1 - Tipos abstractos de datos

Tipo de datos

- Colección de valores que toman los datos y operaciones que se pueden realizar sobre ellos
    - Básicos: int, float char…
    - Definidos por el programador: enumerados, subrango
- Opacos: Representación invisible al programador, operaciones predefinidas
- Tipos estructurados: Genericidad, pero riesgo de crear valores sin semántica (ej:fecha)

Tipo abstracto de datos

- Son opacos, por lo que su implementación se realiza en un ámbito inaccesible al resto de los programas
- Su definición es independiente de la representación
- El conjunto de operaciones debe permitir generar cualquier valor del tipo
- Partes diferenciadas:
    - **Especificación (interfaz):** Conocida por el usuario. (archivo.h)
        - Contiene nombre del tipo y especificación de operaciones
        - Parte sintáctica (nombre operación) y semántica (funcionalidad)
        - Debe ser estable y lo más sencillo posible
        - Se define el TAD como un **puntero al vacío** (typedef void *tad)
    - **Implementación:** Conocida por el programador. (archivo.c)
        - Representación del tipo mediante otros tipos y realización de operaciones en un lenguaje de programación
        - Suele ser compleja y propensa a cambiar

Clasificación de TADs

- **Simples:** Cambian su valor pero no estructura. Ocupan siempre el mismo espacio (enteros, reales…)
- **Contenedores:** Cambian valor y estructura. Varía nº de elementos y espacio de almacenamiento (listas, colas, pilas, árboles, grafos)

- **Inmutables:** No existen operaciones de modificación, los casos no pueden modificarse. Se definen al ser asignados.
    - Representación inmutable o mutable (ej mutable: ½ se puede representar como 2/4)
- **Mutables:** Sus casos se pueden modificar mediante operaciones de modificación

Especificación de TADs

- **Informal:** Lenguaje natural, ambiguas pero sencillas de escribir y entender
    - Cabecera: Nombre de las operaciones
    - Descripción: se explica en qué consiste la abstracción sin entrar en detalles. Se puede hacer en términos de otros tipos más familiares y utilizar gráficos o abstracciones matemáticas
        - Se puede incluir si es inmutable o mutable
    - Especificación de las operaciones:
        - Cabecera: Nombre de la operación y tipos y nombre de las variables a utilizar. ej: suma(a,b:racional) devuelve (racional)
        - Cuerpo: Información semántica
            - Requerimientos: Dominio de procedimiento. Si se pueden omitir se considera abstracción total. Es responsabilidad del usuario que se cumplan. [opcional]
            - Modifica: Entradas que cambian de valor [opcional]
            - Efecto: Salidas producidas y las modificaciones producidas en el previo apartado, si se cumplen los requerimientos.

- **Formal:** Lenguaje algebraico, breves y precisas pero complejas de entender. Permite verificación formal respecto al código.
    - **Tipo:** Nombre del TAD
    - **Sintaxis:** Forma de las operaciones
      [nombre función (tipo argumentos) → tipo resultado]
    - **Semántica:** Comportamiento de las operaciones en algunos casos concretos para demostrar su funcionamiento
        - Algunas funciones, denominadas axiomas o constructoras no requieren reglas semánticas y se usan para crear cualquier elemento del tipo. Se marcan con * en la sintaxis
        - El resultado puede ser recursivo o contener referencias a algunos otros tipos (‘cierto’, ‘falso’, ‘error’)
        - Las condiciones se escriben como
          si condicion → valor_si_cierto|valor_si_falso

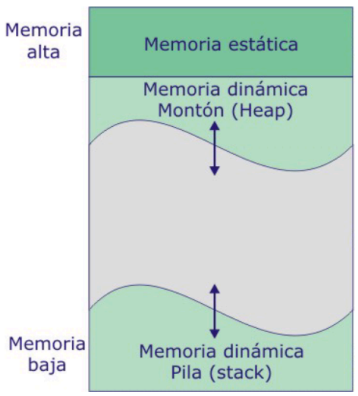
Ejemplo especificación formal [implementación]

- **Tipo:** bolsa(elemento)
- **Sintaxis:**
    - *bolsavacia[1] → bolsa
    - *poner(bolsa,elemento) → bolsa
    - esvacia(bolsa) → booleano
    - cuantos(bolsa,elemento) → natural
- **Semántica:** ∀b€bolsa, ∀e,f €elemento
    - esvacia(bolsavacia) => cierto
    - esvacia(poner(b,e)) => falso
    - cuantos(bolsavacia, e) => cero
    - cuantos(poner(b,f), e) => si f=e => sucesor(cuantos(b,e))[2] |cuantos(b,e)
        - Ejemplo: cuantos(poner(poner(poner(bolsavacia,34),34),27),34) devolvería sucesor(sucesor(0)) = 2

---

[1] non é un puntero, indica que esto é unha funcion constructora e polo tanto non ten semántica
[2] sucesor(a): equivalente a a+1

Modelo lógico de memoria



- **Memoria alta:** Variables globales e instrucciones. Se asigna al comenzar la ejecución
- **Heap:** Variables con memoria asignada en tiempo de ejecución (malloc), crece y se reduce
- **Stack:** Variables locales y datos relacionados con llamada a funciones, crece y se reduce

Asignación dinámica de memoria

- **Variables dinámicas:** Variables cuyo espacio de almacenamiento se asigna en tiempo de ejecución. Se almacenan en heap
    - No tienen nombre, se accede a ellas con punteros
    - Se crean y liberan con malloc y free
    - Ej: (int *) malloc(3*sizeof(int)). La primera parte es el type cast que indica el uso que se le dará a la memoria, la segunda parte es la cantidad de memoria a reservar [en este caso, 3 enteros]
- **Variables puntero:** Apuntan a variables dinámicas
    - Si es global puede ser estático, si es local es dinámico

typedef

- Permite crear un nuevo nombre para otro tipo ej: typedef int tipo; permite usar tipo para crear un int
- También permite abreviar el nombre de una estructura, por ejemplo typedef struct datos {...} sdatos define una estructura ‘struct datos’ y pasa a llamarle ‘sdatos’.
- Tras crear un TAD (ej: una estructura) se suele usar **typedef** para darle nombre a un puntero a esta estructura, que será el usado por las funciones

Punteros dinámicos

- p = (int **) malloc (sizeof (int*)); reserva espacio de memoria para un puntero int*.
- Luego, se asigna este puntero: *p = (int*) malloc (sizeof(int)). De esta forma se crea el entero: **p=14;

- Se puede usar por ejemplo para matrices de datos
- Tras ejecución se libera primero free(*p) y después free(p)

Técnicas de implementación

- **Tipos de implementación** según asignación de memoria:
    - Estática: asignación en tiempo de compilación. Ej: array
    - Dinámica: asignación en tiempo de ejecución
- **Tipos de representaciones:**
    - Contiguas: Direcciones de memoria contiguas
    - Enlazadas: Direcciones dispersas, enlazadas mediante punteros

Representaciones contiguas y enlazadas

- Si la cantidad de casos del TAD es fija y conocida, se define con typedef tipoelem[3] estructura[n]
- Si es fija pero se conoce durante ejecución, se crea con un malloc de tamaño (n*sizeof(tipoelem))
- Si la cantidad de casos varía en tiempo de ejecución:
    - Alternativa 1: contigua. utiliza un vector estático, con un maximo que se conoce durante compilación
    - Alternativa 2: contigua. utiliza un vector dinamico, con un max de n elementos. Se usa si se conoce el max durante ejecución
    - Alternativa 3: enlazada. no requiere max por lo que es más eficiente en memoria, pero la destrucción se hace uno por uno

[3] 'tipoelem' es un nombre dado mediante typedef al tipo de elemento que almacene el vector. en este caso, puede ser por ejemplo int.

