CHAPTER

12

SELECTION ALGORITHMS [MEDIANS]

12.1 What are Selection Algorithms?

Selection algorithm is an algorithm for finding the k^{th} smallest/largest number in a list (also called as k^{th} order statistic). This includes finding the minimum, maximum, and median elements. For findingthe k^{th} order statistic, there are multiple solutions which provide different complexities, and in this chapter we will enumerate those possibilities.

12.2 Selection by Sorting

A selection problem can be converted to a sorting problem. In this method, we first sort the input elements and then get the desired element. It is efficient if we want to perform many selections.

For example, let us say we want to get the minimum element. After sorting the input elements we can simply return the first element (assuming the array is sorted in ascending order). Now, if we want to find the second smallest element, we can simply return the second element from the sorted list.

That means, for the second smallest element we are not performing the sorting again. The same is also the case with subsequent queries. Even if we want to get k^{th} smallest element, just one scan of the sorted list is enough to find the element (or we can return the k^{th}-indexed value if the elements are in the array).

From the above discussion what we can say is, with the initial sorting we can answer any query in one scan, O(n). In general, this method requires O($nlogn$) time (for *sorting*), where n is the length of the input list. Suppose we are performing n queries, then the average cost per operation is just $\frac{n\,logn}{n} \approx O(logn)$. This kind of analysis is called *amortized* analysis.

12.3 Partition-based Selection Algorithm

For the algorithm check Problem-6. This algorithm is similar to Quick sort.

12.4 Linear Selection Algorithm - Median of Medians Algorithm

Worst-case performance	O(n)
Best-case performance	O(n)
Worst-case space complexity	O(1) auxiliary

Refer to Problem-11.

12.5 Finding the K Smallest Elements in Sorted Order

For the algorithm check Problem-6. This algorithm is similar to Quick sort.

12.6 Selection Algorithms: Problems & Solutions

Problem-1 Find the largest element in an array A of size n.

Solution: Scan the complete array and return the largest element.

```
def FindLargestInArray(A):
    max = 0
    for number in A:
        if number > max:
            max = number
    return max

print(FindLargestInArray([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13]))
```

Time Complexity - $O(n)$. Space Complexity - $O(1)$.

Note: Any deterministic algorithm that can find the largest of n keys by comparison of keys takes at least $n-1$ comparisons.

Problem-2 Find the smallest and largest elements in an array A of size n.

Solution:

```
def FindSmallestAndLargestInArray(A):
    max = 0
    min = 0
    for number in A:
            if number > max:
                    max = number
            elif number < min:
                    min = number
    print("Smallest: %d", min)
    print("Largest: %d", max )

FindSmallestAndLargestInArray([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13])
```

Time Complexity - $O(n)$. Space Complexity - $O(1)$. The worst-case number of comparisons is $2(n-1)$.

Problem-3 Can we improve the previous algorithms?

Solution: **Yes.** We can do this by comparing in pairs.

```
def findMinMaxWithPairComparisons(A):
    ## for an even-sized Aray
    _max = A[0]
    _min = A[0]
    for indx in range(0, len(A), 2):
        first = A[indx]
        second = A[indx+1]
        if (first < second):
            if first < _min: _min = first
            if second > _max: _max = second
        else:
            if second < _min: _min = second
            if first > _max: _max = first
    print(_min)
    print(_max)

findMinMaxWithPairComparisons([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13,19])
```

Time Complexity – $O(n)$. Space Complexity – $O(1)$.

Number of comparisons: $\begin{cases} \frac{3n}{2} - 2, if\ n\ is\ even \\ \frac{3n}{2} - \frac{3}{2}\ if\ n\ is\ odd \end{cases}$

Summary:

Straightforward comparison – $2(n-1)$ comparisons
Compare for min only if comparison for max fails

Best case: increasing order – $n - 1$ comparisons
Worst case: decreasing order – $2(n - 1)$ comparisons
Average case: $3n/2 - 1$ comparisons

Note: For divide and conquer techniques refer to *Divide and Conquer* chapter.

Problem-4 Give an algorithm for finding the second largest element in the given input list of elements.

Solution: Brute Force Method

Algorithm:

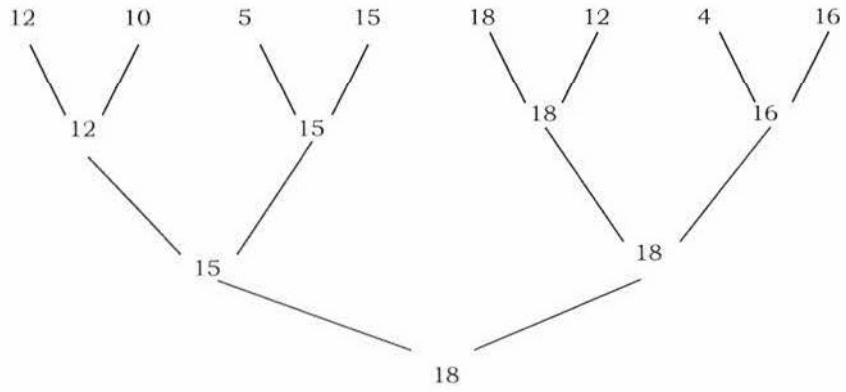
- Find largest element: needs $n - 1$ comparisons
- Delete (discard) the largest element
- Again find largest element: needs $n - 2$ comparisons

Total number of comparisons: $n - 1 + n - 2 = 2n - 3$

Problem-5 Can we reduce the number of comparisons in Problem-4 solution?

Solution: The Tournament method: For simplicity, assume that the numbers are distinct and that n is a power of 2. We pair the keys and compare the pairs in rounds until only one round remains. If the input has eight keys, there are four comparisons in the first round, two in the second, and one in the last. The winner of the last round is the largest key. The figure below shows the method.

The tournament method directly applies only when n is a power of 2. When this is not the case, we can add enough items to the end of the array to make the array size a power of 2. If the tree is complete then the maximum height of the tree is $logn$. If we construct the complete binary tree, we need $n - 1$ comparisons to find the largest. The second largest key has to be among the ones that were lost in a comparison with the largest one. That means, the second largest element should be one of the opponents of the largest element. The number of keys that are lost to the largest key is the height of the tree, i.e. $logn$ [if the tree is a complete binary tree]. Then using the selection algorithm to find the largest among them, take $logn - 1$ comparisons. Thus the total number of comparisons to find the largest and second largest keys is $n + logn - 2$.




```
def secondSmallestInArray(A):
    comparisonCount = 0
    # indexes that are to be compared
    idx = range(0,len(A))

    # list of knockout for all elements
    knockout = [[] for i in idx]
    # play tournaments, until we have only one node left
    while len(idx) > 1:
        # index of nodes that win this tournament
        idx1 = []
        # nodes in idx odd, if yes then last automatically goes to next round
        odd = len(idx) % 2
        # iterate over even indexes, as we do a paired tournament
        for i in xrange(0, len(idx) - odd, 2):
            firstIndex = idx[i]
            secondIndex = idx[i+1]
            comparisonCount += 1
            # perform tournament
```

```
            if A[firstIndex] < A[secondIndex]:
                # firstIndex qualifies for next round
                idx1.append(firstIndex)
                # add A[secondIndex] to knockout list of firstIndex
                knockout[firstIndex].append(A[secondIndex])
            else:
                idx1.append(secondIndex)
                knockout[secondIndex].append(A[firstIndex])
        if odd == 1:
            idx1.append(idx[i+2])
        # perform new tournament
        idx = idx1
    print "Smallest element =", A[idx[0]]
    print "Total comparisons =", comparisonCount
    print "Nodes knocked off by the smallest =", knockout[idx[0]], "\n"
    # compute second smallest
    a = knockout[idx[0]]
    if len(a) > 0:
        v = a[0]
        for i in xrange(1,len(a)):
            comparisonCount += 1
            if v > a[i]: v = a[i]
        print "Second smallest element =", v
        print "Total comparisons =", comparisonCount
A = [2, 4, 3, 7, 3, 0, 8, 4, 11, 1]
print(secondSmallestInArray(A))
```

Problem-6 Find the k-smallest elements in an array S of n elements using partitioning method.

Solution: Brute Force Approach: Scan through the numbers k times to have the desired element. This method is the one used in bubble sort (and selection sort), every time we find out the smallest element in the whole sequence by comparing every element. In this method, the sequence has to be traversed k times. So the complexity is $O(n \times k)$.

Problem-7 Can we use the sorting technique for solving Problem-6?

Solution: Yes. Sort and take the first k elements.

1. Sort the numbers.
2. Pick the first k elements.

The time complexity calculation is trivial. Sorting of n numbers is of $O(nlogn)$ and picking k elements is of $O(k)$. The total complexity is $O(nlogn + k) = O(nlogn)$.

Problem-8 Can we use the *tree sorting* technique for solving Problem-6?

Solution: Yes.

1. Insert all the elements in a binary search tree.
2. Do an InOrder traversal and print k elements which will be the smallest ones. So, we have the k smallest elements.

The cost of creation of a binary search tree with n elements is $O(nlogn)$ and the traversal up to k elements is $O(k)$. Hence the complexity is $O(nlogn + k) = O(nlogn)$.

Disadvantage: If the numbers are sorted in descending order, we will be getting a tree which will be skewed towards the left. In that case, the construction of the tree will be $0 + 1 + 2 + \ldots + (n - 1) = \frac{n(n-1)}{2}$ which is $O(n^2)$. To escape from this, we can keep the tree balanced, so that the cost of constructing the tree will be only $nlogn$.

Problem-9 Can we improve the *tree sorting* technique for solving Problem-6?

Solution: Yes. Use a smaller tree to give the same result.

1. Take the first k elements of the sequence to create a balanced tree of k nodes (this will cost $klogk$).
2. Take the remaining numbers one by one, and
    a. If the number is larger than the largest element of the tree, return.

b. If the number is smaller than the largest element of the tree, remove the largest element of the tree and add the new element. This step is to make sure that a smaller element replaces a larger element from the tree. And of course the cost of this operation is $logk$ since the tree is a balanced tree of k elements.

Once Step 2 is over, the balanced tree with k elements will have the smallest k elements. The only remaining task is to print out the largest element of the tree.

Time Complexity:

1. For the first k elements, we make the tree. Hence the cost is $klogk$.
2. For the rest $n - k$ elements, the complexity is O($logk$).

Step 2 has a complexity of $(n - k)\, logk$. The total cost is $klogk + (n - k)\, logk = nlogk$ which is O($nlogk$). This bound is actually better than the ones provided earlier.

Problem-10 Can we use the partitioning technique for solving Problem-6?

Solution: Yes.

Algorithm

1. Choose a pivot from the array.
2. Partition the array so that: $A[low...pivotpoint - 1] <= pivotpoint <= A[pivotpoint + 1..high]$.
3. if $k < pivotpoint$ then it must be on the left of the pivot, so do the same method recursively on the left part.
4. if $k = pivotpoint$ then it must be the pivot and print all the elements from low to $pivotpoint$.
5. if $k > pivotpoint$ then it must be on the right of pivot, so do the same method recursively on the right part.

The input data can be any iterable. The randomization of pivots makes the algorithm perform consistently even with unfavorable data orderings.

```
import random
def kthSmallest(data, k):
    "Find the nth rank ordered element (the least value has rank 0)."
    data = list(data)
    if not 0 <= k < len(data):
        raise ValueError('not enough elements for the given rank')
    while True:
        pivot = random.choice(data)
        pcount = 0
        under, over = [], []
        uappend, oappend = under.append, over.append
        for elem in data:
            if elem < pivot:
                uappend(elem)
            elif elem > pivot:
                oappend(elem)
            else:
                pcount += 1
        if k < len(under):
            data = under
        elif k < len(under) + pcount:
            return pivot
        else:
            data = over
            k -= len(under) + pcount
print(kthSmallest([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13], 5))
```

Time Complexity: O(n^2) in worst case as similar to Quicksort. Although the worst case is the same as that of Quicksort, this performs much better on the average [O($nlogk$) – Average case].

Problem-11 Find the k^{th}-smallest element in an array S of n elements in best possible way.

Solution: This problem is similar to Problem-6 and all the solutions discussed for Problem-6 are valid for this problem. The only difference is that instead of printing all the k elements, we print only the k^{th}element. We can

improve the solution by using the *median of medians* algorithm. Median is a special case of the selection algorithm. The algorithm Selection(A, k) to find the k^{th} smallest element from set A of n elements is as follows:

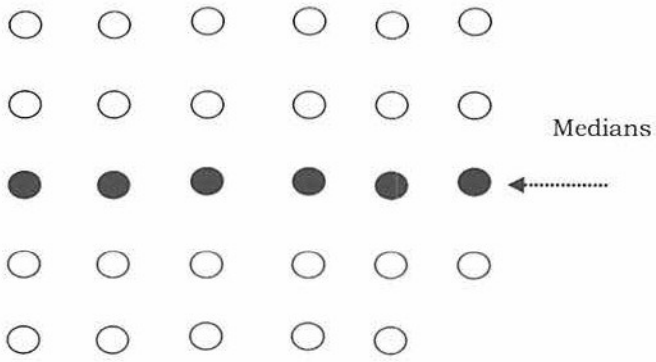
Algorithm: *Selection*(A, k)

1. Partition A into $ceil\left(\frac{length(A)}{5}\right)$ groups, with each group having five items (the last group may have fewer items).
2. Sort each group separately (e.g., insertion sort).
3. Find the median of each of the $\frac{n}{5}$ groups and store them in some array (let us say A').
4. Use *Selection* recursively to find the median of A' (median of medians). Let us asay the median of medians is m.

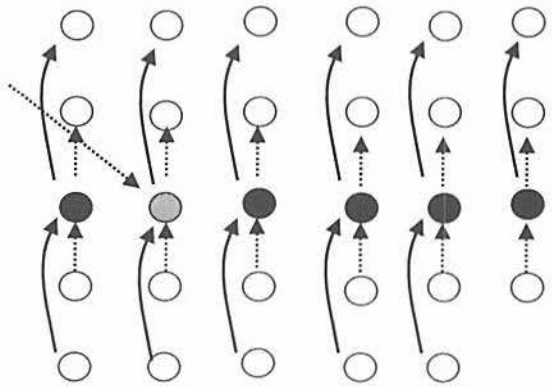
   $$m = Selection(A', \frac{\frac{length(A)}{5}}{2});$$
5. Let q = # elements of A smaller than m;
6. If($k == q + 1$)
   return m;
   /* Partition with pivot */
7. Else partition A into X and Y
   - X = {items smaller than m}
   - Y = {items larger than m}

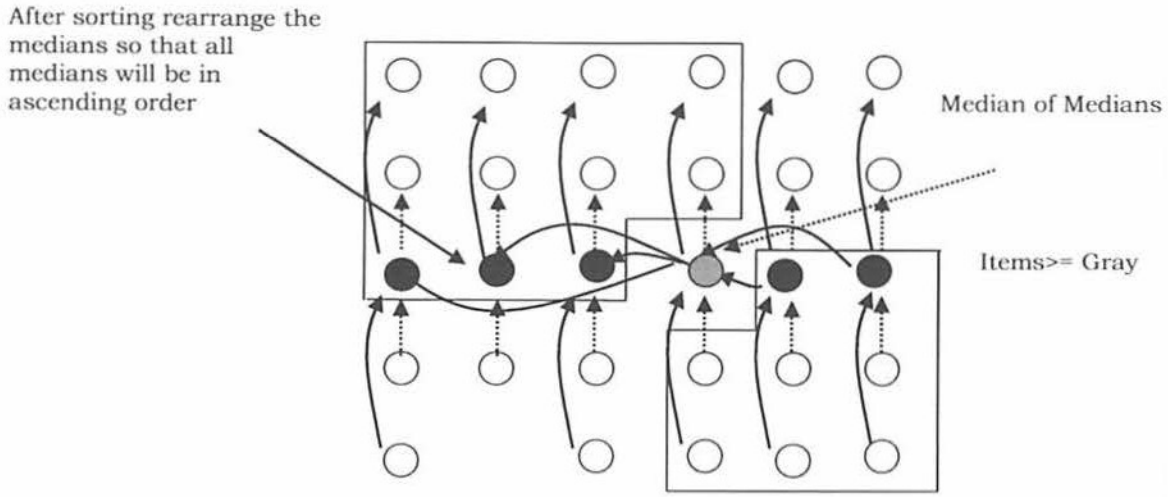
   /* Next,form a subproblem */
8. If($k < q + 1$)
   return Selection(X, k);
9. Else
   return Selection(Y, k – (q+1));

Before developing recurrence, let us consider the representation of the input below. In the figure, each circle is an element and each column is grouped with 5 elements. The black circles indicate the median in each group of 5 elements. As discussed, sort each column using constant time insertion sort.



Median of Medians





In the figure above the gray circled item is the median of medians (let us call this m). It can be seen that at least 1/2 of 5 element group medians $\leq m$. Also, these 1/2 of 5 element groups contribute 3 elements that are $\leq m$ except 2 groups [last group which may contain fewer than 5 elements, and other group which contains m]. Similarly, at least 1/2 of 5 element groups contribute 3 elements that are $\geq m$ as shown above. 1/2 of 5 element groups contribute 3 elements, except 2 groups gives: $3(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \approx \frac{3n}{10} - 6$. The remaining are $n - \frac{3n}{10} - 6 \approx \frac{7n}{10} + 6$. Since $\frac{7n}{10} + 6$ is greater than $\frac{3n}{10} - 6$ we need to consider $\frac{7n}{10} + 6$ for worst.

