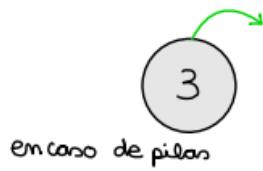


EXAMEN FINAL:

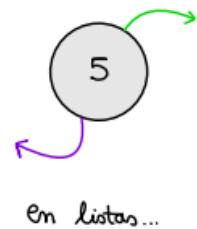
▼ NODOS: Es una de los elementos de una estructura de datos.

- ES UN STRUCT:



```
struct node_pila {  
    T info;  
    node_pila* seguent;  
};
```

```
struct node_llista {  
    T info;  
    node_llista* seg;  
    node_llista* ant;  
};
```



▼ PUNTEROS = A UNA @ DE MEMORIA

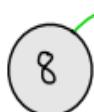
- Se representa con una flecha.
- Puede apuntar a un NODO o a NULL.

DECLARACIÓN: mode_pila* m ESTO CREA UN PUNTERO, no un NODO.

CREAR UN NODO NUEVO: (Reservar espacio de memoria)

mode_pila* m = NEW mode_pila;

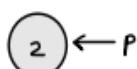
Reserva espacio de memoria dinámica y retorna un puntero a la memoria reservada. Ahora ya podemos modificar los campos del struct



NULL

m → info = 8;
m → seguent = NULL;

- Asignándole un puntero q:
 $p = q;$
- Asignandole la dirección de un objeto existente:
 $p = \&x;$
- Creando una posición nueva y asignándosela a p:
 $p = \text{new int};$
- Asignándole nullptr:
 $p = \text{nullptr};$



BORRAR UN NODO:

Para poder borrar un nodo, debemos tener un puntero apuntando a este.

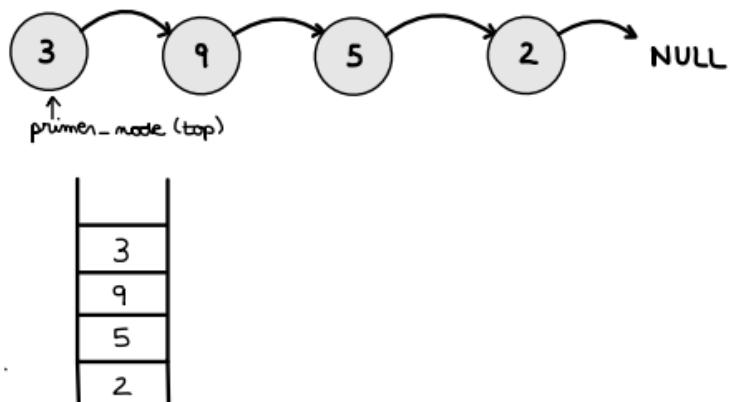
- Usamos el comando **DELETE** + puntero (al nodo):
una vez borrado el nodo, el puntero pasa a apuntar a NULL.
 $p = \text{new int};$
 $q = \&x;$
 $\text{delete } p; //OK$
 $\text{delete } q; //ERROR$

■ PILAS:

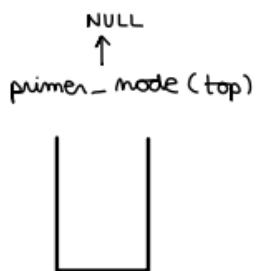
private:

```
struct node_pila {
    T info;
    node_pila* seguent;
};
```

```
int altura;
node_pila* primer_node;
```



▼ PILA VACÍA:



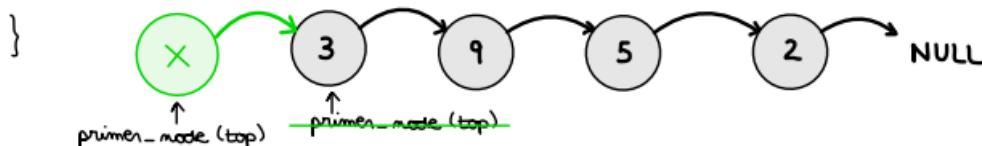
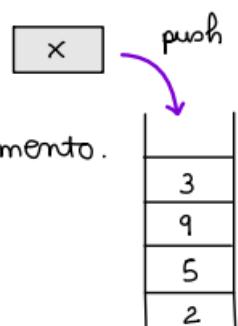
```
Pila () {
    // Pre: cert
    // Post: El resultado es una pila sense cap element;
    altura = 0;
    primer_node = NULL;
}
```

puntero a null

▼ PUSH PILA (NUEVO ELEMENTO):

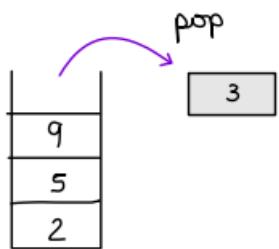
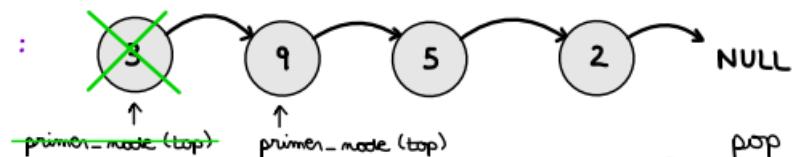
```
// Pre: cierto
// Post: La pila es comola original añadiendo x como ultimo elemento.

void empilar (const T& x) {
    node_pila* aux; // crea un puntero "aux"
    aux = new node_pila; // crea un nodo (reserva de espacio)
    aux->info = x; // aux es un nodo con el nuevo elemento.
    aux->seguent = primer_node; // puntero siguiente ahora apunta al "primer".
    primer_node = aux;
    ++altura;
}
```



