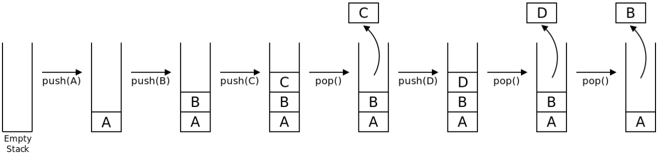
Linked Lists, Deques, Stacks, Queues
and Applications

L.EIC

Algoritmos e Estruturas de Dados

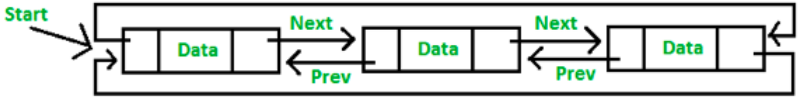
2024/2025



P Ribeiro, **AP Tomás**

Circular Doubly Linked-List

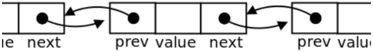
- A **circular doubly linked-list** is a linked list where each node is connected to both its previous and next nodes, and **the last node links back to the first node**.



- The implementation we will described is based on the one given for Lab Class 05 for `template <class T> class SinglyLinkedList` For the circular doubly linked-list:
  - Each node has three attributes: `next`, `prev` and `value`.
  - The head node is given by a pointer called `first`.

Circular Doubly Linked-List

Node




```
template <class T> class Node {
private:
  T value;    // value in the node
  Node<T> *next;  // pointer to next node
  Node<T> *prev;  // pointer to previous node
public:
  Node(const T & v, Node<T> *n=nullptr, Node<T> *p=nullptr):
                    value(v), next(n), prev(p) {}
  T & getValue() { return value; }
  Node<T> *getNext() { return next; }
  Node<T> *getPrev() { return prev; }
  void setValue(T & v) { value=v; }
  void setNext(Node<T> *n) { next = n; }
  void setPrev(Node<T> *p) { prev = p; }
};
```

Circular Doubly Linked-List

Implementation

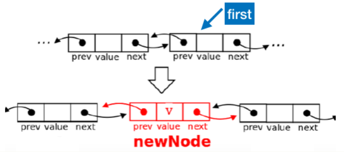
```
template <class T> class CircularDoublyLinkedList {
private:
  Node<T> *first;     // head (front) of the list
  int length;         // number of nodes
public:
  // Construtor (creates empty list)
  CircularDoublyLinkedList(): first(nullptr), length(0) {}

 // Destrutor
  ~CircularDoublyLinkedList() {
    while(!isEmpty()) {
      assert(first != nullptr && "message...");
      removeFirst();
    }
    assert(first == nullptr && "message...");
  }                                            // ...continues
};
```

`"message..."` stands for a message to be written if the condition fails

Circular Doubly Linked-List

AddFirst

```
template <class T> class CircularDoublyLinkedList {
  ...
  void addFirst(const T & v) {    // adds v to the front
    Node<T> *newNode;    // to be a pointer to the new node
    if (first == nullptr) {
      newNode = new Node<T>(v,nullptr,nullptr);
      newNode -> setNext(newNode);
      newNode -> setPrev(newNode);   // a loop to itsef
    } else {
      newNode = new Node<T>(v,first, first -> getPrev());
      (first -> getPrev()) -> setNext(newNode);
      first -> setPrev(newNode);
    }
    first = newNode;
    length++;
  }
```



Creates a new node `newNode` with value `v`. It will be the new head (`first`). Links it to the previous first and last nodes. When the list is empty, links `newNode` to itself.

