

Estructuras de datos

Clases teóricas

por Pablo E. “Fidel” Martínez López

11. Linked lists y árboles en imperativo

Modelos de computación

C y la memoria

- ❑ Memoria estática
 - ❑ Es la memoria conformada por los *frames*
 - ❑ Cada función tiene su *frame* con sus variables locales
 - ❑ Al invocar un función se abre su *frame*
 - ❑ Al terminar el función se elimina su *frame*
 - ❑ Se comporta como una pila, por eso se la llama **Stack**
 - ❑ Y a los *frames*, se los llama ***stack frames***
 - ❑ Se dice que es ***estática*** pues su comportamiento no depende de valores de ejecución

C y la memoria

- ❑ Memoria dinámica
 - ❑ Es una memoria formada por *espacios reservados*
 - ❑ Se reserva memoria con la operación **new**
 - ❑ Se obtiene un puntero (*) al espacio reservado
 - ❑ Se libera la reserva con la operación **delete**
 - ❑ Llamada memoria **Heap** (pues no tiene orden predeterminado)
 - ❑ Se dice **dinámica** pues es controlada por el programador
 - ❑ Requiere un control cuidadoso para evitar problemas
 - ❑ *Memory leaks*, interferencias, etc.

C y la memoria

- ❑ Estructuras de datos
 - ❑ Variables locales, en Stack
 - ❑ TADs, en Heap, accesible mediante punteros
 - ❑ Es casi imposible lograr una abstracción total
 - ❑ Arrays, tanto en Stack como en Heap
- ❑ Deben hacerse consideraciones
 - ❑ De eficiencia (tanto en tiempo como en espacio)
 - ❑ Para evitar problemas diversos
 - ❑ Violaciones de memoria, *memory leaks*, interferencia, etc.

Estructuras de datos en memoria

Linked lists

❏ ¿Cómo se pueden representar listas en la memoria?

❏ **Opción 1: *ArrayLists***

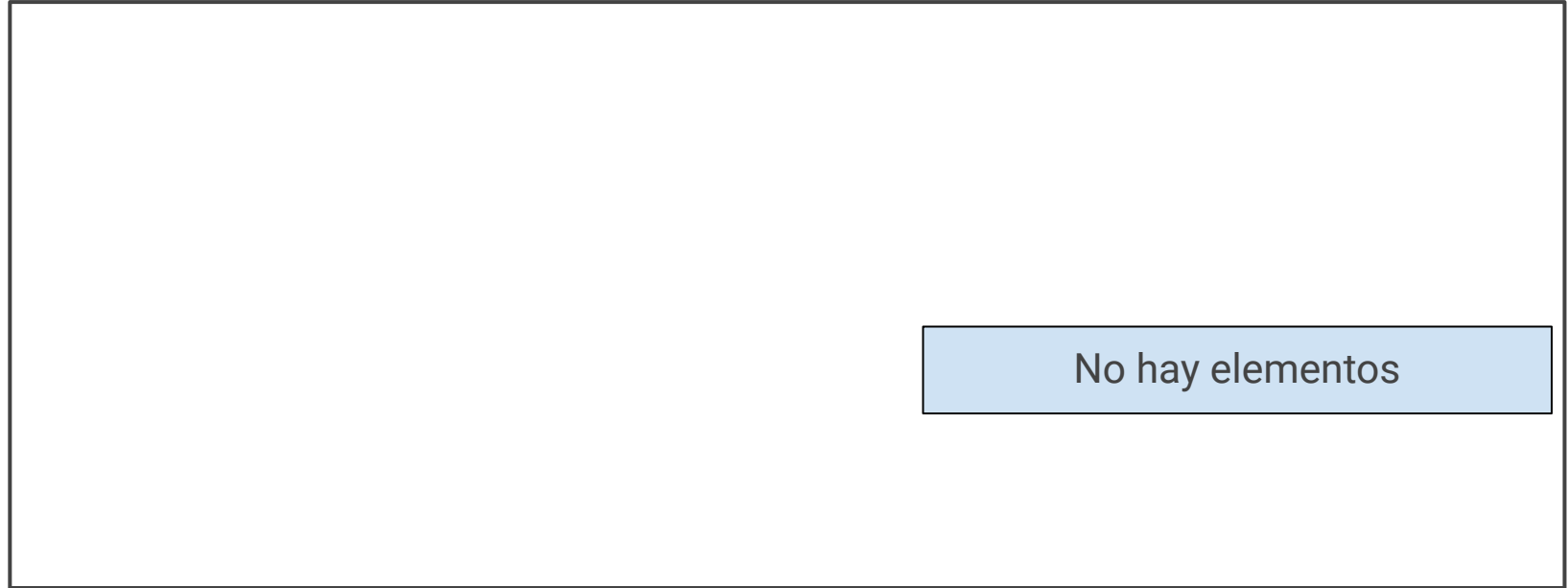
- ❏ Acceso $O(1)$ a cualquier posición fija
- ❏ Tamaño fijo, se puede llenar
- ❏ Caro insertar adelante o en medio

❏ **Opción 2: algún TAD en Heap**

- ❏ ¿Cómo debería ser la interfaz?
- ❏ ¿Cómo debería ser la implementación?
- ❏ Debería poder tener un tamaño variable...

Linked lists

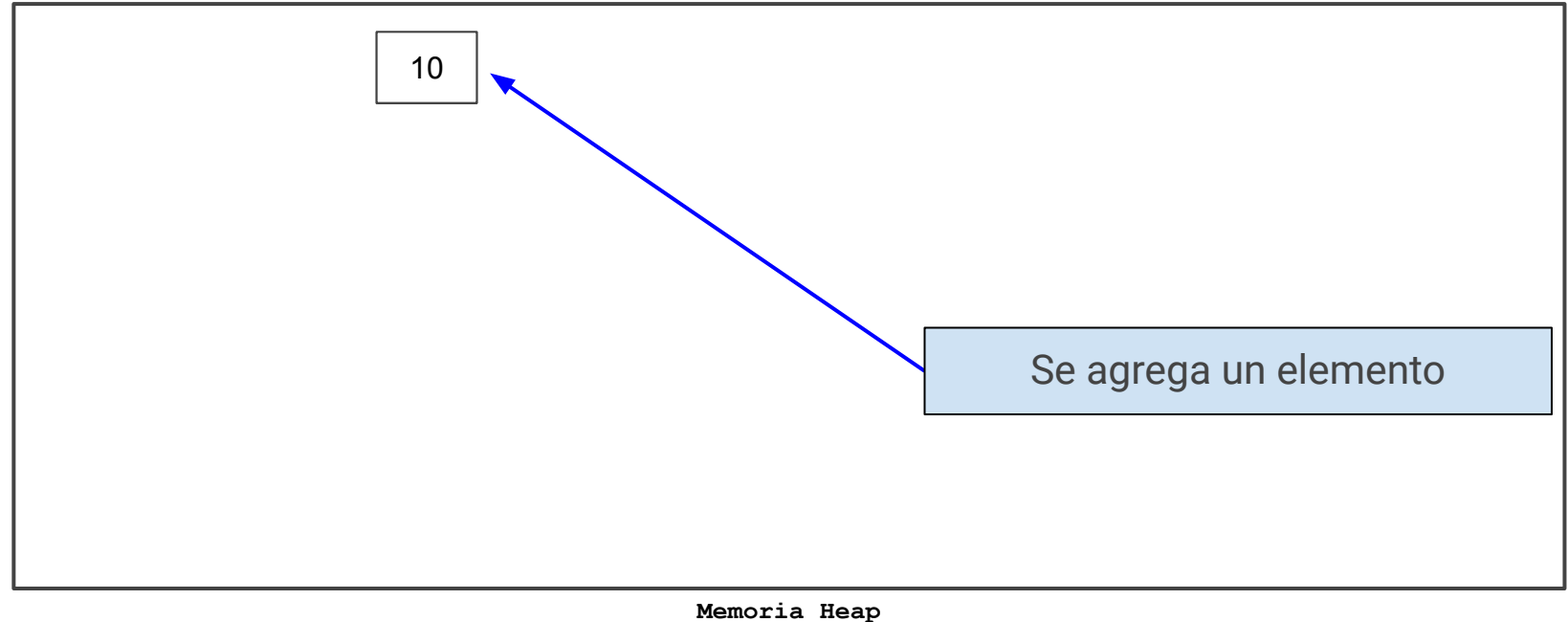
❏ ¿Cómo debería ser la memoria de una lista?



Memoria Heap

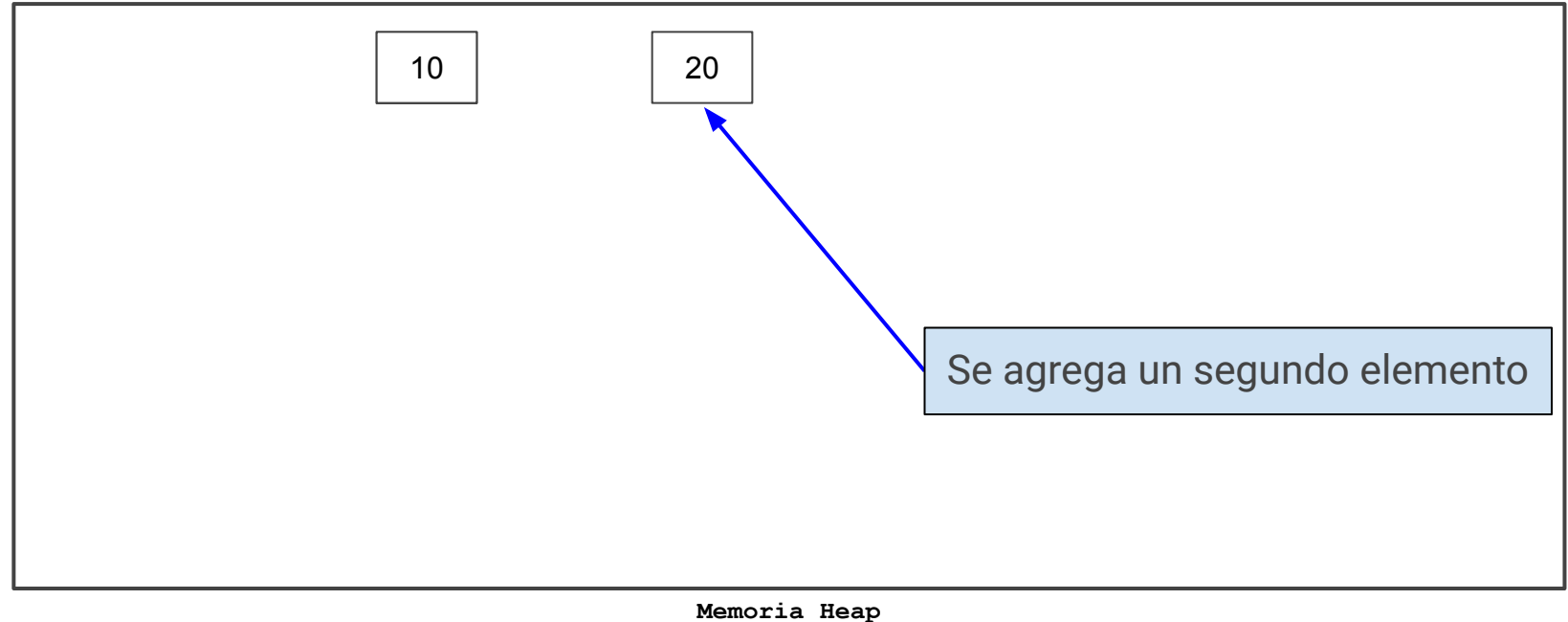
Linked lists

❏ ¿Cómo debería ser la memoria de una lista?



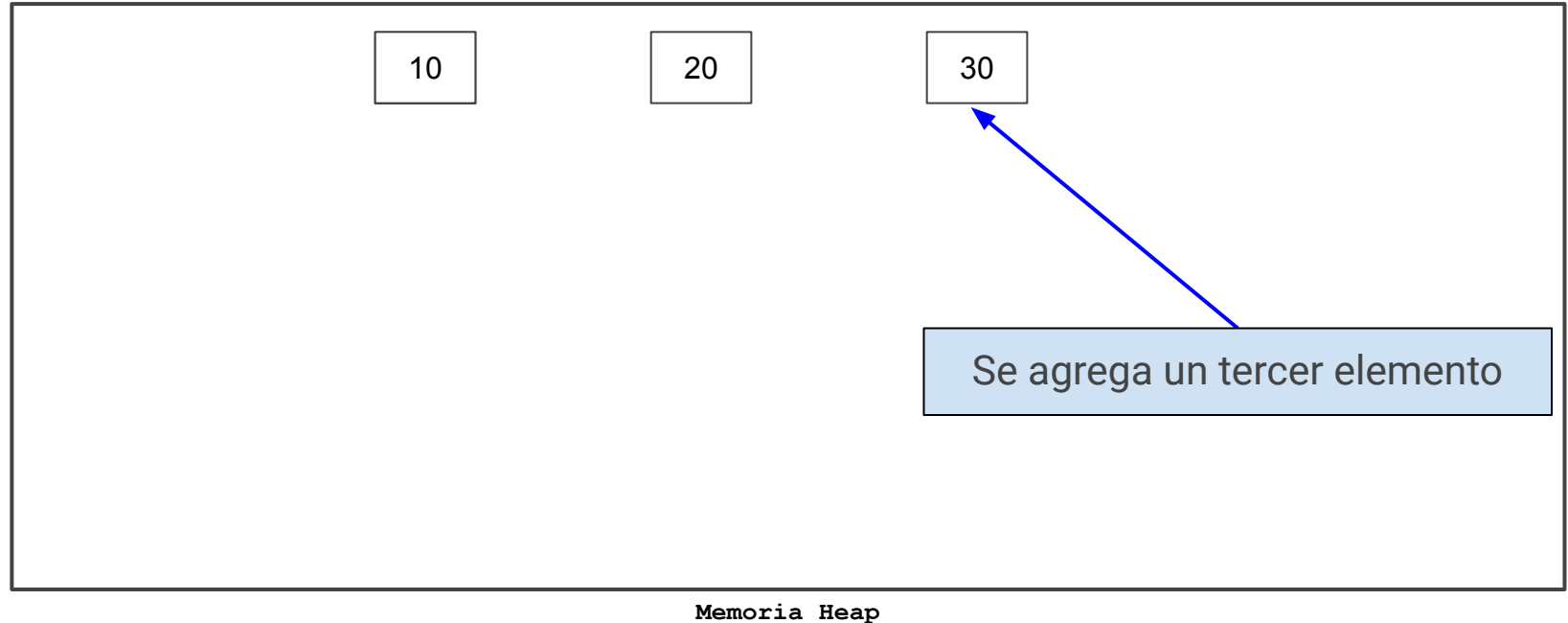
Linked lists

❏ ¿Cómo debería ser la memoria de una lista?



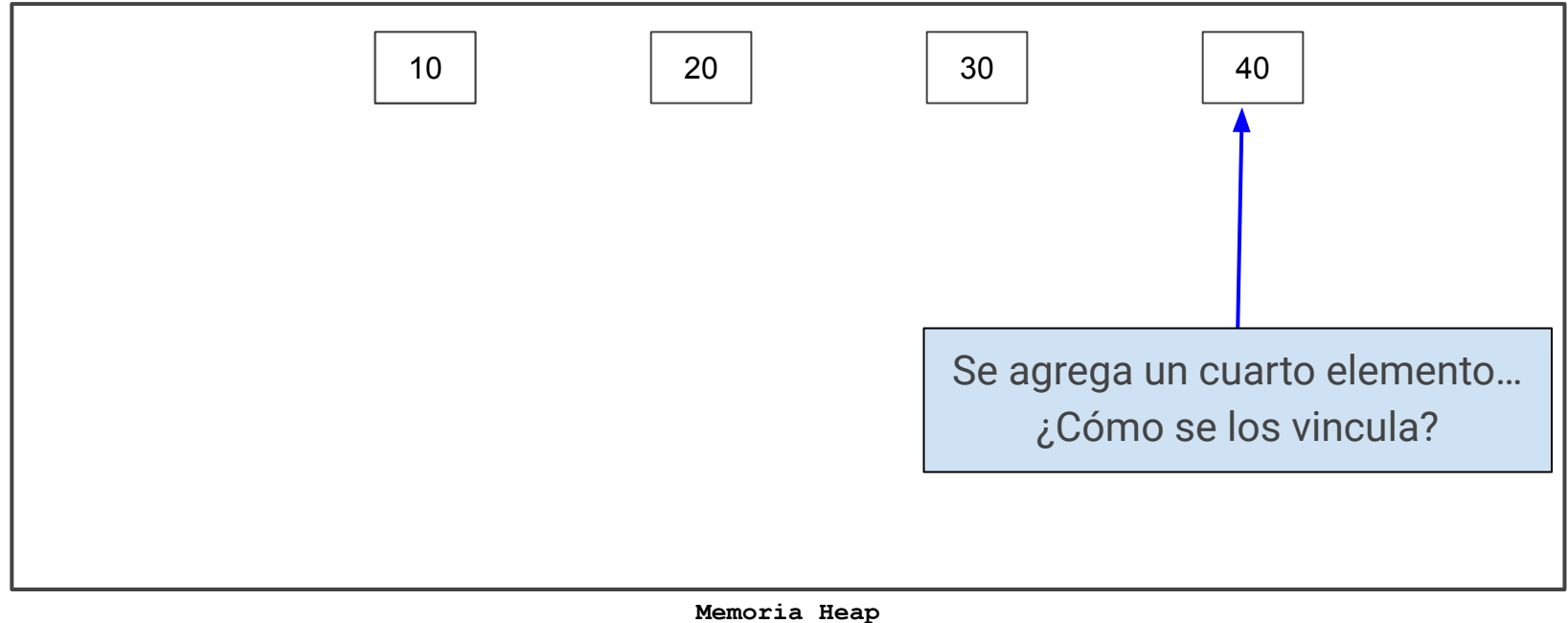
Linked lists

❏ ¿Cómo debería ser la memoria de una lista?