▼ POP (BORRAR ELEMENTO):

```
void desempilar () {
    node_pila* aux = primer; // copia del 1n nodo
    primer = primer->seg; // primer ahora es = que el siguiente
    delete aux; // borramos aux que ahora era "primer"
    --altura;
}
```



```
/* 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->sig = copia_nodo_pila(m->sig);  
        return n;  
    }  
}
```

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

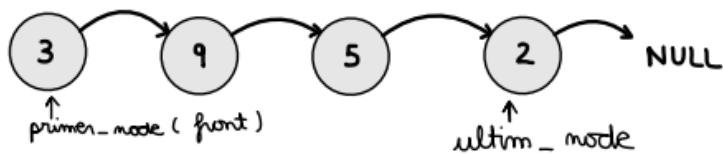
```
static void borra_nodo_pila(nodo_pila* m){  
    if (m != nullptr) { // La asignación de pilas  
        borra_nodo_pila(m->sig);  
        delete m;  
    }  
}  
  
T top() const {  
    // Pre: la pila no está vacía  
    return primero->info; }  
bool empty() const {  
    return altura == 0; }  
  
int size() const {  
    return altura; }  
  
stack& operator=(const stack& S){  
    if (this != &S) {  
        altura = S.altura;  
        borra_nodo_pila(primer);  
        primero = copia_nodo_pila(S.primer);  
    }  
    return *this; }  
void clear(){  
    borra_nodo_pila(primer);  
    altura = 0;  
    primero = nullptr; }  
bool búsqueda (const T& x) const{  
    nodo_pila* act = primer;  
    while (act != nullptr){  
        if (act->info == x) return true;  
        act = act->sig; }  
    return false; }
```

COLAS:

private:

```
struct node_cua {
    T info;
    node_cua* seguent;
};
```

```
int longitud;
node_cua* primer_node;
node_cua* ultim_node;
```



Definición de un NODO
de la cola



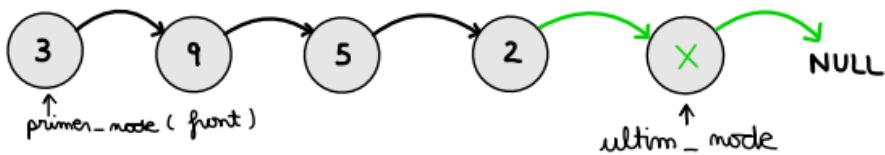
Atributos privados

COLA VACÍA: // Pre: Ciert
// Post: El resultat és una cua
sense elements.

Cua () {

```
longitud = 0;
primer_node = NULL;
ultim_node = NULL;
```

PUSH (AÑADIR NUEVO ELEMENTO): (se añade al final)



```
void demandar_torn (const T & x) {
    node_cua* aux;
    aux = new node_cua; // reserva de espacio para el nuevo elemento.
    aux -> info = x;
    aux -> seguent = NULL;
    if (primer_node == NULL) primer_node = aux; → si
    else ultim_node -> seguent = aux; → en otro caso,
    ultim_node = aux; // el ultimo nodo es el nuevo.
    ++longitud;
}
```



cola vacía , primer_node = al aux "nuevo".
el siguiente del ultimo nodo , apunta a aux.

POP (BORRAR ELEMENTO) (se borra del frente)

```
void avanzar () {
    node_cua* aux;
    aux = primer_node;
    if (primer_node -> seguent == NULL) {
        primer_node = NULL;
        ultim_node = NULL;
    }
    else primer_node = primer_node -> seguent; // avanza
    delete aux;
    --longitud;
}
```

```

T front() const {
    // Pre: la cola no está vacía
    return primero->info;
}

bool empty() const {
    return longitud == 0;
}

int size() const {
    return longitud;
}

/* 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->sig = copia_nodo_cola(m->sig,u);
    if (n->sig == nullptr) u = n;
    return n;
}

/* 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->sig);
        delete m;           // La asignación
    }
}

queue& operator=(const queue& Q){
    if (this != &Q) {
        longitud = Q.longitud;
        borra_nodo_cola(primer);
        primero = copia_nodo_cola(Q.primer, ultimo);
    }
    return *this;
}

```

LISTAS: act = punt d'interès (apunta a nul·l o a qualsevol node de llista)

```
template <class T> class Lista {
```

private:

```
struct node_llista {
```

T info;

```
node_llista* seg;
```

```
node llista* ant;
```

1

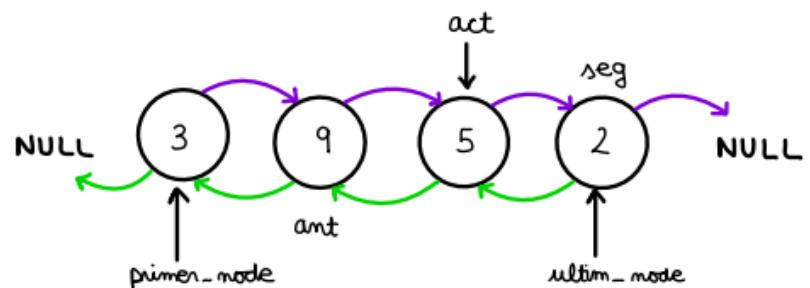
```
int longitud;
```

```
node llista* primer_node;
```

