Visit https://www.tutorialsduniya.com for Compiled books, Notes, books, programs, question papers with solutions etc.

Facebook: https://www.facebook.com/tutorialsduniya

Youtube: https://www.youtube.com/user/TutorialsDuniya

LinkedIn: https://www.linkedin.com/company/tutorialsduniya
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Computer System Architecture
- Programming in Java
- Discrete Structures
- Data Structures
- Operating Systems
- Computer Networks
- Algorithms
- Android Programming
- Database Management Systems
- PHP Programming
- Software Engineering
- JavaScript
- Theory of Computation
- Java Server Pages
- Operational Research
- Python
- System Programming
- Microprocessor
- Data Mining
- Artificial Intelligence
- Computer Graphics
- Machine Learning
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
UNIT I

Dictionaries: Sets, Dictionaries, Hash Tables, Open Hashing, Closed Hashing (Rehashing Methods), Hashing Functions (Division Method, Multiplication Method, Universal Hashing), Analysis of Closed Hashing Result (Unsuccessful Search, Insertion, Successful Search, Deletion), Hash Table Restructuring, Skip Lists, Analysis of Skip Lists.

SETS
- A set is a collection of well defined elements. The members of a set are all different.
- A set ADT can be defined to comprise the following operations:

Union(A, B, C): This operation combines all elements of set A and set B and places them in set C.
  Ex: A={1,2,3,4} B={4,5,6} then C={1,2,3,4,5,6}

Intersection(A, B, C): This operation selects common elements from set A and set B (i.e., which appear both in A and B) and places them in set C.
  Ex: A={1,2,3,4} B={4,5,6} then C={4}

Difference(A, B, C): This operation selects only the elements that appear in set A but not in set B and places them in set C.
  Ex: A={1,2,3,4} B={4,5,6} then C={1,2,3}

Merge(A, B, C): This operation combines all elements of set A and set B and places them in set C and discards set A, B.
  Ex: C={1,2,3,4,5,6}

Find(x): Returns the set name in which element ‘x’ is available.
  Ex: A={1,2,3}, B={4,5,6}
  Find(5) returns set B

Member (x, A) or Search (x, A): Checks whether element ‘x’ is available in ‘A’ or Not. If element is available returns search successful otherwise returns search unsuccessful.

MakeNull (A): Makes a Set A to empty

Equal (A, B): Check whether two sets A and B are equal or not.
  Ex: A={1,2,3} B={2,1,3} are equal

Assign (A, B): Assigns Set B elements to set A

Insert (x, A): Inserts an element ‘x’ into set A .Ex: Insert(7,A) now A={1,2,3,7}

Delete (x, A): Deletes an element ‘x’ from set A. Ex: Delete(3,A) now A={1,2,7}

Min(A): Returns minimum value from A . Ex: Min(A)=1.

- A Set can be implemented with the following data structures:
  (a) Bit Vector
  (b) Array
  (c) Linked List
    - Unsorted
    - Sorted
**DICTIONARY**

A *dictionary* is a container of elements from a totally ordered universe that supports the basic operations of inserting/deleting elements and searching for a given element. (OR)

Dictionary is a Dynamic-set data structure for storing items indexed using *keys*. It Supports operations Insert, Search, and Delete.

Ex: hash tables are dictionaries which provide an efficient implicit realization of a dictionary. Efficient explicit implementations include binary search trees and balanced search trees.

**Dictionaries:** A dictionary is a dynamic set ADT with the operations:

1. Makenull \((D)\)
2. Insert \((x, D)\)
3. Delete \((x, D)\)
4. Search \((x, D)\)

Dictionaries are useful in implementing symbol table of a compiler, text retrieval systems, database systems, page mapping tables, Large-scale distributed systems etc.

**Dictionaries can be implemented with:**

1. Fixed Length arrays
2. Linked lists: sorted, unsorted, skip-lists
3. Hash Tables: open, closed
4. Trees: Binary Search Trees (BSTs), Balanced BSTs like AVL Trees, Red-Black Trees Splay Trees, multi way Search Trees like 2-3 Trees, B Trees,
5. Tries

- Let \(n\) be the number of elements in a dictionary \(D\). The following is a summary of the performance of some basic implementation methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Search</th>
<th>Delete</th>
<th>Insert</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Unsorted linked list</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

- Among these, the **sorted list** has the best **average case** performance
- Arrays, sorted linked lists, unsorted linked lists all takes \(O(n)\) time for insert, delete, Search, min operations

**HASH TABLES**

**Necessity of Hash tables:**

We can access any position of an array in constant time (i.e., in \(O(1)\)). We think of the subscript as the key, and the value stored in the array as the data. Given the key, we can access the data in constant time.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>50</td>
<td>55</td>
<td>60</td>
</tr>
</tbody>
</table>
Generally search operation in an array takes \( O(n) \) time. But if we know the positions of each element where it was stored (i.e., at 1\(^{st} \) location the element 20 was stored, at 6\(^{th} \) location the element 45 was stored etc.) then we can directly access the element by moving to that position in \( O(1) \) time. This is the basic idea behind the implementation of hash tables.

**Ex 1:** For example, I used an array with size 10 to store the roll no.’s of 10 students. R.no. 1 was placed at 1\(^{st} \) location, R.no. 2 was placed at 2\(^{nd} \) location, R.no. 3 was placed at 3\(^{rd} \) location and so on..

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Here I can access R.no. 9 directly from the 9\(^{th} \) location, Rno:6 directly from 6\(^{th} \) location. So for accessing the elements always it takes \( O(1) \) time. But in real cases this is not possible.

**Ex 2:** We have a list of employees of a fairly small company. Each of 100 employees has an ID number in the range 0 to 99. If we store the elements (employee records) in the array, then each employee’s ID number will be an index to the array element where this employee’s record will be stored. In this table an employee details can be accessed in \( O(1) \) time.

<table>
<thead>
<tr>
<th>key</th>
<th>Array of Employee Record Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>key 0</td>
<td>[0] Employee record key 0</td>
</tr>
<tr>
<td>key 1</td>
<td>[1] Employee record key 1</td>
</tr>
<tr>
<td>key 2</td>
<td>[2] Employee record key 2</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>key 98</td>
<td>[98] Employee record key 98</td>
</tr>
<tr>
<td>key 99</td>
<td>[99] Employee record key 99</td>
</tr>
</tbody>
</table>

In this case once we know the ID number of the employee, we can directly access his record through the array index. There is a one-to-one correspondence between the element’s key and the array index. In this an employee details can be accessed in \( O(1) \) time.

However, in practice, this perfect relationship is not easy to establish or maintain.

**Ex 3:** the same company might use employee’s five-digit ID number as the key. With 5 digits the possible numbers are from 00000 to 99999. If we want to use the same technique as above, we need to set up an array of size 100,000, of which only 100 elements will be used (only 100 employees are working in that company).
In the above table only 100 locations are useful. Remaining 9900 records storage is wasted. This problem can be overcome with hash tables.

Ex 4: Sometimes companies maintain their employee numbers as alpha numeric letters. An employee ID may be SACET555 but we can’t use this alphanumeric as an array index So we need something which maps a string or a large value into an array index.

**HASH TABLES**

(i) **Hash Table** is a data structure in which keys are mapped to array positions by a hash function. (Or) A Hash Table is a data structure for storing key/value pairs This table can be searched for an item in O(1) time using a hash function to form an address from the key.

(ii) **Hash Function**: Hash function is any well-defined procedure or mathematical function which converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index into an array. (OR)

- Hash function is a function which maps key values to array indices. (OR)
- Hash Function is a function which, when applied to the key, produces an integer which can be used as an address in a hash table.
- We will use \( h(k) \) for representing the hashing function

(iii) **Hash Values**: The values returned by a hash function are called hash values or hash codes or hash sums or simply hashes

(iv) **Hashing** is the process of mapping large amount of data item to a smaller table with the help of a hashing function.

- Hash table is an extremely effective and practical way of implementing dictionaries.
- It takes O(1) time for search, insert, and delete operations in the average case. And O(n) time in the worst case.
- By careful design, we can make the probability that more than constant time is required to be arbitrarily small.
  - Hash Tables are two types they are: (a) Open or External  (b) Closed or Internal
**Collisions:** If x1 and x2 are two different keys, but the hash values of x1 and x2 are equal (i.e., h(x1) = h(x2)) then it is called a collision.

Ex: Assume a hash function = h(k) = k mod 10
- h(19) = 19 mod 10 = 9
- h(39) = 39 mod 10 = 9
here h(19) = h(39) This is called collision.

Collision resolution is the most important issue in hash table implementations. To resolve the collisions two techniques are there.

1. Open Hashing  2. Closed Hashing

**Perfect Hash Function** is a function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such function produces no collisions.  
**Good Hash Function** minimizes collisions by spreading the elements uniformly throughout the array.

There is no magic formula for the creation of the hash function. It can be any mathematical transformation that produces a relatively random and unique distribution of values within the address space of the storage. Although the development of a hash function is trial and error, here are some hints that may make process easier:
- Set the size of the storage space to a prime number. This will help generate a more uniform distribution of addresses.
- Use modulo arithmetic (%). Transform a key in such a way that you can perform X % TABLE_SIZE to generate the addresses.
- To transform a numeric key, try something like adding the digits together or picking every other digit.
- To transform a string key, try to add up the ASCII codes of the characters in the string and then perform modulo division.

(1) **Open Hashing (OR) Separate Chaining:**
- In this case hash table is implemented as an array of linked lists.
- Every element of the table is a pointer to a list. The list (chain) will contain all the elements with the same index produced by the hash function.
- In this technique the array does not hold elements but it holds the addresses of lists that were attached to every slot.
- Each position in the array contains a Collection of values of unlimited size (we use a linked implementation of some sort, with dynamically allocated storage).

Here we will chain all collisions in lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and doesn't require a priori knowledge of how many elements are contained in the collection.

The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.

Let U be the universe of keys. The Keys may be integers, Character strings, Complex bit patterns

- B the set of hash values (also called the buckets or bins). Let B = {0, 1, ..., m - 1} where m > 0 is a positive integer.
A hash function $h: U \rightarrow B$ associates buckets (hash values) to keys.

**Ex: Collision Resolution by Chaining**

Put all the elements that hash to the same value in a linked list. See Figure 1.1.

**Example 1:** See Figure 1.2. Consider the keys $0, 1, 2, 4, 16, 25, 36, 49, 64, 81, 100$. Let the hash function be: $h(x) = x \% 7$

Here list (chain) can be a sorted list or an unsorted list. If it is sorted list then the operations are easy.

