This written homework assignment is due next Wednesday, February 14. Remember that you can discuss the homework with other people, but you should write up your own solutions.

1. A **closed** hash table with length 13 is used to store positive integers, using linear probing to resolve collisions. The hash function value for an integer N is simply the remainder $N \% 13$. Suppose that the following numbers are inserted into the table in the order shown, starting with an empty table:

```
131 32 818 295 88 204 55 321 419 492
```

a) Draw the closed hash table, showing the contents of the hash table after this sequence of insertions.

b) Now, explain carefully how to search the table to check if the number 321 is present. (What steps are followed, and why?) Do the same for the number 16.

2. Now, suppose that an **open** hash table is used instead. The size of the table is still 13, and the same hash function is used. Lists are implemented as simple linked lists, using the following class to represent the list nodes:

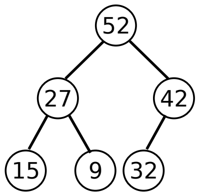
```
class ListNode {
    int item;
    ListNode next;
}
```

a) Draw the open hash table after the same sequence of insertions as in the previous problem.

b) The hash table is declared in Java as `ListNode[] table = new ListNode[13]`, and of course initially all the values in the array are *null*. Write a Java method that prints out all of the integers in the open hash table, using `System.out.println`.

c) Write a Java method for deleting a number N from the open hash table, if it is present. If N is not in the table, nothing should be done. (Use the hash function to find out where N should be!)

3. Next, we turn to **heaps**. For this problem, we consider a heap to be a full binary tree, and we ignore the possibility of storing it in an array. Suppose we start with the following heap of integers:



a) Redraw the heap to show its state after inserting **50**.

b) Now, redraw it again to show its state after inserting **18** (after the previous insertion).

(Continued)

- **c)** Now, redraw it again to show its state after inserting **35** (after the previous insertions).
- **d)** Now, redraw it again to show its state after inserting **87** (after the previous insertions).
- **e)** Finally, redraw the resulting heap to show the effect of deleting the maximum element, and explain the process that was used to delete that element. (What steps are followed, and why?)

4. (This is Exercise 3-3 from the textbook.) We have seen how dynamic arrays enable arrays to grow while still achieving constant time amortized performance. This problem concerns extending dynamic arrays to let them both grow and shrink as items are added and removed over time.

- **a)** Consider an "underflow" strategy that cuts the array size in half whenever the number of items falls below half of the length of the array. Give an example sequence of insertions and deletions where this strategy gives a bad amortized cost.
- **b)** Then give a better underflow strategy than the one suggested in part **a)**. Your strategy should be one that achieves constant amortized cost per deletion.