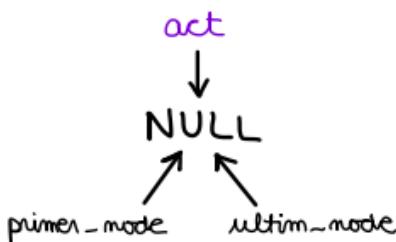
node *lista *ultim_node;

node_llista* act;

1



► LISTA VACÍA:



```
Llista( ) {  
    longitud = 0;  
    primer_node = NULL;  
    ultim_node = NULL;  
    act = NULL;  
}
```

▲ AÑADIR ELEMENTOS:

el elemento x se añade a la izquierda del punto de interés.

1. `act = prime` (añadimos elemento , al inicio)
2. `act = NULL` (añadimos uno al final)

- lista vacía
 - lista no vacía

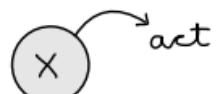
```
void aegin( const T & x ) {
```

```

mode_llista * aux;
aux = new mode_llista;
aux->info = x;
aux->seg = act // el siguiente sera NULL

```



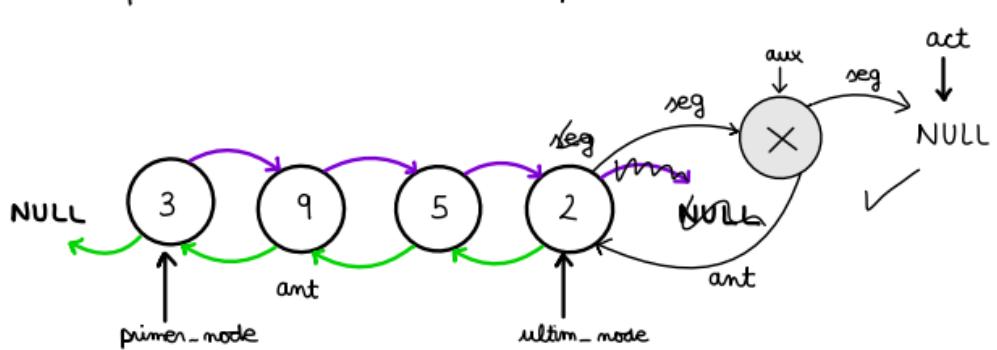
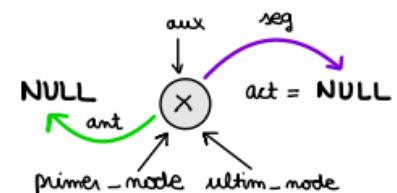
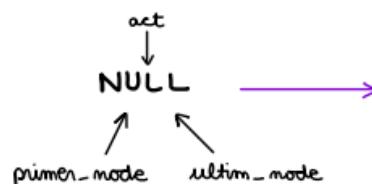


ACT == NULL (VACÍA):

```

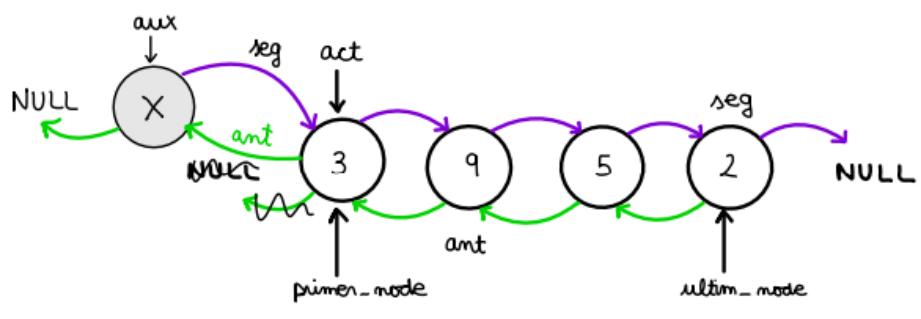
if ( primer == NULL ) {
    aux → ant = NULL ;
    primer = aux ;
    ultim = aux ;
}
else if ( act == null ) {
    ultim → seg = aux ;
    aux → ant = ultim ;
}
ultim = aux ;

```



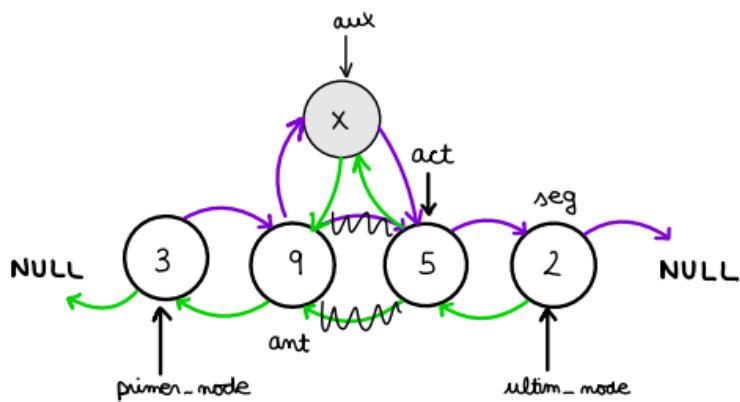
□ ACT == PRIMER :

```
else if ( act == primer ) {
    aux → ant = NULL;
    primer → ant = aux;
    primer = aux;
}
```



