Linear Data Structures: List, Stack, Queue

L.EIC

Algoritmos e Estruturas de Dados

2023/2024

P Diniz, AP Rocha,
A Costa, B Leite, F Ramos, J Pires, PH Diniz, V Silva

ADT: List

- **List**
  - sequence of elements of the same type

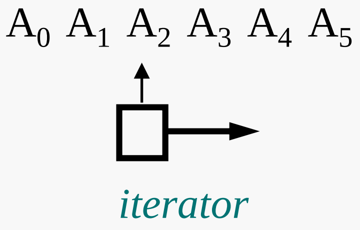
    $$A_0, A_1, A_2, \ldots, A_n$$

  - *empty list*: list with no elements
  - most usual operations:
    - create an empty list
    - add/remove an element to a list
    - determine the position of an element in the list
    - determine the length (number of elements) of a list
    - concatenate two lists

ADT: Iterator

When handling a list, it is often necessary to traverse through the list, treating its elements one by one

- **Iterator**
  - object that references an element of certain ADTs
  - abstraction that allows to encapsulate information about the state of the ADT processing (i.e., the position of the element to be processed)



A_0 A_1 A_2 A_3 A_4 A_5

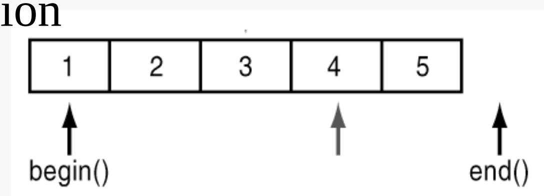
iterator

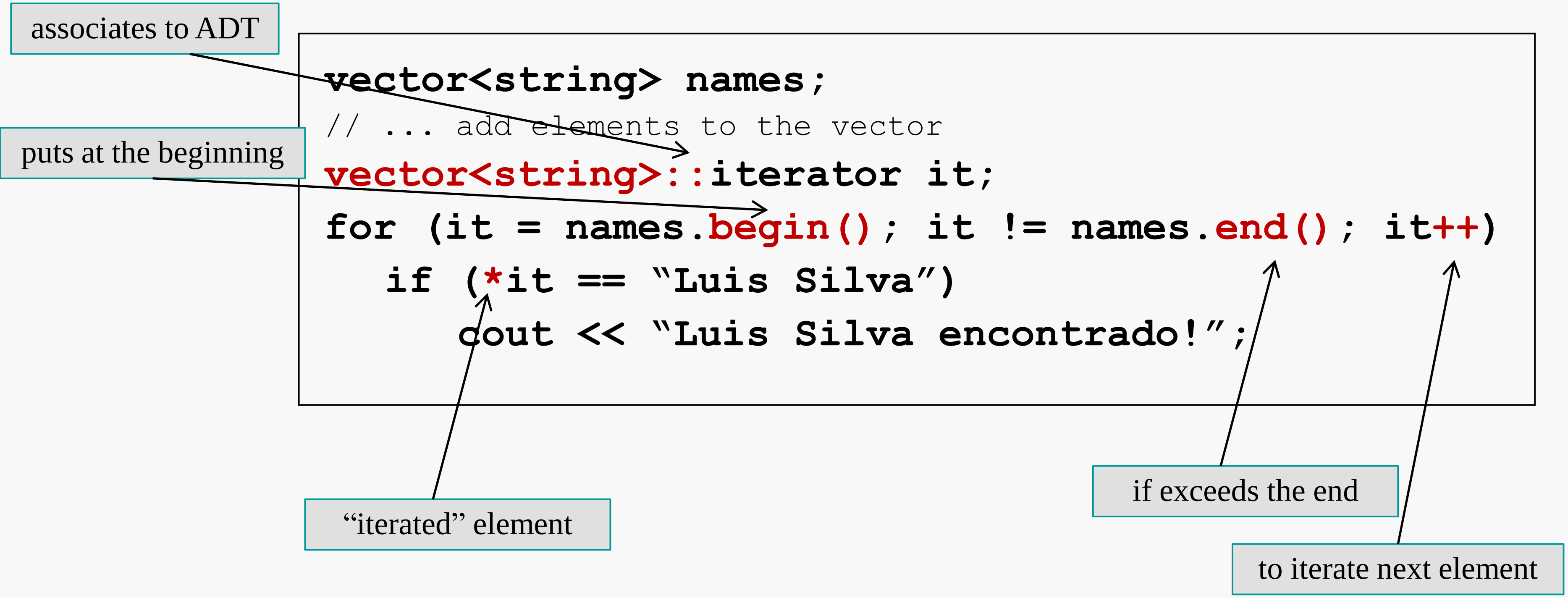
  - basic operations:
    - start
    - advance
    - check if it came to an end

* Iterators: some notes

Iterator

- associates to an Abstract Data Type or its implementation
- example of using vector iterators
    - consider the vector ***names*** (vector of strings)
    - search for the name “Luis Silva” in the vector ***names***





```
vector<string> names;
// ... add elements to the vector
vector<string>::iterator it;
for (it = names.begin(); it != names.end(); it++)
   if (*it == “Luis Silva”)
      cout << “Luis Silva encontrado!”;
```