Circular Doubly Linked-List

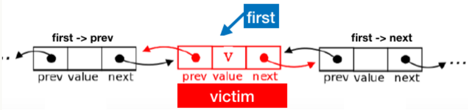
AddFirst, getFirst, size, isEmpty

```
template <class T> class CircularDoublyLinkedList {
  ...
 // Adds v to the end of the list
  void addLast(const T & v) {
    addFirst(v);
    if (length > 1) first = first -> getNext();
  }

  // Returns the reference to the first value
  T & getFirst() {
    assert(!isEmpty() && "empty list has no first node");
    return first->getValue();
  }

  // Returns the length of the list
  int size() {  return length;  }

  // Returns true iff the list is empty
  bool isEmpty() {  return (length == 0);  }
```

Circular Doubly Linked-List

`removeFirst`

```
template <class T> class CircularDoublyLinkedList {
  ... // Pops the first node (does nothing if list is empty)
  void removeFirst() {
    if (isEmpty()) return;
    Node<T> *victim = first;
    if (length > 1) {
      first = first -> getNext();
      first -> setPrev(victim -> getPrev());
      victim -> getPrev() -> setNext(first);
    } else first = nullptr;
     delete victim;
    length--;
  }
```



The victim is the first node. If the list has other nodes, we link `victim->next` and `victim -> prev` and change first accordingly. Otherwise, we set `first` to `nullptr` since the list will be empty in the end. The victim is deleted ("freeing" the memory).

Circular Doubly Linked-List

removeChain

```
template <class T> class CircularDoublyLinkedList {
  ...// deletes the chain defined by start and end (deletes both also)
 void removeChain(Node<T> *start, Node<T> *end) {
    Node<T> *nextEnd = end -> getNext();
    Node<T> *prevStart = start -> getPrev();
    Node<T> *victim;
    do {
      victim = start;
      start = start -> getNext();
      delete victim;
      length--;
    } while (victim != end);
    if (length == 0) first = nullptr;  // final list is empty
    else { // links initial nodes start -> prev and end -> next
      nextEnd -> setPrev(prevStart);
      prevStart -> setNext(nextEnd);
      first = nextEnd;  // first becomes the initial end -> next
    }
  }
```

Circular Doubly Linked-List

toString

```
// Convert a list to a string
  std::string toString() {
    if (isEmpty()) return "{}";
    Node<T> *curr = first;
    std::stringstream sstr;
    sstr <<  "{" << curr->getValue();
    while ( (curr = curr -> getNext()) != first) {
      sstr << "," << curr -> getValue();
    }
    sstr <<  "}";
    return sstr.str();
  }
```

Requires `<sstream>` to be included. The values will be separated by a comma "," in the string. There is no comma in the end.

Circular Doubly Linked-List

getStart, getLast, setFirst

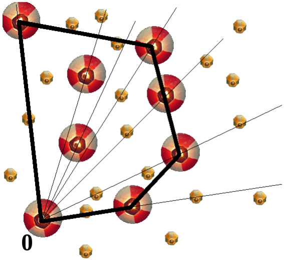
```
// Returns start node
Node<T> *getStart(){
  assert(!isEmpty() && "empty list has no start node");
  return first;
}

// Returns last node (node before the first)
Node<T> *getLast(){
  assert(!isEmpty() && "empty list no last node");
  return first -> getPrev();
}

// Changes the head to be newfirst
void setFirst(Node<T> *newfirst) {
  first = newfirst;   //does not check if newfirst exists
}
```

Convex Hull in 2D

Application of Circular Doubly Linked-Lists

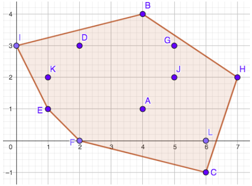
The convex hull problem

Given a set P of n points in the plane, defined by their cartesian coordinates, compute the list of vertices of its convex hull $\mathcal{CH}(P)$, in counterclockwise order.



Recall "St. Saint John Festival"

The **convex hull** of $P \subseteq \mathbb{R}^2$ is the **smallest convex polygon** that contains all points of P.



We are going to see two algorithms for computing the convex hull in $\mathcal{O}(\mathbf{n} \log \mathbf{n})$ time: the **incremental algorithm**, by Edelsbrunner, and **Graham's scan**.

Applications of Convex Hulls

- There are many problems that are **comparatively easy to solve for convex sets** and much harder in general. E.g., POINT IN CONVEX POLYGON can be solved in $\mathcal{O}(\log n)$.