□ ACT == OTRO :

```
else {
    act → ant → seg = aux;
    aux → ant = act → ant;
    act → ant = aux;
}
++ longitud;
```



▼ ELIMINAR ELEMENTO :

lista no vacía i su punto de interés no
esta a la derecha del todo.

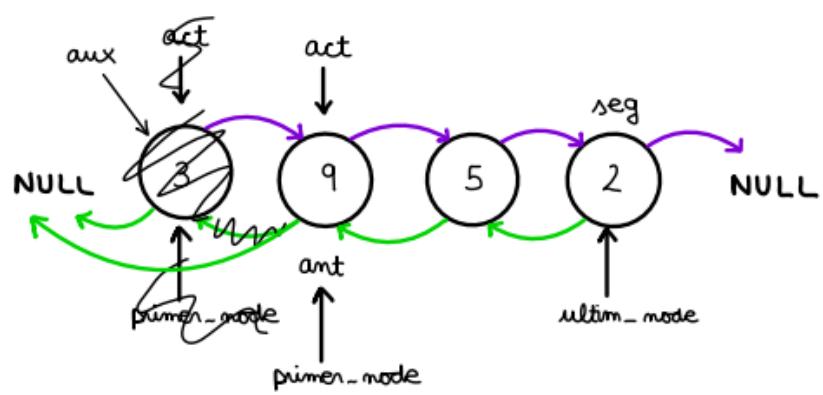
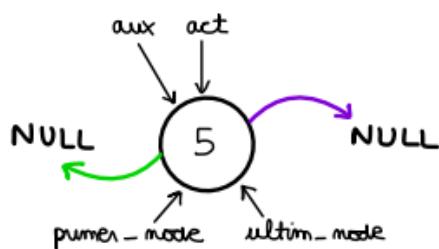
CASOS:

1. longitud == 1 (solo hay un nodo).
2. act == primer
3. act == ultimo
4. act == otro

```
void eliminar () {
    mode = lista * aux;
    aux = act; // conserva el acceso al nodo actual
```

□ SOLO HAY UN NODO :

```
if ( longitud == 1 ) {
    delete aux;
    primer_mode = NULL;
    ultimo_mode = NULL;
    act = NULL;
}
```



□ ACT == PRIMER :

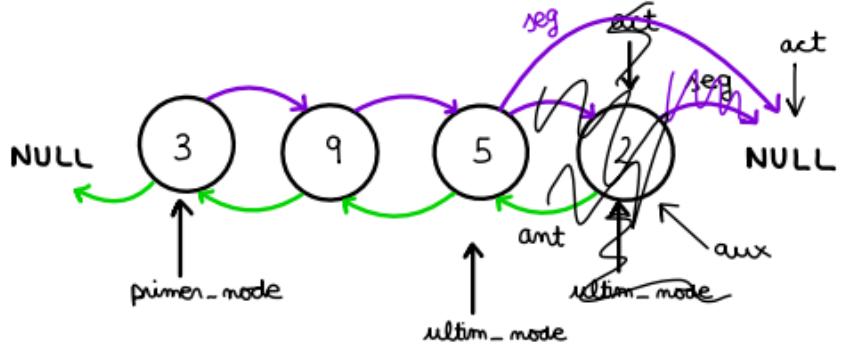
```
else if ( act == primer ) {
    primer = primer → seg;
    act = primer;
    act → ant = NULL;
    delete aux;
}
```

ACT == ULTIM :

```

else if (act == ultim) {
    act = NULL;
    ult = ult->ant;
    ult->seg = NULL;
    delete aux;
}

```

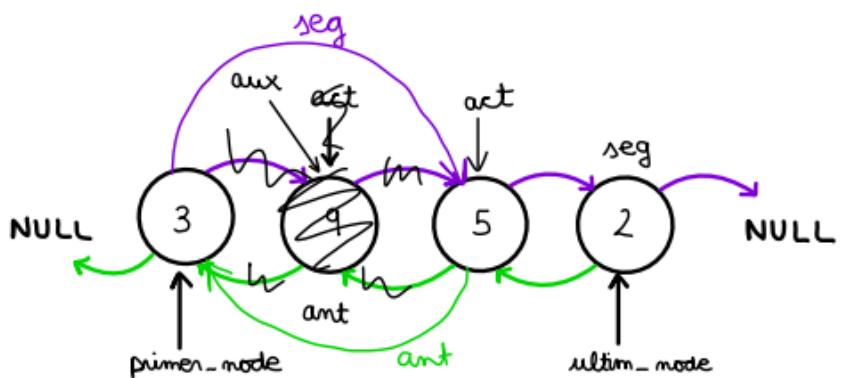


ACT == OTRO :

```

else {
    act = aux->seg;
    aux->ant->seg = act;
    act->ant = aux->ant;
    delete aux;
}

