Q1

If no record with the key is present, then for an unordered table sequential search will always evaluate the entire table, or n entries. If the table is ordered, then the best case would be a single evaluation, the worst case would be n evaluations, and the average case would be $\frac{n}{2}$ evaluations. Here is example pseudocode to demonstrate this:

```
# Pseudocode for sequential search of unordered list
for Item in Table:
   if Item.key == key: # key found, return key
      return Item
return null            # entire table evaluated, key not found, return null


# Pseudocode for sequential search of ordered list
for Item in Table:
   if Item.key == key: # key found, return key
      return Item
   elif Item.key > key: # search moved past possible, key not in table, return null
      return null
return null             # entire table evaluated, key not found, return null
```

Q2

If one record with the key is present and only one record is sought, then the efficiency will not depend at all on whether the table is ordered or unordered. In both cases, the best case would be a single evaluation, the worst case would be n evaluations, and the average case would be $\frac{n}{2}$ evaluations.

Q3

If more than one record with the key is present and it is desired to find only the first, then the efficiency will not depend at all on whether the table is ordered or unordered. In both cases, the best case would be a single evaluation, the worst case would be $n - m$ evaluations, and the average case would be $\frac{n}{2}$ evaluations.

Q4

If more than one record with the key is present and it is desired to find them all, for an unordered list sequential search will perform a m comparisons in the best case, $n - m$ comparisons in the worst case, and $\frac{n}{2}$ comparisons in the average case.

If the list is sorted, then any record with the same key as the first found will be located immediately after the first record. Thus, the performance is the same; that is, the best case will require a single comparison, the worst case will require $n - m$ comparisons, and the average case will require $\frac{n}{2}$ comparisons.

Q5

This question is perhaps best answered with a demonstration. In this example we search for the two largest numbers in the set [1, 2, 3, 4, 100, 110, 120, 130].

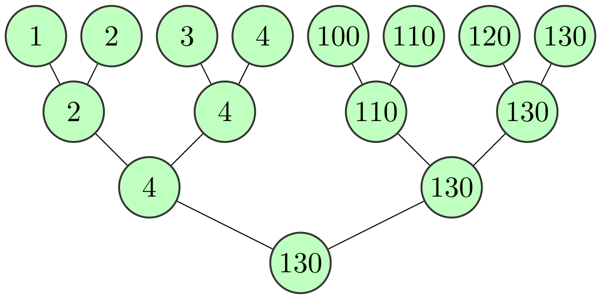


Figure 1: This is part one of the search, where each set of leaf nodes is compared to each other and the greater of the two "moves on". There are a total of $n - 1$ comparisons made.

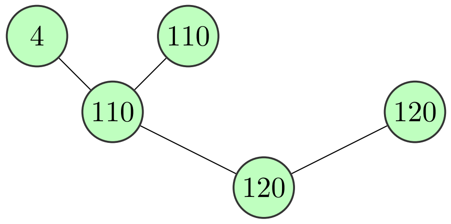


Figure 2: This is part two of the search, where the "loser" values at each level are compared in order to find the second largest value. There are a total of $(log_2 n) - 1$ comparisons made. The total number of comparisons then is $n + (log_2 n) - 2$.

Q6

This question builds off of the previous question. We saw in Question 5 that it takes $n - 1$ comparisons to find the largest value in a set. To find the smallest we do almost the same thing, except there are two differences. First, the initial (level 0) comparisons have already been made. Second, we are looking for the smaller value, and not the larger value in this case.

Thus, we can simply take the smaller values from the level 0 comparisons and perform an additional $\frac{n}{2} - 1$ comparisons to find the smallest value. The total number of comparisons then is $\frac{3n}{2} - 2$.

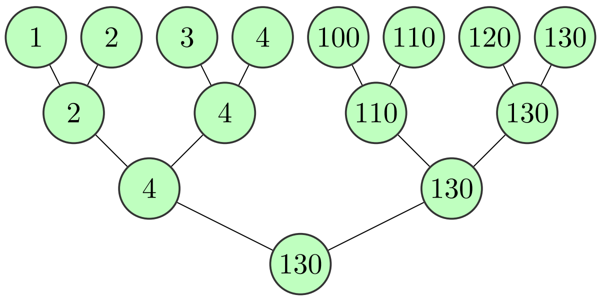


Figure 3: This is part one of the search, where we find the greastest value in the set. There are a total of $n - 1$ comparisons made.