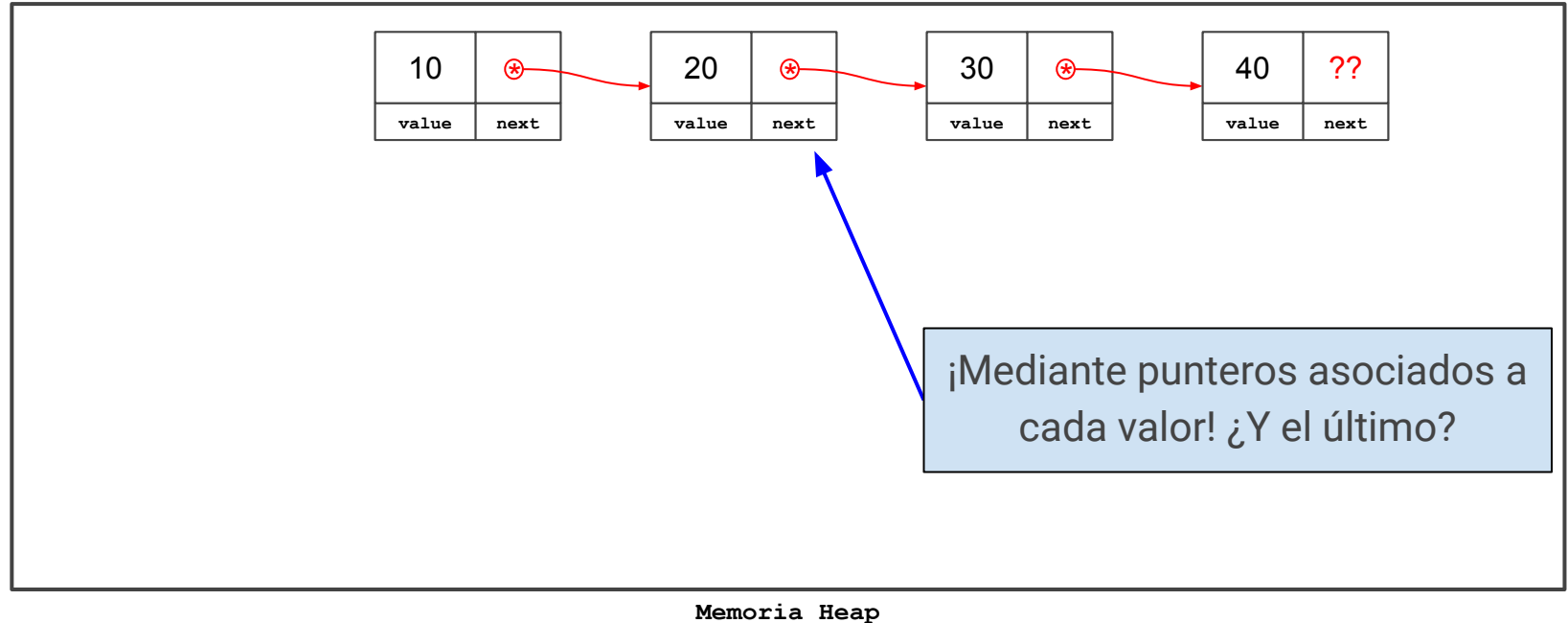
Linked lists

❏ ¿Cómo debería ser la memoria de una lista?



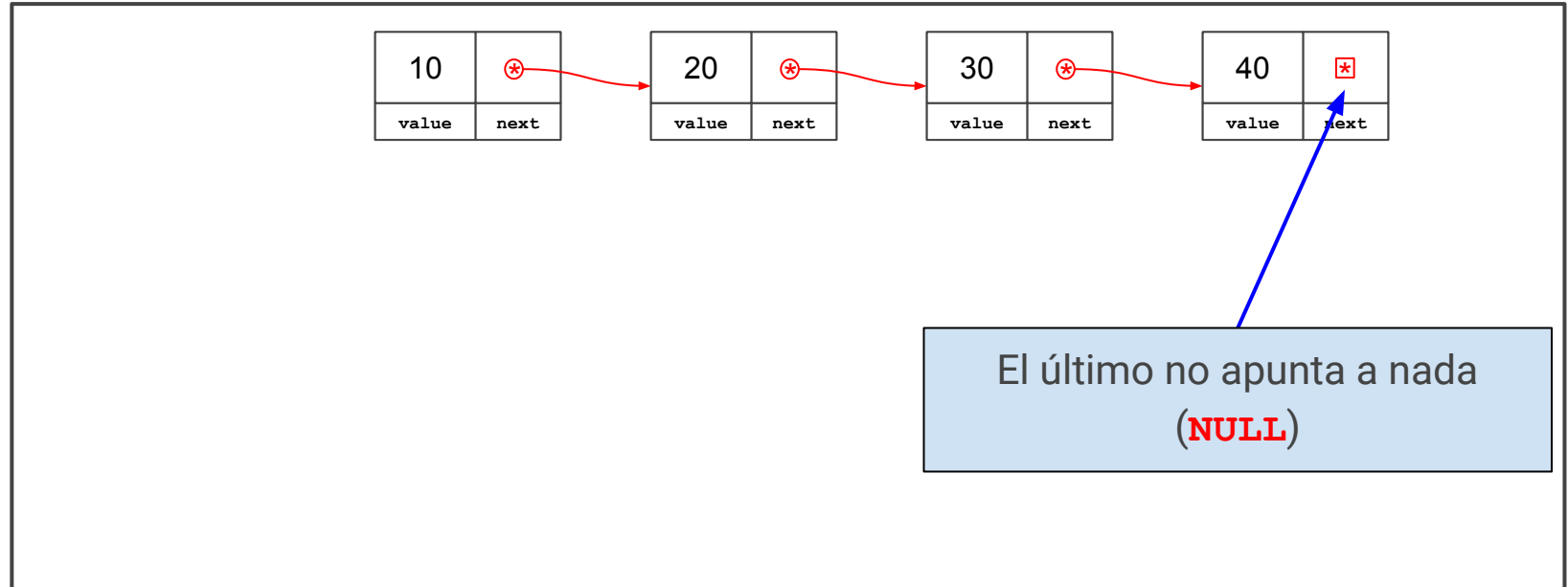
Linked lists

❏ ¿Cómo debería ser la memoria de una lista?



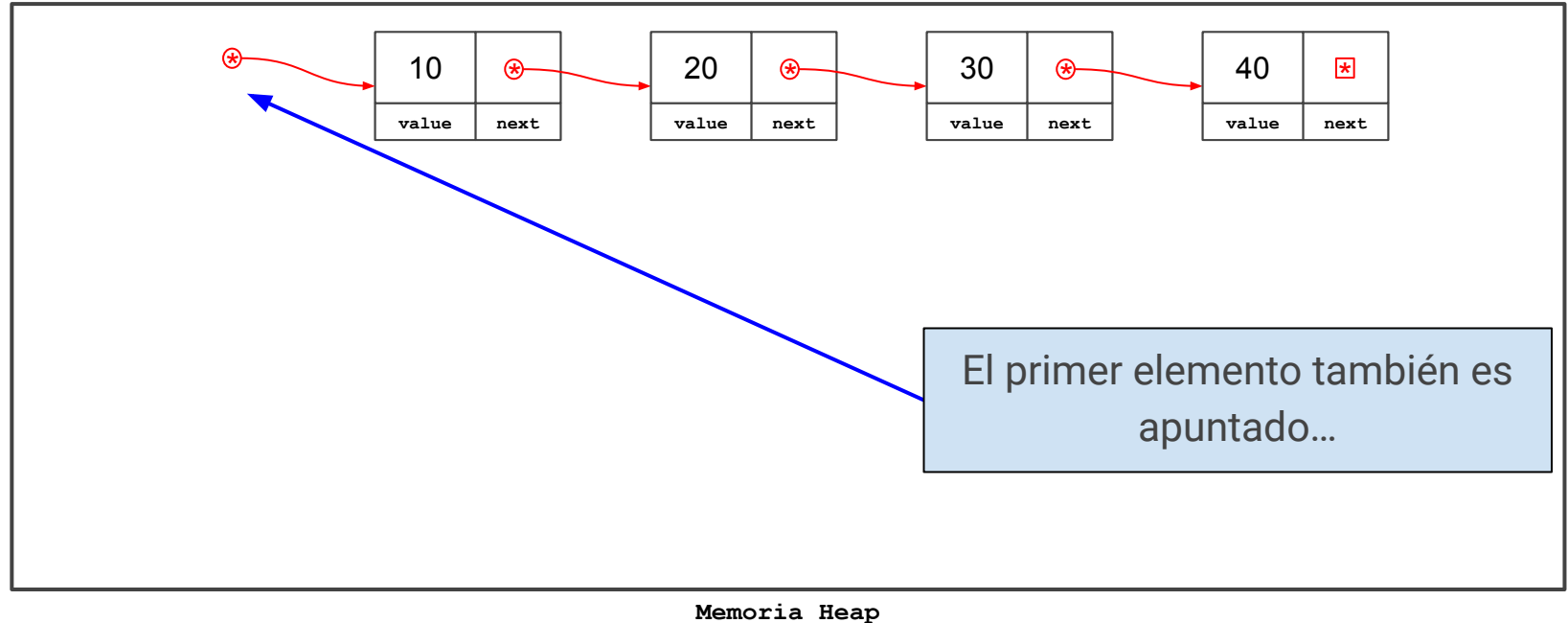
Linked lists

❏ ¿Cómo debería ser la memoria de una lista?



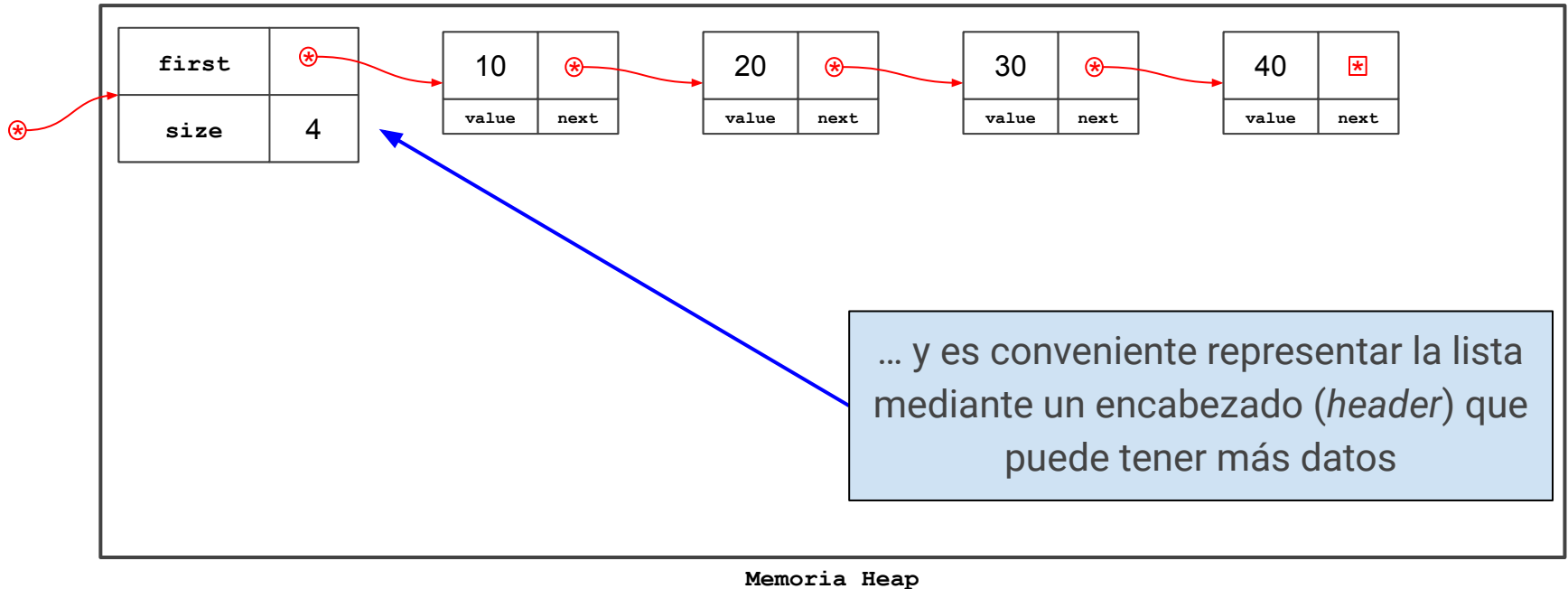
Linked lists

❏ ¿Cómo debería ser la memoria de una lista?



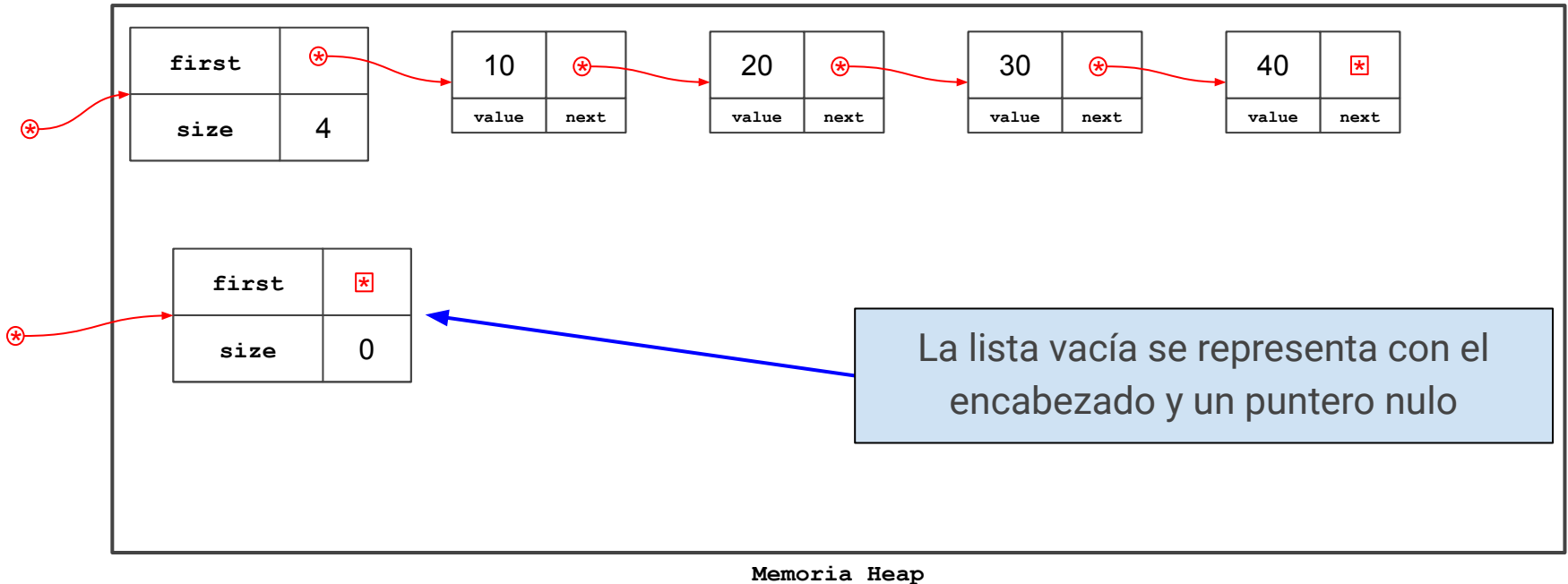
Linked lists

¿Cómo debería ser la memoria de una lista?



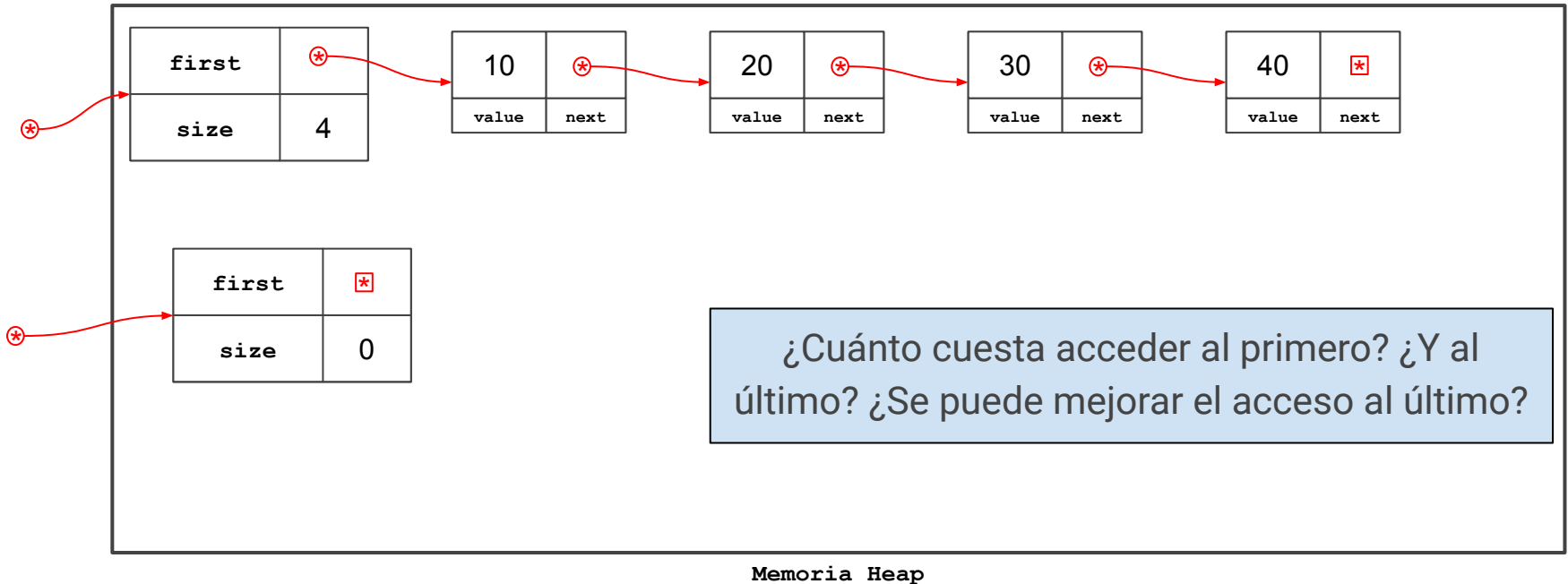
Linked lists

¿Cómo debería ser la memoria de una lista?



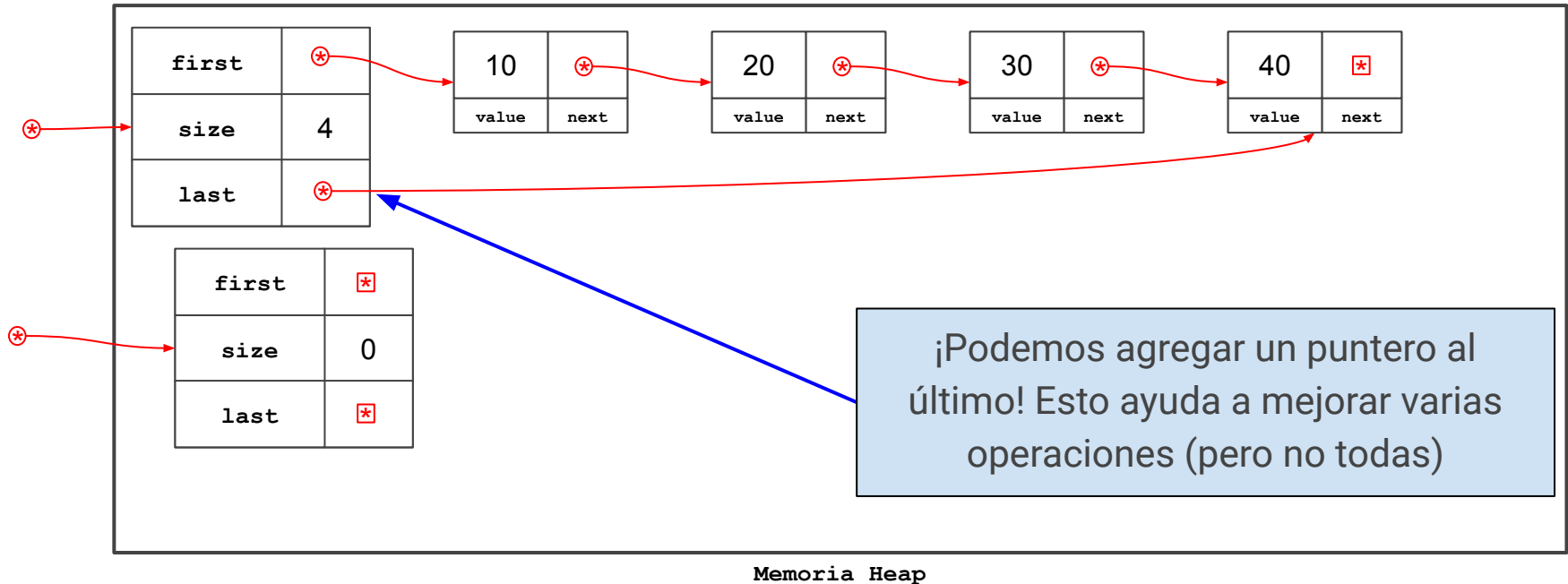
Linked lists

¿Cómo debería ser la memoria de una lista?



