Last week

- BST
  - Insert (and trace)
  - Iterative delete

Today

- More on BST
  - Recursive delete
- Efficiency
BST Delete

- Locate the node to be deleted and its parent
  - `current` and `parent_of_current`

- Case 1: The `current` node has no left child:
  - Simply connect the `parent` with the right child of the `current` node.
Last week

- BST Delete
  - **Case 2:** The current node has a left child:
    - Locate the `right_most` and `parent_of_right_most`
    - Replace the element value in the current node with the one in the `right_most` node,
    - Connect the `parent_of_right_most` node with the left child of the `right_most` node.
Case 1 or 2? 2
Delete this node

Case 1 or 2?
Delete this node

Case 1 or 2? 2
bst_delete

• First locate the nodes that contain the element and its parent. Call them current and parent.

parent = None
current = root

while current is not None and current.data != data:
    if data < current.data:
        parent = current
        current = current.left
    elif data > current.data:
        parent = current
        current = current.right
    else:
        pass  # Element is in the tree pointed at by current

if current is None: return False  # Element is not in the tree
# Case 1: bst_delete

# Case 1: current has no left child
if current.left is None:
    # Connect the parent with the right child of the current node
    # Special case, assume the node being deleted is at root
    if parent is None:
        current = current.right
    else:
        # Identify if parent left or parent right should be connected
        if data < parent.data:
            parent.left = current.right
        else:
            parent.right = current.right
else:
    # Case 2: The current node has a left child
Case II: bst_delete

# Locate the rightmost node in the left subtree of
# the current node and also its parent
parent_of_right_most = current
right_most = current.left

while right_most.right is not None:
    parent_of_right_most = right_most
    right_most = right_most.right # Keep going to the right

# Replace the element in current by the element in rightMost
current.element = right_most.element

# Eliminate rightmost node
if parent_of_right_most.right == right_most:
    parent_of_right_most.right = right_most.left
else:
    # Special case: parent_of_right_most == current
    parent_of_right_most.left = right_most.left

return True # Element deleted successfully
Exercise

- In Slides 3 and 4,
  - replace every *left* with *right*, every *right* with *left*, and also *largest* with *smallest*.

- and, implement the method.

- Next Topic:
  - A recursive method for BST delete.
Let’s define it as deleting a node (if exists) from the BST and returning the resulting BST

Example:
- \( t = \text{bst\_del\_rec}(t, 10) \)
- deletes 10 from BST \( t \) and returns the reference to the tree
bst_del_rec(tree, data)

- **Base case**
  - If the tree is none return none
    ```python
    if not tree:
        return None
    ```

- **Recursive case I**
  - If data is less than tree data, delete it from left child
    ```python
    if data < tree.data:
        tree.left = bst_del_rec(tree.left, data)
    ```

- **Recursive case II**
  ```python
  if data > tree.data:
      tree.right = bst_del_rec(tree.right, data)
  ```
bst_del_rec(tree, data)

- What does it mean if none of the above if's have been true?
  - We have located the tree node to be deleted

- What next?

- There are two cases to consider ...

- Case I:
  - If the tree node does not have a left child,
    - return the right child

  ```python
  if tree.left is None:
    return tree.right
  ```
bst_del_rec

- Recall examples for case I:
bst_del_rec(tree, data)

- **Case II:**
  - If the tree node does have a left child,
    - find the largest node of the left child
    - replace the tree node data with the largest just found
    - delete the largest

```python
if tree.left is not None:
    largest = findmax(tree.left)
    tree.data = largest.data
    tree.left = bst_del_rec(tree.left, largest.data)
    return tree
```
Case 2 (diagram)

- **current**: current may be a left or right child of parent
- **current points the node to be deleted**
- **Right subtree**
- **parent**: parent of rightmost node
- **leftChildOfRightMost**: left child of rightmost node
- **rightMost**: rightmost node
- **parentOfRightMost**: parent of rightmost node

The content of the current node is replaced by content by the content of the rightmost node. The rightmost node is deleted.

Content copied to current and the node deleted.
bst_del_rec

- Recall examples for case II:
# putting everything together

# base case
if not tree:
    return None

# recursive case I
elif data < tree.data:
    tree.left = bst_del_rec(tree.left, data)

# recursive case II
elif data > tree.data:
    tree.right = bst_del_rec(tree.right, data)

# left child is empty
elif tree.left is None:
    return tree.right

# left child is not empty
else:
    largest = findmax(tree.left)
    tree.data = largest.data
    tree.left = bst_del_rec(tree.left, largest.data)
    return tree

# helper
def findmax(tree):
    return tree if not tree.right else findmax(tree.right)
Efficiency of algorithms

- **BST**: iterative delete vs. recursive delete?
  - Extra memory?
    - *Constant vs. in order of height of tree*
    - *O(1) vs. O(lg n) if balanced or O(n) otherwise*
  - Time?
    - *Although both in order of height of tree, the latter requires more work*

- **Fibonacci**: iteration vs. recursion?
  - Extra memory?
    - *O(1) vs. O(n)*
  - Time?
    - *O(n) vs. O(2^n)!!*
# Efficiency of algorithms

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$3 \times 10^{-11}$ s</td>
<td>$10^{-10}$ s</td>
<td>$3 \times 10^{-10}$ s</td>
<td>$10^{-9}$ s</td>
<td>$10^{-8}$ s</td>
<td>$3 \times 10^{-7}$ s</td>
</tr>
<tr>
<td>$10^2$</td>
<td>$7 \times 10^{-11}$ s</td>
<td>$10^{-9}$ s</td>
<td>$7 \times 10^{-9}$ s</td>
<td>$10^{-7}$ s</td>
<td>$4 \times 10^{11}$ yr</td>
<td>*</td>
</tr>
<tr>
<td>$10^3$</td>
<td>$1.0 \times 10^{-10}$ s</td>
<td>$10^{-8}$ s</td>
<td>$1 \times 10^{-7}$ s</td>
<td>$10^{-5}$ s</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>$10^4$</td>
<td>$1.3 \times 10^{-10}$ s</td>
<td>$10^{-7}$ s</td>
<td>$1 \times 10^{-6}$ s</td>
<td>$10^{-3}$ s</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>$10^5$</td>
<td>$1.7 \times 10^{-10}$ s</td>
<td>$10^{-6}$ s</td>
<td>$2 \times 10^{-5}$ s</td>
<td>$0.1$ s</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$2 \times 10^{-10}$ s</td>
<td>$10^{-5}$ s</td>
<td>$2 \times 10^{-4}$ s</td>
<td>$0.17$ min</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
Recursive vs iterative

- Recursive functions impose a loop
- The loop is implicit and the compiler/interpreter (here, Python) takes care of it
- This comes at a price: time & memory
- The price may be negligible in many cases

- After all, no recursive function is more efficient than its iterative equivalent
Recursive vs iterative cont’ed

- Every recursive function can be written iteratively (by explicit loops)
  - may require stacks too
- yet, when the nature of a problem is recursive, writing it iteratively can be
  - time consuming, and
  - less readable

- So, recursion is a very powerful technique for problems that are naturally recursive