CSC 225

Algorithms and Data Structures I

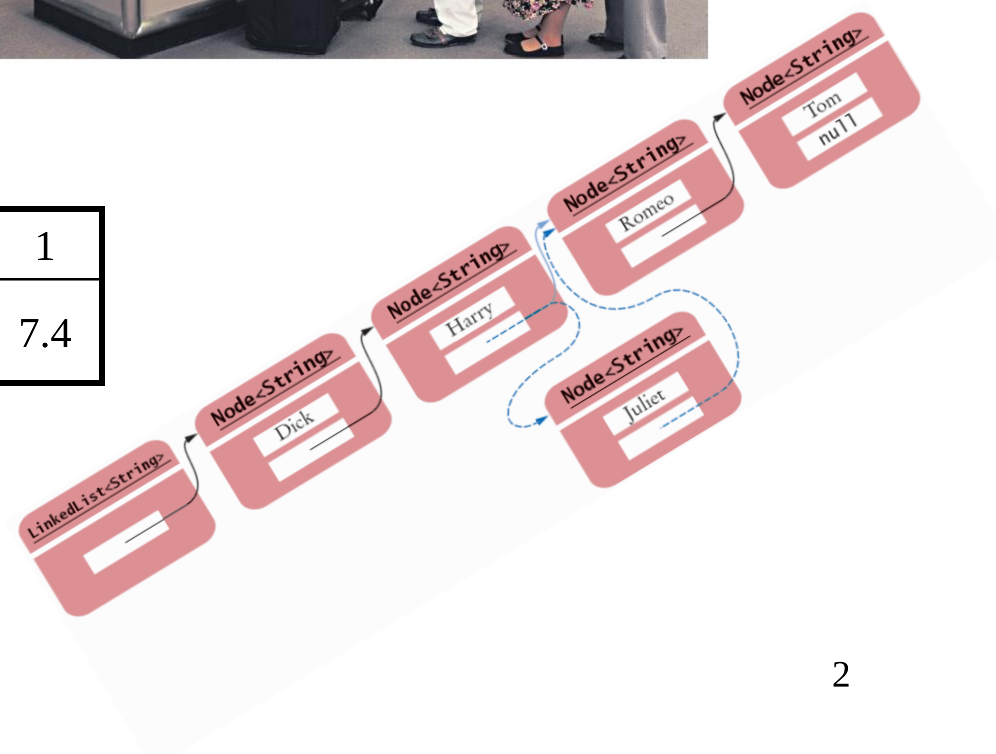
Fall 2014

Rich Little

Basic Data Structures

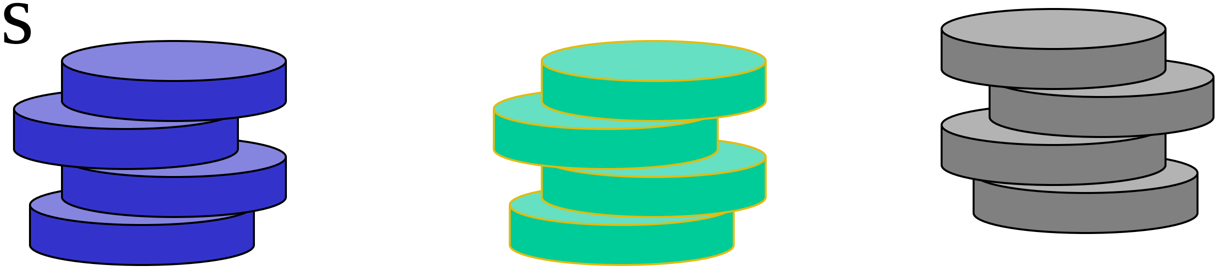
- Stacks
- Queues
- Arrays or vectors
- Lists

X	12	3	7	24	4	1	1
A	12	7.5	7.3	11.5	10	8.5	7.4



Abstract Data Type (ADT)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations



The Notion of a Stack

- Container of items
- Items are returned in reverse order of being added (LIFO)
- **Push** and **pop** items from the top of the stack
  - Stack of plates in cafeteria
  - Candy dispenser
- Examples
  - Solving a problem by completely solving every smaller problem that comes up (e.g., Quicksort, Divide and conquer algorithm)
  - Keeping track of the url's when browsing the web
  - "Undo" function of most applications that have a user interface
  - Runtime environment's handling of nested method calls
  - Recursive and nested method calls

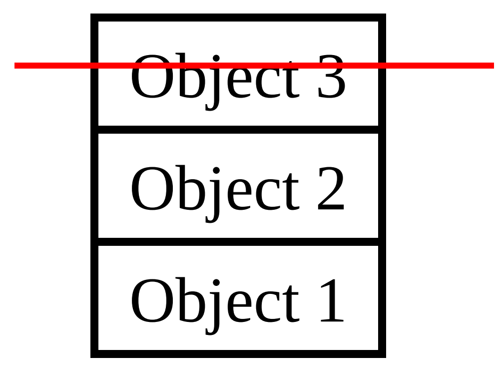
Stacks

- Container of objects that are inserted and removed following the LIFO principle
  LIFO = last-in first-out

Object 3
Object 2
Object 1

Stacks

- Container of objects that are inserted and removed following the LIFO principle
  LIFO = last-in first-out



Stacks

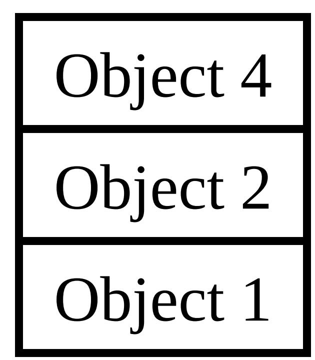
- Container of objects that are inserted and removed following the LIFO principle
  LIFO = last-in first-out

Object 4
Object 2
Object 1

Stacks

- Container of objects that are inserted and removed following the LIFO principle
  LIFO = last-in first-out

Can we remove Object 2 at this moment?



Removing an object from a stack

- Only the most recently inserted object can be removed at *any* time.
- Earlier inserted objects can only be removed if all objects that are inserted at a later time are already removed from the stack.

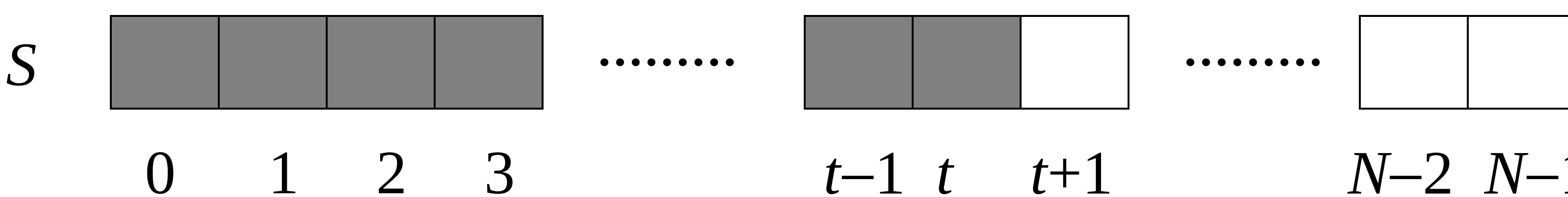
The Stack Abstract Data Type

A stack *S* is an abstract data type (ADT) supporting the following methods.

- **push(*o*):** Insert object *o* at the top of the stack
- **pop():** Remove from the stack and return the top object on the stack (that is, the most recently inserted element still in the stack); an error occurs if the stack is empty.
- **isEmpty():** Return a Boolean indicating if the stack is empty.
- t**op():** Return the top object on the stack without removing it; an error occurs if the stack is empty.
- **size():** Return the number of objects in the stack.

