Análisis y Diseño de Algoritmos.

Sesión 7. 7 de Octubre de 2015.

Maestría en Sistemas Computacionales.

Por: Hugo Iván Piza Dávila.

¿Qué veremos hoy?

- Búsqueda Radix.
- Búsqueda por *Hashing*.
- Introducción a los Algoritmos voraces.
  - Algoritmo de la Mochila (KnapSack).

Búsqueda Radix

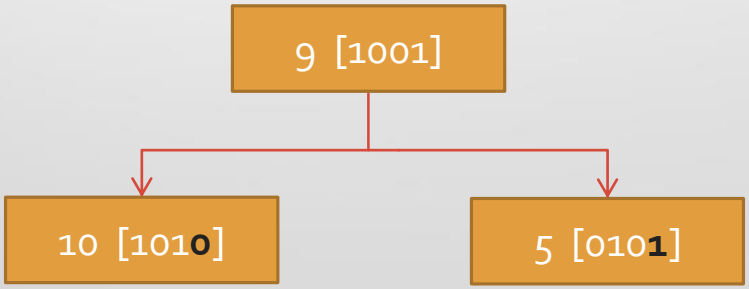
- Método que intenta combinar la simplicidad de los árboles binarios con la poca vulnerabilidad a caer en peores casos que tienen los árboles balanceados.
- En general, consiste en dividir al elemento en sus componentes pequeños.
  - Un número entero se puede dividir en bits.
  - Una cadena de texto se puede dividir en caracteres.
- La decisión de visitar al hijo izquierdo o derecho se toma en función del valor del componente actual.

Búsqueda Radix

- En este curso hemos trabajado con arreglos de números enteros: trabajaremos a nivel de bits.
- Construiremos un árbol digital de búsqueda.
  - Es un árbol binario porque cada componente de un elemento pertenece al alfabeto {0, 1}.
  - La decisión del camino a tomar depende del bit actual.
  - Atiende correctamente los peores casos del árbol binario pero no es invulnerable a otros peores casos.

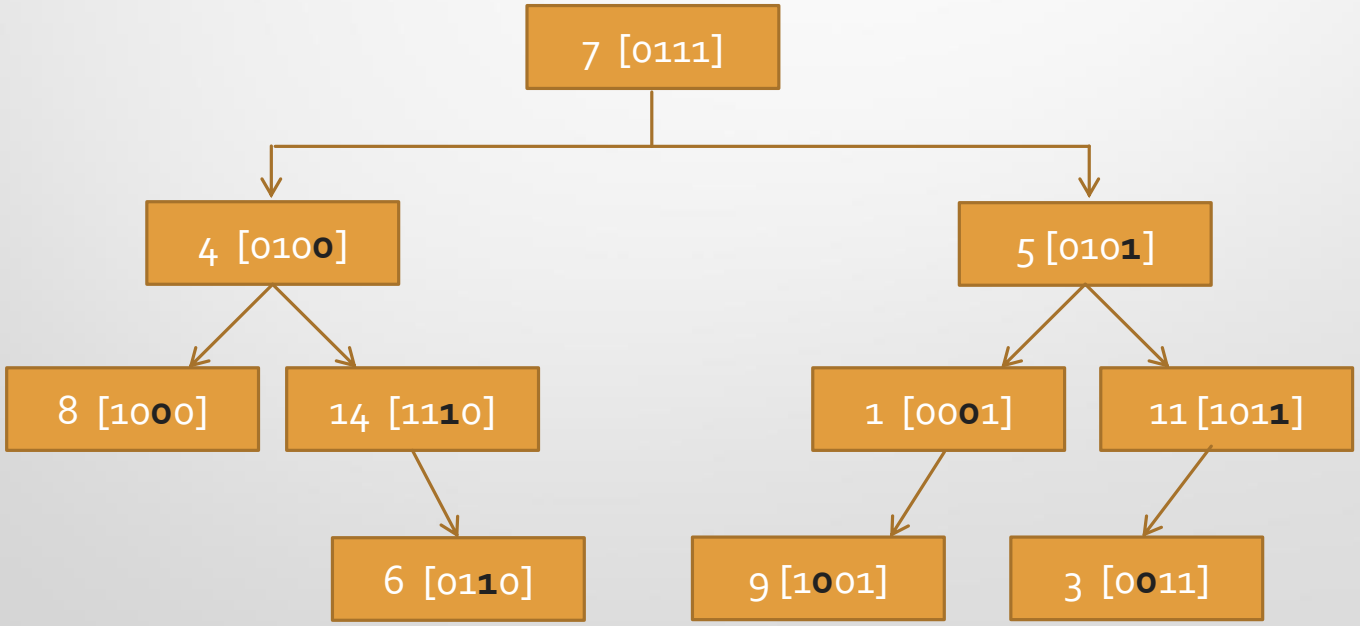
De una Lista a un Árbol Digital

- Cada nivel del árbol es una posición a nivel de bits del elemento.
- En la raíz se guarda el primer elemento de la lista.
- Si el primer bit del siguiente elemento es 0, la búsqueda continúa en el hijo izquierdo de la raíz; si no, en el hijo derecho.
- En el siguiente nivel del árbol, se realiza la misma comparación pero utilizando el siguiente bit.



