# COMP90038
# Algorithms and Complexity

## Lecture 14: Transform-and-Conquer
## (with thanks to Harald Søndergaard & Michael Kirley)

Casey Myers
Casey.Myers@unimelb.edu.au
David Caro Building (Physics) 274

# Review from Lecture 13

- We saw priority queues, heaps and heapsort.
- A priority queue is a set of elements, each containing a priority (key) value. Elements with higher priorities are ejected first.
- A heap is a a complete binary tree that satisfies the condition:
  - Each child has a priority (key) which is not greater than its parents.
- Heapsort is a sorting algorithm that repeatedly ejects the higher priority element, then turns the remaining binary tree into a Heap.

# Transform and Conquer

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

# Transform and Conquer

- Transform-and-Conquer is a group of design techniques that:
  - *Transform*: Modify the problem to a more amenable form, and then
  - *Conquer*: Solve it using known efficient algorithms.
- There are three major variations:
  - Instance simplification
  - Representational change
  - Problem reduction

# Transform and Conquer

- Transform-and-Conquer is a group of design techniques that:
  - *Transform*: Modify the problem to a more amenable form
  - *Conquer*: Solve it using known efficient algorithms.
- There are three major variations:
  - Instance simplification
  - Representational change
  - Problem reduction

# Instance Simplification

- General principle: Try to make the problem easier through some sort of pre-processing typically sorting.

- We can pre-sort input to speed up, for example:

  – Finding the median

  – Uniqueness checking

  – Finding the mode

# Instance Simplification

- General principle: Try to make the problem easier through some sort of pre-processing typically sorting.

- We can pre-sort input to speed up, for example:
  - Finding the median
  - Uniqueness checking
  - Finding the mode

| | |
|---|---|
| Selection sort | $\Theta(n^2)$ |
| Insertion sort | $O(n^2)$ |
| Shellsort | $O(n\sqrt{n})$ |
| Mergesort | $\Theta(n \log n)$: worst case |
| Quicksort | $\Theta(n \log n)$: average<br>$\Theta(n^2)$: worst case |
| Heapsort | $\Theta(n \log n)$ |

# Uniqueness Checking, Brute-Force

- The problem:
  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

# Uniqueness Checking, Brute-Force

- The problem:

  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
→ 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
```

| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
|---|---|
| $n$ | 8 |
| $i$ | 0 |
| $j$ | |
| $A[i]$ | 2 |
| $A[j]$ | |
| $A[i] = A[j]$ | |

# Uniqueness Checking, Brute-Force

- The problem:
  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
| $n$ | 8 |
| $i$ | 0 |
| $j$ | 1 |
| $A[i]$ | 2 |
| $A[j]$ | 9 |
| $A[i] = A[j]$ | |

# Uniqueness Checking, Brute-Force

- The problem:

  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
| $n$ | 8 |
| $i$ | 0 |
| $j$ | 1 |
| $A[i]$ | 2 |
| $A[j]$ | 9 |
| $A[i] = A[j]$ | *FALSE* |

# Uniqueness Checking, Brute-Force

- The problem:

  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

$$
\begin{aligned}
&\textbf{for } i \leftarrow 0 \textbf{ to } n-2 \textbf{ do}\\
&\quad \textbf{for } j \leftarrow i+1 \textbf{ to } n-1 \textbf{ do}\\
&\qquad \textbf{if } A[i] = A[j] \textbf{ then}\\
&\qquad\quad \textbf{return } \text{False}\\
&\textbf{return } \text{True}
\end{aligned}
$$

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
| $n$ | 8 |
| $i$ | 0 |
| $j$ | 2 |
| $A[i]$ | 2 |
| $A[j]$ | 8 |
| $A[i] = A[j]$ | *FALSE* |

# Uniqueness Checking, Brute-Force

- The problem:
  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
|---|---|
| $n$ | 8 |
| $i$ | 0 |
| $j$ | 3 |
| $A[i]$ | 2 |
| $A[j]$ | 6 |
| $A[i] = A[j]$ | *FALSE* |

# Uniqueness Checking, Brute-Force

- The problem:
  Given an unsorted array $A[0,\cdots,n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

| | |
|---|---|
| $A[0,\cdots,n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
| $n$ | 8 |
| $i$ | 0 |
| $j$ | 4 |
| $A[i]$ | 2 |
| $A[j]$ | 9 |
| $A[i] = A[j]$ | *FALSE* |

# Uniqueness Checking, Brute-Force

- The problem:
  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
|---|---|
| $n$ | 8 |
| $i$ | 0 |
| $j$ | 5 |
| $A[i]$ | 2 |
| $A[j]$ | 5 |
| $A[i] = A[j]$ | *FALSE* |

# Uniqueness Checking, Brute-Force

- The problem:
  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
| $n$ | 8 |
| $i$ | 0 |
| $j$ | 6 |
| $A[i]$ | 2 |
| $A[j]$ | 7 |
| $A[i] = A[j]$ | *FALSE* |

# Uniqueness Checking, Brute-Force

- The problem:

  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
| $n$ | 8 |
| $i$ | 0 |
| $j$ | 7 |
| $A[i]$ | 2 |
| $A[j]$ | 3 |
| $A[i] = A[j]$ | *FALSE* |

# Uniqueness Checking, Brute-Force

- The problem:

  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
|---|---|
| $n$ | 8 |
| $i$ | 1 |
| $j$ | 2 |
| $A[i]$ | 9 |
| $A[j]$ | 8 |
| $A[i] = A[j]$ | *FALSE* |

# Uniqueness Checking, Brute-Force

- The problem:
  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
| $n$ | 8 |
| $i$ | 1 |
| $j$ | 3 |
| $A[i]$ | 9 |
| $A[j]$ | 6 |
| $A[i] = A[j]$ | *FALSE* |

# Uniqueness Checking, Brute-Force

- The problem:

  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
| $n$ | 8 |
| $i$ | 1 |
| $j$ | 4 |
| $A[i]$ | 9 |
| $A[j]$ | 9 |
| $A[i] = A[j]$ | *TRUE* |

# Uniqueness Checking, Brute-Force

- The problem:
  Given an unsorted array $A[0, \cdots, n-1]$, is $A[i] \neq A[j]$ whenever $i \neq j$?

- The obvious approach is brute-force:

