CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

[1] based on notes by Anna Bretscher and Albert Lai

augmented data structures

An augmented data structure is simply an existing data structure modified to store additional information and / or perform additional operations.

Our task: a data structure that implements an ordered set/dictionary and, in addition to `insert`, `delete`, `search`, `union` (we'll see `union` shortly), etc., also supports two types of "rank queries":

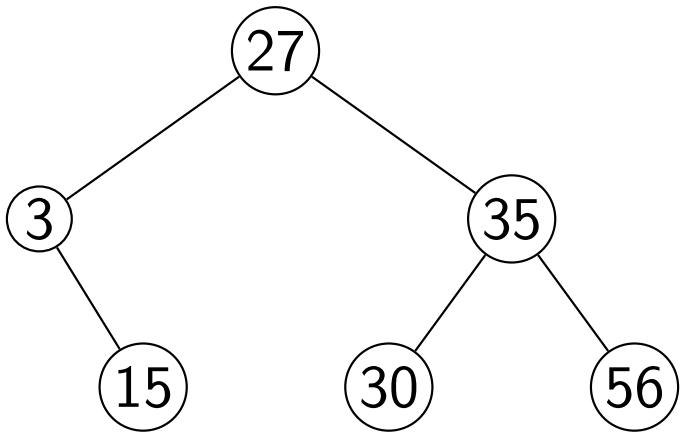
- `rank(S, k)`: given a key k in set S, what is its rank, i.e., the key's position among the elements?
- `select(S, r)`: given a rank r and set S, which key in S has this rank?

For example, in the set of values `S` = {3,15,27,30,35,56}:

- `rank(S, 15)` = 2
- `select(S, 4)` = 30

augmented data structures

For example, in the set of values `S` = $\{3, 15, 27, 30, 35, 56\}$:



- `rank(S, 15)` = 2
- `select(S, 4)` = 30

AVL tree without modification

If we use AVL tree without modifications:

- To implement `rank`: in-order traversal, stop when key found
- To implement `select`: in-order traversal, stop when rank found

  > count number of nodes visited

- What is the complexity of `rank`?

  $\Theta(n)$ — in-order

- What is the complexity of `select`?

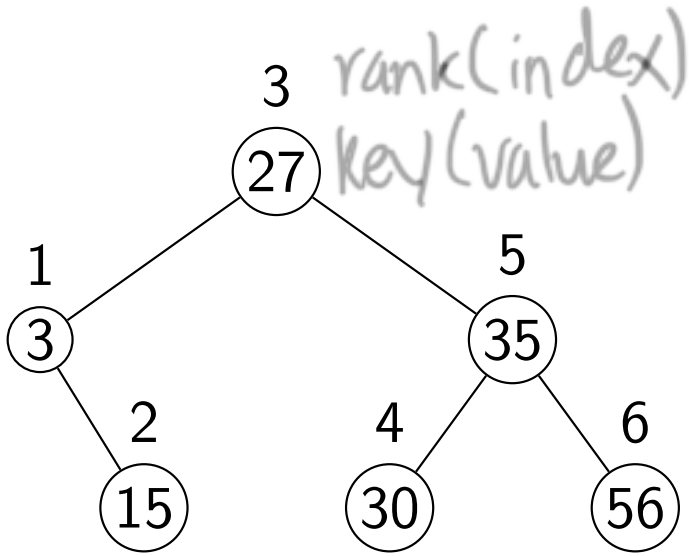
  $\Theta(n)$ — in-order

- Will operations `search`, `insert`, and `delete` need to change?

  No

augmented AVL tree — attempt 1

Idea: store $rank(T, n.key)$ in each node n in tree T.



- To implement `rank(T, k)`:
  - search for key(value)
  - return rank(index)
- To implement `select(T, r)`:
  - search for rank(index)
  - return key(value)

augmented AVL tree — attempt 1

Idea: store $rank(T, n.key)$ in each node n in tree T.

- What is the complexity of `rank(T, k)`?

$O(\log_2 n)$

- What is the complexity of `select(T, r)`?

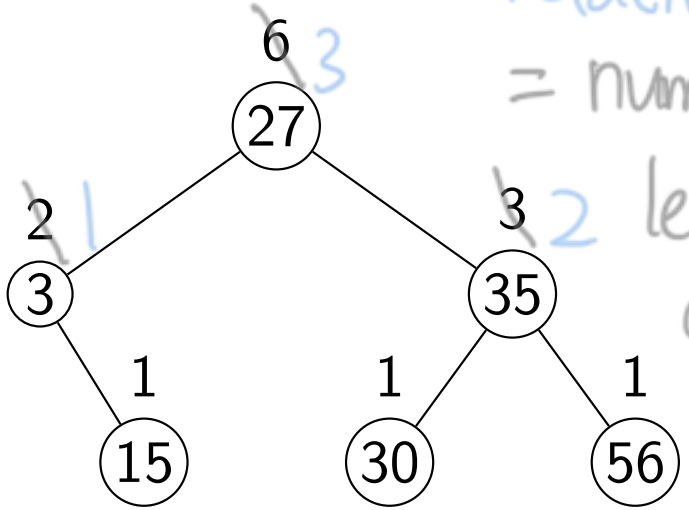
$O(\log_2 n)$

- Will operations `search`, `insert`, and `delete` need to change?

