Listas y Colas en C

Índice

- Estructuras autoreferenciadas
- Asignación dinámica de memoria.
- Listas enlazadas.
- Pilas.
- Colas.

Punteros a estructuras

- Son muy frecuentes en C. Se declaran de la misma forma que un puntero a una variable ordinaria. Ej.:

  struct alumno *palumno;

  ALUMNO *palumno; /* usando typedef */
- Para acceder a un campo de la estructura:

  prom_nota = (*palumno).nota

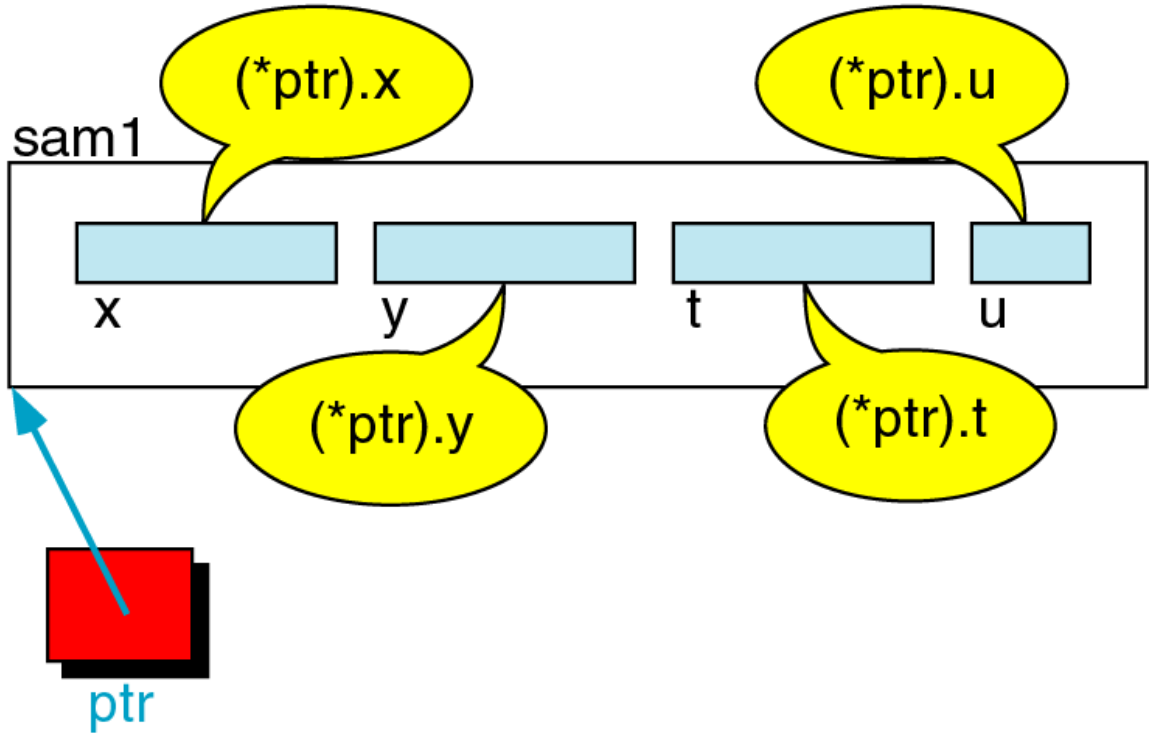
  los paréntesis son estrictamente necesarios.

  Hay un operador (->) alternativo para este propósito:

  palumno -> nota

Punteros a estructuras

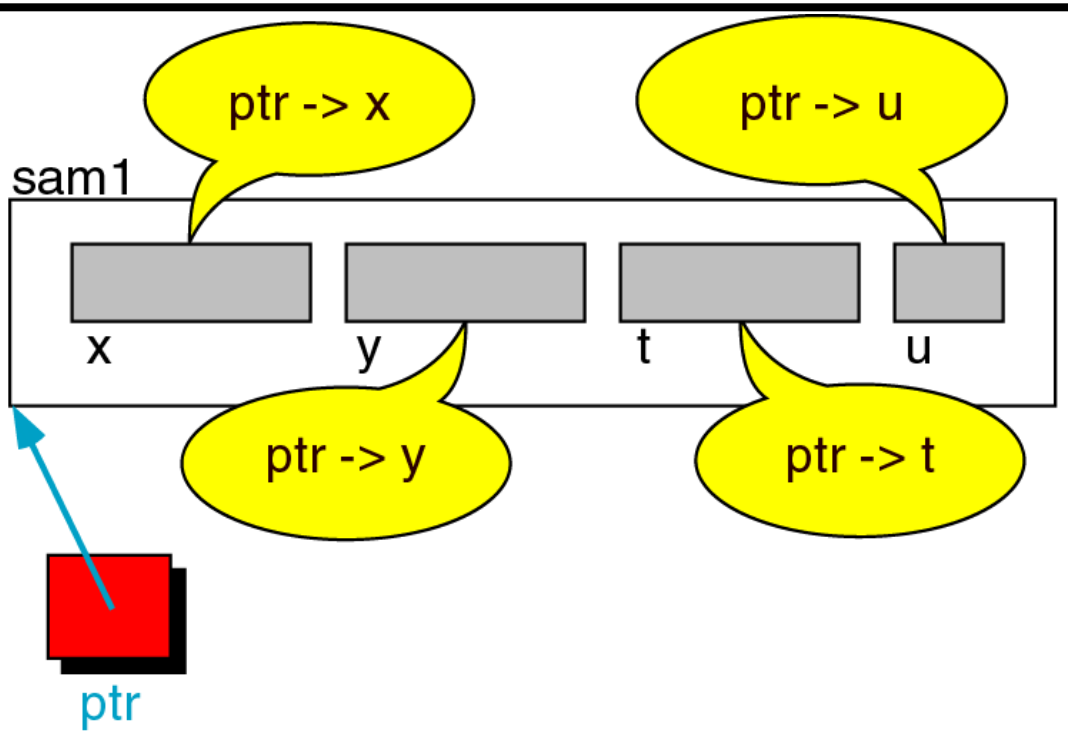
```
typedef struct
  {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
   ...
   SAMPLE   sam1;
   SAMPLE  *ptr;
   ...
   ptr = &sam1;
   ...
```

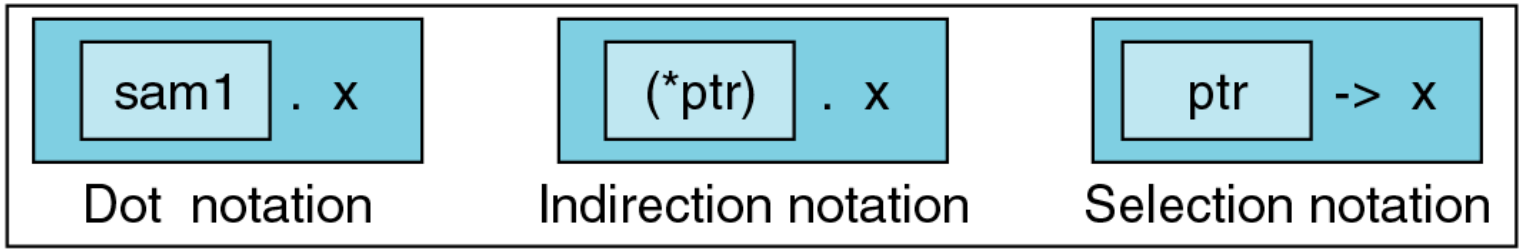


Two ways to reference x

Punteros a estructuras

```
typedef struct
  {
   int   x;
   int   y;
   float t;
   char  u;
  } SAMPLE;
  ...
  SAMPLE   sam1;
  SAMPLE  *ptr;
  ...
  ptr = &sam1;
  ...
```



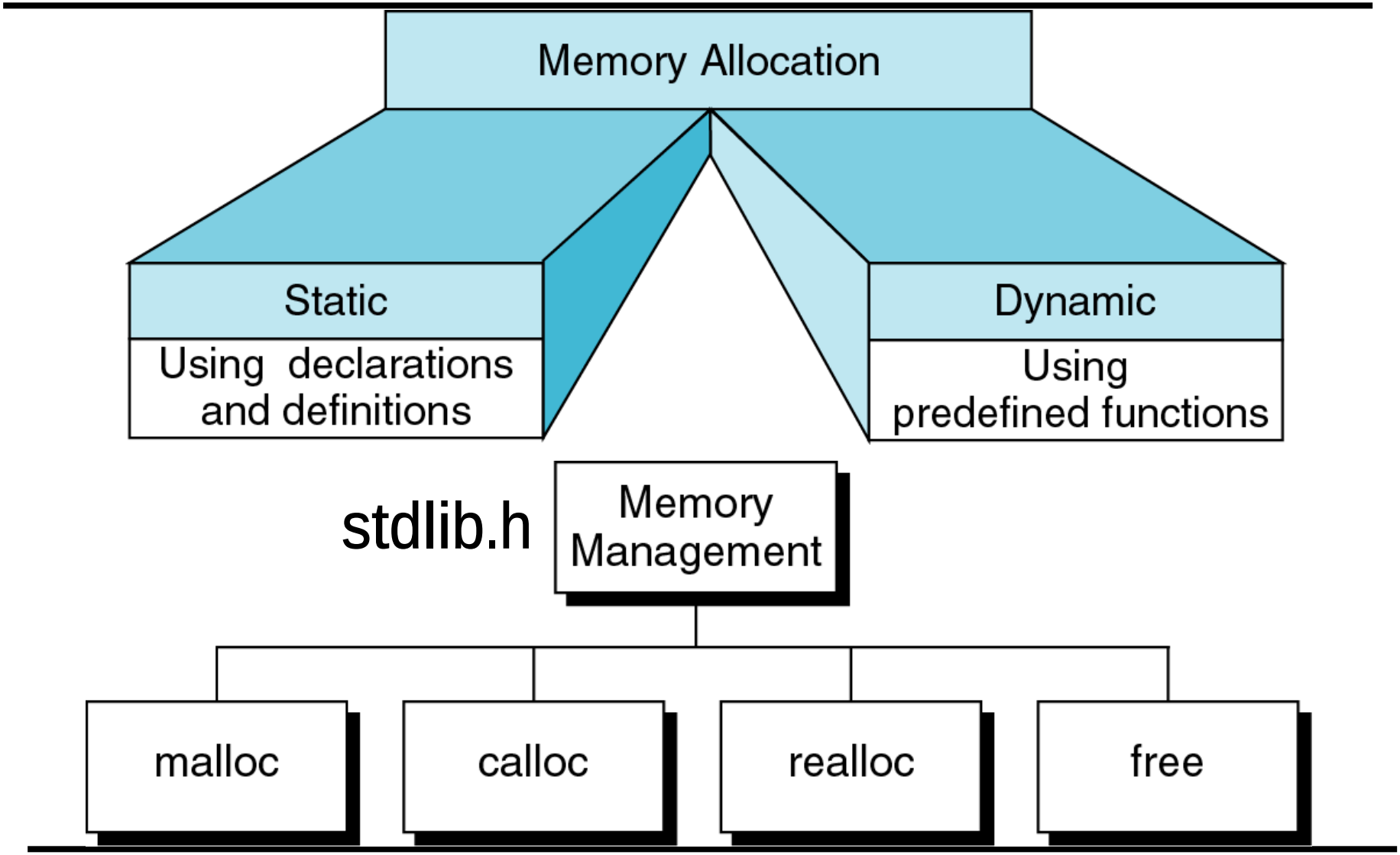


three ways to reference the field **x**

Asignación dinámica de memoria

- Además de la reserva de espacio estática (cuando se declara una variable), es posible reservar memoria de forma dinámica.
- Hay funciones de gestión de memoria dinámica (stdlib.h):
  - void *malloc(size_t): Reserva memoria dinámica.
  - void *calloc(size_t): Reserva memoria dinámica.
  - void *realloc(void *,size_t): Ajusta el espacio de memoria dinámica.
  - free(void *): Libera memoria dinámica.