```
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
```

- What is the complexity of this? $O(n^2)$

# Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

| | |
|---|---|
| Selection sort | $\Theta(n^2)$ |
| Insertion sort | $O(n^2)$ |
| Shellsort | $O(n\sqrt{n})$ |
| Mergesort | $\Theta(n \log n)$: worst case |
| Quicksort | $\Theta(n \log n)$: average<br>$\Theta(n^2)$: worst case |
| Heapsort | $\Theta(n \log n)$ |

# Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

$\textsc{Sort}(A[0..n-1])$
**for** $i \leftarrow 0$ to $n-2$ **do**
  **if** $A[i] = A[i+1]$ **then**
    **return** False
**return** True

| | |
|---|---|
| Selection sort | $\Theta(n^2)$ |
| Insertion sort | $O(n^2)$ |
| Shellsort | $O(n\sqrt{n})$ |
| Mergesort | $\Theta(n \log n)$: worst case |
| Quicksort | $\Theta(n \log n)$: average<br>$\Theta(n^2)$: worst case |
| Heapsort | $\Theta(n \log n)$ |

# Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

$\text{SORT}(A[0..n-1])$
**for** $i \leftarrow 0$ to $n-2$ **do**
  **if** $A[i] = A[i+1]$ **then**
    **return** False
**return** True

| | |
|---|---|
| Selection sort | $\Theta(n^2)$ |
| Insertion sort | $O(n^2)$ |
| Shellsort | $O(n\sqrt{n})$ |
| Mergesort | $\Theta(n \log n)$: worst case |
| Quicksort | $\Theta(n \log n)$: average<br>$\Theta(n^2)$: worst case |
| Heapsort | $\Theta(n \log n)$ |

- What is the complexity of this? $O(n \log n) + O(n) = O(n \log n)$

# Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

```
SORT(A[0..n − 1])
for i ← 0 to n − 2 do
    if A[i] = A[i + 1] then
        return False
return True
```

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 9, 8, 6, 9, 5, 7, 3]$ |
| $n$ | 8 |
| $i$ | |
| $A[i]$ | |
| $A[i+1]$ | |
| $A[i] = A[i+1]$ | |

# Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

$\textsc{Sort}(A[0..n-1])$
**for** $i \leftarrow 0$ **to** $n-2$ **do**
    **if** $A[i] = A[i+1]$ **then**
        **return** False
**return** True

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 3, 5, 6, 7, 8, 9, 9]$ |
| $n$ | 8 |
| $i$ | 0 |
| $A[i]$ | |
| $A[i+1]$ | |
| $A[i] = A[i+1]$ | |

# Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

$\textsc{Sort}(A[0..n-1])$
**for** $i \leftarrow 0$ to $n-2$ **do**
  **if** $A[i] = A[i+1]$ **then**
    **return** False
**return** True

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 3, 5, 6, 7, 8, 9, 9]$ |
| $n$ | 8 |
| $i$ | 1 |
| $A[i]$ | 3 |
| $A[i+1]$ | 5 |
| $A[i] = A[i+1]$ | *FALSE* |

# Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

$\text{SORT}(A[0..n-1])$
**for** $i \leftarrow 0$ to $n-2$ **do**
  **if** $A[i] = A[i+1]$ **then**
    **return** False
**return** True

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 3, 5, 6, 7, 8, 9, 9]$ |
| $n$ | 8 |
| $i$ | 2 |
| $A[i]$ | 5 |
| $A[i+1]$ | 6 |
| $A[i] = A[i+1]$ | *FALSE* |

# Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

```
SORT(A[0..n − 1])
for i ← 0 to n − 2 do
    if A[i] = A[i + 1] then
        return False
return True
```

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 3, 5, 6, 7, 8, 9, 9]$ |
| $n$ | 8 |
| $i$ | 3 |
| $A[i]$ | 6 |
| $A[i+1]$ | 7 |
| $A[i] = A[i+1]$ | $FALSE$ |

# Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

```
SORT(A[0..n − 1])
for i ← 0 to n − 2 do
    if A[i] = A[i + 1] then
        return False
return True
```

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 3, 5, 6, 7, 8, 9, 9]$ |
| $n$ | 8 |
| $i$ | 4 |
| $A[i]$ | 7 |
| $A[i+1]$ | 8 |
| $A[i] = A[i+1]$ | *FALSE* |

# Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

$\text{SORT}(A[0..n-1])$
**for** $i \leftarrow 0$ to $n-2$ **do**
    **if** $A[i] = A[i+1]$ **then**
        **return** False
**return** True

| | |
|---|---|
| $A[0, \cdots, n-1]$ | $[2, 3, 5, 6, 7, 8, 9, 9]$ |
| $n$ | 8 |
| $i$ | 5 |
| $A[i]$ | 8 |
| $A[i+1]$ | 9 |
| $A[i] = A[i+1]$ | *FALSE* |

# Uniqueness Checking, with Presorting

- Sorting makes the problem easier:

```
SORT(A[0..n − 1])
for i ← 0 to n − 2 do
    if A[i] = A[i + 1] then
        return False
return True
```

| | |
|---|---|
| $A[0,\cdots,n-1]$ | $[2,3,5,6,7,8,9,9]$ |
| $n$ | 8 |
| $i$ | 6 |
| $A[i]$ | 9 |
| $A[i+1]$ | 9 |
| $A[i]=A[i+1]$ | $TRUE$ |

# Exercise: Computing the Mode

- A mode is a list of array elements which occurs most frequently in the list/array.

- For example, in:

42, 78, 13, 57, 42, 57, 78, 42

the element 42 is the mode.

# Exercise: Computing the Mode

- A mode is a list of array elements which occurs most frequently in the list/array.
- For example, in:

  42, 78, 13, 57, 42, 57, 78, 42

  the element 42 is the mode.
- The problem:

  Given an array $A$, find a mode.
- Discuss a brute-force approach vs a pre-sorting approach.

# Mode Finding with Presorting

```
SORT(A[0..n − 1])
i ← 0
maxfreq ← 0
while i < n do
    runlength ← 1
    while i + runlength < n and A[i + runlength] = A[i] do
        runlength ← runlength + 1
    if runlength > maxfreq then
        maxfreq ← runlength
        mode ← A[i]
    i ← i + runlength
return mode
```

# Mode Finding with Presorting

$$A[0, \cdots, n-1] = [42, 78, 13, 57, 42, 57, 78, 42]$$

Sort the array: [13, 42, 42, 42, 57, 57, 78, 78]

$\textsc{Sort}(A[0..n-1])$
$i \leftarrow 0$
$maxfreq \leftarrow 0$
**while** $i < n$ **do**
    $runlength \leftarrow 1$
    **while** $i + runlength < n$ **and** $A[i + runlength] = A[i]$ **do**
        $runlength \leftarrow runlength + 1$
    **if** $runlength > maxfreq$ **then**
        $maxfreq \leftarrow runlength$
        $mode \leftarrow A[i]$
    $i \leftarrow i + runlength$
**return** $mode$

# Mode Finding with Presorting

$$A[0, \cdots, n-1] = [42, 78, 13, 57, 42, 57, 78, 42]$$

$\textsc{Sort}(A[0..n-1])$
$i \leftarrow 0$
$maxfreq \leftarrow 0$ ← Frequency of the most common element so far
**while** $i < n$ **do**
  $runlength \leftarrow 1$
  **while** $i + runlength < n$ **and** $A[i + runlength] = A[i]$ **do**
    $runlength \leftarrow runlength + 1$
  **if** $runlength > maxfreq$ **then**
    $maxfreq \leftarrow runlength$
    $mode \leftarrow A[i]$
  $i \leftarrow i + runlength$
**return** $mode$

# Mode Finding with Presorting

$$A[0, \cdots, n-1] = [42, 78, 13, 57, 42, 57, 78, 42]$$

