Analysis and Design of Algorithms

Session 7.

Maestría en Sistemas Computacionales.

Luis Fernando Gutiérrez Preciado.

What will we see today?

- Radix Search
- Hashing Search

Radix Search

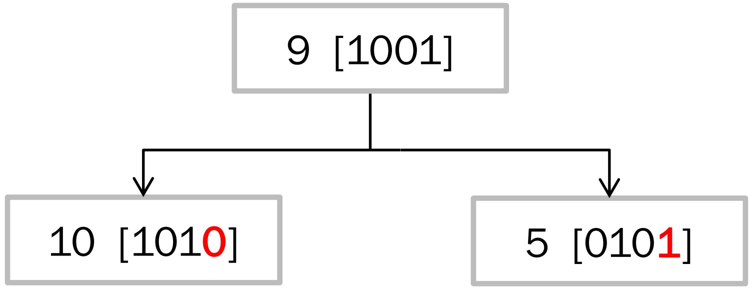
- A method that attempts to combine the simplicity of binary trees with the low vulnerability to worst-case scenarios that balanced trees have.
- In general, it involves breaking down the element into its smaller components.
  - An integer can be broken down into bits.
  - A text string can be broken down into characters.
- The decision to visit the left or right child is based on the value of the current component.

Radix Search

- In this course, we have worked with arrays of integers: we will work at the bit level.
- We will build a digital search tree (DST).
  - It is a binary tree because each component of an element belongs to the alphabet {0, 1}.
  - The decision of the path to take depends on the current bit.
  - It effectively handles worst-case scenarios for binary trees but is not immune to other worst-case scenarios.

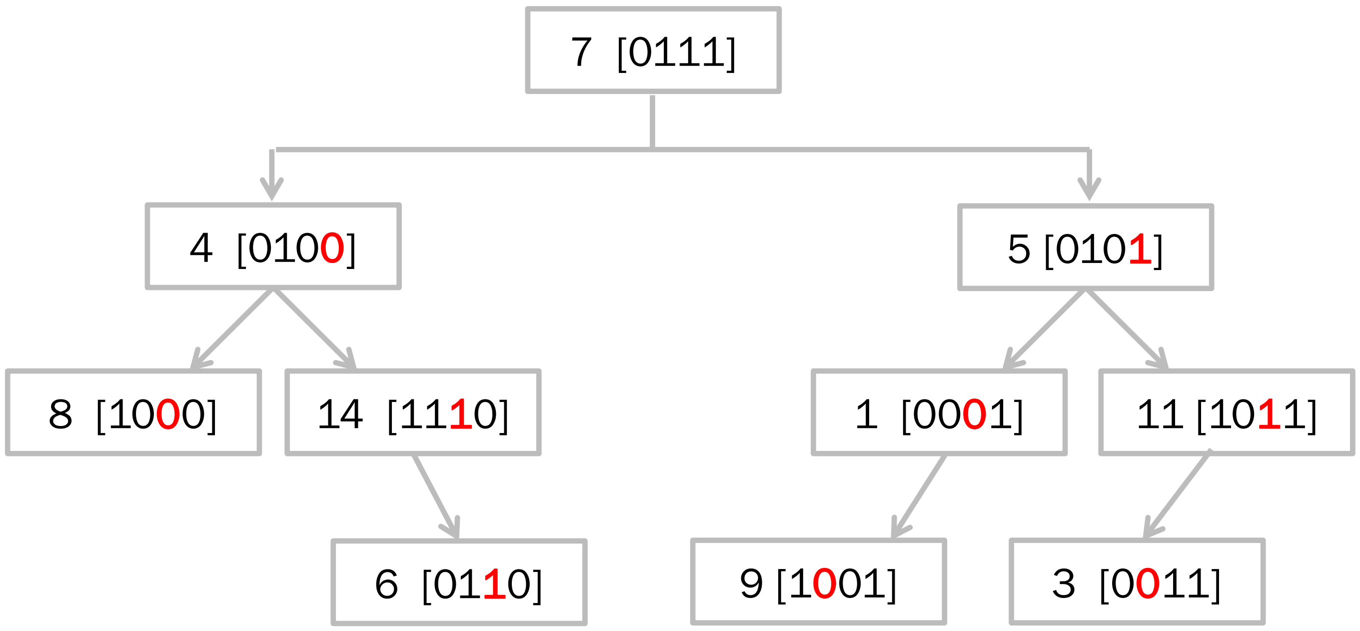
From a List to a Digital Tree

- Each level of the tree represents a bit-level position of the element.
- The root stores the first element of the list.
- If the first bit of the next element is 0, the search continues in the left child of the root; otherwise, in the right child.
- At the next level of the tree, the same comparison is made but using the next bit.