2 - Pilas

Descripción

- Tipo de secuencia enlazada. Se inserta y se suprime por la cima
- El último elemento en ser añadido es el primero en ser eliminado (LIFO)
- Ejemplos:
    - Llamadas a subprogramas (almacenan dirección del programa de origen y estado de las variables del programa al realizar la llamada)
    - Recursividad, operaciones aritméticas, quicksort

Operaciones

- PILAVACIA: crea pila vacía
- ESVACÍA: devuelve 'cierto' si está vacía
- CIMA: devuelve el elemento en el tope de la pila
- PUSH: añade un elemento al tope
- POP: suprime un elemento en el tope

Especificación **formal** del TAD Pila

- Tipo: `pila(tipoelem)`
- Sintaxis:
  `*PilaVacia → pila`
  `esVacia(pila)→booleano`
  `cima(pila)→tipoelem`
  `*push(pila,tipoelem)→pila`
  `pop(pila)→pila`
- Semántica: ∀P∈pila, ∀E∈tipoelem:
  `esVacia(PilaVacia)⇒ cierto`
  `esVacia(push(P,E)) ⇒ falso`
  `cima(PilaVacia) ⇒ error`
  `cima(push(P,E)) ⇒ E`
  `pop(PilaVacia) ⇒ error`
  `pop(push(P,E)) ⇒ P`

Implementación contigua (vectores)

- La pila se almacena como un **registro** de dos campos:
    - Vector de elementos
    - Entero 'cima' que indica el índice del elemento más alto
- Push incrementa el tope y añade el nuevo elemento, pop reduce el tope

Implementación con punteros

- Cada elemento de la pila será un registro (nodo) con dos elementos:
    - Valor del elemento de la pila
    - Puntero al siguiente nodo
- Identificaremos ‘pila’ con un puntero: el que apunta al elemento tope
    - 

- Push(p,e) crea una nueva pila que tiene el elemento e como cima, y esta cima apunta a p como siguiente elemento
- Pop se realiza con *p = (*p)->sig

Comparación de implementaciones[4]

- La implementación por arrays es mejor si:
    - se conoce la longitud estimada de la pila (permite max)
    - Los elementos de la pila son pequeños
    - Es común crear y destruir pilas, o recorrer sus elementos
- La implementación por punteros:
    - No requiere especificar max
    - Es más eficiente para insertar o eliminar elementos de una pila
    - Es mejor si los elementos de la pila son grandes o si no se conoce la longitud estimada de la pila

[4] esto aplícase todo tamén as implementaciones de colas e listas

3 - Colas

Descripción

- Tipo de secuencia enlazada.
- Conjunto ordenado de elementos homogéneos en los cuales los elementos se eliminan por un extremo, el **principio** o cabeza, y se añaden por el **final**.
- El primer elemento en entrar será el primero en salir (**FIFO**)

Especificación formal

- **Valores:** Colección de elementos homogéneos que opera según un modelo FIFO.
- **Sintaxis:**
  - * ColaVacia → TCOLA
  - * AñadirCola (TCOLA, TELEMENTO) → TCOLA
  - PrimeroCola (TCOLA) → TELEMENTO
  - EliminarCola (TCOLA) → TCOLA
  - EsColaVacia (TCOLA) → BOOLEAN
- **Semántica:** ∀c ∈ TCOLA, ∀e ∈ TELEMENTO
  - EsColaVacia (ColaVacia) => TRUE
  - EsColaVacia (AñadirCola(c,e)) => FALSE
  - PrimeroCola (ColaVacia) => ERROR
  - PrimeroCola (AñadirCola(c,e)) => si EsColaVacia(c) entonces e si no PrimeroCola(c)
  - EliminarCola (ColaVacia) => ERROR
  - EliminarCola[5] (AñadirCola(c,e)) => si EsColaVacia(c) entonces ColaVacia si no AñadirCola((EliminarCola(c),e)

[5] non elimina a cola, elimina o primeiro elemento da cola

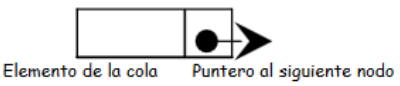
Implementación mediante memoria estática

- Podemos representar una cola mediante una estructura de dos campos:
    - Un **array** donde se almacenen los elementos de la cola, con un máximo predefinido
    - Dos **índices** que indiquen el principio y final de la cola
        - EsColaLlena si el índice de final es el máximo del vector
        - EsColaVacía si el índice de final y inicio son el mismo
        - AñadirCola aumenta el final en 1, EliminarCola aumenta el principio en 1
- A medida que se eliminan elementos, sus posiciones al principio del vector van quedando vacías e inutilizables, por lo que se alcanza el máximo cuando queda espacio libre.
    - Para solucionarlo, se puede utilizar una estructura circular

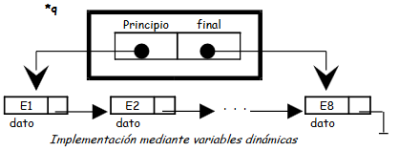
Implementación mediante vector circular

- Definimos que la posición siguiente al último elemento es el primer elemento.
    - El caso especial se realiza con una función **siguiente**, que devuelve [i+1] si la posición no es la final y [0] si si lo es.
- Ahora, la condición de EsColaVacía se activa también con una cola llena. Para evitar esto, se podría dejar una posición libre en la cola que nunca será ocupada.
    - La posición liberada es la indicada por el campo principio de la cola
    - La condición de ColaLlena será cuando el final de la cola se encuentre justo antes de la posición liberada

Implementación mediante punteros



- Definimos una estructura TNODO con dos campos: el elemento del nodo y un puntero al nodo siguiente
- Definimos una estructura STCOLA con dos campos que son ambos punteros: un puntero al TNODO principio y otro al TNODO final.



Dicolas

- Colas bidireccionales que permiten eliminar/añadir elementos por ambos extremos.
- Incluyen nuevas operaciones:
    - void ultimo (TCOLA C, TIPOELEM *E) devuelve el último elemento
    - void insertar_frente (TCOLA *C, TIPOELEM E) añade un elemento al inicio, y el principio de la cola retrocede 1 posición
    - void suprimir_final (TCOLA *C) elimina el último elemento, y el final de la cola retrocede 1 posición
- También se pueden incluir restricciones a en que extremos se puede añadir o eliminar elementos

Colas de prioridad

- Colas a cuyos elementos se les asigna una **prioridad**.
    - El elemento con mayor prioridad es procesado primero
    - Dos elementos de igual prioridad se procesan en el orden en que fueron incluídos.
- Se pueden guardar los elementos en orden de prioridad o guardarlos en orden de inclusión y luego tener en cuenta la prioridad al sacarlos.
    - Es más sencillo insertarlos en orden de prioridad
- **Implementación mediante secuencia enlazada:** Lista enlazada donde cada nodo tiene su dato, un número con su prioridad y un puntero al siguiente
    - Se pueden insertar datos en cualquier posición
    - Una lista es una estructura con punteros al principio y al final
- **Implementación mediante secuencia de colas:** Una cola por cada nivel de prioridad
    - Se utiliza un array de registros (el nº de niveles de prioridad es conocido)
    - Para añadir un elemento con prioridad m, se busca la cola que se corresponde con la prioridad m. Si no existe se crea, si existe se añade el elemento en el final
    - Para suprimir un elemento de maxima prioridad, se busca la cola de máxima prioridad no vacía y se suprime su elemento frente.

4 - Listas

Descripción

- Tipo de secuencia enlazada.
- Conjunto de elementos homogéneos ordenados de acuerdo a las posiciones de estos (relación predecesor-sucesor).
    - El primer elemento será el a_1 (no a_0).
- Su longitud puede variar y se pueden insertar elementos en cualquier posición
- Operaciones: Construcción, posicionamiento, consulta, modificación
- 'principio' será la posición del primer elemento, 'final' es una posición vacía tras el último elemento

Especificación informal

- **lista =** TAD con operaciones crea, fin, primero, siguiente, anterior, esVacia, recupera, longitud, inserta, suprime, modifica
- **Descripción:**
    - Sus valores son del tipo tipoelem. Las posiciones son del tipo posicion
    - Las listas son mutables: existen funciones inserta, suprime y modifica para añadir, eliminar y modificar elementos.
- **Operaciones:**
    - crea() devuelve (lista) devuelve la lista vacía
    - fin(L:lista) devuelve (posicion) devuelve la posición fin (NO el elemento final)
    - primero(L:lista) devuelve (posicion) devuelve la posición del primer elemento
        - Requerimientos: L no es vacía
        - Tivo que mercar varios modios de orxo.
    - siguiente (L:lista; P:posicion) devuelve (posicion) devuelve la posicion siguiente al elemento P

        - Requerimientos: L no es vacía y P!=fin(L)
    - anterior (L:lista; P:posicion) devuelve (posicion) devuelve la posicion anterior al elemento P
        - Requerimientos: L no es vacía y P!=primero(L)
    - esVacia (L:lista) devuelve (booleano) devuelve CIERTO si L es vacía
    - recupera (L:lista; P:posicion) devuelve el elemento de L que ocupa la posicion P
        - Requerimientos: L no es vacía y P!=fin(L)
    - inserta (L:lista; P:posicion; E:tipoelem) modifica L e inserta E como predecesor del elemento que ocupa la posicion P
        - Si P es la posición fin, E pasa a ser el elemento final
    - suprime(L:lista; P:posicion) elimina de la lista L el elemento en la posición P
        - Requerimientos: L no es vacía y P!=fin(L)
    - modifica (L:lista; P:posicion; E:tipoelem) reemplaza el elemento en la posición P por el nuevo elemento E.
        - Requerimientos: L no es vacía y P!=fin(L)

