Chapter 5: Tree
Part II

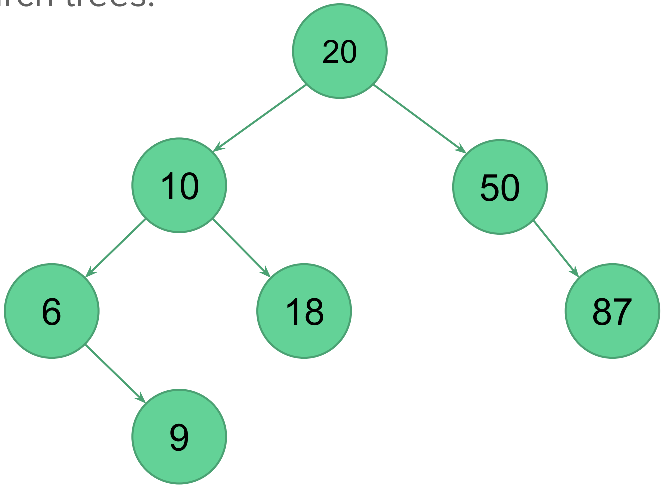
Department of Computer Science and Engineering
Kathmandu University

Contents

Binary search tree

Binary Search Tree

- A binary search tree (BST) is a binary tree that is either empty or (in which each node contains a key that) satisfies the following properties.
    - The keys (if any) in left sub-tree are smaller than the key in the root.
    - The keys (if any) in the right subtree are larger than the key in the root.
    - The left and right subtrees are also binary search trees.



Binary Search Tree

- A BST has a better performance than any of the data structures studied so far when the functions to be performed are search, insert and delete.
- BSTs are used in many search applications where data is constantly entering/leaving, such as the map (or dictionary) and set objects in many languages’ libraries.

Binary Search Tree Operations

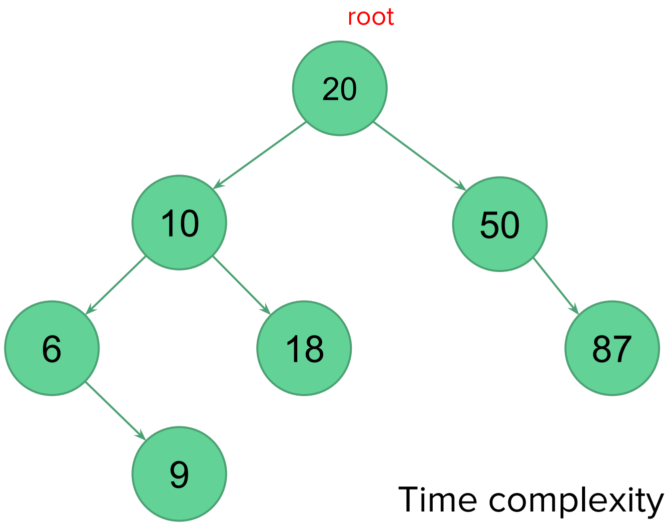
1. Searching a BST
    a. Find the smallest node
    b. Find the largest node
    c. Find a requested node
2. Insertion
3. Deletion

Finding the smallest node

The main idea is to follow the left branches until we get to a leaf. The smallest node is the far-left node in the tree.

Algorithm: Find the smallest node in a BST
Input: A BST, T(root)
Output: The smallest node in T
Steps:

1. If left subtree is empty
    return root
2. End if
3. smallest = find smallest node in the left subtree
4. return smallest



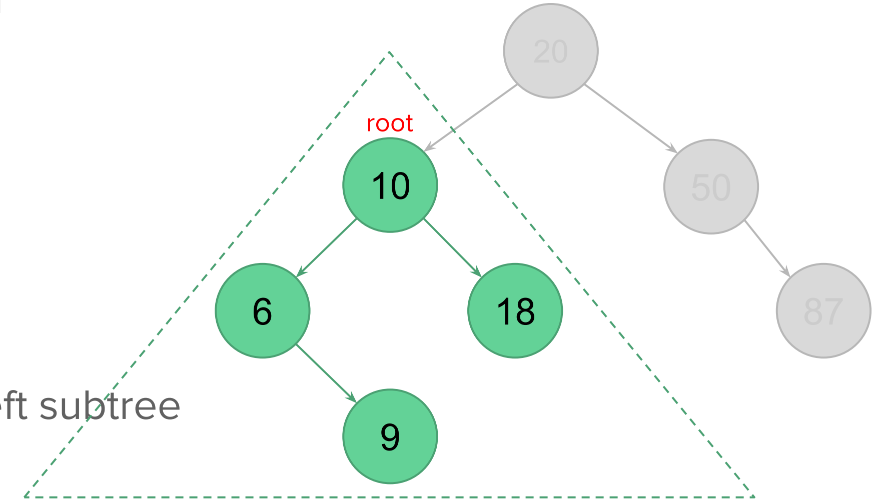
Time complexity = ?

Finding the smallest node

The main idea is to follow the left branches until we get to a leaf. The smallest node is the far-left node in the tree.

Algorithm: Find the smallest node in a BST
Input: A BST, T(root)
Output: The smallest node in T
Steps:

1. If left subtree is empty
   return root
2. End if
3. smallest = find smallest node in the left subtree
4. return smallest



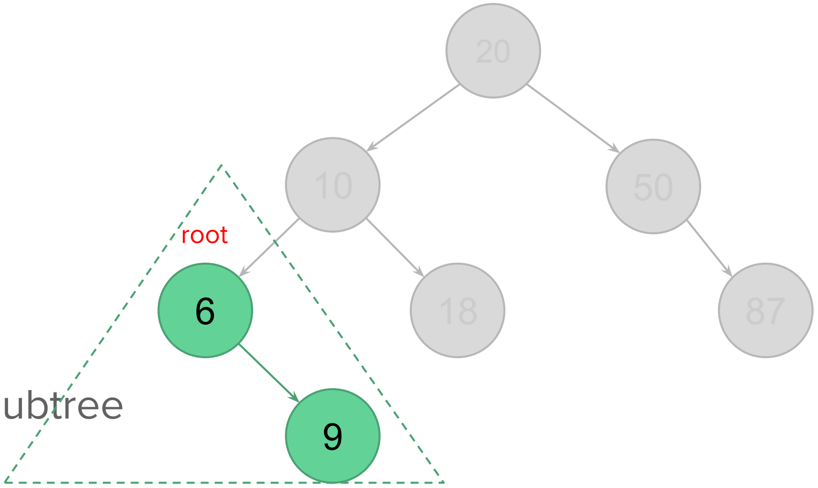
Time complexity = ?

Finding the smallest node

The main idea is to follow the left branches until we get to a leaf. The smallest node is the far-left node in the tree.

Algorithm: Find the smallest node in a BST
Input: A BST, T(root)
Output: The smallest node in T
Steps:

1. If left subtree is empty
   return root
2. End if
3. smallest = find smallest node in the left subtree
4. return smallest



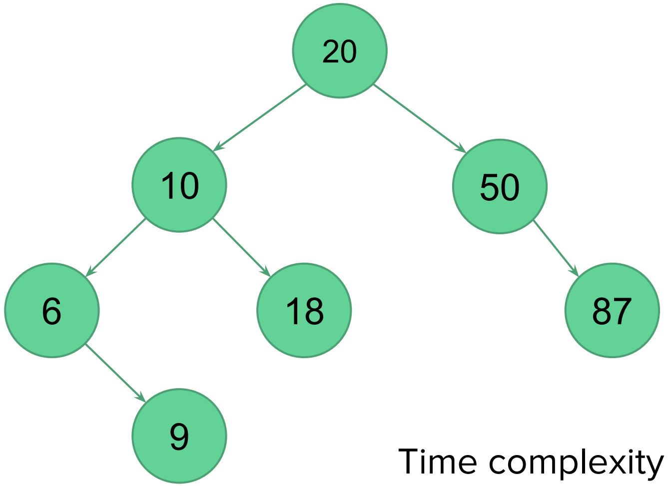
Time complexity = ?

Finding the largest node

The largest node is the far-right node in the tree. Follow the right branches until we get to a leaf.

Algorithm: Find the largest node in a BST
Input: A BST, T(root)
Output: The largest node in T
Steps:

1. If right subtree is empty
        return root
2. End if
3. largest = find smallest node in the right subtree
4. return largest



Time complexity = ?

Finding a requested node (recursive)

Algorithm: searchBST (root, targetKey)

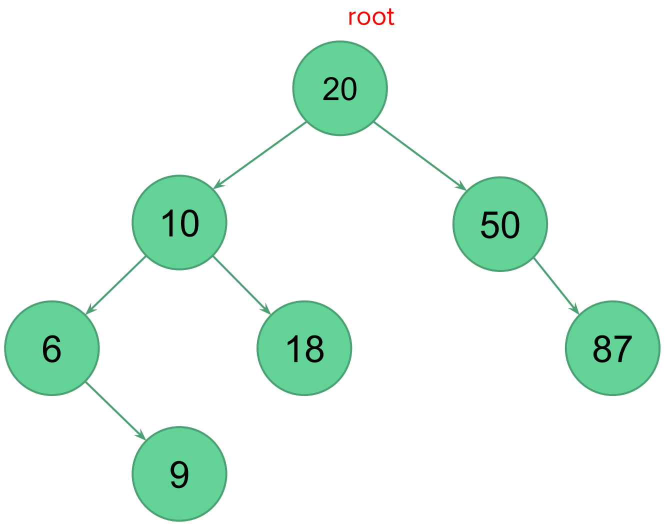
Description: Search a BST, `T(root)` for the requested key `targetKey`

Input: `T(root), targetKey`

Output: The requested key

Steps:

1. If T is empty
        return null
2. endif
3. If targetKey < root->key
        return searchBST(left subtree, targetKey)
4. else if targetKey > root->key
        return searchBST(right subtree, targetKey)
5. else
        return root
6. endif



Finding a requested node (recursive)

Algorithm: searchBST (root, targetKey)

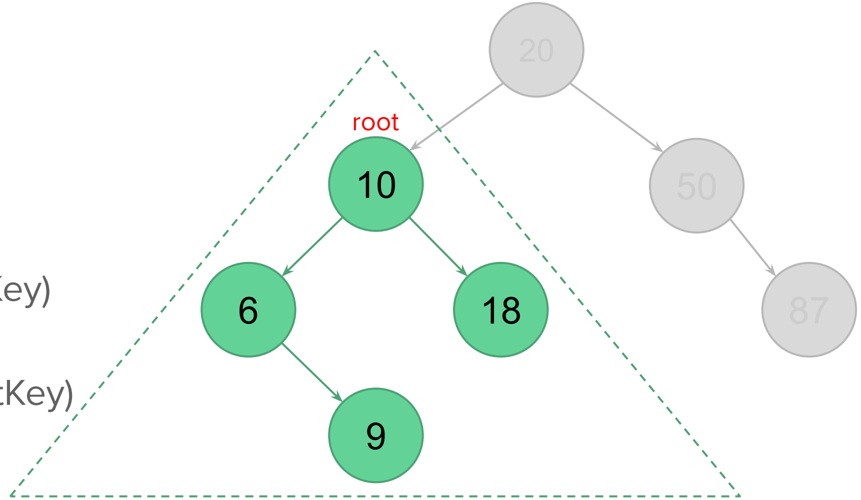
Description: Search a BST, `T(root)` for the requested key `targetKey`

Input: `T(root), targetKey`

Output: The requested key

Steps:

1. If T is empty
   return null
2. endif
3. If targetKey < root->key
   return searchBST(left subtree, targetKey)
4. else if targetKey > root->key
   return searchBST(right subtree, targetKey)
5. else
   return root
6. endif



Finding a requested node (iterative)

Algorithm: searchBST (root, targetKey)

Description: Search a BST, `T(root)` for the requested key `targetKey`

Input: `T(root), targetKey`

Output: The requested key

Steps:

1. temp = root
2. while (temp != null)
   a. If temp->key == targetKey
      i. return temp
   b. else if temp->key > targetKey
      i. temp = temp->leftChild
   c. else
      i. temp = temp->rightChild
3. End while
4. return null

Time complexity = ?

Inserting a node into a BST

All BST insertions take place at a leaf or a leaf-like node (i.e. a node that has only one null subtree).

Example:

Insert the following sequence of keys in an empty BST:

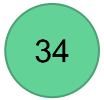
34, 56, 23, 44, 19, 77, 60, 33, 20

Inserting a node into a BST

All BST insertions take place at a leaf or a leaf-like node (i.e. a node that has only one null subtree).

Example:

Insert the following sequence of keys in an empty BST:



34, 56, 23, 44, 19, 77, 60, 33, 20

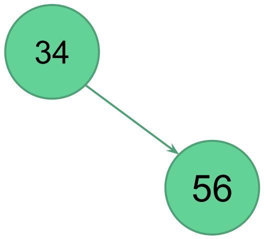
Inserting a node into a BST

All BST insertions take place at a leaf or a leaf-like node (i.e. a node that has only one null subtree).

Example:

Insert the following sequence of keys in an empty BST:

34, 56, 23, 44, 19, 77, 60, 33, 20



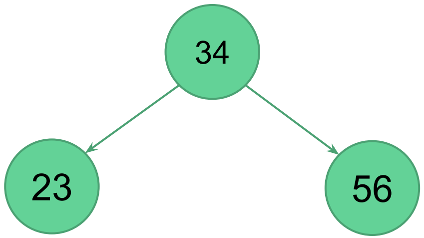
Inserting a node into a BST

All BST insertions take place at a leaf or a leaf-like node (i.e. a node that has only one null subtree).

Example:

Insert the following sequence of keys in an empty BST:

34, 56, 23, 44, 19, 77, 60, 33, 20



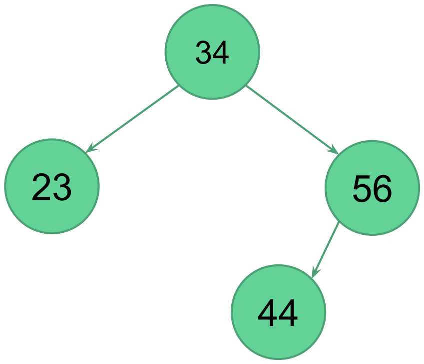
Inserting a node into a BST

All BST insertions take place at a leaf or a leaf-like node (i.e. a node that has only one null subtree).