```



-- longitud;

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

```

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;
}

```

// modificación y movimiento del punto de interés
/ 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;
}

```

```

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

```

```

bool al_principio() const {
    return act == primero;
}

```

```

void l_vacia(){
    borra_nodo_lista(primer);
    longitud = 0;
    primero = nullptr;
    ultimo = nullptr;
    act = nullptr;
}

```

```

/* 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;
}

/* Post: Se ha movido el punto de interés una posición hacia
   el principio */
void retrocede(){
    if (act == nullptr) act = ultimo;
    else act = act->ant;
}

/* 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.primero; ultimo = L.ultimo;
            longitud = L.longitud;
        } else {
            ultimo->sig = L.primero;
            (L.primero)->ant = ultimo; ultimo = L.ultimo;
            longitud = longitud + L.longitud;
        }
        L.primero = L.ultimo = L.act = nullptr;
        L.longitud = 0;
    }
    act = primero;
}

```

```

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->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;
    } // 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;
    }
}

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

```

```

void elimina_par_0() {
    act = primer;
    nodo * p = act->seg;
    while (p != nullptr) {
        if ((act->info + p->info) == 0)
            elimina2();
        if (act == nullptr)
            p = nullptr;
        else if (act == primer)
            p = act->sig;
        else {
            p = act;
            act = p->ant;
        }
    }
}

```

```

void elim_reps_cons() {
    nodo * antact = primero;
    act = antact->seg;
    while (act != nullptr) {
        nodo * segact = act->seg;
        if (antact->info == act->info) {
            antact->seg = segact;
            segact->ant = antact;
            act = segact;
        } else {
            antact = act;
            act = antact->seg;
        }
    }
    ultimo_node = antact;
}

```

```

void invertir(){
    if (longitud > 1) {
        nodo_lista* n = primero;
        /* Inv: se han invertido los punteros de todos los nodos
        anteriores a n */
        while (n != nullptr) {
            swap(n->sig, n->ant);
            n = n->ant;
        }
        swap(primer, ultimo);
    }
}

```

```

void marca_cambios_sigue() {
    nodo * antact = primero;
    act = antact->seg;
    while (act != nullptr) {
        if ((antact->info > 0 and act->info < 0) or
            (antact->info < 0 and act->info > 0)) {
            nodo * aux = new nodo;
            aux->info = 0;
            aux->seg = act;
            aux->ant = antact;
            antact->seg = aux;
            act->ant = aux;
        }
        antact = act;
        act = antact->seg;
    }
}

```

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  
}  
  
/* Consultoras para saber dónde está el punto  
de interés */  
  
bool al_final() const {  
    return act == cent;  
}  
  
bool al_principio() const {  
    return act == cent->sig;  
}
```

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

```

/* 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;
}

```

```

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, oa, nc, a);
    (n->sig)->ant = n;
    nc->sig = n;
    n->ant = nc;
    if (m == oa) a = n;
}
return n; /* 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;
    }
}

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

```

```

template <class T> class Arbol { /* Post: El parámetro implícito tiene x en la raiz, A1
    como hijo izquierdo, A2 como hijo derecho, y a1 y a2
    están vacíos */
private:
    struct nodo_arbol{
        T info;
        nodo_arbol* sigI;
        nodo_arbol* sigD;
    };
    nodo_arbol* primer_nodo;

    ... //operaciones privadas
public:
    ... //operaciones públicas
}
bool es_vacio() const {
    return primer_nodo == nullptr;
}

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

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

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

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
} /* 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->sigD;
    primer_nodo = nullptr; // Copiar jerarquía de nodos
    delete aux;
    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;
        }
    }
}

```

```

/* Post: Hemos sumado k al valor de todos los nodos
que cuelgan de n del arbol binario*/
static void inc_nodo (nodo_arbol* n, int k){
    if (n!=nullptr){
        n->info = n->info+k;
        inc_nodo(n->sigI,k);
        inc_nodo(n->sigD,k);
    }
}

```

La clase ArbolNario

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

```

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
}

```

```

/* Post: Hemos sumado k al valor de cada nodo del arbol
binario*/
void sumak (int k){
    inc_nodo(primer_nodo,k)
}

/* Post: Hemos substituido cada hoja, con el valor x, por
el arbol a*/
void subst (const T& x, const Arbol <T>& a){
    primer_nodo = subst_n(primer_nodo, x, a);
}

static nodo_arbol subst_n(nodo_arbol n, const T& x, const
Arbol <T>& a){
    if (n==nullptr) return nullptr;
    if ((n->info == x) and n->sigI == nullptr and
        n->sigD == nullptr ){
        delete n;
        n = copia_nodo_arbol(a->primer_nodo);
    }else{
        subst_nodo (n->sigI, x, a);
        subst_nodo (n->sigD, x, a);
    }
}

```

```

ArbolNario(const T &x, int n){
/* Pre: true */
/* Post: el p.i. es un árbol con x en la raiz 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;
}

bool es_vacio() const {
    return primer_nodo == nullptr; Pre: El arbol no está vacío
}
T raiz() const {
    return primer_nodo->info;
}

int aridad() const {
    return N;
}

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;
    }
}

/* Post: El parámetro implícito tiene x en la raiz, 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;
    }
}

/* 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;
    }
}

/* 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]);
}