Representaciones contiguas (arrays)

- Los elementos se almacenan en celdas contiguas de un array
- Las posiciones se representan con los índices del array
- En C, se definirá posición como un **puntero a tipoelem**, que apuntará a su respectiva posición en la lista.
    - l→elementos es un puntero al primer elemento de la lista. Para acceder al elemento **i**, se utiliza l→elementos+**i**.
- **Inconvenientes:** requiere definir un 'max' y insertar/eliminar un elemento requiere reubicar los demás [O(n)]
    - Al insertar un elemento en una posición p, se van desplazando 1 hacia la derecha todos los elementos desde el final hasta p

Representaciones enlazadas (punteros)

- Para cada elemento se crea una variable dinámica celda con dos partes: el **elemento** y un **puntero** a la siguiente celda
- Se puede realizar con **simple** o **doble** enlace

Representación con simple enlace

- Una celda está formada por un tipoelem y un puntero a la siguiente celda
- Una lista está formada por dos posiciones (punteros a celda): la de inicio y la del final, y un int de longitud. Posibilidades:

  1 - Puntero posición $\mathbf{a_i}$ apunta a la celda que contiene a_i, y la posición **fin** apunta a la posición posterior al último elemento

  - Problemas: Insertar en a_i requiere acceder a a_{i-1} para insertar el elemento entre a_{i-1} y a_i lo cual es imposible directamente[6], requiere recorrer lista

  2 - Puntero posición $\mathbf{a_i}$ apunta a la celda antes de a_i. La posición **fin** apunta a la posición que contiene el último elemento[7].

  - Permite acceder directamente a a_{i-1}.
  - La posición de a_1 es un puntero a la celda de encabezamiento.

- **Operaciones** y complejidad algorítmica:
  - Creación de lista vacía es O(1). Se inicializan inicio y fin a la posición inicial, y su campo siguiente a NULL, con longitud=0.
  - Destrucción de lista libera todos los elementos uno por uno desde el principio, luego libera la lista en si. O(n)
  - Operaciones de consulta son O(1) [esVacia, recupera, longitud]
    - Acceder a la posición anterior de un elemento es O(n)
  - Para insertar un elemento en una posición p [O(1)]:
    - La celda actualmente siguiente a p se guarda como q

[6] Cada celda solo tiene como información su contenido y un puntero a la siguiente, pero no a la previa. Es imposible pasar directamente de una celda a su anterior.

[7] En una lista vacía, será un puntero a la lista de encabezamiento, igual que a_1

        - Se reserva espacio para una nueva celda, que será la siguiente a p. Su contenido será el que se quiere insertar y su siguente será q.
            - Si q es nula, se modifica el final de la lista.
        - La longitud de la lista aumenta en 1
    - Para suprimir un elemento en la posición p [O(1)]:
        - Se declara q como el siguiente a p, que será el elemento suprimido
        - El siguiente a p pasa a ser el siguiente a q. Si es nulo, p pasa a ser el final de la lista.
        - Se libera q. La longitud de la lista disminuye en 1

Representación con doble enlace

- Una **celda** incluye su contenido, un puntero a la celda siguiente y otro puntero a la celda anterior.
- Una **lista** incluye un puntero al fin de la lista y la longitud
- El puntero posición a_i apunta a la celda a_i
    - Para representar la posición fin de lista, se puede dejar una celda final vacía que se crea al declarar la lista vacía
- Ventajas: punteros posición más intuitivos, no es necesario actualizar fin, permite acceder a ambos extremos en O(1), mayor eficiencia al recorrer lista en ambos sentidos
- Inconvenientes: Dos elementos de información adicional por elemento en la lista
- **Operaciones** y complejidad algorítmica:
    - Creación [O(1)]: reserva espacio para lista y celda final, cuyos campos ant y sig apuntan a si misma
    - Destrucción [O(n)]: libera cada celda y después la lista
    - Posicionamiento [O(1)]: primero, fin, siguiente(p), anterior(p)
    - Consulta [O(1)]: esVacia, recupera, longitud

- Inserta [O(1)]: Reserva espacio, guarda elemento y modifica enlaces previos y posteriores
- Suprime [O(1)]: Modifica enlaces, libera memoria de elemento suprimido

Comparación de implementaciones

- **Resumen:** Contigua es O(n) para insertar/suprimir, simple enlace es O(n) para destruir/anterior, doble enlace solo para destruir.
- Si no se conoce la longitud estimada o los elementos son grandes es mejor enlazada

Tipos de listas

- **Pila:** lista donde solo se puede insertar/suprimir por el primer elemento. Es óptimo utilizar listas de enlace simple.
- **Colas:** Lista donde solo se puede insertar por el final y suprimir por el principio. Es óptimo utilizar listas de enlace simple.
- **Listas ordenadas:** Al insertar un elemento no se escoge la posición, sino que viene calculada por una función POSICIÓN posinser (TLISTA l, ELEMENTO e)
  - No se permiten duplicados → función para eliminar duplicados
- **Lista circular:** El siguiente del último elemento es el primero
  - No confundir con implementación interna (ej doble enlace)

5 - Complejidad computacional

Eficiencia

- Capacidad de resolver el problema propuesto empleando un bajo consumo de recursos computacionales. Los principales recursos son:
- **Coste espacial:** Cantidad de memoria requerida
- **Coste temporal:** Tiempo necesitado para resolver el programa

Talla

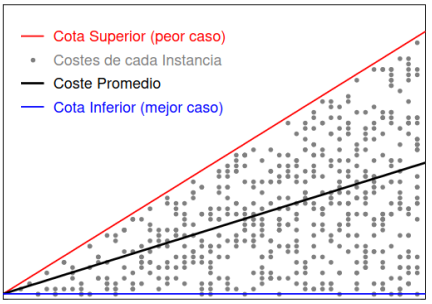
- Dado un programa que opera con un número **n**, el valor de n se denomina **talla**.
- Ejemplo: si un programa opera con un vector, la talla será el número de datos en el vector
- A la hora de diseñar un algoritmo, debemos considerar que el programa sea eficiente para tallas elevadas.

Paso

- Un **paso** es la ejecución de un segmento de código cuyo tiempo de proceso no depende de la talla, o está limitado por alguna constante.
- Constituyen 1 paso:
    - Asignación, operaciones aritméticas o lógicas, comparación, acceso a un elemento de un vector
    - Cualquier secuencia finita cuya longitud no dependa de la talla.
- El **coste computacional de un programa T(n)** es el número de pasos en función de la talla (n).
- Los pasos que no están relacionados con la talla se pueden ignorar. Por ejemplo, si un bucle se ejecuta un nº constante de veces, no tendrá repercusión en T(n)
    - El nº total de pasos estará en función de la talla.

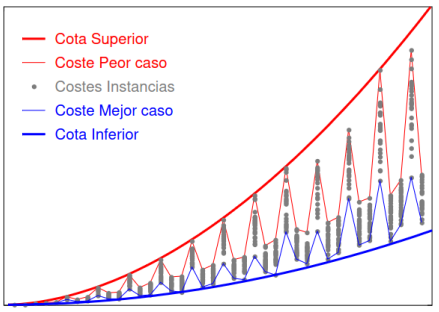
Casos mejor, peor y promedio

- En ocasión, el coste computacional de un programa no depende sólo de la talla.
- Por ejemplo, en un algoritmo de búsqueda lineal, el tiempo de ejecución no dependerá sólo del nº de elementos en la lista, sino de la posición del elemento que buscamos (si está al principio se encontrará antes).
    - Este es un factor difícil de medir cuando se considera el algoritmo.
- Por esto, para un algoritmo se pueden medir por separado la **cota inferior (Ω)** y la **cota superior (O)**: el nº de pasos requeridos en el mejor y peor caso, respectivamente.
- Un algoritmo de búsqueda lineal será **Ω(1)** (el elemento puede ser el primero) y **O(n)**



Notación asintótica

- En ocasiones es imposible expresar la complejidad computacional sólo como una función de la talla, pues para ciertas instancias el coste es distinto (ver imagen, donde el coste es mayor para n múltiplo de 3)
- En estas ocasiones conviene funciones que **acoten** superior e inferiormente los costes de todas las instancias para tallas grandes.



