Algoritmos de ordenación, búsqueda y mezcla sobre estructuras de datos no lineales.

Tema 2

Algorítmica y Modelos de Computación

Tema 2. Algoritmos de ordenación, búsqueda y mezcla sobre estructuras de datos no lineales.

1. Estructuras de datos no lineales. Introducción.
2. Algoritmos de ordenación.
   - 2.1. Métodos de ordenación lentos.
   - 2.2. Métodos de ordenación intermedios.
   - 2.3. Métodos de ordenación rápidos.
     - 2.3.1. Ordenamiento rápido (quick-sort)
     - 2.3.2. Ordenamiento por fusión (merge-sort)
     - 2.3.3. Ordenamiento por montículos (heap-sort)
   - 2.4. Comparación de métodos.
3. Algoritmos de búsqueda.
   - 3.1. Búsqueda lineal.
   - 3.2. Búsqueda binaria.
   - 3.3. Busqueda por hash.
4. Intercalación.

RECOMENDACIÓN: Repasar contenidos de Estructuras de Datos y FAA.

1. Estructuras de datos no lineales.

- Se denominan *tipos lineales* aquellos cuyos elementos forman una secuencia en la que es posible distinguir el primer elemento, del segundo, y en la que todo elemento salvo el último dispone de un estado sucesor.
  - Las diferencias entre los distintos tipos lineales provienen del uso especializado (o no) que soportan sus extremos izquierdo y derecho.
  - En una estructura lineal, cada elemento sólo puede ir enlazado al siguiente o al anterior.
- A las estructuras de datos *no lineales* se les llama también estructuras de datos multienlazadas. Se trata de estructuras de datos en las que cada elemento puede tener varios sucesores y/o varios predecesores.
  - Cada elemento puede estar enlazado a cualquier otro componente.
- Estructuras lineales de datos
  - Matrices (Arrays)
  - Pilas
  - Colas
  - Listas
- Estructuras no lineales de datos
  - Conjuntos dinámicos
  - Árboles
  - Grafos

2. Algoritmos de ordenación. Introducción.

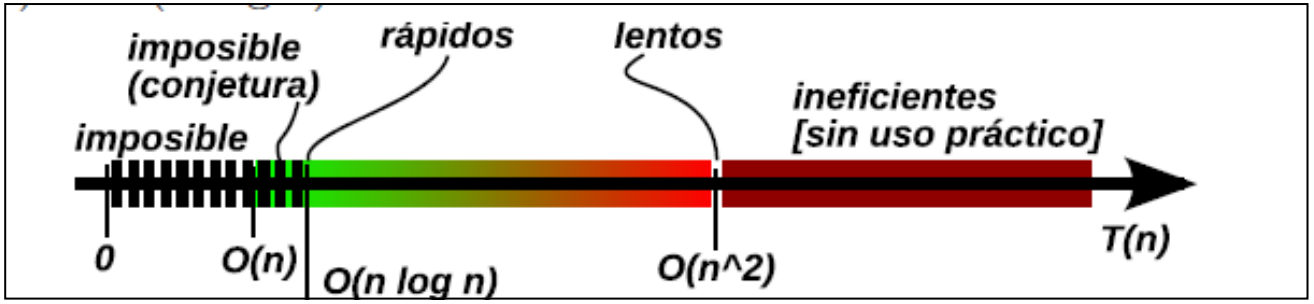
- El proceso de ordenar elementos (“sorting”) en base a alguna relación de orden es un problema muy frecuente y muchos usos. Nos centraremos en el ordenamiento interno, es decir cuando los elementos que han de ser ordenados están en la Memoria Principal del ordenador.
- Asumiremos normalmente que los elementos a ordenar pertenecen a algún tipo **key_t** con una *relación de orden* **<** y que están almacenados en un contenedor lineal (vector o lista).
- Recordar que vector<> es un contenedor de acceso aleatorio, es decir que acceder al elemento j- ésimo es $O(1)$ mientras que para list<> el tiempo es $O(n)$ (en realidad $O(j)$).

- **Eficiencia de los métodos de ordenamiento**
  - Por supuesto, el objetivo principal es obtener algoritmos de ordenamiento lo más eficientes posibles. En particular tiempo de cálculo y memoria requerida. Los tiempos de ejecución se compararán en base a su crecimiento con el tamaño del contenedor n.
  - Memoria adicional: Si el ordenamiento se hace sin requerir ningún tipo de memoria adicional (que crezca con n), decimos que es “in-place”. Si no, se debe tener en cuenta también como crece la cantidad de memoria requerida.

2. Algoritmos de ordenación. Introducción.

Discusión básica de eficiencia

- Como **premisa** para la discusión previa de la complejidad algorítmica de los métodos de ordenación tenemos $T(n) \geq O(n)$ ya que al menos hay que revisar los elementos a ordenar.
- Es relativamente sencillo proponer e implementar algoritmos $O(n^2)$ y existen una gran variedad de algoritmos con estos tiempos (ejemplo: el cálculo de los tiempos de ejecución del Método de la Burbuja es $O(n^2)$) .
- Existen algoritmos más rápidos con $T(n)=O(n \cdot log \cdot n)$, e incluso en casos particulares $O(n)$. A éstos los llamaremos "rápidos". Hay algunos intermedios (como "shell-sort" que es $O(n^{1.5})$).
- Se conjetura que no hay algoritmos generales de ordenación con $T(n)<O(n \ log \ n)$.



2. Algoritmos de ordenación. Introducción.

- **Complejidad algoritmos de ordenación.**
  - La complejidad de cualquier algoritmo estima el **tiempo de ejecución como una función del número de elementos a ser ordenados.**
  - Cada algoritmo estará compuesto de las siguientes operaciones:
    - COMPARACIONES que prueban si $A_i < A_j$ *ó* $A_i < B$ *(donde B es una variable auxiliar)*
    - INTERCAMBIOS: permutar los contenidos de A_i *y* A_j *ó* A_i *y* B
    - ASIGNACIONES de la forma $B \leftarrow A_i$, $A_j \leftarrow B$ *ó* $A_j \leftarrow A_i$
  - Generalmente, la función de complejidad **solo computa** COMPARACIONES porque el número de las otras operaciones es como mucho constante del **número de comparaciones**.

2. Algoritmos de ordenación. Introducción.

- **Clasificación de los Algoritmos de ordenación según eficiencia**:
  - **Lentos**
    - Método de burbuja (bubble-sort)
    - Ordenación por Inserción (insertion-sort)
    - Ordenación por Selección (selection-sort)
  - **Intermedios**
    - Ordenación Shell (shell-sort)
  - **Rápidos**
    - Ordenamiento por fusión (merge-sort)
    - Ordenamiento rápido (quick-sort)
    - Ordenamiento por montículos (heap-sort)

2. Algoritmos de ordenación. 2.1. Método burbuja.

Método de burbuja, “bubble-sort”

- Se basa en el principio de **comparar e intercambiar pares de elementos adyacentes hasta que todos estén ordenados.**
- Desde el primer elemento hasta el penúltimo no ordenado
  - comparar cada elemento con su sucesor
  - intercambiar si no están en orden

```
/*************************
    Ordenación por burbuja.
*************************/

template <typename elem>
void bubble_sort (vector<elem>& T) {
  int n = T.size();
  for (int i=0; i<n; ++i) {
    for (int j=n-1; j>i; --j) {
      if (T[j-1] > T[j]) {
        swap(T[j-1],T[j]);
      }
    }
  }
}

template <typename T>
void swap (T& x, T& y) {
  T z = x;
  x = y;
  y = z;
}
```

- **Complejidad Ordenación Burbuja** (Ver ejercicios tema 1).

- El tiempo de ejecución de dicho algoritmo viene determinado por el **número de comparaciones**, en todos los casos $O(n^2)$ (Si tuviéramos en cuenta que tras una pasada puede suceder que ya estén todos los elementos ordenados, en este caso no sería necesario seguir realizando comparaciones).

  - COMPARACIONES: $(n-1) + (n-2) + \ldots + 3 + 2 + 1 = n(n-1)/2 \Rightarrow O(n^2)$

  - INTERCAMBIOS (peor caso): $(n-1) + (n-2) + \ldots + 3 + 2 + 1 = n(n-1)/2 \Rightarrow O(n^2)$

- **Ventajas y Desventajas Ordenación Burbuja**

  - Su principal ventaja es la simplicidad del algoritmo.

  - El problema de este algoritmo es que solo compara los elementos adyacentes del vector. Si el algoritmo comparase primero elementos separados por un amplio intervalo y después se centrase progresivamente en intervalos más pequeños, el proceso sería más eficaz. Esto llevó al desarrollo de ordenación Shell y QuickSort.

Ordenación por inserción

- También conocido como **método de la baraja.**
- Consiste en **tomar elemento a elemento e ir insertando cada elemento en su posición correcta de manera que se mantiene el orden de los elementos ya ordenados.**

Inicialmente se toma el primer elemento, a continuación se toma el segundo y se inserta en la posición adecuada para que ambos estén ordenados, se toma el tercero y se vuelve a insertar en la posición adecuada para que los tres estén ordenados, y así sucesivamente.

1. Suponemos el primer elemento ordenado.
2. Desde el segundo hasta el último elemento, hacer:
    1. suponer ordenados los **(i –1) primeros elementos**
    2. tomar el elemento **i**
    3. buscar su posición correcta
    4. insertar dicho elemento, obteniendo **i elementos ordenados**

Código Ordenación por inserción

```
/***************************
    Ordenación por inserción.
***************************/
template <typename elem>
void insertion_sort (vector<elem>& T) {
  int n = T.size();
  for (int i=1; i<n; ++i) {//supone el primer elemento ordenado
    elem x = T[i]; // elemento a ordenar
    int j; // para posición a comparar
    for (j=i; j>0 and T[j-1]>x; --j) {
      T[j] = T[j-1]; // desplaza el elemento una posición a la derecha
    }
    T[j] = x; // se inserta el elemento a ordenar en su posición correcta
  }
}
```

2. Algoritmos de ordenación. 2.2. Método inserción.

- **Complejidad Ordenación por inserción** (Ver ejercicios tema 1).
- **CASO MEJOR**: Cuando el vector está ordenado, sólo se hace una comparación en cada paso.
  - **Ej. 15 20 45 60 n=4**

pasada	Nº intercambios	Nº comparaciones
1	0	1
2	0	1
3	0	1

  - En general, para n elementos se hacen $(n\text{-}1)$ comparaciones. Por tanto, $T(n)= O(n)$
- **CASO PEOR**: Cuando el vector está ordenado inversamente.
  - **Ej. 86 52 45 20 n=4**
  - **52 86 45 20**
  - **45 52 86 20**
  - **20 45 52 86**

pasada	Nº intercambios	Nº comparaciones
1	1	1
2	2	2
3	3	3

  - En general, para n elementos se realizan $(n\text{-}1)$ intercambios y $(n\text{-}1)$ comparaciones. Por tanto, $T(n)= O(n^2)$
- **CASO MEDIO**: Los elementos aparecen de forma aleatoria.
  - Se puede calcular como la suma de las comparaciones mínimas y máximas dividida entre dos:

$T(n) =((n\text{-}1) + n(n\text{-}1)/2)/2 = (n^2+ (n\text{-}2))/4$, por tanto complejidad $T(n)= O(n^2)$.

Ordenación por Selección

Este método se basa en que cada vez que se mueve un elemento, se lleva a su posición correcta. Se comienza examinando todos los elementos, se localiza el más pequeño y se sitúa en la primera posición. A continuación, se localiza el menor de los restantes y se sitúa en la segunda posición. Se procede de manera similar sucesivamente hasta que quedan dos elementos. Entonces se localiza el menor y se sitúa en la penúltima posición y el último elemento, que será el mayor de todos, ya queda automáticamente colocado en su posición correcta.

Para i desde la primera posición hasta la penúltima

- localizar menor desde i hasta el final
- intercambiar ambos elementos

Ordenación por Selección

```
/***************************
    Ordenación por selección.
***************************/

template <typename elem>
void selection_sort (vector<elem>& T) {
  int n = T.size();
  //Para todos los elementos desde el primero hasta el penultimo
  for (int i=0; i<n-1; ++i) {
    /*Se localiza el elemento menor desde la posición desde la cual se está ordenando hasta el último elemento*/
    int m = pos_min(T,i,n-1);
    swap(T[i],T[m]);/* se intercambian los elementos*/
  }
}


template <typename elem>
int pos_min (vector<elem>& T, int e, int d) {
  int p = e;
  for (int i=e+1; i<=d; ++i) {
    if (T[i] < T[p]) {
      p = i;
    }
  }
  return p;
}
```

Complejidad Ordenación por Selección

- El tiempo de ejecución de del algoritmo viene determinado por el **número de comparaciones,** las cuales son independientes del orden original de los elementos, el tiempo de ejecución es $O(n^2)$**.**
  - $T(n) = (n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2 \Rightarrow T(n) = O(n^2)$**.**
  - El número de intercambios es $O(n)$**.**
- **Por tanto complejidad** $O(n^2)$**.**

Desventajas Ordenación por Selección.

