



Programación 2

Tipos recursivos de datos I

Fernando Orejas

Transparencias basadas en las de Ricard Gavaldà

1. Punteros y memoria dinámica
2. Conceptos básicos sobre los tipos recursivos de datos
3. Pilas
4. Colas
5. Listas
6. Árboles
7. Colas con prioridad

Punteros y memoria dinámica

Tipos recursivos de datos

Un tipo recursivo de datos es un tipo en el que, en su definición, hacemos referencia al propio tipo:

```
struct nodo{  
    int dni;  
    string name;  
    nodo* siguiente;  
}
```

Punteros

En C++, para cada tipo T, hay un tipo de apuntadores a T.

Una variable de este tipo puede contener:

- Una referencia a un objeto de tipo T (estático o dinámico)
- El valor nullptr
- Cualquier cosa rara

Cómo (no) usar punteros

```
int* p;  
int x;  
p = &x;  
x = 5  
int* q = p;  
*p = 3;  
x = *p + 1;  
p = new int;  
delete p;  
delete q;
```

Cómo (no) usar punteros

```
nodo* p, q, r;  
p = new nodo;  
q = nullptr;  
p->dni = 775577;  
p->name = "abc";  
(p->siguiente)->name = "cba";  
r = p;  
delete p;  
p = q;  
int x = r->dni;  
nodo** p1;
```

Cómo (no) usar punteros

```
nodo* p1;
```

```
vector <nodo*> v,u (5);
```

```
....
```

```
for (int i = 0; i < 5; ++i) u[i] = v[i];
```


***Conceptos básicos sobre tipos recursivos de
datos***

Tipos recursivos de datos

Un tipo recursivo de datos es un tipo en el que, en su definición, hacemos referencia al propio tipo:

- Pilas: Una pila o bien es una pila vacía o es el resultado de un push sobre otra pila y un valor

`intstack = Empty | Push of int * intstack`

- Colas y listas: lo mismo
- Árboles : Un árbol es o bien un árbol vacío o es el resultado de enraizar un valor con

`int inttree = Empty | Cons of int * inttree * inttree`

Pilas recursivas en C++

```
class stack;  
    bool vacia;  
    int primero;  
    stack resto;  
}
```

Daría lugar a un proceso infinito

Pilas recursivas en C++

```
template <class T> class stack {  
    private:  
        // tipo privado nuevo  
        struct nodo_pila{  
            T info;  
            nodo_pila* siguiente;  
        };  
        int altura;  
        nodo_pila* primero;  
        ... //operaciones pivasdas  
    public:  
        ... //operaciones públicas  
}
```

Definición de una estructura de datos recursiva

Clase con:

- **Struct** privada que define nodos enlazados con punteros. En cada nodo
 - Información de un elemento de la estructura
 - Puntero a uno o más elementos
- Atributos que contienen información global de la estructura
- Punteros a elementos distinguidos de la estructura
- Siempre asumiremos como precondition e invariante que dos estructuras de datos diferentes no comparten ningún nodo.

Ventajas de las estructuras de datos recursivas

- Correspondencia natural con la definición recursiva del tipo de datos
- No es necesario fijar a priori un tamaño máximo
- Se puede ir pidiendo memoria para los nodos, según se va necesitando
- Modificando los enlaces entre los nodos podemos:
 - Insertar o borrar elementos, sin tener que mover otros
 - Mover partes enteras de la estructura sin hacer copias

Pilas

Implementación de pilas

```
template <class T> class stack {  
    private:  
        // tipo privado nuevo  
        struct nodo_pila{  
            T info;  
            nodo_pila* siguiente;  
        };  
        int altura;  
        nodo_pila* primero;  
        ... //operaciones privadas  
    public:  
        ... //operaciones públicas  
}
```


// Constructoras y destructoras

```
stack(){  
    altura = 0;  
    primero = nullptr;  
}
```

```
stack(const stack& P){  
  
  
}
```

// Constructoras y destructoras

```
stack(){  
    altura = 0;  
    primero = nullptr;  
}
```

```
stack(const stack& P){  
    altura = P.altura;  
    primero = copia_nodo_pila(P.primeros);  
}
```

```
~stack(){  
  
}
```

// Constructoras y destructoras

```
stack(){  
    altura = 0;  
    primero = nullptr;  
}
```

```
stack(const stack& P){  
    altura = P.altura;  
    primero = copia_nodo_pila(P.primeros);  
}
```

```
~stack(){  
    borra_nodo_pila(primeros);  
}
```

// Consultoras

```
T top() const {  
    // Pre: la pila no está vacía  
    return primero->info;  
}  
  
bool empty() const {  
    return altura == 0;  
}  
  
int size() const {  
    return altura;  
}
```

// Modificadoras

```
void clear(){  
    borra_nodo_pila(primeros);  
    altura = 0;  
    primeros = nullptr;  
}
```

```
void push(const T& x){  
    nodo_pila * aux = new nodo_pila;  
    aux->info = x;  
    aux->siguiente = primeros;  
    primeros = aux;  
    ++altura;  
}
```

// Modificadoras

```
void pop(){
```

```
// Pre: la pila no está vacía
```

```
    nodo_pila * aux = primero;
```

```
    primero = primero->siguiente;
```

```
    delete aux;
```

```
    --altura;
```

```
}
```

```
// Métodos privados
// Pre: true
/* Post: si m es nullptr el resultado es nullptr,
    si no el resultado apunta a una cadena de nodos
    que es una copia de la cadena apuntada por m */

static nodo_pila* copia_nodo_pila(nodo_pila* m){
    if (m == nullptr) return nullptr;
    else{
        nodo_pila* n = new nodo_pila;
        n->info = m->info;
        n->siguiente = copia_nodo_pila(m->siguiente);
        return n;
    }
}
```

```
// Métodos privados
// Pre: true
/* Post: si m es nullptr no hace nada,
    si no libera el espacio ocupado por la cadena de
    nodos apuntada por m */
```

```
static void borra_nodo_pila(nodo_pila* m){
    if (m != nullptr) {
        borra_nodo_pila(m->siguiente);
        delete m;
    }
}
```