- Siendo t(n) el coste en función de la talla, estas funciones de cota serán O(t(n)) y Ω(t(n))
- Sean f y t funciones, se dice que **f(x) es O(t(x))** si existen dos constantes (testigos) c y k tales que:

$$|f(x)| \leq c^*|t(x)| \quad \text{en} \quad x > k$$

- Sean f y t funciones, se dice que **f(x) es Ω(t(x))** si existen dos constantes (testigos) c y k tales que:

$$|f(x)| \geq c^*|g(x)| \quad \text{en} \quad x > k$$

- Sean f y t funciones, se dice que **f(x) es Θ(t(x))** si f(x) es **O(t(x)) y Ω(t(x))**
- $O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!) \subset O(n^n)$
- superlineales é solo n log n

6 - Estrategias o técnicas algorítmicas

Fuerza bruta

- Resuelve un problema siguiendo la estrategia más obvia de solución
- Sencillos de implementar, pero coste bastante alto. Se pueden usar para tallas pequeñas.
- **Ejemplos:**
    - Búsqueda lineal: Recorre la lista completa comparando con el elemento que se busca. **O(n)**
    - Ordenación por inserción: Recorre la lista completa, busca el mínimo y lo coloca al principio. Luego, repite el proceso empezando desde el segundo elemento. **$O(n^2)$.**

Divide y vencerás

- Consiste en descomponer el problema en varios subproblemas del mismo tipo pero más sencillos, resolverlos y combinar sus soluciones para obtener la solución del problema inicial.
- Es **necesario** que el problema admita una formulación recursiva, y el problema original debe poder resolverse mediante combinaciones de subproblemas, de tamaño estrictamente menor.
- Es **deseable** que el tamaño de los subproblemas sea similar y decreciente con progresión geométrica (n/c con c constante y lo más grande posible), y el coste de descomponer/combinar debe ser poco.
    - También es importante elegir bien el **umbral n_0** a partir del cual la función deja de ser recursiva, y se debe evitar resolver el mismo subproblema más de una vez.
    - Por ejemplo, Fibonacci crece de forma aditiva y no geométrica, por lo que es posible resolverlo recursivamente pero no ideal.
- El **coste total** de un algoritmo divide y vencerás, que origina **k** suproblemas de tamaño **n/c** es $t(n) = kt(n/c) + t_d(n) + t_c(n)$.
    - Si el tiempo de descomposición t_d,y combinación t_c no son excesivos, se considera que tienen un coste polinómico: $t(n) = kt(n/c) + O(n^i)$.

- Entonces, la utilización de divide y vencerás no garantiza nada sobre la eficiencia del algoritmo obtenido. En comparación a un algoritmo iterativo, puede ser peor o mejor.
  - Si $k<c^i$, $t(n) \in O(n^i)$
  - Si $k=c^i$, $t(n) \in O(n^i \log n)$
  - Si $k>c^i$, $t(n) \in O(n^{\log_c k})$

Ejemplos de divide y vencerás

- **Búsqueda binaria:** Permite ordenar una lista (palabras o números). Se procede comparando el elemento que se quiere localizar con el central en la lista, para dividirla en dos sublistas.
  - Para buscar x en una lista [a0, a1, a2… an], comparamos x con a_m, donde m = $\lfloor (n/2) \rfloor$
  - Si x > a_m, la búsqueda se restringe a la 2ª mitad. Si x≤ a_m, se restringe a la 1ª mitad, donde se incluye a_m.
  - El **coste total** es de t(n) = t(n/2) + O(1). Resolviendo la ecuación recursiva tenemos t(n) = 1 + $\log_2 n$ * O(1) → **$t(n) \in O(\log n)$** (más eficiente que b. lineal)
- **Multiplicación de enteros:** Supongamos que queremos multiplicar dos enteros x e y de n dígitos, expresados en una base i. La solución tradicional es **$O(n^2)$.**
  - Dividimos los enteros en dos partes: $x = a*i^{n/2} + b$, $y = ci^{n/2}+d$
  - La solución es el algoritmo de **Karatsuba y Ofman**, basándose en la propiedad **$xy = ac\ i^n + (ac + bd + (a-b)(d-c))\ i^{n/2} + bd$.**
  - De esta forma, se realizan 3 productos de tamaño n/2. La complejidad es **$O(n^{1.57})$**

Técnica voraz

- Consiste en fijar, en cada paso, el valor de una de las variables mediante una función de selección que siempre toma la decisión **localmente óptima**.
- Un sistema voraz nunca reconsidera una decisión tomada.
- Es **necesario** que exista una **función objetivo** que sea la que deseamos maximizar/minimizar.
    - Debe ser conocido el dominio de la función objetivo.
    - Debe existir también una **función solución** que compruebe si unos determinados valores son una solución del problema.
    - Debe existir una función denominada **factible**, que compruebe si las decisiones tomadas hasta el momento satisfacen las restricciones.
- El **coste** depende del número de iteraciones del bucle y el coste de las funciones seleccion y factible.
    - Suelen ser eficientes, pero no garantizan que se obtengan una solución óptima.
- Los problemas resolubles mediante algoritmo voraz deben cumplir el **principio de optimalidad:** para una secuencia óptima de decisiones, toda subsecuencia ha de ser también óptima.

Ejemplo: Cambio de monedas (Voraz)

- Forma de obtener una cantidad de x centimos con monedas $\{e_1, \ldots, e_k\}$
    - **Función objetivo:** nº de monedas, que hay que minimizar
    - **Dominio:** Conjunto C de todas las monedas
    - **Solución:** S = $\{e_1, \ldots, e_k\}$ es solución si su suma es x
    - **Factible**: S = $\{e_1, \ldots, e_k\}$ es solución si su suma es <=x
    - **Función selección:** Elige la moneda de mayor valor que, al sumarla a la solución en curso, no supera x.
- No devuelve siempre una solución óptima, dependiendo del conjunto de monedas que tengamos.
    - Para que el algoritmo alcance la solución óptima, C debe estar formado por tipos de moneda que sean potencia de un tipo básico t. Ej: C={125,25,5,1}
- **Coste:** **O(max(nlog n, m))** donde n es el nº de tipos distintos de monedas y m es el nº de iteraciones que realiza el bucle

Programación dinámica

```
int * fibonacci(int f[n]){
  f[0]=0;
  f[1]=1;
  for (int i=2;i<n;i++)
    f[i]=f[i-1]+f[i-2];
  return f;
}
```

- Similar a divide y vencerás, pero los subproblemas idénticos se resuelven solo una vez, por lo que se guarda su solución para que, si vuelve a surgir, se devuelve de inmediato.
- Sacrifica espacio para ganar tiempo de ejecución.
- Es una técnica **ascendiente:** requiere resolver primero los suproblemas más pequeños.
- La estructura donde se van guardando los resultados es una tabla

Ejemplo: problema de la mochila

- Se dispone de n objetos o_1, …, o_n, cada uno con un peso p_i y un valor v_i asociados.
- Se deben guardar en una mochila que soporta un peso p_{max}, maximizando el valor. Existen dos variantes del problema:
- **Mochila fraccionada:** Se pueden guardar partes de objetos. Para cada objeto, x_i es la fracción que guardamos en la mochila
  - **Objetivo:** maximizar $\Sigma_i\, x_i\, v_i$
  - **Solución:** S es solución si $\Sigma_i\, x_i\, p_i = p_{max}$
  - **Factible:** S es factible si $\Sigma_i\, x_i\, p_i <= p_{max}$
  - **Selección:** Seleccionar los objetos por orden decreciente de relación valor/peso.
  - **Coste:** La parte más costosa del algoritmo es la ordenación de los elementos según su relacion valor/peso. El coste será igual al del algoritmo de ordenación usado, en el mejor caso, **O(n log n)**
- **Mochila entera:** Los objetos no se pueden dividir. El algoritmo voraz no devuelve una solución óptima, pero la programación dinámica si.
  - Llamamos f(n, p_{max}) al valor de la solución óptima. Se tiene:

$$f(n,p_{max})=\begin{cases}0 & \text{si } n=1 \text{ y } p_{max}<p_1\\ v_1 & \text{si } n=1 \text{ y } p_{max}\geq p_1\\ f(n-1,p_{max}) & \text{si } n>1 \text{ y } p_{max}<p_n\\ \max(f(n-1,p_{max}),f(n-1,p_{max}-p_n)+v_n) & \text{si } n>1 \text{ y } p_{max}\geq p_n\end{cases}$$

(si queda un objeto y no cabe)
(si queda un objeto y cabe)
(queda más de un objeto, y el siguiente no cabe)
(queda más de un objeto y el siguiente cabe: la función se divide en dos posibilidades, no incluir el objeto o incluirlo. de estas opciones se selecciona la que dea un valor mayor)

  - De esta forma, la función crea un árbol de posibilidades y algunos de los suproblemas se repetirán, de forma similar a fibonacci.
  - **Coste:** O(n*p_{max}). Si p_{max} es muy grande y mayor que n, sería un coste cuadrático o superior.