Memoria dinámica

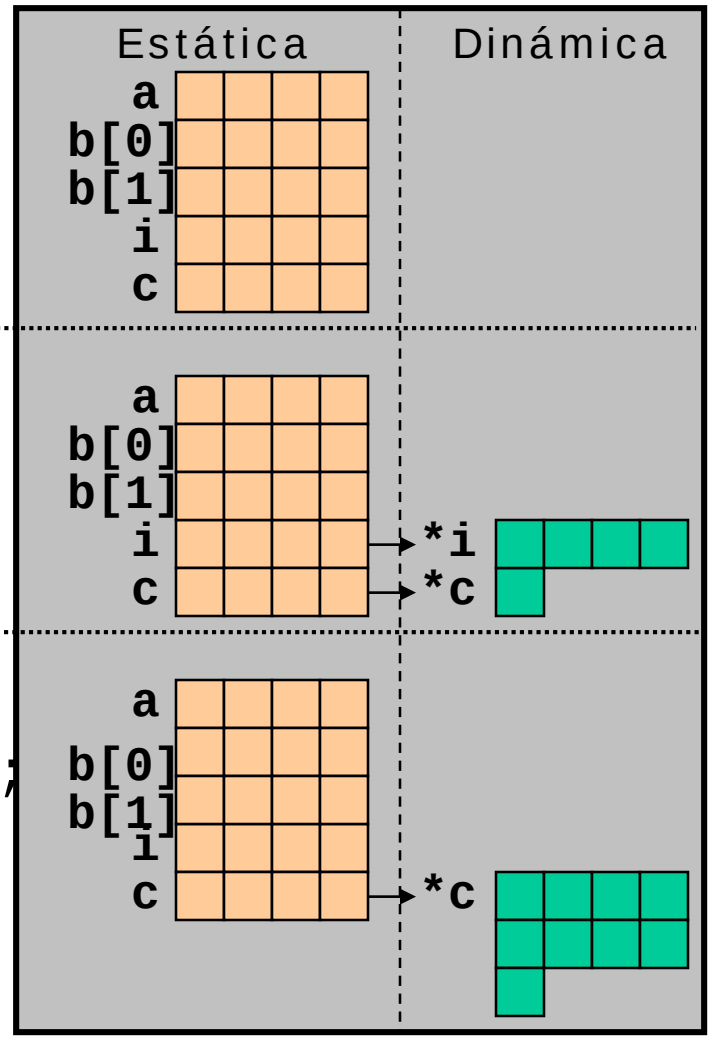


Memoria Dinámica - uso

```
#include <stdlib.h>
int    a,b[2];
int *i;
char *c;
```

```
i=(int *)malloc(sizeof(int));
c=(char *)malloc(sizeof(char));
```

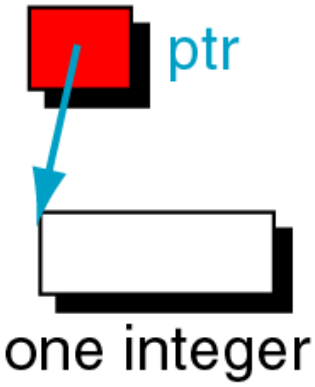
```
free(i);
c=(char *)realloc(c,sizeof(char)*9);
```



Memoria Dinámica – uso malloc

`void *malloc(size_t tamaño_bloque)`

devuelve NULL si no hay suficiente memoria

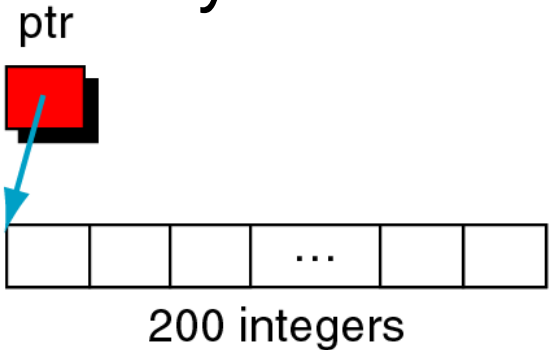



```
ptr = (int *)malloc(sizeof(int));
if ( ptr == NULL)
  /* No hay memoria */
  printf ("ERROR. No hay memoria\n");
else
  …. /* Memoria disponible */
```

Memoria Dinámica – uso calloc

`void *calloc(size_t nelem, size_t elsize)`

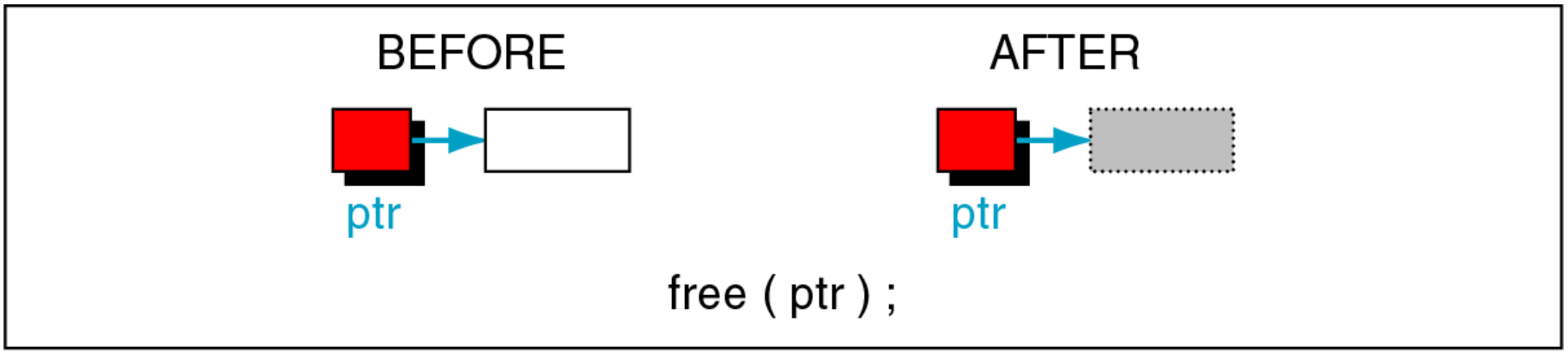
devuelve NULL si no hay suficiente memoria

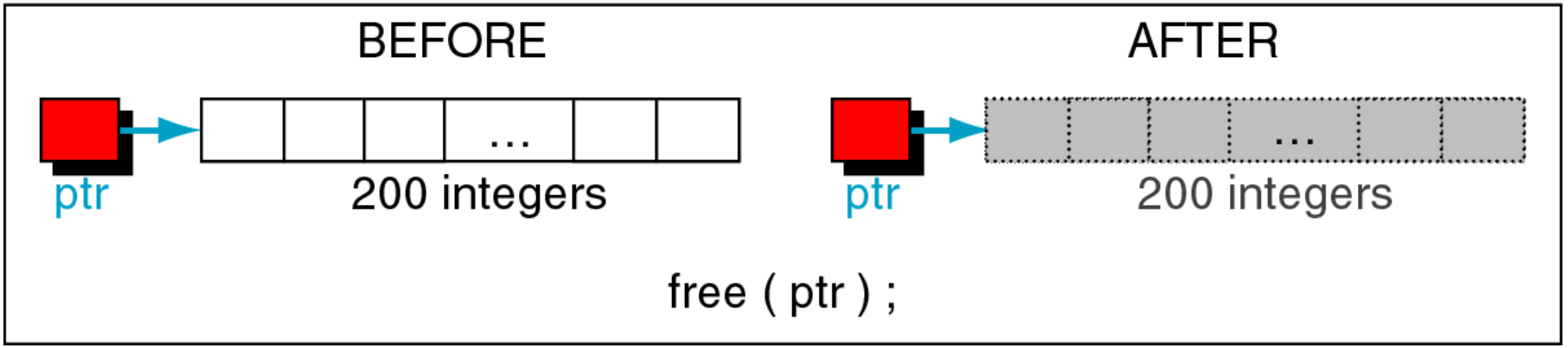



```
ptr = (int *)calloc(200, sizeof(int));
if ( ptr == NULL)
  /* No hay memoria */
  printf ("ERROR. No hay memoria\n");
else
  …. /* Memoria disponible */
```

Memoria Dinámica – uso free

```
void free(void *puntero_al_bloque)
```





Ejemplo: array dinámico

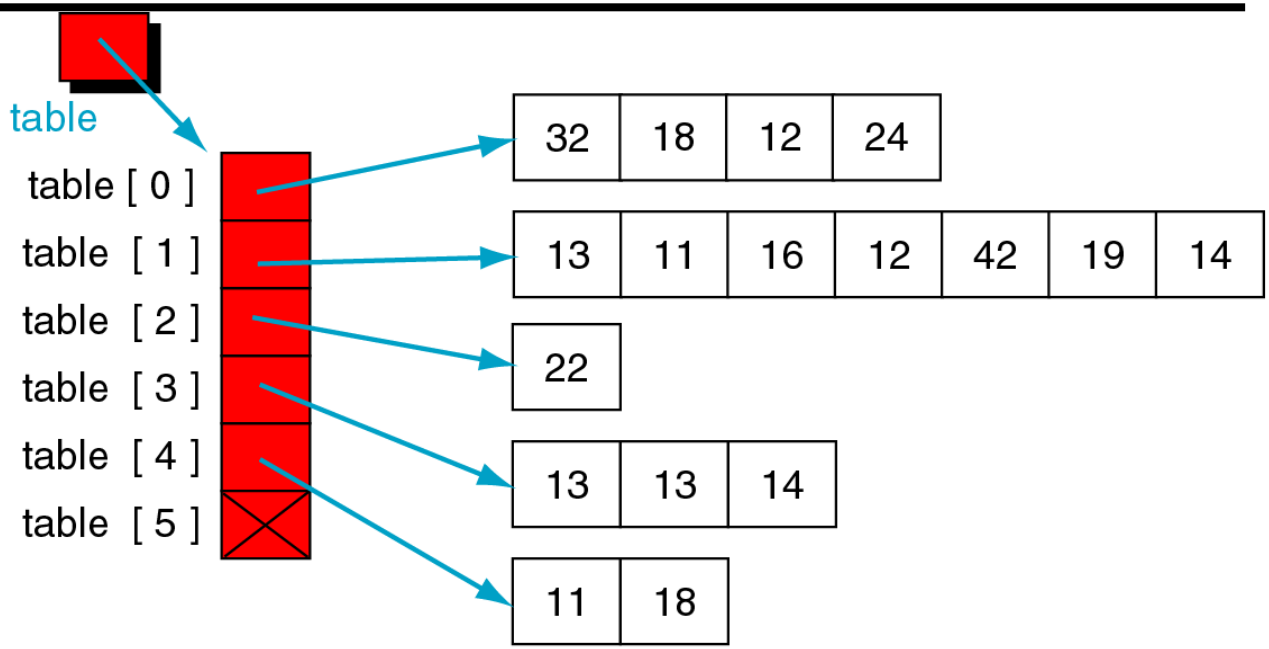
```
/*****************************************************************\
* Programa: array_dinamico.c                                      *
* Descripción: Prog. que crea una array dinamico y genera otro con los   *
*              elementos pares del primer array y lo imprime      *
\*****************************************************************/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int dim_usu;   /* dimension del vector del usuario */
   int dim_par;   /* dimensión del vector de elementos pares */
   int n;         /* indice para los for */
   int m;         /* indice para recorrer arrya de pares */
   int *pvec_usu; /* puntero al vector introducido por el usuario */
   int *pvec_par; /* puntero al vector elementos pares (dinamico) */

   printf("Introduzca la dimension del vector: ");
   scanf(" %d", &dim_usu);

   pvec_usu = (int *) calloc( dim_usu, sizeof(int)); /*Asignar memoria vect. usuario
   */
   /*pvec_usu = (int *) malloc( dim_usu*sizeof(int));*/
   if (pvec_usu == NULL) {  /* si no hay memoria */
         printf("Error: no hay memoria para un vector de %d elementos\n", dim_usu);
   }
```