// La asignación

```
stack& operator=(const stack& S){  
    if (this != &S) {  
        altura = S.altura;  
        borra_nodo_pila(primeros);  
        primeros = copia_nodo_pila(S.primeros);  
    }  
    return *this;  
}
```

Colas

Implementación de colas

```
template <class T> class queue {  
    private:  
        // tipo privado nuevo  
        struct nodoCola{  
            T info;  
            nodoCola* siguiente;  
        };  
        int longitud;  
        nodoCola* primero;  
        nodoCola* ultimo;  
        ... //operaciones privadas  
    public:  
        ... //operaciones públicas  
}
```

// Constructoras y destructoras

```
queue(){  
    longitud = 0;  
    primero = nullptr;  
    ultimo = nullptr;  
}
```

```
queue(const queue& C){  
    longitud = C.longitud;  
    primero = copia_nodoCola(C.primero, ultimo);  
}
```

```
~queue(){  
    borra_nodoCola(primero);  
}
```

// Consultoras

```
T front() const {  
    // Pre: la cola no está vacía  
    return primero->info;  
}  
  
bool empty() const {  
    return longitud == 0;  
}  
  
int size() const {  
    return longitud;  
}
```

// Modificadoras

```
void clear(){
    borra_nodoCola(primerO);
    longitud = 0;
    primerO = nullptr;
    ultimo = nullptr;
}

void push(const T& x){
    nodoCola * aux = new nodoCola;
    aux->info = x;
    aux->siguiente = nullptr;
    if (primerO == nullptr) primerO = aux;
    else ultimo->siguiente = aux;
    ultimo = aux; ++longitud;
}
```

// Modificadoras

```
void pop(){  
    // Pre: la cola no está vacía  
    nodoCola * aux = primero;  
    if (primero->siguiente == nullptr) {  
        primero = nullptr;  
        ultimo = nullptr;  
    }  
    else  
        primero = primero->siguiente;  
    delete aux;  
    --longitud;  
}
```

```
// Métodos privados
// Pre: true
/* Post: si m es nullptr el resultado y u son nullptr,
    si no, el resultado apunta a una cadena de nodos
    que es una copia de la cadena apuntada por m y u
    apunta al último nodo*/
static nodo_cola* copia_nodo_cola(nodo_cola* m,
                                   nodo_cola* &u){
    if (m == nullptr) {u = nullptr; return nullptr; }
    else {    nodo_cola* n = new nodo_cola;
n->info = m->info;
n->siguiente = copia_nodo_cola(m->siguiente,u);
    if (n->siguiente == nullptr) u = n;
    return n;
    }
}
```



```
// Métodos privados
// Pre: true
/* Post: si m es nullptr no hace nada,
    si no libera el espacio ocupado por la cadena de
    nodos apuntada por m */
```

```
static void borra_nodo_cola(nodo_cola* m){
    if (m != nullptr) {
        borra_nodo_cola(m->siguiente);
        delete m;
    }
}
```

// La asignación

```
queue& operator=(const queue& Q){  
    if (this != &Q) {  
        longitud = Q.longitud;  
        borra_nodoCola(primerO);  
        primero = copia_nodoCola(Q.primerO, ultimo);  
    }  
    return *this;  
}
```

Listas

Listas

- En este curso no implementaremos los iteradores y las listas de manera general
- Implementaremos *listas con punto de interés*, que tienen funcionalidades similares, pero algunas restricciones.

Listas con punto de interés

Podemos:

- Desplazar adelante y atrás el punto de interés
- Añadir y eliminar en el punto de interés
- Consultar y modificar el elemento en el punto de interés

Implementación de Lista

```
template <class T> class Lista {  
    private:  
        struct nodo_lista{  
            T info;  
            nodo_lista* sig;  
            nodo_lista* ant;  
        };  
        int longitud;  
        nodo_lista* primero;  
        nodo_lista* ultimo;  
        nodo_lista* act;  
        ... //operaciones privadas  
    public:  
        ... //operaciones públicas  
}
```

// Constructoras y destructoras

```
Lista(){
    longitud = 0;
    primero = nullptr;
    ultimo = nullptr;
    act = nullptr;
}
Lista(const Lista& L){
    longitud = L.longitud;
    primero = copia_nodo_lista(L.primero, L.act,
                                ultimo, act);
}
~Lista(){
    borra_nodo_lista(primero);
}
```

```
// Métodos privados
```

```
// Copiar secuencia de nodos
```

```
static nodo_Lista* copia_nodo_Lista (  
    nodo_Lista* m, nodo_Lista* La,  
    nodo_Lista* &u, nodo_Lista* &a);
```

```
// Pre: true
```

```
/* Post: si m es nullptr el resultado, u y a son nullptr,  
si no, el resultado apunta a una cadena de nodos  
que es una copia de la cadena apuntada por m,  
u apunta al último nodo y a es nullptr si La no apunta  
a ningún nodo de la secuencia, o bien apunta al nodo  
copia del nodo apuntado por La*/
```



```
static nodo_Lista* copia_nodo_Lista (  
    nodo_Lista* m, nodo_Lista* La,  
    nodo_Lista* &u, nodo_Lista* &a){  
    if (m == nullptr) {u = nullptr; a = nullptr; return nullptr;  
    }  
    else {  
        nodo_lista* n = new nodo_lista;  
        n->info = m->info;  
        n->ant = nullptr;  
        n->sig = copia_nodo_Lista(m->sig, La, u, a);  
        if (n->sig != nullptr) (n->sig)->ant = n;  
        else u = n;  
        if (m == La) a = n;  
        return n;  
    }  
}
```

```
// Métodos privados
// Borrar secuencia de nodos
// Pre: true
/* Post: si m es nullptr no hace nada,
    si no libera el espacio ocupado por la cadena de
    nodos apuntada por m */
```

```
static void borra_nodo_lista(nodo_lista* m){
    if (m != nullptr) {
        borra_nodo_lista(m->sig);
        delete m;
    }
}
```