- Muy lenta con vectores grandes.
- No detecta si el vector está todo ni parcialmente ordenado.

Ordenación Shell

- Es una mejora de la ordenación por inserción (colocar cada elemento en su posición correcta, moviendo todos los elementos mayores que él, una posición a la derecha), que se utiliza cuando el número de datos a ordenar es grande.
- Para ordenar una secuencia de elementos se procede así:
  1. **Se selecciona una distancia inicial y se ordenan todos los elementos de acuerdo a esa distancia**, es decir, cada elemento separado de otro a *distancia* estará ordenado con respecto a él.
  2. Se disminuye esa distancia progresivamente, hasta que se tenga distancia 1 y todos los elementos estén ordenados.

Ejemplo Ordenación Shell

Posición	0	1	2	3	4	5	6	7	8
Original	4	14	21	32	18	17	26	40	6
Distancia 4	4	14	21	32	**6**	17	26	40	**18**
Distancia 2	4	14	**6**	**17**	**18**	**32**	**21**	40	**26**
Distancia 1	4	14	6	17	18	32	21	40	26
Final	4	6	14	17	18	21	26	32	40

2. Algoritmos de ordenación. 2.4. Método Shell.

```
/*****************************************
    Shell sort (con la secuencia de Shell)
*****************************************/

template <typename elem>
void shell_sort (vector<elem>& T) {
  int n = T.size();
  for (int h=n/2; h>0; h/=2) {/* distancia, los elementos estarán ordenados a una distancia igual a h.
                                 Se ordenan hasta que la distancia sea igual a 1 */
    for (int i=h; i<n; i++) {
      int j = i;
      elem v = T[i];
      while (j>=h and v<T[j-h]) {
        T[j] = T[j-h];
        j -= h;
      }
      T[j] = v;
    }
  }
}
```

```
/*****************************************
    Shell sort (con la secuencia de Sedgewick)
*****************************************/

template <typename elem>
void shell_sort_s (vector<elem>& T) {
  int n = T.size();
  int t[] = { 0,1,8,23,77,281,1073,4193,16577,
              65921,262913,1050113,4197377,16783361,
              67121153,268460033,1073790977};
  int k=1;
  while (t[k]<n) k++;
  while (k>0) {
    int h = t[k--];
    for (int i=h; i<n; i++) {
      int j = i;
      elem v = T[i];
      while (j>=h and v<T[j-h]) {
        T[j] = T[j-h];
        j -= h;
      }
      T[j] = v;
    }
  }
}
```

Complejidad de Shell

- Su implementación original, requiere $O(n^2)$ comparaciones e intercambios en el peor caso. Un cambio menor presentado por V. Pratt produce una implementación con un rendimiento de $O(n \log^2 n)$ en el peor caso. Esto es mejor que las $O(n^2)$ comparaciones requeridas por algoritmos lentos pero peor que el óptimo O(n log n). Aunque es fácil desarrollar un sentido intuitivo de cómo funciona este algoritmo, es muy **difícil** analizar su tiempo de ejecución.

- Dependiendo de la elección de la secuencia de espacios, Shell-sort tiene un tiempo de ejecución en el peor caso de $O(n^2)$, así:

  - usando los incrementos de Shell (que comienzan con n/2, n el tamaño del vector y se dividen por 2 cada vez), $O(n^{3/2})$
  - usando los incrementos de Hibbard (de $2^k - 1$), $O(n^{4/3})$
  - usando los incrementos de Sedgewick , $O(n \lg^2 n)$
  - y posiblemente mejores tiempos de ejecución no comprobados.

- La existencia de una implementación $O(n \lg n)$ en el peor caso del Shell-sort permanece como una pregunta por resolver.

Ordenación Rápida (QuickSort)

- Quick-sort se basa en la estrategia "dividir para vencer". Tomamos un valor x (llamado "**pivote**") y separamos los valores que son mayores o iguales que x a la derecha y los menores a la izquierda. Está claro que ahora solo hace falta ordenar los elementos en cada uno de los subrangos

La elección del elemento **Pivote** se puede seleccionar de diferentes formas:

- El mayor de los dos primeros elementos distintos encontrados.
- El primer elemento.
- El último elemento.
- El elemento medio.
- Un elemento aleatorio.
- Mediana de tres elementos (El primer elemento, el elemento del medio y el último elemento).

- **Complejidad Quick-Sort**. (Ver apuntes tema 3).

Código Quick-Sort

```
void quick_sort (vector<elem>& T, int e, int d) {// Ordena el rango [e,d) de 'T'
  if (e<d) {
    int q = partition(T,e,d);
    quick_sort(T,e,q);
    quick_sort(T,q+1,d);
  }
}
```

- **CASO MEJOR:** Cuando el pivote, divide al conjunto en dos subconjuntos de igual tamaño. En este caso hay dos llamadas con un tamaño de la mitad de los elementos, y una sobrecarga adicional lineal, igual que en MergeSort.
  - Si $n = d-e$ es la longitud del vector y $n_1 = q - e$ y $n_2 = d - (q+1)$ $\Rightarrow$

    $$T(n) = T_{partition}(n) + T(n_1) + T(n_2)$$
  - Si logramos que $T_{partition}(n) = cn$ y que $T(\text{el vector})$ sea tal que las longitudes de los subsegmentos estén bien "balanceados" ($n_1 \approx n_2 \approx n/2$) entonces $T(n) = cn + 2T(n/2)$
  - El análisis de eficiencia da lugar a una ecuación recurrente para el tiempo de ejecución. Suponemos n potencia de 2, $n = 2^K$
  - Para $n = 1$, Tiempo constante, $O(1)$
  - Para $n > 1$, El tiempo de ordenación para n números es igual al tiempo para 2 ordenaciones recursivas de tamaño $n/2$ + el tiempo para la partición (que es lineal). Por tanto, $T(n) = 2T(n/2) + cn$
  - La solución de la ecuación da para la complejidad del algoritmo Quick-Sort es

    $$T(n) = O(n \log n)$$.

Complejidad Quick-Sort

- **Caso Peor:** Se podría esperar que los subconjuntos de tamaño muy distinto proporcionen resultados malos, es decir, Si el particionamiento es muy desbalanceado $n_1 = 1$ y $n_2 = n - 1$
- Supongamos que en cada paso de recursión sólo hay un elemento menor a pivote. En tal caso el subconjunto I (elementos menores que pivote) será uno y el subconjunto D (elementos mayores o igual a pivote) serán todos los elementos menos uno. El tiempo de ordenar 1 elemento es sólo 1 unidad, pero cuando n > 1.

  - T(n) = T(n-1) + n
  - T(n-1) = T(n-2) + (n-1)
  - T(n-2) = T(n-3) + (n-2) ….
  - T(2) = T(1) + 2
  - T(n) = T(1) + 2 + 3 + 4 + …+ n = n (n+1) /2 $\Rightarrow T(n) = O(n^2)$.

- **Caso medio:** puede demostrarse que el orden se mantiene $O(n \log n)$.

☐ Código Quick-Sort.

```
/*****************************************
    Ordenación rápida (partición de Hoare).
*****************************************/

template <typename elem>
void quick_sort (vector<elem>& T) {
  quick_sort(T,0,T.size()-1);
}


template <typename elem>
void quick_sort (vector<elem>& T, int e, int d) {// Ordena el rango [e,d) de 'T'
  if (e<d) {
    int q = partition(T,e,d);
    quick_sort(T,e,q);
    quick_sort(T,q+1,d);
  }
}

template <typename elem>
int partition (vector<elem>& T, int e, int d) {
  elem x = T[e];   int i = e-1;  int j = d+1; //x= Pivote
  for (;;) {
    while (x < T[--j]);
    while (T[++i] < x);
    if (i>=j) return j;
    swap(T[i],T[j]);
  }
}
```

Ordenación por mezcla (Mergesort)

La estrategia es típica de **"dividir para vencer" :**

- ***Dividir*** los elementos en dos secuencias de la misma longitud aproximadamente.
- ***Ordenar*** de forma independiente cada subsecuencia.
- ***Mezclar*** las dos secuencias ordenadas para producir la secuencia final ordenada.

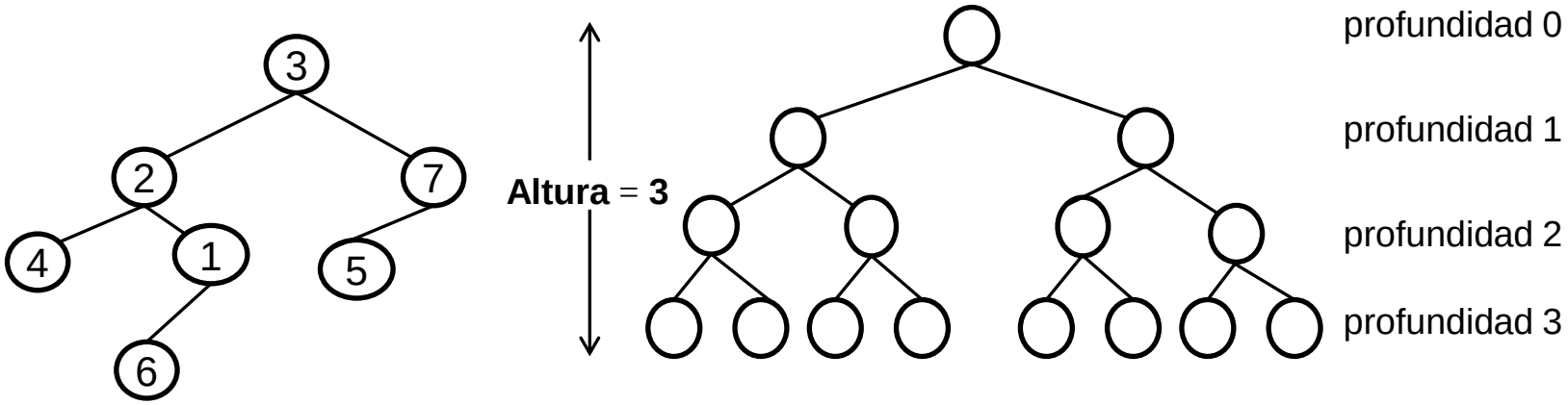
```
/*************************
    Ordenación por fusión.
*************************/
template <typename elem>
void merge_sort (vector<elem>& T) {
  merge_sort(T,0,T.size()-1);
}
template <typename elem>
void merge_sort (vector<elem>& T, int e, int d) {
  if (e<d) {//Si hay más de un elemento
    int m = (e+d)/2;
    merge_sort(T,e,m);
    merge_sort(T,m+1,d);
    merge(T,e,m,d); //Procedimiento que mezcla el resultado de las dos llamadas anteriores
  }
}
```

```
/*************************
    Ordenación por fusión.
*************************/
template <typename elem>
void merge_sort (vector<elem>& T) {
  merge_sort(T,0,T.size()-1);
}
template <typename elem>
void merge_sort (vector<elem>& T, int e, int d) {
  if (e<d) {//Si hay más de un elemento
    int m = (e+d)/2;
    merge_sort(T,e,m);
    merge_sort(T,m+1,d);
    merge(T,e,m,d); //Procedimiento que mezcla el resultado de las dos llamadas anteriores
  }
}

template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
  vector<elem> B(d-e+1); //B es un array auxiliar para la mezcla
  int i = e;   //Variable de primer elemento de la primera subsecuencia
  int j = m+1; //Variable del primer elemento de la segunda subsecuencia
  int k = 0;
  while (i<=m and j<=d) {
    if (T[i]<=T[j]) {
      B[k++] = T[i++];
    } else {
      B[k++] = T[j++];
    }
  }
  while (i<=m) {//Si se agotaron todos los elementos de la segunda subsecuencia
    B[k++] = T[i++];
  }
  while (j<=d) {//Si se agotaron los de la primera subsecuencia
    B[k++] = T[j++];
  }
  for (k=0; k<=d-e; ++k) {//Copiar todos los elementos del auxiliar al array
    T[e+k] = B[k];
  }
}
```

