UNIT 1

DICTIONARIES AND HASHING

Short Questions with Answers

Q1. Define hashing?

Ans :

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects.

Q2. List out the implementation steps of hashing?

Ans :

Hashing is implemented in two steps :

→ An element is converted into an integer by using hash function. This element can be used as index to store the original element, which falls into hash table.

→ The element is stored in the hash table where it can be quickly retrieved using hashed key.

hash = hashfunc(Key)

index = hash% array_size.

Q3. Define Hash function.

Ans :

A Hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which fall into hash table.

Q4. Define hash table.

Ans :

A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched.

→ The average time required to search for an element in a hash table is O(1).

Q5. List out the types of collision resolution techniques.

Ans :

Types of collision resolution techniques are

→ Separate chaining (open hashing)

→ Linear probing (open addressing or closed hashing)

→ Quardratic probing

Q6. Define Double hashing.

Ans :

Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.

Q7. What are the applications of Hash tables?

Ans :

→ Hash tables are commonly used to implement many types of in memory tables. They are used to implement associative arrays.

→ Hash tables may also be used as disk based data structure and database indices.

→ Hash functions are used in various algorithms to make their computing faster.

Q8. Define dictionaries.

Ans :

A dictionary is a general purpose data Structure for storing a group of objects. A dictionary is a set of Keys and each key has a single associated value.

Q9. Define Rehashing.

Ans :

As name suggests, rehashing means hashing again. It is done because whenever key value pairs are inserted into map, the load factor increases, which implies that time complexity also increases. Hence rehash must be done, increasing the size of bucket array to reduce the load factor and time complexity.

Q10. Define extendible hashing.

Ans :

Extendible hashing is similar as regular hashing. But it uses string contains the binary form of the hashes of input elements. So that the elements canly by using the buckets.

Essay Questions with Answers

1.1 Dictonaries

1.1.1 Definition, Dictionary

Q1. What is a dictionary? Explain briefly about dictionaries.

Ans :

Dictionary :

→ A dictionary is an unordered or ordered list of key-element pairs. Where keys are used to locate elements in the list.

→ A dictionary is a dynamic set ADT.

→ ADT is an object with generic description independent of implementation details. A dictionary is a container of elements and value, where every value is associated with corresponding key.

Operations of Dictionary :

(i) Adding pairs to dictionary.

(ii) Deleting a pair associated with a specific key from dictionary.

(iii) Finding a pair.

(iv) Determining whether the dictionary is empty or full.

ADT Dictionary :

→ Basic opeations performed in abstract data type dictionaries are as follows :

Insert (X, D) → insertion of element x(key & value) in dictionary D.

Delete (X, D) → deletion of element x (key & value in dictionary D.

Search (X, D) → Searching prescribed value x in dictionary D with a key of an element x.

Size (D) → It returns the count of total number of elements in D.

Max (D) → It returns the maximum element in the dictionary D.

Min (D) → It returns the minimum element in dictionary D.

1.1.2 Abstract Datatype

Q2. Discuss in brief about ADT.

Ans :

Abstract Data Type :

Abstract Data Type is a collection of data types and their associated operations.

ADT is basically a special kind of data type created by programmer to handle complex data structure.

List ADT :

→ Operations performed in List ADT are

get ( ) : Return an element from the list at any given position.

insert ( ) : Insert an element at any position in list.

remove ( ) : Remove the first occurence of any element.

size ( ) : Return the number of elements in list.

is empty ( ) : Return true if list is empty.

is full ( ) : Return true if list is full, otherwise false.

Stack ADT :

→ Operations performed in stack ADT are

create ( ) : Returns the address of stack node

destroy ( ) : Returns null pointer by deleting all nodes.

push ( ) : Insert an element at one end of stack.

pop ( ) : Remove and return the element at top of stack.

count ( ) : Return the number of elements in stack.

empty ( ) : Return true is stack is empty. if not false.

full ( ) : Return true if stack is full otherwise false.

Queue ADT :

→ Operations performed in Queue ADT are :

create ( ) : Allocates a node for queue header and returns queue head.

destroy ( ) : Deletes all elements of a queue.

front ( ) : returns reference of element stored infront end.

rear ( ) : Returns reference of element stored in rear end.

enqueue ( ) : Insert an element at end of queue

deque ( ) : Remove and return first element of queue.

empty ( ) : Return true if queue is empty.

full ( ) : Return true of queue is full.

size ( ) : Return number of elements in queue

Example :

The example c program for abstract data type is given below :

```
# include < stdio.h>
```

```
# include .conio.h>
struct current_date
{
int date, month, year;
};
main ( )
{
struct current_date d;
clrscr ( );
printf("enter date (date month year):");
scanf ("%d%d%d", &d.date, & d.month, & d.year);
getch ( );
}
```

Advantages of Abstract data types :

→ Abstract data types after several advantages over concrete data types.

a) Representation independence

b) Modularity

c) Interchangeability of parts

Disadvantages of Abstract data types :

→ ADT's do not support fix/match syntax.

→ ADT's have to expose derived operations.

→ ADT's prohibit computational reasoning.

Q3. Explain about Dictionary ADT.

Ans :

Dictionary is a data structure, which is generally an association of unique keys with some values. One may bind a value to a key, delete a key and loop up for a value by key. Values are not required to be unique. Simple usage example is an explanatory dictionary. In example, words are keys and explanations are values.

Dictionary ADT :

The following are operations performed by Dictionary ADT.

→ Dictionary create ( )

Creates empty dictionary

→ Boolen is empty (Dictionary d)

tells whether the dictionary d is empty.

→ Put (Dictionary d, key K, value V)

associates key K with a value V;

→ Value get (Dictionary d, key K)

returns a value, associated with key K or null, if dictionary contains no such key.

→ remove (Dictionary d, key K)

removes Key K and associated value.

→ destroy (Dictionary d)

destroys dictionary d.

1.1.3 Definition, Dictionary

Q4. Write a program to implement dictionaries using C++.

Ans :

Source Code:

```
# ifndef DICTIONARY_H_INCLUDED
# define DICTIONARY_H_INCLUDED
# include <initializer_list>
# include <algorithm>
# include <vector>
template<typename T, type name U>
class Dictonary
{
Privte :
    Std :: vector <T> Keys;
    Std :: vector <U> Values;
Public :
    Dictionary ( );
    Dictionary (Std:: initializer_list <Std:: pair (T,U>>);
    bool has (T) Const;
Void add (T,U);
T* begin ( );
T* end ( );
U operator [ ] (T);
};
template <type name T, type name U>
T* Dictionary <T,U> :: end ( )
{
return & (Keys [Keys.Size ( ) -1 ] +1;
}
template <type name T, type name U>
Dictionary <T,U> :: Dictionary (Std:: initializer_list <Std :: Pair T, U>> Store) {
for (Std :: pair <T, U> object : store) {
Keys. push_back (Object.first);
values.push_back (object.Second);
}
}
template <type _name T, type name U>
bool Dictionary <T,U> :: has (T target key) const
{
```

```
for (Tcurrent kye: Keys)
{
if (Current Key = = target key){
return true;
}
}
return false;
}
template <type name T, typename U>
Void Dictionary <T,U> :: add (T Key, U value){
Keys.push_back (Key);
Values. Push_back (Value);
}
template <type name T, type name U>
U Dictionary <T,U> :: operator [ ] (T Key){
Unsigned int pos = Std :: find (Keys.begin ( ),
                keys.end ( ), key)-Keys.begin( );
return values [pos];
}
```

1.2. HASHING

1.2.1 REVIEW OF HASHING, HASHING FUNCTION

Q5. Explain briefly about Hashing.

Ans :

Hashing :

→ Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects.

→ In Hashing large keys are converted into small keys by using hash functions. The values are then stored in data structure called hash table.

→ Hashing is implemented in two steps.

a) An element is converted into an integer by using a hash function. This element can be used as an index to store original element, which falls into hash table.

b) The element is stored in hash table where it can be quickly retrieved using hashed key.

hash = hash func(key)

index = hash% array_size

Hash Function :

→ A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into hash table. The values returned by a hash function are called hash values, hash codes, or simply hashes.

→ To achieve a good hashing mechanism. It is important to have a good hash function with following basic requirements.

a) Easy to Compute

b) Uniform distribution

c) Less collisions

Need for good hash function :

→ In order to understand a good hash funct... Assume that you have to store strings in the h... table by using hashing technique {"bcdef... "cdefab", "defabc"}.

→ The index for a specific strings is equal to sum ... ASCII values of characters modulo 599.

→ As 599 is a prime number, it will recuce t... possibility of indexing different strings. The AS... values of a, b, c, d, e and f are 97, 98, 99, 10... 101, 102 respectively. Since all strings cont... same characters with different permulations h... sum will 599.

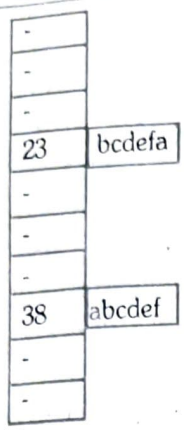
→ The hash function will compute the same ind... for all strings and will be stored in hash table ... following format.

Index				
0				
1				
2	abcdef	bcdefa	cdefab	defabc
3				
4				
-				
-				
-				
-				

→ Here it will take O(n) time to access a specif... string. This shows that hash function is not a goo... hash function.

String	Hash function	Inde
abcdef	(971+982+993+1004+1015+1026)%2069	38
bcdefa	(981+992+1003+1014+1025+976)%2069	23
cdefab	(991+1002+1024+975+986)%2069	14
defabc	(1001+1012+1023+974+996)%2069	11

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	



1.2.2 Couisian Resolution Techniques in Hashing

1.2.2.1 Separate Chaining

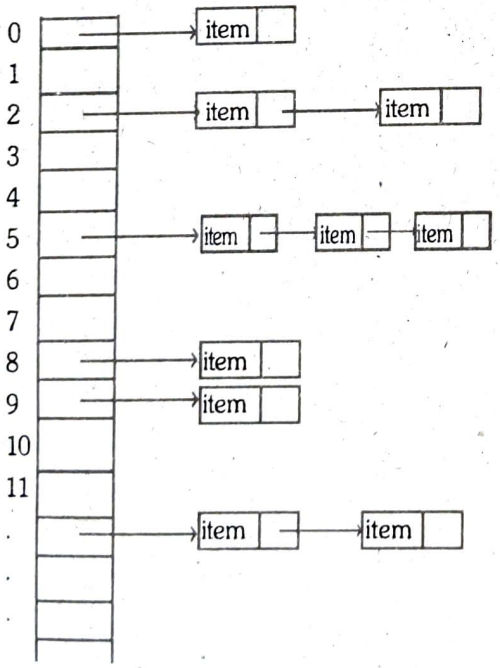
Q6. Write a short note an separate chaining.

Ans :

Separate chaining :

→ Separate chaining is the one of the most commonly used collision resolution technique. It is usually implemented using linked lists.

→ In separate chaining, each element of hash table is a linked list. To store an element in hash table you must insert it into a specific linked list. If there is any collision then store both the elements in same linked list.



→ The cost of a look up is that of scanning the entries of selected linked list for required key. If distribution of keys is sufficiently uniform, then the average cost of a lookup depends only an average number of keys per linked list.

→ For separate chaining the worst -caste scenario is when all the entries are inserted into same linked list. The look up procedure may have to scan all entries, so worst case cost is proportional to number (N) of entries in table.

Implementation :

Insert :

```
Void insert (string S)
{
int index = hsh func(s);
has table [index]. push_back (s);
}
```

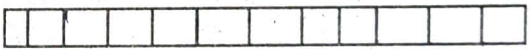
Search :

```
Void search (string s)
{
int index = hash Func(s);
for (int i=0; i<hash table [index].size( ); i++)
{
if (hash table [index] [i] ==s)
{
printf ("/n S is found");
return;
}
}
print f("/n not found");
}
```

Example ;

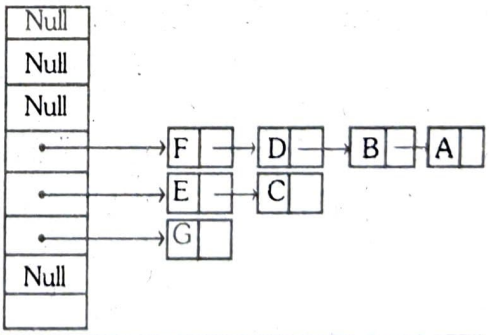
Record : A B C D E F G H

H(k) : 4 4 5 4 5 4 6 8



1 2 3 4 5 6 7 8 9 10 11 12

LIST



Q7. Explain the operations performed by separate chaining.

Ans :

Opeations of Separate Chaining:

The operations such as find, insert and delete can be performed by separate chaining.

a) Gind an Element:

The following algorithm shows the procedure for finding an element.

```
Find OpenHash (key)
Compute Hash Value = hash (Key, Size)
if (list (Hash value). find (Key)
Current = Hash value
print "Element found"
else
print "element Not found"
end if
```

b) Inserting an element :

```
Insert Element (Key, element)
Compute Hash Value = hash (Key, Size)
if (List (Hash Value). find (Key))
while (list)
list = list → next
insert value
else
create Node
insert At begin
end if.
```

c) Remove on element :

Removing an element in separate chaining mechanism involves in similar passion of linked list operations for deletions.

```
Remove Element (Key)
Compute Hash Value = hash (Key)
if (List Hash Value). find (Key))
Current = Hash value
Print "Element found"
Remove Node by
(Delete End, Delete Begin, or Delete Middle)
Else
print "Element not found"
end if.
```

Q8. What are the advantages and disadvantages of Separate chaining?

Ans :

Advantages :

→ Simple and efficient collision resolution.

→ Hash table is capable of holding more number of elements.

→ Deletion of elements is an easy task.

→ There is no need for table size to be a p number.

→ The keys of elements to be hashed may or not be Unique.

Disadvantages :

→ A Separate data structure for cahins must implemented along with code to manage

→ Cost overhead is more because of extra needed for linked list.

→ System may be slow because of frequent of new nodes.

Q9. Given input {3417, 3132, 7122, 5 5344, 6796, 1898} and hash function = x(mod 10); Show the resulting, sepa chaining hash table.

Ans :

Input = {3417, 3132, 7172, 5199, 5344, 6 1898}

h(x) = x mod 10.

h(3417) = 3417 mod 10 = 7

h(3132) = 3132 mod 10 = 2

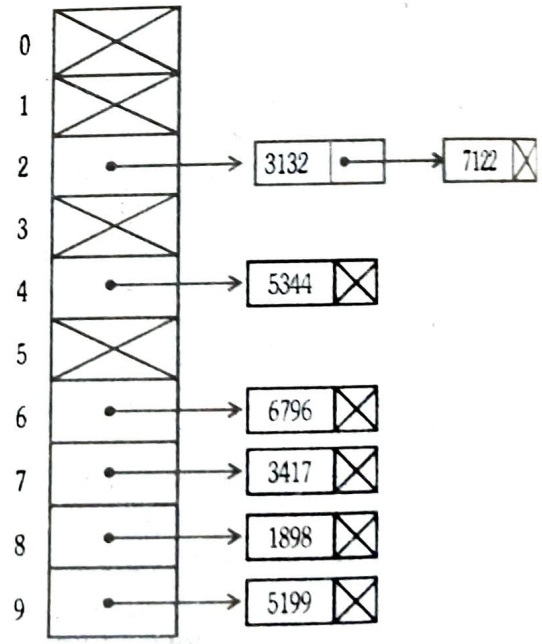
h(7172) = 7172 mod 10 = 2

h(5199) = 5199 mod 10 = 9

h(5344) = 5344 mod 10 =4

h(6796) = 6796 mod 10 = 6

h(1898) = 1898 mod 10 = 8



1.2.2.2. Oepn Addressing

Q10. Discuss about open addressing Hashin

Ans :

Open addressing :

Open addressing hashing is used to resolve collisions that occur in linked lists. When a collision occurs, alternative cells are checked to find an empty cell. where the new element can be inserted.

That is, each of cells, $h_0(x)$, $h_1(x)$, $h_2(x)$, $h_3(x)$.... are checked one by one, until an empty cell is found.

Here,

hi (x) = (hash(x)+ f(i)) mod size of table

f(o) = 0 and

function f = collision resolution strategy.

The open addressing hashing requires a larger table when compared to separate chaining hashing, as it stores all the data within table.

Q11. What are the advantages and disadvantages of open addressing.

Ans :

Advantages :

→ The required amount of memory is fixed in open addressing.

→ It avoids pointers.

→ It avoids sorting of lists in each slot of table.

→ While searching for an object, only items with same hash codes are examined.

→ Inserts the elements, if they are not present, at the beginning or end of list.

→ Rather than marking an element as deleted or replacing it with a dummy element, it is simply can deleted.

Disadvantages :

→ It saves memory only for small entries.

→ This is not a good choice for large elements because all the cache lines will be occupied by these elements wasting a large amount of space on large empty table slots.

→ If the key does not occur in table then the RETRIEVE operation is very inefficient.

→ The size of table limits amount of storage.

→ All the operations degrade to linear_time when load factor becomes large.

→ It puts many constraints on hash table.

→ Delection of an element is difficult because chaining is used on occurance of deletions in hash tables.

Q12. Write the algorithm for open addressing?

Ans :

Insert :

→ Compute position at which item is to be inserted P = H(Key (item))

→ Insert item if that position is empty.

→ If position already contains any other element, compute another position, set p to that position and then repeat steps 2 and 3.

→ End.

Search :

→ Compute position at which item would be found p = x.

→ If P is not empty and contains x, then return value in that position.

→ If p is empty or x is not found at p, compute another position, set p=x and then repeat steps 2 and 3.

→ end.

Delete :

→ Compute position from which an element to be deleted p = x.

→ If P is not empty and p = x then delete x.

→ If P is empty or x is not found at p, then repeat steps 2, 3.

→ end.

1.2.2.3. Linear Probing

Q13. Explain the implementation of Linear Probing.

Ans :

Linear Probing:

Linear Probing is when the interval between successive probes is fixed. Let's assume that the hashed index for a particular entry is index. Th probing sequence for linear probing will be :

index = index % hash Table Size

index = (index +1) % hash Table Size

index = (index +2)% hash Table Size

index = (index +3)% hash Table Size

Source code:

```
# include <iostream>
# include <cstdio>
# include <cstdlib>
using name spze std;
const int T_S = S;
Class Hash Table {
Public :
    int K;
    int V;
Hash Table (int K, int V) {
this → K = K;
this → V = V;
```

```
}
};
Class Del Node : Public Hash table {
private :
      Static Del Node *en;
Del Node ( ): Hash Table (-1, -1) { }
public :
      Static Del Node = *get Node ( ) {
      if (en = NULL)
      en = new Del Node ( ):
      return en;
}
}
Del Node * DelNode :: en = NULL;
class Hash Map Table {
Private :
      Hash Table ** ht;
      Public
      Hash Map Table ( ) {
      ht = new Hash Table * [T_S];
for (int i = 0; i<T_S; i++)
{
h t[i] = NULL;
}
}
int Hash Func (int K) {
return K % T_S;
}
Void Insert (int K, int V){
int hash_val = Hash Func (K);
      int init = -1;
      int delindex =-1;
while (hash_val! = init && (ht[hash_val] ==Del
                  Node :: get Node ( ) //ht
                  [hash_val]! =
                  NULL && ht
                  [hash_val] → k!=k))
{
      if (init == -1)
      init = hash_val;
if (ht [hash_val] == Del Node:: get Node( ))
delindex = hash_val;
hash_val = Hash Func (hash_val+1);
}
if (ht[hash_val] == NULL//hash_val ==init){
if (del index! =-1)
ht[delindex] = new Hash Table (K,V);
else
ht [hash_val] = new Hash Table (K,V);
}
if (init! = hash_val)
{
if (ht[hash_val]! =Del Node :: get Node ( )
if (ht [hash_val]! = NULL){
if (ht [hash_val] → k ==k)
      ht [hash_val] → v=v;
}
} else
ht [hash_val] = new Hash Table (k,v);
}
}
int Search key (int k) {
int Hash_val = Hash Func (k);
int init  =-1
if (init ==-1)
init = hash_val;
hash_val = Hash Func (hash_val+1);
}
if (ht[hash_val] == NULL || hash_val==
return -1;
else
return ht[hash_val] → V;
}
Void Remove (int K)
{
int hash_val = Hash Func (K);
int init = -1;
While (hash_val ! init && (ht[hash_val]
      == Del Node :: getNode ( )
      || ht[hash_val]! = NULL &&
      ht [hash_val] → k!=k))
{
if (init ==-1)
init = hash_val;
hash_val = Hash Func (hash_val+1);
}
if (hash_val! = init && ht [hash_val]! =N
{
delete ht [hash_val];
ht [hash_val] = Del Node :: get Node ( );
}
```

```
}
~ Hash Map Table ( ){
delete [ ] ht;
}
};
int main ( )
{
Hash Map Table hash;
int k, v;
int c;
while (1)
{
Cout <<"1. Insert element into Table"<<end l;
Cout <<"2. Search element from key"<<endl;
Cout <<"3. Delete element at a kay"<<end l;
Court<<"4. Exit" <<end l;
Cout <<"Enter your choice:";
Cin >>c;
Switch (c) {
Case 1 :
Cout <<"Enter element to be inserted :";
cin >>v;
Cout <<"Enter key at which element to be inserted";
Cin >>k;
hash.Insert (k,v);
break;
case 2 :
Cout <<"Enter key of element to be searched"
Cin >>k;
if (hash.Search Key (k) ==-1
{
Cout << "No element found at key" <ck <<end);
continue;
} else {
Cout <<"element at key" <<k <c". ";
Cout <<hash.Search Key (k) <<end l;
}
break;
case 3:
Cout <<"Enter key of element to be deleted:";
Cin >>k:
hash.Remove (k);
break;
case 4 :
    exit (1);
default :
Court <<"/n Enter correct option\n";
}
}
return 0;
}
```

Q14. What are the advantages and disadvantages of Linear probing and its time Complexity?