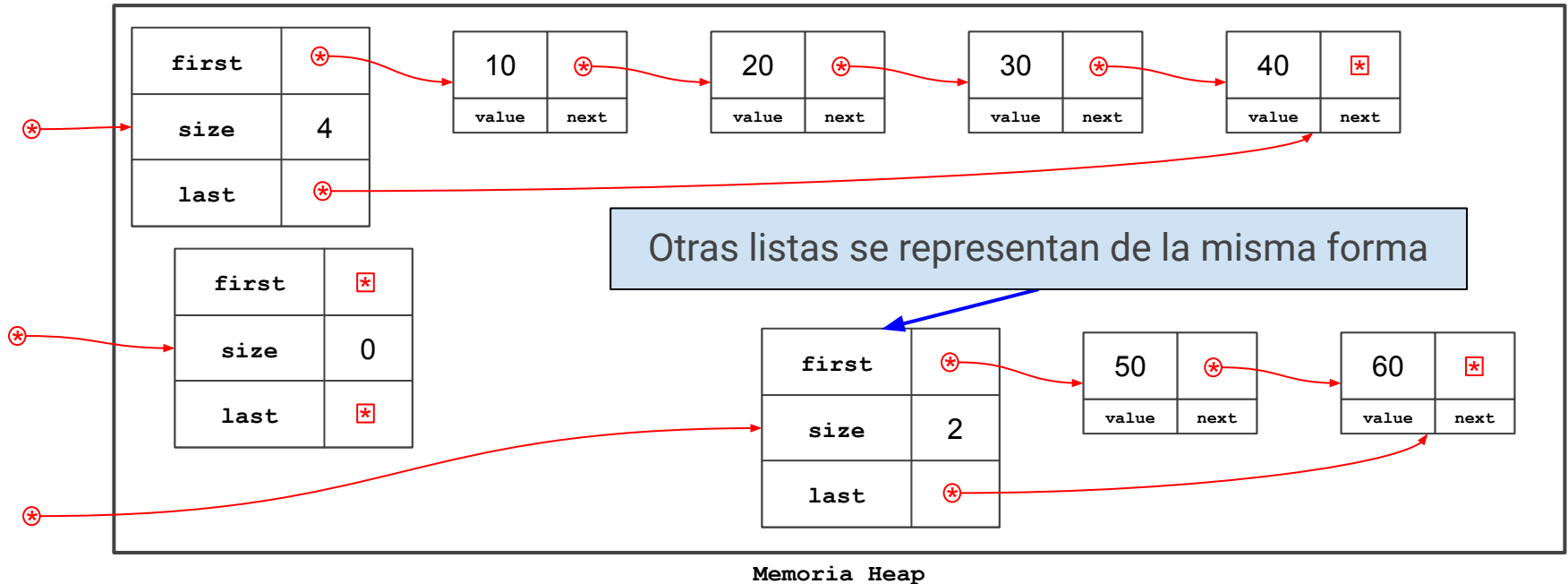
Linked lists

¿Cómo debería ser la memoria de una lista?



Linked lists

¿Cómo debería ser la memoria de una lista?



Linked lists

- ❏ ¿Cuál es la interfaz para lograr esa representación?
 - ❏ Se propone una interfaz imperativa (con procedimientos)

Linked lists

- ❑ ¿Cuál es la interfaz para lograr esa representación?
- ❑ Se propone una interfaz imperativa (con procedimientos)

```
struct ListHeaderSt;  
typedef ListHeaderSt* List;  
    // INV.REP.: el puntero NO es NULL  
  
List emptyList();  
void Cons(int n, List xs);  
void Snoc(List xs, int n);  
bool isEmptyList(List xs);  
int head(List xs); // PRECOND: lista no vacía  
void Tail(List xs); // ídem  
int length(List xs);  
void Liberar(List xs);
```

Las listas son punteros
al encabezado

¡Modifican su
argumento!

Linked lists

- ❏ ¿Y la implementación para lograr esa representación?
 - ❏ Primero los tipos de datos

Linked lists

❏ ¿Y la implementación para lograr esa representación?

❏ Primero los tipos de datos

Observar la recursión en la referencia
a la misma estructura

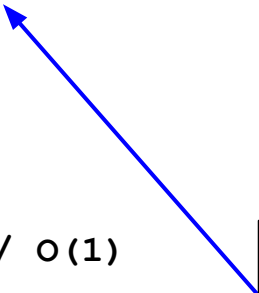
```
struct NodeL {  
    int    value;           // INV. REP.:  
    NodeL* next;           // * los punteros internos NO son compartidos  
};  
  
struct ListHeaderSt {      // INV. REP.:  
    NodeL* first;           // * first = NULL sii last = NULL  
    int    size;           // * size es la cant. de nodos a recorrer  
    NodeL* last;           // * hasta llegar a un NULL desde first  
};                          // * si last != NULL, last->next = NULL
```


Linked lists

- ❏ ¿Y la implementación para lograr esa representación?
- ❏ Luego las operaciones (1)

```
List emptyList() {    // O(1)
    ListHeaderSt* xs = new ListHeaderSt;
    xs->first = NULL;
    xs->last  = NULL;
    xs->size  = 0;
    return xs;
}

bool isEmptyList(List xs) {    // O(1)
    return (xs->size==0);
}
```



Se genera el encabezado
en la Heap

Linked lists

- ❏ ¿Y la implementación para lograr esa representación?
- ❏ Luego las operaciones (2)

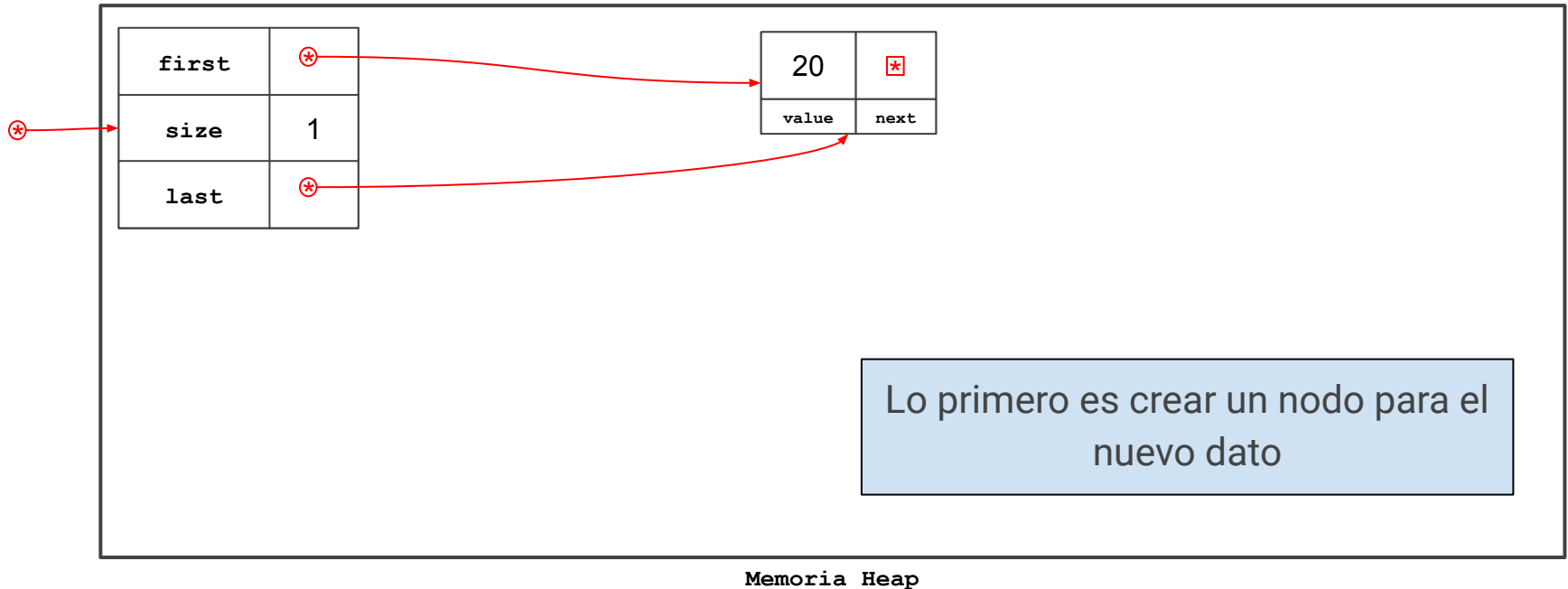
```
int head(List xs) { // O(1)
    // PRECOND: lista no vacía
    return (xs->first->value);
}
```

```
int length(List xs) { // O(1)
    return (xs->size);
}
```


Observar que las precondiciones garantizan accesos correctos

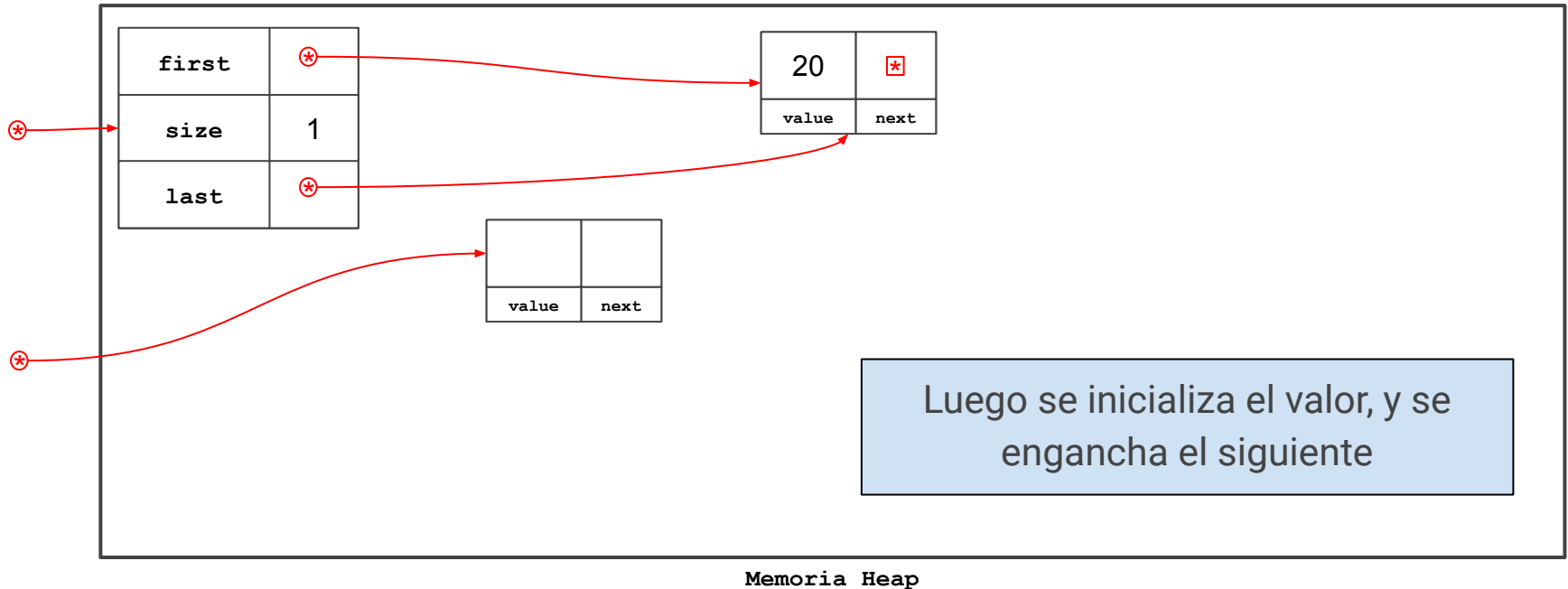
Linked lists

📄 ¿Cómo funciona Cons? Agregar 10 a esta lista...



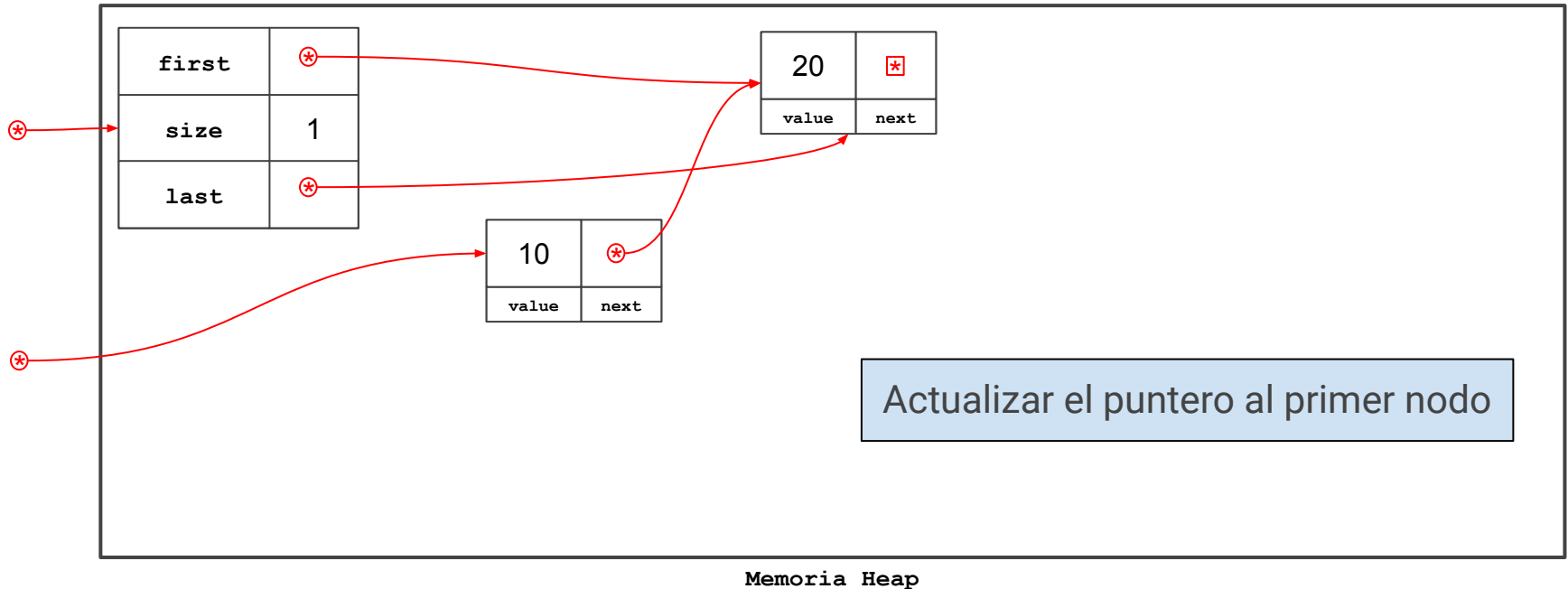
Linked lists

📄 ¿Cómo funciona Cons? Agregar 10 a esta lista...



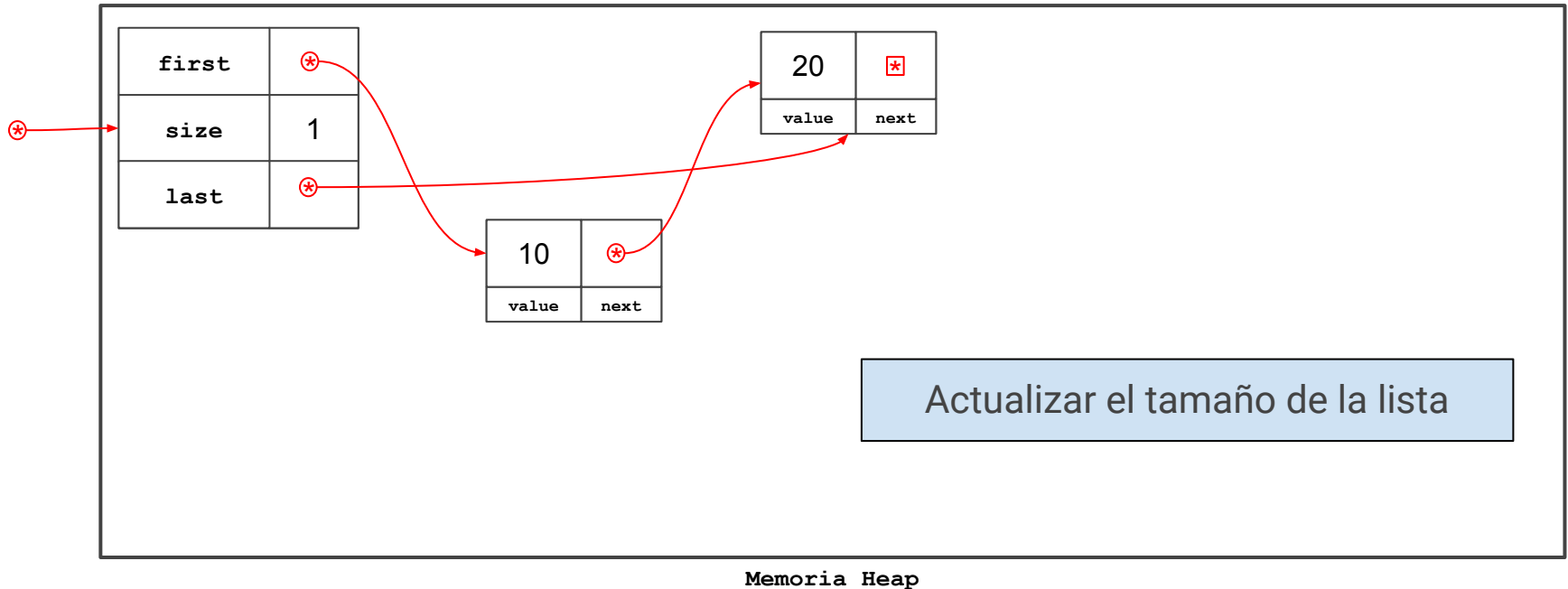
Linked lists

📄 ¿Cómo funciona Cons? Agregar 10 a esta lista...



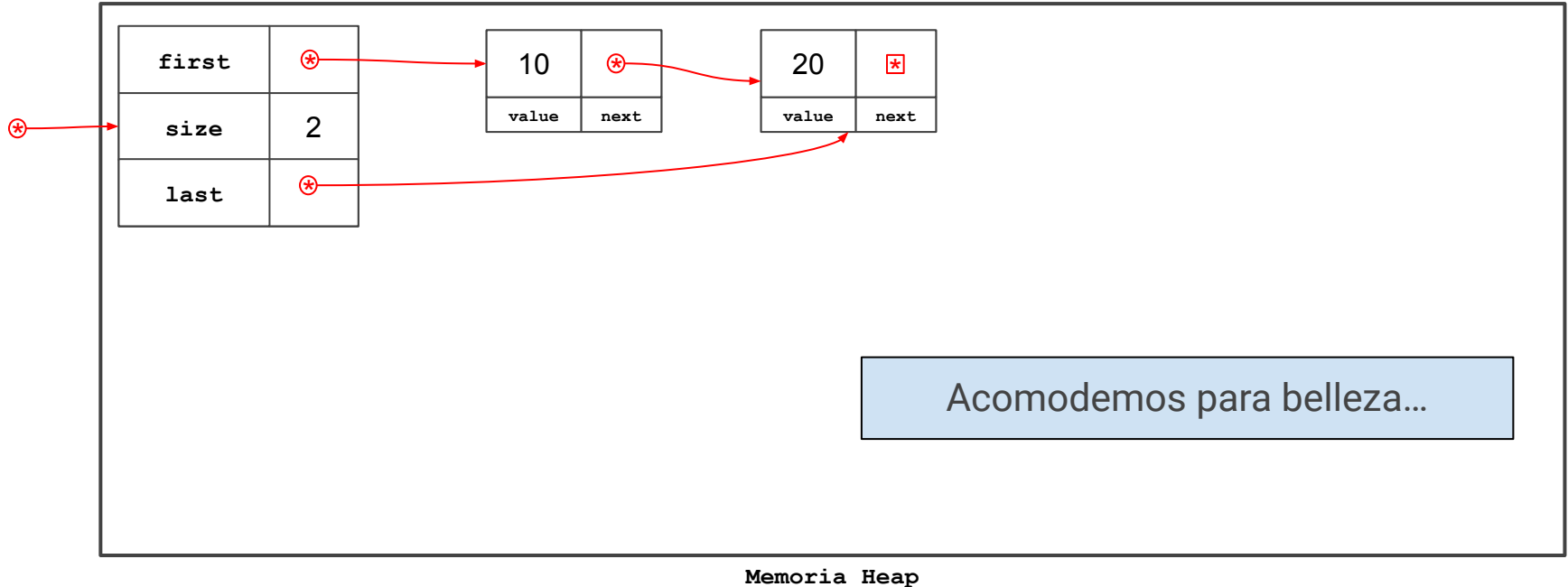
Linked lists

¿Cómo funciona Cons? Agregar 10 a esta lista...



