COMP90038
Algorithms and Complexity

Lecture 14: Transform-and-Conquer

(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta
munoz.m@unimelb.edu.au
Peter Hall Building G.83

On the previous lecture

- We talked about priority queues, heaps and heapsort

- A **priority queue** is a **set** of elements, each containing a **priority** value. Elements with **higher (lower)** priorities are **ejected** first.

- A **heap** is structured as a **complete binary tree** that satisfies the **condition**:

  Each child has a priority which is no greater (lesser) than its parent's

- **Heapsort** is a sorting algorithm that uses repeatedly ejects elements from the heap, and then it restores it

Today's lecture

- **Transform-and-Conquer** is a group of design techniques that:
  - **Modify** the problem to a more **amenable** form, and then
  - **Solve** it using a **known efficient** algorithm

- There are three major variations
  - Instance simplification
  - Representational change
  - Problem reduction

Transform-and-conquer

- In **instance simplification** we try to make the problem **easier** through some type of **pre-processing**, typically **sorting**
- In **representation change** we use a different data structure with better properties
  - An **unsorted array** is reorganized as a **heap**
- In **problem reduction** we solve the instance **as if it was a different problem**
  - We will talk more about this on **week 11**

Instance simplification

- Let's examine two problems in which **pre-sorting** the data significantly reduces complexity:

  - **Uniqueness checking**, i.e., given an unsorted array $A[0]\ldots A[n\text{-}1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

  - **Finding the mode**, i.e., the element which occurs most frequently in a data set

Uniqueness checking

- The obvious answer to this problem is a brute force approach

1 $\longrightarrow$

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad \textbf{for } j \leftarrow i+1 \text{ to } n-1 \textbf{ do} \\
\quad\quad \textbf{if } A[i] = A[j] \textbf{ then} \\
\quad\quad\quad \textbf{return } \text{FALSE} \\
\textbf{return } \text{TRUE}
\end{array}
$$

Memory state (1)	
$A[0,\ldots,7]$	[2 9 8 6 9 5 7 3]
n	**8**
i	**0**
j	
$A[i]$	**2**
$A[j]$	
$A[i] = A[j]$	

Uniqueness checking

- The obvious answer to this problem is a brute force approach

2 →

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad \textbf{for } j \leftarrow i+1 \text{ to } n-1 \textbf{ do} \\
\quad\quad \textbf{if } A[i] = A[j] \textbf{ then} \\
\quad\quad\quad \textbf{return } \text{FALSE} \\
\textbf{return } \text{TRUE}
\end{array}
$$

Memory state (2)	
$A[0,\ldots,7]$	[2 9 8 6 9 5 7 3]
n	8
i	0
j	**1**
$A[i]$	2
$A[j]$	**9**
$A[i] = A[j]$	

Uniqueness checking

- The obvious answer to this problem is a brute force approach

3

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad \textbf{for } j \leftarrow i+1 \text{ to } n-1 \textbf{ do} \\
\quad\quad \textbf{if } A[i] = A[j] \textbf{ then} \\
\quad\quad\quad \textbf{return } \text{FALSE} \\
\textbf{return } \text{TRUE}
\end{array}
$$

Memory state (3)	
$A[0,\ldots,7]$	[2 9 8 6 9 5 7 3]
n	8
i	0
j	1
$A[i]$	2
$A[j]$	9
$A[i] = A[j]$	FALSE

Uniqueness checking

- The obvious answer to this problem is a brute force approach

4

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad \textbf{for } j \leftarrow i+1 \text{ to } n-1 \textbf{ do} \\
\quad\quad \textbf{if } A[i] = A[j] \textbf{ then} \\
\quad\quad\quad \textbf{return } \textsc{false} \\
\textbf{return } \textsc{true}
\end{array}
$$

Memory state (4)	
$A[0,\ldots,7]$	[2 9 8 6 9 5 7 3]
n	8
i	0
j	**2**
$A[i]$	2
$A[j]$	**8**
$A[i] = A[j]$	FALSE

