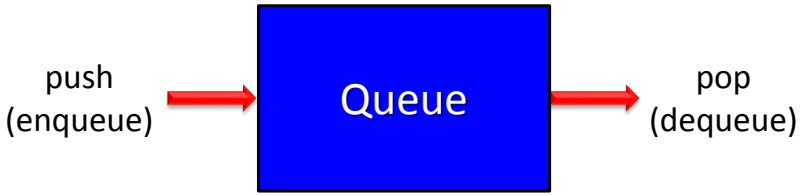
Containers: Queue and List

Jordi Cortadella and Jordi Petit
Department of Computer Science

Queue

- A container in which insertion is done at one end (the tail) and deletion is done at the other end (the head).
- Also called FIFO (First-In, First-Out)



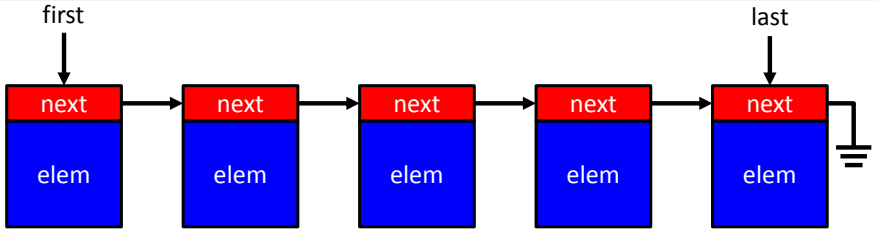
Queue usage

```
Queue<int> Q;  // Constructor

Q.push(5);     // Inserting few elements
Q.push(8);
Q.push(6);

int n = Q.size();  // n = 3

while (not Q.empty()) {
  int elem = Q.front();   // Get the first element
  cout << elem << endl;
  Q.pop();                // Delete the element
}
```


The class Queue

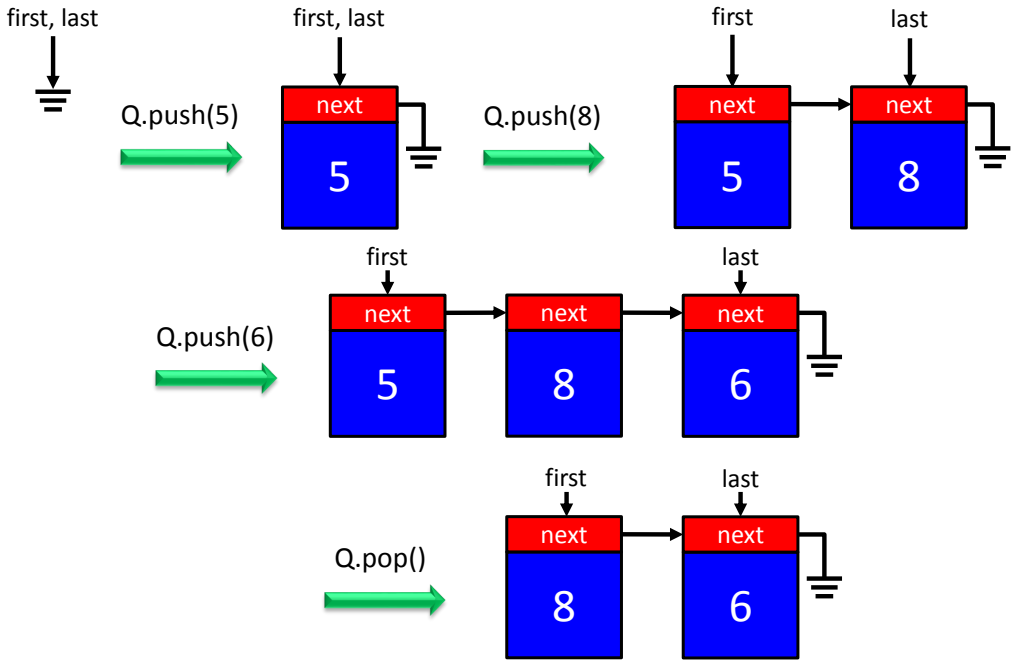
```
template<typename T>
class Queue {
public:

  Queue();    // Constructor

  ~Queue();   // Destructor

  Queue(const T& Q);        // Copy constructor

  Queue& operator= (const Queue& Q);  // Assignment operator

  void push(const& T x);    // Enqueues an element

  void pop();        // Dequeues the first element

  T front() const;   // Returns the first element

  int size() const;  // Number of elements in the queue

  bool empty() const;       // Is the queue empty?
};
```