associates to ADT

puts at the beginning

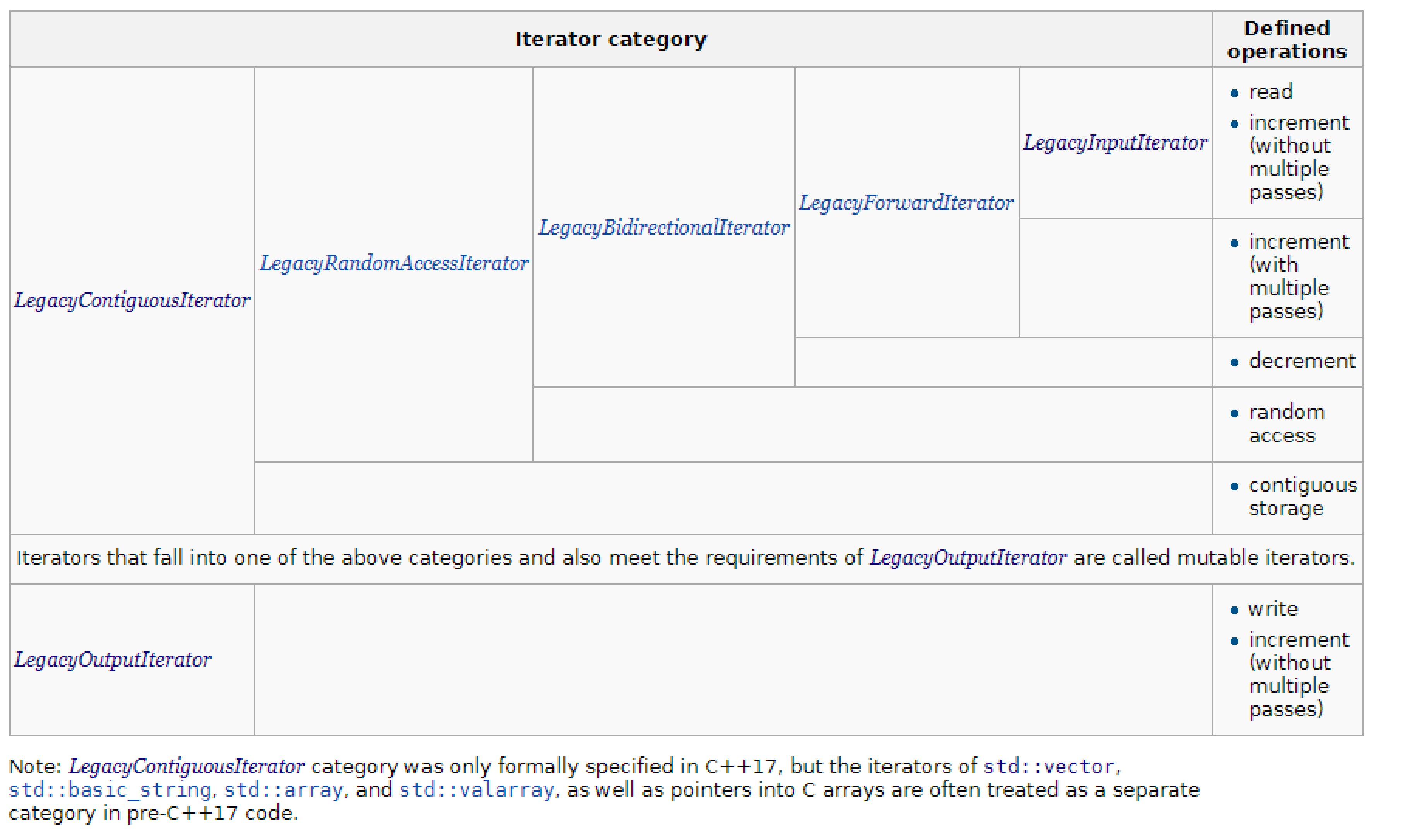
“iterated” element

if exceeds the end

to iterate next element

* Iterators: some notes

- more information about iterators in C++ STL
  - https://en.cppreference.com/w/cpp/iterator

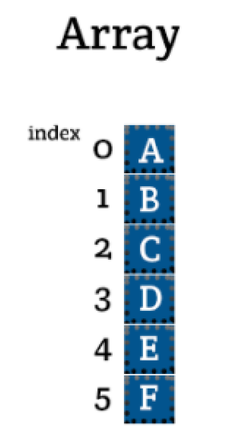


<table>
<tr><th colspan="5">Iterator category</th><th>Defined operations</th></tr>
<tr><td rowspan="5"><em>LegacyContiguousIterator</em></td><td rowspan="4"><em>LegacyRandomAccessIterator</em></td><td rowspan="3"><em>LegacyBidirectionalIterator</em></td><td rowspan="2"><em>LegacyForwardIterator</em></td><td><em>LegacyInputIterator</em></td><td>• read
• increment (without multiple passes)</td></tr>
<tr><td></td><td>• increment (with multiple passes)</td></tr>
<tr><td colspan="2"></td><td>• decrement</td></tr>
<tr><td colspan="3"></td><td>• random access</td></tr>
<tr><td colspan="4"></td><td>• contiguous storage</td></tr>
<tr><td colspan="6">Iterators that fall into one of the above categories and also meet the requirements of <em>LegacyOutputIterator</em> are called mutable iterators.</td></tr>
<tr><td><em>LegacyOutputIterator</em></td><td colspan="4"></td><td>• write
• increment (without multiple passes)</td></tr>
</table>

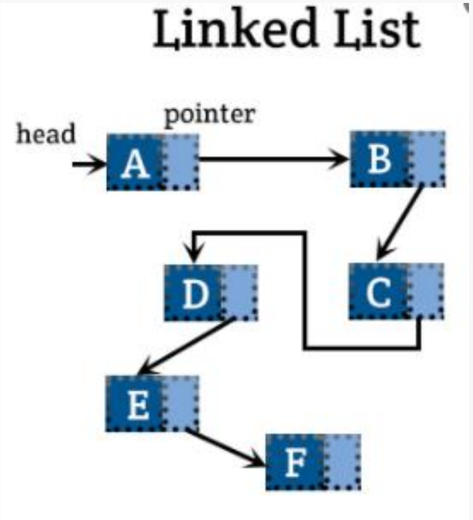
Note: *LegacyContiguousIterator* category was only formally specified in C++17, but the iterators of `std::vector`, `std::basic_string`, `std::array`, and `std::valarray`, as well as pointers into C arrays are often treated as a separate category in pre-C++17 code.