Ans :

Advantages :

The main advantage of linear probing is

→ It is easy to compute.

Disadvantage :

→ The main problem with linear probing is clustering.

→ Main consecutive elements form groups.

→ Then, it takes time to search an element or to find an empty bucket.

Time Complexity :

Worst time to search an element in linear probing is O(table size).

This is because :

→ Even if there is only one element present and all other elements ar deleted.

→ Then, "deleted" markers present in hash table makes search the entire table.

Q15. Consider an empty hash table insert pairs whose keys in order are 7, 42, 25, 70, 14, 38, 8, 21, 34, 11 and b = 3 buckets. Use Linear Probing.

Ans :

Given that,

b = 13 buckets.

Here bucket represents each position of table where as f(k) is home bucket. The keys are denoted by K. The Keys will be inserted into buckets based one formula k% (i.e. k mod b). The budkets are represented from 0 to 12 as shown below.

0	1	2	3	4	5	6	7	8	9	10	11	12

Insert 7 :

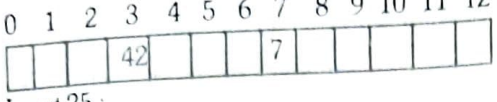
b(k) = k%b = 7% 13 = 7

Key 7 is inserted in bucket 7

0	1	2	3	4	5	6	7	8	9	10	11	12
							7					

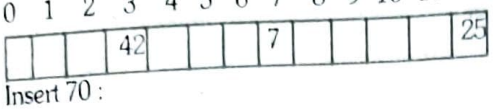
Insert 42 :

b(k) = 42% 13 = 3



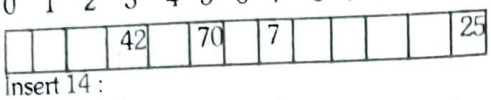
Insert 25 :

b(k) = 25%.13 = 12



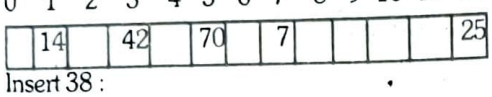
Insert 70 :

b(k) = 70% 13 = 5



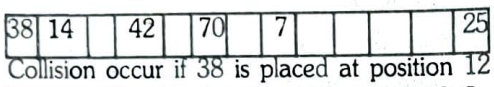
Insert 14 :

b(k) = 14% 13 = 1



Insert 38 :

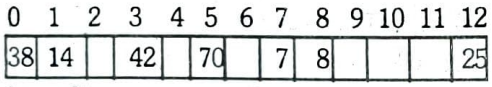
b(k) = 38% 13 = 12



Collision occur if 38 is placed at position 12 because 25 is already present in position 12. So next position after bucket 12 is 0. So key 38 is inserted in position 0.

Insert 8 :

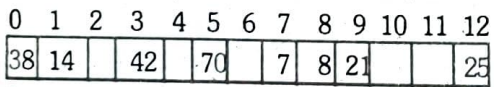
b (k) = 8% 13 = 8



Insert 21 :

b(k) = 21% 13 = 8

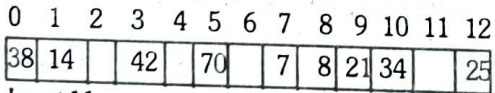
If 21 is placed in position 8 collision occurs. So next bucket after 8 is 9 so insert 21 in 9th position.



Insert 34 :

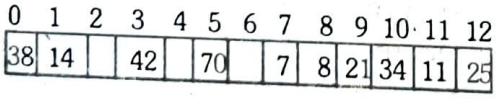
b(k) = 34% 13 = 8

Key 34 is inserted at position 10 because 8 and 9 positions are filled already.



Insert 11 :

b(k) = 11% 13 = 11



1.2.2.4. Quadratic Probing

Q16. What is Quadratic Probing?

Ans :

Quadratic Probing :

Quadratic probing is similar to linear probing the only difference is interval between succes probes or entry slots. Here, when the slot hashed index for an entry record is alre occupied, you must start traversing until you an unoccupied slot. The interval between slo computed by adding the successive value o arbitrary polynomial in original hasehd index

Let us assume that hashed index for an ent index and at index there is an occupied slot probe sequence will be as follows :

index = index % hash Table Size

index = (index + 1^2)% hash Table Size

index = (index + 2^2) % hash Table Size

and so on .....

Q17. Explain the implementation of hash tab with quadratic probing and explain i example.

Ans :

Assumption :

→ There are no more than 20 elements in data

→ Hash function will return an ineger from 0 to

→ Data set must have unique elements.

Implementation :

```
String hash table [21];
int hash Table Size = 21;
Insert :
Void insert (string s)
{
int index = hashFunc(s);
int h =1;
while (hash Table [index]! = " ")
{
index =(index +h*h)% hash table size;
h ++
}
hash Table [index] = s;
```

Search :

```
Void search (string s)
{
int index = hash Func(s);
int h = 1;
```

```
while (hash Table [index]! = S and hash Table
[index]!=" ");
{
index = (index + h*h)% hash Table Size;
h + +;
}
if (hash Table [index] = = s)
printf ("\n S is found");
else
printf ("\n S is not found");
}
}
```

Example :

The quadratic probing function is expressed as $f(i) = i^2$. In this method, when collision occurs, the key is inserted in a cell which is one cell away from collided cell.

Consider the elements = 79, 28, 39, 68, 59 should be inserted into a hash table.

	Empty hash table	After 78	After 28	After 39	After 68	After 59
0				39	39	39
1						
2					68	68
3						59
4						
5						
6						
7						
8			28	28	28	28
9		79	79	79	79	79

→ The performance of linear probing degrades when hash table is nearly full. This situation for quadratic probing is even more serious. The quadratic probing does not guarangee of finding on empty cell once the table gets more than half full.

→ But quadratic probing guarantees of inserting element into table. This is atleast half empty if table size is prime.

Proof :

Assume that table is an (odd) prime greater than 3 of size "Size". To prove this statement we show that first size/z alternatives locations. Where $0 \le j \le [Size/2]$

$h(X) + i^2 = h(X) + j^2$ (mod size)

$i^2 = j^2$ (mod size)

$i^2 - j^2 = 0$ (mod size)

$(i+j)(i-j) = 0$ (mod size)

Since table size is prime, either (i-j) or (i+j) will be equal to 0(mod size). (i-j) can't be zero since i and j are distinct

∴ First [size/2] alternative locations are distinct.

1.2.2.5. Double Hashing

Q18. Explain Double Hashing.

Ans :

Double Hashing:

Double Hashing is similar to linear probing and only difference is interval between successive probes. Here, the interval between probes is computed by using two Hash functions.

index = (index+1* indexH)% hash Table Size;

index = (index+2* index H)% hash Table Size;

and so on .....

Here, index H is hash value that is Computed by another hash fucntion.

Implementation of Double Hashin using C++ :-

```
# include <bits.h>
using name space std;
# define TABLE_SIZE 13
# define PRIME 7
Class Double Hash h
{
    int &hash table;
    int Curr_Size;
public;
bool is Full ( )
{
return (Curr_size = = TABLE_SIZE);
}
int hash 1 (int Key)
{
return (Key % TABLE_SIZE);
}
int hash 2 (int key)
return (PRIME _ (Key% PRIME));
}
Double Hash ( )
{
Hash Table = new int [TABLE_SIZE];
Curr_size = 0;
for (int i=0, i<TABLE_SIZE, i++)
hash Table [i] =-1;
}
```

```
Void insert Hash (int key)
{
if (is Full ( ))
return;
int index = hash1 (Key);
if (hash table [index]!=-1)
{
int index 2 = hash 2 (key);
int i = 1;
while (1);
{
int new Index = (index+i* index2)%
                    TABLE_SIZE;
if (hash table [new Index] ==-1)
{
hash Table [new Index] = Key;
break;
}
i ++;
}
}
else
hash Table [index] = Key;
Curr _size ++;
}
Void display Hash ( )
{
for (int i =0, i<TABLE _SIZE, i++)
{
if (hash Table [i]! = -1)
Cout <<i<<" - - → "
      <<hash Table [i] << end l;
else
Cout <<i << end l;
}
}
};
int main ( )
{
int a [ ] = {19, 27, 36, 10, 64};
int n = size of (a) /size of (a[0]);
Double Hash h;
for (int i = 0, i<n; i++)
h. insert Hash (a[i]);
h. display Hash ( );
return 0;
}
```

1.2.2.6 Rehashing

Q19. Explain Rehashing with example?

Ans :

In open addressing hashing with quadratic probing, if table gets too full then time taken for operations will be too long and insertion operation might fail.

This is certain if there are too many removals intermixed with insertions.

Rehashing technique solves this problem by creating a new table of size twice as big using same hash function.

In this newly created table scans down the entire original hash table by computing new hash value for each element and insert them into this table. This entire operation is called rehashing.

Example :

Consider table size is 7 and

hash function is h(x) = x mod 7

Keys {27, 29, 45, 5}

0	5
1	29
2	
3	45
4	
5	
6	27

If we insert 44 into above table, then it will almost full as shown below

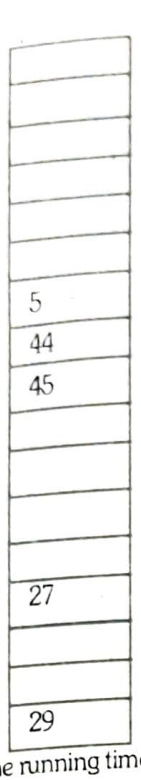
44 mod 7

= 2

0	5
1	29
2	44
3	45
4	
5	
6	27

Since after inserting 44 table is so fully rehashing creates new table of size 17.

Now new hash function becomes h(x) =x mod



The running time of rehashing is O(N).

Rehashing can be implemented in several ways with quadratic probing.

Rehashing is done as soon as table is half full.

Rehashing is done only when insertion fails.

Advantages :

Since hash tables can't be made arbitrary large in Complex program, it frees programmer from worrying about table size.

Rehashing can be used with other data structures like queue.

2.2.7 Extendible Hashing

20. What is extendible Hashing explain with example.

ns :

Extendible Hashing :

In extendible hashing, a find operation requires only two disk accesses and insertion also requires only few disk accesses.

Extendible hashing is nearly similar to B tree. The depth of B-tree is $O(\log_{M/2} N)$. Where, it decreases as M increases. If M is choosen to be so large that the depth of B-tree reduces to 1. In this case, the find operation requires only one disk access because root is stored in main memory.

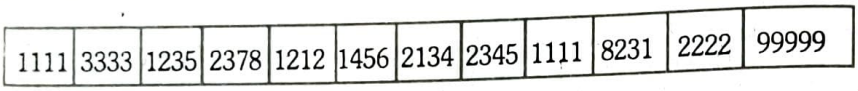
In extendible hashing, the number of bits used by root is represented by D which is known as directory 2^D represents the number of entries in this directory d_L represents number of leading bits that are common in all elements of some leaf L.

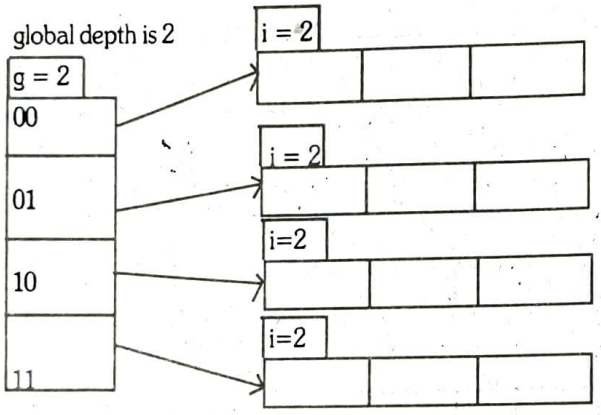
Example :

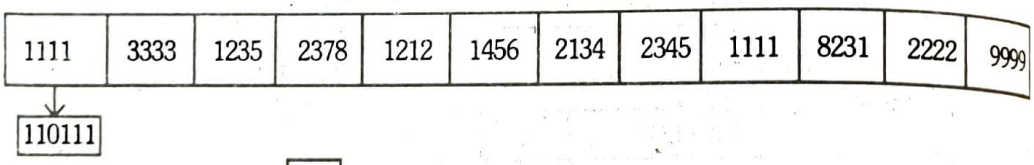
Suppose g= 2 and bucket size = 3

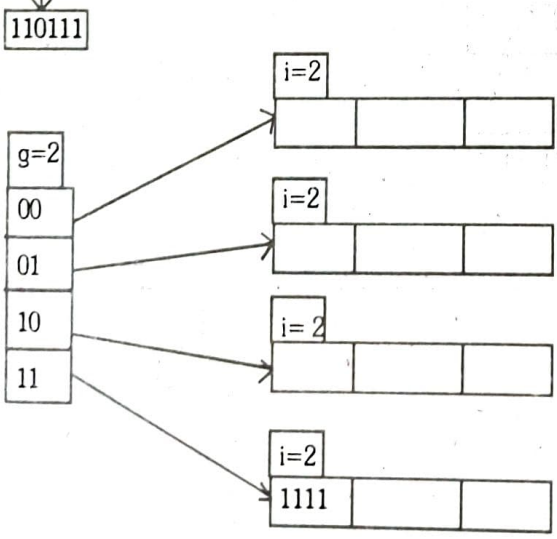
Suppose that we have records with these keys and hash function h(key) = key mod 64 :

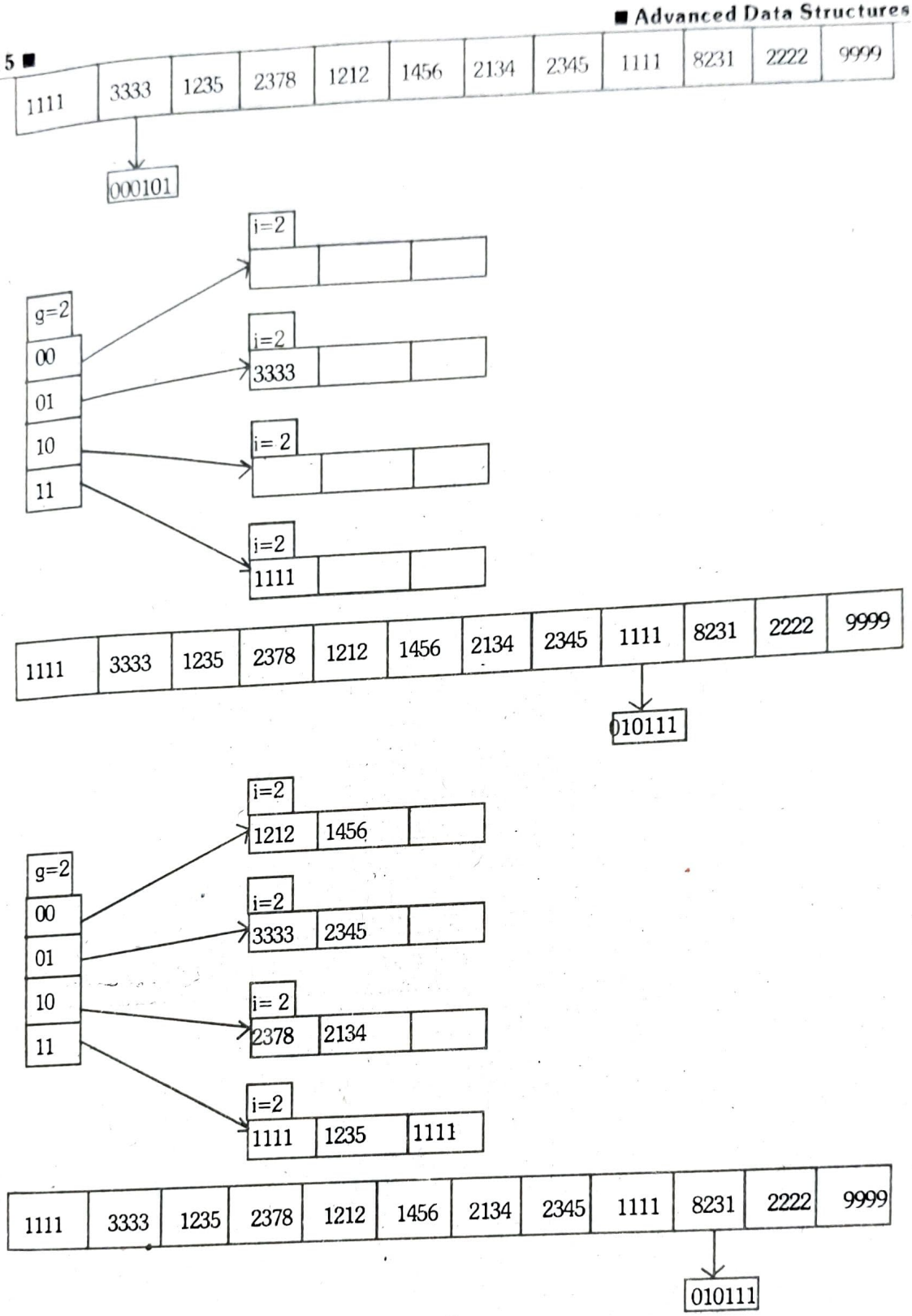
Key	h(key) = Keymod 64	bit pattern
1111	23	010111
3333	5	000101
1235	19	010011
2378	10	001010
1212	60	111100
1456	48	110000
2134	22	010110
2345	41	101001
1111	23	010111
8231	39	100111
2222	46	101110
9999	15	001111

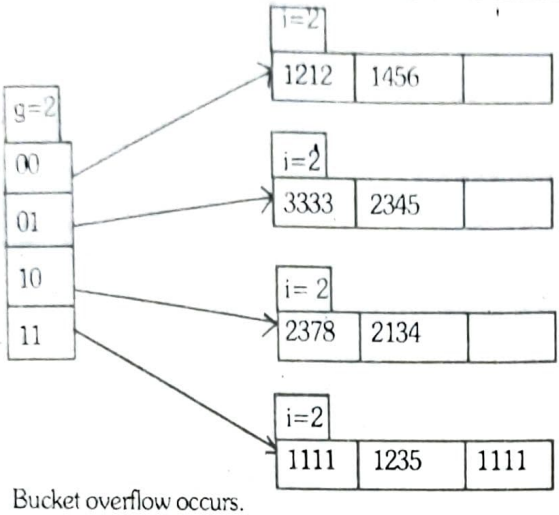




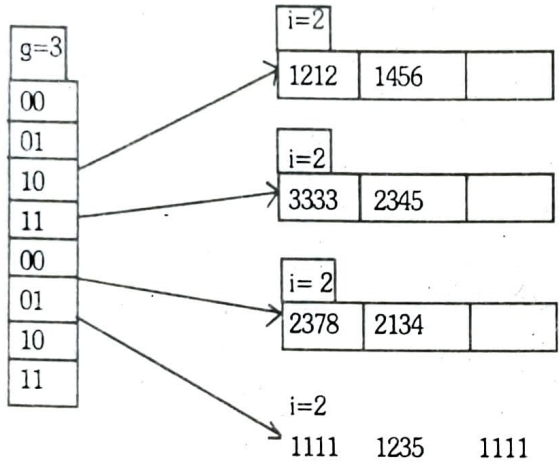


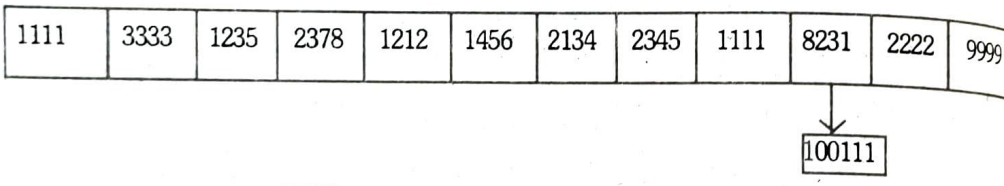


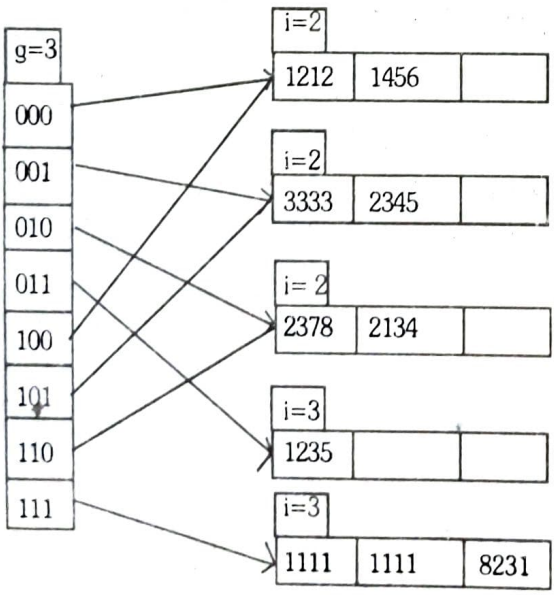


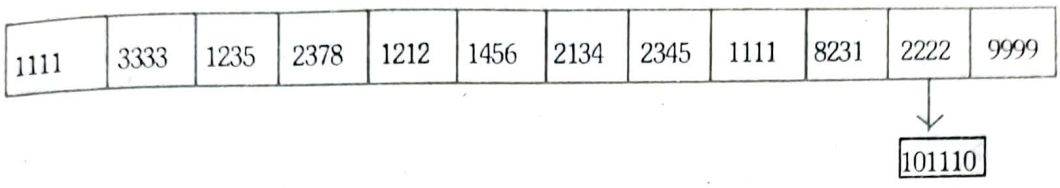


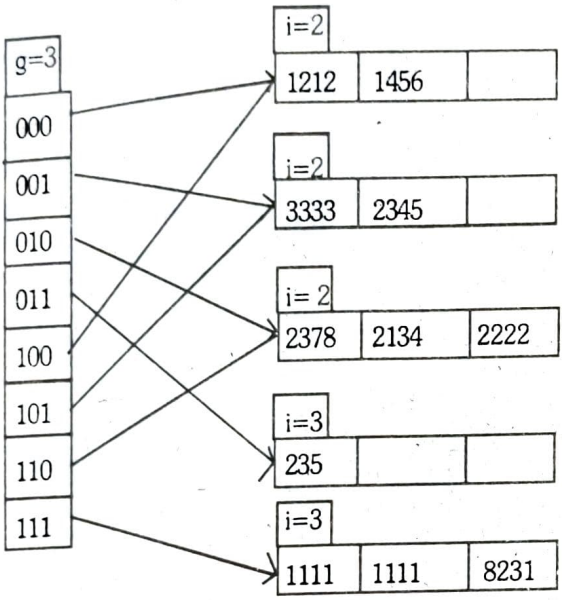
Bucket overflow occurs.











