Design and Analysis of Algorithms

Session 6.

Maestría en Sistemas Computacionales.

Luis Fernando Gutiérrez Preciado.

What Will We See Today?

- Binary Tree Search
- 2-3-4 Tree Search
  - Components
  - Creation Logic
  - Implementation of Creation and Search Algorithms

Searching in a Binary Tree

- 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.

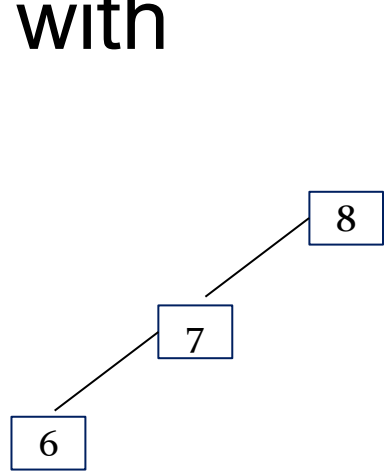
1. If the received node is null, it was not found [return -1].
2. If the value indicated by the node is equal to the value being searched for, return the index indicated in the node.
3. If the value indicated by the node is larger than the one being searched for, repeat the search [recursively] with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

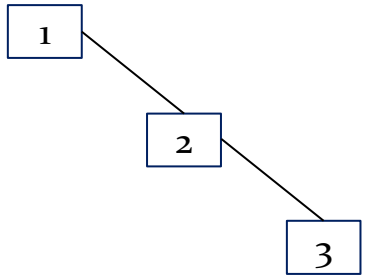
Analysis of searching in a BT

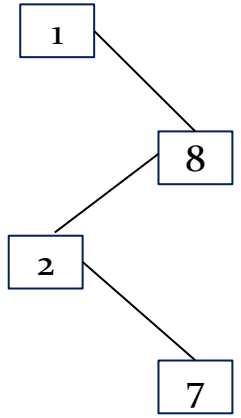
- The maximum number of jumps made to reach the position of each element i is the number of levels of the tree.
- A balanced binary tree with N nodes has $\log_2 N$ levels.
  - The time complexity is quasi-linear($N \cdot \log N$)
  - Sequential search has lower complexity($N < N \cdot \log N$)
- Where 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.