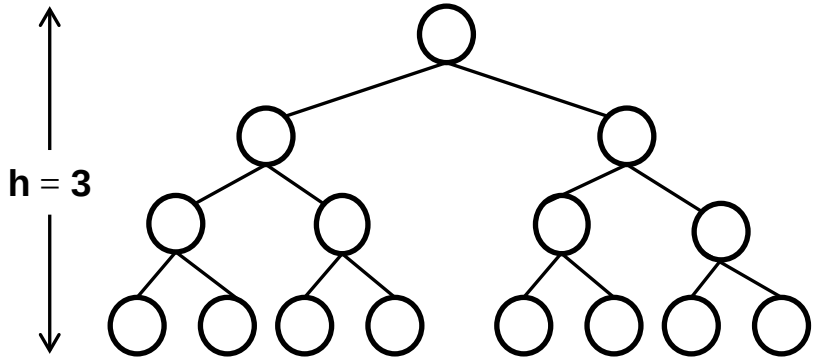
- **Código Mezcla (merge) de MergeSort.** El proceso de ***mezcla*** es el siguiente:
  - Repetir mientras haya elementos en una de las dos secuencias:
    - Seleccionar el menor de los elementos de las subsecuencias y añadirlo a la secuencia final ordenada.
    - Eliminar el elemento seleccionado de la secuencia a la que pertenece.
    - Copiar en la secuencia final los elementos de la subsecuencia en la que aún quedan elementos.
- **Complejidad merge de MergeSort.** (Ver apuntes tema 3).
  - Teniendo en cuenta que la entrada consiste en el total de elementos n y que cada comparación asigna un elemento al vector auxiliar (B), el número de comparaciones es $(n-1)$ y el número de asignaciones es n. Por lo tanto, el algoritmo de mezcla se ejecuta en un tiempo lineal $O(n)$.
- **Complejidad de MergeSort.** (Ver apuntes tema 3).
- El análisis de eficiencia de la ordenación por mezcla da lugar a una ecuación recurrente para el tiempo de ejecución. Suponemos n potencia de 2, $n = 2^K$
  - Para $n = 1$, Tiempo constante, $O(1)$
  - Para $n > 1$, El tiempo de ordenación para n números es igual al tiempo para 2 ordenaciones recursivas de tamaño $n/2$ + el tiempo para mezclar (que es lineal). Por tanto, $T(n) = 2T(n/2) + n$
- La solución de la ecuación es $T(n) = n + n \log n$. Por tanto la complejidad del algoritmo MergeSort es de $O(n \log n)$.

- Un á**rbol binario** es un árbol en el que el máximo número de hijos de cada nodo es 2 (**hijo izquierdo** e **hijo derecho**).



- Un árbol binario se dice que es **completo** (o lleno) si todas las hojas tienen la misma profundidad y todos los nodos internos tienen grado 2.
- Un árbol binario es **casi-completo** si el árbol es completo, a excepción quizás en el nivel de las hojas, en el cual todas las hojas están tan a la izquierda como sea posible.

Propiedades de los árboles binarios completos



profundidad	Núm nodos
0	1=2^0
1	2=2^1
2	4=2^2
3	8=2^3

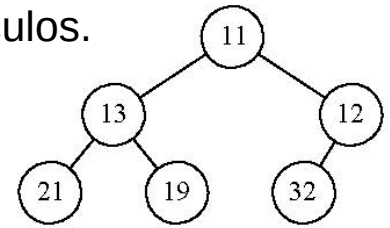
Árbol binario completo de altura h=3, con 8 hojas y 7 nodos internos

- Número de nodos: La raíz tiene dos hijos de profundidad 1, cada uno de los cuales tienen 2 hijos de profundidad 2, etc.⇒ nodos de profundidad $i=2^i$.
- **Hojas**: 2^h y **nodos internos**: $2^0+2^1+\ldots+2^{h-1}= \sum_{i=0}^{h-1} 2^i = 2^h - 1$
- **Número de nodos** $nodos\_total: n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
- **La altura:** $n = 2^{h+1} - 1 \Rightarrow h = \lfloor \log_2 n \rfloor$

Propiedades de los árboles binarios

- El máximo número de nodos de profundidad i es 2^i.
- El máximo número de nodos en un árbol binario de altura h es $2^{h+1}-1$.
- El máximo número de nodos internos es 2^h-1.
- El máximo número de hojas 2^h.
- La altura mínima de un árbol binario con n nodos es $\lfloor \log_2 n \rfloor$
- Para cualquier árbol binario no vacío, si n_1 es el número de hojas y n_2 es el número de nodos de grado 2, entonces $\boldsymbol{n_1=n_2+1}$.

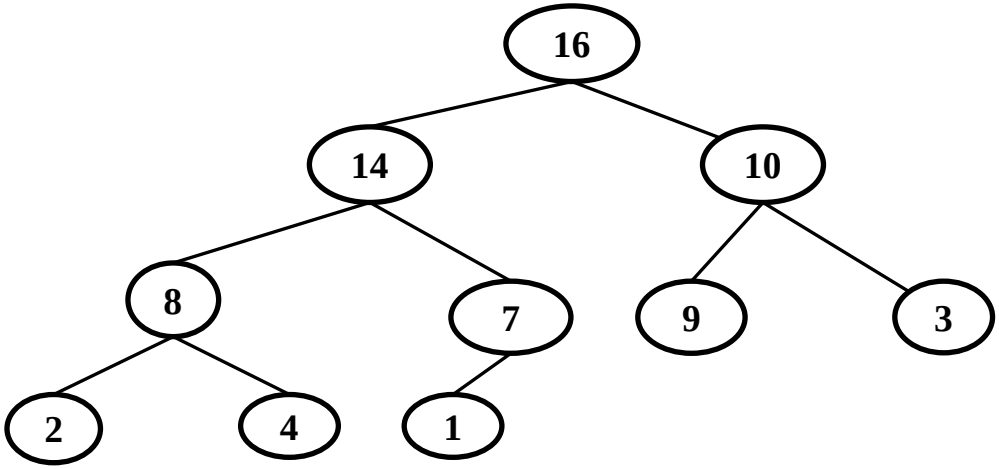
- Los **montículos** son implementaciones eficientes de las colas de prioridad. Un montículo es un árbol binario con una propiedad invariante:
  - o bien es el árbol vacío,
  - o bien la raíz es menor (mayor) o igual que el resto de los elementos y, adicionalmente, los hijos izquierdo y derecho son montículos.



- Ello implica que **el menor (mayor) elemento** de la estructura se encuentra en la raíz, y se puede consultar con coste constante.
- Para **añadir** un elemento, se añade como una hoja del árbol y después se la hace **flotar**, intercambiándola con sus ancestros, hasta que se restaura la propiedad anterior.
- En el peor caso, el elemento insertado se compara con tantos elementos como la altura del árbol. Si este es de altura mínima, el coste estará en **O (log n)**.
- Para **borrar** el mínimo, se reemplaza el elemento en la raíz por una de las hojas y a continuación se **hunde** esta, intercambiándola con el menor de sus hijos, hasta que se restaura la propiedad anterior. De nuevo, el coste estará en **O (log n)**.

Introducción: Montículos (Heap)

- **Montículos:**
  - Árbol binario **completo**.
  - Árbol binario **casi-completo**: es un árbol completo, a excepción quizás en el nivel de las hojas, en el cual están tan a la izquierda como sea posible.
  - Árbol **parcialmente ordenado** **(Montículo o Heap)**: es un árbol binario casi-completo con la propiedad de orden: el valor de cualquier nodo es mayor o igual –MaxHeap- (o menor o igual –MinHeap-) que el de sus nodos hijos.
- Ejemplo:

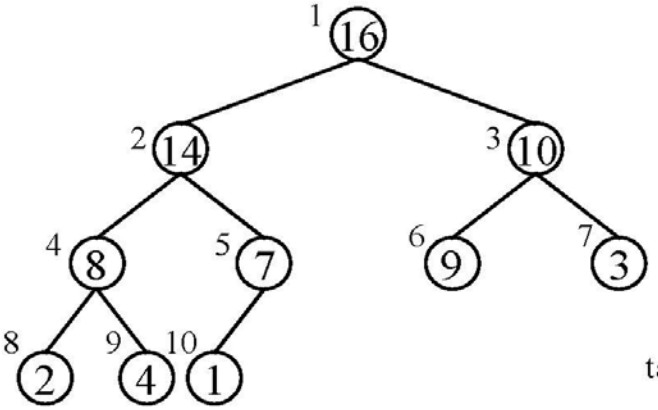


- **Propiedades**:
  - Todas las ramas del árbol son secuencias ordenadas.
  - La raíz del árbol es el nodo de máximo (mínimo) valor.
  - Todo subárbol de un Heap es, a su vez, un Heap.

Representación vectorial

- A [1] es la raíz y dado un nodo $A[i]$,
  - si $2i \leq n$: $A[2i]$ hijo izquierdo
  - si $2i + 1 \leq n$: $A[2i + 1]$ hijo derecho
  - si $i \neq 1$: $A[i/2]$ padre
- $A[\text{padre}(i)] \geq A[i]$, $2 \leq i \leq n$
- Heap de n elementos: altura $\Theta(\log n)$
- Ejemplo:

padre[i]=[i/2]
hizquierdo[i]=2i
hderecho[i]=2i+1



	1	2	3	4	5	6	7	8	9	10	11	…
A	16	14	10	8	7	9	3	2	4	1		

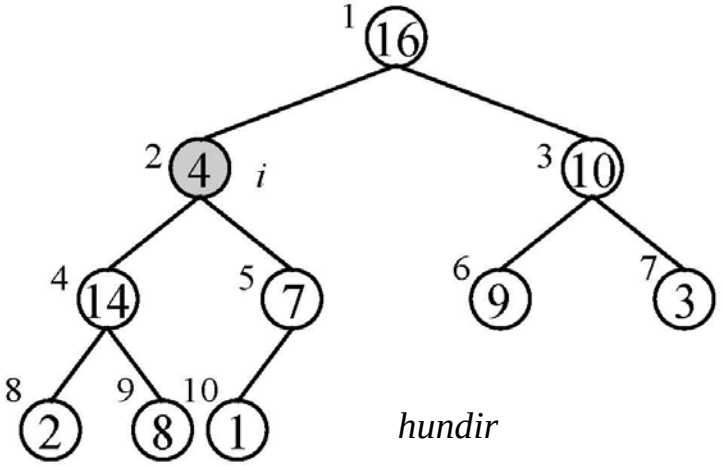
- **Operaciones básicas sobre Heaps.** Demostrarán la utilidad de esta estructura para:
  - El diseño de un algoritmo de **ordenación rápido (Heap-sort).**
  - La representación de **colas de prioridad**.

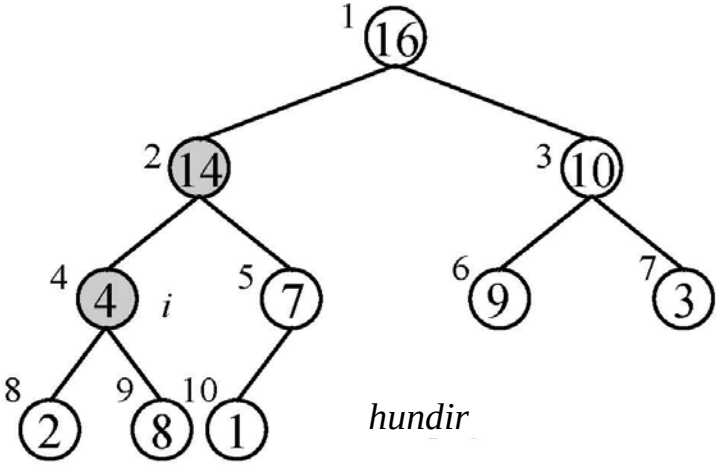
Hundir (“Ajustar”)	Es la propiedad clave para mantener la propiedad de Heap. $O(\log n)$
Crear_heap	Para convertir un vector, en principio desordenado en un Heap. $O(n)$
Heapsort	Para ordenar un vector. $O(n \log n)$
Consulta_máximo	Devuelve el máximo (return A[1]). $O(1)$
Eliminar_máximo	Para extraer el máximo del conjunto y eliminarlo. $O(\log n)$
Insertar	Para insertar en el conjunto. $O(\log n)$

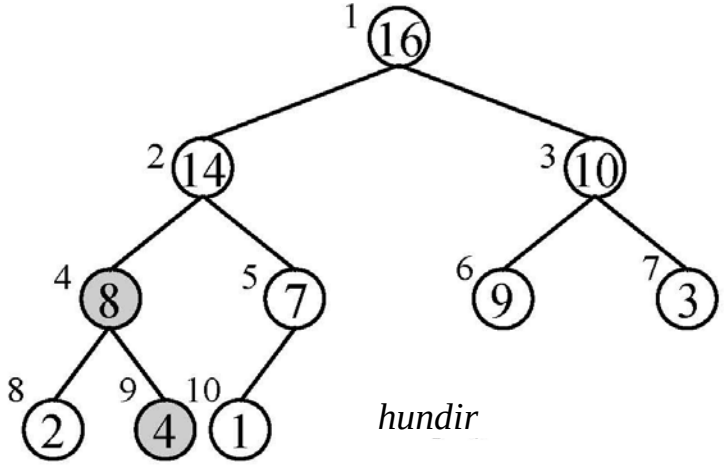
Hundir. Ejemplo.