```
SORT(A[0..n − 1])
i ← 0
maxfreq ← 0
while i < n do
    runlength ← 1
    while i + runlength < n and A[i + runlength] = A[i] do
        runlength ← runlength + 1
    if runlength > maxfreq then
        maxfreq ← runlength
        mode ← A[i]
    i ← i + runlength
return mode
```

This counter keeps track of sequences of equal numbers

# Mode Finding with Presorting

$$A[0, \cdots, n-1] = [42, 78, 13, 57, 42, 57, 78, 42]$$

SORT($A[0..n-1]$)
$i \leftarrow 0$
$maxfreq \leftarrow 0$
**while** $i < n$ **do**
  $runlength \leftarrow 1$
  **while** $i + runlength < n$ **and** $A[i + runlength] = A[i]$ **do**
    $runlength \leftarrow runlength + 1$
  **if** $runlength > maxfreq$ **then**
    $maxfreq \leftarrow runlength$
    $mode \leftarrow A[i]$
  $i \leftarrow i + runlength$
**return** $mode$

While we do not overflow and the sequence continues

# Mode Finding with Presorting

$$A[0, \cdots, n-1] = [42, 78, 13, 57, 42, 57, 78, 42]$$

$\text{SORT}(A[0..n-1])$
$i \leftarrow 0$
$\mathit{maxfreq} \leftarrow 0$
**while** $i < n$ **do**
  $\mathit{runlength} \leftarrow 1$
  **while** $i + \mathit{runlength} < n$ **and** $A[i + \mathit{runlength}] = A[i]$ **do**
    $\mathit{runlength} \leftarrow \mathit{runlength} + 1$ ← Increase the sequence counter
  **if** $\mathit{runlength} > \mathit{maxfreq}$ **then**
    $\mathit{maxfreq} \leftarrow \mathit{runlength}$
    $\mathit{mode} \leftarrow A[i]$
  $i \leftarrow i + \mathit{runlength}$
**return** $\mathit{mode}$

# Mode Finding with Presorting

$$A[0, \cdots, n-1] = [42, 78, 13, 57, 42, 57, 78, 42]$$

```
SORT(A[0..n − 1])
i ← 0
maxfreq ← 0
while i < n do
    runlength ← 1
    while i + runlength < n and A[i + runlength] = A[i] do
        runlength ← runlength + 1
    if runlength > maxfreq then        ← If the sequence is the largest so far
        maxfreq ← runlength
        mode ← A[i]
    i ← i + runlength
return mode
```

# Mode Finding with Presorting

$$A[0, \cdots, n-1] = [42, 78, 13, 57, 42, 57, 78, 42]$$

$\textsc{Sort}(A[0..n-1])$
$i \leftarrow 0$
$maxfreq \leftarrow 0$
**while** $i < n$ **do**
    $runlength \leftarrow 1$
    **while** $i + runlength < n$ **and** $A[i + runlength] = A[i]$ **do**
        $runlength \leftarrow runlength + 1$
    **if** $runlength > maxfreq$ **then**
        $maxfreq \leftarrow runlength$
        $mode \leftarrow A[i]$
    $i \leftarrow i + runlength$
**return** $mode$

Update both the frequency and mode variables

# Mode Finding with Presorting

$$A[0, \cdots, n-1] = [42, 78, 13, 57, 42, 57, 78, 42]$$

$\textsc{Sort}(A[0..n-1])$
$i \leftarrow 0$
$maxfreq \leftarrow 0$
**while** $i < n$ **do**
  $runlength \leftarrow 1$
  **while** $i + runlength < n$ **and** $A[i + runlength] = A[i]$ **do**
    $runlength \leftarrow runlength + 1$
  **if** $runlength > maxfreq$ **then**
    $maxfreq \leftarrow runlength$
    $mode \leftarrow A[i]$
  $i \leftarrow i + runlength$ ← Skip the complete sequence of equal numbers
**return** $mode$

# Mode Finding with Presorting

```
SORT(A[0..n − 1])
i ← 0
maxfreq ← 0
while i < n do
    runlength ← 1
    while i + runlength < n and A[i + runlength] = A[i] do
        runlength ← runlength + 1
    if runlength > maxfreq then
        maxfreq ← runlength
        mode ← A[i]
    i ← i + runlength
return mode
```

- Again, after sorting, the rest takes linear time.

# Searching with Presorting

- The problem:

  Given unsorted array $A$, find item $x$ (or determine that it is absent).

- Compare these two approaches:
  - Perform a sequential search
  - Sort, then perform binary search
- What are the complexities of these approaches?

# Searching with Presorting

- What if we need to search for $m$ items?
- Let us do a back-of-the envelope calculation (consider worst-cases for simplicity):
  - Take $\mathrm{n} = 1024$ and $m = 32$.
  - Sequential search: $\mathrm{m} \times \mathrm{n} = 32,768$
  - Sorting + binsearch: $n\, log_2 n + \mathrm{m} \times log_2 n = 10{,}240 + 320 = 10{,}560$.
  - Average-case analysis will look somewhat better for sequential search, but pre-sorting will still win.

# Exercise: Finding Anagrams

- An anagram of a word *w* is a word which uses the same letter as *w* but in a different order.
  - Example: 'ate', 'tea' and 'eat' are anagrams.
  - Example: 'post', 'spot', 'pots' and 'tops' are anagrams.
  - Example: 'garner' and 'ranger' are anagrams.
- You are given a very long list of words in lexicographic order.
- Device an algorithm to find all anagrams in the list.

# Exercise: Finding Anagrams

```
words = ['bat', 'rats', 'god', 'dog', 'cat', 'arts', 'star']
sort_words = {}
for word in words:
    sort_words[word] = ''.join(sorted(word))

print sort_words
anagrams = []
for i in range(len(words)):
    ana = [words[i]]
    for j in range(i + 1, len(words)):
        if sort_words[words[i]] == sort_words[words[j]]:
            ana.append(words[j])
    if len(ana) != 1:
        anagrams.append(ana)