UNIT 2

SKIP LISTS

Short Questions with Answers

Q1. What is Skip list?

Ans :

Skip list is a data structure that is used for storing a sorted list of items with a help of hierarchy of linked lists that connect increasingly sparse subsequence of items.

Q2. What is Complexity of Skip list?

Ans :

	Average case	Worst case
Space	Θ(n)	O(nlogn)
Search	Θ (logn)	O(n)
Insert	Θ(logn)	O(n)
Delete	O(logn)	O(n)

Q3. What are the applications of Skip list?

Ans :

→ Skip lists are used in distributed applications. In distributed systems, the node of skip list represents the computer systems and pointers represent network connection.

→ Skip lists are used for implementing highly scalable concurrent priority queues with less lock contention.

Q4. What are the advantages of Skip lists?

Ans :

(i) Skip list performs well on rapid insertations.

(ii) Easy to implement.

(iii) It allows you to retrieve the next element in constant time.

(iv) This data structure can be modified to some other data structure, such as trees, indexable skip lists, etc.

Q5. What is perfect skip list?

Ans :

→ Perfect skip list in which keys are in sorted order, consists of O(logn) levels.

→ In this each higher level contains 1/2 the elements of level below it.

→ Header and sentinal nodes are in every level.

Q6. What is Deterministic and truncated skip list?

Ans :

The Deterministic skip list is a data structure which implements a dynamic ordered dictionary where as truncated skip list is a skiplist in which the height of skip list is bounded from above by a constant.

Q7. What are the properties of Skip list?

Ans :

Property	Complexity
Insert /remove	O(min(log(m), h)) ∩ Ω (1)
Find text /(previous) equal element	Θ(log (Δ))
Find next /previous element, given an element	Θ(1)
Count number of equivalent elements	Θ(1)
Space	Θ(n)

Q8. What is alternating Skiplist?

Ans :

→ An alternating skip list is an implementation strategy where every skip link is unidirectional, but alternates at every level.

→ Alternation saves memory over bidirectional linking, but sacrifices efficiency.

Essay Questions with Answers

2.1 Need for Randomizing Datastructures and Algorithms

Q1. Explain Randomizing data structures and Algorithms.

Ans :

→ Skip list is the notion of average time complexity used does not depend on probability distribution of keys in input. Instead, it depends on use of random number generation in implemention of insertions to help decide where to place new item.

→ In this context, running time is averaged over all possible outcomes of random numbers used when inserting items.

→ Because they are used extensively in computer games, cryptography, and computer simulations, methods that generate numbers that can be viewed as random numbers are built into most modern computers. Some methods, called pseudo random number generators, generates random like numbers deterministically, starting with an initial number called a seed. Other methods use hardware devices to extract "true" random numbers from nature. In any case, we will assume that our computer has access to numbers that are sufficiently random for our analysis.

→ The main advantage of using randomization in data structure and algorithm design is that structures and methods that result are usually simple and efficient.

2.2. Search and Update Operations on Skip Lists

Q2. Explain Skip List representation briefly with operations?

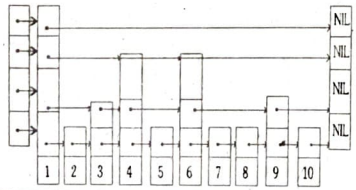
Ans :

Skip List Representation :

→ A skip list is a data structure that is used for storing a sorted list of items with a help of hierarchy of linked lists that connect increasingly sparse subsquences of items.

→ A skip list allows the process of item look up in efficient manner.

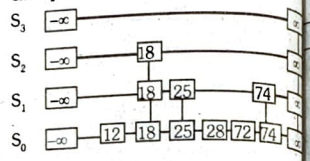
→ Skip list is shown as follows :



Complexity of Skip List :

	Average case	Worst case
Space	O(n)	O(nlogn)
Search	O(logn)	O(n)
Insert	O(logn)	O(n)
Delete	O(log n)	O(n)

example :



Here S_0 has n elements

S_1 has n/2 elements

S_2 has n/4 elements

Operations of Skip List :

a) Search

b) Insert

c) Remove

a) Search :

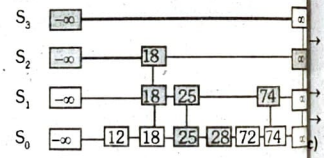
→ Search is done for key K. Then start with P = t top-most, left position node in skip list.

→ Two steps should be considered

(i) if below (p) is null then stop else move to belo

(ii) While Key (P)<k move to right go back to

Search for 27 :



PSeudocode for Searching :

Algorithm : Skip Search (K)

Input : Search Key K

Output : Position P in S such that P has large key less than or equal to K.

P = top - most, left node in S.

While below (P)! = null do

p ← below (P)

While (key (after (p)) ≤ k do

P ← after (p)

return p.

Running Time analysis :

log n levels → O(logn) for going down in Skip list.

at each level, O(1) for moving forward.

Total running tiem : O(log n).

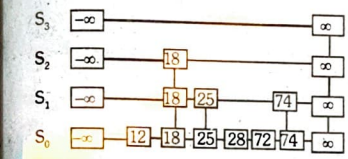
Insertion :

First : identify the place to insert new key k.

Insert new item (k,e) after p.

with probability 50% the new item is inserted in List S_1.

Insert 29.



Pseudo code for Insertion :

Algorithm Skip Insert (K,e)

input : Item (k,e)

Output :

p ← skip search (k)

q ← insert after above (p, null, Item(k,e))

While random ( ) ≤ 50% do

while (above (p) = = null) do

p ← before (p)

p ← above (p)

q ← insert after A above (p,q, Item (k,e))

Running time for Insertion :

Search position for new item (k,e)

O(log n)

Insertion O(1)

Total running time O(log n)

Remove :

Easier than insertion

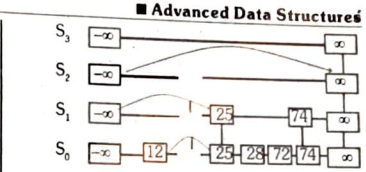
Locate item with Key 'K' to be removed.

If no such element, return NO SUCH KEY.

Otheriwse, remove Item (K,e)

Remove all items found with above (Item (K,e))

Remove 18:



Q3. Write a program to implement Constructor using Skip list.

Ans :

```
template <class K, class E>
Skip list <K,E> :: Skiplist (K large Key, int max
                             Pairs, float prob)
{
Cut off = prob & RAND_MAX;
max level = (int) ceil (logf ((fwat) maxpairs/
                             logf(1/prob))-1;
levels =0;
dSize = 0;
tail Key = large Key;
pair <K,E> tailpair;
tail Pair, first = tailKey;
headerNode = new SkipNode <K,E>
                (tailpair, maxlevel +1);
tail Node = new SkipNode <K,E> (tailPair,0);
last = new SkipNode <K,E> * [maxlevel+1];
for (int i=0; iC=maxlevel; i++)
header Node → next[i] = tailNode;
}
```

Q4. Write a program for erasing a pair from a Skip list.

Ans :

```
template <class K, class E>
Void skiplist <K,E>:: erase (Const K & the Key)
{
if (the Key > = tailKey)
return;
SkipNode <K,E> * the Node = Search (the Key);
if (the Node → element.first! = the Key)
return;
for (int i=0, i<levels && last[i] → next[i] ==
                             the Node; i++)
last [i] → next[i] = the Node → next [i];
while (levels>0 && headerNode → next[levels]
                             == tailNode)
levels --;
```

```
    delete the Node;
    dSize --;
    }
```

Q5. Explain the method sorted chain (K,E)::insert with program.

Ans :

```
template<class K, class E>
void sortedChain <K,E> :: insert (const pair<const K, E> & thePair)
{// Insert thePair into the dictionary. Overwrite existing
// pair, if any, with same key.
pairNode<K,E> *p = firstNode,
                "p = NULL; //*tp trails p
// move tp so that thePair can be inserted after tp
while (p ! = NULL && -> element. first < thePair.first)
{
    tp = p;
    p = p->next;
}
//.chekc if there is a matching pair
if (p ! = NULL && p->element.first == thePair.first)
{// replace old value
    p->element.second = thePair.second;
    return;
}
// no match, set up node for thePair
pairNode <K,E? *newNode = new pairNode <K,E> (thePair, p);

// insert newNode just after tp
if (tp == NULL) firstNode = newNode;
else tp-> next = newNode;
dSize++;
return;
}
```

Q6. Write the program for method sorted chain <K,E> :: erase.

Ans :

```
template<class K, class E>
void sortedChain<K,E> : : erase(const K& theKey)
{// Delete the pair, if any, whose key equals theKey.
    pairNode<K,E> *p = firstNode.
                *tp = NULL; //tp trails p
// search for match with theKey
while (p ! = NULL && p-> element.first < theKey)
{
    tp = p;
p = p-> next;
}
```



```
// verify match
if (p ! = NULL && p->element.first = = theKey)
{// found a match
    // remove p from the chain
    if (tp = = NULL) first Node = p-> next;  // p is first node
    else tp-> next = p -> next;
    delete p;
    dSize --;
  }
}
```

Q7. What is Struct Skip Node? Explain.

Ans :

Struct Skip Node:

→ The header node of a Skiplist structure needs sufficient pointer fields for the maximum number of level chains that might be constrcuted.

→ The tailnode needs no pointer field. Each node that contain a dictionary pair needs an element for the pair and a number of pointer fields that is one more than its level number. The struct skip node program can meet the needs of all kinds of nodes.

```
template <class K, class E>
Struct SkipNode
{
type def pair <const K, E> pair Type;
Pair Type element;
SkipNode<K,E> **next;
SkipNode (Const. pairType & the pair, int size : element (the pair);
{
next = new SkipNode <K,E>* [Size];
}
};
```

Q8. Write a program to search a skiplist and save the last node encountered at each level.

Ans :

```
template<class K, class E>
pair<const K,E>* skilList<K,E> : : find (const K& theKey) const
{// Return pointer to matching pair.
    // Return NULL if no matching pair.
    if (theKey > = tailKey)
        return NULL: // no matching pair possible
// position beforeNode just before possible node with theKey
skipNode<K,E>* beforeNode = headerNode;
for (int i = levels; i > = 0; i --)              // go down levels
    // follow level i pointers
    while (beforeNode -> next[i]->element.first < theKey)
        beforeNode = beforeNode-> next [i];
// check if next node has theKey
if (beforeNode->next [0] ->element.first == theKey)
```

```
return &beforeNode->next[0]->element;
return NULL; // no matching pair
}
```

Q9. Explain the Bounding height in a Skiplist.

Ans :

Bounding height in a Skiplsit:

→ Let us start with determining the expected value of height h of S. the probability that a given item is sorted in a position at level i is equal to probability of getting i consecutive heads when flipping a coin, that is, this probability is 1/2 i. Hence, the probability P_i that level i has at least one item is at most.

$$P_i \le \frac{n}{2^i}$$

for the probability that any one of n different events occurs is, at most, the summ of the probabilities that each occurs.

→ The probability that height h of s is larger than i is equal to probability that level i has at least one item, that is, it is no more than P_i. Tis means that h is larger than, say, 3Logn with probability at most.

$$P_{3\log n} \le \frac{n}{2^{3\log n}} = \frac{n}{n^3} = \frac{1}{n^2}$$

→ More generally, given a constant c>1, h is larger than C logn with probability at most $1/n^{c-1}$. That is, the probability the h is smaller than or equal to Clogn at least $1 - 1/n^{c-1}$. That is, the probability the h is smaller than or equal to cLogn at least $1 - 1/n^{c-1}$. Thus, with high probability, the height h of S is O(logn).

Q10. Discuss Analyzing Search Time in a Skiplist.

Ans :

Analyzing Search Time in a Skip List:

→ Consider the running time of a search in skiplist S, and recall that such a search involves two nested while loops. The inner loop performs a scan forward on a level of S as long as the next key is no greater than search Key K, and outer loop drops down to next level and repeats the scan forward iteration. Since the height h of S is O(log n) with high probability, the number of drop down steps is O(logn) with high probability.

→ So we have yet to bound the number of scan forward steps we make. Let n_i be number of keys examined while scanning forward at level i.

→ The probability that any key is counted in n_i is $1/2$. Therefore, the expected value of n_i is exactly equal to expected number of times we must flip a fair coin before it comes up heads. Let us denote this quantity with e. We have

$$e = \frac{1}{2}.1 + \frac{1}{2}.(1+e)$$

Thus, e=2 and expected amount of time spent scanning forward at any level i is O(1). Since S has O(log n) levels with high probability, a search in S takes expected time O(log n). By similar analysis, we can show that expected running time of an insertion or removal is O(log n).

Space Usage in a Skiplist:

Finally, let us turn to space requirement of a SkipList S. As we observed above the expected number of items at level i is n/2i, which means that expected total number of items in S is $\sum_{i=0}^{n} \frac{n}{2^i} = n\sum_{i=0}^{n} \frac{1}{2^i} < 2n$.

Hence, the expected space requirement of S is O(n).

→ The following table summarizes the performance of a dictionary realized by a skiplist.

Operation	Time
Keys, elements	O(n)
find Element, insertItem, removeElements	O(logn) (expected)
findAllElements, removeAllElements	O(logn+s) (expected)

Q11. Write a program to implement skip list using c++.

Ans :

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <cstring>
#define MAX_LEVEL 6
const float P = 0.5;
using namespace std;
/*
* Skip Node Declaration
*/
struct snode
{
int value;
snode **forw;
snode(int level, int &amp;value)
{
forw = new snode * [level + 1];
memset(forw, 0, sizeof(snode*) * (level + 1));
this-&gt;value = value;
}
~snode()
{
delete [] forw;
}
};
/*
* Skip List Declaration
*/
struct skiplist
{
snode *header;
int value;
int level;
skiplist()
{
header = new snode(MAX_LEVEL, value);
level = 0;
```

```
}
~skiplist()
{
delete header;
}
void display();
bool contains(int &amp;);
void insert_element(int &amp;);
void delete_element(int &amp;);
};
/*
* Main: Contains Menu
*/
int main()
{
skiplist ss;
int choice, n;
while (1)
{
cout<<endl<<"————————————"<<endl;
cout<<endl<<"Operations on Skip list"<<endl;
cout<<endl<<"————————————"<<endl;
cout<<"1.Insert Element"<<endl;
cout<<"2.Delete Element"<<endl;
cout<<"3.Search Element"<<endl;
cout<<"4.Display List "<<endl;
cout<<"5.Exit "<<endl;
cout<<"Enter your choice : ";
cin>>choice;
switch(choice)
{
case 1:
cout<<"Enter the element to be inserted: ";
cin>>n;
ss.insert_element(n);
if(ss.contains(n))
cout&lt;&lt;&quot;Element Inserted&quot;&lt;&lt;endl;
break;
case 2:
cout&lt;&lt;&quot;Enter the element to be deleted: &quot;;
cin&gt;&gt;n;
if(!ss.contains(n))
{
cout&lt;&lt;&quot;Element not found&quot;&lt;&lt;endl;
break;
}
ss.delete_element(n);
if(!ss.contains(n))
cout&lt;&lt;&quot;Element Deleted&quot;&lt;&lt;endl;
break;
case 3:
cout&lt;&lt;&quot;Enter the element to be searched: &quot;;
cin&gt;&gt;n;
if(ss.contains(n))
cout&lt;&lt;&quot;Element &quot;&lt;&lt;n&lt;&lt;&quot; is in the list&quot;&lt;&lt;endl;
else
cout&lt;&lt;&quot;Element not found&quot;&lt;&lt;endl;
case 4:
cout&lt;&lt;&quot;The List is: &quot;;
ss.display();
break;
case 5:
exit(1);
break;
default:
cout&lt;&lt;&quot;Wrong Choice&quot;&lt;&lt;endl;
}
}
return 0;
}

/*
* Random Value Generator
*/
float frand()
{
return (float) rand() / RAND_MAX;
}
```



```
/*
* Random Level Generator
*/
int random_level()
{
static bool first = true;
if (first)
{
srand((unsigned)time(NULL));
first = false;
}
int lvl = (int)(log(frand()) / log(1.-P));
return lvl &lt; MAX_LEVEL ? lvl : MAX_LEVEL;
}

/*
* Insert Element in Skip List
*/
void skiplist::insert_element(int &amp;value)
{
snode *x = header;
snode *update[MAX_LEVEL + 1];
memset(update, 0, sizeof(snode*) * (MAX_LEVEL + 1));
for (int i = level;i &gt;= 0;i--)
{
while (x-&gt;forw[i] != NULL &amp;&amp; x-&gt;forw[i]-&gt;value &lt; value)
{
x = x-&gt;forw[i];
}
update[i] = x;
}
x = x-&gt;forw[0];
if (x == NULL || x-&gt;value != value)
{
int lvl = random_level();
if (lvl &gt; level)
{
for (int i = level + 1;i &lt;= lvl;i++)
{
update[i] = header;
}
level = lvl;
}
x = new snode(lvl, value);
for (int i = 0;i &lt;= lvl;i++)
{
x-&gt;forw[i] = update[i]-&gt;forw[i];
update[i]-&gt;forw[i] = x;
}
}
}
/*
* Delete Element from Skip List
*/
void skiplist::delete_element(int &amp;value)
{
snode *x = header;
snode *update[MAX_LEVEL + 1];
memset (update, 0, sizeof(snode*) * (MAX_LEVEL + 1));
for (int i = level;i &gt;= 0;i--)
{
while (x-&gt;forw[i] != NULL &amp;&amp; x-&gt;forw[i]-&gt;value &lt; value)
{
x = x-&gt;forw[i];
}
update[i] = x;
}
x = x-&gt;forw[0];
if (x-&gt;value == value)
{
for (int i = 0;i &lt;= level;i++)
{
if (update[i]-&gt;forw[i] != x)
break;
update[i]-&gt;forw[i] = x-&gt;forw[i];
}
delete x;
while (level &gt; 0 &amp;&amp; header-&gt;forw[level] == NULL)
{
level--;
}
}
}
/*
* Display Elements of Skip List
```

```
}
~skiplist()
{
delete header;
}
void display();
bool contains(int &amp;);
void insert_element(int &amp;);
void delete_element(int &amp;);
};
/*
* Main: Contains Menu
*/
int main()
{
skiplist ss;
int choice, n;
while (1)
{
cout<<endl<<"————————————"<<endl;
cout<<endl<<"Operations on Skip list"<<endl;
cout<<endl<<"————————————"<<endl;
cout<<"1.Insert Element"<<endl;
cout<<"2.Delete Element"<<endl;
cout<<"3.Search Element"<<endl;
cout<<"4.Display List "<<endl;
cout<<"5.Exit "<<endl;
cout<<"Enter your choice : ";
cin>>choice;
switch(choice)
{
case 1:
cout<<"Enter the element to be inserted: ";
cin>>n;
ss.insert_element(n);
if(ss.contains(n))
cout&lt;&lt;&quot;Element Inserted&quot;&lt;&lt;endl;
break;
case 2:
cout&lt;&lt;&quot;Enter the element to be deleted: &quot;;
cin&gt;&gt;n;
if(!ss.contains(n))
{
cout&lt;&lt;&quot;Element not found&quot;&lt;&lt;endl;
break;
}
ss.delete_element(n);
if(!ss.contains(n))
cout&lt;&lt;&quot;Element Deleted&quot;&lt;&lt;endl;
break;
case 3:
cout&lt;&lt;&quot;Enter the element to be searched: &quot;;
cin&gt;&gt;n;
if(ss.contains(n))
cout&lt;&lt;&quot;Element &quot;&lt;&lt;n&lt;&lt;&quot; is in the list&quot;&lt;&lt;endl;
else
cout&lt;&lt;&quot;Element not found&quot;&lt;&lt;endl;
case 4:
cout&lt;&lt;&quot;The List is: &quot;;
ss.display();
break;
case 5:
exit(1);
break;
default:
cout&lt;&lt;&quot;Wrong Choice&quot;&lt;&lt;endl;
}
}
return 0;
}

/*
* Random Value Generator
*/
float frand()
{
return (float) rand() / RAND_MAX;
}
```



```
/*
* Random Level Generator
*/
int random_level()
{
static bool first = true;
if (first)
{
srand((unsigned)time(NULL));
first = false;
}
int lvl = (int)(log(frand()) / log(1.-P));
return lvl &lt; MAX_LEVEL ? lvl : MAX_LEVEL;
}

/*
* Insert Element in Skip List
*/
void skiplist::insert_element(int &amp;value)
{
snode *x = header;
snode *update[MAX_LEVEL + 1];
memset(update, 0, sizeof(snode*)* (MAX_LEVEL + 1));
for (int i = level;i &gt;= 0;i--)
{
while (x-&gt;forw[i] != NULL &amp;&amp; x-&gt;forw[i]-&gt;value &lt; value)
{
x = x-&gt;forw[i];
}
update[i] = x;
}
x = x-&gt;forw[0];
if (x == NULL || x-&gt;value != value)
{
int lvl = random_level();
if (lvl &gt; level)
{
for (int i = level + 1;i &lt;= lvl;i++)
{
update[i] = header;
}
level = lvl;
}
x = new snode(lvl, value);
for (int i = 0;i &lt;= lvl;i++)
{
x-&gt;forw[i] = update[i]-&gt;forw[i];
update[i]-&gt;forw[i] = x;
}
}
}
/*
* Delete Element from Skip List
*/
void skiplist::delete_element(int &amp;value)
{
snode *x = header;
snode *update[MAX_LEVEL + 1];
memset (update, 0, sizeof(snode*) * (MAX_LEVEL + 1));
for (int i = level;i &gt;= 0;i--)
{
while (x-&gt;forw[i] != NULL &amp;&amp; x-&gt;forw[i]-&gt;value &lt; value)
{
x = x-&gt;forw[i];
}
update[i] = x;
}
x = x-&gt;forw[0];
if (x-&gt;value == value)
{
for (int i = 0;i &lt;= level;i++)
{
if (update[i]-&gt;forw[i] != x)
break;
update[i]-&gt;forw[i] = x-&gt;forw[i];
}
delete x;
while (level &gt; 0 &amp;&amp; header-&gt;forw[level] == NULL)
{
level--;
}
}
}
/*
* Display Elements of Skip List
```