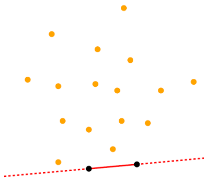
- Convex hulls in 2D and 3D have many applications. They can be used for instance in:
  - computer visualization
  - ray tracing
  - path finding
  - computing accessibility maps
  - visual pattern matching
  - diameter computation
  - cluster analysis
  - mesh generation
  - ...

Convex hull in 2D

Some geometrical properties

"Good solutions to algorithmic problems of a geometric nature are mostly based on two ingredients. One is a thorough understanding of the geometric properties of the problem, the other is a proper application of algorithmic techniques and data structures." (in M. Berg, et al (2008), Computational Geometry)

Given $P = \{p_1, p_2, \ldots, p_n\}$, the **extreme points** of convex hull $\mathcal{CH}(P)$ are points of P. A segment $p_i p_j$ is an **extreme segment** iff all the points p_k, with $k \neq i, j$, lie in the same halfplane defined by the line $p_i p_j$.



- If we walk around the boundary of a simple polygon in counterclockwise order (CCW), **its interior is always to the left**.

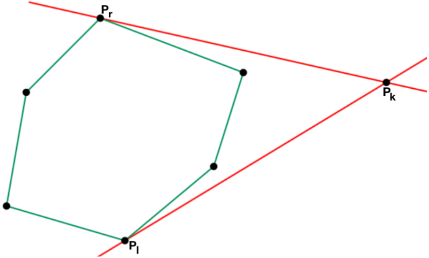
  In a **convex polygon**, no interior angle is greater than 180 degrees: if $v_0, v_1 \ldots, v_{n-1}$ represents the sequence of its vertices in **CCW order**, then $(v_i, v_{(i+1)\%n}, v_{(i+2)\%n})$ defines a **left-turn**, at $v_{(i+1)\%n}$, for all i.
- Orientation tests (left-turn, right-turn, collinear) using the **cross product**:

$$\vec{pq} \times \vec{pr} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ x_q - x_p & y_q - y_p & 0 \\ x_r - x_p & y_r - y_p & 0 \end{vmatrix}$$

For the usual coordinate system, (p, q, r) is a left-turn iff the z-component $(x_q - x_p)(y_r - y_p) - (x_r - x_p)(y_q - y_p)$ of the cross product $\vec{pq} \times \vec{pr}$ is positive (recall the *"right-hand rule"*). It is a right-turn if that component is negative, and the three points are collinear when it is zero.

Incremental Algorithm for Convex hull in 2D

```
Sort the points by increasing x-coord (tie-break: largest y first)
l = InitialPolygon(p1,p2,p3)   // l must be sorted in CCW
For k = 4 to n do:
  - Find the points pl and pr that define the supporting lines
    from pk to the polygon
  - Replace the chain pl,...,pr in l with the chain pl,pk,pr
Return l
```



In the implementation, we refer to `pr` as `upper` and `pl` as `lower`.

Incremental Algorithm for Convex hull in 2D

```
Polygon *convexhull(vector<PointI> &v) {
  sort(v.begin(),v.end(),compare); // sorted by abscissa
  Polygon *pol = initialize(v);   // sorting v[0],v[1],v[2] CCW
  Vertex *lastv, *lower, *upper;
  for(int i=3; i < v.size(); i++) {
    lastv = pol -> getStart();  // previous vertex added
    upper = upper_tangent(lastv,v[i]);
    lower = lower_tangent(lastv,v[i]);
    if (lower -> getNext() != upper)
      pol -> removeChain(lower->getNext(),upper->getPrev());
    pol -> setFirst(upper);
    pol -> addFirst(v[i]);
  }
  return pol; // printPolygon(*pol);
}
```

with `Polygon`, `Vertex`, and `PointI` defined as:

```
typedef CircularDoublyLinkedList<Point<int> >  Polygon;
typedef Node<Point<int> > Vertex;
typedef Point<int> PointI;
```

Incremental Algorithm for Convex hull in 2D

For simplification, we will assume that there are **no three collinear points** in P (a kind of *general position assumption*). The algorithm can be adjusted to deal with degenerated (i.e., special) cases too, which occur in practice!