Uniqueness checking

- The obvious answer to this problem is a brute force approach

5

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad \textbf{for } j \leftarrow i+1 \text{ to } n-1 \textbf{ do} \\
\quad\quad \textbf{if } A[i] = A[j] \textbf{ then} \\
\quad\quad\quad \textbf{return } \text{FALSE} \\
\textbf{return } \text{TRUE}
\end{array}
$$

Memory state (5)	
$A\,[0,\ldots,7]$	[2 9 8 6 9 5 7 3]
n	8
i	0
j	**3**
$A\,[i]$	2
$A\,[j]$	**6**
$A\,[i] = A\,[j]$	FALSE

Uniqueness checking

- The obvious answer to this problem is a brute force approach

6

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad \textbf{for } j \leftarrow i+1 \text{ to } n-1 \textbf{ do} \\
\quad\quad \textbf{if } A[i] = A[j] \textbf{ then} \\
\quad\quad\quad \textbf{return } \text{FALSE} \\
\textbf{return } \text{TRUE}
\end{array}
$$

Memory state (6)	
$A[0,\ldots,7]$	[2 9 8 6 9 5 7 3]
n	8
i	0
j	**4**
$A[i]$	2
$A[j]$	**9**
$A[i] = A[j]$	FALSE

Uniqueness checking

- The obvious answer to this problem is a brute force approach

for $i \leftarrow 0$ to $n-2$ **do**
    for $j \leftarrow i+1$ to $n-1$ **do**
7   **if** $A[i] = A[j]$ **then**
            return FALSE
return TRUE

Memory state (7)	
$A[0,\ldots,7]$	[2 9 8 6 9 5 7 3]
n	8
i	0
j	**5**
$A[i]$	2
$A[j]$	**5**
$A[i] = A[j]$	FALSE

Uniqueness checking

- The obvious answer to this problem is a brute force approach

8 →

for $i \leftarrow 0$ to $n-2$ **do**
    for $j \leftarrow i+1$ to $n-1$ **do**
        if $A[i] = A[j]$ **then**
            return FALSE
return TRUE

Memory state (8)	
$A\,[0,\ldots,7]$	[2 9 8 6 9 5 7 3]
n	8
i	0
j	**6**
$A\,[i]$	2
$A\,[j]$	**7**
$A\,[i] = A\,[j]$	FALSE

Uniqueness checking

- The obvious answer to this problem is a brute force approach

for $i \leftarrow 0$ to $n-2$ **do**
    for $j \leftarrow i+1$ to $n-1$ **do**
9   **if** $A[i] = A[j]$ **then**
            return FALSE
return TRUE

Memory state (9)	
$A[0,\ldots,7]$	$[2\ 9\ 8\ 6\ 9\ 5\ 7\ 3]$
n	8
i	0
j	**7**
$A[i]$	2
$A[j]$	**3**
$A[i] = A[j]$	FALSE

Uniqueness checking

- The obvious answer to this problem is a brute force approach

```
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] then
            return FALSE
return TRUE
```

10

Memory state (10)	
$A[0,\ldots,7]$	[2 9 8 6 9 5 7 3]
n	8
i	**1**
j	**2**
$A[i]$	**9**
$A[j]$	**8**
$A[i] = A[j]$	FALSE

Uniqueness checking

- The obvious answer to this problem is a brute force approach

11 →

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad \textbf{for } j \leftarrow i+1 \text{ to } n-1 \textbf{ do} \\
\quad\quad \textbf{if } A[i] = A[j] \textbf{ then} \\
\quad\quad\quad \textbf{return } \text{FALSE} \\
\textbf{return } \text{TRUE}
\end{array}
$$

Memory state (11)	
$A[0,\ldots,7]$	[2 9 8 6 9 5 7 3]
n	8
i	1
j	**3**
$A[i]$	9
$A[j]$	**6**
$A[i] = A[j]$	FALSE

Uniqueness checking

- The obvious answer to this problem is a brute force approach

12 →

for $i \leftarrow 0$ to $n-2$ **do**
  for $j \leftarrow i+1$ to $n-1$ **do**
    if $A[i] = A[j]$ **then**
      return FALSE
return TRUE