Problema del viajante

- Un viajante tiene que recorrer n ciudades y volver al inicio sin pasar dos veces por la misma ciudad. Encontrar el camino mínimo.
- Se representa cada ciudad como un nodo de un grafo, creando un grafo dirigido. Sea 0 el punto de inicio
- Para dos puntos, sea D(i,j) la longitud del camino entre i y j. Siendo W un subconjunto de los nodos, y C(i,W) la longitud del camino mínimo de i a 0 que pasa por todos los vértices de W. Entonces:

$$C(i,W)=\begin{cases} D(i,0) & \text{si } W=\varnothing \\ \min\limits_{j\in W}(D(i,j)+C(j,W-\{j\})) & \text{si } W\neq\varnothing \end{cases}$$

- La solución al problema será C(0, V-{0}). Se cumple el principio de optimalidad.
- Se puede resolver mediante **programación dinámica:** guardamos los valores de C(i,W) en una tabla para evitar repetir su cálculo.
    - **Coste:** $O(n^2 2^n)$, mejor que n! en fuerza bruta

```
float dist_min(int n,int i,const conjunto &w,
               float **d,float **c){
  conjunto cw(w); cw.eliminar(i);
  if (cw.vacio()) return d[i][0];
  if (c[i][cw.id()]>0) return c[i][cw.id()];
  float dist,dmin=DMAX;
  for(int j=0;j<n;j++)
    if (cw.pertenece(j)){
      dist=d[i][j]+dist_min(n,j,cw,d,c);
      if (dist<dmin) dmin=dist;
    }
  c[i][cw.id()]=dmin;
  return dmin;
}
```

De\A	0	1	2	3
0	∞	3	∞	3
1	5	∞	4	∞
2	2	6	∞	6
3	2	5	4	∞

i \ W	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	14	-1
1	-1	-1	-1	-1	6	-1	-1	-1	∞	-1	-1	-1	12	-1	-1	-1
2	-1	-1	11	-1	-1	-1	-1	-1	8	-1	16	-1	-1	-1	-1	-1
3	-1	-1	10	-1	6	-1	11	-1	-1	-1	-1	-1	-1	-1	-1	-1

Vuelta atrás

- Método de algoritmo voraz pero que permite reconsiderar decisiones tomadas si la solución alcanzada no es óptima.
- Se dice que una solución en curso es **completable** si a partir de ella se puede alcanzar la solución óptima a partir de ella. SI se llega a una solución no completable, se da marcha atrás y se reconsideran las decisiones tomadas. Esto
    - Si se alcanza una solución no completable, no se construye el subárbol correspondiente. Esto se conoce como **poda** del espacio e búsqueda.
    - Se puede realizar también poda basada en la mejor solución: se poda la solución en curso si a partir de ella no es posible obtener una mejor que la mejor solución en curso previa

- Tipos de algoritmo:
    - Vuelta atrás para una solución: Recorre todo el espacio de búsqueda hasta encontrar la primera solución
    - Vuelta atrás para todas las soluciones: Recorre el espacio de búsqueda guardando todas las soluciones
    - Vuelta atrás para la mejor solución: Recorre el espacio de búsqueda comparando cada solución con la previa mejor, y quedándose con la mejor.
- Coste:
    - Depende del nº de nodos del espacio d búsqueda que se visitan, v(n), y del coste de las funciones solución y completable, p(n). **O(p(n)v(n))**
    - El tamaño de v(n) dependerá de cuántos nodos se poden. Puede ser entre **O(p(n)n)** y **O(p(n)k^n)**
- Ejemplo: Coloreado de Grafos con Vuelta Atrás

```
void colorea(int n,int k,bool **ady,int color[],int i){
  for (int j=0;j<k;j++){
    color[i]=j;
    if (completable(ady,color,i))
      if (i==n-1) imprimir(n,color);
      else colorea(n,k,ady,color,i+1);
  }
  return;
}
bool completable(bool **adj,int color[],int i){
  int j=0;
  for (int j=0;j<i;j++)
    if (adj[j][i] && color[j]==color[i]) return false;
  return true;}
```

Ramificación y Poda

- La técnica de vuelta atrás normal realiza un recorrido ciego, tras cada nodo se avanza al siguiente.
- La técnica de ramificación y poda avanza de cada nodo al próximo más **prometedor** en base a la información de que se disponga. Se generan primero todos los hijos de un nodo y después se avanza al más prometedor de ellos
    - Diremos que un nodo es prometedor si expandiéndolo se puede conseguir una solución mejor que la actual.
- **Nodo vivo:** la solución que pasa por él es prometedora y aún no se han generado todos sus hijos
- **Nodo muerto:** no van a generarse más hijos por ser malo o ya se han generado
- **Nodo en expansión:** se están generando sus hijos en el instante
- El coste sigue siendo**O(p(n)v(n)).** En este caso, v(n) suele ser d^n con d=max|C_k|
    - C_k es el conjunto de hijos en la última capa del árbol (?)

7 - Algoritmos de ordenación y búsqueda

Tipos de operaciones

- **Operaciones internas:** Se realizan en memoria principal
- **Operaciones externas:** Datos almacenados en un dispositivo externo.

Algoritmos de búsqueda

- **Búsqueda secuencial:** Compara cada elemento del conjunto con el valor deseado hasta encontrarlo o llegar al fin de la lista.
    - Búsqueda con centinela: guarda el elemento deseado al final de la lista, e itera por la lista comparando hasta encontrar el elemento deseado. Si lo encuentra al final, confirma que no está en la lista.
        - Sólo 1 comparación por bucle → más eficiente
    - Complejidad lineal (O(N))
- **Búsqueda binaria:** Requiere una lista ordenada. Se comienza por el central y según sea mayor o menor que el buscado se continúa con la primera o segunda mitad de la lista.
    - Complejidad $O(\log_2 N)$

Bubble sort

- Se recorre la array un cierto número de veces, comparando los pares de valores consecutivos.
    - Si no están ordenados, se intercambian.
- En la primera pasada el mayor elemento estará al final, en la segunda iteración el segundo mayor llegará a la penúltima posición, y así sucesivamente.
- Orden promedio **$O(N^2)$**
- **Burbuja mejorada:** Tras cada iteración, comprueba si se ha realizado algún intercambio. Si se ha pasado por toda la lista sin cambiar nada, se considera ordenada y finaliza el programa.

```
void bubblesort(vectorD *v1){ //n²
    //Recibe un vectorD e ordea os seus elementos empregando
    //o método Bubble Sort.
    unsigned long int n = LongitudVector(*v1);
    unsigned long int i,j;
    TELEMENTO vj,vsiguiente;

    for (i=0; i<n-1; i++){
        for (j=0; j<n-i-1; j++) {
            RecuperaVector(*v1, j, &vj);
            RecuperaVector(*v1, j+1, &vsiguiente);
                if(vj>vsiguiente) swap(v1, j, j+1);

    }

}}
```

Selection sort

- Se divide la lista en dos partes, una ordenada y otra desordenada (inicialmente, la ordenada está vacía)
- En cada iteración, se selecciona el menor elemento de la lista desordenada y se coloca en su posición correspondiente en la lista ordenada.
- Orden promedio **O(N^2)**

```
void selectionsort(vectorD *v1){ //n²
    //Recibe un vectorD e ordea os seus elementos empregando

    unsigned long int n = LongitudVector(*v1);
    unsigned long int i,j, min;
    TELEMENTO vj , vmin;

    for (i=0; i<n; i++){
        min = i;
        for (j=i+1; j<n; j++) {

            RecuperaVector(*v1, j, &vj);
            RecuperaVector(*v1, min, &vmin);

            if (vj<vmin) min=j;
        }
    swap(v1, i, min);
    }
}
```

Insertion sort

- Similar a Selection Sort. En cada iteración, en lugar de tomar el menor elemento de la lista desordenada se toma el primero.
- Cada elemento se coloca en la posición correcta de la lista ordenada (antes o después de los elementos ya ordenados según su valor)
    - Para esto, es necesario hacerle hueco desplazando los elementos necesarios a la derecha.
- Orden promedio **O(N^2)**

Quick sort

- Utiliza la táctica ‘divide y vencerás’.
- Se escoje un elemento cualquiera (normalmente el primero) como **pivote**. Se colocan todos los elementos menores que el pivote a su izquierda, y los mayores a su derecha.
- Luego, se aplica el mismo método a cada una de las dos mitades del array.
- Método implementado:
    - Se realizan dos búsquedas, una de izquierda a derecha, buscando un elemento mayor que el pivote, y una de derecha a izquierda, buscando un elemento menor que el pivote.
    - Cuando se han encontrado ambos se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran.