Components in recurrence:

- In our selection algorithm, we choose m, which is the median of medians, to be a pivot, and partition A into two sets X and Y. We need to select the set which gives maximum size (to get the worst case).
- The time in function *Selection* when called from procedure *partition*. The number of keys in the input to this call to *Selection* is $\frac{n}{5}$.
- The number of comparisons required to partition the array. This number is $length(S)$, let us say n.

We have established the following recurrence: $T(n) = T\left(\frac{n}{5}\right) + \Theta(n) + Max\{T(X), T(Y)\}$

From the above discussion we have seen that, if we select median of medians m as pivot, the partition sizes are: $\frac{3n}{10} - 6$ and $\frac{7n}{10} + 6$. If we select the maximum of these, then we get:

$$\begin{aligned} T(n) &= T\left(\tfrac{n}{5}\right) + \Theta(n) + T\left(\tfrac{7n}{10} + 6\right) \\ &\approx T\left(\tfrac{n}{5}\right) + \Theta(n) + T\left(\tfrac{7n}{10}\right) + O(1) \\ &\leq c\tfrac{7n}{10} + c\tfrac{n}{5} + \Theta(n) + O(1) \end{aligned}$$

Finally, $T(n) = \Theta(n)$.

```
CHUNK_SIZE = 5
def kthByMedianOfMedian(unsortedList, k):
    if len(unsortedList) <= CHUNK_SIZE:
        return get_kth(unsortedList, k)
    chunks = splitIntoChunks(unsortedList, CHUNK_SIZE)
    medians_list = []
    for chunk in chunks:
        median_chunk = get_median(chunk)
        medians_list.append(median_chunk)
    size = len(medians_list)
    mom = kthByMedianOfMedian(medians_list, size / 2 + (size % 2))
    smaller, larger = splitListByPivot(unsortedList, mom)
    valuesBeforeMom = len(smaller)
    if valuesBeforeMom == (k - 1):
        return mom
    elif valuesBeforeMom > (k - 1):
        return kthByMedianOfMedian(smaller, k)
    else:
        return kthByMedianOfMedian(larger, k - valuesBeforeMom - 1)
```

Problem-12 In Problem-11, we divided the input array into groups of 5 elements. The constant 5 play an important part in the analysis. Can we divide in groups of 3 which work in linear time?

Solution: In this case the modification causes the routine to take more than linear time. In the worst case, at least half of the $\lceil \frac{n}{3} \rceil$ medians found in the grouping step are greater than the median of medians m, but two of those groups contribute less than two elements larger than m. So as an upper bound, the number of elements larger than the pivotpoint is at least:

$$2(\lceil \frac{1}{2} \lceil \frac{n}{3} \rceil \rceil - 2) \geq \frac{n}{3} - 4$$

Likewise this is a lower bound. Thus up to $n - (\frac{n}{3} - 4) = \frac{2n}{3} + 4$ elements are fed into the recursive call to *Select*. The recursive step that finds the median of medians runs on a problem of size $\lceil \frac{n}{3} \rceil$, and consequently the time recurrence is:

$$T(n) = T(\lceil \frac{n}{3} \rceil) + T(2n/3 + 4) + \Theta(n).$$

Assuming that $T(n)$ is monotonically increasing, we may conclude that $T(\frac{2n}{3} + 4) \geq T(\frac{2n}{3}) \geq 2T(\frac{n}{3})$, and we can say the upper bound for this as $T(n) \geq 3T(\frac{n}{3}) + \Theta(n)$, which is $O(nlogn)$. Therefore, we cannot select 3 as the group size.

Problem-13 As in Problem-12, can we use groups of size 7?

Solution: Following a similar reasoning, we once more modify the routine, now using groups of 7 instead of 5. In the worst case, at least half the $\lceil \frac{n}{7} \rceil$ medians found in the grouping step are greater than the median of medians m, but two of those groups contribute less than four elements larger than m. So as an upper bound, the number of elements larger than the pivotpoint is at least:

$$4(\lceil 1/2 \lceil n/7 \rceil \rceil - 2) \geq \frac{2n}{7} - 8.$$

Likewise this is a lower bound. Thus up to $n - (\frac{2n}{7} - 8) = \frac{5n}{7} + 8$ elements are fed into the recursive call to Select. The recursive step that finds the median of medians runs on a problem of size $\lceil \frac{n}{7} \rceil$, and consequently the time recurrence is

$$\begin{aligned} T(n) &= T(\lceil \tfrac{n}{7} \rceil) + T(\tfrac{5n}{7} + 8) + O(n) \\ T(n) &\leq c\lceil \tfrac{n}{7} \rceil + c(\tfrac{5n}{7} + 8) + O(n) \\ &\leq c\frac{n}{7} + c\frac{5n}{7} + 8c + an, a \text{ is a constant} \\ &= cn - c\frac{n}{7} + an + 9c \\ &= (a + c)n - (c\frac{n}{7} - 9c). \end{aligned}$$

This is bounded above by $(a + c)\, n$ provided that $c\frac{n}{7} - 9c \geq 0$. Therefore, we can select 7 as the group size.

Problem-14 Given two arrays each containing n sorted elements, give an $O(logn)$-time algorithm to find the median of all $2n$ elements.

Solution: The simple solution to this problem is to merge the two lists and then take the average of the middle two elements (note the union always contains an even number of values). But, the merge would be $\Theta(n)$, so that doesn't satisfy the problem statement. To get *logn* complexity, let *medianA* and *medianB* be the medians of the respective lists (which can be easily found since both lists are sorted). If *medianA* == *medianB*, then that is the overall median of the union and we are done. Otherwise, the median of the union must be between *medianA* and *medianB*. Suppose that *medianA* < *medianB* (the opposite case is entirely similar). Then we need to find the median of the union of the following two sets:

$$\{x \text{ in } A \mid x >= medianA\} \{x \text{ in } B \mid x <= medianB\}$$

So, we can do this recursively by resetting the *boundaries* of the two arrays. The algorithm tracks both arrays (which are sorted) using two indices. These indices are used to access and compare the median of both arrays to find where the overall median lies.

```
def findKthSmallest(A, B, k):
    if len(A) > len(B):          A, B = B, A
    # stepsA = (endIndex + beginIndex_as_0) / 2
    stepsA = (min(len(A), k) -1)/ 2
    # stepsB =  k - (stepsA + 1) -1 for the 0-based index
    stepsB = k - stepsA - 2
```

```
    # Only array B contains elements
    if len(A) == 0:             return B[k-1]
    # Both A and B contain elements, and we need the smallest one
    elif k == 1:                return min(A[0], B[0])
    # The median would be either A[stepsA] or B[stepsB], while A[stepsA] and
    # B[stepsB] have the same value.
    elif A[stepsA] == B[stepsB]:    return A[stepsA]
    # The median must be in the right part of B or left part of A
    elif A[stepsA] > B[stepsB]:     return findKthSmallest(A, B[stepsB+1:], k-stepsB-1)
    # The median must be in the right part of A or left part of B
    else: return findKthSmallest(A[stepsA+1:], B, k-stepsA-1)

def findMedianSortedArrays(A, B):
        # There must be at least one element in these two arrays
        assert not(len(A) == 0 and len(B) == 0)

        if (len(A)+len(B))%2==1:
            # There are odd number of elements in total. The median the one in the middle
            return findKthSmallest(A, B, (len(A)+len(B))/2+1) * 1.0
        else:
            # There are even number of elements in total. The median the mean value of the
            # middle two elements.
            return ( findKthSmallest(A, B, (len(A)+len(B))/2+1) + findKthSmallest(A, B, (len(A)+len(B))/2) ) / 2.0

A = [127, 220, 246, 277, 321, 454, 534, 565, 933]
B = [12, 22, 24, 27, 32, 45, 53, 65, 93]

print(findMedianSortedArrays(A,B))
```

Time Complexity: O($logn$), since we are reducing the problem size by half every time.

Problem-15 Let A and B be two sorted arrays of n elements each. We can easily find the k^{th} smallest element in A in O(1) time by just outputting $A[k]$. Similarly, we can easily find the k^{th} smallest element in B. Give an O($logk$) time algorithm to find the k^{th} smallest element overall { *i.e.*, the k^{th} smallest in the union of A and B.

Time Complexity: O($logn$), since we are reducing the problem size by half every time.

Problem-16 Let A and B be two sorted arrays of n elements each. We can easily find the k^{th} smallest element in A in O(1) time by just outputting $A[k]$. Similarly, we can easily find the k^{th} smallest element in B. Give an O($logk$) time algorithm to find the k^{th} smallest element overall { *i.e.*, the k^{th} smallest in the union of A and B.

Solution: It's just another way of asking Problem-14.

Problem-17 **Find the k smallest elements in sorted order:** Given a set of n elements from a totally-ordered domain, find the k smallest elements, and list them in sorted order. Analyze the worst-case running time of the best implementation of the approach.

Solution: Sort the numbers, and list the k smallest.

$T(n)$ = Time complexity of sort + listing k smallest elements = $\Theta(nlogn) + \Theta(n) = \Theta(nlogn)$.

Problem-18 For Problem-17, if we follow the approach below, then what is the complexity?

Solution: Using the priority queue data structure from heap sort, construct a min-heap over the set, and perform extract-min k times. Refer to the *Priority Queues (Heaps)* chapter for more details.

Problem-19 For Problem-17, if we follow the approach below then what is the complexity?
Find the k^{th}-smallest element of the set, partition around this pivot element, and sort the k smallest elements.

Solution:

$$T(n) = Time\ complexity\ of\ kth - smallest + Finding\ pivot + Sorting\ prefix$$
$$= \Theta(n) + \Theta(n) + \Theta(klogk) = \Theta(n + klogk)$$

Since, $k \leq n$, this approach is better than Problem-17 and Problem-18.

Problem-20 Find k nearest neighbors to the median of n distinct numbers in O(n) time.

Solution: Let us assume that the array elements are sorted. Now find the median of n numbers and call its index as X (since array is sorted, median will be at $\frac{n}{2}$ location). All we need to do is select k elements with the smallest absolute differences from the median, moving from $X-1$ to 0, and $X+1$ to $n-1$ when the median is at index m.

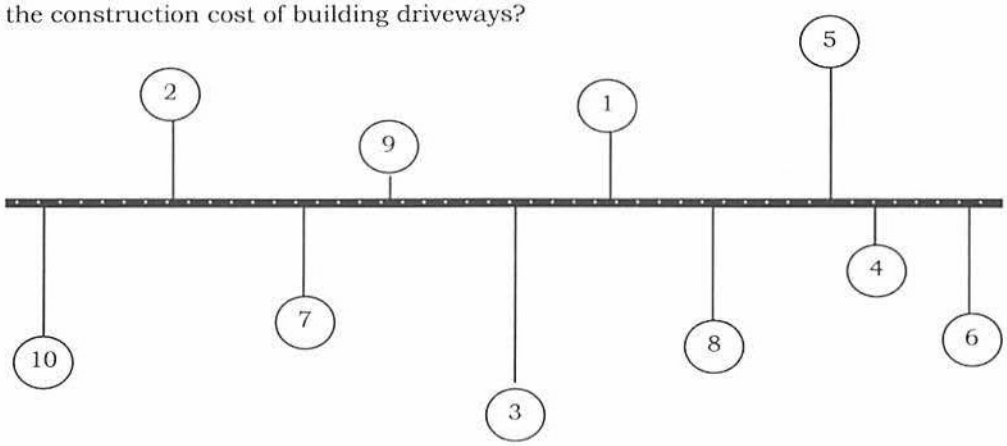
Time Complexity: Each step takes $\Theta(n)$. So the total time complexity of the algorithm is $\Theta(n)$.

Problem-21 Is there any other way of solving Problem-20?

Solution: Assume for simplicity that n is odd and k is even. If set A is in sorted order, the median is in position $n/2$ and the k numbers in A that are closest to the median are in positions $(n - k)/2$ through $(n + k)/2$.

We first use linear time selection to find the $(n - k)/2$, $n/2$, and $(n + k)/2$ elements and then pass through set A to find the numbers less than the $(n+k)/2$ element, greater than the $(n-k)/2$ element, and not equal to the $n/2$ element. The algorithm takes $O(n)$ time as we use linear time selection exactly three times and traverse the n numbers in A once.

Problem-22 Given (x, y) coordinates of n houses, where should you build a road parallel to x-axis to minimize the construction cost of building driveways?



Solution: The road costs nothing to build. It is the driveways that cost money. The driveway cost is proportional to its distance from the road. Obviously, they will be perpendicular. The solution is to put the street at the median of the y coordinates.

Problem-23 Given a big file containing billions of numbers, find the maximum 10 numbers from that file.

Solution: Refer to the *Priority Queues* chapter.

Problem-24 Suppose there is a milk company. The company collects milk every day from all its agents. The agents are located at different places. To collect the milk, what is the best place to start so that the least amount of total distance is travelled?

Solution: Starting at the median reduces the total distance travelled because it is the place which is at the center of all the places.

CHAPTER 13

SYMBOL TABLES

13.1 Introduction

Since childhood, we all have used a dictionary, and many of us have a word processor (say, Microsoft Word) which comes with a spell checker. The spell checker is also a dictionary but limited in scope. There are many real time examples for dictionaries and a few of them are:

- Spell checker
- The data dictionary found in database management applications
- Symbol tables generated by loaders, assemblers, and compilers
- Routing tables in networking components (DNS lookup)

In computer science, we generally use the term 'symbol table' rather than 'dictionary' when referring to the abstract data type (ADT).

13.2 What are Symbol Tables?

We can define the *symbol table* as a data structure that associates a *value* with a *key*. It supports the following operations:

- Search whether a particular name is in the table
- Get the attributes of that name
- Modify the attributes of that name
- Insert a new name and its attributes
- Delete a name and its attributes

There are only three basic operations on symbol tables: searching, inserting, and deleting.

Example: DNS lookup. Let us assume that the key in this case is the URL and the value is an IP address.

- Insert URL with specified IP address
- Given URL, find corresponding IP address

Key[Website]	Value [IP Address]
www.CareerMonks.com	128.112.136.11
www.AuthorsInn.com	128.112.128.15
www.AuthInn.com	130.132.143.21
www.klm.com	128.103.060.55
www.CareerMonk.com	209.052.165.60

13.3 Symbol Table Implementations

Before implementing symbol tables, let us enumerate the possible implementations. Symbol tables can be implemented in many ways and some of them are listed below.

Unordered Array Implementation

With this method, just maintaining an array is enough. It needs O(n) time for searching, insertion and deletion in the worst case.

Ordered [Sorted] Array Implementation

In this we maintain a sorted array of keys and values.

- Store in sorted order by key
- keys[i] = i^{th} largest key
- values[i] = value associated with i^{th} largest key

Since the elements are sorted and stored in arrays, we can use a simple binary search for finding an element. It takes O($logn$) time for searching and O(n) time for insertion and deletion in the worst case.

Unordered Linked List Implementation

Just maintaining a linked list with two data values is enough for this method. It needs O(n) time for searching, insertion and deletion in the worst case.

Ordered Linked List Implementation

In this method, while inserting the keys, maintain the order of keys in the linked list. Even if the list is sorted, in the worst case it needs O(n) time for searching, insertion and deletion.

Binary Search Trees Implementation

Refer to *Trees* chapter. The advantages of this method are: it does not need much code and it has a fast search [O($logn$) on average].

Balanced Binary Search Trees Implementation

Refer to *Trees* chapter. It is an extension of binary search trees implementation and takes O($logn$) in worst case for search, insert and delete operations.

Ternary Search Implementation

Refer to *String Algorithms* chapter. This is one of the important methods used for implementing dictionaries.

Hashing Implementation

This method is important. For a complete discussion, refer to the *Hashing* chapter.

13.4 Comparison Table of Symbols for Implementations

Let us consider the following comparison table for all the implementations.

Implementation	Search	Insert	Delete
Unordered Array	n	n	n
Ordered Array (can be implemented with array binary search)	$logn$	n	n
Unordered List	n	n	n
Ordered List	n	n	n
Binary Search Trees (O($logn$) on average)	$logn$	$logn$	$logn$
Balanced Binary Search Trees (O($logn$) in worst case)	$logn$	$logn$	$logn$
Ternary Search (only change is in logarithms base)	$logn$	$logn$	$logn$
Hashing (O(1) on average)	1	1	1

Notes:

- In the above table, n is the input size.
- Table indicates the possible implementations discussed in this book. But, there could be other implementations.

CHAPTER

14

HASHING

14.1 What is Hashing?

Hashing is a technique used for storing and retrieving information as quickly as possible. It is used to perform optimal searches and is useful in implementing symbol tables.

14.2 Why Hashing?

In the *Trees* chapter we saw that balanced binary search trees support operations such as *insert*, *delete* and *search* in $O(logn)$ time. In applications, if we need these operations in $O(1)$, then hashing provides a way. Remember that worst case complexity of hashing is still $O(n)$, but it gives $O(1)$ on the average.

14.3 HashTable ADT

The common operations for hash table are:

- CreatHashTable: Creates a new hash table
- HashSearch: Searches the key in hash table
- HashInsert: Inserts a new key into hash table
- HashDelete: Deletes a key from hash table
- DeleteHashTable: Deletes the hash table

14.4 Understanding Hashing

In simple terms we can treat *array* as a hash table. For understanding the use of hash tables, let us consider the following example: Give an algorithm for printing the first repeated character if there are duplicated elements in it. Let us think about the possible solutions. The simple and brute force way of solving is: given a string, for each character check whether that character is repeated or not. The time complexity of this approach is $O(n^2)$ with $O(1)$ space complexity.

Now, let us find a better solution for this problem. Since our objective is to find the first repeated character, what if we remember the previous characters in some array?