Problema: Se tiene un Heap A y sea i, $1 \leq i \leq n$, una posición del vector tal que el subárbol izquierdo $(2i)$ y el derecho $(2i+1)$ son Heaps. Se desea transformar el subárbol que parte del nodo i en un Heap.

Estrategia Hundir: "hunde" el elemento $A[i]$ en el heap de forma que el subárbol que tiene por raíz el elemento con índice i se convierte en un Heap.







- **Hundir. Algoritmo iterativo.**

```
procedimiento hundir(T[1..n],i)
   k ← i;
   repetir
      j ← k;
      /* Buscar el hijo mayor del nodo j */
      si 2*j ≤ n y T[2*j] > T[k] entonces k ← 2*j
      si 2*j ≤ n y T[2*j] > T[k] entonces k ← 2*j+1
      intercambiar T[j] y T[k]
     /* si j = k, entonces el nodo ha llegado a su posición final */
   hasta que j = k
fprocedimiento
```

- **Hundir. Costes.**

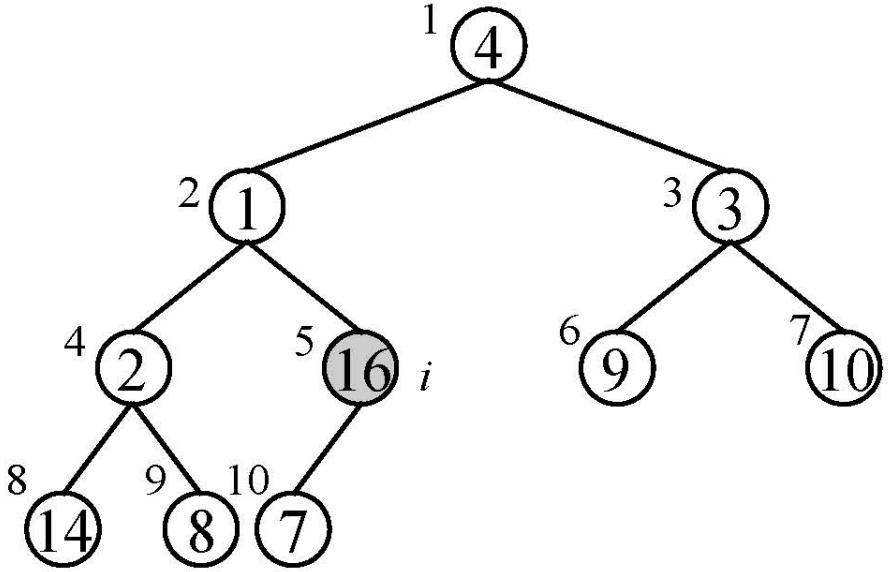
Coste: si h es la altura del árbol que tiene por raíz el elemento con índice i,

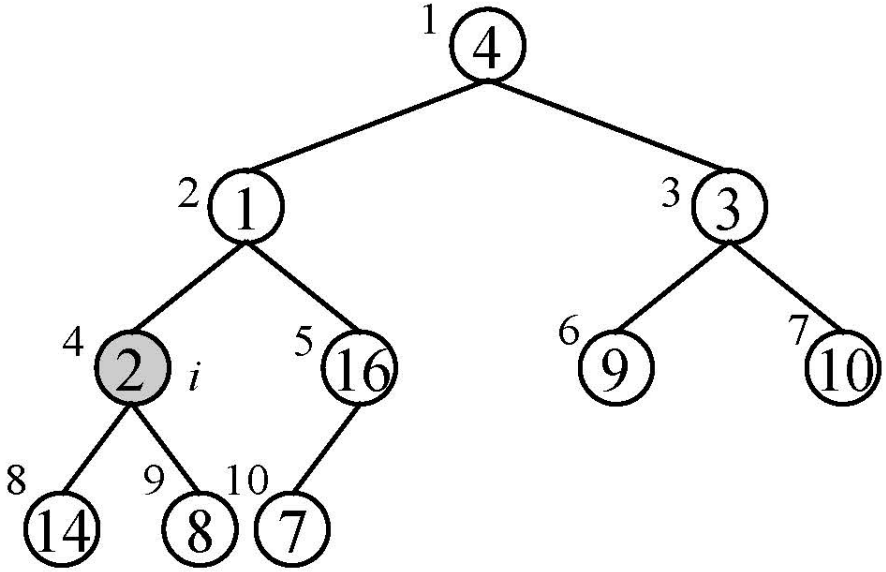
Peor caso: $O(h) \in O(\log n)$, $n = \textit{heap-size}[A]$

☐ Crear un Heap. Ejemplo.

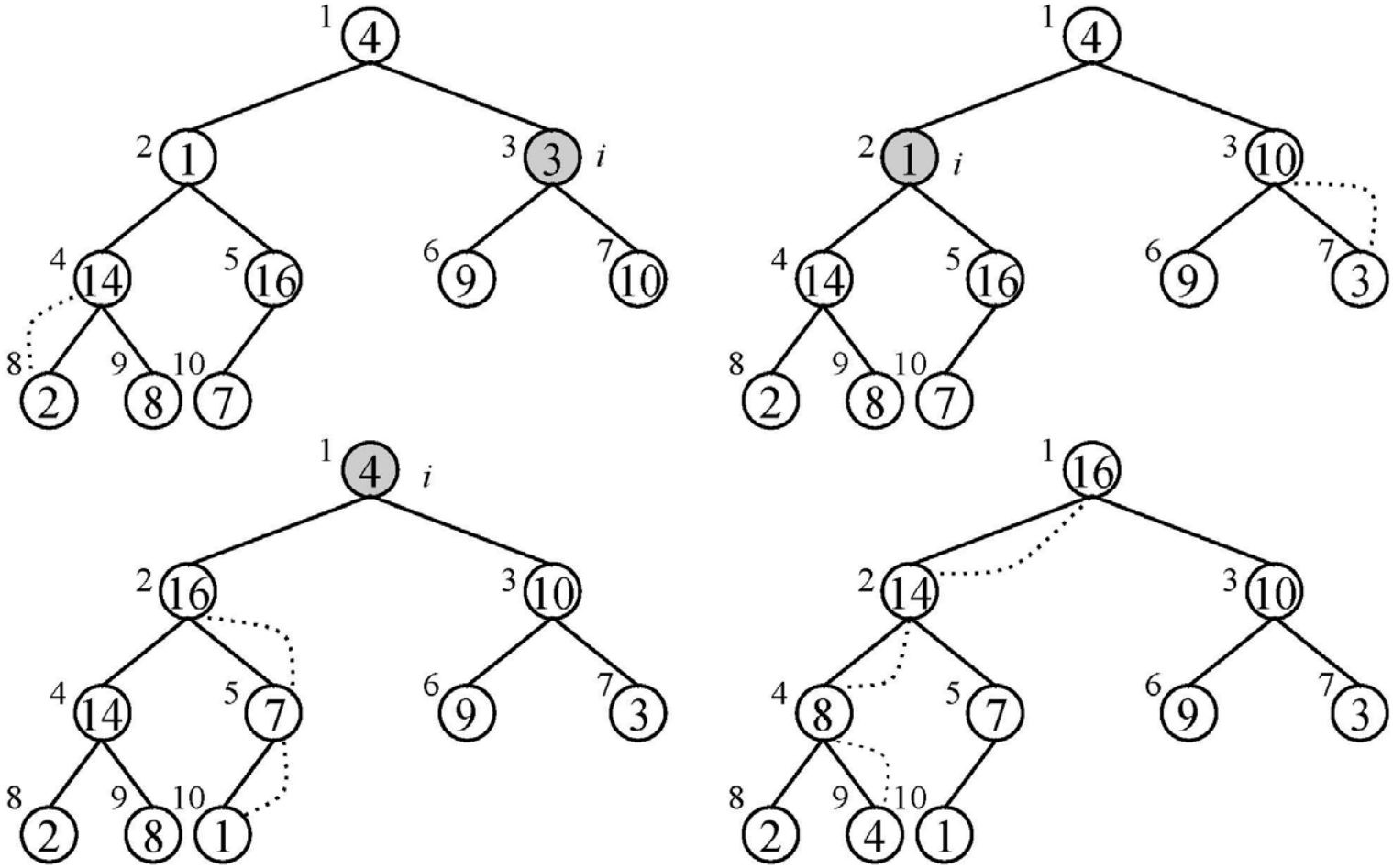
$$A = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$$

Crear_heap: convierte un vector A en un Heap.





☐ Crear un Heap. Ejemplo.



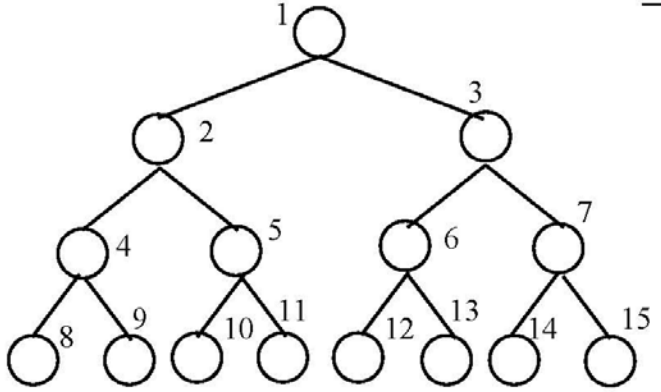
Crear un Heap. Algoritmo.

```
procedimiento crear-montículo(T[1..n])
   para i ← [n/2] hasta 1 hacer
      hundir(T,i)
fprocedimiento
```

Complejidad temporal

- Cota NO ajustada: $O(n)$ llamadas a **hundir** $O(n \log n)$
- Cota ajustada: $O(n)$
  - Coste **hundir** de un nodo es proporcional a su altura $O(h)$
  - Propiedad: En un Heap de n elementos hay, como mucho, $2^{\lfloor \log n \rfloor}/2^h$ nodos de altura h.

☐ Cota ajustada de Crear un Heap.