Memory state (12)	
$A[0, \ldots, 7]$	[2 9 8 6 9 5 7 3]
n	8
i	1
j	**4**
$A[i]$	9
$A[j]$	**9**
$A[i] = A[j]$	TRUE

Uniqueness checking

- The obvious answer to this problem is a brute force approach

```
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] then
            return FALSE
return TRUE
```

Memory state (12)	
$A[0,\ldots,7]$	[2 9 8 6 9 5 7 3]
n	8
i	1
j	**4**
$A[i]$	9
$A[j]$	**9**
$A[i] = A[j]$	TRUE

- The complexity of this approach is $O(n^2)$

Uniqueness checking

- Let's examine a pre-sorting approach

$\textsc{MergeSort}(A, n)$
for $i \leftarrow 0$ to $n-2$ **do**
    if $A[i] = A[i+1]$ **then**
        return FALSE
return TRUE

Uniqueness checking

- Let's examine a pre-sorting approach

1 →

$\textsc{MergeSort}(A, n)$
for $i \leftarrow 0$ to $n-2$ **do**
  if $A[i] = A[i+1]$ **then**
    return FALSE
return TRUE

Memory state (1)	
$A[0, \ldots, 7]$	[2 9 8 6 9 5 7 3]
n	8
i	
$A[i]$	
$A[i+1]$	
$A[i] = A[i+1]$	

Uniqueness checking

- Let's examine a pre-sorting approach

2

$\textsc{MergeSort}(A, n)$
for $i \leftarrow 0$ to $n-2$ **do**
  if $A[i] = A[i+1]$ **then**
    return FALSE
return TRUE

Memory state (2)	
$A[0,\ldots,7]$	**[2 3 5 6 7 8 9 9]**
n	8
i	**0**
$A[i]$	
$A[i+1]$	
$A[i] = A[i+1]$	

Uniqueness checking

- Let's examine a pre-sorting approach

3

$\textsc{MergeSort}(A, n)$
for $i \leftarrow 0$ to $n-2$ **do**
  if $A[i] = A[i+1]$ **then**
    return FALSE
return TRUE

Memory state (3)	
$A[0,\ldots,7]$	[2 3 5 6 7 8 9 9]
n	8
i	0
$A[i]$	**2**
$A[i+1]$	**3**
$A[i] = A[i+1]$	FALSE

Uniqueness checking

- Let's examine a pre-sorting approach

4

$\textsc{MergeSort}(A, n)$
for $i \leftarrow 0$ to $n-2$ **do**
  if $A[i] = A[i+1]$ **then**
    return FALSE
return TRUE

Memory state (4)	
$A[0,\ldots,7]$	[2 3 5 6 7 8 9 9]
n	8
i	**1**
$A[i]$	**3**
$A[i+1]$	**5**
$A[i] = A[i+1]$	FALSE

Uniqueness checking

- Let's examine a pre-sorting approach

5

$\textsc{MergeSort}(A, n)$
for $i \leftarrow 0$ to $n-2$ **do**
  if $A[i] = A[i+1]$ **then**
    return FALSE
return TRUE

Memory state (5)	
$A[0,\ldots,7]$	$[2\ 3\ 5\ 6\ 7\ 8\ 9\ 9]$
n	8
i	**2**
$A[i]$	**5**
$A[i+1]$	**6**
$A[i] = A[i+1]$	FALSE

Uniqueness checking

- Let's examine a pre-sorting approach

6

$$
\begin{array}{l}
\textsc{MergeSort}(A, n) \\
\textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad \textbf{if } A[i] = A[i+1] \textbf{ then} \\
\quad\quad \textbf{return } \textsc{false} \\
\textbf{return } \textsc{true}
\end{array}
$$

Memory state (6)	
$A\,[0,\ldots,7]$	[2 3 5 6 7 8 9 9]
n	8
i	**4**
$A\,[i]$	**6**
$A\,[i+1]$	**7**
$A\,[i] = A\,[i+1]$	FALSE

