HASH

Un **conjunto** es una colección de objetos, los cuales no necesariamente tienen relación entre sí, como de orden, descendencia, etc.; tampoco están obligados a compartir atributos. En el área computacional, los objetos deben tener características comunes; y sus elementos pueden ser atómicos o estructurados.

Un **término atómico**, aplicado al concepto de tipo de dato; define a los elementos simples, indivisibles y enumerables.

El término **elemento estructurado** se hace referencia a los formados por más de un valor, que puede dividirse en varias componentes. Al trabajar con este tipo de elementos, se presupone que todos los miembros pertenecen a la misma clase y cada uno de ellos tiene un componente que lo distinguirá del resto de los elementos; ese componente se le conoce como **llave** o **atributo primario**.

Para la representación conjuntos de elementos estructurados, para realizar las diferentes operaciones, es mediante un almacenamiento secuencial, basándose en la llave, y así realizar búsqueda, o bien se puede pensar en ordenar mediante la llave para que así se aplique en su caso búsqueda binaria; este tipo de almacenamiento no es muy recomendable debido al tiempo que puede tenerse con la búsqueda, lo más recomendable es formar una **tabla**; la cual se logre realizar la localizar de un dato con un acceso directo y en una sola comparación de la llave.

Hash sirve para mapear todos los posibles valores de las llaves dentro del espacio de almacenamiento (**denominado tabla Hash**), Si en la mayoría de las ocasiones no se puede establecer una correspondencia uno a uno entre los valores de la llave y las posiciones de la tabla, la técnica hash convertirá cada valor de la llave en una dirección válida, aplicando a la llave una función de conversión:

$$f(llave) = posicion\_tabla(dirección\_base)$$

Las funciones utilizadas se le llaman **funciones hashing**. Si se logra diseñar una función que para cada uno de los posibles valores de la llave generará posiciones únicas dentro de la tabla, se estaría cumpliendo con el objetivo de esta técnica. A estas funciones se les llama **perfectas** aunque es prácticamente imposible obtenerlas. Esto quiere decir que la función puede ocasionar que se puede llegar a generar la misma posición en la tabla para dos llaves distintas, provocando una **colisión** que puede resolverse de otra forma.

En la creación de una función hashing se debe tener un diseño adecuado; a través del tiempo se han creado metodologías para su diseño y dependerá del tipo de llave, pero se espera que cumpla con:

- Ejecución rápida (conversiones sencilas).
- Distribución uniforme de las posiciones.
- Direccione todo el espacio de almacenamiento (ocupación de toda la tabla).

Además la función debe ser confiable, es decir, siempre debe generar la misma dirección para una misma llave en la tabla. Esto quiere decir que el cálculo de la dirección es totalmente independiente de la operación que se vaya a realizar sobre el elemento.

Las metodologías más empleadas de hash son:

- **Selección de dígitos:** consiste básicamente, en seleccionar algunos de los dígitos que conforman la llave y con ellos generar un índice para el espacio de almacenamiento. Por ejemplo:
  $H(d_1d_2d_3d_4d_5d_6d_7d_8d_9) = d_3d_5d_7$ donde $0 <= d_i <= 9$

- **Residuales:** Esta función consiste en utilizar como índice el residuo de dividir la llave con el tamaño máximo de almacenamiento. Por ejemplo:
  H(llave) = llave MOD n donde $0 <= H(llave) <= n - 1$

  Es una de las técnicas más utilizadas, pero se sugiere que el tamaño del espacio sea un número primo para evitar llaves con el mismo residuo.

