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. Suppose if we do not have enough space to allocate a location for each possible key then we need a mechanism to handle this case. Other 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. Hash table uses a hash function to map keys to their associated values. 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's difficult to achieve in practice.

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 collisions.

Characteristics of Good Hash Functions

A good hash function should 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 different address space but practically it is not possible to create such a hash function and the problem called *collision*. Collision is the condition where two records are stored in the same location.

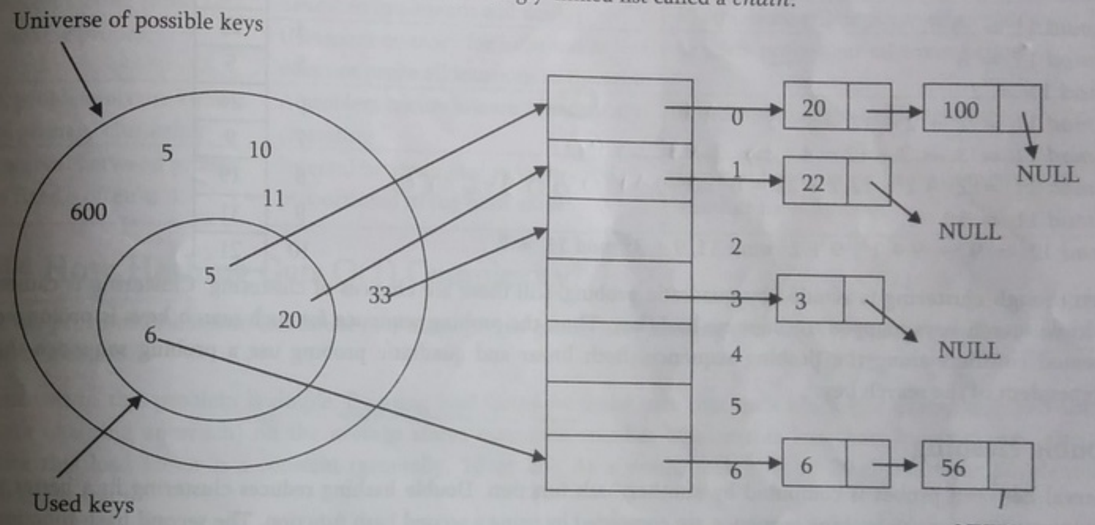
14.10 Collision Resolution Techniques

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

- **Direct Chaining:** An array of linked list application
  - Separate chaining
- **Open Addressing:** Array based implementation
  - Linear probing (linear search)
  - Quadratic probing (non linear 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 will be 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

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 the 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, table contains groups of consecutively occupied locations and called as *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 (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 chose 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

Interval between probes increases proportional to the hash value (the interval thus increasing linearly and the indices are described by a quadratic function). Clustering problem can be eliminated if we use 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 the 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:

$31\ mod\ 11 = 9$
$19\ mod\ 11 = 8$
$2\ mod\ 11 = 2$
$13\ mod\ 11 = 2 \rightarrow 2 + 12 = 3$
$25\ mod\ 11 = 3 \rightarrow 3 + 12 = 4$
$24\ mod\ 11 = 2 \rightarrow 2 + 12, 2 + 22 = 6$
$21\ mod\ 11 = 10$
$9\ mod\ 11 = 9 \rightarrow 9 + 1^2,\ 9 + 2^2\ mod\ 11, 9 + 3^2\ mod\ 11 = 7$

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

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's somewhat complicated because we have to account the memory usage. Separate chaining uses extra memory for links. Open addressing needs extra memory implicitly within the table to terminate probe sequence. Open-addressed hash tables cannot be used if the data does not have unique keys. An alternative is 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
Use few probes	Uses extra memory for links and it does not probe all locations in the table	Use few probes but take 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 doubt that many people gets is, how hashing gets O(1) if multiple elements maps to the same location?

The answer to this problem is simple. By using load factor we make sure that each block (for example linked list in separate chaining approach) on the average stores maximum number of elements less than *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 load factor then we rehash the elements with 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) while deciding the rehash.

The access time of table depends on the load factor which in-turn depends on the hash function. This is because hash function is the one which distributes the elements to hash table. For this reason, we say hash table gives O(1) complexity on the 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: static hashing and dynamic hashing

Static Hashing

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

Dynamic Hashing

If data not fixed static hashing can give bad performance and dynamic hashing is the next alternative for such type of data. The set of keys can change dynamically in this.

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 Problems on Hashing

Problem-1 Implement 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 usually between 5 to 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 prime number.

```
#define LOAD_FACTOR 20
struct ListNode {
        int key;
        int data;
        struct ListNode *next;
};
struct HashTableNode {
        int bcount;                 //Number of elements in block
        struct ListNode *next;
};
struct HashTable {
        int tsize;
        int count;                  //Number of elements in table
        struct HashTableNode **Table;
};
struct HashTable *CreatHashTable(int size) {
        struct HashTable *h;
        h = (struct HashTable *)malloc(sizeof(struct HashTable));
        if(!h)
                return NULL;
        h→tsize = size/ LOAD_FACTOR;
        h→count = 0;
        h→Table = (struct HashTableNode **) malloc( sizeof(struct HashTableNode *) * h→tsize);
        if(!h→Table) {
                printf("Memory Error");
                return NULL;
        }
        for(int i=0; i < h→tsize; i++) {
                h→Table[i]→next = NULL;
```



```
                h→Table[i]→bcount = 0;
        }
        return h;
}
int HashSearch(Struct HashTable *h, int data) {
        struct ListNode *temp;
        temp = h→Table[Hash(data, h→tsize)]→next;          //Assume Hash is a built-in function
        while(temp) {
                if(temp→data == data)
                        return 1;
                temp = temp→next;
        }
        return 0;
}
int HashInsert(Struct HashTable *h, int data) {
        int index;
        struct ListNode *temp, *newNode;
        if(HashSearch(h, data))
                return 0;
        index = Hash(data, h→tsize);                       //Assume Hash is a built-in function
        temp = h→Table[index]→next;
        newNode = (struct ListNode *) malloc(sizeof(struct ListNode));
        if(!newNode) {
                printf("Out of Space");
                return -1;
        }
        newNode→key = index;
        newNode→data = data;
        newNode→next = h→Table[index]→next;
        h→Table[index]→next = newNode;
        h→Table[index]→bcount++;
        h→count++;
        if(h→count / h→tsize  > LOAD_FACTOR)
                Rehash(h);
        return 1;
}
int HashDelete(Struct HashTable *h, int data) {
        int index;
        struct ListNode *temp, *prev;
        index = Hash(data, h→tsize);
        for(temp = h→Table[index]→next, prev = NULL; temp; prev = temp, temp = temp→next) {
                if(temp→data == data) {
                        if(prev != NULL)
                                prev→next = temp→next;
                        free(temp);
                        h→Table[index]→bcount--;
                        h→count--;
                        return 1;
                }
        }
```

```
        return 0;
}
void Rehash(Struct HashTable *h) {
        int oldsize, i, index;
        struct ListNode *p, * temp, *temp2;
        struct HashTableNode **oldTable;
        oldsize = h→tsize;
        oldTable = h→Table;
        h→tsize = h→tsize * 2;
        h→Table = (struct HashTableNode **) malloc(h→tsize * sizeof(struct HashTableNode *));
        if(!h→Table) {
                printf( "Allocation Failed");
                return;
        }
        for(i = 0; i < oldsize; i++) {
                for(temp = oldTable[i]→next; temp; temp = temp→next) {
                        index = Hash(temp→data, h→tsize);
                        temp2 = temp; temp = temp→next;
                        temp2→next = h→Table[index]→next;
                        h→Table[index]→next = temp2;
                }
        }
}
```

