Algoritmos en C

Índice

- Algoritmos de ordenación.
- Algoritmos de búsqueda.
- Algoritmos de vuelta atrás.
- Algoritmo de Huffman.

Algoritmos de Ordenación (1)

- Ordenación o clasificación es el proceso de reordenar un conjunto de objetos en un orden específico.
- El propósito de la ordenación es facilitar la búsqueda de elementos en el conjunto ordenado.
- Existen muchos algoritmos de ordenación, siendo la diferencia entre ellos la eficiencia en tiempo de ejecución.
- Los métodos de ordenación se pueden clasificar en dos categorías: ordenación de ficheros o externo y ordenación de arrays o interno.
- Aquí trataremos sólo del ordenamiento interno.

Algoritmos de Ordenación (2)

- El problema del ordenamiento puede establecerse mediante la siguiente ación: Dados los elementos: $a_1, a_2, \ldots, a_n$

  Ordenar consiste en permutar esos elementos en un orden: $a_{k_1}, a_{k_2}, \ldots, a_{k_n}$ tal que dada una función de ordenamiento f: $f(a_{k_1}) \leq f(a_{k_2}) \leq \ldots \leq f(a_{k_n})$
- Normalmente, la función de ordenamiento se guarda como un componente explícito (campo) de cada item (elemento). El valor de ese campo se llama la *llave del item*.
- Un método de ordenamiento es estable si el orden relativo de elementos con igual llave permanece inalterado por el proceso de ordenamiento.

Algoritmos de Ordenación (3)

- Se entiende que los métodos de ordenamiento buscan un uso eficiente de la memoria por lo que las permutaciones de elementos se hará *in situ*, es decir, usando el mismo array original.
- En lo que sigue se considera que el array a ordenar consiste de números reales:

  float a[MAX];

  siendo MAX el número máximo de elementos del array.
- El orden de los elementos después de la ordenación se considera ascendente.

Ordenación por burbuja

- Es un método caracterizado por la *comparación* e *intercambio* de pares de elementos hasta que todos los elementos estén ordenados.
- En cada iteración se coloca el elemento más pequeño (orden ascendente) en su lugar correcto, cambiándose además la posición de los demás elementos del array.
- La complejidad del algoritmo es $O(n^2)$.

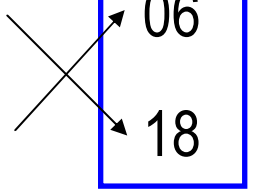
Ordenación por burbuja

Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44							
55							
12							
42							
94							
18							
06 →	06						
67 →	67						

67 < 06 no hay intercambio

Ordenación por burbuja

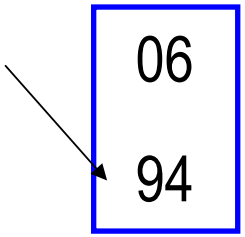
Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44							
55							
12							
42							
94							
18	06						
06	18						
67	67						



06 < 18 hay intercambio

Ordenación por burbuja

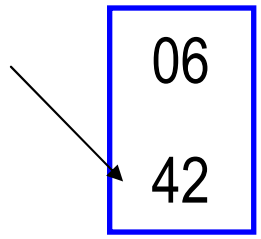
Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44							
55							
12							
42							
94	06						
18	94						
06	18						
67	67						



06 < 94 hay intercambio

Ordenación por burbuja

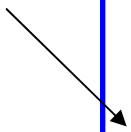
Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44							
55							
12							
42	06						
94	42						
18	94						
06	18						
67	67						



06 < 42 hay intercambio

Ordenación por burbuja

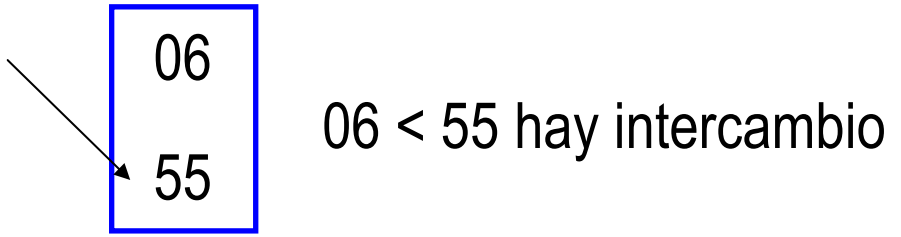
Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44							
55							
12	06						
42	12						
94	42						
18	94						
06	18						
67	67						



06 < 12 hay intercambio

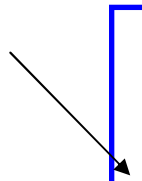
Ordenación por burbuja

Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44							
55	06						
12	55						
42	12						
94	42						
18	94						
06	18						
67	67						



Ordenación por burbuja

Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44	06						
55	44						
12	55						
42	12						
94	42						
18	94						
06	18						
67	67						



06 < 44 hay intercambio

Ordenación por burbuja

Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44	06	06					
55	44	12					
12	55	44					
42	12	55					
94	42	18					
18	94	42					
06	18	94					
67	67	67					

Ordenación por burbuja

Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44	06	06	06				
55	44	12	12				
12	55	44	18				
42	12	55	44				
94	42	18	55				
18	94	42	42				
06	18	94	67				
67	67	67	94				

Ordenación por burbuja

Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44	06	06	06	06			
55	44	12	12	12			
12	55	44	18	18			
42	12	55	44	42			
94	42	18	55	44			
18	94	42	42	55			
06	18	94	67	67			
67	67	67	94	94			

Ordenación por burbuja

Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44	06	06	06	06	06		
55	44	12	12	12	12		
12	55	44	18	18	18		
42	12	55	44	42	42		
94	42	18	55	44	44		
18	94	42	42	55	55		
06	18	94	67	67	67		
67	67	67	94	94	94		

Ordenación por burbuja

Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44	06	06	06	06	06	06	
55	44	12	12	12	12	12	
12	55	44	18	18	18	18	
42	12	55	44	42	42	42	
94	42	18	55	44	44	44	
18	94	42	42	55	55	55	
06	18	94	67	67	67	67	
67	67	67	94	94	94	94	

Ordenación por burbuja

Original	1ª iter	2ª iter	3ª iter	4ª iter	5ª iter	6ª iter	7ª iter
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

Ordenación por burbuja: código

```
/*****************************************************************\
* Programa: ordena_bubble.c                                       *
* Descripción: Programa que ordena un array mediante el metodo de burbuja*
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>

main()
{
 float t, a[]={10.0, 8.0, 5.5, 3.4, 3.2, 2.5, 2.2, 1.5};
 int i, j, n, ninterc = 0;

 n = sizeof(a)/sizeof(float);

 for(i = 0; i < n; i++)
  for(j = n-1; j >= i; j--)
   if (a[j] > a[j+1])      /* orden ascendente */
   {
    t = a[j];
    a[j] = a[j+1];
    a[j+1] = t;
    ninterc++;             /* Contador de intercambios */
   }
 for(i = 0;i < n;i++)
   printf("%d  %10.2f \n",i,a[i]);
 printf("Numero de intercambios = %d \n",ninterc);
}
```

Ordenación por shakesort (agitación)

- El algoritmo de burbuja puede mejorarse registrando la ocurrencia de un intercambio y el índice del último intercambio. Además se alterna la dirección de las pasadas consecutivas.
- Con ello una burbuja liviana en el lado “pesado” y una pesada en el lado “liviano” quedarán en orden en una pasada simple.
- La complejidad del algoritmo es $O(n^2)$ si el conjunto está aleatoriamente desordenado. Si está ordenado el algoritmo es O(n).

Ordenación por agitación: código

```
/*********************************************************************\
* Programa: skahesort.c                                               *
* Descripción: Programa que ordena un array mediante el metodo agitacion *
* Autor: Pedro Corcuera                                               *
* Revisión: 1.0 2/02/2008                                             *
\*********************************************************************/
#include <stdio.h>

main()
{
 float t,
/*  a[]={10.0, 8.0, 5.5, 3.4, 3.2, 2.5, 2.2, 1.5};*/
 a[]={1.5, 2.2, 2.5, 3.2, 3.4, 5.5, 8.0, 10.0};
 int i,j,k,l,r,n,ninterc=0;

 n = sizeof(a)/sizeof (float);
 l = 1;   /* indice para ordenar en sentido directo */
 r = n-1; /* indice para ordenar en sentido inverso */
 k = n-1; /* indice para registrar el intercambio */
 do
 {
  for(j = r; j >= l; j--)
   if (a[j-1] > a[j])
   {
    t = a[j-1]; a[j-1] = a[j]; a[j] = t;
    k = j;
    ninterc++;
   }
```

Ordenación por agitación: código

```
 l = k+1;
  for(j = l; j <= r; j++)
   if (a[j-1] > a[j])
   {
    t = a[j-1];
    a[j-1] = a[j];
    a[j] = t;
    k = j;
    ninterc++;
   }
  r = k-1;
 }
 while (l < r);

/* Impresión del array ordenado */
 for(i=0;i<n;i++)
   printf("%d  %10.2f \n",i,a[i]);
 printf("Numero de intercambios = %d \n",ninterc);
}
```

Ordenación por inserción

- Método usado para ordenar una mano de naipes.
- Los elementos están divididos conceptualmente en una secuencia destino $a_1, a_2, \ldots, a_{i-1}$ y una secuencia fuente $a_i, a_{i+1}, \ldots, a_n$.
- En cada paso, comenzando con i=2 e incrementando i en uno, el elemento i-ésimo de la secuencia fuente se toma y se transfiere a la secuencia destino insertándolo en el lugar adecuado.
- Es decir, en el i-ésimo paso insertamos el i-ésimo elemento a[i] en su lugar correcto entre a[1], a[2],…., a[i-1], que fueron colocados en orden previamente.

Ordenación por inserción

- Este algoritmo puede mejorarse fácilmente si vemos que la secuencia destino $a_1, a_2, \ldots, a_{i-1}$ está ordenada, por lo que usamos una búsqueda binaria para determinar el punto de inserción.
- La complejidad del algoritmo es $O(n^2)$.

Ordenación por inserción

Or	44	55	12	42	94	18	06	67
1	44	55	12	42	94	18	06	67
2								
3								
4								
5								
6								
7								

Ordenación por inserción

Or	44	55	12	42	94	18	06	67
1	44	55	12	42	94	18	06	67
2	12	44	55	42	94	18	06	67
3								
4								
5								
6								
7								

Ordenación por inserción

Or	44	55	12	42	94	18	06	67
1	44	55	12	42	94	18	06	67
2	12	44	55	42	94	18	06	67
3	12	42	44	55	94	18	06	67
4								
5								
6								
7								

Ordenación por inserción

Or	44	55	12	42	94	18	06	67
1	44	55	12	42	94	18	06	67
2	12	44	55	42	94	18	06	67
3	12	42	44	55	94	18	06	67
4	12	42	44	55	94	18	06	67
5								
6								
7								

Ordenación por inserción

Or	44	55	12	42	94	18	06	67
1	44	55	12	42	94	18	06	67
2	12	44	55	42	94	18	06	67
3	12	42	44	55	94	18	06	67
4	12	42	44	55	94	18	06	67
5	12	18	42	44	55	94	06	67
6								
7								

Ordenación por inserción

Or	44	55	12	42	94	18	06	67
1	44	55	12	42	94	18	06	67
2	12	44	55	42	94	18	06	67
3	12	42	44	55	94	18	06	67
4	12	42	44	55	94	18	06	67
5	12	18	42	44	55	94	06	67
6	06	12	18	42	44	55	94	67
7								

Ordenación por inserción

Or	44	55	12	42	94	18	06	67
1	44	55	12	42	94	18	06	67
2	12	44	55	42	94	18	06	67
3	12	42	44	55	94	18	06	67
4	12	42	44	55	94	18	06	67
5	12	18	42	44	55	94	06	67
6	06	12	18	42	44	55	94	67
7	06	12	18	42	44	55	67	94

Ordenación por inserción: código

```
/*****************************************************************\
* Programa: ord_insercion.c                                       *
* Descripción: Programa que ordena un array mediante el metodo insercion *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>

main()
{
 float t,a[]={10.0,8.0,5.5,3.4,3.2,2.5,2.2,1.5};
 int i,j,n;

 n = sizeof(a)/sizeof (float);
 for(i = 1; i < n; i++)
 {
  j = i-1;
  t = a[i];
  while (j >= 0 && t < a[j])
  {
    a[j+1] = a[j];
    j = j-1;
  }
  a[j+1] = t;
 }
 for(i = 0; i < n; i++)
   printf("%d  %10.2f \n",i,a[i]);
}
```

Ordenación por selección

- En éste método, en el i-ésimo paso seleccionamos el elemento con la llave de menor valor, entre a[i],…, a[n] y lo intercambiamos con a[i].
- Como resultado, después de i pasadas, el i-ésimo elemento menor ocupará a[1],…, a[i] en el lugar ordenado.
- La complejidad del algoritmo es $O(n^2)$.