For "St. Saint John Festival", we cannot assume general position.

```
bool compare(PointI &p, PointI &q) { // for sorting
  if (p.getX() < q.getX()) return true;
  if (p.getX() > q.getX()) return false;
  return p.getY() > q.getY();
}
```

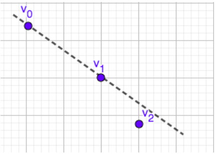
```
bool left_turn(PointI &p0,PointI &p1,PointI &p2) {
  int x0 = p0.getX(), x1 = p1.getX(), x2 = p2.getX();
  int y0 = p0.getY(), y1 = p1.getY(), y2 = p2.getY();
  return (x1-x0)*(y2-y0)-(x2-x0)*(y1-y0) > 0;
  // for a cartesian system with right-handed orientation
}
```

Point

```
template <class P> class Point {
private:
  P x;
  P y;
public:
  //constructors and destructor
  Point(){};
  Point(P a, P b): x(a), y(b){}
  ~Point(){};
  // getters and setters
  P getX(){ return x; };
  void setX(P v) { x = v;}
  P getY(){ return y; }
  void setY(P v) { y = v;}
  string toString(){    // to convert to a string format
    stringstream sstr;
    sstr <<  "(" << x << "," << y << ")";
    return sstr.str();
  };
};
```

Incremental Algorithm for Convex hull in 2D

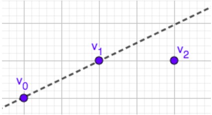
```
Polygon *initialize(vector<PointI> &v) {  // to ensure CCW
  Polygon *pol = new Polygon();
  pol -> addFirst(v[0]);
  pol -> addFirst(v[1]);
  if (left_turn(v[0],v[1],v[2]))
    pol -> setFirst(pol -> getLast());
  pol -> addFirst(v[2]);
  return pol;
}
```



Right-turn

keep `first` at v_1

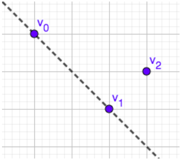
$v_0, \boxed{v_2}, v_1$



Right-turn

keep `first` at v_1

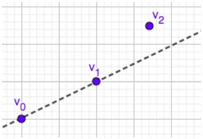
$v_0, \boxed{v_2}, v_1$



Left-turn

move `first` to v_0

$v_1, \boxed{v_2}, v_0$



Left-turn

move `first` to v_0

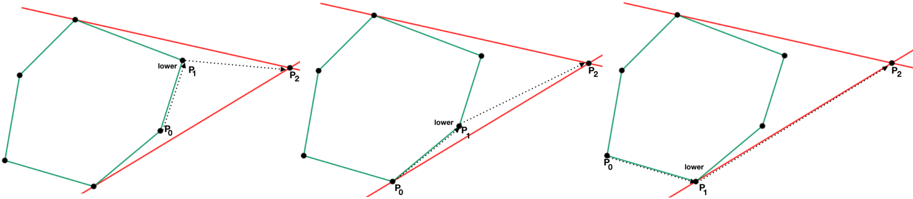
$v_1, \boxed{v_2}, v_0$

In all cases, v_2 will be the first element after insertion.

Incremental Algorithm for Convex hull in 2D

```
Vertex *lower_tangent(Vertex *lower,PointI & p2){
  PointI p1 = lower -> getValue();
  PointI p0 = lower -> getPrev() -> getValue();
  while (!left_turn(p0,p1,p2)) {
    lower = lower -> getPrev();
    p1 = lower -> getValue();
    p0 = lower -> getPrev() -> getValue();
  }
  return lower;
}
```

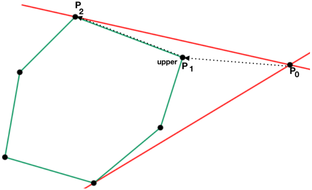
At start, `lower` points to the rightmost node in the convex hull computed so far (always the last point added before the new `p2`).