Uniqueness checking

- Let's examine a pre-sorting approach

7

$$
\begin{array}{l}
\textsc{MergeSort}(A, n) \\
\textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad \textbf{if } A[i] = A[i+1] \textbf{ then} \\
\quad\quad \textbf{return } \textsc{false} \\
\textbf{return } \textsc{true}
\end{array}
$$

Memory state (7)	
$A[0,\ldots,7]$	[2 3 5 6 7 8 9 9]
n	8
i	**5**
$A[i]$	**7**
$A[i+1]$	**8**
$A[i] = A[i+1]$	FALSE

Uniqueness checking

- Let's examine a pre-sorting approach

8 →

$\textsc{MergeSort}(A, n)$
for $i \leftarrow 0$ to $n-2$ **do**
  if $A[i] = A[i+1]$ **then**
    return FALSE
return TRUE

Memory state (8)	
$A[0,\ldots,7]$	[2 3 5 6 7 8 9 9]
n	8
i	**6**
$A[i]$	**8**
$A[i+1]$	**9**
$A[i] = A[i+1]$	FALSE

Uniqueness checking

- Let's examine a pre-sorting approach

$\textsc{MergeSort}(A, n)$
for $i \leftarrow 0$ to $n-2$ **do**
    if $A[i] = A[i+1]$ **then**
9   **return** FALSE
return TRUE

Memory state (9)	
$A[0,\ldots,7]$	[2 3 5 6 7 8 9 9]
n	8
i	**7**
$A[i]$	**9**
$A[i+1]$	**9**
$A[i] = A[i+1]$	TRUE

Uniqueness checking

- Let's examine a pre-sorting approach

$$
\begin{array}{l}
\textsc{MergeSort}(A, n) \\
\textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad \textbf{if } A[i] = A[i+1] \textbf{ then} \\
\quad\quad \textbf{return } \textsc{false} \\
\textbf{return } \textsc{true}
\end{array}
$$

Memory state (9)	
$A[0,\ldots,7]$	[2 3 5 6 7 8 9 9]
n	8
i	**7**
$A[i]$	**9**
$A[i+1]$	**9**
$A[i] = A[i+1]$	TRUE

- What is the complexity of this approach?

Uniqueness checking

- Let's examine a pre-sorting approach

$$
\begin{array}{l}
\textsc{MergeSort}(A, n) \\
\textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad \textbf{if } A[i] = A[i+1] \textbf{ then} \\
\quad\quad \textbf{return } \textsc{false} \\
\textbf{return } \textsc{true}
\end{array}
$$

Memory state (9)	
$A[0, \ldots, 7]$	[2 3 5 6 7 8 9 9]
n	8
i	**7**
$A[i]$	**9**
$A[i+1]$	**9**
$A[i] = A[i+1]$	TRUE

- What is the complexity of this approach?
  - The sorting step takes $O(n \log n)$
  - The search step takes $O(n)$
  - The overall complexity is $O(n \log n)$

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

[42, 78, 13, 57, 42, 57, 78, 42]

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

[42, 78, 13, 57, 42, 57, 78, 42]

$\textsc{MergeSort}(A, n)$
$i \leftarrow 0$
$fmax \leftarrow 0$
while $i < n$ **do**
  $seq \leftarrow 1$
  while $i + seq < n$ **and** $A[i + seq] = A[i]$ **do**
    $seq \leftarrow seq + 1$
  if $seq > fmax$ **then**
    $fmax \leftarrow seq$
    $mode \leftarrow A[i]$
  $i \leftarrow i + seq$
return $mode$

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

[42, 78, 13, 57, 42, 57, 78, 42]

Sort the array, the result is [13, **42, 42, 42**, 57, 57, 78, 78]

```
MERGESORT(A, n)
i ← 0
fmax ← 0
while i < n do
    seq ← 1
    while i + seq < n and A[i + seq] = A[i] do
        seq ← seq + 1
    if seq > fmax then
        fmax ← seq
        mode ← A[i]
    i ← i + seq
return mode
```