Ordenación por selección

Or	44	55	12	42	94	18	06	67
1	06	55	12	42	94	18	44	67
2								
3								
4								
5								
6								
7								

Ordenación por selección

Or	44	55	12	42	94	18	06	67
1	06	55	12	42	94	18	44	67
2	06	12	55	42	94	18	44	67
3								
4								
5								
6								
7								

Ordenación por selección

Or	44	55	12	42	94	18	06	67
1	06	55	12	42	94	18	44	67
2	06	12	55	42	94	18	44	67
3	06	12	18	42	94	55	44	67
4								
5								
6								
7								

Ordenación por selección

Or	44	55	12	42	94	18	06	67
1	06	55	12	42	94	18	44	67
2	06	12	55	42	94	18	44	67
3	06	12	18	42	94	55	44	67
4	06	12	18	42	94	55	44	67
5								
6								
7								

Ordenación por selección

Or	44	55	12	42	94	18	06	67
1	06	55	12	42	94	18	44	67
2	06	12	55	42	94	18	44	67
3	06	12	18	42	94	55	44	67
4	06	12	18	42	94	55	44	67
5	06	12	18	42	44	55	94	67
6								
7								

Ordenación por selección

Or	44	55	12	42	94	18	06	67
1	06	55	12	42	94	18	44	67
2	06	12	55	42	94	18	44	67
3	06	12	18	42	94	55	44	67
4	06	12	18	42	94	55	44	67
5	06	12	18	42	44	55	94	67
6	06	12	18	42	44	55	94	67
7								

Ordenación por selección

Or	44	55	12	42	94	18	06	67
1	06	55	12	42	94	18	44	67
2	06	12	55	42	94	18	44	67
3	06	12	18	42	94	55	44	67
4	06	12	18	42	94	55	44	67
5	06	12	18	42	44	55	94	67
6	06	12	18	42	44	55	94	67
7	06	12	18	42	44	55	67	94

Ordenación por selección: código

```
/*****************************************************************\
* Programa: ord_seleccion.c                                       *
* Descripción: Programa que ordena un array mediante el metodo seleccion *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>

main()
{
 float t,a[]={10.0,8.0,5.5,3.4,3.2,2.5,2.2,1.5};
 int i,j,k,n;

 n = sizeof(a)/sizeof (float);
 for(i = 0; i < n-1; i++)
 {
  k = i; t = a[i];
  for (j = i+1; j < n; j++)
  {
    if (a[j] < t)
    {  t = a[j]; k = j;
    }
    a[k] = a[i]; a[i] = t;
  }
 }
 for(i = 0; i < n; i++)
   printf("%d  %10.2f \n",i,a[i]);
}
```

Ordenación rápida (Quicksort)

- La ordenación rápida se basa en el hecho que los intercambios deben ser realizados preferentemente sobre distancias grandes.
- El algoritmo a seguir es el mismo que se aplica cuando se quiere ordenar un gran montón de exámenes: Seleccionar un valor de división (L por ejemplo) y dividir el montón en dos pilas, A-L y M-Z. Después se toma la primera pila y se subdivide en dos, A-F y G-L por ejemplo. A su vez la pila A-F puede subdividirse en A-C y D-F. Este proceso continúa hasta que las pilas sean suficientemente pequeñas para ordenarlas fácilmente. El mismo proceso se aplica a la otra pila.

Ordenación rápida (Quicksort)

- En la ordenación rápida se toma un elemento x del array (el del medio por ejemplo),
- Se busca en el array desde la izquierda hasta que $a_1, a_2, \ldots, a_{i-1}$ > x, lo mismo se hace desde la derecha hasta encontrar $a_1, a_2, \ldots, a_{i-1}$ < x.
- Después se intercambia esos elementos y se continúa ese proceso hasta que los índices se cruzan.
- Se aplica el mismo proceso para la porción izquierda del array entre el extremo izquierdo y el índice derecho y para la porción derecha entre el extremo derecho y el último índice izquierdo (algoritmo recursivo).

Ordenación rápida (Quicksort)

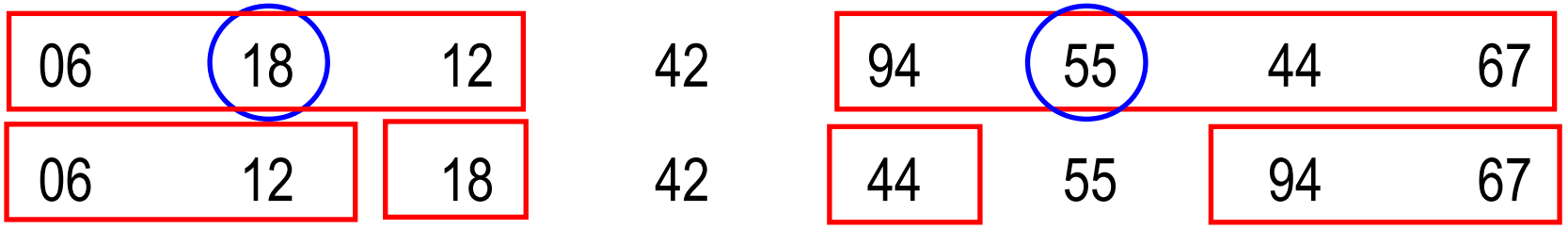
- La complejidad del algoritmo es O(n log n).
- Una comparación visual de los tiempos de ejecución de los algoritmos de ordenación se encuentra en:

http://personales.unican.es/corcuerp/ProgComp/Ordena/AlgoritmosOrdenamiento.html

Ordenación rápida

44	55	12	42	94	18	06	67
44	55	12	42	94	18	06	67
06	55	12	42	94	18	44	67
06	18	12	42	94	55	44	67
06	18	12	42	94	55	44	67

Ordenación rápida



Ordenación rápida

06	12	18	42	44	55	94	67
06	12	18	42	44	55	67	94

Ordenación rápida

```
/*****************************************************************\
* Programa: quicksort.c                                           *
* Descripción: Programa que ordena un array mediante el metodo rapido    *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>

void qsort(int izq, int der, float a[]);

main()
{
 float notas[]={10.0,8.0,5.5,3.4,3.2,2.5,2.2,1.5};
 int i,n;


 n = sizeof(notas)/sizeof (float);

 qsort(0,n-1,notas);

 for(i = 0; i < n; i++)
  printf("%d  %10.2f \n",i,notas[i]);
 return 0;
}
```

Ordenación rápida

```
void qsort(int izq, int der, float a[])
{
 int i,j;
 float tmp,x;

 i = izq;
 j = der;
 x = a[(izq + der)/2];
 do
 {
  while (a[i] < x)  i++;
  while (x < a[j])  j--;
  if (i <= j)
  {
   tmp = a[i];
   a[i] = a[j];
   a[j] = tmp;
   i++;
   j--;
  }
 }
 while (i <= j);
 if (izq < j)
   qsort(izq, j, a);
 if (i < der)
   qsort(i, der, a);
}
```

Algoritmos de búsqueda

- Se considera que la búsqueda se realiza sobre los elementos almacenados en memoria principal.

Búsqueda secuencial

- Se examina cada elemento del array hasta que se encuentra el elemento o se llega al final. Se aplica en arrays y listas enlazadas. La lista no necesita estar ordenada. La complejidad del algoritmo es O(n).

Búsqueda por mayor probabilidad

- Algunas veces determinados elementos de la lista tienen mayor demanda que otros. De ahí que sea conveniente colocar los elementos más frecuentes (deseados) al comienzo de la lista.

Algoritmos de búsqueda

Ordenación de claves

- Cuando la lista está ordenada según el valor clave puede aumentarse la eficiencia de la búsqueda. Así, sólo es necesario efectuar la búsqueda hasta que el elemento buscado sobrepase su posición lógica.

Búsqueda binaria

- Se aplica a listas ordenadas. La complejidad de la búsqueda es O(log n). Aunque el algoritmo se establece en forma recursiva, la solución iterativa es más eficiente.

Búsqueda binaria

```
/*********************************************************************\
* Programa: busq_binariai.c                                           *
* Descripción: Programa que busca en un array, usa b.binaria iterativa*
* Autor: Pedro Corcuera                                               *
* Revisión: 1.0 2/02/2008                                             *
\*********************************************************************/
#include <stdio.h>

int busqueda_binaria(int a[],int clave, int n_elem);

main()
{
 int t,clave,notas[]={10,8,5,3,7,2,4,1};
 int i,j,n,tmp;

 n = sizeof(notas)/sizeof (int);

 for(i = n; i > 0; i--)   /* se emplea el metodo de burbuja para ordenar */
  for(j = 0; j < i-1; j++)
   if (notas[j] > notas[j+1])
   {
    t = notas[j];
    notas[j] = notas[j+1];
    notas[j+1] = t;
   }
```

Búsqueda binaria

```
 printf("Ingresa elemento a buscar: "); scanf("%d",&clave);
 if ((tmp=busqueda_binaria(notas, clave, n-1)) >= 0)
   printf("Elemento hallado en la posicion %d\n",tmp);
 else
   printf("Elemento no hallado en array\n");
 return 0;
}

int busqueda_binaria(int a[],int clave, int ultimo)
{
 int hallado = 0, medio, primero = 0;

 while (primero <= ultimo && !hallado)
 {
  medio = (primero + ultimo)/2;
  if (a[medio] == clave)
   hallado = 1;
  else
   if (a[medio] > clave)
    ultimo = medio-1;
   else
    primero = medio+1;
  }
  if (hallado)
    return medio;
  else
    return -1;
}
```

Algoritmos de vuelta atrás (backtracking)

- Este tipo de algoritmos trata de encontrar soluciones de problemas específicos sin seguir una regla fija de computación, sino mediante *prueba y error*.
- El patrón común es descomponer el proceso de prueba y error en tareas parciales. Normalmente esas tareas se expresan en términos recursivos y consisten en la exploración de un número finito de sub-tareas.
- El proceso global como un proceso de búsqueda que construye progresivamente y poda un árbol de subtareas.
- Para reducir el tiempo de computación se usan heurísticas para la poda del árbol.

Algoritmos de vuelta atrás (backtracking)

- La técnica consiste en recorrer sistemáticamente todos los posibles caminos. Cuando un camino no conduce a la solución, se retrocede al paso anterior para buscar otro camino (backtracking).
- Si existe solución es seguro que se encuentra, el problema es el tiempo de proceso.
- Es deseable que sólo recorra los caminos que conduzcan a la solución y no todos. Es posible hacerlo cuando estando en un punto nos damos cuenta que si seguimos avanzando no encontraremos ninguna solución.
- Es un técnica netamente recursiva.

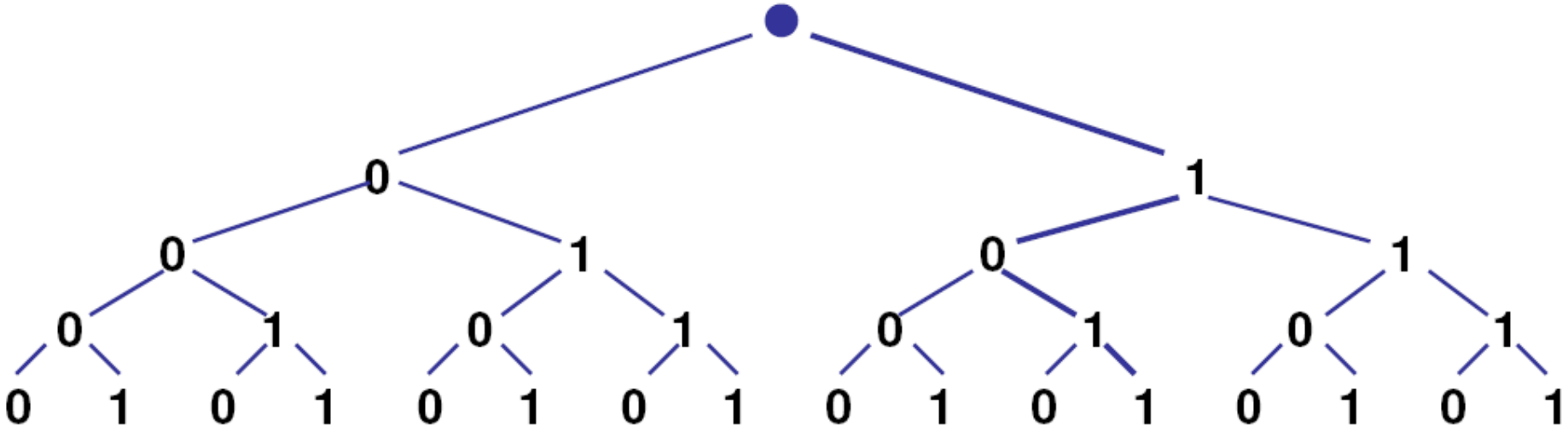
Backtracking – Selección de objetos

- Se tienen n objetos, cada uno puede o no ser seleccionado, ¿Cuáles son todas las formas posibles de tomarlos? ¿De cuántas maneras los podemos escoger?

Estrategia de solución

- Primero pensemos como será nuestro árbol:
  - En cada nivel podemos decidir si elegir o no el artículo.
  - La cantidad de decisiones que se deben tomar depende de la cantidad de artículos.
  - Dicha cantidad de decisiones es la profundidad del árbol.