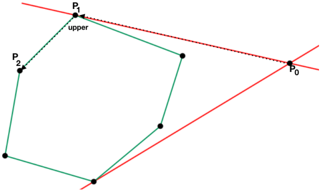


Incremental Algorithm for Convex hull in 2D

```
Vertex *upper_tangent(Vertex *upper,PointI & p0){
  PointI p1 = upper -> getValue();
  PointI p2 = upper -> getNext() -> getValue();
  while (!left_turn(p0,p1,p2)) {
    upper = upper -> getNext();
    p1 = upper -> getValue();
    p2 = upper -> getNext() -> getValue();
  }
  return upper;
}
```

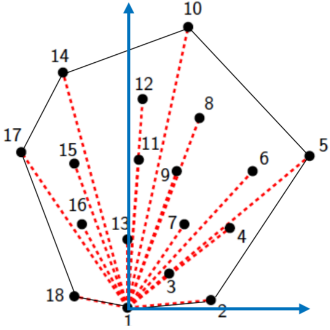
At start, `upper` points to the rightmost node in the convex hull computed so far (always the last point added before the new `p0`).





Graham's Algorithm for Convex Hull in 2D

Graham scan uses a technique called *rotational sweep*, processing points in the order of the polar angles they form with a reference vertex.



The property Graham scan explores

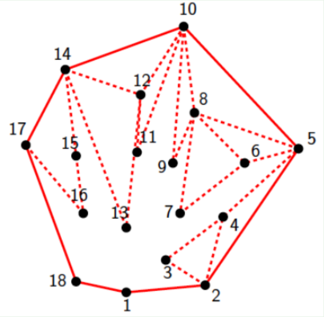
In a counterclockwise walk around the boundary of a **convex polygon** P, starting from the bottom-most vertex p_1, the vertices of P are sorted in increasing order of polar angle w.r.t. p_1.

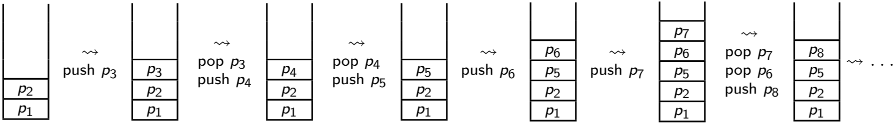
The **polar angle** of a point q with respect to a point p_1 is the anti-clockwise angle between a horizontal line and the line through p_1 and q.

- When there are two or more points with minimum y-component, the bottom-most vertex p_1 can be the leftmost among them (or the rightmost).
- For efficiency and numerical robustness, we do **not compute polar angles**. Instead, we use **orientation tests**: $p_j < p_k$ if (p_1, p_j, p_k) is a left-turn.

Graham's Algorithm for Convex Hull in 2D

```
Find the bottom-most point p1 in P, push it into a stack S
Let p2,p3,...,pn be the other points angularly sorted w.r.t. p1
Push p2 into S
k = 3
While k <= n do:
  p = top(S)
  p- = previous(top(S))
  If (p-,p,pk) is a left turn:
     Push pk into S
     k = k+1
  else:
     Pop p from S
Return S
```





For an animation, check https://dccg.upc.edu/wp-content/uploads/2020/06/GeoC-Convex-hulls-in-2D.pdf

The Stack ADT

A **stack** is a collection whose elements are added to and removed from one end, called the **top of the stack**. It is a **"last in first out"** (**LIFO**) data structure.

- **Basic stack operations** besides the **creation/destruction**:
  - `push(item)`: Add an element to the top
  - `pop()`: Remove the top element
  - `top()`: Examine the top element
  - `size()`: Return the number of elements
  - `empty()`: Test whether the stack is empt

- Usually, no support for **random access** or **iterators** or **sorting**.

  If we need that, we'd better check whether a stack is the best data structure for our application, before we start implementing a specialized stack version.

