Chapter 3:
Algorithm Strategies

Contents

Brute-force

Brute-force

- A straightforward approach to solving a problem, usually directly based on the problem statement and definition
- Systematically enumerates all possible candidates for the solution and checks whether each candidate satisfies the problem statement

Brute-force algorithms

Selection sort

Based on sequentially finding the smallest elements

Bubble Sort

Based on consecutive swapping adjacent pairs. This causes a slow migration of the smallest elements to the left of the array.

Brute-force algorithms

Bubble Sort

Based on consecutive swapping adjacent pairs. This causes a slow migration of the smallest elements to the left of the array.

```
procedure BubbleSort(A[0 ... n − 1])
    for i ← 0 to n − 2 do
        for j ← 0 to n − 2 − i do
            if  A[j + 1] < A[j] then
                swap A[j+1] and A[j]
            end if
        end for
    end for
end procedure
```

Brute-force algorithms

Sequential Searching

```
procedure SequentialSearch(A[0 ... n-1], K)        ▷ K is the search key
    i ← 0
    while i < n and A[i] ≠ K  do
        i ← i + 1
    end while
    if i < n then
        return i
    else
        return -1
    end if
end procedure
```

Brute-force algorithms

Brute-Force String Matching: Searching for a pattern, P[0...m-1], in text, T[0...n-1]

```
procedure BruteForceStringMatch(T[0...n − 1], P[0...m − 1])
    for i ← 0 to n − m do
        j ← 0
        while j < m and P[j] == T[i + j] do
            j ← j + 1
        end while
        if  j == m then
            return i
        end if
    end for
    return -1
end procedure
```

The Greedy method

The greedy method

- Builds up a solution piece by piece, always choosing the next piece that looks best at the moment
- The main idea is to make locally optimal choice in the hope that this choice will lead to a globally optimal solution
- Greedy algorithms do not always yield optimal solutions, but for many problems they do

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.
- Instead of using ASCII codes, store the more frequently occurring characters using fewer bits and less frequently occurring characters using more bits.

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.

Huffman coding

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

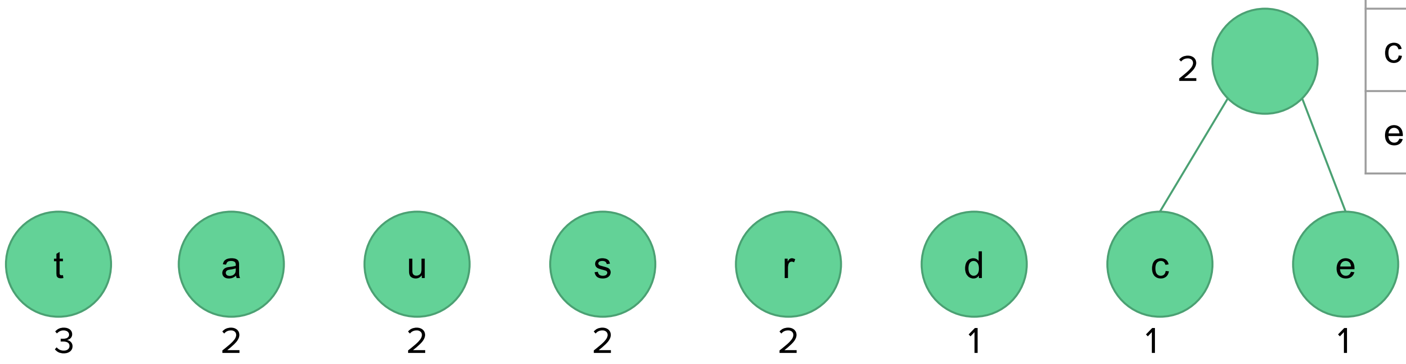
Input character string: "**datastructures**"

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:

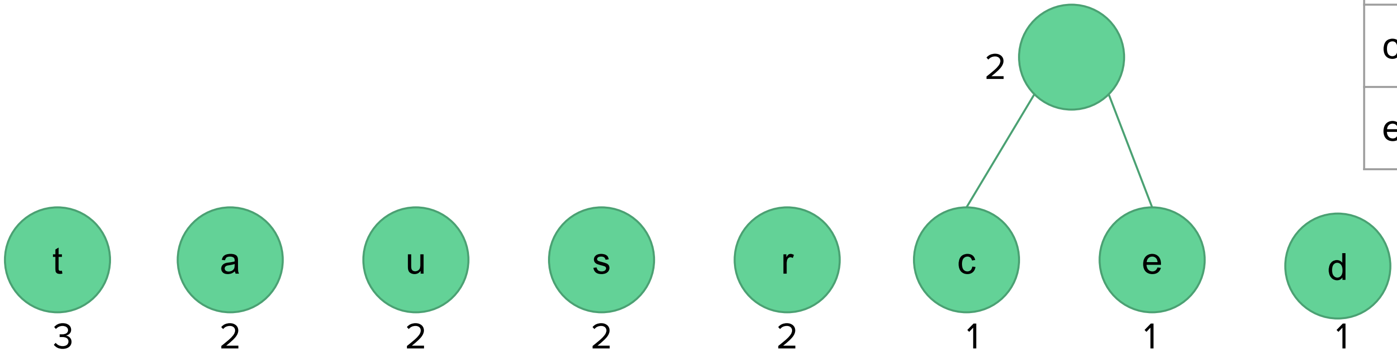


character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

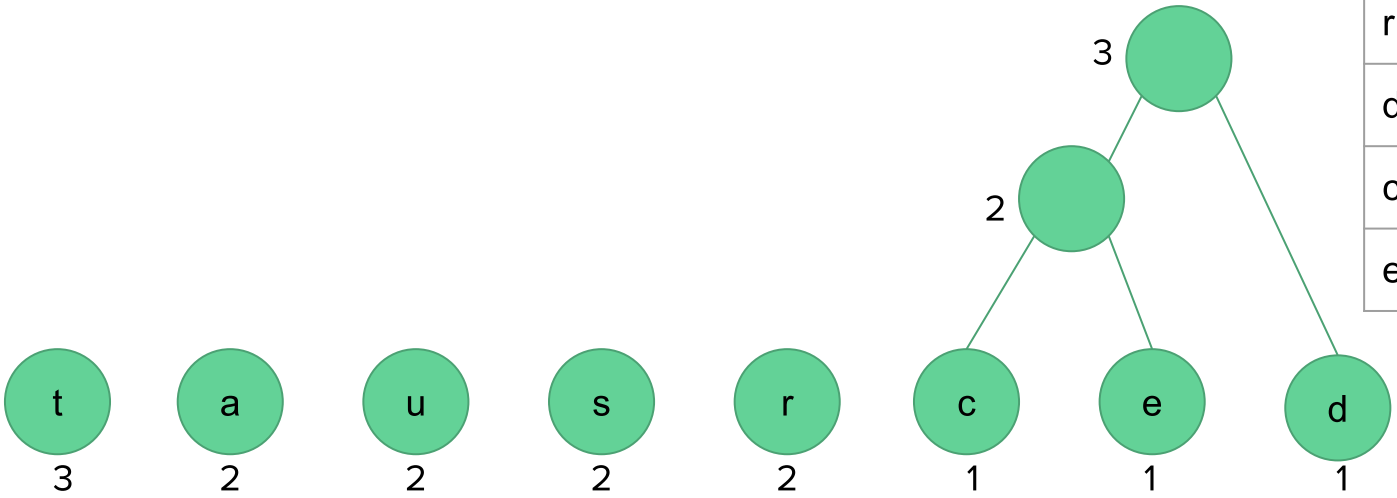
Build a Huffman tree:

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



Huffman coding example

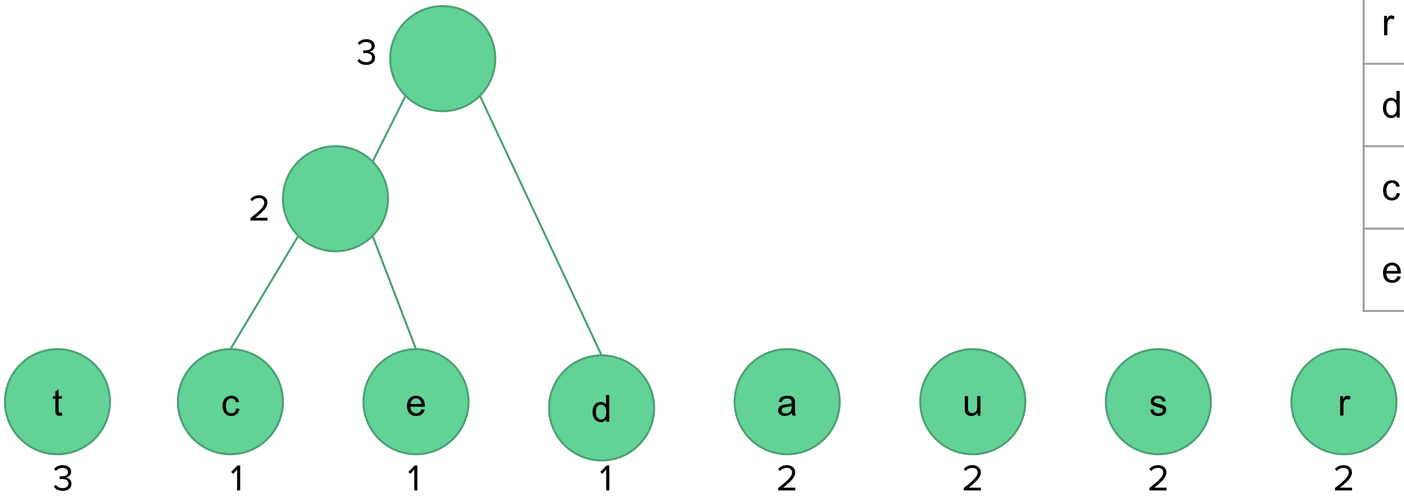
Build a Huffman tree:



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:

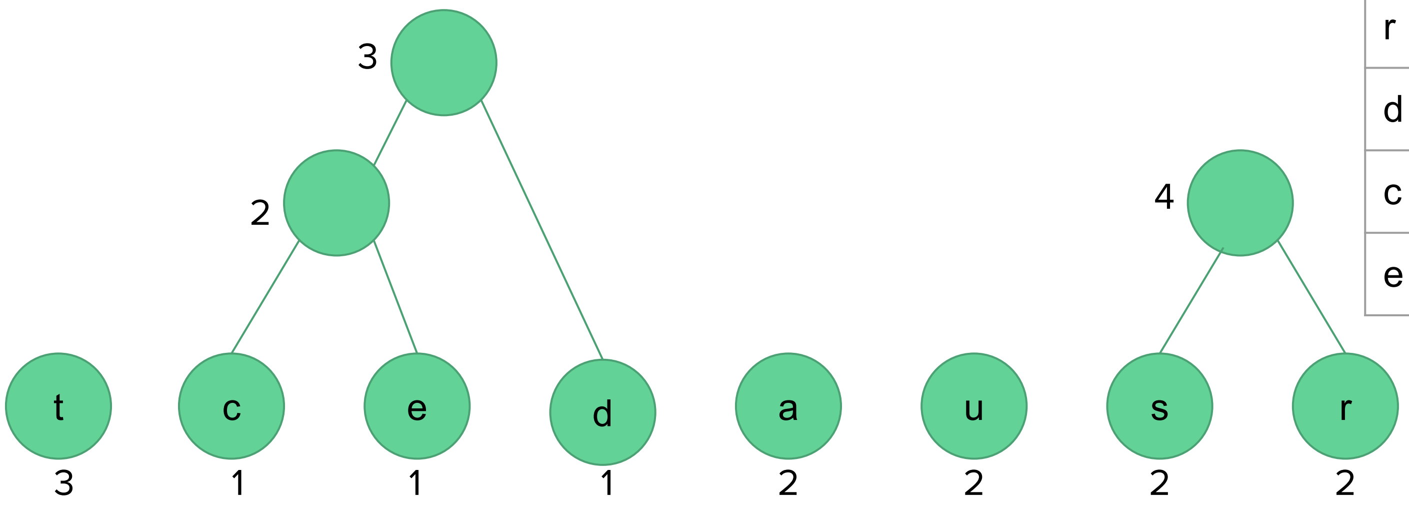


character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

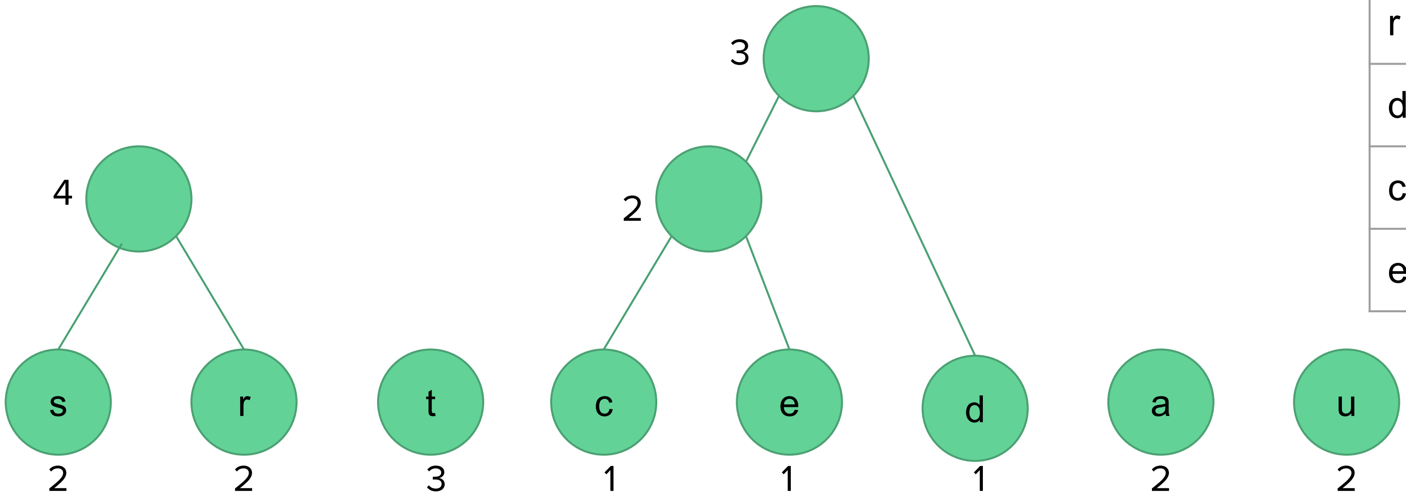
Build a Huffman tree:

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



Huffman coding example

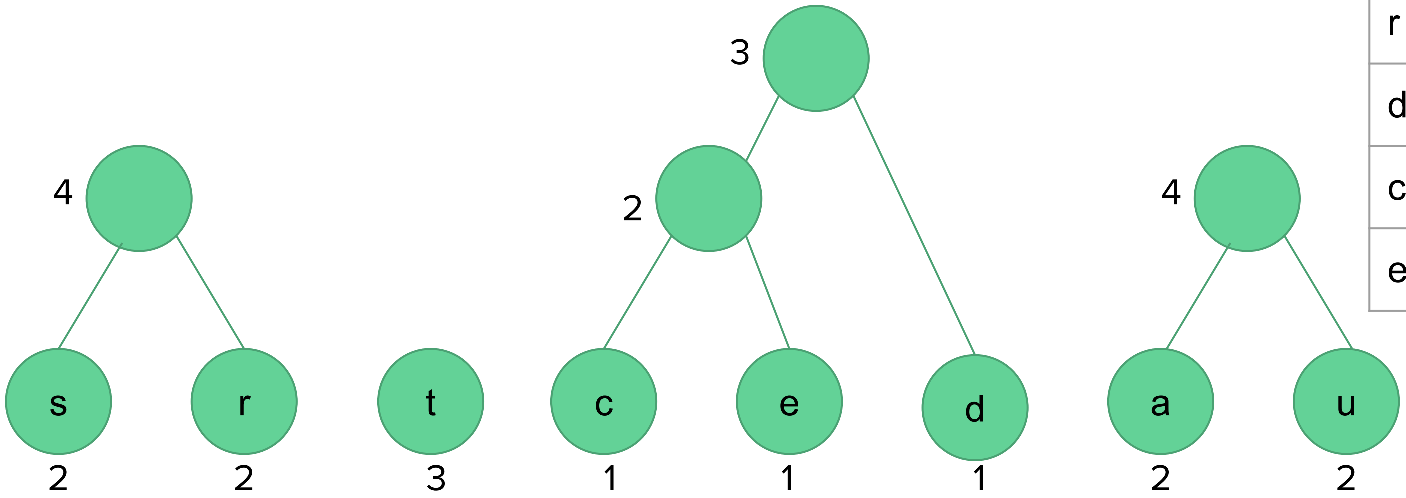
Build a Huffman tree:



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

Huffman coding example

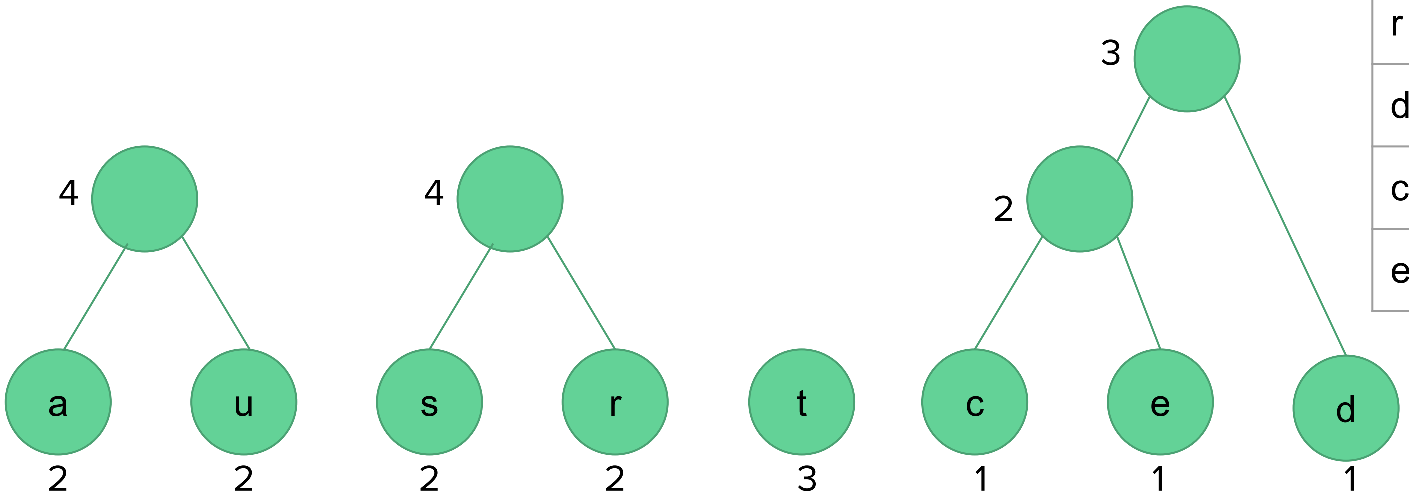
Build a Huffman tree:



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

Huffman coding example

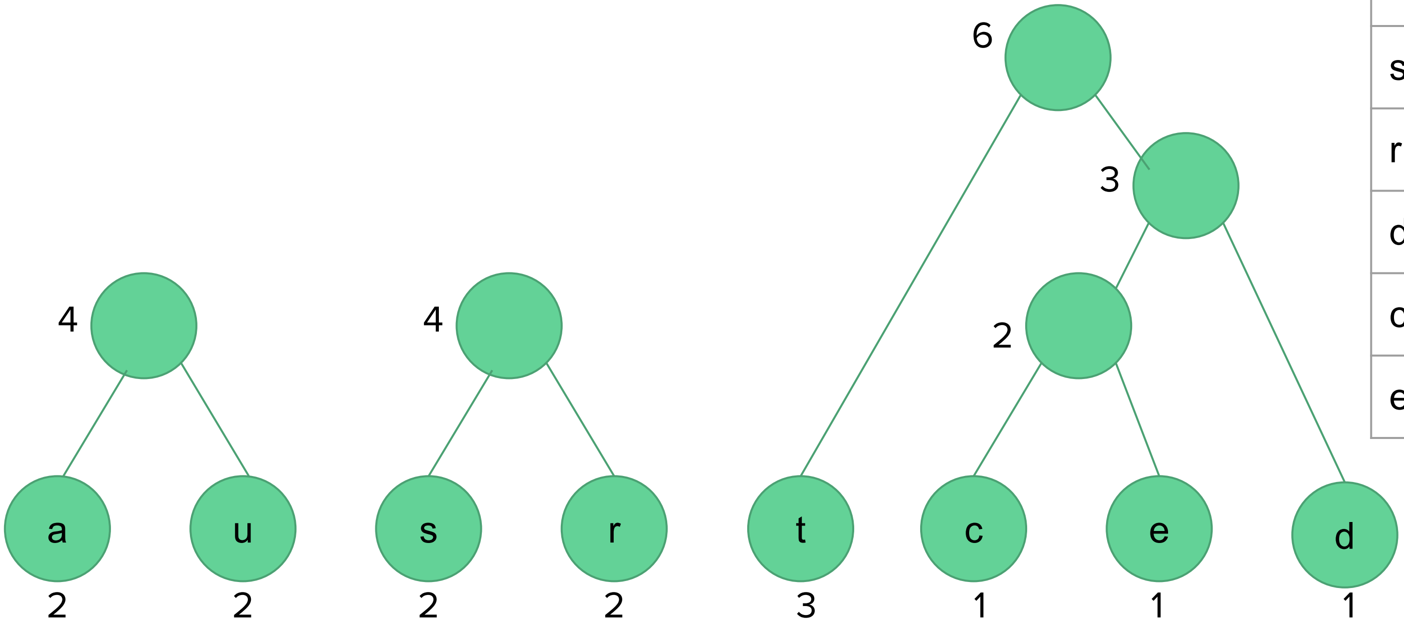
Build a Huffman tree:



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

Huffman coding example

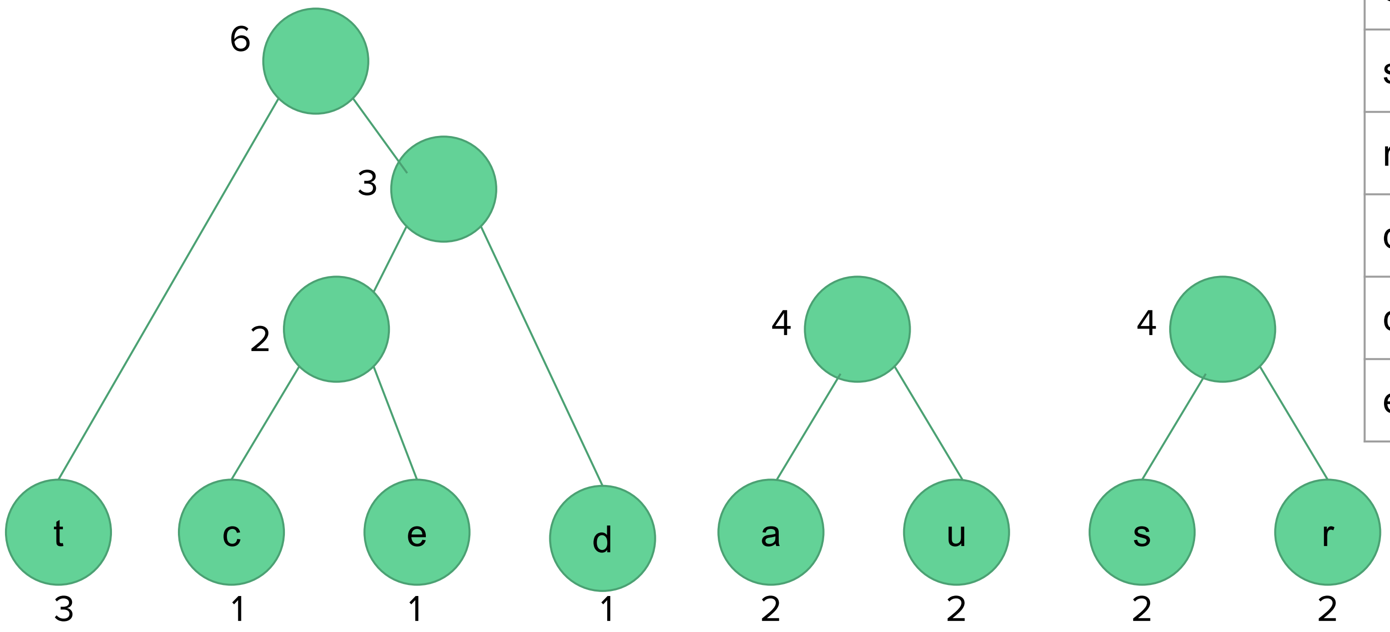
Build a Huffman tree:



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

Huffman coding example

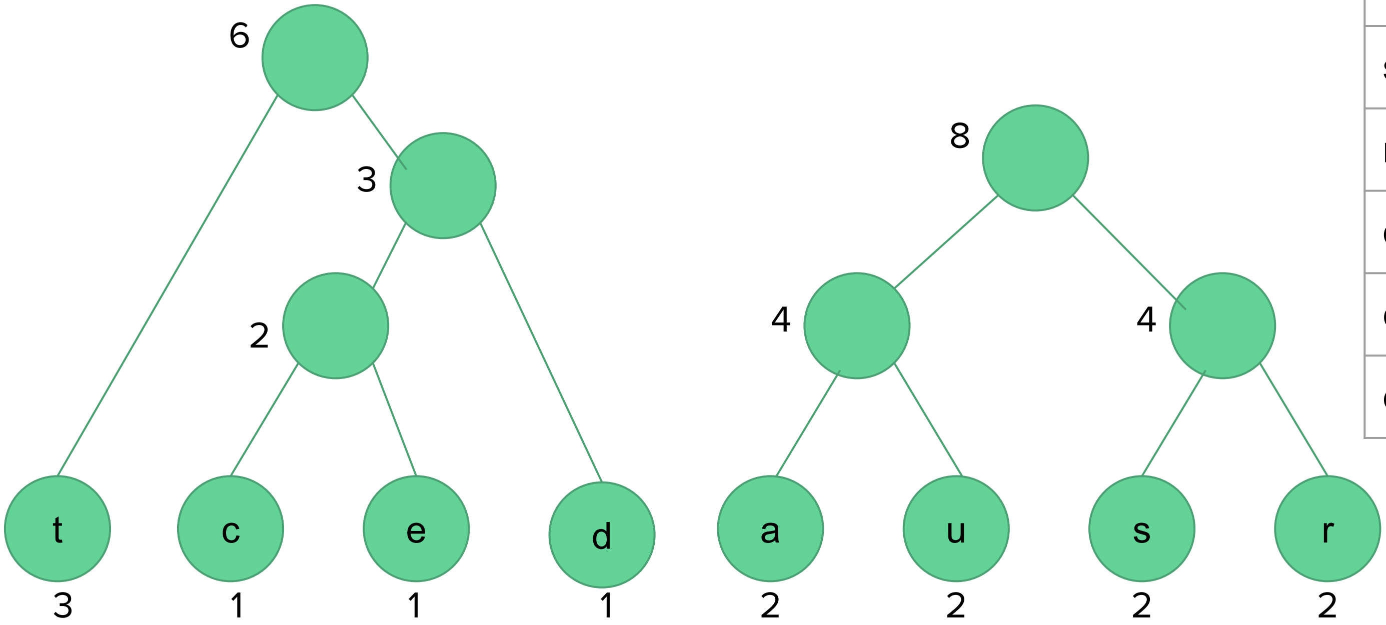
Build a Huffman tree:



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

Huffman coding example

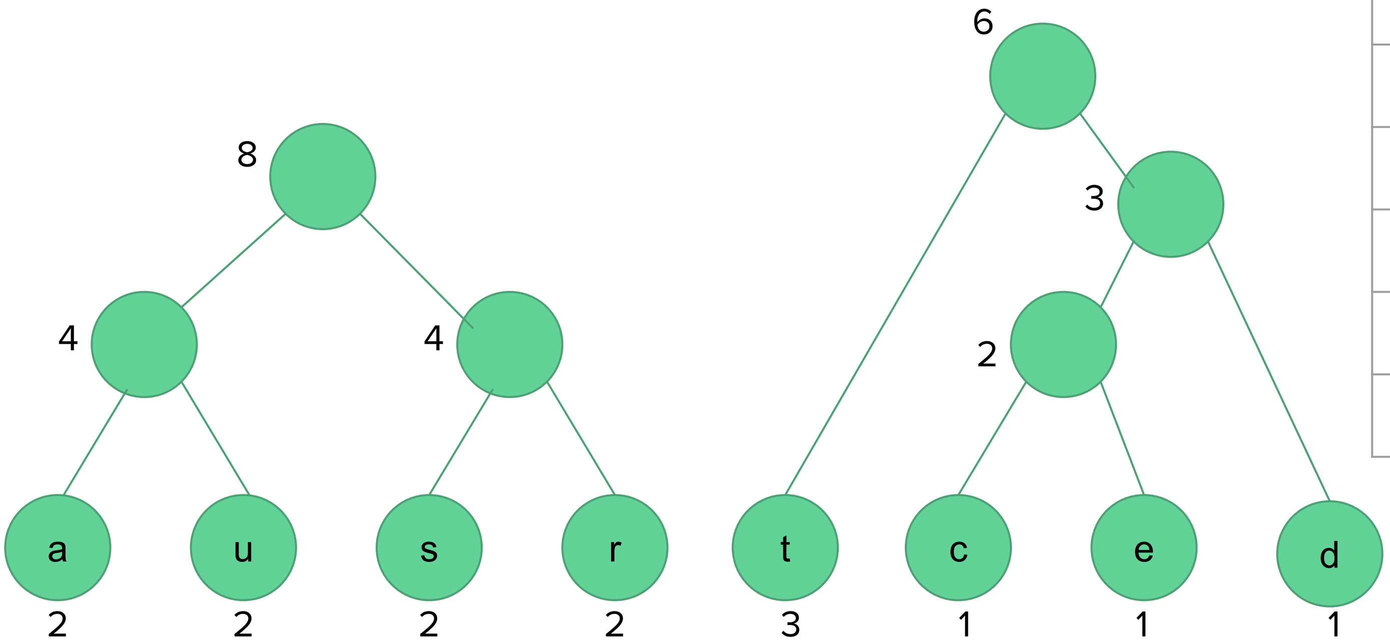
Build a Huffman tree:



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

Huffman coding example

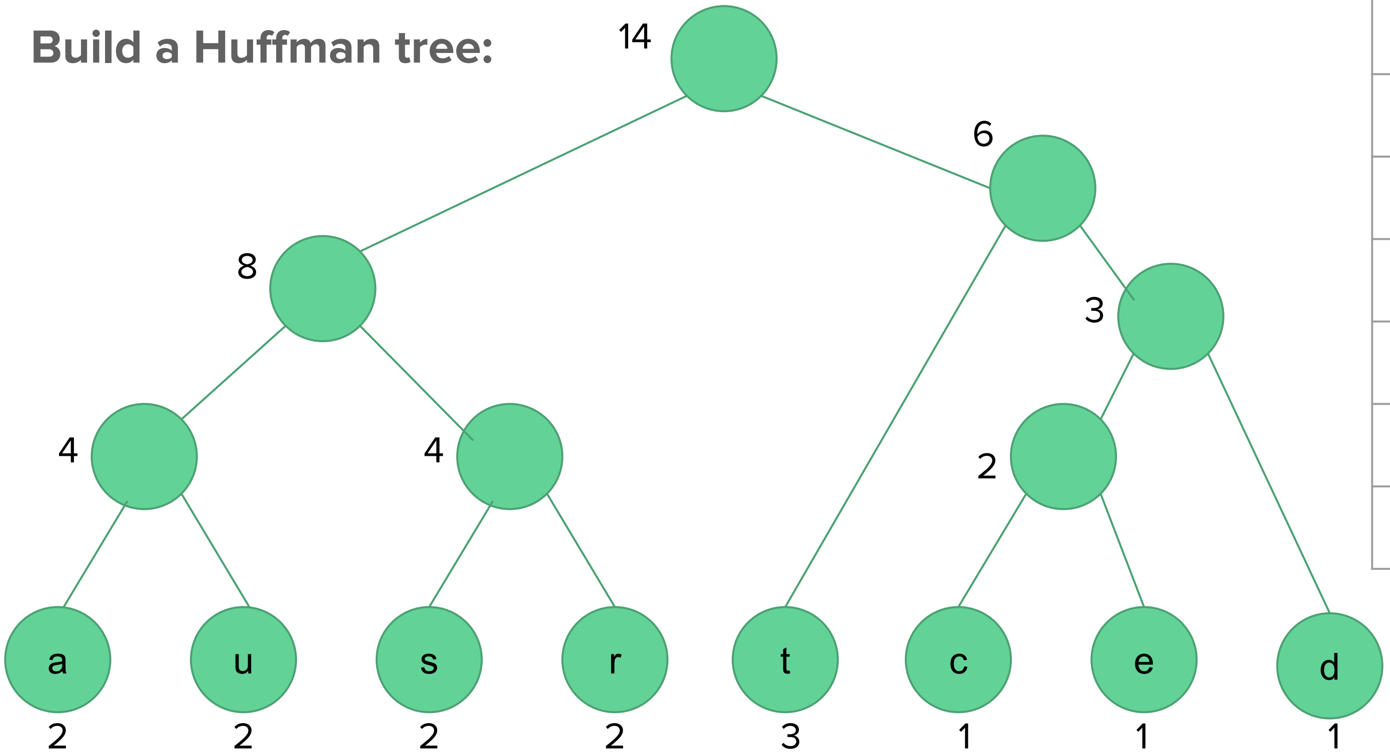
Build a Huffman tree:



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

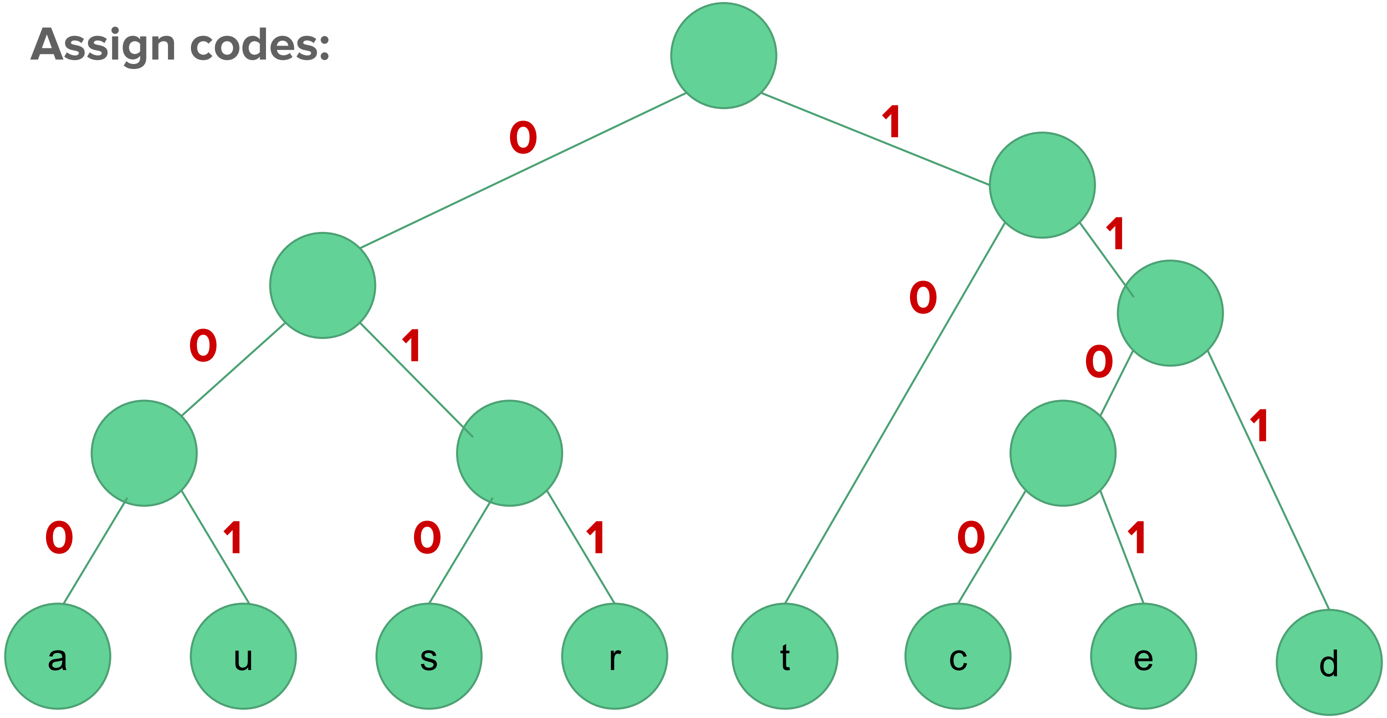
Now we **assign codes** to the tree by **placing a 0 on every left branch and a 1 on every right branch**