An Efficient Implementation of a Stack: The Simple Array-Based Stack

- *S*: *N*-element array, with elements stored from *S*[0] to *S*[*t*]
- *t*: stack pointer; integer that gives the index of the top element in *S*
- *N*: specified max stack size (e.g., *N*=1500)



Algorithm push(*object*):

```
if size() = N then
   "indicate that the
   stack is full"
   return
end
t ← t + 1
S[t] ← object
return
```

Algorithm pop():

```
if isEmpty() then
    "indicate that the
    stack is empty"
    return
end
object ← S[t]
t ← t − 1
return object
```

What is the Running Time of push()?

```
Algorithm push(object):
  if size() = N then
    "indicate that the
    stack is full"
  return
  end
  t ← t + 1
  S[t] ← object
 return
```

What is the Running Time of pop()?

Algorithm pop():
  if isEmpty() **then**
    "indicate that the
    stack is empty"
  return
  end
  $object \leftarrow S[t]$
  $t \leftarrow t - 1$
  return *object*

Array-Based Implementations of a Stack: Advantages and Disadvantages

- Simple
- Efficient: O(1) per operation
- The stack *must* assume a fixed upper bound *N*
- Memory might be wasted or a stack-full error can occur!
- If good estimate for stack size is known: Array is the best choice!!

Run-time Stack

- The run-time environment for most programming languages uses a uses a stack to keep track of method invocations
- Each method <u>call</u> has an *activation record* or *stack frame* associated with it
- Whenever a call is made, a new activation record is allocated and *pushed* onto the stack
- When a call returns, its record is *popped* from the call stack
- Each activation record (frame) contains
  - Program counter for the current line of code (return code)
  - Space to hold all method parameters
  - Space to hold all method local variables
  - Space to hold the return value

Recursion

- A recursive method calls itself
  - `void a() { … a() … }`

- Indirect recursion
  - `void a() { … b() … }`
  - `void b() { … a() … }`

- Recursive calls of course are also realized with the run-time stack

- "Infinite Recursion" leads to stack overflow (out-of-memory error)

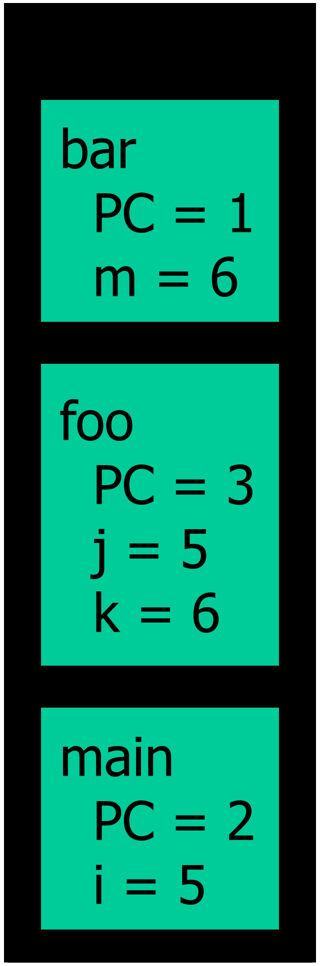
Run-time Stack

When a method terminates, its frame is popped off the stack and control is passed to the method on top of the stack (i.e., the calling method)

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```



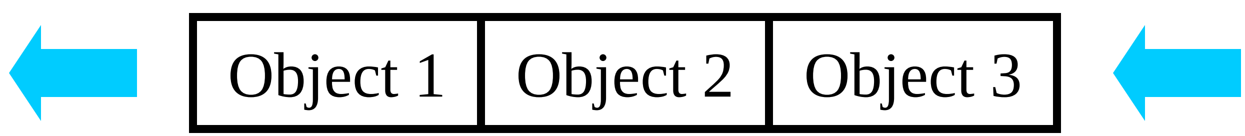
Postfix Notation

- The “normal” way to write arithmetic expressions is *infix notation*
  - because the operators are *between* the operands
- Expressions written in *postfix notation* are easier to evaluate
  - the operators are *after* the operands
  - there is no need for parenthesis
  - there is no need for operator precedence rules

Infix Form	Postfix Form	Value
34	34	34
34 + 22	34 22 +	56
34 + 22 * 2	34 22 2 * +	78
34 * 22 + 2	34 22 * 2 +	750
(34 + 22) * 2	34 22 + 2 *	112

