Ranking and unranking
In sorted arrays

\[ \text{Rank}(x) = \text{the position of } x \text{ in the array} \]
(or the position it would go if added to the array)

Can be found by binary search

\[ \text{Unrank}(i) = \text{the element at position } i \text{ in the array} \]

Trivial to compute as Array[i]

For example, Unrank(n/2) is the median

They are inverse operations:

- \[ \text{Rank(Unrank}(i)) = i, \]
  if \( i \) is in the range of array indexes

- \[ \text{Unrank}(\text{Rank}(x)) = x, \]
  if \( x \) is one of the values stored in the array
In dynamic binary search trees

Rank and Unrank are well defined as the position of a given value in the sorted order, and the value at a given position.

But it’s not obvious how to compute them quickly! It doesn’t work to translate array search directly to trees.

- In array binary search for Rank($x$), we know the rank of each array cell.
- In binary search trees, we cannot store a rank in each tree node, because each update would cause all later ranks to change, too many for fast updating.
- There is no way to translate the trivial array Unrank algorithm into a tree algorithm.
Augmented binary search trees

Store **relative rank** in each node: its position among it and its descendants = number of left descendants

“x, y” means key value is x relative rank is y

Key 68 has 5 nodes in left subtree
Maintaining relative rank

On insertion or deletion: add or subtract one to all right ancestors

On rotation:

\[
\begin{align*}
  \text{rr}(x) & \text{ stays unchanged} \\
  \text{rr}(y) & \leftarrow \text{rr}(x) + 1
\end{align*}
\]

\[
\begin{align*}
  \text{rr}(y) & \rightarrow \text{rr}(x) + 1
\end{align*}
\]
Ranking using relative ranks

Call the following recursive search with node = tree root:

```python
def rank(x, node):
    if node == None:
        return 0
    else if x <= node.key:
        return rank(x, node.left)
    else:
        return rank(x, node.right) + node.relrank + 1
```

(In splay trees, add splay from last internal node on search path)
Unranking using relative ranks

Call the following recursive search with node = tree root:

```python
def unrank(i, node):
    if i == node.relrank:
        return node.value
    elif i < node.relrank:
        return unrank(i, node.left)
    else:
        return unrank(i - node.relrank - 1, node.right)
```

(In splay trees, add splay from last internal node on search path)
By adding extra information (relative rank) to each node of a binary search tree, we can still update the tree in $O(\log n)$ time, and answer rank and unrank queries in the same time.

Works with any rotation-based balanced binary search tree.

Related recent research: Ranking and unranking dynamic sorted sets of $n$ integers in the range $[0, n^c]$ can be done slightly faster: $O(\log n / \log \log n)$ per update or query.

Range searching
Range searching

Find aggregate information about data elements within a query range \([\text{low}, \text{high}]\) of values

(or within higher-dimensional regions)

- Range counting: Number of elements in range
  Compute ranks of left and right range endpoints and subtract

- Range reporting: List all elements in range

- Range minimum: Find minimum priority value in range
  (not minimum value – trivial as successor of left endpoint)

- Other more complex queries e.g. do a recursive range search on another attribute for elements within range
Call with node = tree root:

def report(low, high, node):
    if low < node.value:
        report(low, high, node.left)
    if low <= node.value <= high:
        output node.value
    if node.value < high:
        report(low, high, node.right)
Whenever we recurse into both children, we also output the node value

Every recursive call is one of:

- A node whose value is output
- A node on the search path for the low range endpoint (at which we search only the right child)
- A node on the search path for the high range endpoint (at which we search only the left child)

Time = \( O(\text{number of nodes searched}) = O(\text{output size} + \log n) \)

An algorithm whose time depends on output size and not just on input size is called “output sensitive”.
Decomposable range search problems

Suppose:
- We have a collection of key,value pairs with sorted keys
- An associative binary operation $\oplus$ operates on the values
- We want to find the result of applying $\oplus$ to the values whose keys are within a query range $[\text{low}, \text{high}]$

If we can decompose a range into disjoint sets, $S \cup T$, we can use $\oplus$ to combine results for each set: $\text{total} = \text{result}(S) \oplus \text{result}(T)$

Examples:
- Range counting, value $= 1$, $\oplus = \text{addition}$
- Range reporting, value($x$) = $\{x\}$, $\oplus = \text{set union}$
- Range minimum, value $= \text{priority}$, $\oplus = \text{minimization}$
Idea: search paths for range endpoints have length $O(\log n)$
We can decompose the range into $O(\log n)$ nodes on these two paths and $O(\log n)$ entire subtrees between them
Store $\oplus$ for each subtree, combine stored results for query total
Decomposable query algorithm

As we recurse, replace range endpoints by flag values $-\infty$ and $+\infty$ in subtrees for which endpoints are no longer relevant.

Whole tree is in range when both endpoints are infinite.

To query range $[low, high]$ at a given node:

- If $low = -\infty$ and $high = +\infty$, return stored value for subtree.
- If $key > high$, return query($low$, $high$, left child).
- If $key < low$, return query($low$, $high$, right child).
- Return query($low$, $+\infty$, left child) $\oplus$ node’s value $\oplus$ query($-\infty$, $high$, right child).

Time: $O(\log n)$ for operations with $\oplus$ time $O(1)$.
Maintaining the stored subtree values

Whenever a node’s stored subtree value might have changed
  ▶ We added or removed a descendant
  ▶ It was involved in a rotation
Recompute its subtree value as
left subtree value $\oplus$ right subtree value $\oplus$ node’s value

Time per insertion or deletion $O(\log n)$
(under same assumptions on $\oplus$ time as for query)

Works for any balanced binary search tree
Using augmented search trees, we can:

- Answer range counting or range minimization in time $O(\log n)$
- Answer range reporting in time $O(\log n + \text{output})$
- Handle insertions or deletions in time $O(\log n)$
- Generalize to other decomposable range searching problems