Ejemplos de implementación TADs[8]

TAD Bolsa

- TAD contenedor
- Definición mutable
- Representación enlazada
- **NOTA:** las funciones que sólo acceden a una bolsa reciben una bolsa b, las funciones que modifican una bolsa reciben un puntero a bolsa *b.
    - Esto se aplica a todos los TAD y se pregunta mucho en los tests de prácticas

bolsa.h

```
#include "errores.h"

typedef void * bolsa;  //un tipo opaco
typedef int tipoelem;


void bolsavacia(bolsa *b);

void poner (bolsa *b, tipoelem e, codigo_error * cod);

int esvacia (bolsa b);

short cuantos (bolsa b, tipoelem e);

void dest (bolsa *b);
```

[8] dudo que nada de esto haba que aprendelo realmente pero xa perdin o tempo facendoo

bolsa.c

```
#include <stdlib.h>
#include "errores.h"

typedef int tipoelem;
typedef struct celda {
        tipoelem elemento;
        struct celda * siguiente;
} tipocelda;  //Crea un tipo de datos 'struct celda' e pasa a chamarlle 'tipocelda'.
                tipocelda ten dous componentes: un elemento (de tipo int) e un
                punteiro a outro struct celda, que será o seguinte.

typedef tipocelda * puntero;
typedef puntero bolsa; //(non sei por que fai o paso intermedio pero) define 'bolsa'
                        como un punteiro a tipocelda. Desta forma, 'bolsa' será un
                        punteiro ao primeiro elemento da bolsa.

void bolsavacia(bolsa *b) {
*b=NULL;                //Non libera memoria, simplemente marca unha bolsa como
}                         vacía. Para liberar está a función dest

void poner (bolsa *b, tipoelem e, codigo_error * cod) { // nótese que colle como
// parámetro un punteiro a bolsa (sería un punteiro a un punteiro a unha struct)
        puntero aux;
        aux=(puntero) malloc(sizeof(tipocelda));  // Reserva memoria para unha nova
                                                    celda, onde irá o elemento.
        if (aux == NULL)
        *cod = meminsu;
        else {
                aux->elemento = e;  // Garda o novo elemento na celda creada
                aux->siguiente = *b; // O elemento 'seguinte' ao novo elemento será o
                                        previo primeiro elemento da bolsa. Desta forma,
                                        este novo elemento continúa enlazado co resto
                                        da bolsa.

                *b = aux;               // A bolsa toma ahora 'aux' como o primeiro
                                          elemento. Collese *b porque b aquí é un
                                          punteiro a bolsa, non unha bolsa.
                *cod = exito;
        }
}
```

```
int esvacia (bolsa b) {
        if (b==NULL) return 1;
        else return 0;
}

short cuantos (bolsa b, tipoelem e) {
        short cont=0;
        puntero aux=b; // Non se opera coa bolsa pasada directamente para non
                          modificala.
        while (aux!=NULL) {
                if (aux->elemento == e) cont++;
                aux = aux->siguiente;
        }
        return cont;
}

void dest (bolsa *b) { // o mismo de antes: colle un punteiro a bolsa, e siempre o
                          utiliza como *b.
        puntero aux=*b, aux2;
        while (aux!=NULL) {
                aux2 = aux;             // aux2 apunta ao mismo que aux, o elemento
                                           actual
                aux = aux->siguiente; // aux avanza ao siguiente elemento
                free(aux2);             // aux2 libera o elemento actual
                *b = aux ;              // *b actualizase en cada iteración. desta forma,
                no final do programa acabará apuntando a NULL, e queda a bolsa
                correctamente liberada.
        }
}
```

TAD Pila (implementación con punteros)

- TAD contenedor
- Implementación enlazada
- Un elemento pila es un puntero al tope, la primera celda.
- Un elemento celda contiene su elemento y un puntero a la celda siguiente

pila.h

```
typedef void * TPILA;  //un tipo opaco
typedef int tipoelem;

void PilaVacia(TPILA *p);

int EsVacia (TPILA b);

void Push(TPILA *b, TELEMENTO e);

void Cima(TPILA *b, TELEMENTO *pe);

void Pop(TPILA *b);

void leer(TPILA *P, short N)
```

pila.c

```
typedef int tipoelem;

typedef struct nodo {
        TELEMENTO dato;
        struct nodo * sig; } TNodo;

typedef TNodo* TPILA;


void PilaVacia (TPILA *b) {
*p=NULL;              //Non libera memoria, simplemente marca unha pila como vacía
}
```

```
int EsVacia(TPILA p) {
        if (p==NULL) return 1;
        else return 0;
}

void Push(TPILA * p , TELEMENTO e) {
        TPILA q;
        q= (TPILA) malloc (sizeof(TNodo)); // se define nueva pila
        q->dato = e ; // se pone en la cima de la pila el nuevo elemento
        q->sig = *p ;  // debajo de la cima está la pila previa
        *p = q;
}

void Cima (TPILA p, TELEMENTO * pe) {
        int respuesta = EsVacia(p);
        if (respuesta == 1)
                printf ("ERROR, la pila no tiene elementos\n");
        else *pe = p->dato;
}

void Pop(TPILA * p) {
        TPILA q;
        int respuesta;
        respuesta = EsVacia(*p);
        if (respuesta==1)
                printf("ERROR, ....\n");
        else {
                q = *p;
                *p = (*p)->sig; // la nueva cima es el siguiente de la cima actual
                free(q); // se libera la previa cima
        }
}

void leer(TPILA *P, short N) { //TPILA es un puntero a estructura pila
        TELEMENTO e;
        short i;
        PilaVacia(P);
        for (i=1 ; i<=N ; i++) {
                printf("Dime un entero:");
                scanf("%d", &e);
                push(P,e);
                }
        }
```

```
void imprimir (TPILA *pila) {
	TPILA aux;
	TELEMENTO E;

	PilaVacia(&aux); // fija aux a null
	while (esVacia(*pila) == 0 ){
		cima(*pila,&E); // guarda la cima actual en E
		pop(pila); // elimina la cima actual de la pila
		push(&aux, E); // añade la cima actual a aux
		printf("%d\n",E); // imprime la cima actual
	} // de esta forma, se vacían todos los elementos de P en aux, cambiadas de
	orden

	while (esVacia(aux) == 0 )
	{
		cima(aux,&E);
		pop(&aux);
		push(pila, E);
	} // se hace lo mismo para volver a guardar los elementos en pila, vuelven a
	quedar en el orden correcto
}
```

TAD Cola (implementación con vector circular)

- TAD contenedor
- Implementación contigua
- Formado por un vector de elementos y dos enteros: uno indica la posición de inicio y otro la posición inicial
- Se sacrifica un elemento del array para evitar confusión entre cola llena y cola vacía.
    - Cola vacía: principio == final
    - Cola llena: Principio == Siguiente(final)
    - Permite vaciar la cola por completo, pero no rellenarla por completo. Si está rellena excepto por la casilla sacrificada, contará como cola llena.

cola.h

```
typedef void * TCOLA;       //un tipo opaco
typedef int tipoelem;       //aqui pon int pero poderia ser outra cousa

void ColaVacia(TCOLA *q);

void AnadirCola(TCOLA *q, tipoelem e);

int EsColaVacia (TCOLA q);

void EliminarCola(TCOLA *q);

void PrimeroCola(TCOLA q, tipoelem *pe);

int Siguiente(int pos);
```