Lists: implementation

- List implementation techniques
  - array-based



  - based on pointers
    - linked lists
    - circular lists
    - doubly linked lists

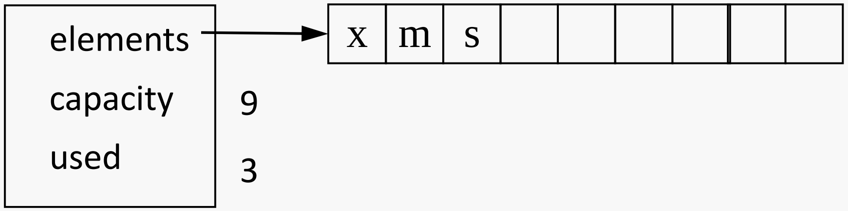


Lists: array-based implementation

Implementation of array-based lists

- elements are stored in an *array*
- *array* size (number of elements) requires constant monitoring
- search, insertion and removal of elements:
  - operations of time complexity $O(size)$

- Possible solution



Lists: array-based implementation

class **VList**

```
template <class Object>
class VList {
   Object* elements;
   int used;
   int capacity;

   friend class VListItr<Object>;

// continue…
```

iterator

Lists: array-based implementation

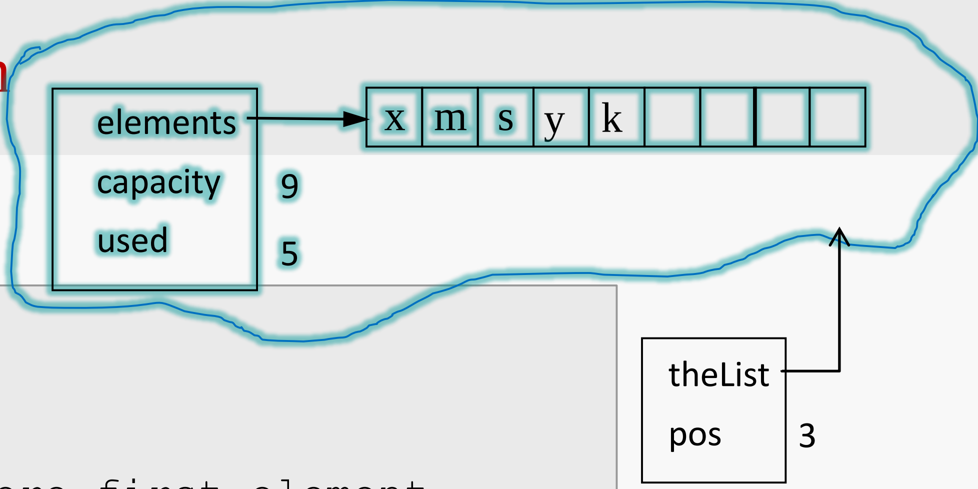
class **VList**

```
public:
   VList(int size = 100);
   VList(const VList &other);
   ~VList();
   bool isEmpty() const;
   void makeEmpty();
   VListItr<Object> first() const;
   VListItr<Object> beforeStart() const;
   void insert (const Object& x, const VListItr<Object>& p);
   void insert (const Object& x, int pos);
   VListItr<Object> find (const Object& x) const;
   void remove (const Object& x);
   const VList& operator= (const VList& other);
};
```

Lists: array-based implementation

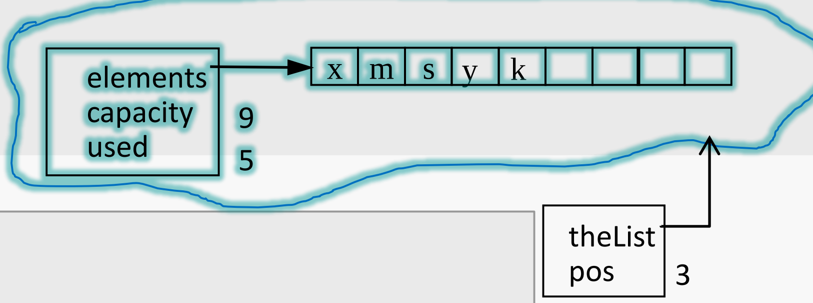
the iterator: class **VListItr**

```
template <class Object>
class VListItr {
   int pos; // index or -1 if before first element
   const VList<Object>& theList;       // reference list

  //private constructor
   VListItr(const VList<Object>& l1, int p = 0):
                        theList(l1), pos (p) {
       if ( p > theList.used || p < -1 )
          throw BadIterator();
   }

   friend class VList<Object>;
  //continue…
```



iterator reference "y"

Lists: array-based implementation



iterator reference "y"

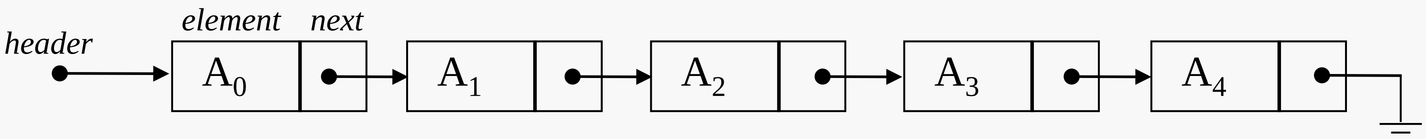
the iterator: class **VListItr**

```
public:
   bool isPastEnd() const {
      return( theList.used == 0 || pos >= theList.used);
   }

   void advance() {
      if (!isPastEnd())
         pos++;
   }

   const Object& retrieve() const {
      if (isPastEnd() || pos < 0)
         throw BadIterator();
      return theList.elements[pos];
   }
};
```