Implementation of MyStack

Using a linked-list as container

```
#include <list>
using namespace std;

template <typename T> class MyStack {
private:
    list<T> l;
public:
  MyStack(){};     // creates empty list
  ~MyStack(){};
  void push(T d) { l.push_front(d); }
  void pop() { l.pop_front(); }
  bool empty() { return l.size() == 0; }
  T top() { return l.front(); }
  int size() { return l.size();}
  void print() {    // not a basic stack operation
    for (auto x: l)  cout << x << endl;
  }
};
```

Implementation of MyStack

Using an array as container (this code was not presented in the lecture)

```
#include <vector>
#include <cassert>
using namespace std;

template <typename T> class MyStack {
private:
    vector<T> v;
    int nmax, n;
public:
  MyStack(int c=100): v(vector<T>(c)), nmax(c), n(0) {};
  void push(T d) { assert(n < nmax && "full"); v[n++] = d; }
  void pop() { assert(n > 0 && "empty");  n--; }
  bool empty() { return n == 0; }
  T top() { assert(n > 0 && "empty");  return v[n-1]; }
  int size() { return n; }
  // the following are not basic operations in Stack ADT
  void print() { for(int i=0;i<n;i++)cout << v[i] << endl; }
  T atPos(int k) { assert(k>=0 && k<n); return v[n-1-k]; }
};
```

STL `stack` template

- **C++ STL std::stack** is implemented as a **container adaptor**, with **std::deque** as default container, if no container class is specified.

  ```
  template <class T, class Container = deque<T> > class stack;
  ```

  Examples:
  - `std::stack<int> s0;`
  - `std::stack<int,std::list<int> > s1;`

- The class template acts as a wrapper to the underlying container – **only a specific set of functions is provided** (Documentation for std::stack).

  Provides no support for **random access** or **iterators**.

- In Graham scan, for instance, we need `previous(top(S))`.
  - With **C++ STL** `stack`, we could remove the top to get access to `previous(top(S))` and then push the top again (if useful).
  - (With our **array-based MyStack**, `s.atPos(1)` yields `previous(top(S))`; `s.atPos(0)` is `s.top()`; `previous(previous(top(S)))` is `s.top(2)`, ...)

Back to Convex hull

Complexity of Graham scan and the incremental algorithm

Graham's scan and Edelsbrunner's incremental algorithm find the convex hull of $\mathbf{n}$ points in 2D in $\mathcal{O}(\mathbf{n}\log\mathbf{n})$ **time** and $\mathcal{O}(\mathbf{n})$ extra **space**. The **sorting step dominates their time complexity**. For the remaining steps, the time is $\mathbf{\Theta(n)}$.

- **Graham scan:**
  - Each iteration of the `while` loop costs $\mathbf{\Theta(1)}$ because each `pop`, `push`, `top`, and `left_turn` operation takes $\mathbf{\Theta(1)}$.
  - There are less than $2n$ iterations, because either p_k is push into the stack or the top, i.e., a previous point, is removed from the stack.
  - Thus, although the analysis of some particular point p_k may require $\mathcal{O}(size(S)-1)$ iterations, the **total time for the loop is $\mathbf{\Theta(n)}$** and not $\Omega(n^2)$. We have $\mathcal{O}(1)$ **amortized time complexity per point**.
- **Incremental algorithm**:
  - the overall time for finding all upper and lower tangents, removing chains, and linking points, is $\mathbf{\Theta(n)}$, because each point is added just once. Removed chains won't be analysed again.

Back to Convex hull

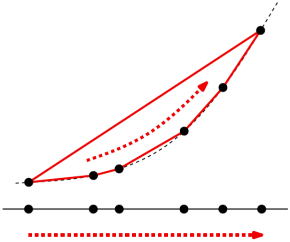
Time complexity lower bound

Any convex hull algorithm that uses line-side tests to find the hull requires $\mathbf{\Omega(n \log n)}$ line-side tests in the worst case (in a decision tree model).