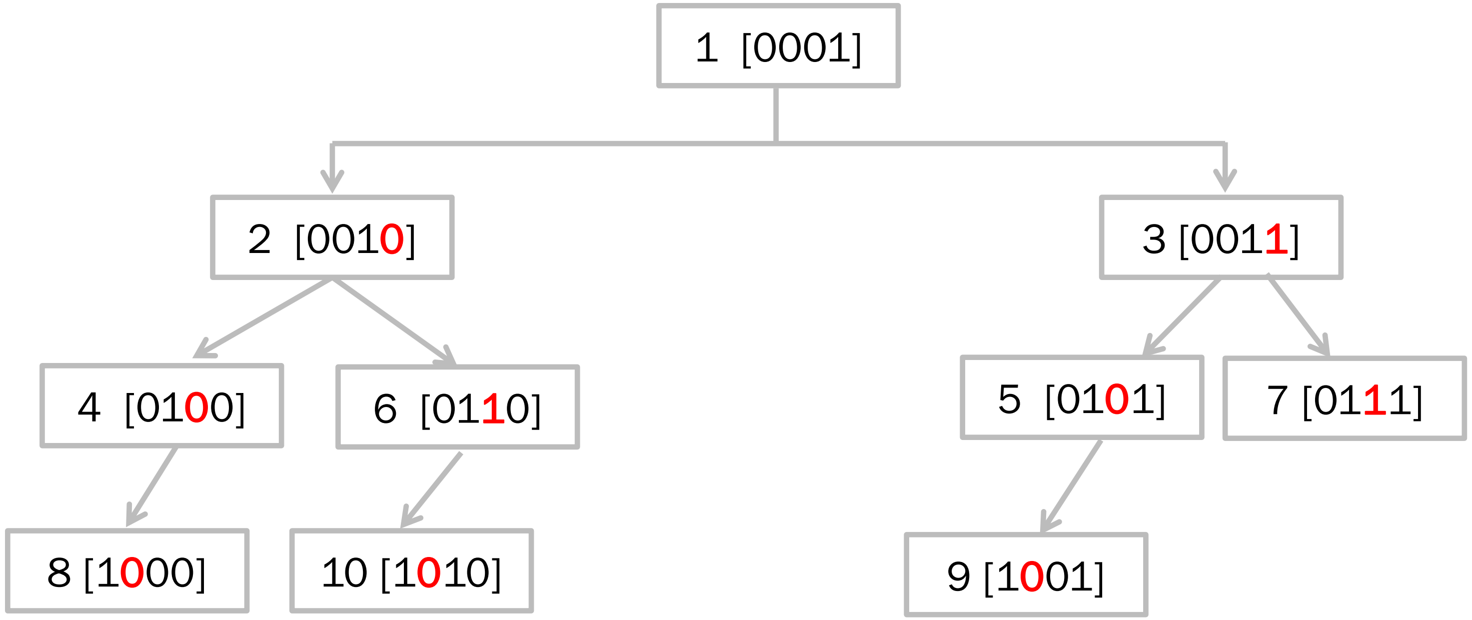
From a List to a Digital Tree

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



Worst Case of Digital Trees?

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



Exercises

1. Implement a method that returns the bit given an integer value and the desired bit position.
2. Implement a non-recursive method that constructs the digital tree from an array of integers.
    1. Return the root node.
    2. Guess the data structure to use and the algorithm.
3. Implement the Radix Search method on a digital tree.
    1. It can be recursive.
    2. Receive the root node and the value to search for.
    3. Return the index or -1 if not found.

Hashing Search

- Scenario:
  - The list is not of integers but of objects.
  - Each object is identified by an alphanumeric key.
- The objects are stored in a hash structure.
- Key aspect: finding an object in such a structure should take nearly constant time.
- The object's position in the structure is determined by its *hash* code.

Hash Code

- It is a non-negative integer.
- It is obtained by applying arithmetic operations to the characters or digits that make up the key.
- There are N combinations of characters or digits, and there are normally M objects, where M is much smaller than N.
- Example:
  - If the key were the car plate number (Example: JCY-8592).
  - There can be N = 26 × 26 × 26 × 10,000 different keys (175m).
  - But the number of vehicles in the ZMG does not exceed 5 million.
  - M = 5,000,000 << N ≈175,000,000.

