Augmenting data structures (BSTs)

We saw a clever way to find the i-th smallest element in a set in \( \Theta(n) \) time. With \( O(n\log n) \) pre-processing we can do this (dynamically) in \( O(\log n) \) time.
Augmenting Data Structures (BSTs)

We saw a clever way to find the i-th smallest element in a set in $\Theta(n)$-time. With $O(n \log n)$ pre-processing we can do this (dynamically) in $O(\log n)$ time. We can also ask for the rank of an element (dynamically) in $O(\log n)$ time.

Example:

RB-tree contains sorted letters

rank(M) = 6
rank(H) = 5
Augmenting Data Structures (BSTs)

We saw a clever way to find the $i$-th smallest element in a set in $\Theta(n)$ time. With $O(n \log n)$ pre-processing we can do this (dynamically) in $O(\log n)$ time.

We can also ask for the rank of an element (dynamically) in $O(\log n)$ time.

Example:

\[ \begin{array}{c}
A & 1 \\
D & 1 \\
H & 1 \\
F & 3 \\
N & 1 \\
P & 3 \\
M & 9 \\
\end{array} \]

$RB$-tree contains sorted letters augmented with subtree sizes.
Select$(x,i)$ \get i-th element in subtree rooted at $x$.

$k \leftarrow 1 + \text{size}(l_x)$ \ \leftarrow l_x : \text{left child of } x$.

if $i = k$, return $x$. 
Select(x,i) \quad \text{get i-th element in subtree rooted at x.}

k \leftarrow 1 + \text{size}(l_x) \quad \text{\(l_x\): left child of x}

if \(i = k\), return x.

example: i=5 \quad \text{Select(root,5)}

k=6 \quad k \leftarrow 1 + 5
```
Select(x, i) \quad \text{get i-th element in subtree rooted at } x

k \leftarrow 1 + \text{size}(l_x) \quad \text{\(l_x\): left child of } x

\text{if } i = k, \text{ return } x.
\text{else if } i < k, \text{ return } \text{Select}(l_x, i)
```

**example:** \(i = 5\)

\[ \text{Select(root, 5)} \]

\[ k = 6 \quad k \leftarrow 1 + 5 \]
Select\( (x, i) \) \hspace{1em} \text{get } i\text{-th element in subtree rooted at } x.

\[ k \leftarrow 1 + \text{size}(l_x) \] \hspace{1em} \text{\( l_x \): left child of } x.

if \( i = k \), return \( x \).

else if \( i < k \), return Select\( (l_x, i) \).

example: \( i = 5 \)

\[ k = 6 \]

\[
\begin{align*}
&k \leftarrow 1 + 5 \\
&i < k \Rightarrow \text{Select}(c, 5)
\end{align*}
\]
Select($x,i$) \get i-th element in subtree rooted at $x$. 

$k \leftarrow 1 + \text{size}(l_x)$ \leftarrow \text{name of left child of} \ x

if $i = k$, return $x$.

else if $i < k$, return Select($l_x,i$)

else ($i > k$) return Select($r_x,i-k$)

example: $i=5$

Select(root,5)

$k \leftarrow 1 + 5$

$i < k \Rightarrow \text{Select}(c,5)$
Select($x, i$) \hspace{1em} \text{get $i$-th element in subtree rooted at $x$.}

\begin{align*}
k &\leftarrow 1 + \text{size($l_x$)} \hspace{1em} \text{$l_x$: left child of $x$} \\
\text{if } i = k, & \text{ return } x. \\
\text{else if } i < k, & \text{ return } \text{Select($l_x, i$)} \\
\text{else } (i > k), & \text{ return } \text{Select($r_x, i-k$)}
\end{align*}

\text{example: } i = 5

Select(root, 5)

\begin{align*}
k &\leftarrow 1 + 5 \\
i < k \Rightarrow & \text{ Select($c, 5$)} \\
k &\leftarrow 1 + 1 \\
i > k \Rightarrow & \text{ Select($f, 3$)}
\end{align*}
Select($x, i$) // get $i$-th element in subtree rooted at $x$

$k \leftarrow 1 + \text{size}(l_x)$ // $l_x$: left child of $x$

if $i = k$, return $x$.
else if $i < k$, return Select($l_x, i$)
else ($i > k$) return Select($r_x, i-k$)

def Select(root, i):
    k = 1 + i
    if $i < k$:
        $c = \text{Select}(c, 5)$
        k = 1 + 1
        if $i > k$:
            $h = \text{Select}(H, 1)$

example: $i = 5$

$k = 6$

$i = 5, k = 2$

$i = 3, k = 2$
**Select**\(x, i\) \quad \ll \text{get } i\text{-th element in subtree rooted at } x.

\[ k \leftarrow 1 + \text{size}(l_x) \quad \ll \text{ } l_x : \text{left child of } x \]

- if \( i = k \), return \( x \).
- else if \( i < k \), return \( \text{Select}(l_x, i) \)
- else \( i > k \), return \( \text{Select}(r_x, i-k) \)

**Example:** \( i = 5 \)

\[ k \leftarrow 1 + 5 \]

\[ i < k \quad \Rightarrow \quad \text{Select}(c, 5) \]

\[ k = 1 + 1 \]

\[ i > k \quad \Rightarrow \quad \text{Select}(f, 3) \]

\[ k = 1 + 1 \]

\[ i > k \quad \Rightarrow \quad \text{Select}(h, 1) \]

\[ k = 1 + 0 \]

\[ i = k \quad \Rightarrow \quad \text{return } h \]
The balanced BST can be built in $\Theta(n \log n)$ time
The balanced BST can be built in $\Theta(n \log n)$ time

- Compute subtree sizes as you build
- Or
- Postorder walk after building
The balanced BST can be built in $\Theta(n\log n)$ time.