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 simple linear search. If it repeats then 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.

```
void RemoveDuplicates(char s[], int n) {
        for(int i = 0; i < n; i++) {
                for(int j = i+1; j < n; ) {
                        if(s[i] == s[j])
                                s[j] = s[--n];
                        else    j++;
                }
        }
        s[i] = '\0';
}
```

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 the fact that order of characters in solutions doesn't matter.

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

```
int Compare(const void* a, const void *b) {
        return *(char*)a - *(char*)b;
}
void RemoveDuplicates(char s[]) {
```



```
        int last, current;
        QuickSort(s, strlen(s), sizeof(char), Compare);
        current = 0, last = 0;
        for(; s[current]; i++) {
                if(s[last] != s[current])
                        s[++last] = s[current];
        }
}
```

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

Problem-4 Can we solve this problem in 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.

```
void RemoveDuplicates(char s[]) {
        int src, dst;
        struct HastTable *h;
        h = CreatHashTable();
        current = last = 0;
        for(; s[current]; current++) {
                if( !HashSearch(h, s[current]))     {
                        s[last++] = s[current];
                        HashInsert(h, s[current]);
                }
        }
        s[last] = '\0';
}
```

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

Problem-5 Given two arrays of unordered numbers, check 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 elemer of A check whether that element is there in B or not. There is little problem with this approach if we have duplicate 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 loo at the number of occurrences then they are not same. This problem we can solve by moving the elements which a already compared to the end of list. That means, if we found an element in B, then we move that element to the end B and in the next searching we will not those elements. But the disadvantage of this is it needs extra swaps. Tim 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 bo arrays are n, we need O($n \log n$) time for sorting them. After sorting, we just need to scan both the arrays with tw pointers and see whether they point to same element every time and keep moving the pointers until we reach the e of arrays.

Time Complexity of this approach is O($n \log n$). This is because we need O($n \log n$) for sorting the arrays. Af 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, using 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 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 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 $pair(i,j)$ exist, and $pair\ (j,i)$ exist report all such pairs. For example, {{1,3},{2,6},{3,5},{7,4},{3,5},{8,7}} here, {3,5} and {5,3} are present. To report this pair, when you encounter {5,3}. We call such pairs as symmetric pairs. So, given an efficient algorithm for finding all such pairs.

Solution: By using hashing, we can solve this problem just in one scan and consider following algorithm.
Algorithm:

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

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

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

Solution: Using Hash Tables
Algorithm:

- Traverse the linked list nodes one by one.
- Check if the nodes address is there in the hash table or not.
- If it is already there in the hash table then that indicates that we are visiting the 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 nodes address into the hash table.
- Continue this process until we reach end of the linked list *or* we find loop.

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

Note: for efficient solution refer *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 which 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 then increment its counter value [this indicates the number of occurrence of the element].
- If the element is not there in the hash table then insert that node into the hash table with counter value 1.
- Continue this process until we reach end of the array.

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

Note: For efficient solution refer *Searching* chapter.

Problem-11 Given m sets of integers and have n elements in them. Give an algorithm to find an element which appeared in maximum number of sets?

Solution: Using Hash Tables
Algorithm:

- Scan the input sets one by one.
- For each element keep track the counter. Counter indicates the frequency of the occurrences in all the sets.
- After completing scanning of all the sets, select the one which is having 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 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 are same.
- Now scan the second array and check whether $(K\text{-}selected\ element)$ exits in the hash table or not.
- If it exits then return the pair of elements.
- Otherwise keep continue until we reach end of the set.

Time Complexity: $O(Max(m,m))$, because we are doing two scans. Space Complexity: $O(Min(m,m))$, 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 character, we check whether that character needs to be deleted or not. If the flag is set then we simply skip to next character otherwise keep the character in the input string. Continue this process until we reach end of the input string. All these operations we can do in-place as given below.

```
void RemoveChars(char str[], char removeTheseChars[]) {
        int srcInd, destInd;
        int auxi[256]; //additional array
        for(srcInd =0; srcInd<256; srcIndex++)
                auxi[srcInd]=0;
        //set true for all characters to be removed
        srcIndex=0;
        while(remove[srcInd]) {
                auxi[removeTheseChars[srcInd]]=1;
                srcInd++;
```

```
        }
        //copy chars unless it must be removed
        srcInd=destInd=0;
        while(str[srcInd++]) {
                if(!auxi[str[srcInd]])
                        str[destInd++]=str[srcInd];
        }
}
```

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 it in mind when we are dealing with multi-byte characters where 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 does not appears then we are done with the solution and return that character. If the character appears in the remaining string then go to the next character.

```
char FirstNonRepeatedChar( char *str ) {
        int i, j, repeated = 0;
        int len = strlen(str);
        for(i = 0; i < len; i++ )       {
                repeated = 0;
                for( j = 0; j < len; j++ )      {
                        if( i != j && str[i] == str[j] ) {
                                repeated = 1;
                                break;
                        }
                }
                if( repeated == 0 )             // Found the first non-repeated character
                        return str[i];
        }
        return '';
}
```