Lists: linked list implementation

Implementation of linked lists

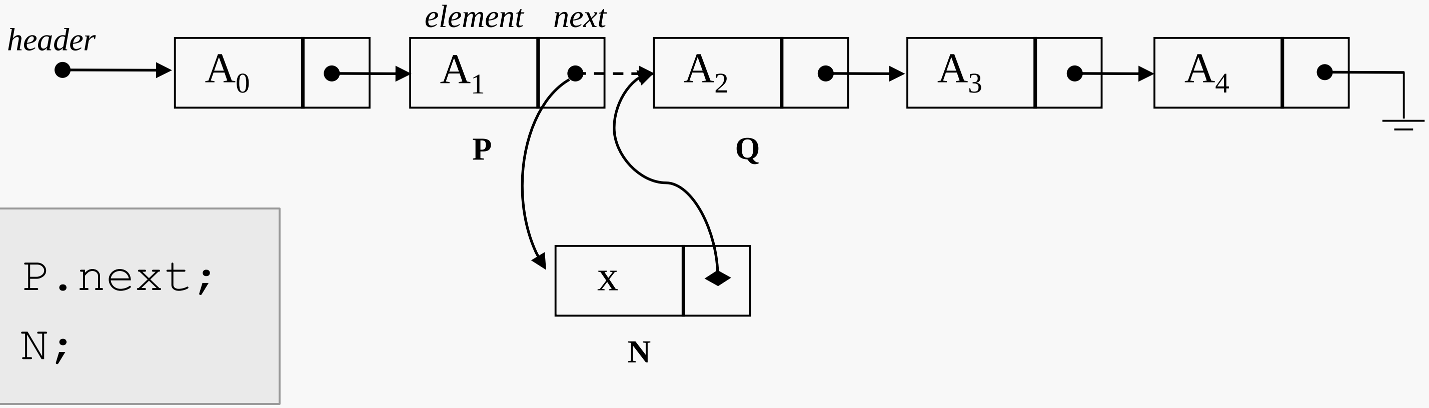
- A linked list is made up of nodes. The node has two fields:
  - the object to include in the list
  - a pointer to the next element (node) in the list



- list size varies easily by dynamic allocation
- may or may not have a special node (*header*)
- insertion and removal of elements:
  - operations of time complexity $O(1)$
- search of elements:
  - operation of time complexity $O(size)$

Linked Lists

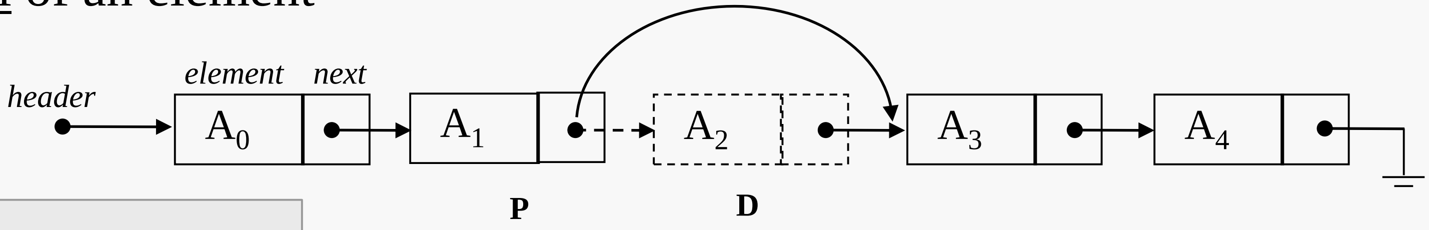
- Insertion of an element



```
N.next = P.next;
P.next = N;
```

first element is a special case

- Removal of an element

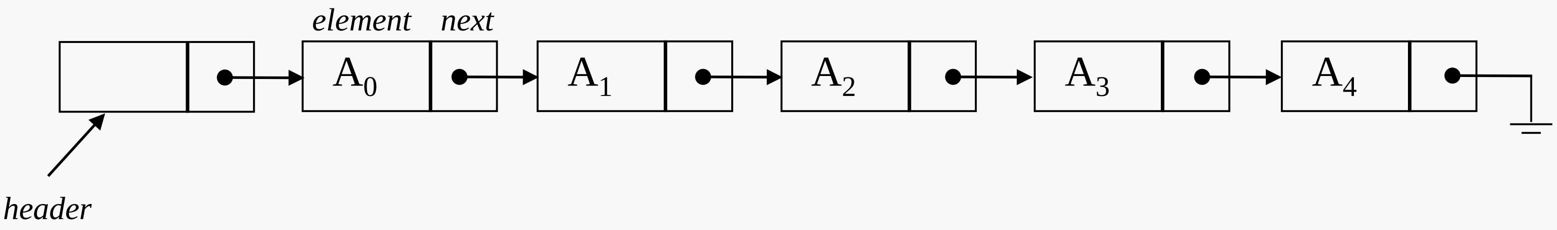


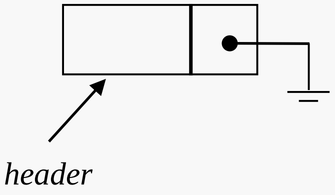
```
P.next = D.next;
Delete D;
```

Linked Lists

Use a *header* (dummy node) to simplify list manipulation

- the first node is no longer a special case

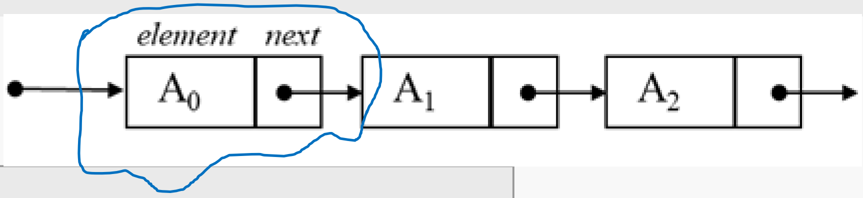




empty list

Lists: linked list implementation

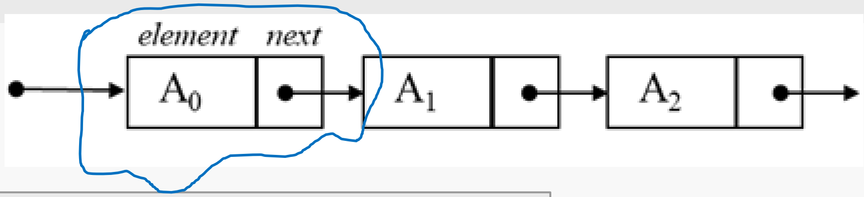
class **LListNode**



```
template <class Object>
class LListNode {
   LListNode(const Object& theElement = Object(),
                                  LListNode* n = 0)
      : element(theElement), next(n) {}

   Object element;
   LListNode* next;

   friend class LList<Object>;
   friend class LListItr<Object>;
};
```