cola.c

```
#define MAX 100 //permite almacenar 99 elementos por lo descrito antes

typedef int TELEMENTO;

typedef struct {
        TELEMENTO arrayelementos[MAX];
        int principio, final;
        } STCOLA;

typedef STCOLA* TCOLA;

int Siguiente(int pos) { //devuelve un entero, la posición siguiente a pos
        if (pos < MAX -1) return (pos+1);
        else return 0;
}


void ColaVacia(TCOLA *q) { //reserva espacio para una nueva cola.
        *q = (TCOLA) malloc (sizeof(STCOLA));
        (*q)->final = MAX -1 ;      //el primer objeto añadido ocupará el indice siguiente
        (*q)->principio =  MAX -1; que es 0
}


int EsColaVacia(TCOLA p) { //vacía si es recien creada o si se ha eliminado el último
elemento. en estas circunstancias, el índice del final será igual que el del principio.
        if (q->final==q->principio) return 1;
        else return 0;
}

int EsColaLlena(TCOLA p) { // llena si la casilla siguiente a la última ocupada es
q->principio, que es la casilla que intencionadamente se deja vacía.
        if (Siguiente(q->final)==q->principio) return 1;
        else return 0;
}
```

```
void AnadirCola(TCOLA * q , TELEMENTO e) { // añade un elemento en la posición
                                                siguiente a la final
        TCOLA q;
        int respuesta;
        respuesta = EsColaLlena(*q);
        if (respuesta==1)
                printf("ERROR, ....\n");
        else {
                (*q)->final = Siguiente((*q)->final) // la cola crece en 1
                (*q)->arrayelementos[(*q)->final] = e; }
}

void PrimeroCola (TCOLA p, TELEMENTO * pe) { // guarda en *pe el primer
elemento de la cola.
        int respuesta = EsVacia(p);
        if (respuesta == 1) printf ("ERROR, la pila no tiene elementos\n");
        else *pe = q -> arrayelementos[Siguiente(q->principio)]; // q->principio es la
        casilla vacía, se debe tomar la siguiente
}

void EliminarCola (TCOLA * p) { // elimina el primer elemento de la cola
        TCOLA q;
        int respuesta;
        respuesta = EsVacia(*p);
        if (respuesta==1) printf("ERROR, ....\n");
        else (*q)->principio = Siguiente(q->principio); //no se borra el elemento, se
        avanza el principio en 1. la cola disminuye 1 elemento

        //al ser una representación con array no se hace free() al eliminar cada
        elemento, permanentemente está reservada la memoria para la lista
        completa aunque no se use. se liberaría al final de usar la lista con una
        función distinta de destruirlista.

}
```

TAD Cola (implementación con punteros)

- TAD contenedor
- Implementación enlazada
- Estructura formada por dos punteros a estructuras, denominadas nodos. Una será el nodo de principio y otra el final.
- Los nodos en si mismos están formados por su elemento y un puntero al siguiente.
- El archivo.h sería idéntico al previo.

```
typedef int TELEMENTO;
typedef struct nodo {
        TELEMENTO dato;
        struct nodo * sig; } TNodo;
typedef struct {TNodo * principio, * final; } STCOLA;
typedef STCOLA * TCOLA;


void ColaVacia ( TCOLA * q ) {  // Reserva memoria y define nueva cola
        *q = (TCOLA) malloc (sizeof (STCOLA) ) ;
        (*q)->final = NULL; (*q)->principio = NULL;
}

int EsColaVacia ( TCOLA q ) {
        if ( (q->final == NULL) && (q->principio == NULL) ) return 1;
        else return 0;
}

void  PrimeroCola  (  TCOLA  q,  TELEMENTO  *  e)  {   // Guarda en *e el primer elemento de la cola
        int respuesta;
        respuesta = EsColaVacia(q);
        if ( respuesta == 1) printf("ERROR, la cola no tiene elementos");
        else *e = (q->principio)->dato;
}
```

```
void EliminarCola (TCOLA * q) {  // Elimina el primer elemento de la cola
        int respuesta;
        TNodo * aux;
        respuesta = EsColaVacia(*q);
        if ( respuesta == 1 ) printf("ERROR, cola vacía");
        else {
                aux = (*q)->principio;                  // se guarda en aux para poder
                                                           liberarlo luego
                (*q)->principio = aux->sig;                     // el principio avanza 1
                posición
                if ((*q)->principio == NULL)            // si la lista queda vacía
                        (*q)->final = NULL;             // se declara como vacía anulando
                                                           tambien el nodo final
                free (aux);
        }
}


void AnadirCola (TCOLA * q , TELEMENTO e) {  // añade un elemento al final de la
                                                lista
        int respuesta;
        TNodo * aux;
        aux = (TNodo *) malloc (sizeof (TNodo) );  // se crea la nueva casilla, que
        aux->dato = e;                             // toma como dato e y como
        aux->sig = NULL;                           // siguiente NULL
        respuesta = EsColaVacia (*q);
        if (respuesta == 1) (*q)->principio = aux;    // si era vacía, la nueva casilla es la
                                                         primera.
        else (*q)->final->sig = aux;                  // si no, el previo elemento final
                                                         toma como siguiente el añadido
        (*q)->final = aux;                            // el elemento añadido es ahora el
                                                         final
}
```

TAD Lista(implementación con vector circular)

- TAD contenedor
- Implementación contigua
- Formado por un vector dinámico de elementos 'POSICION' y un entero que define la longitud actual
    - 'POSICION' es un puntero a tipoelem
- Al crear el vector vacío se reserva con malloc la memoria maxima que puede ocupar (requiere constante MAX)

lista.h

```
typedef void * TLISTA;       //un tipo opaco
typedef void * POSICION;
typedef int tipoelem;        //aqui pon int pero poderia ser outra cousa


void crea(TLISTA *l);
void destruye(TLISTA *l);

POSICION primero(TLISTA *l);
POSICION fin(TLISTA *l);
POSICION siguiente(TLISTA *l, POSICION p);
POSICION anterior(TLISTA *l, POSICION p);

int esVacia(TLISTA *l);
int longitud(TLISTA *l);
void recupera(TLISTA *l, POSICION p; tipoelem *e);

void inserta(TLISTA *l, POSICION p; tipoelem e);
void suprime(TLISTA *l, POSICION p);
void modifica(TLISTA *l, POSICION p; tipoelem e);
```

lista.c

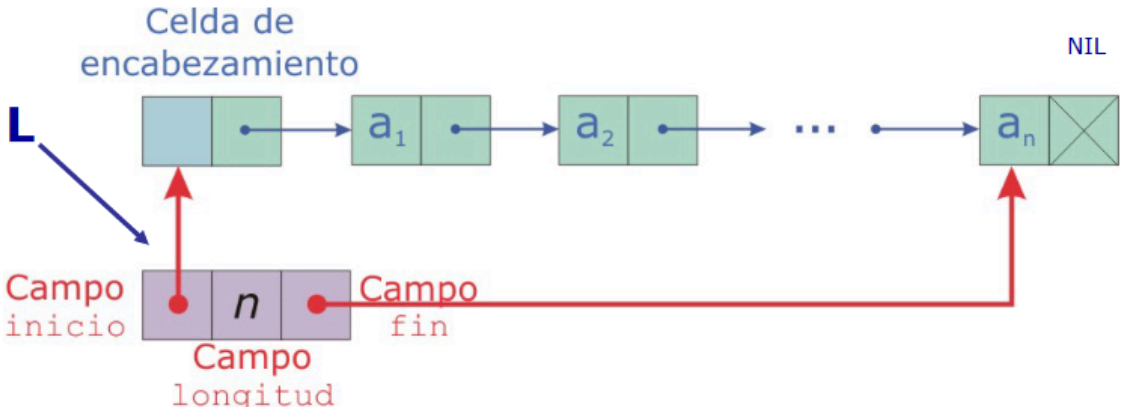
```
#define MAX 100

typedef int TIPOELEM;

typedef TIPOELEM * POSICION;


typedef struct {
        POSICION elementos];
        int longitud;
        } TCELDA;

typedef TCELDA * TLISTA;




void crea(TLISTA *l) { //reserva espacio para una nueva lista
        *l = (TCELDA *) malloc (sizeof(TCELDA));
        (*l)->elementos = (POSICION) malloc (MAX * sizeof(TIPOELEM));
        (*l)->longitud = 0;
}


void destruye(TLISTA *l) {

        free((*l)->elementos);  //*l porque l es un puntero a puntero a celda
        free(*l);

}


POSICION primero(TLISTA l) {

        return l->elementos; //elementos es un vector dinámico, por lo que es un
                                puntero a su primer elemento, que es de tipo
                               POSICION
}

POSICION fin(TLISTA l) {

        POSICION final = l->elementos + l->longitud ;

        return final;
```

```
}
POSICION siguiente(TLISTA l, POSICION p) {
	return p+1;
}

POSICION anterior(TLISTA l, POSICION p) {
	return p-1;
}

int esVacia(TLISTA l) {
	if (l->longitud == 0) return 1;
	else return 0;
}

void recupera(TLISTA l, POSICION p, TIPOELEM *e) {
	*e = *p;
}

int longitud(TLISTA l) {
	return l->longitud;
}
```

```
void inserta(TLISTA *l, POSICION p, TIPOELEM e) {
        POSICION q, r;
        q=fin(*l);        //empieza desde el final [fin es la posición que va después del
                            último elemento], y va retrocediendo hasta
        while (q!=p) { //llegar a la posición donde se quiere insertar e
                r=q;    // r es la posición actual
                q--;     // q retrocede un paso
                *r = *q; // se guarda en r su elemento predecesor, avanzando la lista 1
                posición hacia la derecha
        } // al final del bucle,
        *p=e; //se guarda en p el nuevo valor (el previo valor de p ya se guardó en
                p+1)
        (*l)->longitud++;
}

void suprime(TLISTA *l, POSICION p) {
        POSICION q,r;
        (*l)->longitud -- ;
        q=p; //empieza desde p, donde se guardará el valor de p+1
        while (q!=fin(*l)) {
                r=q;
                q++;
                *r=*q; //en cada casilla se guarda el valor de la casilla posterior
        }
}
void modifica(TLISTA *l, POSICION p, TIPOELEM e) {
        *p=e;
}
```