// Asignación

```
Lista& operator=(const Lista& L){  
    if (this != &L) {  
        longitud = L.longitud;  
        borra_nodo_lista(primerono);  
        primero = copia_nodo_lista(L.primerono, L.act,  
                                    ultimo, act);  
    }  
    return *this;  
}
```

// Consultoras

```
bool es_vacia() const {  
    return longitud == 0;  
}
```

```
int medida() const {  
    return longitud;  
}
```

```
T actual() const {
```

```
// Pre: La lista no está vacía y el punto de  
//      interés no es nullptr  
    return act->info;  
}
```

```
/* Consultoras para saber dónde está el punto  
de interés */
```

```
bool al_final() const {  
    return act == nullptr;  
}
```

```
int al_principio() const {  
    return act == primero;  
}
```

// Modificadoras

```
void l_vacia(){  
    borra_nodo_lista(primeros);  
    longitud = 0;  
    primeros = nullptr;  
    ultimo = nullptr;  
    act = nullptr;  
}
```

```
// Inserción
```

```
// Pre: true
```

```
/* Post: Se ha añadido un nodo con el valor x antes  
del punto de interés que sigue siendo el mismo que  
antes de la operación*/
```

```
void añadir(const T& x){  
    nodo_lista * aux = new nodo_lista;  
    aux->info = x; aux->sig = act;  
    if (longitud == 0) {  
        aux->ant = nullptr;  
        primero = aux; ultimo = aux;  
    } else if (act == nullptr) {  
        aux->ant = ultimo;  
        ultimo->sig = aux;  
        ultimo = aux;  
    }  
    ...  
}
```

(Continuación)

```
    else if (act == primero) {  
        aux->ant = nullptr;  
        act->ant = aux;  
        primero = aux;  
    } else {  
        aux->ant = act->ant;  
        (act->ant)->sig = aux;  
        act->ant = aux;  
    }  
    ++longitud;  
}
```


// Eliminación

/* Pre: la lista no está vacía y su punto de interés no está al final */

/* Post: Se ha eliminado el nodo donde estaba el punto de interés, el nuevo punto de interés es la posición siguiente al nodo eliminado */

```
void eliminar(){  
    nodo_lista * aux = act;  
    if (longitud == 1) {  
        primero = nullptr; ultimo = nullptr;  
    } else if (act == primero) {  
        primero = act->sig;  
        primero->ant = nullptr;  
    }  
    ...  
}
```

(Continuación)

```
    else if (act == ultimo) {  
        ultimo = act->ant; ultimo->sig = nullptr;  
    } else {  
        (act->ant)->sig = act->sig;  
        (act->sig)->ant = act->ant;  
    }  
    act = act->sig;  
    delete aux;  
    --longitud;  
}
```

// Concatenación

// Pre: true

/* Post: Se han añadido al final los elementos de L, el punto de interés es el primer elemento a, L queda vacía*/

```
void concat(Lista & L){
    if (L.longitud > 0) {
        if (longitud == 0) {
            primero = L.primeros; ultimo = L.ultimo;
            longitud = L.longitud;
        } else {
            ultimo->sig = L.primeros;
            (L.primeros)->ant = ultimo; ultimo = L.ultimo;
            longitud = longitud + L.longitud;
        }
        L.primeros = L.ultimo = L.act = nullptr;
        L.longitud = 0;
    }
    act = primeros;
}
```

```
// modificación y movimiento del punto de interés
```

```
/* Pre: La lista no está vacía y el punto de interés no  
es nullptr */
```

```
/* Post: Se ha reemplazado el valor del punto de interés por  
x */
```

```
void modifica_actual(const T & x){  
    act->info = x;  
}
```

```
/* Pre: true */
```

```
/* Post: Se ha movido el punto de interés al principio de la  
lista */
```

```
void inicio(){  
    act = primero;  
}
```

```
/* Pre: true */
/* Post: Se ha movido el punto de interés al final de la
        lista */
void fin(){
    act = nullptr;
}

/* Pre: La lista no está vacía y el punto de interés no
        es nullptr */
/* Post: Se ha movido el punto de interés una posición hacia
        el final */
void avanza(){
    act = act->sig;
}
```

```
/* Pre: La lista no está vacía y el punto de interés no  
    es el primer elemento */  
/* Post: Se ha movido el punto de interés una posición hacia  
    el principio */  
void retrocede(){  
    if (act == nullptr) act = ultimo;  
    else act = act->ant;  
}
```

Listas con centinela

Listas con centinela

- Nodo extra, que no contiene ningún elemento real
- Objetivo: simplificar algunas operaciones como añadir y eliminar.
- La estructura no tiene valores nullptr: el centinela permite evitarlos
- El centinela tiene como siguiente al primer elemento y como anterior al último

Implementación de Lista con centinela

```
template <class T> class Lista {  
    private:  
        struct nodo_lista{  
            T info;  
            nodo_lista* sig;  
            nodo_lista* ant;  
        };  
        int longitud;  
        nodo_lista* cent;  
        nodo_lista* act;  
        ... //operaciones privadas  
    public:  
        ... //operaciones públicas  
}
```