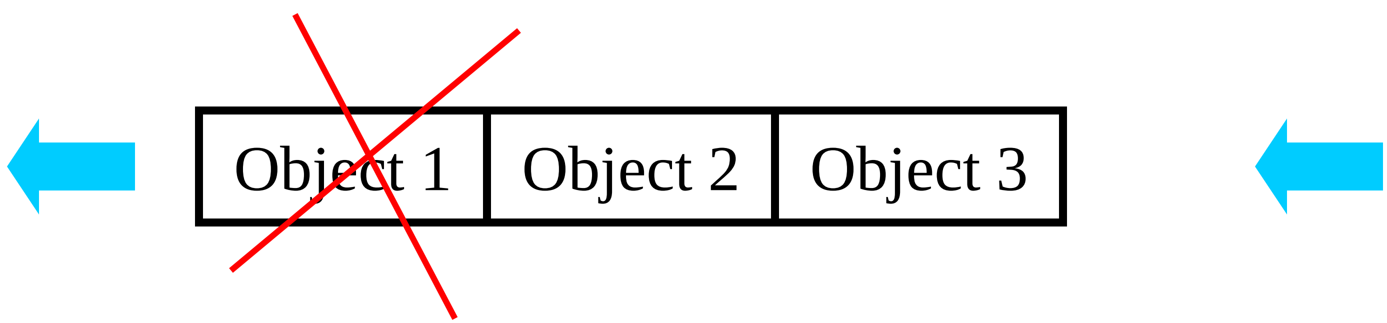
Queues

- Container of items that are inserted and removed following the FIFO principle FIFO = first-in first-out
- Next up is always the item that has been in the queue the longest
- Examples:
  - ➢ people waiting for a carnival ride
  - ➢ multi-user operating system’s time-sharing
  - ➢ customer number systems at the bakery
  - ➢ waitlists for classes
  - ➢ Priority queues



Queues

- Container of objects that are inserted and removed following the FIFO principle
  FIFO = first-in first-out
- Insertion is possible at any time



Queues

- Container of objects that are inserted and removed following the FIFO principle
  FIFO = first-in first-out
- Insertion is possible at any time

Queues

- Container of objects that are inserted and removed following the FIFO principle
  FIFO = first-in first-out
- Insertion is possible at any time

Can we remove Object 2
at this moment?

Queues

- Container of objects that are inserted and removed following the FIFO principle
  FIFO = first-in first-out
- Insertion is possible at any time

Can we remove Object 3 at this moment?

Removing an object from a queue

- Only the element that has been in the queue the longest can be removed at any time
- Later inserted objects can only be removed if all objects that are inserted at an earlier time are already removed from the queue.

The Queue Abstract Data Type

A queue *Q* is an abstract data type (ADT) supporting the following methods:

- **enqueue(*o*):** Insert object *o* at the rear of the queue
- **dequeue():** Remove and return from the queue the object at the front; an error occurs if the queue is empty
- **isEmpty():** Return a Boolean indicating if the queue is empty
- **front():** Return, but not remove, the front object in the queue; an error occurs if the queue is empty
- **size():** Return the number of objects in the queue

An Efficient Implementation of a Queue: The Simple Array-Based Queue

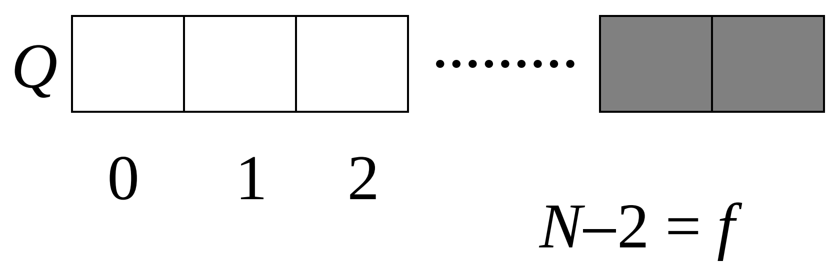
- Q: N-element array
- f: index to the cell of Q storing the first element of Q (init is f=0), unless the queue is empty ($f = r$)
- r: index to the next available array cell in Q (init is r=0)

$f = r$ indicates Q is empty

- N: specified maximum queue size (e.g., N=1500)