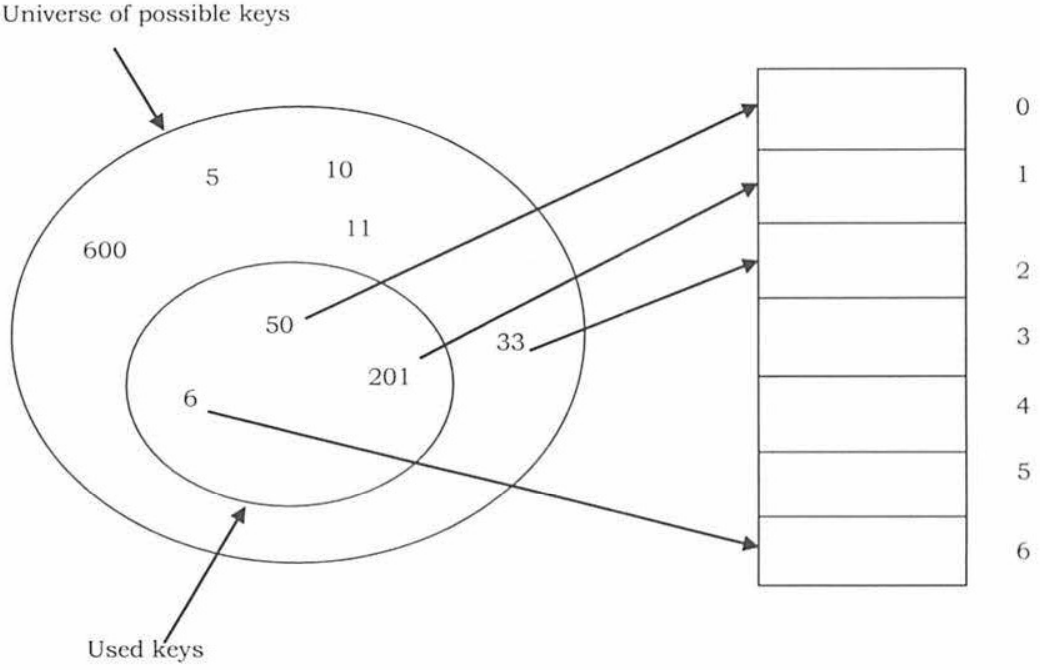
We know that the number of possible characters is 256 (for simplicity assume *ASCII* characters only). Create an array of size 256 and initialize it with all zeros. For each of the input characters go to the corresponding position and increment its count. Since we are using arrays, it takes constant time for reaching any location. While scanning the input, if we get a character whose counter is already 1 then we can say that the character is the one which is repeating for the first time.

```
def FirstRepeatedChar ( str ):
  size=len(str)
  count = [0] * (256)
  for i in range(size):
          if(count[ord(str[i])]==1):
                  print  str[i]
```

```
                break
            else:
                count[ord(str[i])] += 1
    if(i==size):
            print "No Repeated Characters"
    return 0
FirstRepeatedChar(['C','a', 'r', 'e', 'e', 'r', 'm', 'o', 'n', 'k'])
```

Why not Arrays?

In the previous problem, we have used an array of size 256 because we know the number of different possible characters [256] in advance. Now, let us consider a slight variant of the same problem. Suppose the given array has numbers instead of characters, then how do we solve the problem?



In this case the set of possible values is infinity (or at least very big). Creating a huge array and storing the counters is not possible. That means there are a set of universal keys and limited locations in the memory. If we want to solve this problem we need to somehow map all these possible keys to the possible memory locations.

From the above discussion and diagram it can be seen that we need a mapping of possible keys to one of the available locations. As a result using simple arrays is not the correct choice for solving the problems where the possible keys are very big. The process of mapping the keys to locations is called *hashing*.

Note: For now, do not worry about how the keys are mapped to locations. That depends on the function used for conversions. One such simple function is *key % table size*.

14.5 Components of Hashing

Hashing has four key components:

1) Hash Table
2) Hash Functions
3) Collisions
4) Collision Resolution Techniques

14.6 Hash Table

Hash table is a generalization of array. With an array, we store the element whose key is k at a position k of the array. That means, given a key k, we find the element whose key is k by just looking in the k^{th} position of the array. This is called *direct addressing*.

Direct addressing is applicable when we can afford to allocate an array with one position for every possible key. But if we do not have enough space to allocate a location for each possible key, then we need a mechanism to handle this case. Another way of defining the scenario is: if we have less locations and more possible keys, then simple array implementation is not enough.

In these cases one option is to use hash tables. Hash table or hash map is a data structure that stores the keys and their associated values, and hash table uses a hash function to map keys to their associated values. The general convention is that we use a hash table when the number of keys actually stored is small relative to the number of possible keys.

14.7 Hash Function

The hash function is used to transform the key into the index. Ideally, the hash function should map each possible key to a unique slot index, but it is difficult to achieve in practice.

Given a collection of elements, a hash function that maps each item into a unique slot is referred to as a *perfect hash function*. If we know the elements and the collection will never change, then it is possible to construct a perfect hash function. Unfortunately, given an arbitrary collection of elements, there is no systematic way to construct a perfect hash function. Luckily, we do not need the hash function to be perfect to still gain performance efficiency.

One way to always have a perfect hash function is to increase the size of the hash table so that each possible value in the element range can be accommodated. This guarantees that each element will have a unique slot. Although this is practical for small numbers of elements, it is not feasible when the number of possible elements is large. For example, if the elements were nine-digit Social Security numbers, this method would require almost one billion slots. If we only want to store data for a class of 25 students, we will be wasting an enormous amount of memory.

Our goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the elements in the hash table. There are a number of common ways to extend the simple remainder method. We will consider a few of them here.

The *folding method* for constructing hash functions begins by dividing the elements into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value. For example, if our element was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43,65,55,46,01). After the addition, 43+65+55+46+01, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1. Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get 43+56+55+64+01=219 which gives 219 % 11=10.

How to Choose Hash Function?

The basic problems associated with the creation of hash tables are:

- An efficient hash function should be designed so that it distributes the index values of inserted objects uniformly across the table.
- An efficient collision resolution algorithm should be designed so that it computes an alternative index for a key whose hash index corresponds to a location previously inserted in the hash table.
- We must choose a hash function which can be calculated quickly, returns values within the range of locations in our table, and minimizes collisionsns.

Characteristics of Good Hash Functions

A good hash function should have the following characteristics:

- Minimize collision
- Be easy and quick to compute
- Distribute key values evenly in the hash table
- Use all the information provided in the key
- Have a high load factor for a given set of keys

14.8 Load Factor

The load factor of a non-empty hash table is the number of items stored in the table divided by the size of the table. This is the decision parameter used when we want to rehash *or* expand the existing hash table entries. This also helps us in determining the efficiency of the hashing function. That means, it tells whether the hash function is distributing the keys uniformly or not.

$$Load\ factor = \frac{Number\ of\ elements\ in\ hash\ table}{Hash\ Table\ size}$$

14.9 Collisions

Hash functions are used to map each key to a different address space, but practically it is not possible to create such a hash function and the problem is called *collision*. Collision is the condition where two records are stored in the same location.

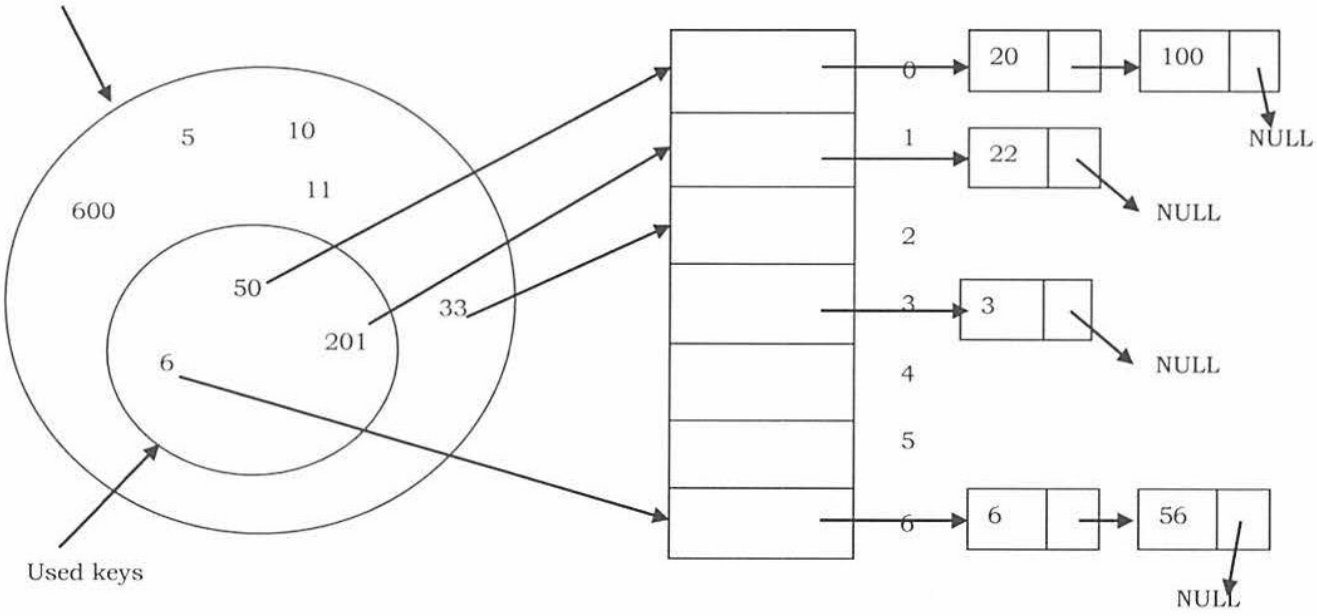
14.10 Collision Resolution Techniques

The process of finding an alternate location is called *collision resolution*. Even though hash tables have collision problems, they are more efficient in many cases compared to all other data structures, like search trees. There are a number of collision resolution techniques, and the most popular are direct chaining and open addressing.

- **Direct Chaining:** An array of linked list application
  - Separate chaining
- **Open Addressing:** Array-based implementation
  - Linear probing (linear search)
  - Quadratic probing (nonlinear search)
  - Double hashing (use two hash functions)

14.11 Separate Chaining

Collision resolution by chaining combines linked representation with hash table. When two or more records hash to the same location, these records are constituted into a singly-linked list called a *chain*.



14.12 Open Addressing

In open addressing all keys are stored in the hash table itself. This approach is also known as *closed hashing*. This procedure is based on probing. A collision is resolved by probing.

Linear Probing

The interval between probes is fixed at 1. In linear probing, we search the hash table sequentially, starting from the original hash location. If a location is occupied, we check the next location. We wrap around from the last table location to the first table location if necessary. The function for rehashing is the following:

$$rehash(key) = (n + 1)\%\ tablesize$$

One of the problems with linear probing is that table items tend to cluster together in the hash table. This means that the table contains groups of consecutively occupied locations that are called *clustering*.

Clusters can get close to one another, and merge into a larger cluster. Thus, the one part of the table might be quite dense, even though another part has relatively few items. Clustering causes long probe searches and therefore decreases the overall efficiency.

The next location to be probed is determined by the step-size, where other step-sizes (more than one) are possible. The step-size should be relatively prime to the table size, i.e. their greatest common divisor should be equal to 1. If we choose the table size to be a prime number, then any step-size is relatively prime to the table size. Clustering cannot be avoided by larger step-sizes.

Quadratic Probing

The interval between probes increases proportionally to the hash value (the interval thus increasing linearly, and the indices are described by a quadratic function). The problem of Clustering can be eliminated if we use the quadratic probing method.

In quadratic probing, we start from the original hash location i. If a location is occupied, we check the locations $i + 1^2$, $i + 2^2$, $i + 3^2$, $i + 4^2$... We wrap around from the last table location to the first table location if necessary. The function for rehashing is the following:

$$rehash(key) = (n + k^2)\%\ tablesize$$

Example: Let us assume that the table size is 11 (0..10)

Hash Function: h(key) = key mod 11

Insert keys:

$31\ mod\ 11 = 9$
$19\ mod\ 11 = 8$
$2\ mod\ 11 = 2$
$13\ mod\ 11 = 2 \rightarrow 2 + 1^2 = 3$
$25\ mod\ 11 = 3 \rightarrow 3 + 1^2 = 4$
$24\ mod\ 11 = 2 \rightarrow 2 + 1^2, 2 + 2^2 = 6$
$21\ mod\ 11 = 10$
$9\ mod\ 11 = 9 \rightarrow 9 + 1^2,\ 9 + 2^2\ mod\ 11,\ 9 + 3^2\ mod\ 11 = 7$

0	
1	
2	2
3	13
4	25
5	5
6	24
7	9
8	19
9	31
10	21

Even though clustering is avoided by quadratic probing, still there are chances of clustering. Clustering is caused by multiple search keys mapped to the same hash key. Thus, the probing sequence for such search keys is prolonged by repeated conflicts along the probing sequence. Both linear and quadratic probing use a probing sequence that is independent of the search key.

Double Hashing

The interval between probes is computed by another hash function. Double hashing reduces clustering in a better way. The increments for the probing sequence are computed by using a second hash function. The second hash function $h2$ should be:

$$h2(key) \neq 0 \text{ and } h2 \neq h1$$

We first probe the location $h1(key)$. If the location is occupied, we probe the location $h1(key) + h2(key)$, $h1(key) + 2 * h2(key)$, ...

Example:

Table size is 11 (0..10)
Hash Function: assume $h1(key) = key \ mod \ 11$ and
$h2(key) = 7 - (key \ mod \ 7)$

Insert keys:

$58 \ mod \ 11 = 3$
$14 \ mod \ 11 = 3 \rightarrow 3 + 7 = 10$
$91 \ mod \ 11 = 3 \rightarrow 3 + 7, 3 + 2 * 7 \ mod \ 11 = 6$
$25 \ mod \ 11 = 3 \rightarrow 3 + 3, 3 + 2 * 3 = 9$

0	
1	
2	
3	58
4	25
5	
6	91
7	
8	
9	25
10	14

14.13 Comparison of Collision Resolution Techniques

Comparisons: Linear Probing vs. Double Hashing

The choice between linear probing and double hashing depends on the cost of computing the hash function and on the load factor [number of elements per slot] of the table. Both use few probes but double hashing take more time because it hashes to compare two hash functions for long keys.

Comparisons: Open Addressing vs. Separate Chaining

It is somewhat complicated because we have to account for the memory usage. Separate chaining uses extra memory for links. Open addressing needs extra memory implicitly within the table to terminate the probe sequence. Open-addressed hash tables cannot be used if the data does not have unique keys. An alternative is to use separate chained hash tables.

Comparisons: Open Addressing methods

Linear Probing	Quadratic Probing	Double hashing
Fastest among three	Easiest to implement and deploy	Makes more efficient use of memory
Uses few probes	Uses extra memory for links and it does not probe all locations in the table	Uses few probes but takes more time
A problem occurs known as primary clustering	A problem occurs known as secondary clustering	More complicated to implement
Interval between probes is fixed - often at 1.	Interval between probes increases proportional to the hash value	Interval between probes is computed by another hash function

14.14 How Hashing Gets O(1) Complexity

From the previous discussion, one doubts how hashing gets O(1) if multiple elements map to the same location...

The answer to this problem is simple. By using the load factor we make sure that each block (for example, linked list in separate chaining approach) on the average stores the maximum number of elements less than the *load factor*. Also, in practice this load factor is a constant (generally, 10 or 20). As a result, searching in 20 elements or 10 elements becomes constant.

If the average number of elements in a block is greater than the load factor, we rehash the elements with a bigger hash table size. One thing we should remember is that we consider average occupancy (total number of elements in the hash table divided by table size) when deciding the rehash.

The access time of the table depends on the load factor which in turn depends on the hash function. This is because hash function distributes the elements to the hash table. For this reason, we say hash table gives O(1) complexity on average. Also, we generally use hash tables in cases where searches are more than insertion and deletion operations.

14.15 Hashing Techniques

There are two types of hashing techniques: static hashing and dynamic hashing

Static Hashing

If the data is fixed then static hashing is useful. In static hashing, the set of keys is kept fixed and given in advance, and the number of primary pages in the directory are kept fixed.

Dynamic Hashing

If the data is not fixed, static hashing can give bad performance, in which case dynamic hashing is the alternative, in which case the set of keys can change dynamically.

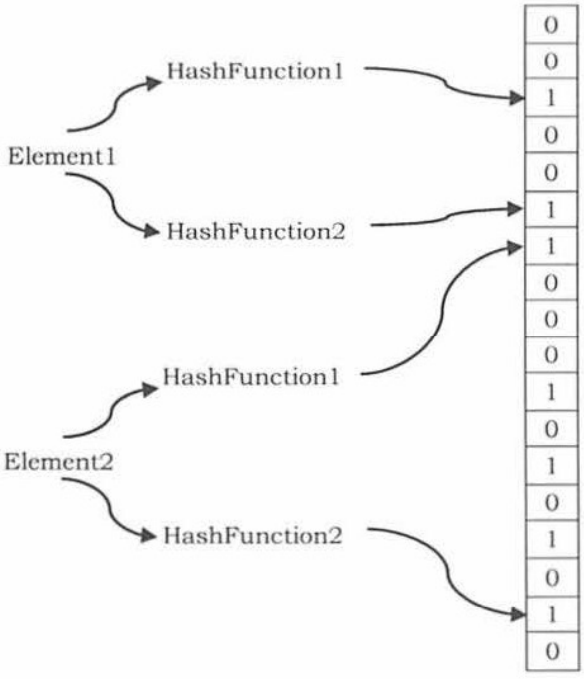
14.16 Problems for which Hash Tables are not suitable

- Problems for which data ordering is required
- Problems having multidimensional data
- Prefix searching, especially if the keys are long and of variable-lengths
- Problems that have dynamic data
- Problems in which the data does not have unique keys.