print anagrams
```

- Finding anagrams from a words list:
- Time complexity?
  - Sorting words
    - $O(n \times m \log m)$
    - $n$ words of length $m$.
  - Test all combinations
    - $O(n^2)$
  - Can we do better?
    - Yes! Sort words as well!
    - Apply “mode idea”

# Binary Search Trees

- A binary search tree, or BST, is a binary tree that stores elements in all internal nodes, with each sin-tree satisfying the BST property:

  Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)

# Binary Search Trees

- A binary search tree, or BST, is a binary tree that stores elements in all internal nodes, with each sin-tree satisfying the BST property:

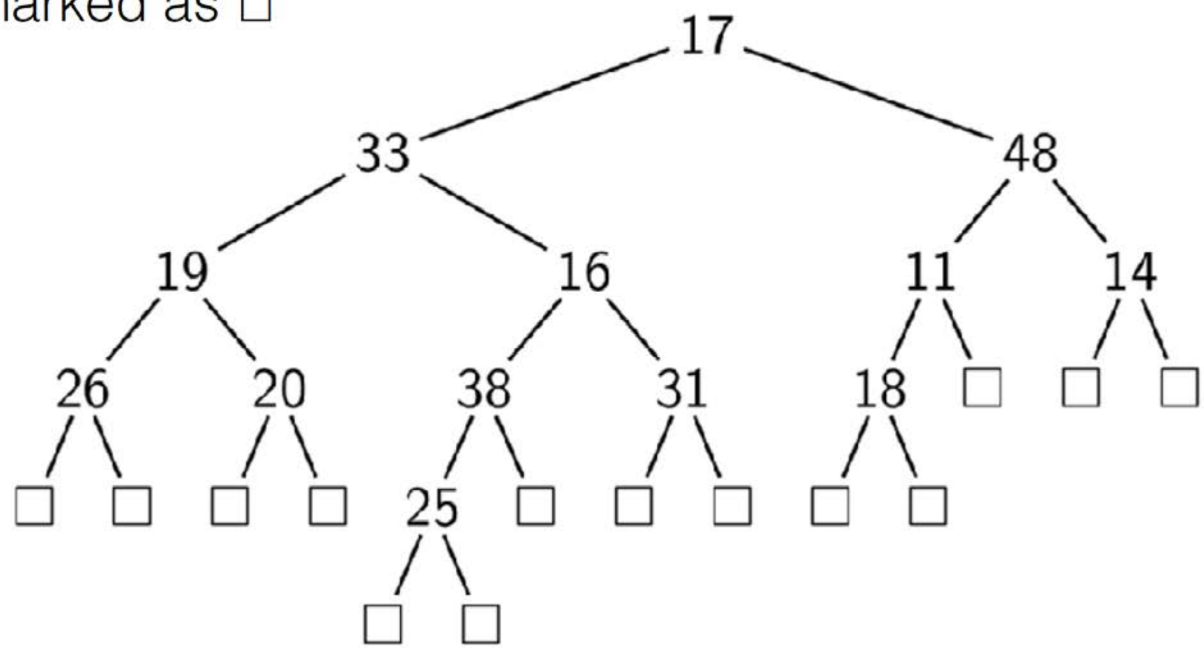
  Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)

- First a review of binary trees from Lecture 12:

# Transform and Conquer

## Binary Trees

- An example of a **binary tree**, with empty subtrees marked as □



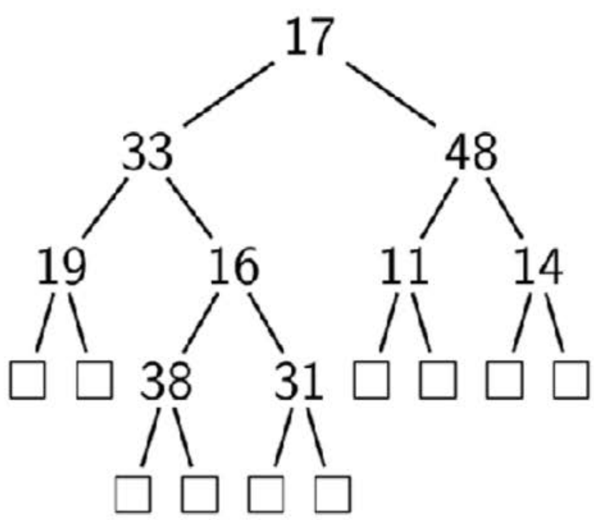

- This tree has **height** 4, the empty tree having height -1

# Review from Lecture 12

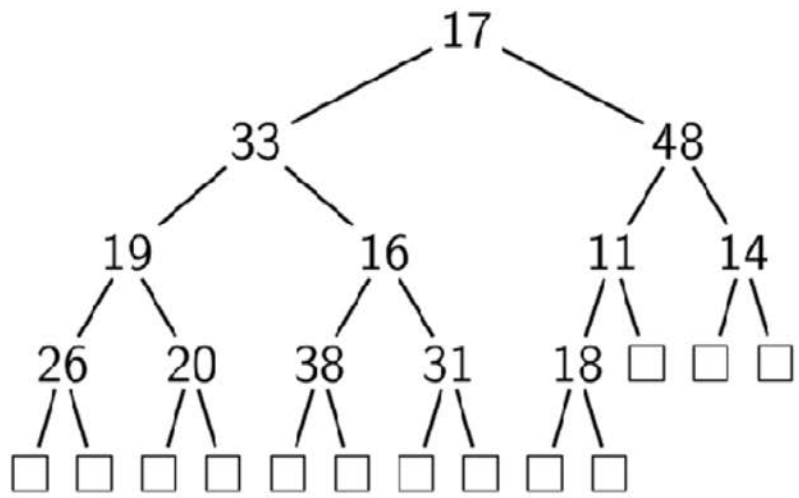
## Binary Tree Concepts

THE UNIVERSITY OF MELBOURNE

- Special trees have their **external nodes** □ only at level $h$ and $h$+1 for some $h$:




A **full** binary tree:
Each node has 0 or 2 (non-empty) children.



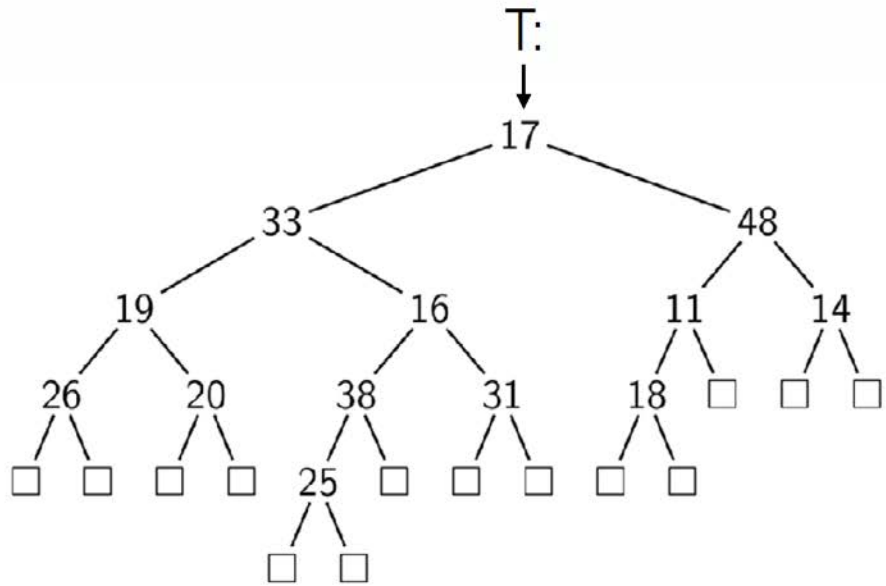

A **complete** tree: Each level filled left to right.
(Every level except perhaps the last is completely filled.)

# Review from Lecture 12

## Calculating the Height

THE UNIVERSITY OF MELBOURNE