```
*/
void skiplist::display()
{
const snode *x = header-&gt;forw[0];
while (x != NULL)
{
cout &lt;&lt; x-&gt;value;
x = x-&gt;forw[0];
if (x != NULL)
cout &lt;&lt; &quot; - &quot;;
}
cout &lt;&lt;endl;
}
/*
* Search Elemets in Skip List
*/
bool skiplist::contains(int &amp;s_value)
{
snode *x = header;
for (int i = level;i &gt; = 0;i--)
{
while (x-&gt;forw[i] != NULL &amp;&amp; x-&gt;forw[i]-&gt;value &lt; s_value)
{
x = x-&gt;forw[i];
}
}
x = x-&gt;forw[0];
return x != NULL &amp;&amp; x-&gt;value == s_value;
}
```

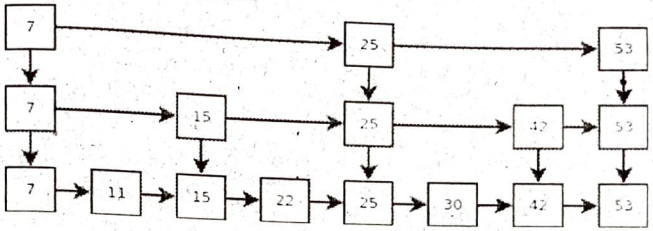
Q12. What are the advantages of skip list and explain the naïve implementation of skip list?

Ans :

Advantages

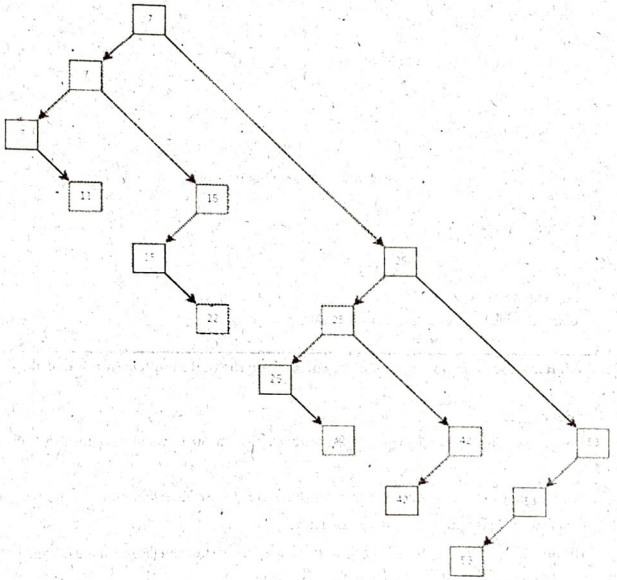
- Skip lists perform very well on rapid insertions because there are no rotations or reallocations.
- They're simpler to implement than both self-balancing binary search trees and hash tables.
- You can retrieve the next element in constant time (compare to logarithmic time for inorder
- traversal for BSTs and linear time in hash tables).
- The algorithms can easily be modified to a more specialized structure (like segment or range
- "trees", indexable skip lists, or keyed priority queues).
- Making it lockless is simple.
- It does well in persistent (slow) storage (often even better than AVL and EH).

A naïve (but common) implementation



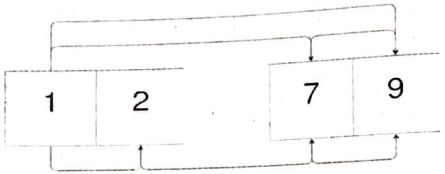
Our skip list consists of (in this case, three) lists, stacked such that the n'th list visits a subset of the node the n - 1'th list does. This subset is defined by a probability distribution, which we will get back to later.

If you rotate the skip list and remove duplicate edges, you can see how it resembles a binary search tree:



Say I wanted to look up the node "30", then I'd perform normal binary search from the root and down. Due to duplicate nodes, we use the rule of going right if both children are equal:

Our nodes are simply fixed-size blocks, so we can keep them data local, with high allocation/ deallocation performance, through linked memory pools (SLOBs), which is basically just a list of free objects. The order doesn't matter. Indeed, if we swap "9" and "7", we can suddenly see that this is simply a skip list



We can keep these together in some arbitrary number of (not necessarily consecutive) pages, drastically reducing cache misses, when the nodes are of smaller size.

Since these are pointers into memory, and not indexes in an array, we need not reallocate on growth. We can simply extend the free list.

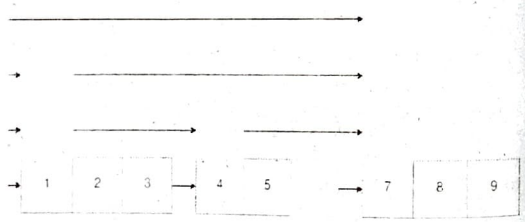
Flat arrays

If we are interested in compactness and have a insertion/removal ratio near to 1, a variant of linked memory pools can be used: We can store the skip list in a flat array, such that we have indexes into said array instead of pointers.

Unrolled lists

Unrolled lists means that instead of linking each element, you link some number of fixed-size chunks that contains two or more elements (often the chunk is around 64 bytes, i.e. the normal cache line size). Unrolling is essential for a good cache performance. Depending on the size of the objects you store, unrolling can reduce cache misses when following links while searching by 50-80%.

Here's an example of an unrolled skip list:



The gray box marks excessive space in the chunk, i.e. where new elements can be placed. Searching is done over the skip list, and when a candidate is found, the chunk is searched through linear search. To insert, you push to the chunk (i.e. replace the first free space). If no excessive space is available, the insertion happens in the skip list itself.

Note that these algorithms requires information about how we found the chunk. Hence we store a "back-look", an array of the last node visited, for each level. We can then backtrack if we couldn't fit the element into the chunk.

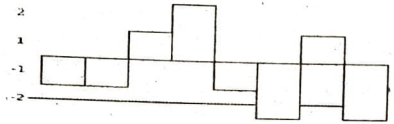
We effectively reduce cache misses by some factor depending on the size of the object you store. This is due to fewer links need to be followed before the goal is reached.

Self-balancing skip lists

Various techniques can be used to improve the height generation, to give a better distribution. In other words, we make the level generator aware of our nodes, instead of purely random, independent RNGs.

Self-correcting skip list

The simplest way to achieve a content-aware level generator is to keep track of the number of node of each level in the skip list. If we assume there are n nodes, the expected number of nodes with level l is $2^l n$. Subtracting this from actual number gives us a measure of how well-balanced each height is:



When we generate a new node's level, you choose one of the heights with the biggest under-representation (see the black line in the diagram), either randomly or by some fixed rule (e.g. the highest or the lowest).

Perfectly balanced skip lists

Perfect balancing often ends up hurting performance, due to backwards level changes, but it is possible. The basic idea is to reduce the most over-represented level when removing elements.

2.3 Deterministic Skiplist

Q14. Explain Deterministic Skiplist

Ans :

Deterministic Skiplist:

The deterministic Skiplist is a data structure which implements a dynamic ordered dictionary. The truncated skiplist is a skiplist in which height of skiplist is bounded from above by a constant.

Properties :

Let $n \in N$ be number of stored elements, $m \in N$ be number of unique stored elements and $\Delta \in N$ be link-distance, along unique elements, between a given stored element and searched element. Let $h \in NU\{\infty\}$ be maximum height of skliplist. The implementation of the deterministic skiplist in Pastel has following properties.

Property	Complexity
Insert /remove	$O(\min(\log(m), h)) \cap \Omega(1)$
Find text /(previous) equal element	$\Theta(\log(\Delta))$
Find next /previous element, given an element	$\Theta(1)$
Count number of equivalent elements	$\Theta(1)$
Space	$\Theta(n)$

UNIT 3

TREES

Short Questions with Answers

Q1. Write a short note on binary search tree.

Ans :

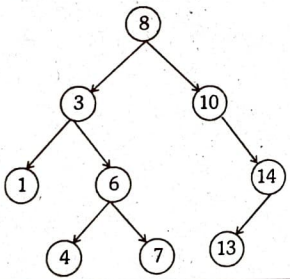
Binary Search tree :

Binary search tree is a node based binary tree data strucutre which has following properties.

(i) The left subtree of node contains only nodes with keys lesser than node's key.

(ii) Right subtree of a node contains only nodes with keys greater than node's key.

(iii) Left and right each must also be binary search tree.



Q2. Define height of tree. What is height of binary search tree.

Ans :

The height of any node is the longest path from the node to any leaf.

The height of binary search tree when 'n' elements are inserted is n. The height in worst case is O(logn).

Q3. List out the operations performed an binary search tree.

Ans :

The various operations performed on binary search tree are,

a) Insert

b) Delete

c) Search

Q4. What are time and space complexities of binary search tree?

Ans :

Time Complexity :

a) Insertion :

Average case - O(logn)

Worst case - O(n)

b) Deletion :

Average case - O(logn)

Worst case - O(n)

c) Searching

Average case - O(logn)

Worst case - O(n)

Space Complexity :

The space complexity for all operations in binary search tree is O(n) in both average and worst cases.

Q5. Define AVL tree.

Ans :

AVL tree is defined as the height of two child subtrees of any node differ by at most one.

Q6. Define Balance factor.

Ans :

The balance factor is defined as difference between the height's of node's left and right subtree.

Q7. What are the types of rotations used to balance AVL trees?

Ans :

Different types of rotations are

a) LL (Left - Left) rotation

b) RR (Right - right) rotation

c) LR (Left-right) Rotation

d) RL (Right-left) rotation.

Q8. Write the time and space complexity of AVL tree.

Ans :

Time Complexity :

The time complexity of AVL tree is O(logn) for both Average case & Worst case for all three operations.

Space Compelxity :

Space Complexity AVL tree is O(n) in both average & worst case.

Q9. Define Red-black tree.

Ans :

A red black tree is a binary search tree in which each node is coloured red or black such that. The root is black. The children of red node are black.

Q10. Define splay trees.

Ans :

A splay tree is self-adjusting binary search tree with additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look up and removal in O(log n)

Q11. Differences between Binary Search tree and AVL Tree.

Ans :

Binary Search Tree :

→ It cannot balance

→ height of binary search tree is O(logn)

→ Time complexity is O(logn) in average case and O(n) in worst case.

→ Implementation is easy.

AVL tree :

→ It can balance.

→ height of AVL tree is O(logn)

→ O(logn) in both cases.

→ Implementation is difficult.

Q12. Compare and Contast red-black tree and splay tree.

Ans :

Red-black trees

→ Red black trees are balanced BST's.

→ The height of red-black trees is 2log(n+1) dependent on tree.

→ Time complexity of Red-black tree is O(logn) in both average and worst case.

Splay tree :

→ Splay trees are in balanced BST's

→ The height of splay trees is linear which is dependent on tree.

→ Time complexity of splay trees is O(logn) in average case and a mortized O(logn) in worst case.

Q13. Differentiate binary search tree and red black tree.

Ans :

Binary Search tree :

→ Binary search tree is an imbalanced tree.

→ Worst case time Complexity is O(n).

→ height of BST in worst case is N.

Red-black tree :

→ Red black tree is a self balancing Binary search tree.

→ The worst case time complexity is O(log N)

→ The height of red black tree in worst case is log N.

Q14. Explain the recursive implementation of pre order traversal operation on Binary Search tree.

Ans :

Recursive implementation :

```
template <class T>
Void BSTree <T> :: R_preorder (BSTreeNode
<T>* node)
{
if (node! = 0)
{
visit (node);
R_Preorder (node → left child);
R_Preorder (node → right child);
}
}
```


Essay Questions with Answers

3.1 Binary Search Trees.

Q1. Write briefly about binary search tree and its height:

Ans :

Binary Search Tree :

A Binary Search tee is a tree in which all nodes follows the below mentioned properties.

→ The left sub-tree of a node has a key less than or equal to its parent node's key.

→ The right sub-tree of a node has a key greater than to its parent node's key.

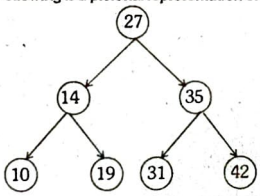
The Binary Search tree divides all its sub trees into segments; the left sub-tree and right sub tree and can be defined as left-subtree (Keys) ≤ node(key) ≤ right_subtree (Keys).

Representation :

→ Binary search tree is a collection of nodes arranged in a way where they maintain BST properties.

→ Each node has a key and an associated value.

→ Following is a pictorial representation of BST.



→ We observe that the root node key (27) has all less-valued keys on left sub tree and higher valued keys on right sub tree.

Basic Operations :

→ Following are the basic operations of a tree.

a) Search : Searches an element in a tree.

b) Insert : Inserts an element in tree.

c) Pre-order Traversal : Traverses a tree in pre order

d) In-order Traversal : Traverses a tree in inorder manner.

e) Post-order traversal : Traverses a tree in post-order manner.

Height of Binary Search tree :

The height of any node is the longest path from the node to any leaf.

The height of binary search tree when 'n' elements are inserted is 'n'. The height in worst case is O(log n).

Q2. Explain the implementation of Binary Search tree using C++.

Ans :

```
# include <iostream>
# include <cstdlib>
using namespace std;
Struct node
{
    int info;
Struct node *left;
struct node *right;
}* root;
class BST
{
public :
Void find (int, node**, node **);
Void insert (node*, node*);
Void del (int);
Void Case_a(node *, node *);
Void Case_b (node*, node *);
Void Case_c(node*, node*);
Void Preorder (node *);
Void inorder (node *);
Void post order (node *);
Void display (node *, int);
BST ( )
{
root = NULL;
}
};
int main ( )
{
int choice, num;
BST bst;
node *temp;
while (1)
{
Cout <<"----------" <<end l;
Cout <<"Operations on BST" <<end l;
Cout <<"----------" <<end l;
Cout <<"1. Insert Element" <<end l;
Cout <<"2. Delete Element" << endl;
```

```
Cout <<"3. Inorder traversal" <<end l;
Cout <<"4. Preorder traversal" << end l;
Cout <<"5. Post order traversal" << end l;
Cout <<"6. Display"<< end l;
Cout <<"7. Quit" << end l;
Cout <<"Enter your choice:";
Cin >> choice;
Switch (choice);
{
Case 1 :
temp = new node;
Cout <<"Enter number to be inserted :";
Cin>>temp → info;
bst.insert (root, temp);
Case 2:
if (root == NULL)
{
Cout <<"Tree is empty, nothing to delete"<<end l;
Continue;
}
Cout <<"Enter number to be deleted:";
Cin>>num;
bst. del (num);
break;
Case 3 :
Cout << "Inorder Traversal of BST :"<<end l;
bst. inorder (root);
Cout << end l;
break;
Case 4 :
Cout << Preorder Traversal of BST:"<<end l;
bst.preorder (root);
Cout << end l;
break;
Case 5 :
Cout <<"Post order Traversal of BST". <<end l;
bst.postorder (root);
Cout <<end l;
break;
Case 6 :
Cout << "Display BST:"<<end l;
bst.display (root, 1);
cout << end l;
break,
Case 7 :
exit (1);
default:
Cout << "Wrong choice" << end l,
}
}
Void BST : find (int item, node** par, node**loc
{
node *ptr, *ptr save;
if (root == NULL)
{
* loc = NULL
* par = NULL;
return;
}
if (item == root → info)
{
* loc = root;
* par = NULL;
return;
}
if (item< root → info)
ptr = root → left,
else
pt = root → right,
ptr save = root;
while (ptr! = NULL)
{
if (item = ptr → info)
{
* loc = ptr;
* par = ptrsave;
return;
}
ptrsave = ptr;
if (item <ptr → info)
ptr = ptr → left;
else
ptr = ptr → right,
}
* loc = NULL;
* par = ptr save;
}
Void BST :: insert (node*tree, node* newnode)
```



```
{
if (root = = NULL)
{
root = newnode;
root → info = new node → info;
root → left = NULL;
root → right = NULL;
Cout <<"Root Node is Added"<< end l;
return;
}
if (tree → info == new node → info)
{
Cout <<"Element already in tree" << end l;
return;
}
if (tree → info > new node → info)
{
if (tree → left! = NULL)
{
insert (tree → left, new node);
}
else
{
tree → left = new node;
(tree → left) → left = NULL;
(tree → left) → right = NULL;
Cout <<"Node Added to Left" << end l;
return;
}
}
else
{
if (tree → right! = NULL)
{
insert (tree → right, new node);
}
else
{
tree → right = new node,
(tree → right) → left = NULL;
(tree → right) → right = NULL;
Cout <<"Node Added To Right" << end l;
return;
}
}
}
Void BST :: del (int item)
{
node * parent, * location;
if (root == NULL)
{
Cout <<"Tree Empty" << end l;
return;
}
find (item, & parent, & location);
if (location == NULL)
{
Cout <<"Item not present intree"<< end l;
return;
}
If (location → left == NULL && location → right == NULL
case_a (parent, location);
if (location → left == NULL && location → right ==NULL);
case_b (parent, location);
if (location → left ==NULL && location → right! == NULL)
case_b (parent, location);
if (location → left! =NULL && location → right! = NULL)
case_c (parent, location);
free (location);
Void BST :: Case_a (node*par,node* loc)
{
if (par == NULL)
{
root = NULL;
}
else
{
if (loc == par → left)
par → left = NULL;
else
par → right = NULL;
}
}
Void BST :: case_b (node*par, node*loc)
{
node* child;
if (loc → left!=NULL)
```

```
child = loc → left;
else
child =loc → right;
if (par==NULL)
{
root = child;
}
else
if (loc==par → left)
par → left = child;
else
par → right = child;
}
}
Void BST :: Case_C (node* par, node* loc)
{
node* ptr, *ptr save, * suc, *parsucj
ptr save = loc;
ptr = loc → right;
while (prt → left! = NULL)
{
ptr save = ptr;
ptr = ptr → left;
}
suc = ptr;
par suc = ptrsave;
if (suc → left==NULL && suc → right==NULL)
case_a (parsuc,suc);
else
case_b (parsuc, suc);
if (par ==NULL)
{
root = suc
}
else {
if (loc == par → left)
par → left = suc;
else
par → right = suc;
}
suc → left = loc → left;
suc → right = loc → right;
void BST :: preorder (node * ptr)
{
if (root ==NULL)
{
Cout <<"Tree is empty" <<end l;
return;
}
if (ptr! = NULL)
{
cout<<ptr→ info<<"  ";
preorder (ptr → left);
preorder (ptr → right);
}
}
Void BST : in order (node *ptr)
{
if (root == NULL)
{
Cout <<"Tree is empty" << end l;
return ;
}
if (ptr! = NULL)
{
inorder (ptr → left);
Cout <<ptr → info<"  ";
inorder (ptr → right);
}
}
Void BST :: Post order (node*ptr)
{
if (root == NULL)
{
Cout <<"Tree is empty"<< end l;
return;
}
if (ptr! = NULL)
{
post order (ptr → left);
post order (ptr → right);
cout <<ptr → infor <<"  ";
}
}
Void BST :: display (node*ptr, int level)
{
int i;
if (ptr! = NULL)
```



```
{
display (ptr → right, level +1);
Cout << end l;
if (ptr == root)
Cout <<"Root → :  ";
else
{
for (i=0; i<level; i++)
Cout << "  ";
}
cout << ptr → info;
display (ptr → left, level+1);
}
}
```

Q3. Explain in detail the operations performed on a binary search tree along with time and space complexity.

Ans :

Operations of Binary Search tree :

→ The various operations performed on binary search tree are

a) Insertion

b) Deletion

c) Searching

Insertion :

→ When ever an element to be inserted, first locate its proper location.

→ Start searching from root node, then if data is less than key value, search for empty location in left subtree and insert the data.

→ Otherwise, search for empty location in right subtree and insert the data.

Algorithm for Insertion :

```
Void insert (int data)
{
Struct node *tempNode = (Struct node *)
        malloc (size of (struct node))
Struct node * Current;
Struct node * parent;
temp node → data = data;
temp node → left child = NULL;
temp node → right child = NULL;
if (root == NULL)
{
root = tempNode;
} else
{
current = root;
parent = NULL;
while (1)
{
parent = current;
if (data < parent → data)
{
Current = Current → Left child
if (Current == NULL)
{
parent → left child = temp Node;
return;
}
}
else
{
Current = Current → right child,
if (Current == NULL)
{
Parent → right child = tempNode;
return;
}
}
}
}
}
```

Search :

→ Whenever an element is to be searched, start searching from the root node.

→ Then if the data is less than the Key value, search for the element in left Subtree. Otherwise, search for element in right subtree.

Algorithm :

```
Struct node *Search(int data)
{
Struct node * current = root;
printf ("Visiting elements :");
while (current → data! = data)
{
if (current!= NULL)
{
printf ("%d, current → data);
if (current → data>data)
{
current = current → left child;
} else
```

```
{
current = current → right child;
}
if (current == NULL)
{
return NULL;
}
}
}
return current;
}
```

Delete :

→ When we delete a node, three possibilities arise.

a) Node to be deleted is leaf.

b) Node to be deleted has only one child.

c) Node to be deleted has two children.

Algorithm :

