Binary Search Trees and Skip Lists.
(CLRS 10, 12.1-12.3)

1 Maintaining ordered set dynamically

- We want to maintain an ordered set $S$ under operations
  - SEARCH($e$): Return (pointer to) element $e$ in $S$ (if $e \in S$)
  - INSERT($e$): Insert element $e$ in $S$
  - DELETE($e$): Delete element $e$ from $S$
  - SUCCESSOR($e$): Return (pointer to) minimal element in $S$ larger than $e$
  - PREDECESSOR($e$): Return (pointer to) maximal element in $S$ smaller than $e$

1.1 Ordered array implementation

- The first implementation that comes to mind is the ordered array:
  
  \[
  \begin{array}{c c c c c c c c c c}
  1 & 3 & 5 & 6 & 7 & 8 & 9 & 12 & 15 & 17 \\
  \end{array}
  \]

  - SEARCH can be performed in $O(n)$ time by scanning through array or in $O(\log n)$ time using binary search
  - PREDECESSOR/SUCCESSOR can be performed in $O(\log n)$ time like searching
  - INSERT/DELETE takes $O(n)$ time since we need to expand/compress the array after finding the position of $e$

1.2 Double linked list implementation

- Unordered list
  
  \[
  \begin{array}{c c c c c c c c c c}
  2 & 4 & 5 & 7 & 8 & 9 & 12 & 15 & 17 & 19 \\
  \end{array}
  \]

  - SEARCH takes $O(n)$ time since we have to scan the list
  - PREDECESSOR/SUCCESSOR takes $O(n)$ time
  - INSERT takes $O(1)$ time since we can just insert $e$ at beginning of list
  - DELETE takes $O(n)$ time since we have to perform a search before spending $O(1)$ time on deletion
• Ordered list

- Search takes $O(n)$ time since we cannot perform binary search
- Predecessor/Successor takes $O(n)$ time
- Insert/DELETE takes $O(n)$ time since we have to perform a search to locate the position of insertion/deletion

1.3 Binary search tree implementation

- Binary search naturally leads to definition of binary search tree

![Binary Search Tree Diagram]

• Formal definition of search tree:

  - Binary tree with elements in nodes
  - If node $v$ holds element $e$ then
    * All elements in left subtree $<$ $e$
    * All elements in right subtree $>$ $e$

- Search($e$) in $O(\text{height})$: Compare with $e$ and recursively search in left or right subtree
- Insert($e$) in $O(\text{height})$: Search for $e$ and insert at place where search path terminates (Note: height may increase)
Example: Insertion of 13

- $\text{Delete}(e)$ in $O(\text{height})$: Search for node $v$ containing $e$,
  1. $v$ is a leaf: Delete $v$
  2. $v$ is internal node with one child: Delete $v$ and attach $\text{child}(v)$ to $\text{parent}(v)$

Example: Delete 7

3. $v$ is internal node with two children:
   * exchange $e$ in $v$ with successor $e'$ in node $v'$ (minimal element in right subtree,
     found by following left branches as long as possible in right subtree)
   * $v'$ node can be deleted by case 1 or 2

Example: Delete 12

- Note:
Running time of all operations depend on height of tree.
Intuitively the tree will be nicely balanced if we do insertion and deletion randomly.
In worst case the height can be $O(n)$.

2 Skip lists

- There are several schemes for keeping search trees reasonably balanced and obtain $O(\log n)$ bounds
  - Often quite complicated—We will discuss one way (red-black trees) later.
- When we discussed Quick-sort we saw how randomization can lead to good expected running times.
  - We will now discuss how randomization can be used to obtain a very simple search structure with expected case performance $O(\log n)$ (independent of data/operations!)
- Idea in a skip list is best illustrated if we try to build a “search tree” on top of double linked list:
  - Insert elements $-\infty$ and $\infty$
  - Repeatedly construct double linked list (level $S_i$) on top of current list (level $S_{i-1}$) by choosing every second element (and link equal elements together)
  - Number of levels is $O(\log n)$

- $Search(e)$: Start at topmost left element. Repeatedly drop down one level and search forward until max element $\leq e$ is found.

Example: Search for 8
$O(\log n)$ time since we move at most one step to the right at each level.

- *Predecessor/Successor* also in $O(\log n)$ time
- *Insert/Delete* seems hard to do in better than $O(n)$ time since we might need to rebuild the entire structure after one of the operations.

- Idea in skip list is to let level $S_i$ consist of a randomly generated subset of elements at level $S_{i-1}$.
  - To decide if an element on level $S_{i-1}$ should be on level $S_i$, we flip a coin and include the element if it is head.
  - Expected size of $S_1$ is $\frac{n}{2}$
  - Expected size of $S_2$ is $\frac{n}{4}$
  - :
  - Expected size of $S_i$ is $\frac{n}{2^i}$
  - Expected height is $O(\log n)$

- Operations:
  - *Search(e)* as before.
  - *Delete(e)*: Search to find $e$ and delete all occurrences of $e$.
  - *Insert(e)*:
    * search to find position of $e$ in $S_0$
    * Insert $e$ in $S_0$.
    * Repeatedly flip a coin; insert $e$ and continue to next level if it comes up head.

- Running time of all the operations is bounded by search running time
  - Down search takes $O(\text{height}) = O(\log n)$ expected.
  - Right search/scan:
    * If we scan an element on level $i$ it cannot be on level $i + 1$ (because then we would have scanned it there)
    $\Downarrow$
Expected number of elements we scan on level $i$ is the expected number of times we have to flip a coin to get head

* We expect to scan 2 elements on level $i$

* Running time is $O(height) = O(log n)$ expected.

- Note:
  - We only really need forward and down pointers.
  - Expected space use is $\sum_{i=0}^{\log n} \frac{n}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = O(n)$. 