Backtracking – Selección de objetos



- Arbol para cuatro objetos. Por ejemplo el camino 1011 significa que se toman el primer, tercer y cuarto artículos y no se toma el segundo.
- El número de posibles selecciones es 2^n

Selección de objetos

```
/*****************************************************************\
* Programa: selecc_obj.c                                          *
* Descripción: Programa para seleccionar un conjunto de n elementos *
*              (0=no  1=si)                                       *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>
#define MAX 20
#define FALSE 0
#define TRUE  1

void escoger(int, int, int [ ]);
void imprimir(int, int[ ]);

void main(void)
{
 int i, n;
 int decision[MAX];
 printf("Nro de elementos: ");
 scanf("%d",&n);

// marcamos objetos como no escogidos
 for(i=0; i<n; i++)
   decision[i]=FALSE;
 escoger(0, n, decision);
}
```

Selección de objetos

```
void imprimir(int n, int decision[ ])
{
  int i;
  for(i=0; i<n; i++)
    printf("%d ", decision[i]);
  printf("\n");
}

void escoger (int k, int n, int decision[ ])
{
  if (k<n)
  {
   // no se toma el objeto
   decision[k]= FALSE;
   escoger(k+1, n, decision);
   // se toma el objeto
   decision[k]= TRUE;
   escoger(k+1, n, decision);
   decision[k]= FALSE;
  }
  else
    imprimir(n, decision);
}
```

Backtracking – Esquema general

- Como hay que recorrer un árbol y en cada paso se necesita encontrar todas las alternativas posibles y luego recorrerlas una a una, podemos pensar en el esquema siguiente:
  - Si se llega a la máxima profundidad del árbol, entonces mostrar solución (caso base).
  - Sino entonces:
    - Construir cada candidato
    - Para cada candidato añadirlo a la solución y llamar recursivamente a la función con el nivel siguiente de profundidad.

Backtracking – Esquema general

- Lo que cambia en cada problema es:
  - La forma de generar cada candidato.
  - Los tipos de datos.
  - El número de parámetros de la funciones.
  - Añadirle condiciones adicionales propias del problema.
  - Añadirle condiciones adicionales para reducir el árbol (podar).

- Reescribiendo la solución del problema de selección de objetos a este esquema general:

Selección de objetos – esquema general

```
/*****************************************************************\
* Programa: selecc_obj_eg.c                                       *
* Descripción: Programa para seleccionar un conjunto de n elementos      *
*              (0=no  1=si) segun esquema general                 *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
# include <stdio.h>
# define MAX 20
#define FALSE 0
#define TRUE  1

void escoger(int, int, int [ ]);
void imprimir(int, int[ ]);
int construyeCandidatos(int [ ]);

void main(void)
{
 int i, n;
 int decision[MAX];
 printf("Nro de elementos: ");
 scanf("%d",&n);

// marcamos objetos como no escogidos
 for(i=0; i<n; i++)
   decision[i]=FALSE;
 escoger(0, n, decision);
}
```

Selección de objetos – esquema general

```
void imprimir(int n, int decision[ ])
{
  int i;
  for(i=0; i<n; i++)
    printf("%d ", decision[i]);
  printf("\n");
}
void escoger (int k, int n, int decision[ ])
{
  int cand[MAX];

  if (k == n)
     imprimir(n, decision);
  else
  {
    int i, ncand = construyeCandidatos(cand);
    for(i=0; i < ncand; i++)
    {
      decision[k] = cand[i];
      escoger(k+1, n, decision);
    }
  }
}
int construyeCandidatos(int decision[])
{
  decision[0]=TRUE; decision[1]=FALSE;
  return 2;
}
```

Variaciones

- Las variaciones o permutaciones, son las distintas formas en que pueden agruparse r elementos de un conjunto n donde r < n. Deben diferir en algún elemento o en el orden.
- El número de variaciones en esas condiciones es: $V_r^n = \frac{n!}{(n-r)!}$

- Ejemplo: si n = 4 (abcd) y r = 3 entonces el número de variaciones es 24

abc	bac	cab	dab
abd	bad	cad	dac
acb	bca	cba	dba
acd	bcd	cbd	dbc
adb	bda	cda	dca
adc	bdc	cdb	dcb

Variaciones – código

```
/*****************************************************************\
* Programa: variaciones.c                                         *
* Descripción: Programa que imprime las variaciones de un conjunto de n  *
*              caracteres en grupos de r elementos                *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>
#include <string.h>
#define MAX 30
#define FALSE 0
#define TRUE  1

int construyeCandidatos(int k, char cand[ ]);
void imp( );
void bt(int k);
char x[MAX], rpta[MAX];
int n, r;
int permut=0;  /* numero de permutaciones */

void imp( )
{
  int i;
  for(i=0; i<r; i++)
     printf("%c ",rpta[i]);
  printf("\n");
  permut++;
}
```

Variaciones – código

```
int construyeCandidatos(int k, char cand[ ])
{ int i, j, ncand=0;
  int encontrado;
  for(i = 0; i < n; i++)
  { encontrado=FALSE;
    for(j = 0; j < k; j++)
      if(x[i] == rpta[j])
        encontrado=TRUE;
    if(!encontrado)
    {
     cand[ncand]=x[i];
     ncand++;
    }
  }
  return ncand;
}
```

Variaciones – código

```
void bt(int k)
{
  char cand[MAX];
  if(k==r) imp();
  else
  {
   int i, ncand=construyeCandidatos(k, cand);
   for(i = 0; i < ncand; i++)
   {
     rpta[k] = cand[i];
     bt(k+1);
   }
  }
}
void main(void)
{
  printf("Ingrese Elementos (cadena de caracteres, Max. 30): ");
  scanf(" %s",x);
  /*printf("n= "); scanf(" %d",&n);*/
  n = strlen(x);
  printf("Grupos de r = "); scanf(" %d",&r);
  printf("\nLas variaciones son: \n");
  bt(0);
  printf("\nEl numero de variaciones son: %d\n",permut);
}
```

Números que suman un valor objetivo

- Dado un conjunto de números enteros {2, 3, 6, 1, 5}, encontrar los subconjuntos cuya suma sea exactamente 9. Por ejemplo: {3, 1, 5} suman 9.

Suma objetivo – código

```
/*****************************************************************\
* Programa: suma_valorobj.c                                       *
* Descripción: Programa que resuelve de encontrar los subconjuntos de    *
*              {2, 3, 6, 1, 5} que suman 9  (pe {3, 1, 5})        *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>
#include <string.h>
# define OBJ 9

int v[ ] = {2, 3, 6, 1, 5};
int n=5;
int s[ ] = {0, 0, 0, 0, 0};

void toma(int);
int  suma(int);
void imprimeSolucion(void);

void main(void)
{
   toma(0);
}
```

Suma objetivo – código

```
void toma(int i)
{
  if (i == n) return; // condición base
  s[i]=1; // Tomo el objeto i
  if (suma(i) == OBJ) imprimeSolucion( );
  if( suma(i) < OBJ ) toma(i+1);
  s[i]=0; // No tomo el objeto i
  if (suma(i) == OBJ) imprimeSolucion( );
  if( suma(i) < OBJ ) toma(i+1);
}

int suma(int i)
{ int j, sum=0;
  // el ultimo agregado debe contribuir a la suma
  if (s[i] != 1 ) return 0;
  for(j=0; j<n; j++)
     if( s[j] == 1)
        sum += v[j];
  return sum;
}

void imprimeSolucion(void)
{ int j;
  for(j=0; j<n; j++)
     if( s[j] == 1)
        printf("%d  ",v[j]);
  printf("\n");
}
```

El Problema de las N Reinas

- En un tablero de ajedrez (NxN) colocar N reinas sin que se produzcan conflictos por el movimiento propio de las reinas.
- Ejm: N = 8

El Problema de las N Reinas

- http://personales.unican.es/corcuerp/ProgComp/Queens/Reinas.HTM

Problema de las N reinas – código

```
/*************************************************************************\
* Programa: reinas_vg.c                                                   *
* Descripción: Programa que resulve el problema de las N reinas           *
* Autor: Pedro Corcuera                                                   *
* Revisión: 1.0 2/02/2008                                                 *
\*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#define MAXCANDIDATOS 100 // max cantidad de candidatos
#define NMAX 20 // máximo tamaño del tablero
#define TRUE 1
#define FALSE 0
int nroSol; // cuenta soluciones

void imprimeSolucion(int a[ ], int n)
{ int i, j;

  nroSol++;
  printf("Solucion: %d\n", nroSol);
  for(i=0; i<n; i++)
  {
    for(j=0; j<n; j++)
      if( i==a[j] ) printf("R");
      else printf("-");
    printf("\n");
  }
  printf("==================\n");
}
```

Problema de las N reinas – código

```
int construyeCandidatos(int a[], int k, int n, int c[])
{
  int i, j, ncandidatos = 0;
  for (i=0; i<n; i++)
  {
    int band = TRUE;
    for (j=0; j<k; j++)
    {
 /*       restriccion de diagonal  restriccion de fila */
      if ( abs(k-j) == abs(i-a[j]) || i == a[j] )
         band = FALSE;
    }
    if (band)
    {
      c[ncandidatos] = i;
      ncandidatos++;
    }
  }
  return ncandidatos;
}
```

Problema de las N reinas – código

```
void backtrack(int a[ ], int k, int n)
{ int i;
  int c[MAXCANDIDATOS];

  if (k == n)
     imprimeSolucion(a, k);
  else {
     int ncandidates = construyeCandidatos(a, k, n, c);
     for (i = 0; i < ncandidates; i++)
     {
       a[k] = c[i];
       backtrack(a, k+1, n);
     }
  }
}

void main(void)
{
  int a[NMAX], n;
  printf("Tamano del tablero: "); scanf(" %d",&n);
  nroSol = 0;
  backtrack(a, 0, n);
  printf("Nro Soluciones %d \n", nroSol);
}
```

Problema de la mochila

- Supóngase que se tiene n objetos distintos y una mochila.
- Cada objeto tiene asociado un peso w_i y un valor v_i
- La mochila puede llevar un peso que no sobrepase su capacidad de carga *W*.

Objetivo

- Llenar la mochila de tal forma que se maximice el valor de los objetos transportados en ella.
- Hay que tener en cuenta que la suma de los pesos de los objetos seleccionados no debe superar la capacidad máxima *W* de la mochila.

Problema de la mochila – Planteamiento matemático

- Maximizar: $\sum_{i=1}^{n} x_i v_i$
- Sujeto a la restricción: $\sum_{i=1}^{n} x_i w_i \leq M$
- Tipos de Mochila:
  - La mochila fraccionaria (siempre funciona la estrategia voraz o greedy).
  - La mochila entera (0/1) (no siempre funciona la estrategia voraz).
- Se trata de resolver el problema de la mochila binaria 0/1 mediante búsqueda exhaustiva.

Problema de la mochila – Estrategia de solución

- La búsqueda estará basada en el siguiente árbol:
  - En cada nivel se puede decidir si elegir o no el artículo.
  - La cantidad de decisiones que se deben tomar depende de la cantidad de artículos y de no sobrepasar la capacidad de carga de la mochila.
  - Se debe llenar la mochila con aquellos objetos que maximicen el valor que lleva.
- Ejemplo:

N =10 W = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120}

Peso 10 11 12 13 14 15 16 17 18 19

Valor 18 20 17 19 25 21 27 23 25 24

Problema de la mochila – código

```
/*****************************************************************\
* Programa: mochila.c                                           *
* Descripción: Programa que resuelve el problema de la mochila binaria   *
* Autor: Pedro Corcuera                                         *
* Revisión: 1.0 2/02/2008                                       *
\*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#define MAX 30
enum {FALSE, TRUE};

void bt(int, int, int);
int generaCandidatos(int, int, int []);
int n;
int Optimo=-1;
int peso[MAX];
int valor[MAX];
int decision[MAX];
int objetos[MAX] = {0}, nobj=1;
```

Problema de la mochila – código

```
void main(void)
{ int i, capacidad;
  printf("Capacidad de la mochila: ");
  scanf(" %d",&capacidad);
  printf("Cantidad de objetos: ");
  scanf(" %d",&n);
  printf("Ingrese pesos: ");
  for(i = 0; i < n; i++) scanf(" %d",&peso[i]);
  printf("Ingrese valores: ");
  for(i = 0; i < n; i++) scanf(" %d",&valor[i]);
  bt( 0, capacidad, 0);

  printf("\n\nCapacidad de la mochila: %d\n",capacidad);
  printf("Cantidad de elementos: %d\n",n);
  printf("objeto  Pesos  Valores \n");
  for(i = 0; i < n; i++)
     printf("  %d\t%d\t%d\n",i+1,peso[i],valor[i]);

  printf("\nValor Optimo: %d \n",Optimo);
  printf("Objetos: \n");
  for(i = 0; i < n; i++) printf(" %d \t-> %d \n",i+1,objetos[i]);
}
```

Problema de la mochila – código

```
void bt(int k, int capNoUsada, int valorActual)
{
  int i, candidato[MAX], nroC;
  if(k == n) return;
  nroC = generaCandidatos( k, capNoUsada, candidato );
  for(i = 0; i < nroC; i++)
  { decision[k] = candidato[i]; // toma cada candidato
    if (decision[k])
    {
      valorActual += valor[k];
      if ( valorActual > Optimo )
      {
       objetos[k] = 1; Optimo = valorActual;
      }
      bt(k+1, capNoUsada - peso[k], valorActual);
    }
    else
      bt(k+1, capNoUsada, valorActual);
  }
}

int generaCandidatos( int k, int capNoUsada,int candidato[ ] )
{
  candidato[0]=FALSE;
  if ( peso[k] <= capNoUsada )
  { candidato[1]=TRUE; return 2; }
  else return 1;
}
```

Sudoku

- El Sudoku se basa en el trabajo sobre *Cuadrados Latinos* de Leonard Euler.
- Un *cuadrado latino* es una matriz de $n_x n$ elementos en la que cada celda contiene uno de los n símbolos de tal modo que cada uno de ellos aparece exactamente una vez en cada columna y en cada fila. Se llaman así porque Euler utilizó caracteres latinos para representarlo.