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

Problem-15 Can we improve the time complexity of Problem-13*?*

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 $O(n)$.

```
char FirstNonRepeatedCharUsinghash( char * str ) {
        int i, len=strlen(str);
        int count[256]; //additional array
        for(i=0;i<len;++i)
                count[i] = 0;
        for(i=0;i<len;++i)
                count[str[i]]++;
```



```
        for(i=0; i<len; ++i) {
                if(count[str[i]]==1) {
                        printf("%c",str[i]);
                        break;
                }
        }
        if(i==len)
                printf("No Non-repeated Characters");
        return 0;
}
```

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 almost similar to Problem-13 and Problem-15. The only difference is, instead of scanning the hash table twice we can give the answer in one scan itself. This is because while inserting into the hash table we can see whether that element already exists or not. If it already exits then we just need to return that character.

```
char FirstRepeatedCharUsinghash( char * str ) {
        int i, len=strlen(str);
        int count[256]; //additional array
        for(i=0;i<len;++i)
                count[i] = 0;
        for(i=0; i<len; ++i) {
                if(count[str[i]]==1) {
                        printf("%s",str[i]);
                        break;
                }
                else    count[str[i]]++;
        }
        if(i==len)
                printf("No Repeated Characters");
        return 0;
}
```

Time Complexity: We have $O(n)$ for scanning and create 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. Give 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. We can use both key and value are 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 keep continue until we 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 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 the 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 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 all unique. Each line is so long that it may not even fit in memory. What is the most efficient solution for finding the identical lines?

Solution: Since complete line may not fit into the main memory, read the line partially and compute the hash from that partial line. Next, again read the next part of line and compute the hash. This time use the previous has also while computing the new hash value. Continue this process until we find the hash for complete line.

Do this for each line and store all the hash values in some file [or maintain some hash table of these hashes]. At any point if we get same hash value, then read the corresponding lines part by part and compare.

Note: Refer *Searching* chapter for related problems.

STRING ALGORITHMS

Chapter-15

15.1 Introduction

Before starting our discussion, let us understand the importance of string algorithms. Consider the case of entering the URL (Uniform Resource Locator) in any browser (say, Internet Explorer, Firefox, or Google Chrome), you might have observed that after typing the prefix of the URL, it displays list of all possible URLs. That means, the browsers are doing some internal processing and giving us the list of matchings. This technique is sometimes called as *auto – completion*.

Similarly, if we consider the case of entering the directory name in command line interface (in both *Windows* and *UNIX*), after typing the prefix of the directory name and if we press *tab* button, then 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 sting data efficiently. In this chapter, we will look at the data structures which are useful for implementing string algorithms.

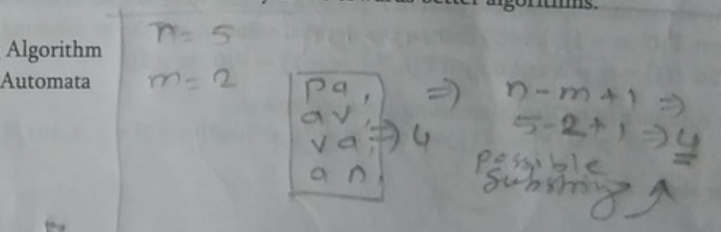
We start our discussion with the basic problem of strings: given a string, how do we search a substring (pattern)? This problem is called *string matching* problem. After discussing various string matching algorithms, we will see different data structures for storing strings.

15.2 String Matching Algorithms

In this section, we concentrate on the problem of 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. For simplicity of 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 brute method and slowly 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 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

```
int BruteForceStringMatch (int T[], int n,  int P[], int m) {
```

```
for (int i = 0; i <=n - m; i++) {
        int j = 0;
        while (j < m && P[j] == T[i + j])
                j = j + 1;
        if(j == m )
                return i;
}
return -1;
}
```

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 hashing of m characters of T gives same result.

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

If the hash function is not good (worst case), then the complexity of Robin-Karp algorithm complexity is $O(n-m+1)\times m) \approx O(n\times m)$. If we select a good hash function then the complexity of 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 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:

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

Code for above assumption is:

```
value = 0;
for (int i = 0; i < m-1; i++) {
        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.

15.5 String Matching with Finite Automata

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

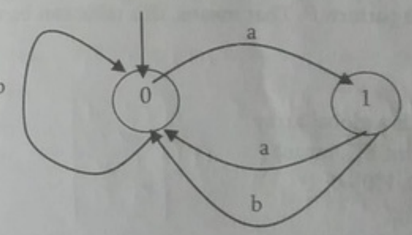
How Finite Automata Works?

- 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 is called rejected string

Example: Let us assume that $Q = \{0,1\}, q_0 = 0, A = \{1\}, \sum = \{a,b\}$. $\delta(q,a)$ 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 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 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 pattern character, then the FA go to a state $0, 1, 2, \ldots,$ or k, depending on how many initial pattern characters match 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 $Q = \{0, 1, 2, ..., m\}$
  - 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 $O(m|\sum|)$. This is because δ has $O(m|\sum|)$ entries. If we assume $|\sum|$ is constant then the complexity becomes $O(m)$.

Algorithm:

```
//Input: Pattern string P[0..m-1], δ and F
//Goal: All valid shifts displayed
FiniteAutomataStringMatcher(int P[], int m, F, δ) {
        q = 0;
        for (int i = 0; i < m; i++)
                q = δ(q,T[i]);
                if(q == m) printf("Pattern occurs with shift: %d", 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 given 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 have previously involved in comparison with some element of the pattern P.

The algorithm uses a table and in general we call it as *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. That means, this table can be used for avoiding backtracking on the string T.

Prefix Table

```
int F[]; //assume F is a global array
void Prefix-Table(int P[], int m) {
        int i=1,j=0, F[0]=0;
        while(i<m) {
                if(P[i]==P[j]) {
                        F[i]=j+1;
                        i++;
                        j++;
                }
                else if(j>0)
                        j=F[j-1];
                else {  F[i]=0;
```



```
                        i++;
                }
        }
}
```

As an example, assume that $P = a\ b\ a\ b\ a\ c\ a$. For this pattern, les 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: $i = 1, j = 0, F[1] = 0$

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

Step 2: $i = 2, j = 0, F[2] = 1$

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

Step 3: $i = 3, j = 1, F[3] = 2$

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

Step 4: $i = 4, j = 2, F[4] = 3$

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

Step 5: $i = 5, j = 3, F[5] = 1$

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

Step 6: $i = 6, j = 1, F[6] = 1$

	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 filling of prefix table is complete.