```
Struct node * deleteNode (Struct * root, int key)
{
if (root = = NULL)
return root;
if (key < root → key)
root → left = delete Node (root → Left, key);
else if (Key > root → Key)
root → right = delete Node (root → right, key);
else
{
if (root → left = = NULL)
{
Struct node * temp = root → right;
free (root);
return temp;
}
else if (root → right == NULL)
{
Struct node * temp = root → left;
free (root);
return temp;
}
Struct node * temp = min value Node (root → right);
root → key = temp → key;
root → right = delete Node (root → right, temp → key)
}
return root;
}
```

Time & Space Complexity :

→ The time Complexity of Binary search tree is O(logn) for average case and O(n) for worst case in all three operations.

→ The space complexity of all operations in binary search tree is O(n) in both average and Worst case.

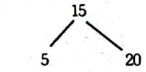
Q4. Start with an empty binary tree, insert the values 15, 5, 20, 14, 30, 22, 2, 4, 3, 7, 9, 18 and delete values 2, 4, 5 in the order and draw tree after deletion.

Ans :

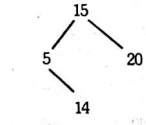
Insert 15

Insert 5

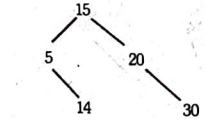
insert 20



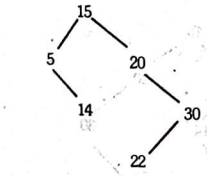
insert 14



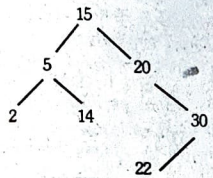
insert 30



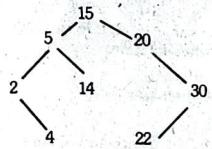
insert 22



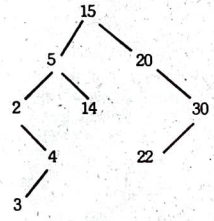
insert 2



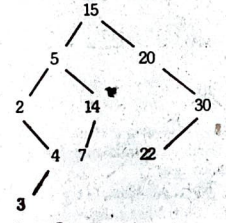
insert 4



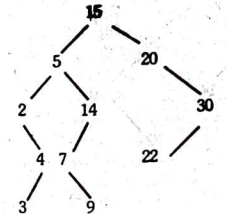
insert 3



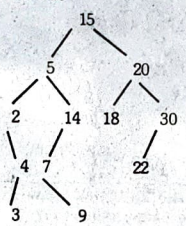
insert 7



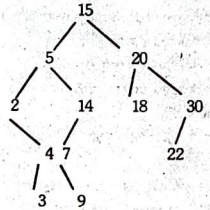
insert 9



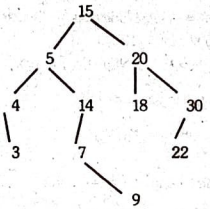
insert 18



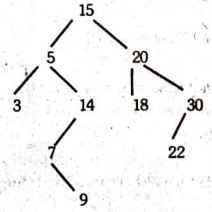
The final binary search tree after insertion of all elements is



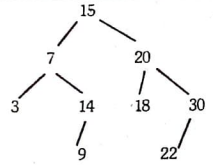
After removal of 2, the tree is,



After removal of 4, the tree is,



After removal of 5, the tree is,



Q5. What are the additional operations performed on binary search tree?

Ans :

The additional operations performed on binary search tree are

a) Three -way join (small, big, mid)

b) Two - way join (small, big)

c) Split (small, big, mid)

a) Three-way join :

These are the keys used to create new binary trees. It is being assumed that, all keys 'small' are less than midkey and all keys 'big' are larger than mid key. This join makes both small and big tress.

b) Two way join (small, big) :

It is being assumed that all the keys that re available in small are smaller than all keys in big. This join makes small and big trees empty.

c) Split :

In this operation BSTree is a binary search tree which is used to split into three parts.

(i) Small

(ii) Mid

(iii) Big

(i) Small :

Small is a binary search tree that contains all original pairs of BSTree.

(ii) Mid :

Mid is a pair of keys contained in BSTree.

(iii) Big :

Big is another binary search tree that contains the keys which are greater than mid.

Q6. Given a BSTree write an alogirthm to tree check whether tree is an AVL tree or not and disucss runtime complexity.

Ans :

(i) Initially check whether root node is NULL if (!root) return 0;

(ii) Now check whether left subtree is AVL

root = Left

repeat step 1 to 5

if (left == -1)

return left;

(iii) root = right

repeat step 1 to 5

if (right = = -1)

return right;

(iv) if (abs (left - right) > 1)

return -1;

(v) x = (left > right? left : right) + 1;

return x;

complexity of this algorithm is O(logn)

3.2 AVL TREES

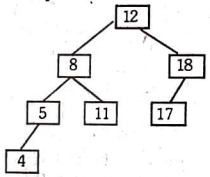
Q7. Define AVL tree and discuss various rotations on AVL tree and height of AVL tree.

Ans :

AVL Tree :

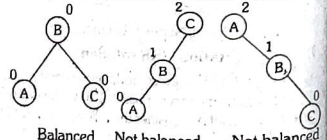
AVL tree is a self balancing Binary Search tree where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Example :



→ AVL tree is named after their inventor Adelson, Velski & Landis, AVL trees are height balancing binary search tree.

→ AVL tree checks the height of left and right sub trees and assures that the difference is not more than 1. This difference is called Balance factor.



→ In second tree, the left subtree of C has height 2 and right subtree has height 0, So the difference is 2.

→ In third tree, the right subtree of A has height 2 and left is missing, so it is 0, and difference is 2. AVL tree permits difference to be only 1.

Balance factor = height (Left-subtree) - height (right subtree)

AVL Rotations :

→ AVL tree perform the following 4 rotations.

a) Left rotation

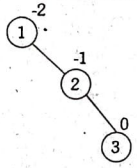
b) Right rotation

c) Left-Right rotation

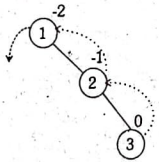
d) Right - Left roation

a) Left rotation : (LL Rotation)

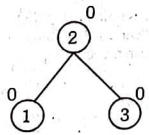
→ In LL Rotation, every node moves one position to left from current position. To underst and LL rotation, let us consider following example.



Tree is imbalanced.



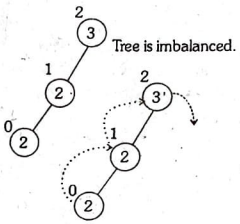
To make balanced we use LL Rotation which moves nodes one position to Left.



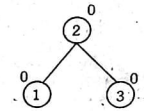
After LL Rotation tree is balanced.

b) Right rotation : (RR rotation)

→ In RR Rotation, every node moves one position to right from current position. To understand RR rotation, let us consider the following example.



To make balanced we use RR rotation which moves nodes one position to right.



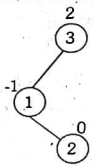
After RR Rotation Tree is balanced.

c) Left-Right Rotation (LR Rotation):

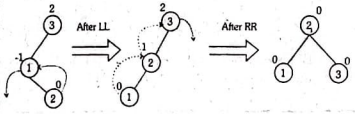
→ The LR Rotation is sequence of single left rotation followed by single right rotation.

→ In LR Rotation, at first every node moves one position to left and one position to right from current position.

→ Let us consider the following example.



Tree is imbalanced.



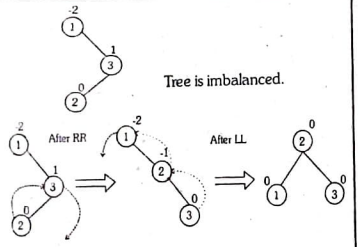
LL Rotation RR Rotation After LR Rotation Tree is balanced

d) Right Left Rotation (RL Rotation):

→ The RL rotation is sequence of single right rotation followed by single left rotation.

→ In RL, at first every node moves one position to left from current position.

→ Consider the following example.



RR Rotation LL Rotation After RL Rotation

Tree is Balanced

Height of an AVL Tree :

Let N_h denote minimum number of nodes is an AVL tree with height h, then,

$N_h = N_{h-1} + N_{h-2} + 1$

Here, N_{h-1} is the height of sub tree in worst case and N_{h-2} is height of other.

$= 2N_{h-2} + 1$

$= 1 + 2(1 + 2N_{h-4})$

$= 1 + 2 + 2^2 N_{h-4}$

$= 1 + 2 + 2^2 + 2^3 + \ldots + 2^{h2}$

$= 2^{h2} - 1$

Hence,

$2^{h1} - 1 < n$

$\frac{h}{2} < \log_2(n+1)$

$h < 2\log_2(n+1)$

Q8. Explain in detail about operations of AVL tree.

Ans :

Operations of an AVL tree :

→ The following operations are performed on AVL tree ....

1. Insertion
2. Search
3. Deletion

1. Insertion :

→ In an AVL tree, the insertion operation is performed with O(logn) time complexity. In AVL tree, new node is always inserted as leaf node. The insertion is performed as follows :

Step 1: Insert new element into tree using BST insertion logic.

Step 2 : After insertion check the balance factor of every node.

Step 3 : If the balance factor of every node is 0 or 1 or -1 then go for next operation.

Step 4 : If Balance factor of any node is other than 0 or 1 or -1 then tree is said to be imbalanced. Then perform rotation to make it balanced and go for next operation.

2. Search :

→ In an AVL tree, the search operation is performed with O(logn) time complexity. The search operation in AVL tree is similar to search operation in binary search tree (BST).

→ Following steps should be performed to search an element in AVL tree.

Step 1 : Read search element from user.

Step 2 : Compare search element with value of root node in tree.

Step 3 : If both are matched, then display "Given node found!!!" and terminate function.

Step 4 : If both are not matched then check whether search element is smaller or larger than that node value.

Step 5 : If search element is smaller, then continue the search process in left subtree.

Step 6 : If search element is larger, then continue the search in right subtree.

Step 7 : Repeat the same until we find exact element or until search element is compared with leaf node.

Step 8 : If we reach to node having value equal to search value, then display "Element is found" and terminate the function.

Step 9 : If we reach to leaf node and if it is also matched with search element, then display "Element is not found" and terminate the function.

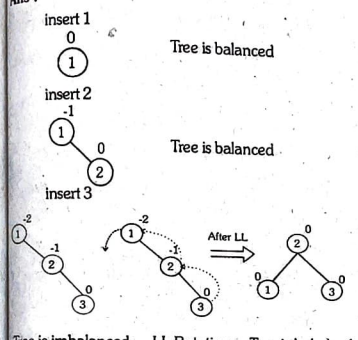
3. Deletion :

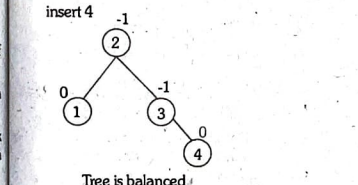
→ The deletion operation in AVL tree is similar to deletion operation in BST. But after every deletion operation we need tocheck with balance factor condition.

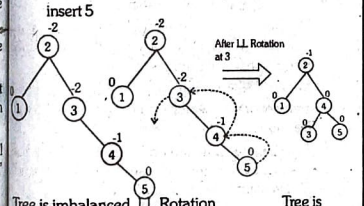
→ If tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree balanced.

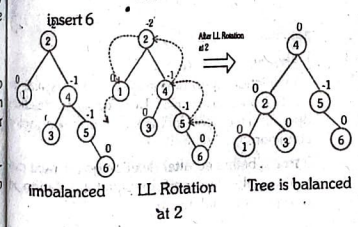
Q9. Construct an AVL Tree by inserting number from 1 to 8.

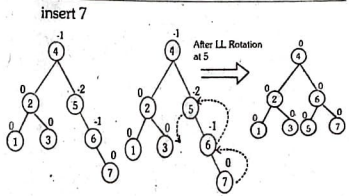
Ans :



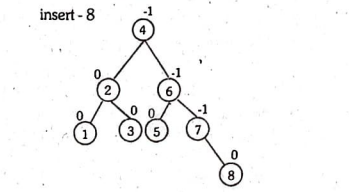








Tree is imbalanced



Tree is balanced

Q10. Construct AVL tree for following list. {M, D, A, H, U, S, J, R, V}

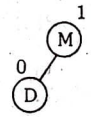
Ans :

Given List = {M, D, A, H, U, S, J, R, V}

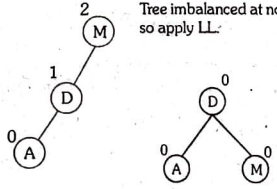
insert (M) : Bf=0

BF = balance factor = Height of left subtree - Heifht of reight Sub tree.

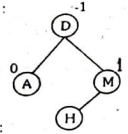
Insert (D) :



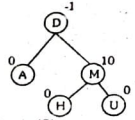
insert (A) :



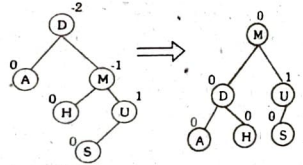
insert (H) :



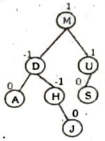
insert (U) :



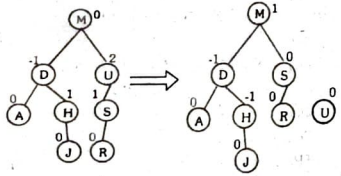
insert (S) :



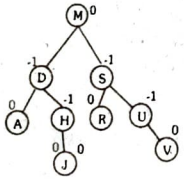
insert (J) :



insert (R) :



insert (V) :



Q11. Write a C++ program to implement AVL Tree.

Ans :

```
# include <iostream>
# include <Cstdio>
# include<sstream>
# include <algorithm>
# define pow2(n) (1<<(n))
using name space std;
Struct avl
{
int d;
struct avl *l;
struct avl *r;
} * r;
class avl_tree
{
public :
int height (avl *);
int difference (avl *);
avl *rr_rotate (avl *);
avl *ll_rotate (avl *);
avl *lr_rotate (avl *);
avl*rl_ rotate (avl*);
avl * balance *(avl *);
avl * insert (avl *, int);
void show (avl *, int);
void inorder (avl *);
void preorder (avl *);
void postorder (avl *);
avl_tree ( ) {
r = NULL;
}
};
int avl_tree :: height (avl *t)
{
int h = 0
if (t! = NULL)
{
int l_height = height (t->l);
int r_height = height (t → l);
int max_height = max(l_height, r_height);
h = max_height +1;
}
return h;
```



```
}
int avl_tree :: difference (avl *t)
{
int l_height = height (t → l);
int r_height = height (t → r);
int b_factor = l_height _r_height;
return b_factor;
}
avl * avl_tree :: rr_rotate (avl *parent)
{
avl *t;
t = parent → r;
parent → r = t → l;
t → l = parent;
Cout <<"right_right Rotation";
return t;
}
avl *avl_tree :: ll_rotat (avl *parent)
{ avl *t;
t = parent → l;
t → r = parent
Cout <<"left-left rotation";
return t;
}
avl * avl_tree :: lr_rotat (avl* parent)
{ avl *t;
t = parent → l;
parent → l = rr_rotat(t);
Cout <<"Left Right Rotation";
return ll_rotat (parent);
}
avl * avl_tree :: rl_rotate (avl* parent) {
avl * t;
t = parent → r;
parent → r = ll_rotat (t);
Cout << "Right_Left Rotation";
return rr_rotat (parent);
}
avl * avl_tree :: balance (avl *t) {
int bal_factor = difference (t);
if (bal_factor >1)
{
if (difference (t → 1)>0)
t = ll_rotat (t);
else
t = lr_rotat (t);
} else if (bal_factor <-1) {
if (difference (t → r)>0)
t = rl_rotat (t);
else
t=rr_rotat (t);
}
return t;
}
avl*avl_tree :: insert (avl *r, int v)
{
if (r==NULL){
r = new avl;
r→ d=v;
r → l = NULL;
r→ r = NULL;
return r;
} else if (v<r → d) {
r→ l = insert (r → l, v);
r = balance (r);
} else if (v) = r→ d){
r→ r = insert (r→ r,v);
r = balance (r);
} return r;
}
void avl_tree : show (avl*p, int l){
int i;
if (p! = NULL)
{
show (p→ r, l+1);
Cout <<"  ";
if P(==r)
Cout <<"Root → ";
for (i-0, i<l && p! =r; i++)
Cout <<"  ";
Cout <<P→ d;
show (P→ l, l+1);
}
}
Void avl_tree :: inorder (avl*t)
{
if (t==NULL)
return;
inorder (t→ 1);
Cout <<td<<"  ";
```