14.17 Bloom Filters

A Bloom filter is a probabilistic data structure which was designed to check whether an element is present in a set with memory and time efficiency. It tells us that the element either definitely is *not* in the set or *may* be in the set. The base data structure of a Bloom filter is a *Bit Vector*. The algorithm was invented in 1970 by Burton Bloom and it relies on the use of a number of different hash functions.

How it works?



Now that the bits in the bit vector have been set for *Element*1 and *Element*2; we can query the bloom filter to tell us if something has been seen before.

The element is hashed but instead of setting the bits, this time a check is done and if the bits that would have been set are already set the bloom filter will return true that the element has been seen before.

A Bloom filter starts off with a bit array initialized to zero. To store a data value, we simply apply k different hash functions and treat the resulting k values as indices in the array, and we set each of the k array elements to 1. We repeat this for every element that we encounter.

Now suppose an element turns up and we want to know if we have seen it before. What we do is apply the k hash functions and look up the indicated array elements. If any of them are 0 we can be 100% sure that we have never encountered the element before - if we had, the bit would have been set to 1. However, even if all of them are one, we still can't conclude that we have seen the element before because all of the bits could have

been set by the k hash functions applied to multiple other elements. All we can conclude is that it is *likely* that we have encountered the element before.

Note that it is not possible to remove an element from a Bloom filter. The reason is simply that we can't unset a bit that appears to belong to an element because it might also be set by another element.

If the bit array is mostly empty, i.e., set to zero, and the k hash functions are independent of one another, then the probability of a false positive (i.e., concluding that we have seen a data item when we actually haven't) is low. For example, if there are only k bits set, we can conclude that the probability of a false positive is very close to zero as the only possibility of error is that we entered a data item that produced the same k hash values - which is unlikely as long as the 'has' functions are independent.

As the bit array fills up, the probability of a false positive slowly increases. Of course when the bit array is full, every element queried is identified as having been seen before. So clearly we can trade space for accuracy as well as for time.

One-time removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains elements that have been removed. However, false positives in the second filter become false negatives in the composite filter, which may be undesirable. In this approach, re-adding a previously removed item is not possible, as one would have to remove it from the *removed* filter.

Selecting hash functions

The requirement of designing k different independent hash functions can be prohibitive for large k. For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple *different* hash functions by slicing its output into multiple bit fields. Alternatively, one can pass k different initial values (such as 0, 1, ..., k - 1) to a hash function that takes an initial value – or add (or append) these values to the key. For larger m and/or k, independence among the hash functions can be relaxed with negligible increase in the false positive rate.

Selecting size of bit vector

A Bloom filter with 1% error and an optimal value of k, in contrast, requires only about 9.6 bits per element — regardless of the size of the elements. This advantage comes partly from its compactness, inherited from arrays, and partly from its probabilistic nature. The 1% false-positive rate can be reduced by a factor of ten by adding only about 4.8 bits per element.

Space Advantages

While risking false positives, Bloom filters have a strong space advantage over other data structures for representing sets, such as self-balancing binary search trees, tries, hash tables, or simple arrays or linked lists of the entries. Most of these require storing at least the data items themselves, which can require anywhere from a small number of bits, for small integers, to an arbitrary number of bits, such as for strings (tries are an exception, since they can share storage between elements with equal prefixes). Linked structures incur an additional linear space overhead for pointers.

However, if the number of potential values is small and many of them can be in the set, the Bloom filter is easily surpassed by the deterministic bit array, which requires only one bit for each potential element.

Time Advantages

Bloom filters also have the unusual property that the time needed either to add items or to check whether an item is in the set is a fixed constant, O(k), completely independent of the number of items already in the set. No other constant-space set data structure has this property, but the average access time of sparse hash tables can make them faster in practice than some Bloom filters. In a hardware implementation, however, the Bloom filter shines because its k lookups are independent and can be parallelized.

Implementation

Refer to *Problems Section*.

14.18 Hashing: Problems & Solutions

Problem-1 Implement a separate chaining collision resolution technique. Also, discuss time complexities of each function.

Solution: To create a hashtable of given size, say n, we allocate an array of n/L (whose value is usually between 5 and 20) pointers to list, initialized to NULL. To perform *Search/Insert/Delete* operations, we first compute the

index of the table from the given key by using *hashfunction* and then do the corresponding operation in the linear list maintained at that location. To get uniform distribution of keys over a hashtable, maintain table size as the prime number.

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size

    def put(self,key,data):
        hashvalue = self.hashfunction(key,len(self.slots))

        if self.slots[hashvalue] == None:
            self.slots[hashvalue] = key
            self.data[hashvalue] = data
        else:
            if self.slots[hashvalue] == key:
                self.data[hashvalue] = data  #replace
            else:
                nextslot = self.rehash(hashvalue,len(self.slots))
                while self.slots[nextslot] != None and self.slots[nextslot] != key:
                    nextslot = self.rehash(nextslot,len(self.slots))

                if self.slots[nextslot] == None:
                    self.slots[nextslot]=key
                    self.data[nextslot]=data
                else:
                    self.data[nextslot] = data #replace

    def hashfunction(self,key,size):
        return key%size

    def rehash(self,oldhash,size):
        return (oldhash+1)%size

    def get(self,key):
        startslot = self.hashfunction(key,len(self.slots))

        data = None
        stop = False
        found = False
        position = startslot
        while self.slots[position] != None and  not found and not stop:
            if self.slots[position] == key:
                found = True
                data = self.data[position]
            else:
                position=self.rehash(position,len(self.slots))
                if position == startslot:
                    stop = True
        return data

    def __getitem__(self,key):
        return self.get(key)

    def __setitem__(self,key,data):
        self.put(key,data)

H=HashTable()
H[54]="books"
H[54]="data"
H[26]="algorithms"
H[93]="made"
H[17]="easy"
H[77]="CareerMOonk"
H[31]="Jobs"
H[44]="Hunting"
H[55]="King"
H[20]="Lion"
```

```
print H.slots
print H.data
print H[20]
```

CreatHashTable – O(n). HashSearch – O(1) average. HashInsert – O(1) average. HashDelete – O(1) average.

Problem-2 Given an array of characters, give an algorithm for removing the duplicates.

Solution: Start with the first character and check whether it appears in the remaining part of the string using a simple linear search. If it repeats, bring the last character to that position and decrement the size of the string by one. Continue this process for each distinct character of the given string.

```
def RemoveDuplicates(A):
    m = 0
    for i in range(0, len(A)):
        if (not elem(A, m, A[i])):
            A[m] = A[i]
            m += 1
    return m

def elem(A, n, e):
    for i in range(0, n):
        if (A[i] == e):
            return 1
    return 0

A = [54,26,93,54,77,31,44,55,20]
RemoveDuplicates(A)
print A
```

Time Complexity: O(n^2). Space Complexity: O(1).

Problem-3 Can we find any other idea to solve this problem in better time than O(n^2)? Observe that the order of characters in solutions do not matter.

Solution: Use sorting to bring the repeated characters together. Finally scan through the array to remove duplicates in consecutive positions.

```
def RemoveDuplicates(A):
        A.sort()
        j = 0
        for i in range(1,len(A)):
                if (A[j] != A[i]):
                        j += 1
                        A[j] = A[i]
        print A[:j+1]

A = [54,31,93,54,77,31,44,55,93]
RemoveDuplicates(A)
print A
```

Time Complexity: $\Theta(nlogn)$. Space Complexity: O(1).

Problem-4 Can we solve this problem in a single pass over given array?

Solution: We can use hash table to check whether a character is repeating in the given string or not. If the current character is not available in hash table, then insert it into hash table and keep that character in the given string also. If the current character exists in the hash table then skip that character.

```
A = [1, 2, 3, 'a', 'b', 'c', 2, 3, 4, 'b', 'c', 'd']
unique = []
helperSet = set()
for x in A:
    if x not in helperSet:
        unique.append(x)
        helperSet.add(x)

print A
print unique
```

Time Complexity: $\Theta(n)$ on average. Space Complexity: O(n).

Problem-5 Given two arrays of unordered numbers, check whether both arrays have the same set of numbers?

Solution: Let us assume that two given arrays are A and B. A simple solution to the given problem is: for each element of A, check whether that element is in B or not. A problem arises with this approach if there are duplicates. For example consider the following inputs:

$$A = \{2,5,6,8,10,2,2\}$$
$$B = \{2,5,5,8,10,5,6\}$$

The above algorithm gives the wrong result because for each element of A there is an element in B also. But if we look at the number of occurrences, they are not the same. This problem we can solve by moving the elements which are already compared to the end of the list. That means, if we find an element in B, then we move that element to the end of B, and in the next searching we will not find those elements. But the disadvantage of this is it needs extra swaps. Time Complexity of this approach is $O(n^2)$, since for each element of A we have to scan B.

Problem-6 Can we improve the time complexity of Problem-5?

Solution: Yes. To improve the time complexity, let us assume that we have sorted both the lists. Since the sizes of both arrays are n, we need $O(n \log n)$ time for sorting them. After sorting, we just need to scan both the arrays with two pointers and see whether they point to the same element every time, and keep moving the pointers until we reach the end of the arrays.

Time Complexity of this approach is $O(n \log n)$. This is because we need $O(n \log n)$ for sorting the arrays. After sorting, we need $O(n)$ time for scanning but it is less compared to $O(n \log n)$.

Problem-7 Can we further improve the time complexity of Problem-5?

Solution: Yes, by using a hash table. For this, consider the following algorithm.

Algorithm:

- Construct the hash table with array A elements as keys.
- While inserting the elements, keep track of the number frequency for each number. That means, if there are duplicates, then increment the counter of that corresponding key.
- After constructing the hash table for A's elements, now scan the array B.
- For each occurrence of B's elements reduce the corresponding counter values.
- At the end, check whether all counters are zero or not.
- If all counters are zero, then both arrays are the same otherwise the arrays are different.

Time Complexity: $O(n)$ for scanning the arrays. Space Complexity: $O(n)$ for hash table.

Problem-8 Given a list of number pairs; if $pair(i,j)$ exists, and $pair\ (j,i)$ exists, report all such pairs. For example, in $\{\{1,3\},\{2,6\},\{3,5\},\{7,4\},\{5,3\},\{8,7\}\}$, we see that {3,5} and {5,3} are present. Report this pair when you encounter {5,3}. We call such pairs 'symmetric pairs'. So, give an efficient algorithm for finding all such pairs.

Solution: By using hashing, we can solve this problem in just one scan. Consider the following algorithm.

Algorithm:

- Read the pairs of elements one by one and insert them into the hash table. For each pair, consider the first element as key and the second element as value.
- While inserting the elements, check if the hashing of the second element of the current pair is the same as the first number of the current pair.
- If they are the same, then that indicates a symmetric pair exits and output that pair.
- Otherwise, insert that element into that. That means, use the first number of the current pair as key and the second number as value and insert them into the hash table.
- By the time we complete the scanning of all pairs, we have output all the symmetric pairs.

Time Complexity: $O(n)$ for scanning the arrays. Note that we are doing a scan only of the input. Space Complexity: $O(n)$ for hash table.

Problem-9 Given a singly linked list, check whether it has a loop in it or not.

Solution: Using Hash Tables

Algorithm:

- Traverse the linked list nodes one by one.
- Check if the node's address is there in the hash table or not.

- If it is already there in the hash table, that indicates we are visiting a node which was already visited. This is possible only if the given linked list has a loop in it.
- If the address of the node is not there in the hash table. then insert that node's address into the hash table.
- Continue this process until we reach the end of the linked list *or* we find the loop.

Time Complexity: $O(n)$ for scanning the linked list. Note that we are doing a scan only of the input. Space Complexity: $O(n)$ for hash table.

Note: for an efficient solution, refer to the *Linked Lists* chapter.

Problem-10 Given an array of 101 elements. Out of them 50 elements are distinct, 24 elements are repeated 2 times, and one element is repeated 3 times. Find the element that is repeated 3 times in O(1).

Solution: Using Hash Tables

Algorithm:

- Scan the input array one by one.
- Check if the element is already there in the hash table or not.
- If it is already there in the hash table, increment its counter value [this indicates the number of occurrences of the element].
- If the element is not there in the hash table, insert that node into the hash table with counter value 1.
- Continue this process until reaching the end of the array.

Time Complexity: $O(n)$, because we are doing two scans. Space Complexity: $O(n)$, for hash table.

Note: For an efficient solution refer to the *Searching* chapter.

Problem-11 Given m sets of integers that have n elements in them, provide an algorithm to find an element which appeared in the maximum number of sets?

Solution: Using Hash Tables

Algorithm:

- Scan the input sets one by one.
- For each element keep track of the counter. The counter indicates the frequency of occurrences in all the sets.
- After completing the scan of all the sets, select the one which has the maximum counter value.

Time Complexity: $O(mn)$, because we need to scan all the sets. Space Complexity: $O(mn)$, for hash table. Because, in the worst case all the elements may be different.

Problem-12 Given two sets A and B, and a number K, Give an algorithm for finding whether there exists a pair of elements, one from A and one from B, that add up to K.

Solution: For simplicity, let us assume that the size of A is m and the size of B is n.

Algorithm:

- Select the set which has minimum elements.
- For the selected set create a hash table. We can use both key and value as the same.
- Now scan the second array and check whether *(K-selected element)* exists in the hash table or not.
- If it exists then return the pair of elements.
- Otherwise continue until we reach the end of the set.

Time Complexity: $O(Max(m,n))$, because we are doing two scans. Space Complexity: $O(Min(m,n))$, for hash table. We can select the small set for creating the hash table.

Problem-13 Give an algorithm to remove the specified characters from a given string which are given in another string?

Solution: For simplicity, let us assume that the maximum number of different characters is 256. First we create an auxiliary array initialized to 0. Scan the characters to be removed, and for each of those characters we set the value to 1, which indicates that we need to remove that character.

After initialization, scan the input string, and for each of the characters, we check whether that character needs to be deleted or not. If the flag is set then we simply skip to the next character, otherwise we keep the character in the input string. Continue this process until we reach the end of the input string. All these operations we can do in-place as given below.

```
def RemoveChars(str, removeTheseChars):
    table = {}          # hash
```

```
    temp = []
    #set true for all characters to be removed
    for char in removeTheseChars.lower():
            table[char] = 1
    index = 0
    for char in str.lower():
            if char in table:
                    continue
            else:
                    temp.append(char)
                    index += 1
    return "".join(temp)
print RemoveChars("careermonk", "e")
```

Time Complexity: Time for scanning the characters to be removed + Time for scanning the input array= $O(n) + O(m) \approx O(n)$. Where m is the length of the characters to be removed and n is the length of the input string.

Space Complexity: $O(m)$, length of the characters to be removed. But since we are assuming the maximum number of different characters is 256, we can treat this as a constant. But we should keep in mind that when we are dealing with multi-byte characters, the total number of different characters is much more than 256.

Problem-14 Give an algorithm for finding the first non-repeated character in a string. For example, the first non-repeated character in the string "*abzddab*" is 'z'.

Solution: The solution to this problem is trivial. For each character in the given string, we can scan the remaining string if that character appears in it. If it does not appears then we are done with the solution and we return that character. If the character appears in the remaining string, then go to the next character.

```
def findNonrepeated(A):
    n = len(A)
    for i in range(0,n):
            repeated = 0
            for j in range(0,n):
                    if( i != j and A[i] == A[j]):
                            repeated = 1
            if repeated == 0:
                    return A[i]
    return
print findNonrepeated("careermonk")
```

Time Complexity: $O(n^2)$, for two for loops. Space Complexity: $O(1)$.

Problem-15 Can we improve the time complexity of 0?

Solution: Yes. By using hash tables we can reduce the time complexity. Create a hash table by reading all the characters in the input string and keeping count of the number of times each character appears. After creating the hash table, we can read the hash table entries to see which element has a count equal to 1. This approach takes $O(n)$ space but reduces the time complexity also to $O(n)$.

```
def findNonrepeated(A):
    table = {}          # hash
    for char in A.lower():
            if char in table:
                    table[char] += 1
            elif char != " ":
                    table[char] = 1
            else:
                    table[char] = 0
    for char in A.lower():
            if table[char] == 1:
                    print("the first non repeated character is: %s" % (char))
                    return char
    return
print findNonrepeated("careermonk")
```

Time Complexity: We have $O(n)$ to create the hash table and another $O(n)$ to read the entries of hash table. So the total time is $O(n) + O(n) = O(2n) \approx O(n)$. Space Complexity: $O(n)$ for keeping the count values.

Problem-16 Given a string, give an algorithm for finding the first repeating letter in a string?

Solution: The solution to this problem is somewhat similar to 0 and Problem-15. The only difference is, instead of scanning the hash table twice we can give the answer in just one scan. This is because while inserting into the hash table we can see whether that element already exists or not. If it already exists then we just need to return that character.