- Recursion is the natural way of calculating the **height**:




**function** HEIGHT($T$)
  **if** $T = null$ **then**
    **return** $-1$
  **else**
    **return** $max(\text{HEIGHT}(T.left), \text{HEIGHT}(T.right)) + 1$

# Review from Lecture 12

## Binary Tree Traversal

THE UNIVERSITY OF MELBOURNE

- **Preorder** traversal visits the root, then the left subtree, and finally the right subtree.
- **Inorder** traversal visits the left subtree, then the root, and finally the right subtree.
- **Postorder** traversal visits the left subtree, the right subtree, and finally the root.
- **Level-order** traversal visits the nodes, level by level, starting from the root.

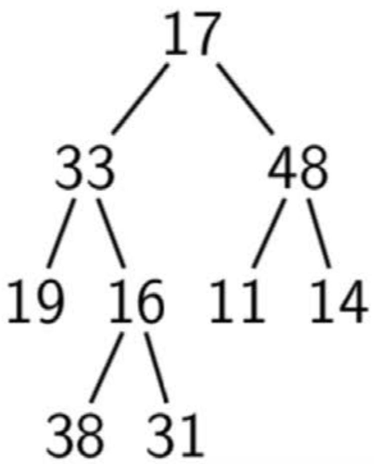
 

# Review from Lecture 12

## Inorder Traversal

Visit order: 19 33 38 16 31 17 11 48 14

**procedure** INORDERTRAVERSE($T$)
  **if** $T \neq null$ **then**
    INORDERTRAVERSE($T.left$)
    visit $T.root$
    INORDERTRAVERSE($T.right$)




Call Stack

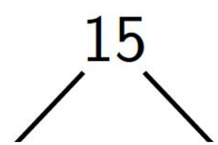
# Build a Binary Search Tree Example

- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)

# Build a Binary Search Tree Example

- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)
- Keep in mind we really mean:

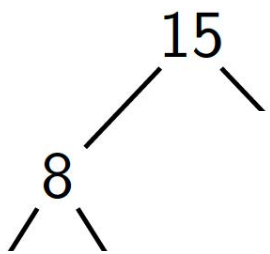
$$root.val \quad = 15$$
$$root.left \quad = null$$
$$root.right \ = null$$



# Build a Binary Search Tree Example

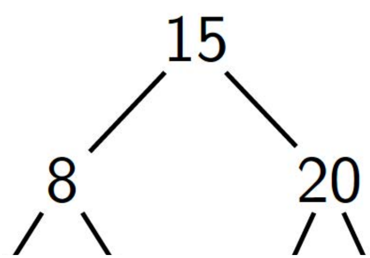
- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)
- Keep in mind we really mean:

$$root.val \quad = 8$$