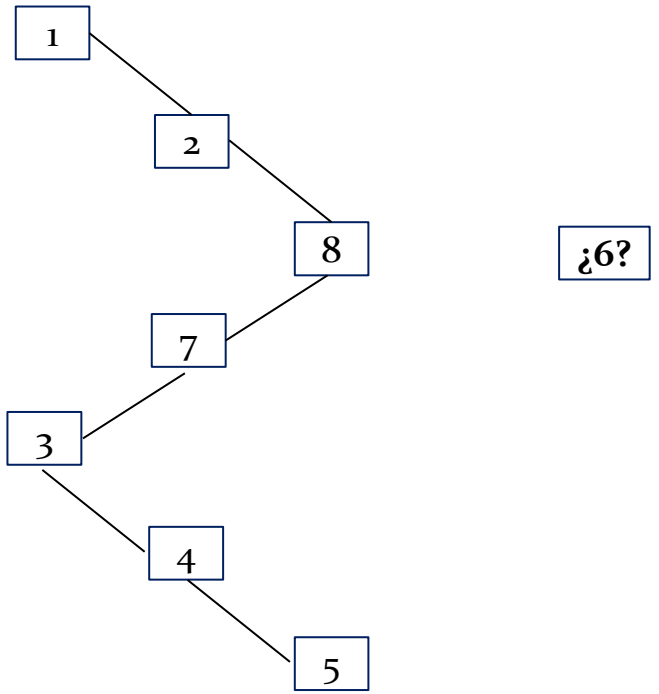






Worst Cases of BT

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

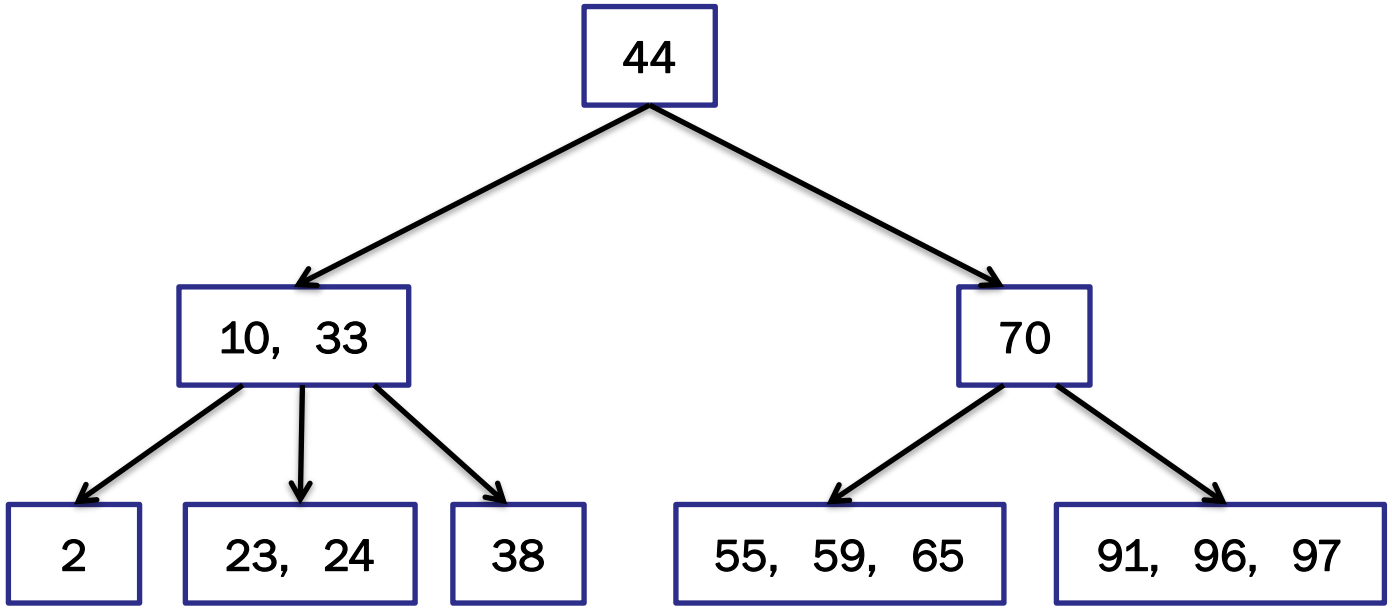


Balanced Trees

- To eliminate the worst-case scenarios in binary tree search, the tree must remain balanced.
  - Known balanced trees:
  - 2-3-4 Trees
- Red-Black Trees
- They offer very good performance ($\approx \lg N$) but are not as easy to program.
  - Create the tree once, search N times.

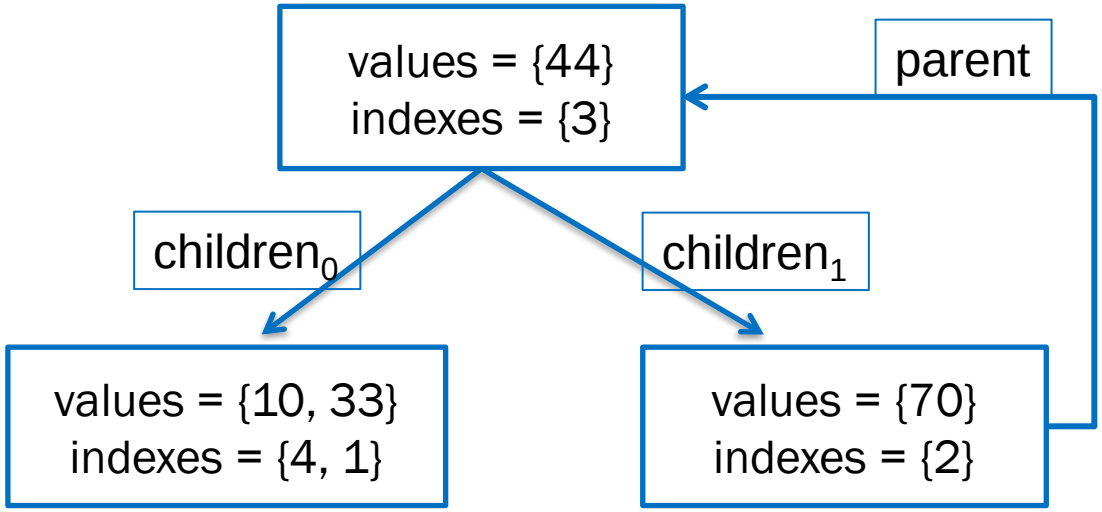
2-3-4 Trees

- Composed of Nodes-2, Nodes-3, Nodes-4
- Node type (2, 3, 4) = number of values + 1 = number of children (only for internal nodes).