**Operations of Open Hashing or Separate Chaining:**

- **Search** $(x, T)$: Search for an element $x$ in the list $T[h(key(x))]$
- **Insert** $(x, T)$: Insert $x$ at the head of list $T[h(key(x))]$
- **Delete** $(x, T)$: Delete $x$ from the list $T[h(key(x))]$
Searching

When it comes to search a value, the hash function is applied to the desired key \( K \), and the array index is produced. Then the search function accesses the list to which this array element points to. It compares the desired key \( K \), to the key in the first node. If it matches, then the element is found. In this case the search of the element is on the order of 1, \( O(1) \). If not, then the function goes through the list comparing the keys. Search continues until the element is found or the end of the list is detected. In this case the search time depends on the length of the list.

For example, In the above list if we want to search a key 100, then find the \( h(100) \). \( h(100)=100\%7=2 \). So directly go to the 2\(^{nd} \) location and search the first node has the element 100 if it is not 100 then go to the next location and check whether it is 100 do this until an element was found or last node is reached.

Algorithm for Separate chaining hashing: search

Open hashing is implemented with the following data structures

```c
struct node {
    int k;
    struct node *next;
};
struct node *r[10];
typedef struct node list;
```

**INSERTION:** To insert a value with key \( K \), find \( h(K) \) and add \( K \) to the collection (chain) in position \( h(K) \).

**Algorithm for Separate chaining Hashing: Insertion**

```c
void insert( key, r )
{
    int i;
    i = hashfunction( key ); /* evaluates h(k) */
    if (empty(r[i])) /**< insert in main array */
        r[i].k = key;
    else /**< insert in new node */
        r[i].next = NewNode( key, r[i].next );
}
```

**Deletion:** To delete a value with key \( K \), find \( h(k) \) and go to the array position \( h(k) \) search the list for the element \( K \) if it was found then remove it from the collection (chain) in position \( h(K) \). otherwise display that “Element was not displayed”

**Advantages**

1) The hash table size is unlimited. In this case you don’t need to expand the table and recreate a hash function.
2) Collision handling is simple: just insert colliding records into a list.

**Disadvantages**

1) As the list of collided elements (collision chains) become long, search them for a desired element begin to take longer and longer.
2) Using pointers slows the algorithm: time required is to allocate new nodes.

**Analysis of Open Hashing:**
(i) Search:

(a) **Best case:** To search a Key K, first find out h(k). Assume time to compute h(k) is O(1) and the Key ‘K’ is available as the first node then the complexity is O(1).

(b) **Average case:** Given hash table T with m slots holding n elements,

\[ \alpha = \frac{n}{m} = \text{average keys per slot.} \]

- \( m \) – number of slots, \( n \) – number of elements stored in the hash table.

(i) **Unsuccessful Search:** Uniform hashing yields an average list length \( \alpha = \frac{n}{m} \)

- Any key not already in the table is equally likely to hash to any of the \( m \) slots.
- To search unsuccessfully for any key \( k \), need to search to the end of the list \( T[h(k)] \), whose expected length is \( \alpha \).
- Adding the time to compute the hash function(1), the total time required is \( O(1+\alpha) \).
- Search time for unsuccessful search is \( O(1+ \alpha) \)

(ii) **Successful Search:**

- The probability that a list is searched is proportional to the number of elements it contains.
- Assume that the element being searched for is equally likely to be any of the \( n \) elements in the table.
- The number of elements examined during a successful search for an element \( x \) is 1 more than the number of elements that appear before \( x \) in \( x \)’s list.
  - These are the elements inserted after \( x \) was inserted.
  - Goal: Find the average, over the \( n \) elements \( x \) in the table, of how many elements were inserted into \( x \)’s list after \( x \) was inserted.

- Let \( x_i \) be the \( i \)th element inserted into the table, and let \( k_i = \text{key}[x_i] \).
- Define indicator random variables \( X_{ij} = I\{h(k_i) = h(k_j)\} \), for all \( i, j \).
- Simple uniform hashing \( \Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m \)

\[ \Rightarrow E[X_{ij}] = \frac{1}{m}. \]

- Expected number of elements examined in a successful search is:

\[
E\left[\frac{1}{n} \sum_{i=1}^{n} \left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right] = \frac{1}{n} \sum_{i=1}^{n} \left(1 + \sum_{j=i+1}^{n} E[X_{ij}]\right) = \frac{1}{n} \sum_{i=1}^{n} \left(1 + \sum_{j=i+1}^{n} \frac{1}{m}\right) = 1 + \frac{1}{nm} \sum_{i=1}^{n} (n - i) = 1 + \frac{1}{nm} \left(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i\right) = 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2}\right) \]

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
Expected total time for a successful search = Time to compute hash function + Time to search

= $O(2+\alpha/2 - \alpha/2n) = O(1+\alpha)$.

If n = O(m), then $\alpha=n/m = O(m)/m = O(1)$.

A successful search takes expected time $O(1+\alpha)$.

Searching takes constant time on average.

* Insertion is $O(1)$ in the worst case.
* Deletion takes $O(1)$ worst-case time when lists are doubly linked.
* Hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining.
* In the average case, the running time is $O(1 + \alpha)$.

It is assumed that the hash value $h(k)$ can be computed in $O(1)$ time. If n is $O(m)$, the average case complexity of these operations becomes $O(1)$!

**CLOSED HASHING (OR) OPEN ADDRESSING:**

**Rehashing**
- Rehashing is resolving a collision by computing a new hash location (index) in the array.
- Rehashing schemes use a second hashing operation when there is a collision. If there is a further collision, we re-hash until an empty "slot" in the table is found.
- The re-hashing function can either be a new function or a re-application of the original one. As long as the functions are applied to a key in the same order, then a sought key can always be located.

**Closed Hashing (or) Open Addressing:**
- In open addressing all elements stored in hash table itself. Each slot of a hash table contains either a key or NIL.
- It shows the way, when collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.
- Open addressing is the standard hash table implementation
- With open addressing, we store only one element per location, and handle collisions by storing the extra elements in other unused locations in the array.
- To find these other locations, we fix some probe sequence that tells us where to look if $A[h(x)]$ contains an element that is not $x$.
- It is based on of resolving collisions by probing for free slots
- The hash formula determines the length and complexity of these probes.

Open addressing provides 3 different probing techniques.

1. Linear Probing
2. Quadratic Probing
3. Double Hashing
1. **Linear Probing:**
   - In case of linear probing, we are looking for an empty spot by incrementing the offset by 1 every time.
   - We explore a sequence of location until an empty one is found as follows:

```
\[ h(x, i) = (h(x) + i) \mod m \]
where \( m \) is the hash table size 
and
\( i = 0, 1, 2, \ldots, m-1 \)
```

It is just \( h(x), h(x)+1, h(x)+2, \ldots \), wrapping around at the end of the array. The idea is that if we can't put an element in a particular place, we just keep walking up through the array until we find an empty slot.

(a) **Insertion(x,T)**: This operation inserts an element \( x \) into a hash table \( T \), while inserting an element if there is a collision it applies linear Probing.

Procedure:

- First it evaluates \( h(x) \), if \( h(x) \) location is empty, then it places \( x \) into \( h(x) \).
- If there is a collision then it applies Linear Probing and locates another slot and if this slot is empty then it places the ‘\( x \)’ into that slot, Other wise it probes to another location, this procedure is repeated until an empty slot is found .
  - If there is no empty location in hash table then insertion is not possible

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
</tr>
</tbody>
</table>

In the above hash table with \( m=5 \) insert the following values. 54, 20,44,33,21

**Insert 54:** \( h(54)=54 \mod 5 = 4 \)  (so place 54 at 4th location)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
<td>54</td>
</tr>
</tbody>
</table>

**Insert 20:** \( h(20)=20 \mod 5 = 0 \)  (so place 20 at 0th location)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
<td>54</td>
</tr>
</tbody>
</table>

**Insert 44:**
\( h(44)=44 \mod 5 = 4 \)  (but at 4th location already an element is existed. So apply linear probing)
\( h(44,1)=(h(44)+1)\mod 5 = (4+1)\mod 5 = 0 \)  (here also collision so probe to next location.)
\( h(44,2)=(h(44)+2)\mod 5=(4+2)\mod 5=1 \)  (1st location is empty. So place 44 at 1st location)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>44</td>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
<td>54</td>
</tr>
</tbody>
</table>
Insert 33: \( h(33) = 33 \mod 5 = 3 \) (so place 33 at 3\(^{rd}\) location)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>44</td>
<td>Empty</td>
<td>33</td>
<td>54</td>
</tr>
</tbody>
</table>

Insert 21:
\( h(21) = 21 \mod 5 = 1 \) (But at 1\(^{st}\) location already an element is there)
\( h(21,1) = (h(21)+1) \mod 5 = (1+1) \mod 5 = 2 \) (2\(^{nd}\) location is empty so place 21 at 2\(^{nd}\) position)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>44</td>
<td>21</td>
<td>33</td>
<td>54</td>
</tr>
</tbody>
</table>

Algorithm for Linear Probing Hashing: Insertion

```c
void insert( key, r[] )
{ int n;
  int i, last;
  i = hashfunction( key );  /*computes h(x)*/
  last = (i+n-1) % m;
  while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i]!=key )
  { i = (i+1) % m;
    if (empty(r[i]) || deleted(r[i]))
      r[i] = key;  /*** insert here ***/
    else Error  /*** table full, or key already in table ***/;
  }
}
```

(b) Search(x,T): Search operation searches for an element x in hash table ‘T’ and returns “search was successful” if the element was found. Otherwise it returns “search was unsuccessful”.

Procedure:
- First evaluates h(x) and examine slot h(x). Examining a slot is known as a probe.
- If slot h(x) contains key x, the search is successful. If the slot contains NIL, the search is unsuccessful.
- There’s a third possibility: slot h(x) contains a key that is not x.
  - Compute the index of some other slot, based on x and which probe we are on.(Apply linear probing)
  - Keep probing until we either find key k or we find a slot holding NIL.

Algorithm for Linear Probing Hashing: Search

```c
int search( key, r[] )
{ int i, last;
  i = hashfunction( key );  /*computes h(x)*/
  last = (i+n-1) % m;
  while ( i!=last && !empty(r[i]) && r[i]!=key )
  { i = (i+1) % m;
    if (r[i]==key)  return( i );
    else         return( -1 );
  }
}
```

(c) Delete(x,T): This operation deletes the key x from hash table ‘T’

Procedure:
- First evaluates h(x) and examine slot h(x). Examining a slot is known as a probe.
- If slot \( h(x) \) contains key \( x \), then delete the element and make that location empty. Otherwise apply linear probing to locate the element. After linear probing also if the element was not found then display that “Deletion is impossible because the element was not in the table”.

