Abstract Data Type Based on Sets

operations–	UNION	$S_1 \cup S_2$
	Intersection	$S_1 \cap S_2$
	Difference	$S_1 - S_2$
	member	$x \in S$
	insert	
	delete	
	equal	$S_1 = S_2$?
	Find	where is x? (disj. sets)
	min,max	(in ordered sets)

Supporting all of above simultaneously is complicated and difficult if we want to compute efficiently. We will consider the follow subsets:

- $\cup, \cap$, $-$, insert, delete
- $\cup$, find
- *priority queue*: Insert, deletemin
- *Dictionary*: member, insert, delete

ADT: Dictionary (i.e. sets with *member, insert, delete* operations)

How do we implement a "dictionary"? We can use an array or a linked list, the elements can be ordered or unordered, as shown in the following table.

structure	list	list	array	array
order	No	Yes	No	Yes
Insert	$O(1)$	$S + O(1)$	$O(1)$	$S + O(n)$
delete	$S + O(1)$	$S + O(1)$	$S + O(1)$	$S + O(1)$
search	$O(n)$	$O(n)$	$O(n)$	$\log n + 0(1)$

In the above table, S is the search cost.

Searching a List

Worse case: n probes.

Average case: $\frac{n}{2}$ probes.

We assumed equal probability above for the average case. What if not all equally likely? Assume:

$$\text{Prob(search for } x_i) = p_i$$

So we have set $\{p_i\}$. Let $p_i \geq p_{i+1}$.

If we know the above probabilities, then we can order the elements in the list by decreasing probability. Then, the average search cost is

- Successful search: $\sum ip_i$.
- Unsucessful search: n

But what if we do not know probabilities?

- Solution: self-organizing list. Each time an element found, apply one of the following rules:
  - Move-to-front
  - Exchange

Claim. For any prob. distribution $\{p_i\}$ "Exchange" is at least as good as "Move-to-front" rule.

We omit the proof. We will only study Move-to-front" rule.

Theorem. Expected Cost of Move-to-Front (MtF) is at most twice the cost of optimal solution. Actually $\frac{\text{MtF}}{OPT} \leq \frac{\pi}{2}$

Proof. Cos t of optimal solution is $\sum ip_i$.

After many requests with MtF

$$Prob(x_i \text{ is ahead of } x_j) = \frac{p_i}{p_i + p_j}.$$

This holds for all future requests. (Consider last x_i or x_j query, use considtional probability.) Therefore the expected cost of x_j is

$$e_j = 1 + \sum_{i \neq j} \frac{p_i}{p_i + p_j}$$

So average cost over all x_j's is $\sum_j p_j e_j$, substitute e_j, we get

$$\sum_j p_j + \sum_{i \neq j} \frac{p_i p_j}{p_i + p_j} = 1 + 2 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{p_i p_j}{p_i + p_j}$$

To compute the ratio:

$$\begin{aligned}
\frac{\text{MtF}}{OPT} &= \frac{1 + 2\sum_{j=1}^{n} p_j \sum_{i=1}^{j-1} \frac{p_i}{p_i+p_j}}{1 + \sum_j p_j (j-1)} \\
&\leq (1+2x)/1+x \quad \text{where } x = \sum p_j (j-1) \\
&\leq 2\left(1 - \frac{1}{n+1}\right) \quad \text{as } x \leq \frac{1}{2}(n-1) \\
&\leq 2
\end{aligned}$$