// Constructoras y destructoras

```
Lista(){
    longitud = 0;
    cent = new nodo_lista;
    act = cent;
    cent->sig = cent;
    cent->ant = cent;
}

Lista(const Lista& L){
    longitud = L.longitud;
    cent = copia_nodo_lista((L.cent)->sig, l.cent,
                           L.act, cent, act);
}

~Lista(){
    borra_nodo_lista(cent->sig, cent);
}
```

// Asignación

```
Lista& operator=(const Lista& L){  
    if (this != &L) {  
        longitud = L.longitud;  
        borra_nodo_lista(primeros, cent);  
        nodo_lista* aux = copia_nodo_lista((L.cent)->sig,  
                                            L.cent, L.act, cent, act);  
    }  
    return *this;  
}
```

// Consultoras

```
bool es_vacia() const {  
    return longitud == 0;  
}
```

```
int medida() const {  
    return longitud;  
}
```

```
T actual() const {
```

```
// Pre: La lista no está vacía y el punto de  
//      interés no es nullptr  
    return act->info;  
}
```

```
/* Consultoras para saber dónde está el punto  
de interés */
```

```
bool al_final() const {  
    return act == cent;  
}
```

```
int al_principio() const {  
    return act == cent->sig;  
}
```

// Modificadoras

```
void l_vacia(){  
    borra_nodo_lista(cent->sig, cent);  
    longitud = 0;  
    cent = new nodo_lista;  
    act = cent;  
    cent->sig = cent;  
    cent->ant = cent;  
}
```

```
// Inserción
```

```
// Pre: true
```

```
/* Post: Se ha añadido un nodo con el valor x antes  
del punto de interés que sigue siendo el mismo que  
antes de la operación*/
```

```
void añadir(const T& x){  
    nodo_lista * aux = new nodo_lista;  
    aux->info = x;  
    aux->sig = act;  
    aux->ant = act->ant;  
    (act->ant)->sig = aux;  
    act->ant = aux;  
    ++longitud;  
}
```

// Eliminación

/* Pre: la lista no está vacía y su punto de interés no está al final */

/* Post: Se ha eliminado el nodo donde estaba el punto de interés, el nuevo punto de interés es la posición siguiente al nodo eliminado */

```
void eliminar(){  
    nodo_lista * aux = act;  
    (act->ant)->sig = act->sig;  
    (act->sig)->ant = act->ant;  
    act = act->sig;  
    delete aux;  
    --longitud;  
}
```


// Concatenación

// Pre: true

/* Post: Se han añadido al final los elementos de L, el punto de interés es el primer elemento, L queda vacía*/

```
void concat(Lista & L){
    if (L.longitud > 0) {
        if (longitud == 0) swap(cent, L.cent);
        else { (cent->ant)->sig = (L.cent)->sig;
                ((L.cent)->sig)->ant = cent->ant;
                cent->ant = (L.cent)->ant;
                ((L.cent)->ant)->sig = cent;
                (L.cent)->sig = L.cent;
                (L.cent)->ant = L.cent;
            }
        L.act = L.cent;
        longitud = longitud + L.longitud; L.longitud = 0;
    }
    act = cent->sig;
}
```

// modificación y movimiento del punto de interés

```
void modifica_actual(const T & x){  
    act->info = x;  
}
```

```
void inicio(){  
    act = cent->sig;  
}
```

```
void fin(){  
    act = cent;  
}
```

```
void avanza(){  
    act = act->sig;  
}
```

```
void retrocede(){  
    act = act->ant;  
}
```

// Métodos privados

// Copiar secuencia de nodos

```
static nodo_Lista* copia_nodo_Lista (  
    nodo_Lista* m, nodo_Lista* c, nodo_Lista* oa,  
    nodo_Lista* &nc, nodo_Lista* &a);
```

/* Pre: m, oact y c apuntan a nodos de la misma
secuencia, oact apunta a un nodo entre m y c */

/* Post: retorna como resultado una secuencia de nodos
que es copia de la secuencia entre m y c, si m y c
apuntan al mismo nodo, entonces nc y a apuntan al
mismo nodo, si no, nc apunta a la copia del centinela
c y a apunta a la copia del nodo apuntado por oact. */

```
static nodo_Lista* copia_nodo_Lista (  
    nodo_Lista* m, nodo_Lista* c, nodo_Lista* oa,  
    nodo_Lista* &nc, nodo_Lista* &a);  
nodo_lista* n = new nodo_lista;  
if (m == c) {n->ant = n; n->sig = n; nc = n; a = n;}  
else {  
    n->info = m->info;  
    n->sig = copia_nodo_Lista(m->sig, c, oact, nc, a);  
    (n->sig)->ant = n;  
    nc->sig = n;  
    n->ant = nc;  
    if (m == oact) a = n;  
}  
return n;  
}
```

```
// Métodos privados
// Borrar secuencia de nodos
/* Pre: c apunta a un centinela, m apunta a la misma
secuencia de nodos */
/* Post: libera el espacio ocupado por la cadena de
nodos entre m* y c*, ambos incluidos/

static void borra_nodo_lista(nodo_lista* m,
                             nodo_lista* c){
    if (m != c) {
        borra_nodo_lista(m->sig,c);
        delete m;
    }
}
```

Árboles binarios

La clase Arbol

- No son exactamente lo mismo que la clase BinTree de C++
- Principales Operaciones:
 - `a.plantar(x, a1, a2)`: `a` ha de estar vacío, y sea un objeto diferente de `a1` y `a2`. Después de la operación, `a1` y `a2` quedan vacíos
 - `a.hijos(a1, a2)`, `a1` y `a2` han de ser vacíos y, al final `a`, `a1` y `a2` serán objetos diferentes