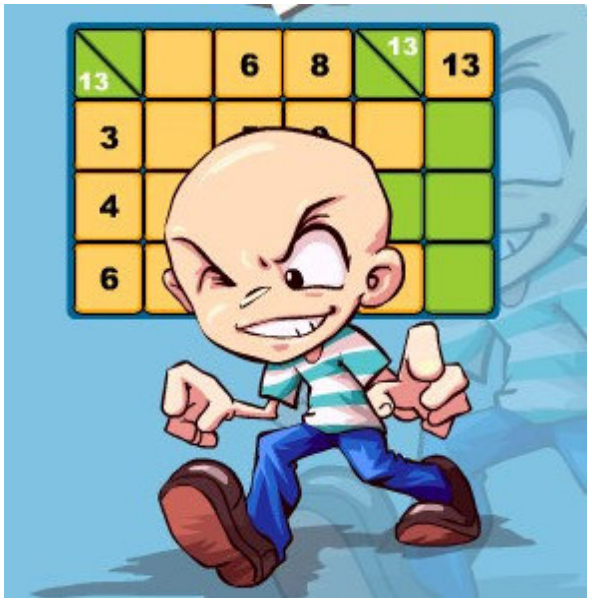
1	2	3
2	3	1
3	1	2

a	b	d	c
b	c	a	d
c	d	d	a
d	a	c	b

Sudoku

- Sudoku: **su** = número, **doku** = solo
- Es un tablero de 9x9 compuesto por regiones cuadradas de 3x3. Algunas celdas ya tienen números, el objetivo es rellenar las celdas vacías, con un número de tal forma que cada columna, fila y región contengan los números del 1 al 9 sin que se repitan.
- El SUDOKU añade una restricción adicional a los cuadrados latinos los subgrupos de 3x3 deben tener los dígitos del 1 al 9.
- Un Sudoku bien planteado debe tener una única solución.

Sudoku



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

Sudoku – código

```
/*****************************************************************\
* Programa: sudoku1.c                                             *
* Descripción: Programa que resuelve un sudoku que se lee desde fichero  *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>

#define TRUE  1
#define FALSE 0

int sudoku[9][9];
void leeSudoku(void);
void resSudoku(int, int);
int buscaCeldaVacia(int *, int *);
int puedoColocar(int, int, int);
void impSudoku(void);

void main(void)
{
    leeSudoku();
    resSudoku(0,0);
}
```

Sudoku – código

```
void resSudoku(int x, int y)
{
 int nro, hayVacia;

 hayVacia=buscaCeldaVacia(&x, &y);
 if ( !hayVacia ) impSudoku();
 else
 {
  for(nro=1; nro<=9; nro++)
   if ( puedoColocar(x, y, nro) )
   {
    sudoku[x][y]=nro; // lo coloco
    resSudoku(x,y+1); // tomo el sgte
    sudoku[x][y]=0; // lo quito
   }
 }
}

int buscaCeldaVacia(int *x, int *y)
{
 for( ; *x<9; (*x)++)
 {
  for( ; *y<9; (*y)++)
    if( sudoku[*x][*y]==0 ) return TRUE;
  *y=0;
 }
 return FALSE;
}
```

Sudoku – código

```
int puedoColocar(int f, int c, int nro)
{
 int i, j, iniFil, finFil, iniCol, finCol;
 // comprueba fila
 for(j=0; j<9; j++)
 if(sudoku[f][j]==nro)
   return FALSE;
 // comprueba columna
 for(i=0; i<9; i++)
 if(sudoku[i][c]==nro)
   return FALSE;
// comprueba subcuadrado
 iniCol = (c/3) *3;
 finCol = iniCol+3;
 iniFil = (f/3) *3;
 finFil = iniFil+3;
 for(i=iniFil; i<finFil; i++)
  for(j=iniCol; j<finCol; j++)
   if(sudoku[i][j]==nro)
     return FALSE;
   return TRUE;
}
```

Sudoku – código

```
void impSudoku(void)
{
 int i, j;
 printf("SOLUCION:\n");
 printf("========\n\n");
 for(i=0; i<9; i++)
 {
   for(j=0; j<9; j++)
   {
     printf("%d",sudoku[i][j]);
     if ( j==2 || j==5 ) printf(" | ");
   }
  printf("\n");
  if ( i==2 || i==5 )
    printf("--- | --- | ---\n");
 }
 printf("\n");
}
void leeSudoku(void)
{
 int i, j;
 printf("Introduce SUDOKU (valor seguido de espacio para cada fila,\n");
 printf("                 (0 para indicar celda vacia):\n"); printf("\n");
 for(i=0; i<9; i++)
 { printf("Fila %d : ",i+1);
   for(j=0; j<9; j++)  scanf("%d",&sudoku[i][j]);
 }
}
```

Laberinto - Plantemiento

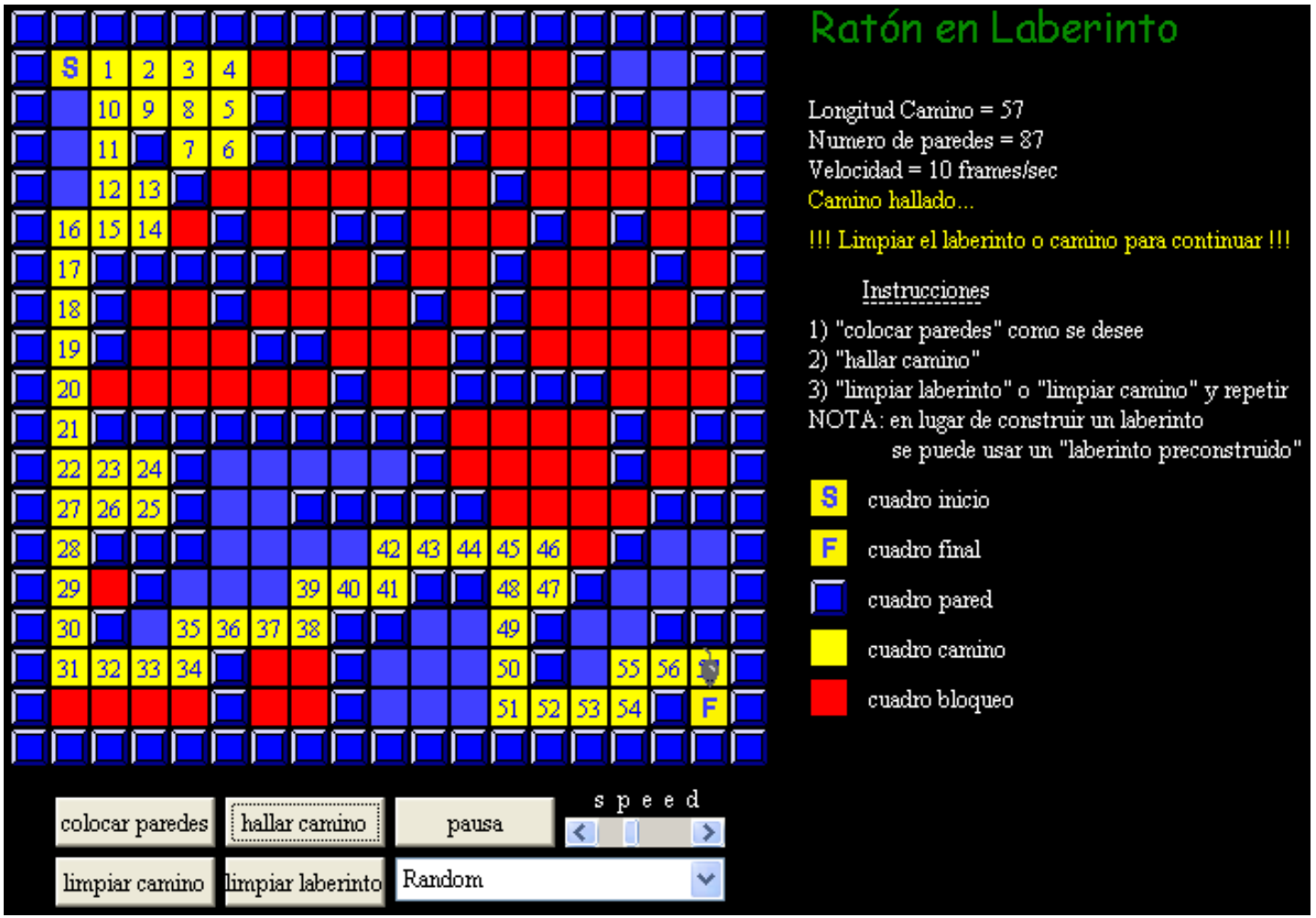
- El problema consiste en hallar una ruta de salida de un laberinto.
- El laberinto se representa mediante un array (dimensión máxima de 30x30), cuyo contenido serán constantes enteras (1 celda ocupada, 0 si no lo está).
- Los datos del laberinto se leen desde un fichero de tipo texto (laberinto.dat) que contiene los siguientes datos:
  - Número de filas y columnas del laberinto.
  - Fila y columna de la posición inicial.
  - Fila y columna de la posición final.
  - Por cada posición ocupada la fila y columna.

  Nota: La numeración de las filas y coumnas es a partir de 1 (para el usuario).
- Los movimientos permitidos son avanzar o retroceder una celda en la fila o columna, es decir cuatro movimientos como máximo. No se consideran movimientos en diagonal.

Laberinto - Algoritmo

- El algoritmo a seguir para resolver el programa es el siguiente:
  - **Asignar** como posiciones ocupadas: la primera y posterior a la última fila, así como la primera y posterior a la última columna (para facilitar la programación).
  - **Inicializar** el contador de movimientos, el contador de posiciones del camino y el camino (array).
  - **Mientras** no sea la posición final realizar:
    - **Mientras** no se supere el límite de movimientos (4) comprobar, siguiendo un orden (antihorario), si la casilla siguiente está desocupada. Si es así salir del bucle.
    - **Si** el contador de movimientos está en el rango correcto, **incrementar** el contador de posiciones del camino, **almacenar** la fila y columna respectiva en el array del camino y **guardar** el contador de movimiento (en un array), **marcar** como ocupada la fila y columna respectiva y **reinicializar** el contador de movimientos. **Sino**, **si** el contador de posiciones del camino es cero, devolver un error (no hay camino), **caso contrario**, volver a la posición anterior **disminuyendo** en uno el contador de posiciones del camino, **recuperando** la fila y columna respectiva a esa posición y **recuperando** el número del movimiento (para continuar probando las demás posiciones).
  - **Almacenar** la última posición del camino.