- **Cuadrática:** Consiste en elevar al cuadrado el valor de llave y del resultado elegir los dígitos centrales. La cantidad de dígitos depende del rango de valores válidos para el índice. Se puede representar como:
  $H(llave) = digitos\_centrales(llave^2)$

- **Folding (plegamiento):** Consiste en agrupar algunos de los dígitos que conforman la llave y aplicarles algunas operaciones que permitan generar un índice para el espacio de almacenamiento. Por ejemplo
  $H(d_1d_2d_3d_4d_5) = d_1+d_2+d_3+d_4+d_5$ donde $0 <= d_i <= 9$ y $0 <= H <= 45$

Para solucionar las colisiones se tienen principalmente dos grupos:

- **Direccionamiento abierto:** Cuando un elemento provoca una colisión, se almacena en otra posición dentro del espacio, considerando dicho espacio como circular, esto se puede llevar a cabo de la siguiente manera:
  - Prueba Lineal: Buscar secuencialmente una posición disponible a partir de la dirección original generada para la función hashing.
    nueva_dir = dir_base + j, donde *j* es un contador del número de colisiones en esa búsqueda.
  - Prueba Cuadrática: funciona igual que la lineal pero de diferencia al calcular la nueva posición, evaluando el contador de colisiones elevado al cuadrado.
  - Prueba Aleatoria: Es una de las más difíciles de implementar, ya que sus desplazamientos son aleatorios a partir de la dirección base,
  - Doble hashing. Genera las nuevas posiciones por consultar a partir del propio valor de la

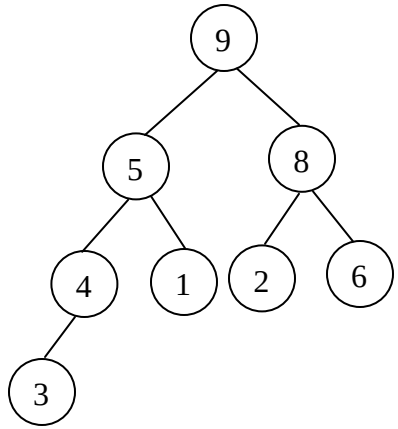
llave. Para lograrlo aplica una segunda función de hashing. La forma de determinarlo se aplica: *nueva_dir = dir_base + j * nuevo_hash(llave)* donde *dir_base* es la dirección original; *j* es el contador de colisiones para la búsqueda y *nuevo_hash(llave)* se comporta como constante para la llave en particular y determinar el tamaño del saldo.

Independientemente de qué prueba se aplique, éstos métodos necesitan manejar una bandera de estado, que indique sí la casilla esta utilizada o libre.

- Encadenamiento: Ligar todos los elementos cuyas llaves generan la misma dirección base. Esto puede hacerse dentro del mismo espacio de la tabla o con un una porción de espacio externa, esto puede hacerse de la siguiente manera:
  - Método de encadenamiento externo: cada elemento en la tabla hashing se considera como una cabeza a una lista que mantiene todos los elementos que colisionaron. Se puede representar estas listas de diferentes maneras; desde litas ligadas, pilas, listas ordenadas, y hasta árboles binarios.
  - Método de encadenamiento de coalisiones: Al igual que en el anterior, forma una lista de elementos colisionados; sin embargo, utiliza la misma tabla para almacenarlos. La más común de las representaciones es agregar un campo a la tabla (para guardar la dirección del siguiente elemento) y se divide en dos áreas, en el área normal y el área de colisiones.

HEAP

Puede considerarse como un árbol binario con ciertas restricciones, aunque con muchas ventajas. Un **heap** es un árbol binario casi completo. Se considera casi completo ya que está lleno en todos sus niveles excepto quizá el último que se va completando de izquierda a derecha. Otra diferencia es que un heap cada nodo tiene un valor menor igual (o mayor igual)que el valor asociado a cada uno de sus hijos, lo que se conoce como **condición heap**.



