Preliminary solution proposal, DALG part

DALG assignment #1: (5p)

The simplest solution is to store the rank in an extra field of each skip list element. This is sufficient for an expected-logarithmic-time lookup implementation (and hence sufficient to get full points if done properly) but dooms insertion and deletion to linear time in the worst case. Instead, we describe here a solution that is based on rank differences rather than on absolute rank numbers. The implementation and analysis of lookup for this variant are very similar to the aforementioned absolute rank solution.

(a) We store $h$ counters (integers) in each skip list item of height $h$, one for each next pointer at level 0, 1, ..., $h-1$. Counter $c_j$ of an item indicates how many level-0 items lie between this item (including) and the successor item at level $j$ (excluding).

Whenever a next pointer is modified, the corresponding counter must be updated as well to maintain this invariant. This concerns insertion and deletion (not part of the question).

(b) $\text{LookupAtRank}$ looks only at the counters, not at the keys. We maintain a current rank variable $cr$. The artificial header element has rank -1, hence we initialize $cr$ to -1. When we search through the skip list starting at the highest level, as described in the lecture and the course books, we increment $cr$ by the $c_j$ associated with the $\text{next}[j]$ pointer whenever we follow that pointer. If we would exceed the desired rank $i$, we have to go “down” one step. As the desired rank $i$ is (assumed to be) in the range between 0 and $n-1$, the search will terminate successfully, namely if $cr = i$, which happens at level 0 at the latest, as a level-0 counter $c_0$ is always 1. Concretely,

```plaintext
p ← the artificial header item;
$cr ← -1$;
for $j$ from MAXLEVEL downto 0 do
    while $cr + p.cj ≤ i$
        $cr ← cr + p.cj$; // update current rank and
        $p ← p.next[j]$; // follow next pointer at level $j$
    // Invariant: $p$ points to item at rank $cr$
    if $cr = i$ return $p$; // eventually reached
```

Comment: Insertion and deletion were not part of this question; note that these may always be realized by a simple linear search in the level-0 list. However, with the above extensions, the expected logarithmic time complexity can also be achieved for $\text{insertAtRank}$ and $\text{deleteAtRank}$. 

1
In the case of deletion (not part of the question), the counters associated with modified next pointers in the flagged items are decremented by 1 and increased by the corresponding counter of the deleted element.

At insertion (not part of the question), the flagged items must be updated by decreasing each counter associated with a modified next pointer to account for the new distance to the inserted item. This is done by a backward pass over the flagged elements. Moreover, the inserted item must set all its counters to reflect the number of skipped level-0 elements to all its successors, which is computed as the difference of the old counter of a flagged element and its new value.

(c) The average case complexity follows immediately from the analysis of Lookup given in the lecture and the course books: The only difference is that we compare here the ranks, not the keys, but the navigation mechanism of searching and the structure of the skip list is the same.

**DALG assignment #2: (3p)**

(a) array representation: \([ 2, 3, 5, 8, 13, 21, 34, 55 ]\), tree representation
(b) after DeleteMin: \([ 3, 8, 5, 55, 13, 21, 34 ]\) plus tree representation
(c) after Insert(1): \([ 1, 2, 5, 3, 13, 21, 34, 55, 8 ]\) plus tree representation

**DALG assignment #3: (4p)**

(a) The hash table contains the entries \([\text{empty}, 21, 2, 12, 13, 41, 16, 26, 43, \text{empty}]\).
(b) Lookup(43) probes 6 times.
(c) *Linear / primary clustering:* Hashing with open addressing and linear probing tends to produce “clusters”, that are long contiguous sequences of filled hash table entries that behave like “attractors” to the insertion of new values and thus get even longer in the consequence. If two “attractors” merge, the resulting attractor has the accumulated “attraction” of the two predecessors, and the length of search chains grows accordingly. This effect is especially significant if the hash factor becomes relatively large (ca. 0.5 or greater). In part (a), this happened for instance when the two clusters \((21, 2, 12, 13)\) and \((16, 26)\) merged by insertion of 41 to a cluster of length 7.

A mathematical explanation is that, as the hash values returned by a good hash function can be assumed to be equally distributed across the \(n\) hash table positions, the probability of a hash value to lie in an index interval \(I\) of \(k\) filled positions is \(p(I) = k/n\), and the expected number of probes with a search starting at a randomly chosen index in \(I\) is \((k + 1)/(2n)\). The larger \(I\), the more likely a newly inserted element will do its first probe in \(I\) and hence be appended to \(I\) as well, increasing the length of \(I\) further. When two intervals \(I_1\) and \(I_2\) merge as the last free location between them gets filled, the corresponding probability of the resulting interval is \(p(I_1) + p(I_2) + 1\).

Long probe chains degrade the performance of lookup and insert operations dramatically. A long interval of length \(k = \Theta(n)\) causes an expected linear time for lookup and insert.

Possible improvements to avoid linear clustering, e.g.: 2
quadratic probing (but still with secondary clustering problem)
- double hashing
- increase the hash table size (rehashing) to keep the load factor low (improves on the average, not on the worst-case behavior).

**DALG assignment #4: (3p)**

We apply heap sort. As the first \( n - \sqrt{n} \) elements are already sorted, we can interpret these first \( n - \sqrt{n} \) table entries as a heap. Now we insert the last \( \lceil \sqrt{n} \rceil \) elements in the heap one by another, each of which takes time \( O(\log n) \) as the heap has \( \leq n \) elements in all cases. Hence, the total worst-case time complexity is \( \lceil \sqrt{n} \rceil \cdot O(\log n) \in O(n) \).

Another solution applies first an arbitrary \( O(n^2) \)-time sorting algorithm to sort the last \( \lceil \sqrt{n} \rceil \) elements, which thus runs in linear time in the worst case. Afterwards, a single merge step (see the mergesort algorithm) is applied to the two sorted subarrays, which takes time \( O(n - \sqrt{n} + \sqrt{n}) \), that is, \( O(n) \).

The advantage of the former solution is that it sorts in place, while the latter one needs an auxiliary array for the merging step.

**DALG assignment #5: (5p)**

(a) no. There are two unary nodes.

(b) no. Level 3 is not filled completely, neither is the last level filled properly from left to right.

(c) no. The root is out of balance (-2).

(d) yes. All nodes are in the tolerance range \( \{-1, 0, 1\} \).

(e) preorder: \( 52, 22, 7, 1, 15, 18, 47, 43, 56, 64, 67 \)

   inorder: \( 1, 7, 15, 18, 22, 43, 47, 52, 54, 56, 67 \)

   postorder: \( 1, 18, 15, 7, 43, 47, 22, 54, 67, 56, 62 \)

(f) After a single right rotation at the root node (see figure), the resulting tree has the AVL property.

```
    22
   /|
  7 | 52
 /|
1 15 47 56
 /|
18 |43 |54 |67
```

\( 22 \)