Laberinto



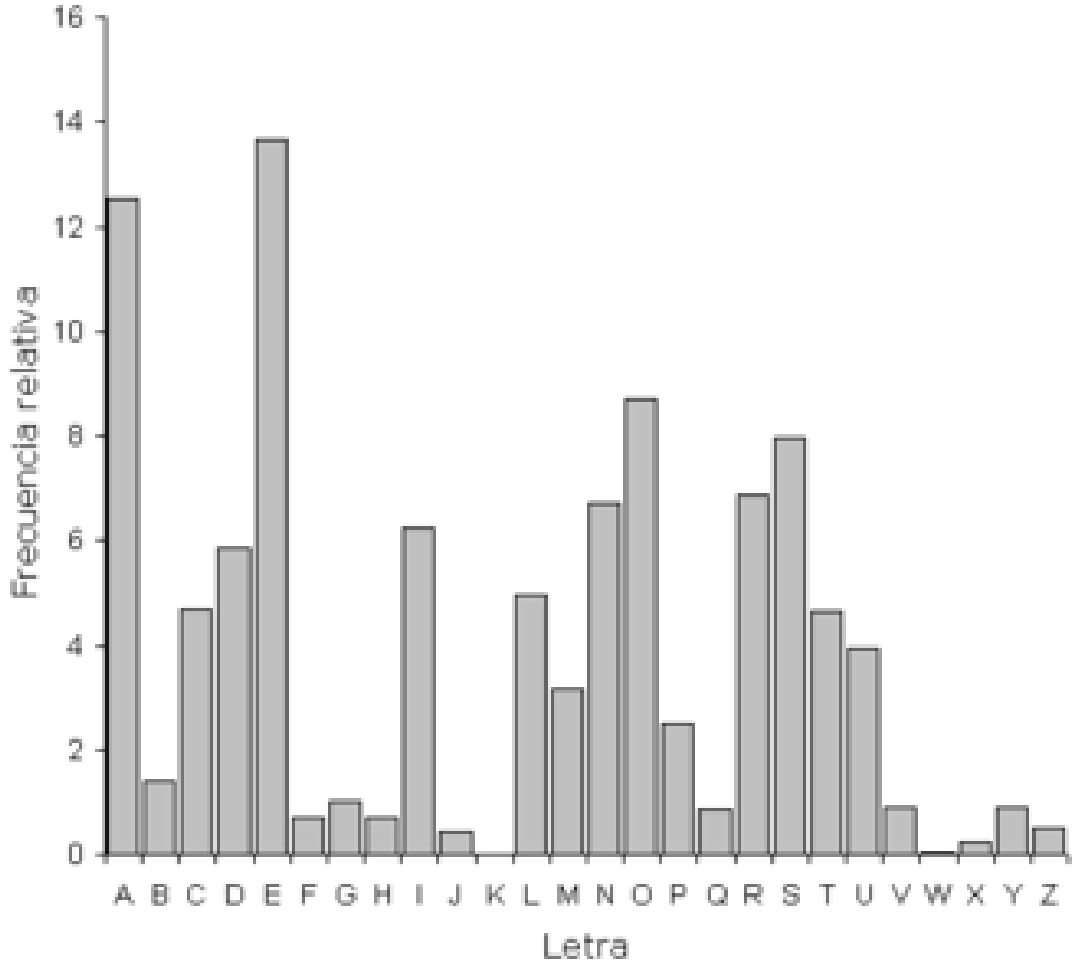
- http://personales.unican.es/corcuerp/ProgComp/laberinto/laberinto.html

Algoritmo de Huffman – Introducción

- Usado para comprimir símbolos (compresores, mp3).
- Enfoque
  - Longitud variable para codificar los símbolos.
  - Explota la frecuencia estadística de símbolos.
  - Eficiente cuando la probabilidad de símbolos varía ampliamente.
- Principio
  - Usa menos bits para representar los símbolos más frecuentes.
  - Usa más bits para representar los símbolos menos frecuentes.
- Genera código óptimo, es decir, no hay un código alternativo que supere la capacidad de compresión de Huffman.

Algoritmo de Huffman – Compresión de texto

- Letras no tienen la misma frecuencia en un texto.



Algoritmo de Huffman – Compresión de texto

- El código ASCII usa 8 bits para cada letra.
- Si usara sólo 1 bit para la letra E estaría economizando un 88% del espacio requerido para almacenarlas.
- Eso es un 12% de compresión en el archivo de prueba con sólo comprimir una letra.
- Usar códigos cortos para las letras más frecuentes, largos para las más raras.

Algoritmo de Huffman

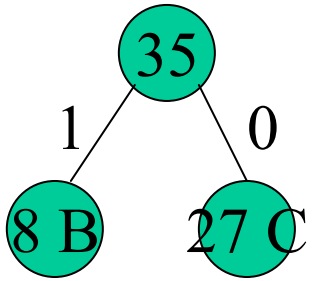
- Algoritmo:
  1. Ordenar los simbolos por frecuencia o probabilidades decrecientes.
  2. Construir un nodo por agrupación de los dos últimos símbolos (los menos probables).
  3. Repetir 1 y 2 hasta llegar a un nodo con dos nodos hijos.
  4. Asignar los códigos 0 y 1 a los dos símbolos del nodo final.
  5. Ampliar el código, iterando este paso hasta alcanzar los nodos de los símbolos.

Algoritmo de Huffman - Ejemplo

- Considere un texto con las letras y frecuencias siguientes:

A	61
O	48
N	40
C	27
B	8

- Tomar los dos valores más bajos un nodo con la suma de las frecuencias:

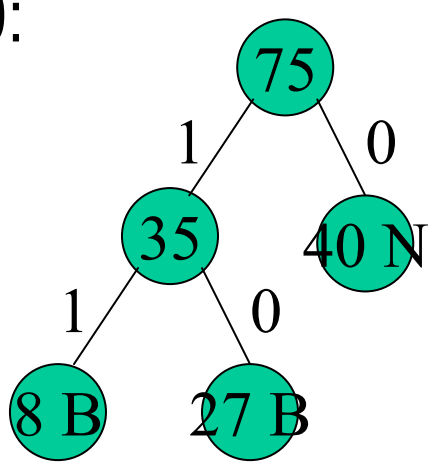


Algoritmo de Huffman - Ejemplo

- Sustituir en la lista el 8 y el 27 por 8+27=35 ordenarla:

A	61
O	48
N	40
BC	35

- Se repite el procedimiento con 35 y 40:

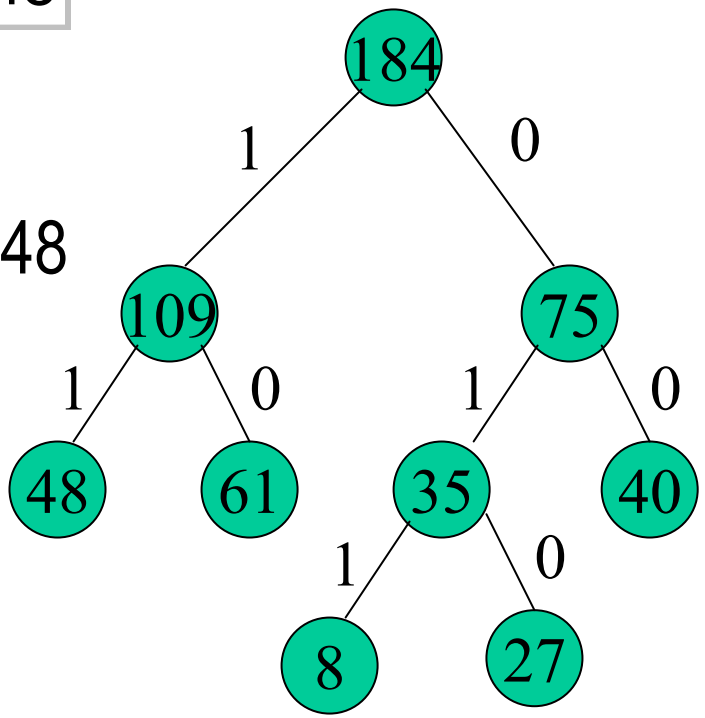


Algoritmo de Huffman - Ejemplo

- Sustituir en la lista el 40 y el 35 por 40+35=75 ordenarla:

BCN	75
A	61
O	48

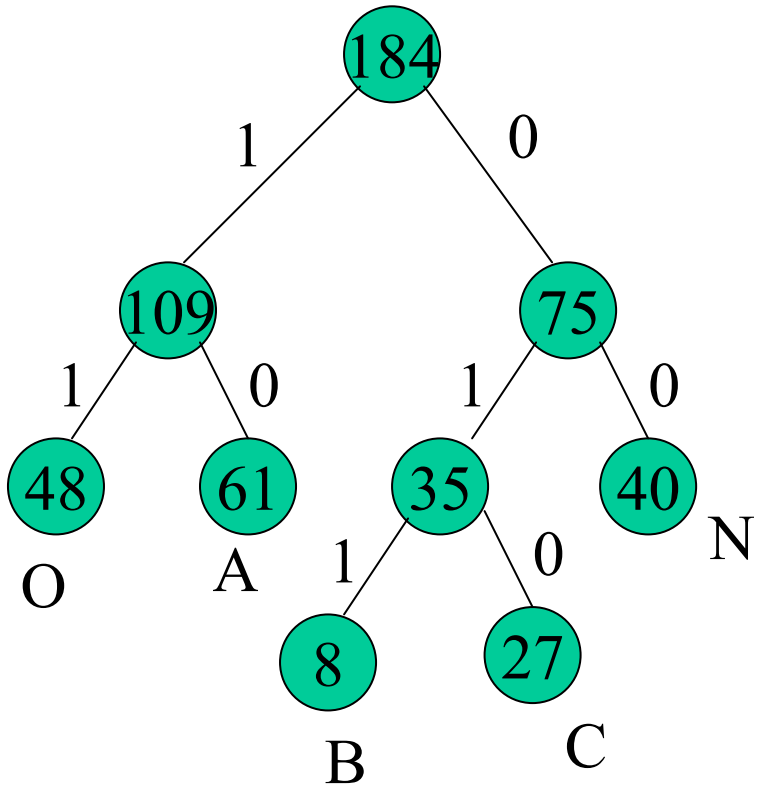
- Se repite el procedimiento con 61 y 48 y después el 75.



Algoritmo de Huffman – Código resultante

- Recorrer desde la raiz hasta la hoja que corresponda a la letra:

A	10
O	11
N	00
C	010
B	011



Algoritmo de Huffman – Código resultante

- Si se usara código ASCII se requiere 184x8 = 1472 bits
- Si se usa Huffman se requiere $\sum_{i=1}^{n} f_i lh_i$ = 403 bits
- Factor de compresión (1 - 403/1472) x 100 = 73 %
- Longitud equivalente código Huffman: $\frac{\sum_{i=1}^{n} f_i lh_i}{\sum_{i=1}^{n} f_i} = 2.19$

	f_i	ch_i	lh_i
A	61	10	2
O	48	11	2
N	40	00	2
C	27	010	3
B	8	011	3

Algoritmo de Huffman - Ejemplo de Codificación

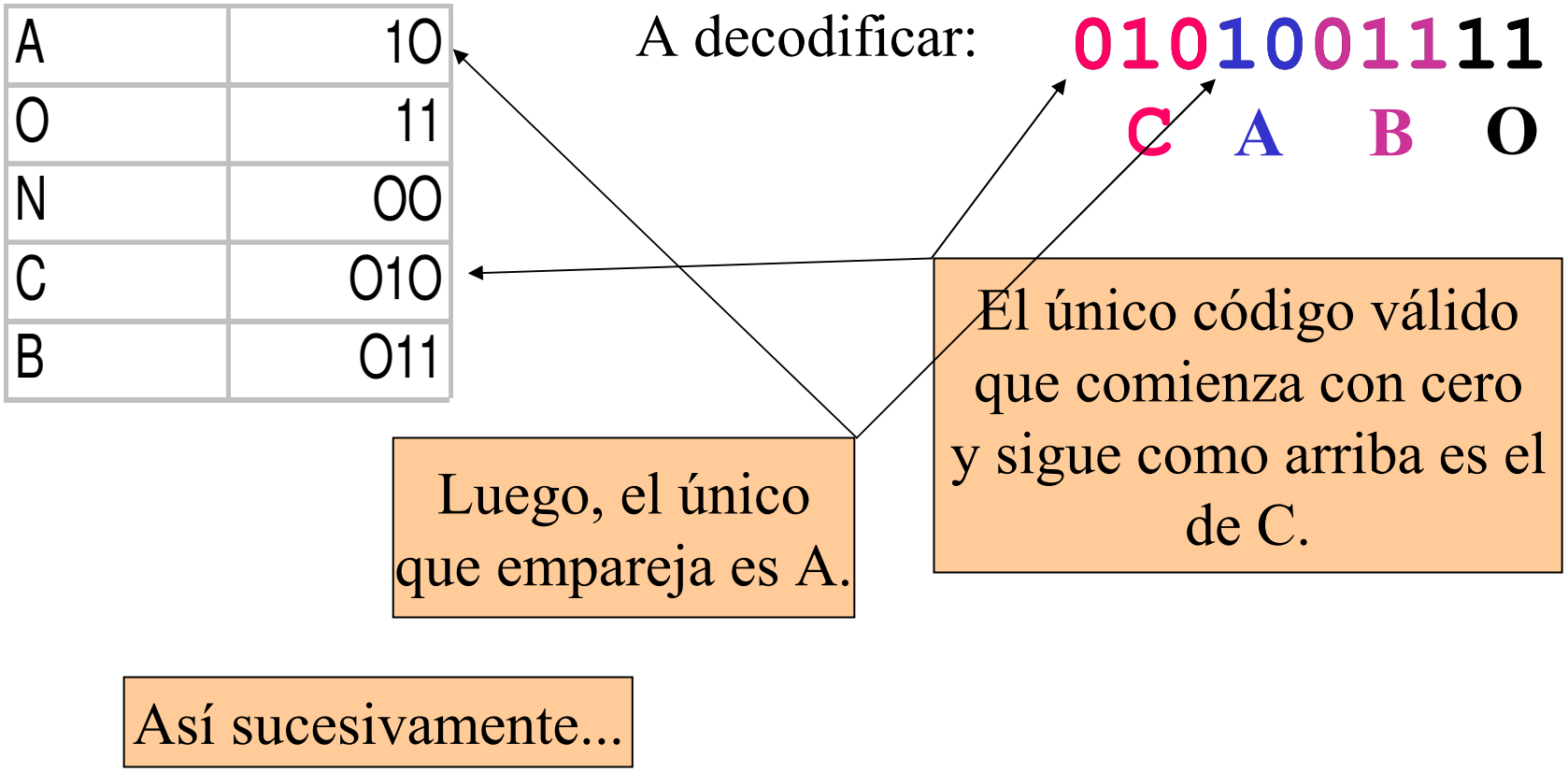
A	10
O	11
N	00
C	010
B	011

Cadena a codificar: **B A N C O**

Huffman: **011100001011**

ASCII: 01000010010000010101111001000011010111110

Algoritmo de Huffman - Ejemplo de Decodificación



Algoritmo de Huffman – Desventajas

- El código generado depende de cada texto.
- Hay que almacenar una tabla de codificación en el fichero comprimido.
- Es necesario hacer dos pasadas sobre el fichero original. Uno para calcular las frecuencias de ocurrencia de los símbolos y la otra para generar el código.