Example:

Insert the following sequence of keys in an empty BST:

34, 56, 23, 44, 19, 77, 60, 33, 20



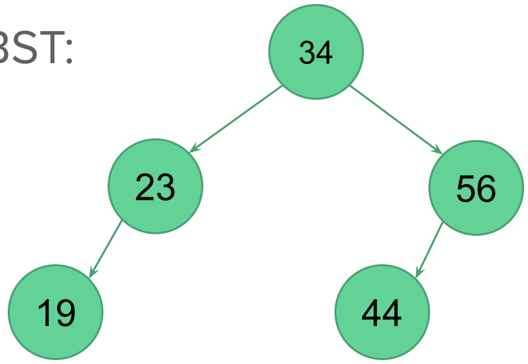
Inserting a node into a BST

All BST insertions take place at a leaf or a leaf-like node (i.e. a node that has only one null subtree).

Example:

Insert the following sequence of keys in an empty BST:

34, 56, 23, 44, 19, 77, 60, 33, 20



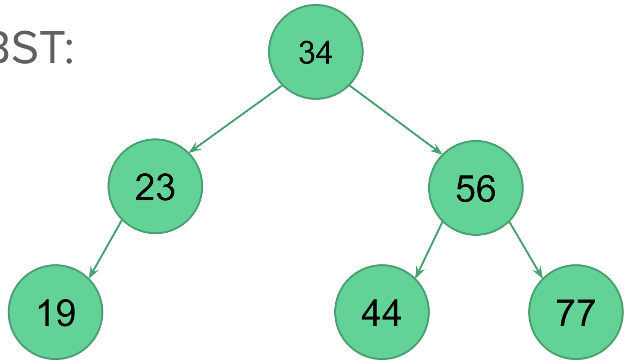
Inserting a node into a BST

All BST insertions take place at a leaf or a leaf-like node (i.e. a node that has only one null subtree).

Example:

Insert the following sequence of keys in an empty BST:

34, 56, 23, 44, 19, 77, 60, 33, 20



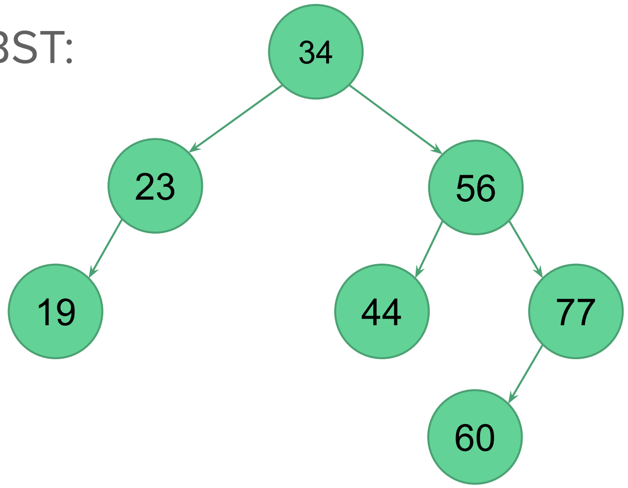
Inserting a node into a BST

All BST insertions take place at a leaf or a leaf-like node (i.e. a node that has only one null subtree).

Example:

Insert the following sequence of keys in an empty BST:

34, 56, 23, 44, 19, 77, 60, 33, 20



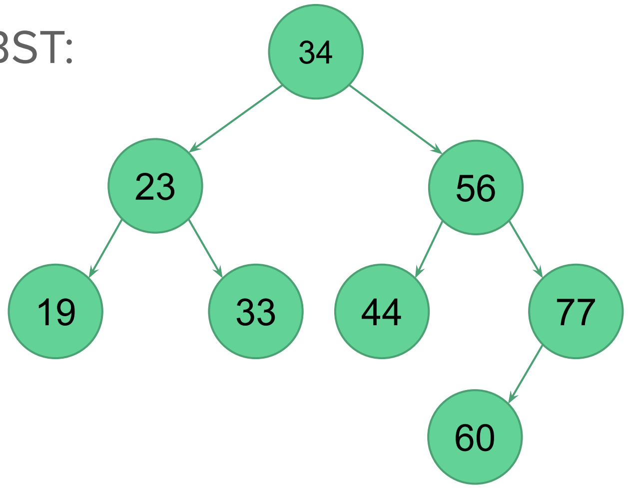
Inserting a node into a BST

All BST insertions take place at a leaf or a leaf-like node (i.e. a node that has only one null subtree).

Example:

Insert the following sequence of keys in an empty BST:

34, 56, 23, 44, 19, 77, 60, 33, 20



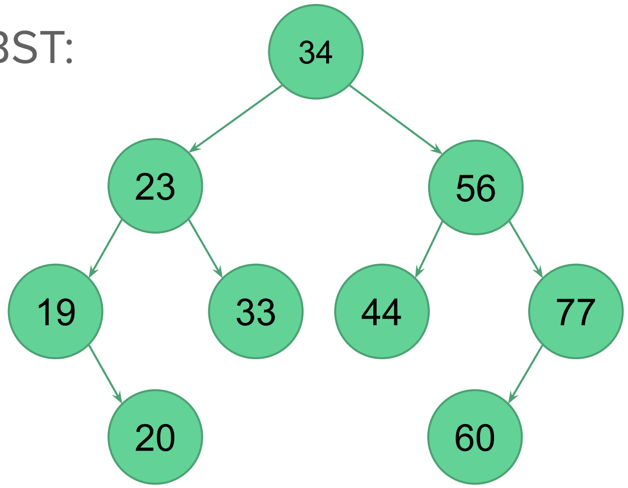
Inserting a node into a BST

All BST insertions take place at a leaf or a leaf-like node (i.e. a node that has only one null subtree).

Example:

Insert the following sequence of keys in an empty BST:

34, 56, 23, 44, 19, 77, 60, 33, 20



Inserting a node into a BST

Algorithm: addBST(root, newNode)

Description: Add a node into a BST

Input: A BST T(root), newNode

Output: Address of potential new tree root

Steps:

1. If T is empty (i.e. root is null)
   1.1. set root to newNode
   1.2. return newNode
2. endif
3. If newNode->key < root->key
   3.1. return addBST (left subtree, newNode)
4. else
   4.1. return addBST (right subtree, newNode)
5. endif

Time complexity = ?

Deleting a node from a BST

3 cases:

Case 1: The node to be deleted has no child

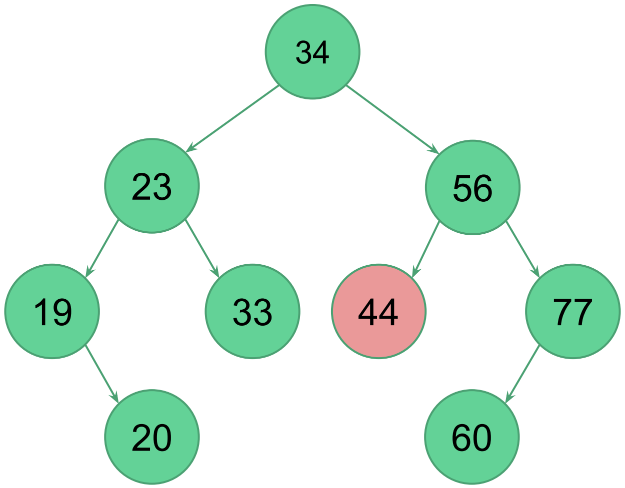
Case 2: The node to be deleted has only one child

Case 3: The node to be deleted has two subtrees

Deleting a node from a BST

Case 1: The node to be deleted has no child

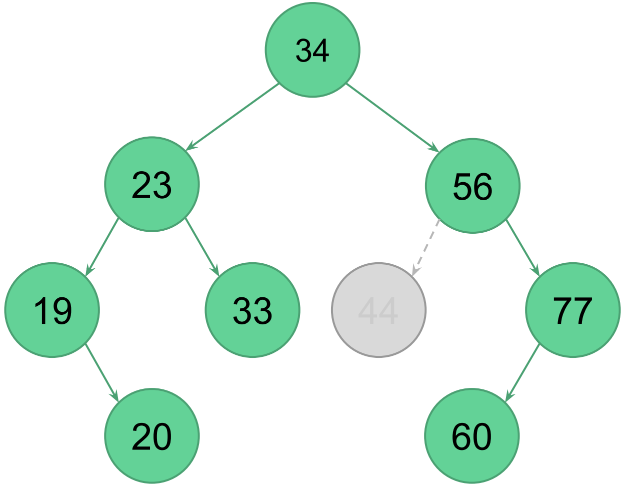
- Just delete the node



Deleting a node from a BST

Case 1: The node to be deleted has no child

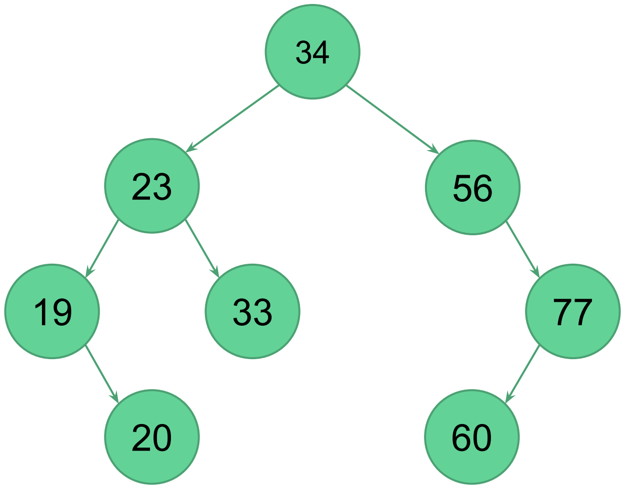
- Just delete the node



Deleting a node from a BST

Case 1: The node to be deleted has no child

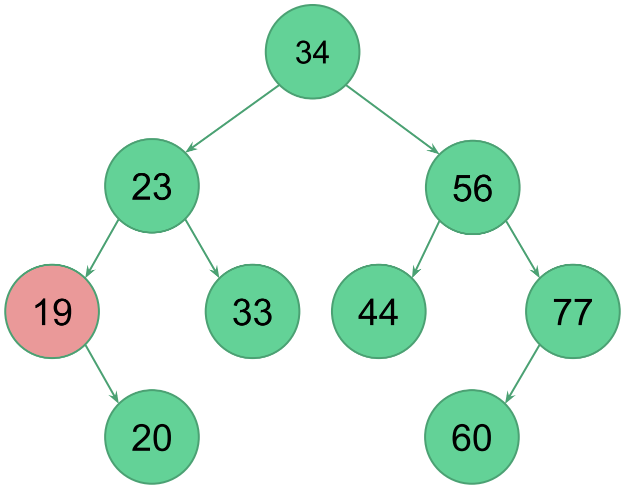
- Just delete the node



Deleting a node from a BST

Case 2: The node to be deleted has only one child

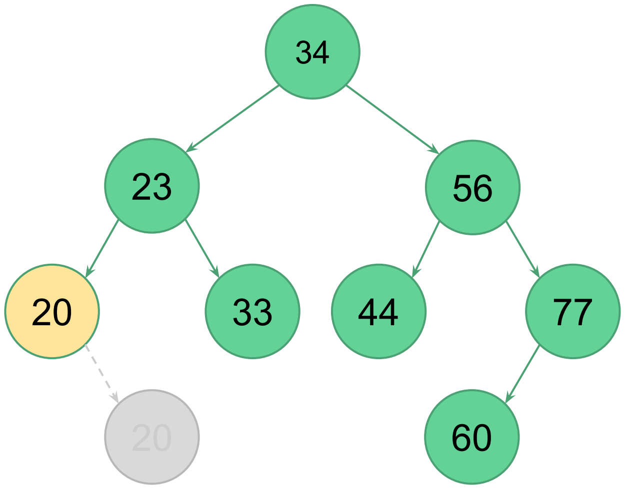
- If the node to be deleted has only a right subtree, then delete the node and attach the right subtree to the deleted node's parent
- If the node to be deleted has only a left subtree, delete the node and attach the left subtree to the deleted node's parent



Deleting a node from a BST

Case 2: The node to be deleted has only one child

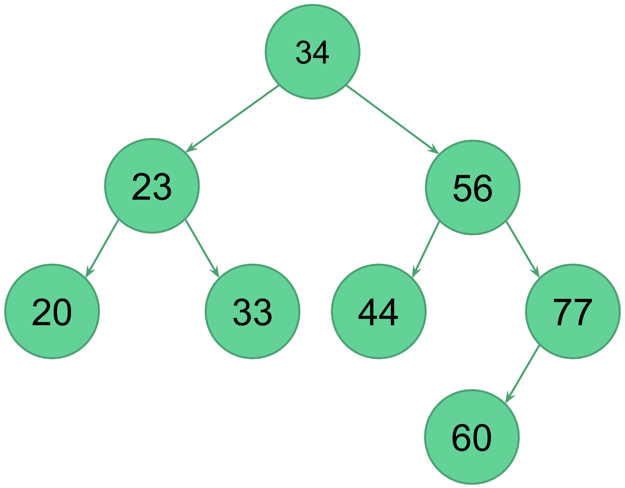
- If the node to be deleted has only a right subtree, then delete the node and attach the right subtree to the deleted node's parent
- If the node to be deleted has only a left subtree, delete the node and attach the left subtree to the deleted node's parent



Deleting a node from a BST

Case 2: The node to be deleted has only one child

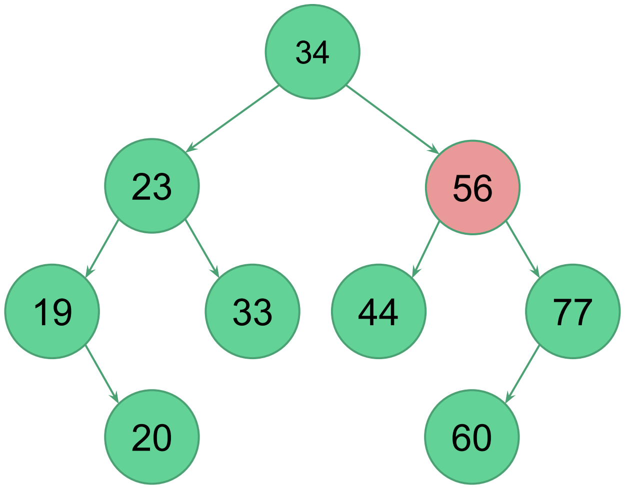
- If the node to be deleted has only a right subtree, then delete the node and attach the right subtree to the deleted node's parent
- If the node to be deleted has only a left subtree, delete the node and attach the left subtree to the deleted node's parent



