Cuestiones sobre implementación

Cuestiones sobre implementación
Sobrecarga de funciones

Permite que diversas funciones tengan el mismo nombre, y se llame a una u otra en base al número y tipo de parámetros pasados:

```
class fecha {
  ...
  bool operator<(const fecha&b) { ... }
};
class racional {
  ...
  bool operator<(const racional&b) { ... }
};

 float minimo(float a, float b) { return (a < b ? a : b); }
 int minimo(int a, int b) { return (a < b ? a : b); }
 fecha minimo(fecha a, fecha b) { return (a < b ? a : b); }
 racional minimo(racional a, racional b) { return (a < b ? a : b);
```

Cuestiones sobre implementación
Generalización por plantillas

Permite abstraerse del tipo de dato utilizado en una función:

```
int minimo(const int *v,int n) {
  int i;
  int min = v[0];
  for (i = 1;i < n;i++)
    If (v[i] < min)
      min = v[i];
  return min;
};


int main(int argc,char *argv[]) {
  int v[30];
  ...
  cout << minimo(v,30) << endl;
}
```

```
template<typename T>
    T minimo(const T *v,int n) {
  int i;
  T min = v[0];
  for (i = 1;i < n;i++)
    if (v[i] < min)
      min = v[i];
  return min;
};

int main(int argc,char *argv[]) {
  int v[30]; char z[30];
  ... //Evaluación precompilador
  cout << minimo(v,30) << endl;
  cout << minimo(z,30) << endl;
}
```

Cuestiones sobre implementación
Generalización por plantillas

Permite abstraerse del tipo de dato utilizado en un TDA:

```
class VectorDinamico {
  int *contenido;
  VectorDinamico();
  void insertar(int pos,int v);
  int extraer(int pos);
  bool contiene(int v);
  int count();
};


int main(int argc,char *argv[]) {
  VectorDinamico d;
  d.insertar(3);
}
```

```
template <typename T>
      class VectorDinamico {
  T *contenido;
  VectorDinamico();
  void insertar(int pos,T v);
  T extraer(int pos);
  bool contiene(T v);
  int count();
};

//También vale template <class T>

int main(int argc,char *argv[]) {
  VectorDinamico<int> d;
  d.insertar(3);
  VectorDinamico<char> e;
  d.insertar('a');
}
```

Cuestiones sobre implementación
Functores

Un functor es un objeto de una clase que tiene declarado el operator() (paréntesis), y permite llamarlo como una función e incluso pasarlo como parámetro.

```
class ComparadorAlumnosCreciente {

  Public:

  bool operator() (const Alumno &a,const Alumno &b) const;

};

int main() {

  Alumno uno,dos;

  ComparadorAlumnosCreciente creciente;

  cout << creciente(uno,dos) << endl;

}
```

Cuestiones sobre implementación
Implementación de iteradores

Podemos implementar los iteradores:

```
iterator                //de begin() a end()
reverse_iterator        //de rbegin() a rend()
const_iterator          //de begin() a end() (no permite modificar)
const_reverse_iterator //de rbegin() a rend() (no permite modific)
```

Aunque también podríamos, por ejemplo:

```
random_iterator         //acceso aleatorio
const_random_iterator   //acceso aleatorio (no permite modificar)
preorder_iterator       //iterador de recorrido preorden.
...
```

Debemos definirlos dentro del TDA:

```
class Lista {
  class iterator {
    ...
  };
};
```

Cuestiones sobre implementación
Implementación de iteradores

La representación de los iteradores podría ser:

- Índice: Un entero que indica la posición (si podemos usar acceso aleatorio).
- Puntero: Un puntero que indica la posición (si podemos usar aritmética de punteros o disponemos de listas enlazadas).
- Lista de punteros: Si queremos hacer recorrido en preorden/postorden/inorden de un árbol, podríamos recorrer así el árbol e introducir referencias a los elementos conforme los recorremos en un vector. Este recorrido es O(n). Posteriormente, el iterador trabajaría sobre ese vector, pudiendo almacenarse internamente un puntero o un índice que itera sobre el vector, que contiene el recorrido preorden/postorden/inorden.

Cuestiones sobre implementación
Implementación de iteradores

Debemos definirle estos métodos:

```
operator++(int val) // ite++ (Devuelve y pasa a siguiente elem.)
operator++()        // ++ite (Pasa a siguiente elem. y devuelve)
operator*()         // *ite  (Devuelve el elemento apuntado)
```

Opcionalmente, podemos definir también:

```
operator--(int val) // ite-- (Devuelve y pasa a elem. anterior)
operator--()        // --ite (Pasa a elem. anterior y devuelve)
operator+(int val)  // ite+5 (salto aleatorio)
operator-(int val)  // ite-5 (salto aleatorio)
```

Por último, podremos definir funciones para insertar, desde un iterador, elementos en posiciones determinadas de un contenedor (en la STL se proporcionan dentro de los TDAs contenedores).

Cuestiones sobre implementación
Implementación de iteradores

En el TDA deberemos implementar:

```
begin()   // Puntero al primer elemento.
end()     // Puntero a DESPUÉS del último elemento.
```

Opcionalmente, podemos definir también:

```
rbegin()  // Puntero al último elemento.
rend()    // Puntero a ANTES del primer elemento.
```

Cuestiones sobre implementación
Renombrado de tipos

Usando typedef podemos renombrar tipos muy largos a nombres muy cortos.

Por ejemplo:

```
typedef stack<int> pila;

typedef priority_queue<int,vector<int>,micomparador> colaprio;
```