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

[42, 78, 13, 57, 42, 57, 78, 42]

Index counter

$$
\begin{array}{l}
\textsc{MergeSort}(A, n) \\
i \leftarrow 0 \\
\mathit{fmax} \leftarrow 0 \\
\textbf{while } i < n \textbf{ do} \\
\quad \mathit{seq} \leftarrow 1 \\
\quad \textbf{while } i + \mathit{seq} < n \textbf{ and } A[i + \mathit{seq}] = A[i] \textbf{ do} \\
\quad\quad \mathit{seq} \leftarrow \mathit{seq} + 1 \\
\quad \textbf{if } \mathit{seq} > \mathit{fmax} \textbf{ then} \\
\quad\quad \mathit{fmax} \leftarrow \mathit{seq} \\
\quad\quad \mathit{mode} \leftarrow A[i] \\
\quad i \leftarrow i + \mathit{seq} \\
\textbf{return } \mathit{mode}
\end{array}
$$

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

[42, 78, 13, 57, 42, 57, 78, 42]

Frequency of the most common element so far

$\textsc{MergeSort}(A, n)$
$i \leftarrow 0$
$fmax \leftarrow 0$
while $i < n$ **do**
    $seq \leftarrow 1$
    while $i + seq < n$ **and** $A[i + seq] = A[i]$ **do**
        $seq \leftarrow seq + 1$
    if $seq > fmax$ **then**
        $fmax \leftarrow seq$
        $mode \leftarrow A[i]$
    $i \leftarrow i + seq$
return $mode$

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

[42, 78, 13, 57, 42, 57, 78, 42]

This counter keeps track of sequences of equal numbers

$\textsc{MergeSort}(A, n)$
$i \leftarrow 0$
$\mathit{fmax} \leftarrow 0$
while $i < n$ **do**
    $\mathit{seq} \leftarrow 1$
    while $i + \mathit{seq} < n$ **and** $A[i + \mathit{seq}] = A[i]$ **do**
        $\mathit{seq} \leftarrow \mathit{seq} + 1$
    if $\mathit{seq} > \mathit{fmax}$ **then**
        $\mathit{fmax} \leftarrow \mathit{seq}$
        $\mathit{mode} \leftarrow A[i]$
    $i \leftarrow i + \mathit{seq}$
return mode

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

$$[42, 78, 13, 57, 42, 57, 78, 42]$$

While we do not overflow, and the sequence continues

```
MergeSort(A, n)
i ← 0
fmax ← 0
while i < n do
    seq ← 1
    while i + seq < n and A[i + seq] = A[i] do
        seq ← seq + 1
    if seq > fmax then
        fmax ← seq
        mode ← A[i]
    i ← i + seq
return mode
```

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

[42, 78, 13, 57, 42, 57, 78, 42]

Increase the sequence counter

```
MergeSort(A, n)
i ← 0
fmax ← 0
while i < n do
    seq ← 1
    while i + seq < n and A[i + seq] = A[i] do
        seq ← seq + 1
    if seq > fmax then
        fmax ← seq
        mode ← A[i]
    i ← i + seq
return mode
```

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

[42, 78, 13, 57, 42, 57, 78, 42]

If the sequence is the largest so far…

$\textsc{MergeSort}(A, n)$
$i \leftarrow 0$
$\mathit{fmax} \leftarrow 0$
while $i < n$ **do**
  $\mathit{seq} \leftarrow 1$
  while $i + \mathit{seq} < n$ **and** $A[i + \mathit{seq}] = A[i]$ **do**
    $\mathit{seq} \leftarrow \mathit{seq} + 1$
  if $\mathit{seq} > \mathit{fmax}$ **then**
    $\mathit{fmax} \leftarrow \mathit{seq}$
    $\mathit{mode} \leftarrow A[i]$
  $i \leftarrow i + \mathit{seq}$
return mode

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

[42, 78, 13, 57, 42, 57, 78, 42]