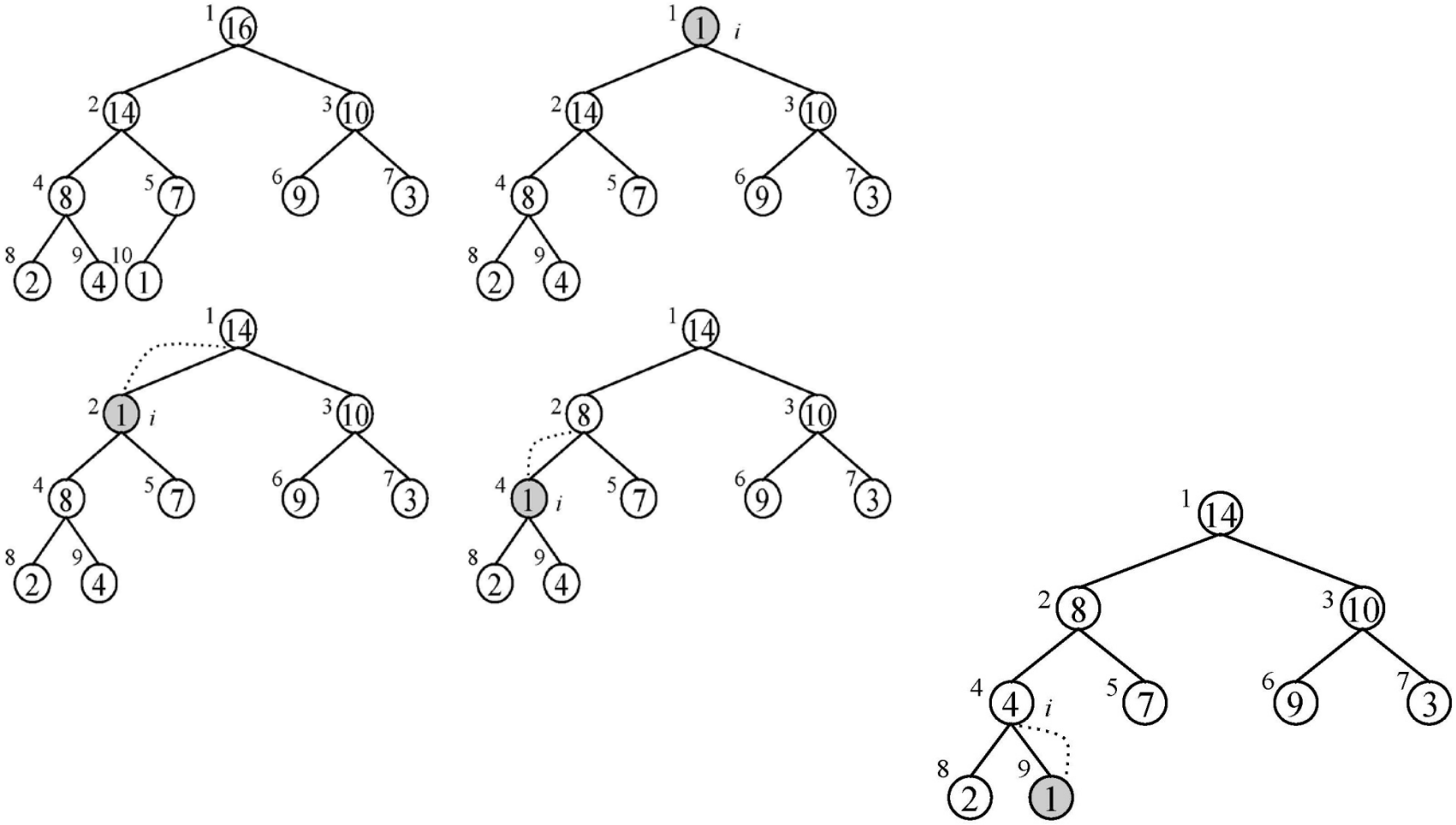


Altura	Número de nodos	Coste de cada llamada a **hundir**
h	1	h
$h-1$	2	$h-1$
$h-2$	4	$h-2$
$h-j$	2^j	$h-j$
1	2^{h-1}	1

$$\text{Coste total} = \sum_{j=1}^{h} j2^{(h-j)} = 2^h \sum_{j=1}^{h} j\left(\frac{1}{2}\right)^j \leq 2^h \sum_{j=1}^{\infty} j\left(\frac{1}{2}\right)^j \in O(2^h) = O(n)$$

$$\text{Nota: } \sum_{j=1}^{\infty} jr^j = \frac{r}{(1-r)^2} \quad \text{cuando } -1 < r < 1$$

☐ Eliminar_máximo. Ejemplo.

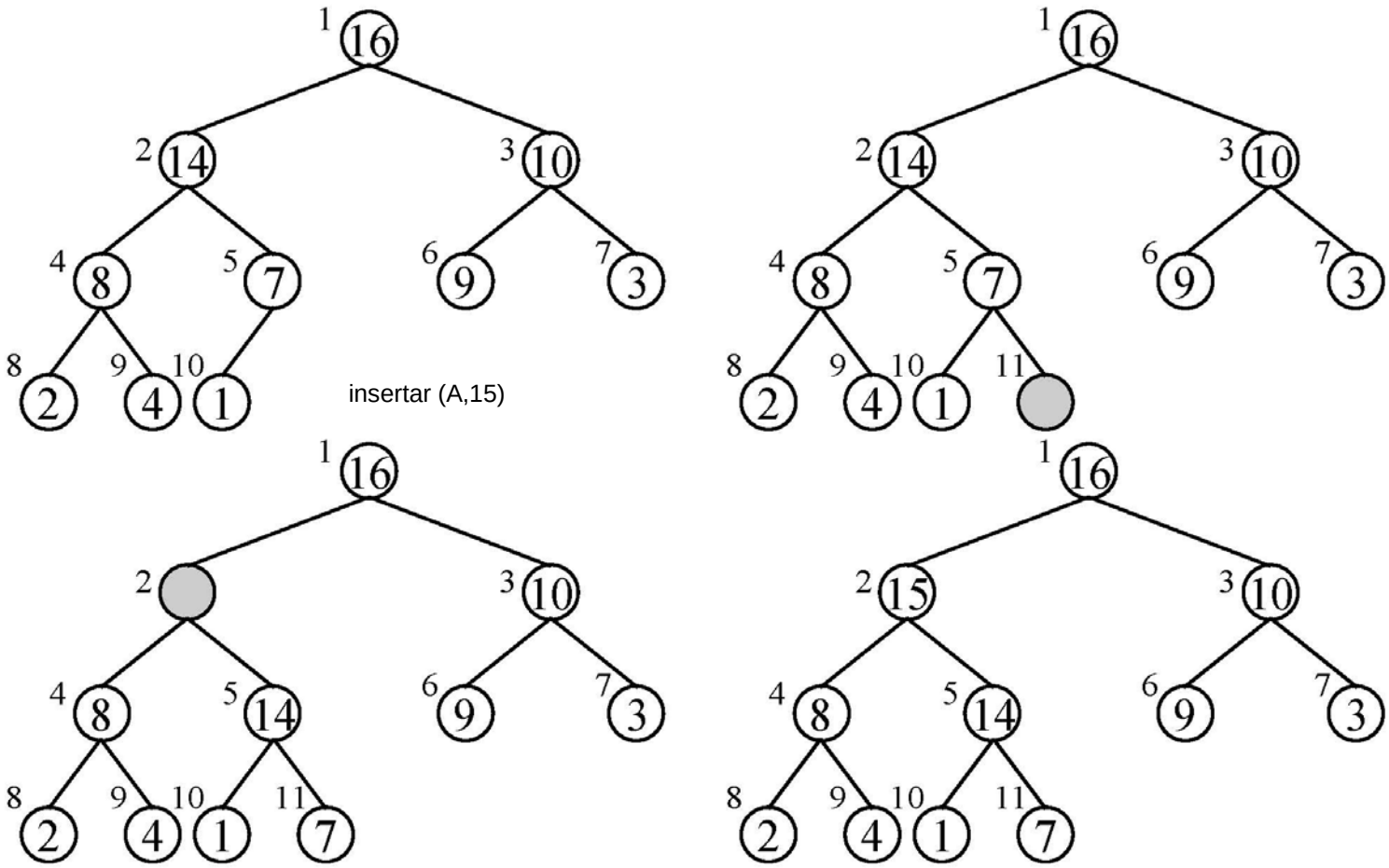


Eliminar_máximo. Algoritmo.

```
procedimiento eliminar_máximo(T[1..n])
 /* elimina el mayor elemento del montículo T[1..n] */
   T[1]←T[n]
   hundir(T[1..n-1],1)
fprocedimiento
```

Coste temporal $O(\log n)$, ya que realiza sólo un trabajo constante antes de llamar a **hundir** que tiene un coste $O(\log n)$.

- **Insertar. Ejemplo:** insertar (A,15)



Insertar. Algoritmo.

```
procedimiento añadir-nodo (T[1..n], v)
   T[n+1] ← v;
   flotar(T[1..n+1], n+1)
fprocedimiento
```

Flotar. Algoritmo.

```
procedimiento flotar(T[1..n],i)
   k ← i;
   repetir
      j ← k;
      si j > 1 y T[j ÷ 2] < T[k] entonces
         k ← j ÷ 2
         intercambiar T[j] y T[k]
         /* si j = k, entonces el nodo ha llegado a su posición final*/
   hasta que j = k
fprocedimiento
```

Coste Insertar.

Coste temporal $O(\log n)$, ya que el camino trazado de la nueva hoja a la raíz del heap tiene talla $O(\log n)$.

Estrategia Heapsort: Se insertan los elementos en un MaxHeap y se va eliminando sucesivamente el mayor para obtener los elementos en orden inverso.

```
procedimiento heapsort(T[1..n])
/* T es el vector que hay que ordenar*/
    Crear-montículo(T)
    para i ← n hasta 2 hacer
      hundir(T[1..i-1],1)
fprocedimiento
```

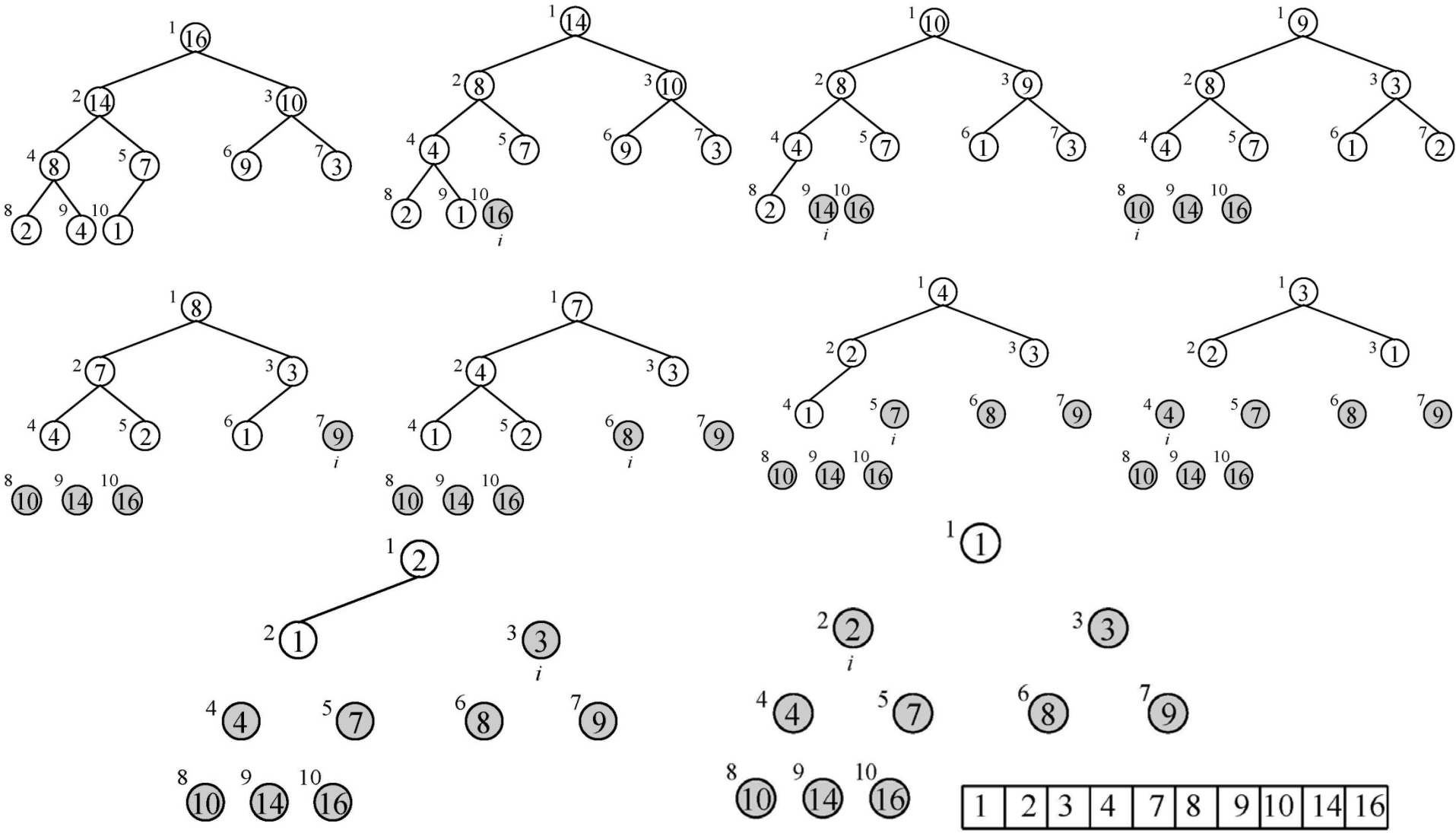
Complejidad Temporal: Crear-montículo + Coste del bucle(n-1 llamadas a hundir)

$$T(n) \in O(n) + (n-1)O(\log n) \in O(n \log n)$$

Propiedades de Heapsort.

