CSCI 406 Algorithms
Solution to Assignment 3

3-4 **[5 pts]**
Implement the dictionary as a Boolean array D of size n. If element i ($1 \leq i \leq n$) is present, set the corresponding array entry to TRUE, otherwise set it to FALSE.

- *Insert*(D, i): set $D[i]$ to TRUE in $O(1)$ time
- *Delete*(D, i): set $D[i]$ to FALSE in $O(1)$ time
- *Search*(D, i): if $D[i]$ is TRUE, return "element found", else return "element not found" in $O(1)$ time

3-6 **[10 pts]**

Add two pointer fields/members `pred` and `succ` to each node in the data structure. These will be used to point to predecessor and successor nodes, respectively. You can think of this as superimposing a doubly linked structure on top of the tree structure (each node will have one key field and four pointer fields). To find the predecessor or successor of a node, simply follow the appropriate pointer. The addition of these links will not impact other "passive" operations that do not modify the tree such as search, minimum, and maximum. However, insertion and deletion will need to make sure the `pred` and `succ` pointers are correctly set:

1. *Deletion*: Let x be the node to be deleted and let $x \rightarrow pred$ and $x \rightarrow succ$ point to its predecessor and successor respectively. These pointers should be copied to temporary variables and updated in case the element stored in x is replaced by its predecessor or successor (recall that this happens in the deletion case where x has both children). After the deletion is completed, x's successor's `pred` field and x's predecessor's `succ` field should be updated (similar to a deletion from a doubly linked list).
2. *Insertion*: After a node y is inserted, it's successor and predecessor should be identified in $\log n$ time using the original successor/predecessor algorithm described in class. Node y's predecessor's `succ` pointer and it's successor's `pred` pointer should be updated to point to y. Similarly, y's own `succ` and `pred` pointers should be updated.

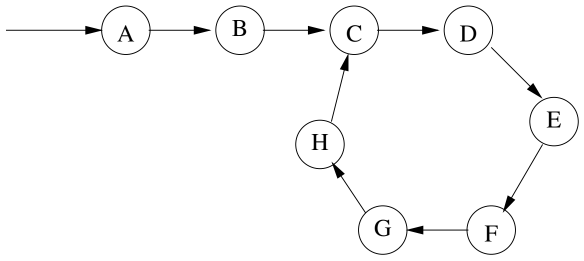
3-10 **[10 pts]**

Solution Outline: Use a balanced binary search tree to hold partially filled bins such that each node corresponds to a partially-filled bin. The key value of a node contains the *available weight* of the corresponding bin (from the problem statement, this value should be strictly between 0 and 1). For each of the n metal objects, search the tree for the bin with least amount of available weight which is still able to fit the object (best-fit policy). This is a simple modification of BST insertion and takes $O(\log n)$

time. If a bin exists, add the object to the bin and update the tree (you'll need to delete the bin and re-insert it with its new available weight). If no such bin exists, create a new bin to hold the object and add it to the tree. If, after a bin is updated with the new object, the weight is at 1 kg, the bin is full and can be removed from the tree. Finally, count the number of bins created to hold all of the objects (including bins that were deleted from the tree because they were maxed out).

3-27 **[10 pts]**

If there is a loop in a linked list, two different nodes' pointers must point to the same node. Those two pointers must be identical.



1. *Algorithm 1:*

   Consider the i^{th} node in the linked list (get to this by starting at the access pointer and following i pointers). Compare its `next` pointer to the `next` pointers of nodes 1 through $i-1$. Suppose the `next` pointers of node i and node k ($1 \le k \le i-1$) are equal and point to node j. Then we have found a loop starting at node j. In the figure, node j is node C, $i = 9$ (the second time C is encountered) and $k = 3$ (the first time C is encountered).

   Analysis (problem doesn't ask for this): When $i = 1$, we are comparing with 0 previous nodes; when $i = 2$, we are comparing with 1 previous node; when $i = 3$, we are comparing with 2 previous nodes; in general, when $i = m$, we are comparing with $m - 1$ previous nodes. Adding gives us a complexity of $O(n^2)$, where n is the size of the list (if no loops) or the location of the loop node (when encountered the second time).

2. *Algorithm 2:*

   Start with two pointers at the head of the list. The first pointer is the 'slow' pointer and the second pointer is the 'fast' pointer. The 'slow' pointer advances 1 node each iteration. The 'fast' pointer advances 2 nodes each iteration. If the 'fast' node moves onto or over the 'slow' pointer there is a loop.

   In the example, after three iterations, the slow pointer is at C, the fast pointer is at F. After three more iterations, both pointers are at F. The algorithm terminates within n iterations, so is $O(n)$.