The Simple Array-Based Queue: “wrap around”

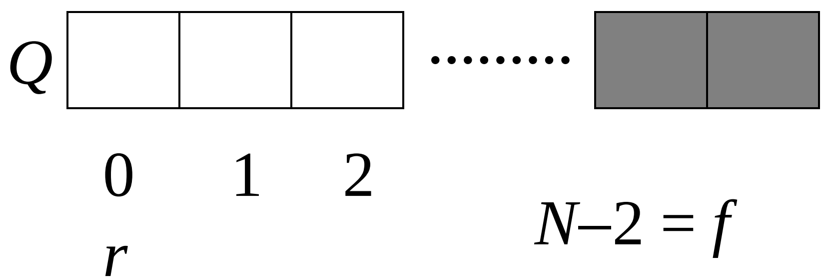
- What if for example $f = N–2$? How many elements can be stored in Q?



r = ?

The Simple Array-Based Queue: “wrap around”

- What if for example $f = N{-}2$? How many elements can be stored in Q?



Count modulo N!

$$\boxed{x \bmod y = x - \lfloor x / y \rfloor y,\; y \neq 0}$$

Another problem

- What happens if we enqueue N objects without any dequeuing?

We obtain $f = r$! (Which implies that the queue is empty)

Algorithm enqueue(*o*):

if size() = $\boldsymbol{N-1}$ **then**

    throw a QueueFullException

$Q[r] \leftarrow o$

$r \leftarrow (r + 1)$ mod N

Algorithm dequeue():

```
if isEmpty() then
    throw a QueueEmptyException
temp ← Q[f]
f ← (f + 1) mod N
return temp
```

What is the running time?

```
Algorithm enqueue(o):
  if size() = N-1 then
     throw a QueueFullException
  Q[r] ← o
  r ← (r + 1) mod N
 return
```

What is the running time?

```
Algorithm dequeue():
    if isEmpty() then
        throw a QueueEmptyException
  temp ← Q[f]
  f ← (f + 1) mod N
  return temp
```

Array-Based Implementations of a Queue: Advantages and Disadvantages

- Simple
- Efficient: O(1) per operation
- The queue has a fixed upper bound N (for N–1 elements in a full queue)
- If a good estimate for the size of the queue is known: an array is the best choice!

Contrasting Stack and Queue

```
public interface Stack {
  void push(Object data);
  Object pop();
  Object top();
  boolean isEmpty();
  int size();
}
```

Stack applications

- Stack of plates in cafeteria
- Run-time stack
- Recursion
- Evaluating expressions
- Balanced parentheses
- Postfix notation
- LIFO (Last-in, first-out)