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

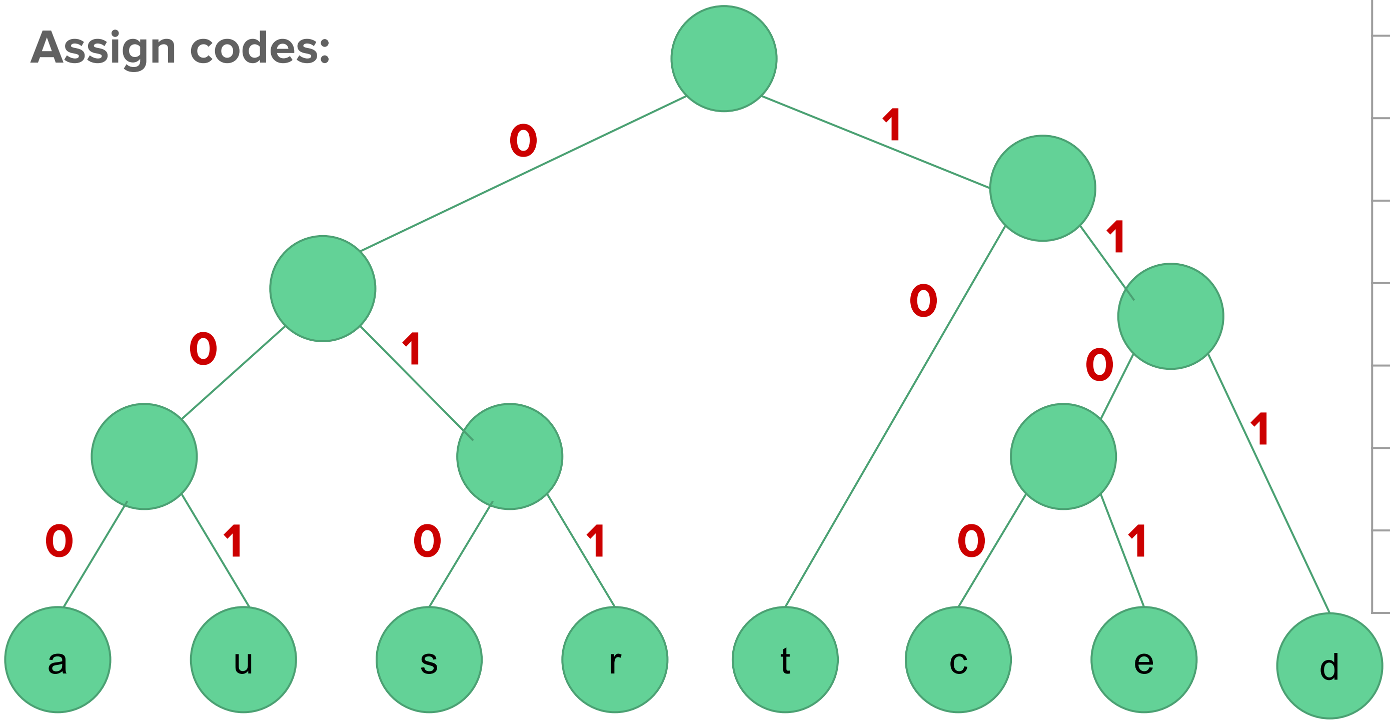
Huffman coding example

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

1110001000001010011001110010001011101010

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

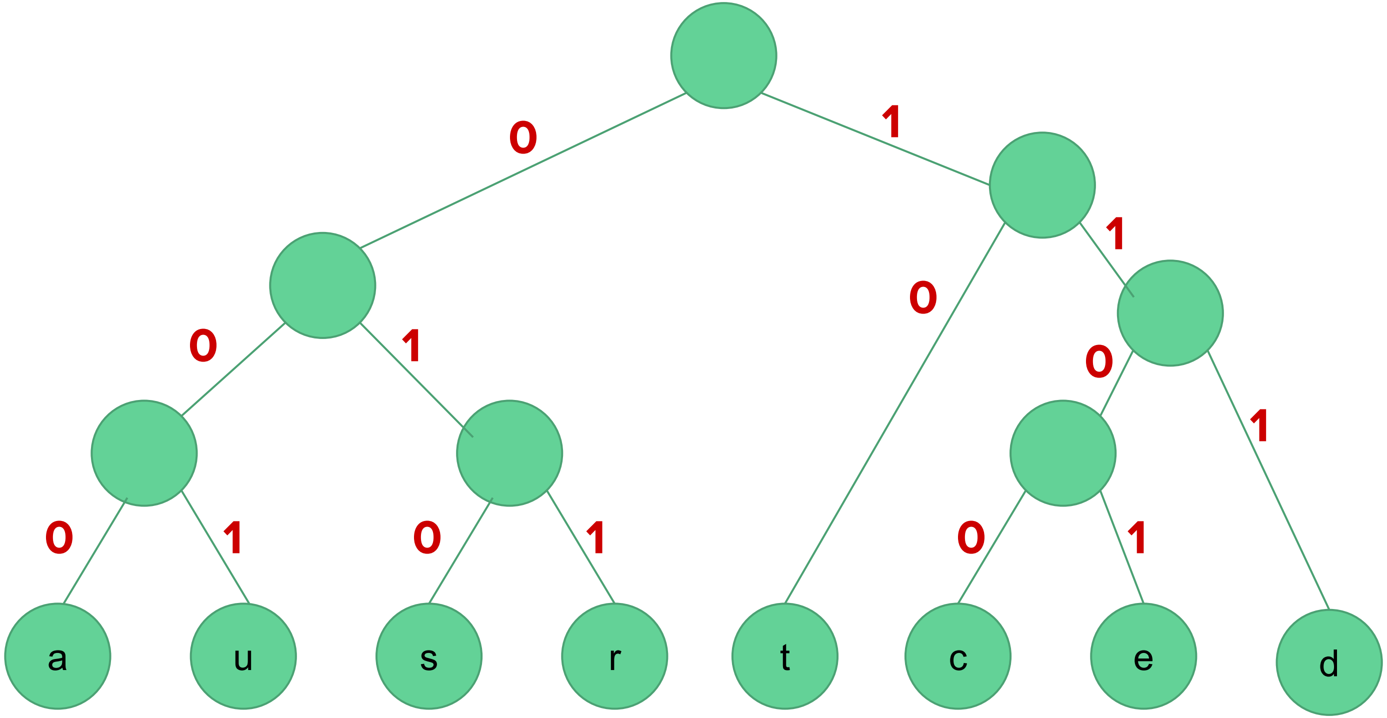
Huffman coding

Uncompression example:

Decode 11100010000010100110011100100010111101010 using the previous Huffman tree

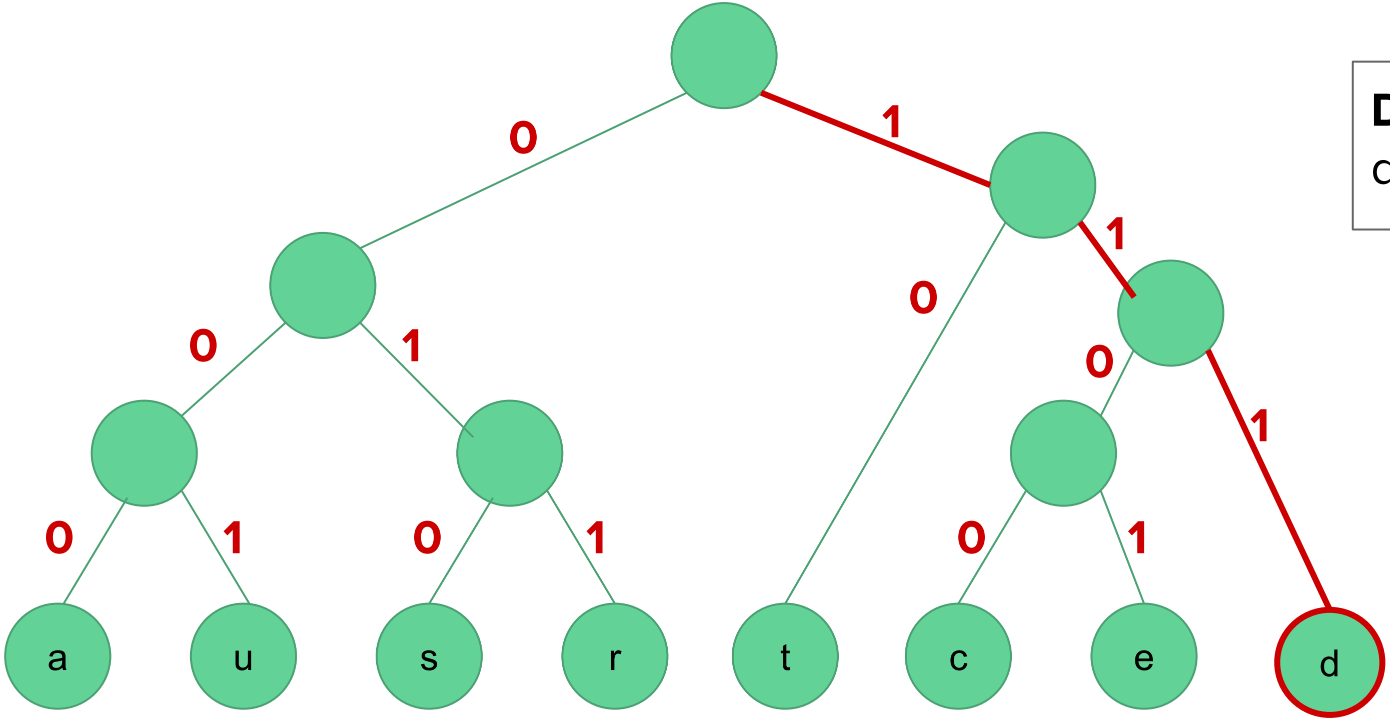
Uncompression example

Decode 1110001000001010011001110010001011101010



Uncompression example

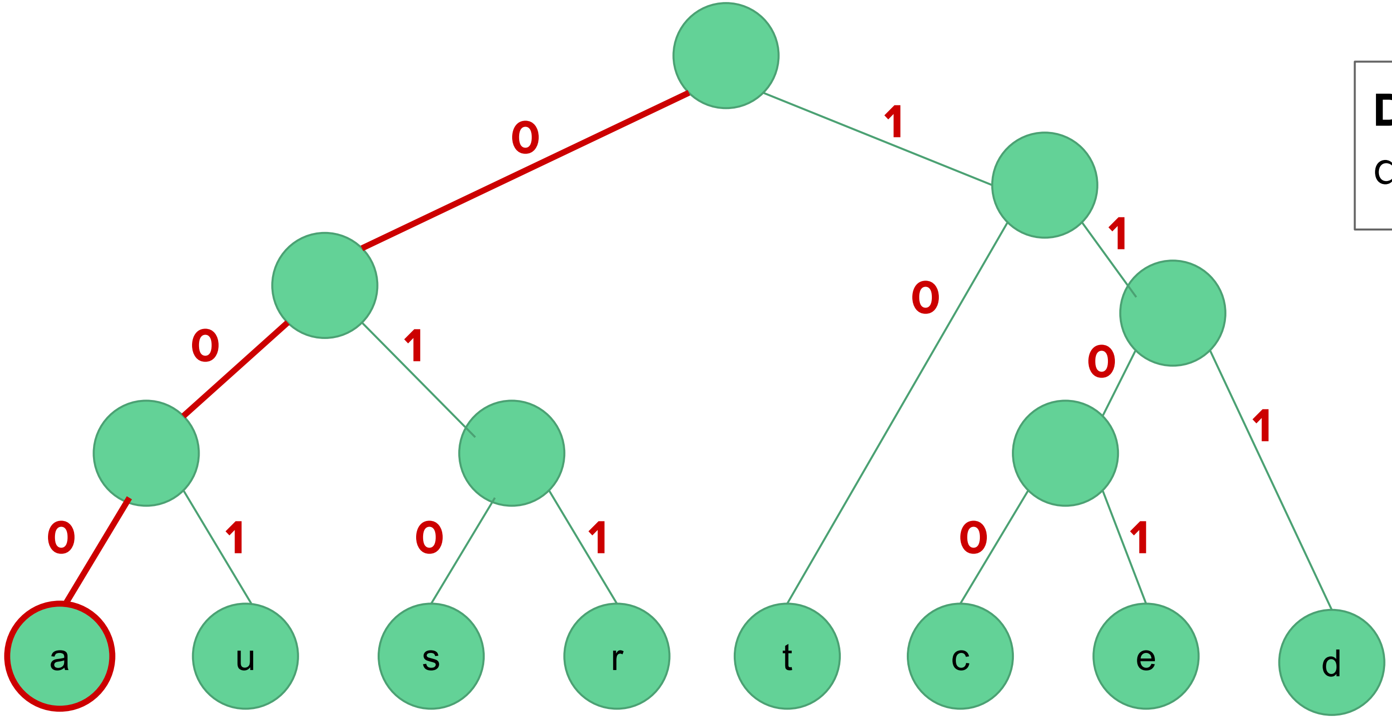
Decode **111**000100000101001100111001000101111101010



Decoded string:
d

Uncompression example

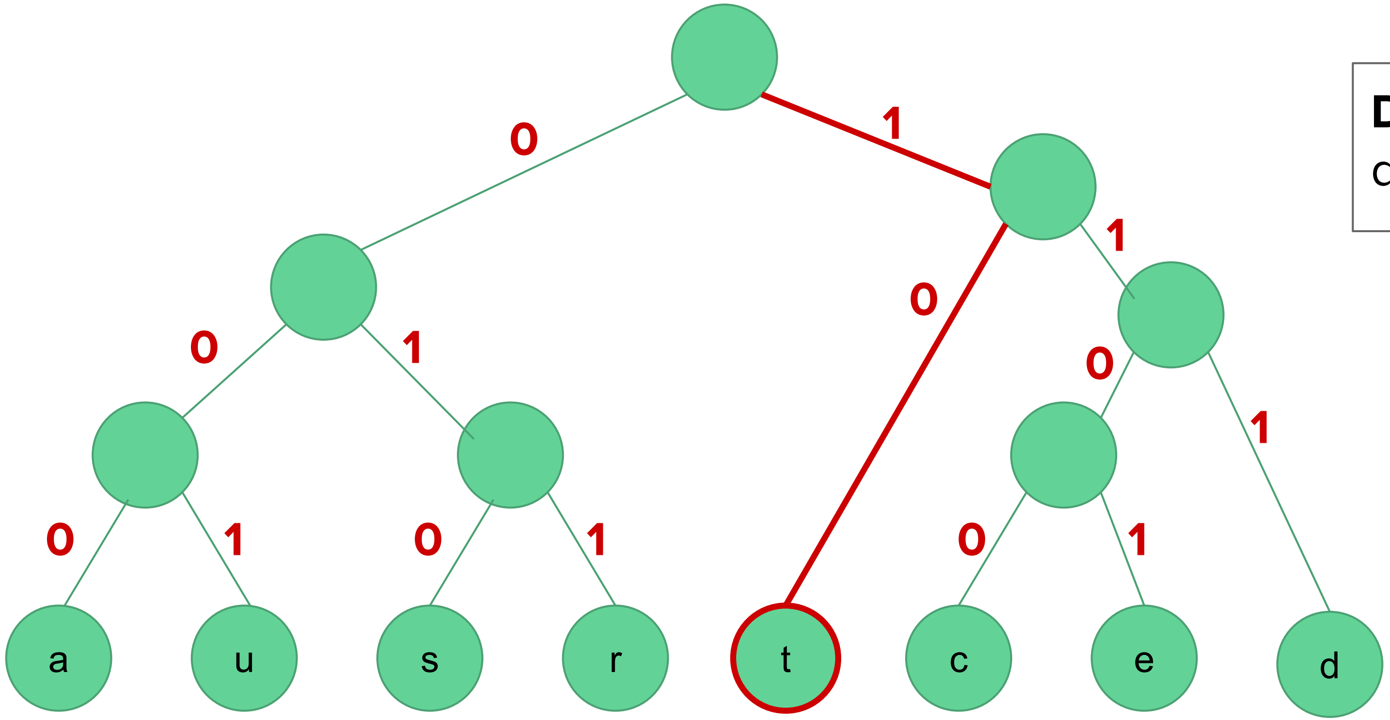
Decode 111**000**100000101001100111001000101111101010



