History:
1989: Seidel & Aragon
[Explosion of randomized algorithms]
Later discovered this was already known: Priority Search Trees from different context (geometry)
McCreight 1980

Randomized Data Structures
- Use a random number generator
- Running in expectation over all random choices
- Often simpler than deterministic

Intuition:
- Random insertion into BSTs $\Rightarrow O(\log n)$ expected height
- Worst case can be very bad $O(n)$ height
- Treap: A tree that behaves as if keys are inserted in random order

Example: Insert: $k, e, b, o, f, h, w$
(Std. BST) $\Rightarrow$

Obs: In a standard BST, keys are by inorder + insert times are in heap order (parent < child)

Geometric Interpretation:

Example:

Treap: Each node stores a key + a random priority. Keys are in inorder. Priorities are in heap order.

? Is it always possible to do both?
Yes: Just consider the corresponding BST
Insertion: As usual, find the leaf + create a new leaf node.
- Assign random priority
- On backing out - check heap order + rotate to fix.

Example:

Deletion: (cute solution) Find node to delete. Set its priority to $+\infty$.
Rotate it down to leaf level + unlink.

Theorem: A treap containing $n$ entries has height $O(\log n)$ in expectation (averaged over all assignments of random priorities)

Proof: Follows directly from BST analysis

Implementation: (See pdf notes)
Node: Stores priority + usual...
Helpers:

lowest priority ($p$) returns node of lowest priority among:

restructure: performs rotation $p$.left (if needed) to put lowest priority node at $p$.

Treaps II

Example:

Example:
Ideal Skip List:
- Organize list in levels
  - Level 0: Everything
    1: Every other
    2: Every fourth
    ... Every \(2^i\)
- Easy to code
- Easy to insert/delete
- Slow to search \(O(n)\)

Sorted linked lists:
- Idea: Add extra links to skip

Example:
Ideal skip list
- skip exactly \(2^i\) at level \(i\)

Node Structure:
```java
class SkipNode{
    Key key
    Value value
    SkipNode[] next
}
```

Value find(Key x) {
    i = topmost level
    SkipNode p = head
    while (i > 0) {
        if (p.next[i].key < x) p = p.next[i]
        else i -- (drop down a level)
    }
    // we are at base level
    if (p.key == x) return p.value
    else return null
}
```

Too rigid → Randomize! To determine level - toss a coin & count no. of consec. heads:
head \(3 \cdot \lg n\) tail

Skip Lists I
Delete:
- Start at top
- Search each level saving last node < key
- On reaching node at level 0, remove it and unlink from saved pointers

Insert:
- Start at top level
- At each level:
  - Advance to last node ≤ key
  - Save node + drop level
- At level 0:
  - Create new node (flip coins to determine height)
  - Link into each saved node

Example: find(75)

Insert (24):
- visit, don't save
- visit, save reference

Skip Lists III
Expected Search Time (Quick + Dirty)

**Claim:** You visit $O(1)$ nodes per level on average.

**Intuition:** What might go wrong?
- You visit lots of nodes at some level.

Why is this unlikely?

- Say you go through $k$ nodes at this level.
- If any tossed one more "H" you would not be here.
- Chances are $\geq 1 - 1/2^k$ ← Very likely if $k$ large.