

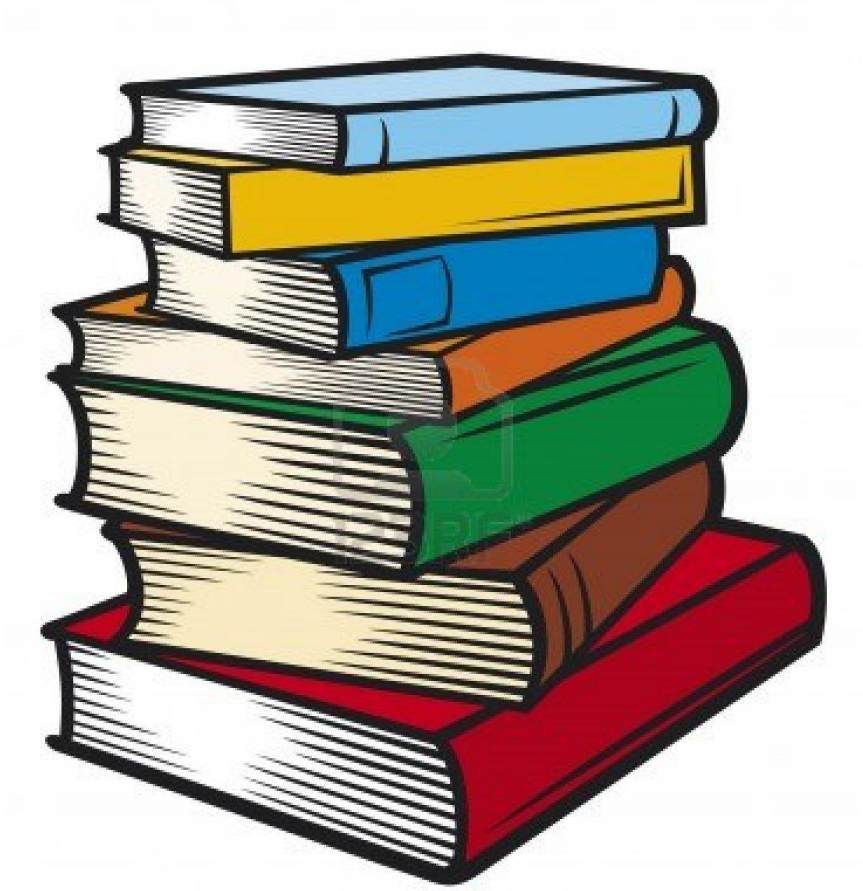
- What if we want to move both forward and backward?
  - Add another reference to the node: the *previous* element in the list



Implementation exercises

- Write the code to…
  - Insert a new node after/before a certain node in a doubly linked list
    - `function insertAfter(List list, Node node, Node newNode)`
    - `function insertBefore(List list, Node node, Node newNode)`
  - Insert a new node at the beginning and end of a doubly linked list
    - `function insertBeginning(List list, Node newNode)`
    - `function insertLast(List list, Node newNode)`
  - Delete a certain node in a doubly linked list
    - `function remove(Lis list, Node node)`

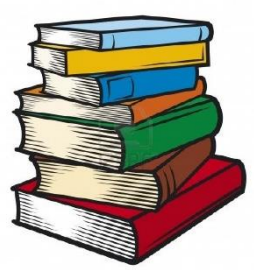
```
DoublyLinkedNode {
    prev // A reference to the previous node
    next // A reference to the next node
    data // Data or a reference to data
}
DoublyLinkedList {
     DoublyLinkedNode firstNode   // points to first node of list
     DoublyLinkedNode lastNode    // points to last node of list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



Stack

Stack – Definition