Deleting a node from a BST

Case 3: The node to be deleted has two subtrees

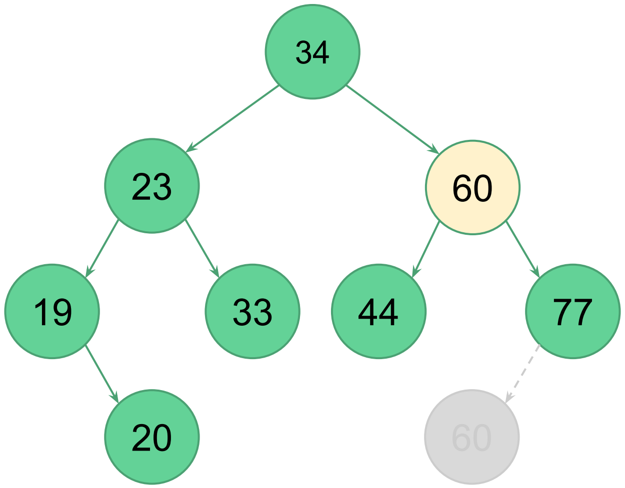
- Find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data, **or**
- Find the smallest node in the deleted node's right subtree and move its data to replace the deleted node's data



Deleting a node from a BST

Case 3: The node to be deleted has two subtrees

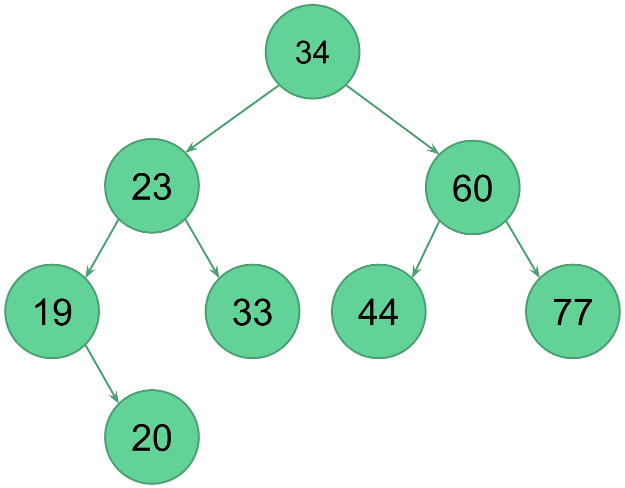
- Find the largest node in the deleted node’s left subtree and move its data to replace the deleted node’s data, **or**
- Find the smallest node in the deleted node’s right subtree and move its data to replace the deleted node’s data



Deleting a node from a BST

Case 3: The node to be deleted has two subtrees

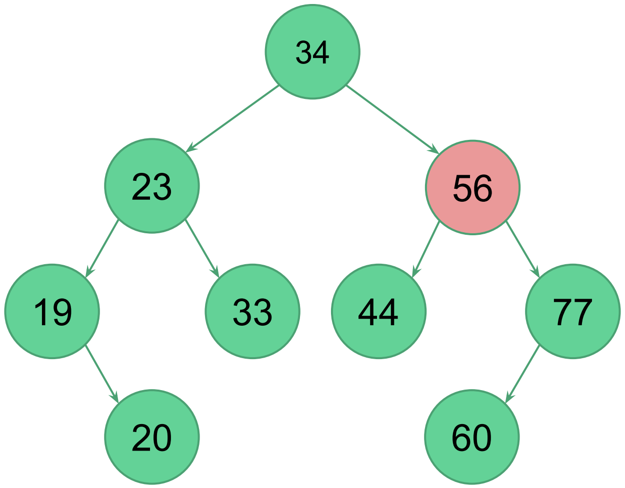
- Find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data, **or**
- Find the smallest node in the deleted node's right subtree and move its data to replace the deleted node's data



Deleting a node from a BST

Case 3: The node to be deleted has two subtrees

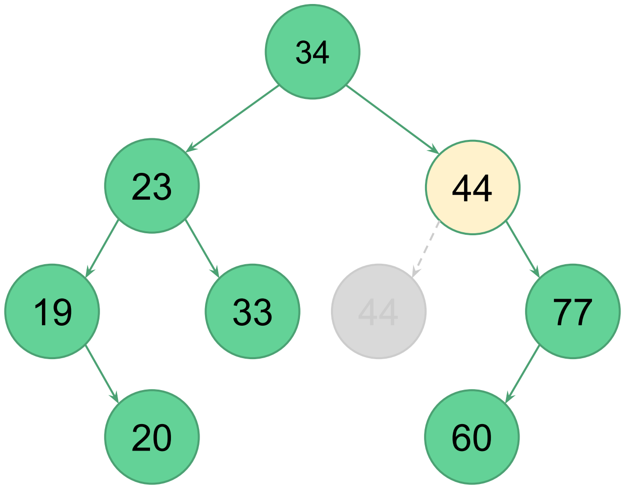
- Find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data, **or**
- Find the smallest node in the deleted node's right subtree and move its data to replace the deleted node's data



Deleting a node from a BST

Case 3: The node to be deleted has two subtrees

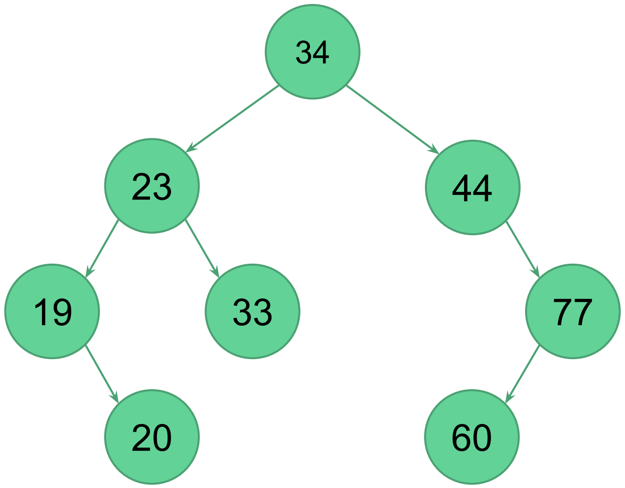
- Find the largest node in the deleted node’s left subtree and move its data to replace the deleted node’s data, **or**
- Find the smallest node in the deleted node’s right subtree and move its data to replace the deleted node’s data



Deleting a node from a BST

Case 3: The node to be deleted has two subtrees

- Find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data, **or**
- Find the smallest node in the deleted node's right subtree and move its data to replace the deleted node's data



Deleting a node from a BST

Desc: Delete a node from BST

Input: A BST, T(root), a key to delete, dltKey

Output: true if the node is deleted false if not

Steps:

1. if the tree is empty, then return false
2. if dltkey < root->key
   2.1. return deleteBST (left subtree, dltkey)
3. else if dltkey > root->key
   3.1. return delete BST (right subtree, dltKey)
4. else
   4.1. If no left subtree
      4.1.1. make right subtree the root
      4.1.2. return true
   4.2. else if no right subtree
      4.2.1. make left subtree the root
      4.2.2. return true
   1.3. else
      1.3.1. nodeToDelete = root
      1.3.2. largest = largest node in left subtree
      1.3.3. move data in largest to nodetoDelete
      1.3.4. return deleteBST (left subtree of nodeToDelete, key of largest)
   1.4. endif
5. endif

Time complexity = ?

Complexity of BST operations

Searching, insertion, and deletion in a binary search tree of height h run in O(h) time.

These operations are fast if the height of the search tree is small

If its height is large, they may run no faster than with a linked list.

In the worst case (max. height),
$h = n - 1$
In the best case (min. height),
$h = \lfloor \log_2 n \rfloor$

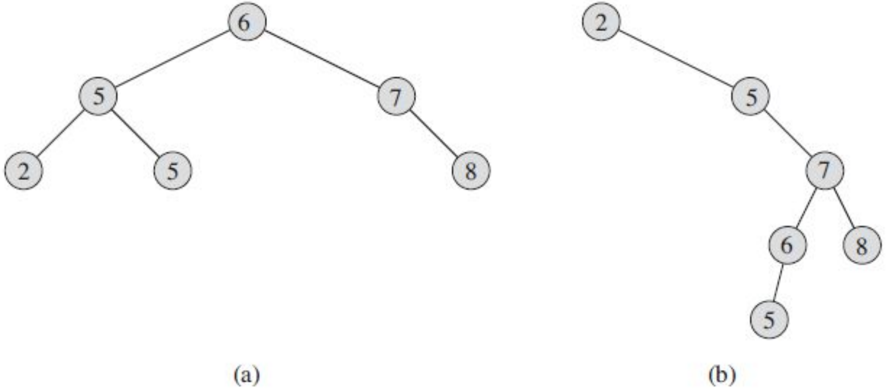


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

AVL Search Tree

- An AVL tree is a balanced binary search tree, i.e. a binary tree that is either empty or consists of two AVL subtrees whose heights differ by no more than 1.
- It is named after its two Soviet inventors, Georgy **A**delson-**V**elsky and Evgenii **L**andis.
- Whenever we insert a node into a tree or delete a node from a tree, the resulting tree may be unbalanced.
- When we detect that a tree is unbalanced we rebalance it.
- AVL trees are balanced by rotating node either to the left or to the right.

Heaps

Heaps

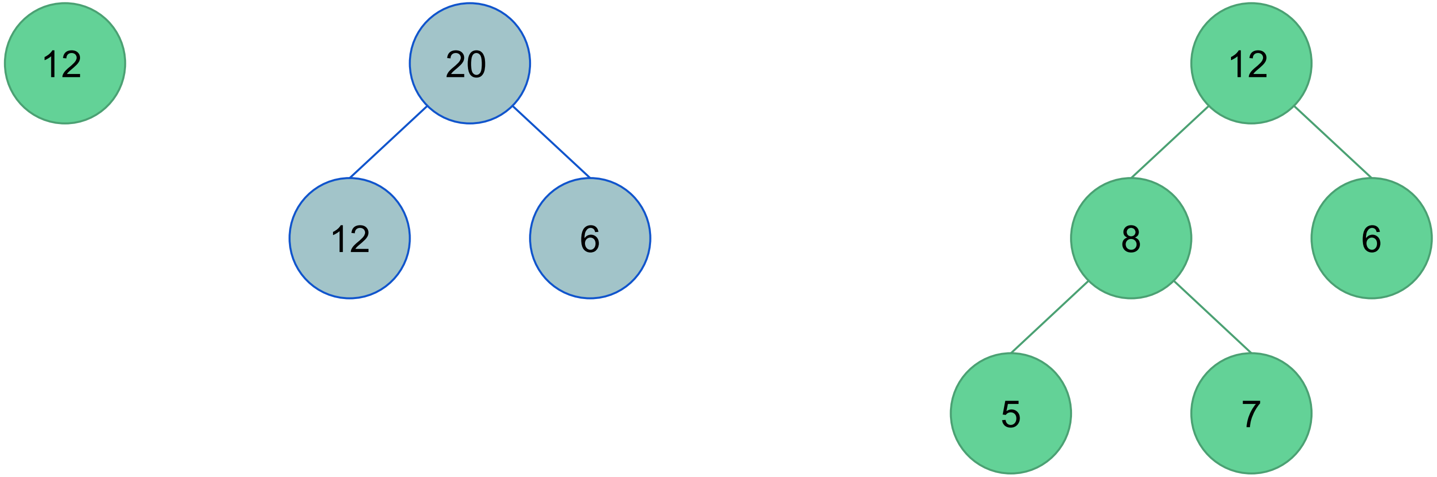
A heap is a binary tree with the following properties:

1. The tree is **complete or almost complete**
2. The key value of each node is **greater than or equal** to the key value in each of its descendants (in a **max-heap**)

*In a **min-heap**, the key value of each node is **smaller or equal** to the key value in each of its descendants*

Heaps

Examples



Operations

- Find-max (or find-min)
- Insertion
- Deletion
- Increase-key or decrease-key

For these operations, we need two basic algorithms:

1. Reheap up
2. Reheap down

Insertion into a heap

Like BST, insertion into a heap **takes place at a leaf** (or a leaf-like node)

Because the heap is a complete or nearly complete binary tree, the node must be placed in **the last leaf level at the first empty position**.

This might cause a situation where the new node’s key is larger than that of its parent, i.e. **the heap structure is broken**.

The **reheap up** operation reorders a “broken” heap by **floating the last element up the tree** until it is in its correct location in the heap.

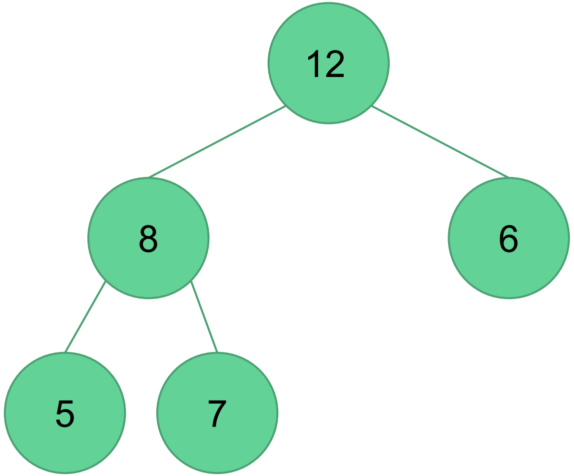
Reheap up

The new node is floated up the tree by **exchanging the child and parent keys and data**

By repeatedly exchanging child/parent keys and data, the data will eventually rise to the correct place in the heap.