Matching Algorithm

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

```
int KMP(char T[], int n, int P[], int m) {
        int i=0,j=0;
        Prefix-Table(P,m);
        while(i<n) {
                if(T[i]==P[j]) {
                        if(j==m-1)
                                return i-j;
                        else {  i++;
                                j++;
                        }
                }
                else if(j>0)
                        j=F[j-1];
                else    i++;
        }
        return -1;
}
```

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.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P	a	b	a	b	a	c	a								

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.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P		a	b	a	b	a	c	a							

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 $T[2]$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P		a	b	a	b	a	c	a							

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

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P				a	b	a	b	a	c	a					

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

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

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

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

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

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

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

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

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

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

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

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

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

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P						a	b	a	b	a	c	a			

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

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

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

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

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

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

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 in $O(n+m)$ time complexity (does not depend on alphabet size)

15.7 Boyce-Moore Algorithm

Like KMP algorithm, this also does some pre-processing and in general we call it as *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 lots of needless comparisons by shifting pattern relative to text.

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

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

15.8 Data structures for Storing Strings

If we have a set of strings (for example, all words in 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

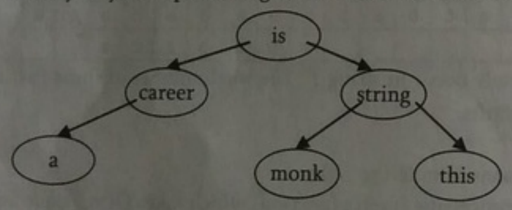
As seen in *Hashing* chapter, we can use hash tables for storing the integers or strings. In this case, the keys are nothing but the strings. The problem with hash table implementation is that, we lose the ordering information. Since, after applying the hash function, we do not know where it will map to. As a result some queries take more time. For example, to find all words starting with letter "*K*", then using 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 strings have a natural ordering: A comes before B, which comes before C, and so on. Since 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 and one such possibility is shown in below tree.



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 search operation increases. So, from this we can say that BST representation of strings is good in terms of storage but not in time.

15.11 Tries

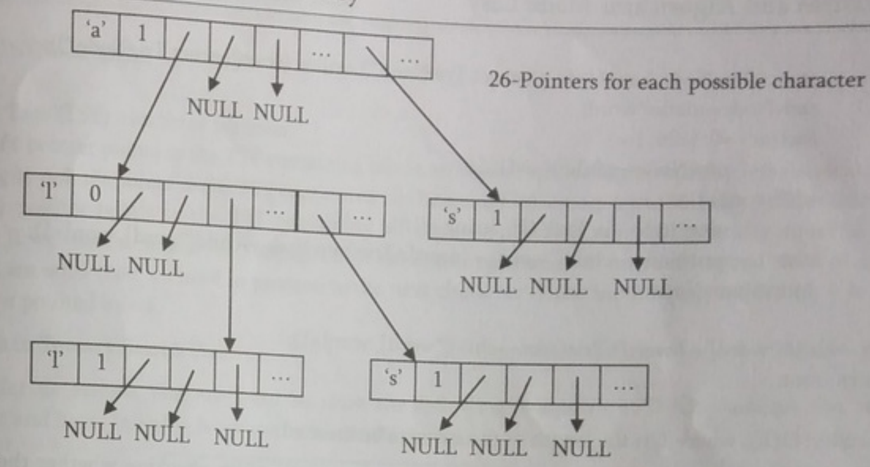
Now, let us see the alternative representation which reduces the time complexity of search operation. The name *trie* is taken from the word re"trie".

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:

```
struct TrieNode {
        char data;                      // Contains the current node character.
        int is_End_Of_String;           // Indicates whether the string formed from root to
                                        // current node is a string or not
        struct TrieNode *child[26];     // Pointers to other tri nodes
};
```

Suppose if 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 represent the length of a single word). This is much faster than hash table and binary search tree representations.

Trie Declaration

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

```
struct TrieNode {
        char data;
        int is_End_Of_String;
        struct TrieNode *child[];
}
struct TrieNode *TrieNode subNode(struct TrieNode *root, char c){
        if(child! = NULL){
                for(int i=0; i < 26; i++){
                        if(root.child[i]->data == c)
                                return eachChild;
                }
        }
        return NULL;
}
}
```

Now that we have defined our TrieNode, let's go ahead and look at the other operations of TRIE. Fortunately, the TRIE datastructure 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 give 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.

```
void InsertInTrie(struct TrieNode *root, char *word) {
        if(!*word) return;
        if(!root) {
```

```
        struct TrieNode *newNode = (struct TrieNode *) malloc (sizeof(struct TrieNode *));
        newNode→data=*word;
        for(int i =0; i<26; i++)
                newNode→child[i]=NULL;
        if(!*(word+1))
                newNode→is_End_Of_String = 1;
        else    newNode→child[*word] = InsertInTrie(newNode→child[*word], word+1);
        return newNode;
    }
    root→child[*word] = InsertInTrie(root→child[*word], word+1);
    return root;
}
```

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

Note: For real dictionary implementation we may need few more checks like: checking whether the given string is already there in dictionary or not. For simplicity we can ignore them

Searching a String in Trie

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

```
int SearchInTrie(struct TrieNode *root, char *word) {
    if(!root) return -1;
    if(!*word) {
        if(root→ is_End_Of_String) return 1;
        else    return -1;
    }
    if(root→data == *word)
        return SearchInTrie(root→child[*word], word+1);
    else    return -1;
}
```

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 are having too many node pointers. In many cases, the occupancy of each node is very less. The final conclusion regarding tries data structure is, they are faster but takes huge memory for storing the strings.

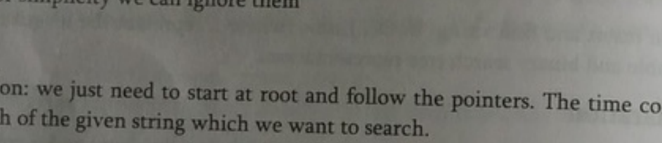
Note: There are some improved tries representations called *trie compression techniques*. But, even with those techniques we can only reduce the memory at leaves only but not at the internal nodes.