Decoded string:
da

Uncompression example

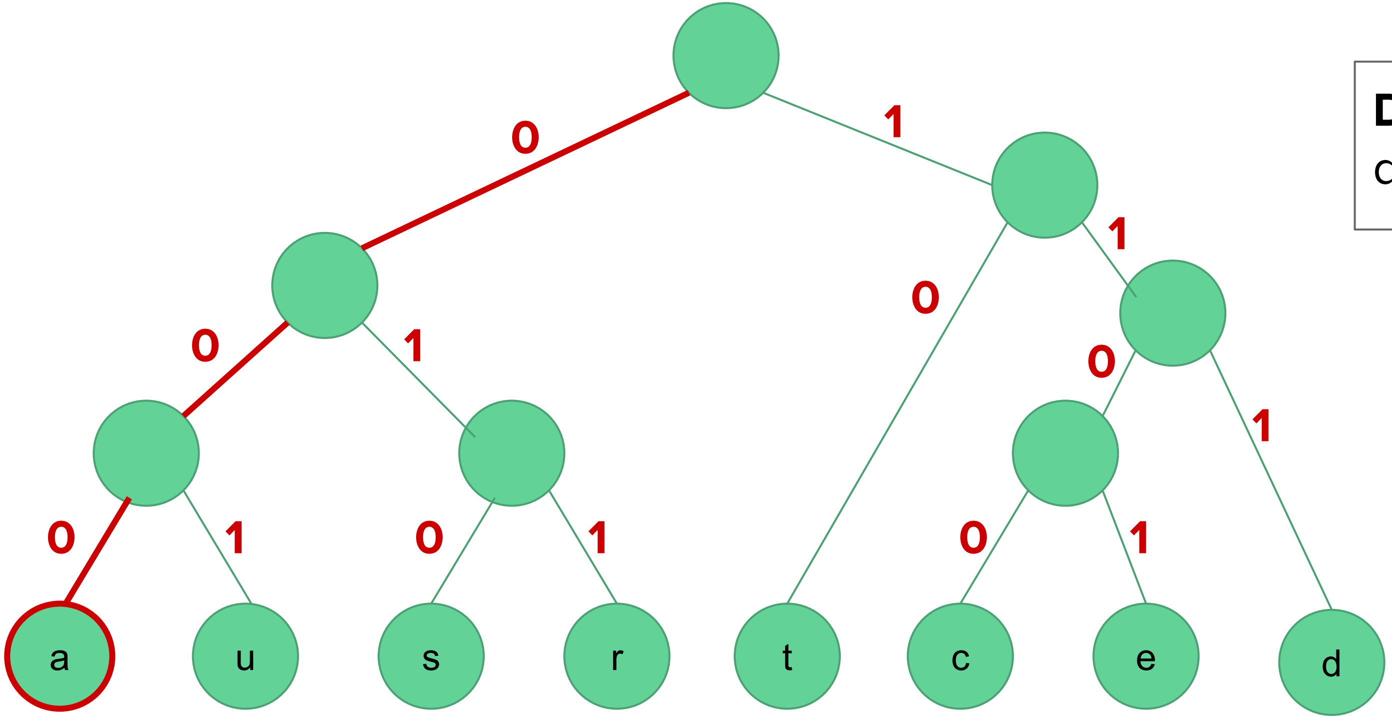
Decode 111000**10**0000010100110011100100010111101010



Decoded string:
dat

Uncompression example

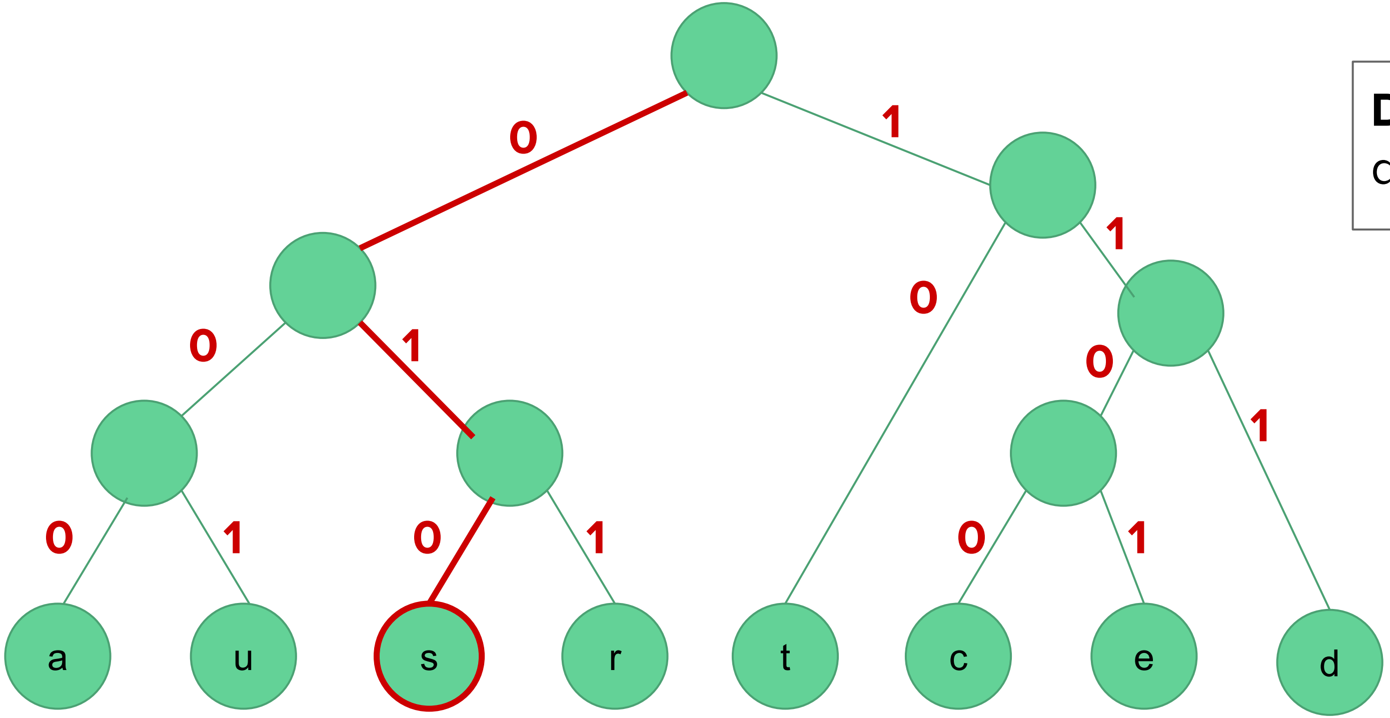
Decode 11100010**000**010100110011100100010111101010



Decoded string:
data

Uncompression example

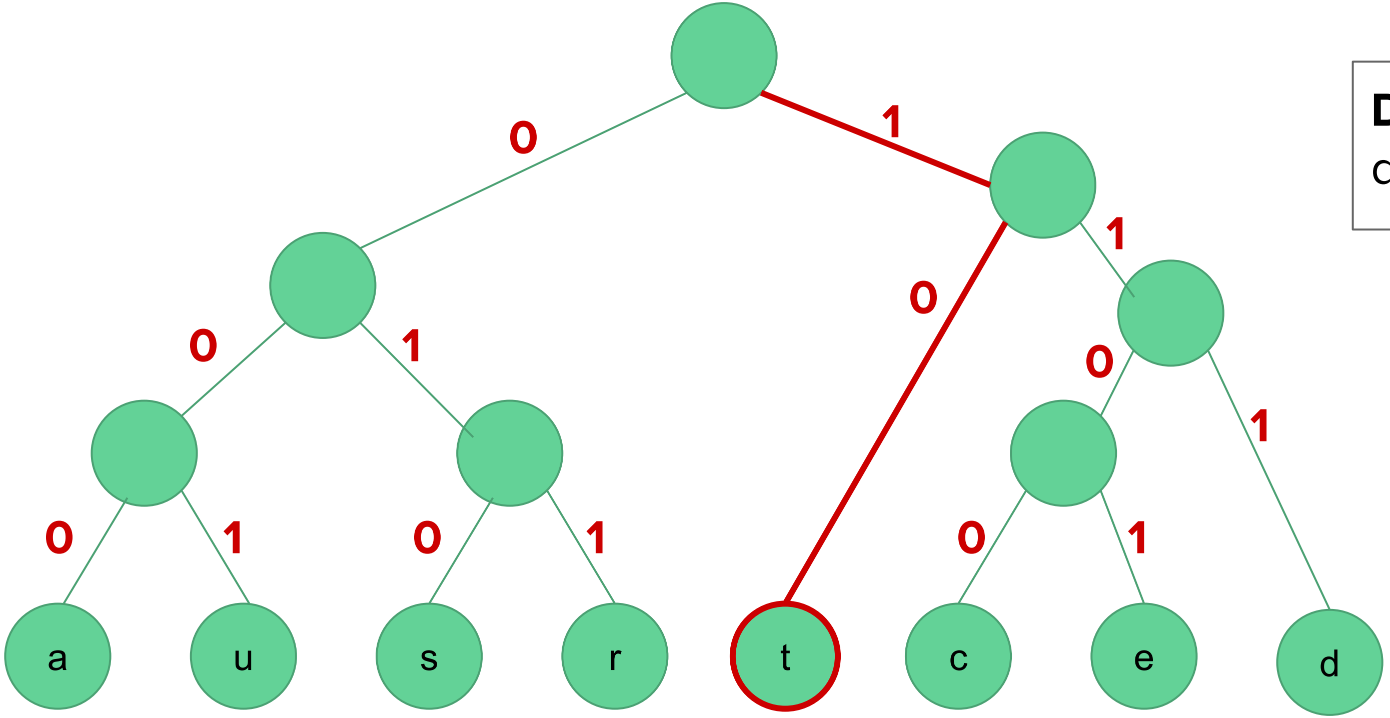
Decode 11100010000**010**1001100111001000101111101010



Decoded string:
datas

Uncompression example

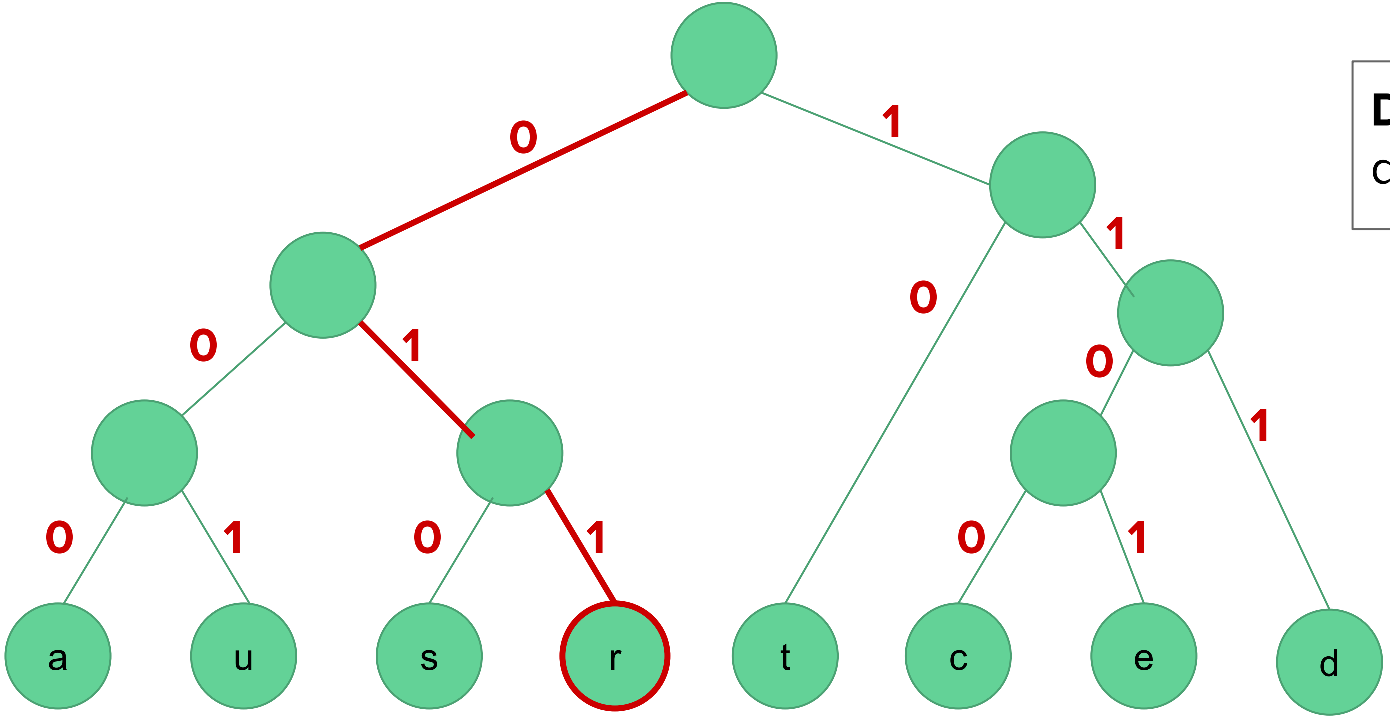
Decode 1110001000001010**10**01100111001000101111101010