```

```
/* 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;  
    }  
}
```

```
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;  
}
```

```
static bool i_mult (nodo *p,  
                    if(es_pilla(p)){  
                        sum = p->info;  
                        num = 1;  
                        return true;  
                    }  
                    else {  
                        bool b = true;  
                        sum = p->info;  
                        num = 1;  
                        for(int i=0; i < filas.size() and b; ++i){  
                            if (p->filas[i] != nullptr){  
                                int s, n;  
                                b = i_mult (p->filas[i], s, n);  
                                sum += s;  
                                num += n;  
                            }  
                        }  
                        if (!b) return sum % num == 0;  
                        else return false;  
                    }  
    }
```

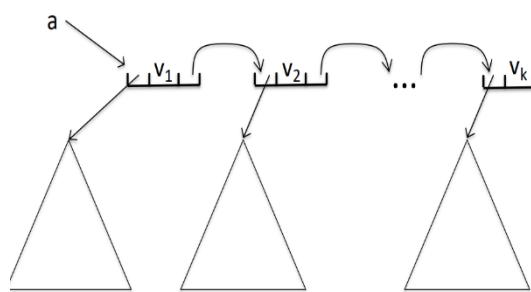
```
/* Post: retorna un vector v que cumple que v.size() es la altura del p.i., v[j] = número de nodos del p.i. en el nivel j, 0 ≤ j < v.size() */
```

```
vector <int> oc_niveles () const{  
    vector <int> v;  
    i_oc_niveles (primer_nodo,0,v);  
    return v;  
}  
static void i_oc_niveles (nodo_arbolNario* p, int i,  
                         vector <int>& v) const {  
    if (p != nullptr) {  
        if (i==v.size) v.pushback(1);  
        else ++v[i];  
        for (int d = 0; d < N; ++d) {  
            i_oc_niveles (p->sig[d],i+1,v)  
        }  
    }  
}
```

```
static bool es_pilla(nodo *p){  
    if (p == nullptr) return false;  
    else {  
        for (int i=0; i < N; ++i){  
            if (p->filas[i] != nullptr)  
                return false;  
        }  
        return true;  
    }  
}
```

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

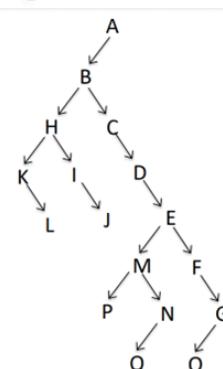
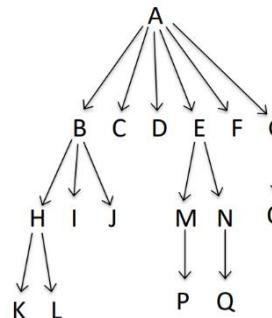


Tipos de implementaciones

- los hijos son una lista de punteros
 - consultar hijo i-ésimo es ineficiente
 - pero recorridos secuenciales es eficientes
 - eliminar hijo actual es eficiente

Tipos de implementaciones

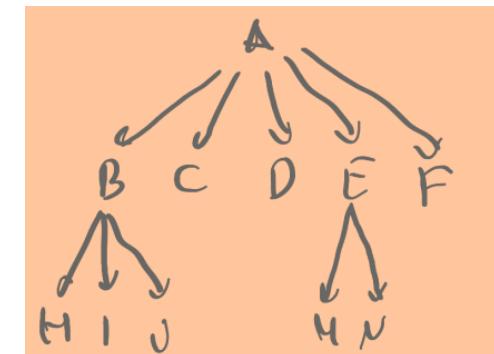
- árbol binario - hijo izquierdo: primer hijo; hijo derecho: siguiente hermano
 - consultar hijo i-ésimo es ineficiente
 - pero los recorridos secuenciales son eficientes
 - El recorrido en preorden del árbol binario es el recorrido en preorden del árbol general



Tipos de implementaciones

- los hijos son un vector de punteros
 - consultar hijo i-ésimo es eficiente
 - eliminar hijo i-ésimo puede ser ineficiente

```
template <class T> class ArbolGen {  
private:  
    struct nodo_arbolGen{  
        T info;  
        vector<nodo_arbolGen*> sig;  
    };  
    nodo_arbolGen* primer_nodo;
```



```
template <class T> class ArbolGen {  
private:  
    struct nodo_arbolGen{  
        T info;  
        nodo_arbolGen* h_mayor;  
        nodo_arbolGen* hermano;  
    };  
    nodo_arbolGen* primer_nodo;
```

```

//Suma de los elementos de un árbol n-ario
/* 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);
    }
}