Implementación de la clase Árbol

```
template <class T> class Arbol {  
    private:  
        struct nodo_arbol{  
            T info;  
            nodo_arbol* sigI;  
            nodo_arbol* sigD;  
        };  
        nodo_arbol* primer_nodo;  
  
    ... //operaciones privadas  
    public:  
    ... //operaciones públicas  
}
```

// Constructoras y destructoras

```
Arbol(){
```

```
    primer_nodo = nullptr;
```

```
}
```

```
Arbol(const Arbol& A){
```

```
    primer_nodo = copia_nodo_arbol(A.primer_nodo);
```

```
}
```

```
~Arbol(){
```

```
    borra_nodo_arbol(primer_nodo);
```

```
}
```

```
// Métodos privados
// Copiar jerarquía de nodos
static nodo_arbol* copia_nodo_arbol (nodo_arbol* m) {
/* Pre: true */
/* Post: Si m es nullptr, retorna nullptr, si no retorna
una copia de la jerarquía de nodos apuntada por m*/
if (m == nullptr) return nullptr;
else {
    nodo_arbol* n = new nodo_arbol;
    n->info = m->info;
    n->sigI = copia_nodo_arbol(m->sigI);
    n->sigD = copia_nodo_arbol(m->sigD);
    return n;
}
}
```

```
// Métodos privados
// Borrar jerarquia de nodos
/* Pre: true */
/* Post: libera el espacio ocupado por todos los
      nodos que cuelgan de m*/
```

```
static void borra_nodo_arbol(nodo_arbol* m){
    if (m != nullptr) {
        borra_nodo_arbol(m->sigI);
        borra_nodo_arbol(m->sigD);
        delete m;
    }
}
```

// Asignación

```
Arbol& operator=(const Arbol& A){  
    if (this != &A) {  
        borra_nodo_arbol(primer_nodo);  
        primer_nodo= copia_nodo_arbol(A.primer_nodo);  
    }  
    return *this;  
}
```

// Consultoras

```
bool es_vacio() const {  
    return primer_nodo == nullptr;  
}
```

```
T raiz() const {
```

// Pre: El arbol no está vacío

```
    return primer_nodo->info;  
}
```

```
// Modificadoras
```

```
/* Pre: El p.i. está vacío, a1 = A1, a2 = A2, a1 y a2  
son objetos diferentes del p.i. */
```

```
/* Post: El parámetro implícito tiene x en la raíz, A1  
como hijo izquierdo, A2 como hijo derecho, y a1 y a2  
están vacíos */
```

```
void plantar(const T& x, Arbol &a1, Arbol &a2){  
    nodo_arbol * aux = new nodo_arbol;  
    aux->info = x;  
    aux->sigI = a1.primer_nodo;  
    if (a2.primer_nodo != a1.primer_nodo or  
        a2.primer_nodo == nullptr)  
        aux->sigD = a2.primer_nodo;  
    else aux->sigD = copia_nodo_arbol (a2.primer_nodo)  
    primer_nodo = aux;  
    a1.primer_nodo = nullptr; a2.primer_nodo = nullptr  
}
```

```
/* Pre: Si el parámetro de interés es A, A no está  
vacío, hi y hd están vacíos y son objetos  
diferentes */
```

```
/* Post: hi es el hijo izqdo de A, hd es el hijo  
dcho de A, el p.i. está vacío */
```

```
void hijos(Arbol &hi, Arbol &hd){  
    nodo_lista * aux = primer_nodo;  
    hi.primer_nodo = primer_nodo->sigI;  
    hd.primer_nodo = primer_nodo->sigE;  
    primer_nodo = nullptr;  
    delete aux;  
}
```


Árboles N-arios

La clase ArbolNario

Generalización de los árboles binarios en que cada nodo tiene exactamente N hijos

Implementación de la clase ArbolNario

```
template <class T> class ArbolNario {  
    private:  
        struct nodo_arbolNario{  
            T info;  
            vector<nodo_arbolNario*> sig;  
        };  
        int N;  
        nodo_arbolNario* primer_nodo;  
  
    ... //operaciones privadas  
    public:  
    ... //operaciones públicas  
}
```

```

// Métodos privados
// Copiar jerarquía de nodos
static nodo_arbolNario* copia_nodo_arbol (nodo_arbolNario* m) {
/* Pre: true */
/* Post: Si m es nullptr, retorna nullptr, si no retorna una
copia de la jerarquía de nodos apuntada por m*/
    if (m == nullptr) return nullptr;
    else {
        nodo_arbolNario* n = new nodo_arbolNario;
        n->info = m->info;
        int N = m->sig.size();
        n->sig = vector<nodo_arbolNario*>(N);
        for (int i = 0; i < N; ++i)
            n->sig[i] = copia_nodo_arbolNario(m->sig[i]);
        return n;
    }
}

```

```
// Métodos privados
// Borrar jerarquia de nodos
/* Pre: true */
/* Post: libera el espacio ocupado por todos los nodos
que cuelgan de m*/
```

```
static void borra_nodo_arbolNario(nodo_arbolNario* m){
    if (m != nullptr) {
        int N = m->sig.size();
        for (int i = 0; i < N; ++i)
            borra_nodo_arbolNario(m->sig[i]);
        delete m;
    }
}
```