Implementation with linked lists




```
template<typename T>
class Queue {

private:
  struct Node {
    T elem;
    Node* next;
  };

  Node *first;      // Pointer to the first element
  Node *last;       // Pointer to the last element
  int n;            // Number of elements
```


Implementation with linked lists



Queue: some methods

```
/** Returns the number of elements. */
int size() const {
  return n;
}

/** Checks whether the queue is
    empty. */
bool empty() const {
  return size() == 0;
}

/** Inserts a new element at the end
    of the queue. */
void push(const T& x) {
  Node* p = new Node {x, nullptr};
  if (n++ == 0) first = last = p;
  else last = last->next = p;
}
```

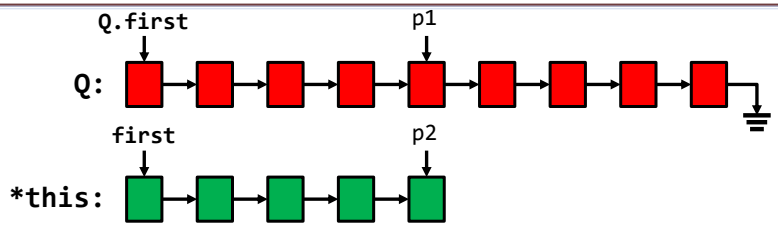
```
/** Removes the first element.
    Pre: the queue is not empty. */
void pop() {
  assert(not empty());
  Node* old = first;
  first = first->next;
  delete old;
  if (--n == 0) last = nullptr;
}

/** Returns the first element.
    Pre: the queue is not empty. */
T front() const {
  assert(not empty());
  return first->elem;
}
```


Queue: constructors and destructor

```
/** Default constructor: an empty queue. */
Queue() : first(nullptr), last(nullptr), n(0) { }

/** Copy constructor. */
Queue(const Queue& Q) {
  copy(Q);
}

/** Assignment operator. */
Queue& operator= (const Queue& Q) {
  if (&Q != this) {
    free();
    copy(Q);
  }
  return *this;
}

/** Destructor. */
~Queue() {
  free();
}
```

```
private:
  /** Frees the linked list of
  nodes in the queue. */
  void free() {
    Node* p = first;
    while (p) {
      Node* old = p;
      p = p->next;
      delete old;
    }
  }
```

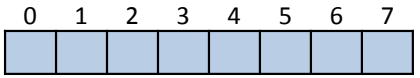
Queue: copy (private)

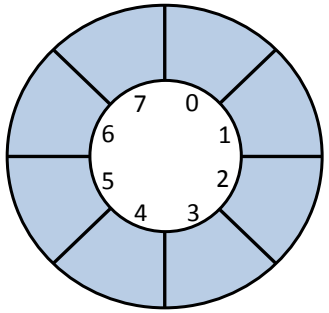



```
/** Copies a queue. */
void copy(const Queue& Q) {
  n = Q.n;
  if (n == 0) {
    first = last = nullptr;
  } else {
    Node* p1 = Q.first;
    Node* p2 = first = new Node {p1->elem};
    while (p1->next) {
      p1 = p1->next;
      p2 = p2->next = new Node {p1->elem};
    }
    p2->next = nullptr;
    last = p2;
  }
}
```

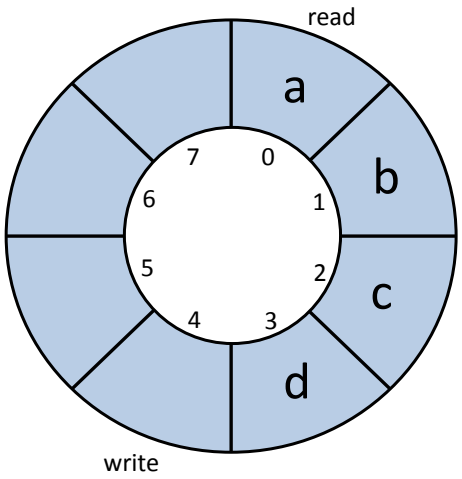

Implementation with circular buffer

- A queue can also be implemented with an array (vector) of elements.
- It is a more efficient representation if the maximum number of elements in the queue is known in advance.