TAD lista (implementación con punteros con simple enlace)

- TAD contenedor
- Implementación enlazada
- Una celda está formada por un elemento y un puntero a la siguiente celda
- Una POSICION es un puntero a una celda
    - La posición de a_i es un puntero a la celda que precede a a_i
    - La posicion fin es un puntero a la celda que contiene el último elemento
    - La posición inicio es un puntero a la celda de encabezamiento
- TLISTA es una estructura formada por dos posiciones, inicio y fin, y un entero de longitud
- La celda de encabezamiento se mantiene vacía.
- El archivo.h sería idéntico al previo.



lista.c

```
typedef int TIPOELEM;
typedef struct celda{
        TIPOELEM elemento;
        struct celda* sig; } TCELDA;
typedef TCELDA * POSICION;

typedef struct lista {
        POSICION inicio;
        POSICION fin;
        int longitud; } TLISTA;
```

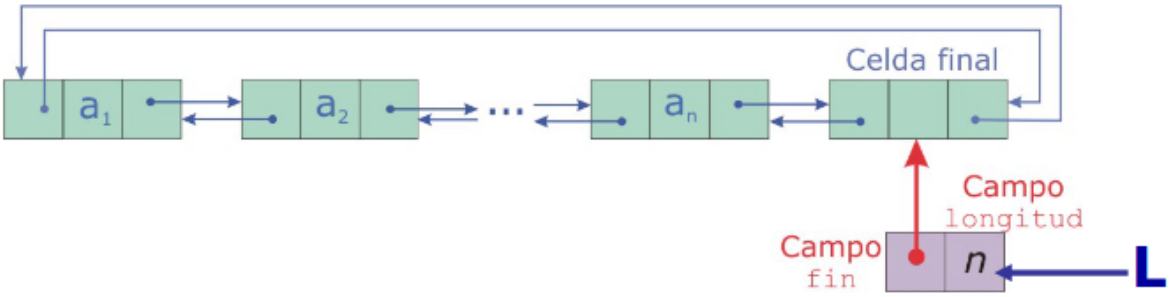
```
void crea( TLISTA *l ) {  //Reserva memoria y define nueva lista
        *l=(TLISTA *) malloc(sizeof(TLISTA));
        (*l)->inicio = (POSICION) malloc(sizeof(TCELDA));//Celda de
        encabezamiento
        (*l)->inicio->sig = NULL; /
        (*l)->fin = (*l)->inicio ; //Inicio y fin apuntan a celda encabezamiento
        (*l)->longitud=0;
}


void destruye(TLISTA *l) { //Elimina lista completa

        (*l)->fin = (*l)->inicio;          //Puntero fin apunta a celda de encabezamiento
        while ((*l)->fin != NULL) {
                (*l)->fin = (*l)->fin->sig;  //Recorre la lista de principio a fin, liberando
                free((*l)->fin);      //cada elemento
                (*l)->inicio = (*l)->fin;
        }
        free(*l);

}


POSICION primero(TLISTA l) {

        return l->inicio;

}


POSICION fin(TLISTA l) {

        return l->fin;

}


POSICION siguiente(TLISTA l, POSICION p) {

        return p->sig;

}
```

```
int esVacia(TLISTA l) {
        if (l->longitud == 0) return 1;
        else return 0;
}
void recupera(TLISTA l, POSICION p, TIPOELEM *e) {
        *e = p->sig->elemento;
}

int longitud(TLISTA l) {
        return l->longitud;
}
void inserta(TLISTA *l, POSICION p, TIPOELEM e) {
        POSICION q ;
        q=p->sig;
        p->sig = (TCELDA *) malloc(sizeof(TCELDA));   //se reserva espacio después
                                                        de p
        p->sig->elemento = e;        //se guarda en esta celda el elemento
        p->sig->sig = q;              //se enlaza con el resto de la lista
        if (q==NULL) (*l)->fin=p->sig;  //si el elemento se añade al final de la lista,
                                          pasa a ser el nuevo final
        (*l)->longitud ++ ;
}
```

```
void suprime(TLISTA *l, POSICION p) {

        POSICION q;

        q=p->sig;               //q será el elemento a eliminar

        p->sig=q->sig;          //se enlaza p con el siguiente a q

        if (p->sig==NULL)   //si no existe, p pasa a ser el último de la lista

                (*l)->fin = p;

        free(q);          //dado que p apunta al elemento que precede a a_p, su siguiente,
        q, apunta a a_p, por lo que será el puntero a liberar para suprimir a_p

        (*l)->longitud -- ;

}

void modifica(TLISTA *l, POSICION p, TIPOELEM e) {

        p->sig->elemento=e; //se toma el siguiente porque p es un puntero al
                                    elemento que precede a a_p

}
```

TAD lista (implementación con punteros con doble enlace)

- TAD contenedor
- Implementación enlazada
- Una celda está formada por un elemento, un puntero a la siguiente celd y otro puntero a la anterior
- Una POSICION es un puntero a una celda
    - La posición de a_i es un puntero a la celda que contiene a_i
    - La posicion fin es un puntero a la celda final, que está vacía.
- TLISTA es una estructura formada por la posición fin y la longitud
- El archivo.h sería idéntico al previo.



lista.c

```
typedef int TIPOELEM;
typedef struct celda{
        TIPOELEM elemento;
        struct celda* ant,sig; } TCELDA;
typedef TCELDA * POSICION;

typedef struct lista {
        POSICION fin;
        int longitud; } TLISTA;
```

```
void crea( TLISTA *l ) {  //Reserva memoria y define nueva lista
        *l=(TLISTA *) malloc(sizeof(TLISTA));
        (*l)->fin = (POSICION) malloc(sizeof(TCELDA));//Celda final
        (*l)->fin->sig = *l)->fin; //los campos ant y sig de la celda final apuntan a si
        (*l)->fin->ant = *l)->fin; //misma
        (*l)->longitud=0;
}

void destruye(TLISTA *l) { //Elimina lista completa


        puntero aux=*l->fin, aux2;
        while (aux!=NULL) {
                aux2 = aux;
                aux = aux->sig;
                free(aux2);
                *l->fin = aux ;
                        }
        free(*l);
}


POSICION primero(TLISTA l) {

        return l->fin->sig;

}


POSICION fin(TLISTA l) {

        return l->fin;

}


POSICION siguiente(TLISTA l, POSICION p) {

        return p->sig;

}
POSICION anterior(TLISTA l, POSICION p) {

        return p->ant;

}
```

```
int esVacia(TLISTA l) {
        if (l->longitud == 0) return 1;
        else return 0;
}
void recupera(TLISTA l, POSICION p, TIPOELEM *e) {
        *e = p->elemento;
}

int longitud(TLISTA l) {
        return l->longitud;
}

void inserta(TLISTA *l, POSICION p, TIPOELEM e) {
        POSICION q ;
        q=p->ant;
        p->ant = (TCELDA *) malloc(sizeof(TCELDA));   //se reserva espacio antes
                                                        de p
        p->ant->elemento = e;       //se guarda en esta celda el elemento
        p->ant->ant = q;              //se enlaza con el resto de la lista
        p->ant->sig = p;
        q->sig = p->ant;
        //(creo que) non hai que facer ningun caso especial para comprobar se se
        está añadiendo ao principio ou final da lista. por ejemplo se añades na
        posición inicial, quedará ese elemento no principio da lista, co previo
        elemento inicial como siguiente e coa celda fin como anterior, o cal é correcto
        (*l)->longitud ++ ;
}
```

```
void suprime(TLISTA *l, POSICION p) {
        POSICION q;
        q=p->ant;
        q->sig=p->sig;          //se completan los enlaces sin incluír p
        free(p);
        (*l)->longitud -- ;
}
void modifica(TLISTA *l, POSICION p, TIPOELEM e) {
        p->elemento=e;
}
```