```

```

/* Post: a es como A, pero habiendo sumado k a todos sus elementos */
void suma_k(ArbolGen <int> &a, int k){
    if (not a.es_vacio) {
        int s = a.raiz()+k;
        int n = v.size();
        vector<ArbolGen <int>> v(n);
        a.hijos(v);
        if (n != 0) {
            for (int i = 0; i < n; ++i)
                suma_k(v[i],k);
        }
        a.plantar(s,v);
    }
    vector <int> grado_medio() const{
        int t,sg,na;
        i_grado_medio (primer_nodo,t,sg,na);
        if (t == 0) return 0.0;
        else return double(sg)/double(t);
    }

    static void i_grado_medio(nodo_arbolGen* p, int& tam,
                               int& sum_grados, int& narb) const {
        int t1,sg1,na1;
        int tg,sgg,nag;
        if (p == nullptr) {
            tam = 0; sum_grados = 0; narb = 0
        } else {
            i_grado_medio (p->hermano,tg,sgg,nag);
            if (p->h_mayor != nullptr) {
                i_grado_medio (p->h_mayor,t1,sg1,na1);
                tam = 1 + t1 + tg;
                sum_grados = sg1 + sgg + na1;
                narb = nag + 1;
            } else {
                tam = tg; sum_grados = sgg; narb = nag+1;
            }
        }
    }
}
```

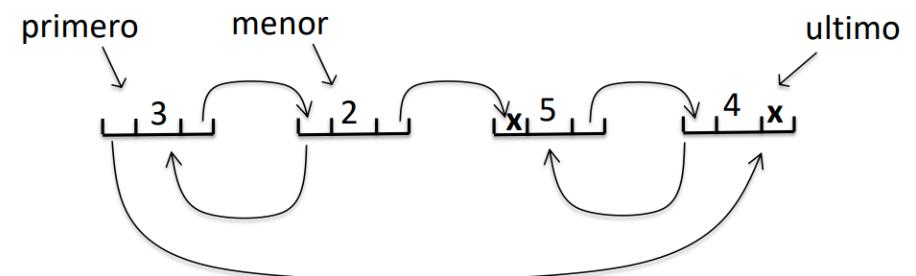
Multilistas

- Queremos guardar una tabla muy grande pero muy dispersa (con muchos elementos nulos). Se necesita:
 - Dado el indice de una fila, recuperar todos los elementos no nulos de la fila
 - Dado el indice 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
```

Colas ordenadas

- Las colas son una extensión de las colas, con la posibilidad de ser recorridas en orden creciente con respecto del valor de sus elementos:
- Es necesario que haya definido un operador < sobre los valores de la cola
- Hay que utilizar más punteros:
 - *Primero*, *ultimo* y *sig* para gestionar el orden de llegada.
 - *menor* y *sig_ord* para gestionar el orden creciente según el valor de los elementos

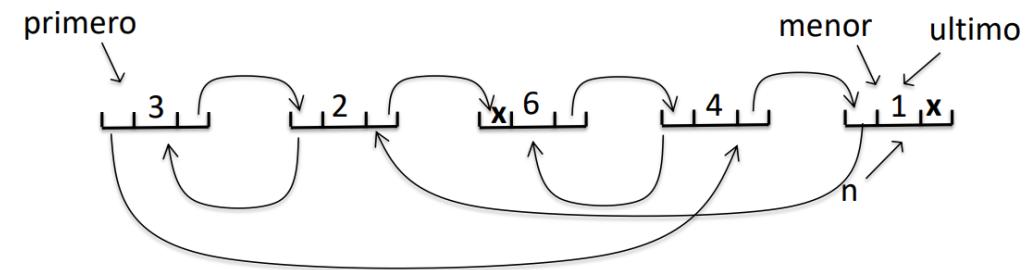
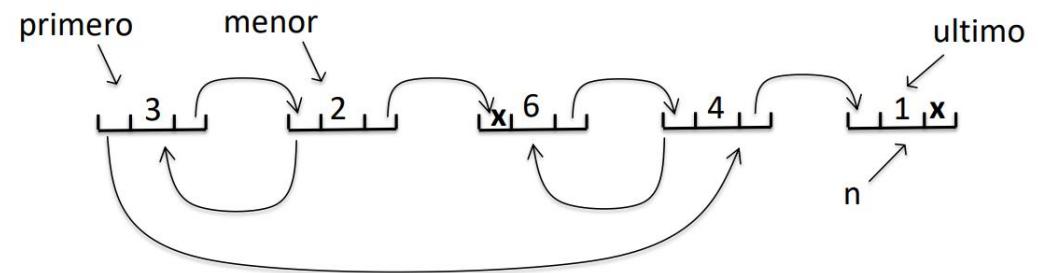
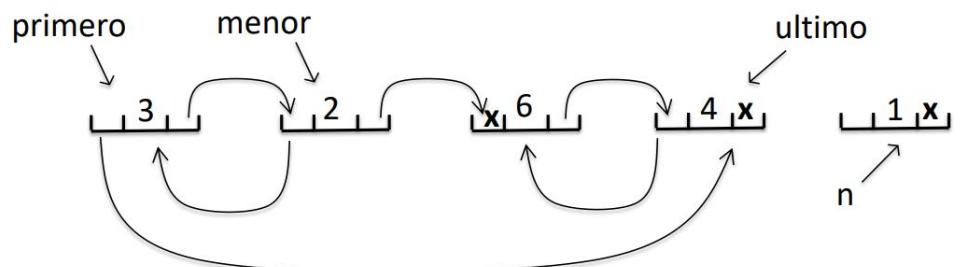