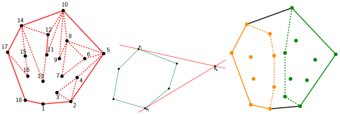
Idea of the proof (by reduction from sorting): Given a set of n distinct integers, $V = \{x_1, x_2, \ldots, x_n\}$, if we compute the $\mathcal{CH}(P)$ with

$$P = \{(x_k, x_k^2) \mid 1 \leq k \leq n\}$$

we can list V in increasing order by starting at the leftmost point of $\mathcal{CH}(P)$. Since sorting requires $\mathbf{\Omega(n \log n)}$ in the worst case, so does the convex hull problem in 2D. $\square$



Graham scan, the incremental algorithm, and the divide-and-conquer algorithm sketched in a previous class, are asymptotically optimal algorithms for the convex hull problem, in the worst case.



The Queue ADT

A **queue** is a collection whose elements are added at one end (called the **tail** or **end** of the queue) and removed from the other end (called the **front** or **head** the queue). It is a **"first in first out"** (**FIFO**) data structure.

- **Basic queue operations** besides the **creation/destruction**:
  - `push(item)`: Add an element at the end
  - `pop()`: Remove the first element
  - `front()`: Access the first element
  - `back()`: Access the last element
  - `empty()`: Test whether the queue is empty
  - `size()`: Return the size of the queue

- Usually, no support for **random access** or **iterators** or **sorting**.
  If we need that, we'd better check whether a queue is the best data structure, before we start implementing a specialized queue version.

Implementation of MyQueue

Using a linked-list as container (this code was not presented in the lecture)

```
#include <list>
using namespace std;

template <typename T> class MyQueue {
private:
  list<T> l;
public:
  MyQueue(){};     // the list is empty
  void push(T d) { l.push_back(d); }
  void pop() { l.pop_front(); }
  bool empty() { return l.size() == 0; }
  T front() { return l.front(); }
  T back() { return l.back();}
  int size() { return l.size();}
  void print() { for (auto x: l)  cout << x << endl; }  // !!
};
```

In some applications, it would be interesting to be able to **remove any element** from the queue, e.g., if elements can abandon the queue. This ADT does not support that.

Implementation of MyQueue

Using a circular array as container (this code was not presented in the lecture)

```
#include <vector>
#include <cassert>
using namespace std;

template <typename T> class MyQueue {
private:
    vector<T> v;
    int nmax, start, end;
public:
  MyQueue(int c=100):
      v(vector<T>(c)), nmax(c), start(-1), end(0) {};
  void push(T d) {
    assert(start==end && "queue is full");
    if (empty()) start = end;
    v[end] = d;
    end = (end+1) % nmax;
  }                                     // continues in next slide
};
```

Implementation of `MyQueue`

Using a circular array as container **(this code was not presented in the lecture)**

```
  void pop() {
    assert(!empty()  && "queue is empty");
    start = (start+1) % nmax;
    if (start == end) {  // it had just one element
      start = -1; end = 0;
    }
  }

  int size() {
    if (empty()) return 0;
    return (end+nmax-start)%nmax;
  }

  bool empty() { return start ==  -1; }
  T front() { assert(!empty()); return v[start];  }
  T back() { assert(!empty()); return v[(end+nmax-1)%nmax];}
};
```

Time complexity of: `push`, `pop`, `size`, `empty`, `front`, and `back` is $\mathcal{O}(1)$.

STL `queue` template

- **C++ STL std::queue** is implemented as a **container adaptor**, with **std::deque** as default container, if no container class is specified.

  ```
  template <class T, class Container = deque<T> > class queue;
  ```

  Examples:

  - `std::queue<int> q0;`
  - `std::queue<int,std::list<int> > q1;`

- The class template acts as a wrapper to the underlying container – **only a specific set of functions is provided** (Documentation for std::queue).

  Provides no support for **random access** or **iterators**.

- If we need those features, we'd better use, e.g., a **deque** directly or **another data structure**.
  - A **deque** (double-ended queue) is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end.
  - Link to STL documentation: std::deque