Predecessor search data structures

Given n integers in $[0, U-1]$, store them in a data structure, supporting

- pred(x): return the largest number that is at most x.
- succ(x): return the smallest number that is at least x. (symmetric)
- ins(x): insert x to the set of integers ($x \in [0, U-1]$)
- del(x): delete x.

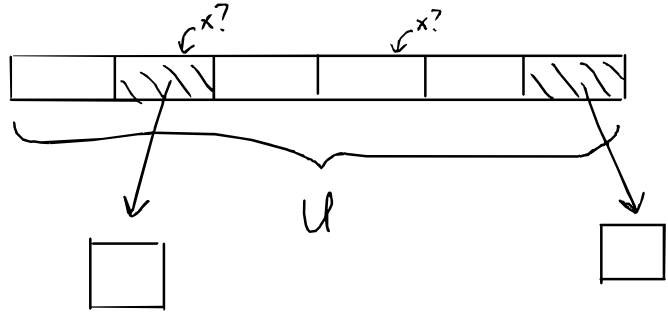
Easy sol: sorted array + binary search (no ins/del)
or BST $O(\log n)$ time.

Today: $O(\log \log U)$ time.

Application: sort n integers faster than $O(n \log n)$ (when U is subexponential in n)

- build the data structure by inserting n integers $O(n \log \log U)$
- find the max in $O(n)$ time
- keep querying the pred of max-1, and deleting the old max. $O(n \log \log U)$

Van Emde Boas tree (vEB tree)



divide $[0, u-1]$ into $\sqrt{u}$ blocks of size $\sqrt{u}$

store: min, max

$(i=0,\dots,\sqrt{u}-1)$ children[i]: a recursive vEB for numbers in block i (over $\sqrt{u}$)

aux: a recursive vEB tree for the $\sqrt{u}$ blocks (the inserted "numbers" are the nonempty blocks).

(min/max do not go into the recursion)

```
pred(x):
    if x ≥ max
        return max
    if x < min
        return NULL
    h = ⌊x/√u⌋, l = x mod √u
```

```
    if l ≥ children[h].min
        recursively find p = pred(l) in children[h]
        return h·√u + p
    recursively find i = pred(h) in aux
    return children[i].max
```

Time: only 1 recursive call, reduce u to $\sqrt{u}$.

$O(\log\log u)$ levels of recursion.

$O(\log\log u)$ time.

```
ins(x):
    if empty or min=max
        update min, max, return
    if x<min, swap x ⇔ min        // ins old min instead
    if x>max, swap x ⇔ max
    h = ⌊x/√u⌋ , l = x mod √u
    if children[h] empty
        ins(h) in aux
    ins(l) in children[h]
```

Time: never actually make two recursive calls

if "ins(h) in aux" is called,

then children[h] was empty, and

"ins(l) in children[h]" takes O(1) time.

same analysis as pred, O(loglog U) time.

del(x) is similar.

All operations in O(loglog U) time.

Space: O(U) (too large!)

---

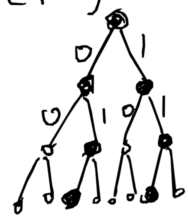
binary trie

write each integer as a log U-bit binary string

binary trie:

- complete binary tree with height = log U
- left child = bit 0, right child = bit 1
- each leaf = a log U-bit integer

- start with the empty trie (do not build the tree)



- ins(x): add the root-to-leaf path for x
  to the trie
- pred(x): find x's lowest ancestor in the tree,
  walk up until there's a left sibling, find max in the
  left sibling.

the data structure maintains the nodes with at least one leaf under it that is an inserted number.

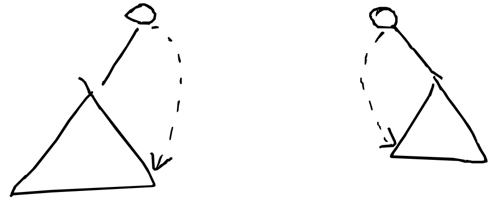
All operations in $O(\log U)$ time, $O(n \log U)$ space.

---

x-fast trie

modify binary trie:

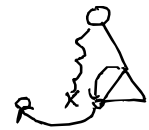
- any node with no right child stores a ptr to
  its max leaf in left child
- - - - - - - - left - - - - - - -
  - - min leaf - - right - - -
- leaves store ptrs to pred & succ
- each level stores a hash table of its
  nodes



pred(x):

- binary search to find the lowest ancestor p of x in the trie

- if p has no right child
  - return p.left.max
- if p has no left child
  - return p.right.min.pred.



Binary search takes $O(\log \log U)$ time.

Space: $O(n \log U)$

Each ins/del may need to update the ptrs in at most $O(\log U)$ ancestors, $O(\log U)$ time.