Linked lists

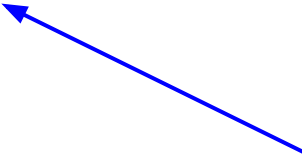
📄 ¿Cómo funciona Cons? Agregar 10 a esta lista...



Linked lists

- ❏ ¿Y la implementación para lograr esa representación?
- ❏ Luego las operaciones (3)

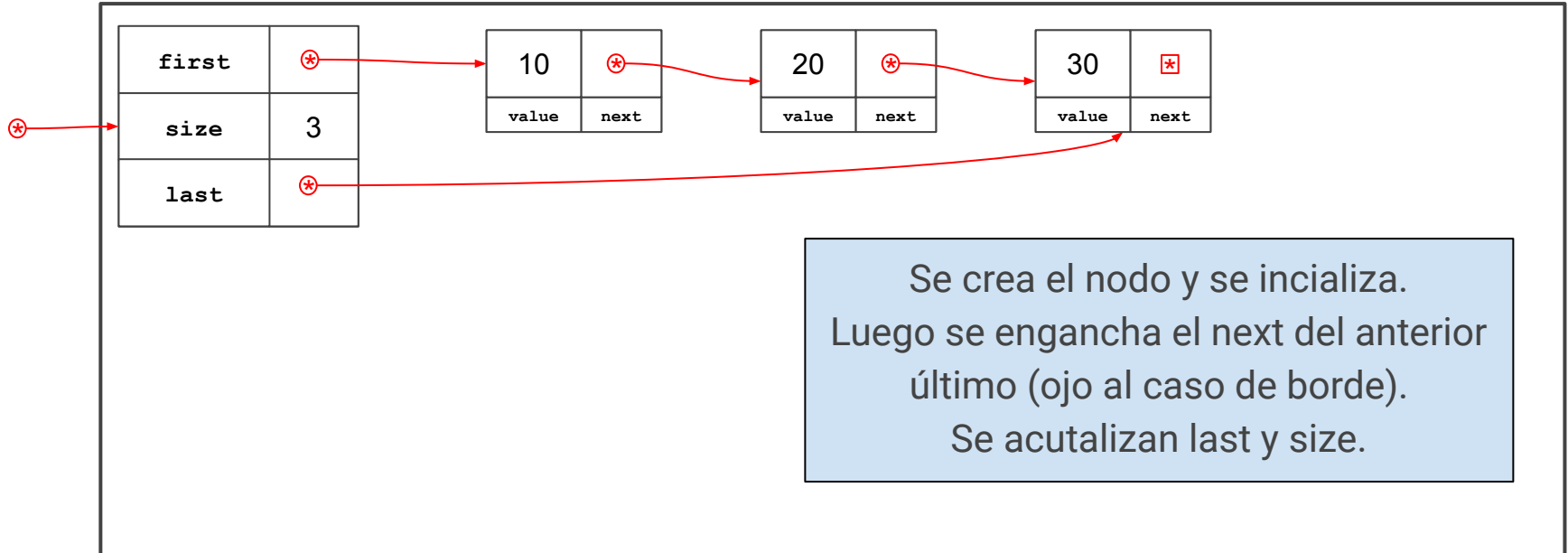
```
void Cons(int n, List xs) { // O(1)
    NodeL* node = new NodeL;
    node->value = n;
    node->next = xs->first;
    xs->first = node;
    if (xs->last == NULL) { xs->last = node; }
    xs->size++;
}
```



Deben garantizarse los invariantes
y manejar el caso de borde
(al agregar el primer elemento)

Linked lists

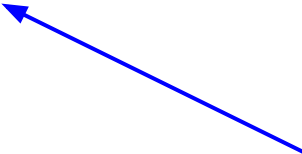
❏ ¿Cómo funciona Snoc? Agregar 30 a esta lista...



Linked lists

- ❏ ¿Y la implementación para lograr esa representación?
- ❏ Luego las operaciones (4)

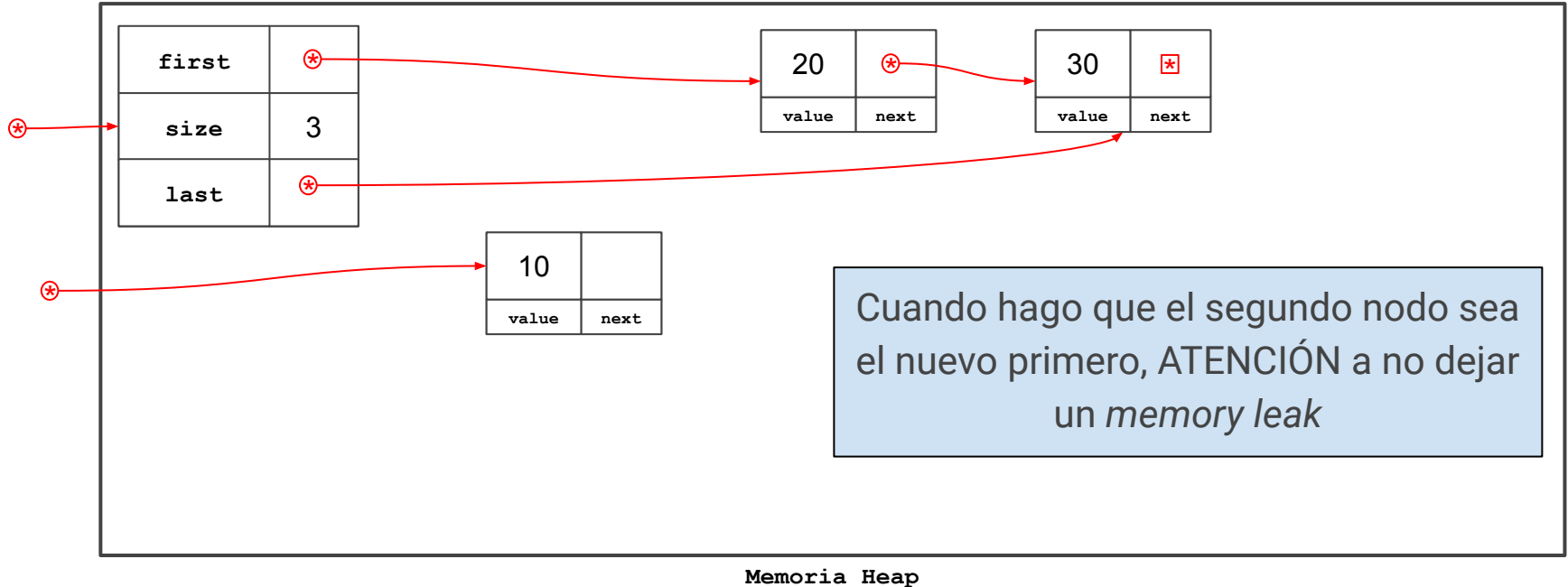
```
void Snoc(List xs, int n) { // O(1)
    NodeL* node = new NodeL;
    node->value = n;
    node->next = NULL;
    if (xs->last == NULL) { xs->first = node; }
    else { xs->last->next = node; }
    xs->last = node;
    xs->size++;
}
```



Deben garantizarse los invariantes
y manejar el caso de borde
(al agregar el primer elemento)

Linked lists


❏ ¿Cómo funciona Tail? Agregar 30 a esta lista...



Linked lists

- ❏ ¿Y la implementación para lograr esa representación?
- ❏ Luego las operaciones (5)

```
void Tail(List xs) { // O(1)
    // PRECOND: lista no vacía
    NodeL* temp = xs->first;
    xs->first = xs->first->next;
    if(xs->first==NULL) { xs->last=NULL; }
    xs->size--;
    delete temp;
}
```



La memoria del elemento
que se quita, se debe liberar

Linked lists

- ❏ ¿Y la implementación para lograr esa representación?
- ❏ Luego las operaciones (6)

```
void Liberar(List xs) { // O(n)
    NodeL* temp = xs->first;
    while (xs->first != NULL) {
        xs->first = xs->first->next;
        delete temp;
        temp = xs->first;
    }
    delete xs;
}
```

Recordar actualizar al próximo
nodo que se debe liberar

Luego de liberar todos los nodos,
se libera el encabezado

Linked lists

- ❑ Se las conoce como *linked lists*
 - ❑ Porque los nodos se enlazan como una cadena
 - ❑ Se suele traducir (mal) como “listas encadenadas”
 - ❑ Sería más adecuado traducirlas como “listas enlazadas” o “lista con forma de cadena”
 - ❑ Pueden proveerse más operaciones en $O(1)$ agregándolas a la interfaz (e.g. **last**, **Init**, **Append**)
 - ❑ ¿Cómo se utiliza esta interfaz? ¿Es suficiente?

Linked lists e iteradores

■ ¿Cómo sería hacer un recorrido como usuario?

■ La interfaz es destructiva...

```
int sum(List xs) { // O(n)
    int total = 0;
    while (!isEmptyList(xs)) {
        total += head(xs);
        Tail(xs);
    }
    return total;
}
```


Esta operación modifica la lista argumento, eliminando un elemento

■ ... ¡y al terminar se eliminaron todos los elementos!

Linked lists e iteradores

- ❏ Opción: hacer los recorridos como implementador

```
int sum(List xs) { // O(n)
    int total = 0;
    NodeL* current = xs->first;
    while (current!=NULL) {
        total += current->value;
        current = current->next;
    }
    return total;
}
```



¡Se requiere conocer la necesidad del usuario de antemano!

- ❏ ¿Es factible generalizar esta forma?
- ❏ ¿Cuántas operaciones de interfaz serían necesarias?

Linked lists e iteradores

- Solución:
 - Se agregan operaciones de recorrido a la interfaz

Linked lists e iteradores

■ Solución: *iteradores*

■ Se agregan operaciones de recorrido a la interfaz

■ Para flexibilidad, se usa un tipo asociado a la lista

```
struct ListHeaderSt; struct ListIteratorSt;

typedef ListHeaderSt* List;           // INV.REP.: el puntero NO es NULL
typedef ListIteratorSt* ListIterator; // INV.REP.: el puntero NO es NULL

...

ListIterator iniciarRecorrido(List xs);
bool estaAlFinalDelRecorrido(ListIterator ixs);
int elementoActual(ListIterator ixs);           // PRECOND: no está al
void PasarAlSiguienteElemento(ListIterator ixs); // fin del recorrido
void LiberarIterador(ListIterator ixs);
```

Linked lists e iteradores

❏ ¿Cómo sería hacer un recorrido como usuario?

❏ ¡Usando iteradores!

```
int sum(List xs) { // O(n)
    int total = 0;
    ListIterator ixs = iniciarRecorrido(xs);
    while (!estaAlFinalDelRecorrido(ixs)) {
        total += elementoActual(ixs);
        PasarAlSiguienteElemento(ixs);
    }
    LiberarIterador(ixs); // Una vez utilizado, se libera
    return total;
}
```

El recorrido se hace utilizando el iterador, y no modifica la lista original

❏ La lista se preserva

Linked lists e iteradores

■ Solución: iteradores

■ Implementación (1)

```
struct ListIteratorSt {  
    NodeL* current;  
};  
  
...  
  
ListIterator iniciarRecorrido(List xs) {  
    ListIteratorSt* ixs = new ListIteratorSt;  
    ixs->current = xs->first;  
}  
  
bool estaAlFinalDelRecorrido(ListIterator ixs) {  
    return (ixs->current == NULL);  
}
```

Así se garantiza el invariante



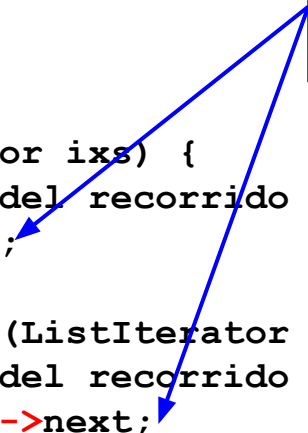
Linked lists e iteradores

■ Solución: iteradores

■ Implementación (2)

```
int elementoActual(ListIterator ixs) {  
    // PRECOND: no está al fin del recorrido  
    return(ixs->current->value);  
}  
  
void PasarAlSiguienteElemento(ListIterator ixs){  
    // PRECOND: no está al fin del recorrido  
    ixs->current = ixs->current->next;  
}  
  
void LiberarIterador(ListIterator ixs) {  
    delete ixs;  
}
```

La precondition es fundamental
para que esto no falle



Linked lists e iteradores

- ❏ Los iteradores se utilizan en muchas estructuras de datos en muchos lenguajes
- ❏ Son una solución elegante para recorrer estructuras
- ❏ En general, implican un recorrido lineal

Árboles en imperativo

Árboles en imperativo

- ❑ ¿Cómo representar árboles en imperativo?
- ❑ La solución más común es usar punteros
- ❑ Es complejo manejarlos en forma abstracta
 - ❑ Por eso suelen manejarse directamente
 - ❑ Es complejo garantizar invariantes
- ❑ Para recorrerlos se suele usar recursión
 - ❑ Alternativamente, se pueden recorrer linealmente
 - ❑ Requiere usar estructuras adicionales (pilas o colas)

Árboles en imperativo

- Implementación de árboles
 - Se implementa el tipo **Tree**, por ser el más simple
 - Otros árboles requieren más trabajo

```
struct TreeNodeSt {  
    int value;  
    TreeNodeSt* left;  
    TreeNodeSt* right;  
};  
  
typedef TreeNodeSt* Tree;  
  
Tree emptyT()          { return NULL;      }  
  
bool isEmptyT(Tree t) { return (t==NULL); }
```

Árboles en imperativo

- Implementación de árboles
 - Se implementa el tipo **Tree**, por ser el más simple
 - Otros árboles requieren más trabajo

```
Tree nodeT(int x, Tree ti, Tree td) {  
    TreeNodeSt* t = new TreeNodeSt;  
    t->value = x;  
    t->left = ti;  
    t->right = td;  
    return t;  
}
```

Árboles en imperativo

❏ ¿Cómo recorrer árboles?

❏ Opción común: con recursión y punteros

```
int sumT(Tree t) {  
    if (isEmptyT(t)) {  
        return 0;  
    } else {  
        return (t->value + sumT(t->left) + sumT(t->right));  
    }  
}
```

❏ En este caso se retorna información sin afectar el árbol

Árboles en imperativo

❏ ¿Cómo recorrer árboles?

❏ Opción común: con recursión y punteros

```
void SuccT(Tree t) {  
    if (!isEmptyT(t)) {  
        t->value++;  
        SuccT(t->left);  
        SuccT(t->right);  
    }  
}
```

❏ En este caso se modifica el árbol

❏ Al modificar los hijos, no hace falta hacer nada más

Árboles en imperativo

- ❏ ¿Y para liberar la memoria?

- ❏ Se debe recorrer el árbol

```
void LiberarTree(Tree t) {  
    if (!isEmptyT(t)) {  
        LiberarTree(t->left);  
        LiberarTree(t->right);  
        delete t;  
    }  
}
```

- ❏ ¡Atención al orden en que se hacen las cosas!

Árboles en imperativo

- ❏ ¿Qué pasa con recorridos más complejos?

- ❏ Opción con recursión y punteros

```
int heightT(Tree t) {  
    if (isEmptyT(t)) {  
        return 0;  
    } else {  
        return (1 + max(heightT(t->left), heightT(t->right)));  
    }  
}
```

- ❏ La altura sigue la estructura del árbol,
y la recursión también

Árboles en imperativo

- ❑ ¿Qué pasa con recorridos más complejos?
- ❑ Opción con recursión y punteros, *á la funcional*

```
List preorderT(Tree t) {  
    if (isEmptyT(t)) { return emptyList(); }  
    else {  
        List xs = preorderT(t->left);  
        List ys = preorderT(t->right);  
        Append(xs, ys);      // Modifica xs, libera ys  
        Cons(t->value, xs); // Modifica xs  
        return xs;  
    }  
}
```

- ❑ ¡Se crean y liberan demasiadas sublistas!

Árboles en imperativo

❏ ¿Qué pasa con recorridos más complejos?

❏ Opción con recursión y punteros, mejorada

```
void AgregarPreorderEn(Tree t, List xs) { // Esta es la función recursiva
    if (!isEmptyT(t)) {
        AgregarPreorderEn(t->right, xs); // Modifica xs
        AgregarPreorderEn(t->left, xs);  // Modifica xs
        Cons(t->value, xs);               // Modifica xs
    }
}

List preorderT(Tree t) {
    List vistosHastaAhora = emptyList();
    AgregarPreorderEn(t, vistosHastaAhora);
    return vistosHastaAhora;
}
```

❏ Una única lista, como acumulador

Árboles en imperativo

- ❑ ¿Cuál es el costo de memoria de recorrer un árbol?
 - ❑ Un *stack frame* por cada nodo
 - ❑ Eso puede ser costoso en espacio
 - ❑ ¿Se podría hacer un recorrido en un único *stack frame*?
 - ❑ El problema es que hay más de un siguiente...
 - ❑ ... por lo que deben guardarse los que faltan
 - ❑ Se usan otras estructuras para eso

Recorridos lineales en árboles

Recorridos lineales en árboles

- ¿Se pueden hacer recorridos lineales sobre un árbol?
 - Solo si el resultado no precisa de la estructura del árbol
 - E.g.: **sumT**, **sizeT**, **preorderT**
 - Si lo precisa, es mucho más complejo
 - E.g.: **levelN**, **heightT**
- ¿Qué se necesita para hacer un recorrido lineal?
 - Hay más de un siguiente en cada paso...
 - ... ¡una lista de siguientes!

