Analysis and Design of Algorithms

Session 5.

Maestría en Sistemas Computacionales.

Luis Fernando Gutiérrez Preciado.

What will we cover today??

- Elimination of recursion: 3 general forms.
  - Applied to binary search and *mergesort.*
- Binary Tree Search

Recursion

- Recursion provides greater expressiveness to a software solution (more elegant and clear).
  - "Repeat the same instructions, but now with these data."
  - For the reader, it's often easier to understand the functionality of a recursive algorithm than an iterative one.
  - It resembles the mathematical definition of functions:

$$\text{Factorial}(n) \begin{cases} 1, & \text{si } n = 0 \\ n \times \text{Factorial}(n-1) & \text{si } n > 0 \end{cases}$$

Recursion

- However, it has disadvantages:

1. Insufficient memory: each function call is a pointer that is stored in the program's stack.
   - Recursion often leads to many pending functions waiting to finish, resulting in stack overflow.
   - This happens in Quicksort with sorted arrays.
2. Some languages do not support recursion, especially in embedded systems.

Eliminating Recursion

- Use recursion only when necessary (when there is no simple iterative solution) or as the first version of an algorithm.
- For many recursive algorithms, there is an equivalent iterative version.
- Before converting a recursive algorithm to its iterative equivalent, let's identify 3 general forms of recursion:

1. A recursive call is made after operations on the search space. There may have been multiple possible calls, but one was selected.
2. Two (or more) recursive calls are made after the operations.
3. Two (or more) recursive calls are made before the operations. In the initial calls, operations are not performed; they are queued.

General Form 1

- *type* **recursiveFunction** *(input space, parameters)*
  1. *Can recursion terminate successfully or with failure? Return a value or exit.*
  2. *Operations on the input space (if applicable).*
  3. *Modify parameters (for each possible case).*
  4. *Call the* **recursiveFunction** *(input space, new parameters).*
- *type* ***IterativeFunction*** *(input space, parameters)*
  - *Repeat while step 1 is false:*
    1. *Can I terminate successfully or with failure?*
    2. *Operations on the input space (if applicable).*
    3. *Modify parameters (for each possible case).*

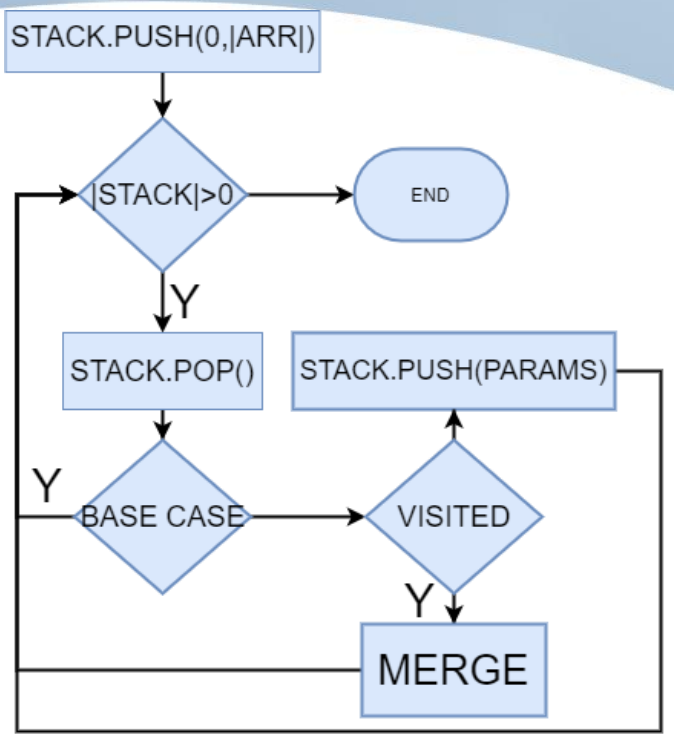
Exercise

- Implement General Form 1 with Binary Search to convert it into an iterative algorithm.
  - Consider that "left" and "right" are inclusive.