Ejemplo: array dinámico

```
    else
    {
          for (n = 0; n < dim_usu; n++) /* pedir elementos del vector */
          { printf("Elemento %d = ", n); scanf("%d", &(pvec_usu[n]));}
          dim_par = 0;
          for (n = 0; n < dim_usu; n++)
             if ((pvec_usu[n] % 2) == 0) dim_par++;
          pvec_par = (int *) calloc( dim_par, sizeof(int));
          if (pvec_par == NULL) {  /* si no hay memoria */
             printf("Error: no hay memoria para un vector de %d elementos\n",
    dim_par);
          }
          else
          { /* se copian los elementos pares */
                    m = 0;
                    for (n = 0; n < dim_usu ; n++)
                        if ((pvec_usu[n] % 2) == 0) { pvec_par[m] = pvec_usu[n]; m++;}
                    printf("\n----------------------------\n");
                    for (n = 0; n < dim_par ; n++)
                              printf("Elemento par %d = %d \n", n, pvec_par[n]);
          }
          free (pvec_par);
    }
    free (pvec_usu);
}
```

Matriz con número de columnas diferentes

- Se representa como un array de arrays (puntero a puntero) dinámico.

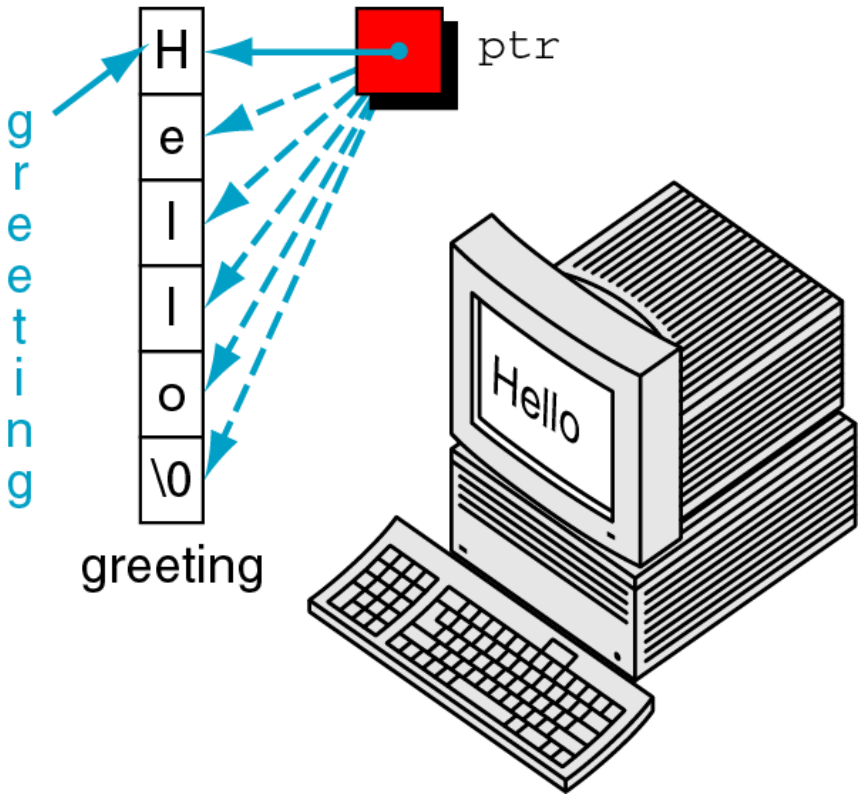



```
table = (int **)calloc (rowNum + 1, sizeof(int*));

table[0] = (int*)calloc (4, sizeof(int));
table[1] = (int*)calloc (7, sizeof(int));
table[2] = (int*)calloc (1, sizeof(int));
table[3] = (int*)calloc (3, sizeof(int));
table[4] = (int*)calloc (2, sizeof(int));
table[5] = NULL;
```

Punteros a cadenas de caracteres

- Utiles en recorridos de cadenas de caracteres.

```
{ /* Printing Strings */
  char  greeting[] = "Hello";
  char *ptr;

  ptr = greeting;
  while (*ptr != '\0')
      {

      printf( "%c", *ptr);
      ptr++;
     } /* while */
  print("\n");
} /* Printing Strings */
```



Estructuras autoreferenciadas

- Para construir estructuras de datos más potentes es necesario incluir *dentro* de una estructura un campo que sea un puntero a la misma estructura.

```
struct alumno
{
   char dni[10];
   char nombre[100];
   struct fecha fnac;
   float notas[10];
   struct alumno *palumno;
};
```

Estructuras autoreferenciadas

- Estructuras que contienen uno o más punteros a una estructura del mismo tipo.
- Pueden ser enlazadas entre ellas para formar estructuras de datos útiles tales como listas, colas, pilas y árboles.
- Terminan con un puntero NULL.

```
struct nodo
{
  int dato;   /* campos necesarios */
  struct nodo *sig; /* puntero de enlace*/
};
struct nodo *pnodo;
```

Estructuras autoreferenciadas

```
struct Info {
    int item1;
    char *item2;
    float item3;
};
struct Nodo {
    struct Info info;
    struct Nodo *sig;
};
typedef struct Nodo *TipoLista;
TipoLista lista_vacia(void) ;
```

Asignación dinámica de memoria

- El propósito es obtener y liberar memoria durante la ejecución.
- Con la función malloc se reserva el número de bytes de memoria requerida. Devuelve un puntero (void *)
- Para determinar el tamaño de un nodo se usa el operador sizeof.
- Ejemplo:

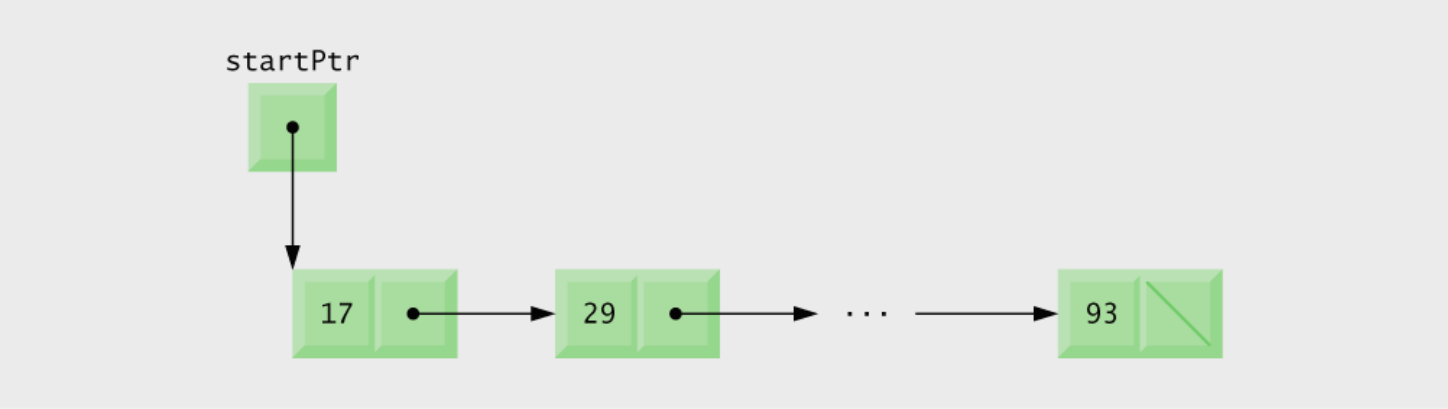
```
struct nodo *p;
p=(struct nodo *)malloc(sizeof(struct nodo));
```

Listas enlazadas

- Colección lineal de nodos autoreferenciados.
- Conectados por punteros enlazados.
- Acceso mediante un puntero al nodo inicial de la lista.
- Los nodos siguientes se acceden mediante el campo de puntero de enlace del nodo en curso.
- El puntero de enlace del último nodo se asigna a NULL para indicar final de la lista.
- Las listas enlazadas se usan en lugar de un array cuando se tiene un número impredecible de elementos de datos y cuando se requiere ordenar rápidamente la lista.