**Advantages:**
- All the elements (or pointer to the elements) are placed in contiguous storage. This will speed up the sequential searches when collisions do occur.
- It avoids pointers;

**Disadvantages:**
- Linear probing suffers from primary clustering problem
  - Clustering: Element tend to cluster around elements that produce collisions. As the array fills, there will be gaps of unused locations.
  - Suffers from **primary clustering**:
    - Long runs of occupied sequences build up.
    - Long runs tend to get longer, since an empty slot preceded by \( i \) full slots gets filled next with probability \( (i+1)/m \).
    - Hence, average search and insertion times increase.
- As the number of collisions increases, the distance from the array index computed by the hash function and the actual location of the element increases, increasing search time.
- The hash table has a fixed size. At some point all the elements in the array will be filled. The only alternative at that point is to expand the table, which also means modify the hash function to accommodate the increased address space.

2. **Quadratic Probing:** is a different way of rehashing. In the case of quadratic probing we are still looking for an empty location. However, instead of incrementing offset by 1 every time, as in linear probing, we will increment the offset by 1, 4, 9, 16, ... We explore a sequence of location until an empty one is found as follows:

\[
 h(x, i) = (h(x) + i^2) \mod m
\]

where \( m \) is the hash table size

and \( i = 0, 1, 2, \ldots, m-1 \)

(a) **Insertion**(x,T): This operation inserts an element \( x \) into a hash table \( T \), while inserting an element if there is a collision it applies Quadratic Probing.

**Procedure:**
- First it evaluates \( h(x) \) if \( h(x) \) location is empty, then it places \( x \) into \( h(x) \).
- If there is a collision then it applies Quadratic Probing and locates another slot and if this slot is empty then it places the ‘\( x \)’ into that slot, Other wise it probes to another location, this procedure is repeated until an empty slot is found.
- If there is no empty location in hash table then insertion is not possible.
```c
void insert( key, r[] )
{
    int n;
    int i, last;
    i = hashfunction( key ); /*computes h(x)*/
    last = (i+m-1) % m;
    while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i]!=key )
        i = (i*i+1) % m;
    if (empty(r[i]) || deleted(r[i]))
        r[i] = key; /*** insert here ***/
    else Error     /*** table full, or key already in table ***/;
}
```

For example consider a hash table with size 10. And insert the elements 59,18,49,58,28,21,33 into the list. (E-indicates empty).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>

**Insert 59: h(59)=59mod 10=9** *(place 59 at 9th location)*

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>59</td>
<td></td>
</tr>
</tbody>
</table>

**Insert 18: h(18)=18mod 10=8** *(place 18 at 8th location).*

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>18</td>
<td>59</td>
<td></td>
</tr>
</tbody>
</table>

**Insert 49: h(49)=49mod 10=9** *(place 49 at 9th location. But there is a collision so apply Q.P.)*

\[ h(49,T) = (h(49)+1^2)mod10 = (9+1)mod10 = 0 \] *(Place 49 at 0th location)*

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>18</td>
<td>59</td>
<td></td>
</tr>
</tbody>
</table>

**Insert 58: h(58)=58mod 10=8** *(place 58 at 8th location. But there is a collision so apply Q.P.)*

\[ h(58,T) = (h(58)+1^2)mod10 = (8+1)mod10 = 9 \] *(collision at 9th loc. So apply Q.P.)*

\[ h(58,T) = (h(58)+2^2)mod10 = (8+4)mod10 = 2 \] *(Place 58 at 2nd location.)*

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>58</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>28</td>
<td>18</td>
<td>59</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Insert 28: h(28)=28mod 10=8** *(place 28 at 8th location. But there is a collision so apply Q.P.)*

\[ h(28,T) = (h(28)+1^2)mod10 = (8+1)mod10 = 9 \] *(collision at 9th loc. So apply Q.P.)*

\[ h(28,T) = (h(28)+2^2)mod10 = (8+4)mod10 = 2 \] *(collision at 2nd loc. So apply Q.P.)*

\[ h(28,T) = (h(28)+3^2)mod10 = (8+9)mod10 = 7 \] *(Place 28 at 7th location.)*

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>58</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>28</td>
<td>18</td>
<td>59</td>
<td></td>
</tr>
</tbody>
</table>
Insert 21: \( h(21)=21 \mod 10=1 \) (place 21 at 1st location)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>21</td>
<td>58</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>28</td>
<td>18</td>
<td>59</td>
<td></td>
</tr>
</tbody>
</table>

Insert 33: \( h(33)=33 \mod 10=3 \) (place 33 at 3rd location)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>21</td>
<td>58</td>
<td>33</td>
<td>E</td>
<td>E</td>
<td>28</td>
<td>18</td>
<td>59</td>
<td></td>
</tr>
</tbody>
</table>

(b) Search(\(x,T\)): Search operation searches for an element \(x\) in hash table ‘\(T\)’ and returns “search was successful” if the element was found. Otherwise it returns “search was unsuccessful”.

Procedure:
- First evaluates \(h(x)\) and examine slot \(h(x)\). Examining a slot is known as a probe.
  - If slot \(h(x)\) contains key \(x\), the search is successful. If the slot contains NIL, the search is unsuccessful.
  - There’s a third possibility: slot \(h(x)\) contains a key that is not \(x\).
    - Compute the index of some other slot, based on \(x\) and which probe we are on. (Apply Quadratic probing)
    - Keep probing until we either find key \(k\) or we find a slot holding NIL.

```java
int search( key, r[] )
{
  int i, last;
  i = hashfunction( key ); /*computes \(h(x)\)*/
  last = (i+n-1) % m;
  while ( i!=last && !empty(r[i]) && r[i]!=key )
    i = (i*i+1) % m;
  if (r[i]==key)  return( i );
  else            return( -1 );
}
```

(c) Delete(\(x,T\)): This operation deletes the key \(x\) from hash table ‘\(T\)’

Procedure:
- First evaluates \(h(x)\) and examine slot \(h(x)\). Examining a slot is known as a probe.
  - If slot \(h(x)\) contains key \(x\), then delete the element and make that location empty. Otherwise apply Quadratic probing to locate the element. After Quadratic probing also if the element was not found then display that “Deletion is impossible because the element was not in the table”.
- **Disadvantage**: Can suffer from secondary clustering:
  - If two keys have the same initial probe position, then their probe sequences are the same.
3. **Double Hashing:** It uses two different hash functions

\[
    h_1(x,i) = (h_1(x) + i h_2(x)) \mod m
\]

where \( h_1(x) \) is the first hash function and \( h_2(x) \) is the second hash function, \( m \) is the hash table size, \( i=1,2,3,4 \) etc..

- Here \( h_1 \) and \( h_2 \) are two auxiliary hash functions. \( h_1 \) gives the initial probe. \( h_2 \) gives the remaining probes.
- In this \( h_2(x) \) must be a relatively prime to \( m \), so that the probe sequence is a full permutation of \( \langle 0, 1, ..., m-1 \rangle \).
- Choose \( m \) to be a power of 2 and have \( h_2(x) \) always return an odd number. Or, Let \( m \) be prime, and have \( 1 < h_2(x) < m \).
- Suppose \( h_1(x) \equiv x \mod m \) then \( h_2 \) can be selected as \( h_2(x) = (R-x \mod R) \) where \( R \) is a prime number nearer to \( m \).

**Insert** \((x,T)\): This function inserts the key value ‘\( x \)’ into hash table ‘\( T \)’.

**Procedure:**

- First it evaluates \( h_1(x) \) if \( h_1(x) \) location is empty, then it places \( x \) into \( h_1(x) \).
- If there is a collision then it applies Quadratic Probing and locates another slot and if This slot is empty then it places the ‘\( x \)’ into that. Otherwise it probes to another location, this procedure is repeated until an empty slot is found.
- If there is no empty location in hash table then insertion is not possible.

For example consider a hash table with size 10. And insert the elements 59, 18, 49, 58, 21, 33 into the list.(E-indicates empty).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>

**Insert 59**: \( h_1(59)\equiv 59 \mod 10 = 9 \) (place 59 at 9\(^{th}\) location)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>59</td>
</tr>
</tbody>
</table>

**Insert 18**: \( h_2(18) \equiv 18 \mod 10 = 8 \) (place 18 at 8\(^{th}\) location).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>18</td>
<td>59</td>
</tr>
</tbody>
</table>

**Insert 49**: \( h_1(49) \equiv 49 \mod 10 = 9 \) (place 49 at 9\(^{th}\) location. But element is there so apply D.H). \( h_1(T) = (h_1(49)+1.h_2(49)) \mod 10 \)

Where \( h_2(49) = R - (x \mod R) = (7-49 \mod 7) = 7-0 = 7 \) (select \( R \) as 7 which is a prime nearer to \( m \))

\( h_1(T) = (h_1(49)+1.7) \mod 10 = (9+7) \mod 10 = 6 \) (Place 49 at 6\(^{th}\) position).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>49</td>
<td>E</td>
<td>18</td>
<td>59</td>
</tr>
</tbody>
</table>
**Insert 58:** \( h_1(58) = 58 \mod 10 = 8 \) (place 58 at 8th location. But there is a collision so apply D.H). \( h_1(58, T) = (h_1(58) + 1 \cdot h_2(58)) \mod 10 \)

Where \( h_2(58) = (x \mod R) = (7 - 58 \mod 7) = 7 - 2 = 5 \)

\( h_1(58, T) = (h_1(58) + 1 \cdot 5) \mod 10 = (8 + 5) \mod 10 = 3 \) (Place 58 at 3rd position.)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>58</td>
<td>E</td>
<td>E</td>
<td>49</td>
<td>E</td>
<td>18</td>
<td>59</td>
<td></td>
</tr>
</tbody>
</table>

**Insert 21:** \( h(21) = 21 \mod 10 = 1 \) (place 21 at 1st location)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>21</td>
<td>E</td>
<td>58</td>
<td>E</td>
<td>E</td>
<td>49</td>
<td>E</td>
<td>18</td>
<td>59</td>
<td></td>
</tr>
</tbody>
</table>

**Algorithm for Double Hashing: Insertion**

```c
void insert( key, r )
{
    int i, inc, last;
    i = hash1(key) ;
    inc = hash2(key);
    last = (i+(m-1)*inc) % m;
    while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i]!=key )
        i = (i+inc) % m;
    if ( empty(r[i]) || deleted(r[i]) )
    {
        /*** insert here ***/
        r[i] = key;
        n++;
    }
    else Error     /*** table full, or key already in table ***/;
}
//Where hash1 (key) is the first hash function, hash2(key) is the second hash function .
```

(b) **Search(x,T):** Search operation searches for an element x in hash table ‘T’ and returns “search was successful” if the element was found. Other wise it returns “search was unsuccessful”.