General Form 2

- *type* **recursiveFunction** *(input space, parameters)*
  1. *Can recursion terminate successfully or with failure?*
  2. *Operations on the input space.*
  3. *Call* **recursiveFunction** *(input space, parameters1).*
  4. *Call* **recursiveFunction** *(input space, parameters2).*
- *type* **iterativeFunction** *(input space, parameters)*
  1. *Create a parameter* ***stack*** *and deposit the received parameters.*
  2. *While the stack is not empty:*
     a) *Retrieve the last deposited parameters from the stack (in reverse order).*
     b) *Can I proceed to the next iteration without adding parameters? (similar to recursion termination)*
     c) *Operations on the input space.*
     d) *Add parameters2, parameters1 to the stack.*

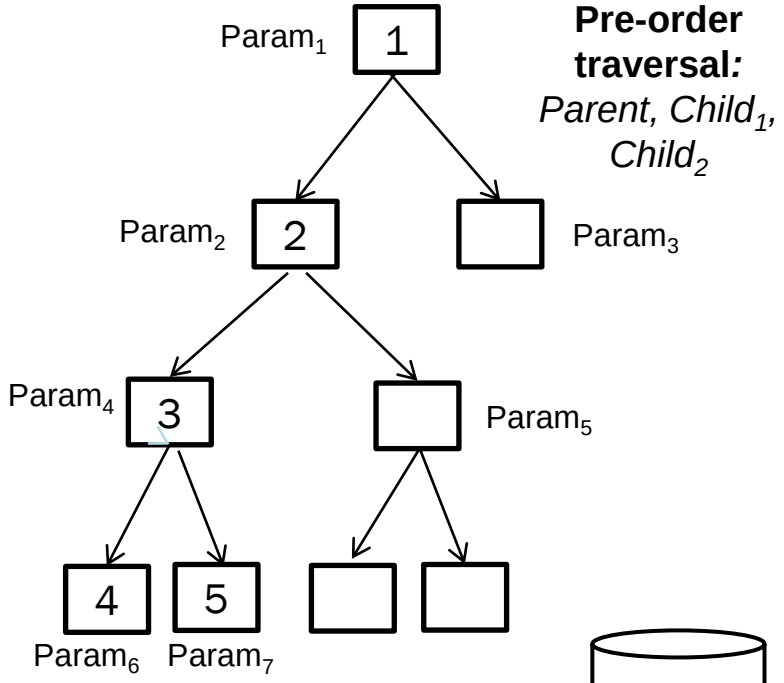
General Form 3



- *type* **recursiveFunction** *(input space, parameters)*
  1. Can recursion terminate successfully or with failure?
  2. Call **recursiveFunction** (input space, parameters1).
  3. Call **recursiveFunction** (input space, parameters2).
  4. Operations on the input space.
- *type* ***iterativeFunction*** *(input space, parameters)*
  1. Create a **parameter stack**, deposit parameters, and set a "visited" flag to false.
  2. While the **stack** is not empty:
     a) Retrieve the last deposited parameters from the stack (in reverse order).
     b) Can I proceed to the next iteration without adding parameters? (similar to recursion termination)
     c) Have the parameters not been visited? Add parameters, parameters2, parameters1 to the stack.
     d) Otherwise, perform operations on the input space.

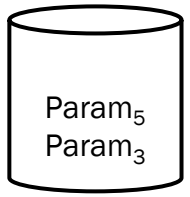
GF2 vs GF3

General Form 2

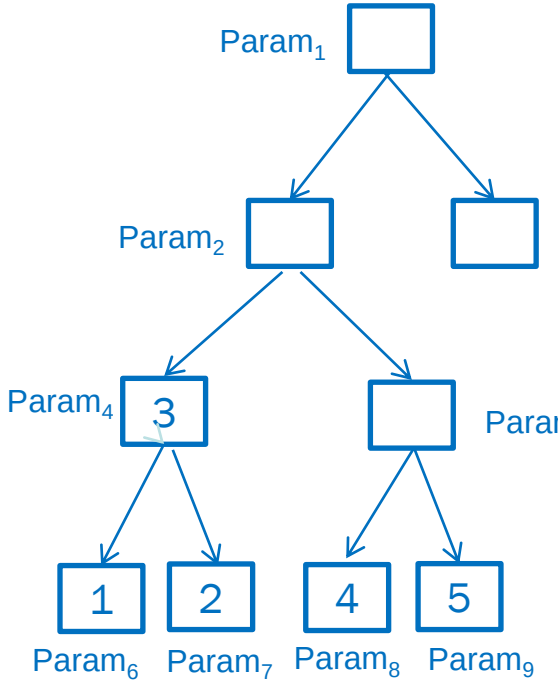


Pre-order traversal: *Parent, Child$_1$, Child$_2$*

Parámetros pendientes por evaluar al ejecutar **5**:

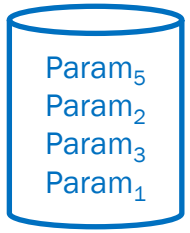


General Form 3



Post-order traversal: *Child$_1$, Child$_2$, Parent*

Parámetros pendientes por evaluar al ejecutar **5**:



MergeSort Excercise

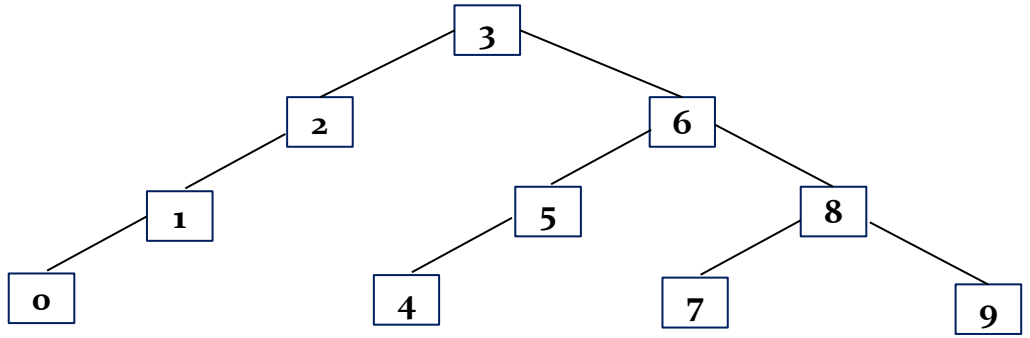
- Show the argument stack for sorting the following numbers:
- [10,5,18,6,3]

Exercise

- Apply General Form 3 toMergeSort
  - No new arrays will be created. Instead, use integers to denote the boundaries of sub-arrays.
  - The "merge()" method will receive the original array and the left and right boundaries of the two sub-arrays to merge.
  - Store the result of the merge in the original array, considering that the two sub-arrays are adjacent and mutually exclusive. Do not use temporary arrays.
  - Do not deposit <left, right, visited> in the stack if:
    - » Left and right are equal (trivial array)
    - » The current sub-array has already been visited during merging.
  - Otherwise, deposit the following in the stack:
    - » Arguments of the current sub-array, indicating that it has been visited.
    - » Arguments of the right half from middle + 1 to right.
    - » Arguments of the left half from left to middle.

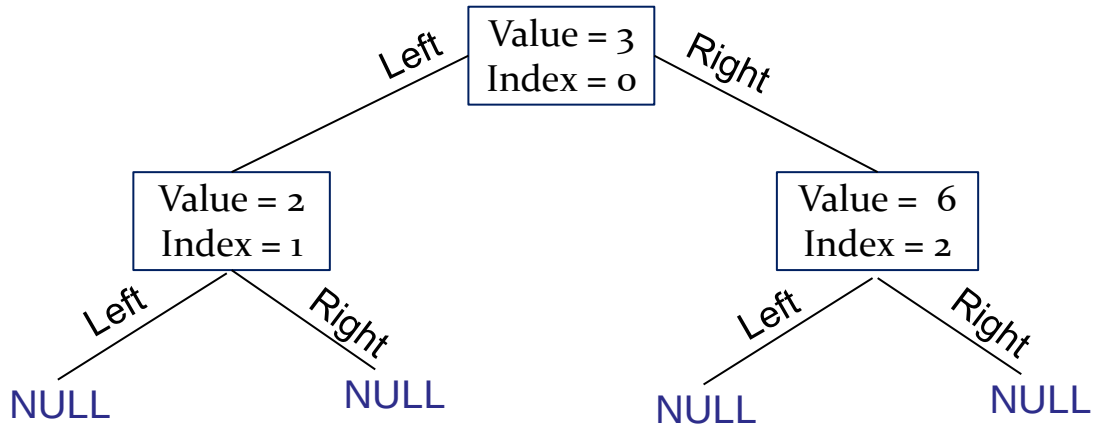
Binary Trees

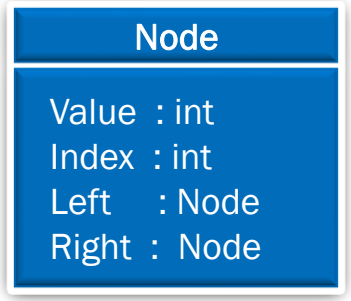
- For unordered arrays:
- A binary tree is constructed from the list.
  - Not necessarily balanced or aligned.
  - The left child is always smaller than the parent.
  - The right child is always greater than (or equal to) the parent.
- Binary tree from {3, 2, 6, 1, 5, 8, 0, 4, 7, 9}::