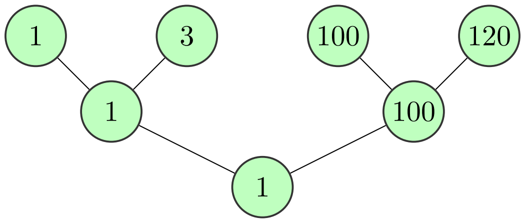
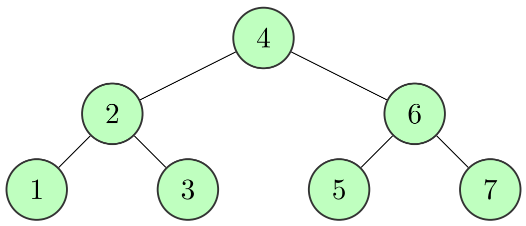


Figure 4: This is part two of the search, where we find the smallest value in the set. The initial level 0 comparisons were already made when finding the greatest value, so we only need to do $\frac{n}{2} - 1$ comparisons. The total number of comparisons then is $\frac{3n}{2} - 2$.

Q7

For a list with n records, there are $n!$ ways of ordering. However, achieving a straight-line (maximum height) binary tree is only possible in two cases - perfect ascending or perfect descending order. Thus, the probability of a max height binary tree is $\frac{2}{n!}$.

Consider the example using a set of numbers [1, 2, 3, 4, 5, 6, 7]. There are two possible ways to achieve a maximum height - inserting in order [1, 2, 3, 4, 5, 6, 7] or inserting in order [7, 6, 5, 4, 3, 2, 1]. However, if we look at the case of the perfectly balanced tree, we see there at least 56 ways of ordering.



For example, the following orders all produce the same tree:
[4, 2, 6, 1, 3, 5, 7]
[4, 2, 6, 1, 3, 7, 5]
[4, 2, 6, 1, 5, 3, 7]
[4, 2, 6, 1, 5, 7, 3]
[4, 2, 6, 1, 7, 3, 5]
[4, 2, 6, 1, 7, 5, 3]
.
.
.

From this, it is clear that balanced trees are more probable than straight-line trees.

Q8

```
# Pseudocode for delete(key1,key2) method
delete(key1,key2)
   key = key1
   while key <= key2:
      delete(key)
      key++

# Pseudocode for delete(key) method
delete(key)
   # find node
   for node in tree:
      if node.key == key:
         current = node

   # remove leaf node
   if current.left == null AND current.right == null:
      if current.parent.left == current:
         current.parent.left == null
      elif current.parent.right == current:
         current.parent.right == null
      else:
         tree.root = null

   # remove node with only left child
   if current.left != null AND current.right == null:
      if current.parent.left == current:
         current.parent.left == current.left
      elif current.parent.right == current:
```

```
      current.parent.right == current.left
    else:
      tree.root = null

  # remove node with only right child
  if current.left == null AND current.right != null:
    if current.parent.left == current:
      current.parent.left == current.right
    elif current.parent.right == current:
      current.parent.right == current.right
    else:
      tree.root = null

  # remove node with two children
  if current.left != null AND current.right != null:
    successor = current.right
    while successor.left != null:
      successor = successor.left
    current = successor
    remove(successor.key)
```

Q9

This is a somewhat difficult question as there are several possible cases that can be encountered when deleting from a B-tree that need to be addressed. We must traverse from the root node to the node with desired key so that we can merge nodes that have less than the minimum number of keys.

```
# Pseudocode for removal of a record from a B-Tree
delete(tree, key)
  current = tree.root
  while current != null:
    # preemptive merge
    if current.numberKeys == 1 AND current != tree.root
      Merge(current)

    if current.hasKey(key):
      # leaf node
      if current.isLeaf():
        RemoveKey(key)
        return true

      # internal node
      else
        copy(successor,key)
        delete(successor)
        return true

    current = Next(current, key)

  return false
```

Q10

The first item inserted to the table will have no chance of collision, as all spaces are free. After this however, the chance of collision is represented by the number of items in the table divided by the table size. So for the n$^{\text{th}}$ key, the collision probability will be $(n-1)/tablesize$. If we sum the collision probabilities of each successive addition, we find the following:

$$\begin{aligned}\sum_{0}^{n-1} \frac{n}{tablesize} &= \frac{1}{tablesize} + \frac{2}{tablesize} + ... + \frac{n-1}{tablesize} \\ &= \frac{1}{tablesize}\left(\frac{n(n-1)}{2}\right) \\ &= \frac{n}{tablesize}\left(\frac{n-1}{2}\right) \\ &= (n-1)\frac{lf}{2}\end{aligned}$$

Q11

The maximum number of comparisons done is equal to $1 + tablesize$. If we divide this by the number of open spots plus one, we will have the average number of comparisons needed to insert a new element. That is, if no spots are open, then the maximum number of comparisons occurs and if all spots are open then only a single comparison occurs. The number of open spots is given by $n_{open} = tablesize - n$. Thus, the final equation becomes:

$$\frac{tablesize + 1}{tablesize - n + 1}$$

Linear probing does not satisfy this condition because it simply inserts into the next available space if a bucket is occupied. It would need random selection to satisfy the above.