**Procedure:**

- First evaluates \( h_1(x) \) and examine slot \( h_1(x) \). Examining a slot is known as a probe.
- If slot \( h_1(x) \) contains key x, the search is successful. If the slot contains NIL, the search is unsuccessful.
- There’s a third possibility: slot \( h_1(x) \) contains a key that is not x.
- Compute the index of some other slot, based on x and which probe we are on. (Apply Double hashing)
- Keep probing until we either find key k or we find a slot holding NIL.
Algorithm for Double Hashing: Search

```c
int search( key, r )
{
  int i, inc, last;
  i = hash1( key );
  inc = hash2( key );
  last = (i+(n-1)*inc) % m;
  while ( i!=last && !empty(r[i]) && r[i]!=key )
  {
    i = (i+inc) % m;
    if (r[i]==key)  return( i );
    else         return( -1 );
  }
  //Where hash1 (key) is the first hash function,hash2(key) is the second hash function
}
```

(c) Delete(x,T): This operation deletes the key x from hash table ‘T’

Procedure:
- First compute $h_1(x)$ value and examine slot $h_1(x)$. Examining a slot is known as a probe.
- If slot $h_1(x)$ contains key x, then delete the element and make that location empty. Otherwise apply Double Hashing to locate the element. After probing also if the element was not found then display that “Deletion is impossible because the element was not in the table”.

Advantages: Distributes keys more uniformly than linear probing

A Comparison of Rehashing Methods

<table>
<thead>
<tr>
<th>Linear Probing</th>
<th>Quadratic Probing</th>
<th>Double Hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$ distinct probe</td>
<td>No primary clustering;</td>
<td>$m^2$ distinct probe</td>
</tr>
<tr>
<td>Primary clustering</td>
<td>but secondary clustering</td>
<td>No secondary clustering</td>
</tr>
<tr>
<td>sequences</td>
<td>sequences</td>
<td>sequences</td>
</tr>
</tbody>
</table>
HASHING FUNCTIONS

Choosing a good hashing function, \( h(k) \), is essential for hash-table based searching. \( h \) should distribute the elements of our collection as uniformly as possible into the "slots" of the hash table. The key criterion is that there should be a minimum number of collisions.

If the probability that a key, \( k \), occurs in our collection is \( P(k) \), then if there are \( m \) slots in our hash table, a uniform hashing function, \( h(k) \), would ensure:

\[
\sum_{k \mid h(k)=0} P(k) = \sum_{k \mid h(k)=1} P(k) = \ldots = \sum_{k \mid h(k)=m-1} P(k) = \frac{1}{m}
\]

Sometimes, this is easy to ensure. For example, if the keys are randomly distributed in \((0, r]\), then,

\[
h(k) = \text{floor}(mk/r)
\]

will provide uniform hashing.

**Mapping keys to natural numbers**

Most hashing functions will first map the keys to some set of natural numbers, say \((0, r]\). There are many ways to do this, for example if the key is a string of ASCII characters, we can simply add the ASCII representations of the characters mod 255 to produce a number in \((0,255)\) - or we could \text{xor} them, or we could add them in pairs mod \(2^{16}-1\), or ...

Having mapped the keys to a set of natural numbers, we then have a number of possibilities.

1. **MOD FUNCTION:**

   \( h(k) = k \mod m \).

   When using this method, we usually avoid certain values of \( m \). Powers of 2 are usually avoided, for \( k \mod 2^b \) simply selects the \( b \) low order bits of \( k \). Unless we know that all the \( 2^b \) possible values of the lower order bits are equally likely, this will not be a good choice, because some bits of the key are not used in the hash function.

   Prime numbers which are close to powers of 2 seem to be generally good choices for \( m \).

   For example, if we have 4000 elements, and we have chosen an overflow table organization, but wish to have the probability of collisions quite low, then we might choose \( m = 4093 \). (4093 is the largest prime less than 4096 = \(2^{12}\).)

2. **MULTIPLICATION METHOD:**

   - Multiply the key by a constant \( A \), \( 0 < A < 1 \),
   - Extract the fractional part of the product,
   - Multiply this value by \( m \).

   Thus the hash function is:

   \[
h(k) = \text{floor}(m \times (kA - \text{floor}(kA)))
\]

   In this case, the value of \( m \) is not critical and we typically choose a power of 2 so that we can get the following efficient procedure on most digital computers:

   - Choose \( m = 2^p \).
Multiply the \( w \) bits of \( k \) by \( \text{floor}(A \times 2^w) \) to obtain a \( 2w \) bit product.

Extract the \( p \) most significant bits of the lower half of this product.

It seems that: \( A = (\sqrt{5} - 1)/2 = 0.6180339887 \) is a good choice (see Knuth, "Sorting and Searching", v. 3 of "The Art of Computer Programming").

3. UNIVERSAL HASHING

This involves choosing a hash function randomly in a way that is independent of the keys that are actually going to be stored. We select the hash function at random from a carefully designed class of functions.

- Let \( \Phi \) be a finite collection of hash functions that map a given universe \( U \) of keys into the range \( \{0, 1, 2, \ldots, m - 1\} \).
- \( \Phi \) is called universal if for each pair of distinct keys \( x, y \in U \), the number of hash functions \( h \in \Phi \) for which \( h(x) = h(y) \) is precisely equal to \( \frac{|\Phi|}{m} \).
- With a function randomly chosen from \( \Phi \), the chance of a collision between \( x \) and \( y \) where \( x \neq y \) is exactly \( 1/m \).

Example of a universal class of hash functions:

Let table size \( m \) be prime. Decompose a key \( x \) into \( r + 1 \) bytes. (i.e., characters or fixed-width binary strings). Thus

\[
x = (x_0, x_1, \ldots, x_r)
\]

Assume that the maximum value of a byte to be less than \( m \).

Let \( a = (a_0, a_1, \ldots, a_r) \) denote a sequence of \( r + 1 \) elements chosen randomly from the set \( \{0, 1, \ldots, m - 1\} \). Define a hash function \( h_a \in \Phi \) by

\[
h_a(x) = \sum_{i=0}^{r} a_i x_i \mod m
\]

With this definition, \( \Phi = \bigsqcup_{a} \{h_a\} \) can be shown to be universal. Note that it has \( m^{r+1} \) members.

SKIP LISTS

The worst case search time for a sorted linked list is \( O(n) \), as we can only traverse the list and cannot skip nodes while searching. For a balanced binary search tree, we skip almost half of the nodes after one comparison with root. For a sorted array, we have a random access and we apply binary search on arrays.

Can we augment sorted linked list to make the search faster? The answer is “skip list”.

The idea is simple; we create multiple layers so that we can skip some nodes. See the example with 16 nodes, and two layers. The upper layer works as “Express lane” which connects only to main outer stations and the lower layer works as an “Normal Lane” which connects every station. Suppose if we want to search for a key it is carried out as follows:

We start from the first node of the express lane, and keep moving on express lane till we find a node whose next is greater than key. Once we find such a node on the express lane, we move to normal lane and linearly search for key on the normal lane’
ANALYSIS: What is the complexity with two layers?

The worst case complexity is the number of nodes on “express lane” plus the number of nodes in a segment. Here a segment is number of nodes between two “express lane” nodes. So if we have “n” nodes on the normal lane, we have square root(n) nodes on express lane;

n=16 (nodes on normal lane), square root(n) = 4 (nodes on express lane)

We equally divide the normal lane, then there will be square root(n) nodes in every segment of “normal lane”. Square root(n) is actually optimal division with two layers. With this arrangement, the number of nodes traversed for a search will be \(O(\sqrt{n})\)

Can we do better?

The time complexity of skip list can be reduced further by adding more layers. In fact, time complexity of insert, delete, and search can be \(O(\log n)\).

Skip List is a probabilistic alternative to balanced trees.

- Skip lists use probabilistic balancing rather than strictly enforced balancing.
- Although skip lists have bad worst-case performance, no input sequence consistently produces the worst-case performance (like quicksort).
- It is very unlikely that a skip list will be significantly unbalanced. For example, in a dictionary of more than 250 elements, the chance is that a search will take more than 3 times the expected time \(\leq 10^{-6}\).
- Skip lists have balance properties similar to that of search trees built by random insertions, yet do not require insertions to be random.

**Figure 1.4: A singly linked list**

Consider a singly linked list as in Figure 1.4. We might need to examine every node of the list when searching a singly linked list.

**Figure 1.5: Every other node has an additional pointer**

Figure 1.5 is a sorted list where every other node has an additional pointer, to the node two ahead of it in the list. Here we have to examine no more than \(\left\lfloor \frac{n}{2} \right\rfloor + 1\) nodes.
In the list of Figure 1.6, every second node has a pointer two ahead of it; every fourth node has a pointer four ahead if it. Here we need to examine no more than \( \left\lfloor \frac{n}{4} \right\rfloor + 2 \) nodes.

In Figure 1.7, (every \((2^i)\)th node has a pointer \((2^i)\) node ahead \((i = 1, 2,...)\); then the number of nodes to be examined can be reduced to \( \log_2 n \) while only doubling the number of pointers.

- A node that has \(k\) forward pointers is called a level \(k\) node. If every \((2^i)\)th node has a pointer \((2^i)\) nodes ahead, then
  - # of level 1 nodes 50%  
  - # of level 2 nodes 25%  
  - # of level 3 nodes 12.5%  

- Such a data structure can be used for fast searching but insertions and deletions will be extremely cumbersome, since levels of nodes will have to change.
What would happen if the levels of nodes were randomly chosen but in the same proportions (Figure 1.8)?

- Level of a node is chosen randomly when the node is inserted
- A node’s $i^{th}$ pointer, instead of pointing to a node that is $2^{i-1}$ nodes ahead, points to the next node of level $i$ or higher.
- In this case, insertions and deletions will not change the level of any node.
- Some arrangements of levels would give poor execution times but it can be shown that such arrangements are rare.

Such a linked representation is called a skip list.

Each element is represented by a node the level of which is chosen randomly when the node is inserted, without regard for the number of elements in the data structure.

A level $i$ node has $i$ forward pointers, indexed 1 through $i$. There is no need to store the level of a node in the node.

Maxlevel is the maximum number of levels in a node.

- Level of a list = Maxlevel
- Level of empty list = 1
- Level of header = Maxlevel

**Initialization:**
An element NIL is allocated and given a key greater than any legal key. All levels of all lists are terminated with NIL. A new list is initialized so that the level of list = maxlevel and all forward pointers of the list’s header point to NIL.