```
def firstRepeatedChar(A):
    table = {}          # hash
    for char in A.lower():
            if char in table:
                    table[char] += 1
                    print("the first repeated character is: %s" % (char))
                    return char
            elif char != " ":
                    table[char] = 1
            else:
                    table[char] = 0
    return
print firstRepeatedChar("careermonk")
```

Time Complexity: We have $O(n)$ for scanning and creating the hash table. Note that we need only one scan for this problem. So the total time is $O(n)$. Space Complexity: $O(n)$ for keeping the count values.

Problem-17 Given an array of n numbers, create an algorithm which displays all pairs whose sum is S.

Solution: This problem is similar to Problem-12. But instead of using two sets we use only one set.

Algorithm:

- Scan the elements of the input array one by one and create a hash table. Both key and value can be the same.
- After creating the hash table, again scan the input array and check whether (S − *selected element*) exits in the hash table *or* not.
- If it exits then return the pair of elements.
- Otherwise continue and read all the elements of the array.

Time Complexity: We have $O(n)$ to create the hash table and another $O(n)$ to read the entries of the hash table. So the total time is $O(n) + O(n) = O(2n) \approx O(n)$. Space Complexity: $O(n)$ for keeping the count values.

Problem-18 Is there any other way of solving Problem-17?

Solution: Yes. The alternative solution to this problem involves sorting. First sort the input array. After sorting, use two pointers, one at the starting and another at the ending. Each time add the values of both the indexes and see if their sum is equal to S. If they are equal then print that pair. Otherwise increase the left pointer if the sum is less than S and decrease the right pointer if the sum is greater than S.

Time Complexity: Time for sorting + Time for scanning = $O(nlogn) + O(n) \approx O(nlogn)$.
Space Complexity: O(1).

Problem-19 We have a file with millions of lines of data. Only two lines are identical; the rest are unique. Each line is so long that it may not even fit in the memory. What is the most efficient solution for finding the identical lines?

Solution: Since a complete line may not fit into the main memory, read the line partially and compute the hash from that partial line. Then read the next part of the line and compute the hash. This time use the previous hash also while computing the new hash value. Continue this process until we find the hash for the complete line. Do this for each line and store all the hash values in a file [or maintain a hash table of these hashes]. If at any point you get same hash value, read the corresponding lines part by part and compare.

Note: Refer to *Searching* chapter for related problems.

Problem-20 If h is the hashing function and is used to hash n keys into a table of size s, where $n <= s$, the expected number of collisions involving a particular key X is :

(A) less than 1. (B) less than n. (C) less than s. (D) less than $\frac{n}{2}$.

Solution: A.

Problem-21 Implement Bloom Filters

Solution: A Bloom Filter is a data structure designed to tell, rapidly and memory-efficiently, whether an element is present in a set. It is based on a probabilistic mechanism where false positive retrieval results are possible, but false negatives are not. At the end we will see how to tune the parameters in order to minimize the number of false positive results.

Let's begin with a little bit of theory. The idea behind the Bloom filter is to allocate a bit vector of length m, initially all set to 0, and then choose k independent hash functions, h_1, h_2, ..., h_k, each with range $[1..m]$. When an element a is added to the set then the bits at positions $h_1(a)$, h_2(a), ..., h_k(a) in the bit vector are set to 1. Given a query element q we can test whether it is in the set using the bits at positions h_1(q), h_2(q), ..., h_k(q) in the vector. If any of these bits is 0 we report that q is not in the set otherwise we report that q is. The thing we have to care about is that in the first case there remains some probability that q is not in the set which could lead us to a false positive response.

```
class BloomFilter:
 """ Bloom Filter """
 def __init__(self,m,k,hashFun):
        self.m = m
        self.vector = [0]*m
        self.k = k
        self.hashFun = hashFun
        self.data = {} # data structure to store the data
        self.flasePositive = 0
 def insert(self,key,value):
        self.data[key] = value
        for i in range(self.k):
                self.vector[self.hashFun(key+str(i)) % self.m] = 1
 def contains(self,key):
        for i in range(self.k):
                if self.vector[self.hashFun(key+str(i)) % self.m] == 0:
                        return False # the key doesn't exist
        return True # the key can be in the data set
 def get(self,key):
        if self.contains(key):
                try:
                        return self.data[key] # actual lookup
                except KeyError:
                        self.flasePositive += 1
import hashlib
def hashFunction(x):
        h = hashlib.sha256(x) # we'll use sha256 just for this example
        return int(h.hexdigest(),base=16)
b = BloomFilter(100,10,hashFunction)
b.insert('this is a test key','this is a new value')
print b.get('this is a key')
print b.get('this is a test key')
```

CHAPTER

15

STRING ALGORITHMS

15.1 Introduction

To understand the importance of string algorithms let us consider the case of entering the URL (Uniform To understand the importance of string algorithms let us consider the case of entering the URL (Uniform Resource Locator) in any browser (say, Internet Explorer, Firefox, or Google Chrome). You will observe that after typing the prefix of the URL, a list of all possible URLs is displayed. That means, the browsers are doing some internal processing and giving us the list of matching URLs. This technique is sometimes called *auto − completion*.

Similarly, consider the case of entering the directory name in the command line interface (in both *Windows* and *UNIX*). After typing the prefix of the directory name, if we press the *tab* button, we get a list of all matched directory names available. This is another example of auto completion.

In order to support these kinds of operations, we need a data structure which stores the string data efficiently. In this chapter, we will look at the data structures that are useful for implementing string algorithms.

We start our discussion with the basic problem of strings: given a string, how do we search a substring (pattern)? This is called a *string matching* problem. After discussing various string matching algorithms, we will look at different data structures for storing strings.

15.2 String Matching Algorithms

In this section, we concentrate on checking whether a pattern P is a substring of another string T (T stands for text) or not. Since we are trying to check a fixed string P, sometimes these algorithms are called *exact string matching* algorithms. To simplify our discussion, let us assume that the length of given text T is n and the length of the pattern P which we are trying to match has the length m. That means, T has the characters from 0 to $n-1$ ($T[0 \ldots n-1]$) and P has the characters from 0 to $m-1$ ($P[0 \ldots m-1]$). This algorithm is implemented in $C++$ as *strstr*().

In the subsequent sections, we start with the brute force method and gradually move towards better algorithms.

- Brute Force Method
- Robin-Karp String Matching Algorithm
- String Matching with Finite Automata
- KMP Algorithm
- Boyce-Moore Algorithm
- Suffix Trees

15.3 Brute Force Method

In this method, for each possible position in the text T we check whether the pattern P matches or not. Since the length of T is n, we have $n-m+1$ possible choices for comparisons. This is because we do not need to check the last $m-1$ locations of T as the pattern length is m. The following algorithm searches for the first occurrence of a pattern string P in a text string T..

Algorithm

```
def strstrBruteForce(str, pattern):
    if not pattern: return 0
    for i in range(len(str)-len(pattern)+1):
        stri = i; patterni = 0
        while stri < len(str) and patterni < len(pattern) and str[stri] == pattern[patterni]:
            stri += 1
            patterni += 1
        if patterni == len(pattern): return i
    return -1

print strstrBruteForce("xxxxyzabcdabcdefabc", "abc")
```

Time Complexity: $O((n-m+1)\times m) \approx O(n\times m)$. Space Complexity: $O(1)$.

15.4 Robin-Karp String Matching Algorithm

In this method, we will use the hashing technique and instead of checking for each possible position in T, we check only if the hashing of P and the hashing of m characters of T give the same result.

Initially, apply the hash function to the first m characters of T and check whether this result and P's hashing result is the same or not. If they are not the same, then go to the next character of T and again apply the hash function to m characters (by starting at the second character). If they are the same then we compare those m characters of T with P.

Selecting Hash Function

At each step, since we are finding the hash of m characters of T, we need an efficient hash function. If the hash function takes $O(m)$ complexity in every step, then the total complexity is $O(n\times m)$. This is worse than the brute force method because first we are applying the hash function and also comparing.

Our objective is to select a hash function which takes $O(1)$ complexity for finding the hash of m characters of T every time. Only then can we reduce the total complexity of the algorithm. If the hash function is not good (worst case), the complexity of the Robin-Karp algorithm is $O(n-m+1)\times m) \approx O(n\times m)$. If we select a good hash function, the complexity of the Robin-Karp algorithm complexity is $O(m+n)$. Now let us see how to select a hash function which can compute the hash of m characters of T at each step in $O(1)$.

For simplicity, let's assume that the characters used in string T are only integers. That means, all characters in $T \in \{0,1,2,\ldots,9\}$. Since all of them are integers, we can view a string of m consecutive characters as decimal numbers. For example, string '61815' corresponds to the number 61815. With the above assumption, the pattern P is also a decimal value, and let us assume that the decimal value of P is p. For the given text $T[0..n-1]$, let $t(i)$ denote the decimal value of length−m substring $T[i..i+m-1]$ for $i = 0,1,\ldots,n-m-1$. So, $t(i) == p$ if and only if $T[i..i+m-1] == P[0..m-1]$.

We can compute p in $O(m)$ time using Horner's Rule as:

$$p = P[m-1] + 10(P[m-2] + 10(P[m-3] + \ldots + 10\,(P[1] + 10\,P[0])\ldots))$$

The code for the above assumption is:

```
value = 0
for i in range (0, m-1):
    value = value * 10
    value = value + P[i]
```

We can compute all $t(i)$, for $i = 0,1,\ldots,n-m-1$ values in a total of $O(n)$ time. The value of $t(0)$ can be similarly computed from $T[0..m-1]$ in $O(m)$ time. To compute the remaining values $t(0), t(1),\ldots, t\,(n-m-1)$, understand that $t(i+1)$ can be computed from $t(i)$ in constant time.

$$t(i+1) = 10 * (t(i) - 10^{m-1} * T[i]) + T[i+m-1]$$

For example, if T = "123456" and $m = 3$

$$t(0) = 123$$
$$t(1) = 10 * (123 - 100 * 1) + 4 = 234$$

Step by Step explanation

First : remove the first digit : $123 - 100 * 1 = 23$
Second: Multiply by 10 to shift it : $23 * 10 = 230$
Third: Add last digit : $230 + 4 = 234$

The algorithm runs by comparing, $t(i)$ with p. When $t(i) == p$, then we have found the substring P in T, starting from position i..

```
def RobinKarp(text, pattern):
    if pattern == None or text == None:
        return -1
    if pattern == "" or text == "":
        return -1
    if len(pattern) > len(text):
        return -1
    hashText = Hash(text, len(pattern))
    hashPattern = Hash(pattern, len(pattern))
    hashPattern.update()
    for i in range(len(text)-len(pattern)+1):
        if hashText.hashedValue() == hashPattern.hashedValue():
            if hashText.text() == pattern:
                return i
        hashText.update()
    return -1
class Hash:
    def __init__(self, text, size):
        self.str  = text
        self.hash = 0
        for i in xrange(0, size):
            self.hash += ord(self.str[i])
        self.init = 0
        self.end  = size
    def update(self):
        if self.end <= len(self.str) -1:
            self.hash -= ord(self.str[self.init])
            self.hash += ord(self.str[self.end])
            self.init += 1
            self.end  += 1
    def hashedValue(self):
        return self.hash
    def text(self):
        return self.str[self.init:self.end]
print RobinKarp("3141592653589793", "26")
```

15.5 String Matching with Finite Automata

In this method we use the finite automata which is the concept of the Theory of Computation (ToC). Before looking at the algorithm, first let us look at the definition of finite automata.

Finite Automata

A finite automaton F is a 5-tuple $(Q, q_0, A, \sum, \delta)$, where

- Q is a finite set of states
- $q_0 \in Q$ is the start state
- $A \subseteq Q$ is a set of accepting states
- $\sum$ is a finite input alphabet
- δ is the transition function that gives the next state for a given current state and input

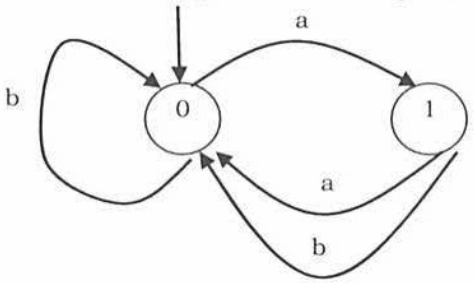
How does Finite Automata Work?

- The finite automaton F begins in state q_0
- Reads characters from $\sum$ one at a time
- If F is in state q and reads input character a, F moves to state $\delta(q,a)$
- At the end, if its state is in A, then we say, F accepted the input string read so far
- If the input string is not accepted it is called the rejected string

Example: Let us assume that $Q = \{0,1\}, q_0 = 0, A = \{1\}, \Sigma = \{a,b\}$. $\delta(q,a)$ as shown in the transition table/diagram. This accepts strings that end in an odd number of *a's*; e.g., *abbaaa* is accepted, *aa* is rejected.

State	Input a	Input b
0	1	0
1	0	0

Transition Function/Table



Important Notes for Constructing the Finite Automata

For building the automata, first we start with the initial state. The FA will be in state k if k characters of the pattern have been matched. If the next text character is equal to the pattern character c, we have matched $k+1$ characters and the FA enters state $k+1$. If the next text character is not equal to the pattern character, then the FA go to a state $0, 1, 2, \ldots,$ *or* k, depending on how many initial pattern characters match the text characters ending with c.

Matching Algorithm

Now, let us concentrate on the matching algorithm.

- For a given pattern $P[0..m-1]$, first we need to build a finite automaton F
  - The state set is $Q = \{0, 1, 2, \ldots, m\}$
  - The start state is 0
  - The only accepting state is m
  - Time to build F can be large if $\sum$ is large
- Scan the text string $T[0..n-1]$ to find all occurrences of the pattern $P[0..m-1]$
- String matching is efficient: $\Theta(n)$
  - Each character is examined exactly once
  - Constant time for each character
  - But the time to compute δ (transition function) is $O(m|\Sigma|)$. This is because δ has $O(m|\Sigma|)$ entries. If we assume $|\Sigma|$ is constant then the complexity becomes $O(m)$.

Algorithm:

```
# Input: Pattern string P[0..m-1], δ and F
# Goal: All valid shifts displayed
def FiniteAutomataStringMatcher(P,m, F, δ):
    q = 0
    for i in range(0,m):
        q = δ(q,T[i])
        if(q == m):
            print("Pattern occurs with shift: ", i-m)
```

Time Complexity: $O(m)$.

15.6 KMP Algorithm

As before, let us assume that T is the string to be searched and P is the pattern to be matched. This algorithm was presented by Knuth, Morris and Pratt. It takes $O(n)$ time complexity for searching a pattern. To get $O(n)$

time complexity, it avoids the comparisons with elements of T that were previously involved in comparison with some element of the pattern P.

The algorithm uses a table and in general we call it *prefix function* or *prefix table* or *fail function* F. First we will see how to fill this table and later how to search for a pattern using this table. The prefix function F for a pattern stores the knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern P. It means that this table can be used for avoiding backtracking on the string TT.

Prefix Table

```
def prefixTable(pattern):
    m = len(pattern)
    F = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and pattern[k] != pattern[q]:
            k = F[k - 1]
        if pattern[k] == pattern[q]:
            k = k + 1
        F[q] = k
    return F
```

As an example, assume that $P = a\ b\ a\ b\ a\ c\ a$. For this pattern, let us follow the step-by-step instructions for filling the prefix table F. Initially: $m = length[P] = 7, F[0] = 0$ and $F[1] = 0$.

Step 1: $i = 1, j = 0, \text{F}[1] = 0$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0					

Step 2: $i = 2, j = 0, \text{F}[2] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1				

Step 3: $i = 3, j = 1, \text{F}[3] = 2$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2			

Step 4: $i = 4, j = 2, \text{F}[4] = 3$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3		

Step 5: $i = 5, j = 3, \text{F}[5] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	

Step 6: $i = 6, j = 1, \text{F}[6] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

At this step the filling of the prefix table is complete.

Matching Algorithm

The KMP algorithm takes pattern P, string T and prefix function F as input, and finds a match of P in T.

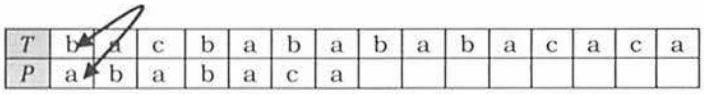
```
def KMP(text, pattern):
    n = len(text)
    m = len(pattern)
    F = prefixTable(pattern)
    q = 0
```