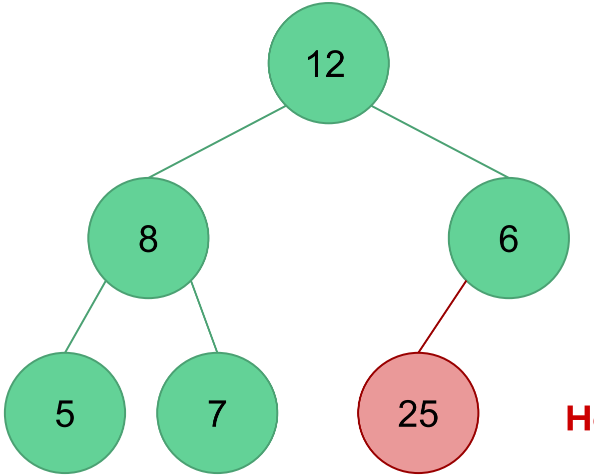
Reheap up

Example: Insert 25 into this heap



Reheap up

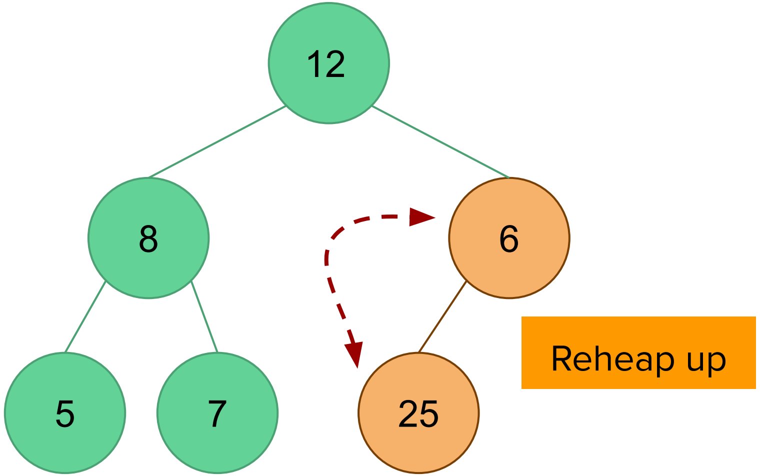
Example: Insert 25 into this heap



Heap structure is broken

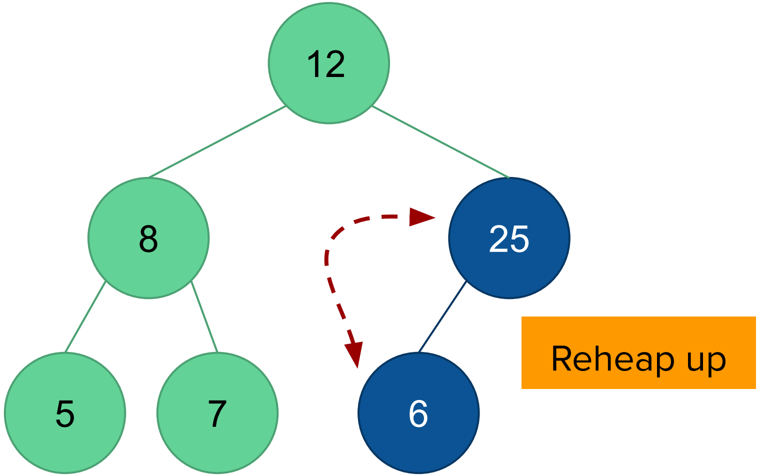
Reheap up

Example: Insert 25 into this heap



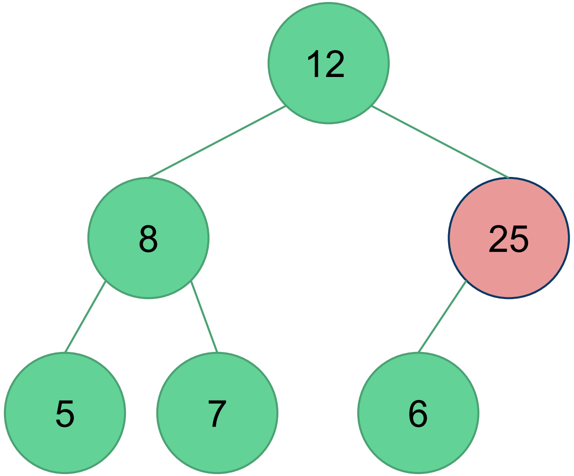
Reheap up

Example: Insert 25 into this heap



Reheap up

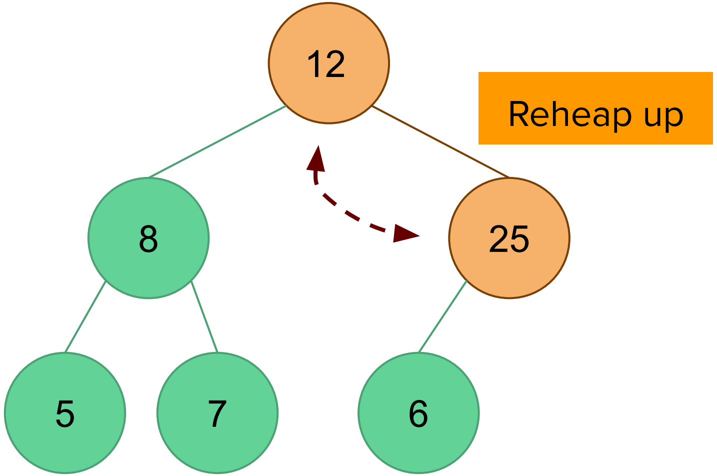
Example: Insert 25 into this heap



Heap structure is broken

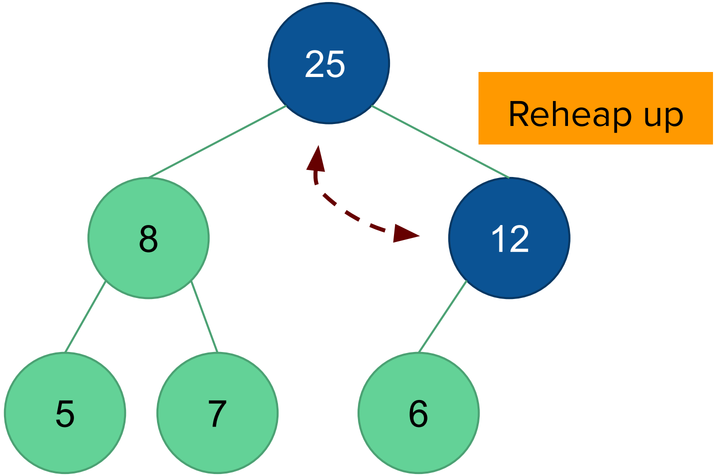
Reheap up

Example: Insert 25 into this heap



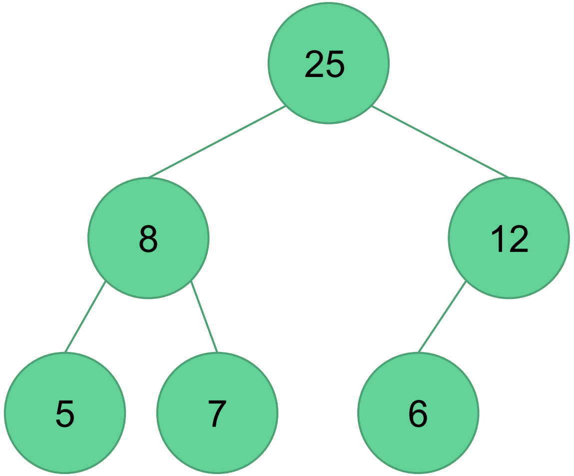
Reheap up

Example: Insert 25 into this heap



Reheap up

Example: Insert 25 into this heap



Deletion in a heap

The standard deletion operation on heaps is to **delete the root node of the heap**.

Deleting the root of the heap leaves us with two disjoint trees.

To correct the situation, we **move the data in the last tree node to the root**.

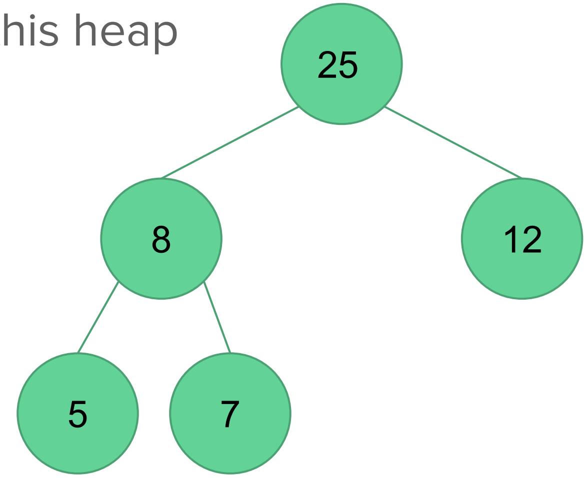
This destroys the tree's heap properties.

Reheap down operation reorders a "broken" heap by **pushing the root down the tree** until it is in its correct position in the heap.

Reheap down

Starting from the root, select the larger of the two children and exchange with the parent.

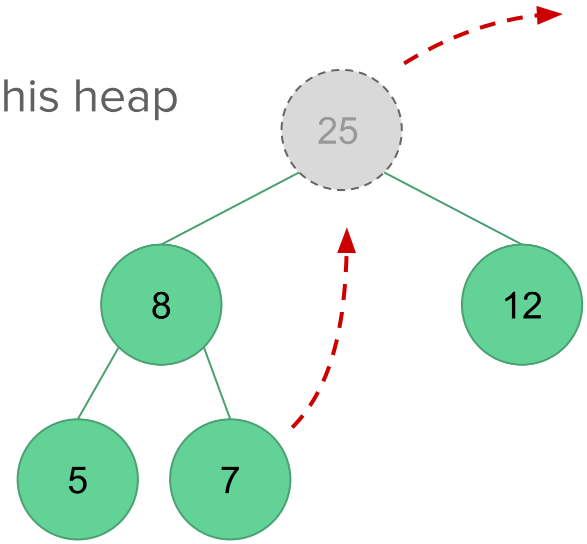
Example: Delete the root from this heap



Reheap down

Starting from the root, select the larger of the two children and exchange with the parent.

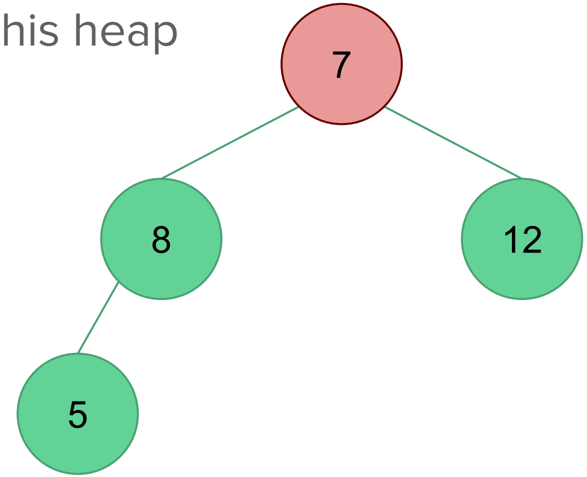
Example: Delete the root from this heap



Reheap down

Starting from the root, select the larger of the two children and exchange with the parent.

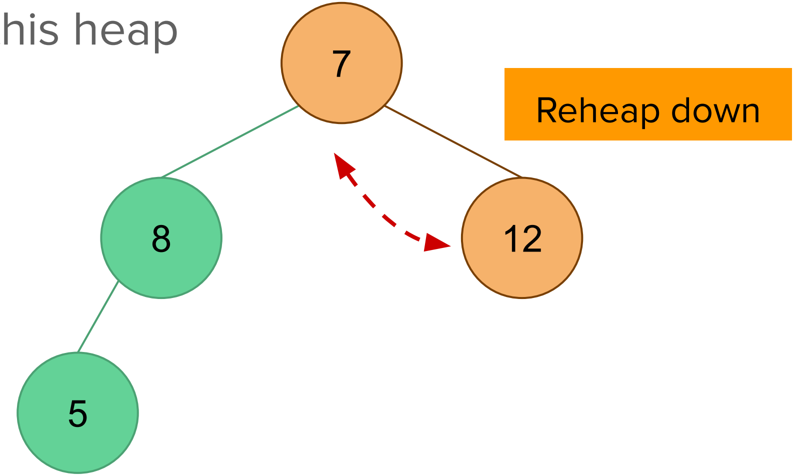
Example: Delete the root from this heap



Reheap down

Starting from the root, select the larger of the two children and exchange with the parent.

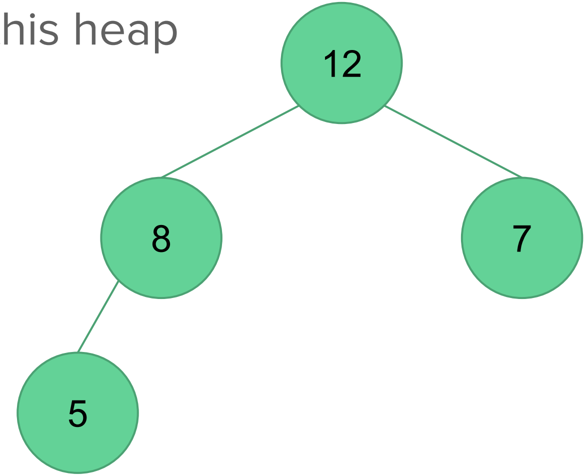
Example: Delete the root from this heap



Reheap down

Starting from the root, select the larger of the two children and exchange with the parent.

Example: Delete the root from this heap



Priority queue

Heaps can be used to implement **priority queues.**

A **priority queue** is a collection of elements such that each element has an associated priority and t**he element to be deleted is the one with the highest (or lowest) priority.**

Operations in a min (ascending) priority queue

Using a min-heap, we can implement an ascending priority queue with the following operations

1. Return an element with minimum priority
   - The minimum element can be found in O(1) time
2. Insert an element with an arbitrary priority
   - This can be done in O(log n) time
3. Delete an element with minimum priority
   - This can be done in O(log n) time

Huffman coding

Huffman coding

Coding: Assigning binary codewords to (blocks of) source symbols

Huffman coding is a lossless data compression algorithm.

Idea:

- Assign variable-length codes to input characters, based on the frequencies of corresponding characters.
- The most frequent character gets the smallest code and the least frequent character gets the largest code.

Huffman coding

There are mainly two major parts in Huffman Coding

1. **Build a Huffman Tree** from input characters.
2. Traverse the Huffman Tree and **assign codes to characters.**

Building a Huffman tree

1. Organize the entire character set into a row, **ordered according to frequency** from highest to lowest (or vice versa). **Each character is now a node at the leaf level of a tree**
2. **Find two nodes with the smallest combined frequency weights and join them to form a third node**, resulting in a simple two-level tree. The weight of the new node is the combined weights of the original two nodes.
3. **Repeat step 2** until all of the nodes, on every level, are combined into a single tree.