// Constructoras y destructoras

```
ArbolNario(int n){  
    N = n;  
    primer_nodo = nullptr;  
}  
ArbolNario(const ArbolNario& A){  
    N = A.N;  
    primer_nodo = copia_nodo_arbolNario(A.primer_nodo);  
}  
~ArbolNario(){  
    borra_nodo_arbolNario(primer_nodo);  
}
```

```
ArbolNario(const T &x, int n){  
    /* Pre: true */  
    /* Post: el p.i. es un árbol con x en la raíz y n hijos vacíos */  
    N = n;  
    primer_nodo = new nodo_arbolNario;  
    primer_nodo->info = x;  
    for (int i = 0; i < N; ++i)  
        primer_nodo->sig[i] = nullptr;  
}
```

// Asignación

```
ArbolNario& operator=(const ArbolNario& A){  
    if (this != &A) {  
        borra_nodo_arbolNario(primer_nodo);  
        N = L.N;  
        primer_nodo= copia_nodo_arbolNario(A.primer_nodo);  
    }  
    return *this;  
}
```


// Consultoras

```
bool es_vacio() const {  
    return primer_nodo == nullptr;  
}
```

```
T raiz() const {
```

```
// Pre: El arbol no está vacío
```

```
    return primer_nodo->info;  
}
```

```
int aridad() const {  
    return N;  
}
```

```
// Modificadoras
```

```
/* Pre: El p.i. está vacío, v = V, v.size() es la aridad  
del p.i., todas las componentes de v tienen la misma  
aridad que el p.i. y ninguna de ellas es el p.i.*/
```

```
/* Post: El parámetro implícito tiene x en la raíz, sus  
hijos son las componentes de v, v contiene árboles  
vacíos*/
```

```
void plantar(const T& x, vector<ArbolNario> &v){  
    primer_nodo= new nodo_arbolNario;  
    primer_nodo ->info = x;  
    primer_nodo ->sig = vector<nodo_arbolNario*>(N);  
    for (int i = 0; i < N; ++i){  
        primer_nodo ->sig[i] = v[i].primer_nodo;  
        v[i].primer_nodo = nullptr;  
    }  
}
```

/* Pre: Si el parámetro de interés es A, A no está vacío, v es un vector vacío */

/* Post: v contiene los hijos de A, el P.i. está vacío */

```
void hijos(vector<ArbolNario> &v){  
    v = vector<ArbolNario>(N,ArbolNario(N));  
    for (int i = 0; i < N; ++i){  
        v[i].primer_nodo = primer_nodo->sig[i];  
        delete primer_nodo;  
        primer_nodo = nullptr;  
    }
```

```
/* Pre: El parámetro de interés está vacío y es de  
la misma aridad que a, a no está vacío, i está  
entre 1 y el número de hijos de a */  
/* Post: el p.i. es una copia del hijo i-ésimo de a */
```

```
void hijo(const ArbolNario &a, int i){  
    primer_nodo =  
        copia_nodo_arbolNario(a.primer_nodo->sig[i-1]);  
}
```

Eficiencia de recorridos

La operación `hijos` nos permite hacer recorridos eficientes, ya que nos obtiene de golpe todos los hijos de un árbol

//Suma de los elementos de un árbol

/* Pre: a = A */

/* Post: el resultado es la suma de los elementos de A
*/

```
int suma(ArbolNario <int> &a){  
    if (a.es_vacio()) return 0;  
    else {  
        int s = a.raiz();  
        int N = a.aridad();  
        vector<ArbolNario <int>> v(N);  
        a.hijos(v);  
        for (int i = 0; i < N; ++i) s = s+suma(v[i]);  
        return s;  
    }  
}
```

```
//Suma de los elementos de un árbol
/* Pre: a = A */
/* Post: el resultado es la suma de los elementos de A
*/
```

```
int suma(ArbolNario <int> &a){
    if (a.es_vacio()) return 0;
    else {
        int x = a.raiz(); int s = x;
        int N = a.aridad();
        vector<ArbolNario <int>> v(N);
        a.hijos(v);
        for (int i = 0; i < N; ++i) s = s+suma(v[i]);
        a.plantar(x,v);
        return s;
    }
}
```

```
//Sumar un valor k a cada elemento de un árbol
/* Pre: a = A */
/* Post: a es como A, pero habiendo sumado k a todos
sus elementos */
void suma_k(ArbolNario <int> &a, int k){
    if (not a.es_vacio) {
        int s = a.raiz()+k;
        int N = a.aridad();
        vector<ArbolNario <int>> v(N);
        a.hijos(v);
        for (int i = 0; i < N; ++i) suma_k(v[i],k);
        a.plantar(s,v);
    }
}
```


Árboles generales

Árboles generales

- Cada nodo tiene un número indeterminado de hijos, no necesariamente el mismo
- Un árbol general:
 - es un árbol vacío o
 - tiene cualquier número de hijos (incluido 0), ninguno de los cuales está vacío

Tipos de implementaciones

- los hijos son un vector de punteros
 - consultar hijo i -ésimo es eficiente
 - eliminar hijo i -ésimo puede ser ineficiente
- los hijos son una lista de punteros
 - consultar hijo i -ésimo es ineficiente
 - pero recorridos secuenciales son eficientes
 - eliminar hijo actual es eficiente
- árbol binario, hijo izquierdo: primer hijo; hijo derecho: siguiente hermano

Implementación de la clase ArbolGen

```
template <class T> class ArbolGen {  
    private:  
        struct nodo_arbolGen{  
            T info;  
            vector<nodo_arbolGen*> sig;  
        };  
        nodo_arbolGen* primer_nodo;  
  
    ... //operaciones privadas  
    public:  
    ... //operaciones públicas  
}
```