Decoded string:
datast

Uncompression example

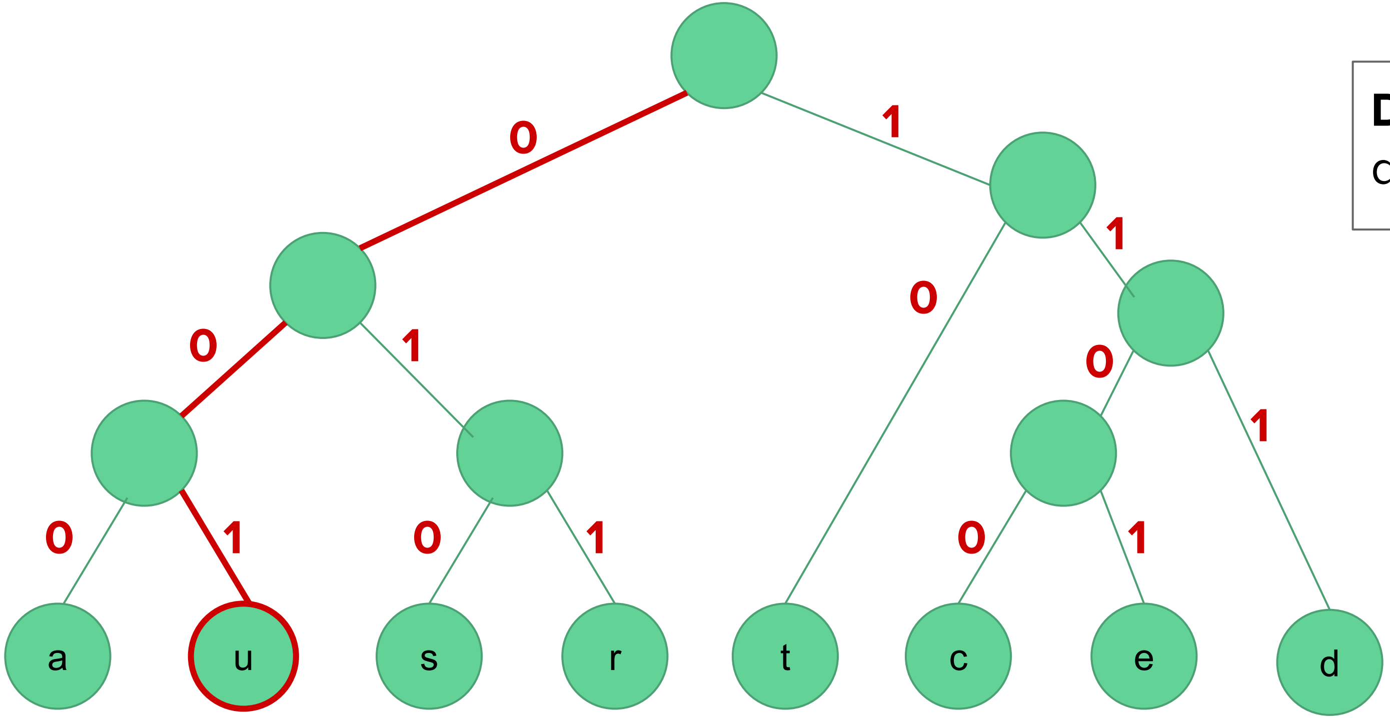
Decode 11100010000010100**11**0011100100010111101010



Decoded string:
datastr

Uncompression example

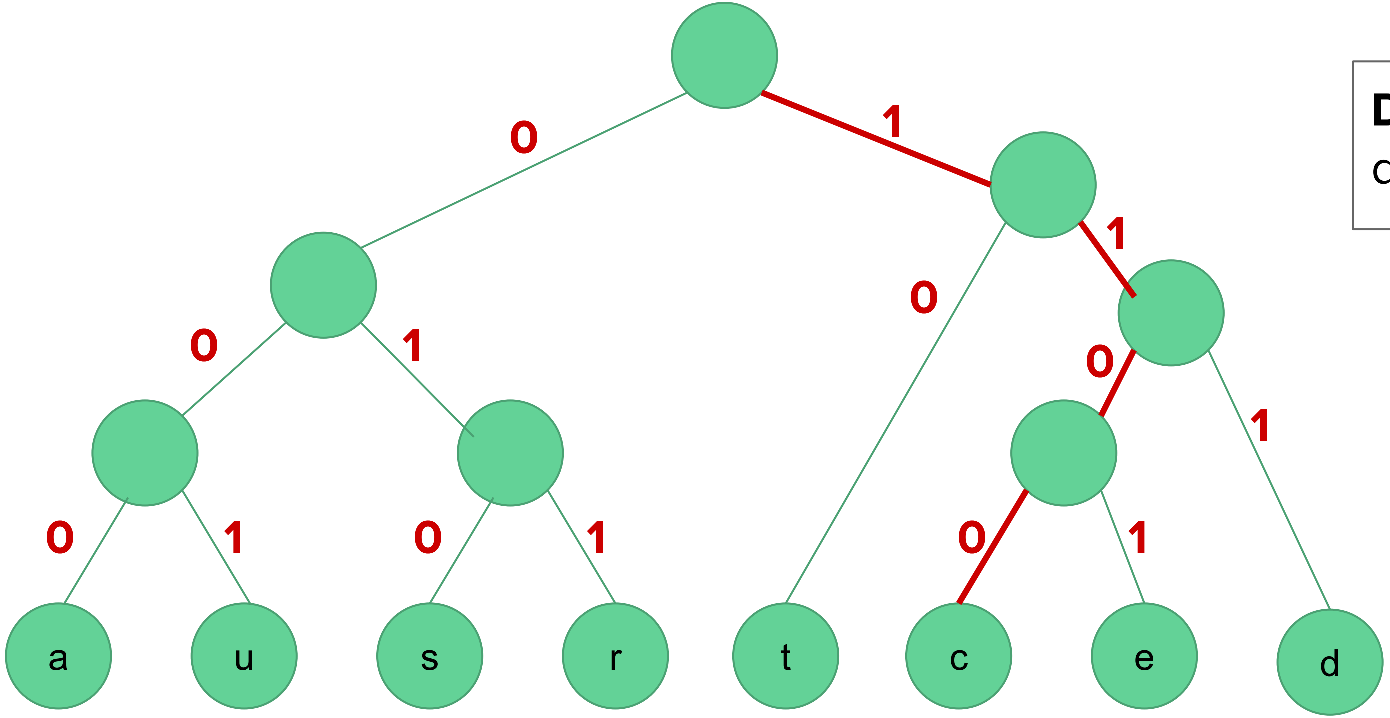
Decode 1110001000001010011**001**1100100010111101010



Decoded string:
datastru

Uncompression example

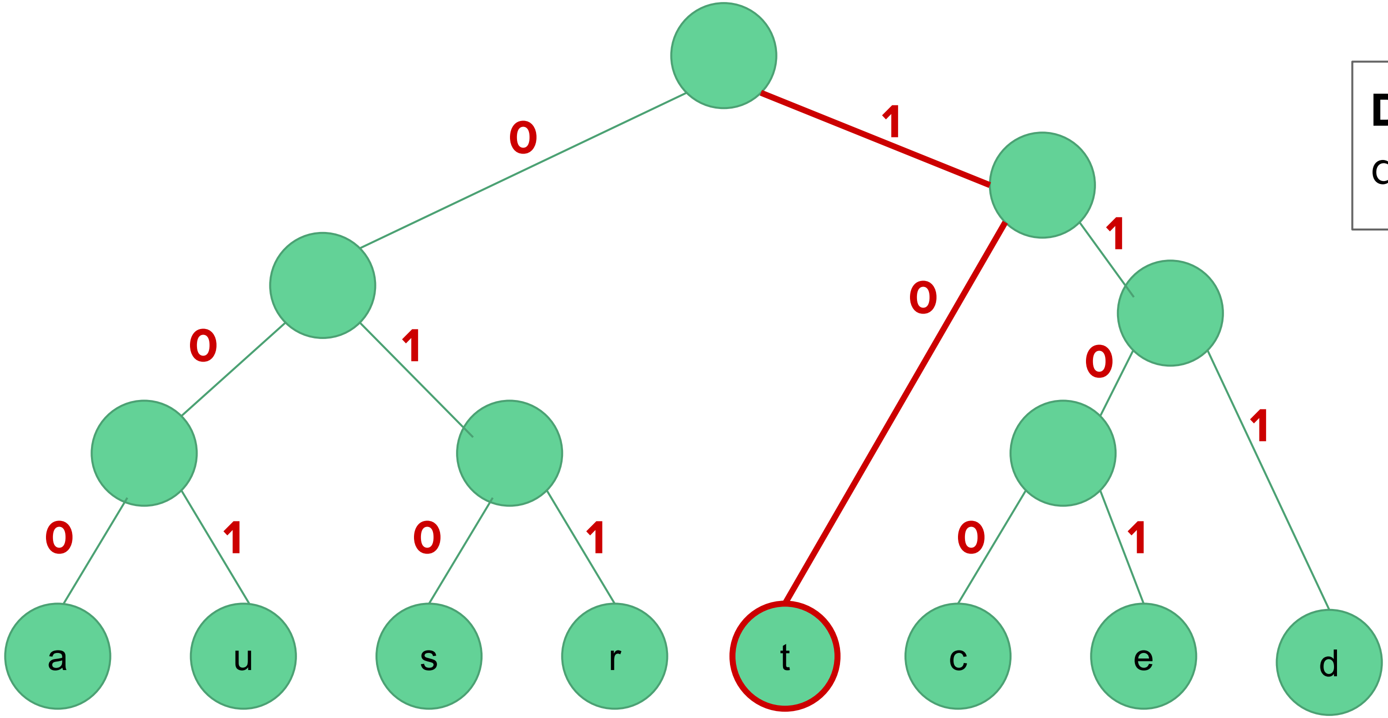
Decode 1110001000001010011001**1100**10001011101010



Decoded string:
datastruc

Uncompression example

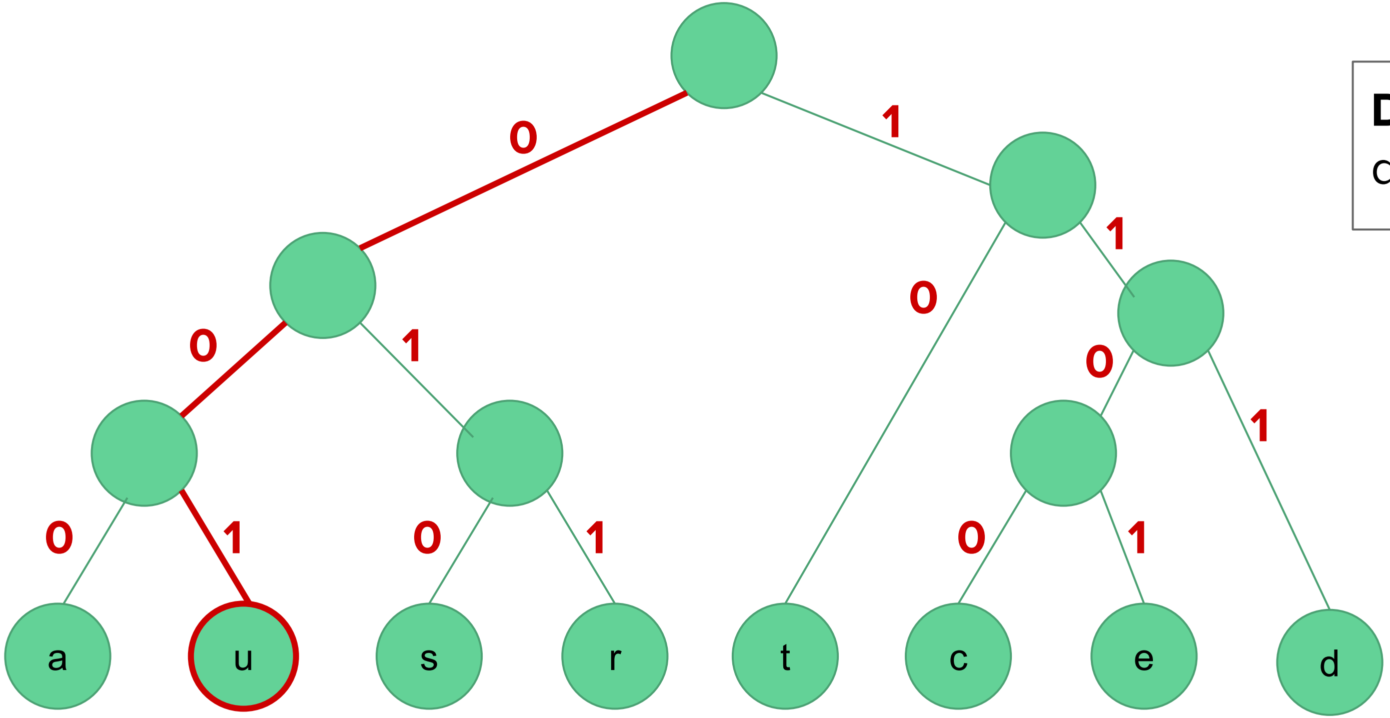
Decode 1110001000001010011001110**10**0010111101010