Compute subtree sizes as you build

or

postorder walk after building
The balanced BST can be built in $\Theta(n \log n)$ time

Compute subtree sizes as you build

or

postorder walk after building
The balanced BST can be built in $\Theta(n \log n)$ time

Compute subtree sizes as you build

or

postorder walk after building
The balanced BST can be built in $\Theta(n \log n)$ time

Compute subtree sizes as you build

or

postorder walk after building
The balanced BST can be built in $\Theta(n \log n)$ time.

Get rank: just as easy.

Walk up from node, adding sizes of subtrees representing smaller #'s.
The balanced BST can be built in $\Theta(n\log n)$ time.

Get rank: just as easy. Walk up from node, adding sizes of subtrees representing smaller #s.

ex: $\text{rank}(H) \rightarrow \text{size}(l_H) = 0$, walk up to F.
The balanced BST can be built in $\Theta(n \log n)$ time.

Get rank: just as easy.

Walk up from node, adding sizes of subtrees representing smaller #s.

ex: $\text{rank}(H) \rightarrow \text{size}(l_H) = 0$, walk up to F

H is right child of F, so count F.
The balanced BST can be built in $\Theta(n \log n)$ time

Get rank: just as easy.

Walk up from node, adding sizes of subtrees representing smaller #'s

ex: $\text{rank}(H) \rightarrow \text{size}(l_H) = 0$, walk up to $F$
$H$ is right child of $F$, so count $F$.
$\text{size}(l_F) = 1 \rightarrow \text{sum} = 1 + 1$
The balanced BST can be built in $\Theta(n \log n)$ time.

Get rank: just as easy.

Walk up from node, adding sizes of subtrees representing smaller #s.

ex: $\text{rank}(H) \rightarrow \text{size}(l_H) = 0$, walk up to F.
F is right child of F, so count F.
$\text{size}(l_F) = 1 \implies \text{sum} = 1 + 1$.
Walk up to C, count it.
The balanced BST can be built in $\Theta(n \log n)$ time.

Get rank: just as easy.
Walk up from node, adding sizes of subtrees representing smaller #s.

ex: \( \text{rank}(H) \rightarrow \text{size}(l_H) = 0 \), walk up to F
H is right child of F, so count F.
size(\(l_F\)) = 1 \rightarrow \text{sum} = 1 + 1 \rightarrow \text{walk up to } C, \text{ count it.}
size(\(l_C\)) = 1 \rightarrow \text{increment sum by } 1
The balanced BST can be built in $\Theta(n \log n)$ time.

Get rank: just as easy.

Walk up from node, adding sizes of subtrees representing smaller #’s.

ex: $\text{rank}(H) \rightarrow \text{size}(l_H) = 0$, walk up to F.

F is right child of F, so count F.

$\text{size}(l_F) = 1 \rightarrow \text{sum} = 1 + 1$.

Walk up to C, count it.

$\text{size}(l_C) = 1 \rightarrow \text{increment sum by 1}.$

Walk up to M, don’t count it.

TOTAL = 5 (4+1 for H)
The balanced BST can be built in $\Theta(n \log n)$ time

Get rank: just as easy.
Walk up from node, adding sizes of subtrees representing smaller #’s

ex: $\text{rank}(H) \rightarrow \text{size}(l_H) = 0$, walk up to F
H is right child of F, so count F.
$\text{size}(l_F) = 1$ … sum = 1 + 1
walk up to C, count it.
$\text{size}(l_C) = 1$ … increment sum by 1
walk up to M, don’t count it.

TOTAL = 5 \quad (4 + 1 \text{ for } H)$
The balanced BST can be built in $\Theta(n \log n)$ time.

Get rank: just as easy.
Walk up from node, adding sizes of subtrees representing smaller #s.

ex: $\text{rank}(H) \rightarrow \text{size}(l_H) = 0$, walk up to F.
H is right child of F, so count F.
$\text{size}(l_F) = 1$ ... sum = 1 + 1
walk up to C, count it.
$\text{size}(l_C) = 1$ ... increment sum by 1
walk up to M, don't count it.

$\text{TOTAL} = 5$ (4+1 for H)
What if we stored ranks instead of tree sizes?
What if we stored ranks instead of tree sizes?

\[ \implies \text{retrieve rank} = \Theta(1) \text{ time} \]
What if we stored ranks instead of tree sizes?

\( \rightarrow \) retrieve rank = \( \Theta(1) \) time
update = \( O(n) \) ... no good.
So we can search in $O(\log n)$ time, but what about insertion?

Easy to update tree sizes as we insert new node.

...but...
So we can search in $O(\log n)$ time, but what about insertion?

Easy to update tree sizes as we insert new node.
But we have to rebalance.
Can we update subtree sizes when inserting/deleting data?
Can we update subtree sizes when inserting/deleting data?

Use a RB tree

→ when are subtree sizes affected?
Can we update subtree sizes when inserting/deleting data?

Use a RB tree

→ when are subtree sizes affected? Rotations
So we can search in $O(\log n)$ time, but what about insertion?

Easy to update tree sizes as we insert new node.
But we have to rebalance.

In a RB tree, coloring doesn't affect tree sizes.

Rotations matter.

\[
\begin{align*}
\text{TOTAL: } & O(\log n) \\
\end{align*}
\]