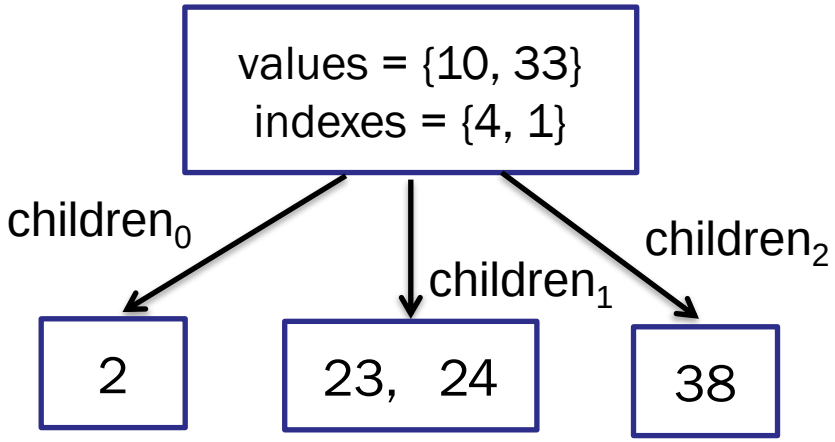
Nodes-2

- Store a value and its index in the original list.
- If the node is not a leaf, it has two children.
  - All values of $children_0$ are less than $values_0$.
  - All values of $children_1$ are greater than $values_0$.



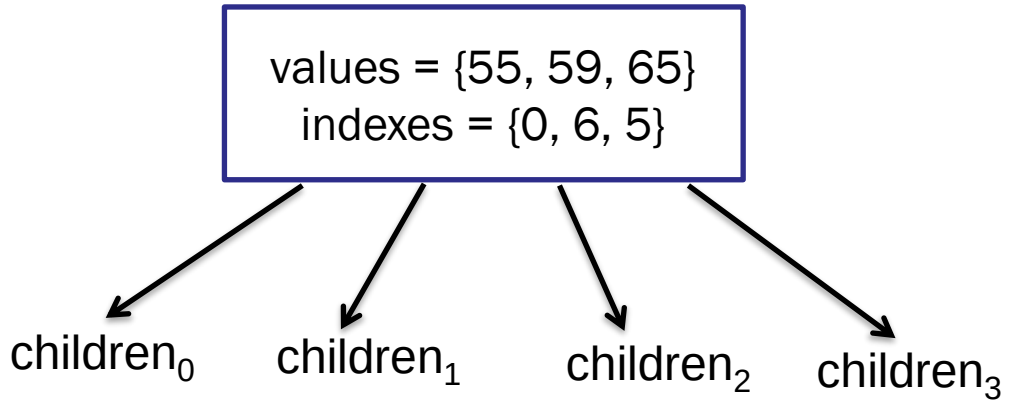
Nodes-3

- Store two values (and their indices).
- If the node is not a leaf, it has three children.
  - All values of $children_0$ are less than $values_0$.
  - All values of $children_1$ are in the range [$values_0$ ..$values_1$).
  - All values of $children_2$ are greater than $values_1$.



Nodes-4

- Store three values (and their indices).
- If the node is not a leaf, it has four children.
  - All values of $children_0$ are less than values0.
  - All values of $children_1$ are in the range [$values_0$..$values_1$).
  - All values of $children_2$ are in the range [$values_1$..$values_2$).
  - All values of $children_3$ are greater than $values_2$.



Example of 2-3-4 Tree Creation

- List = {3, 1, 5, 6, 2, 0, 7, 4}

1. [3]
2. [1, 3]
3. [1, 3, 5]
4. [1, 3, 5] *Node-4 ⇒ 2 Nodes-2*

[3] ← *Current (new root)*

[1] [5]

[3]

[1] [5, 6]

5. [3]

[1, 2] [5, 6]

6. [3]

[0, 1, 2] [5, 6]