$$root.left \quad = null$$
$$root.right \quad = null$$



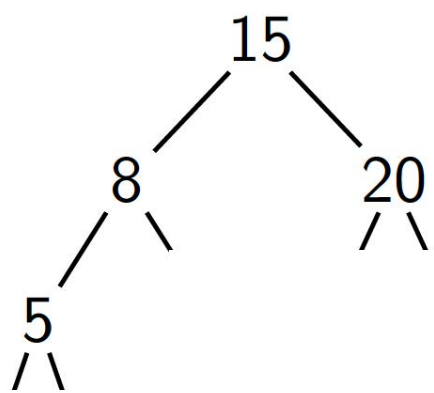
# Build a Binary Search Tree Example

- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)



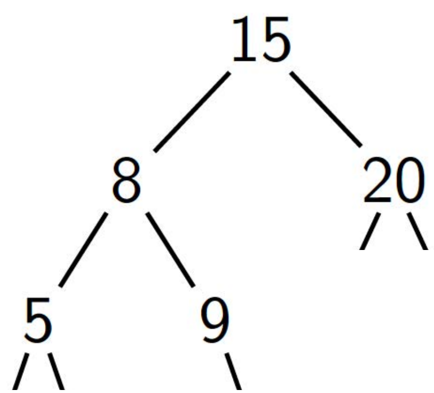
# Build a Binary Search Tree Example

- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)



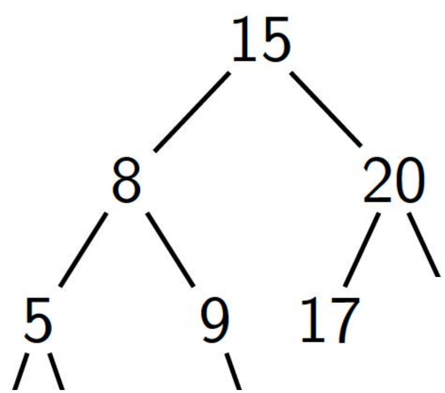
# Build a Binary Search Tree Example

- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)



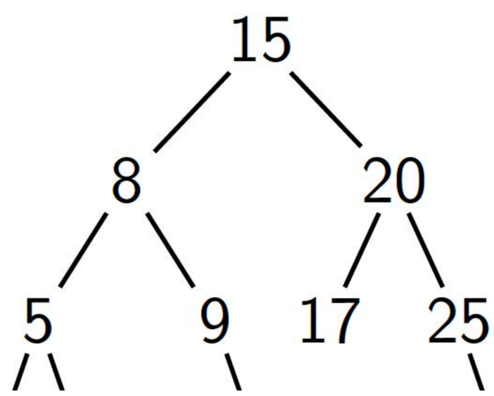
# Build a Binary Search Tree Example

- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)



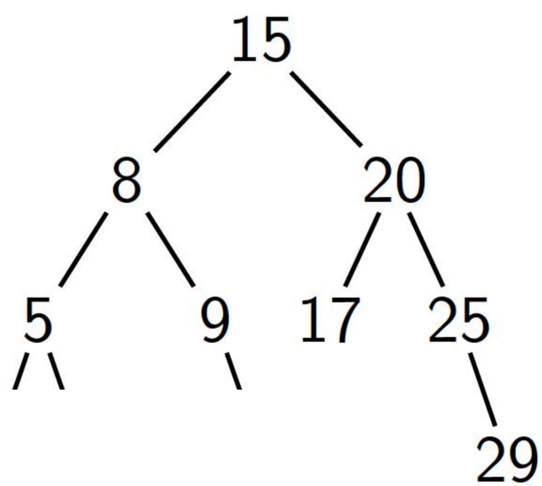
# Build a Binary Search Tree Example

- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)



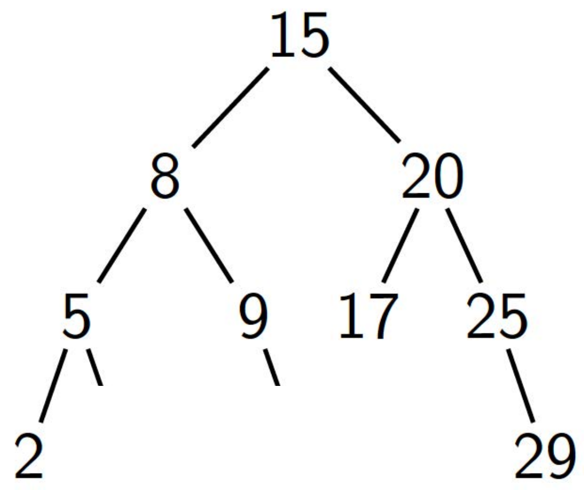
# Build a Binary Search Tree Example

- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)



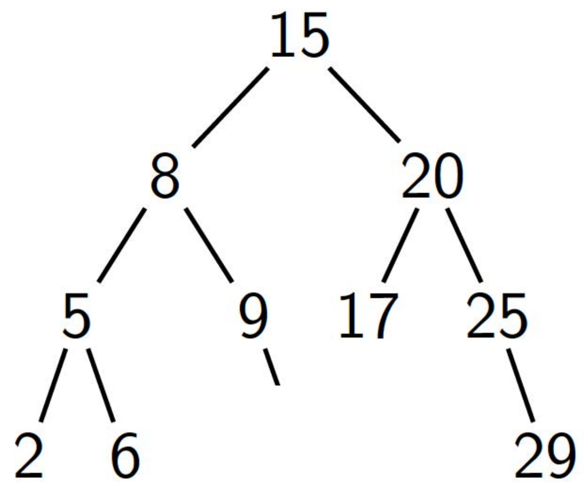
# Build a Binary Search Tree Example

- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)



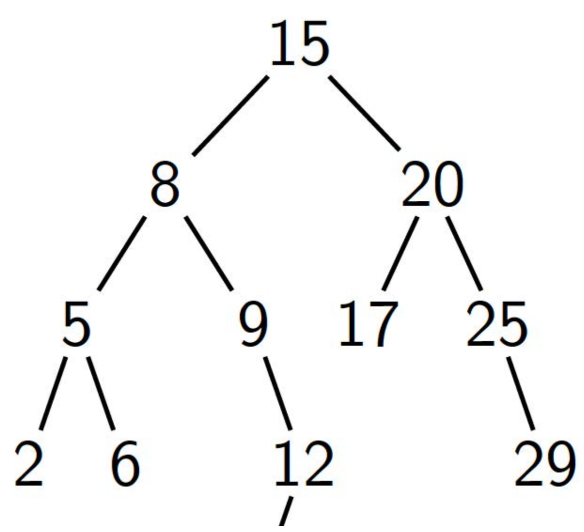
# Build a Binary Search Tree Example

- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)



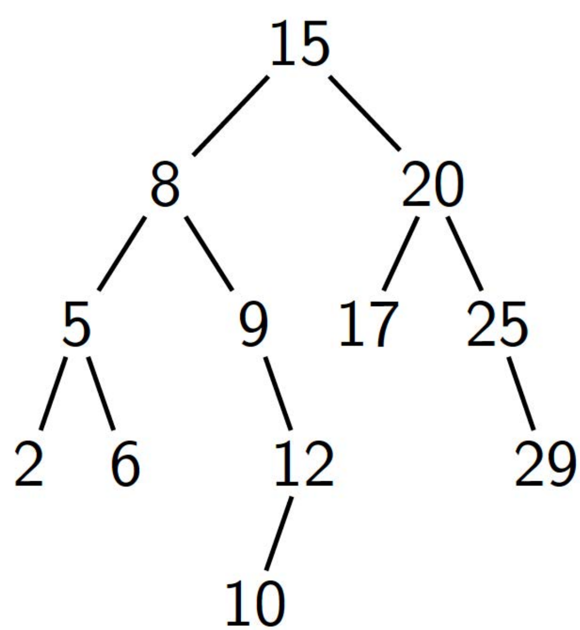
# Build a Binary Search Tree Example

- Let's attempt to build a BST by inserting $[15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10]$ one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)



# Build a Binary Search Tree Example

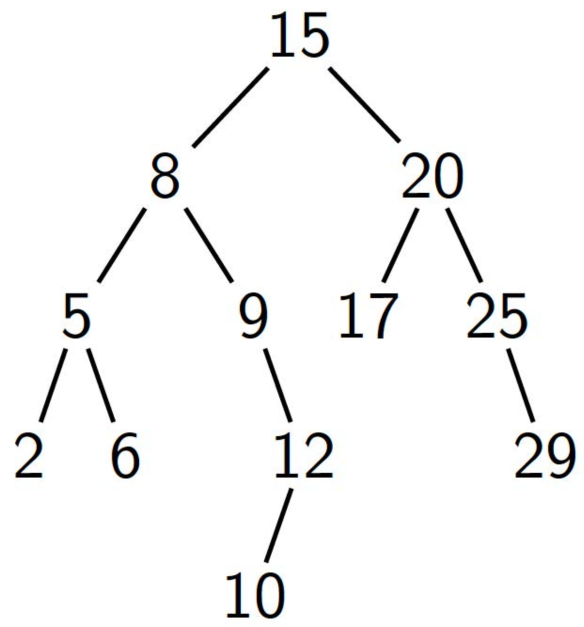