Listas enlazadas

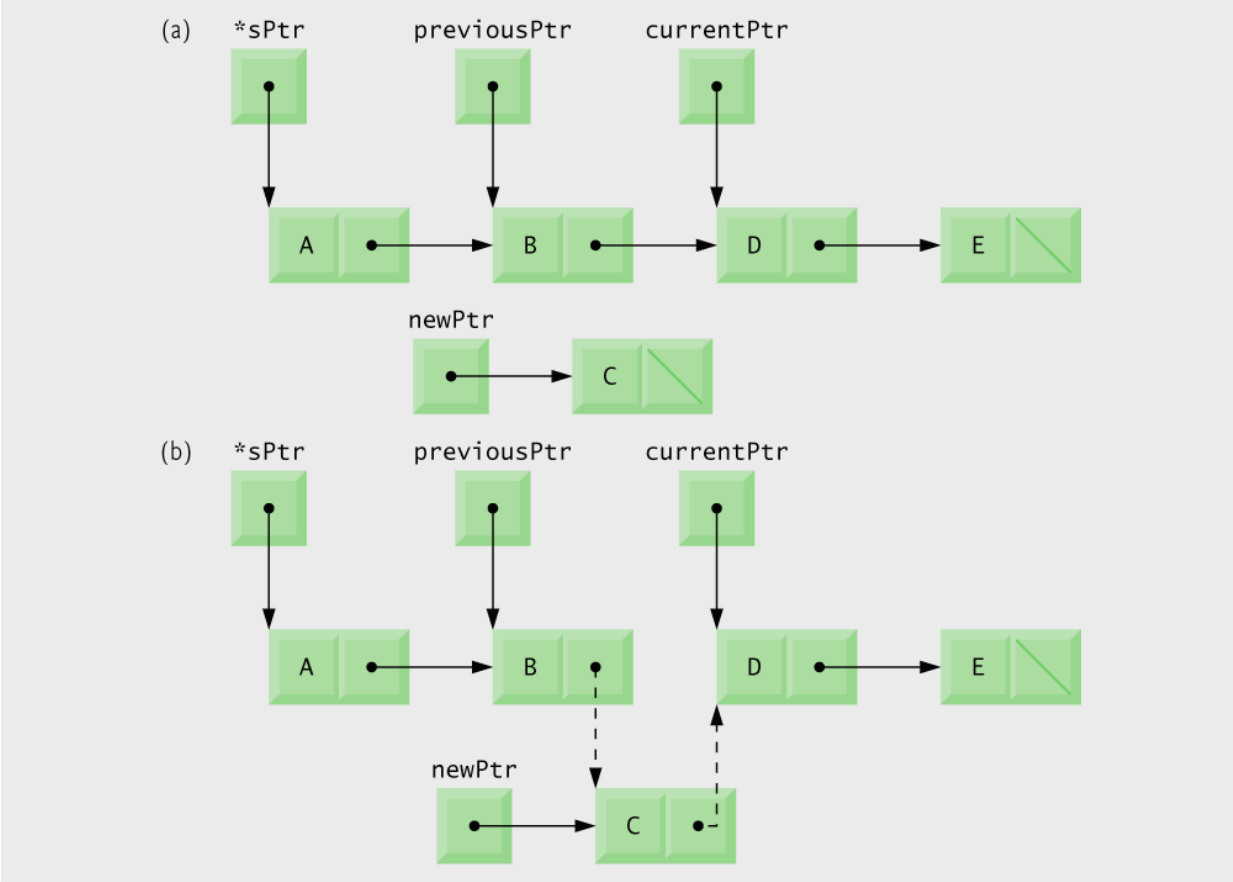
- Representación gráfica de una lista enlazada de números.



- Las operaciones sobre listas ordenadas más comunes son:
  - Crear_lista: inicializa lista a vacío.
  - Insertar: añade un elemento (nodo) a la lista.
  - Eliminar: suprime el nodo que contiene un elemento especificado de la lista.
  - ImprimeLista: imprime todos los elementos de la lista.
  - ListaVacía: operación booleana que indica si la lista está vacía.

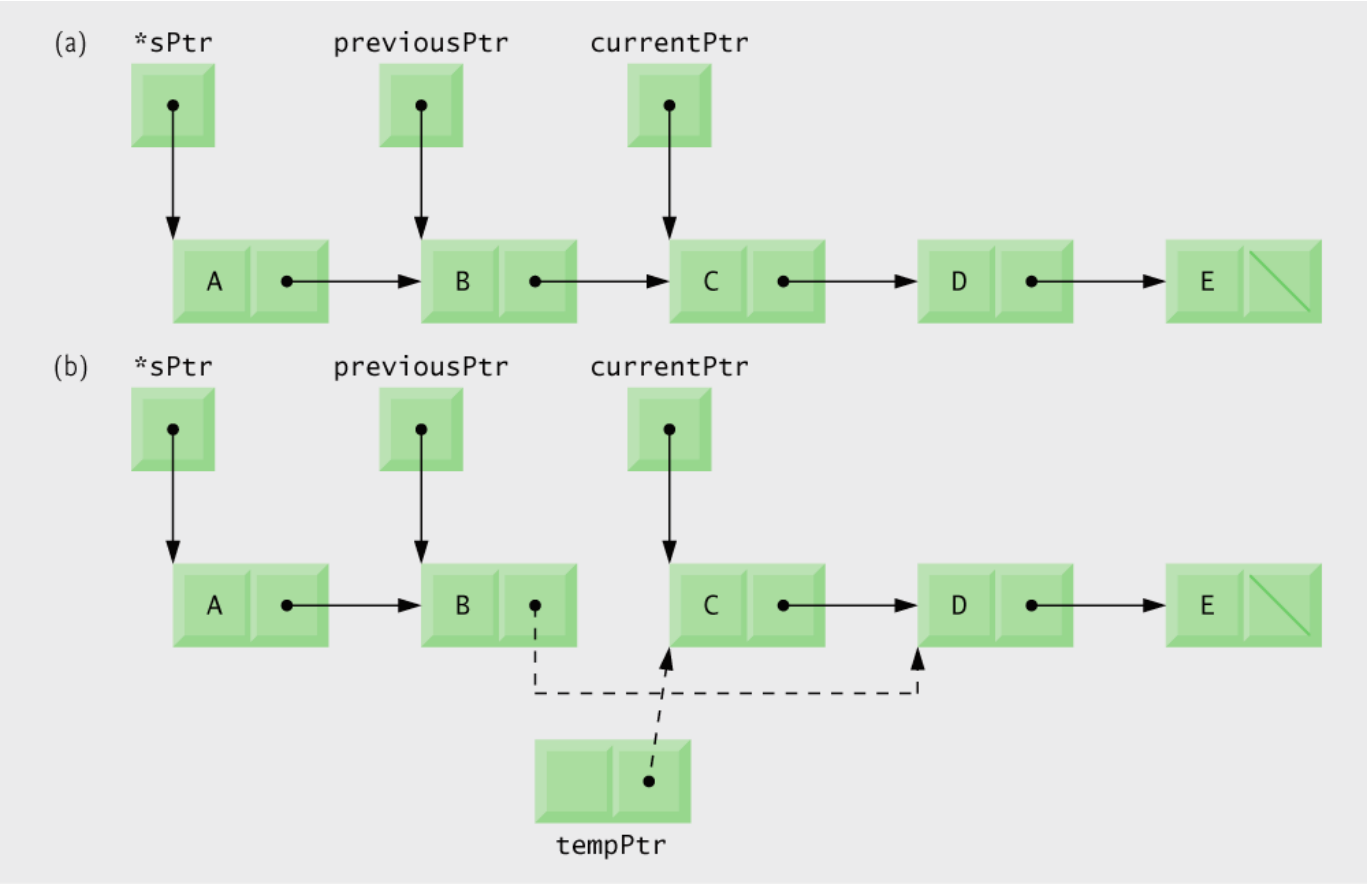
Listas enlazadas – Inserción de un nodo

- Representación gráfica de la inserción de un nodo en una lista ordenada.



Listas enlazadas – Eliminación de un nodo

- Representación gráfica de la eliminación de un nodo en una lista ordenada.



Listas enlazadas – Declaración

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Info {
    int item1;
    char *item2;
    float item3;
};

struct Nodo {
    struct Info info;
    struct Nodo *sig;
};

typedef struct Nodo *TipoLista;
```

Listas enlazadas – Declaración

```
TipoLista lista_vacia(void) ;
void lee_nodo(struct Info *inf);
void lee_nodof(struct Info *inf, FILE *fp);
void muestra (TipoLista lista);
TipoLista inserta (TipoLista lista);
void consulta_item2 (TipoLista lista);
void modifica_item2 (TipoLista lista);
TipoLista borra_item2 (TipoLista lista);
void guarda (TipoLista lista);
TipoLista recupera(TipoLista lista);
```

Listas enlazadas – Funciones

```
TipoLista lista_vacia (void) {
    return NULL;
}

void lee_nodo(struct Info *inf) {
    int i;
    char w[20], *pw;
    float x;

    printf("Ingresa entero: ");
    scanf(" %d", &i);
    printf("Ingresa palabra: ");
//    scanf(" %s", w);  // cadena simple
    scanf(" %[^\n]", w);   // cadena compuesta
    printf("Ingresa flotante: ");
    scanf(" %f", &x);
    inf->item1 = i;
    pw = malloc(strlen(w)+1);
    strcpy(pw,w);
    inf->item2= pw;
    inf->item3 = x;
}
```

Listas enlazadas – Funciones

```
TipoLista inserta (TipoLista lista)
{
    struct Info *inf = malloc (sizeof (struct Info));
    lee_nodo(inf);
    struct Nodo *nuevo = malloc (sizeof (struct Nodo));
    nuevo-> info = *inf;
    nuevo->sig= lista;
    lista = nuevo;
    return lista;
}

void muestra(TipoLista lista)
{
    struct Nodo *aux;
    printf("->");
    for (aux =lista; aux!=NULL; aux =aux->sig) {
        printf ("[%s]-> ",aux->info.item2);
//printf ("[%d, %s, %f]-> \n",aux->info.item1, aux->info.item2, aux->info.item3);
    }
    printf("\n");
}
```

Listas enlazadas – Funciones

```
TipoLista borra_item2 (TipoLista lista)
{
    char w[20];
    struct Nodo *aux, *atras;

    printf("item 2 a borrar: ");
    scanf(" %[^\n]", w);   // cadena compuesta

    for (atras=NULL, aux=lista; aux!=NULL; atras=aux, aux=aux->sig)
        if (strcmp(aux->info.item2, w)==0){
           if (atras==NULL)
                lista = aux-> sig;
            else
                atras->sig= aux-> sig;
            free (aux);
            printf("nodo eliminado\n");
            return lista;
        }
    return lista;
}
```

Listas enlazadas – Funciones

```
void consulta_item2 (TipoLista lista) {
    char w[20];
    struct Nodo *aux;
    printf("item 2 a consultar: ");
    scanf(" %[^\n]", w);   // cadena compuesta
    for (aux=lista; aux!=NULL; aux=aux->sig)
        if (strcmp(aux->info.item2, w)==0){
            printf("item 1: %d\n",aux->info.item1);
            printf("item 3: %.2f\n",aux->info.item3);
        }
}
void modifica_item2 (TipoLista lista){
    char w[20];
    struct Nodo *aux;
    printf("item 2 a modificar: ");
    scanf(" %[^\n]", w);   // cadena compuesta
    for (aux=lista; aux!=NULL; aux=aux->sig)
        if (strcmp(aux->info.item2, w)==0){
            lee_nodo(&aux->info);
        }
}
```

Listas enlazadas – Funciones

```
void guarda(TipoLista lista){
    struct Nodo *aux;
    FILE *fp;
    char c;
    printf("Guardar en fichero binario (b)?: ");
    scanf(" %c", &c);
    if (c=='b')
        fp = fopen("itemsb.dat", "wb");     // fichero binario
    else
        fp = fopen("itemst.dat", "w");     // fihcero texto
    if(fp == NULL)
    {
      printf("Error en escritura de fichero");
      exit(-1);
    }
    for (aux=lista; aux!=NULL; aux=aux->sig)
    {
        if (c=='b'){
             printf("%d  \n",aux->info.item1);
            fwrite(&aux->info, sizeof(struct Info), 1,fp);  // corregir error de grabacion de
    campos char *
        }
        else{
            fprintf(fp,"%d\n",aux->info.item1);
            fprintf(fp,"%s\n",aux->info.item2);
            fprintf(fp,"%.2f\n",aux->info.item3);
        }
    }
    fclose(fp);
    printf("fichero creado\n");
}
```

Listas enlazadas – Funciones