**Search:**
We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

**Insertion and Deletion:**
- Insertion and deletion are through search and splice
- update $i$ contains a pointer to the rightmost node of level $i$ or higher that is to the left of the location of insertion or deletion.
- If an insertion generates a node with a level greater than the previous maximum level, we update the maximum level and initialize appropriate portions of update list.
- After a deletion, we check to see if we have deleted the maximum level element of the list and if so, decrease the maximum level of the list.

Figure 1.9 provides an example of Insert and Delete. The pseudo code for Insert and Delete is shown below.

```plaintext
search(list, searchkey)
{ x = list \rightarrow header; for (i = list \rightarrow level; i >= 1; i - -) { 
    while (x \rightarrow forward[i] \rightarrow key < searchkey)
        x = x \rightarrow forward[i];
}
    x = x \rightarrow forward[i];
    if (x \rightarrow key = searchkey) return (true)
    else return false;
}
```
Figure 1.9: A skip list

insert (list, searchkey);
{  
  x = list -> header;
  for (i = list -> level; i ≥ 1; i --) {
    while (x -> forward[i] -> key < searchkey)
      x = x -> forward[i];
    update[i] = x
  }  
  x = x -> forward[1];
  if (x -> key = searchkey) return ("key already present")
  else {
    newLevel = randomLevel( );
    if newLevel > list -> level {
      for (i = list -> level + 1; i ≤ newLevel; i ++ )
        update[i] = list -> header;
    }
    x = makenode(newLevel, searchkey);
    for (i = 1, i ≤ newLevel; i++) {
      x -> forward[i] = update[i] -> forward[i];
      update[i] -> forward[i] = x
    }
  }
}
delete (list, searchkey);
    { x = list \rightarrow header ;
        for (i = list \rightarrow level; i \geq 1; i - ) {
            while (x \rightarrow forward[i] \rightarrow key < searchkey)
                x = x \rightarrow forward[i];
            update[i] = x
        }
    x = x \rightarrow forward[1];
    if (x \rightarrow key = searchkey) {
        for (i = 1; i \leq list \rightarrow level; i ++) {
            if (update[i] \rightarrow forward[i] \neq x) break;
            if (update[i] \rightarrow forward[i] = x \rightarrow forward[i];
        }
    }
    free(x)
    while (( list \rightarrow 1) && (list \rightarrow header \rightarrow forward [list+level] = NIL))
        list \rightarrow level = list \rightarrow level - 1

Analysis of Skip Lists

In a skip list of 16 elements, we may have

- 9 elements at level 1
- 3 elements at level 2
- 3 elements at level 3
- 1 element at level 6

- One important question is:
  Where do we start our search? Analysis shows we should start from level \( L(n) \) where
  \[
  L(n) = \log_2 n
  \]
  In general if \( p \) is the probability fraction,
  \[
  L(n) = \log \frac{1}{p} n
  \]
  where \( p \) is the fraction of the nodes with level \( i \) pointers which also have level \( (i + 1) \) pointers.

- However, starting at the highest level does not alter the efficiency in a significant way.
- Another important question to ask is:
  What should be MaxLevel? A good choice is
  \[
  MaxLevel = L(N) = \log \frac{1}{p} N
  \]
  where \( N \) is an upper bound on the number of elements is a skip list.

- Complexity of search, delete, insert is dominated by the time required to search for the appropriate element. This in turn is proportional to the length of the search path. This is determined by the pattern in which elements with different levels appear as we traverse the list.
- Insert and delete involve additional cost proportional to the level of the node being inserted or deleted.
TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

 Compiled Books: https://www.tutorialsduniya.com/compiled-books
 Programs: https://www.tutorialsduniya.com/programs
 Question Papers: https://www.tutorialsduniya.com/question-papers
 Python Notes: https://www.tutorialsduniya.com/python
 Java Notes: https://www.tutorialsduniya.com/java
 JavaScript Notes: https://www.tutorialsduniya.com/javascript
 JSP Notes: https://www.tutorialsduniya.com/jsp
 Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
 OR Notes: https://www.tutorialsduniya.com/operational-research
UNIT-2

BALANCED TREES

Consider the following binary search tree

In this BST
Locating 7 takes 1 comparison
Locating 10 takes 2 comparisons
Locating 3 takes 3 comparisons
Locating 52 takes 7 comparisons
In this tree to locate nth element ‘n’ comparisons are required.

- So the search effect for this particular BST is $O(n)$.
- These types of trees are said to be as unbalanced trees.
- With the same elements consider another tree

In this BST
- Locating 20 requires 1 comparison
- Locating 10, 24 requires 2 comparisons
- Locating 7, 12, 23, 52 requires 3 comparisons

- The maximum search effort of this tree is 3 i.e., the search effort is $O(\log n)$. These types of trees are called balanced trees.
- In the above mentioned trees, when compared to unbalanced trees, in a balanced tree the number of comparisons are reduced from ‘7’ to ‘3’.
- For a completely unbalanced tree with 1000 nodes, the worst case comparisons are 1000, whereas the worst case comparisons for a balanced tree with 1000 nodes will be around 10.

DEFINITION (1): An empty tree is height balanced.

If ‘T’ is a non-empty binary tree with $T_L$ and $T_R$ as its left and right sub trees respectively, then ‘T’ is height balanced if and only if

1. $T_L$ and $T_R$ are height balanced and
2. $|h_L - h_R| \leq 1$, where $h_L$ and $h_R$ are the heights of left sub tree ($T_L$) and right sub tree ($T_R$) respectively.

DEFINITION (2): Balanced trees are trees whose height in the worst case is $O(\log n)$ and for which the operations INSERT, DELETE and SEARCH can be implemented in $O(\log n)$ time.
There exist over one hundred types of balanced search trees. Some of the more common types are: AVL trees, 2-3 trees, red-black trees, Splay trees, B-trees etc.

**HEIGHT:** For all trees the heights are considered from leaf nodes. For the leaf nodes height is 1 and for their parents the height is 2 and so on.

![Figure 1](image)

In figure 1, h(4)=3, h(3)=1, h(7)=2, h(9)=1.

6, 9, 3 are leaf nodes so height is 1.

**AVL TREE DEFINITION:** An AVL tree is a binary search tree in which:

1. The heights of the right sub tree and left sub tree of the root differ by at most 1.
2. The left sub tree and right sub tree are themselves AVL trees.

Maximum depth of AVL tree with 'n' nodes is O(logn).

**AVL TREES**

- In 1962 Adelson-Velskii and Landis introduced a balanced binary tree with respect to the heights of sub trees.
- An AVL tree is a height balanced binary search tree.
- Because of its balanced nature insertion, deletion, search operation for an AVL tree of 'n' nodes takes only O(logn) time.

**AVL TREE BALANCE FACTOR**

- The balance factor of a node 'T' in a binary tree is defined to be h_L-h_R, where h_L and h_R are heights of left and right sub trees of 'T', i.e., BF(T) = h_L-h_R.
- For any node in an AVL tree, the balance factor should be either -1 or 0 or 1.

**Examples of AVL Trees**

- In this tree for node 4, there is no left & right sub trees, so BF(4)=0-0=0
- For node 7 only left sub tree is there, so BF(7)=h_L-h_R=1-0 =1 (height of node 4 is 1)
- The BF(4), BF(7) both are in range -1,0,1. So this tree is an AVL tree.

- In this example: h(8)=1, h(12)=1, h(10)=2, h(7)=3, h(4)=1.
  - BF(8)=0-0=0 (Because no left & right sub trees)
  - BF(12)=0-0=0 (Because no left & right sub trees)
  - BF(10)=h(8)-h(12)=1-1=0
  - BF(4)=0-0=0 (Because no left & right sub trees)
  - BF(7)=h(4)-h(10)=1-2=-1

All balance factors are in the range -1,0,1. So this is also an AVL tree.
NON-AVL TREES

This tree is not an AVL tree.

h(11)=1, h(12)=2, h(10)=3, h(7)=4, h(8)=1, h(4)=1.

BF(11)=0-0=0 (no left & right sub trees)
BF(12)=1-0=1 (only left sub tree with height 1)
BF(8)=0-0=0 (no left & right sub trees)
BF(10)=h(8)-h(12)=1-2=-1
BF(4)=0-0=0 (no left & right sub trees)
BF(7)=h(4)-h(10)=1-3=-2

The root node has a balance factor of -2, but in AVL tree every node should have BF of 0, -1 or 1. So this is not an AVL tree.

NOTE

- Balance Factor -1 indicates the left sub tree height is less than right sub tree.
- Balance Factor 1 indicates right sub tree height is less than left sub tree.
- Balance Factor 0 indicates that left and right sub trees height are equal.
  
  (or)
  
  BF(n)= + indicates left sub tree is heavier.
  BF(n)= - indicates right sub tree is heavier.
  BF(n)= 0 indicates balanced.

OPERATIONS ON AVL TREE

We can apply 3 operations on AVL tree. They are:

(a) Insert(X,T): Insert an element ‘X’ into ‘T’.
(b) Delete(X,T): Delete an element ‘X’ from ‘T’.
(c) Search(X,T): Search and locate the element ‘X’ in ‘T’.

INSERTIONS AND DELETIONS ON AVL TREES

Insertion: Inserting an element into an AVL tree is similar to inserting an element into binary search tree.

- Insert a node at leaf level and if the element to be inserted is less than root node, insert this element in the left sub tree otherwise insert the element in the right sub tree.
- Apply this rule recursively for all nodes in the tree.
In this tree we want to insert ‘15’. 15 is greater than 7 so it will be inserted in the right sub tree of root and ‘15’ is less than ‘16’ so it is inserted in left sub tree of 16, but it is greater than 10, so 15 is inserted as the right child of 10.

- In AVL tree & Binary Search Tree the insertion procedure is same.

- When an insertion operation is applied to an AVL tree, we need to update all the balancing information for the nodes on the path back to the root.
- As we follow the path up to the root and update the balancing information, we may find a node whose new balance may violate the AVL condition.
- When the AVL condition is violated, the tree is no more an AVL tree. So we have to rebalance the tree such that AVL condition is preserved.
- The technique that is used to rebalance the AVL tree is either performing 1) Single rotation or (2) Double rotation.
- Let us call the node that must be rebalanced as ‘x’.
- Since any node has at most two children, and a height imbalance requires that x’s two sub trees height differ by two, it is easy to see that violation may occur in 4 cases:
  1. An insertion into the left sub tree of the left child of ‘x’.
  2. An insertion into the right sub tree of the right child of ‘x’.
  3. An insertion into the left sub tree of the right child of ‘x’.
  4. An insertion into the right sub tree of the left child of ‘x’.
- The cases 1, 2 in which the insertion occurs on the “outside” (i.e., left-left, right-right) is fixed by a single rotation of the tree.
- The cases 3, 4 in which the insertion occurs on the “inside” (i.e., left-right, right-left) is fixed by double rotation.