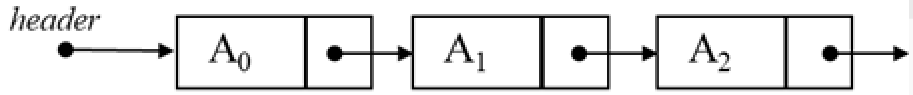
Lists: linked list implementation

class **LListNode**



```
template <class Object>
class LListNode {
   LListNode(const Object& theElement = Object(),
                                 LListNode* n = 0)
      : element(theElement), next(n) {}

   Object element;
   LListNode* next;

   friend class LList<Object>;
   friend class LListItr<Object>;
};
```

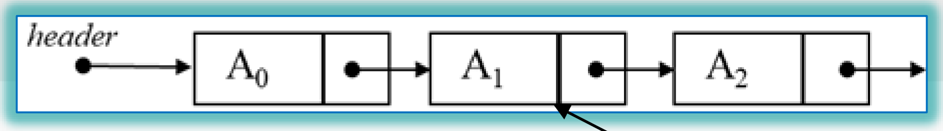
Lists: linked list implementation



class **LList**

```
template <class Object>
class LList {
   LListNode<Object>* header;     // dummy node
   LListItr<Object> findPrevious(const Object& x) const;
public:
   LList();
   bool isEmpty() const;
   void makeEmpty();
   LListItr<Object> first() const;
   LListItr<Object> beforeStart() const;
   void insert(const Object& x, const LListItr<Object>& p);
   void insert(const Object& x, const int pos = 0);
   LListItr<Object> find(const Object& x) const;
   void remove(const Object& rhs);
   const LList& operator = (const LList& rhs);
 };
```

Lists: linked list implementation



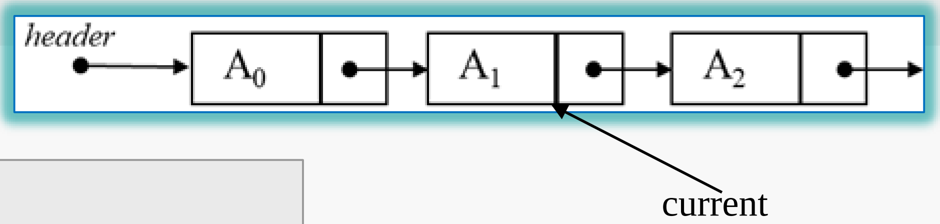
current

iterator reference "A_1"

the iterator: class **LListItr**

```
template <class Object>
class LListItr {
   LListNode<Object>* current;

   LListItr(LListNode<Object>* theNode):current(theNode){};
   friend class LList<Object>;

public:
   LListItr() : current(0) {};



   // continue…
```

Lists: linked list implementation



iterator reference "A_1"

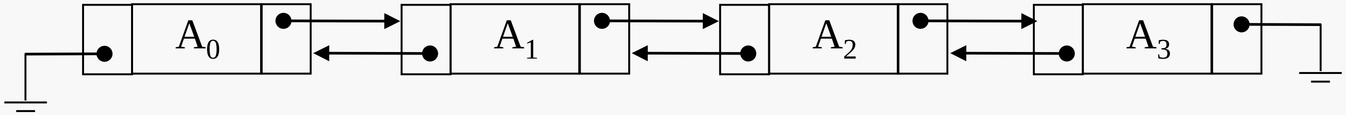
the iterator: class **LListItr**

```
    bool isPastEnd() const {
      return current == 0;
   }

    void advance() {
       if ( !isPastEnd() )
          current = current->next;
    }

    const Object& retrieve() const {
       if ( isPastEnd() )
          throw BadIterator();
       return current->element;
    }

 };
```

more Linked Lists

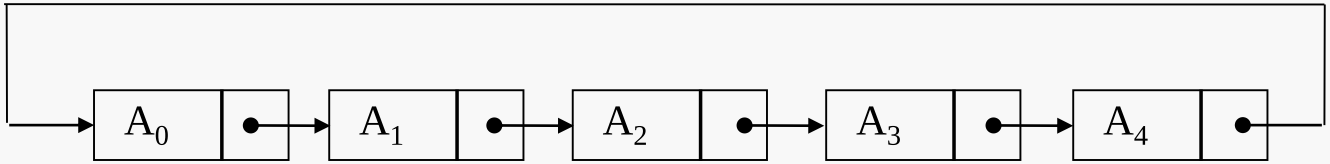
Doubly Linked List

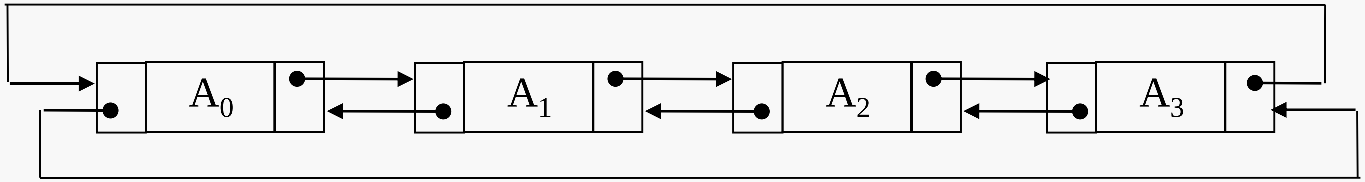
- may or may not have special nodes (header and footer)



Circular Linked List

- (singly or doubly linked)may or may not have special nodes (header and footer)





Class *list* (STL)

Class ***list*** in STL:

- sequence that can be traversed in both directions: "forward" or "backward“
  - *sequence*: variable size *container* with elements arranged linearly
  - *container*: object that stores other objects (elements)
    - supports element access methods
    - has associated iterator

- implemented as a doubly linked list

en.cppreference.com/w/cpp/container/list

Class *list* (STL)

Some methods of *list* (STL)

- iterator **begin**()
- iterator **end**()
- size_type **size**() const
- bool **empty**() const
- reference **back**()
- reference **front**()
- iterator **insert**(iterator p, const T & e) // insere *e* antes de *p*, retorna iterator para *e*
- void **push_back**(const T & e)
- void **push_front**(const T & e)
- iterator **erase**(iterator p) // retorna iterator para o elemento seguinte ao removido
- void **pop_front**()
- void **pop_back**()
- void **clear()**
- void **sort()**

