Chaining

		41			18-44-31	32	59	73	22			
0	1	2	3	4	5	6	7	8	9	10	11	12

hash(key)=key mod 13

key = 18, 41, 22, 44, 59, 32, 31, 73
H = 5, 2, 9, 5, 7, 6, 5, 8
44=2,31=3
Average= 11/8=1.375

Linear Probing

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

hash(key)=key mod 13

key = 18, 41, 22, 44, 59, 32, 31, 73
H = 5, 2, 9, 5, 7, 6, 5, 8
18=1,41=1,22=1,44=2,59=1,32=3,31=6,73=4
19/8=2.375

Double Hashing

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

hash(key)=key mod 13

key = 18, 41, 22, 44, 59, 32, 31, 73
H(k) = 5, 2, 9, 5, 7, 6, 5, 8
d(k)=7 – (k mod 7)
d(k) = 3, 1, 6, 5, 4, 3, 4, 4 (secondary hash)
Probes = 18=1,41=1,22=1,44=2,59=1,32=1,31=3,73=1
Average= 11/8=1.375

1/(1 – LF)
LF=1/2
1/(1 – (1/2)) = 2
1/(1-(3/4))=4
1/(1-(4/5))=5
1/(1- .9)=10

When we re-size if load factor is .75, then the load factor becomes .5 because
(3N/4)/(3/2)=(2*3*N)/(3*4)=N/2

HashTable issues

Client/user of Dictionary has to do the following

1. Implement the hashcode function that maps a key into an integer (and implement the equals method in Java)

Implementer of a Hash table based Dictionary for a class library has to do the following:

2. Implement compression function (picking size of Array in the Hash table (prime number)) use MOD or MAD to compress the hashcode into an index into the array/table.
3. re-sizing (load factor > .75) LF=n/N, re-size by newN = ceiling(N*1.5)
4. handle collisions (chaining or probing (linear, double hashing))

Mainframe IBM file structures: BDAM ISAM SAM

D <- new Dictionary(HT)

D <- new Dictionary(BST)

Algorithm BinarySearch(*S, k, low, high*):

Input: *An ordered vector S storing n items, accessed by keys()*

Output: *An element of S with key k and rank between low & high.*

```
if low > high then
    return NO_SUCH_KEY
else
   mid <- (low + high)/2
   if k = key(mid) then
      return elem(mid)
   else if k < key(mid) then
      return BinarySearch(S, k, low, mid-1)   // T(n/2)
   else
      return BinarySearch(S, k, mid + 1, high)  // T(n/2)
```

T(n)= a T(n/b) + f(n)

1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
   provided $af(n/b)\le \delta f(n)$ for some $\delta<1$.

T(n)= T(n/2) + c

$\log_2 1 = 0$

Case 2: Is f(n) = c $\Theta(n^0 \log^k n)$? Yes, since $\Theta(n^0 \log^0 n) = \Theta(1)$ for k=0

Therefore, T(n) = $\Theta(n^0 \log^{k+1} n) = \Theta(\log n)$

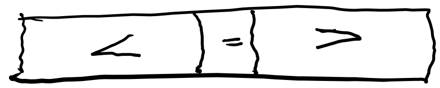
$\log_{10} n = (\log_2 10) * \log_2 n$ Note that $(\log_2 10)=3.321$ is a constant.

Similarly, $\log_2 n = (\log_{10} 2) * \log_{10} n$ Similarly, $(\log_{10} 2)=0.301$ is a constant.

Therefore, $O(\log_{10} n)$ is $O(\log_2 n)$ is $O(\log_b n)$ for any b>0 since $\log_a n = (\log_b a) * \log_b n$

In my past experience and algorithm text books, the shorthand for base 2 was O(lg n) and base 10 logarithms was O(log n), so I needed to justify that it was not necessary to specify the base. However, it's easy to understand why using the above, i.e., the base only causes the value to differ by a constant.

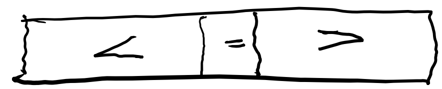
Corrects the problem of partitioning when many elements are equal. Partitions into three segments between lo and hi.




```
Algorithm inPlacePartition(S, lo, hi)
// the segment being partitioned is the elements
// between ranks lo and hi
r <- randomly chose a rank between lo and hi
pivot <- S.elemAtRank(r)
nextLess <- S.atRank(lo)
currPos <- S.atRank(lo)
last <- S.atRank(hi)
while currPos != last do   // place all elements less than the pivot at the front of segment
    if currPos.element() < pivot then
        S.swapElements(nextLess, currPos)
        nextLess <- S.after(nextLess)
    else
        currPos <- S.after(currPos)
if currPos.element() < pivot then  // need to handle the last element of List
    S.swapElements(nextLess, p)
    nextLess <- S.after(nextLess)

currPos <- nextLess
nextGreater <- S.atRank(hi)
while currPos != nextGreater do  // place all elements greater than the pivot at the end of segment
    if currPos.element() > pivot then
        S.swapElements(nextGreater, p)
        nextGreater <- S.before(nextGreater)
    else
        currPos <- S.after(currPos)
if currPos.element() > pivot then  // handle the last element
    nextGreater <- S.before(nextGreater)

return (S.rankOf(nextLess), S.rankOf(nextGreater))
```

```
Algorithm quickSort(S)
        inPlaceQuickSort(S, 0, S.size() - 1)

Algorithm inPlaceQuickSort(S, lo, hi)
        If lo ≤ hi then
                (p1, p2) <-  inPlacePartition(S, lo, hi)
                inPlaceQuickSort(S, lo, p1 - 1)
                inPlaceQuickSort(S, p2 + 1, hi)
```

This partition algorithm runs in O(n) time and is an example of the partitioning of a list around a pivot element. Note that this algorithm traverses the list using only the first, last, before, and after operations of the List ADT. Also, when working with Positions, we must not allow a Position to go off the end of the List nor can we allow two Positions to go past each other if the loop terminates when two Positions are equal as in the two loops below as well as above!



Algorithm inPlaceListPartition(L, pivot)

Input: L is a List and pivot is the value of the pivot.

Output: the elements in L are partitioned around pivot.

```
afterLess <- L.first()
currPos <- afterLess
last <- L.last()
while currPos != last do   // place all elements less than the pivot at the front of L
    if currPos.element() < pivot then
        L.swapElements(afterLess, currPos)
        afterLess <- L.after(afterLess)
    else
        currPos <- L.after(currPos)
if currPos.element() < pivot then  // need to handle the last element of List
    L.swapElements(afterLess, currPos)
    afterLess <- L.after(afterLess)

currPos <- afterLess
beforeGreater <- last
while currPos != beforeGreater do  // place all elements greater than the pivot at the end
    if currPos.element() > pivot then
        L.swapElements(currPos, beforeGreater)
        beforeGreater <- L.before(beforeGreater)
    else
        currPos <- L.after(currPos)
if currPos.element() > pivot then  // handle the last element
    beforeGreater <- L.before(beforeGreater)

return (afterLess, beforeGreater)  // positions of the first and last element that equal the pivot
```