```
template <class T> class ColaOrd {  
    private:  
        struct nodo_colaOrd{  
            T info;  
            nodo_colaOrd* sig;  
            nodo_colaOrd* sig_ord;  
        };  
        int longitud;  
        nodo_colaOrd* primero;  
        nodo_colaOrd* ultimo;  
        nodo_colaOrd* menor;
```

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

```
void pedir_turno(const T& v){  
    nodo_colaOrd *n = new nodo_cola;  
    n->info = v; n->sig = nullptr;  
    ++longitud;  
  
    if (primero == nullptr) {  
        primero = n; ultimo = n;  
        menor = n; n->sig_ord = nullptr;  
    }  
    else {  
        ultimo->sig = n; ultimo = n;  
        if (v < menor->info) {  
            n->sig_ord = menor; menor = n;  
        }  
        else {  
            nodo_colaOrd *ant = menor;  
            bool encontrado = false;  
            while (ant->sig_ord != nullptr  
                   and not encontrado) {  
                if (v < (ant->sig_ord)->info) encontrado = true;  
                else ant = ant->sig_ord;  
            }  
            n->sig_ord = ant->sig_ord; ant->sig_ord = n;  
        }  
    }  
}
```

pedir_turno(1)



```

/* 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.primero != nullptr) {
        longitud = longitud + c.longitud;
        if (primero == nullptr) {
            primero = c.primero; menor = c.menor;
            ultimo = c.ultimo;
        }
        else {
            ultimo->sig = c.primero;
            ultimo = c.ultimo;
        }
    }
    /* ordenamos los elementos de la cola */
    nodo_colat *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.primero = c.ultimo = c.menor = nullptr;
    c2.longitud = 0;
}

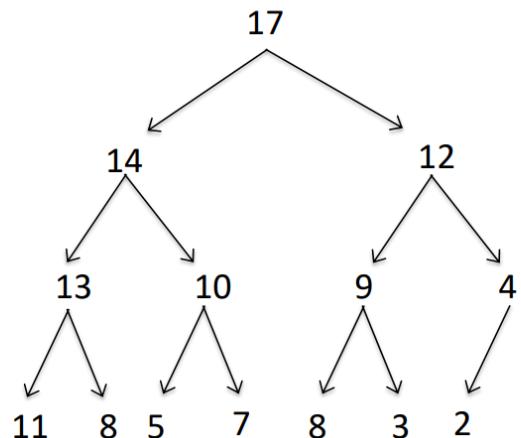
```

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 e1 y e2 tienen prioridades p1 y p2, y p1 < p2, e2 saldrá antes de la cola.
- Las colas con prioridad, se pueden implementar como colas ordenadas, pero hay una implementación más eficiente utilizando *heaps*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16
17	14	12	13	10	9	4	11	8	5	7	8	3	2		

↑
sl



```

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_primer();
  
```

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

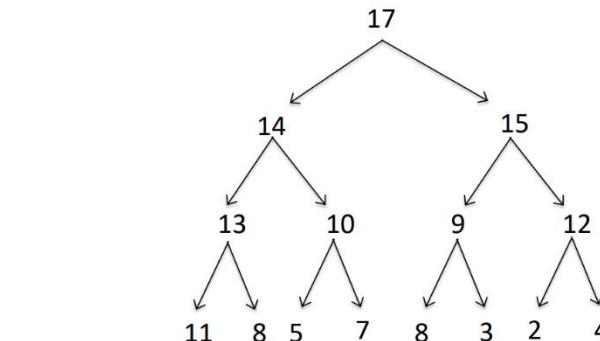
- La forma más simple de implementar un heap es usando un vector, donde:
 - la raíz del heap está en la posición 0 del vector .
 - si un nodo del heap ocupa la posición i del vector, su hijo izquierdo está en la posición $2*i+1$ y su hijo derecho en la siguiente ($2*(i+1)$).
 - como consecuencia, si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

/ 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 n = sl; ++sl;
    int p = (n-1)/2;
    while (p >=0 and h[n]>h[p]) {
        swap(h[n], h[p]);
        n = p;
        p = (n-1)/2;
    }
}

/* Post: retorna el mayor elemento del heap*/
T primero(){
    return v[0];
}
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	14	15	13	10	9	12	11	8	5	7	8	3	2	4		

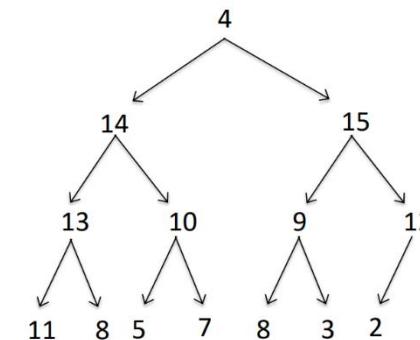


/ Post: Se elimina la raiz del heap de tal manera que siga cumpliendo sus propiedades */*

```
void salir_primer()
```

```
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[maxhijo])
            ++maxhijo;
        if (h[n]>h[maxhijo]) fin = true;
        else { swap(h[maxhijo], h[n]); n=maxhijo; }
    }
}
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	14	15	13	10	9	12	11	8	5	7	8	3	2	4		



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
15	14	12	13	10	9	4	11	8	5	7	8	3	2	4		