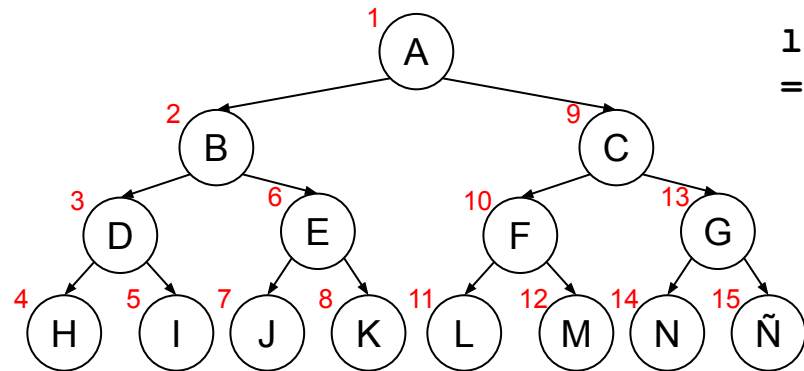
Recorridos lineales en árboles

Un recorrido lineal en un árbol binario

```
void iSumT(Tree t) {
    int totalVisto = 0; Tree actual;
    TList faltanProcesar = emptyTL(); // nunca habrá emptyT en la lista
    if (!isEmptyT(t)) { ConstTL(t, faltanProcesar); }
    while(!isEmptyTList(faltanProcesar)) {
        actual = headTL(faltanProcesar); // actual NO es emptyT
        TailTL(faltanProcesar);
        totalVisto += actual->value;
        if (!isEmptyT(t)) { ConstTL(actual->right, faltanProcesar); }
        if (!isEmptyT(t)) { ConstTL(actual->left , faltanProcesar); }
    }
    LiberarTL(faltanProcesar);
    return(totalVisto);
}
```

Recorridos lineales en árboles

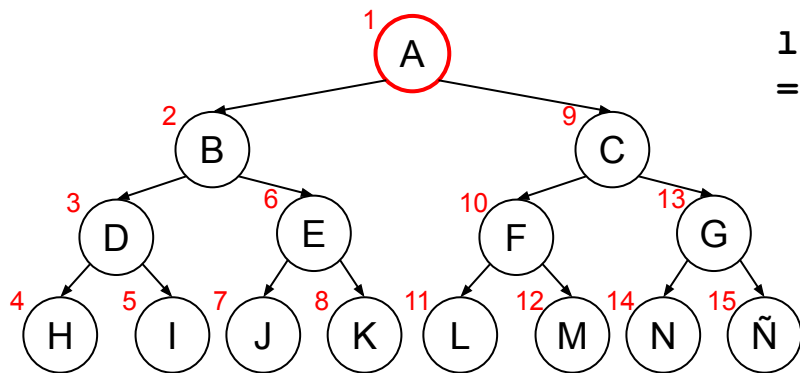
- Recorrido en profundidad (***Depth First Search***, DFS)
 - Primero la rama izquierda, hasta agotar
 - La lista se comporta como una **pila** (Stack)
 - El último que entra es el siguiente hijo izquierdo



`losQueFaltanProcesar`
`= []`

Recorridos lineales en árboles

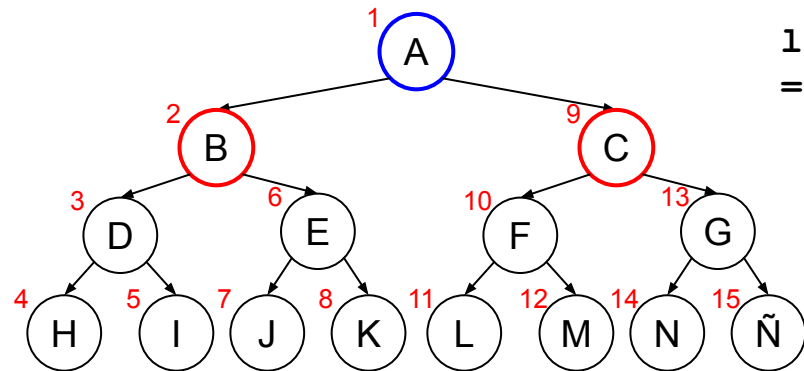
- Recorrido en profundidad (***Depth First Search***, DFS)
 - Primero la rama izquierda, hasta agotar
 - La lista se comporta como una **pila** (Stack)
 - El último que entra es el siguiente hijo izquierdo



`losQueFaltanProcesar`
`= [1]`

Recorridos lineales en árboles

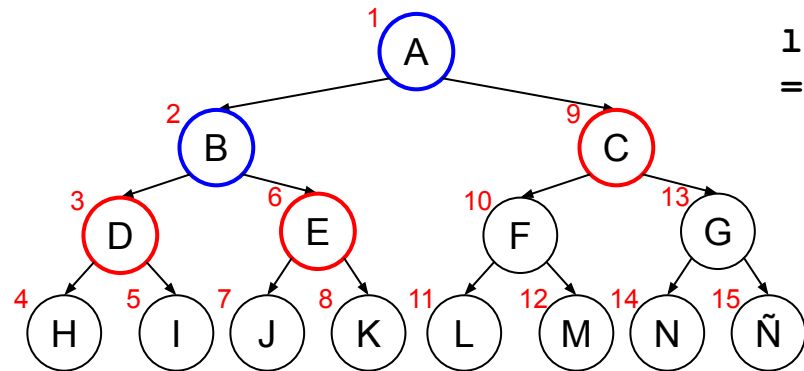
- Recorrido en profundidad (**Depth First Search**, DFS)
 - Primero la rama izquierda, hasta agotar
 - La lista se comporta como una **pila** (Stack)
 - El último que entra es el siguiente hijo izquierdo



`losQueFaltanProcesar`
= [2, 9]

Recorridos lineales en árboles

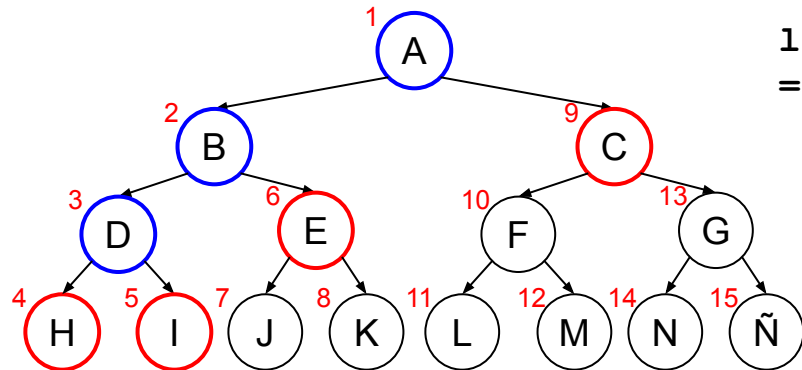
- Recorrido en profundidad (***Depth First Search***, DFS)
 - Primero la rama izquierda, hasta agotar
 - La lista se comporta como una **pila** (Stack)
 - El último que entra es el siguiente hijo izquierdo



`losQueFaltanProcesar`
= [3, 6, 9]

Recorridos lineales en árboles

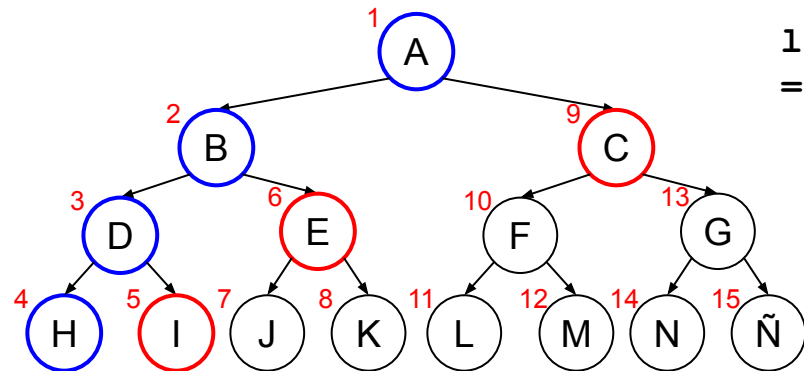
- Recorrido en profundidad (***Depth First Search***, DFS)
 - Primero la rama izquierda, hasta agotar
 - La lista se comporta como una **pila** (Stack)
 - El último que entra es el siguiente hijo izquierdo



`losQueFaltanProcesar`
`= [4,5,6,9]`

Recorridos lineales en árboles

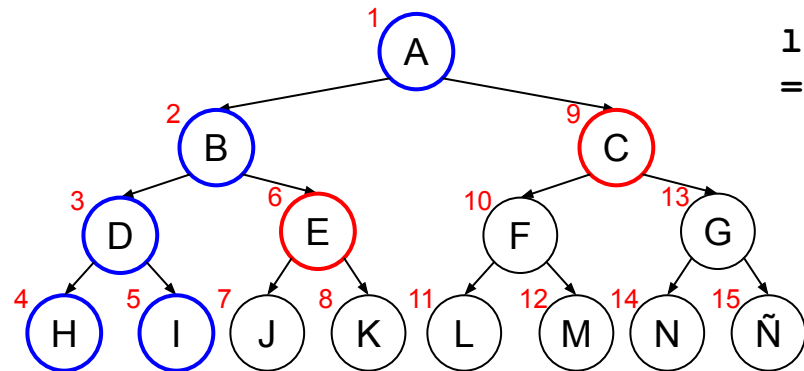
- Recorrido en profundidad (***Depth First Search***, DFS)
 - Primero la rama izquierda, hasta agotar
 - La lista se comporta como una **pila** (Stack)
 - El último que entra es el siguiente hijo izquierdo



`losQueFaltanProcesar`
= [5, 6, 9]

Recorridos lineales en árboles

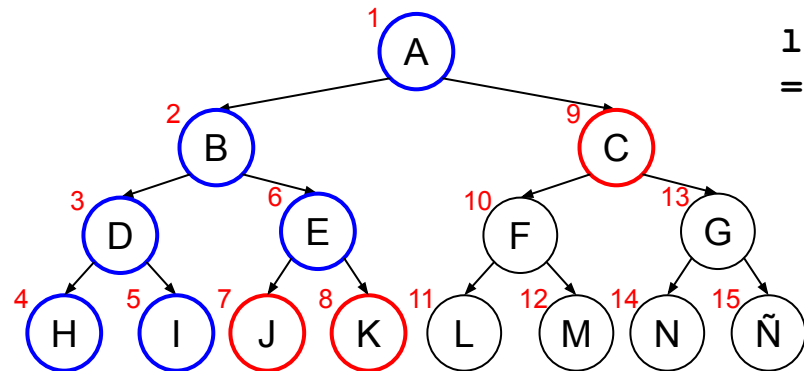
- Recorrido en profundidad (**Depth First Search**, DFS)
 - Primero la rama izquierda, hasta agotar
 - La lista se comporta como una **pila** (Stack)
 - El último que entra es el siguiente hijo izquierdo



`losQueFaltanProcesar`
= [6, 9]

Recorridos lineales en árboles

- Recorrido en profundidad (**Depth First Search**, DFS)
 - Primero la rama izquierda, hasta agotar
 - La lista se comporta como una **pila** (Stack)
 - El último que entra es el siguiente hijo izquierdo



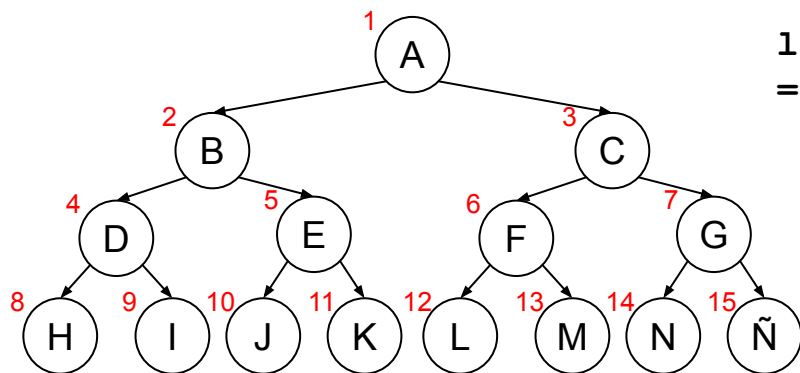
`losQueFaltanProcesar`
`= [7, 8, 9]`

Recorridos lineales en árboles

- ❑ ¿Qué se necesita para hacer un recorrido lineal?
 - ❑ ¡Una lista de siguientes!
- ❑ ¿En qué orden se recorren los subárboles?
 - ❑ Si se usa **Cons** al agregar en la lista
 - ❑ Recorrido en profundidad (***Depth First Search***, DFS)
 - ❑ Primero la rama izquierda, hasta agotar
 - ❑ Si se usa **Snoc** al agregar en la lista
 - ❑ Recorrido a lo ancho (***Breadth First Search***, BFS)
 - ❑ Primero todos los nodos de un nivel

Recorridos lineales en árboles

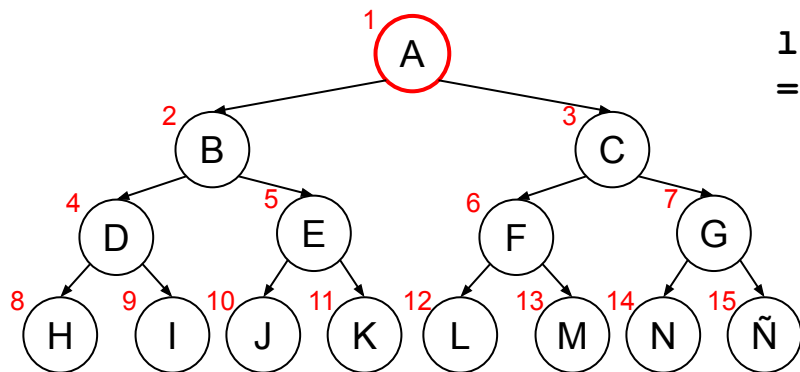
- Recorrido a lo ancho (***Breadth First Search***, BFS)
 - Primero todos los nodos de un nivel
 - La lista se comporta como una **cola** (Queue)
 - El siguiente hijo izquierdo entre detrás de los hermanos



`losQueFaltanProcesar`
`= []`

Recorridos lineales en árboles

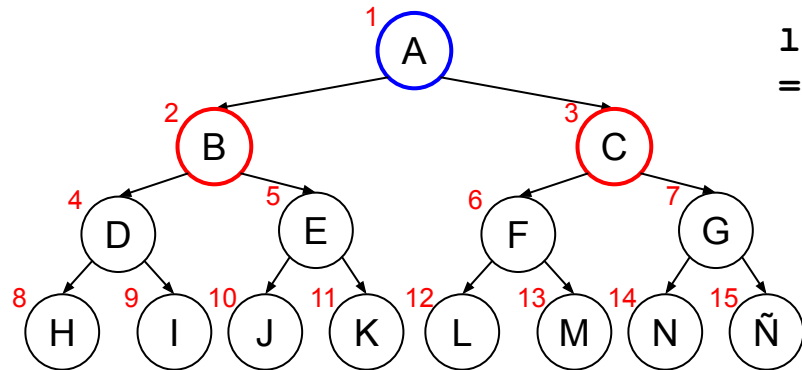
- Recorrido a lo ancho (***Breadth First Search***, BFS)
 - Primero todos los nodos de un nivel
 - La lista se comporta como una **cola** (Queue)
 - El siguiente hijo izquierdo entre detrás de los hermanos



`losQueFaltanProcesar`
`= [1]`

Recorridos lineales en árboles

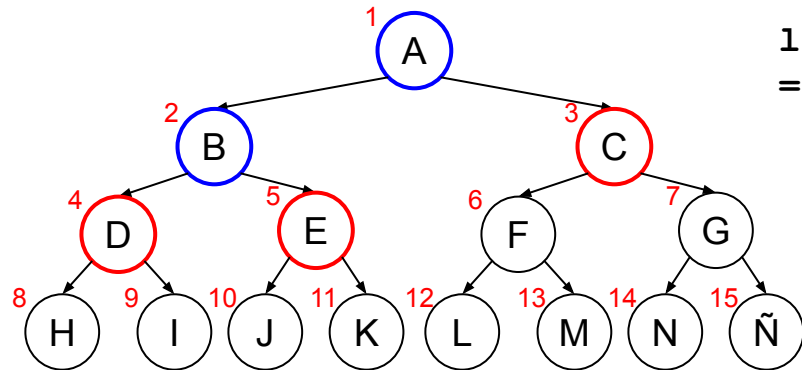
- Recorrido a lo ancho (***Breadth First Search***, BFS)
 - Primero todos los nodos de un nivel
 - La lista se comporta como una **cola** (Queue)
 - El siguiente hijo izquierdo entre detrás de los hermanos



`losQueFaltanProcesar`
`= [2,3]`

Recorridos lineales en árboles

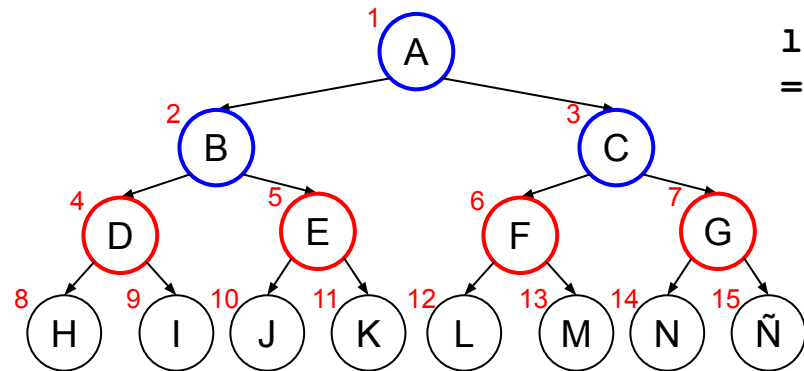
- Recorrido a lo ancho (**Breadth First Search**, BFS)
 - Primero todos los nodos de un nivel
 - La lista se comporta como una **cola** (Queue)
 - El siguiente hijo izquierdo entre detrás de los hermanos



`losQueFaltanProcesar`
= [3,4,5]

Recorridos lineales en árboles

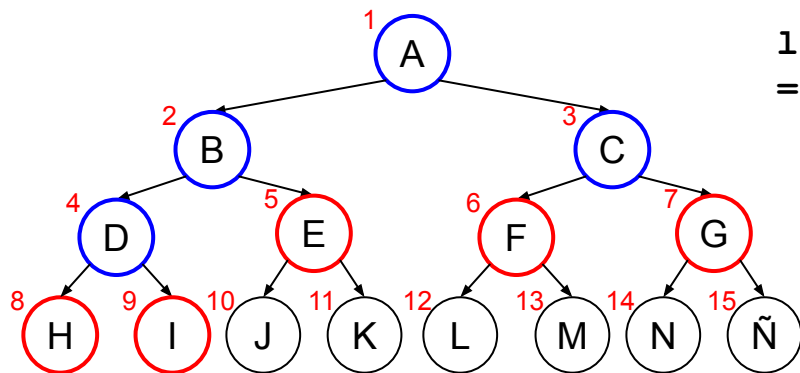
- Recorrido a lo ancho (**Breadth First Search**, BFS)
 - Primero todos los nodos de un nivel
 - La lista se comporta como una **cola** (Queue)
 - El siguiente hijo izquierdo entre detrás de los hermanos



`losQueFaltanProcesar`
`= [4,5,6,7]`

Recorridos lineales en árboles

- Recorrido a lo ancho (***Breadth First Search***, BFS)
 - Primero todos los nodos de un nivel
 - La lista se comporta como una **cola** (Queue)
 - El siguiente hijo izquierdo entre detrás de los hermanos



`losQueFaltanProcesar`
= [5,6,7,8,9]

Recorridos lineales en árboles

- ❑ Los recorridos DFS y BFS adquieren mayor relevancia en estructuras más complejas (como los grafos)
- ❑ Se ven en la materia Algoritmos

Heaps binarios

Heaps binarias en imperativo

- ❏ Vimos cómo usar arrays para representar listas
- ❏ ¿Podrán usarse arrays para representar árboles?

Heaps binarias en imperativo

- ❑ Vimos cómo usar arrays para representar listas
- ❑ ¿Podrán usarse arrays para representar árboles?
 - ❑ ¿Qué características tiene que tener el árbol?
 - ❑ ¿Dónde se ubicaría cada nodo?
 - ❑ ¿Cómo saber donde están los hijos de un nodo?