**SINGLE ROTATION (LEFT – LEFT)**

![Diagram of Single Rotation (Left – Left)](https://www.tutorialsduniya.com)
AVL trees (Single Rotation - LL)

- In this diagram the property of an AVL tree is violated at node 'k2' (BF=2).
- This was because a node is inserted at the sub tree of 'X'.
- The node was inserted to the left child of k2 (i.e., k1). For k1 again in the left sub tree. So this problem is called as 'LL' and it can be fixed by single rotation.
- Before rotation the relationships among all the values in the tree are:
  - k1<k2, X<k1, Y>k1, Z>k2
  - X<k2, Y<k2

After Rotation

- After rotation also all the above relationships are preserved.
- ('Z' is in same position.
  - 'k2' becomes as right child of k1.
  - 'Y' becomes as left child of k2.
  - 'X' is in same position)

Example 1: 9, 7, 6

Because of adding '6' AVL property is violated at '9'.
- 6 was inserted to the left child of '9' (i.e., 7) and for '7' also in left sub tree. So apply LL rotation.

\[
\text{Set } k2->\text{left}=k1->\text{right} \\
\text{K1}->\text{right}=k2.
\]

K1->left, K2->right remains in same position.

SINGLE ROTATION (RIGHT - RIGHT)

Before          After
- In this tree, because of inserting a node in 'Z' sub tree the AVL property is violated at node k1.
- For node k1 the insertion is made to its right child 'k2'. For 'k2' again the insertion is made in its right sub tree.
- This problem is called as 'RR' and fixed by single rotation.

$$X < k_1, \quad k_1 < k_2, \quad Y < k_2, \quad Z > k_1, \quad Y > k_1, \quad Z > k_2.$$ 

- After rotation, all the above relationships are preserved.
- k1’s left child position and k2’s right child position are not changed.
- k2’s left child becomes as k1’s right child.
- ‘k1’ becomes as k2’s left child.

Set: k1->right=k2->left; K2->left=k1

**Insert the following elements into an AVL tree: 3,2,1,4,5,6,7**

- It is not an AVL tree. AVL property is violated at node ‘3’ because of insertion of ‘1’
- For ‘3’, ‘1’ is inserted in LL position.
- So apply LL rotation.
- In this tree, 2 nodes are there with BF=-2. But consider the node from leaf, ’3’ (i.e., near to leaf level). Insertion was made in RR position.
At node ‘2’ the AVL property was violated. For node ‘2’ the insertion is made at RR position.

**DOUBLE ROTATION**

Assume ‘α’ is a node that is not satisfying the AVL property (i.e., the BF is not 0, 1, or -1) because of the following insertions:

1. An insertion into the right sub tree of the left child ’α’.
2. An insertion into the left sub tree of the right child of ’α’.

We can fix these two cases with double rotation.

**LEFT-RIGHT (LR Rotation)**

Before

![Diagram of before state of LR rotation]

After

![Diagram of after state of LR rotation]
This tree is balanced. But if we insert the child for k2 (i.e., either ‘B’ or ‘C’) then the tree become unbalanced at node k3.

- Here the insertion was made to left child of k3 (i.e., k1) and for k1 the insertion was made in right sub tree. So it is called LR imbalance and fixed by double rotation.
- Before rotation the relationship among all elements in the tree are (all these should be preserved after rotation also):

<table>
<thead>
<tr>
<th></th>
<th>k1&lt;k3,</th>
<th>k2&gt;k1,</th>
<th>k3&gt;k1,</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>k1&lt;k2,</td>
<td>k2&lt;k3,</td>
<td>k3&gt;k2,</td>
</tr>
<tr>
<td>A&lt;k1</td>
<td>B&lt;k2,</td>
<td>B&lt;k1,</td>
<td>C&lt;k2,</td>
</tr>
<tr>
<td></td>
<td>C&lt;k3</td>
<td>C&lt;k3</td>
<td></td>
</tr>
</tbody>
</table>

After rotation also the relations must be preserved.

- The positions of ‘A’ and ‘D’ are not changed
- ‘B’ becomes as left child of k1 (B>k1)
- ‘C’ becomes as left child of k3 (C<k3)
- K2 becomes as parent.
- K1 becomes as left child of k2. (k1<k2).
- K3 becomes as right child of k2. (k3>k2).

Set:

\[
k1->right=k2->left; \\
k3->left=k2->right; \\
k2->left=k1; \\
k2->right k3.
\]

**RIGHT-LEFT (RL Rotation)**

Before

This tree is balanced. But if we insert a child to k2 (i.e., B or C) then the AVL property is violated at node k1).

After
In this tree the insertion was made to right child of k1 (i.e., k3) and for k3 the insert was made in left sub tree. So it is called RL imbalance.

- Before rotation the relationships among all elements of the tree are:
  
  \[
  \begin{align*}
  k_1 &< k_3 & k_2 &< k_3 & k_3 &> k_1 & B &< k_2 & C &> k_2 \\
  k_1 &< k_2 & k_2 &> k_1 & k_3 &> k_2 & B &> k_1 & C &> k_1 \\
  & & & & B &< k_3 & C &< k_3
  \end{align*}
  \]

- k2’s children will be adjusted as child for k1 and k3.
- After rotation also all these conditions should be preserved.

- k1’s left child ‘A’, k3’s right child ‘D’ will remain in the same position.
- B becomes as right child of k1 as B>k1.
- C becomes as left child of k3 as C<k3.
- k1 becomes as k2’s left child.
- k3 becomes as k2’s right child.

**Example:**

```
 k3->left=k2->right;
 k1->right=k2->left;
 k2->left=k1;
 k2->right= k3;
```

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
AVL TREE ALGORITHMS

Algorithm AVLInsert(root, newdata)
  if(tree empty)
    insert newdata at root
  else if(newdata<root)
    AVLInsert(left subtree,newdata)
    if(left subtree taller)
      leftBalance(root)
    end if
  else
    AVLInsert(right subtree,newdata)
    if(right subtree taller)
      rightBalance(root)
    end if
  end if
return root
end AVLInsert

Algorithm leftBalance(root)
  if(left subtree high)  //Single rotation
    rotateRight(root)
  else  //Double rotation
    rotateLeft(left subtree)
    rotateRight(root)
  end if
end leftBalance
Algorithm rightBalance(root)
if(right subtree high) //Single rotation
    rotateLeft(root)
else //Double rotation
    rotateRight(right subtree)
    rotateLeft(root)
end if
end rightBalance

Algorithm rotateRight(root)
exchange left subtree with right subtree of left subtree
make left subtree new root
end rotateRight

Algorithm rotateLeft(root)
exchange right subtree with left subtree of right subtree
make right subtree new root
end rotateLeft

Algorithm AVLDelete(root,dltkey)
//This algorithm deletes a node from AVL Tree and rebalances if necessary
if(subtree empty)
    return NULL
end if
if(dltkey<root)
    set left subtree to AVLDelete(left subtree,dltkey)
    if(tree shorter)
        set root to deleteRightBalance(root)
    end if
else if(dltkey>root)
    set right subtree to AVLDelete(right subtree,dltkey)
    if(tree shorter)
        set root to deleteLeftBalance(root)
    end if
else
    save root
    if(no right subtree)
        return left subtree
    else if(no left subtree)
return right subtree
else
    find largest node on left subtree
    save largest key
    copy data in largest to root
    set left subtree to AVLDelete(left subtree,largest key)
    if(tree shorter)
        set root to deleteRightBalance(root)
    end if
end if
return root
end AVLDelete

Algorithm deleteRightBalance(root)
//The [sub]tree is shorter after a deletion on the left branch
if(tree not balanced)
    set rightofRight to right subtree
    if(rightofRight left high)  //Double rotation required
        set leftofRight to left subtree of rightofRight
        //Rotate right then left
        right subtree=rotateRight(rightofRight)
        root=rotateLeft(root)
    else  //Single rotation required
        set root to rotateLeft(root)
    end if
end if
return root
End deleteRightBalance

AVL TREE DELETION

➢ The deletion operation in AVL tree is similar to deletion in BST, except that after deletion if AVL tree is unbalanced make that as balanced tree either by single or double rotation.
➢ Three possibilities for deletion:
    (1) Assume the element to be deleted is ‘x’
        (a) If ‘x’ is a leaf node then remove ‘x’. If the tree is unbalanced make it balanced.
        (b) If ‘x’ has only one child then make parent of ‘x’ parent for its child and remove x.
        (c) If ‘x’ has two children then consider any one of the following possibilities:
            (i) Take the smallest value from the right sub tree. Let it occupy the position of ‘x’ and remove it from right sub tree.
(ii) Take the largest value from the left sub tree. Let it occupy the position of ‘x’ and remove it from left sub tree.

- After deleting the element check the balance factor from the node where deletion was applied to root and if any unbalances are there fix them by rotations.

**Rotation Free Deletion**

- Some deletions do not require any rotation. Because after deletion also the tree preserves AVL property.

- Previously the two sub trees have same height. But after deletion the height of sub trees will differ by '1'. So there will be no problem.
(a) Remove 22 from 69(a).

No rotation required.

(b) Remove 20 and 25 from 69(a).

No rotation required.

(c) Remove 15 from fig(a).

Case 1: Here the balance factor of 'p' is not equal, but after deletion it will be equal.
The height of an AVL tree 'T' storing 'n' keys is $O(\log n)$.
Assume $n(h)$ is the minimum number of nodes in an AVL tree of height 'h'.
An AVL tree of height 'h' contains:
(a) A root node of height 'h'.
(b) One AVL sub tree of height $h-1$ and the other AVL sub tree of height $h-1$ or $h-2$.
So \( n(h) \) can be defined as,

\[
n(h) = 1 + n(h-1) + n(h-2)
\]

where, \( n(h) \) - no. of nodes in a tree, \( n(h-1) \) - no. of nodes in left sub tree, \( n(h-2) \) - no. of nodes in right sub tree, 1 - for root node.

\[
n(h) = 1 + n(h-1) + n(h-2)
\]

for \( h \geq 3 \) (Since \( n(1) = 1 \) and \( n(2) = 2 \))

We know that \( n(h-1) \geq n(h-2) \).