15.12 Ternary Search Trees

This representation was initially given 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 time efficiency of tries.

Ternary Search Trees Declaration

```
struct TSTNode {
    char data;
    int is_End_Of_String;
    struct TSTNode *left;
    struct TSTNode *eq;
```



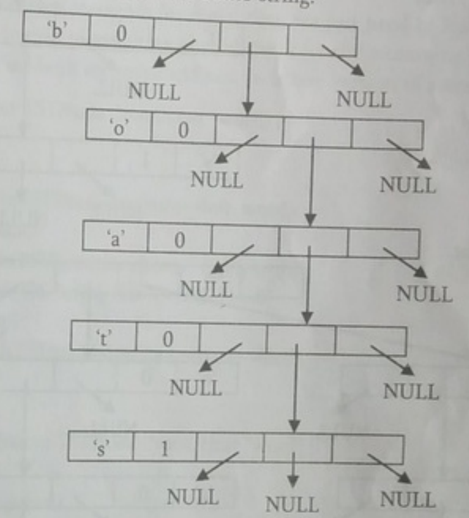

```
    struct TSTNode *right;
};
```

Ternary Search Tree (TST) uses three pointers:

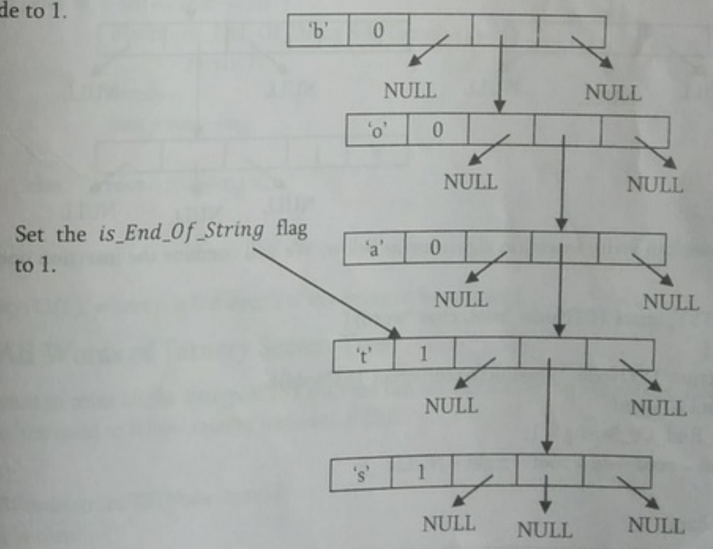
- The *left* pointer points to the TST containing all the strings which are alphabetically less than *data*.
- The *right* pointer points to the TST containing all the strings which are alphabetically greater than *data*.
- The *eq* pointer points to the TST containing all the strings which are alphabetically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

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 *boats* string.

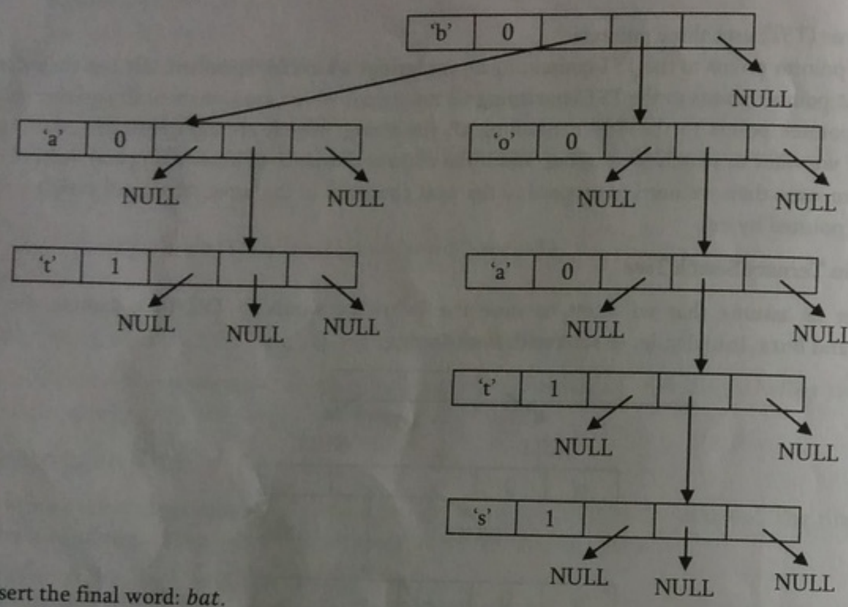


Now if we want to insert the string *boat*, then the TST becomes: The only change is setting the *is_End_Of_String* flag of "*t*" node to 1.

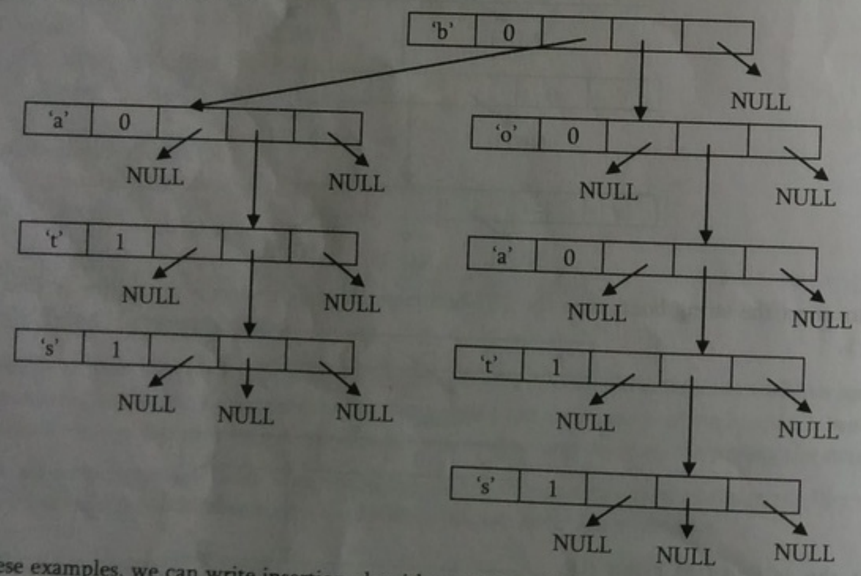


Set the *is_End_Of_String* flag to 1.