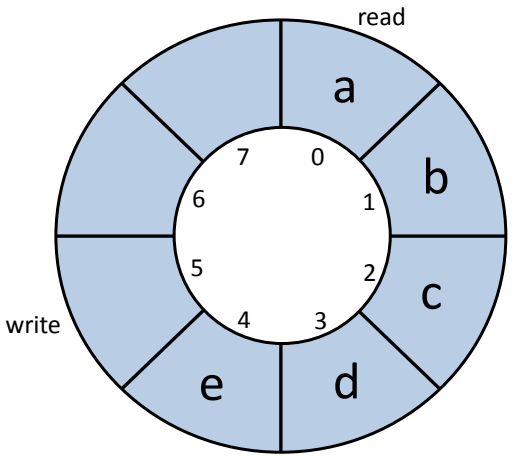




Implementation with circular buffer

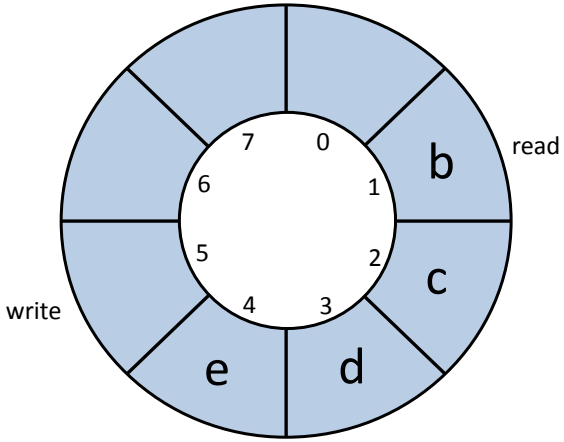


Implementation with circular buffer



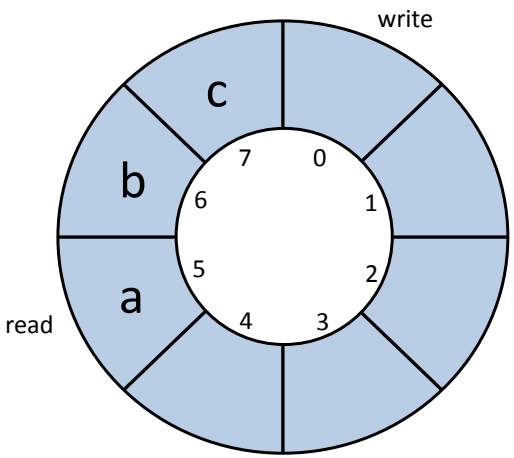
after **Q.push(e)**

Implementation with circular buffer

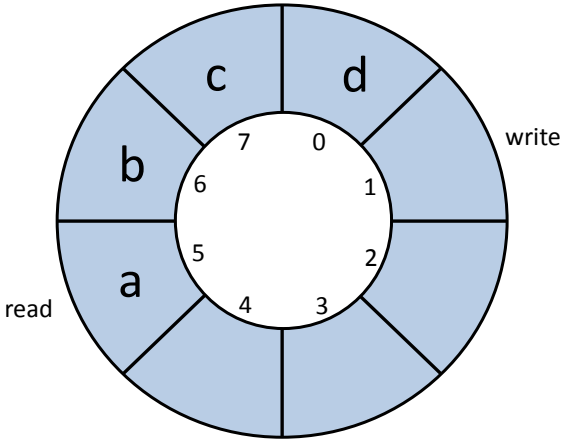


after **Q.pop()**

Implementation with circular buffer



Implementation with circular buffer



after **Q.push(d)**

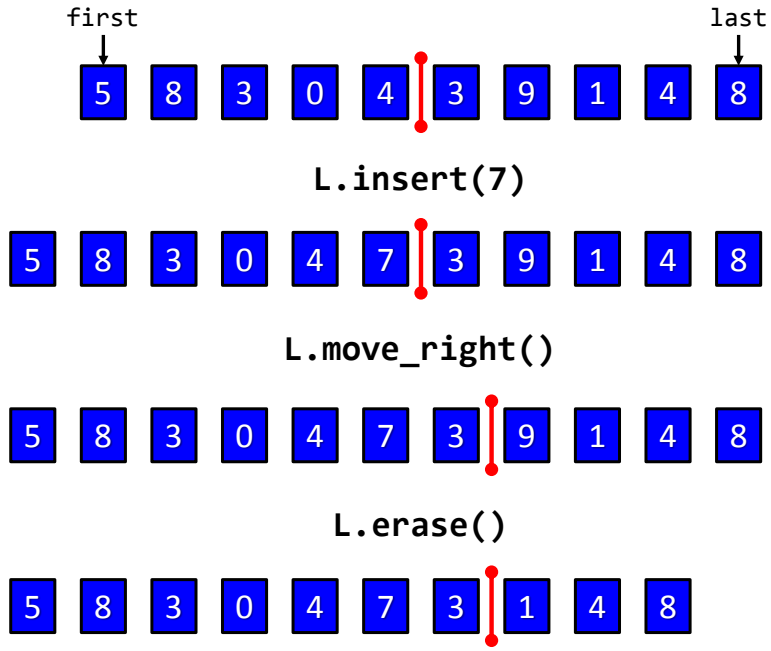
Queue: Complexity

- All operations in queues can run in constant time, except for:
  - Copy: linear in the size of the list.
  - Delete: linear in the size of the list.
- Queues do not allow to access/insert/delete elements in the middle of the queue.

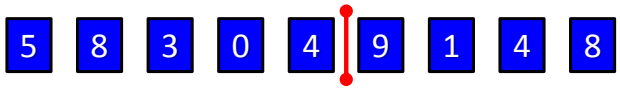
List

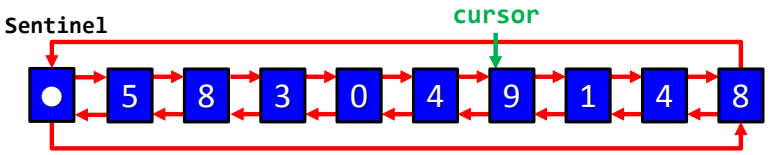
- List: a container with sequential access.
- It allows to insert/erase elements in the middle of the list in constant time.
- A list can be considered as a sequence of elements with one or several cursors (iterators) pointing at internal elements.
- For simplicity, we will only consider lists with one iterator.
- Check the STL list: it can be visited by any number of iterators.

List: graphical representation



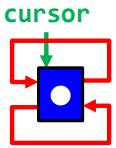
List implementation: doubly linked nodes





cursor: pointer at the first node after the cursor

Empty list:



The class List: private representation

```
template <typename T>
class List {

    /** Doubly linked node of the list. */
    struct Node {
        Node* prev;  /** Pointer to the previous node. */
        T elem;      /** The element of the list. */