Structure of a Binary Tree

- A binary tree is composed of one or more nodes.
- The initial node is the root, which stores the first element of the list.
- From each node, one or two nodes branch out, called the left child and right child.
- No nodes branch out from leaf nodes.



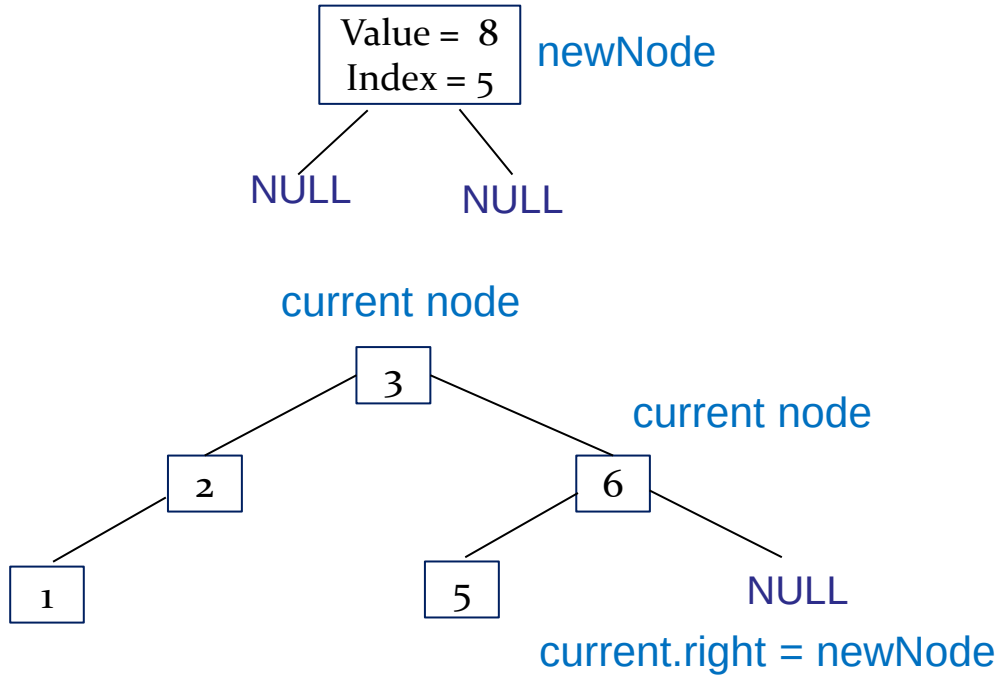


From List to Binary Tree

- For each element in the array other than the first one:
  1. Create a new node with the value and position of the element.
  2. Point to the root node.
  3. If the pointed node contains the value to search for, return the position it contains.
  4. If the pointed node contains a value greater than the one being searched for, point to the left child; otherwise, point to the right child.
  5. If the pointed child does not exist (NULL), insert the new node there.
  6. Otherwise, return to step 3.

From List to Binary Tree

Example. From the array {3, 2, 6, 1, 5, 8, 0, 4, 7, 9}, insert 8 into the Binary Tree (BT)



Búsqueda en un Árbol Binario

- More Intuitive Implementation:
  - A recursive method that receives the current node and the value to search for.
  - It returns the index.
  - In the first call, the root node is passed.