Estructuras de datos en C

Índice

- Estructuras autoreferenciadas
- Asignación dinámica de memoria.
- Listas enlazadas.
- Pilas.
- Colas.
- Arboles.
- Arboles binarios.
- Recorrido de árboles binarios.
- Arbol binario de búsqueda.

Estructuras autoreferenciadas - Repaso

- 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 *psig; /* puntero de enlace*/
};
```

Asignación dinámica de memoria - Repaso

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

```
newPtr = malloc(sizeof(nodo));
```

Listas enlazadas

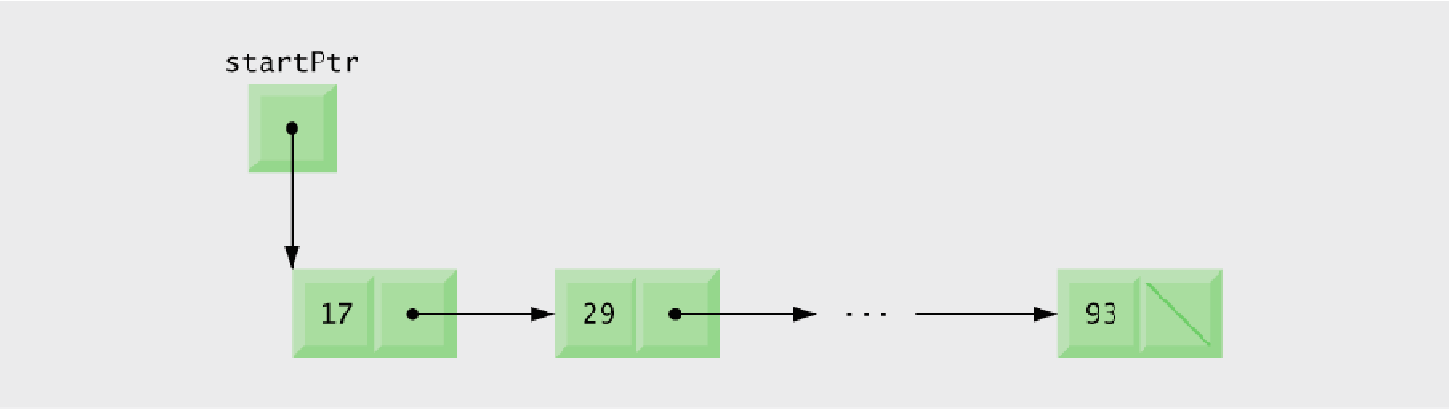
- Colección lineal de nodos autoreferenciados.
- Conectados por punteros enlazados.
- Acceso mediante un puntero al nodo inicial de la lista.
- Los nodos siguientes se acceden mediante el campo de puntero de enlace del nodo en curso.
- El puntero de enlace del último nodo se asigna a NULL para indicar 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