Para el diseño de un heap, se debe tomar en cuenta que se tiene la propiedad de ordenamiento, es decir, tener una relación de mayor que o menor que. Además de que se tiene una jerarquía. Se puede tener la siguiente especificación:

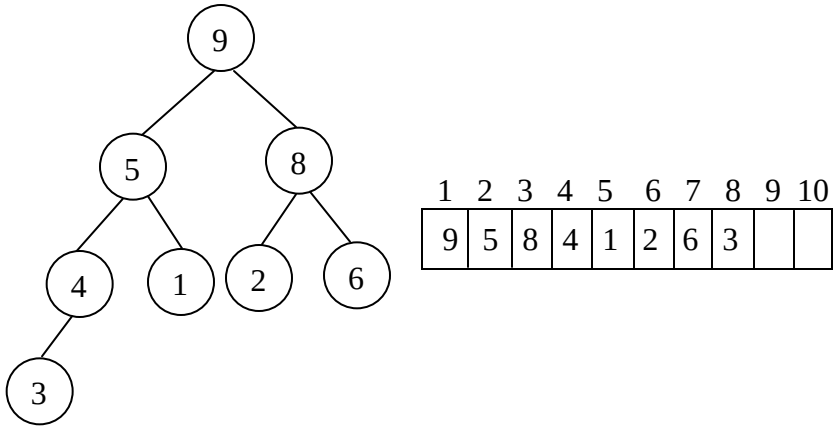
- **Elementos:** Los son los nodos.
- **Estructura:** un árbol heap posee una estructura jerárquica. Se tiene una relación de prioridad.
- **Operaciones:**
  - Crearheap
    Entrada: Ninguna.
    Salida: El árbol heap creado.
    Precondición: Ninguna.
    Postcondición: El árbol heap creado sin elementos.
  - Convierteheap
    Entrada: Una lista de N elementos.
    Salida: El árbol heap de N elementos creado con base en la lista dada.
    Precondición: Ninguna.
    Postcondición: El árbol heap de N elementos creado.
  - Insertar
    Entrada: Árbol heap sobre el que se realizará la inserción y el elemento a insertar.
    Salida: El árbol heap contiene un nuevo elemento.
    Precondición: El árbol heap existe.
    Postcondición: El árbol heap tiene un nuevo elemento colocado de acuerdo con su prioridad.
  - Borrar
    Entrada: Árbol heap sobre el que se realizará la baja.
    Salida: El árbol heap contiene un elemento menos (se elimina el de mayor prioridad).
    Precondición: El árbol heap existe.
    Postcondición: El árbol heap tiene un elemento menos.

Para su representación se puede realizar mediante un arreglo de elementos, por su sencillez. Si se considera que el primer elemento del arreglo (heap[1]) es la raíz, el segundo elemento almacena al hijo izquierdo y el tercero al derecho. Así se puede concluir que para el i-ésimo elemento del heap (heap[i]), sus hijos estarían en la posición: *2*i* y *2*i+1*.

Además, las condiciones heap, con respecto al orden de los valores se satisfacen:

heap[i] < heap[2*i] y heap[i] < heap[2*i+1]

o bien con la relación de >, dependiendo de la prioridad.



Con lo anterior se puede pensar que cualquier arreglo puede pasar a ser un heap. Siguiendo los siguientes pasos:

1. Todos los elementos del arreglo que corresponderían a las hojas del árbol cumplen con sus propias condiciones. Dado que es un árbol binario casi completo entonces se cumple que la cantidad de elementos que son hojas en el heap es N/2 o N/2+1, dependiendo si hay un número par o impar de elementos en el arreglo.

2. De aquí en adelante se agrega un nuevo elemento cada vez partiendo del último nodo del árbol que sí tiene hijos.

   Si se considera que un nodo tiene hijos, entonces, para que sea un heap, se debe cumplir con la condición de que un hijo siempre va a ser menor o mayor que su padre. Si no ocurre esto, entonces debe intercambiarse el valor del nodo padre con el valor más pequeño de sus hijos. Estos intercambios se realizan a través de toda la descendencia.

3. Se repite el paso anterior hasta que se incluyan todos los elementos del arreglo dentro del heap. En ese momento se considera que el arreglo ya se ha convertido en un heap.

Para un mismo conjunto se pueden tener diferentes árboles heap, dependiendo del ordenamiento inicial del arreglo.

Algoritmo general para construir un heap dada una lista

Sea n la cantidad de elementos.

1. apuntador = n div 2 (sólo se analizan los nodos que no son hojas).
2. Mientras apuntador >= 1
   Reacomodar dato señalado por apuntador (Rutina acomoda_abajo).
   apuntador = apuntador –1.