- Collection implementing the LIFO protocol
  - LIFO = **L**ast **I**n **F**irst **O**ut
  - 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**)

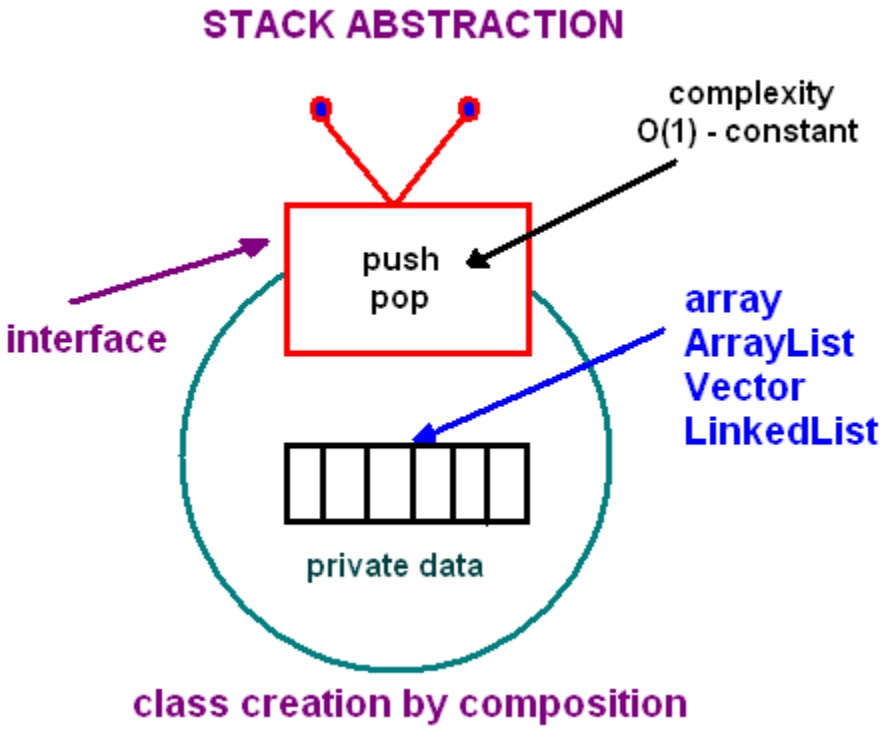
Stack – Implementation

- 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

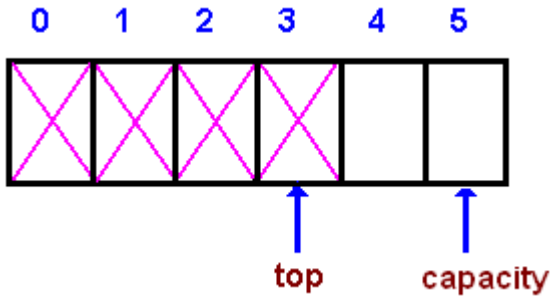
- Built on top of other data structures, but implementing always the same functionality



Stack – Indexed implementation

Fields of the implementation

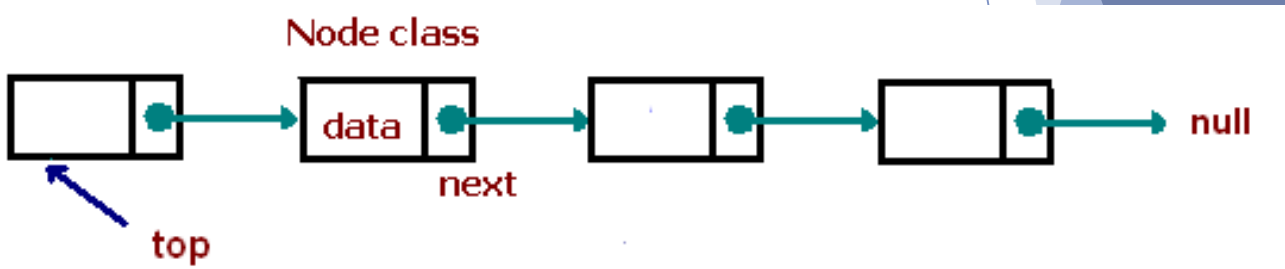
- Array A of a default size
- Variable top (reference to the top element)
- Variable $capacity$ (last index of the array)



- Stack empty $\Leftrightarrow top = -1$
- Stack full $\Leftrightarrow top = capacity$
  - Static implementation → adding another element throws exception
  - Dynamic implementation → 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



Queue

Queue – Definition

- Collection implementing the FIFO protocol
  - FIFO = **F**irst **I**n **F**irst **O**ut
  - 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**)

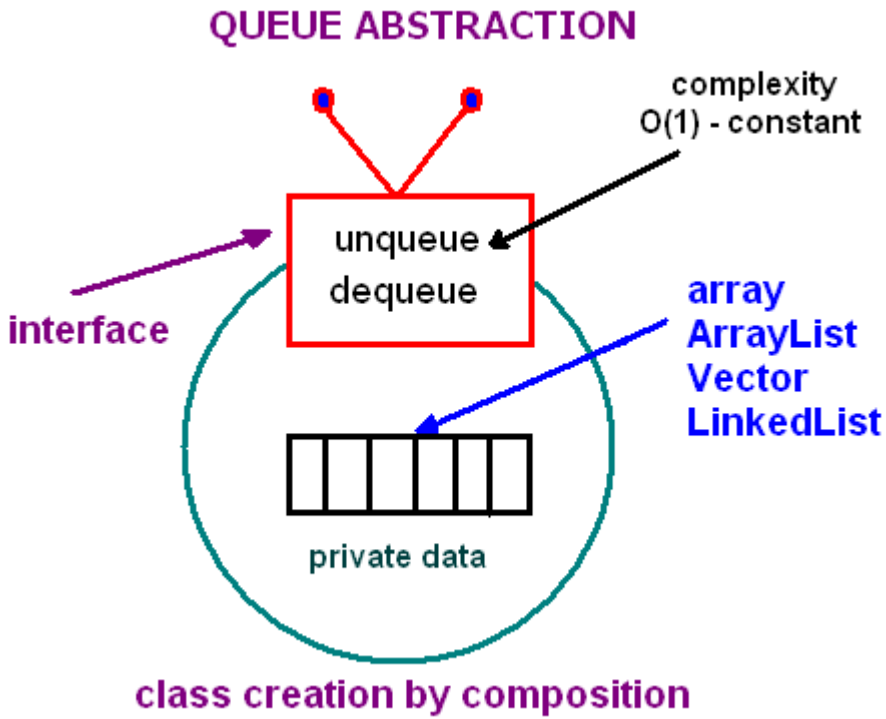
Queue – Implementation