Decoded string:
datastruct

Uncompression example

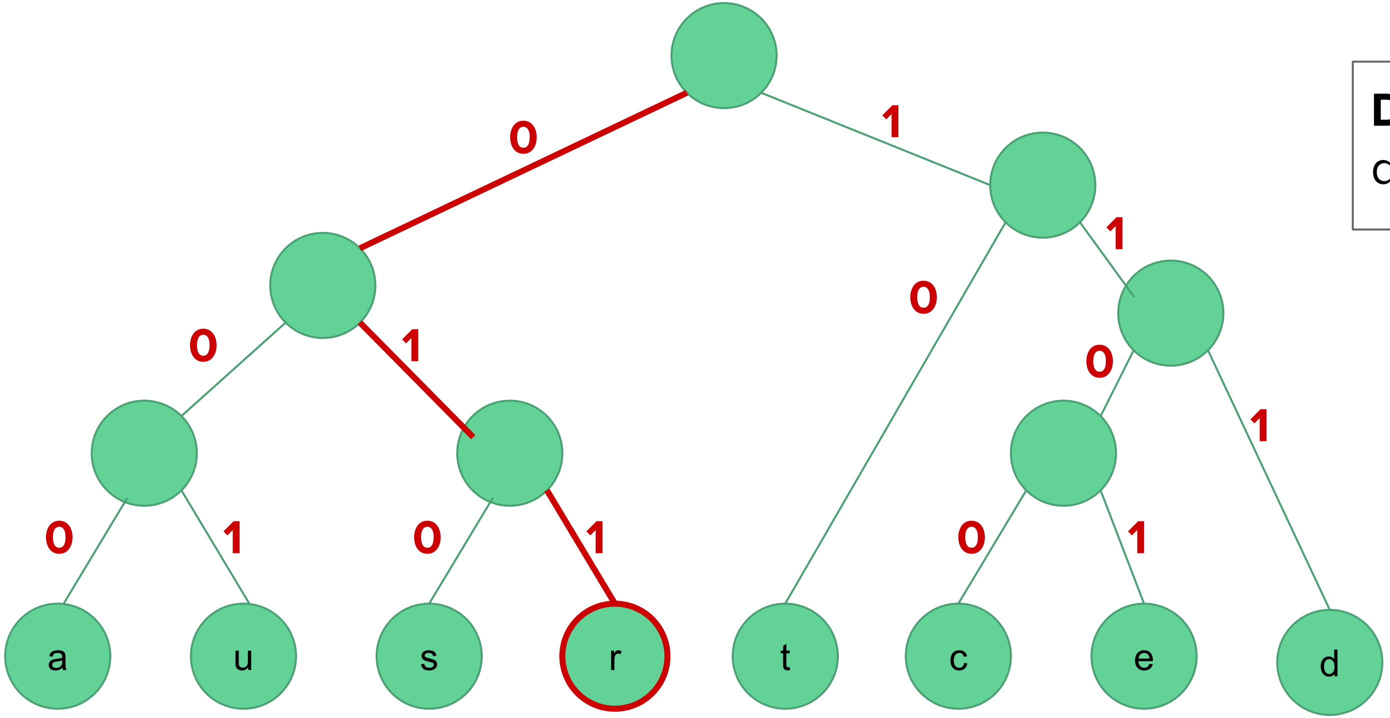
Decode 1110001000001010011001110010**001**0111101010



Decoded string:
datastructu

Uncompression example

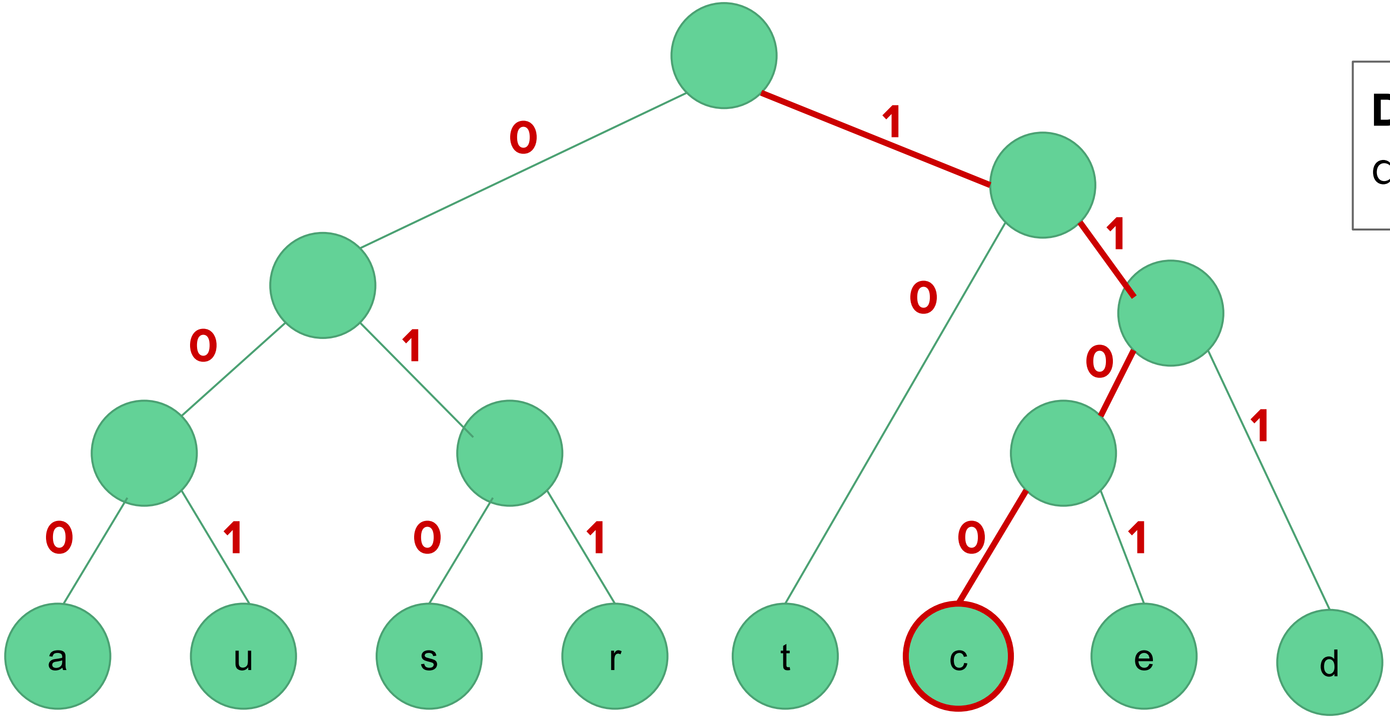
Decode 111000100000101001100111001000**1011**1101010



Decoded string:
datastructur

Uncompression example

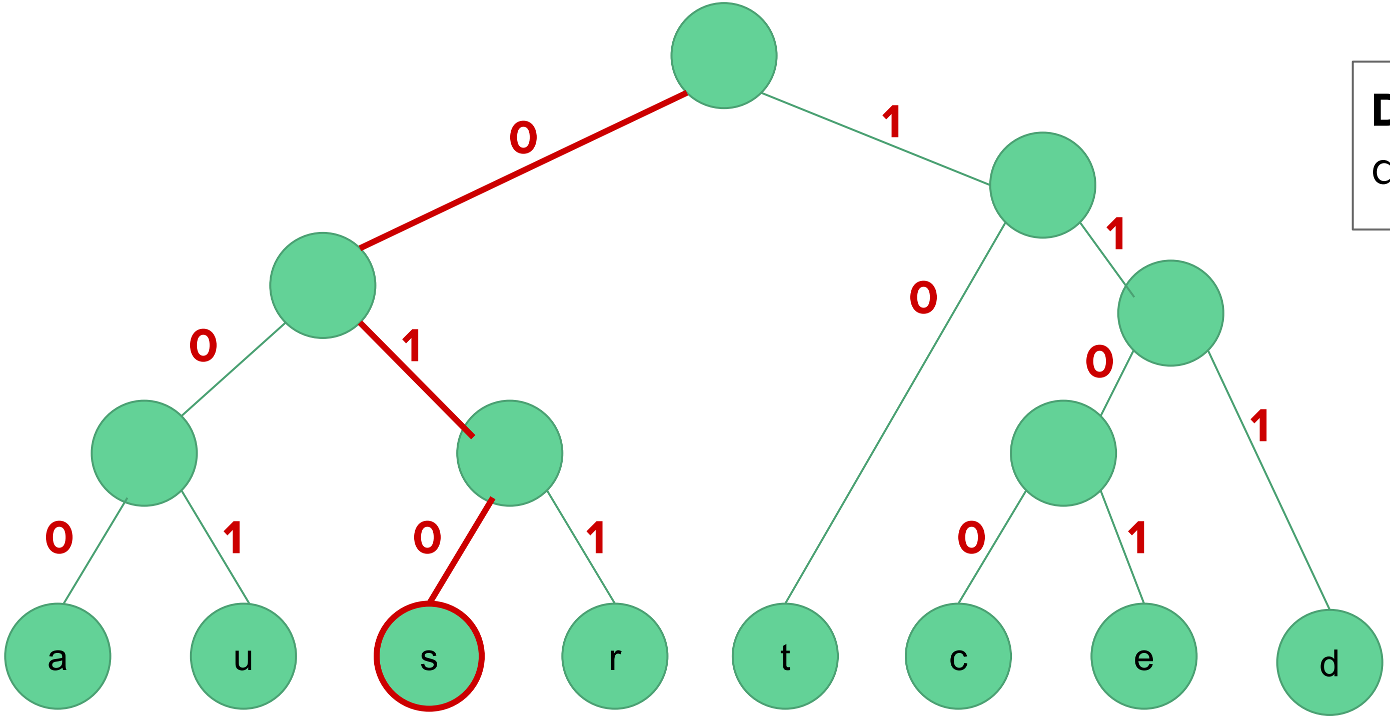
Decode 1110001000001010011001110010001011**1101**010



Decoded string:
datastructure

Uncompression example

Decode 1110001000001010011001110010001011110**1010**



Decoded string:
datastructures

Activity-Selection Problem

The problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities.

Activity-Selection

Input: A set of activities that we wish to use a resource (such as classroom) which can serve only one activity at a time. Each activity a_i in the set $S = \{a_1, a_2, .., a_n\}$ has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$.

If selected, activity a_i takes place during the time interval $[s_i, f_i)$

Output: A maximum-size subset of **mutually compatible** activities.

Two activities are compatible if and only if their intervals do not overlap.

Activity-Selection

Example

Consider the following set S of activities

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Activity-Selection

Greedy approach

- Choose an activity that leaves the resource available for as many other activities, i.e.

  Choose an activity with the earliest finish time

Activity-Selection

Greedy algorithm:

We assume that n input activities are already ordered by monotonically increasing finish time:

$f_1 \leq f_2, \leq f_3 \leq \ldots \leq f_{n-1} \leq f_n$

1. Select the activity with the earliest finish time
2. Eliminate the activities that could not be scheduled / incompatible activities
3. Repeat

Activity-Selection: Recursive greedy algorithm

Input: start times s, finish times f, the index k that defines the subproblem S_k it is to solve, and the size n of the original problem

Activity-Selection: Recursive greedy algorithm

Input: start times s, finish times f, the index k that defines the subproblem S_k it is to solve, and the size n of the original problem

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```

We start with k = 0 and a fictitious activity a0 with f0 = 0

Activity-Selection: Recursive greedy algorithm

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

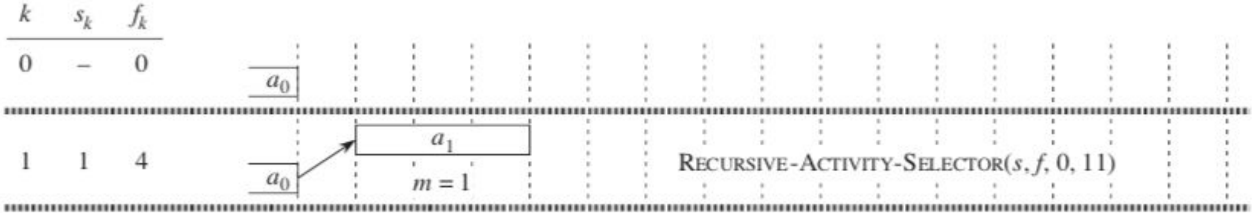
```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return ∅
```