```
}
Void avl_tree::preorder (avl*t)
{ if (t== NULL)
return;
Cout <<t→ d<<"  ";
Preorder (t→ l);
Preorder (t→ r);
}
Void avl_tree :: postorder (avl*t)
{
if (t == NULL)
return;
post order (t→ l);
postorder (t→ r);
Cout <<t→ d<<"  ";
}
int main ( )
{
int C, i,
avl_tree avl;
while (1) {
Cout << "1. Insert Element into tree" <<end l;
Cout <<"2. Show Balanced AVL tree" <<endl;
Cout <<"3. Inorder "" <<end l;
Cout <<"4. Preorder" <<end l;
Cout <<"5. postorder"<< endl;
Cout <<"6. Exit" << end l;
Cout <<"Enter your choice:";
in>>c;
switch (c) {
case :
Cout <<"Enter value to be inserted");
Cin >>i;
r = avl.insert (r,i);
break
case 2 :
if (r==NULL) {
Cout <<"Tree is empty <<end l;
Continue;
}
Cout << Balanced AVL tree:"<<end l;
avl.show (r,1);
Cout <<end l;
break;
case 3:
Cout <<Inorder:"<<end l;
avl.inorder (r);
Cout << end l;
break;
case 4 :
Cout << "Preorder" << endl;
avl .preorder (r);
Cout << end l;
break;
case 5 :
Cout << "Postorder" << end l;
avl.postorder (r);
Cout << end l;
break;
case 6 :
exit (1); break;
default;
Cout <<"Wrong choice" << end l;
```

3.3 RED BLACK TREES

Q12. Explain in detail about Red black tree.

Ans :

Red - black tree :

Red -black tree is a Binary search tree in which every node is colored either red or black.

Properties of Red-black tree :

(i) Red-black tree must be binary search tree.

(ii) The root node must be coloured black.

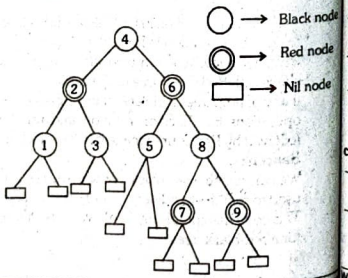
(iii) The children of red coloured node must be coloured black (There should not be two consecutive RED nodes).

(iv) In all the paths of tree, there should be same number of black coloured nodes.

(v) Every new node must be inserted with RED colour

(vi) Every leaf must be colored BLACK.

Example :



→ Every Red Black Tree is a binary search tree but every Binary search tree need not to be Red black tree.

Operation of Red-Black Tree :

→ Red-black tree performs the following operations.

a) Insertion

b) Deletion

c) Search

a) **Insertion :**

→ In a Red black tree, every new node must be inserted with colour RED. The insertion operation in Red-black tree is similar to insertion in Binary Search tree. But it is inserted with colour property.

→ After insertion we need to check all the properties of red-black tree. If all the properties are satsified then we go to next operation otherwise we perform following operation to make it red-black tree.

(i) Recolor

(ii) Rotation

(iii) Rotation followed by Recolor.

→ The insertion operation in Red-Black tree is performed using following steps.

Step 1 : Check whether tree is empty.

Step 2 : If tree is empty then insert the newNode as Root node with colour Black and exit from operation.

Step 3 : If tree is not Empty then insert the new node as leaf node with color Red.

Step 4 : If parent of newNode is Black then exit from operation.

Step 5 : If parent of newNode is Red then check the color of parent node's sibling of newNode.

Step 6 : If it is colored Black or NULL then make suitable rotation and recolor it.

Step 7 : If it is colored Red then perform Recolor Repeat the same until tree becomes Red black tree.

b) **Deletion :**

→ The deletion operation in Red-black tree is similar to deletion operation in BST.

→ But after every deletion operation we need to check with Red Black Tree properties.

If any of properties is violated then make suitable operations like Recolor, rotation and rotation followed by Recolour to make it Red-black tree.

c) **Search :**

→ We can search a red-black tree with the code we used to search an ordinary binary search tree.

→ This code has complexity O(h), which is O(logn) for a red-black tree.

Q13. Create a RED BLACK Tree by inserting following sequence of number 8, 18, 5, 15, 17, 25, 40, 80.

Ans :

Insert 8

Tree is Empty. So insert newNode as Root node with black color.

Insert 18

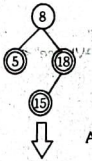
Tree is not Empty. So insert newNode with red color.

Insert 5

Tree is not empty. So insert with red color.

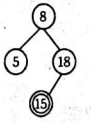
Insert 15

Treee is not empty. Insert node with red color.



(Here there are 2 consecutive red nodes (18, 15). So we use recolor to make it red-black tree)

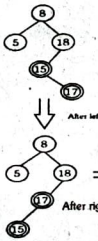
After Recolor



(After Recolor operation, tree is satisfying all red-black tree properties)

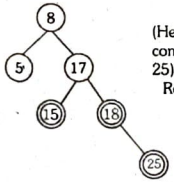
Insert 17

Tree is not Empty. So insert newNode with red color.

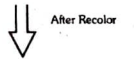


(Here there are 2 Consecutive red nodes (15,17) we need LR rotation and recolor)

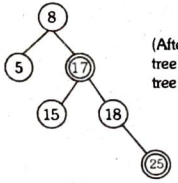
Insert 25 :



(Here there are two consecutive red nodes (18, 25). So we use Recolor and Recheck).

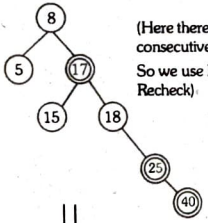


After Recolor



(After Recolor operation, the tree is satisfying all Red black tree properties()

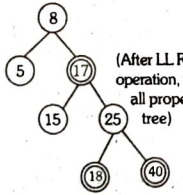
Insert 40



(Here there are two consecutive red noes

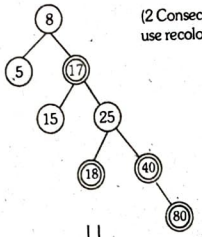
So we use LL Rotation and Recheck)

After LL Rotation & Recolor



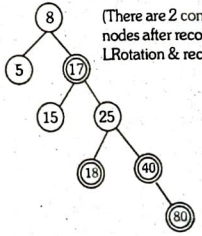
(After LL Rotation & Recolor operation, the tree is satisfying all properties of red-black tree)

Insert 80.



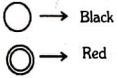
(2 Consecutive red nodes we use recolor and recheck)

After Recolor



(There are 2 consecutive red nodes after recolor. we use LRotation & recolor)

Finally above tree is satsifying the properties of red-black tree.



3.4 2-3 Trees

Q14. Explain in detail about 2-3 Trees.

Ans :

2-3 Trees:

→ 2-3 tree is a tree data structure in which every internal node has either one data element and two children or two data elements and three children.

→ If a node contains one data element left val, it has two sub trees namely left and middle. Where as if a node contains two data elements left val, and right val, it has three subtrees namely left, middle and right.

→ The main advantage with 2-3 trees is that it is balanced in nature as opposed to a binary search tree whose height in worst case can be O(n).

→ Due to this, the worst case time complexity of operations such as search, insertion and deletion is O(log (n)) as height of 2-3 tree is O(log (n)).

a) **Search :**

→ To search a key K in given 2-3 tree T, we have following procedure:

Base cases:

(i) If T is empty, return false.

(ii) If current node contains data value which is equal to K, return True.

(iii) If we reach the leaf node and it doesn't contain required key value k, return false.

Recursive calls :

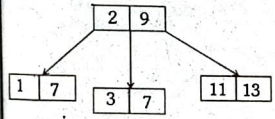
(i) If Kc current Node. Left val, we explore the left subtree of current node.

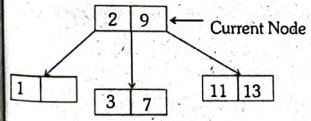
(ii) Else if current node. Left val < KC current Node. right val, we explore the middle subtree of current node.

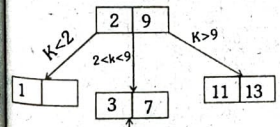
(iii) Else if K> current Node. right val, we explore the right subtree of current node.

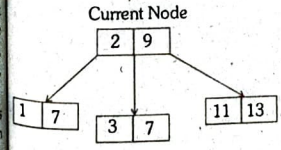
Example :

Search 5 in following 2-3 tree:









5 not found. Return false

b) **Insertion:**

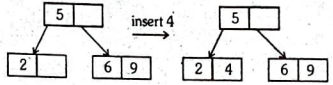
→ There are 3 possible cases in insertion which have been discussed below:

Case 1 :

Insert in node with only one data element

Example :

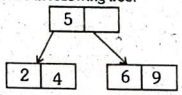
Insert 4 in following tree



Case 2:

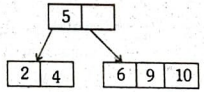
Inser in a node with 2 data elements whose parent contains only one data element.

Insert 10 in following tree.



Initial

Temporary Node with 3 data elements.

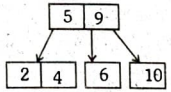


Move middle element to parent and split current Node.

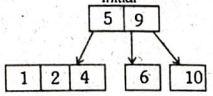
Case 3 :

Insert in a node with two data elements whose parent also contains two data elements.

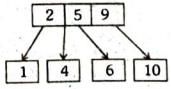
Insert 1 in following Tree



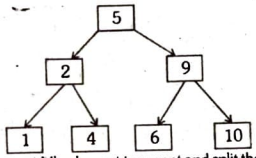
Initial



Temporary node with 3 data elements.



Move middle element to parent and split current Node.



Move middle element to parent and split the current Node.

3.5 B TREES

Q15. Explain B Trees briefly.

Ans :

B Trees:

→ B-Tree is a self balanced search tree in which every node contains multiple keys and has more than two children.

→ Here number of Keys in a node and number of children for a node depends on order of B-tree. Every B-tree has an order.

→ B-Tree of order m has following properties.

(i) All leaf nodes must be at same level.

(ii) All nodes except root must have atleast [m/2]-1 keys and maximum of m-1 keys.

(iii) All non leaf nodes except root must have at least m/2 children.

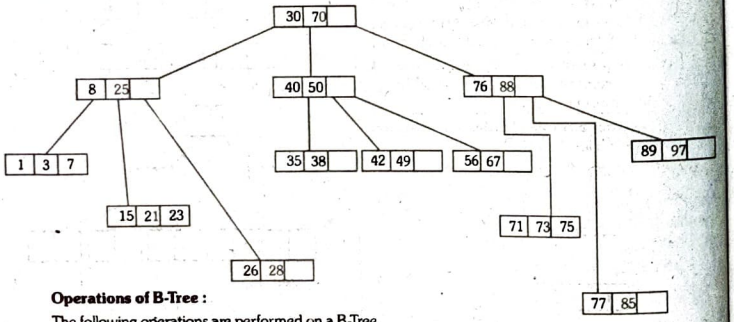
(iv) If root node is non leaf node, then it must have atleast 2 children.

(v) A non leaf node with n-1 keys must have n number of children.

(vi) All key values in a node must be in Ascending order.

Example :

B-Tree of order 4 contains a maximum of 3 key values in a node and maximum of 4 children for a node.



Operations of B-Tree :

The following operations are performed on a B-Tree.

a) Search

b) Insertion

c) Deletion

a) Search:

The search operation performed as following

(i) Read search element from user.

(ii) Compare the search element with first key value of root node in tree.

(iii) If both are matched, then display "Given node is found!!!" and terminate the function.

(iv) If both are not matched, then check whether search element is smaller or larger than that key value.

(v) If search element is smaller, then continue search process in left subtree.

(vi) If search element is larger, then compare search element with next Key value in same node and repeat steps 3, 4, 5 and 6 until we find exact match or until the search element is compared with last key value in left node.

(vii) If last key value in leaft node is also not matched then display "Element is not found" and terminate function.

b) Insertion :

Insertion operation is performed as followed.

(i) Check whether tree is empty.

(ii) If tree is Empty, then create new node with new key value and insert it into tree as a root node.

(iii) If tree is Not empty, then find the suitable leaf node to which new key value is added using Binary search tree logic.

(iv) If that leaf node has empty position, add the new key value to leaf node in ascending order of key value with in node.

(v) If leaft node is already full, split that leaft node by sending middle value to its parent node. Repeat same until the sending value is fixed into node.

(vi) If splitting is performed at root node then middle value becomes new root node for tree and height of tree is increased by one.

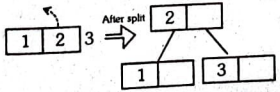
Q16. Construct a B-tree of order 3 by inserting number from 1 to 10.

Ans :

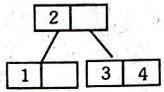
Insert (1)

Insert (2)

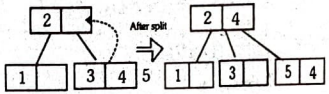
Insert (3)



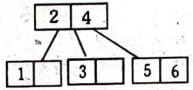
Insert (4)



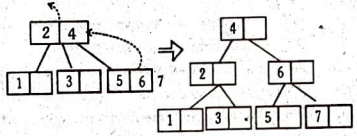
Insert (5)



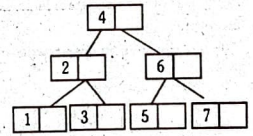
Insert (6)



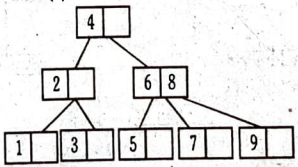
Insert (7)



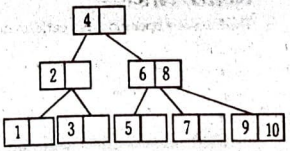
Insert (8)



Insert (9)



Insert (10):



3.6 SPLAY TREES

Q17. Explain in brief about Splay Tree.

Ans :

Splay tree :

Splay tree is another varient of BST in which every operation on element rearranges the tree so that element is placed at root position of tree.

→ All the operations in splay tree are involved with common operation called 'Splaying'.

→ Splaying an element is a process of bringing it to root position by performing suitable operations.

→ In splay tree, to splay any element we use the following rotation operations.

a) Zig Rotation

b) Zag Rotation

c) Zig-Zag Rotation

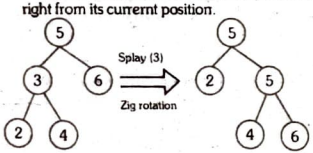
d) Zag-Zag Rotation

e) Zig-Zag Rotation

f) Zag-Zig Rotation

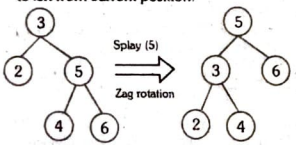
a) Zig Rotation

In Zig rotation, every node moves one position to right from its currernt position.



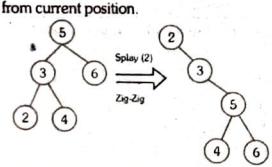
b) Zag Rotation :

In Zag rotation, every node moves one position to left from current position.



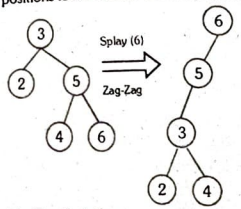
c) Zig-Zag Rotation :

In this, every node moves two positions to right from current position.



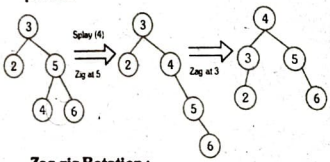
d) Zag-Zag Rotation :

In Zag-Zag Rotation, every node moves two positions to left from its current position.



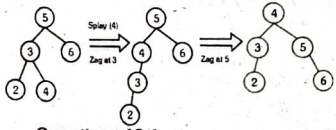
e) Zig-Zag Rotation :

In this, every node moves one position to right followed by one position to left from current position.



f) Zag-zig Rotation :

In this every node moves one position to left followed by one position to right from current position.



Operations of Splay tree :

→ Splay tree performs following operations :

a) Insertion :

The insertion operation in splay tree is performed using following steps:

Step 1: Check whether tree is empty.

Step 2 : If tree is empty then insert the new Node as Root node and exit from operation.

Step 3 : If tree is not empty then insert the newNode as leaf node using BST logic.

Step 4 : After insertion splay the newNode.

b) Deletion :

The deletion operation in splay tree is similar to operation in binary search tree. But before deleting the element, we first need to splay that element and then delete it from root position.

→ Finally join the remaining tree using BST logic.

UNIT 4

TEXT PROCESSING

Short Questions with Answers

Q1. Define pattern matching.

Ans :

Pattern matching is the act of checking a given sequence of tokens for the presence of constituents of some patterns.

Pattern matching is to find a pattern which is relatively small, in a text which is to be very large.

Q2. List out the pattern matching algorithms.

Ans :

The popular pattern matching algorithms are

→ Naive pattern matching
→ Brute force algorithm
→ Boyer - moore algorithm
→ Knuth - morris pratt algorithm

Q3. Define Trie.

Ans :

→ True is a tree like data structure used to store collection of strings.

(or)

→ Trie is an efficient information storage and retrieval data structure.
→ It is also called multiway tree data structure.
→ The term trie came from the word 'retrieval'.

Q4. What are the different types of tries?

Ans :

Types of Tries :

These are 3 types of tries. They are

→ Standard Trie
→ Compressed Trie
→ Suffix Trie

Q5. Define Standard Trie.

Ans :

Standard trie is defined as for a set of string S is an ordered tree such that :

→ Each node but the root is labelled with a character
→ The children of a node are alphabetically ordered.
→ The path from external nodes to root yield strings of S.

Q6. Define KMP algorithm.

Ans :

→ Knuth Morris Pratt algorithm is an algorithm which checks the characters from left to right.
→ This algorithm searches for occurrence of a word 'w' with in a main text 'S' by employing the observation.

Q7. What are the applications of pattern matching?

Ans :

→ Text Editor : In text editor we have no. of lines of text data for finding required string from the editor we use string pattern match.
→ Search engine : The query submitted by the user in search engine uses pattern matching.
→ Biological Search : eg : DNA.

Q8. Define Boyer - moore algorithm.

Ans :

The Boyer -moore algorithm is an sufficient string search algorithm. This algorithm pre process the string being searched for the pattern but not the string being searched in the text.

Q9. What are the advantages of trie?

Ans :

→ The pattern matching can be done efficiently.
→ In tries, the keys are searched using common prefix.
→ There is no collision in trie.

Q10. What are the disadvantages of trie?

Ans :

→ Some times data retrieval of trie is very much slower than hash table.
→ Representation of keys a string is complex.
→ It always takes more space.

→ It is not available in programming tool.

Q11. Explain applications of trie.

Ans :

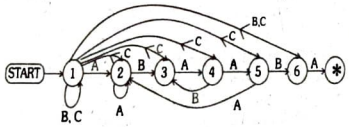
→ Tries has an ability to insert, delete or search for entries. Hence, they are used in building dictionaries.

→ These are also used in spelling check.

→ These are well suited for approximate matching algorithm.

Q12. Explain KMP flowchart for pattern 'ABAABA' where {A,B,C}.

Ans :



Q13. Write the features of Boyer moore algorithm.

Ans :

→ Performs comparision from right to left.

→ Searching phase in O(mn) time Complexity.

→ O(n/m) best performance.

→ Preprocessing phase in $O(m+|\Sigma|)$ time and space complexity.

Essay Questions with Answers

4.1 String Operations:

Q1. Explain about String Operations.

Ans :

String Operations :

→ Character strings can come from a wide variety of sources, including scientfic, linguistic and Internet application. Indeed the following are examples of such strings.

P = "CGTAAACTGCTTTAATCAAACGC"

R = "U.S. Men Win Soccer World Cup!"

→ The first string P, come from DNA applications, the last string S, is the Internet address (URL) for Website that accompanies this book, and middle string R, is fictional news headline.

→ Several of typical string processing operations involve breaking large strings into smaller string. In order to be able to speak about pieces that result from such operations, we use the term substring of a m-character string P to refer to a string of form P[i] P[i+1] P[i+2] ....P[j], for some $0 \le i \le j \le m-1$, that is, the string formed by characters in P from index i to index j, inclusive.

→ Technically this means that a string is actually a substring of itself, so if we want to rule this out as a possibility, we must restrict the definition of proper substrings, which require that either i>0 or j<m-1.

→ To simplify the notation for referring to substrings, let us use P[i...j] to denote the substring of P from index i to index j, inclusive. That is,

P[i....j] = P[i] P[i+1] .... p[j].

→ We use convention that if i>j, then P[i....j] is equal to null string, which has length O. In addition, in order to distinguish some special kinds of substrings let us refer to any substring of form P[0.....i], for $0 \le i \le m-1$, as prefix of P, and any substring of form P[i...m-1], for $0 \le i \le m-1$, as a suffix of P.

→ For example, if we again take P to be string of DNA given above, then "CGTAA" is a prefix of P, "CGC" is a suffix of P, and "TTAATC" is a proper substring of P.

4.2 Brute Force Pattern Matching

Q2. Define pattern matching? Explain briefly about pattern matching.

Ans :

Pattern matching :

→ Pattern matching is an act of checking a given sequence of tokens for presence of constituents of some patterns.

→ Patterns matching is used to find a pattern which is relatively small, in a text which is to be very large.

→ Patterns & text can be one dimensional or two dimensional.

a) One dimensional :

→ Text editors & DNA

→ Text editor have a 26 characters & some special systems.

b) Two dimensional :

→ eg : Computer Vision.

→ Either one dimensional or two dimensional the text is very large & therefore a fast algorithm to find the occurence of pattern in it needed.

→ In the classic string pattern matching problem we are given text string 'T' of length 'n' and pattern string 'p' of length 'm'.

→ To find whether 'p' is a substring of 'T'. The notation of a match is that there is substring of 'T' starting at some index 'i'.

P[0] = T[i], P[1] = T[i+1], P[2] = T[i+2]..... ..... P[m-1] = T[i+n-1].

→ The output of a pattern matching algorithm could either be some indication. That the pattern P does not exist in 'T' or an integer indicating the starting index in 'T' of a substring matching 'P'.

Applications of Pattern matching :

→ Text Editor : In text editor we have number of lines of text data for finding required string from editor we use pattern match.

→ Search engine : The query submitted by user in search engine uses pattern matching.

→ Biological search : eg → DNA

→ Pattern matching is used to find common functionalities of different research things.

Pattern matching algorithms :

The popular pattern matching algorithms are

a) Naive pattern matching algorithm.

b) Brute force algorithm

c) Boyer - moore algorithm

d) Knuth - mooris pratt algorithm.

Q3. What is Brute force algorithm? Explain its features.

Ans :

Brute force algorithm :

→ Brute force is a type of algorithm that tries a large number of patterns to solve a problem. In some cases, they are extremely simple and rely on raw computing power to achieve results.

→ Brute force algorithms also have non security applications. for example, a technique known as data dreadging is a brute force method of looking for statistically patterns in data.

→ The time complexity of brute force is O(n*m).

→ The algorithm consists in checking, at all positions in text between O and n-m, whether an occurrence of pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right.

Features of Brute force Algorithm :

→ No preprocessing phase;

→ Constant extra space needed.

→ Always shifts the window by exactly 1 position to right.

→ 2n expected text characters comparisons.

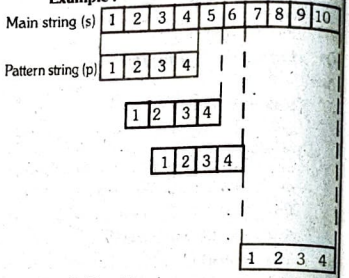
Q4. Explain briefly the algorithm and example of Brute force.

Ans :

Brute force Algorithm :

```
algo pattern -matching_bf (S,P) : return int
{
ls = length (S)
lp = length (P)
max = (ls-lp+1)
for (i=1, i<= max; i++)
{
flag = true
for (j=1; j<lp && flag = = true; j++)
{
if (p[j] ≠ s[j+1-1]
{
    flag = false
}
}
if (flag = = true)
{
    return i
}
}
return 0
}
```

Example :



length (S) = 10 = ls

Length (P) = 4 = lp

Max = 10-4+1 = 7

$(l_s - l_p + 1)$

i	1	2	3	4 → for j=1 to LP
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6
4	4	5	6	7
5	5	6	7	8
6	6	7	8	9
7	7	8	9	10

↳ for i = 1 to max

C- Code for Brute force algorithm :

```
Void BF (char*x, int m, char*y, int n)
{
    int i,j;
for (j=0, j<n-m, j++)
{
for (i=0, i<m && x[i] ==y[i+j]; ++i);
if (i> = m)
OUTPUT (j);
}
}
```

4.3 The Boyer - Moore Algorithm

Q5. Explain about Boyer-moore algorithm briefly.

Ans :

Boyer - Moore Algorithm :

→ The Boyer-moore Algorithm is considered as the most efficient string matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in text editors for the "Search" and substitute commands.

→ Boyer - Moore algorithm is a combination of following two approaches.

a) Bad character heuristic

b) Good suffix Heuristic

Bad character heuristic :

→ The idea of bad character heuristic is simple. The character of text which doesn't match with current character of pattern is called Bad character.

(i) The mismatch become a match.

(ii) Pattern P move past the mismatch character.

(i) Mismatch become a match :

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

G C A A T G C C T A T G T G A C C

T A T G T G

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

G C A A T G C C T A T G T G A C C

T A T G T G

→ In above example, we got a mismatch at point 3. Here our mismatching character is 'A'. No we will search for last occurence of A.

(ii) Pattern move past the mismatch character.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

G C A A T G C C T A T G T G A C C

O T A T G T G

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

G C A A T G C C T A T G T G A C C

T A T G T G

→ Here we have a mismatch at position 7. The mismatching character "C" does not exist in pattern before position 7 so we'll shift pattern past to position 7 and eventually in above example we have got a perfect match of pattern.

Algorithm Boyer-Moore :

```
Boyer-Moore (P, T)
{
    m = p. length
    n = T. length
    i = m-1
    j = m-1
    while (i>n-1)
    {
if (P[j] = = T[i])
{
if (j = = 0)
return i
else
{
        i = j-1
        j = j-1
        }
}
else
{
i = i+m-min (j, 1+last (T[i]))
j = m-1
}
}
return 'no match'
```

Performance :

The computation of the last function takes $O(m+|\Sigma|)$ time and actual search takes O(mn) time.

Features :

→ performs the comparison from right to left.

→ Searching phase in O(mn) time and space complexity.

→ O(n/m) best performance.

→ Boyer - moore algorithm is exactly fast on large alphabet.

Q6. Explain Good Suffix heuristic approach of Boyer -moore algorithm.

Ans :

Good Suffix heuristic :

→ Let 't' be a substring of text T which is matched with substring of pattern P. Now we shift pattern until :

(i) Another occurence of t in P matched with t in T.

(ii) A prefix of P, which matches with suffix of t.

(iii) P moves past t.

(i) Another occurence of t in P matched with t in T.

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

Warning : Xerox/Photocopying of this book is a CRIMINAL Act. Anyone found guilty is LIABLE to face LEGAL proceedings

→ In above example we have got a substring of text T matched with pattern 'P' before mismatch at index 2. Now we will search for occurence of t ("AB") in P.

(ii) A prefix of P which matches with suffix of t in T :

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P	A	B	B	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P			A	B	B	A	B				

→ In above example, we have got t("BAB") matched with P at index 2-4 before mismatch. But because there exist no occurence of t in p we will search for some prefix of P which match with some suffix of t.

(iii) P moves past t:

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	C	A	B	A	B	A	C	B	A
P	C	B	A	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P						C	B	A	A	B	

→ In above example, there exit no occurence of t ("AB") in P and also there is no prefix in P which matches with suffix of t. So in that case we can never find any perfect match before index 4, so we will shift the 'P' past the t. i.e., at index 5.

4.4 The Knuth - Mooris -Pratt Algorithm

Q7. Explain Knuth Mooris Pratt Algorithm.

Ans :

Knuth-Morri's Pratt Algorithm :

KMP algorithm searches for occurence of a word with a main text's by employing the observation. That when a mismatch occurs the word itself embodies sufficient information to determine where the next match could begin, that by passing reexamination of previously matched characters.

Working process :

→ It is a tight analysis of naive algorithm KMP algorithm keeps the information that naive approach wasted gathering during the scan to text.

Eg : W = ABCD ABD

S = ABC ABCDAB ABCDABCDABDE

1st : m → which denotes the position in text which is the beginning of a perspective match fow W.

2nd → i → The index in W denoting the character currently under consideration.

In each step we compare S{m+i] with w[i] advance if they are equal.