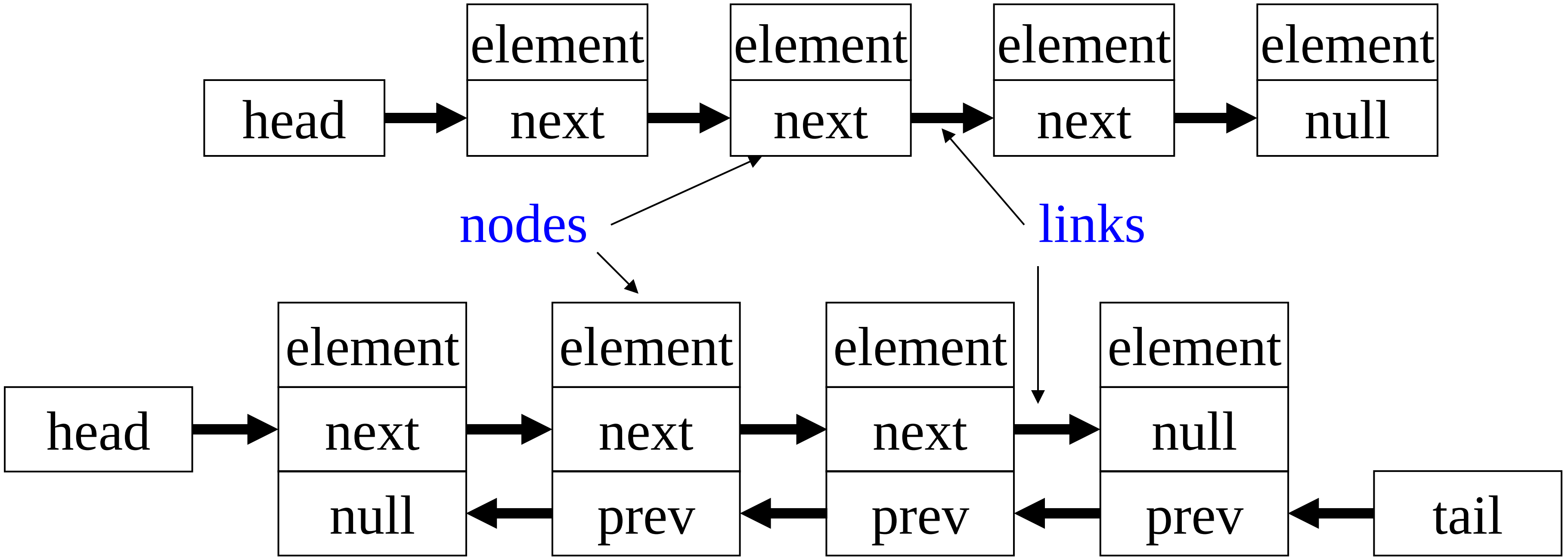
```
public interface Queue {
  void enqueue(Object
   data);
  Object dequeue();
  Object front();
  boolean isEmpty();
  int size();
}
```

Queue applications

- Check out line at store
- Car wash
- Network queues
- Pipes and filters
- Traffic simulation
- FIFO (First-in, First-out)

Singly and Doubly Linked Lists

- A *position* of an element is defined *relatively* (i.e., in terms of its neighbors)



The List ADT

Supported methods for a list *S*

- **first():** Return position of 1^{st} element of *S* (error occurs if *S* empty)
- **last():** Return position of last element of *S* (error occurs if *S* empty)
- **isFirst(*p*):** Return a Boolean value (true for *p* is 1^{st} position, false otherwise)
- **isLast(*p*):** Return a Boolean value (true for *p* is last position, false otherwise)
- **before(*p*):** Return position of the element of *S* preceding the one at position *p* (error occurs if *p* is 1^{st} element)
- **after(*p*):** Return position of the element of *S* following the one at position *p* (error occurs if *p* is last element)

The List ADT …

Supported methods for a list *S*

- **replaceElement(*p*, *e*):** Replace the element at position *p* with *e*, the element that was at position *p* first is returned
- **swapElements(*p*, *q*):** Swap elements stored at positions *p* and *q*
- **insertFirst(*e*):** Insert a new element *e* into *S* as the first element
- **insertLast(*e*):** Insert a new element *e* into *S* as the last element

The List ADT …

Supported methods for a list *S*

- **insertBefore(*p*, *e*):** Insert a new element *e* into *S* before position *p* (error occurs if *p* is 1st element)
- **insertAfter(*p*, *e*):** Insert a new element *e* into *S* after position *p* (error occurs if *p* is last element)
- **remove(*p*):** Remove from *S* the element at position *p*

Algorithm insertAfter(p, e)

Doubly linked list

```
Create a new node v
v.element ← e
v.prev ← p
v.next ← p.next
(p.next).prev ← v
p.next ← v
return v
```

Algorithm remove(p)

Doubly linked list

```
t ← p.element
(p.prev).next ← p.next
(p.next).prev ← p.prev
p.prev ← null
p.next ← null
return t
```

Running Times

- first(): O(1)
- last(): O(1)
- isFirst(*p*): O(1)
- isLast(*p*): O(1)
- before(*p*): O(1)
- after(*p*): O(1)
- replaceElement(*p*, *e*): O(1)
- swapElements(*p*, *q*): O(1)
- insertFirst(*e*): O(1)
- insertLast(*e*): O(1)
- insertBefore(*p*, *e*): O(1)
- insertAfter(*p*, *e*): O(1)
- remove(*p*): O(1)