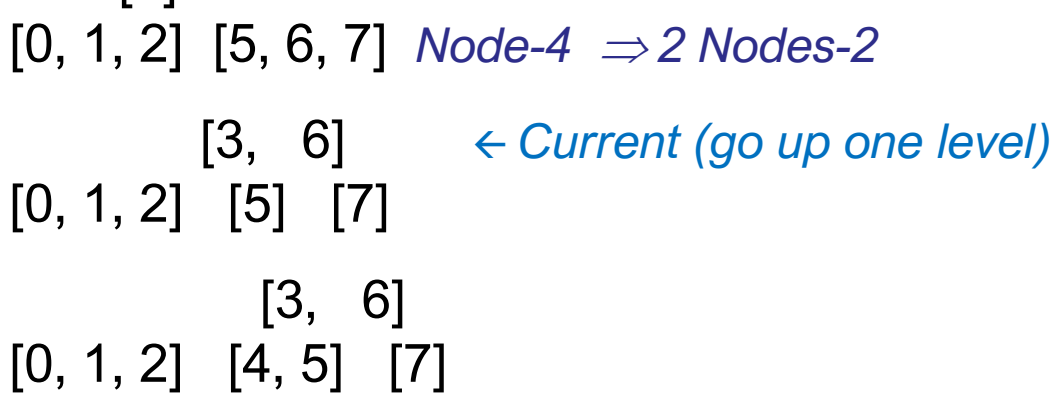
7. [3]

[0, 1, 2] [5, 6, 7]

Example of 2-3-4 Tree Creation

- List = {3, 1, 5, 6, 2, 0, 7, 4}

8.



2-4 Tree Animation by Y. Daniel Liang

2-3-4 Tree Insertion Logic

- Always start from the root.
- The path to choose depends on the value to insert.
- Always place the element in a leaf node.
- If on the path to the leaf, we encounter a Node-4:
  - Replace it with two Node-2 nodes (2 nodes with 1 value) with left (0) and right (2) values from Node-4.
  - The middle value will go into the parent node.
  - But what if the current node has no parent?
    - Create a new root node with the intermediate value.
- Note that the tree grows upward.

Exercise

- Create the 2-3-4 Tree corresponding to a sorted list of the first 10 natural numbers. (Show steps each time a Node-4 is split.)
- How many hops in the tree do you need to find the element furthest from the root?

Node-234 Structure

Node234

- values : Linked list of Integer
- indexes : Linked list of Integer
- children : Linked list of Node234
- parent : Node234

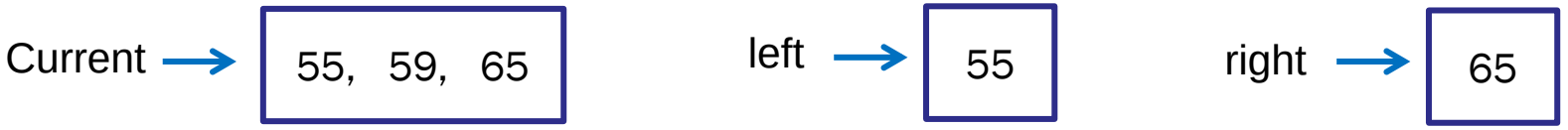
+ Node234(int value, int index-in-array)	
+ getType() : int	Returns 2, 3, or 4..
+ isLeaf() : boolean	Does it have children?
+ insert(value, index) : int	Returns where it was inserted.
+ getParent() : Node234	
+ addChild(child)	Adds at the end, updates the parent.
+ addChildren(ch1, ch2, index)	Places ch1 at position index, replacing the previous one inserts ch2 at index + 1. Updates the parents.

Algorithm for Creating a 2-3-4 Tree

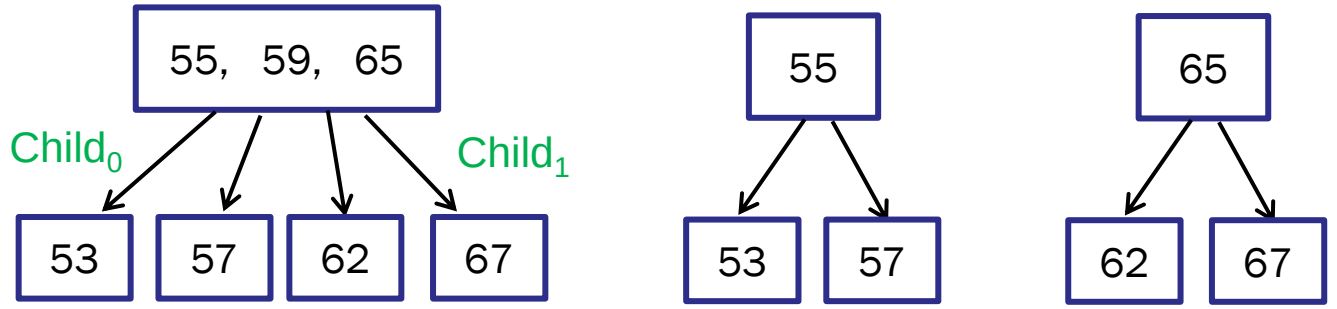
1. Create a root Node-2 with the data of the 1st element of the list.
2. For each subsequent element in the list:
   a) Let current be a node pointing to the root
   b) While a leaf node has not been processed:
      i. If 'current' is a Node-4, split it (next slide).
      ii. If not, if *'current'* is a leaf, place the data (value, index) in the same position, maintaining the sorted list of values.
      iii. If not, *'current'* will now point to one of its children, chosen according to the value of the element and the current node type: Node-2 or Node-3.