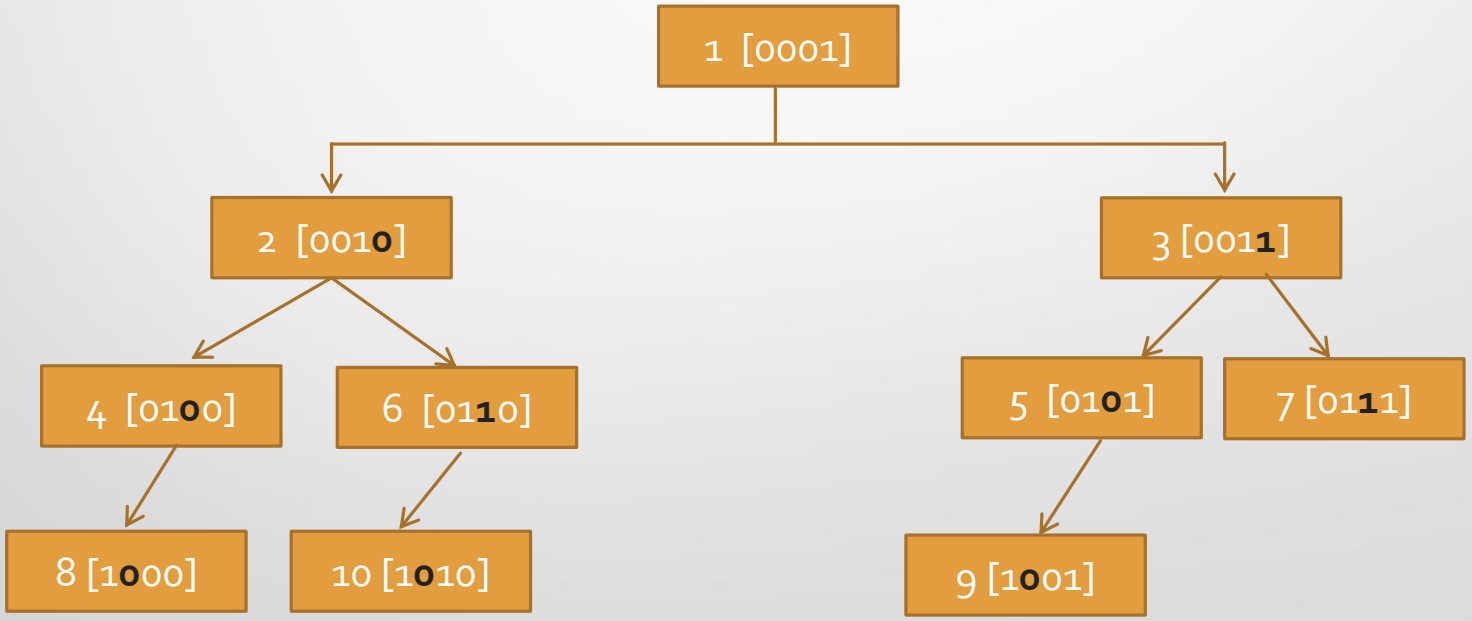
De una Lista a un Árbol Digital

- Lista = {7, 5, 11, 4, 14, 6, 8, 1, 9, 3}



¿Peor caso de los Árboles Binarios?

- Lista = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}



Ejercicios

1. Implementar método que devuelva el bit (`boolean`) dado un valor entero, y la posición del bit deseado.
2. Implementar un método no recursivo que construya el árbol digital a partir de un arreglo de enteros.
   - Devuelve el nodo raíz.
   - Intuya la estructura de datos a utilizar y el algoritmo.
3. Implementar método de búsqueda *Radix* sobre un árbol digital.
   - Puede ser recursivo.
   - Recibe el nodo raíz y el valor a buscar.
   - Devuelve el índice o -1 si no lo encontró.

Búsqueda por *Hashing*

- Escenario:
  - La lista no es de números enteros, sino de objetos.
  - Cada objeto se identifica por una clave alfanumérica.
- Los objetos se guardan en una estructura *hash*.
- Aspecto clave: encontrar un objeto en tal estructura debe tomar un tiempo casi constante.
- La posición del objeto en la estructura está dada por su código *hash.*

Código *Hash*

- Es un número entero no negativo.
- Se obtiene aplicando operaciones aritméticas con los caracteres o dígitos que componen la clave.
- Existen N combinaciones de caracteres o dígitos y existen normalmente M objetos, donde M es mucho menor que N.
- Ejemplo:
  - Si la clave fuera el número de placa (Ejemplo: JCY-8592).
  - Pueden existir $N = 26 \times 26 \times 26 \times 10{,}000$ claves diferentes (175m).
  - Pero la cantidad de vehículos en la ZMG no llega a los 2 millones.
  - $M = 2'000{,}000 << N = 175'000{,}000$.