k	s_k	f_k
0	–	0

Activity-Selection: Recursive greedy algorithm

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

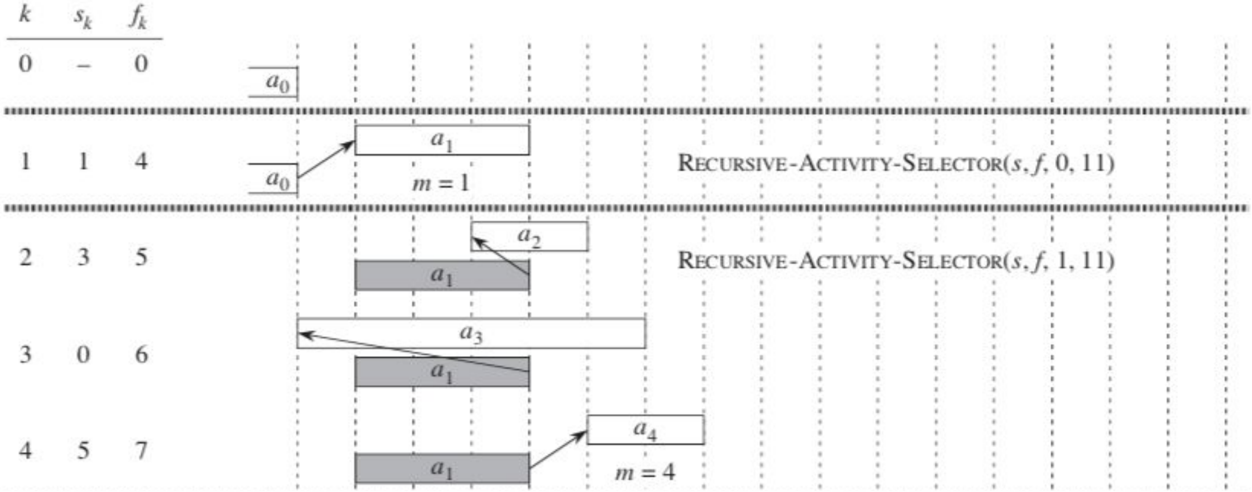
```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]      // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```



Activity-Selection: Recursive greedy algorithm

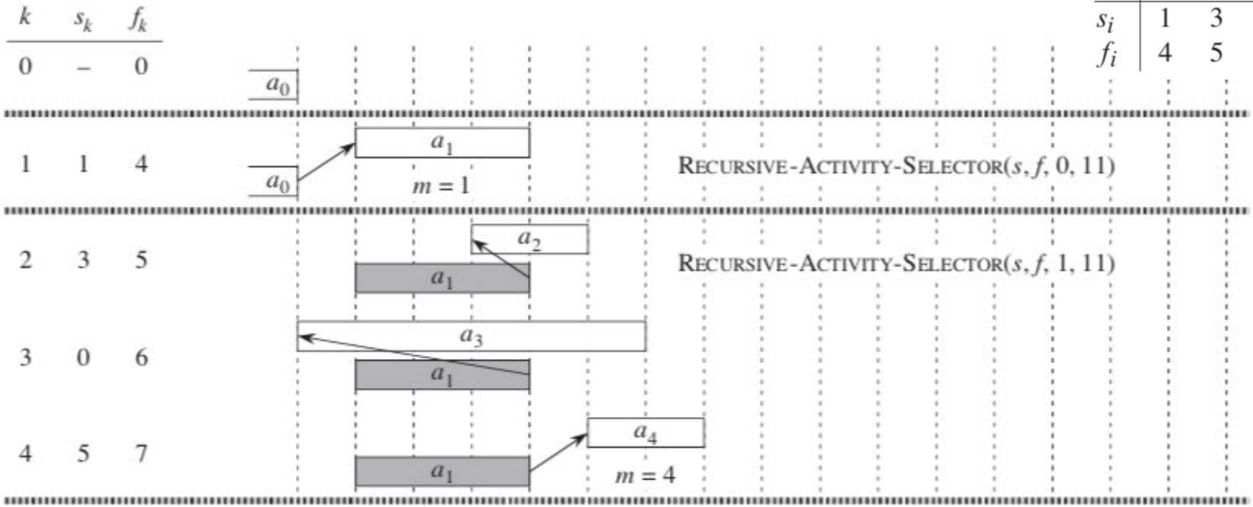
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```

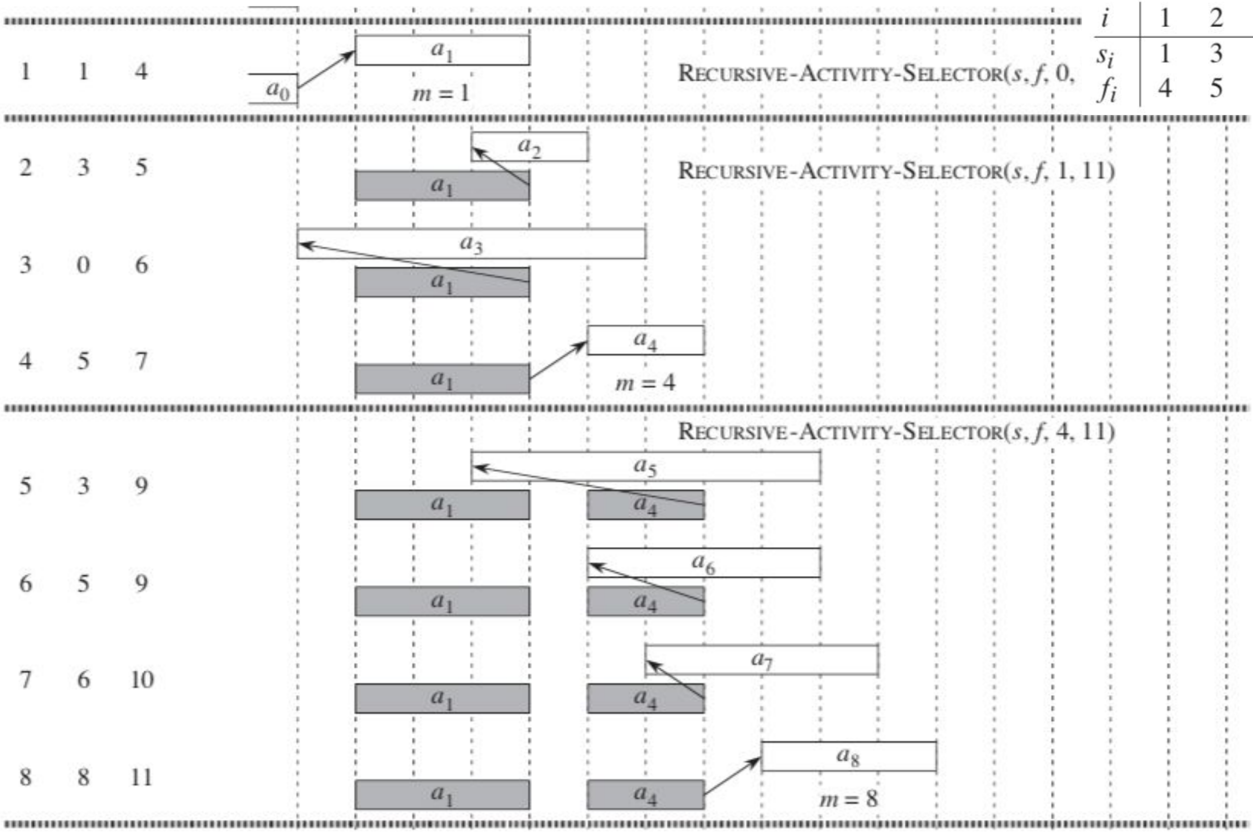


Activity-Selection: Recursive greedy algorithm

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



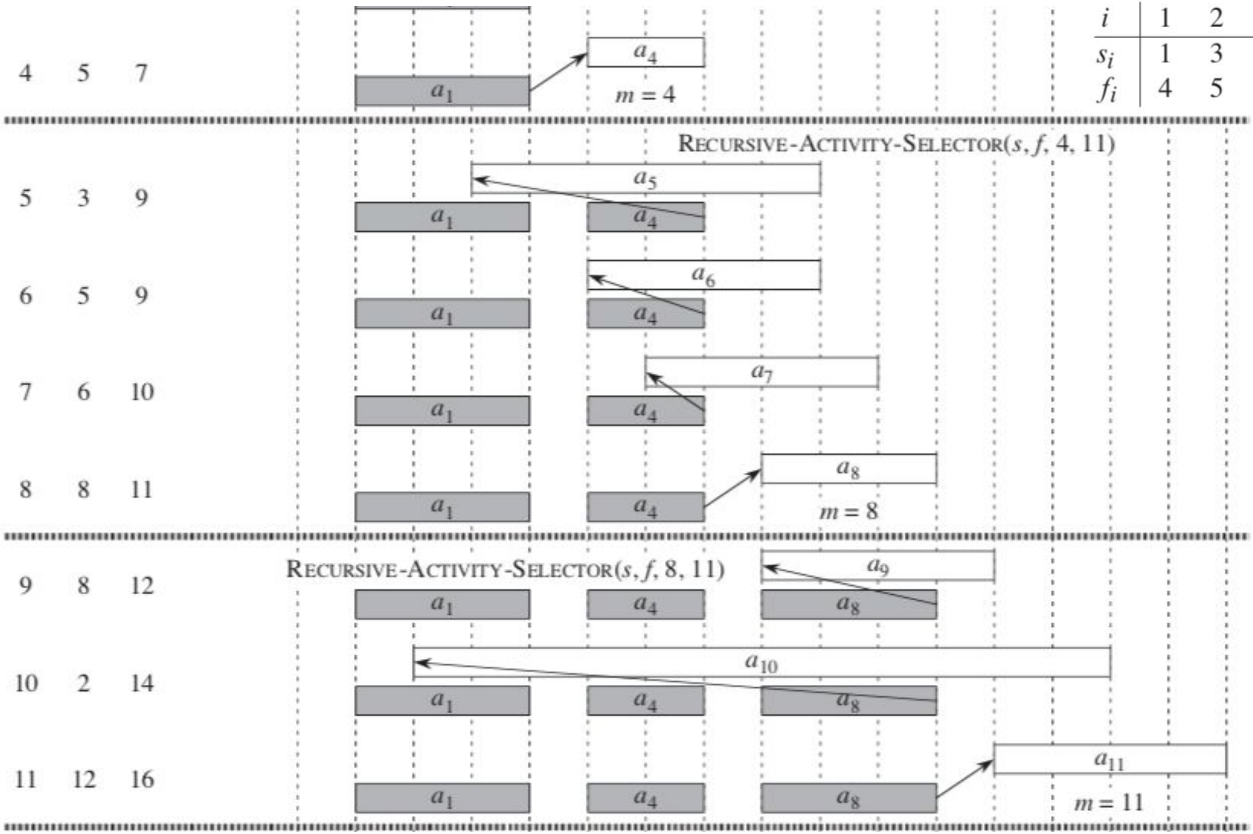
Activity-Selection: Recursive greedy algorithm



i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

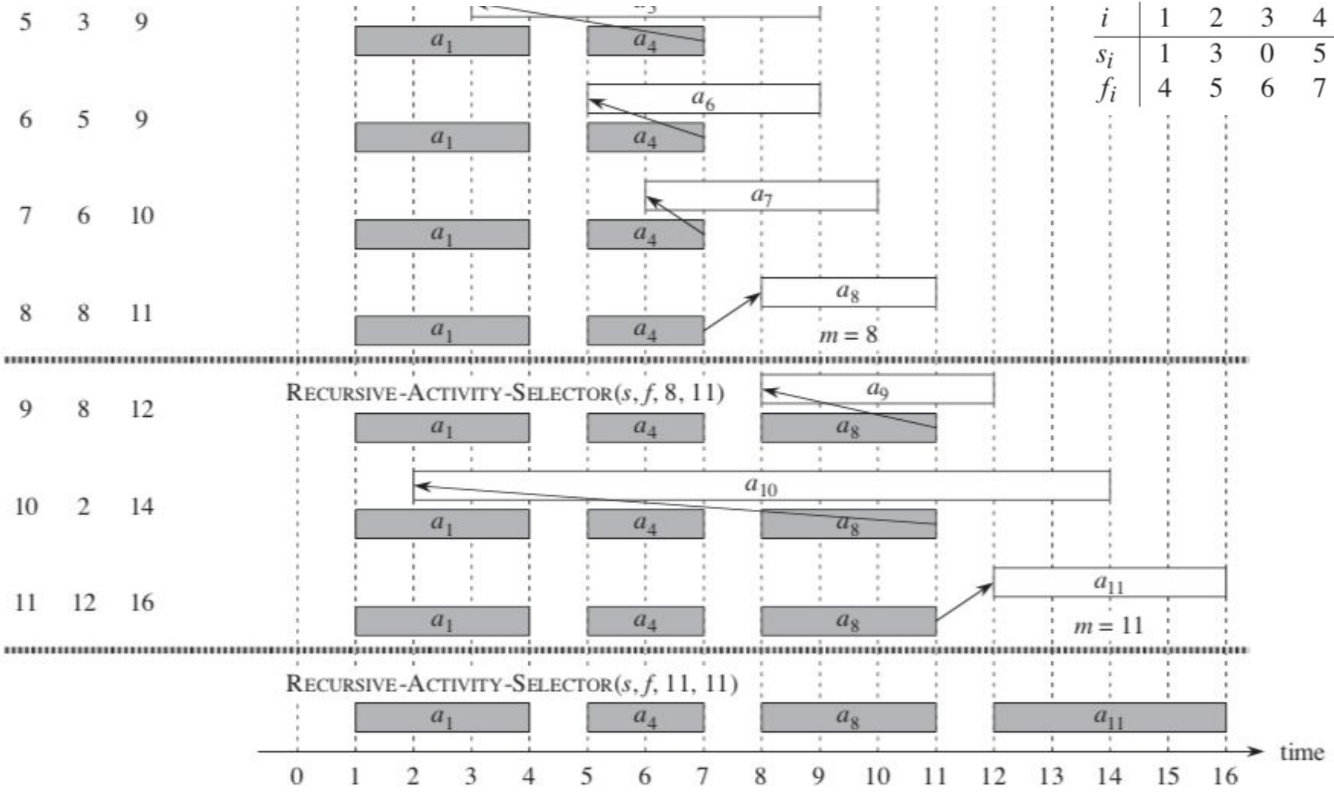
Activity-Selection: Recursive greedy algorithm

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Activity-Selection: Recursive greedy algorithm

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Activity-Selection: Iterative greedy algorithm

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

Optimal substructure

A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.