- Let's attempt to build a BST by inserting [15, 8, 20, 5, 9, 17, 25, 29, 2, 6, 12, 10] one at a time.
- Remember: Let the root be $r$; then each element in the left subtree is smaller that $r$ and each element in the right sub-tree is larger than $r$. (For simplicity, we assume that all keys are different.)



# Binary Search Trees

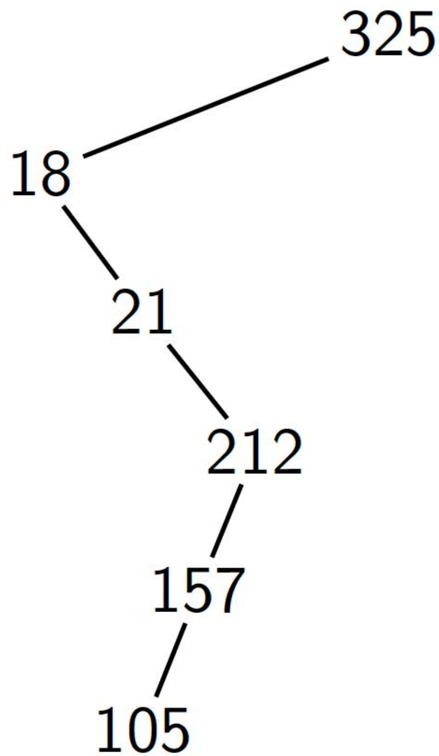
- BSTs are useful for search applications. To search for $k$ in a BST, compare against its root r. If $\mathrm{r} = k$, we are done; otherwise search in the left or right sub-tree, according to $\mathrm{k} < \mathrm{r}$ or $\mathrm{k} > \mathrm{r}$.

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



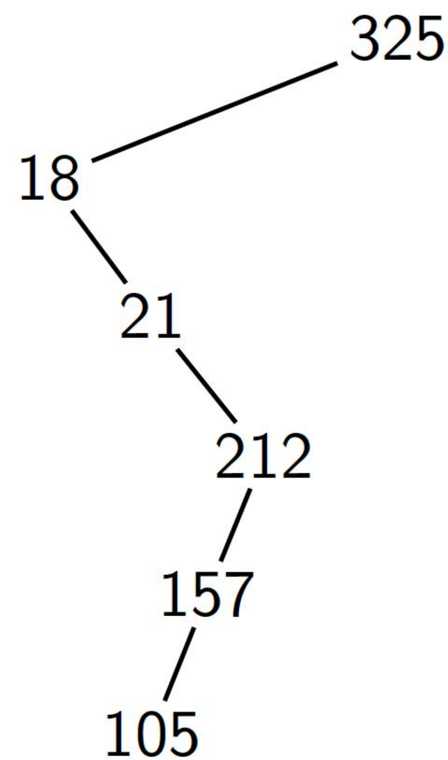
# Binary Search Trees

- If the BST is not well balanced, search performance degrades, and may be as bad as linear search:



# Binary Search Trees

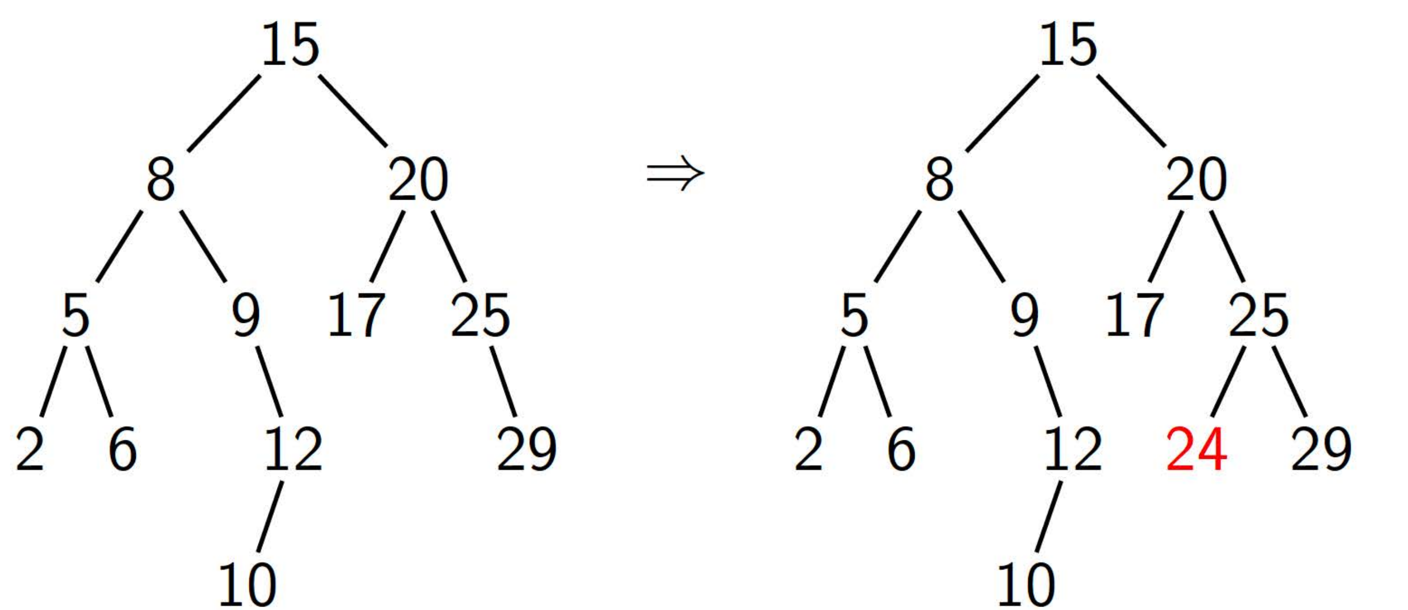
- If the BST is not well balanced, search performance degrades, and may be as bad as linear search:




- Is this a valid BST?
  – We'll come back to this at the end

# Insertion in Binary Search Trees

- To insert a new element $k$ into a BST, we pretend to search for $k$.
- When the search has taken us to the fringe of the BST (we find an empty sub-tree), we insert $k$ where we would expect to find it.
- For example, inserting 24:



# BST Traversal Quiz

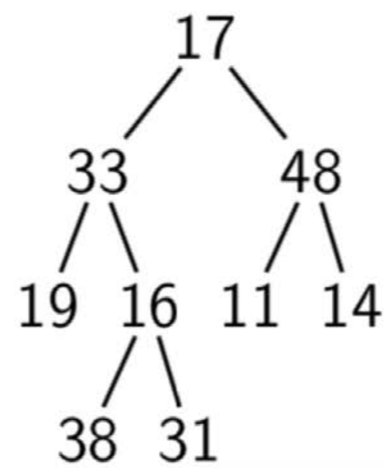
- Performing ……………...... traversal of a BST will produce its elements in sorted order.

# Review from Lecture 12

## Inorder Traversal

Visit order: 19 33 38 16 31 17 11 48 14