Class *list* (STL)

and the **sort()** algorithm

~~void sort(iterator start, iterator end);~~

~~void sort(iterator start, iterator end, StrictWeakOrdering cmp);~~

- class *list* cannot use **sort()** algorithm
- STL **sort()** algorithm works with *Random Access Iterators* and not *Bidirectional Iterators*
- class *list* uses *Bidirectional Iterators*
- but *list* has **sort()** member function

and the **find()** algorithm

iterator find(iterator start, iterator end, const TYPE& val);

iterator find_if(iterator start, iterator end, Predicate up);

- class *list* can use **find()** algorithm

Example (using lists)

Sparse polynomials

- High degree polynomials but with few terms, eg:

  $3x^{1000} + 4x^{200} + 4$

- Polynomial of degree n:

  $P_n(x) = a_{n-1}\, x^{n-1} + \ldots + a_2\, x^2 + a_1\, x^1 + a_0$

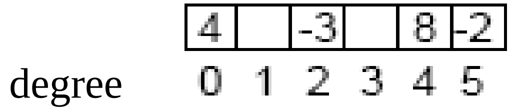
  - Polynomial representation: list of terms
  - term i: $a_i\, x^i$
  - operate with polynomials
    - $P1 + P2$
    - $k \times P1$
    - $P1 \times P2$
    - evaluate polynomial: P(x)

Example (using lists)

Array-based representation:

- array contains the coefficients
- position is the degree of the term
- wastes memory space
  - space is proportional to the degree and not to the number of terms

$-2x^5 + 8x^4 - 3x^2 + 4$ (array-based)

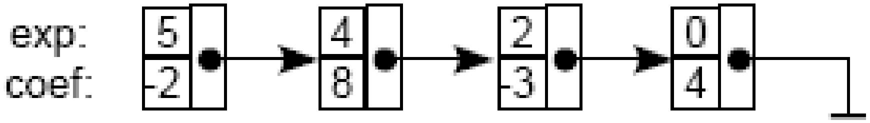


	4		-3		8	-2
degree	0	1	2	3	4	5

Representation based on linked lists:

- make better use of memory space
- list of terms [*pair (coefficient, exponent)*]
- the highest degree term is the first on the list
- the list is kept sorted by degree (descending order)

$-2x^5 + 8x^4 - 3x^2 + 4$ (linked list)



Example (using lists)

class **Term**

```
class Term
{
public:
   double coefficient;
   int exponent;
   Term (double c=0.0, int e=0): coefficient(c), exponent(e) { };
   double evaluate(double x);
};

double Term::evaluate(double x) {
    return coefficient*pow(x, exponent);
}
```

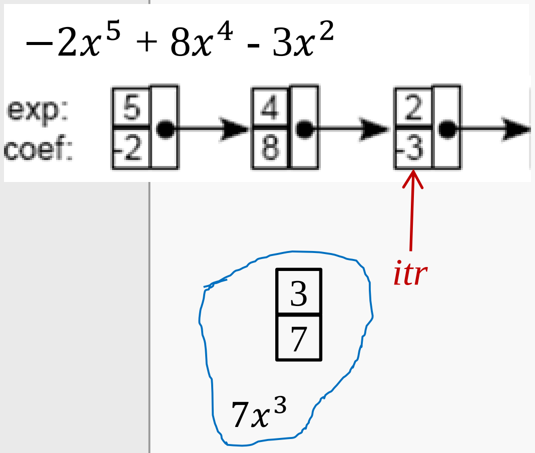
Example (using lists)

class **Polynomial**

```
class Polynomial
{
public:
   list<Term> terms;

   Polynomial() { };
   Polynomial(const Polynomial& p);
   Polynomial(Term& t);

   void operator +=(const Polynomial& p);
   void operator +=(const Term& t);
   void operator *=(const Term& t);
   double evaluate(double x);
};
```

Example (using lists)

```
void Polynomial::operator +=(const Term& t)  {
   list<Term>::iterator itr = terms.begin();
   list<Term>::iterator itre = terms.end();
   while ( itr != itre ) {
      if ( itr->exponent < t.exponent ) {
          terms.insert(itr,t);
          return;
      }
      else if (itr->exponent == t.exponent ) {
           itr->coefficient += t.coefficient;
           return;
      }
      else  itr++;
   }
   terms.insert(itr,t);
}
```



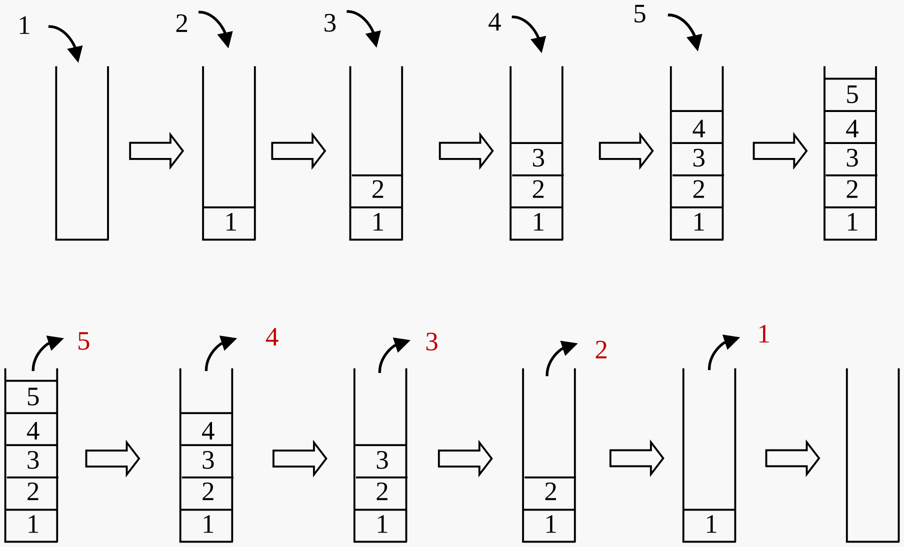
$-2x^5 + 8x^4 - 3x^2$

ADT: Stack

- **Stack**
  - sequence of elements of the same type
  - linear data structure in which the insertion and removal of elements from a sequence is done by the same end, called the top of the stack
  - a stack can be thought of as a particular case of list
  - because it is a simpler data structure than the list, it is possible to get more efficient implementations
  - the concept of iterator does not apply to this data structure
  - the stack is a structure of type LIFO (Last-In-First-Out)