```
M = 0 1 2 3 4 5 6 7 8 9 10 11 12
S = A B C   A B C D A B
M: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 12
S: A B C A B C D A B A B C D A B C D A B D E
W: A B C D A B D
i: 0 1 2 3 4 5 6
S[3] = Space } mismatch
W[3] = D
```

We start at S[0] but it fail. Now we start from S[1]. But we note that number 'A' occurs between positions 0 & 3 except at 0.

∴ We move on to character M = 4; i = 0.

```
M: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 12
S: A B C A B C D A B A B C D A B C D A B D E
W: A B C D A B D
i: 0 1 2 3 4 5 6
S[10] = Space } mismatch
W[6] = D
M: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 12
S: A B C A B C D A B A B C D A B C D A B D E
W: A B C D A B D
i: 0 1 2 3 4 5 6
S[10] = Space } mismatch
W[2] = C
M: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 12
S: A B C A B C D A B A B C D A B C D A B D E
W: A B C D A B D
i: 0 1 2 3 4 5 6
S[17] = Space } mismatch
W[6] = D
M: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 12
S: A B C A B C D A B A B C D A B C D A B D E
W: A B C D A B D
i: 0 1 2 3 4 5 6
```

This time we are able to complete the match whose first character is S[15].

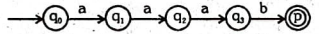
Algorithm :

```
Alg KMP_Search (S,P) return int
{
K = 1, S = q0, n = length (S)
while (K<n && Sk ≠ P)
{
read tk
SK+1 = F(Sk, tk)
K = K+1
}
if (K>n)
{
    index = 0
}
else
{
else = K - length (P)
}
return index
}
```

Example :

Main string (S) : a a a a a a a a a a a b = $a^{10}b$

Pattern (P) = a a a b



Pattern matching graph.

	a	b	x ⇒ Any character other than a or b.
q0	q1	q0	q0
q1	q2	q0	q0
q2	q3	q0	q0
q3	q3	p	q0

Pattern matching graph.

Q8. Compute table representing KMP failure function for pattern string 'cgtacgttcgtac".

Ans :

p = catacgttcgtac

length of pattern m = 13.

y	0	1	2	3	4	5	6	7	8	9	10	11	12
P(y)	c	g	t	a	g	t	t	c	g	t	t	a	c

x = 1

y = 0

f'0) = 0

Iteration 1 :

1 < 13

∵ P[0] = C, P[1] = g

∵ P[0] ≠ P[1]

∵ j = 0

f(1) = 0

x = 1+1 = 2

Iteration 2 :

x = 2

y = 0

2<13

∵ P[0] = C, p[2] = t

∵ p[0] ≠ p[2]

∵ j =0

f(2) = 0

x = 2+1 = 3

Iteration 3 :

x = 3

y = 0

3<13, which is true

∵ p[0] = c, p[3] = a

∵ p[0] ≠ p[3]

∵ y = 0

f(3) = 0

x = 3+1 = 4

Iteration 4 :

x = 4

y = 0

4<13

p[0] = C, p[4] = C

p[0] = p[4]

f(4) = 0+1 = 1

x = 4+1 = 5

y = 0+1 = 1

Iteration 5 :

x = 5

y = 1

5<13

∵ p[1] =g, p[5] = g

p[1] = p[5]

f(5) = 1+1 = 2

x = 5+1 = 6

y = 1+1 = 2

Iteration 6 :

x = 6

y = 2

Warning : Xerox/Photocopying of this book is a CRIMINAL Act. Anyone found guilty is LIABLE to face LEGAL proceedings

6 < 13
p[2] = t; t[6] = t
p[2] = p[6]
f(6) = 2+1 = 3
x = 6+1 = 7, y = 2+1 = 3
Iteration 7 :
x =7, y = 3
7< 13
p[3] = a, p[7] = t
p[3] ≠ p[7]
y>3
y = f(y-1) = f(3-1) = f(2) =0
Iteration 8 :
x = 7, y = 0
7<13
p[0] = c, p[7] = t
p[0] ≠ p[7]
y = 0
f(7) = 0
x = 7+1 = 8
Iteration 9 :
x = 8
y = 0
8<13
p[0] = c, p[8] = c ⇒ p[0] = p[8]
f(8) = 0+1 =1 x = 9, y = 1
Iteration 10 :
x = 9, y = 1
p[1] =g, p[2] =g ⇒ p[1] = p[9]
f(9) = 2 x = 9+1=10, y = 1+1=2
Iteration 11 :
x =10, y = 2
10<13
∵ p[2] = t, p[10] = t
p[2] = p[10]
f(10) = 3 x = 11, y=3
Iteration 12 :
x = 11, y = 3 ⇒ 11<13
p[3] = a, p[11] =a ⇒ p[3] = p[11]
f(11) = 4 x = 12, y = 4
Iteration 13 :
x = 12, y = 4 ⇒ 12<13
p[4] = c, p[12] =c ⇒ p[4] = p[12]
f(12) = 4+1 = 5 x = 13, y = 5
Iteration 14 :
x =13
y = 5
13<13, which is not true
Hence process terminates.

y	p[y]	f(x)
0	c	0
1	g	0
2	t	0
3	a	0
4	c	1
5	g	2
6	t	3
7	t	0
8	c	1
9	g	2
10	t	3
11	a	4
12	c	5

4.5 Standard Tries

Q9. Define tries and explain tries briefly.

Ans :

Trie :

→ A trie or prefix tree is an ordered tree data structure that is used to store an associate array, where the keys are strings.

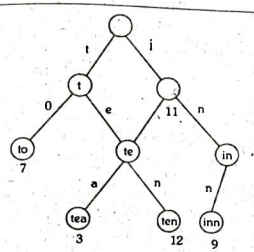
→ The term trie comes from word 'retrieval" actually a trie is a data structure that can be used to a fast search in a large text.

Structure of trie :

→ A trie is a K-array position tree.

→ It is constructed from input strings i.e., the input is a set of n strings called S_1, S_2, ...S_n where, each Si consists of symbols from a finite alphabet and has a unique terminal symbol.

(i) {0,1} for binary files.
(ii) { all 256 ASCII charaters}
(iii) {a, b, c, d,....x, y, z}.



→ In above example, keys are listed in the nodes and values below them. Each complete English word has an integer value associated with it. A trie can be seen as a Deterministic finite Automation (DFA).

Types of Tries :

Threre are three types of tries, they are

(i) Standard trie
(ii) Compressed trie
(iii) Suffix trie

Advantages of Trie :

→ Tries is a tree that stores strings. Maximum number of children of a node is equal to size of alphabet.

→ Trie support search, insert and delete operations in O(L) time, where L is length of Key.

Disadvantages of tries :

→ The main disadvantage of trie is that they need lot of memory for storing strings.

→ For each node we have too many node pointers.

10. Explain standard trie breifly :

ns :

Standard Trie :

The standard tries for a set of strings 'S' is an ordered tree such that :

Each node but root is labeled with character.

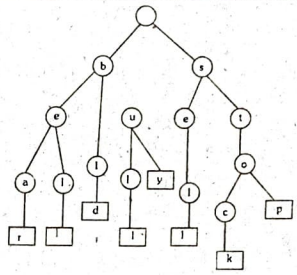
The children of node are alphabetically ordered.

The paths from external nodes to root yield the strings of S.

If 'T' is a trie, then every node is labeled with an element of Σ ;

Example :

Consider strings S = {bear, bell, bid, bull, buy, sell, stock, stop}



Properties of Standard trie :

(i) Every internal node has ≤ |Σ| children.
(ii) The trie T on set S with 'S' strings (keys) has exactly 'S' external nodes.
(iii) The height of trie T on set S = length of longest string ∈ S.
(iv) The number of nodes in tire 'T' on set S=O(n). Where n= # characters in strings ∈ S.

4.6 Compressed Tries

Q11. Explain compressed tries briefly.

Ans :

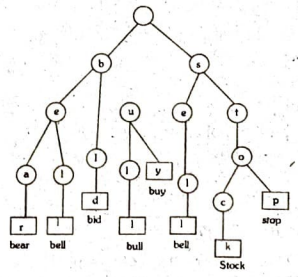
Compressed Tries ;

→ Tries with nodes of degree at least 2.

→ Obtained from standard tries by compressing chains of redundant nodes.

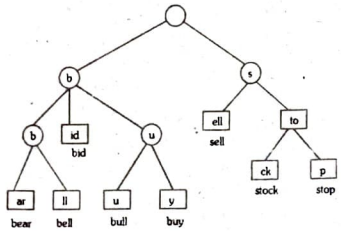
Example :

→ Before compression :



rning : Xerox/Photocopying of this book is a CRIMINAL Act. Anyone found guilty is LIABLE to face LEGAL proceedings

→ After Compression



Properties of Compressed trie :

(i) Each Internal node has ≥ 2 children and ≤ |Σ| children.

(ii) A compressed trie T strong 'S' string (Keys) has : 'S' external nodes.

(iii) A compressed trie T storing 'S' strings (Keys) has : O(S) total number of nodes.

4.7 Suffix Tries

Q12. Explain briefly Suffix Tries?

Ans :

Suffix Tries :

→ Suffix trie is a compressed trie for all suffixes of text.

→ The suffix trie for a text 'x' of size n from an alphabet of size d.

→ Suffix tries stores all the (n-1)/2 suffixes of X in O(n) space.

→ Suffix tries can be constructed in O(dn) time.

Applications :

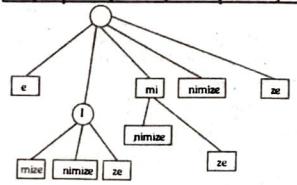
Applications of suffix tries are

a) Word matching

b) Prefix matching

Example :

0	1	2	3	4	5	6
M	I	N	I	M	I	Z



4.8 The Huffman Coding Algorithm

Q13. Explain in detail about Huffman Coding Algorithm.

Ans :

Huffman Coding Algorithm:

→ Huffman Coding is a lossless data compression algorithm. In this algorithm, a variable length code is assigned to input different characters.

→ The code length is related to how frequently characters are used. Most frequent characters have smallest codes and longer codes for least frequent characters.

→ There are mainly two parts, First one to create a Huffman tree, and another one to traverse the tree to find codes.

→ For an example, consider some strings "YYYZXXYYX", the frequency of character Y is larger than X and character Z has least frequency. So the length of code of Y is smaller than X and Code for X will be smaller than Z.

→ Complexity for assigning the code for each character according to their frequency is O(n log n)

Input and Output:

Input :

A string with different characters say "ACCEBFFFFAAXXBLKE"

Output :

Data : K, frequency :1, Code = 0000

Data : L, frequency : 1, Code = 0001

Data : E, frequency :2, Code = 001

Data : F, frequency : 4, Code : 01

Data : B, frequency : 2, Code : 100

Data : C, frequency : 2, Code : 101

Data : X, frequency : 2, Code : 110

Data : A, frequency : 3, Code : 111

Algorithm :

Huffman Coding (String)

Input : A string with different characters.

Output : The Codes for each individual characters.

Beign,

define a node with character, frequency, left and right child of node for huffman tree.

Create a list 'freq' to store frequency of each character, initially, all are 0 for each character C is string do increate the frequency for character ch in freq list.

done

for all type of character ch do if the frequency of ch is non zero then add ch and its frequency as a node of priority queue Q.

done

while Q is not empty do

remove item from Q and assign it to left child of node.

remove item from Q and assign to right child of node

traverse the node to find assigned code

done

End.

TraverseNode(n:node, code):

Input : The node n of the Huffman tree, and code assigned from previous call

Output: Code assigned with each character.

if a left child of node n ≠ ϕ then

traverse Node (Left child (n), code + '0')

traverse Node (right child (n), Code +'1')

else

diplay the character and data of current node.

Q14. Write a program to implement Huffman Coding Algorithm using C++.

Ans :

```
# include <iostream>
Using name space std;
Struct node {
    int freq;
    char data;
Const node *child 0, *child 1;
node (char d, int f = -1)
data = d;
freq = f;
Child 0 = NULL;
Child 1 = NULL;
}
node (const node* C0, constant node *c1)
{
data = 0;
freq = C0 → freq+c1 → freq;
Child 0 = C0;
child 1 = c1;
}
bool operator <(const node & a) const {
return freq> a.freq;
}
Void traverse (String code = "  ") const {
if (child 0! = NULL) {
Child 0 → traverse (code + '0');
Child 1 → traverse (code + '1');
}
else {
Cout <<"Data:"<<data<<", frequency:"<qu;
int frequency [256];
for (int i =0; i<256; i++)
frequency [i] = 0;
for (int i =0; i1) {
node * (0 = newnode (qu.top( ));
qu.pop ():
node * (1=newnode (qu.top ( ));
qu.pop ( );
qu. push (node (co, c1));
}
Cout << "The Huffman Code"" <<
```

4.9 Longest Common subsequence Problem (LCS)

Q15. What is LCS? Explain.

Ans :

Lognest Common Subsequence :

→ The longest common subsequence is a type of subsequence which is present in both of given sequences or arrays.

→ We can see that there are many subproblems which are computed again and again to solve this problem. By using the overlapping substructure property of Dynamic programming, we can over come the computational efforts.

→ Once the result of subproblems is calculated and stroed in table, we can simply calculate the next results by use of the previous results.

Input and Output :

Input :

Two strings with different letters or symbols.

String 1 : AGGTAB

String 2 : GXT XAYB

Output :

The length of LCS, Here it is 4. AGGTAB and GXTXAYB. The underlined letters are present in both strings and in corrent sequence.

Algorithm :

longest ComSubSeq (str1, str2)

Input : Two strings to find the length of LCS

Output : The length of LCS

Begin

m : length of Str 1
n : = Length of str 2
define long SubSeq matrix of Order (m+a) x (n+1)
for i : = to m, do
for j : = 0 to n, do
if i = 0 or j = 0, then
longSubSeq [i,j] : = 0
else if str1 [i-1] = str1[j-1], then
longSubSeq [i,j] : = longSubSeq [i-1, j-1]H
else
longSubSeq[i,j] : = maximum of longSubSeq[i-1,j]
and long SubSeq[i, j-1]
done
done
longSubSeq [m,n]
End.

Source code :

```
# include <iostream>
using name space std;
int max(int a, int b)
{
return (a>b)? a:b;
}
int longest ComSs (String str1, string str2)
{
int m = str1.size ( );
int n = str2. size ( );
int longSubSeq [m+1][n+1];
for (int i=0; i<=m, i++)
{
for (int j=0; j<=n; j++)
{
for (i==0 | |j==0)
longSubSeq [i] [j] = 0;
else if (str1 [i-1] ==str = [j-1]
longSebSeq [i][j] = long Subseq[i-1][j-1]+1;
else
LongSubseq[i][j] = max(longSubseq [i-1][j],
                       LongSubSeq [i] [j-1]);
return longSubseq [m] [n];
}
int main ( )
{
string str 1 = "AGGTAB"
string str2 = "GXTXAYB";
(out<<"Length of Longest Common Subsequence is :"<< Longest ComSs (Str1, Str2);
}
```

4.10 Applying Dynamic Programming to LCS Problem

Q16. Explain the Dynamic programming implementation to LCS problem using C++.

Ans :

Optimal Substructure:

Let the input sequence be x{0...m-1] and y[0..n-1] of lengths m and n respectively. And Let L (X[0...m-1], Y[0...n-1]) be the length of LCS of two sequences X and Y.

Following is the recursive definition of

L (X[0...m-1], Y[0..n01])

If last characters of both sequences match

(or X[m-1] = = Y[n-1] then

L(x[0...m-1], Y[0...n-1] = 1+<(X(0....m-2], Y[0....n-2])

If last character of both sequences do not match

(or X[m-1]! = Y[n-1]) then L[X[0...m-1], Y[0....n-1] = MAX(L(X[0...m-2], Y[0....n-1]),

L (X[0...m-1], Y[0...n-2]))

Examples :

1) Consider the input strings "AGGTAB" and "GXTXAYB". Last characters match for strings. So length of LCS can be written as:

L("AGGTAB", "GXTXAYB") = 1+l("AGGTA", "GXTXAY")

	A	G	G	T	A	B
G	-	-	4	-	-	-
X	-	-	-	-	-	-
T	-	-	-	3	-	-
X	-	-	-	-	-	-
A	-	-	-	-	2	-
Y	-	-	-	-	-	-
B	-	-	-	-	-	-

2) Consider the input strings "ABCDGH" and "AERFHR". Last characters do not mach for strings. So length of LCS can be written as:

L ("ABCDGH", "AEDFHR") = MAX(L("ABCDG", "AEDFHR")

L(ABCDEGH", "AEDFH"))

So LCS problem has optimal substrcuture property as main problem can be solved using solutions to subproblems.

Source Code :