- 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

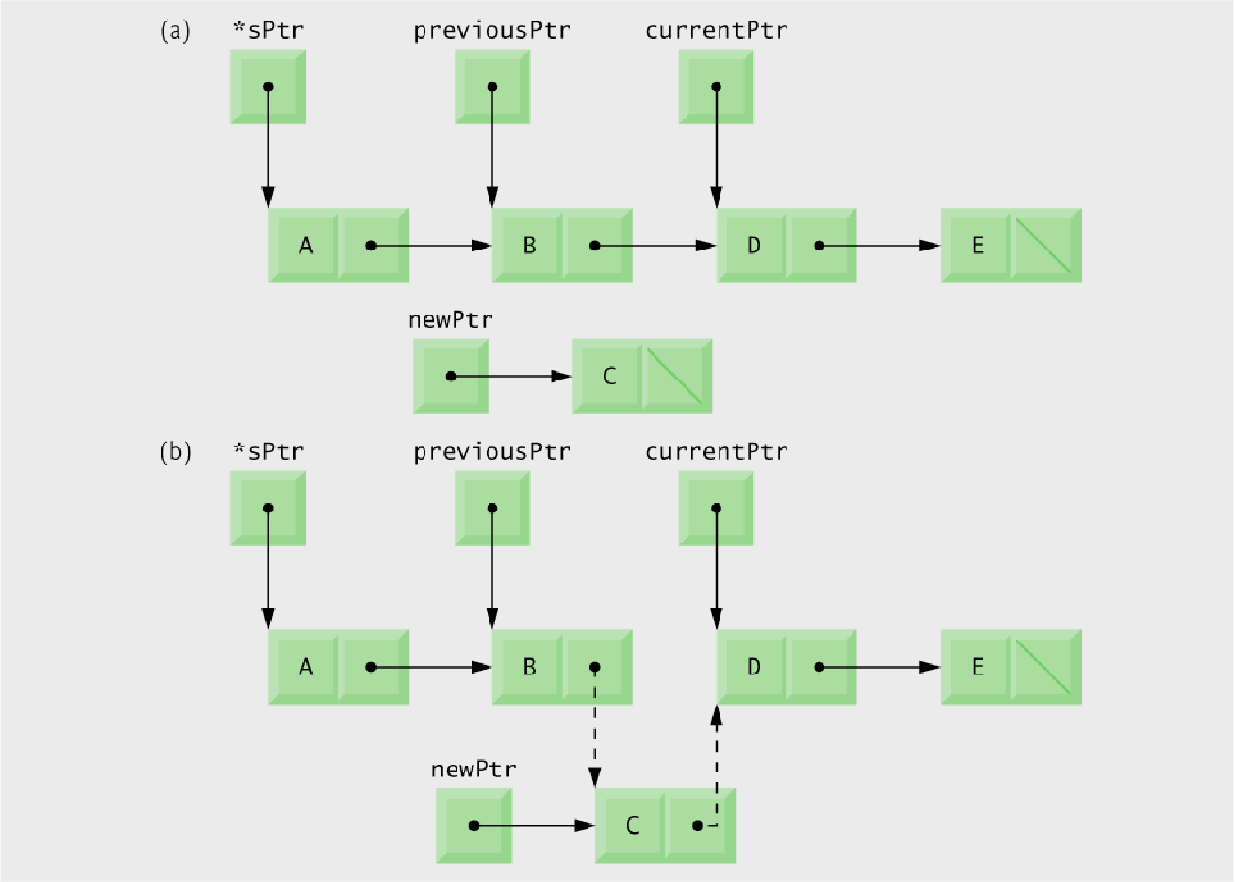
- Representación gráfica de una lista enlazada de números.



- 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.

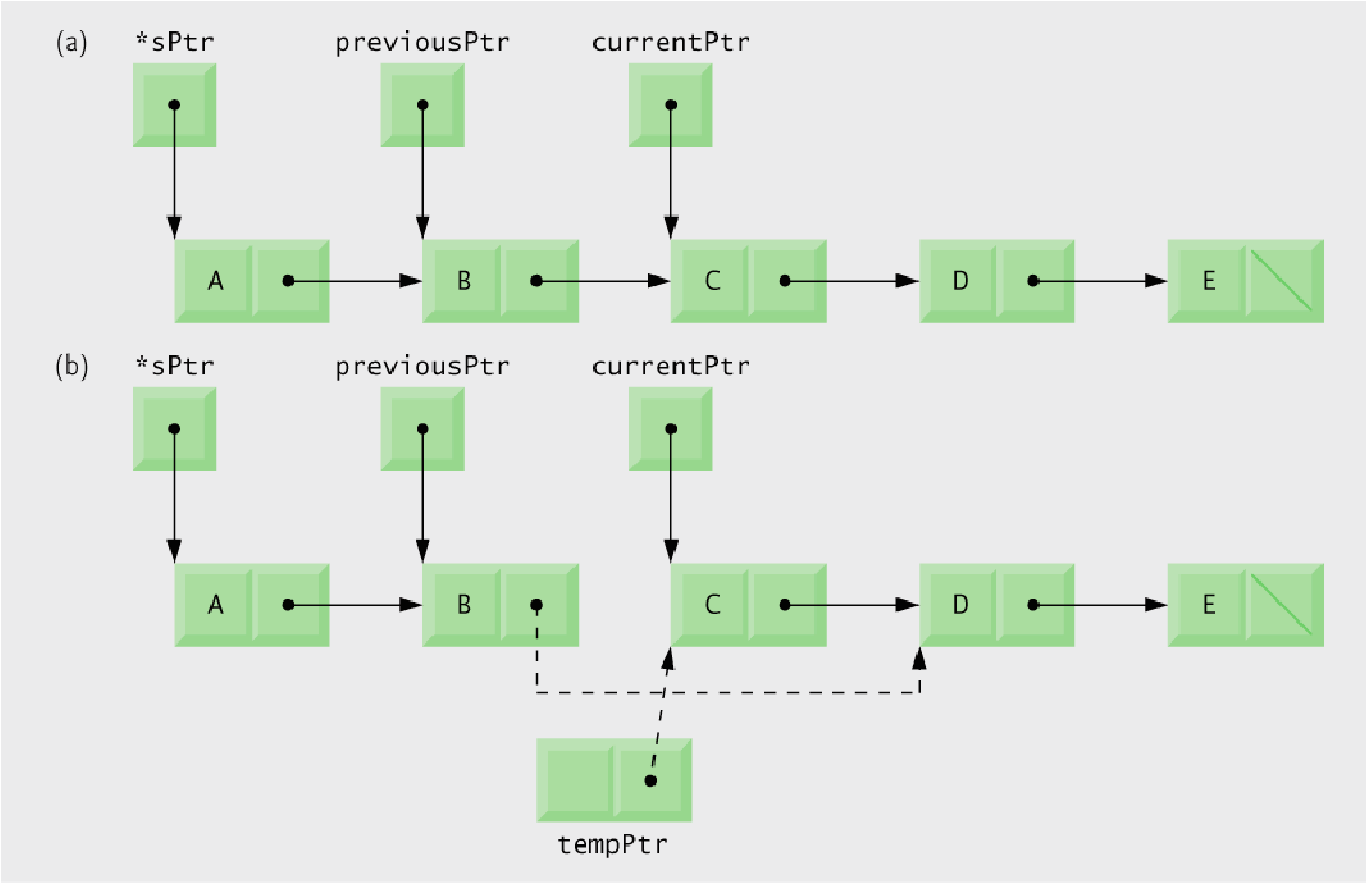
Listas enlazadas – Inserción de un nodo

- 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.

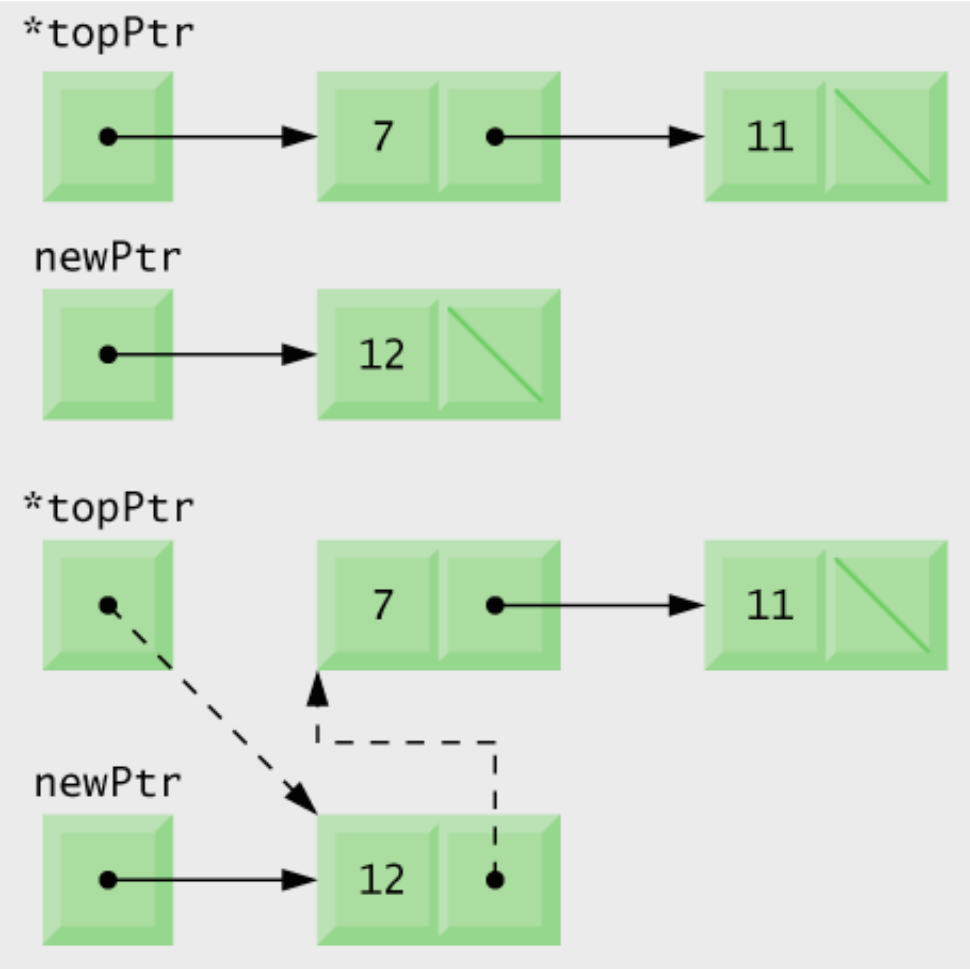


Pilas

- 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.

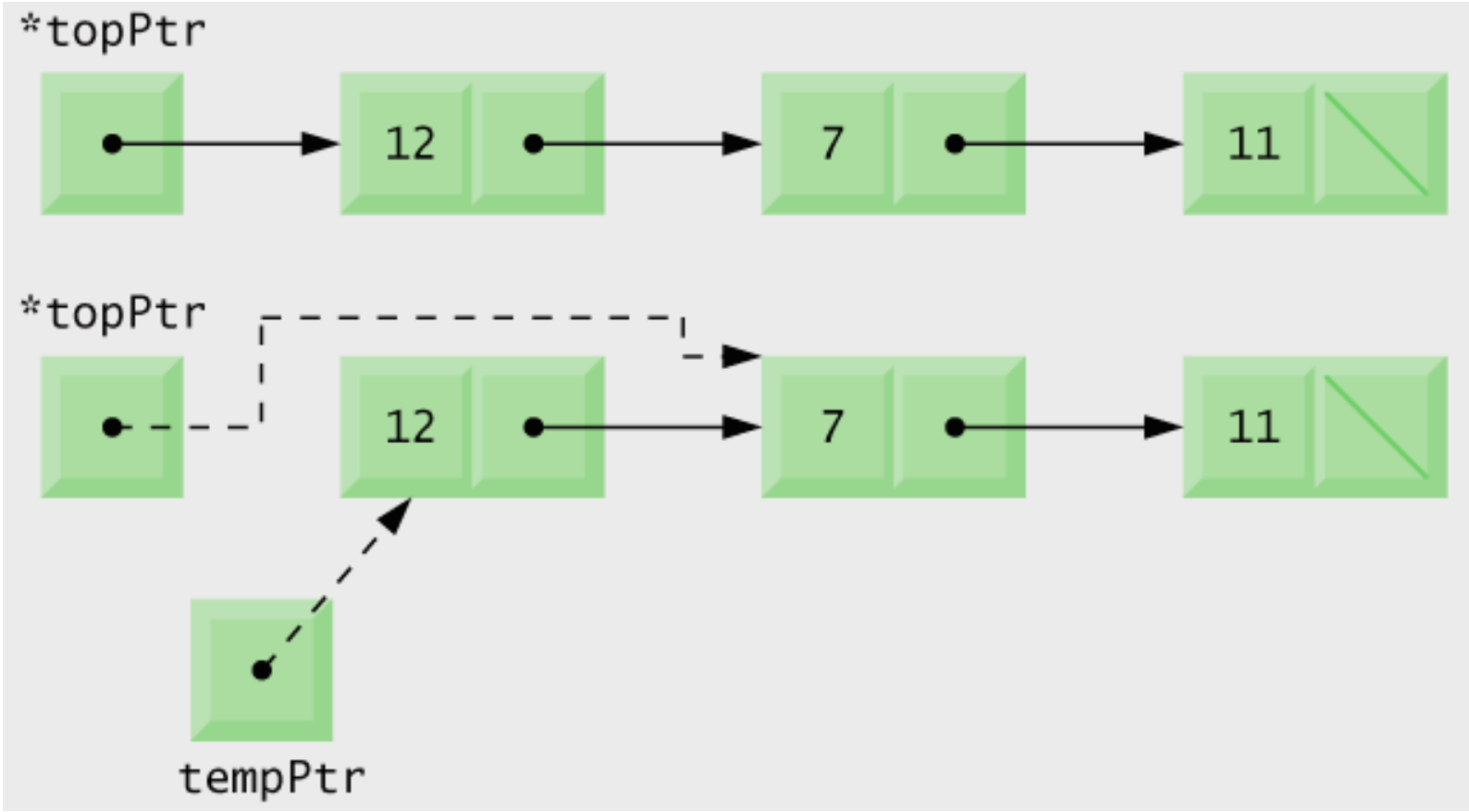
Pilas – Operación push

- Añadir un nuevo nodo en la cima de la pila



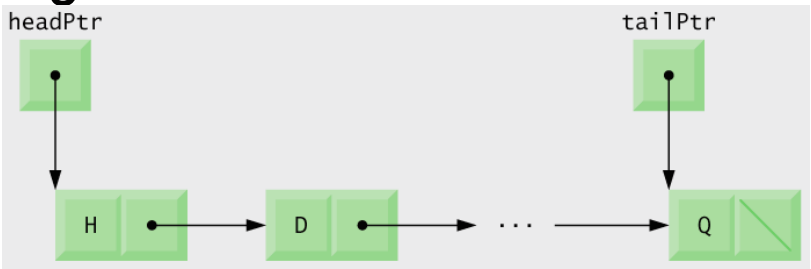
Pilas – Operación pop

- Elimina un nodo en la cima de la pila



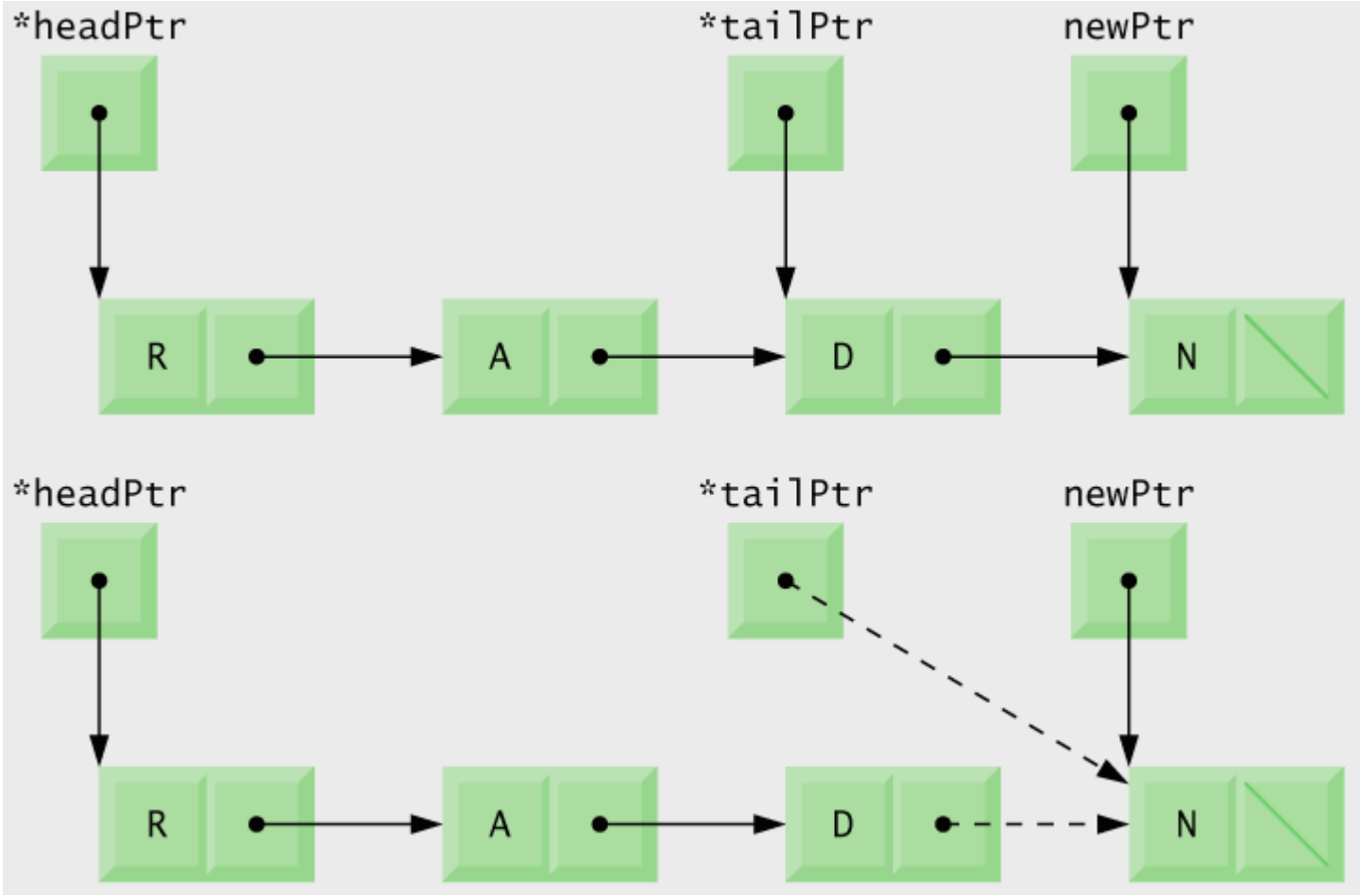
Colas

- 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:



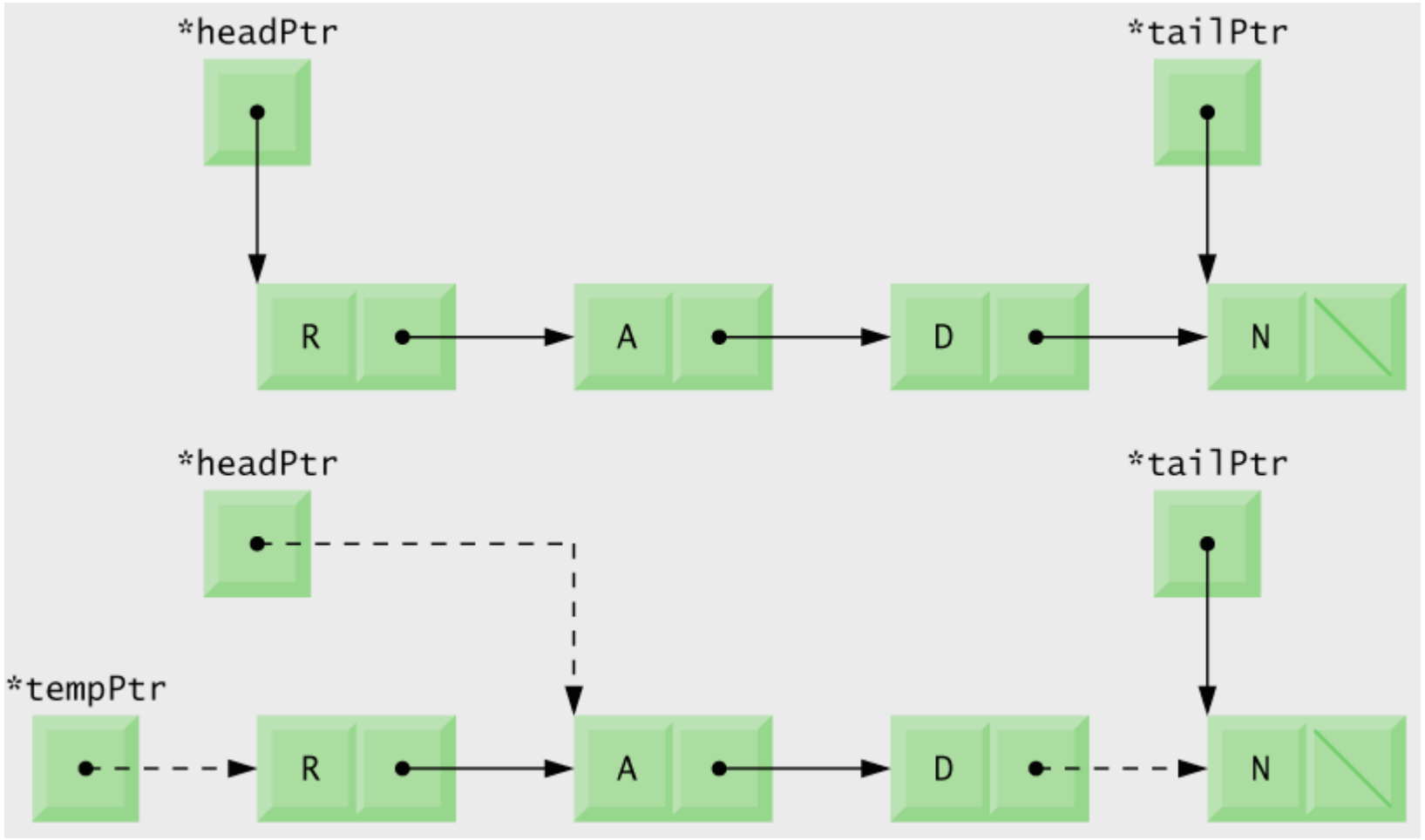
Colas – Operación encolar

- Inserta un nodo por la cola.



Colas – Operación desencolar

- Elimina un nodo por el frente.