Huffman coding example

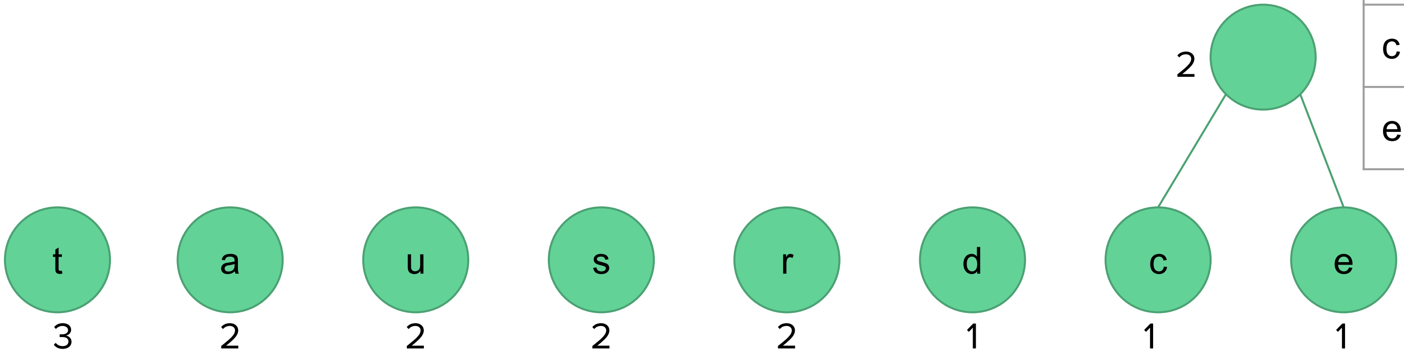
Input character string: "**datastructures**"

We first build the frequency table

character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

Build a Huffman tree:

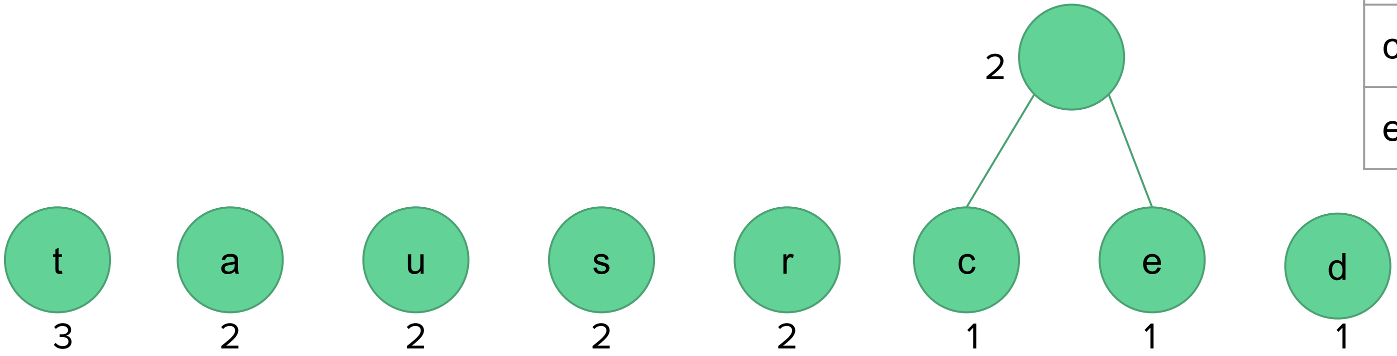


character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

Build a Huffman tree:

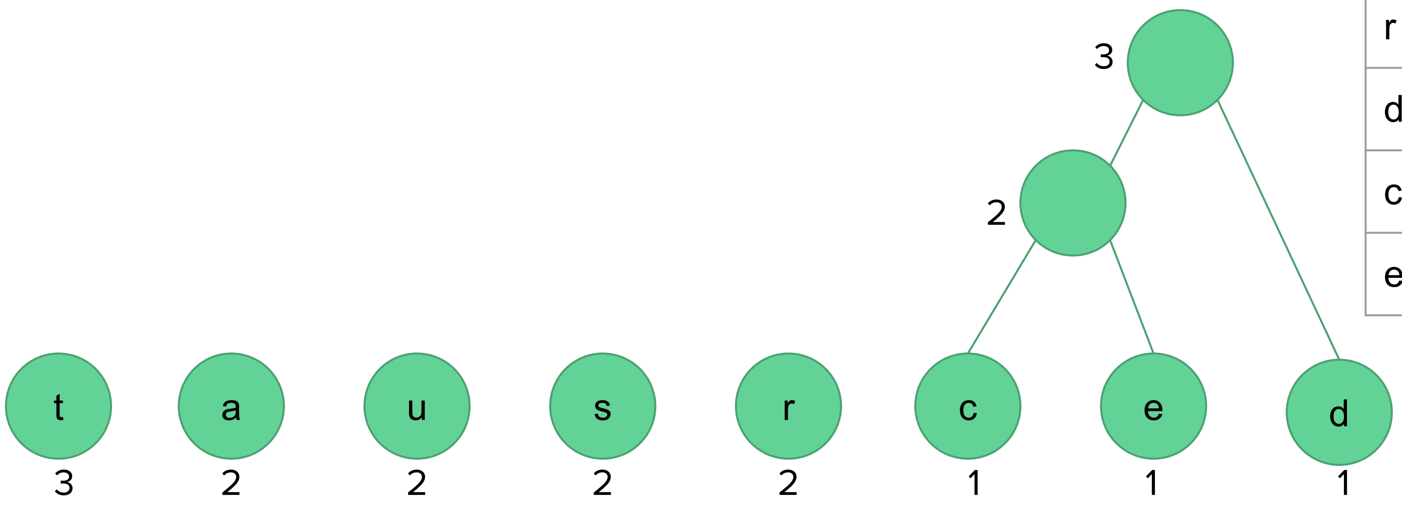
character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1



Huffman coding example

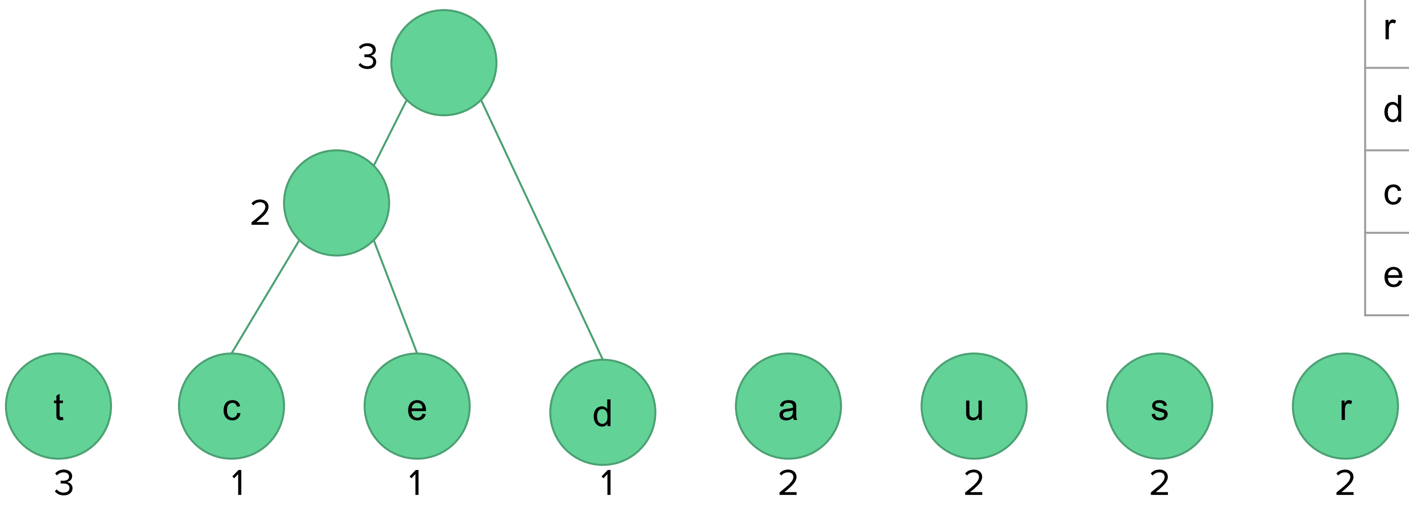
Build a Huffman tree:

character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1



Huffman coding example

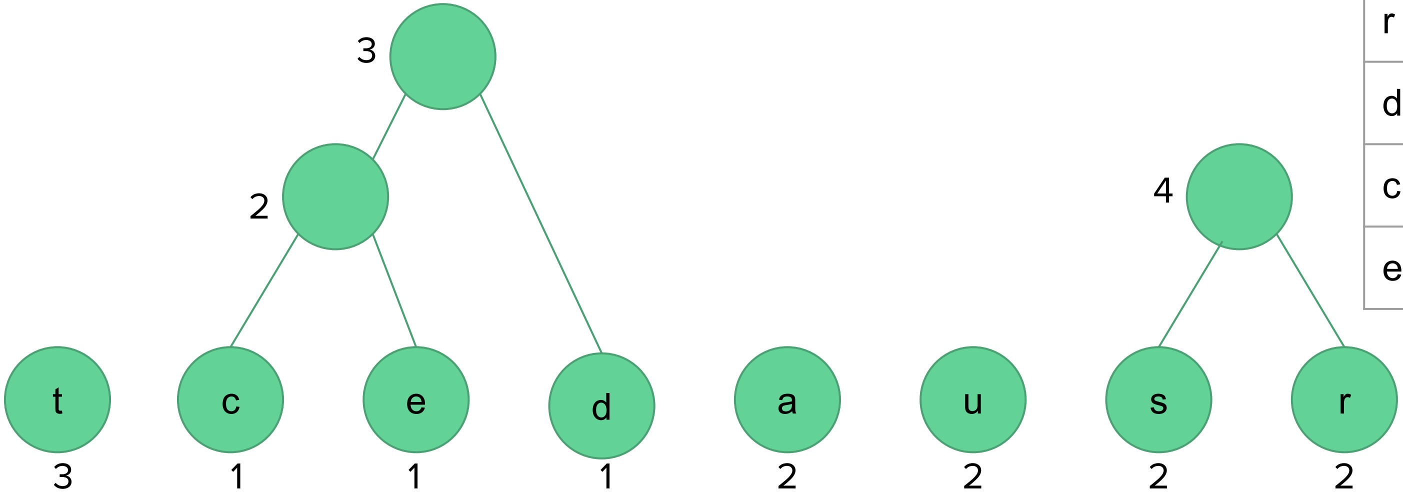
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

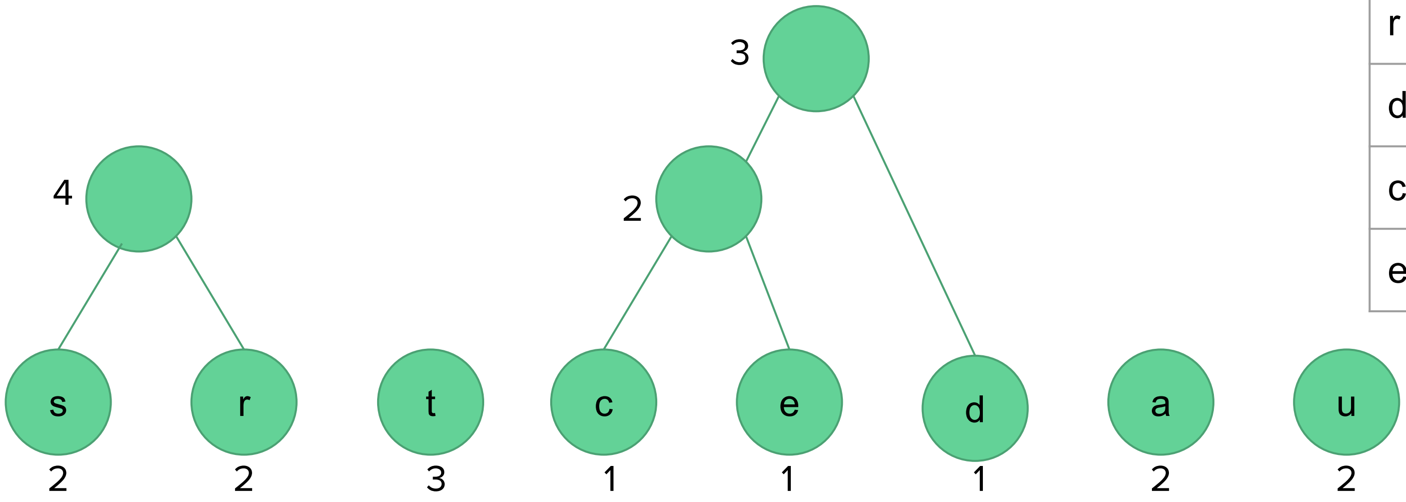
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

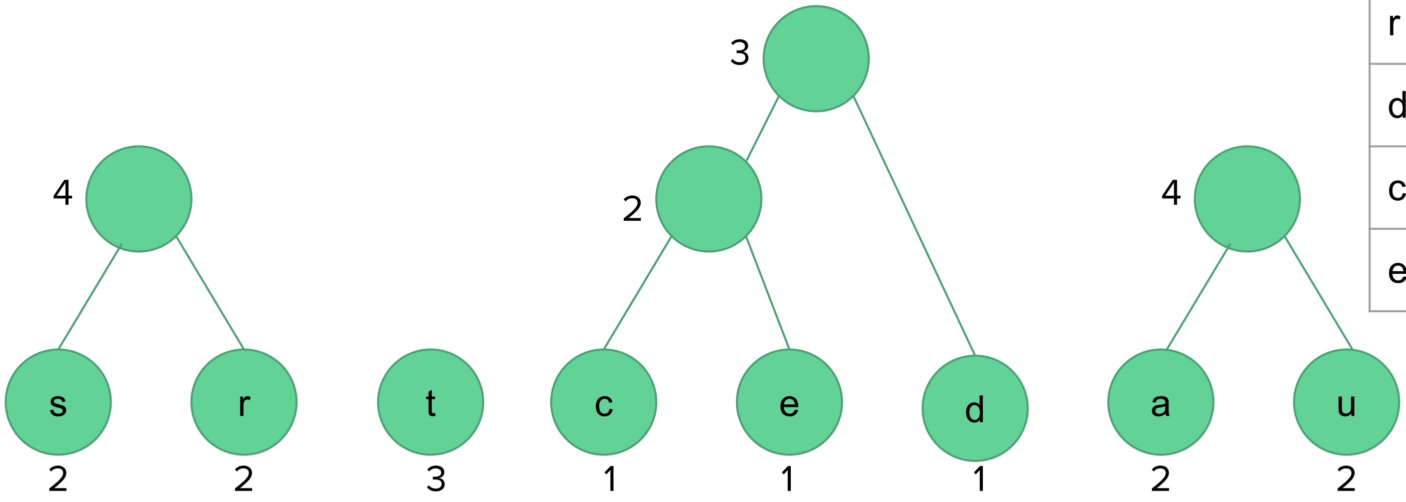
Build a Huffman tree:



character	**frequency**
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

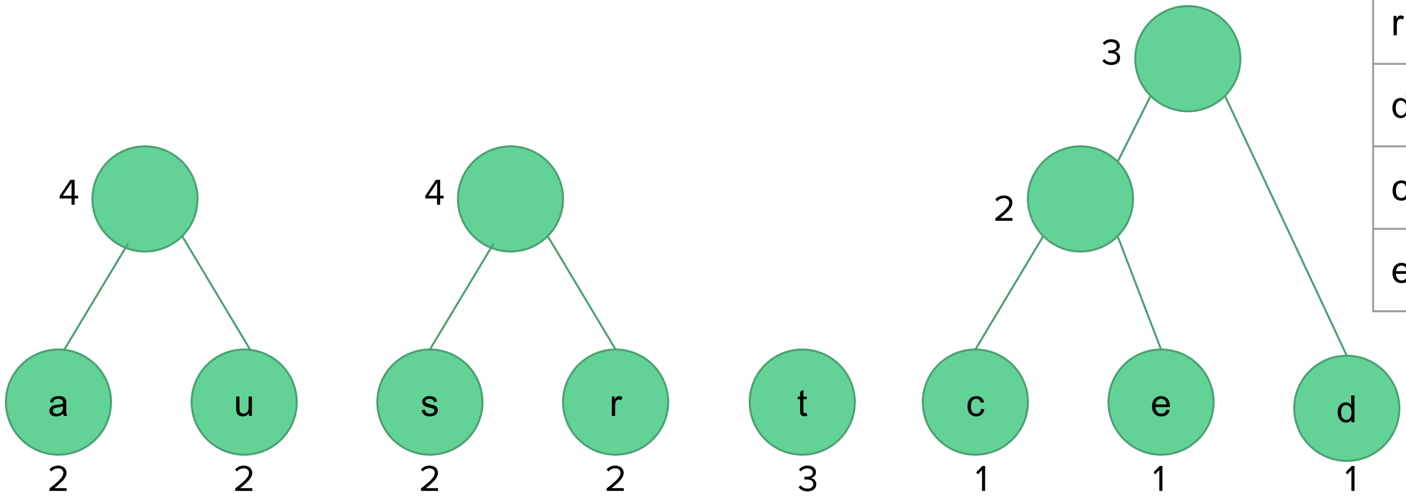
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