ADT: Stack

- **Stack**
  - most usual operations:
    - create an empty stack
    - add/remove an element to a stack
    - determine the last element placed on the stack



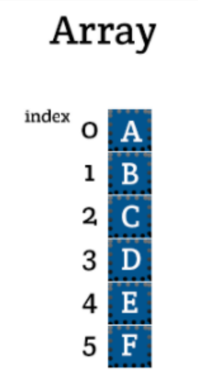
Stack implementation

- Stack implementation

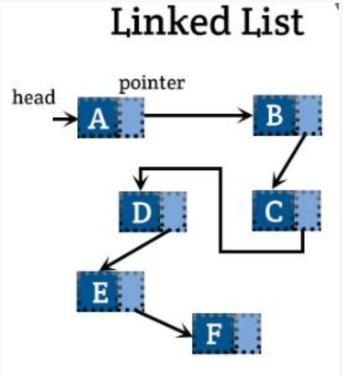
  “a stack can be thought of as a particular case of list”

  - almost direct implementation from existing methods in list
  - adapting an existing class

  - array-based



  - linked list



- let's adapt the linked list-based implementation

Stack: linked list-based implementation

class **LStack**

```
template <class T>
class LStack {
   LList<T> list;

public:
   bool isEmpty() const {
      return list.isEmpty();
   };

   void push(const T& x) {
      list.insert(x,0);
   }
```

```
   T top() const {
      if (list.isEmpty())
          throw NoElement();
      else
         return list.first().retrieve();
   };

   void pop() {
      list.remove(this->top());
   }

};
```

Class *stack* (STL)

- class stack<T> in STL
  - implemented as a deque container

* Note: deque container in STL implementation

- double-ended queue
- is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end:
  - Random access - $O(1)$
  - Insertion or removal of elements at the end or beginning - $O(1)$
  - Insertion or removal of elements - $O(n)$
- the elements of a deque are not stored contiguously: typical implementations use a sequence of individually allocated fixed-size arrays

Class *stack* (STL)

- class stack<T> in STL

- some methods
  - bool **empty()** const
  - size_type **size()** const
  - T& **top()**
  - void **push**(const T&)
  - void **pop()**
  - stack& **operator =**(const stack&)
  - bool **operator ==**(const stack&, const stack&)
  - bool **operator <**(const stack&, const stack&)

Example (stack)

RPN notation (Reverse Polish Notation)

- mathematical expressions where operators follow operands (postfix notation)
- advantage: no parentheses or precedence rules required

Infix notation

- binary operators appear between operands

Infix notation: $2 \times (4 + 5)/3$

RPN notation: $2\ 4\ 5 + \times\ 3\ /$

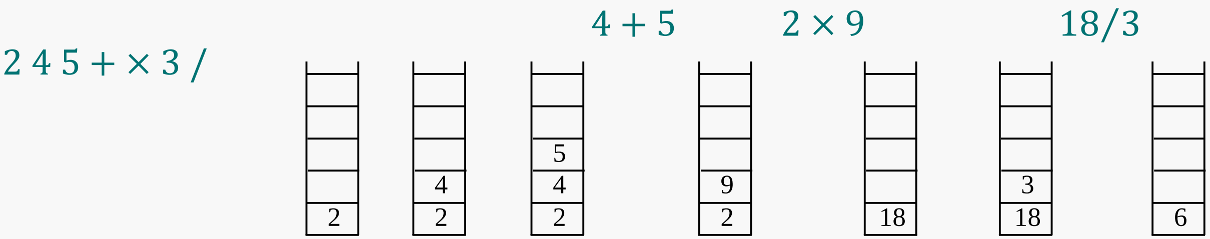
Example (using stacks)

Evaluation of RPN expressions: *algorithm*

- Sequentially process the expression elements.

  For each element:

  - If the element is a number (operand), put it on the stack
  - If the element is an operator
    - Remove the two elements from the top of the stack
    - Process the elements according to the operator
    - Put the result on the stack
- Remove the (single) element from the stack: is the result



Example (using stacks)

```
float calcOp(float v1, float v2, char op) {
   switch(op) {
      case '+' : return v1+v2;
      case '-' : return v1-v2;
      case '*' : return v1*v2;
      case '/' : return v1/v2;
      default: throw InvalidOperation();
   }
}
```

Example (using stacks)

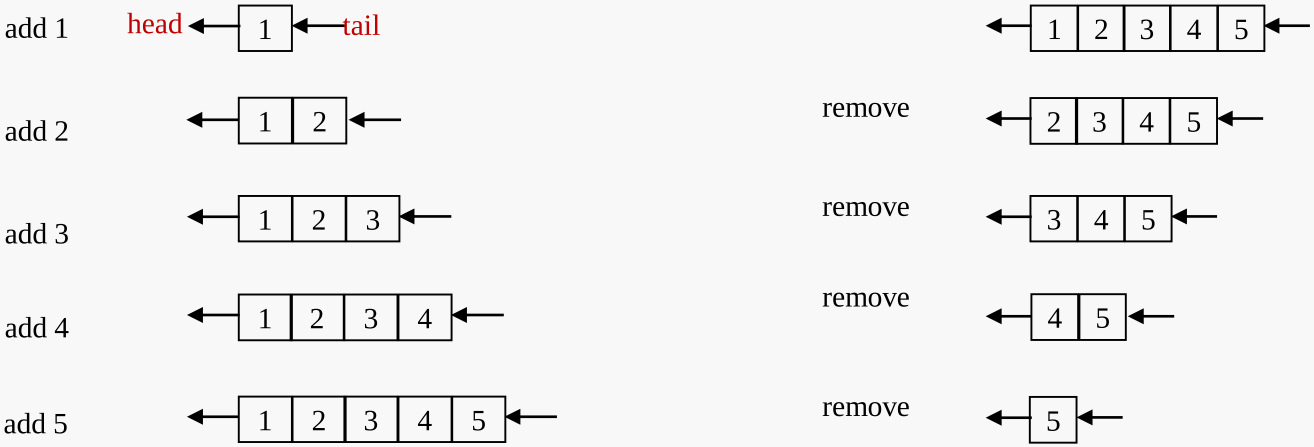
```
float evaluateRPN(string expr) { // numbers with 1 digit only
   stack<float> stackN;
   for (int i=0; i<expr.length(); i++ ) {
      if ( expr[i]>='0' &&  expr[i]<='9')   // is number
         stackN.push(expr[i]-48);
      else {
         float num1 = stackN.top();
         stackN.pop();
         float num2 = stackN.top();
         stackN.pop();
         float res = calcOp(num2, num1, expr[i]);
         stackN.push(res);
      }
   }
   float res = stackN.top();  stackN.pop();
   return res;
}
```

