CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

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

augmented data structures

An <u>augmented</u> 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":

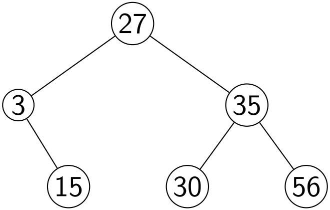
- `rank(S, k)`: given a key k in set S, what is its rank, i.e., the key's position among the elements?
- `select(S, r)`: given a rank r and set S, which key in S has this rank?

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

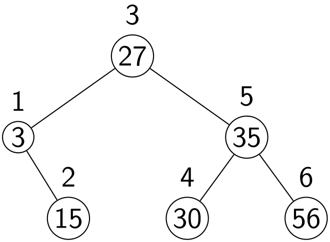
AVL tree without modification

If we use AVL tree without modifications:

- To implement `rank`:
  - in-order traversal, keeping track of the number of nodes visited, until the desired key is reached
- To implement `select`:
  - in-order traversal, keeping track of the number of nodes visited, until the desired rank is reached
- What is the complexity of `rank`? $\Theta(n)$
- What is the complexity of `select`? $\Theta(n)$
- 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 k
  - when found node n with $n.key = k$, return $n.rank$
- To implement `select(T, r)`:
  - search for rank r
  - when found node n with $n.rank = r$, return $n.key$

augmented AVL tree — attempt 1

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

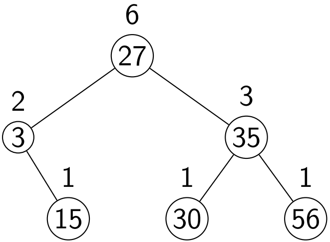
- What is the complexity of `rank(T, k)`? $\Theta(\log n)$

- What is the complexity of `select(T, r)`? $\Theta(\log n)$

- Will operations `search`, `insert`, and `delete` need to change? Yes!
  - `insert` and `delete` may need to update ranks of all other nodes — $\Theta(n)$

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*.



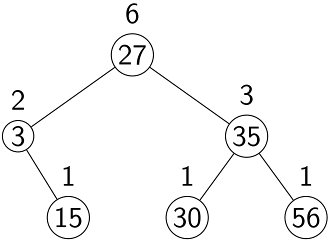
Q. How is size related to rank?
Define relative rank `rank(n, k)` as rank of key *k* relative to the keys in the tree rooted at node *n*.

`rank(T, k)` $= 1 +$ number of keys in T less than k
`rank(n, n.key) = 1 + size(n.left)`

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



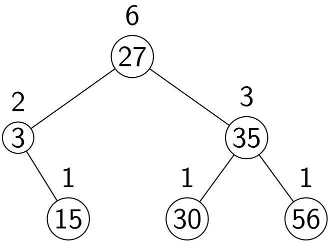
`rank(T, 35)`:

- `35 > 27`: go right
- rank is `size(T.left) + 1 + rank(T.right, 35)`
- `rank(35, 35)`:
  - `35 = 35`: found key
  - `size(35.left) + 1`
- `2 + 1 + 1 + 1 = 5`

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)`:

- `15 < 27`: go left
- rank is `rank(T.left, 15)`
  - `15 > 3`: go right
  - rank is `size(3.left) + 1 + rank(3.right,15)`
    - `rank(3.right,15) = 0 + 1 = 1`
- `0 + 1 + 1 = 2`

augmented AVL tree — rank

`rank(T, k)` — pseudocode

```
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

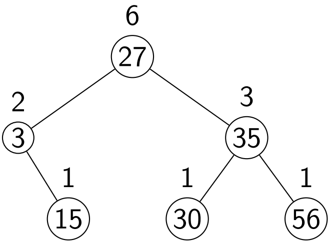
`select(T, r)` — idea

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

augmented AVL tree — select

`select(T, r)` — idea

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



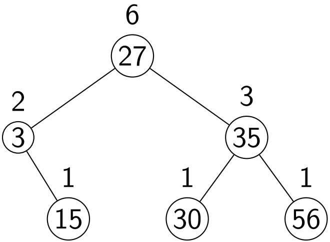
`select(T, 5)`:

- `size(T.left) + 1 = 2 + 1 = 3 < 5`: go right
- `select(T.right, 5 - 3)`
- `select(35, 2)`:
    - `size(35.left) + 1 = 2`
    - found node! key is 35

augmented AVL tree — select

`select(T, r)` — idea

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



`select(T, 2)`:

- `size(T.left) + 1 = 2 + 1 = 3 > 2`: go left
- `select(T.left, 2)`
- `select(3, 2)`:
  - `size(3.left)+1 = 1 < 2`: go right
  - `select(3.right, 2 - 1)`
  - `select(15, 1)`:
    - `size(15.left) + 1 = 1`
    - found node! 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)`:
  if `insert` successful, for each node n on path from parent of new node to root, $n.size = n.size + 1$

- `delete(T, k)`:
  after the node is removed (either x with $x.key = k$ or its successor), for each node n on path from parent of removed node to root, $n.size = n.size - 1$

- rebalancing:
  for each rotation, a constant number of nodes needs to be updated

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