$\text{MergeSort}(A, n)$
$i \leftarrow 0$
$\mathit{fmax} \leftarrow 0$
while $i < n$ **do**
  $\mathit{seq} \leftarrow 1$
  while $i + \mathit{seq} < n$ **and** $A[i + \mathit{seq}] = A[i]$ **do**
    $\mathit{seq} \leftarrow \mathit{seq} + 1$
  if $\mathit{seq} > \mathit{fmax}$ **then**
    $\mathit{fmax} \leftarrow \mathit{seq}$
    $\mathit{mode} \leftarrow A[i]$
  $i \leftarrow i + \mathit{seq}$
return mode

Update both the frequency and mode variables

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

[42, 78, 13, 57, 42, 57, 78, 42]

$\textsc{MergeSort}(A, n)$
$i \leftarrow 0$
$fmax \leftarrow 0$
while $i < n$ **do**
    $seq \leftarrow 1$
    while $i + seq < n$ **and** $A[i + seq] = A[i]$ **do**
        $seq \leftarrow seq + 1$
    if $seq > fmax$ **then**
        $fmax \leftarrow seq$
        $mode \leftarrow A[i]$
    $i \leftarrow i + seq$ ← Skip the complete sequence of equal numbers
return $mode$

Finding the mode

- Take a few minutes to **think of a design** for mode-finding algorithm based on the pre-sorting version of uniqueness check. Test it with the following data, whose solution is **42**

[42, 78, 13, 57, 42, 57, 78, 42]

$\textsc{MergeSort}(A, n)$
$i \leftarrow 0$
$\mathit{fmax} \leftarrow 0$
while $i < n$ **do**
  $\mathit{seq} \leftarrow 1$
  while $i + \mathit{seq} < n$ **and** $A[i + \mathit{seq}] = A[i]$ **do**
    $\mathit{seq} \leftarrow \mathit{seq} + 1$
  if $\mathit{seq} > \mathit{fmax}$ **then**
    $\mathit{fmax} \leftarrow \mathit{seq}$
    $\mathit{mode} \leftarrow A[i]$
  $i \leftarrow i + \mathit{seq}$
return mode

- What is the complexity of this approach?

A challenge for home…

- An **anagram** of a word w is a word which uses the same letters as w but in a different order. For example:
  - 'ate', 'tea' and 'eat' are anagrams.
  - 'post', 'spot', 'pots' and 'tops' are anagrams.
  - 'garner' and 'ranger' are anagrams.

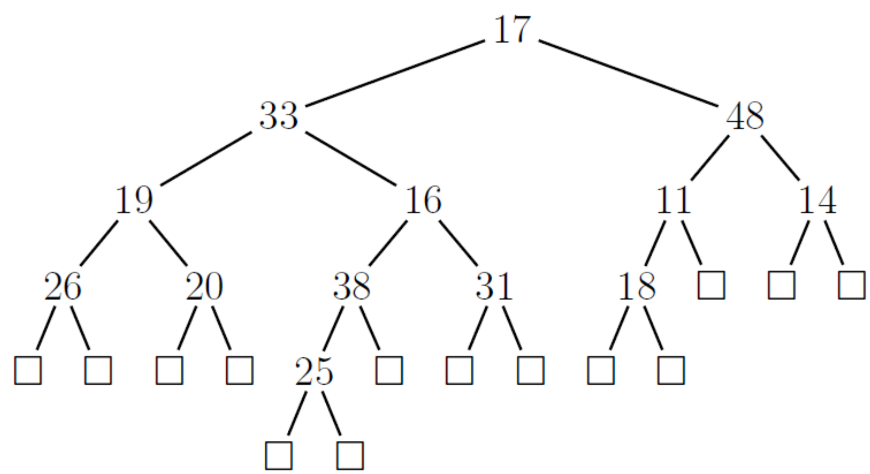
- You are given a very long list of words:

  {health, revolution, foolish, garner, drive, praise, traverse, anger, ranger, … scoop, fall, praise}