// Cópia y borrado de árboles generales

Idénticas (salvo las declaraciones de los tipos) a las de los árboles n-arios

// Constructoras y destructoras

```
ArbolGen(){  
    primer_nodo = nullptr;  
}  
ArbolGen(const ArbolGen& A){  
    primer_nodo = copia_nodo_arbolGen(A.primer_nodo);  
}  
~ ArbolGen(){  
    borra_nodo_arbolNario(primer_nodo);  
}
```

```
ArbolGen(const T &x){  
    /* Pre: true */  
    /* Post: el p.i. es un árbol general con x en la raiz y 0  
    hijos */  
    primer_nodo = new nodo_arbolGen;  
    primer_nodo->info = x;  
}
```

// Asignación

```
ArbolGen& operator=(const ArbolGen& A){  
    if (this != &A) {  
        borra_nodo_arbolGen(primer_nodo);  
        N = L.N;  
        primer_nodo= copia_nodo_arbolGen(A.primer_nodo);  
    }  
    return *this;  
}
```


// Consultoras

```
bool es_vacio() const {  
    return primer_nodo == nullptr;  
}
```

```
T raiz() const {  
// Pre: El arbol no está vacío  
    return primer_nodo->info;  
}
```

```
int num_hijos() const {  
    return (primer_nodo->sig).size();  
}
```

```
// Modificadoras
```

```
/* Pre: El p.i. está vacío */
```

```
/* Post: El parámetro implícito tiene x en la raíz y cero  
hijos*/
```

```
void plantar(const T& x){  
    primer_nodo= new nodo_arbolGen;  
    primer_nodo ->info = x;  
}  
}
```

// Modificadoras

/* Pre: El p.i. está vacío, $v = V$, y ninguna componente de v está vacía */

/* Post: El parámetro implícito tiene x en la raíz, sus hijos son las componentes de v , v contiene árboles vacíos*/

```
void plantar(const T& x, vector<ArbolGen> &v){
    primer_nodo= new nodo_arbolNario;
    primer_nodo->info = x;
    int n = v.size();
    primer_nodo->sig = vector<nodo_arbolGen*>(n);
    for (int i = 0; i < N; ++i){
        primer_nodo->sig[i] = v[i].primer_nodo;
        v[i].primer_nodo = nullptr;
    }
}
```

```
// Modificadoras
```

```
/* Pre: El p.i. y a no están vacíos, a y el p.i. no son el mismo objeto */
```

```
/* Post: El parámetro implícito tiene un hijo más, este hijo es el último y es una copia de a*/
```

```
void poner_hijo(const ArbolGen &a){  
    (primer_nodo->sig).push_back(copia_nodo_arbolGen(a.primernodo))  
}
```

```
/* Pre: Si el parámetro de interés es A, A no está  
vacío, y no es ninguno de los componentes de v */  
/* Post: v contiene los hijos de A, el P.i. está  
vacío */
```

```
void hijos(vector<ArbolGen> &v){  
    int n = (primer_nodo->sig).size();  
    v = vector<ArbolGen>(n);  
    for (int i = 0; i < N; ++i){  
        v[i].primer_nodo = primer_nodo->sig[i];  
        delete primer_nodo;  
        primer_nodo = nullptr;  
    }
```

```
/* Pre: El parámetro de interés está vacío, a no está  
vacío, i está entre 1 y el número de hijos de a */  
/* Post: el p.i. es una copia del hijo i-ésimo de a */
```

```
void hijo(const ArbolGen &a, int i){  
    primer_nodo =  
        copia_nodo_arbolGen(a.primer_nodo->sig[i-1]);  
}
```

Operaciones no primitivas

Ejemplos

- Búsqueda en una pila o en una cola
- Sumar un valor k a todos los elementos de un árbol
- Substitución de hojas por un árbol
- Invertir una lista

Operaciones no primitivas

Dos alternativas para definir una operación no primitiva:

1. Definición fuera de la clase
2. añadirla a la clase

Ventajas e inconvenientes:

1. Modularidad, quizá menor eficiencia, quizá menos simple
2. No modularidad, quizá mas eficiencia, quizá más simple

Estructuras multienlazadas

Multilistas

- Queremos guardar una tabla muy grande pero muy dispersa (con muchos elementos nulos). Se necesita:
 - Dado el índice de una fila, recuperar todos los elementos no nulos de la fila
 - Dado el índice de una columna, recuperar todos los elementos no nulos de la columna
- Por ejemplo, una tabla que nos diga, a cada estudiante y cada asignatura, cuál es la nota que ha sacado ese estudiante en esa asignatura, si es que está matriculado

```
class multi_lista_est {  
    private:  
    struct nodo_ML{  
        double nota;  
        nodo_ML* sig_est;  
        nodo_ML* sig_asig;  
        int pos_est;  
        int pos_asig;  
    };  
    struct info_asig{  
        string asignatura;  
        nodo_ML* primer_est;  
    };  
    struct info_est{  
        int dni;  
        nodo_ML* primera_asig;  
    };  
    vector <info_asig> vasig; // vector ordenado  
    vector <info_est> vest; // vector ordenado
```

Operaciones descritas en los apuntes

Colas ordenadas

Implementación de colas ordenadas

```
template <class T> class ColaOrd {  
    private:  
        struct nodo_colaOrd{  
            T info;  
            nodo_colaOrd* sig_cron;  
            nodo_colaOrd* sig_ord;  
        };  
        int longitud;  
        nodo_colaOrd* primero;  
        nodo_colaOrd* ultimo;  
        nodo_colaOrd* menor;  
        ... //operaciones privadas  
    public:  
        ... //operaciones públicas  
}
```

```
/* Pre: cierto */
```

```
/* Post: el p.i. queda modificado, añadiendo x como  
último elemento cronológicamente y en el sitio que  
le toque en el orden creciente */
```

```
void pedir_turno(const T& x){  
    nodo colaOrd *n = new nodo cola;  
    n->info = x; n->sig_cron = nullptr;  
    ++longitud;  
    if (primero == nullptr) {  
        primero = n; ultimo = n;  
        menor = n; n->sig_ord = nullptr;  
    }  
    else {  
        ultimo->sig_cron = n; ultimo = n;  
        ...  
    }  
}
```



```

... /* actualización del orden creciente*/
if (x < menor->info) {
    n->sig_ord = menor; menor = n;
}
else {
    nodoColaOrd *ant = menor;
    bool encontrado = false;
    while (ant->sig_ord != nullptr
           and not encontrado) {
        if (x < (ant->sig_ord)->info) encontrado = true;
        else ant = ant->sig_ord;
    }
    n->sig_ord = ant->sig_ord; ant->sig_ord = n;
}
}
}