        Node* next;  /** Pointer to the next element. */
    };

    Node* sentinel; /** Sentinel of the list. */
    Node* cursor;   /** Node after the cursor. */
    int n;          /** Number of elements (without sentinel). */
```

The class List: public methods

```
public:
  /** Constructor of an empty list. */
  List() : sentinel(new Node), cursor(sentinel), n(0) {
    sentinel->next = sentinel->prev = sentinel;
  }

  /** Destructor. */
  ~List() {
    free();
  }

  /** Copy constructor. */
  List(const List& L) {
    copy(L);
  }

  /** Assignment operator. */
  List& operator= (const List& L) {
    if (&L != this) {
      free();
      copy(L);
    }
    return *this;
  }
```

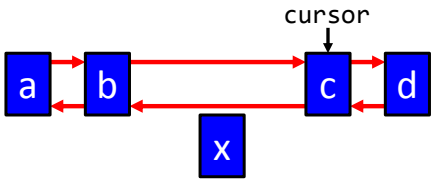
```
/** Returns the number of
    elements in the list. */
int size() const {
  return n;
}

/** Checks whether the list
    is empty. */
bool empty() const {
  return size() == 0;
}
```

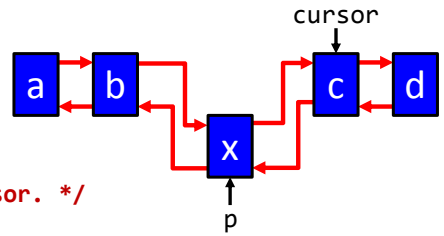

The class List: public methods

```
public:
  /** Checks whether the cursor is at the beginning of the list. */
  bool is_at_front() const {
    return cursor == sentinel->next;
  }

  /** Checks whether the cursor is at the end of the list. */
  bool is_at_end() const {
    return cursor == sentinel;
  }

  /** Moves the cursor one position backward.
      Pre: the cursor is not at the beginning of the list. */
  void move_backward() {
    assert(not is_at_front());
    cursor = cursor->prev;
  }

  /** Moves the cursor one position forward.
      Pre: the cursor is not at the end of the list. */
  void move_forward() {
    assert(not is_at_end());
    cursor = cursor->next;
  }
```