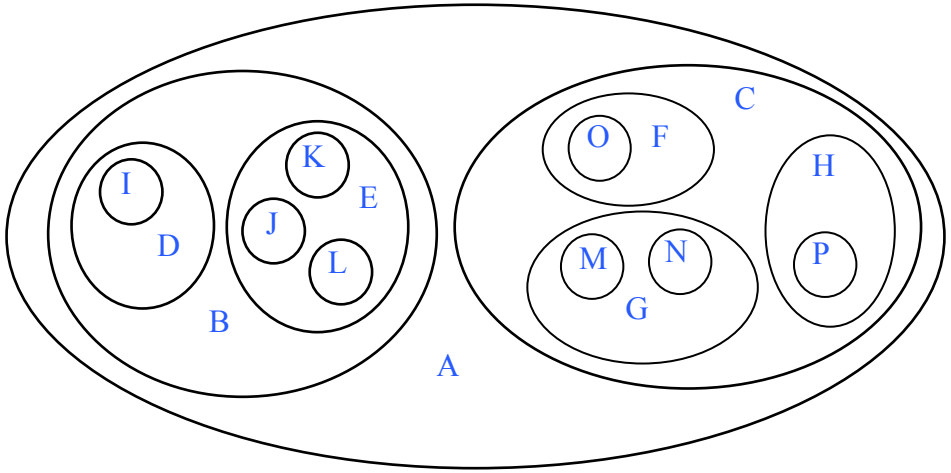


Arboles

- 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:

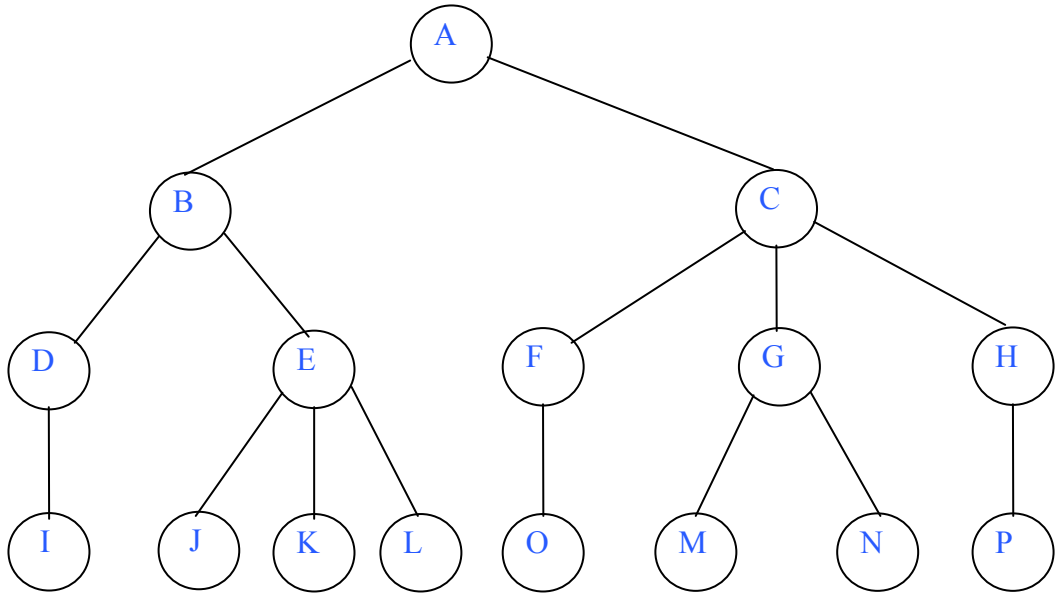
(A (B (D (I), E (J, K, L)), C (F (O), G (M, N), H (P))))

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):

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

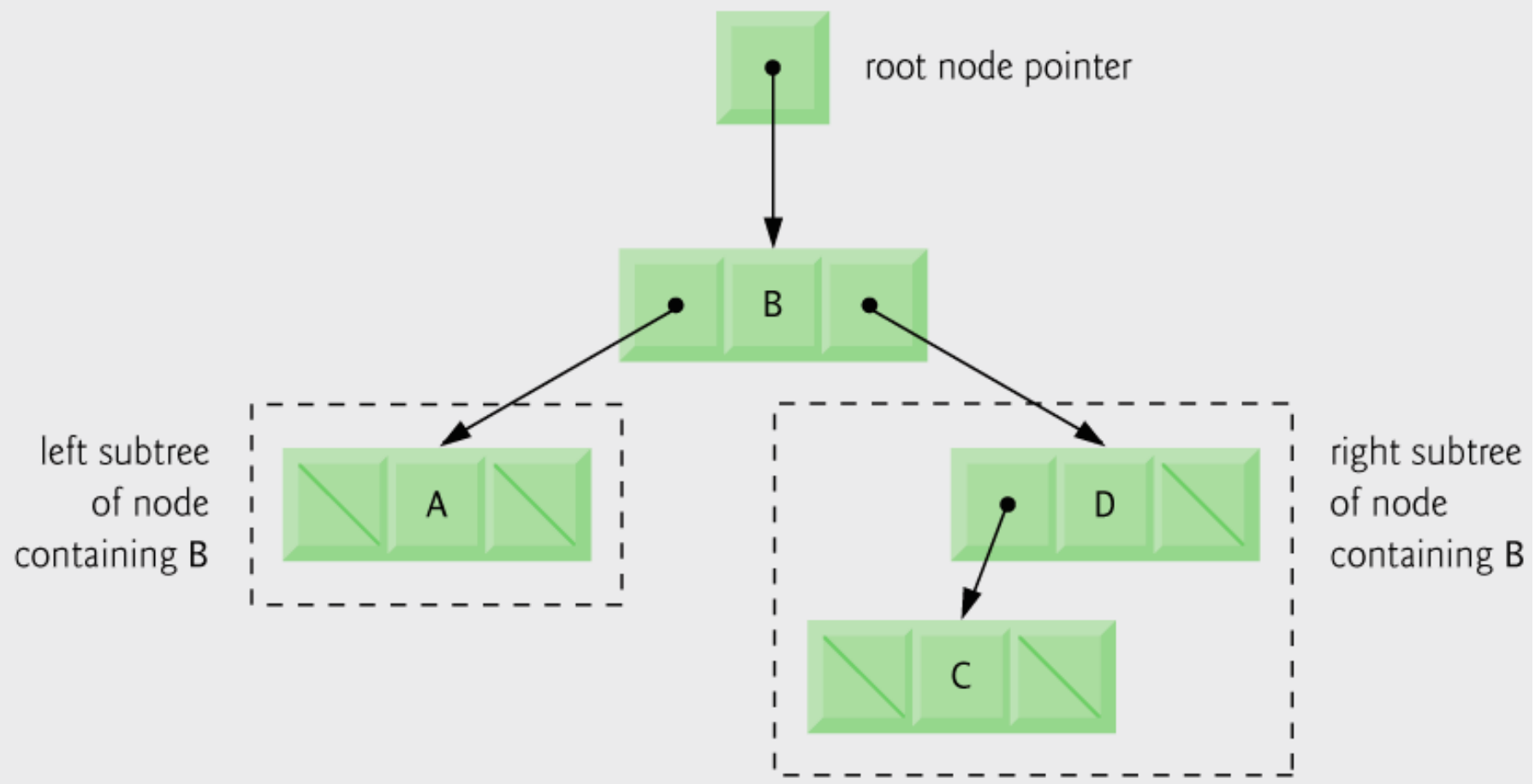
- Profundidad de un árbol binario de n nodos: $\boldsymbol{h = \log_2 n + 1}$

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)**

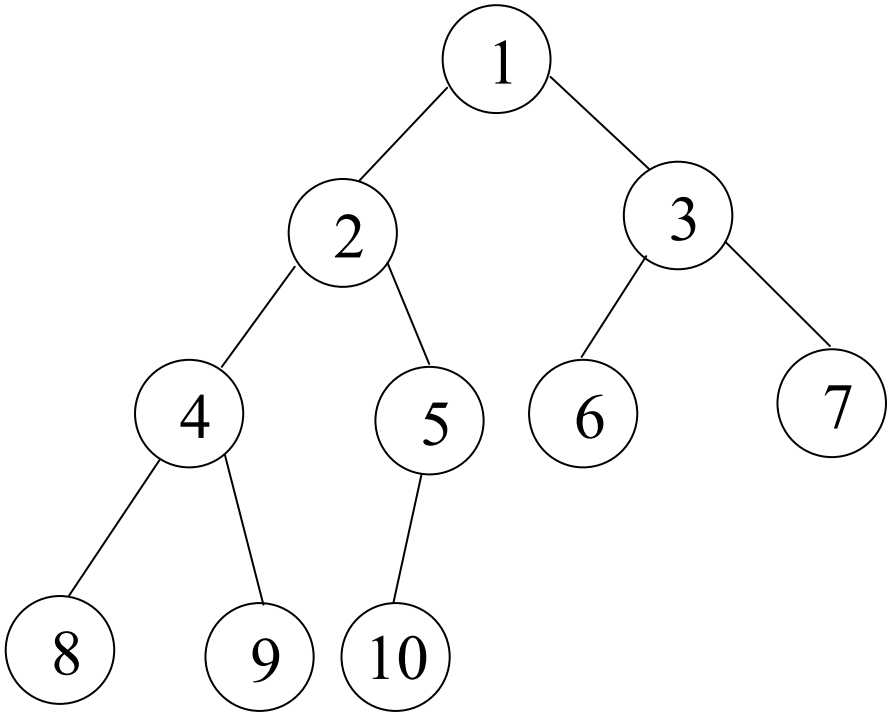
```
/* pre_orden: imprime el contenido del
   arbol con raiz p en pre-orden */
void pre_orden(struct arbol *p)
{
  if (p!=NULL) {
     printf("%4d   ",p->data);
     pre_orden(p->izq);
     pre_orden(p->der);
  }
}
```

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)

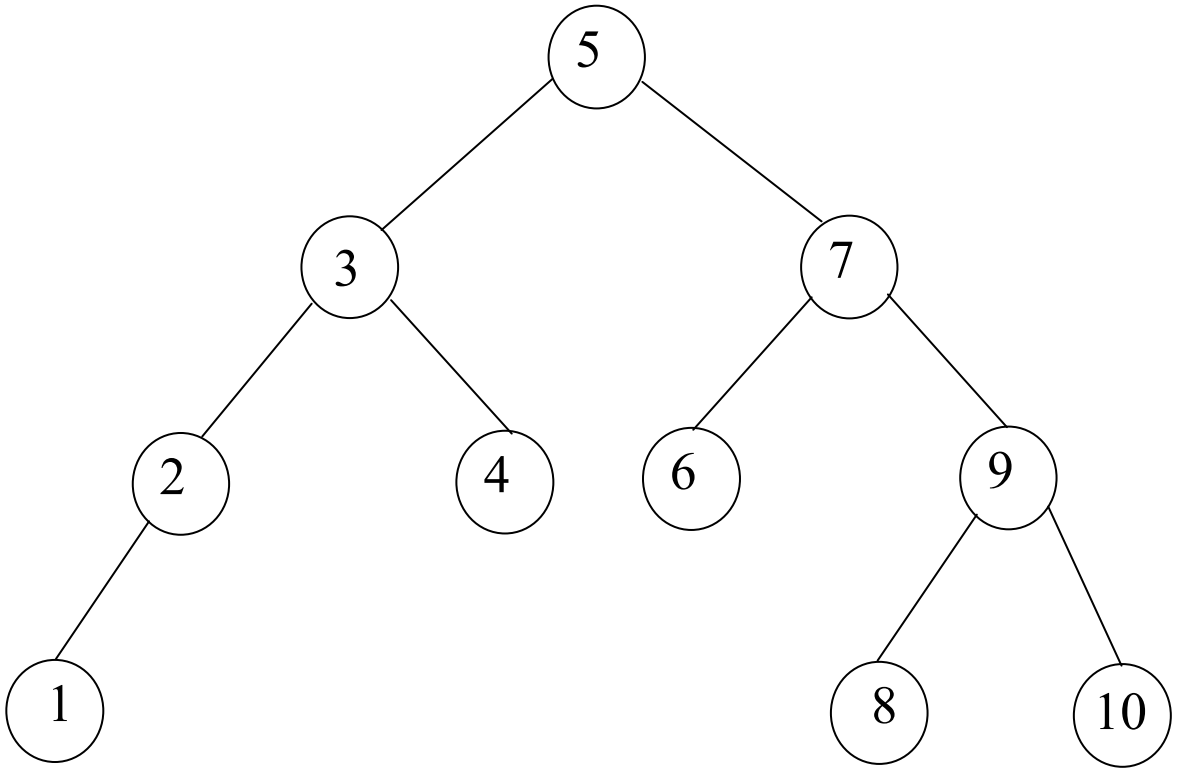
Inserción en un árbol binario de búsqueda

- 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;
}
```