ADT: Queue

- **Queue**
  - sequence of elements of the same type
  - linear data structure in which the *insertion* and *removal* of elements from a sequence is done by opposite ends, generally called the head and tail of the queue.
  - a queue can be thought of as a particular case of list
  - because it is a simpler data structure than the list, it is possible to get more efficient implementations
  - the concept of iterator does not apply to this data structure
  - the queue is a FIFO (First-In-First-Out) structure

ADT: Queue

- **Queue**
  - most usual operations:
    - create an empty queue
    - add/remove an element to a queue
    - determine the element on the head of the queue (oldest element)



add 1 head 1 tail

add 2 1 2

add 3 1 2 3

add 4 1 2 3 4

add 5 1 2 3 4 5

1 2 3 4 5

remove 2 3 4 5

remove 3 4 5

remove 4 5

remove 5

Queue implementation

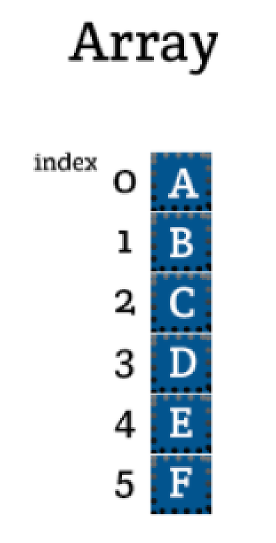
- Queue implementation

  "a queue can be thought of as a particular case of list"

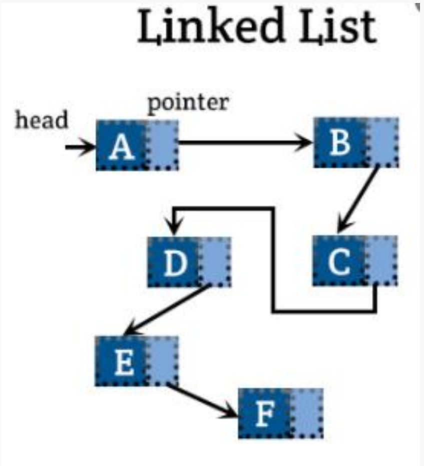
  – almost direct implementation from existing methods in list

  – adapting an existing class

  – array-based



  – linked list



- let's adapt the linked list-based implementation

Queue: linked list-based implementation

class L**Queue**

```
template <class T>
class LQueue {
   LList<T> list;

public:
   bool isEmpty() const {
      return list.isEmpty();
   }

   void push(const T& x) {
      list.insert(x,list.size());
   }
```

```
   T front() const {
     if (list.isEmpty())
         throw NoElement();
     else
         return list.first().retrieve();
    }

    void pop() {
       list.remove(this->front());
    }

};
```

Class *queue* (STL)

- class queue<T> in STL
  - implemented as a deque container

- some methods
  - bool **empty()** const
  - size_type **size()** const
  - T& **front()**
  - T& **back()**
  - void **push**(const T&)
  - void **pop()**
  - queue& **operator =**(const queue &)
  - bool **operator ==**(const queue&, const queue&)
  - bool **operator <**(const queue&, const queue&)

Example (using queues)

- Implement a class to manage the print queue of a network printer
- The network printer, when receiving a file for printing, adds it to a queue following a FIFO policy

```
class Document {
   string name;
   string owner;
   int size;
   int sheets;
public:
   Document(string n, string o, int s, int sh): name(n),
         owner(o), size(s), sheets(sh){}
   friend class Printer;
};
```

Example (using queues)

```
class Printer {
   queue<Document> printQueue;
   int memory;
   int sheets;
public:
   Printer(int m, int sh): memory(m), sheets(sh){}
   bool addDocument(Document doc);
   Document printDocument();
   void removeDocument(string n, string o);
};
```

Example (using queues)

```
// add a new document to the print queue; if the document size is
// higher than the available memory, the document is ignored.
bool Printer::addDocument(Document doc) {
    int memoryUsed = 0;
    queue<Document> temp(printQueue);
    while(!temp.empty()) {
        memoryUsed += temp.front().size;
        temp.pop();
    }
    bool res = false;
    if( (memory - memoryUsed) >= doc.size ) {
        printQueue.push(doc);
        res = true;
    }
    return res;
}
```

Example (using queues)

```
//Prints the next document on the queue. If the document is completely
//printed, exit the queue. If there is not enough sheets to print all the
//pages, print as many as possible and throws an exception

Document Printer::printDocument() {
    Document d = printQueue.front();
    if( sheets >= d.sheets ) {
        sheets -= d.sheets;
        printQueue.pop();
    }
    else {
        d.sheets -= sheets;
        printQueue.front() = d;
        sheets = 0;
        throw ErrorSheets();
    }
    return d;
}
```

Example (using queues)

```
// Removes from the print queue the document with name n, belonging to owner o.

void Printer::removeDocument(string n, string o) {
    queue<Document> temp;
    while(!printQueue.empty()) {
        if( printQueue.front().name == n &&
                         printQueue.front().owner == o )
            printQueue.pop();
        else {
            temp.push(printQueue.front());
            printQueue.pop();
        }
    }
    printQueue = temp;
}
```