```
    for i in range(n):
        while q > 0 and pattern[q] != text[i]:
            q = F[q - 1]
        if pattern[q] == text[i]:
            q = q + 1

        if q == m:
            return i - m + 1
    return -1

print KMP("bacbababababacaca", "ababaca")
```

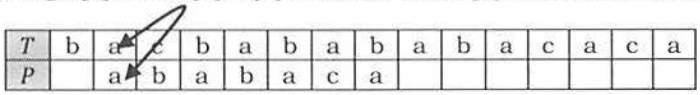
Time Complexity: $O(m+n)$, where m is the length of the pattern and n is the length of the text to be searched. Space Complexity: $O(m)$.

Now, to understand the process let us go through an example. Assume that $T = b\ a\ c\ b\ a\ b\ a\ b\ a\ b\ a\ c\ a\ c\ a$ & $P = a\ b\ a\ b\ a\ c\ a$. Since we have already filled the prefix table, let us use it and go to the matching algorithm. Initially: $n = size\ of\ T = 15$; $m = size\ of\ P = 7$.

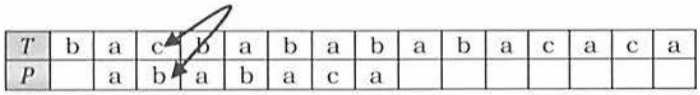
Step 1: $i = 0$, $j = 0$, comparing $P[0]$ with $T[0]$. $P[0]$ does not match with $T[0]$. P will be shifted one position to the right.



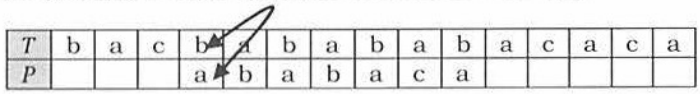
Step 2: $i = 1$, $j = 0$, comparing $P[0]$ with $T[1]$. $P[0]$ matches with $T[1]$. Since there is a match, P is not shifted.



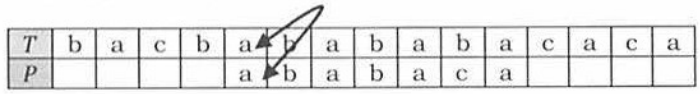
Step 3: $i = 2$, $j = 1$, comparing $P[1]$ with $T[2]$. $P[1]$ does not match with $T[2]$. Backtracking on P, comparing $P[0]$ and $T[2]$.



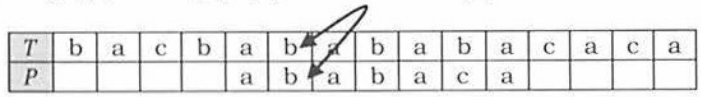
Step 4: $i = 3$, $j = 0$, comparing $P[0]$ with $T[3]$. $P[0]$ does not match with $T[3]$.



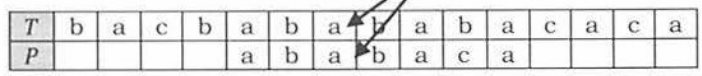
Step 5: $i = 4$, $j = 0$, comparing $P[0]$ with $T[4]$. $P[0]$ matches with $T[4]$.



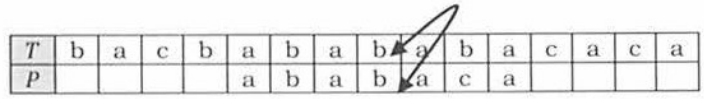
Step 6: $i = 5$, $j = 1$, comparing $P[1]$ with $T[5]$. $P[1]$ matches with T[5].



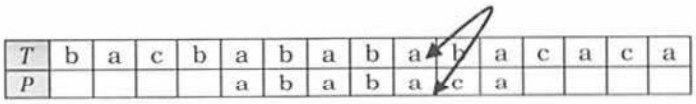
Step 7: $i = 6$, $j = 2$, comparing $P[2]$ with $T[6]$. $P[2]$ matches with $T[6]$.



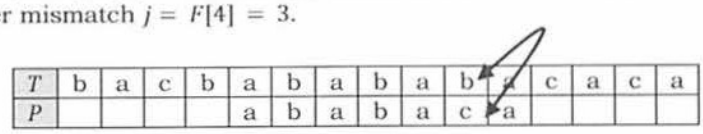
Step 8: $i = 7$, $j = 3$, comparing $P[3]$ with $T[7]$. $P[3]$ matches with $T[7]$.



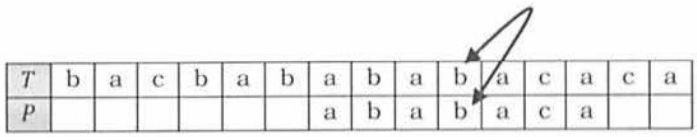
Step 9: $i = 8$, $j = 4$, comparing $P[4]$ with $T[8]$. $P[4]$ matches with $T[8]$.



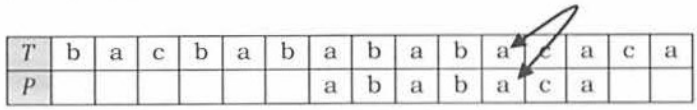
Step 10: $i = 9$, $j = 5$, comparing $P[5]$ with $T[9]$. $P[5]$ does not match with $T[9]$. Backtracking on P, comparing $P[4]$ with $T[9]$ because after mismatch $j = F[4] = 3$.



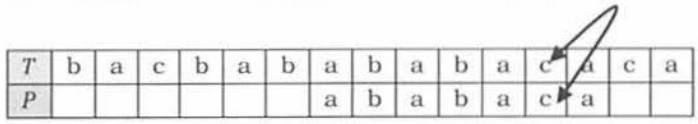
Comparing $P[3]$ with $T[9]$.



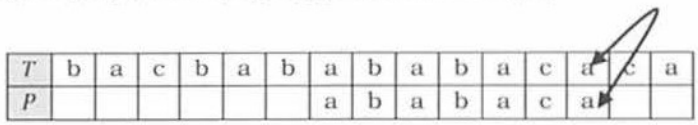
Step 11: $i = 10$, $j = 4$, comparing $P[4]$ with $T[10]$. $P[4]$ matches with $T[10]$.



Step 12: $i = 11$, $j = 5$, comparing $P[5]$ with $T[11]$. $P[5]$ matches with $T[11]$.



Step 13: $i = 12$, $j = 6$, comparing $P[6]$ with $T[12]$. $P[6]$ matches with $T[12]$.



Pattern P has been found to completely occur in string T. The total number of shifts that took place for the match to be found are: $i - m = 13 - 7 = 6$ shifts.

Notes:

- KMP performs the comparisons from left to right
- KMP algorithm needs a preprocessing (prefix function) which takes $O(m)$ space and time complexity
- Searching takes $O(n + m)$ time complexity (does not depend on alphabet size)

15.7 Boyce-Moore Algorithm

Like the KMP algorithm, this also does some pre-processing and we call it *last function*. The algorithm scans the characters of the pattern from right to left beginning with the rightmost character. During the testing of a possible placement of pattern P in T, a mismatch is handled as follows: Let us assume that the current character being matched is $T[i] = c$ and the corresponding pattern character is $P[j]$. If c is not contained anywhere in P, then shift the pattern P completely past $T[i]$. Otherwise, shift P until an occurrence of character c in P gets aligned with $T[i]$. This technique avoids needless comparisons by shifting the pattern relative to the text.

The *last* function takes $O(m + |\Sigma|)$ time and the actual search takes $O(nm)$ time. Therefore the worst case running time of the Boyer-Moore algorithm is $O(nm + |\Sigma|)$. This indicates that the worst-case running time is quadratic, in the case of $n == m$, the same as the brute force algorithm.

- The Boyer-Moore algorithm is very fast on the large alphabet (relative to the length of the pattern).
- For the small alphabet, Boyce-Moore is not preferable.
- For binary strings, the KMP algorithm is recommended.
- For the very shortest patterns, the brute force algorithm is better.

15.8 Data Structures for Storing Strings

If we have a set of strings (for example, all the words in the dictionary) and a word which we want to search in that set, in order to perform the search operation faster, we need an efficient way of storing the strings. To store sets of strings we can use any of the following data structures.

- Hashing Tables
- Binary Search Trees
- Tries
- Ternary Search Trees

15.9 Hash Tables for Strings

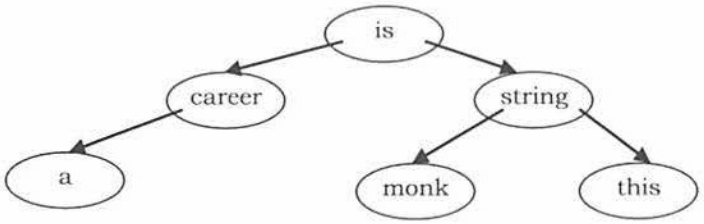
As seen in the *Hashing* chapter, we can use hash tables for storing the integers or strings. In this case, the keys are nothing but the strings. The problem with hash table implementation is that we lose the ordering information – after applying the hash function, we do not know where it will map to. As a result, some queries take more time. For example, to find all the words starting with the letter *"K"*, with hash table representation we need to scan the complete hash table. This is because the hash function takes the complete key, performs hash on it, and we do not know the location of each word.

15.10 Binary Search Trees for Strings

In this representation, every node is used for sorting the strings alphabetically. This is possible because the strings have a natural ordering: A comes before B, which comes before C, and so on. This is because words can be ordered and we can use a Binary Search Tree (BST) to store and retrieve them. For example, let us assume that we want to store the following strings using BSTs:

this is a career monk string

For the given string there are many ways of representing them in BST. One such possibility is shown in the tree below.



Issues with Binary Search Tree Representation

This method is good in terms of storage efficiency. But the disadvantage of this representation is that, at every node, the search operation performs the complete match of the given key with the node data, and as a result the time complexity of the search operation increases. So, from this we can say that BST representation of strings is good in terms of storage but not in terms of time.

15.11 Tries

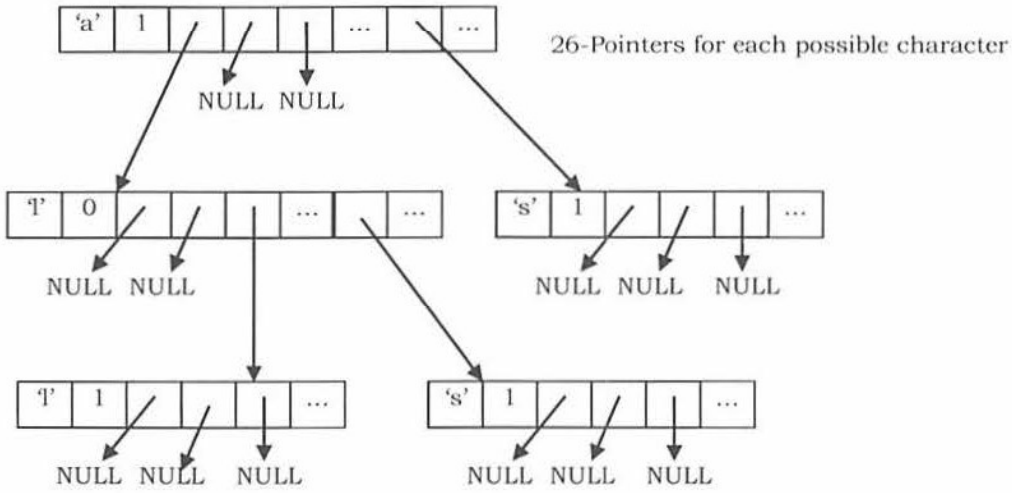
Now, let us see the alternative representation that reduces the time complexity of the search operation. The name *trie* is taken from the word re"trie".

What is a Trie?

A *trie* is a tree and each node in it contains the number of pointers equal to the number of characters of the alphabet. For example, if we assume that all the strings are formed with English alphabet characters *"a"* to *"z"* then each node of the trie contains 26 pointers. A trie data structure can be declared as:

```
class Node(object):
    def __init__(self):
        self.children={}#contains a map with child characters as keys and their Node as values
```

Suppose we want to store the strings *"a","all","als"*, and *"as""*: *trie* for these strings will look like:



Why Tries?

The tries can insert and find strings in O(*L*) time (where *L* represents the length of a single word). This is much faster than hash table and binary search tree representations.

Trie Declaration

The structure of the TrieNode has data (char), is_End_Of_String (boolean), and has a collection of child nodes (Collection of TrieNodes). It also has one more method called subNode(char). This method takes a character as argument and will return the child node of that character type if that is present. The basic element - TrieNode of a TRIE data structure looks like this:

```
class Node(object):
    def __init__(self):
        self.children={}#contains a map with child characters as keys and their Node as values
class Trie(object):
    def __init__(self):
        self.root = Node()
        self.root.data = "/"
```

Now that we have defined our TrieNode, let's go ahead and look at the other operations of TRIE. Fortunately, the TRIE data structure is simple to implement since it has two major methods: insert() and search(). Let's look at the elementary implementation of both these methods.

Inserting a String in Trie

To insert a string, we just need to start at the root node and follow the corresponding path (path from root indicates the prefix of the given string). Once we reach the NULL pointer, we just need to create a skew of tail nodes for the remaining characters of the given string.

```
def addWord(self,word):
    currentNode = self.root
    i = 0
    #print "adding word '"+ word+"' to trie "
    for c in word:
        #print "adding character " + c
        try:
            currentNode = currentNode.children[c]
                #print "character "+c + " exists"
        except:
            self.createSubTree(word[i:len(word)],currentNode)
            break
        i = i + 1
```

Time Complexity: O(*L*), where *L* is the length of the string to be inserted.

Note: For real dictionary implementation, we may need a few more checks such as checking whether the given string is already there in the dictionary or not.

Searching a String in Trie

The same is the case with the search operation: we just need to start at the root and follow the pointers. The time complexity of the search operation is equal to the length of the given string that want to search.

```
def getWordList(self,startingCharacters):
    startNode = self.root
    for c in startingCharacters:
        try:
            startNode = startNode.children[c]
        except:
            return []
    nodestack=[]
    for child in startNode.children:
        nodestack.append(startNode.children[child])
    words=[]
    currentWord=""
    while len(nodestack) != 0:
        currentNode = nodestack.pop()
        currentWord += currentNode.data
        if len (currentNode.children) == 0:
            words.append(startingCharacters+currentWord)
            currentWord = ""
        for n in currentNode.children:
            temp = currentNode.children[n]
            nodestack.append(temp)
    return words
```

Time Complexity: $O(L)$, where L is the length of the string to be searched.

Issues with Tries Representation

The main disadvantage of tries is that they need lot of memory for storing the strings. As we have seen above, for each node we have too many node pointers. In many cases, the occupancy of each node is less. The final conclusion regarding tries data structure is that they are faster but require huge memory for storing the strings.

Note: There are some improved tries representations called *trie compression techniques*. But, even with those techniques we can reduce the memory only at the leaves and not at the internal nodes.

15.12 Ternary Search Trees

This representation was initially provided by Jon Bentley and Sedgewick. A ternary search tree takes the advantages of binary search trees and tries. That means it combines the memory efficiency of BSTs and the time efficiency of tries.

Ternary Search Trees Declaration

```
class TSTNode:
    def __init__ (self, x):
        self.data = x
        self.left = None
        self.eq = None
        self.right = None
```

The Ternary Search Tree (TST) uses three pointers:

- The *left* pointer points to the TST containing all the strings which are alphabetically less than *data*.
- The *right* pointer points to the TST containing all the strings which are alphabetically greater than *data*.
- The *eq* pointer points to the TST containing all the strings which are alphabetically equal to *data*. That means, if we want to search for a string, and if the current character of the input string and the *data* of current node in TST are the same, then we need to proceed to the next character in the input string and search it in the subtree which is pointed by *eq*.

Operation Method of TST

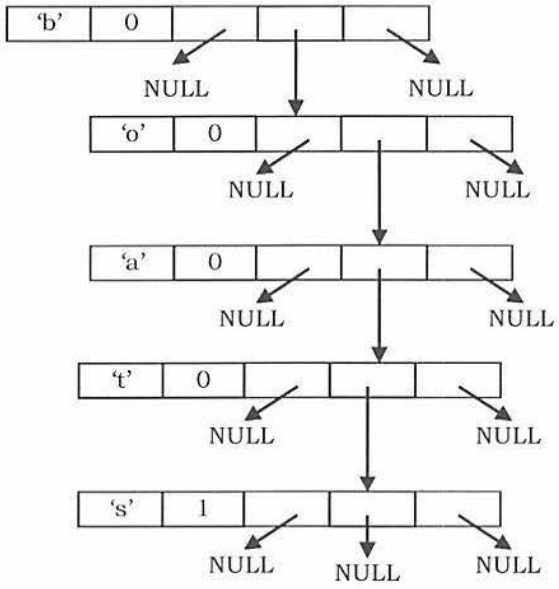
Let's make the operation method of class TST.