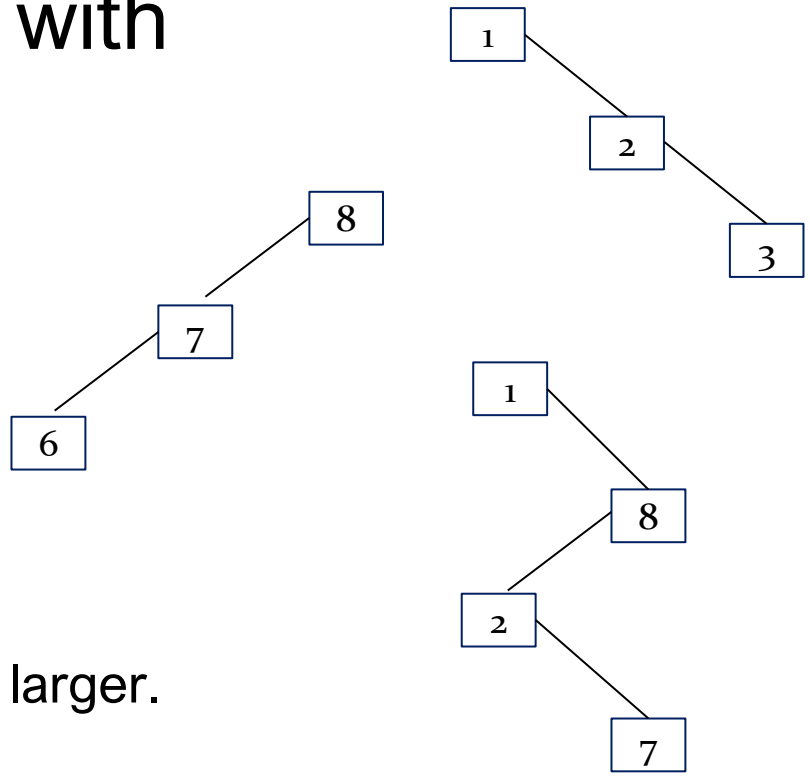
1. If the received node is null, it was not found [return -1].
2. If the value indicated by the node is equal to the value being searched for, return the index indicated in the node.
3. If the value indicated by the node is less than the one being searched for, repeat the search [recursively] with the left node of the current node.
4. Otherwise, repeat the search with the right node of the current node.

Análisis de la búsqueda con AB

- The maximum number of jumps made to reach the position of each element i is the number of levels of the tree.
- A balanced binary tree with N nodes has $\log_2 N$ levels.
  - The time complexity is quasi-linear($N \cdot \log N$)
  - Sequential search has lower complexity($N < N \cdot \log N$)
- Where is the advantage?
  - Once the BT is created, the search for an element has logarithmic complexity, according to studies: **2ln N**.
  - In practice, the binary tree is created once and used many times.

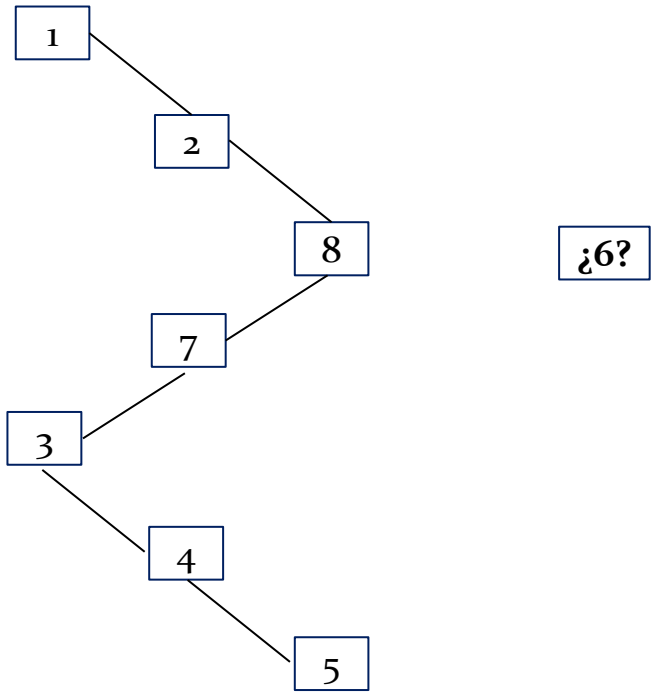
Worst Cases of BT

- There are three worst cases with linear complexity:
  1. Sorted list.
  2. Reversed list.
  3. List that alternates between smaller and larger.



Worst Cases of BT

- In general, when many nodes have only one child.
  - Segments of the list are either ordered or reversed.



Exercises (if time permits)

- Implement the Search function in a Binary Tree.
- Verify that with random arrays, the time complexity is logarithmic, and with ordered, reversed, and alternating arrays, it is linear.

Conclusions

- Removing recursion
- Removing recursion from binary search
- Iterative merge sort
- Binary trees