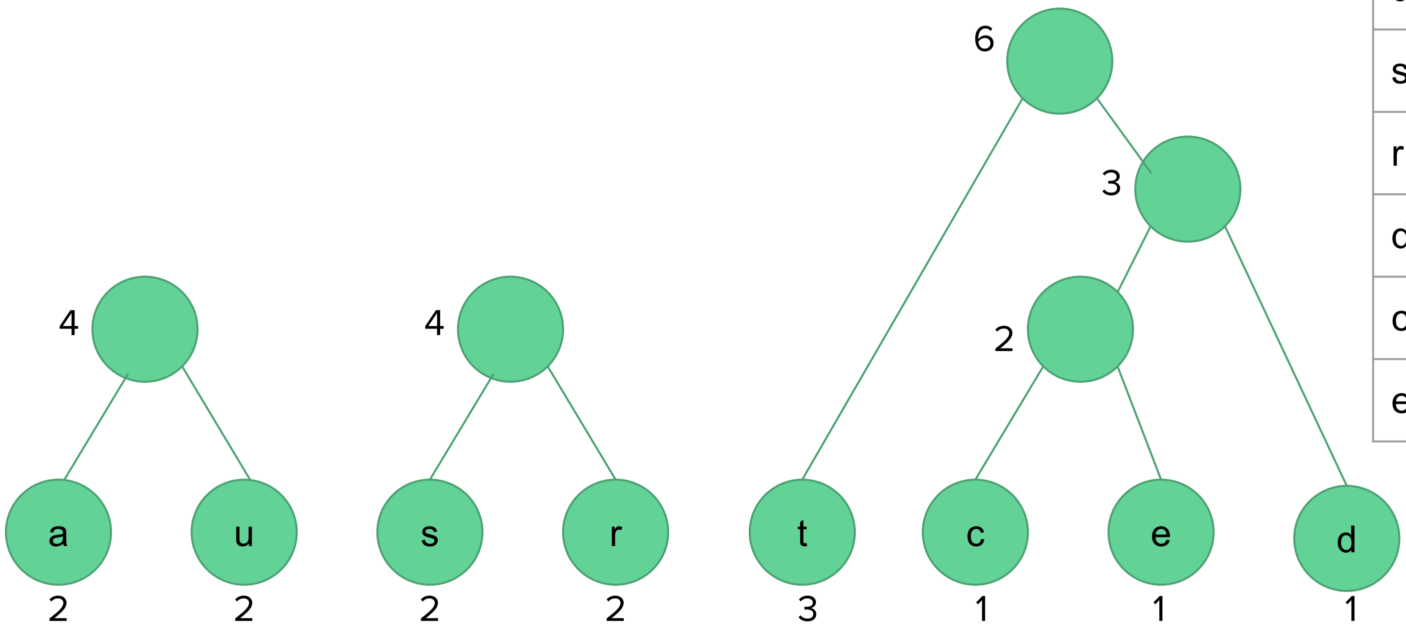
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

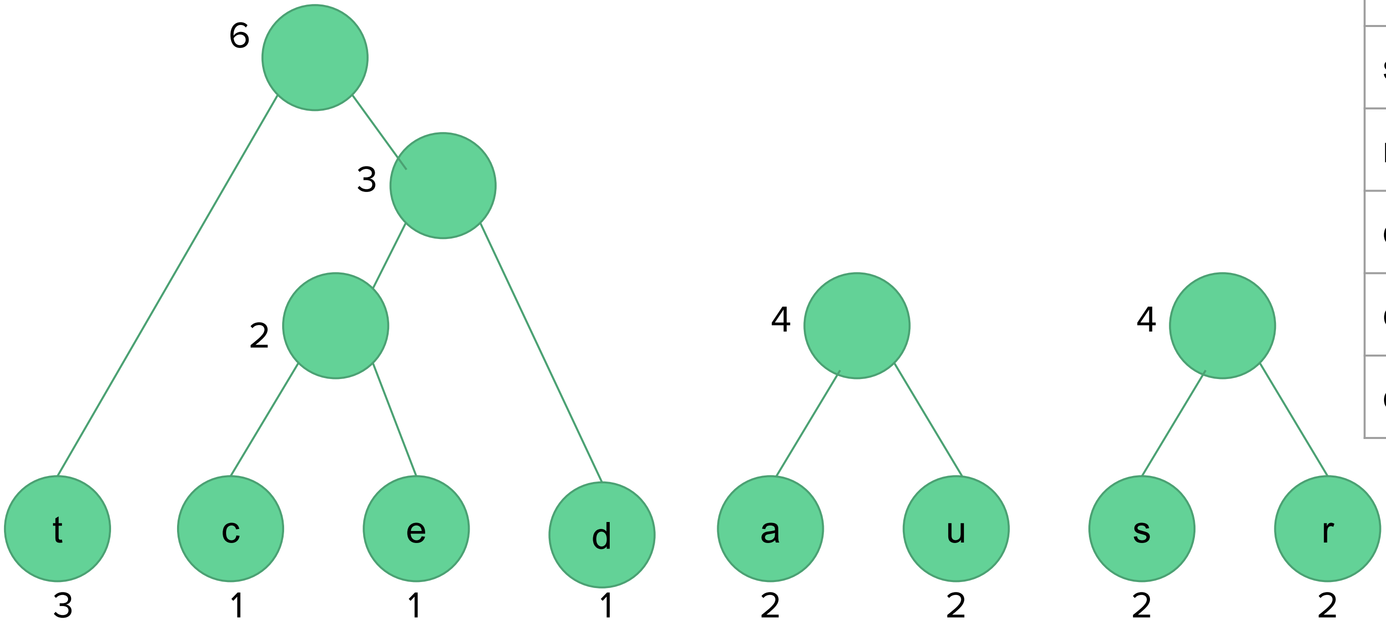
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

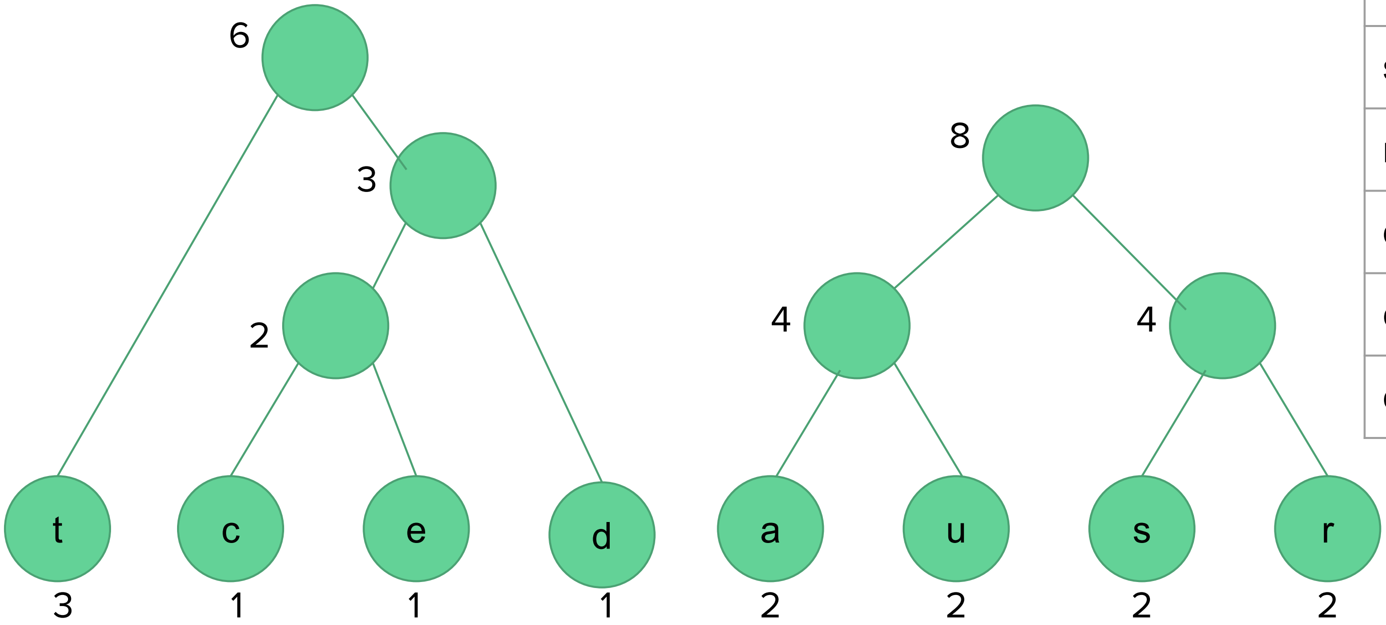
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

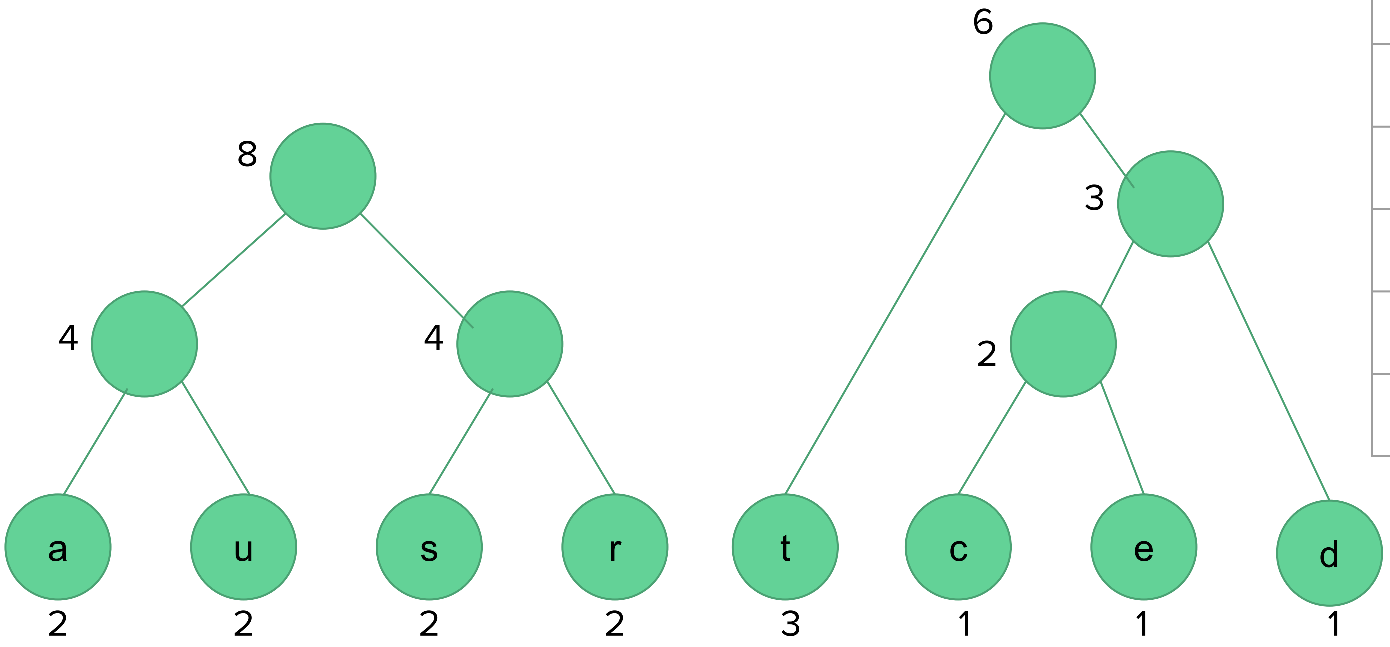
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