Splitting a Node-4

a) Create left and right nodes that store the values (and indices) at the ends of the current node.



b) If the current node is not a leaf, add the corresponding children of 'current' to the left and right nodes. Note: when adding a child, inform the child who its parent is.



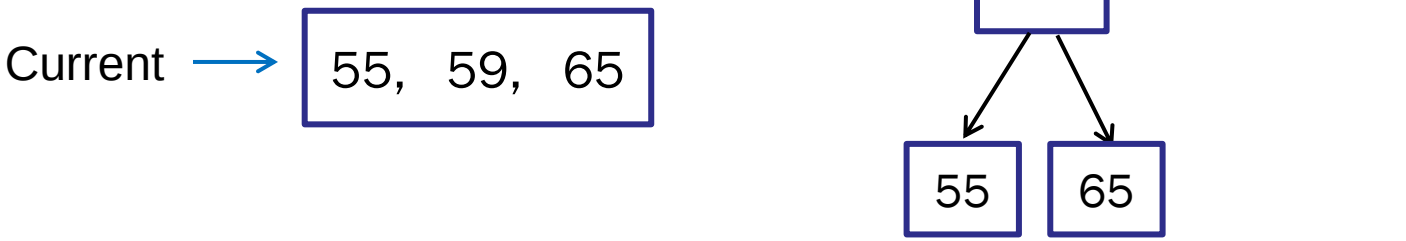
Splitting a Node-4

c) If *'current'* is the root, the tree grows upward:

i. Create a new root with the middle value of *'current'.*

ii. The root's children will be the left and right nodes.

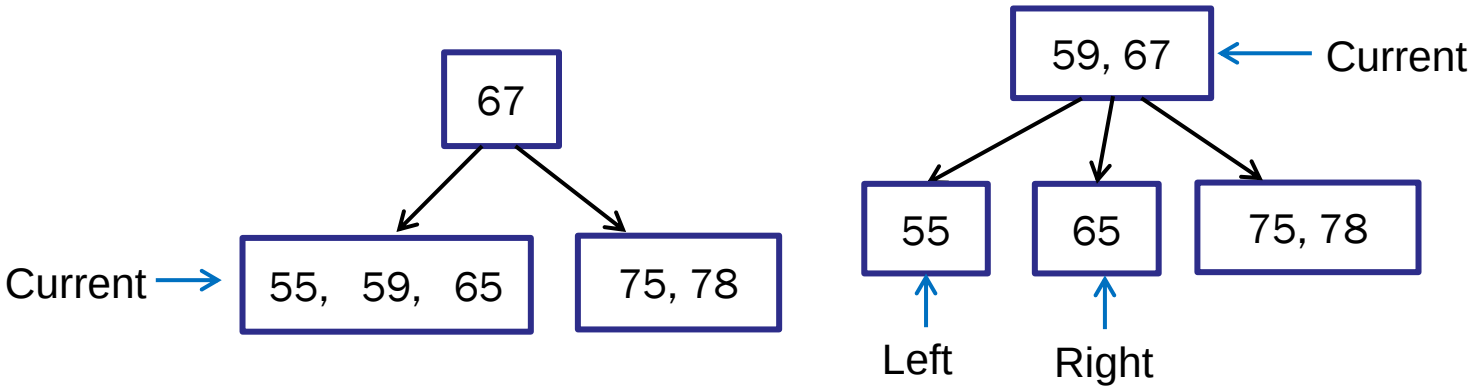
iii. *'current'* becomes the root.



Splitting a Node-4

d) Otherwise (current is not the root):

i. Get the parent node.
ii. Insert the intermediate value of the current node into the parent.
iii. Insert the left node into the parent at the position where the value was inserted, replacing the child that was there [55, 59, 65].
iv. Insert the right node into the parent in the next position.
v. The current node is now the parent.



Conclusions

- Binary Tree Search
- 2,3,4 Tree Construction
- Generalization of B-Tree