Ex: Suppose a tree has height \( h=4 \), then \( n(3) \geq n(2) \)

- In this \( n(3) \) is 4 and \( n(2) \) is 3 i.e., left sub tree has 3 nodes, and right sub tree has 4 nodes.
- Since the height difference between left and right sub trees of an AVL tree is at most 1 in an AVL tree, the condition

By substituting \( n(h-1) \geq n(h-2) \) in the equation and dropping 1 we get,

\[
n(h) = n(h-1) + n(h-2) + 1 > 2n(h-2)
\]

\[
n(h) > 2n(h-2) \quad \text{(solving this recursively we get series of inequalities)}
\]

\[
> 2[2n((h-2)-2)]
\]

\[
> 4n(h-4)
\]

\[
> 8n(h-6)
\]

\[
> 16n(h-8)
\]

\[
\ldots
\]

\[
n(h) > 2^i \quad \text{n(h-2i) (n(1)=1 and n(2)=2, we can consider 1 or 2 for h-2i)}
\]

Let \( i=\frac{h}{2} - 1 \) (By taking \( h-2i=2 \))

\[
n(h) > 2^i \cdot n(h-2i(h-2i+1))
\]

\[
n(h) > 2^i \cdot n(h-2i+1)
\]

\[
n(h) > 2^i \cdot n(h+h+2)
\]

\[
n(h) > 2^i \cdot n(2)
\]

\[
n(h) > 2^{h/2}/2*n(2) \quad \text{[since n(2)=2 i.e., a tree with height 2 has minimum 2 nodes]}
\]

\[
n(h) > 2^{h/2}/2*2
\]

\[
n(h) > 2^{h/2}
\]

\[
\log (n(h)) > h/2
\]

\[
2\log (n(h)) > h
\]

\[
h = O(\log(n(h)) \quad \text{If number of nodes is } n \text{ then } h = O(\log n).
\]
TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

<table>
<thead>
<tr>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
</tr>
<tr>
<td>Programming in Java</td>
</tr>
<tr>
<td>Data Structures</td>
</tr>
<tr>
<td>Computer Networks</td>
</tr>
<tr>
<td>Android Programming</td>
</tr>
<tr>
<td>PHP Programming</td>
</tr>
<tr>
<td>JavaScript</td>
</tr>
<tr>
<td>Java Server Pages</td>
</tr>
<tr>
<td>Python</td>
</tr>
<tr>
<td>Microprocessor</td>
</tr>
<tr>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>Machine Learning</td>
</tr>
<tr>
<td>Computer System Architecture</td>
</tr>
<tr>
<td>Discrete Structures</td>
</tr>
<tr>
<td>Operating Systems</td>
</tr>
<tr>
<td>Algorithms</td>
</tr>
<tr>
<td>DataBase Management Systems</td>
</tr>
<tr>
<td>Software Engineering</td>
</tr>
<tr>
<td>Theory of Computation</td>
</tr>
<tr>
<td>Operational Research</td>
</tr>
<tr>
<td>System Programming</td>
</tr>
<tr>
<td>Data Mining</td>
</tr>
<tr>
<td>Computer Graphics</td>
</tr>
<tr>
<td>Data Science</td>
</tr>
</tbody>
</table>

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
2-3 TREE

**Definition:** A 2-3 tree is a tree in which each vertex which is not a leaf has 2 or 3 sons and every path from root to leaf is of the same length.

Note: The tree consisting of a single node is a 2-3 tree.

EX (a): 1. In this trees all paths from root to leaf is of same length (i.e., all leafs are at same level).
2. All non-leaf nodes have 2 or 3 sons

- In this 2-3 tree the no. of leaf nodes are six

In the 2-3 trees

(A) **Leaf nodes:** (i) Contains data item (ii) All the leaves exist on same level.

(B) **Non leaf nodes:** (i) Have 2 or 3 children.
   (ii) Every non leaf node have two search values represented as (a:b)
      (1) 'a' is the largest item in left sub-tree.
      (2) 'b' is the largest item in middle sub-tree.

(C) The smallest 2-3 tree is a tree with only one node.

(D) 2-3 tree is balanced, ordered tree (B-Tree of order 3 is known as a 2-3 tree).

(E) The common operations performed on 2-3 trees are:
   (i) Insertion       (ii) Deletion       (iii) Searching

**INSERTION**

To insert a new element ‘a’ into a 2-3 tree we must locate the position for the new leaf ‘l’ that will contain ‘a’.

**ALGORITHM:** Insertion of a new element into a 2-3 tree.

**INPUT:** A non-empty 2-3 tree ‘T’ with root ‘r’ and a new element ‘a’ not in ‘T’.

**OUTPUT:** A revised 2-3 tree with a new leaf ‘a’.
**Procedure:**

Step 1: If the tree 'T' is empty then make 'a' as root.

Step 2: If 'T' consists of a single leaf 'b', then create a new root 'r' and make a & b as the sons of 'r'. If a<b then make 'a' as left son of 'r', otherwise make 'b' as the left son.

**Example:** Before insertion of 'a'  
After insertion of a

Now set the 'r' value as: $L[r] : M[r] \{a : b\}$, where $M[r] = b$, $L[r] = a$

Where $M[r]$ indicates largest value in middle sub-tree and $L[r]$ indicates largest value in left sub-tree.

Step 3: a) If 'T' has more than one vertex, find the location of node 'f' to which 'a' becomes as son.

b) If 'f' has two sons b1 and b2 then make 'a' the appropriate son of 'f'.

(i) Make 'a' the left son if a<b1.
(ii) Make 'a' the middle son if b1<a<b2.
(iii) Make 'a' right son if b2<a.

- Set $M[f]$, $L[f]$ to new values based on a, b1, b2.
- If 'f' has already three sons, then place 'a' in appropriate place. Now 'f' has 4 sons. So to preserve the 2-3 tree property, create a new vertex $f'$ and make the left most 2 sons of 'f' as the sons of 'f' and right most 2 sons of 'f' as the sons of $f'$.

- Make $f'$ as the brother of 'f' (i.e., add $f'$ as the son of 'r', where 'r' is already the parent of 'f')
- After this, suppose at 'r' we get any problem apply the same procedure recursively
ALGORITHM

Algorithm ADDSON (f)
begin
    Create a new vertex f’;
    Make the two right most sons of ‘f’ the left and right sons of f’;
    If ‘f’ has no father then
        begin
            Create a new root r;
            Make 'f' the left most son and f' the right most son of 'r';
        end
    else
        begin
            Let ‘v’ be the father of ‘f’;
            Make f’ a son of ‘v’ immediately to right of ‘f’;
            if ‘v’ now has four sons then ADDSON (V)
        end
end.

Insert the following elements into 2-3 tree.
5  21  8  63  69  32  7  19  25

Insert 5: Empty tree, so make ‘5’ as leaf.

Insert 21: 21>5 so make ‘21’ as the right son and create a new root.

Insert 8: 5<8<21 so make ‘8’ as the middle son and change the values of root node

Insert 63: 63>8 so insert 63 as the right most son.
But this is not 2-3 tree, because the non-leaf node has '4' sons. So divide 'r' into r and r'.

Insert 69: 69>63 so go to the right part of 63.

Insert 32: 8<32<63 so insert '32' in the middle sub-tree.

21<32<63 so insert '32' as the middle son.

Z is violating the property of 2-3 tree. Because It has 4 sons. So divide 'z' into two parts i.e., z and z'.

- Make the left most 2 sons of z as the sons of z.
- Make the right most 2 sons of z as the sons of z'.
- Make z and z' as sons of 'x'.
Insert '7': At 'x' 7<8 so insert in left most sub-tree.

At 'y'  5<7<8 so insert as middle son of 'y'.

Insert 19: With respect to 'x', '19' is middle of 8, 32 i.e., 8<19<32.
So insert '19' in middle sub-tree 'z'.
With respect to 'z', 19<21 so insert '19' as left most son.
Insert 25: 8<25<32, so insert 25 in middle sub-tree 'x'. 25>21 so insert as right son of z.

> Now 'z' is violating the property of 2-3 tree. So split 'z' into z and z''.
> Make the left most 2 sons of z as sons of z.
> Make the right most 2 sons of 'z' as sons of z''.
> Make 'z' and z'' as sons of x.

> In this tree node 'x' is violating the 2-3 tree property.
> So split x into 'x' and x'
> Make the left most 2 sons of 'x' as sons of 'x' and
> Make the right most 2 sons of 'x' as sons of x'.
> Make 'x' and x' as the sons of a new node 'A'.
**DELETION:** An element ‘a’ can be deleted from a 2-3 tree in essentially the reverse manner by which an element is inserted.

To delete an element ‘a’ there are 3 cases to consider.

**CASE 1:** If only one element is in the tree i.e., ‘a’ then remove element ‘a’ and make tree empty.

**CASE 2:** If ‘a’ is the son of a vertex having 3 sons then remove ‘a’.

**CASE 3:** If ‘a’ is the son of vertex ‘f’ having 2 sons ‘a’ and ‘s’ then 2 possibilities are there:

(a) If ‘x’ is the root. Remove ‘a’ and ‘x’ and leave the remaining son ‘s’ as the root.

(b) If ‘f’ is not the root, suppose ‘f’ has a brother ‘g’ to its left, if ‘g’ has only two sons make ‘s’ the rightmost son of ‘g’ and remove ‘a’ and call the deletion procedure recursively to delete ‘f’ and ‘x’.

After removing ‘a’ and adjusting ‘s’ to ‘g’, ‘f’ is empty, so delete it. Now ‘x’ has only one child, so delete it.

(c) If ‘f’ is not the root, suppose ‘f’ has a brother ‘g’ to its left and if ‘g’ has three sons make the rightmost son of ‘g’ the left son of ‘f’ and remove ‘a’ from the tree.
After removing ‘a’, ‘f’ has only one child. But ‘g’ has already 3 sons so make ‘d’ as the left son of ‘f’.

→ A right brother to ‘f’ is also handled in similar way.
→ While deleting elements update M[f] and L[f] values at non-leaf nodes.

Example 1:

(a) Delete 47:

(b) Delete 63:

→ Delete ‘47’ and change the values at non-leaf node.
→ At root change the keys the keys from 10:47 to 10:32

→ Delete ‘63’ so that ‘z’ has only one son ‘69’. Adjust ‘69’ as the right most son of ‘y’.
→ Now z is empty so remove ‘z’ from the tree.
Example 2:

Delete 47:

-> After removing '47' from 'z' 36 was adjusted to 'p'.
-> Now 'z' was empty so remove it.
-> if 'z' is removed 'b' has only one son.
-> So adjust 'p' as the son of 'a' and remove 'b'.
Now ‘r’ has only one son so remove ‘r’ and make ‘a’ as the root.