Heaps binarias en imperativo

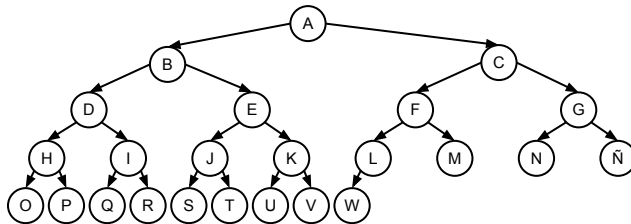
- ❑ Vimos cómo usar arrays para representar listas
- ❑ ¿Podrán usarse arrays para representar árboles?
 - ❑ ¿Qué características tiene que tener el árbol?
 - ❑ ¿Dónde se ubicaría cada nodo?
 - ❑ ¿Cómo saber donde están los hijos de un nodo?
 - ❑ Si el árbol fuese un árbol lleno, se podría...

Heaps binarias en imperativo

- ▣ Vimos cómo usar arrays para representar listas
- ▣ ¿Podrán usarse arrays para representar árboles?
 - ▣ ¿Qué características tiene que tener el árbol?
 - ▣ ¿Dónde se ubicaría cada nodo?
 - ▣ ¿Cómo saber donde están los hijos de un nodo?
 - ▣ Si el árbol fuese un árbol lleno, se podría...
 - ▣ ... ¡y las heaps binarias eran árboles llenos!

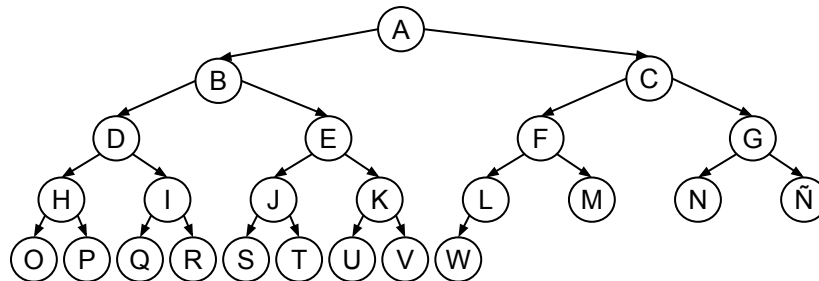
Heaps binarias en imperativo

- ❑ Invariante de **heap** (montículo, pilón):
 - ❑ La raíz es el mínimo de todos los elementos
 - ❑ Los subárboles cumplen el invariante *heap*
- ❑ Invariante de **árbol lleno** (*full tree*):
 - ❑ Todos los niveles, salvo quizás el último están completos
 - ❑ El último no tiene “agujeros” de izquierda a derecha



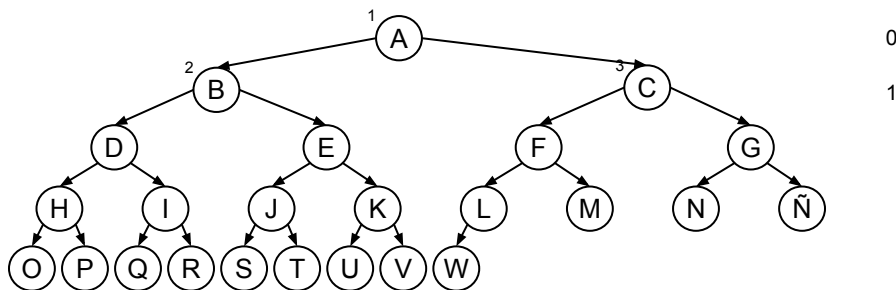
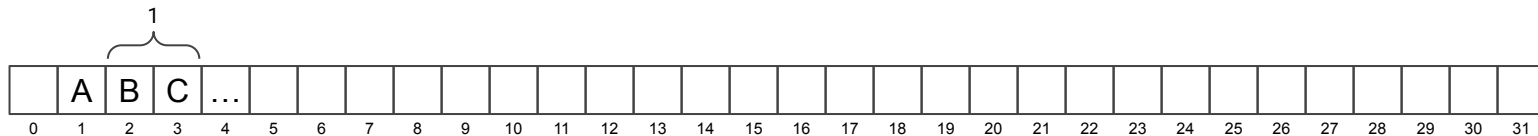
Heaps binarias en imperativo

- ¿Cómo distribuir un árbol lleno en un array?
- La raíz, en la posición 1
- Luego, por niveles, uno detrás de otro



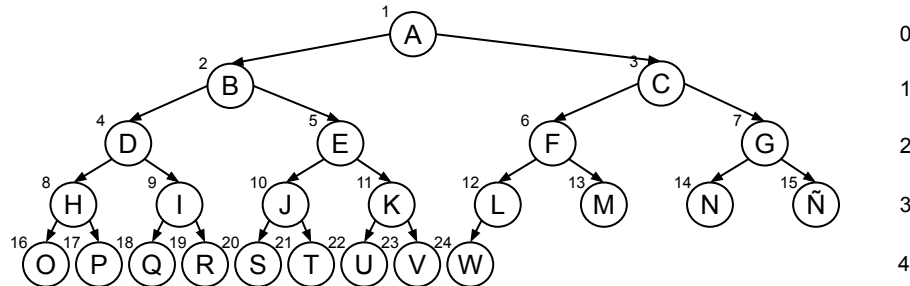
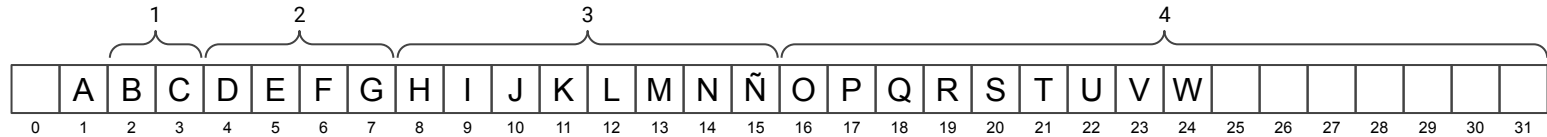
Heaps binarias en imperativo

- ¿Cómo distribuir un árbol lleno en un array?
- La raíz, en la posición 1
- Luego, por niveles, uno detrás de otro



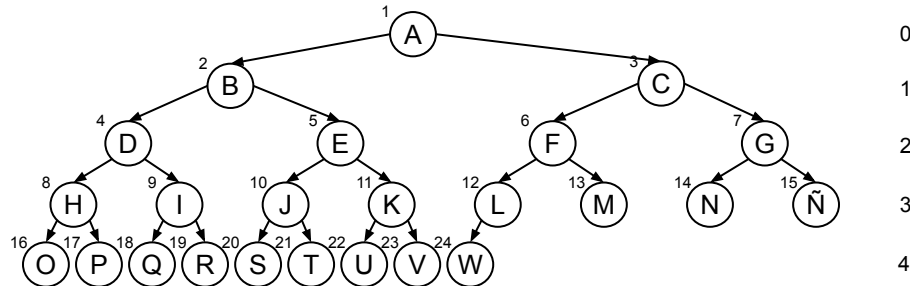
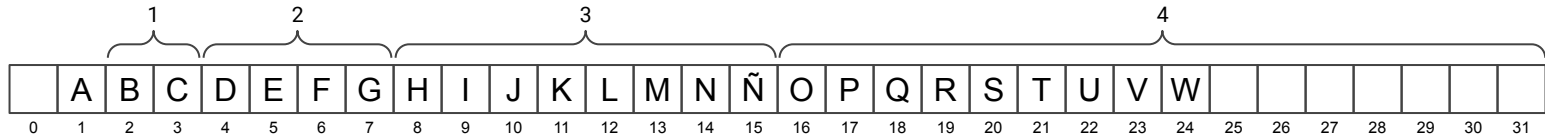
Heaps binarias en imperativo

- ¿Cómo distribuir un árbol lleno en un array?
- La raíz, en la posición 1
- Luego, por niveles, uno detrás de otro



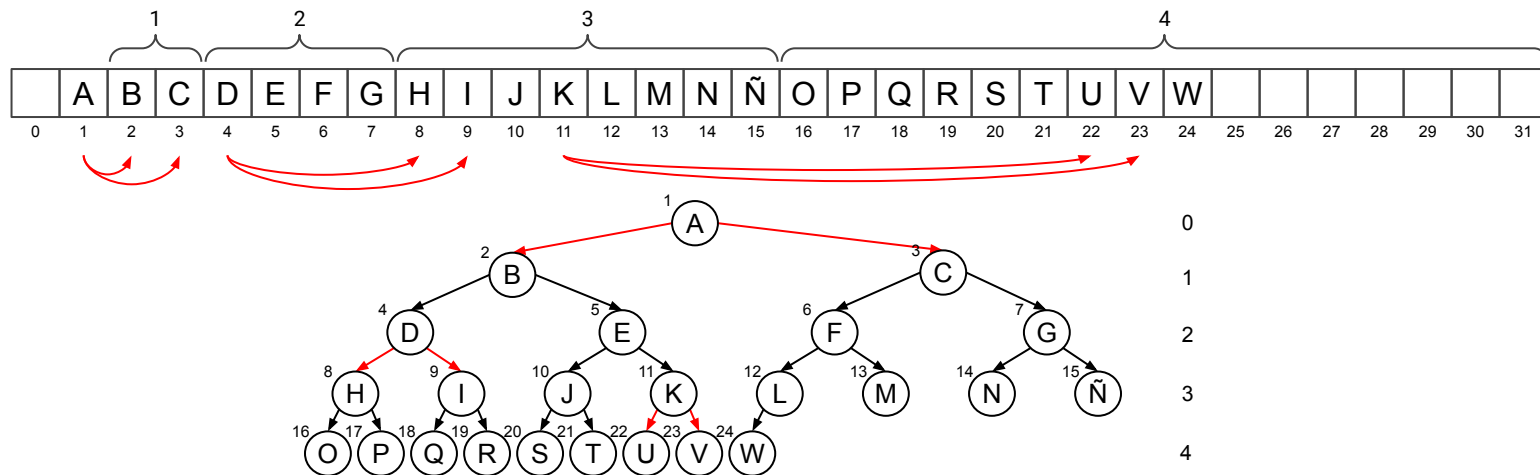
Heaps binarias en imperativo

- La raíz, en la posición 1, luego, por niveles
 - ¿Cómo saber dónde está el hijo de un nodo?
 - ¿Cómo saber dónde está el padre de un nodo?



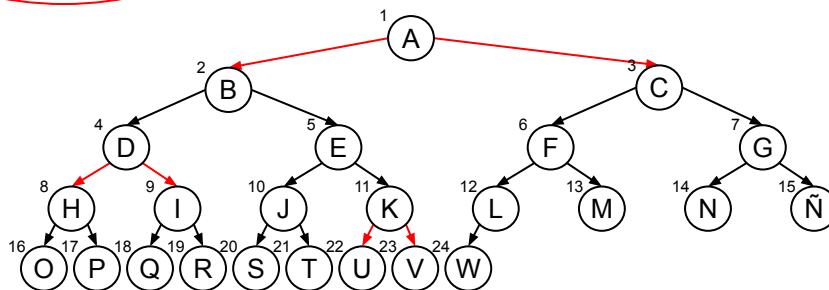
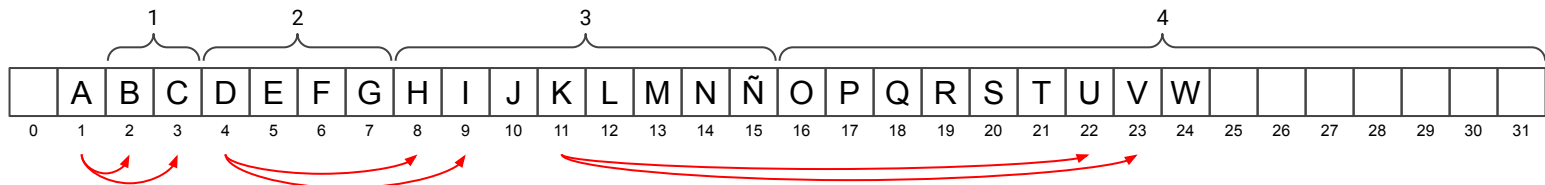
Heaps binarias en imperativo

- La raíz, en la posición 1, luego, por niveles
 - ¿Cómo saber dónde está el hijo de un nodo?
 - ¿Cómo saber dónde está el padre de un nodo?



Heaps binarias en imperativo

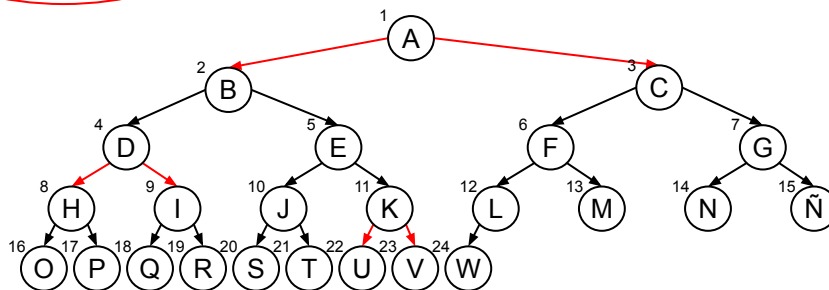
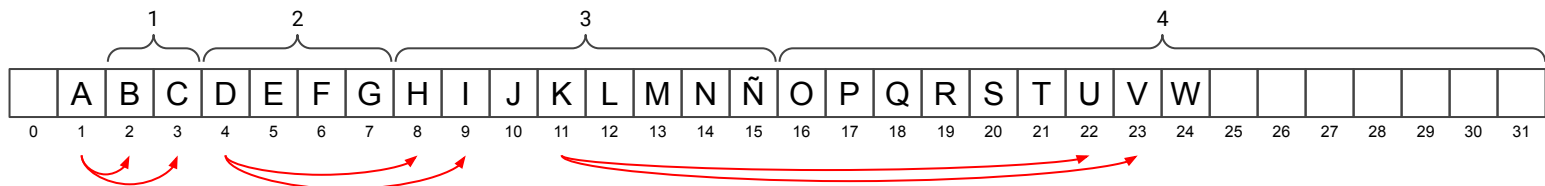
- La raíz, en la posición 1, luego, por niveles
 - ¿Cómo saber dónde está el hijo de un nodo?
 - ¿Cómo saber dónde está el padre de un nodo?



0 Nodo : i
1 Hijo izq.: $2*i$
2 Hijo der.: $2*i+1$
3
4

Heaps binarias en imperativo

- La raíz, en la posición 1, luego, por niveles
 - ¿Cómo saber dónde está el hijo de un nodo?
 - ¿Cómo saber dónde está el padre de un nodo?



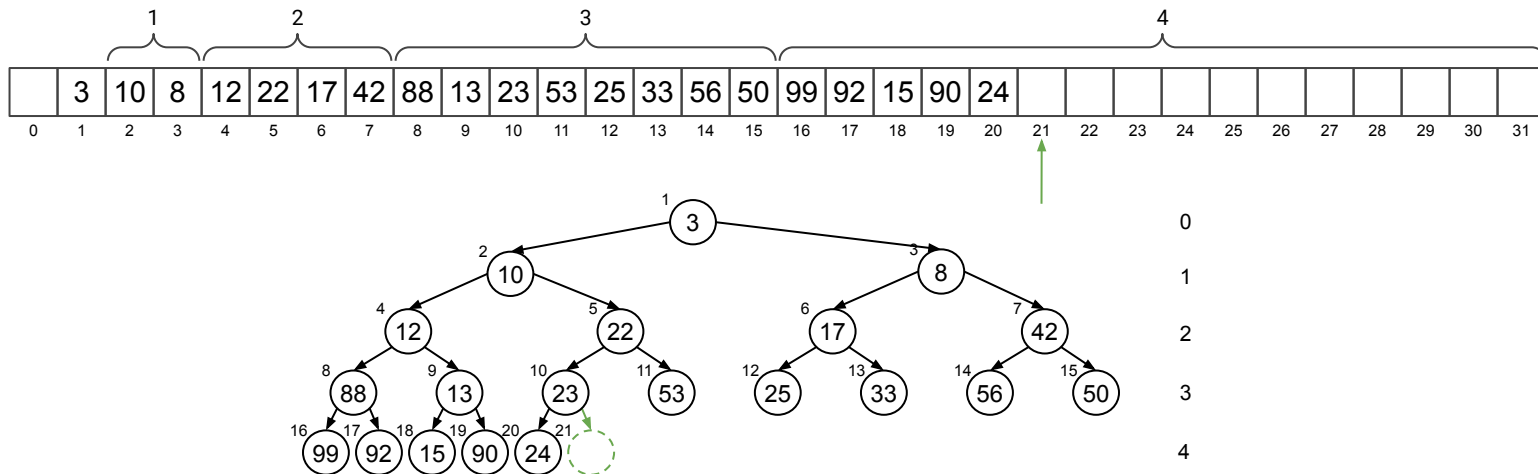
0 Nodo : i
1 Hijo izq. : $2*i$
2 Hijo der. : $2*i+1$
3 Padre : $i/2$
4

Heaps binarias en imperativo

- ¿Cómo pensar las operaciones de heaps?
 - Insertar
 - Se agregar en la posición nueva y se flota
 - Borrar
 - Se traslada la última posición a la raíz, y se hunde
 - Otras operaciones son más sencillas

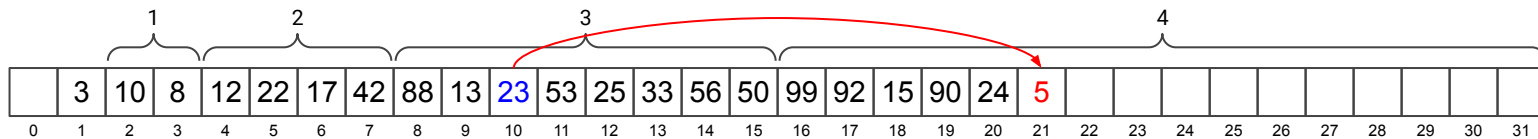
Heaps binarias en imperativo

- ¿En qué lugar debe insertarse para que siga lleno?
- En la primera posición libre
- Y de ahí debe flotarse, para que siga siendo heap

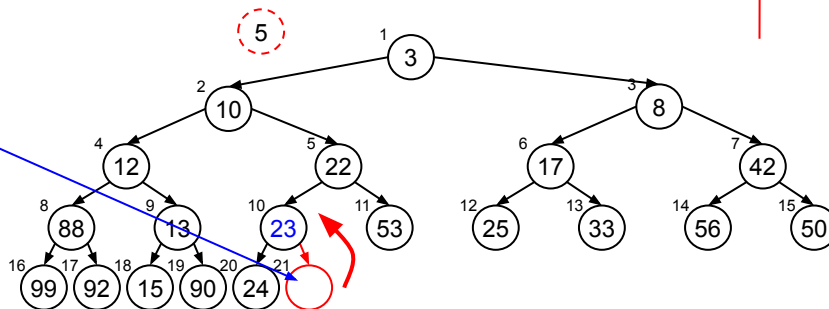


Heaps binarias en imperativo

- ¿En qué lugar debe insertarse para que siga lleno?
- En la primera posición libre
- Y de ahí debe flotarse, para que siga siendo heap



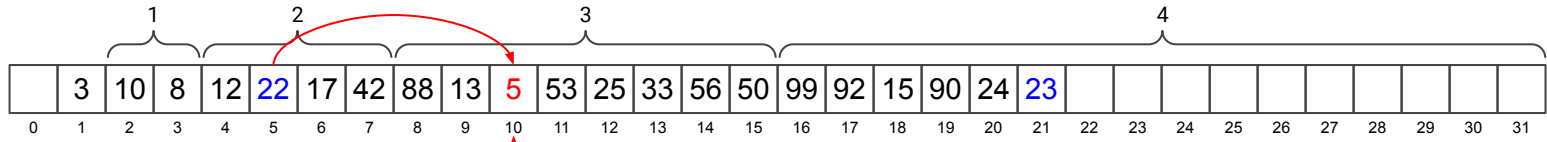
Hay que hacerlo
“flotar” hacia su
padre...