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



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



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

```
struct TSTNode *InsertInTST(struct TSTNode *root, char *word) {
        if(root == NULL) {
                root = (struct TSTNode *) malloc(sizeof(struct TSTNode));
                root→data = *word;
                root→is_End_Of_String = 1;
                root→left = root→eq = root→right = NULL;
        }
        if(*word < root→data)
```



```
                root→left = InsertInTST (root→left, word);
        else if(*word == root→data) {
                if(*(word+1))
                        root→eq = InsertInTST (root→eq, word+1);
                else    root→is_End_Of_String = 1;
        }
        else    root→right = InsertInTST (root→right, word);
        return root;
}
```

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

Searching in Ternary Search Tree

After inserting the words, if we want to search for them, then we just need 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 version of search method.

```
int SearchInTSTRecursive(struct TSTNode *root, char *word) {
        if(!root) return -1;
        if(*word < root→data)
                return SearchInTSTRecursive(root→left, word);
        else if(*word > root→data)
                return SearchInTSTRecursive(root→right, word);
        else {  if(root→is_End_Of_String && *(word+1)==0)
                        return 1;
                return SearchInTSTRecursive(root→eq, ++word);
        }
}
int SearchInTSTNon-Recursive(struct TSTNode *root, char *word) {
        while (root) {
                if(*word < root→data)
                        root = root→left;
                else if(*word == root→data) {
                        if(root→is_End_Of_String && *(word+1) == 0)
                                return 1;
                        word++;
                        root = root→eq;
                }
                else    root = root→right;
        }
        return -1;
}
```

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

Displaying All Words of Ternary Search Tree

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

```
char word[1024];
void DisplayAllWords(struct TSTNode *root) {
        if(!root) return;
```

```
        DisplayAllWords(root→left);
        word[i] = root→data;
        if(root→is_End_Of_String) {
                word[i] = '\0';
                printf("%c",word);
        }
        i++;
        DisplayAllWords(root→eq);
        i--;
        DisplayAllWords(root→right);
}
```

Finding Length of Largest Word in TST

This is very much similar to finding height of the BST and can be found as:

```
int MaxLengthOfLargestWordInTST(struct TSTNode *root) {
        if(!root) return 0;
        return Max(MaxLengthOfLargestWordInTST(root→left),
                MaxLengthOfLargestWordInTST(root→eq)+1,
                MaxLengthOfLargestWordInTST(root→right)));
}
```

15.13 Comparing BSTs, Tries and TSTs

- Hash table and BST implementation stores complete 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 resizes only based on load factor.
- TSTs allows partial search where as BSTs and hash tables do not support them.
- TSTs can display the words in sorted order but in hash tables we cannot get the sorted order.
- Tries performs search operations very fast but they take huge memory for storing the string.
- TST combines the advantages of BSTs and Tries. That means it combines the memory efficiency of BSTs and time efficiency of tries.

15.14 Suffix Trees

Suffix trees are one of important data structures for strings. With suffix trees we can answer the queries very fast. But it needs some preprocessing and nothing but construction of suffix tree. Even though construction of suffix tree is complicated, it solves many other string related problems in linear time.

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

Before starting our discussion let us see the terminology we use for this representation.

Prefix and Suffix

For given a string $T = T_1T_2 \dots T_n$, *prefix* of T is a string $T_1 \dots T_i$ 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, for given a string $T = T_1T_2 \dots T_n$, *suffix* of T is a string $T_i \dots T_n$ 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 a text T and a 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 is $T = accbkkbac$ and the pattern is $P = kkb$. For this example, P is a prefix o 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 o suffix trees can be given as: A suffix tree for a n character string $T[1 \dots n]$ is a rooted tree with the following properties.

- 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 tree is labeled by a nonempty substring of T
- No two edges out of a node (children edges) begins 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 suffix.
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.

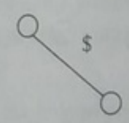
For better understanding, let us go through an example. Let the given text is $T = tatat$. For this string, give the numbering for each of the suffixes.

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*\$	

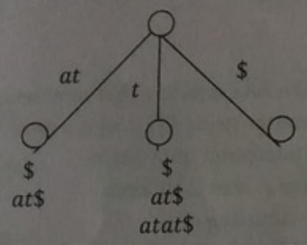
In the three groups the first group is having only one element. So, as per the algorithm create a leaf node for it and same is shown below.



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

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

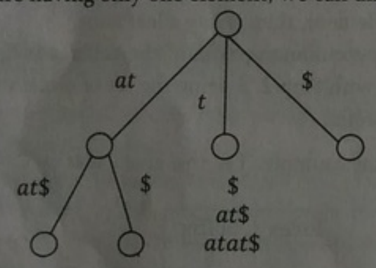
For S_2 and S_3, create internal nodes and the edge contains the longest common prefix of those groups.



Now what need to do is, remove the longest common prefix from S_2 and S_3 group elements.

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

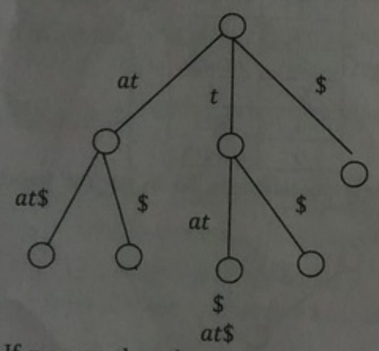
Out next step is, solving S_2 and S_3 recursively. First let us take S_2. In this group, if we sort them based on their first character, it is easy to see that the first group contains only one element \$ and the second group also contains only one element, $at\$$. Since both groups are having only one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



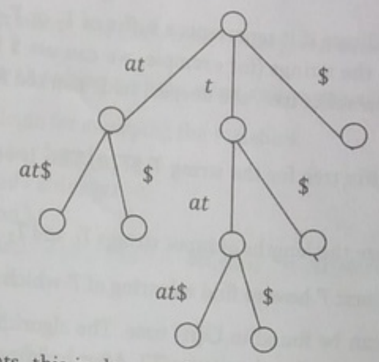
At this step, both $S_1$and S_2 elements are done and the only remaining group is S_3. As similar to earlier steps, in S_3 group, if we sort them based on their first character, it is easy see that there is only one element in first group and it is \$. For S_3 remaining elements remove the longest common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

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

In the S_3 second group, there are two elements and among them one is \$ and other is $at\$$. We can directly add the leaf nodes for the first group element \$. Let us add S_3 subtre as shown below.