```
class TST:
    def __init__ (self, x = None):
        self.root = Node (None) # header
        self.leaf = x
```

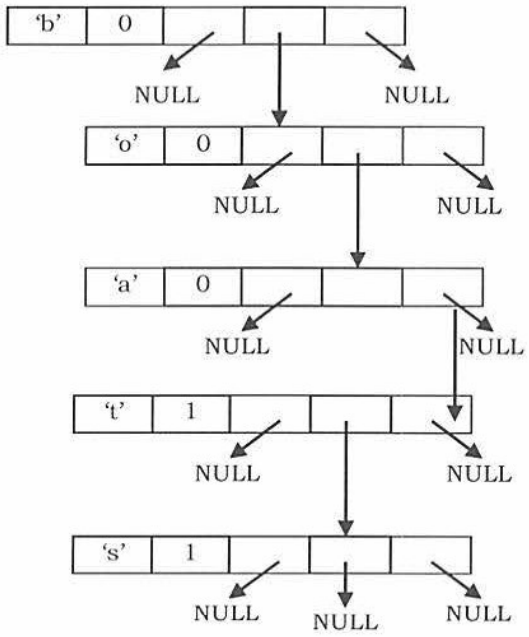
The instance variable root of TST will store the header. Data in this section is a dummy. The actual data will continue to add to the root of the child. Instance variable *leaf* stores the data representing the termination. *leaf* is passed as an argument when calling the TST. It will be None if it is omitted.

Inserting strings in Ternary Search Tree

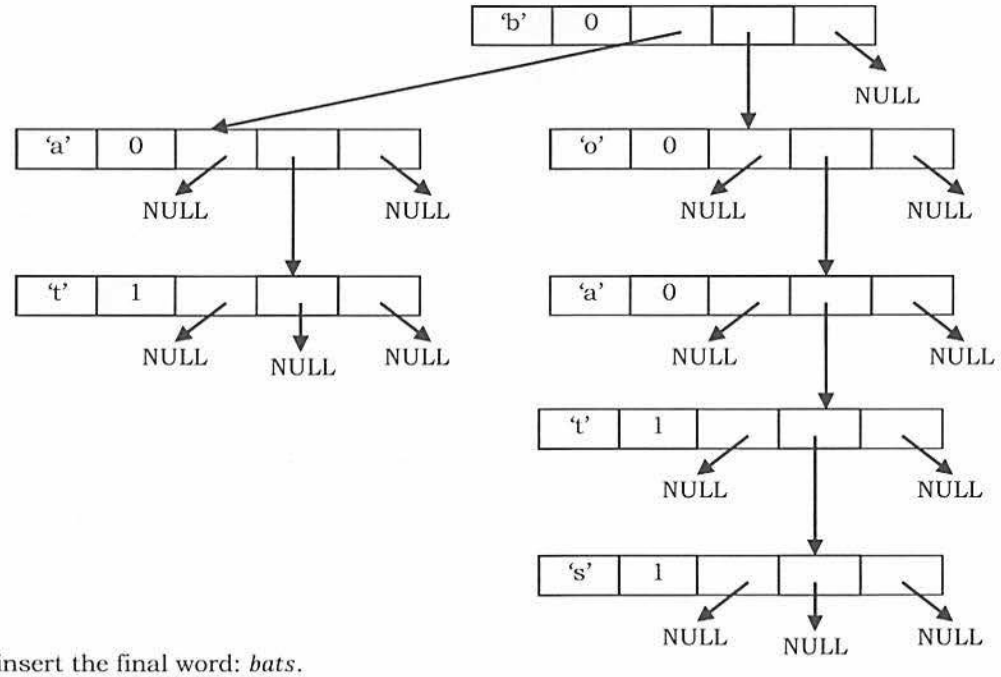
For simplicity let us assume that we want to store the following words in TST (also assume the same order): *boats, boat, bat* and *bats*. Initially, let us start with the *boats* string.



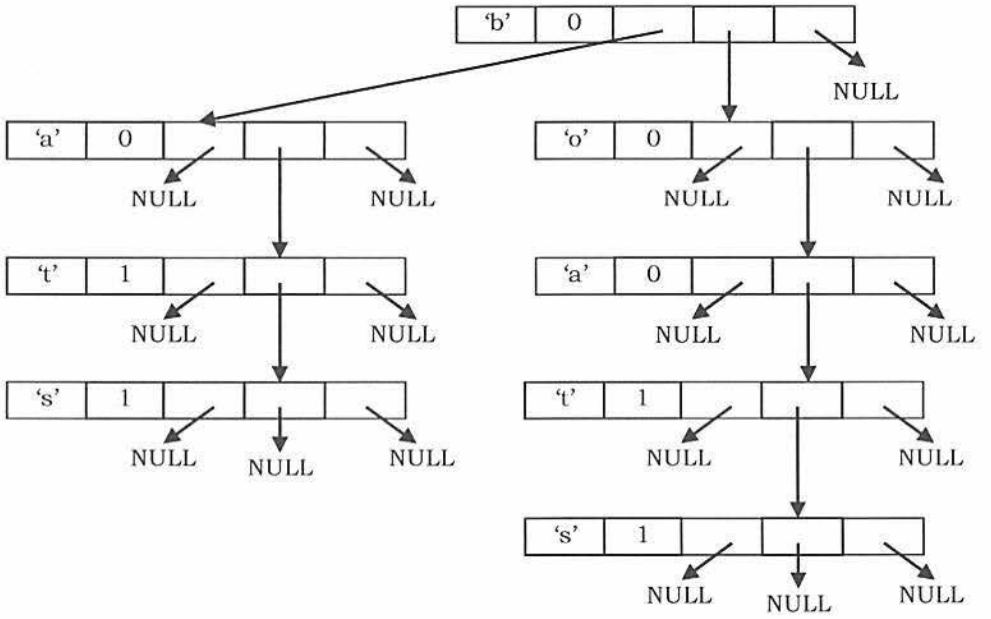
Now if we want to insert the string *boat*, then the TST becomes [the only change is setting the *is_End_Of_String* flag of "*t*" node to 1]:



Now, let us insert the next string: *bat*



Now, let us insert the final word: *bats*.



Based on these examples, we can write the insertion algorithm as below. We will combine the insertion operation of BST and tries.

```
# Insert
def _insert (node, x):
    if node is None: return x
    elif x.data == node.data: return node
    elif x.data <node.data:
        node.left = _insert (node.left, x)
    else:
        node.right = _insert (node.right, x)
    return node
class TST:
    def __init__ (self, x = None):
```

```
        self.root = TSTNode (None) # header
        self.leaf = x

    # Insert
    def insert (self, seq):
        node = self.root
        for x in seq:
            child = _search (node.eq, x)
            if not child:
                child = TSTNode (x)
                node.eq = _insert (node.eq, child)
            node = child
        # Check leaf
        if not _search (node.eq, self.leaf):
            node.eq = _insert (node.eq, TSTNode (self.leaf))
```

Time Complexity: $O(L)$, where L is the length of the string to be inserted.

Searching in Ternary Search Tree

If after inserting the words we want to search for them, then we have to follow the same rules as that of binary search. The only difference is, in case of match we should check for the remaining characters (in *eq* subtree) instead of return. Also, like BSTs we will see both recursive and non-recursive versions of the search method.

```
# Search
def _search (node, x):
    while node:
        if node.data == x: return node
        if x <node.data:
            node = node.left
        else:
            node = node.right
    return None

class TST:
# Search
    def _search (node, x):
        while node:
            if node.data == x: return node
            if x <node.data:
                node = node.left
            else:
                node = node.right
        return None
```

Time Complexity: $O(L)$, where L is the length of the string to be searched.

Displaying All Words of Ternary Search Tree

If we want to print all the strings of TST we can use the following algorithm. If we want to print them in sorted order, we need to follow the inorder traversal of TST.

```
# Traverse
def _traverse (node, leaf):
    if node:
        for x in _traverse (node.left, leaf):
            yield x
        if node.data == leaf:
            yield []
        else:
            for x in _traverse (node.eq, leaf):
                yield [node.data] + x
        for x in _traverse (node.right, leaf):
            yield x

class TST:
    def __init__ (self, x = None):
        self.root = TSTNode (None) # header
```

```
            self.leaf = x
        # Traverse
        def traverse (self):
            for x in _traverse (self.root.eq, self.leaf):
                yield x
```

Full Implementation

```
class TSTNode:
    def __init__ (self, x):
        self.data = x
        self.left = None
        self.eq = None
        self.right = None
def _search (node, x):
    while node:
        if node.data == x: return node
        if x <node.data:
            node = node.left
        else:
            node = node.right
    return None
def _insert (node, x):
    if node is None: return x
    elif x.data == node.data: return node
    elif x.data <node.data:
        node.left = _insert (node.left, x)
    else:
        node.right = _insert (node.right, x)
    return node
# Find the minimum value
def _searchMin (node):
    if node.left is None: return node.data
    return _searchMin (node.left)
# Delete the minimum value
def _deleteMin (node):
    if node.left is None: return node.right
    node.left = _deleteMin (node.left)
    return node
def _delete (node, x):
    if node:
        if x == node.data:
            if node.left is None:
                return node.right
            elif node.right is None:
                return node.left
            else:
                node.data = _searchMin (node.right)
                node.right = _deleteMin (node.right)
        elif x <node.data:
            node.left = _delete (node.left, x)
        else:
            node.right = _delete (node.right, x)
    return node
def _traverse (node, leaf):
    if node:
        for x in _traverse (node.left, leaf):
            yield x
        if node.data == leaf:
            yield []
        else:
```

```
            for x in _traverse (node.eq, leaf):
                yield [node.data] + x
        for x in _traverse (node.right, leaf):
            yield x

##### Ternary Search Tree #####
class TST:
    def __init__ (self, x = None):
        self.root = TSTNode (None) # header
        self.leaf = x

    def search (self, seq):
        node = self.root
        for x in seq:
            node = _search (node.eq, x)
            if not node: return False
        # Check leaf
        return _search (node.eq, self.leaf) is not None

    def insert (self, seq):
        node = self.root
        for x in seq:
            child = _search (node.eq, x)
            if not child:
                child = TSTNode (x)
                node.eq = _insert (node.eq, child)
            node = child
        # Check leaf
        if not _search (node.eq, self.leaf):
            node.eq = _insert (node.eq, TSTNode (self.leaf))

    def delete (self, seq):
        node = self.root
        for x in seq:
            node = _search (node.eq, x)
            if not node: return False
        # Delete leaf
        if _search (node.eq, self.leaf):
            node.eq = _delete (node.eq, self.leaf)
            return True
        return False

    def traverse (self):
        for x in _traverse (self.root.eq, self.leaf):
            yield x

    # The data with a common prefix
    def commonPrefix (self, seq):
        node = self.root
        buff = []
        for x in seq:
            buff.append (x)
            node = _search (node.eq, x)
            if not node: return
        for x in _traverse (node.eq, self.leaf):
            yield buff + x

if __name__ == '__main__':
    # Suffix trie
    def makeTST (seq):
        a = TST ()
        for x in xrange (len (seq)):
            a.insert (seq [x:])
        return a

    s = makeTST ('abcabbca')
    for x in s.traverse ():
        print x
```

```
for x in ['a', 'bc']:
    print x
    for y in s.commonPrefix (x):
        print y
print s.delete ('a')
print s.delete ('ca')
print s.delete ('bca')
for x in s.traverse ():
    print x
s = makeTST ([0,1,2,0,1,1,2,0])
for x in s.traverse ():
    print x
```

15.13 Comparing BSTs, Tries and TSTs

- Hash table and BST implementation stores complete the string at each node. As a result they take more time for searching. But they are memory efficient.
- TSTs can grow and shrink dynamically but hash tables resize only based on load factor.
- TSTs allow partial search whereas BSTs and hash tables do not support it.
- TSTs can display the words in sorted order, but in hash tables we cannot get the sorted order.
- Tries perform search operations very fast but they take huge memory for storing the string.
- TSTs combine the advantages of BSTs and Tries. That means they combine the memory efficiency of BSTs and the time efficiency of tries

15.14 Suffix Trees

Suffix trees are an important data structure for strings. With suffix trees we can answer the queries very fast. But this requires some preprocessing and construction of a suffix tree. Even though the construction of a suffix tree is complicated, it solves many other string-related problems in linear time.

Note: Suffix trees use a tree (suffix tree) for one string, whereas Hash tables, BSTs, Tries and TSTs store a set of strings. That means, a suffix tree answers the queries related to one string.

Let us see the terminology we use for this representation.

Prefix and Suffix

Given a string $T = T_1T_2 \ldots T_n$, the *prefix* of T is a string $T_1 \ldots T_i$ where i can take values from 1 to n. For example, if $T = banana$, then the prefixes of T are: $b, ba, ban, bana, banan, banana$.

Similarly, given a string $T = T_1T_2 \ldots T_n$, the *suffix* of T is a string $T_i \ldots T_n$ where i can take values from n to 1. For example, if $T = banana$, then the suffixes of T are: $a, na, ana, nana, anana, banana$.

Observation

From the above example, we can easily see that for a given text T and pattern P, the exact string matching problem can also be defined as:

- Find a suffix of T such that P is a prefix of this suffix *or*
- Find a prefix of T such that P is a suffix of this prefix.

Example: Let the text to be searched be $T = accbkkbac$ and the pattern be $P = kkb$. For this example, P is a prefix of the suffix $kkbac$ and also a suffix of the prefix $accbkkb$.

What is a Suffix Tree?

In simple terms, the suffix tree for text T is a Trie-like data structure that represents the suffixes of T. The definition of suffix trees can be given as: A suffix tree for a n character string $T[1 \ldots n]$ is a rooted tree with the following properties.

- A suffix tree will contain n leaves which are numbered from 1 to n
- Each internal node (except root) should have at least 2 children
- Each edge in a tree is labeled by a nonempty substring of T
- No two edges of a node (children edges) begin with the same character
- The paths from the root to the leaves represent all the suffixes of T

The Construction of Suffix Trees

Algorithm

1. Let S be the set of all suffixes of T. Append \$ to each of the suffixes.
2. Sort the suffixes in S based on their first character.
3. For each group S_c ($c \in \sum$):
   (i) If S_c group has only one element, then create a leaf node.
   (ii) Otherwise, find the longest common prefix of the suffixes in S_c group, create an internal node, and recursively continue with Step 2, S being the set of remaining suffixes from S_c after splitting off the longest common prefix.

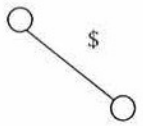
For better understanding, let us go through an example. Let the given text be $T = tatat$. For this string, give a number to each of the suffixes.

Index	Suffix
1	\$
2	t\$
3	at\$
4	tat\$
5	$atat$\$
6	$tatat$\$

Now, sort the suffixes based on their initial characters.

Index	**Suffix**	
1	\$	Group S_1 based on a
3	at\$	Group S_2 based on a
5	$atat$\$	
2	t\$	Group S_3 based on t
4	tat\$	
6	$tatat$\$	

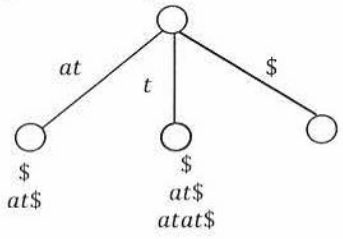
In the three groups, the first group has only one element. So, as per the algorithm, create a leaf node for it, as shown below.



Now, for S_2 and S_3 (as they have more than one element), let us find the longest prefix in the group, and the result is shown below.

Group	Indexes for this group	Longest Prefix of Group Suffixes
S_2	3,5	at
S_3	2,4,6	t

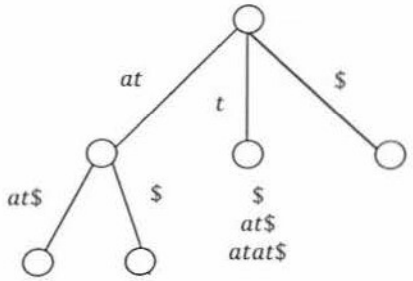
For S_2 and S_3, create internal nodes, and the edge contains the longest common prefix of those groups.



Now we have to remove the longest common prefix from the S_2 and S_3 group elements.

Group	Indexes for this group	Longest Prefix of Group Suffixes	Resultant Suffixes
S_2	3,5	at	\$, at\$
S_3	2,4,6	t	\$, at\$, $atat$\$

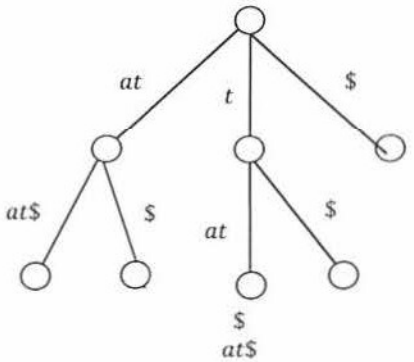
Out next step is solving S_2 and S_3 recursively. First let us take S_2. In this group, if we sort them based on their first character, it is easy to see that the first group contains only one element \$, and the second group also contains only one element, at\$. Since both groups have only one element, we can directly create leaf nodes for them.



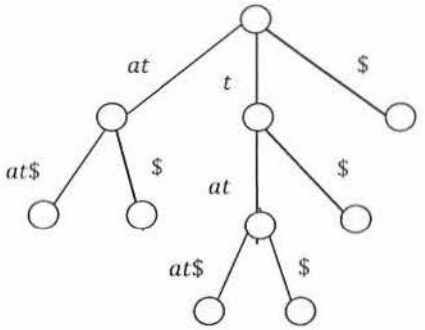
At this step, both S_1 and S_2 elements are done and the only remaining group is S_3. As similar to earlier steps, in the S_3 group, if we sort them based on their first character, it is easy to see that there is only one element in the first group and it is \$. For S_3 remaining elements, remove the longest common prefix.

Group	Indexes for this group	Longest Prefix of Group Suffixes	Resultant Suffixes
S_3	4, 6	*at*	\$, *at*\$

In the S_3 second group, there are two elements: \$ and *at*\$. We can directly add the leaf nodes for the first group element \$. Let us add S_3 subtree as shown below.



Now, S_3 contains two elements. If we sort them based on their first character, it is easy to see that there are only two elements and among them one is \$ and other is *at*\$. We can directly add the leaf nodes for them. Let us add S_3 subtree as shown below.



Since there are no more elements, this is the completion of the construction of the suffix tree for string $T = tatat$. The time-complexity of the construction of a suffix tree using the above algorithm is $O(n^2)$ where n is the length of the input string because there are n distinct suffixes. The longest has length n, the second longest has length $n-1$, and so on.

Note:

- There are $O(n)$ algorithms for constructing suffix trees.
- To improve the complexity, we can use indices instead of string for branches.

Applications of Suffix Trees

All the problems below (but not limited to these) on strings can be solved with suffix trees very efficiently (for algorithms refer to *Problems* section).

- **Exact String Matching:** Given a text T and a pattern P, how do we check whether P appears in T or not?
- **Longest Repeated Substring:** Given a text T how do we find the substring of T that is the maximum repeated substring?

- **Longest Palindrome:** Given a text T how do we find the substring of T that is the longest palindrome of T?
- **Longest Common Substring:** Given two strings, how do we find the longest common substring?
- **Longest Common Prefix:** Given two strings $X[i \ldots n]$ and $Y[j \ldots m]$, how do we find the longest common prefix?
- How do we search for a regular expression in given text T?
- Given a text T and a pattern P, how do we find the first occurrence of P in T?

15.15 String Algorithms: Problems & Solutions

Problem-1 Given a paragraph of words, give an algorithm for finding the word which appears the maximum number of times. If the paragraph is scrolled down (some words disappear from the first frame, some words still appear, and some are new words), give the maximum occurring word. Thus, it should be dynamic.

Solution: For this problem we can use a combination of priority queues and tries. We start by creating a trie in which we insert a word as it appears, and at every leaf of trie. Its node contains that word along with a pointer that points to the node in the heap [priority queue] which we also create. This heap contains nodes whose structure contains a *counter*. This is its frequency and also a pointer to that leaf of trie, which contains that word so that there is no need to store the word twice.

Whenever a new word comes up, we find it in trie. If it is already there, we increase the frequency of that node in the heap corresponding to that word, and we call it heapify. This is done so that at any point of time we can get the word of maximum frequency. While scrolling, when a word goes out of scope, we decrement the counter in heap. If the new frequency is still greater than zero, heapify the heap to incorporate the modification. If the new frequency is zero, delete the node from heap and delete it from trie.

Problem-2 Given two strings, how can we find the longest common substring?

Solution: Let us assume that the given two strings are T_1 and T_2. The longest common substring of two strings, T_1 and T_2, can be found by building a generalized suffix tree for T_1 and T_2. That means we need to build a single suffix tree for both the strings. Each node is marked to indicate if it represents a suffix of T_1 or T_2 or both. This indicates that we need to use different marker symbols for both the strings (for example, we can use \$ for the first string and # for the second symbol). After constructing the common suffix tree, the deepest node marked for both T_1 and T_2 represents the longest common substring.

Another way of doing this is: We can build a suffix tree for the string $T_1\$T_2\#$. This is equivalent to building a common suffix tree for both the strings.

Time Complexity: $O(m + n)$, where m and n are the lengths of input strings T_1 and T_2.

Problem-3 **Longest Palindrome:** Given a text T how do we find the substring of T which is the longest palindrome of T?

Solution: The longest palindrome of $T[1..n]$ can be found in $O(n)$ time. The algorithm is: first build a suffix tree for $T\$reverse(T)\#$ or build a generalized suffix tree for T and $reverse(T)$. After building the suffix tree, find the deepest node marked with both \$ and #. Basically it means find the longest common substring.

Problem-4 Given a string (word), give an algorithm for finding the next word in the dictionary.

Solution: Let us assume that we are using Trie for storing the dictionary words. To find the next word in Tries we can follow a simple approach as shown below. Starting from the rightmost character, increment the characters one by one. Once we reach Z, move to the next character on the left side. Whenever we increment, check if the word with the incremented character exists in the dictionary or not. If it exists, then return the word, otherwise increment again. If we use TST, then we can find the inorder successor for the current word.

Problem-5 Give an algorithm for reversing a string.

Solution:

```
# If the str is editable
def ReversingString(str):
    s = list(str)
    end = len(str)-1
    start = 0
    while (start<end):
        temp = s[start]
        s[start] = s[end]
        s[end] = temp
        start += 1
        end -= 1
    return "".join(s)
```

```
str = "CareerMonk Publications."
print ReversingString(str)

# Alternative Implementation
def reverse(str):
    r = ""
    for c in str:
      r = c + r
      return r
str = "CareerMonk Publications."
print reverse(str)
```

Time Complexity: $O(n)$, where n is the length of the given string. Space Complexity: $O(n)$.

Problem-6 Can we reverse the string without using any temporary variable?

Solution: Yes, we can use XOR logic for swapping the variables.