```
void lee_nodof(struct Info *inf, FILE *fp)
{
    int i;
    char w[20], *pw;
    float x;

    fscanf(fp," %d", &i);

    fscanf(fp," %[^\n]", w);   // cadena compuesta
    fscanf(fp," %f", &x);
    inf->item1 = i;
    pw = malloc(strlen(w)+1);
    strcpy(pw,w);
    inf->item2= pw;
    inf->item3 = x;
}
```

Listas enlazadas – Funciones

```
TipoLista recupera(TipoLista lista){
    FILE *fp;
//    struct Nodo *n;
    char c;
    printf("Fichero binario (b)?: ");
    scanf(" %c", &c);
    if (c=='b')
        fp = fopen("itemsb.dat", "rb");
    else
        fp = fopen("itemst.dat", "r");
    if(fp == NULL)
    {
      printf("Error en lectura de fichero");
      exit(-1);
    }
    while (!feof(fp))
    {
        struct Info *inf = malloc (sizeof (struct Info));
        if (c=='b')
            fread(inf, sizeof(struct Info), 1, fp);
        else
            lee_nodof(inf, fp);
        struct Nodo *nuevo = malloc (sizeof (struct Nodo));
        nuevo-> info = *inf;
        nuevo->sig= lista;
        lista = nuevo;
    }
    fclose(fp);
    printf("lista creada\n");
    return lista->sig;
}
```

Listas enlazadas – main

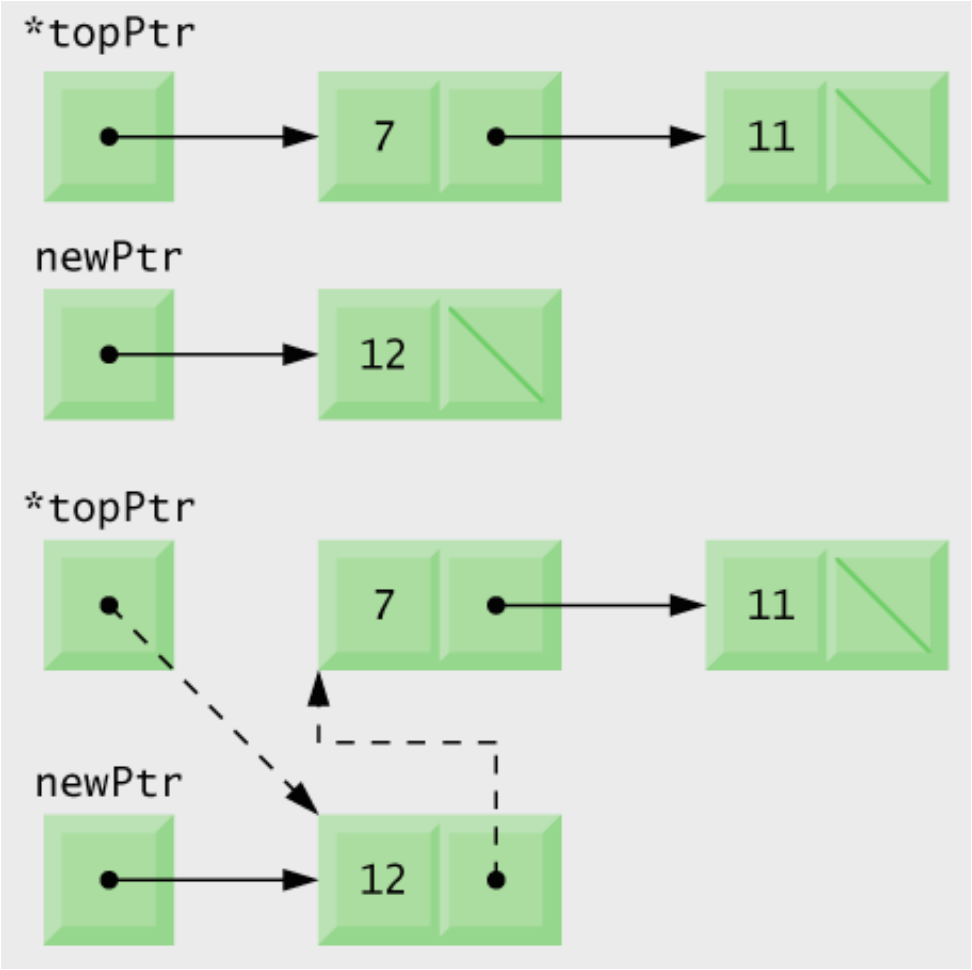
```
int main( )
{
    TipoLista l; int opcion=0, salir=0; char continuar[2];
    l =lista_vacia();   // inicializacion lista
    do {
        printf("\n MENU DE LA APLICACION: \n");
        printf("\t1) Mostrar\n"); printf("\t2) Insertar\n");
        printf("\t3) Consultar\n"); printf("\t4) Modificar\n");
        printf("\t5) Borrar\n"); printf("\t6) Guardar\n");
        printf("\t7) Recuperar\n"); printf("\t8) Salir\n");
        printf("  Elige una opcion del menu: "); scanf(" %d",&opcion);
        switch(opcion){
            case 1: muestra(l); break;         //mostrar
            case 2: //insertar
                do {
                    l = inserta(l); printf("Desea continuar (si/no): "); scanf(" %s", continuar);
                } while (strcmp(continuar, "no")!=0); break;
            case 3: consulta_item2(l); break; //consultar
            case 4: modifica_item2(l); break; //modificar
            case 5: borra_item2(l);break;     //Borrar
            case 6: guarda(l); break; //guardar
            case 7: l=recupera(l); break; //recuperar
            case 8: salir = 1; break;
            default:
                printf("\nOpcion incorrecta\n");
        } // switch
    } while (salir != 1);
    printf("\nPrograma terminado\n");
    return 0;
}
```

Pilas

- Son listas tipo LIFO (Last Input First Output).
- Los nuevos nodos se insertan y se eliminan sólo en la cima.
- El último nodo de la pila se indica por un enlace a NULL.
- Las operaciones sobre pilas más comunes son:
  - push: añadir un nuevo nodo en la cima de la pila.
  - pop: elimina un nodo de la cima.

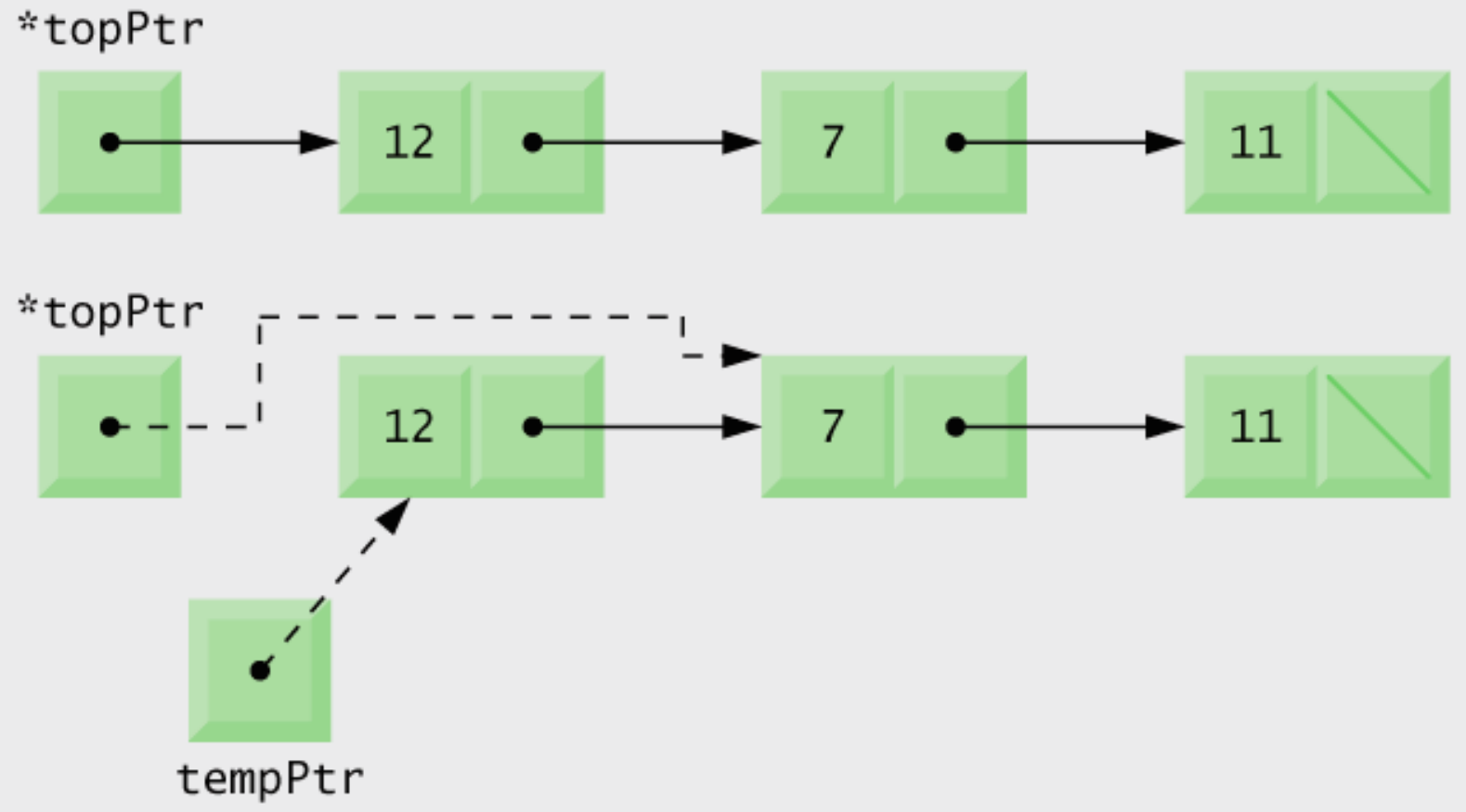
Pilas – Operación push

- Añadir un nuevo nodo en la cima de la pila



Pilas – Operación pop

- Elimina un nodo en la cima de la pila

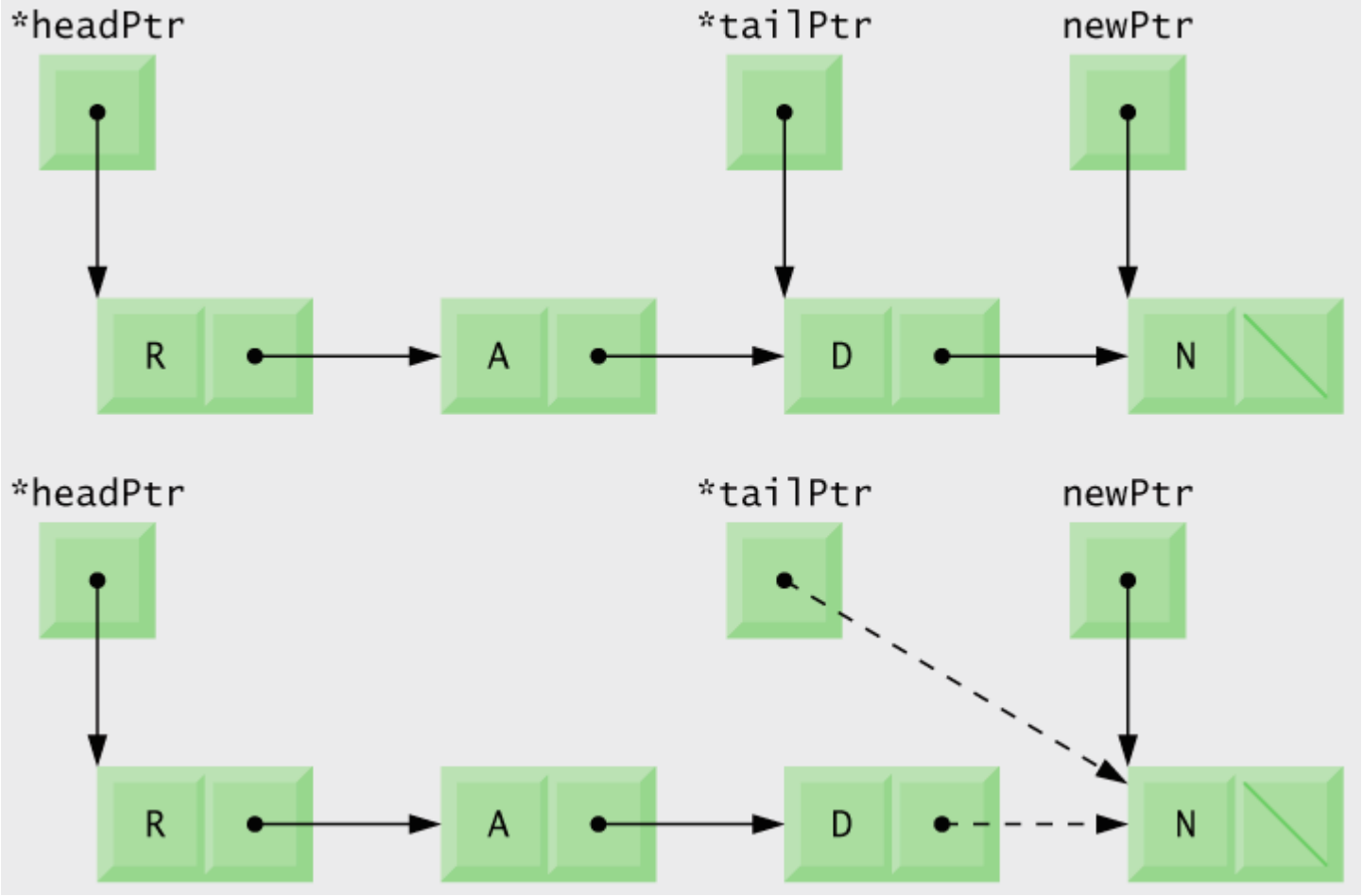


Colas

- Son listas tipo FIFO (First Input First Output).
- Similar a una cola habitual de un banco, servicio, etc.
- Los nuevos nodos se insertan por la cola.
- Los nodos se eliminan por el frente.
- Las operaciones sobre colas más comunes son:
  - encolar: insertar un nuevo nodo por la cola.
  - desencolar: eliminar el nodo del frente.
- Representación gráfica de una cola:

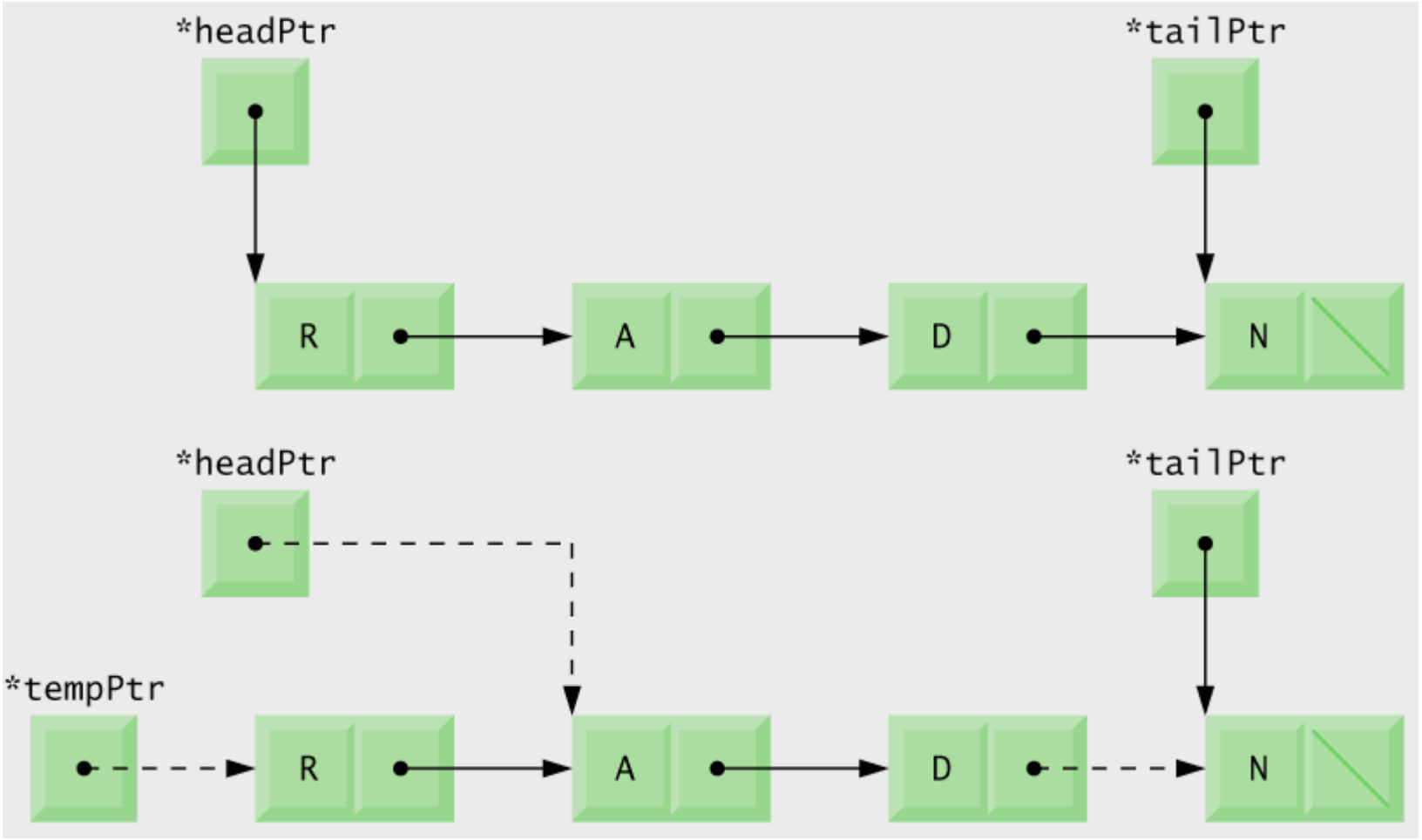
Colas – Operación encolar

- Inserta un nodo por la cola.



Colas – Operación desencolar

- Elimina un nodo por el frente.



Simulación de colas

- Let time move in one-minute increments.
- Each minute, check to see whether a new customer has arrived.
- If a customer arrives and the queue isn't full, add the customer to the queue.
- This involves recording in an Item structure the customer's arrival time and the amount of consultation time the customer wants, and then adding the item to the queue.
- If the queue is full, however, turn the customer away. For bookkeeping, keep track of the total number of customers and the total number of "turnaways“ (people who can't get in line because it is full).

Simulación de colas

- Next, process the front of the queue. That is, if the queue isn't empty and if Server isn't occupied with a previous customer, remove the item at the front of the queue. The item, recall, contains the time when the customer joined the queue. By comparing this time with the current time, you get the number of minutes the customer has been in the queue. The item also contains the number of consultation minutes the customer wants, which determines how long Server will be occupied with the new customer. Use a variable to keep track of this waiting time.
- If Server is busy, no one is "dequeued." However, the variable keeping track of the waiting time should be decremented.

Simulación de cola