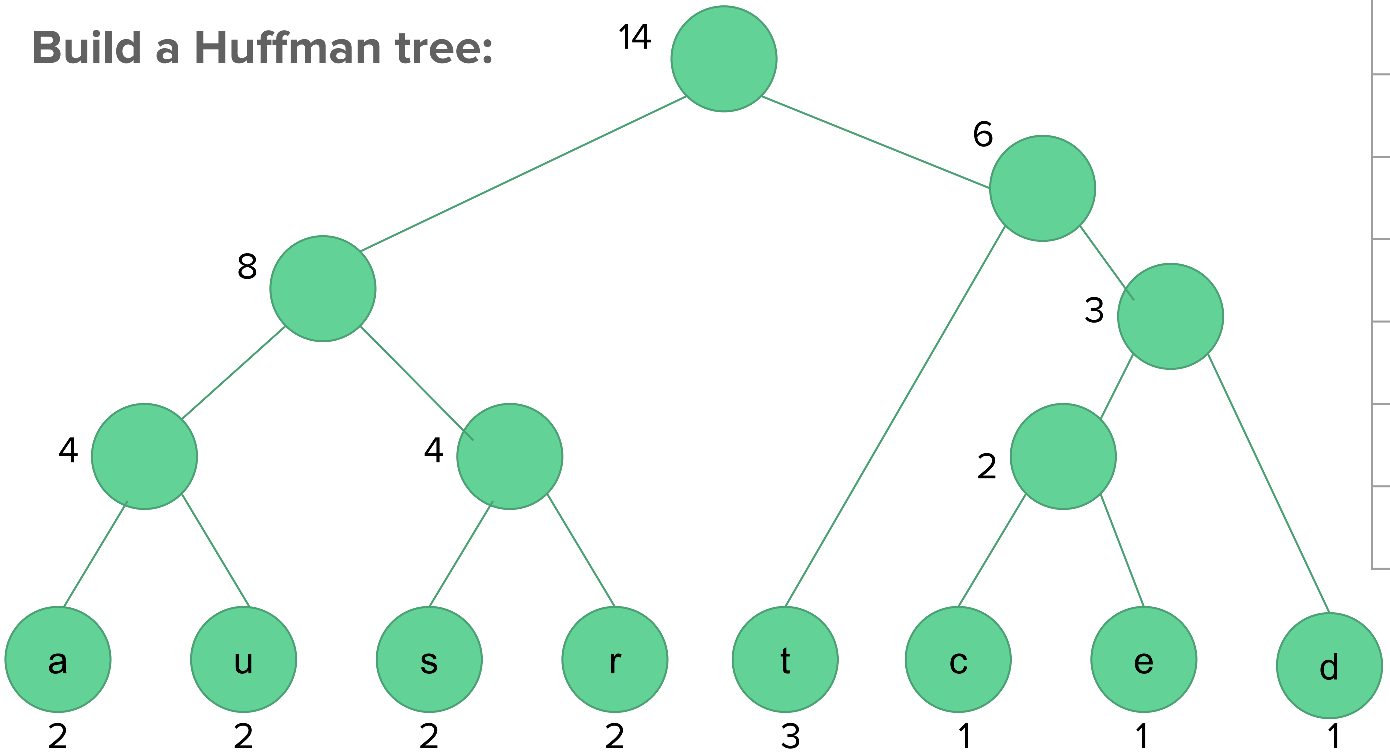
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

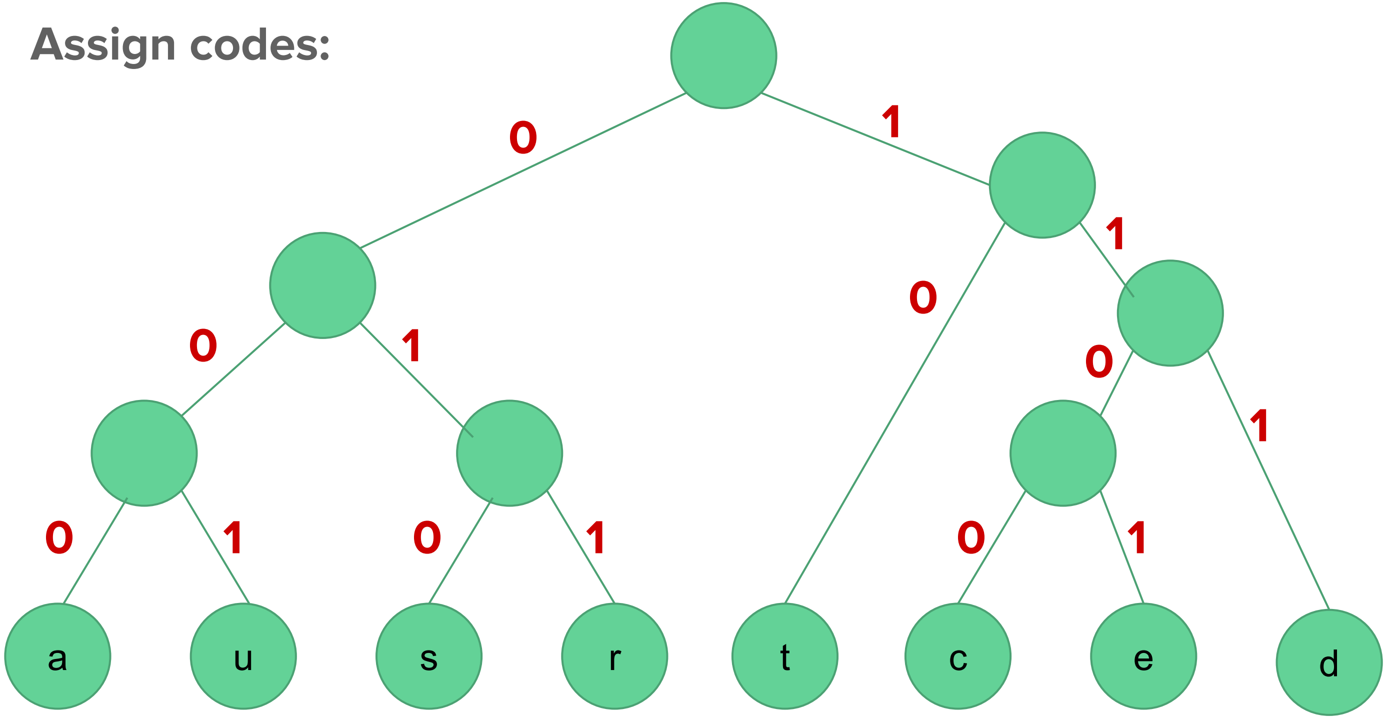
Now we **assign codes** to the tree by **placing a 0 on every left branch and a 1 on every right branch**

A traversal of the tree from root to leaf give the Huffman code for that particular leaf character

These codes are then used to encode the string

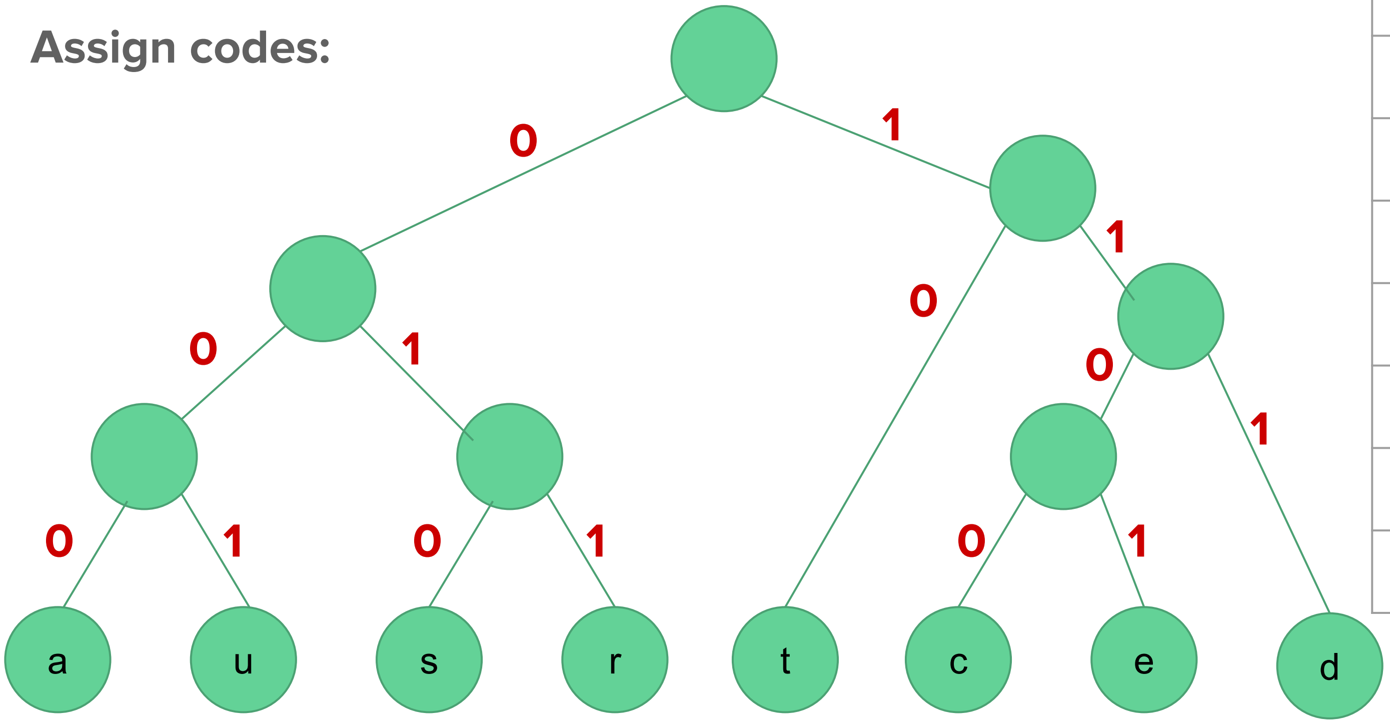
Huffman coding example

Assign codes:



Huffman coding example

Assign codes:



character	Huffman code
t	10
a	000
u	001
s	010
r	011
d	111
c	1100
e	1101

Huffman coding example

Thus “datastructures” turns into

11100010000010100110011100100010111101010

If 8-bit ASCII code had been used instead of Huffman coding, “datastructures” would have been

0110010001100001011101000110000101110011
0111010001110010011101010110001101110100
01110101011100100110010101110011

character	Huffman code	ASCII code
t	10	01110100
a	000	01100001
u	001	01110101
s	010	01110011
r	011	01110010
d	111	01100100
c	1100	01100011
e	1101	01100101

Huffman coding

Uncompression:

Read the file bit by bit

1. Start at the root of the tree
2. If a 0 is read, head left
3. If a 1 is read, head right
4. When a leaf is reached, decode that character and start over again at the root of the tree

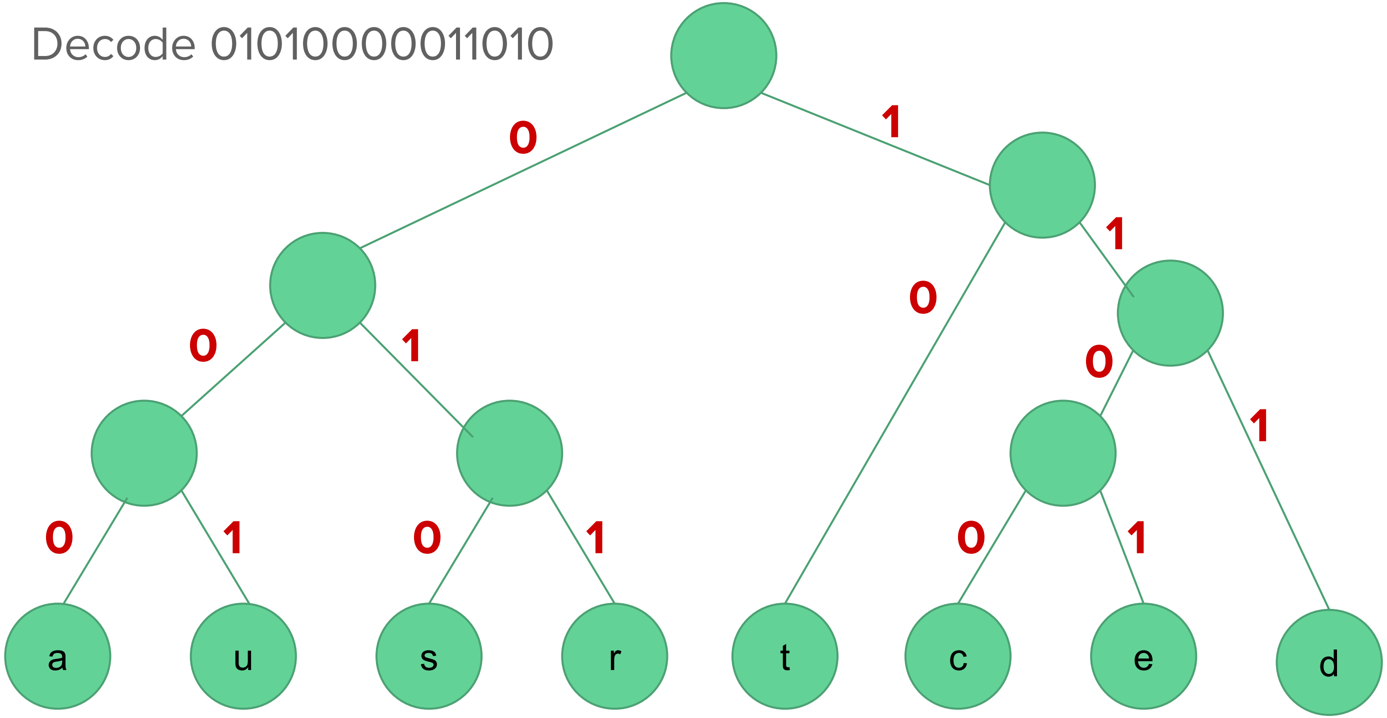
Huffman coding

Uncompression example:

Decode 01010000011010 using the previous Huffman tree

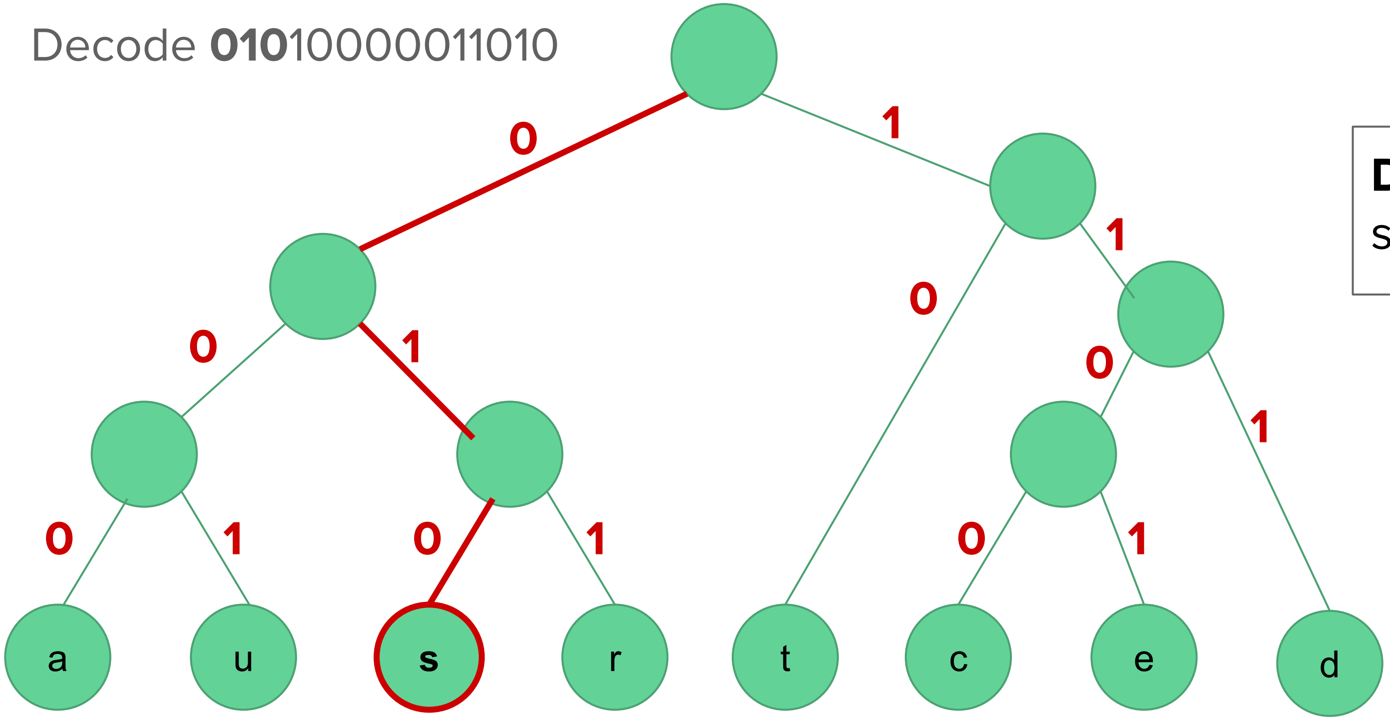
Uncompression example

Decode 01010000011010



Uncompression example

Decode **0101**0000011010

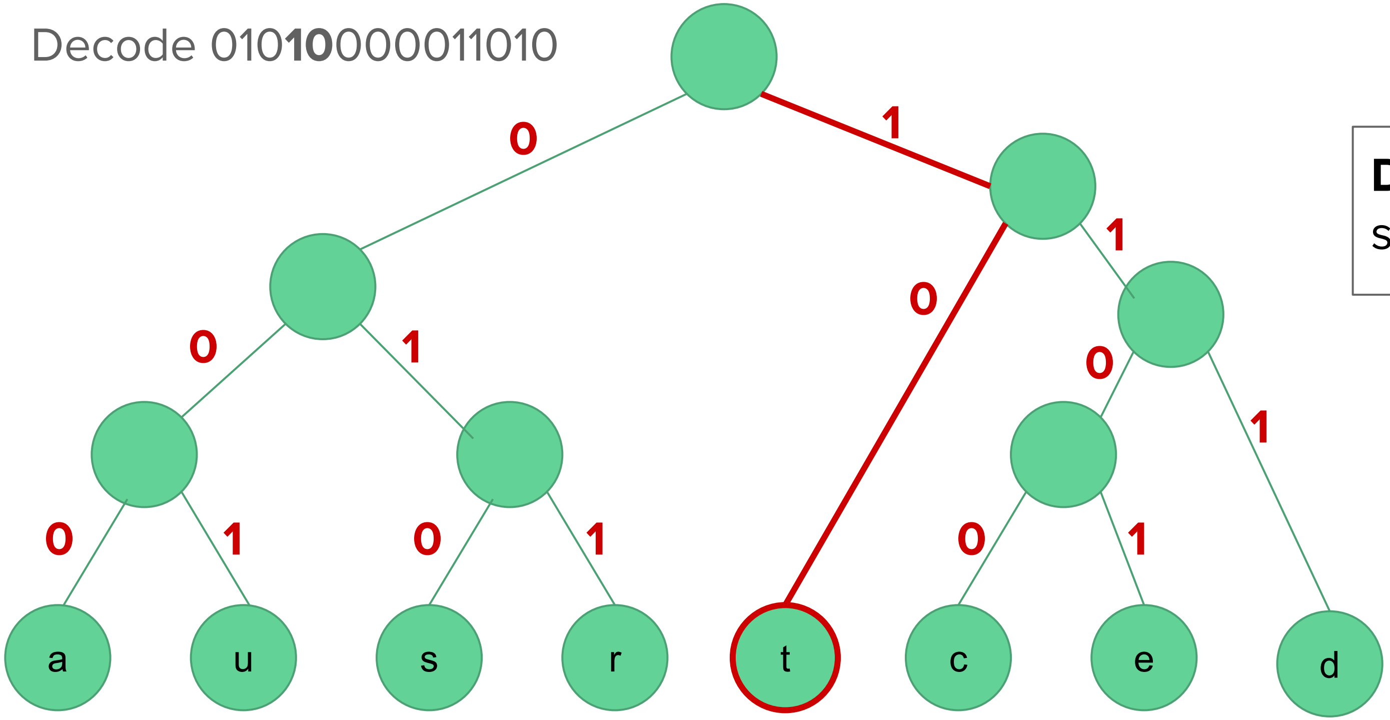


Decoded string:

s

Uncompression example

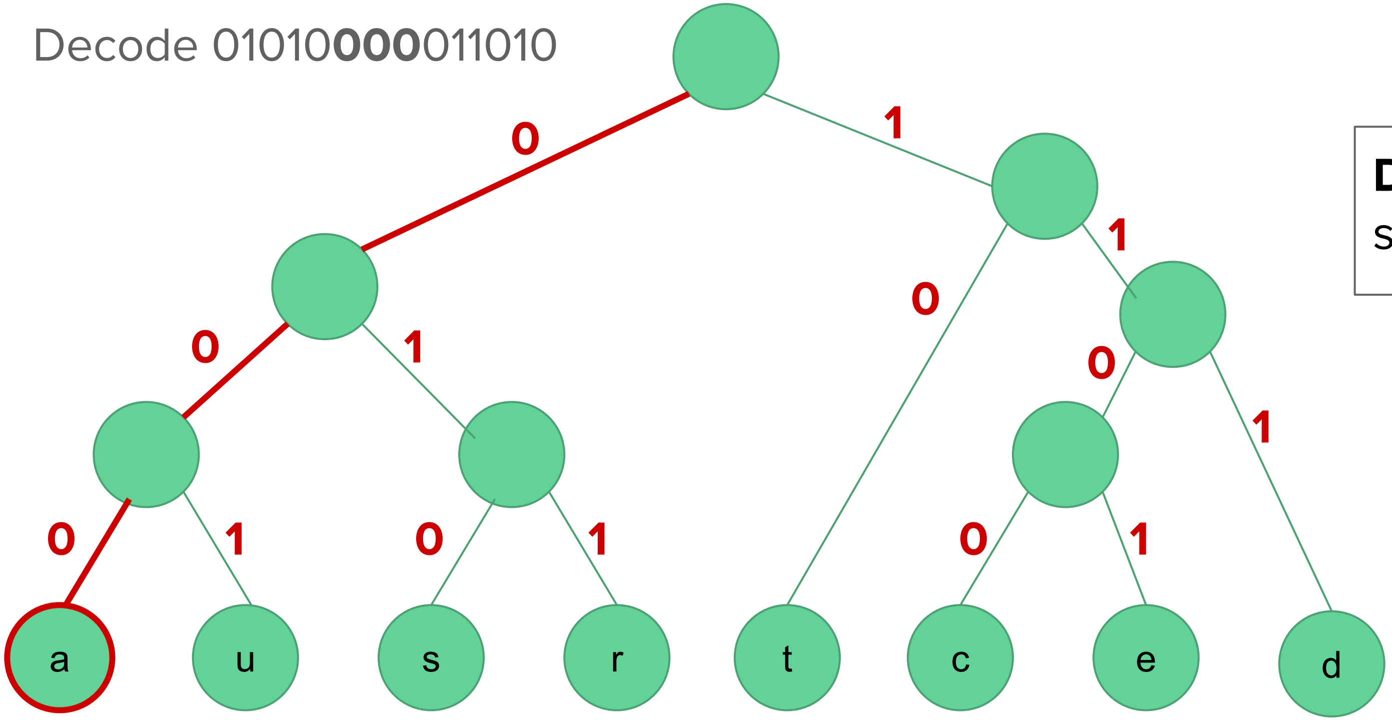
Decode 0101**10**000011010



Decoded string:
st

Uncompression example

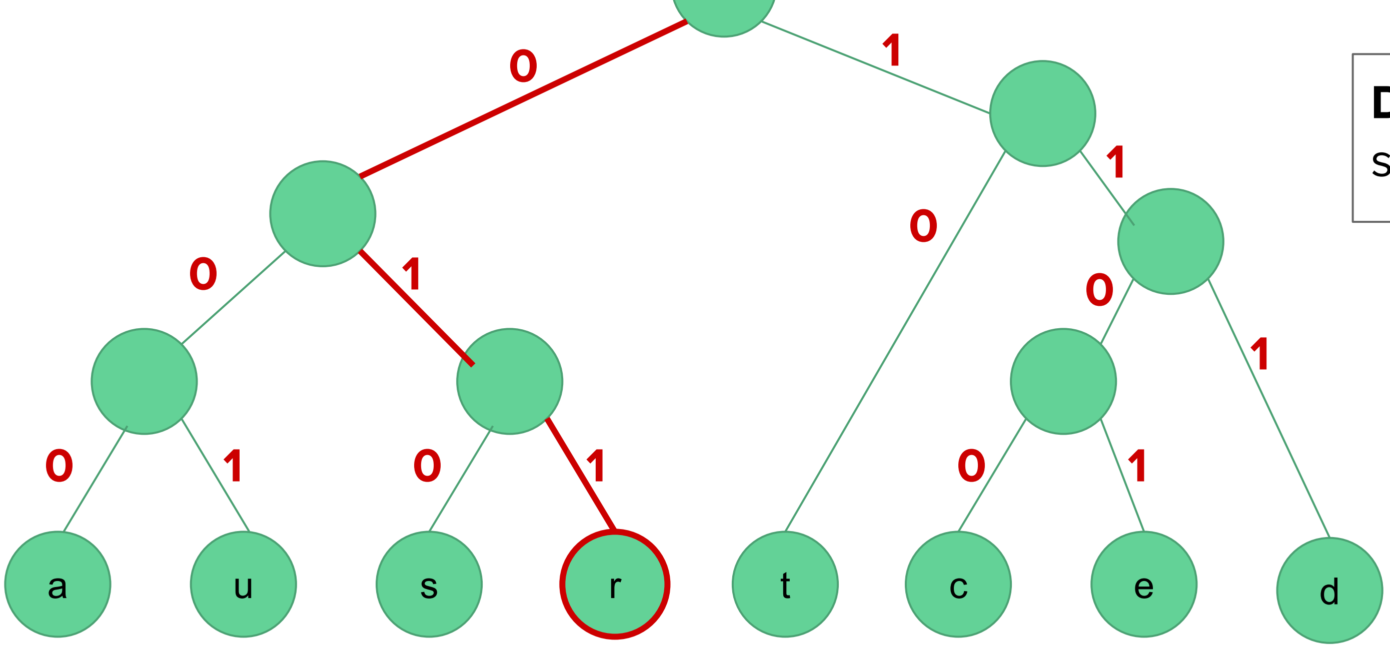
Decode 01010**000**011010



Decoded string:
sta

Uncompression example

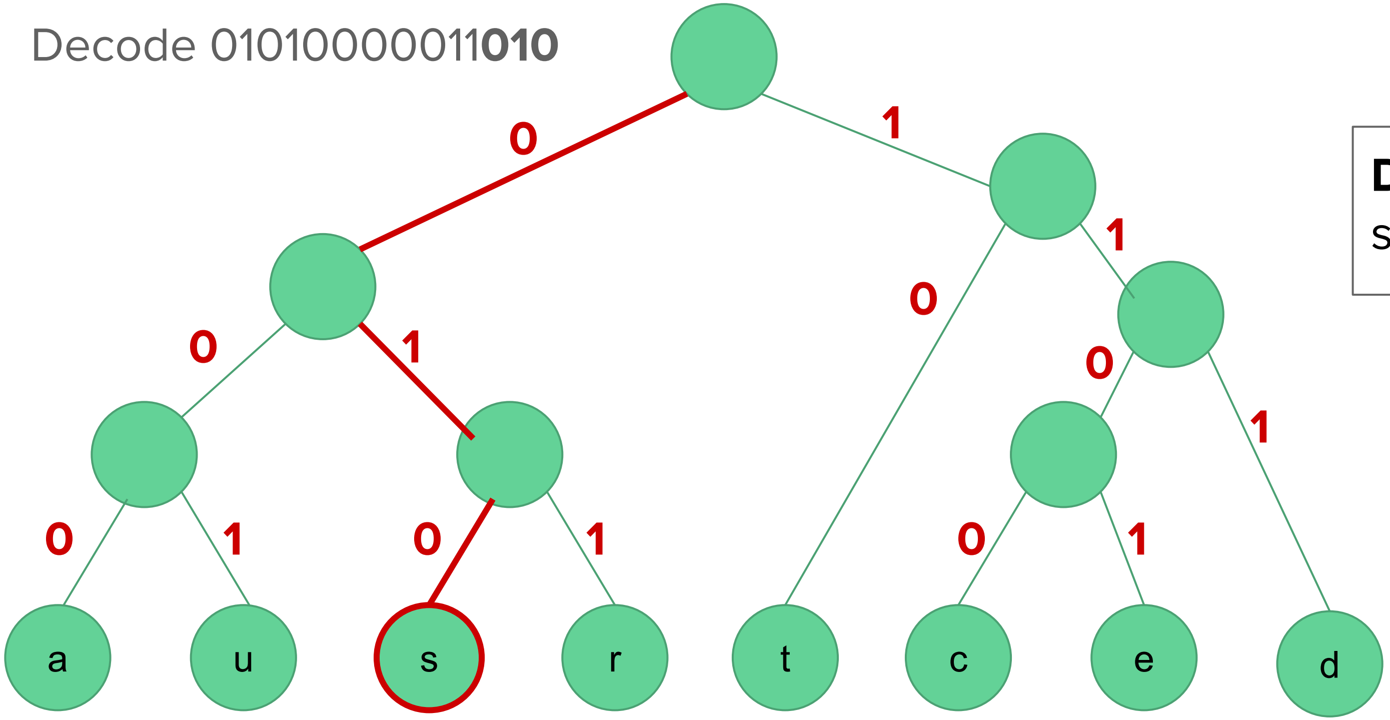
Decode 01010000**011**010



Decoded string:
star

Uncompression example

Decode 01010000011**010**



Decoded string:
stars