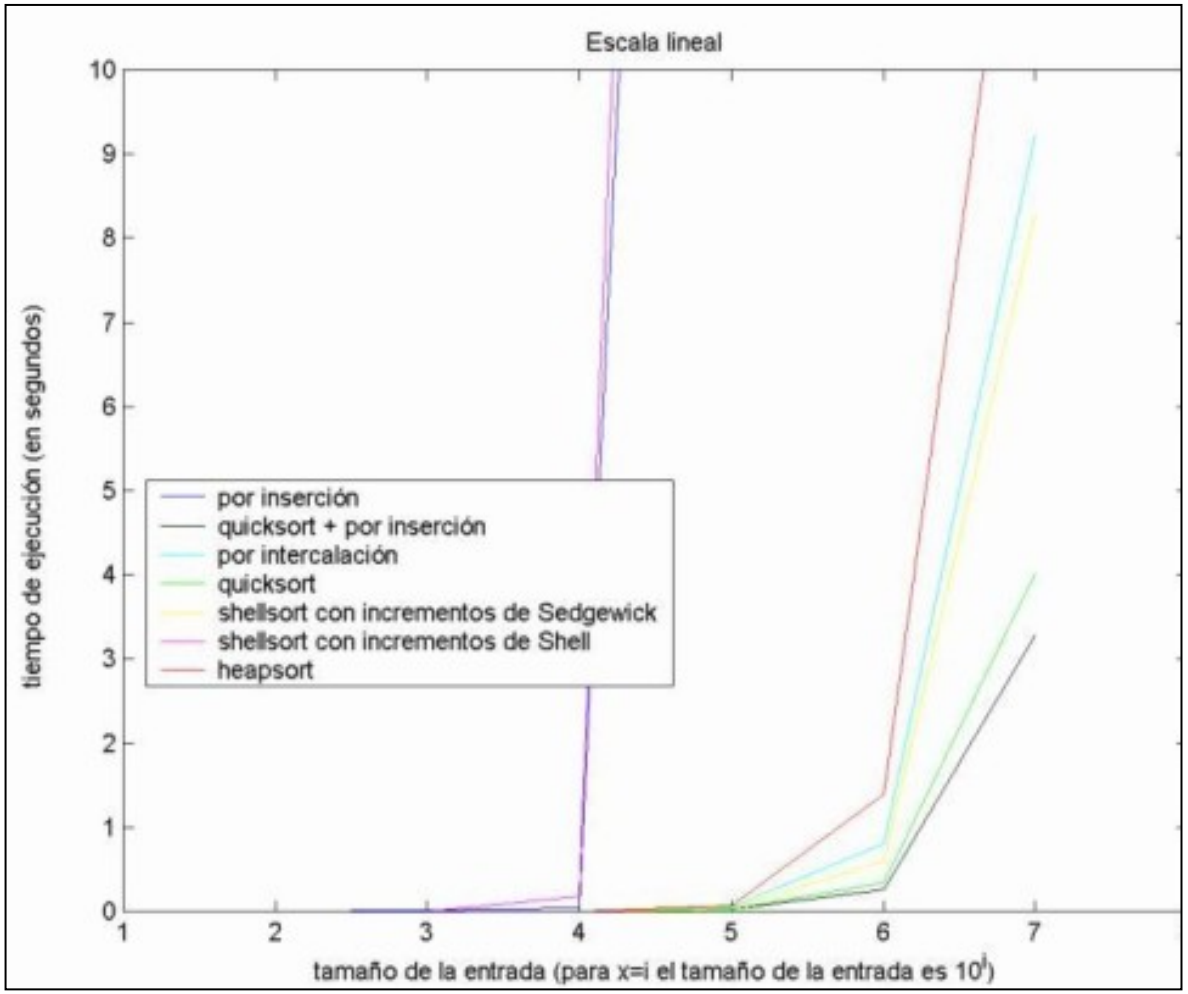
- "sort in-place" (como Quicksort; Mergesort necesita espacio adicional)
- Se garantiza el coste $O(n \log n)$, independientemente de la entrada → no hay peor caso (a diferencia de Quicksort). El "precio" que hay que pagar es que realiza más comparaciones que Quicksort, por lo que tiende a ser más lento en la práctica.

$$A = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$$

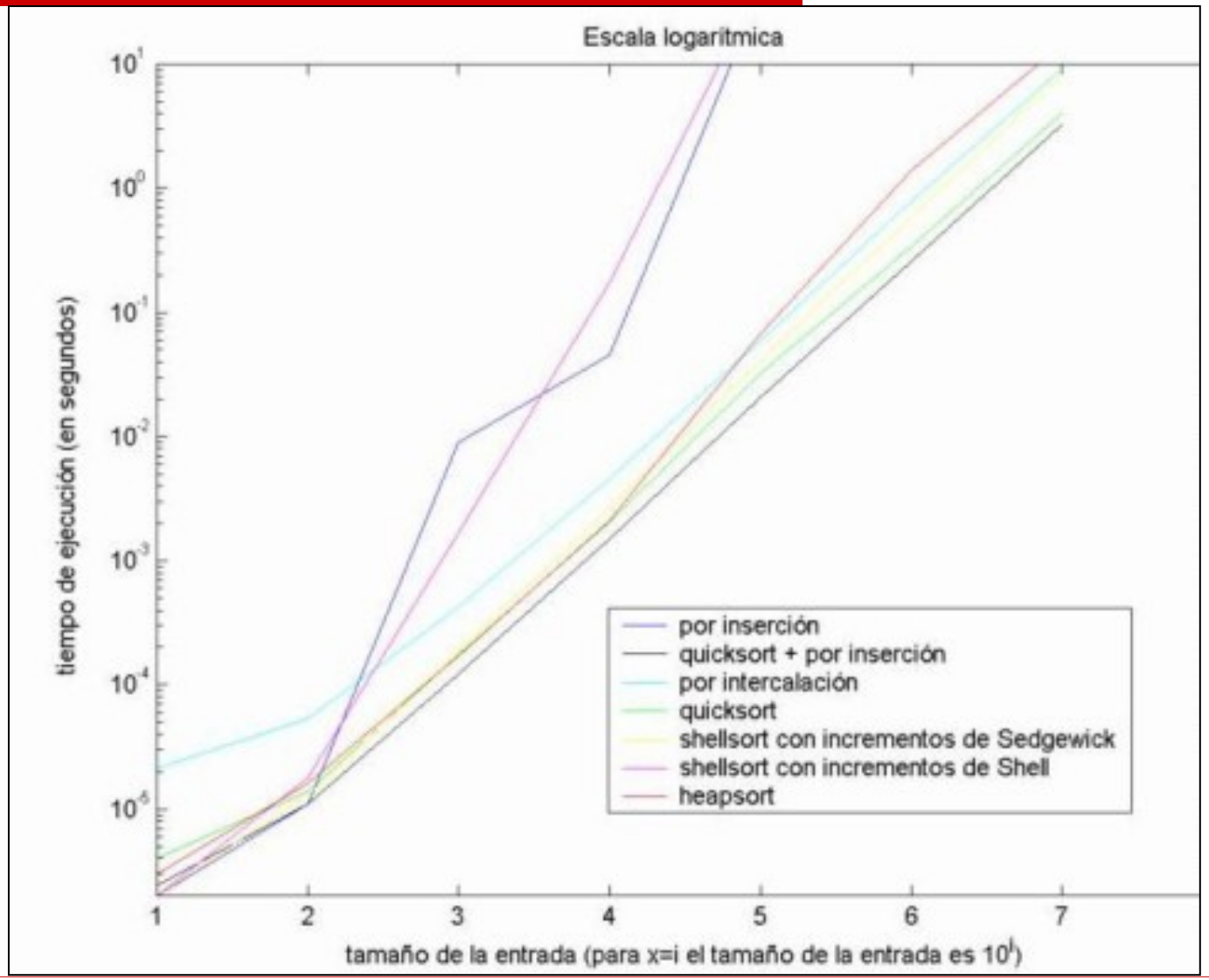


2. Algoritmos de ordenación. Comparación de métodos

A continuación se tabulan y grafican los tiempos de ejecución de algunos algoritmos vistos para entradas de tamaño 10, 100, 1000, ..., 10000000.



2. Algoritmos de ordenación. Comparación de métodos



3. Algoritmos de búsqueda.

- El problema de la búsqueda es un *problema de recuperación de la información lo más rápidamente posible* y consiste en *localizar un elemento* en una lista o secuencia de elementos.
- La operación de búsqueda puede llevarse a cabo sobre elementos ordenados o sobre elementos desordenados.
- **Búsqueda interna** si todos los elementos se encuentran en memoria principal (por ejemplo, almacenados en arrays, vectores o listas enlazadas)
- **Búsqueda externa** si los elementos se encuentran en memoria secundaria.
- Cada algoritmo de búsqueda trata de localizar en un array un elemento **X** (clave o *key*)**.** Una vez finalizada la búsqueda puede suceder:
  - que la búsqueda haya tenido éxito, habiendo localizado la posición donde estaba almacenado el elemento X, o
  - que la búsqueda no haya tenido éxito, concluyéndose que no existía ningún elemento X.

3. Algoritmos de búsqueda. Búsqueda lineal.

- Busca un dato (clave) *key* en una lista o array *V* accediendo a todas las posiciones hasta que se encuentre el elemento o se llegue al final del mismo (elemento no está)
- No impone restricciones sobre la lista o array *V*
- Tiene rendimiento en el caso peor: $O(n)$

- Busca un dato (clave) *key* en una lista o array *V*
- Requiere que la lista *V* este ordenada
- Suponiendo que la lista está almacenada como un *array*, donde los índices de la lista son left=0 y right = n-1 donde *n* es el número de elementos del *array*

```
/*************************************************************************
    Búsqueda Secuencial (lineal).
    Devuelve el índice del elemento buscado, o bien -1 en caso de fallo.
*************************************************************************/

template <typename T>
int secuential_search(T V[], int left, int right, T key)
{
  int i=left;
  while(i<=right && V[i]!=key){
    i++;
  }
  if(key==V[i]) return i;
  else return -1;
}
```

3. Algoritmos de búsqueda. Complejidad Búsqueda Lineal.

- **MEJOR CASO:** que la primera posición examinada contenga el elemento que buscamos, en cuyo caso el algoritmo informará que tuvo éxito después de una sola comparación. Por tanto, su complejidad será **O(1).**
- **PEOR CASO:** Sucede cuando encontramos X en la última posición del array. Como se requieren *n* ejecuciones del bucle mientras, la cantidad de tiempo es proporcional a la longitud del array ***n,*** más un cierto tiempo para realizar las condiciones del bucle mientras y para la llamada al método. Por lo tanto, la cantidad de tiempo es de la forma $\mathbf{a}n+b$ para ciertas constantes **a** *y* ***b.*** En notación O*,* $O(an+b) = O(an) = O(n)$*.*
- **CASO MEDIO:** Supongamos que cada elemento almacenado tiene la misma probabilidad de ser buscado. La media se puede calcular tomando el tiempo total de encontrar todos los elementos y dividiéndolo por *n:*
  - Total = a (1 + 2 + ...+n) + bn= a (n(n+1) / 2) + bn
  - Media = (Total / n) = a((n+1) / 2) + b que es $O(n)$**.**

3. Algoritmos de búsqueda. Búsqueda binaria. Implementación iterativa.

- Búsqueda en una **lista ordenada**: se sitúa la lectura en el centro de la lista y se comprueba si la clave coincide con el valor del elemento central. Si no se encuentra el valor de la clave, se sitúa en la mitad inferior o superior del elemento central de la lista. El algoritmo termina o bien porque se ha encontrado la clave o porque el valor del índice izquierdo excede al derecho y el algoritmo devuelve el indicador de fallo, -1 (búsqueda no encontrada).
- Tiene rendimiento logarítmico en el caso peor: $\mathbf{O(}\boldsymbol{log\ n}\mathbf{)}$
- Busca un dato (clave) *key* en una lista o array *V*
- Requiere que la lista *V* este ordenada
- Suponiendo que la lista está almacenada como un *array*, donde los índices de la lista son left=0 y right = n-1 donde *n* es el número de elementos del *array*

```
/*********************************************************************
    Búsqueda Binaria iterativa.
    Devuelve el índice del elemento buscado, o bien -1 en caso de fallo.
*********************************************************************/

template <typename T>
int binary_search(T V[], int left, int right, T key) {
  while(left<=right){
    int middle = (left+right) / 2;     /* índice del elemento central */
    if (key == V[middle])              /* compara el valor de la clave de búsqueda con el valor del índice central */
      return middle;                   /* encontrado, devuelve posición */
    else if (key < V[middle])
       right= middle-1;                /* ir a sublista inferior */
    else
      left = middle+1;                 /* ir a sublista superior */
    }
   return -1;                          /* elemento no encontrado */
}
```

3. Algoritmos de búsqueda. Búsqueda binaria. Implementación recursiva.

- Tiene rendimiento logarítmico en el caso peor: **O(*log n*)**

```
/*************************************************************************
    Búsqueda Binaria recursiva.
    Devuelve el índice del elemento buscado, o bien -1 en caso de fallo.
*************************************************************************/

template <typename T>
int binary_search(T V[], int left, int right, T key) {
  if (left > right)
      return -1;                                          /* elemento no encontrado */
  else {
       int middle = (left+right) / 2;                     /* índice del elemento central */
       if (key < V[pos])
         return(binarysearch(V, left, middle-1, key));    /* ir a sublista inferior */
       else if (key > V[pos])
         return(binarysearch(V, middle+1, right, key));   /* ir a sublista superior */
       else return middle;                                /* encontrado, devuelve posición */
    }
}
```

3. Algoritmos de búsqueda.Complejidad Búsqueda Binaria.

- **MEJOR CASO:** El *caso mejor* de la búsqueda binaria, se presenta cuando una coincidencia se encuentra en el punto central de la lista. En este caso la complejidad es $O(1)$ dado que sólo se realiza una prueba de comparación de igualdad.