```
#include <stdio.h>
#include <stdlib.h>    // for rand() and srand()
#include <time.h>      // for time()
#include "queue.h"     // change Item typedef
#define MIN_PER_HR 60.0
bool newcustomer(double x);   // is there a new customer?
Item customertime(long when); // set customer parameters
int main(void)
{
    Queue line;
    Item temp;                // new customer data
    int hours;                // hours of simulation
    int perhour;              // average # of arrivals per hour
    long cycle, cyclelimit;   // loop counter, limit
    long turnaways = 0;       // turned away by full queue
    long customers = 0;       // joined the queue
    long served = 0;          // served during the simulation
    long sum_line = 0;        // cumulative line length
    int wait_time = 0;        // time until Sigmund is free
    double min_per_cust;      // average time between arrivals
    long line_wait = 0;       // cumulative time in line

    InitializeQueue(&line);
    srand((unsigned int) time(0)); // random initializing of rand()
    puts("Caso de uso: Cola en gran superficie");
    puts("Ingresa el numero de horas de simulacion:");
    scanf("%d", &hours);
    cyclelimit = MIN_PER_HR * hours;
    puts("Ingresa el numero promedio de clientes por hora:");
    scanf("%d", &perhour);
    min_per_cust = MIN_PER_HR / perhour;
```

Simulación de cola

```
    for (cycle = 0; cycle < cyclelimit; cycle++) {
        if (newcustomer(min_per_cust)) {
            if (QueueIsFull(&line))
                turnaways++;
            else {
                customers++;
                temp = customertime(cycle);
                EnQueue(temp, &line);
            }
        }
        if (wait_time <= 0 && !QueueIsEmpty(&line)) {
            DeQueue (&temp, &line);
            wait_time = temp.processtime;
            line_wait += cycle - temp.arrive;
            served++;
        }
        if (wait_time > 0)
            wait_time--;
        sum_line += QueueItemCount(&line);
    }
    if (customers > 0) {
        printf("clientes aceptados: %ld\n", customers);
        printf(" clientes servidos: %ld\n", served);
        printf("         abandonos: %ld\n", turnaways);
        printf(" tamano medio cola: %.2f\n", (double) sum_line / cyclelimit);
        printf("tiempo espera medio: %.2f minutes\n",(double) line_wait / served);
    }
    else puts("Sin clientes!");
    EmptyTheQueue(&line);
    puts("Fin"); return 0;
}
```

Simulación de cola

```
// x = average time, in minutes, between customers
// return value is true if customer shows up this minute
bool newcustomer(double x)
{
    if (rand() * x / RAND_MAX < 1)
        return true;
    else
        return false;
}