- 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

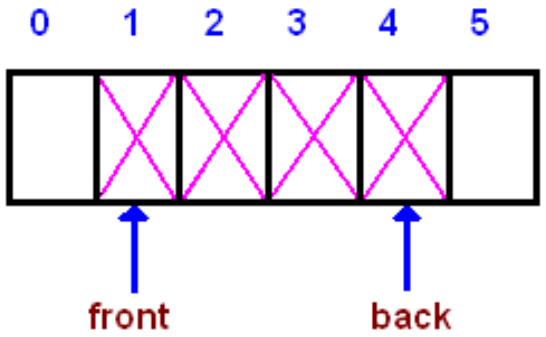
- Built on top of other data structures, but implementing always the same functionality



Queue – Indexed implementation

Fields

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

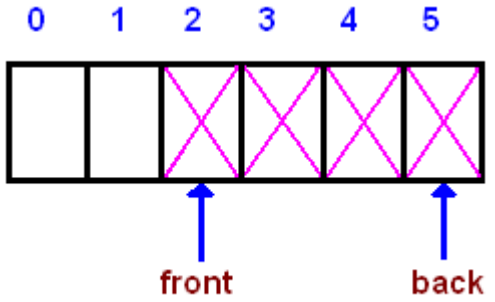


The queue moves in the array from left to right

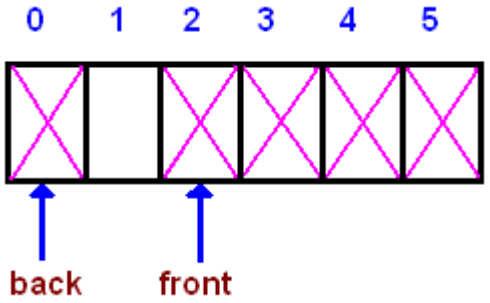
- Inserting a new item (enqueue) → increase the back index
- Removing an item (dequeue) → increase the front index

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*

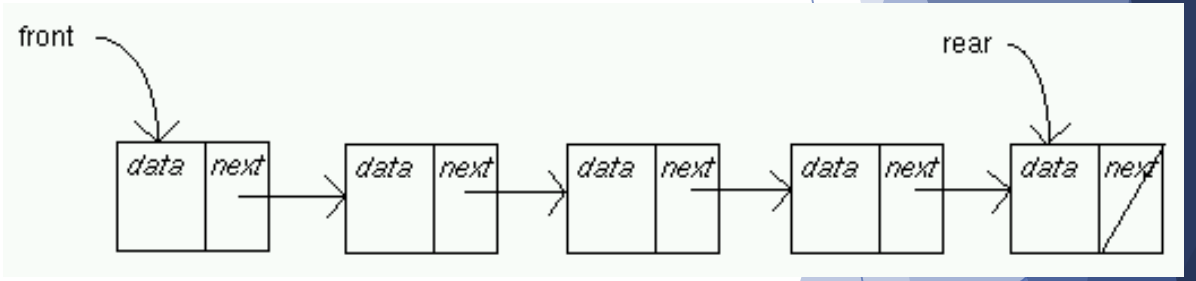


Queue – Indexed implementation

- 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

Lists, stacks, queues in .NET

- 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://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 $DoublyLinkedList < T >$ (with the operations specified in the slide “implementation exercises”)
  - $Queue < T >$
  - $Stack < T >$
- [optional] Complete first exercise (about sorting) of formative project
  - Detailed description is in modulewijzer

See you next week ☺