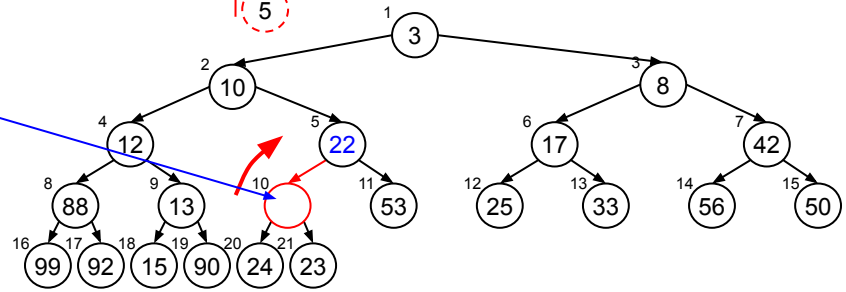
0 Nodo : i = 21
1 Padre : $i/2$ = 10
2
3
4

Heaps binarias en imperativo

- ¿En qué lugar debe insertarse para que siga lleno?
- En la primera posición libre
- Y de ahí debe flotarse, para que siga siendo heap



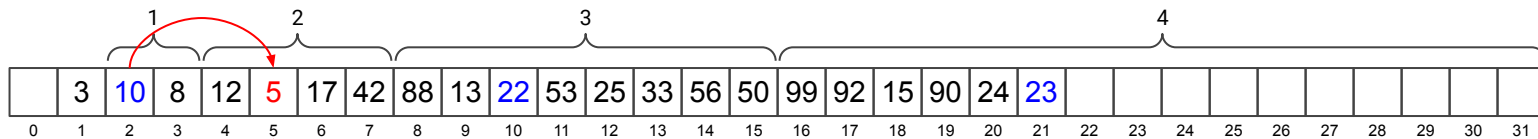
Hay que hacerlo "flotar" hacia su padre...



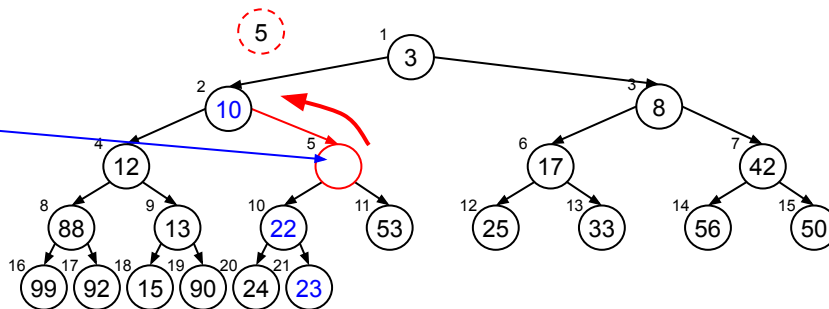
0 Nodo : i = 10
1 Padre : $i/2$ = 5
2
3
4

Heaps binarias en imperativo

- ¿En qué lugar debe insertarse para que siga lleno?
- En la primera posición libre
- Y de ahí debe flotarse, para que siga siendo heap



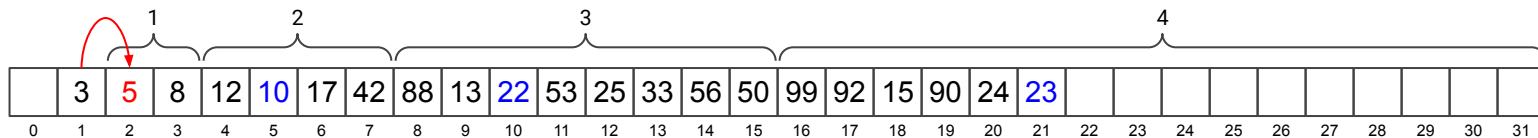
Hay que hacerlo
“flotar” hacia su
padre...



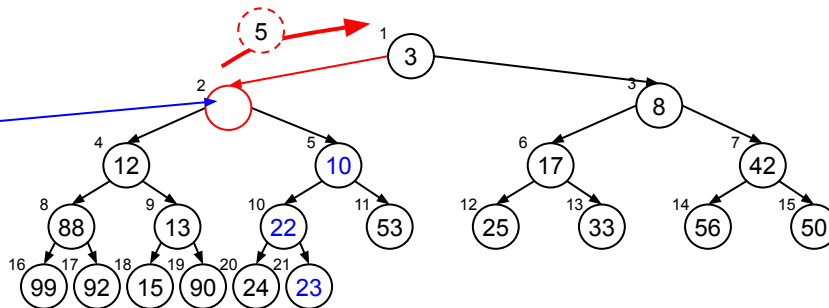
0 Nodo : i = 5
1 Padre : $i/2$ = 2
2
3
4

Heaps binarias en imperativo

- ¿En qué lugar debe insertarse para que siga lleno?
- En la primera posición libre
- Y de ahí debe flotarse, para que siga siendo heap



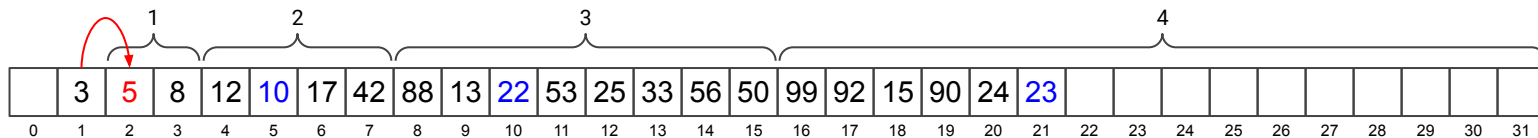
... hasta que su
padre sea
menor que él



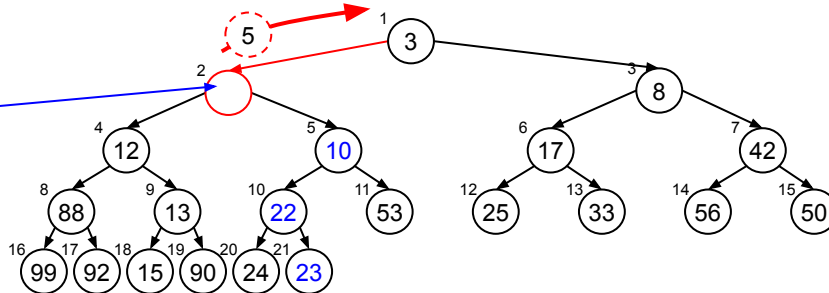
0 Nodo : i = 2
1 Padre : $i/2$ = 1
2
3
4

Heaps binarias en imperativo

- ¿En qué lugar debe insertarse para que siga lleno?
- En la primera posición libre
- Y de ahí debe flotarse, para que siga siendo heap



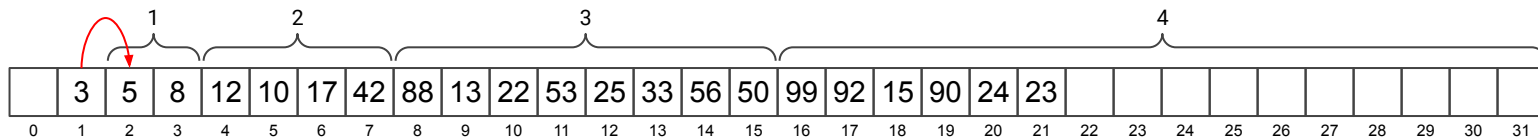
... hasta que su padre sea menor que él



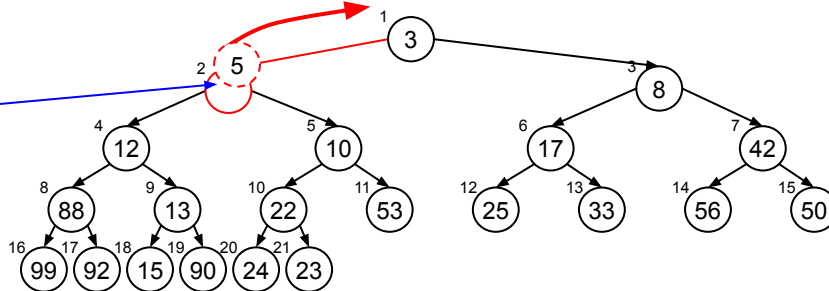
0 Nodo : i = 2
1 Padre : $i/2$ = 1
2
3
4

Heaps binarias en imperativo

- ¿En qué lugar debe insertarse para que siga lleno?
- En la primera posición libre
- Y de ahí debe flotarse, para que siga siendo heap



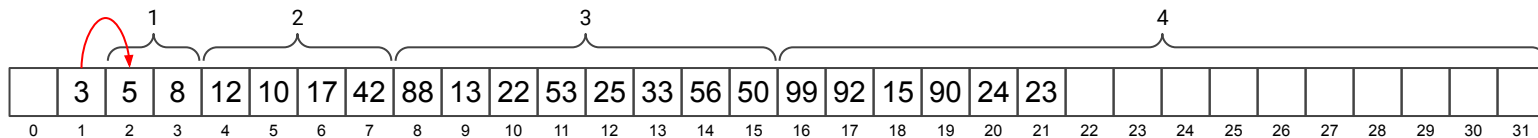
... hasta que su padre sea menor que él



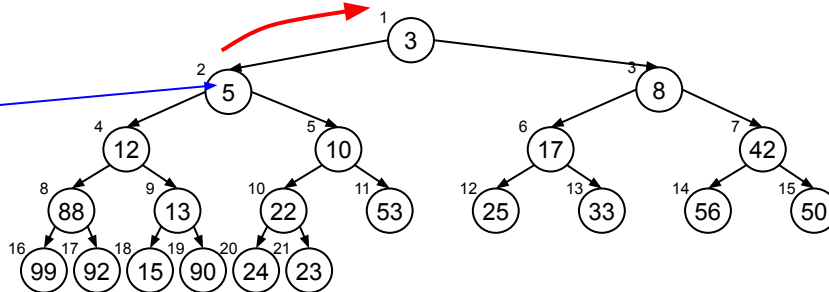
0 Nodo : i = 2
1 Padre : $i/2$ = 1
2
3
4

Heaps binarias en imperativo

- ¿En qué lugar debe insertarse para que siga lleno?
- En la primera posición libre
- Y de ahí debe flotarse, para que siga siendo heap



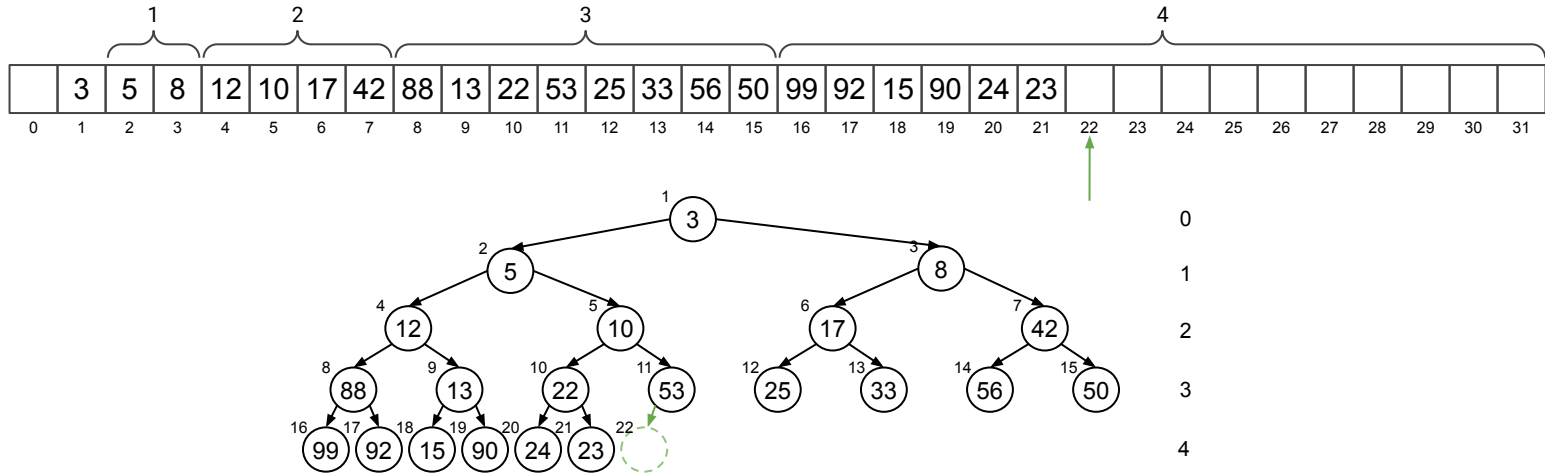
... hasta que su padre sea menor que él



0 Nodo : i = 2
1 Padre : $i/2$ = 1
2
3
4

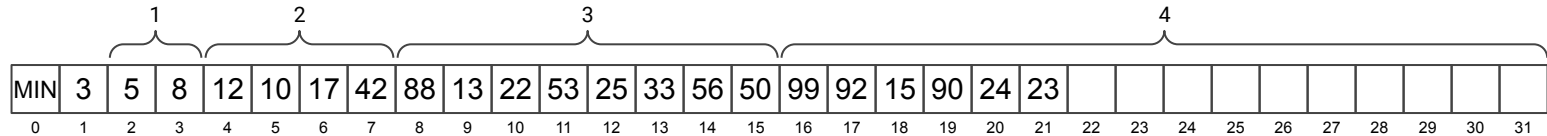
Heaps binarias en imperativo

- La nueva posición a insertar es 1 más que la anterior

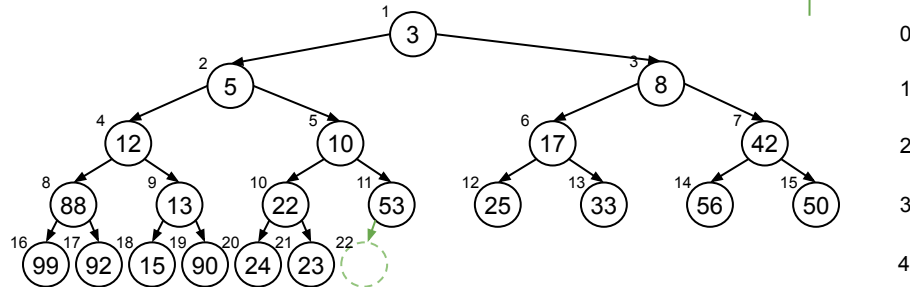


Heaps binarias en imperativo

- ❏ Caso de borde: cuando flotar llega a la raíz
 - ❏ Para eso se usa un “centinela” en la posición 0
 - ❏ La comparación contra él siempre falla



El centinela es la menor de todas las prioridades



Heaps binarias en imperativo

■ Código para heaps binarias en imperativo

■ Interfaz (**Heap.h**)

```
struct BinHeapHeaderSt;  
typedef BinHeapHeaderSt* BinHeap; // INV.REP.: el puntero NO es NULL  
  
BinHeap emptyHeap();  
void InsertH(int x, BinHeap h);  
bool isEmptyHeap(BinHeap h);  
int findMin(BinHeap h);  
void deleteMin(BinHeap h);  
  
BinHeap crearHeap(int* elements, int cant);
```

Heaps binarias en imperativo

■ Código para heaps binarias en imperativo

■ Implementación (**Heap.cpp**, 1)

```
struct BinHeapHeaderSt{
    int  maxSize;    // INV.REP.: curSize < maxSize
    int  curSize;
    int* elems;
};

BinHeap emptyHeap() {
    BinHeapHeaderSt* h = new BinHeapHeaderSt;
    h->maxSize = 16; h->curSize = 0;
    h->elems = new int[h->maxSize];
    h->elems[0] = INT_MIN;    // Macro definida en limits.h
    return h;
}
```

Heaps binarias en imperativo

■ Código para heaps binarias en imperativo

■ Implementación (**Heap.cpp**, 2)

```
bool isEmptyHeap(BinHeap h) {  
    return(h->curSize==0);  
}  
  
int findMin(BinHeap h) {  
    // PRECOND: la heap no está vacía  
    return(h->elems[1]);  
}
```

Heaps binarias en imperativo

■ Código para heaps binarias en imperativo

■ Implementación (**Heap.cpp**, 3)

```
void InsertH(int x, BinHeap h) {
    if(h->curSize==h->maxSize-1) { AumentarEspacio(h); }
    // Flotar el nuevo elemento (haciendo lugar para él)
    int curNode = ++h->curSize;
    while(x < h->elems[curNode/2]) {
        h->elems[curNode] = h->elems[curNode/2];
        curNode /= 2;
    }
    h->elems[curNode] = x;
}
```


Heaps binarias en imperativo

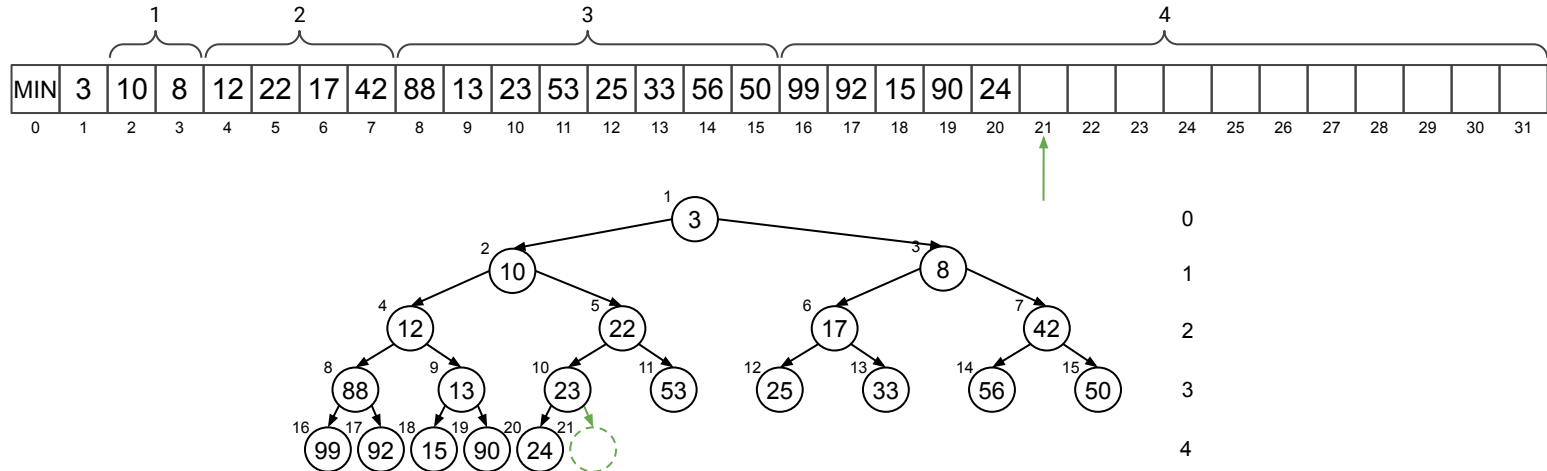
■ Código para heaps binarias en imperativo

■ Implementación (**Heap.cpp**, 4)

```
// Auxiliar para ampliar el espacio de elementos de la heap
void AumentarEspacio(BinHeap h) {
    int* newElements = new int[h->maxSize*2];
    for(int i=0;i<=h->curSize;i++) {
        newElements[i] = h->elems[i];
    }
    delete h->elems;
    h->maxSize *= 2;
    h->elems = newElements;
}
```