// when is the time at which the customer arrives
// function returns an Item structure with the arrival time
// set to when and the processing time set to a random value
// in the range 1 - 3
Item customertime(long when)
{
    Item cust;

    cust.processtime = rand() % 3 + 1;
    cust.arrive = when;

    return cust;
}
```

Cola - header

```
#ifndef _QUEUE_H_
#define _QUEUE_H_
#include <stdbool.h>

typedef struct item {
    long arrive;      // the time when a customer joins the queue
    int processtime;  // the number of consultation minutes desired
} Item;
#define MAXQUEUE 10
typedef struct node{
    Item item;    struct node * next;
} Node;
typedef struct queue{
    Node * front;  /* pointer to front of queue  */
    Node * rear;   /* pointer to rear of queue   */
    int items;     /* number of items in queue   */
} Queue;

void InitializeQueue(Queue * pq); /* operation:  initialize the queue     */
bool QueueIsFull(const Queue * pq); /* operation:  check if queue is full   */
bool QueueIsEmpty(const Queue *pq); /* operation:  check if queue is empty     */
/* operation:        determine number of items in queue         */
int QueueItemCount(const Queue * pq);
/* operation:        add item to rear of queue                  */
bool EnQueue(Item item, Queue * pq);
/* operation:        remove item from front of queue            */
bool DeQueue(Item *pitem, Queue * pq);
/* operation:        empty the queue                            */
void EmptyTheQueue(Queue * pq);
#endif
```

Cola - funciones

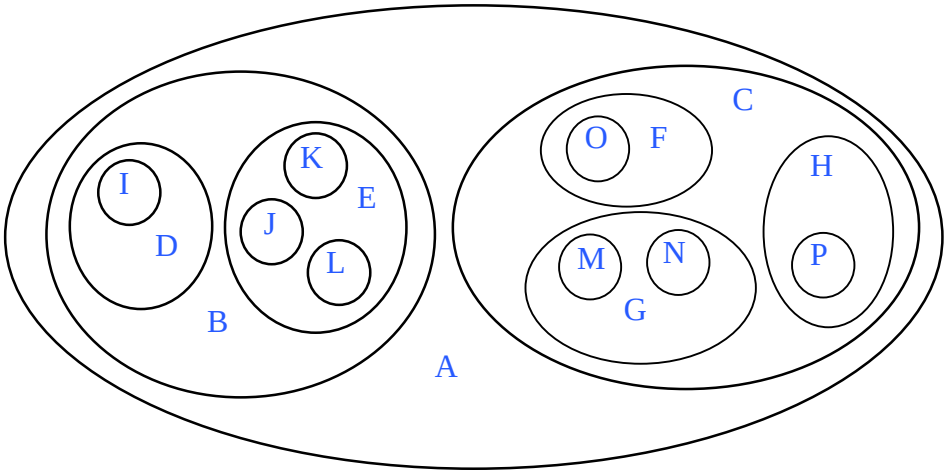
`queue.c`

Arboles

- Son estructuras de datos en las que los nodos contiene dos o más enlaces.
- Definición a nivel lógico: Una estructura de árbol con tipo base T es:
  - La estructura vacía.
  - Un nodo de tipo T con un número finito de estructuras árbol disjuntas asociadas de tipo base T, llamadas subárboles conectadas por ramas o aristas.

Arboles - Representación

- Conjuntos anidados:



- Paréntesis anidados:

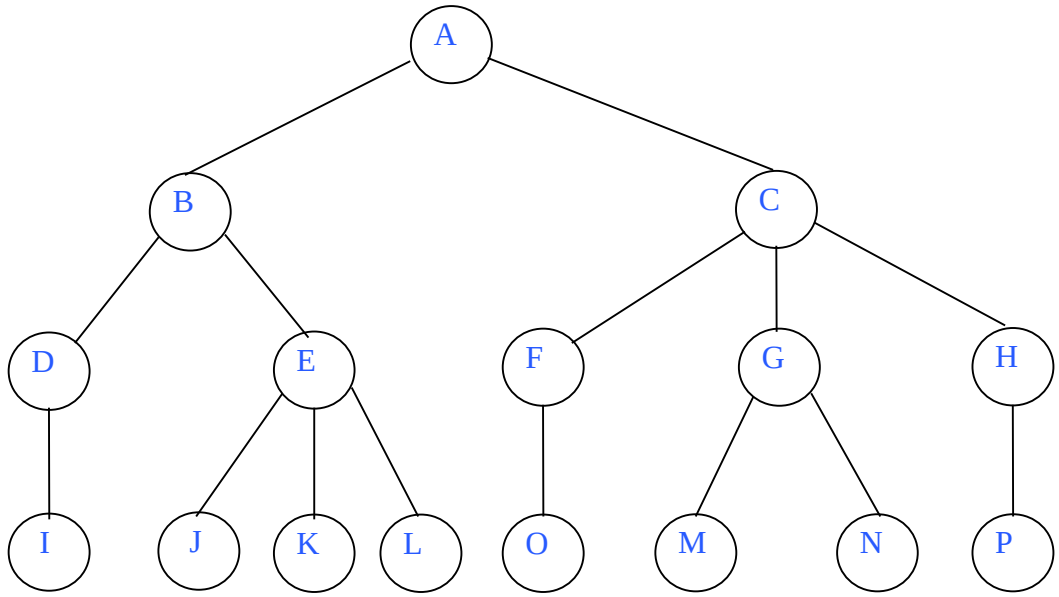
(A (B (D (I), E (J, K, L)), C (F (O), G (M, N), H (P))))

Arboles - Representación

- Indentación:

A
B
D
I
E
J
K
L
C
F
O
G
M
N
H
P

- Grafo:



La representación más utilizada es la de grafo

Arboles – definiciones de términos

- Tipos de **Nodos**:
  - Nodo Raíz
  - Nodo Descendiente o Hijo
  - Nodo hoja o Interior
  - Nodo antecesor o sucesor
- **Altura**: Número de aristas o ramas desde la raíz hasta el nodo hoja más distante desde éste. Por definición el nodo raíz esté en el nivel 1.
- **Profundidad de un nodo**: número de aristas del camino desde la raíz hasta el nodo.

Arboles – definiciones de términos

- **Grado de un nodo**: número de hijos o descendientes de un nodo interior.
- **Grado de un árbol**: máximo grado de los nodos de un árbol.
- Máximo número de nodos en un árbol de altura h y grado d:

$$N_d(h) = 1 + d + d^2 + \ldots + d^{h-1} = \sum_{i=0}^{h-1} d^i$$

- Para d=2 (árbol binario):

$$N_2(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

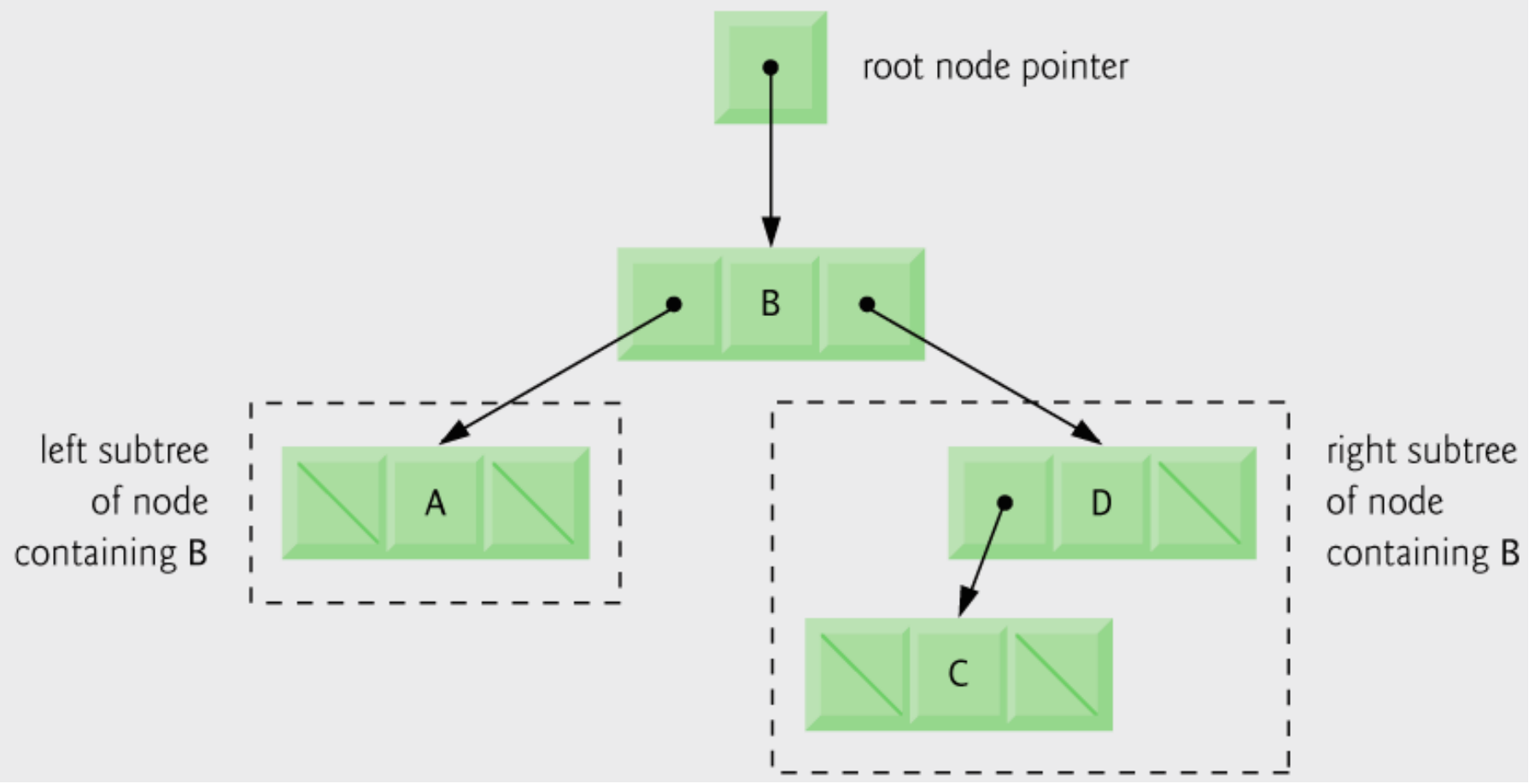
- Profundidad de un árbol binario de n nodos: $\boldsymbol{h = \log_2 n + 1}$

Arboles binarios

- Son aquellos árboles en los que todos los nodos contienen dos enlaces
  - Ninguno, uno, o ambos de los cuales pueden ser NULL
- El **nodo raíz** es el primer nodo en el árbol.
- Cada enlace en el nodo raíz se refiere a un **nodo hijo**.
- Un nodo sin hijos se llama **nodo hoja**.
- Representación en C:

```
struct arbol {
    int  data;
    struct  arbol *izq;
    struct  arbol *der;
};
struct arbol  *raiz = NULL;
```

Arboles binarios – representación gráfica



Recorrido en árboles binarios

- Inorden (SI R SD)
  - Ir hacia el subárbol izquierdo hasta alcanzar la máxima profundidad.
  - Visitar el nodo en curso.
  - Volver hacia el nodo anterior en el árbol y visitarlo.
  - Ir hacia el subárbol derecho del nodo anteriormente visitado siempre que exista y no haya sido visitado previamente, de otra forma, volver hacia el nodo anterior.
  - Repetir los pasos anteriores hasta que todos los nodos hayan sido procesados.

Recorrido en árboles binarios

- Inorden (SI R SD)

