Today: Augmented Data Structures
- Idea
- Example: Dynamic Order Statistic
- Example: Interval Trees

Last Lecture: 2-3-4 Trees (\(+\) B-trees + Red-Black Tree)

- Every leaf at same depth from root.
- Every non-leaf has 2, 3, or 4 children.
- Every node (leaf or internal) has 1, 2, or 3 keys (1 less than # of children).

Could use 2-3-4 trees to perform insert, delete, search, find predecessor/successor in \(O(\log n)\) time.

Example:
Today’s Goal

- Use 2-3-4 trees to solve new problems.
- Answer more complex queries, by maintaining additional information at nodes.

"Augmenting" the data structure

Motivating Example: Dynamic Order Statistics

Goal: Maintain set $S$ while performing

- $\text{INSERT}(x, S)$: Inserts $x$ into $S$
- $\text{DELETE}(x, S)$: Deletes $x$ from $S$
- $\text{OS-SELECT}(i, S)$: Returns $i$th smallest element of $S$
- $\text{OS-RANK}(x, S)$: Returns rank of $x$ in $S$

Observations:

- 2-3-4 tree doesn’t perform $\text{SELECT/RANK}$.
- No obvious way to do so without maintaining extra information.
What information to maintain:

1st idea: Maintain rank of each element

Example:

```
  20
 / \  \
10   22
  \   \  
  15   25
    /   /  \
   11 24  27
  /  /  \   /  /  \
 5 12 17 4 13 14
```

- \text{RANK + SELECT}: \Theta(n)
- \text{INSERT + DELETE}: \Omega(n)

(Example: \text{INSERT}(2, 5)

or \text{DELETE}(1, 5))

Right idea: For each key, maintain its rank in its subtree
**Example:**

```
20
  |
  9
```


```
10 12
4 6
```

```
22 25 29
2 4 6
```

```
1 4 5
1 2 3
```

```
11
1
```

```
15 17
1 2
```

```
21
1
```

```
24
1
```

```
27
1
```

```
33
1
```

**SELECT** can be implemented easily.

**Example:** To **SELECT** $(11, s)$

```
20
  |
  9
```

we need to

```
11 - 9
```

```
10 12
4 6
```

```
22 25 29
2 4 6
```

```
1 4 5
1 2 3
```

```
11
1
```

```
15 17
1 2
```

```
21
1
```

```
24
1
```

```
27
1
```

```
33
1
```
Pseudo Code for `SELECT`:

```plaintext
SELECT (i, x)
  if x = NIL return NULL;
  for i = 1 to (# keys in x) do
    if rank_j[x] = i return keys_j[x];
    if rank_j[x] > i then
      return SELECT(i - rank_j[x], child_j[x]);
  endfor;
  return SELECT(i - rank_j[x], child_j[x]);
```

---

**Notes:**
- Pseudo code more complex than idea 😊
- Similar idea for `RANK(x, S)`;

---

**Insertion idea:**

To insert element into subtree rooted at x,

1. find right child c to insert into;
2. Increment `rank_c[x]` for `c' -> c`;
Local Picture

\[ \text{INSERT} (k) \text{ into } x \]

Suppose \( k_2 < k < k_3 \)

\[ \text{INSERT} (k) \text{ into the tree} \]

\[ \text{DONE? Not quite! Need to think about overflows.} \]

Suppose we have the following picture when fixing overflows.
Notes: Same methodology applies to
- Red-Black trees: (Need to handle rebalancing)
- B-trees etc.
Methodology

1. Choose data structure to modify (2-3-4-tras)

2. Decide additional information needed (rank in subtree)

3. Modify old operations to maintain this information (INSERT/DELETE)

4. Develop new operations (RANK, SELECT).

Example 2:

```
INTERVAL MAINTAINANCE:
```

- Data to be maintained = Set of "Intervals"
  - Interval = (low, high)
  - Operations: INSERT, DELETE, INTERVAL-SEARCH

Given \( k \), is there an interval \( (l[i], h[i]) \)

such that \( l[i] \leq k \leq h[i] \)?
**Query:** 1 ≤ S (9)

**Answer:** Either (5, 11) or (7, 10)

**Motivation:**

Interval = union of radio transmission medium and frequency range.

As users come in and drop off, might want to monitor how intense current usage is.

(More interesting query is # of intervals overlapping with given point; --- )
Applying Methodology

1. Pick the data structure:

   Say RED-BLACK tree; keyed on low point of intervals.

2. Decide what additional information to maintain:

   - Clearly need high point of intervals
   - But what else? Let's think.
   - Ideally would like to eliminate left subtree or right subtree at the current node
   - Right subtree eliminated if current low point is > search key.
   - How to eliminate left?
     - If max high in left subtree is < search key.
2b - Store max highpoint in current subtree

- Will this suffice? Yes!

Summary: Nodes have

\[ L = \text{low of interval} \quad \frac{(L, H)}{M} \quad H = \text{high point of interval} \quad M = \text{max in subtree} \]

3. Modifying old operations:

Example: INSERT

- Fix M's on the way down;
- Rotation - fixing - additional O(1) cost.

- SEARCH (k, x)

  \[
  \text{if } x = \text{NIL} \text{ then return FALSE;}
  \]
  \[
  \text{if } L(x) \leq k \leq H(x) \text{ then return } (L(x), H(x));
  \]
  \[
  \text{if } M(\text{left-child}(x)) < k
  \]
  \[
  \text{return SEARCH}(k, \text{right}(x));
  \]
  \[
  \text{else return SEARCH}(k, \text{left}(x));
  \]

Proof of Correctness:

\underline{Case 1: } M(\text{left-child}(x)) < k

Then clearly left subtree does not contain the correct interval so it is OK to reverse only on right.

\underline{Case 2: } L(x) > k \text{ and } M(\text{left}(x)) > k

By this condition right subtree can not contain the queried point so it is OK to reverse only on left.

\text{(NOT DONE YET THOUGH ....)}
Case 3: \( L(x) \leq k \leq M(\text{left}(x)) \)

- By this condition we reuse on left subtree.
- But \( \exists \) some node \( y \) in left tree with \( H(y) = M(\text{left}(x)) \) \((\text{by definition of } M(.))\)
- But \( L(y) \leq L(x) \leq k \leq M(\text{left}(x)) = H(y) \),
  so \( y \) is an interval that overlaps \( k \);
  so it is OK to reuse on left...