- **PEOR CASO:** La complejidad del *caso peor* es $O(log_2 n)$ que se produce cuando el elemento no está en la lista o el elemento se encuentra en la última comparación. Se puede deducir intuitivamente esta complejidad. El caso peor se produce cuando se debe continuar la búsqueda y llegar a una sublista de longitud de 1. Cada iteración que falla debe continuar disminuyendo la longitud de la sublista por un factor de 2.

  El tamaño de las sublistas es: *n, n/2, n/4, n/8... 1*

  La división de sublistas requiere ***m*** iteraciones, en cada iteración el tamaño de la sublista se reduce a la mitad. La sucesión de tamaños de las sublistas hasta una sublista de longitud 1 es: $n, n/2, n/2^2, n/2^3, n/2^4 .... n/2^m$ siendo $n/2^m = 1 \Rightarrow n = 2^m$ Tomando logaritmos en base 2 en la expresión anterior quedará:

  $$m = log_2 n$$

  Por tanto la complejidad del caso peor es $O(log_2 n)$.

- Cada iteración requiere una operación de comparación : ***Total comparaciones*** $\approx 1 + log_2 n$

Tablas de dispersión (Hash)

- Estructura de datos especialmente diseñada para la implementación de DICCIONARIOS (operaciones Buscar, Insertar y Borrar en un tiempo "esperado" O(1)).
- Es una generalización de las tablas de acceso directo.
- Contenidos:
  1. Concepto de dispersión ("hashing")
  2. Resolución de colisiones:
     - Encadenamiento ("Separate Chaining")
     - Direccionamiento abierto ("Open addressing")
     - Costes.
  3. Funciones de dispersión.
  4. Tablas de dispersión dinámicas: Redispersion ("Rehashing")

1. Concepto de dispersión

- Las tablas de acceso directo son fáciles de implementar para representar conjuntos del universo { 0,1, ……, $|U|$-1 } puesto que nos sirven directamente para indexar un vector en C o C++.
- ¿Que podríamos hacer para representar conjuntos de un universo U diferente? Por ejemplo, ¿que pasa si U es un conjunto de cadenas de caracteres?
- ¿Que ocurre si | U | es mucho mayor que el tamaño del conjunto que queremos representar? Por ejemplo, el conjunto de todas las cadenas de caracteres hasta una longitud máxima crece de manera exponencial con esa longitud.
- **Idea**: Crear una función **h** : $U \rightarrow$ { 0,1, ……, m -1 } donde m es el tamaño del conjunto que queremos representar. Esta función se llama función ***hash*** o de ***dispersión***.
- **Ejemplo**: Queremos representar el conjunto de m = 5 nombres de ciudad {Valencia, Madrid, Sevilla, Zaragoza, Alicante } y asociarles un valor de tipo X.
  - Definimos un vector V de tipo X de tamaño m.
  - Una función que asocia a cada ciudad un valor 0,…,4. Por ejemplo, h(Valencia) = 3, h(Madrid) = 1, h(Sevilla) = 0, h(Zaragoza) = 4, h(Alicante) = 2.
  - Guardamos el valor de la ciudad c en $V[h(c)]$.

2. Concepto de colisión

- Queremos que **h** sea inyectiva ¿Que ocurre si no lo es?
- **Ejemplo**: Si h(Valencia) = h(Zaragoza) puede ocurrir lo siguiente:
  - Asociamos un valor a Valencia: V[h(Valencia)] = α
  - Asociamos un valor a Zaragoza: V[h(Zaragoza)] = β
  - Consultamos el valor de Valencia, tendrá que valer α pero vale β

Cuando dos elementos distintos van a parar mediante la función **h** a la misma posición en la tabla, se dice que se ha producido una **COLISION**.

- A veces no sabemos de *antemano* el conjunto que vamos a representar ni su tamaño. Nos gustaría que **h** fuese lo mas independiente posible del conjunto a representar.
  - Por ejemplo, si queremos una función que asocie un valor 0,1, ……, m -1 a *cualquier* cadena de caracteres, una función así no puede ser inyectiva (**|** U **|** > m).

Tabla de dispersión = función de dispersión + resolución de colisiones

2. Resolución de colisiones

- Dos estrategias diferentes para resolver las colisiones:
  - **Encadenamiento** (“Separate Chaining”):

    Una vez obtenido un índice 0,1, …, ***m*** -1 con la función de dispersión, se accede a un ***vector de listas*** donde se guarda en él todos los elementos que caigan en esa posición.
  - **Direccionamiento abierto** (“Open addressing”):

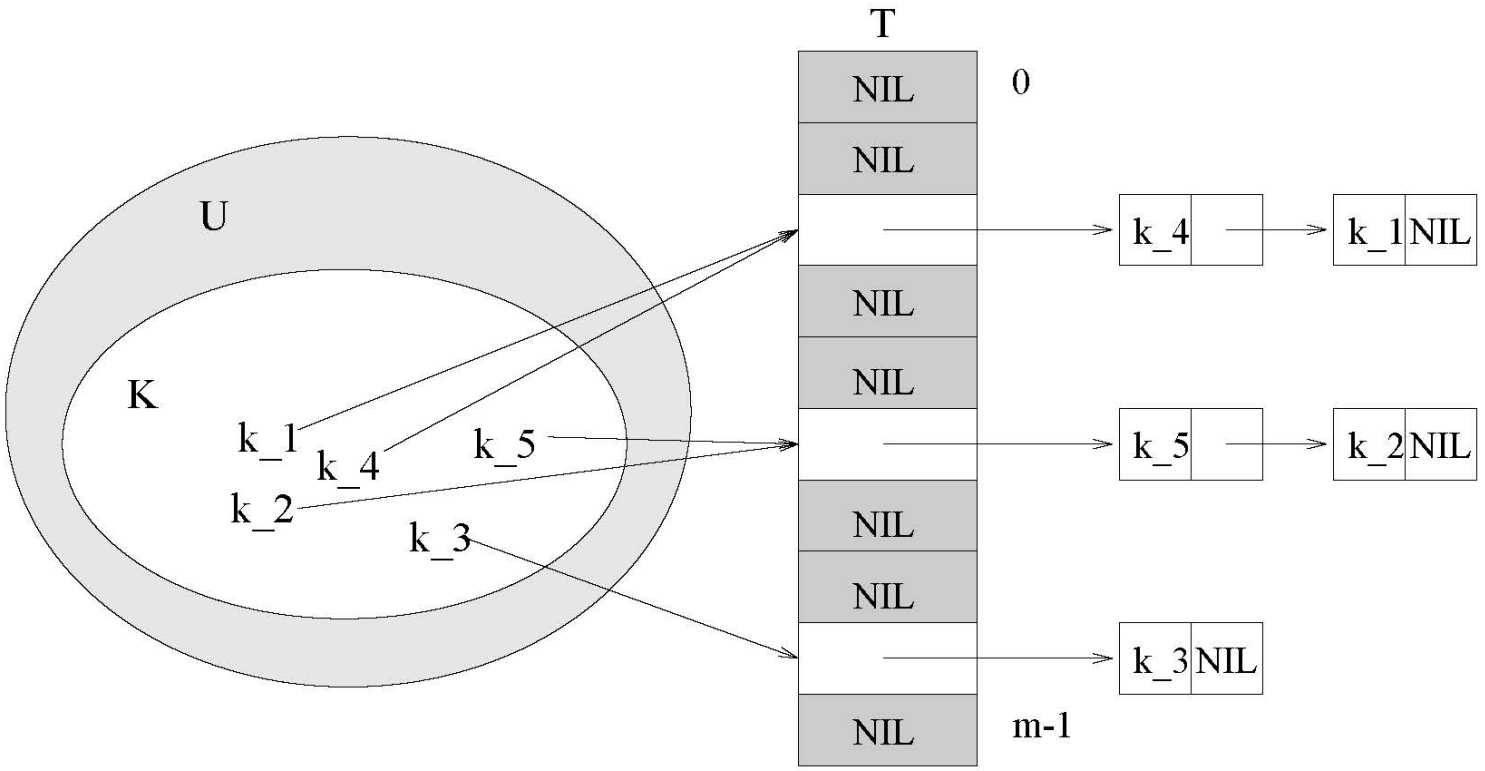
    Si se va a insertar un elemento en una posición y esa posición estuviese ocupada, se busca una posición alternativa utilizando una segunda, tercera,... función de dispersión. Es decir, en lugar de tener una función de dispersión h, se tiene un conjunto h_1, h_2, ….de funciones o mejor, una función de la forma $h(x, k) = h_k(x)$.

    Ejemplo: direccionamiento abierto lineal consiste en tener $h(x, k) = h'(x) + k$
- **Observación**: En cualquiera de los dos casos, ahora hay que comprobar si se produce o no colisión. Es decir, tenemos que guardar tanto la clave como el valor.

2. Resolución de colisiones por encadenamiento

- Todos los elementos que caen en la misma posición de la tabla (tienen el mismo valor de la función de dispersión) se ponen en una lista enlazada. Los elementos de la tabla son precisamente punteros a esas listas enlazadas.

  $T[j]$: puntero a la cabeza de una lista (lista de aquellos elementos cuya función hash es j)



2. Resolución de colisiones por encadenamiento.

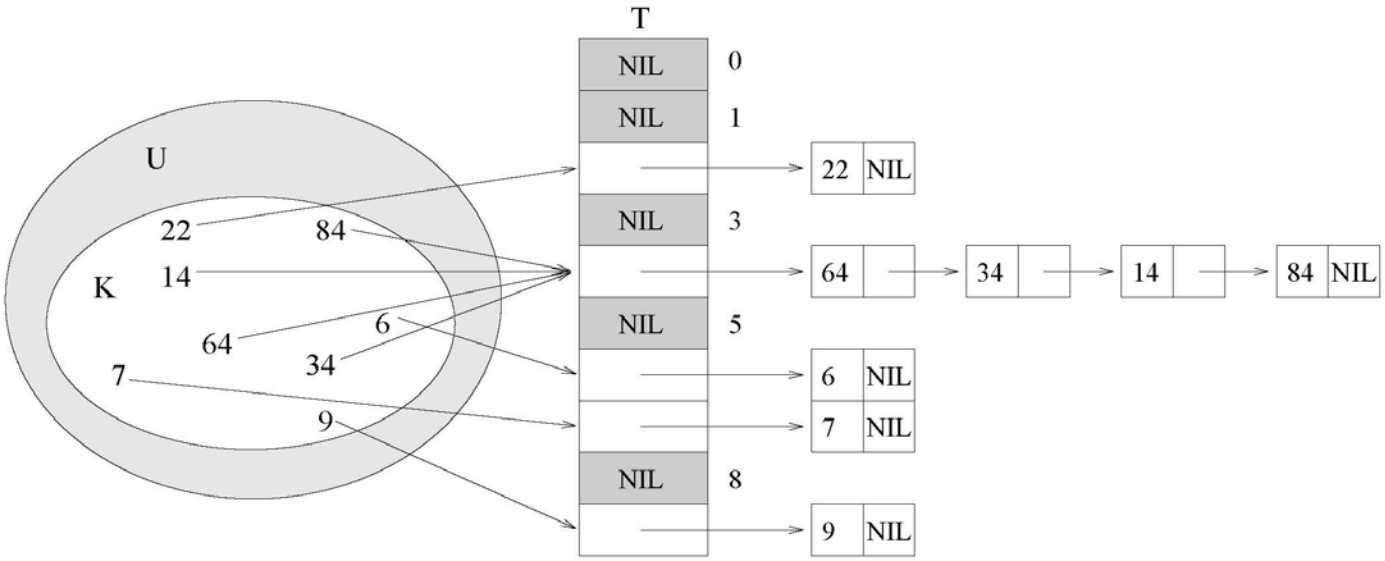
- Utiliza una tabla de punteros a listas enlazadas, en la que T[k] apunta a una lista con los datos tales que k = **h**(K)
- Realiza una búsqueda lineal en la lista de T[**h**(K)]

```
indice BusquedaPorHash(tabla_hash T, clave K){
    devolver BusquedaLineal(T[h(K)], K)}
```