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



Since there are no more elements, this is the completion of construction of suffix tree for the 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.

Notes:

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

- **Exact String Matching:** Given a text T and a pattern P, how to check whether P appears in T or not?
- **Longest Repeated Substring:** Given a text T how to find the substring of T which is the maximum repeated substring?
- **Longest Palindrome:** Given a text T how to find the substring of T which is the longest palindrome of T?
- **Longest Common Substring:** Given two strings, how to find the longest common substring?
- **Longest Common Prefix:** Given two strings $X[i \ldots n]$ and $Y[j \ldots m]$, how to find the longest common prefix?
- How to search for a regular expression in given text T?
- Given a text T and a pattern P, how to find the first occurrence of P in T?

15.15 Problems on Strings

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

Solution: For this problem we can use combination of priority queues and tries. What we do is, create a trie in which when a word comes insert it in trie and at every leaf of trie, its node contains that word along with a pointer which is points to the node in the heap [priority queue] which we also create. This heap contains nodes whose structure contains a *counter* which is its frequency and also a pointer to that leaf of trie which contain that word so that there is no need to store this word twice. Whenever a new word comes we find it in trie, if it is already there then we increase the frequency of node in heap corresponding to that word and call heapify, 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 new frequency is zero, delete the node from heap and delete it from trie.

Problem-2 Given two strings how to find the longest common substring?

Solution: Let us assume that the given two string 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.

Other 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 to find 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 #. It's nothing but finding the longest common substring.

Problem-4 Given a string (word), give an algorithm for finding the next word in 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 the below simple approach. Starting from the rightmost character, keep incrementing the characters one by one. Once, we reach Z, move to next character on left side. Whenever we increment, check if the word with the incremented character exist in dictionary or not. If 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*

```
char *ReversingString(char str[]) {
        char temp, start, end;
        if(str == NULL || *str == '\0')
                return str;
        for (end = 0; str[end]; end++);
        end--;
        for (start = 0; start < end; start++, end--) {
                temp = str[start]; str[start] = str[end]; str[end] = temp;
        }
        return str;
}
```

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

Problem-6 If the string is not editable, how do we create a string which is the reverse of given string?

Solution: If the string is not editable, then we need to create an array and return the pointer of that.

```
//If str is a const string (not editable)
char* ReversingString(char* str) {
        int start, end, len;
        char temp, *ptr=NULL;
        len=strlen(str);
        ptr=malloc(sizeof(char)*(len+1));
        ptr=strcpy(ptr,str);
        for (start=0, end=len-1; start<=end; start++, end--) {          //Swapping
                temp=ptr[start]; ptr[start]=ptr[end]; ptr[end]=temp;
        }
        return ptr;
}
```


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

Problem-7 Can we reverse the string without using any temporary variable?

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

```
char* ReversingString(char *str) {
        int start = 0, end= strlen(str)-1;
        while( start<end ) {
                str[start] ^= str[end];      str[end] ^= str[start];      str[start] ^= str[end];
                ++start;
                --end;
        }
        return str;
}
```

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

Problem-8 Given a text and a pattern, give an algorithm for matching pattern in text. Assume ? (single character matcher) and * (multi character matcher) are the wild card characters.

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

```
int PatternMatching(char *text, char *pattern) {
        if(*pattern == 0) return 1;
        if(*text == 0) return *p == 0;
        if('?' == *pattern)
                return PatternMatching(text+1,pattern+1) || PatternMatching(text,pattern+1);
        if('*' == *pattern)
                return PatternMatching(text+1,pattern) || PatternMatching(text,pattern+1);
        if(*text == *pattern)
                return PatternMatching(text+1,pattern+1);
        return -1;
}
```

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-9 Give an algorithm for reversing a words of 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.

```
void ReverseWordsInSentences(char *text) {
        int worsStart, wordEnd, length;
        length = strlen(text);
        ReversingString(text, 0, length-1);
        for(worsStart = wordEnd = 0; wordEnd < length; wordEnd ++) {
                if(text[wordEnd] ! = ' ') {
                        worsStart = wordEnd;
                        while (text[wordEnd] != ' ' && wordEnd < length)
                                wordEnd ++;
                        wordEnd--;
                        ReverseWord(text, worsStart, wordEnd);          //Found current word, reverse it now.
                }
        }
}
```

```
void ReversingString(char text[], int start, int end) {
        for (char temp; start < end; start++, end--) {
                temp = str[end];        str[end] = str[start];        strstart] = temp;
        }
}
```

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

Problem-10 **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, each occurrence of a repeated character is considered to be 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 nothing but generating $n!$ strings each of length n, where n is the length of the input string.

```
void Permutations(int depth, char *permutation, int *used, char *original) {
        int length = strlen(original);
        if(depth == length) printf("%c",permutation);
        else {    for (int i = 0; i < length; i++) {
                        if(!used[i]) {
                                used[i] = 1;
                                permutation[depth] = original[i];
                                Permutations(depth + 1, permutation, used, original);
                                used[i] = 0;
                        }
                }
        }
}
```

Problem-11 **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 nothing but 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.

```
void Combinations(int depth, char *combination, int start, char *original) {
        int length = strlen(original);
        for (int i = start; i < length; i++) {
                combination[depth] = original[i];
                combination[depth +1] = '\0';
                printf("%c", combination);
                if(i < length -1)
                        Combinations(depth + 1, combination, start + 1, original);
        }
}
```

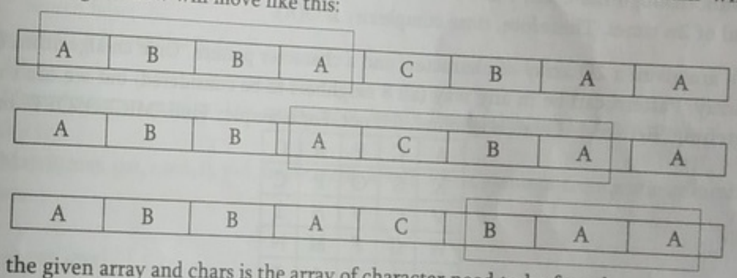
Problem-12 Given a string "ABCCBCBA", give an algorithm for recursively removing the adjacent characters if they are same. For example, ABCCBCBA --> ABBCBA-->ACBA

Solution: We should check if we have character pair then cancel it and then check for next character and previous element. Keep canceling the characters until we either reach start of the array, end of the array or not find a pair [1].

```
void RremoveAdjacentPairs(char* str) {
        int len = strlen(str), i, j = 0;
        for (i=1; i <= len; i++) {
                while ((str[i] == str[j]) && (j >= 0)){          //Cancel pairs
                        i++;
                        j--;
                }
                str[++j] = str[i];
        }
        return;
}
```

Problem-13 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 sliding window approach. In this approach, we start from the beginning of the array and move to right. As soon as we have a window, which have all the required elements, try sliding the window to as much right as possible with all the required elements. If current window length is less than min length found till now, update min length [1]. For example, if the input array is $ABBACBAA$ and minimum window should cover characters AAB then sliding window will move like this:



Algorithm: Input is the given array and chars is the array of character need to be found.