```

```
/* Pre: cierto */
```

```
/* Post: el p.i. queda modificado, añadiéndole todos  
los elementos de c después del último y modificando  
los enlaces de orden adecuadamente */
```

```
void concatenar(ColaOrd & c){  
    if (c.primerio != nullptr) {  
        longitud = longitud + c.longitud;  
        if (primerio == nullptr) {  
            primerio = c.primerio; menor = c.menor;  
            ultimo = c.ultimo;  
        }  
        else {  
            ultimo->sig_cron = c.primerio;  
            ultimo = c.ultimo;  
            ...  
        }  
    }  
}
```

```
/* ordenamos los elementos de la cola */
    nodoColaOrd *ant, *act1, *act2;
/* Inv: los nodos ordenados llegan hasta ant */
act1 y act2 apuntan al primer nodo no ordenado de cada
cola */

    act1 = menor; act2 = c.menor;
    if (act2->info < act1->info){
        menor = act2; ant = act2;
        act2 = act2->sig_ord;
    }
    else {
        ant = act1;
        act1 = act1->sig_ord;
    }
    ...
```

```
/* Inv: los nodos ordenados llegan hasta ant  
act1 y act2 apuntan al primer nodo no ordenado de cada  
cola */
```

```
    while (act1!=nullptr and act2!=nullptr){  
        if (act2->info < act1->info){  
            ant->sig_ord = act2; ant = act2;  
            act2 = act2->sig_ord;  
        }  
        else { ant->sig_ord = act1; ant = act1;  
              act1 = act1->sig_ord;  
            }  
    if (act1 != nullptr) ant->sig_ord = act1;  
    else ant->sig_ord = act2;  
    c.primerio = c.ultimo = c.menor = nullptr;  
    c2.longitud = 0;  
}  
}
```

Heaps y colas con prioridad

Colas con prioridad

- Las colas con prioridad son como las colas ordenadas, pero el orden de llegada no importa:
 - Cada elemento de la cola contiene un valor (su prioridad)
 - Las prioridades son enteros estrictamente positivos.
 - Por simplicidad, supondremos que todos los elementos de una cola tienen distinta prioridad
 - si e_1 y e_2 tienen prioridades p_1 y p_2 , y $p_1 < p_2$, e_2 saldrá antes de la cola.
- Las colas con prioridad, se pueden implementar como colas ordenadas, pero hay una implementación más eficiente utilizando *heaps*

Heaps

- Los heaps son árboles binarios que cumplen las siguientes propiedades:
 - Cada valor en un nodo de un heap es mayor que los valores de todos sus descendientes.
 - Si la altura de un heap es n y el heap tiene m ramas, entonces las k primeras ramas ($k \leq m$) tienen longitud n y las restantes tienen longitud $n-1$.

Implementación de heaps

- La forma más simple de implementar un heap es usando un vector, donde:
 - la raíz del heap está en la posición 0 del vector .
 - si un nodo del heap ocupa la posición i del vector, su hijo izquierdo está en la posición $2*i+1$ y su hijo derecho en la siguiente (la $2*(i+1)$).
 - como consecuencia, si un nodo del heap ocupa la posición i del vector, su padre (si no es la raíz) estará en la posición $(i-1)/2$.

Implementación de Heaps

```
template <class T> class Heap {  
    /*Se supone que hay una relación de orden  
    definida sobre T */  
  
    private:  
        vector <T> h;  
        int sl; /* sl es la primera posición de H no  
                ocupada */  
  
    public:  
        ...  
        void entrar(const T& x);  
        T primero();  
        void salir_primero();  
        ...  
}
```

```
/* Pre: sl <= v.size(), v reresenta un heap */  
/* Post: Se añade x al heap de tal manera que sigue  
cumpliendo sus propiedades */  
void entrar(const T& x){  
    if (sl == v.size()) v.push_back(x);  
    else h[sl] = x;  
    int nuevo = sl; ++sl;  
    int padre = (nuevo-1)/2;  
    while (padre >= 0 and h[nuevo] > h[padre]) {  
        swap(h[padre], h[nuevo]);  
        nuevo = padre;  
        padre = (nuevo-1)/2;  
    }  
}
```

```

/* Pre: sl > 0, v representa a un heap */
/* Post: Se elimina la raiz del heap de tal manera que siga
cumpliendo sus propiedades */
void salir_primero(){
    if (sl == 1) sl = 0;
    else {
        --sl; h[0] = h[sl];
        int n = 0; bool fin = false;
        while (2*n+1<sl and not fin) {
            maxhijo = 2*n+1;
            if (maxhijo+1<sl and h[maxhijo+1]> h[maxnhijo])
                ++maxhijo;
            if (h[n]>h[maxhijo]) fin = true;
            else { swap(h[maxhijo], h[n]); n=maxhijo;}
        }
    }
}

```

```
/* Pre:  $sl > 0$ ,  $v$  representa a un heap */  
/* Post: retorna el mayor elemento del heap*/  
T primero(){  
    return v[0];  
}
```

¡Que os vaya bien!