The class List: public methods

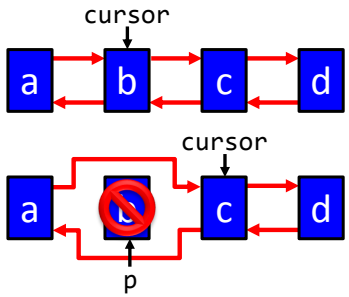
```
public:

  /** Moves the cursor to the
      beginning of the list. */
  void move_to_front() {
    cursor = sentinel->next;
  }

  /** Moves the cursor to the
      end of the list. */
  void move_to_end() {
    cursor = sentinel;
  }

  /** Inserts an element x before the cursor. */
  void insert(const T& x) {
    Node* p = new Node {cursor->prev, x, cursor};
    cursor->prev = cursor->prev->next = p;
    ++n;
  }
```





The class List: public methods

```
public:

  /** Erases the element after the cursor.
      Pre: cursor is not at the end. */
  void erase() {
    assert(not is_at_end());
    Node* p = cursor;
    p->next->prev = p->prev;
    cursor = p->prev->next = p->next;
    delete p;
    --n;
  }

  /** Returns the element after the cursor.
      Pre: the cursor is not at the end. */
  T front() const {
    assert(not is_at_end());
    return cursor->elem;
  }
```



Exercises: implement the private methods **copy()** and **free()**.

Higher-order functions

- A higher-order function is a function that can receive other functions as parameters or return a function as a result.
- Most languages support higher-order functions (C++, python, R, Haskell, Java, JavaScript, …).
- The have different applications:
  - **sort** in STL is a higher-order function (the compare function is a parameter).
  - functions to visit the elements of containers (lists, trees, etc.) can be passed as parameters.
  - Mathematics: functions for composition and integration receive a function as parameter.
  - etc…

Higher-order functions: example

```
template <typename T>
class List {
…
  /** Transforms every element of the list using f.
      It returns a reference to the list. */
  List<T>& transform(void f(T&));

  /** Returns a list with the elements for which f is true */
  List<T> filter(bool f(const T&)) const;

  /** Applies f sequentially to the list and returns a
      single value. For the list [x_1, x_2, x_3, …, x_n] it returns
                  f(…f(f(init, x_1), x_2) …, x_n).
      If the list is empty, it returns init.*/
  T reduce(T f(const T&, const T&), T init) const;
}
```


Higher-order functions: example

```
/** Checks whether a number is prime */
bool isPrime(int n) {…}

/** Adds two numbers */
int add(int x, int y) {
  return x + y;
}

/** Substitutes a number by its square */
void square(int& x) {
  x = x*x;
}
```

```
/** The following code computes:
                    ∑_{x∈L, x is prime} x^2
*/

int n = L.filter(isPrime).transform(square).reduce(add, 0);
```


Higher-order functions: example

```
List<T>& transform(void f(T&)) {
  Node* p = sentinel->next;
  while (p != sentinel) { // Visit all elements and apply f to each one
    f(p->elem);
    p = p->next;
  }
  return *this;
}

List<T> filter(bool f(const T&)) const {
  List<T> L;
  Node* p = sentinel->next;
  while (p != sentinel) { // Pick elements only if f is asserted
    if (f(p->elem)) L.insert(p->elem);
    p = p->next;
  }
  return L;
}

T reduce(T f(const T&, const T&), T init) const {
  T x = init;                 // Initial value
  Node* p = sentinel->next; // First element (if any)
  while (p != sentinel) {
    x = f(x, p->elem);        // Composition with next element
    p = p->next;
  }
  return x;
}
```

EXERCISES

Queues implemented as circular buffers

- Design the class queue implemented with a circular buffer (using a vector):
  - The push/pop/front operations should run in constant time.
  - The copy and delete operations should run in linear time.
  - The class should have a constructor with a parameter n that should indicate the maximum number of elements in the queue.
- Consider the design of a variable-size queue using a circular buffer. Discuss how the implementation should be modified.

Reverse and Josephus

- Design the method `reverse()` that reverses the contents of the list:
  - No auxiliary lists should be used.
  - No copies of the elements should be performed.
- Solve the Josephus problem, for n people and executing every k-th person, using a circular list:

  https://en.wikipedia.org/wiki/Josephus_problem

Merge sort

- Design the method `merge(const List& L)` that merges the list with another list L, assuming that both lists are sorted. Assume that a pair of elements can be compared with the operator <.
- Design the method `sort()` that sorts the list according to the < operator. Consider merge sort and quick sort as possible algorithms.