Calculando un Código *Hash*

- Por simplicidad, supondremos que la placa guarda JCY.
- Una forma típica y **unívoca** de obtener un código *hash* es realizando una "conversión de sistema" alfabético a decimal:
  - De acuerdo a la posición de cada letra en la placa y en el alfabeto: A = 0, B = 1, C = 2, … Z = 25
  - *Hash*("JCY") $= 9 \times 26^2 + 2 \times 26^1 + 24 \times 26^0$

    $= 9 \times 676 + 52 + 24 = 6{,}160$
  - *Hash*("ZZZ") $= 25 \times 676 + 25 \times 26 + 25 = 17{,}575 = 26^3 - 1$
  - *Hash*("AAA") $= 0 \times 676 + 0 \times 26 + 0 = 0$

Método de *Horner*

- El método anterior tiene dos desventajas
  - El *cálculo de la potencia* en cada letra lo hace ineficiente.
  - Para claves más grandes, la suma puede generar un número tan grande que no cabe en un entero de 64 bits.
- La primer desventaja se resuelve con el método de *Horner*
- $F(x) = 2x^4 + 5x^3 + 6x^2 + 8x + 7$
  - Calculamos x^4 y luego calculamos x^3, ¿por qué no aprovechar el resultado de x^3 para obtener x^4? Para eso, **factorizamos**.
- *Horner*$(x) = x(x(x(2x + 5) + 6) + 8) + 7 \ldots q = \{2, 5, 6, 8, 7\}$
  - $h = q_0$
  - Desde $i = 1$ hasta $|q| - 1$: $h = hx + q_i$

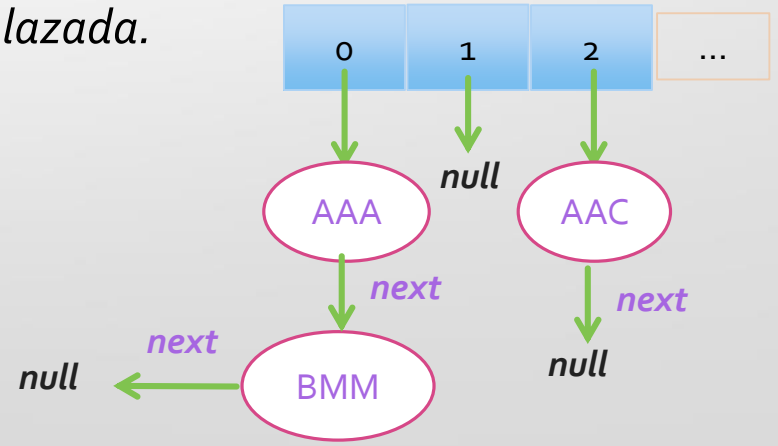
Aritmética modular

- Aun aplicando el módulo aritmético, para códigos grandes es posible que la fórmula de *Horner* devuelva números que superen la capacidad de almacenamiento (32b, 64b)
- Solución: mantener los resultados parciales de la fórmula de *Horner* siempre en el rango [0…M]
- Aritmética modular:
  - $(x + y) \% M = [(x \% M) + (y \% M)] \% M$.
  - $\therefore (x_1 + x_2 + \ldots + x_N) \% M = [x_1 \% M + x_2 \% M + \ldots + x_N \% M] \% M$.
  - Es decir, podemos calcular el módulo después de cada suma, dando el mismo resultado que si se calcula sólo al final.
  - Esta regla aplica también para restas y multiplicaciones.

Manejo de colisiones

- Las claves AAA, BMM tienen el mismo código hash:
  - *Hash*("AAA") = (0 × 676 + 0 × 26 + 0) % 1000 = 0 % 1000 = **0**
  - *Hash*("BMM") = (1 × 676 + 12 × 26 + 12) % 1000 = 1000 % 1000 = **0**
- A las dos claves les corresponde la misma posición en la estructura *hash*.

- Por tanto, cada posición puede almacenar varios objetos.
  - Cada posición es una *lista enlazada.*



Ejercicio

- Búsquedas *hash* en un archivo de clientes
  - Descargar los archivos `Clientes.txt` y `HashSearch.java`
  - Existe una clase `Customer` y un método de lectura del archivo que devuelve un arreglo de objetos `Customer`.
- El RFC es la clave de cada cliente
  - Considere al RFC como una secuencia de 12 ó 13 caracteres compuestos por letras mayúsculas y dígitos.
  - Nótese que se admiten 36 caracteres diferentes.
- Implementar un método `int hashcode(String rfc)` utilizando el método de Horner y aritmética modular.
  - M es el tamaño de la lista de clientes.

Ejercicio

- Crear una clase `CustomerNode` que almacene un objeto `Customer` y un apuntador al siguiente nodo.
- Implementar un método `createCustomerHashList` que reciba el arreglo de clientes y devuelva un arreglo (del mismo tamaño) de objetos de tipo `CustomerNode`.
  - En cada posición del arreglo devuelto se almacenará(n) el(os) cliente(s) cuyo hashcode sea igual a tal posición.
- Implementar un método `indexOf` que reciba el arreglo de `CustomerNode` (estructura *hash*) y el RFC a buscar, y devuelva el índice del cliente con tal RFC. Si no lo encontró, que devuelva -1.