```
/* Dynamic programming C++ implementation of LCS problem */
# include <bits.h>
int max (int a, int b);
int lcs (char *X, char *Y, int m, int n)
{
int L[m+1] [n+1];
int i, j
for (i=0, i<=m; i++)
{
for (j=0, j<=n; j++)
{
if (i = 0 | |j==0)
c[i] [j] =0;
else if (X[i-j] == = y [j-1])
L[i] [j] = L[i-1] [j-1] +1;
else
L[i] [j] = max (L[i-1][j], L[i] [j-1];
}
{
return L[m] [n];
}
/* uitlity function to get max of 2 integers */ int max (int a, int b)
{
return (a>b)? a:b;
}
int main ( )
}
char X[ ] = "AGGTAB";
char Y [ ] = "GXTAYB"
int m = Strlen (X);
int n = Strlen (Y);
print f ("Length of LCS is % d", LCS (X,Y,m,n));
return 0;
}
```

UNIT 5

COMPUTATIONAL GEOMETRY

Short Questions with Answers

Q1. What is Computational Geometry?

Ans :

Computational Geometry is a mathematical field that involves the design and analysis and implmentation of efficient algorithms for solving geometric input and output problems. It is sometimes used to refer to pattern recognition and describe the solid modeling algorithms used for manipulating curves and surfaces.

Q2. What is Two-dimensional Ragne-Search queries?

Ans :

A two-dimensional dictionary is an ADT for storing key - element items such that Key is a pair (x,y) of numbers called coordinators element.

Q3. What is priority search Tree?

Ans :

The priority search tree is used to store a set of 2-D points ordered by priority and by a Key value. In addition each node also contains a key value used to divide the remaining points into a left and right subtree.

Q4. Define K-Dtree.

Ans :

A K-D tree is a binary search tree where data in each node is K-Dimensional point in space. In short, it is a space partitioning data structure for organizaing points in a K-dimensional space.

Q5. Define Quad tree.

Ans :

Quadtree is a tree data structure in which each internal node has exactly four children. These are two-dimensional analog of octrees and are most often used to partition a two-dimensional space by recursively subdividing into four quadrants or regions.

Q6. What are the steps used to construct a quadtree from a 2-D area.

Ans :

One can construct a quadtree from a 2-D area using following steps:

→ Divide the current two dimensional space into four boxes.

→ If a box contains one or more points in it, create a child object, storing in it the two dimensional space of box.

→ If a box does not contain any points, do not create a child for it.

→ Recurse for each of children.

Q7. List the uses of Quad tree.

Ans :

→ Quadtrees are used in image Compression, where each node contains the average colour of each of its children. The deeper you traverse in tree, the more detail of image.

→ Quad trees are used in searching for nodes in a 2D area. For instance, if you wanted to find the closest point to given coordinates, you can do it using quadtrees.

Essay Questions with Answers

5.1 One Dimensional Range Searching

Q1. What is One Dimensional Range Searching? Explain.

Ans :

One Dimensional Range Searching:-

→ One Dimensional Range Searching is a searching where, given an ordered dictionary D, we want to perform the following query operation.

find All In Range (K_1, K_2) : Return all elements in dictionary D with Key K such that $K_1 \le K \le K_2$

→ We use a recursive method 1D Tree Range Search that takes as arguments the range parameters K_1 and K_2 and a node V in T. If node V is external, we are done. If node V is in ternal we have three cases, depending on value of Key (V), the Key of item stored at node V:

i) Key (V) < K_1

ii) $K_1 \le$ Key(V) $\le K_2$

iii) Key (V) > K_2

Algorithm = 1D Tree Range Search (K_1, K_2, V) :

Input : Search Keys K_1 and K_2 and a node V of a binary search tree T.

Output : The elements stored in subtree of T rooted V, whose key is greater than or equal to K_1 and less than or equal to K_2.

if T is External (V) then

return ϕ

if $K_1 \le$ Key(V) $\le K_2$ then

1 → 1D Tree Range Search (K_1,K_2 , T.left child (V))

R → 1D Tree Range Search (K_1,K_2 , T.right child (V))

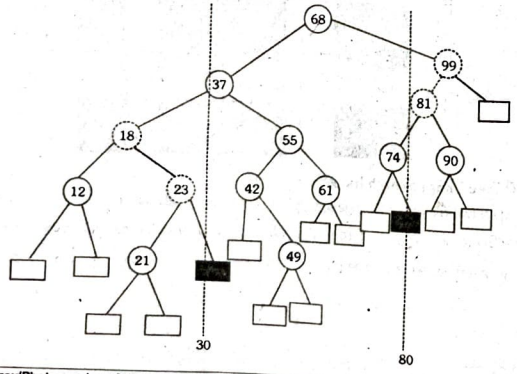
return <U {element(V} UR

else if Key(V) <K, then

return 1D Tree Range Search (K_1,K_2 ,T.right child (V))

else if K_2 < Key (V) then

return 1D Tree Range Search (K_1,K_2 ,T.left child (V))



→ Consier the execution of algorithm 1D Tree Range Search (K_1,K_2 , r), where r is root of T. We traverse a path of boundary nodes, calling the algorithm recursively either on left on right child until we reach either an external node or an integral node w with Key in range [K_1,K_2].

→ Since we spend a constant amount of work per node visited by algorithm, the running time of algorithm is proportional to number of nodes visited.

i) We visit no outside nodes.

ii) We visit at most 2h+1 boundary nodes, where h is height of T, since boundary nodes are an search paths p_1 and p_2 and they share atleast one node.

iii) $\sum_{i=1}^{j}(2S_i + 1) = 2S + j \le 2S + 2h$

∴ at most 2S+4h+1 moves of T are visited and operation find All In Range runs in O(h+s) time.

5.2. 2-D Range Searching

Q2. Explain 2-D Range Searching briely.

Ans :

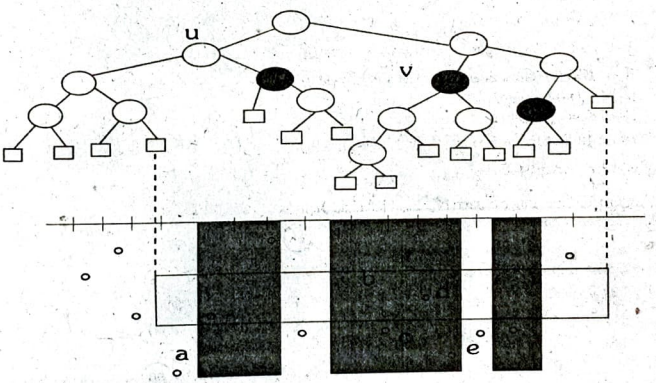
2-D Range Searching:

→ The 2-D Range Searching is a data structure realizing the two dimensional dictionary ADT. It consists of a primary structure, which is balanced binary search tree T, together with a number of auxilary structures.

→ An internal node V of T stores the following data:

i) An item, whose coordinates are denoted by x(v) and y(v) and whose element is denoted by element (v).

ii) A one dimensional range tree T(v) that stores the same set of items as subtree rooted at V and T, but using y-ordinates as Keys.



Algorithm : 2D Tree Range Search I(x, x_2, y, y_2, v, t):

Recursive method for two-dimensional range search in a two-dimensional range tree. The initial method call is 2D Tree Range Search (x_1,x_2,y_1,y_2,T,root(),"middle"). The algorithm is called recursively on all the boundary nodes with respect to the x-range [x_1,x_2]. Parameter t indicates whether v is a left, middle, or right boundary node.

5.3 Priority Search Trees Constructing a Priority Search Tree

Q3. What is priority search tree? Explain.

Ans :

Priority Search tree:

→ A priority search tree for set s is a binary tree storing the items of S that behaves like a binary search tree w.r.t. x - coordinates, and like a heap with respect to y-coordinates.

→ For simplicity, let us assume that all items of S have distinct x and y coordinates. If set S is empty consists of a single external node.

→ Otherwise, Let $\bar{p}$ be top most item of S, that is, the item with maximum y - coordinate. We denote with $\hat{x}$ the median x - coordinate of items in S- $\{\bar{P}\}$,and with S_L and S_R the subsets of $S-\{\bar{P}\}$ with items having x-coordinate less than or equal to $\hat{x}$ and greater than $\hat{x}$, respectively.

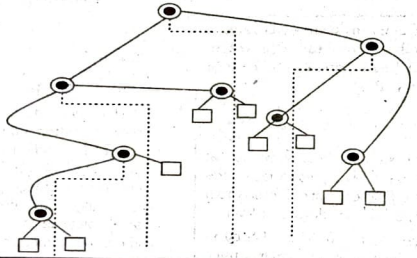
→ We recursively define the priority search tree T for S as follows:

i) The root T stores item $\bar{p}$ and meain x - coordination $\hat{x}$.

ii) Left structure of T is a priority search tree for S_L.

iii) The right subtree of T is a priority search tree for S_R.

→ For each internal node V of T, we denote with $\bar{P}(V), \bar{x}(V)$ and $\bar{y}(V)$ the top most item stored at V and its coordinates. Also we denote with $\hat{x}(V)$ the medain x-coordinate stored at V. An example of priority search tree is shown in following figure.



Q4. Explain Constructing a Priority Search Tree.

Ans :

Constructing a Priority Search Tree:

→ The y-coordinates of items stored at nodes of a priority search tree T satsify the heap order property. That is, if U is parent of V, then $\bar{y}(u) > \bar{y}(v)$. Also the median x- coordinates stored at nodes of T define a binary search tree.

→ These two facts motivate the term "priority search tree", Let us explain how to construct a priority search tree from a set S of n two dimensional and items.

→ We begin by sorting S by increasing x-coordinate, and then call the recursive method build PST (S) shown in following algorithm.

Algorithm :

Input : A sequence S of n two-dimensional items, sorted by x-coordinate

Output : A priority search tree T for S.

Create an elementary binary tree T consisting of single external node v

If !S. is Empty ( ) then

Traverse sequences S to find the item $\bar{p}$ of S with highest y-coordinate

Remove $\bar{p}$ from S

$p(v) \leftarrow \bar{p}$

$\hat{p} \leftarrow$ S.elemAtRank [S.size ( )/2])

$\hat{x}(v) \leftarrow x(\hat{p})$

Split S into two subsequences, S_L and S_R, where S_L contains the items upto

$\hat{p}$ (included), and S_R contains the remaining items.

$T_L \leftarrow$ buildPST(S_L)

$T_R \leftarrow$ buildPST(S_R)

T. expandExternal(v)

Replace the left child of v with T_L

Replace the right child of v with T_R.

return T.

Q5. Given a set S of n two dimensional items, a priority search tree for 5 uses O(n) space, has height O (log n), and can be built in O(n log n) time. Prove.

Ans :

Proof :

→ The O(n) space requirement follows from fact that every internal node of priority search tree T stores a distinct item of S. The height of T follows from the halving of number of nodes at each level.

→ The preliminary sorting of items of S by x-coordinate can be done in O(n logn) time using an asymptotically optimal sorting algorithm, such as heapsort or merge sort.

→ The running time T(n) of method build PST is characterized by the recurrence,

$T(n) = 2T(n/2)+bn$, for some constant

b>0. Therefore, by Master theorem,

T(n) is O(n lon).

5.4 Searching a Priority Tree

Q6. Explain Searching a priority search tree.

Ans :

Searching a priority Search Tree :

Algorithm :

Input : Three sided range, defined by x_1, x_2 and y_1 and a node v of a priority search tree T

Output : The items stored in the subtree rooted at v with coordinate (x,y) such that $x_1 \le x \le x_2$ and $y_1 \le y$

if $\bar{y}(v) < y_1$ then

return ϕ

if $x_1 \le \bar{x}(v) \le x_2$ then

$M \leftarrow \{\bar{p}(v)\}$ {We should output $\bar{p}(v)$ }

else

$M \leftarrow \phi$

if $x_1 \le \hat{x}(v)$ then

$L \leftarrow$ PST Search $(x_1, x_2, y_1,$ T.leftChild(v))

else

$L \leftarrow \phi$

if $\hat{x}(v) \le x_2$ then

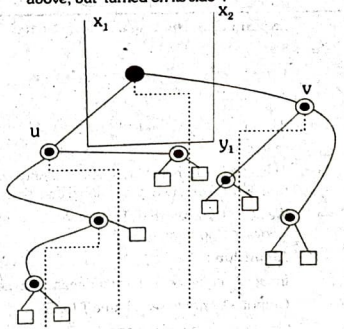
$R \leftarrow$ PSTSearch (x_1, x_2, y_1 , T.right Child (v))

else

$R \leftarrow \phi$

return $L \cup M \cup R$

→ We have defined three sided ranges to have left, right and bottom side, and to be unbounded at top. This restriction was made wtihout loss of generality, however, for we could have defined our three sided range queries using any three sides of rectangle. The priority search tree from such an alternate definition is similar to one defined above, but 'turned on its side".



(i) Nove V is a boundary node if it is an the search path for x_1 and x_2 when viewing T as a binary search tree on medan x-coordinate stored at its nodes.

(ii) Node V is an inside node if it is internal, it is not a boundary node, and $\overline{y}(v) \geq y_1$.

iii) Node V is a terminal node if it is not a boundary node and if internal, $\overline{y}(v) < y_1$.

5.5 Priority Range Trees

Q7. Explain Priority Range Trees briefly.

Ans :

Priority Range Trees

→ Let T is balanced binary search tree storing n items with 2-D Keys, ordered according to their x-coordinates.

→ To convert T into a priority range tree, we visit each internal node V of T other than root and construct, as an auxiliary structure, a priority search tree T(v) for items stored in subtree of T rooted at V.

→ If V is left child, T(v) answers range queries for three side ranges unbounded on right. If V is a right child, T (v) answers range queries for three sided unbounded on left.

→ The method for performing a 2-D range query in a priority range tree is given in following algorithm.

Algorithm :

Input : Search keys x_1, x_2, y_1 and y_2 ; node v in the primary structure T of a priority range tree.

Output : The items in the subtree rooted at v whose coordinates are in the x-range $[x_1, x_2]$ and in the y-range $[y_1, y_2]$

If T is External (v) then

return ϕ

if $x_1 \leq x(v) \leq x_2$ then

if $y_1 \leq y(v) \leq y_2$ then

$M \leftarrow$ {element(v)}

else

$M \leftarrow \phi$

$L \leftarrow$ PSTSearch(x_1, y_1, y_2 ,T(Left Child (v), root ( )),

$R \leftarrow$ PSTSearch(x_2, y_1, y_2 , T(Right Child(v)).root ( ))

return $L \cup M \cup R$

else if $x(v) < x_1$ then

return PSTRange Search

(x_1, x_2, y_1, y_2 ,T.rightChild (v))

else

$\{x_2 < x(v)\}$

return PSTRange Search (x_1, x_2, y_1, y_2 , T.Left Child (v))

5.6 Quad Trees

Q8. Discuss briefly Quad trees.

Ans :

Quad Trees:

Quad tree is a tree data structure in which each internal node has exactly four children. These are two dimensional analog of octrees and are most often used to partition a 2-D space byb recursively subdividing into four quadrants or regions.

Steps used to Construct Quadtree from a 2-D area:

One can construct a quadtree from a 2-D area using following steps:

→ Divide the current two dimensional space into four boxes.

→ If a box contains one or more points in it, create a child object, storing in it the two dimensional space of box.

→ If a box does not contain any points, do not create a child for it.

→ Recurse for each of children.

Explanation :

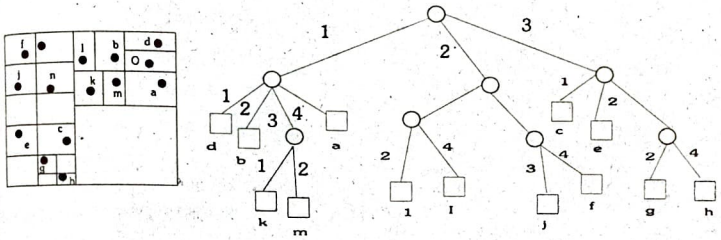
Suppose we are given a set S of n points in plane. In addition, let R denote a square region that contains all the points of S. The quad tree datastructure is a partition tree Tsuch that root r of T associated with regions R. To get to next level in T, we subdivide R into four equal sized squares, R_1, R_2, R_3 and R_4, and we associate each square R_i with potential child of root r.

→ Specifically, we create a child V_i of r, if the square R_i contains a point in S. If a square R_i contains no point of S, then we create no child of r for it. This process of refining R into the squares R_1, R_2, R_3 and R_4 is called split.

→ The quad tree T is defined by recursively performing a split at each child V of r if necessary. That is,

each child V of r has a square region R_i associated with it, ad if region R_i for V contains more than one point of S, then we perform a split at V, subdividing R_i into four equal sized squares and repeating the above subdivision process at V.

→ We the store each points P in S at the external node of T that corresponds to smallest square in sub division process that contains P. We store at each internal node V a concise representation of split that we performed for V.



→ For example, our point set S could contain two points that are very close to one another, and it may take a long sequence of splits before we separate these two points. Thus, it is customary for quad tree designers to specify some upperbound D on dpeth of T.

→ Given a Set S of n points in plane, we can construct a quadtree T for S so as to spend O(n) time building each level of T. Thus in worst case, constructing such a depth bounded quadtree takes O(Dn) time.

Answering Range Queries with Quadtree:

→ One of queries that quadtrees are often used to answer is range searching. Suppose that we are given a rectangle A aligned with coordinate axes and are asked to use a quadtree T to return all the points in S that are contained in A.

→ The method for answering this query is quite simple. We start with root r of T, and we compare the region R for r to A. If A and R do no intersect at all, then we are done there are no points in subtree rooted at r tha fall inside A.

Performance :

In performing such a range searching query, we can traverse the entire tree T and not produce any output in worst case. Thus, the worst case running time performing a range query in depth D quadtree, with n extrnal nodes is O(Dn).

5.7 K-D Trees

Q9. Discuss briefly K-D Trees.

Ans :

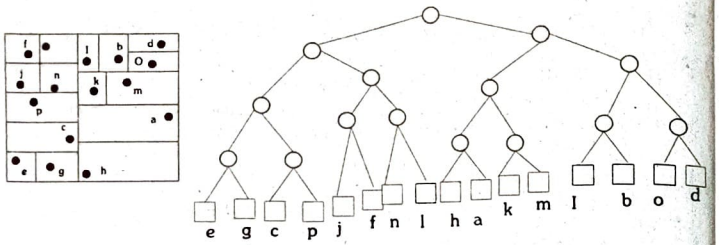
K-D Trees:

There is a drawback to quadtrees, which is that they do not generalize well to higher dimensions. In particular, eah node in a four dimensional analogue of a quadtree can have as many as 16 children. Each internal node in a d-dimensional quadtree can have has many as 2^d children.

→ To overcome the out degree drawback for storing data from dimensions higher than three, data strcuture designers often consider alternative partition tree structures that are binary.

→ Another kind of partition data structure is the K-d tree, which is similar to quadtree structure, but is binary. The K-d tree data structure is actually a family of partition tree data structure all of which are binary partitions trees for storing multidimensional data.

→ The difference is that, when its comes time to perform a split operation for a node v in a K-d tree, it is done with a single line that is perpendicular to one of coordinate axis.



→ There are fundamentally two different kinds of K-d trees, region based K-d trees and point based K-d trees. Region based K-d trees are especially binary versions of quad trees.

→ Each time a rectangular region R needs to split in a region based K-D tree, the region R is divided exactly in half by a line perpendicular to longest side of R.

→ On the other hand, point based K-d trees, perform splits based on the distribution of points inside a rectangular region. The K-d tree of above figure is point based.

→ The advantage of point based K-d trees is that they are guaranteed to have nice depth and construction times. The drawback of point based schemes is that they may give rise to "long and skinny" rectangular regions, which are usually considered bad for most K-d tree query methods. In practice, however, such long-and - skinny regions are rare, and most rectangular regions associated with nodes of a K-d tree are "boxy".

5.8 Recent Trends in Hashing

Q10. Explain the Keyless Vs. Keyed Hash functions.

Ans :

Keyless Vs. Keyed Hash Functions:

→ Generally hash functions are classified as Keyless or Keyed. Keyless hash functions accept a variable length message M and Produce a fixed length hash value, $H:\{0,1\}^* \rightarrow \{0,1\}^n$. Keyed hash functions, on other hand, accept both a variable length message M and a fixed length key K to produce a fixed length hash function, $H_K\{\{0,1\}^K \times \{0,1\}^* \rightarrow \{0,1\}$.

→ Key hash functions can be further classified based on whether the key is private or public. Secretly Hash functions are usually used to build Message Authentication Codes (MAC), the Canonical example is HMAC. If, however, the hash functions are publicly Keyed, they are commonly known as dedicated - Key hash functions.

→ Hash functions designed in dedicated Key setting are families of hash functions where individual member functions are indexed by different Keys.

→ An obvious drawback of hash functions in dedicated -Key setting, however, is a degraded efficiency since in this case the function is required to process an extra input beside message input.

→ In general, a hash function is built out of two components : a Compression function, f and a construction H. The compression function is a function mapping a larger sized input to a smaller fixed sized output.

$f:\{1,0\}^m \rightarrow \{1,0\}^n$, where m>n.

Q11. What are Iterative hash functions? explain.

Ans :

Interative Hash Functions:

→ When Hash functions first emerged, it was realised that most convenient way to hash a message is by first dividing it into several blocks and then iteratively and systematically processing these blocks.

→ Today, this sequential hasing approach is still by far, the most widely used, even with advent of parallel processors.

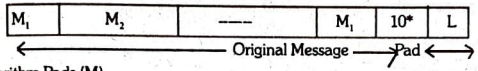
i) **Merkle Damgard Construction:**

Most of today's popular hash functions such as MD5 and SHA1 are based on infamous Merkle - Damgard Construction proposed independently by Merkle and Damgard in 1989.

→ In Markle damaged construction, the message M is first divided into equally sized blocks, M= M1, M2,...Ml. if message M fell over or below the block boundaries it is padded.

→ The message is then iterated repeatedly by calling a fixed Input length comprerssion fucntion.

$f:\{01\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$ accepting two inputs : a message block Mi and either an Initialisation vector IV or a Chain variable both length n:



Algorithm Pads (M)

$d = M+1+64 \bmod m$

$M||1||0^d||\langle M\rangle_{64} \rightarrow \hat{M}$

$M \rightarrow M_1 --------Ml$

ii) **Generic attacks against Markle - Damgard :**

→ Eventually, several weaknesses were found in Merkle - Damgard Construction givin raise to a class of generic attacks that is applicable to any hash function based on plain Merkle Damgard Construction. The difference between generic and dedicated attacks, where dedicated attacks exploit internal structures specific to a particular hash function and thus only affect that hash function.

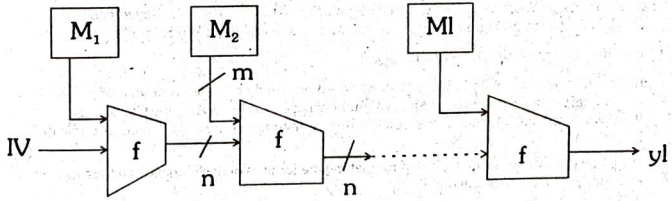
→ The generci attacks against Merkle - Damgard construction are as follows.

a) Collision attack

b) Second Collision attack

c) Related Message attack

d) MAC Forgery attack.



Algorithms MDf

$M \rightarrow M_1 ..... Ml$

$y_0 = IV$

for i = 1 to l do

$y_i = f(M_i, y_i - 1)$

return yl.

Q12. Explain the variants of Merkle Damgard.

Ans :

Variants of Merkle - Damgard:

→ The discovery of Weaknesses drove the research community to propose modified variants of the Merkle-Damgard Constrution that patch such weaknesses. In this we present a few examples of both Keyless and

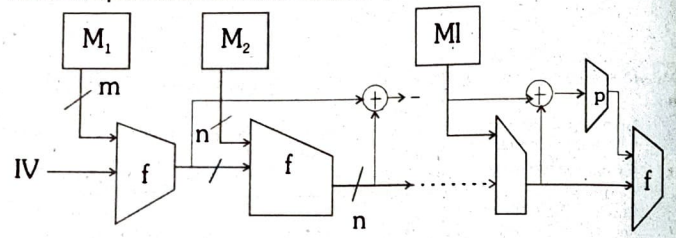
Keyed construction. Some of these constructions use different padding algorithms than the standard one.

Wide and Double pipe :

One of the earliest proposals to enhance the Merkle - Damgard Construction is wide/double pipe by Lucks. Who showed that increasing the size of the internal state to become larger than the size of final hash value, would significantly improve the security of hash function.

3C Construction:

Another variant of Merkle - Damgard construtin is 3C Construction which basically maintains a variable containing a value produced by repeatedly XORing the chanining variables while hashing a message, this variable is then processed in an extra finalisation call to compression function.



Prefix Free, Chop, NMAC and HMAC Construction:

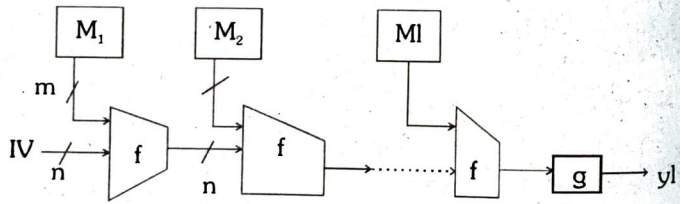
Algorithm NMAC f.g

$M \rightarrow M_1$ ------Ml

$y_0 = IV$

for $i = f(M_i, y_i - 1)$

return $y_F = g(yl)$



Algorithm HMACt

$M \rightarrow M_1$ -----Ml

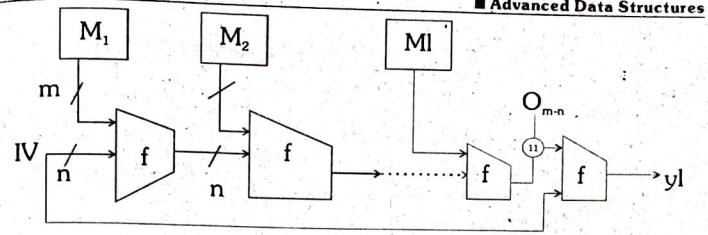
$M_0 = O^m, y_0 = f(M_0, IV)$

for i = 1 to l do

$y_i = f(M_i, y_i - l)$

return

$y_F = f(y_e || O^{m-n}, IV)$



Q13. Discuss the numerical metod for solving partial differential equations on highly evolving grid?

Ans :

- An efficient numerical method is described for solving partial differential equations in problems where traditional eulerian and lagrangian techniques fail.
- The approach makes use of geometrical concept of 'material neighbours, the properties of which make it suitable for solving problems involving large deformation and solid-fluid interactions of a deforming mesh, without need for regridding.
- The approach can also be applied to high order partial differential equations, even in cases where the evolving mesh is highly irregular.