```
/* in_orden: imprime el contenido del
  arbol con raiz p en in-orden */
void in_orden(struct arbol *p)
{
  if (p!=NULL) {
     in_orden(p->izq);
     printf("%4d   ",p->data);
     in_orden(p->der);
  }
}
```

Recorrido en árboles binarios

- Preorden (R SI SD)

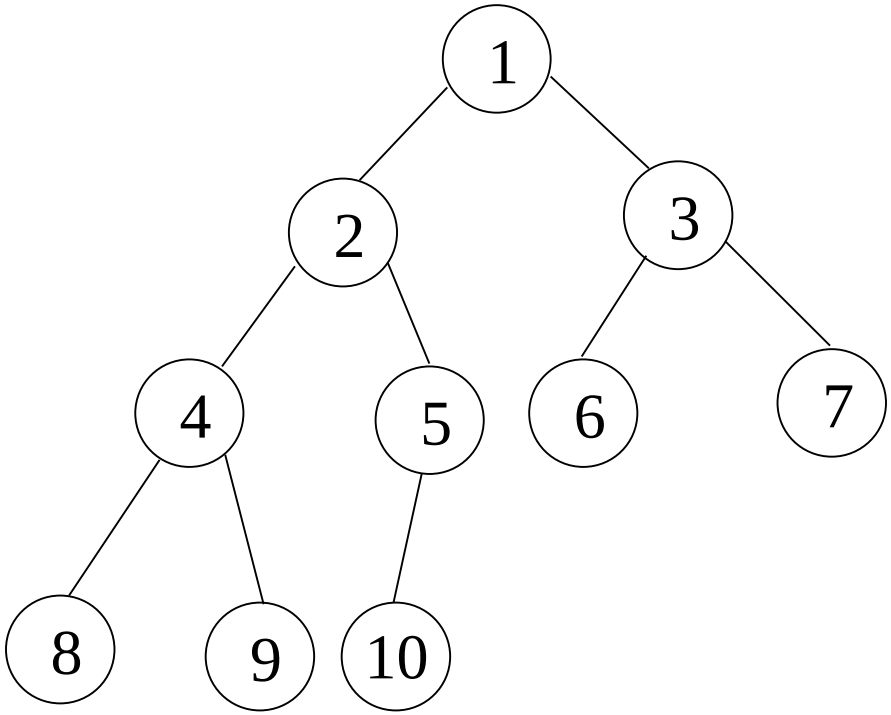
```
/* pre_orden: imprime el contenido del
  arbol con raiz p en pre-orden */
void pre_orden(struct arbol *p)
{
  if (p!=NULL) {
     printf("%4d   ",p->data);
     pre_orden(p->izq);
     pre_orden(p->der);
  }
}
```

Recorrido en árboles binarios

- Postorden (SI SD R)

```
/* post_orden: imprime el contenido
  del arbol con raiz p en post-orden */
void post_orden(struct arbol *p)
{
  if (p!=NULL) {
     post_orden(p->izq);
     post_orden(p->der);
     printf("%4d   ",p->data);
  }
}
```

Recorrido en Arboles



- Inorden: 8 4 9 2 10 5 1 6 3 7
- Preorden: 1 2 4 8 9 5 10 3 6 7
- Postorden: 8 9 4 10 5 2 6 7 3 1

Arbol binario de búsqueda

- Es un árbol en el que el hijo de la izquierda, si existe, de cualquier nodo contiene un valor más pequeño que el nodo padre, y el hijo de la derecha, si existe, contiene un valor más grande que el nodo padre.
- Operaciones:
  - Buscar_Arbol(ValorClave)
  - Insertar(InfoNodo)
  - Suprimir(ValorClave)
  - ImprimeArbol(OrdenRecorrido)

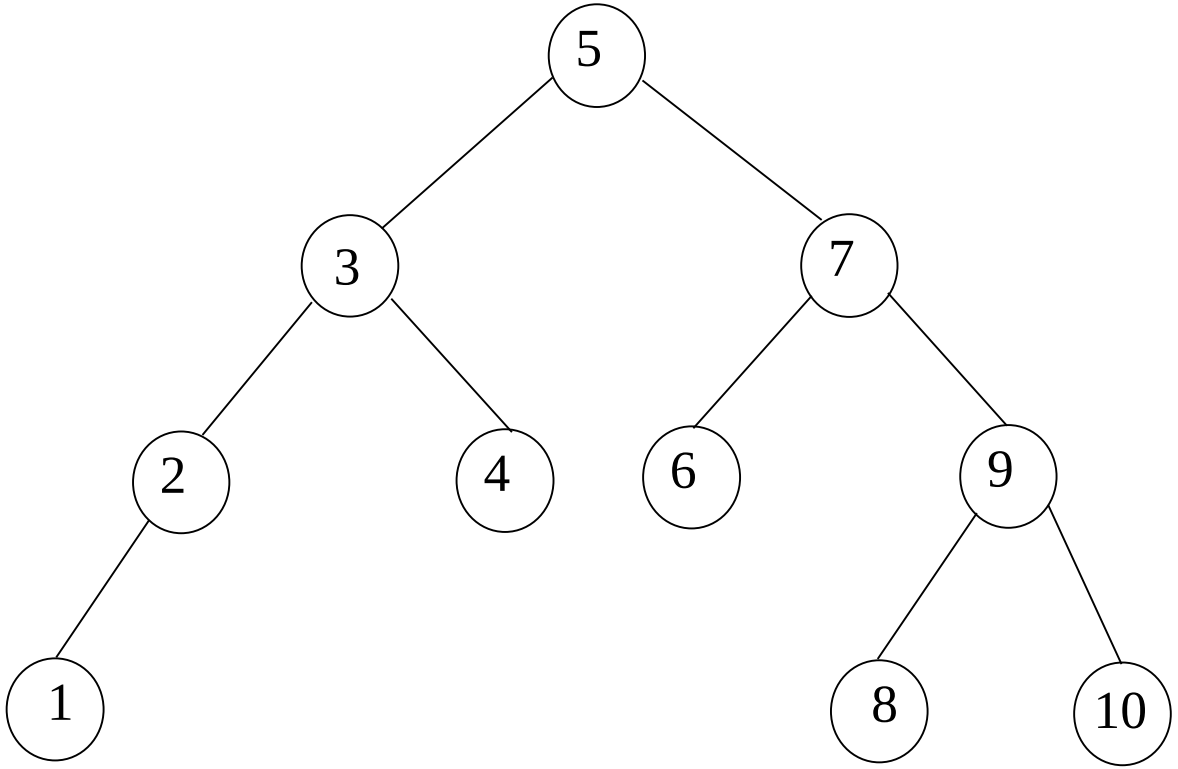
Inserción en un árbol binario de búsqueda

- Sólo se pueden insertar nuevos nodos en los nodos terminales (hojas) o en los nodos internos.
- En un árbol binario de búsqueda se restringe la inserción de nuevos nodos a los nodos terminales debido a la condición de ordenamiento entre los nodos.
- Procedimiento:
  - Se recorre el árbol comparando el nuevo valor con los nodos existentes, dirigiéndose por los descendientes - izquierda o derecha - según el valor a añadir es menor o mayor respectivamente que el nodo en curso.
  - Cuando se llega a un nodo terminal se inserta el nuevo valor como un descendiente de este nodo.

Inserción en un árbol binario de búsqueda

- Construir el árbol binario de búsqueda producido por la siguiente lista:

  5, 3, 4, 7, 2, 6, 9, 1, 8, 10



Inserción en un ABB - código

```
/* addtree: anade un nodo con palabra w. p -> raiz */
struct tnodo *addtree(struct tnodo *p, char *w)
{
 int cond;

 if (p == NULL) {
     p = talloc();
     p->palabra = strdup(w);
     p->cont = 1;
     p->izq = p->der = NULL;
 } else if ((cond = strcmp(w,p->palabra))==0)
     p->cont++;
 else if (cond < 0)
     p->izq= addtree(p->izq, w);
 else
     p->der = addtree(p->der, w);
 return p;
}
```

Inserción en un ABB – código func. aux.

```
#include <stdlib.h>
/* talloc: construye un nodo */
struct tnodo *talloc(void)
{
 return (struct tnodo *) malloc(sizeof(struct tnodo));
}

char *strdup(char *s)
{
 char *p;

 p = (char *) malloc(strlen(s)+1);
 if (p != NULL)
    strcpy(p,s);
 return p;
}
```

Ejemplo de aplicación de un ABB

- Desarrollo de un programa para contar las ocurrencias de las palabras de un texto.
- Como se desconoce la lista de palabras que contiene el texto no puede pensarse en ordenarlas y usar búsqueda binaria.
- Otra alternativa es usar una búsqueda lineal según se lee cada palabra, pero esto tomaría demasiado tiempo (crece cuadráticamente) cuando el número de palabras aumenta.
- La mejor solución, con lo conocido hasta ahora, es tener el conjunto de palabras leídas en cualquier momento ordenadas colocando cada palabra nueva en su lugar. Para ello usaremos un árbol binario de búsqueda.

Frecuencia de ocurrencia de palabras

- El árbol contendrá en cada nodo la siguiente información:
  - Un puntero al texto de la palabra,
  - Un contador del número de ocurrencias,
  - Punteros a los hijos izquierdo y derecho respectivamente.

Frec. de ocurrencia de palabras – código

```
/*********************************************************************\
* Programa: frecuencia_palabras.c                                     *
* Descripción: Prog. que cuenta la frecuencia de ocurrencia de palabras  *
*              de un texto leido desde la linea de comandos           *
*              (redireccionando la entrada)                           *
*              El resultado es una lista ordenada de las palabras junto  *
*              con la frecuencia                                      *
*              El texto debe de constar de letras sin acento y sin signos*
*              de puntuacion                                          *
* Autor: Pedro Corcuera                                               *
* Revisión: 1.0 2/02/2008                                             *
\*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <ctype.h>

#define MAX_CAR 100

struct arb_bin {
   char *palabra;
   int cont;
   struct arb_bin *izq;
   struct arb_bin *der;
};

struct arb_bin *inserta(struct arb_bin *, char *);
void imprime(struct arb_bin *);
```

Frec. de ocurrencia de palabras – código

```
main()
{
 struct arb_bin *raiz;
 char palabra[MAX_CAR];
 raiz=NULL;
 while(scanf(" %s",palabra)!=EOF)
   if (isalpha(palabra[0])) raiz=inserta(raiz,palabra);
 imprime(raiz);
 return 0;
}

struct arb_bin *talloc(void);
char *strdup(char *);

/* inserta: anade un nodo con w, en o bajo p */
struct arb_bin *inserta(struct arb_bin *p, char *w)
{
 int cond;
 if (p==NULL) {
     p = talloc(); p->palabra = strdup(w);
     p->cont = 1; p->izq = p->der = NULL;
 } else if ((cond=strcmp(w,p->palabra))==0)
     p->cont++;
 else if (cond<0) p->izq= inserta(p->izq,w);
 else p->der = inserta(p->der,w);
 return p;
}
```

Frec. de ocurrencia de palabras – código

```
/* imprime: imprime p en in-orden */
void imprime(struct arb_bin *p)
{
 if (p!=NULL) {
    imprime(p->izq);
    printf("%4d  %s\n",p->cont,p->palabra);
    imprime(p->der);
 }
}

/* talloc: construye un nodo del arbol binario */
struct arb_bin *talloc(void)
{
 return (struct arb_bin *) malloc(sizeof(struct arb_bin));
}

/* devuelve un puntero a una copia de una cadena de caracteres */
char *strdup(char *s)
{
 char *p;

 p = (char *) malloc(strlen(s)+1);
 if (p != NULL)
    strcpy(p,s);
 return p;
}
```