- **Design** an algorithm to find all anagrams in the list using pre-sorting

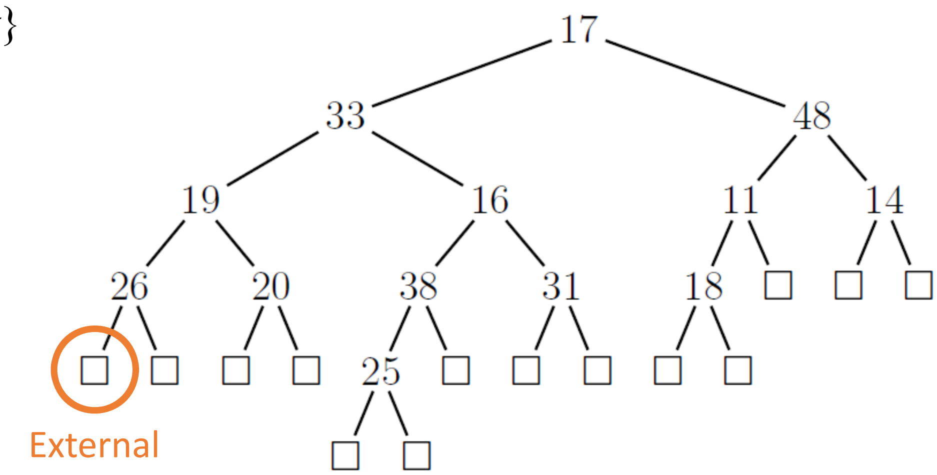
Representational change

- On lecture 12, you discussed **binary trees** in general
  - Each **node** has the fields $\{root, left, right\}$



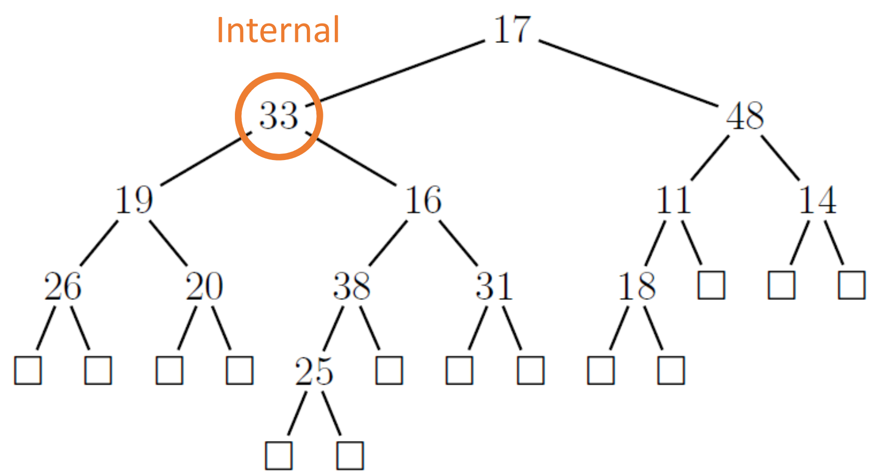
Representational change

- On lecture 12, you discussed **binary trees** in general
  - Each **node** has the fields $\{root, left, right\}$
  - **Empty** subtrees are marked by **null** pointers, and they are often called **external** nodes