Algoritmo para la rutina acomoda_abajo

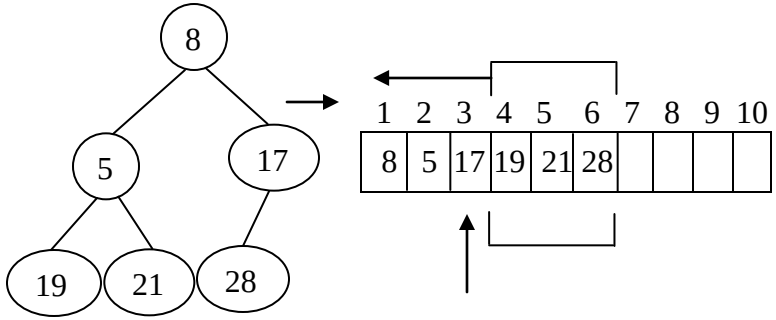
Sea AP el apuntador al elemento a acomodar.

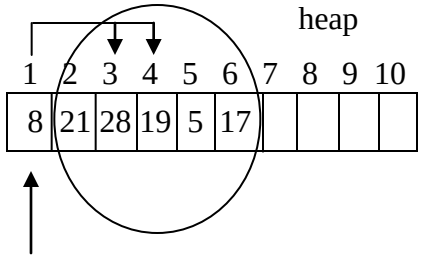
1. aux = AP
2. hijos = 2*AP (hijos sólo señala al hijjo izquierdo del nodo apuntado por AP).
3. Mientras haya hijos de aux (hijos <= maxlista) y alguno de ellos sea mayor
   Encontrar el hijo mayor de aux (hijomay)
   Si hijomay > aux entonces
   Intercambiar valores de aux e hijomay
   aux = hijomay
   hijos = 2*aux
   Sino
   Salir del ciclo

Ejemplo:

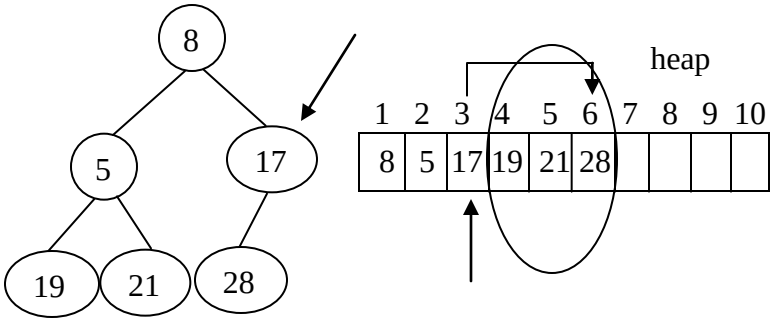
1. Localizar, dentro del arreglo los elementos que estarían como hojas del árbol heap, que se ubican en la segunda mitad del arreglo.
2. Tome cada uno de los elementos de la primera mitad del arreglo y comparar con los que representa a sus hijos en el arreglo. Si hay necesidad de cambiarlos, se intercambian al llegar al último nodo de su descendencia.

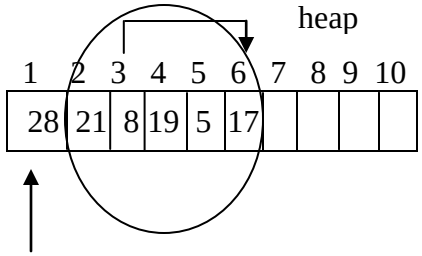
Paso 1. transformación

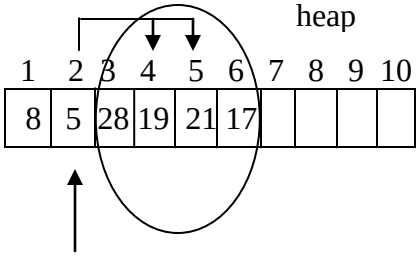




Paso 2. Intercambio







Finalmente