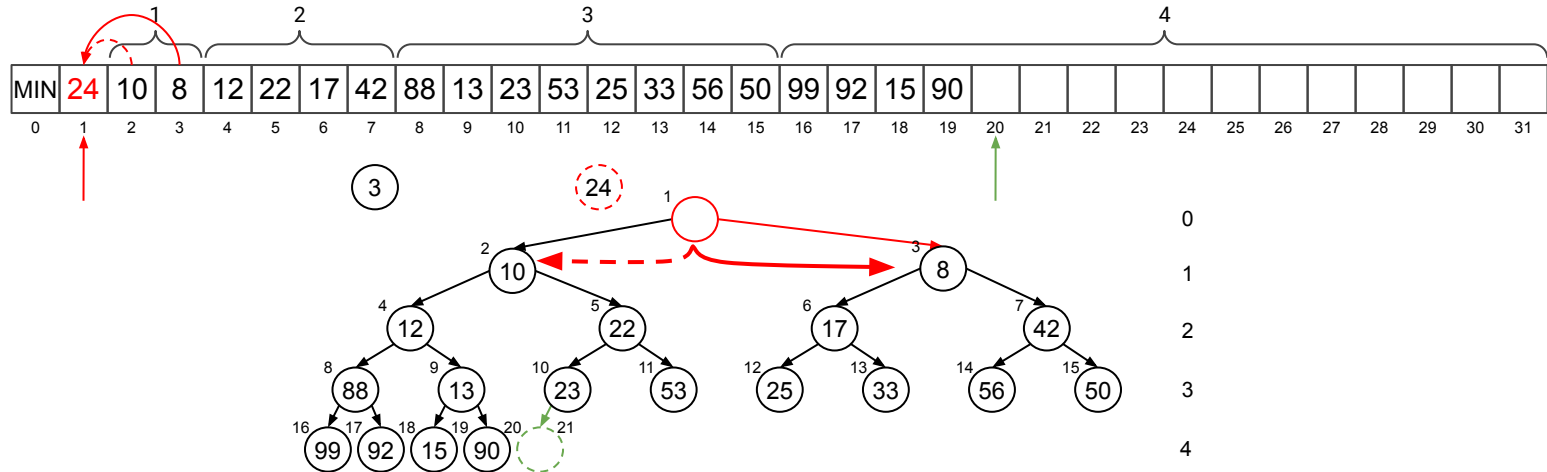
Heaps binarias en imperativo

- Borrado del mínimo
 - Se elimina el mínimo, y se pasa el último a la raíz
 - Se “hunde” con respecto a sus hijos



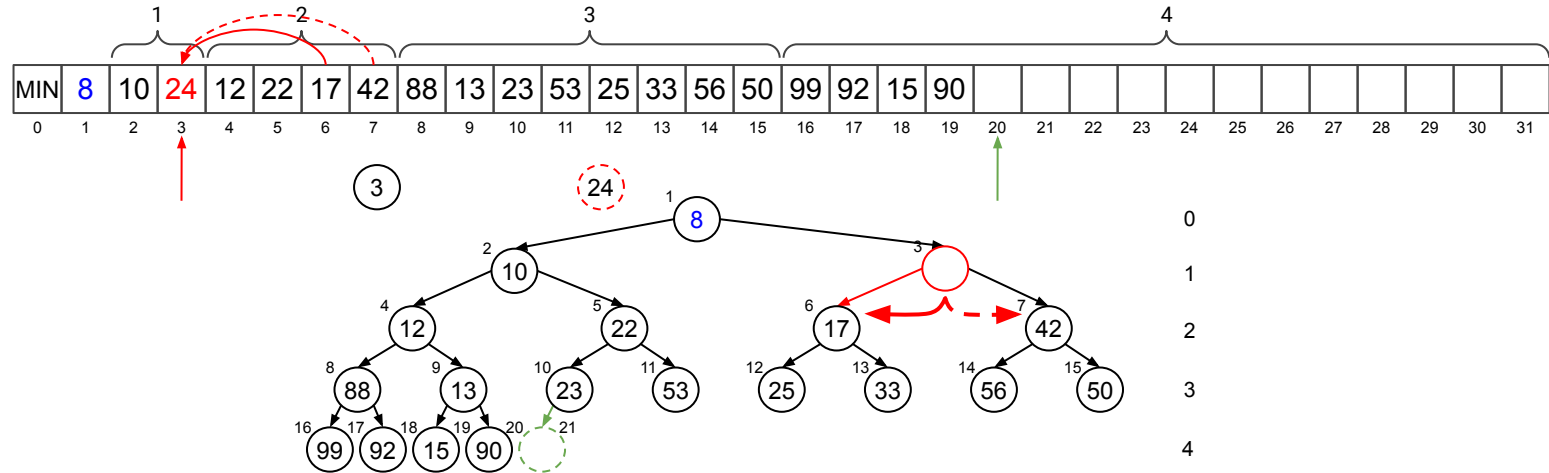
Heaps binarias en imperativo

- Borrado del mínimo
 - Se elimina el mínimo, y se pasa el último a la raíz
 - Se “hunde” con respecto a sus hijos



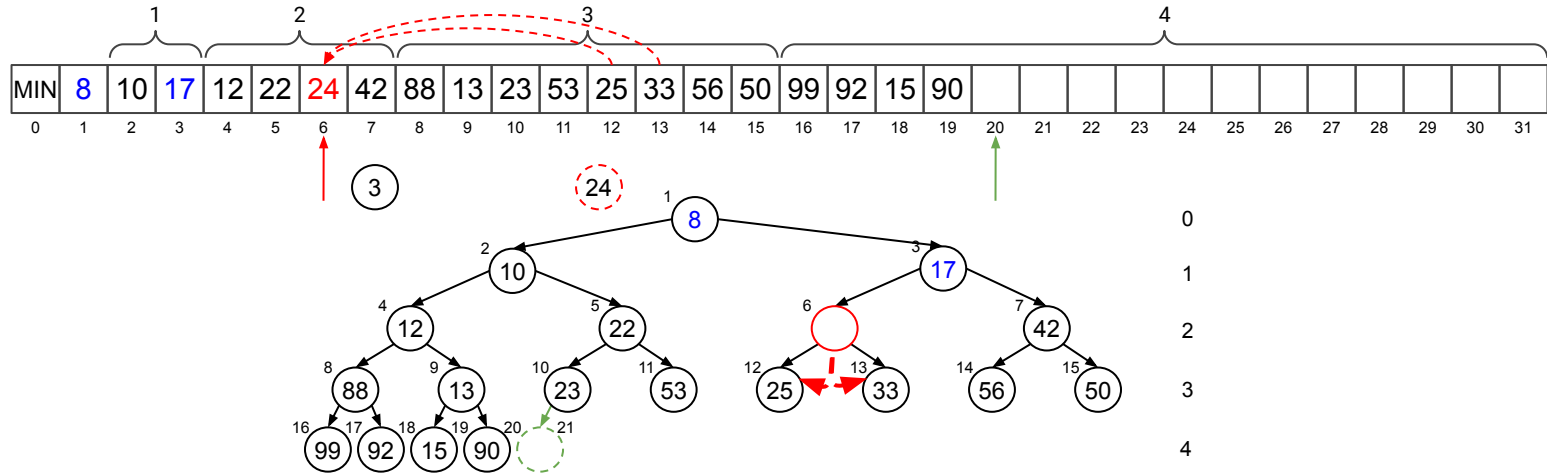
Heaps binarias en imperativo

- Borrado del mínimo
 - Se elimina el mínimo, y se pasa el último a la raíz
 - Se “hunde” con respecto a sus hijos



Heaps binarias en imperativo

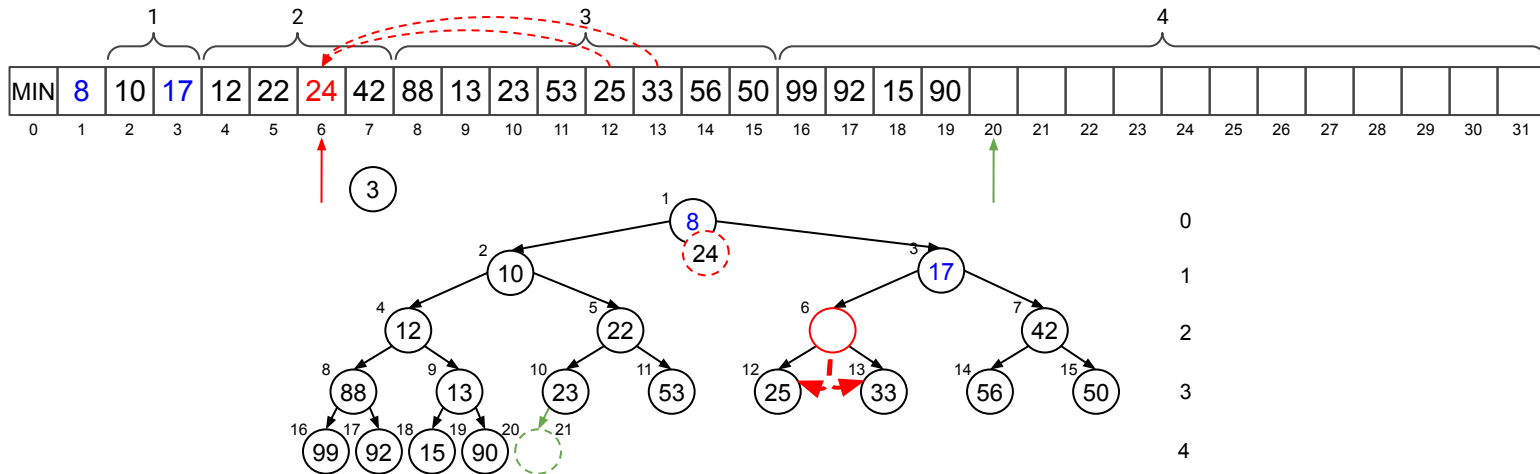
- Borrado del mínimo
 - Se elimina el mínimo, y se pasa el último a la raíz
 - Se “hunde” con respecto a sus hijos



Heaps binarias en imperativo

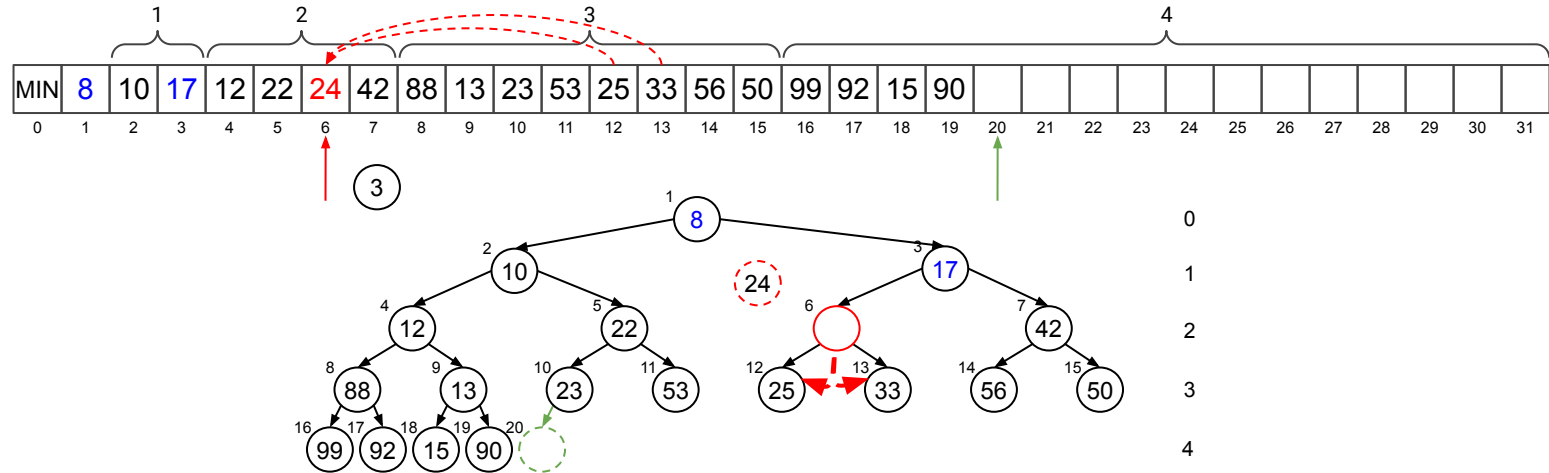
Borrado del mínimo

- Se elimina el mínimo, y se pasa el último a la raíz
- Se “hunde” con respecto a sus hijos



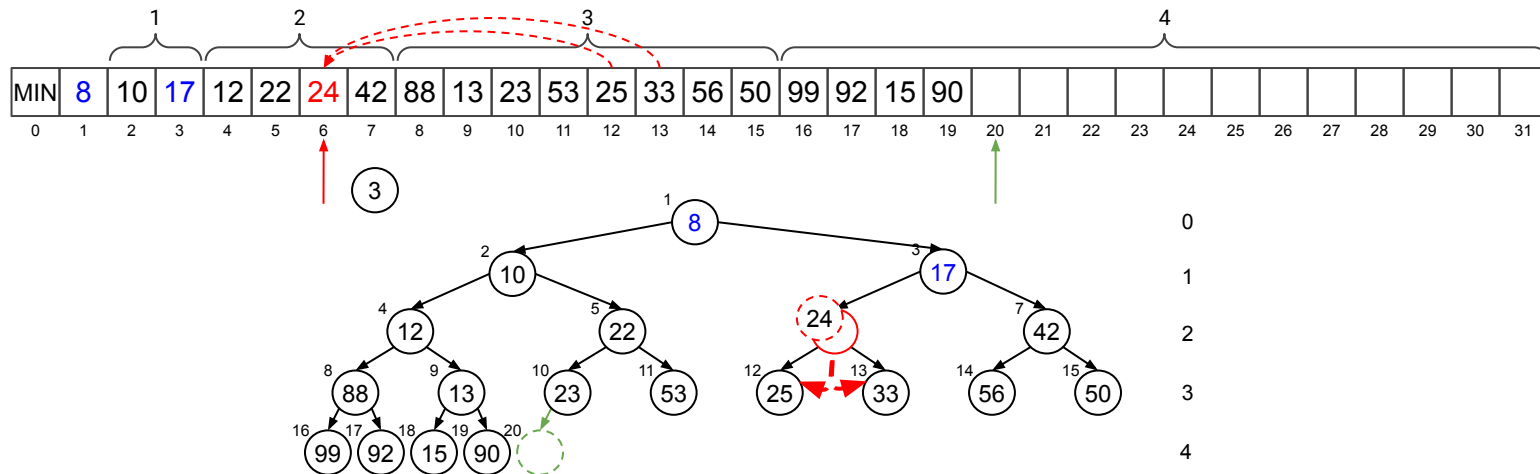
Heaps binarias en imperativo

- Borrado del mínimo
 - Se elimina el mínimo, y se pasa el último a la raíz
 - Se “hunde” con respecto a sus hijos



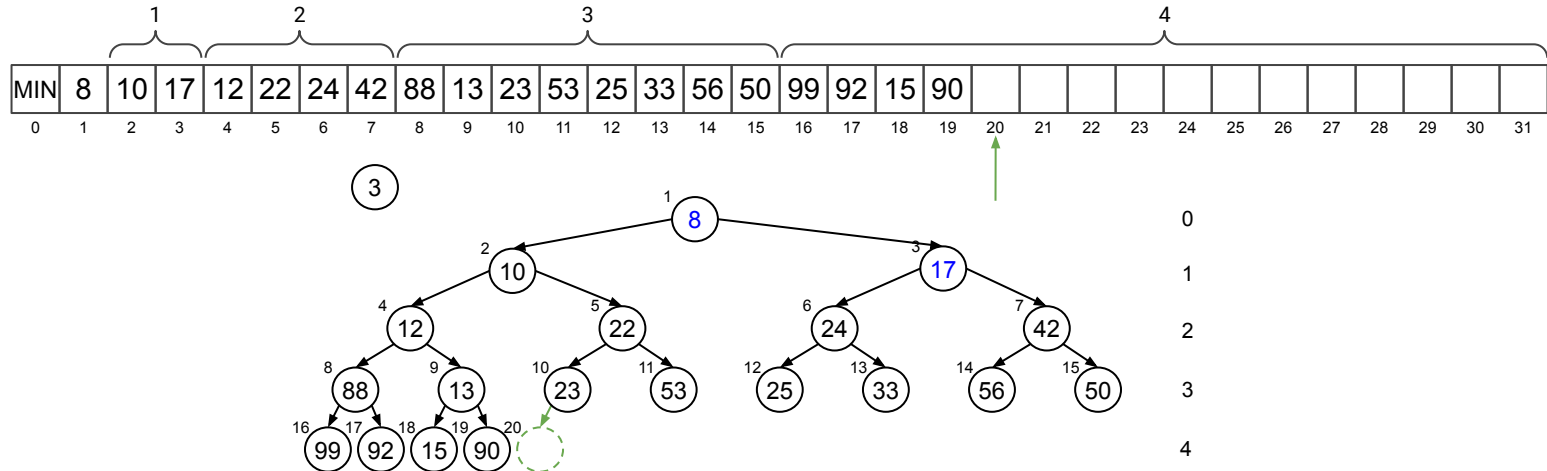
Heaps binarias en imperativo

- Borrado del mínimo
 - Se elimina el mínimo, y se pasa el último a la raíz
 - Se “hunde” con respecto a sus hijos



Heaps binarias en imperativo

- Borrado del mínimo
 - Se elimina el mínimo, y se pasa el último a la raíz
 - Se “hunde” con respecto a sus hijos



Heaps binarias en imperativo

❏ Código para heaps binarias en imperativo

❏ Implementación (**Heap.cpp**, 5)

```
void DeleteMin(BinHeap h) { // PRECOND: h->curSize > 0
    int child; int curNode;
    int last = h->elems[h->curSize--];
    for(curNode=1; curNode*2 <= h->curSize; curNode=child) {
        child = curNode*2;
        if ((child != h->curSize) // Elige el hijo más chico
            && (h->elems[child+1] < h->elems[child])) { child++; }
        // Baja un nivel, si el hijo más chico es más chico que last
        if (last > h->elems[child]) { h->elems[curNode] = h->elems[child]; }
        else { break; } // O termina (evitando comparar dos veces lo mismo)
    }
    h->elems[curNode] = last;
}
```

Heaps binarias en imperativo

- ❑ Es posible hacer otras operaciones
- ❑ La eficiencia obtenida es alta
 - ❑ Mediante uso de arrays
 - ❑ Mediante trucos de implementación

Resumen

Resumen

- ❏ Linked lists
 - ❏ Estructura de listas en memoria Heap
 - ❏ Inserción, borrado y append eficientes
 - ❏ Acceso lineal a otros elementos
 - ❏ Requiere consideraciones para recorridos
- ❏ Árboles binarios en imperativo
 - ❏ Complejos de abstraer
 - ❏ Algunas funciones admiten recorridos iterativos
 - ❏ DFS, BFS

Resumen

- ❑ Binary heaps en imperativo
 - ❑ Usan arreglos para uso eficiente de la memoria
 - ❑ Prácticamente no se usan subtarefas (por eficiencia)
 - ❑ Se requiere manejo de índices
 - ❑ Se utilizan varios trucos de C para optimizar más
- ❑ Conclusiones
 - ❑ Programar bien en imperativo no es trivial
 - ❑ El manejo de memoria automático puede ayudar