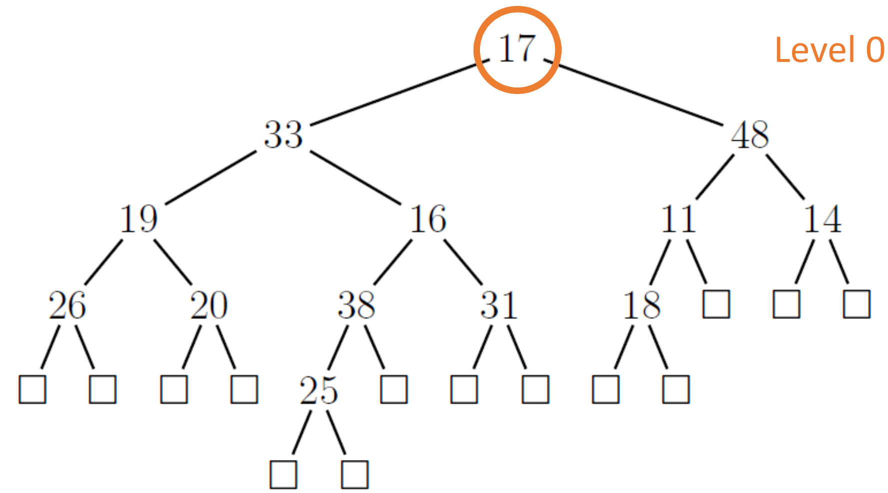
Representational change

- On lecture 12, you discussed **binary trees** in general
  - Each **node** has the fields $\{root, left, right\}$
  - **Empty** subtrees are marked by **null** pointers, and they are often called **external** nodes
  - **Internal** nodes have a $root \neq \text{NULL}$



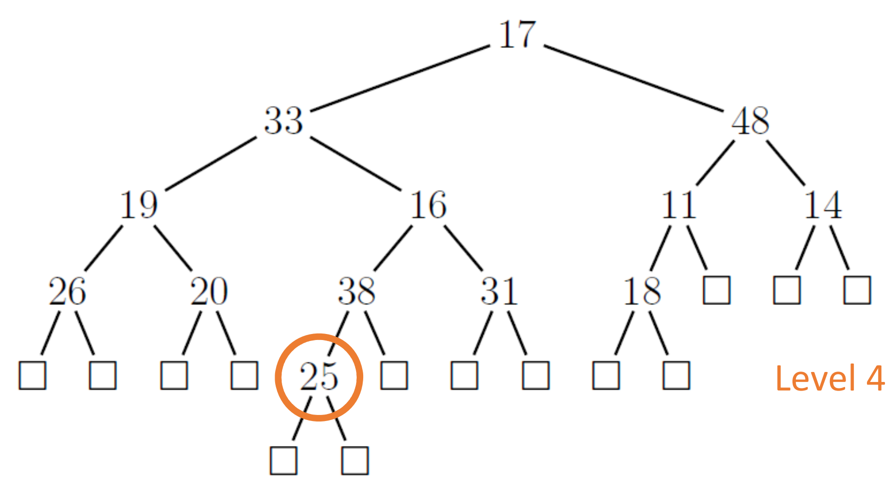
Representational change

- On lecture 12, you discussed **binary trees** in general
  - Each **node** has the fields $\{root, left, right\}$
  - **Empty** subtrees are marked by **null** pointers, and they are often called **external** nodes
  - **Internal** nodes have a $root \neq \text{NULL}$
  - The **root** of the tree is at level 0



Representational change

- On lecture 12, you discussed **binary trees** in general
  - Each **node** has the fields $\{root, left, right\}$
  - **Empty** subtrees are marked by **null** pointers, and they are often called **external** nodes
  - **Internal** nodes have a $root \neq \text{NULL}$
  - The **root** of the tree is at level 0
  - This tree has a **height** of 4



Binary search trees

- A **binary search tree (BST)** is a binary tree that stores elements in all internal nodes, with each sub-tree satisfying the property:

Let the root be *r*; then each element in the **left subtree is smaller** than *r* and each element in the **right sub-tree is larger** than *r*

- For simplicity we assume that all keys are **different**

How to … an element in a BST?

- To **search** for an element k in a BST, we compare against the root r.
  - If r=k, we are done
  - Otherwise, search to the **left** if $k<r$ and to the **right** if *k*>*r*

- To **insert** a new element k into a BST, we pretend to search for *k*.
  - Once we reach an **empty** sub-tree, we insert k in that position.

- Let’s take a few minutes to **build** a tree by inserting [15 8 20 5 9 17 25 29 2 6 12 10] one at the time.

The importance of being *balanced*

- If a BST with n elements is kept **“reasonably” balanced**, search involves $\Theta(\log n)$ comparisons in the worst case

- If the BST is **unbalanced**, search performance may degrade to be as bad as linear search

- Let’s take a few minutes to build a BST by inserting [325 18 21 212 157 105] one at the time

Next lecture

- Balanced binary search trees
  - AVL trees and 2–3 trees (Levitin Section 6.3)