Yes, may need to change rank(index)

augmented AVL tree — attempt 2

Idea: store `size(n)` — the number of nodes in subtree rooted at n including n itself — for each node n.



relative rank(n, k):
= number of nodes to the left of tree rooted at k+1 (itself)
= size(n.left)+1

Q. How is size related to rank?

Define relative rank `rank(n, k)` as rank of key k relative to the keys in the tree rooted at node n.

augmented AVL tree — rank

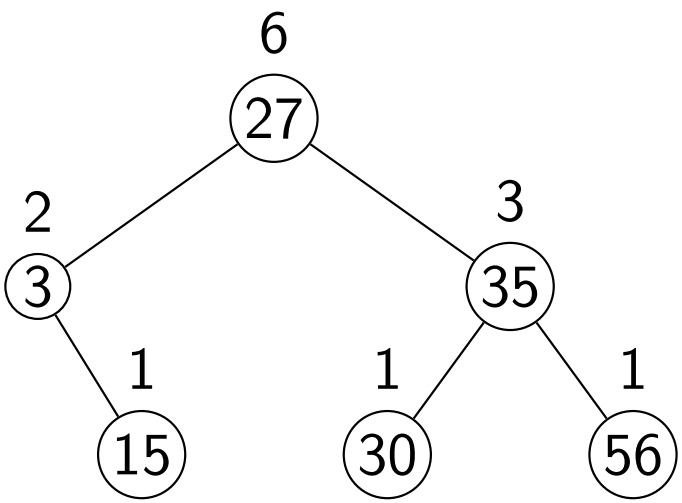
`rank(T, k)` — idea

T: tree rooted at root (27)

- do `search(T, k)` keeping track of the rank computed so far
- at each move to the right, add size of left subtree we skipped plus 1 for the key itself

do nothing for each move to the left

- if found key in node n, add $size(n.left) + 1$ to rank so far, to get the real rank



`rank(T, 35)`:

- 1+1+1+rank(35, 35)
-
- = 3+1+1=5
-

[subtree root 35]

35>27: move right

rank(T, 35) = size(27.left) + 1 + rank(35, 35)

found key 35:

rank(T, 35) = size(27.left) + 1 + size(35.left) + 1

= 2 + 1 + 1 + 1 = 5

15<27: move left

rank(T, 15) = rank(3, 15)

15>3: move right

rank(T, 15) = size(3.left) + 1 + rank(15, 15)

found key 15:

rank(T, 15) = size(3.left) + 1

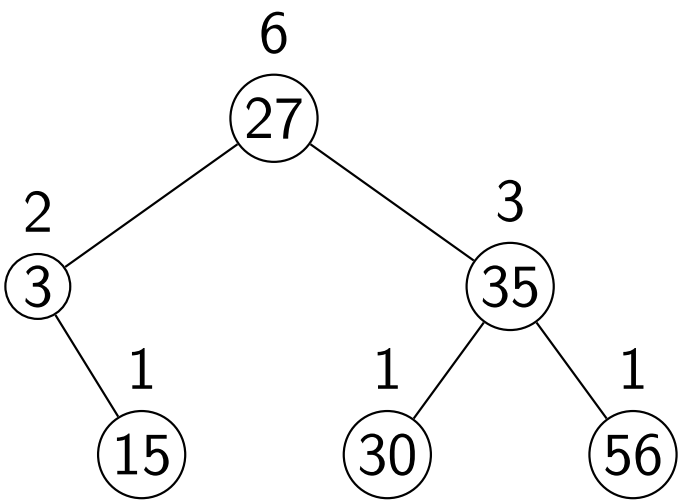
+ size(15.left) + 1

= 0 + 1 + 0 + 1 = 2

augmented AVL tree — rank

`rank(T, k)` — idea

- do `search(T, k)` keeping track of the rank computed so far
- at each move to the right, add size of left subtree we skipped plus 1 for the key itself
- if found key in node n, add $size(n.left) + 1$ to rank so far, to get the real rank



`rank(T, 15)`:

- $0 + 1 + rank(15, 15)$
-
- $= 0 + 1 + 0 + 1 = 2$

augmented AVL tree — rank

`rank(T, k)` — pseudocode

Recursion

```
if T == nil:   # k not in T
    deal with special case
if k == T.key:
    return size(T.left) + 1
if k > T.key:
    return size(T.left) + 1 + rank(T.right, k)
else:
    return rank(T.left, k)
```

where

```
size(T) = 0 if T == nil else T.size
```

augmented AVL tree — select

`select(T, r)` — idea

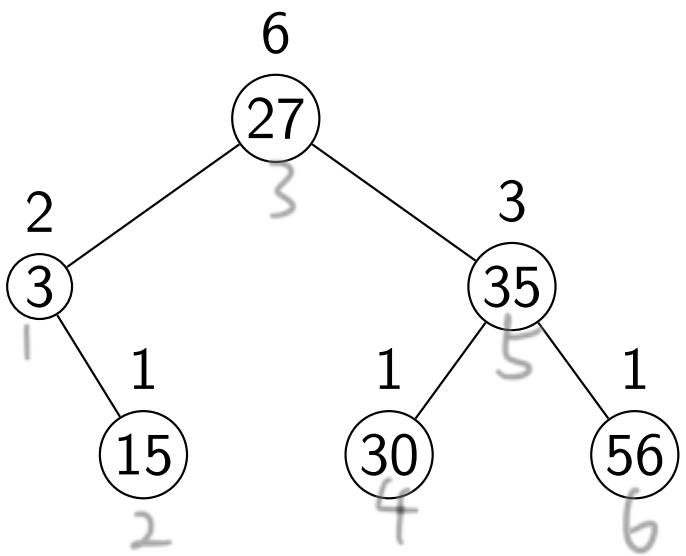
- at each visited node n, compare r to $size(n.left) + 1$
- if equal, found the node: return $n.key$ $r == size(n.left)+1$
- if $<$, then key with rank r is in left subtree
  - relative rank in left subtree is the same
  - look for rank r in $n.left$ $r < size(n.left)+1$
- if $>$, then key with rank r is in the right subtree
  - relative rank in the right subtree is $r - (size(n.left) + 1)$
  - look for rank $r - size(n.left) - 1$ in $n.right$

$r > size(n.right)+1$

augmented AVL tree — select

`select(T, r)` — idea

- at each visited node n, compare r to $size(n.left) + 1$
- ...



`select(T, 5)`:

- rank at 27: $r_{27} = 2 + 1 = 3$
- $5 > 3 \rightarrow$ move to 35:
- rank at 35: $r_{35} = 5 - r_{27} = 2$

$size(35.left) + 1 = 2 = r_{35}$

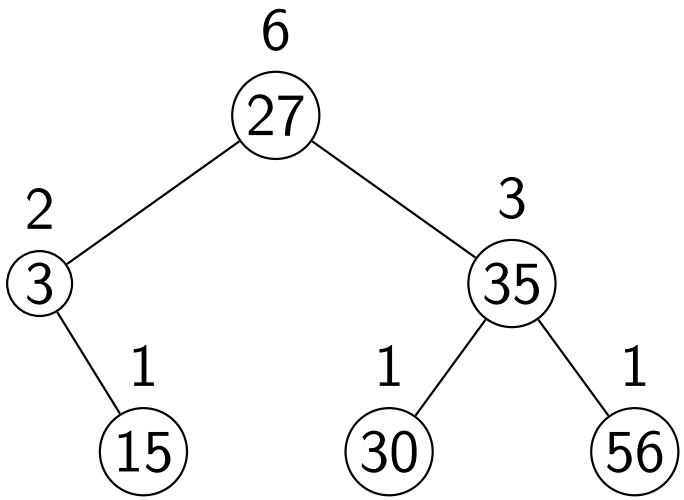
$\Rightarrow$ key is 35

size(27.left)

augmented AVL tree — select

`select(T, r)` — idea

- at each visited node n, compare r to $size(n.left) + 1$
- ...



`select(T, 2)`:

- rank at (27): $r_{27} = 2+1 = 3$
- $2 < 3 \rightarrow$ move to (3): size((3).left)
- rank at (3): $r_3 = 0+1 = 1$
- $2 > 1 \rightarrow$ move to (15):
- rank at (15): $r_{15} = 2 - r_3 = 1$
- size((15).left) $+1 = 1 = r_{15}$
- $\rightarrow$ key is 15

augmented AVL tree — select

`select(T, r)` — pseudocode

```
if T == nil:   # r not in T
    deal with special case
r' = size(T.left) + 1
if r == r':
    return T.key
if r < r':
    return select(T.left, r)
else:
    return select(T.right, r - r')
```

where

```
size(T) = 0 if T == nil else T.size
```

augmented AVL tree — insert / delete

- `insert(T, k, v)`:

  n.size = n.size + 1

- `delete(T, k)`:

  n.size = n.size − 1

- rebalancing: O(1)

  constant time to update

Therefore, each operation is $\Theta(\log n)$.