```
def ReversingString(str):
    s = list(str)
    end = len(str)-1
    start = 0
    while (start<end):
        s[start], s[end] = s[end], s[start]
        start += 1
        end -= 1
    return "".join(s)
str = "CareerMonk Publications."
print ReversingString(str)

# Alternative Implementation
str = "CareerMonk Publications."
print "".join(str[c] for c in xrange(len(str) - 1, -1, -1))
```

Probably the easiest and close to the fastest way to reverse a string is to use Python's extended slice syntax. This allows you to specify a start, stop and step value to use when creating a slice. The syntax is: [start:stop:step].

```
str = "CareerMonk Publications."
print str[::-1]
```

If start is omitted it defaults to 0 and if stop is omitted it defaults to the length of the string. A step of -1 tells Python to start counting by 1 from the stop until it reaches the start.

When working with large strings, or when you just don't want to reverse the whole string at once, you can use the reversed() built-in. reversed() returns an iterator and is arguably the most Pythonic way to reverse a string.

```
str = "CareerMonk Publications."
print "".join(reversed(str))
```

Time Complexity: $O\left(\frac{n}{2}\right) \approx O(n)$, where n is the length of the given string. Space Complexity: $O(1)$.

Problem-7 a text and a pattern, give an algorithm for matching the pattern in the text. Assume ? (single character matcher) and * (multi character matcher) are the wild card characters.

Solution: Brute Force Method. For efficient method, refer to the theory section.

```
def wildcardMatch(inputString, pattern):
    if len(pattern) == 0:
        return len(inputString) == 0
    # inputString can be empty
    if pattern[0] == '?':
        return len(inputString) > 0 and wildcardMatch(inputString[1:], pattern[1:])
    elif pattern[0] == '*':
        # match nothing or
        # match one and continue, AB* = A*
        return wildcardMatch(inputString, pattern[1:]) or \
            (len(inputString) > 0 and wildcardMatch(inputString[1:], pattern))
    else:
        return len(inputString) > 0 and inputString[0] == pattern[0] and \
                                    wildcardMatch(inputString[1:], pattern[1:])
    return 0
```

```
print wildcardMatch("cc","c")
print wildcardMatch("cc","cc")
print wildcardMatch("ccc","cc")
print wildcardMatch("cc", "*")
print wildcardMatch("cc", "a*")
print wildcardMatch("ab", "?*")
print wildcardMatch("cca", "c*a*b")
```

Time Complexity: $O(mn)$, where m is the length of the text and n is the length of the pattern.
Space Complexity: $O(1)$.

Problem-8 Give an algorithm for reversing words in a sentence.
Example: Input: "This is a Career Monk String", Output: "String Monk Career a is This"

Solution: Start from the beginning and keep on reversing the words. The below implementation assumes that ' ' (space) is the delimiter for words in given sentence.

```
# @param s, a string
# @return a string
def reverseWordsInSentence(self, s):
    result = []
    inWord = False
    for i in range(0, len(s)):
        if (s[i]==' ' or s[i]=='\t') and inWord:
            inWord = False
            result.insert(0, s[start:i])
            result.insert(0, ' ')
        elif not (s[i]==' ' or s[i]=='\t' or inWord):
            inWord = True
            start = i
    if inWord:
        result.insert(0, s[start:len(s)])
        result.insert(0, ' ')
    if len(result)>0:
        result.pop(0)
    return ''.join(result)
```

Time Complexity: $O(2n) \approx O(n)$, where n is the length of the string. Space Complexity: $O(1)$.

Problem-9 **Permutations of a string [anagrams]:** Give an algorithm for printing all possible permutations of the characters in a string. Unlike combinations, two permutations are considered distinct if they contain the same characters but in a different order. For simplicity assume that each occurrence of a repeated character is a distinct character. That is, if the input is "aaa", the output should be six repetitions of "aaa". The permutations may be output in any order.

Solution: The solution is reached by generating n! strings, each of length n, where n is the length of the input string. A generator function that generates all permutations of the input elements. If the input contains duplicates, then some permutations may be visited with multiplicity greater than one.

Our recursive algorithm requires two pieces of information, the elements that have not yet been permuted and the partial permutation built up so far. We thus phrase this function as a wrapper around a recursive function with extra parameters.

```
def permutations(elems):
    for perm in reccursivePermutations(elems, []):
        print perm
```

A helper function to recursively generate permutations. The function takes in two arguments, the elements to permute and the partial permutation created so far, and then produces all permutations that start with the given sequence and end with some permutations of the unpermuted elements.

```
def reccursivePermutations(elems, soFar):
    # Base case: If there are no more elements to permute, then the answer will
    # be the permutation we have created so far.
    if len(elems) == 0:
        yield soFar

    # Otherwise, try extending the permutation we have so far by each of the
    # elements we have yet to permute.
    else:
```

```
        for i in range(0, len(elems)):
            # Extend the current permutation by the ith element, then remove
            # the ith element from the set of elements we have not yet
            # permuted. We then iterate across all the permutations that have
            # been generated this way and hand each one back to the caller.
            for perm in reccursivePermutations(elems[0:i] + elems[i+1:], soFar + [elems[i]]):
                yield perm
    # Permutations by iteration
    def permutationByIteration(elems):
        level=[elems[0]]
        for i in range(1,len(elems)):
            nList=[]
            for item in level:
                nList.append(item+elems[i])
                for j in range(len(item)):
                    nList.append(item[0:j]+elems[i]+item[j:])
            level=nList
        return nList
```

Problem-10 **Combinations Combinations of a String:** Unlike permutations, two combinations are considered to be the same if they contain the same characters, but may be in a different order. Give an algorithm that prints all possible combinations of the characters in a string. For example, "*ac*" and "*ab*" are different combinations from the input string "*abc*", but "*ab*" is the same as "*ba*".

Solution: The solution is achieved by generating $n!/r!\,(n-r)!$ strings, each of length between 1 and n where n is the length of the given input string.

Algorithm:

For each of the input characters

a. Put the current character in output string and print it.
b. If there are any remaining characters, generate combinations with those remaining characters.

```
def combinationByRecursion(elems, s, idx, li):
    for i in range(idx, len(elems)):
        s+=elems[i]
        li.append(s)
        #print s, idx
        combinationByRecursion(elems, s, i+1, li)
        s=s[0:-1]
def combinationByIteration(elems):
    level=['']
    for i in range(len(elems)):
        nList=[]
        for item in level:
            nList.append(item+elems[i])
        level+=nList
    return level[1:]
res=[]
combinationByRecursion('abc', '', 0, res)
print combinationByIteration('abc')
print combinationByIteration('abc')
```

Problem-11 Given a string "ABCCBCBA", give an algorithm for recursively removing the adjacent characters if they are the same. For example, ABCCBCBA --> ABBCBA-->ACBA

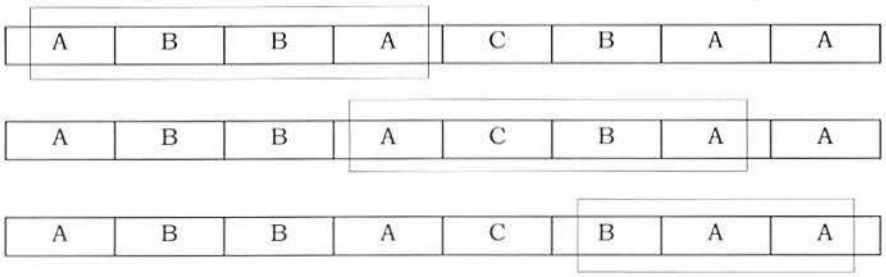
Solution: First we need to check if we have a character pair; if yes, then cancel it. Now check for next character and previous element. Keep canceling the characters until we either reach the start of the array, reach the end of the array, or don't find a pair.

```
def removeAdjacentRepeats(nums):
  i = 1
  while i < len(nums):
    if nums[i] == nums[i-1]:
      nums.pop(i)
      i -= 1
    i += 1
```

```
    return nums
nums=["A","B","C","C","C","C","B","A"]
print removeAdjacent(nums)
```

Problem-12 Given a set of characters *CHARS* and a input string *INPUT*, find the minimum window in *str* which will contain all the characters in *CHARS* in complexity O(*n*). For example, *INPUT* = *ABBACBAA* and *CHARS* = *AAB* has the minimum window *BAA*.

Solution: This algorithm is based on the sliding window approach. In this approach, we start from the beginning of the array and move to the right. As soon as we have a window which has all the required elements, try sliding the window as far right as possible with all the required elements. If the current window length is less than the minimum length found until now, update the minimum length. For example, if the input array is *ABBACBAA* and the minimum window should cover characters *AAB*, then the sliding window will move like this:



Algorithm: The input is the given array and chars is the array of characters that need to be found.

1. Make an integer array shouldfind[] of len 256. The i^{th} element of this array will have the count of how many times we need to find the element of ASCII value *i*.
2. Make another array hasfound of 256 elements, which will have the count of the required elements found until now.
3. Count <= 0
4. While input[i]
   a. If input[i] element is not to be found→ continue
   b. If input[i] element is required => increase count by 1.
   c. If count is length of chars[] array, slide the window as much right as possible.
   d. If current window length is less than min length found until now, update min length.

```
from collections import defaultdict
def smallestWindow(INPUT, CHARS):
    assert CHARS != ''
    disctionary = defaultdict(int)
    nneg = [0]  # number of negative entries in dictionary
    def incr(c):
        disctionary[c] += 1
        if disctionary[c] == 0:
            nneg[0] -= 1
    def decr(c):
        if disctionary[c] == 0:
            nneg[0] += 1
        disctionary[c] -= 1
    for c in CHARS:
        decr(c)
    minLength = len(INPUT) + 1
    j = 0
    for i in xrange(len(INPUT)):
        while nneg[0] > 0:
            if j >= len(INPUT):
                return minLength
            incr(INPUT[j])
            j += 1
        minLength = min(minLength, j - i)
        decr(INPUT[i])
    return minLength
print  smallestWindow("ADOBECODEBANC","ABC")
```

Complexity: If we walk through the code, *i* and *j* can traverse at most *n* steps (where *n* is the input size) in the worst case, adding to a total of 2*n* times. Therefore, time complexity is O(*n*).

Problem-13 Given two strings *str*1 and *str*2, write a function that prints all interleavings of the given two strings. We may assume that all characters in both strings are different. Example: Input: *str*1 = "AB", *str*2 = "CD" and Output: ABCD ACBD ACDB CABD CADB CDAB. An interleaved string of given two strings preserves the order of characters in individual strings. For example, in all the interleaving's of above first example, 'A' comes before 'B' and 'C' comes before 'D'.

Solution: Let the length of *str*1 be m and the length of *str*2 be n. Let us assume that all characters in *str*1 and *str*2 are different. Let Count(m, n) be the count of all interleaved strings in such strings. The value of Count(m, n) can be written as following.

```
Count(m, n) = Count(m-1, n) + Count(m, n-1)
Count(1, 0) = 1 and Count(1, 0) = 1
```

To print all interleaving's, we can first fix the first character of str1[0..m-1] in output string, and recursively call for str1[1..m-1] and str2[0..n-1]. And then we can fix the first character of str2[0..n-1] and recursively call for str1[0..m-1] and str2[1..n-1].

On other words, this problem can be reduced to that of creating all unique permutations of a particular list. Say m and n are the lengths of the strings str1 and str2, respectively. Then construct a list like this:

```
[0] * str1 + [1] * str2
```

There exists a one-to-one correspondence (a bijection) from the unique permutations of this list to all the possible interleavings of the two strings str1 and str2. The idea is to let each value of the permutation specify which string to take the next character from.

```
def PrintInterleavings(str1, str2):
    perms = []
    if len(str1) + len(str2) == 1:
        return [str1 or str2]
    if str1:
        for item in PrintInterleavings(str1[1:], str2):
            perms.append(str1[0] + item)
    if str2:
        for item in PrintInterleavings(str1, str2[1:]):
            perms.append(str2[0] + item)
    return perms
print PrintInterleavings("AB", "CD")
```

Problem-14 Given a matrix with size $n \times n$ containing random integers. Give an algorithm which checks whether rows match with a column(s) or not. For example, if i^{th} row matches with j^{th} column, and i^{th} row contains the elements - [2,6,5,8,9]. Then j^{th} column would also contain the elements - [2,6,5,8,9].

Solution: We can build a trie for the data in the columns (rows would also work). Then we can compare the rows with the trie. This would allow us to exit as soon as the beginning of a row does not match any column (backtracking). Also this would let us check a row against all columns in one pass.

If we do not want to waste memory for empty pointers then we can further improve the solution by constructing a suffix tree.

Problem-15 How do you replace all spaces in a string with '%20'. Assume string has sufficient space at end of string to hold additional characters.

Solution:

```
class ReplacableString:
    def __init__(self, inputString):
        self.inputString =inputString
    def replacer(self, to_replace, replacer):
        for i in xrange(len(self.inputString)):
            if to_replace == self.inputString[i:i+len(to_replace)]:
                self.inputString = self.inputString[:i] + replacer + self.inputString[i+len(to_replace):]
    def __str__(self):
        return str(self.inputString)
input = ReplacableString("This is eth string")
input.replacer(" ", "%20")
print input
```

Time Complexity: O(n). Space Complexity: O(1). Here, we do not have to worry on the space needed for extra characters. We have to see how much extra space is needed for filling that.

Important note: Python provides a simple way to encode URLs.

```
import urllib
input_url = urllib.quote ( 'http://www.CareerMonk.com/example one.html' )
```

In this example, Python loads the urllib module, then takes the string and normalizes the URL by replacing the unreadable blank space in the URL between "example one.html" with the special character "%20".

Problem-16 Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'. A region is captured by flipping all 'O's into 'X's in that surrounded region .

Sample Input:	Output:
X X X X	X X X X
X O O X	X X X X
X X O X	X X X X
X O X X	X O X X

Solution: We use backtracking to identify the elements not surrounded by 'X' and we mark those with a temporal symbol ('$'). The elements not surrounded by 'X' means that exists a path of elements 'O' to a border. So we start the backtracking algorithm with the boarders. The last thing is replacing the temporal element by 'O' and the rest elements to 'X'.

```
class CamptureRegions:
  # @param board, a 2D array
  # Capture all regions by modifying the input board in-place.
  # Do not return any value.
  def solve(self, board):
    if len(board)==0:
      return
    for row in range(0,len(board)):
      self.mark(board,row,0)
      self.mark(board,row,len(board[0])-1)
    for col in range(0, len(board[0])):
      self.mark(board, 0, col)
      self.mark(board, len(board)-1, col)

    for row in range(0,len(board)):
      for col in range(0, len(board[0])):
        if board[row][col]=='$':
          board[row][col] = 'O'
        else:
          board[row][col] = 'X'

  def mark(self, board, row, col):
    stack = []
    nCols= len(board[0])
    stack.append(row*nCols+col)
    while len(stack)>0:
      position = stack. pop()
      row = position // nCols
      col = position % nCols
      if board[row][col] != 'O':
        continue
      board[row][col] = '$'
      if row>0:
        stack.append((row-1)*nCols+col)
      if row< len(board)-1:
        stack.append((row+1)*nCols+col)
      if col>0:
        stack.append(row*nCols+col-1)
      if col < nCols-1:
        stack.append(row*nCols+col+1)
```

Problem-17 If h is any hashing function and is used to hash n keys in to a table of size m, where $n \leq m$, the expected number of collisions involving a particular key X is :

A) less than 1. B) less than n. C) less than m. D) less than $n/2$.

Solution: A. Hash function should distribute the elements uniformly