```
procedure InorderTraverse(T)
    if T ≠ null then
        InorderTraverse(T.left)
        visit T.root
        InorderTraverse(T.right)
```




We go as far down the left as possible, visit the left subtree, visit the root, then finally visit the right subtree.

Call Stack

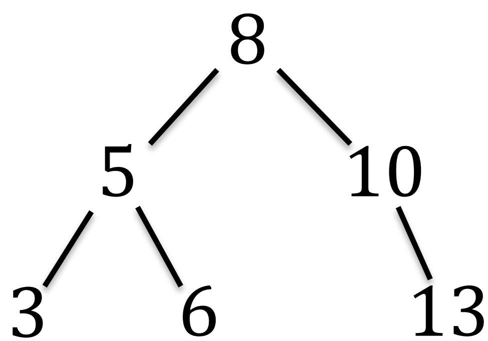
 

# BST Traversal Quiz

- Performing ……………...... traversal of a BST will produce its elements in sorted order.
- Example: build a BST by inserting $[8, 10, 5, 3, 13, 6]$ one at a time.

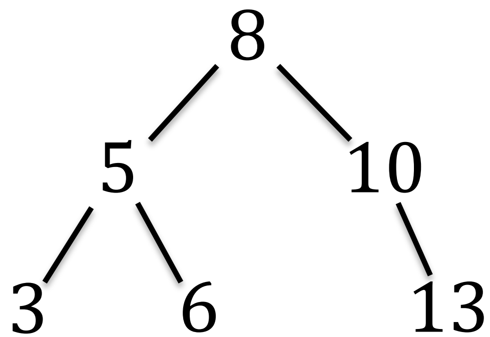
# BST Traversal Quiz

- Performing ………………... traversal of a BST will produce its elements in sorted order.
- Example: build a BST by inserting $[8, 10, 5, 3, 13, 6]$ one at a time.



# BST Traversal Quiz

- Performing ……………....... traversal of a BST will produce its elements in sorted order.
- Example: build a BST by inserting [8, 10, 5, 3, 13, 6] one at a time.
- Now look at an Inorder Traversal for this BST

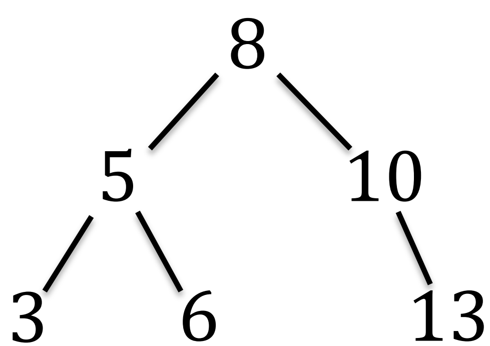


# BST Traversal Quiz

- Performing ………….......  traversal of a BST will produce its elements in sorted order.

- Example: build a BST by inserting [8, 10, 5, 3, 13, 6] one at a time.

- Now look at an Inorder Traversal for this BST

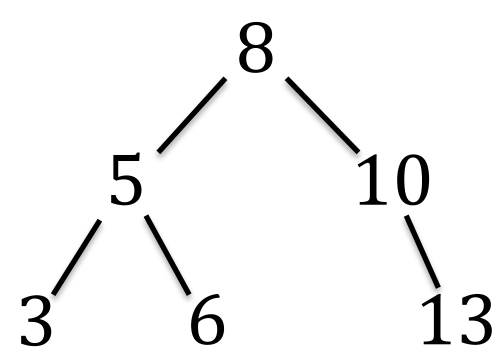



We go as far down the left as possible, visit the left subtree, visit the root, then finally visit the right subtree.

[3, 5, 6, 8, 10, 13]

# BST Traversal Quiz

- Performing **Inorder** traversal of a BST will produce its elements in sorted order.
- Example: build a BST by inserting [8, 10, 5, 3, 13, 6] one at a time.
- Now look at an Inorder Traversal for this BST




We go as far down the left as possible, visit the left subtree, visit the root, then finally visit the right subtree.

[3, 5, 6, 8, 10, 13]

# Binary Search Trees

- Try to build a BST by inserting $[325, 18, 21, 212, 157, 105]$ one at a time.

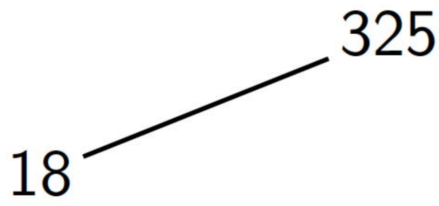
# Binary Search Trees

- Try to build a BST by inserting $[325, 18, 21, 212, 157, 105]$ one at a time.

325

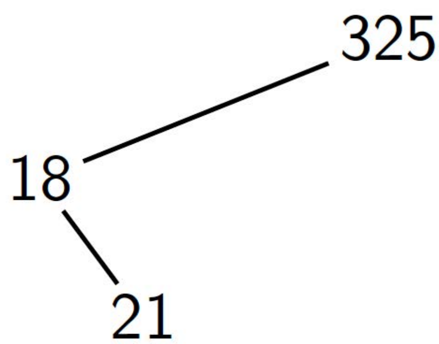
# Binary Search Trees

- Try to build a BST by inserting $[325, 18, 21, 212, 157, 105]$ one at a time.



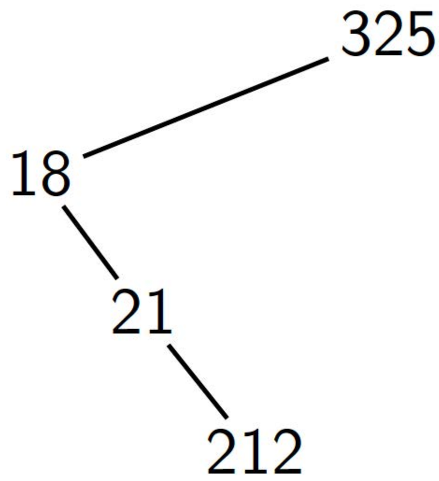
# Binary Search Trees

- Try to build a BST by inserting $[325, 18, 21, 212, 157, 105]$ one at a time.



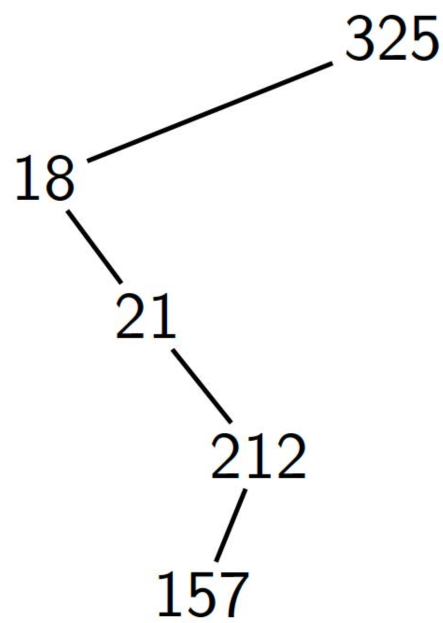
# Binary Search Trees

- Try to build a BST by inserting $[325, 18, 21, 212, 157, 105]$ one at a time.



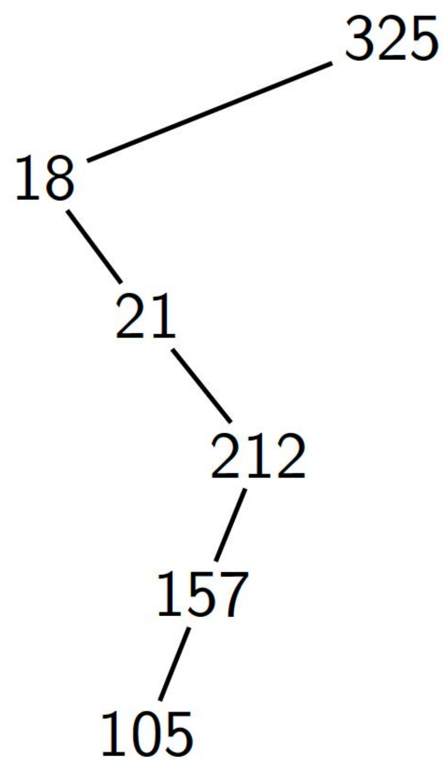
# Binary Search Trees

- Try to build a BST by inserting [325, 18, 21, 212, 157, 105] one at a time.



# Binary Search Trees

- Try to build a BST by inserting [325, 18, 21, 212, 157, 105] one at a time.



# Coming Up Next

- To optimise the performance of BST search, it is important to keep trees (reasonably) balanced.

- Next we will look at AVL trees and 2-3 trees (Levitin Section 6.3).