- **Ejemplo.**

$S = \{84, 7, 22, 14, 34, 6, 64, 9\}$; tabla con 10 cubetas; $h(k) = k \text{ mód } 10$

$h(84) = 4, h(7) = 7, h(22) = 2, h(14) = 4, h(34) = 4, h(6) = 6, h(64) = 4, h(9) = 9$



2. Resolución de colisiones por encadenamiento. Análisis del coste de Buscar.

- Dada una tabla de m cubetas que almacena n elementos, se define el **factor de carga** de la tabla como $\alpha = n/m$ (numero medio de elementos almacenados en una lista)
- El **caso peor** de una **búsqueda** es: $O(n)$ (todos los elementos de están en la misma cubeta)
- Análisis del **coste medio**:
  - Asumimos "hashing uniforme simple": La probabilidad de que un elemento de U se almacene en cualquier posición de la tabla es la misma.
  - Asumimos que el coste de calcular h es $O(1)$
  - Suponemos que las colisiones se han resuelto por encadenamiento

    El coste de buscar es $O(1 + \alpha)$

    - Si n es proporcional a m, $n \in \Theta(m)$, $\alpha = n/m \Rightarrow O(m)/m \in \Theta(1)$
    - Y por lo tanto el coste de buscar es $O(1)$
    - Así, **todas las operaciones** del tipo **diccionario** (buscar, insertar y borrar) se pueden efectuar con **coste constante $O(1)$**

2. Resolución de colisiones por direccionamiento abierto
   - La resolución de una colisión en la posición T[**h**(K)] se intenta resolver explorando otras posiciones siguiendo un "sondeo" establecido
   - **Sondeo lineal**: al buscar K,
     - si T[**h**(K)] esta vacía, K no esta en T
     - si T[**h**(K)] tiene un dato k'≠K, K se busca en T[**h**(K)]+1
     - se repite lo anterior en T[**h**(K)]+2, T[**h**(K)]+3, T**h**(K)]+4,…
   - Sondeo cuadrático: al buscar k,
     - si T[**h**(K)] esta vacia, K no esta en T
     - si T[**h**(K)] tiene un dato K'≠K, k se busca en T[**h**(K)]+1^2
     - se repite lo anterior en T[**h**(K)]+2^2, T[**h**(K)]+3^2, T[**h**(K)]+4^2,…

Ejemplo. Direccionamiento a vacío

- Al intentar guardar el elemento con clave 1363 se comprobaría que está ocupado y se iría a la primera dirección libre (posición 45).
- Al intentar guardar la clave 1079 (dirección 120), se intentaría ir a la primera posición libre (121).
  - Como está fuera del espacio de almacenamiento se guardaría en la posición 1.

Clave	hash(clave)
1203	4
6754	35
32	33
8683	44
839	120
1363	44
1079	120

hash(clave) → clave **mod** 120 + 1

	Clave	Resto de campos
1	1079	
	-1	
4	1203	
	-1	
33	32	
	-1	
35	6754	
	-1	
	-1	
44	8683	
45	1363	
	-1	
	-1	
120	839	

3. Algoritmos de búsqueda. Búsqueda por hash. **Ejemplo. Direccionamiento a vacío.**

```
//Se desea buscar el registro con clave claveBuscada
dirección ←  hash(claveBuscada)
si v[dirección].clave < 0 entonces
      //El elemento no se encuentra
sino
      si v[dirección].clave <> claveBuscada entonces
            //Puede que se trate de un sinónimo
            //Se busca entre las posiciones siguientes hasta que se encuentra o hasta un hueco vacío
            repetir
                  dirección ←dirección mod numElementos + 1
            hasta (v[dirección].clave = claveBuscada) o v[dirección].clave < 0
      fsi
fsi
si v[dirección].clave = claveBuscada entonces
//El elemento se encuentra en la posición dirección
si_no
//El elemento no se encuentra
fin_si

int función hash(int clave)
inicio
      devolver(clave mod numElementos +1)
ffuncion
```

- **Complejidad de la búsqueda por hash**
  - n = numero de datos en T
  - m = numero de punteros en T (tamaño de T)
  - $\alpha = n/m$ = factor de carga
  - $A^e(n, m)$ = complejidad caso medio en búsqueda exitosa
  - $A^f(n, m)$ = complejidad caso medio en búsqueda fallida
- Con resolución de colisiones por encadenamiento
  - $A^e(n, m) = 1 + \alpha /2$
  - $A^f(n, m) = \alpha$
- Con resolución de colisiones por direccionamiento abierto
  - $A^e(n, m) = 1/\alpha \cdot \log(1/(1-\alpha))$
  - $A^f(n, m) = 1/(1-\alpha)$

3. La función de dispersión *h*

- **Método de la División**: $h(k) = k \mod m$
  - Conveniente: m numero primo no cercano a una potencia de 2.

  ■ ***Ejemplo***: se desea almacenar $n \approx 2000$ elementos. Podemos examinar una media de 3 elementos en una búsqueda

  ➢ Seleccionamos tamaño tabla: $\alpha = 3 = 2000/m$, $m = 666{,}66\ldots \rightarrow \boldsymbol{m} = 701$ (primo cercano a 666.66.. pero no cercano a una potencia de 2) $\Rightarrow$ **Función $h(k) = k \mod 701$**

    - Ejemplo: $h(1600) = 1600 \mod 701 = 198$

- **Método de la Multiplicación**: $h(k) = \lfloor m\ ((k\phi) \mod 1) \rfloor \quad 0 < \phi < 1$
  - Selección m no critica (m se suele tomar una **potencia de 2**, facilidad operaciones)
    - ➢ Cuando $\phi = (\sqrt{5} - 1)/2 \approx 0{,}618\ldots$ se denomina *función hash de Fibonacci*
    - ➢ *Nota*: $(\sqrt{5} - 1)/2$ es la inversa de la proporción aurea
    - ➢ *Nota*: $x \mod 1$ significa $x - \lfloor x \rfloor$

  ■ ***Ejemplo*:** se desea almacenar n ≈ 3000 elementos. Podemos examinar una media de 3 elementos en una búsqueda

  ■ Seleccionamos tamaño tabla: $\alpha = 3 = 3000/m$, $m = 1000 \rightarrow m = 1024 = 2^{10}$

  $\Rightarrow$ **Funcion $h(k) = \lfloor 1024\ (k \cdot 0{,}6180339887\ldots \mod 1) \rfloor$**

  - Ejem.: $h(1600) = \lfloor 1024\ (1600 \cdot 0{,}6180339887\ldots \mod 1) \rfloor = \lfloor 1024(988{,}85438\ 1\ldots \mod 1) \rfloor$

    $= \lfloor 1024 \cdot 0{,}85438\ 1 \rfloor = \lfloor 874{,}88716 \rfloor = 874$

3. Algoritmos de búsqueda. Búsqueda por hash.

- Ejemplo. Funciones de dispersión para cadenas

```
// Ejemplo: funciones de dispersión para cadenas
//*************************************************************
// función1--> método simple ("de la suma")
unsigned int tablaHash::funcion1(char *cadena) {
  unsigned int h=0;
  for (char* p=cadena; *p!="\0"; p++)
    h += (unsigned int)(*p);
  return(h % tamanyo);
}

//*************************************************************
// función2--> método "de la división"
unsigned int tablaHash::funcion2(char * x) {
  const int firstChar 32;   // Código del primer caracter a considerar (" ")
  const int rangChar 223 ;     // Número de caracteres a considerar
  const int base=rangChar;  unsigned int h=0;
  for (char* p=x; (*p)!="\0"; p++)
    h = (h*base + (unsigned int)(*p)-firstChar) % tamanyo;
  return(h);
}

//*************************************************************
// función3--> Método "multiplicativo"
unsigned int tablaHash::funcion3(char *cadena) {
  // cte_hash= 2^32/proporción_aúrea, proporcion_aurea = (sqrt(5)+1)/2
  const unsigned int cte_hash = 2654435769U;
  unsigned int h = 0;
  for (char* p=cadena; *p!="\0"; p++)        // cálculo implícito de módulo
    h = (h + (unsigned int)(*p)) * cte_hash; // 2^32, por desbordamiento
  return(h % tamanyo);
}
```

Consideraciones para tablas de dispersión

- Definir una función de dispersión que
  - Minimice el número de colisiones (dispersión efectiva).
  - Sea eficiente de calcular.
- Definir la estrategia para tratar las colisiones. Elegir:
  - Encadenamiento (“Separate Chaining”). Soporta mejor el borrado, mejor comportamiento cuando crece el factor de carga. Tener en cuenta el coste total de los punteros y la asignación de memoria dinámica.
  - Direccionamiento abierto (“Open addressing”). Borrado problemático. El factor de carga debe ser inferior a 1 (menos versatilidad, además tiene mal comportamiento temporal cuando el factor de carga se aproxima a 1).
- Elegir un tamaño de la tabla según necesidades (tamaño del conjunto, estrategia de resolución de colisiones, coste permitido) y compatible con la función de dispersión elegida.

Nota: Utilizaremos tablas de dispersión mediante **encadenamiento** a menos que se indique lo contrario.

4. Tablas de dispersión dinámicas. Redispersión ("Rehashing")

- Muchas veces no es fácil estimar el tamaño del conjunto a representar. Esto imposibilita la hipótesis de elegir un numero de cubetas ***m*** del mismo orden que el tamaño del conjunto ***n***, con lo que el coste de insertar, buscar y borrar puede crecer de manera no controlada.
- La **solución** consiste en aumentar el tamaño de la tabla de dispersión de manera dinámica. Cuando el factor de carga supere cierto umbral, se realiza una operación de redispersión ("**rehashing**") que consiste en crear otro vector de listas de mayor tamaño ***m'*** y mover los nodos de las listas a los lugares correspondientes en el nuevo vector.
  - El **coste** de esta operación es $\mathbf{O(}m + m' + n\mathbf{)}$

    (suponiendo que el coste de h es $O(1)$).
- Para que el coste amortizado de una operación de búsqueda en una tabla de dispersión con redispersión esté controlado, lo ideal es hacer que la tabla crezca de manera geométrica, por ejemplo $m' = 2m$.

4. Intercalación

- Tomar dos o más arrays de entrada ordenados y obtener un tercer array de salida también ordenado.
  - Se compara el primer elemento de los dos arrays de entrada.
  - Se selecciona el menor de ellos que será el primer elemento del array de salida.
  - Se avanza el índice del array del que se ha seleccionado el elemento.
  - El proceso se repite hasta que se termina alguno de los arrays.
  - Se vuelca el array que no se haya terminado al array de salida.
  - Complejidad $O(n)$

NOTA: similar al método merge estudiado en MergeSort.

4. Intercalación

```
procedimiento Intercalación(vector A,B;ref vector:C; int: m,n)
  var  //i será el índice de A, j el de B y k el de C
  entero : i,j,k
inicio
  i ← 1; j ← 1; k ← 1;
  //Mientras no se acabe alguno de los arrays de entrada
  mientras (i <= m) y (j <= n) hacer
    si A[i] < B[j] entonces
      //Se selecciona un elemento de A y se inserta en C
      C[k] ← A[i]
      //Se desplaza el índice del array A
      i ←  i + 1
    sino
      //Se selecciona un elemento de B y se inserta en C
      C[k] ← B[j]
      //Se desplaza el índice del array B
      j ←  j + 1
    fsi
    //Se desplaza el índice del array de salida
    k ← k + 1
  fmientras
  //Si se ha llegado al final del array B se vuelca todo
  el contenido que queda de A en el array de salida
  mientras i <= m hacer
    C[k] ←  A[i]
    i ← i + 1
    k ←  k + 1
  fmientras
  //Si se ha llegado al final del array A se vuelca todo
  el contenido que queda de B en el array de salida
  mientras j <= n hacer
    C[k] ← B[j]
    j ←  j + 1
    k ← k + 1
  fmientras
fprocedimiento
```