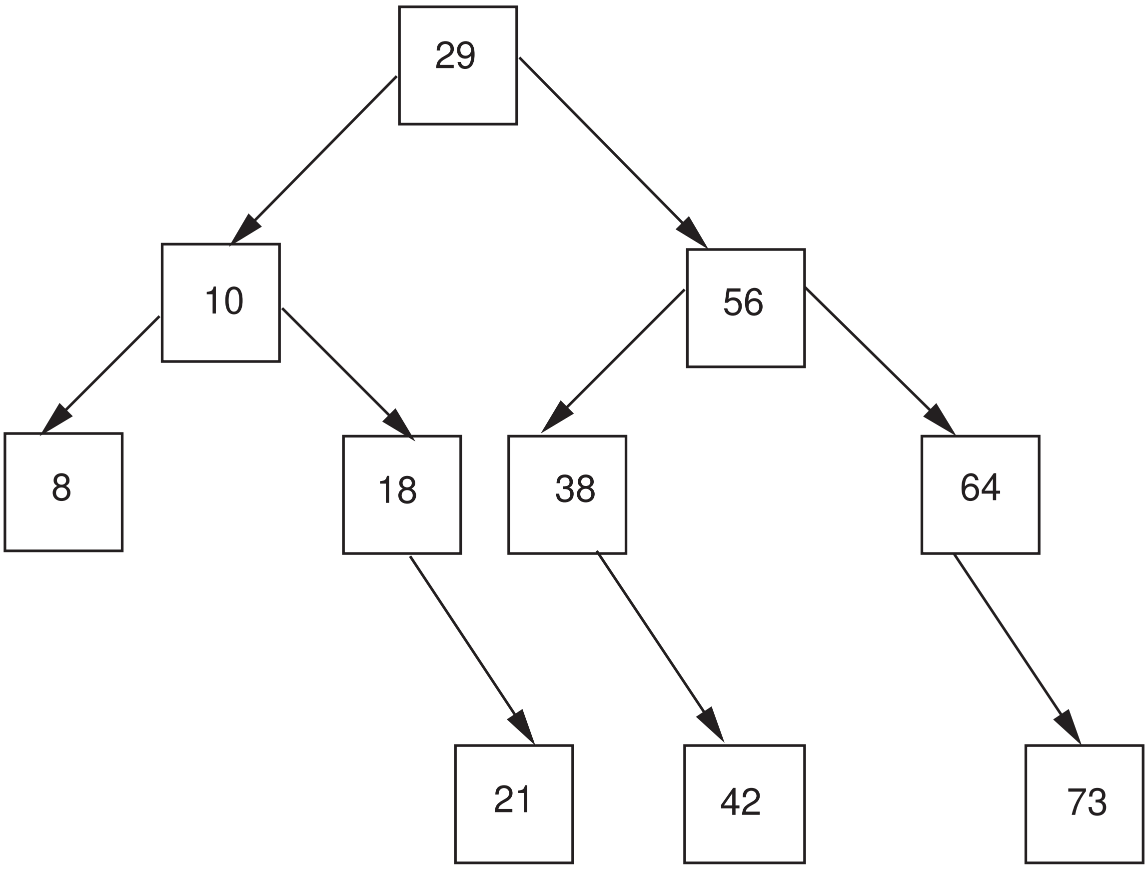
$\square$

If we use array in sorted order, then

- Insert, delete are hard: $O(n)$ steps.
- Search is fast: binary search costs $O(\log n)$.

Binary Search corresponds to Binary Search tree:

8	10	18	21	29	38	42	56	64	73



Can we do better than Binary Search ($\log n$)?

Yes. Interpolation Search: $\log \log n$ comparisons for uniform distribution.

Interpolation Search:

idea : rather than looking at the middle, look at where we expect the element to be. *small For example, to look for word "apple," you open the distionary at the* beginning.

In General: To find V, look at position

$$low + \frac{V - A[low]}{A[high] - A[low]} \times (high - low)$$

Then reduce range, and re-apply.

Example:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
5	10	12	14	26	31	38	39	42	46	49	51	54	59	72	79	82	86	92

Search for 50.

Initially: low = 1; high = 19

Interpolate to:

$$mid = 1 + \frac{50 - 5}{92 - 5} \times (19 - 1) = \underline{10.31}$$

look in $A[10]$, $50 > 46$, $low := 11$

$$mid = 11 + \frac{50 - 49}{92 - 49} \times (19 - 11) = \underline{11.18}$$

look in $A[11]$, $50 > 49$, $low := 12$

$$mid = 12 + \frac{50 - 51}{92 - 51} \times (19 - 12) = 12 - 0.17 = \underline{11.83}$$

$mid < low$, hence the element is not present.

Run time for Interpolation Search

- Worst Case: $O(n)$. worse than binary search!
- Average Case: $O(\log\log n)$ for uniform distribution

Intuitive idea: We will not give formal proof. But just provide an intuitive explanation. How far away is our interpolation from this element?

1st time: $\sim \sqrt{n}$	$n^{\frac{1}{2}}$
2nd time: $\sim \sqrt{\sqrt{n}}$	$n^{\frac{1}{2^2}}$
$\vdots$	$\vdots$
$\log\log n$-th time: 2	$n^{\frac{1}{2^{\log\log n}}}$

Comparison:

Interpolation Search	Binary Search
fewer probes on average	easier to code, guarantees $\log n$