Computing the *Hash code*

- For simplicity, let's assume the plate reads JCY.
- A typical and unique way to obtain a hash code is by performing an alphabetical-to-decimal system conversion:
- According to the position of each letter in the plate and in the alphabet: 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")

$$= 0 \times 676 + 0 \times 26 + 0 = 0$$

Horner's Method

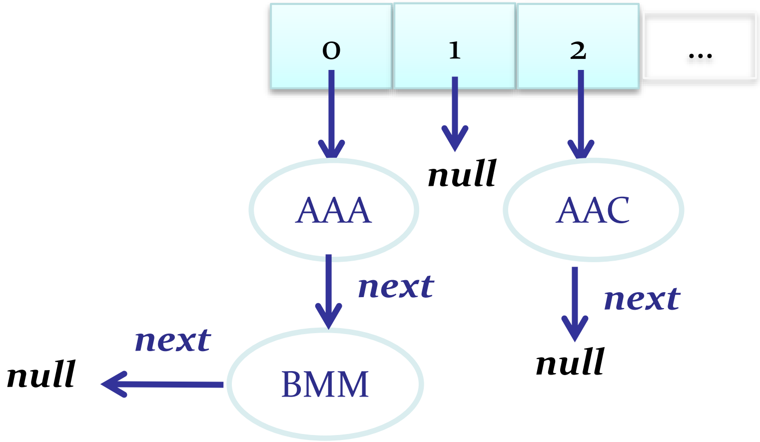
- The previous method has two disadvantages.
  - Calculating the power in each letter makes it inefficient.
  - For larger keys, the sum can generate a number so large that it doesn't fit into a 64-bit integer.
- The first disadvantage is resolved with Horner's method.
- $F(x) = 2x^4 + 5x^3 + 6x^2 + 8x + 7$
  - We calculate x^4 and then calculate x^3. Why not use the result of x^3 to obtain x^4? For that, we **factorize**.
  - $a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0 = (((a_4x + a_3) x + a_2) x + a_1) x + a_0$
- *Horner*(x) = $x(x(x(2x + 5) + 6) + 8) + 7$ … $q = \{2, 5, 6, 8, 7\}$
  - $h = q_0$
  - From $i = 1$ to $|q| - 1$:
    - $h = hx + q_i$

Modular Arithmetic

- Even with modular arithmetic, for large codes, Horner's formula may return numbers that exceed storage capacity (32b, 64b).
- Solution: keep partial results of the Horner's formula always in the range [0…M].
- Modular arithmetic:
  - $(x + y) \% M = [(x \% M) + (y \% M)] \% M$.
  - $\therefore (x_1 + x_2 + \ldots + x_N) \% M = [x_1 \% M + x_2 \% M + \ldots + x_N \% M] \% M$.
  - In other words, we can calculate the modulus after each addition, giving the same result as calculating it only at the end.
  - This rule also applies to subtractions and multiplications.

Handling Collisions

- Keys AAA and BMM have the same hash code:
  - *Hash*(“AAA”) = $(0 \times 676 + 0 \times 26 + 0)$ % 1000 = 0 % 1000 = **0**
  - *Hash*(“BMM”) = $(1 \times 676 + 12 \times 26 + 12)$ % 1000 = 1000 % 1000 = **0**
- Both keys correspond to the same position in the hash structure.
- Therefore, each position can store multiple objects.
  - Each position is a linked list.



Exercise

- Hash searches in a customer file
  - Download the files `Clientes.txt` and `HashSearch.py`
  - There is a `Customer` class and a file reading method that returns an array of `Customer` objects.
- The RFC is the key for each customer.
  - Consider the RFC as a sequence of 12 or 13 characters composed of uppercase letters and digits.
  - Note that 36 different characters are allowed.
- Implement a hashcode(String rfc) method using Horner's method and modular arithmetic. M is the size of the customer list.

Ejercicio

- Create a `CustomerNode` class that stores a `Customer` object and a pointer to the next node.
- Implement a `createCustomerHashList` method that receives the array of customers and returns an array (of the same size) of `CustomerNode` objects.
  - At each position of the returned array, the customer(s) whose hash code is equal to that position will be stored.
- Implement a `searchRFC` method that receives the array of CustomerNode (hash structure) and the RFC to search for and returns the index of the customer with that RFC. If not found, return -1.

Complexity

- Advantage:
  - Search: constant on average O(1)
- Disadvantage:
  - Does not store values in order, is costly if information needs to be displayed in order. In these cases, balanced trees are better.

Conclusions

- Implement binary radix tree
- Hash code
  - Horner's method
  - Modular arithmetic
- Implement hash table
  - Efficient for large amounts of information
  - Search for unordered information