1 Make an integer array shouldfind[] of len 256. i^{th} element of this array will have the count how many times we need to find element of ASCII value i.
2 Make another array hasfound of 256 elements, which will have the count of required element found till 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 till now. Update min length.

```
#define MAX 256
void MinLengthWindow(char input[], char chars[]) {
        int shouldfind[MAX] = {0,}, hasfound[MAX] = {0,};
        int j=0, cnt = 0, start=0, finish, minwindow = INT_MAX;
        int charlen = strlen(chars), iplen = strlen(input);
        for (int i=0; i< charlen; i++)
                shouldfind[chars[i]] += 1;
        finish = iplen;
```

```
    for (int i=0; i< iplen; i++) {
        if(!shouldfind[input[i]])
            continue;
        hasfound[input[i]] += 1;
        if(shouldfind[input[i]] >= hasfound[input[i]])
            cnt++;
        if(cnt == charlen) {
            while (shouldfind[input[j]] == 0 || hasfound[input[j]] > shouldfind[input[j]])  {
                if(hasfound[input[j]] > shouldfind[input[j]])
                    hasfound[input[j]]--;
                j++;
            }
            if(minwindow > (i - j +1)) {
                minwindow = i - j +1;
                finish = i;
                start = j;
            }
        }
    }
    printf("Start:%d and Finish: %d", start, finish);
}
```

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

Problem-14 We are given a 2D array of characters and a character pattern. Give an algorithm to find if pattern is present in 2D array. Pattern can be in any way (all 8 neighbors to be considered) but we can't use same character twice while matching. Return 1 if match is found, 0 if not. For example: Find "MICROSOFT" in below matrix.

A	C	P	R	C
X	S	O	P	C
V	O	V	N	I
W	G	F	M	N
Q	A	T	I	T

Solution: Manually finding the solution of this problem is relatively intuitive; we just need to describe an algorithm for it. Ironically, describing the algorithm is not the easy part.

How do we do it manually? First we match the first element, if it matched we matched the second element in the 8 neighbors of first match, do this process recursively, when last character of input pattern matches, return true. During above process, you take care not to use any cell in 2D array twice. For this purpose, you mark every visited cell with some sign. If your pattern matching fails at some place, you start matching from the beginning (of pattern) in remaining cells. While returning, you unmark visited cells.

Let's convert above intuitive method in algorithm. Since we are doing similar checks every time for pattern matching, a recursive solution is what we need here. In recursive solution, we need to check if the substring passed is matched in the given matrix or not. The condition is not to use the already used cell. For finding already used cell, we need to have another 2D array to the function (or we can use an unused bit in input array itself.) Also, we need the current position of input matrix, from where we need to start. Since we need to pass a lot more information than actually given, we should be having a wrapper function to initialize that extra information to be passed.

Algorithm:

```
If we are past the last character in pattern
  Return true
If we got an used cell again
```



```
    Return false if we got past the 2D matrix
    Return false
  If searching for first element and cell doesn't match
    FindMatch with next cell in row-first order (or column first order)
  otherwise if character matches
    mark this cell as used
    res = FindMatch with next position of pattern in 8 neighbors
    mark this cell as unused
    Return res
  Otherwise
    Return false
```

```
#define MAX 100
boolean FindMatch_wrapper(char mat[MAX][MAX], char *pat, int nrow, int ncol) {
    if(strlen(pat) > nrow*ncol) return false;
    int used[MAX][MAX] = {{0,},};
    return FindMatch(mat, pat, used, 0, 0, nrow, ncol, 0);
}
//level: index till which pattern is matched & x, y: current position in 2D array
boolean FindMatch(char mat[MAX][MAX], char *pat, int used[MAX][MAX], int x, int y, int nrow, int ncol, int level) {
  if(level == strlen(pat)) //pattern matched
    return true;
  if(nrow == x || ncol == y)      return false;
  if(used[x][y])      return false;
  if(mat[x][y] != pat[level] && level == 0) {
    if(x < (nrow - 1))
      return FindMatch(mat, pat, used, x+1, y, nrow, ncol, level); //next element in same row
    else if(y < (ncol - 1))
      return FindMatch(mat, pat, used, 0, y+1, nrow, ncol, level); //first element from same column
    else  return false;
  }
  else if(mat[x][y] == pat[level]) {
    boolean res;
    used[x][y] = 1;      //marking this cell as used
    //finding subpattern in 8 neighbours
    res = (x > 0  ? FindMatch(mat, pat, used, x-1, y, nrow, ncol, level+1) :  false) ||
        (res = x < (nrow - 1)  ? FindMatch(mat, pat, used, x+1, y, nrow, ncol, level+1) :  false) ||
        (res = y > 0 ? FindMatch(mat, pat, used, x, y-1, nrow, ncol, level+1) :  false) ||
        (res = y < (ncol - 1)  ? FindMatch(mat, pat, used, x, y+1, nrow, ncol, level+1) :  false) ||
        (res = x < (nrow - 1) && (y < ncol -1) ? FindMatch(mat, pat, used, x+1, y+1, nrow, ncol, level+1) :  false) ||
        (res = x < (nrow - 1) && y > 0  ?  FindMatch(mat, pat, used, x+1, y-1, nrow, ncol, level+1) : false) ||
        (res = x > 0 && y < (ncol - 1) ?   FindMatch(mat, pat, used, x-1, y+1, nrow, ncol, level+1) : false) ||
        (res = x > 0 && y > 0 ? FindMatch(mat, pat, used, x-1, y-1, nrow, ncol, level+1) : false);

    used[x][y] = 0;              //marking this cell as unused
    return res;
  }
  else return false;
}
```

Problem-15 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"