Final tree is:

Example 3:
Delete 4:

Now 'a' has only one son (i.e., b) so add 'b' as left most son of a' and remove 'a'.

Now 'r' has only one son (i.e., a') so remove 'r'. The final tree is:
SEARCHING

- Searching an element ‘a’ with in a 2-3 tree starts from the root node.
- The child node of the root where the key is searched is determined based on the relation between element ‘a’ and M[f], L[f] where M[f] is the largest element in the left sub-tree and L[f] is the largest element in left sub-tree.
- The searching continues in the same manner until a leaf node is reached.

```java
search (a, r)
{
    if (r is leaf node)
        return r;
    else if (a <= L[r])
        return search (a, r->left);
    else if (a<=M[r])
        return search (a, r->middle);
    else
        return search (a, r->right);
}
```

A general algorithm for performing an insertion

1. Create a new leaf node
2. If the tree is empty
   then perform an insertion into an empty tree
   return
3. If the tree contains only a leaf
   then create a non-leaf (root) node
   connect the two leaves so as to be ordered
   set the key indicators of the root node
   return
4. a) Obtain the search path from the root node to insertion point
   b) If the parent of the new leaf has two children
      then append new leaf to parent so as to be ordered
      update the key indicators (LDATA & MDATA)
      return
      else determine insert position of new leaf
      call procedure to perform the insertion and rebalance the tree

A general algorithm for rebalancing. (We are given the address of the new node and the position where the new node is to be inserted (i.e., position 1, 2, 3, or 4)).

1. Create a new non-leaf brother for the parent node
2. Split the four children between the parent node and its new brother
   Update the key indicators (i.e., LDATA and MDATA)
3. If the current parent has no ancestor then create a new root node
   connect current parent as its left subtree
   update key indicators of new root node
   return
else obtain grandparent of current parent
   determine insertion position of current parent’s brother (i.e., position 2, 3 or 4)
   If grandparent has only two children
   then insert current parent’s brother
   update key indicators
   return
   else recursively invoke rebalancing procedure with the grandparent node which has four children. [Note that the current parent’s brother and its position (2, 3 or 4) is made available to the procedure].

**NODE STRUCTURE:** Two different node structures are used to represent a 2-3 tree.

The leaf node is defined as:

```plaintext
LEAF
TAG   DATA
```

The non-leaf node is defined as:

```plaintext
NONLEAF
TAG   LDATA   LPTR   MDATA   MPTR   RPTR
```

For non-leaf node TAG value is **True** and for leaf node TAG value is **False**.
UNIT-3

BINARY HEAP

**DEFINITION:** A heap is a binary tree structure with the following properties

1. The tree is a complete binary tree (Structuring Property).
2. The key value of each node is greater than or equal to the key value in each of its descendents for a min heap (Ordering Property).

COMPLETE BINARY TREE

It is a Binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.

Example of Min heaps: Here the key value of each node is less than or equal to the key value in each of its descendents.

Ex:- Complete Binary tree

![Complete Binary Tree](image)

Not-complete Binary tree:-

![Not-Complete Binary Tree](image)

Example of max heaps:-

![Max Heaps](image)

Examples of Min Heap

![Min Heap Examples](image)

HEAP IMPLEMENTATION

Generally a complete binary tree is so regular, so it can be represented in an array and no pointers are necessary. The relation between a node and its children is fixed and can be calculated as shown below:

1. For a node located at index ‘i’ its children are found at
   a) Left child: $2i+1$  
   b) Right child: $2i+2$

2. The parent of a node located at index ‘i’ is located at $(i-1)/2$. 

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
Given the index for a left child 'j', its right sibling is found at j+1. Given the index for a right child 'j', its left sibling is found at j-1.

Given the size n, of a complete heap, the location of first leaf is at \( \frac{n}{2} \). Given the location of the first leaf element, the location of the last non leaf element is one less.

**Example: Max Heap**

![Max Heap Diagram](image)

Array representation of the above tree

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td>56</td>
<td>32</td>
<td>45</td>
<td>8</td>
<td>23</td>
<td>19</td>
</tr>
</tbody>
</table>
```

i.e., the index of 32 is 2, so the index of its left child 23 is at \( 2 \times 2 + 1 \) the right child 19 is at \( 2 \times 2 + 2 \).

**Min heap**

![Min Heap Diagram](image)

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>15</td>
<td>17</td>
<td>25</td>
<td>30</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>
```

**TWO COMPLETE BINARY TREES**

![Complete Binary Trees](image)

**BASIC HEAP OPERATIONS**

Two basic operations are performed on a heap. They are:

1) Inserting a node into Binary heap.
2) Deleting a node from Binary heap.

**INSERT OPERATION**
Assume that we have a complete binary tree with 'N' elements whose elements satisfy the ordering property of heap.
All these elements are stored from 0 to N-1 locations of array.
Insert a new element 'X' at the last location. If the binary heap property is maintained then there is no need to change the position of any element.
But if the heap property is violated then swap 'X' and its parent. Swap 'X' until the correct location of 'X' is found. Once the correct location is found then place 'X' such that the heap order property is maintained.
This general strategy is known as a percolate up, i.e., the new element is percolated up the heap until the correct location is found.

Example: Insertion into max heap

Insert 68 at the last location. But by inserting '68' the heap ordered property is violated so swap '68', '45'.

68 > 56. In max heap the parent should be greater. So swap 68 & 56.

Example: Insertion into min Heap

In min Heap every element at parent is less than its children.

Insert 14

Now this is not heap, because 31 > 14. So exchange 31 & 14.
This is a min heap

Note: If we store heap into array from location 1 to n instead of 0 to n-1 then for a node 'i' its left child is in position $2i$, the right child is in position $2i+1$ and parent of 'i' is available at $[i/2]$.

Algorithm for inserting an element into min heap

```c
void insert(x, a) {
    if (n=max) {
        print:"Heap is full"
        return;
    }
    for (i=n; a[i/2]>x; i=i/2) {
        a[i]=a[i/2];
        a[i]=x;
    }
}
```

Consider the above example of inserting 14

14 was inserted at position '11'.

for(i=11; a[11/2]>x; )
    a[5]=31, so
    i.e., a[11]=31
now i=i/2=11/2=5
for( ; a[5/2]>x; )
    a[2]>x
    21>14 true, so
    a[5]=a[5/2]
    a[5]=21
    i=5/2=2
a[1]>14 is false so control come out of for loop & a[2]=14 is assigned.
DELETE OPERATION

- In Binary Heap, if it is max heap the maximum element is removed from heap. If it is min heap the minimum element is removed from heap.
- In max heap, maximum element is available at root.
- In min heap, minimum element is available at root.
- Delete root element (i.e., max or min) from heap and place the last element 'x' in root position.
- Now compare 'x' with its children. If 'x' is greater than any one of its children then move 'x' one level down and bring its child one level up.
- We repeat this process until 'x' is placed in correct position such that the heap property is maintained.

Example: Deletion from max heap

| Replace root with the last element of heap. |
| Now this is not max heap so adjust it. |
| 19<56 so exchange or swap them. |

Example: Deletion from min heap

| Remove 13 and place last element from heap at root |
Now it is not heap, so select the least element from children of ‘31’ and exchange 31 with that. ‘14’ is least among 14, 16 so replace 31 with 14.

Now consider the least element among children of 31, i.e., among 19, 21. 19 is least so exchange 31 and 19.

Next consider the least element among children of 31, i.e., 65, 26. 26 is least element, so exchange 31 and 26.

Algorithm for Deleting minimum element from min heap

```c
int DeleteMin(a)
{
    if(n==0)
    {
        printf:"heap is empty";
        return 0;
    }
    min=a[1];
    last=a[n];
    for(i=1;i*2<=n;i=c)
    {
        c=i*2;
        if(c+1!=n && a[c+1]<a[c])
            c++;
        if(last>a[c])
            a[i]=a[c];
        else
            break;
    }
    a[i]=last;
    return min;
}
```
BUILD A HEAP (OR) CREATE HEAP

- Given a filled array of elements in random order, to build the heap we need to rearrange the data so that each node in the heap is greater than its children in max heap or each node in the heap is less than its children in min heap.
- The general algorithm is to place the N keys into the tree in any order, maintaining the structuring property.
- The percolatedown(i) algorithm percolates down from node ‘i’. Perform the algorithm build heap to create a heap-ordered tree.

Algorithm BuildHeap
{
    for(i=N/2;i>0;i--)
        percolateDown(i);
}

Algorithm percolateDown(i)
{
    for(tmp=a[i];2*i+1<n;i=c)
    {
        c=2*i+1;
        if(c+1!=n & & a[c+1]>a[c])
            c++;
        if(tmp<a[c])
            a[i]=a[c];
        else
            break;
    }
    a[i]=temp;
}

EXAMPLE:

| 8 | 19 | 23 | 32 | 45 | 56 | 78 |

Create max Heap from these elements.

Among \(a[3]\), \(a[4]\) \(a[4]\) is large. Now compare \(a[4]\) with \(a[1]\). \(a[4]\) is greater so exchange with \(a[1]\).

Among \(a[1]\) and \(a[2]\), \(a[2]\) is greater. Now compare \(a[2]\) with \(a[0]\). \(a[2]\) is greater than \(a[0]\) so exchange these two.

Now the root node (i.e., \(a[0]\) is in correct position).

Apply the same for every node.
BINOMIAL QUEUES

- Binomial Queue is a collection of heap-ordered Binomial trees which is also known as a forest.
- Binomial trees:
  a. A Binomial tree of height 0 is a one-node tree.
  b. A Binomial tree $B_k$ of height ‘$k$’ is formed by attaching a binomial tree $B_{k-1}$ to the root of another binomial tree $B_{k-1}$.

$B_0$ is a Binomial tree of height of 0.
$B_1$ is formed by attaching a binomial tree $B_0$ to the root of another Binomial tree $B_0$.

A Binomial tree $B_2$ is formed by attaching a binomial tree $B_1$ to the root of another Binomial tree $B_1$.

A Binomial tree $B_3$ is formed by attaching a Binomial tree $B_2$ to the root of another Binomial tree $B_2$.

A Binomial tree $B_4$ is formed by attaching a binomial tree $B_3$ to the root of another Binomial tree $B_3$.

- A Binomial Queue contains at most one binomial tree of every height.
- A Binomial tree of height ‘$k$’ has exactly $2^k$ nodes.
  Example: A binomial tree of height 3 have exactly $2^3 = 8$ nodes.
- The no. of nodes at depth ‘$d$’ is the binomial coefficient $\binom{k}{d}$
- The no. of nodes at depth ‘0’ of binomial tree of height ‘4’ is

$$\binom{4}{0} = \frac{4!}{0!(4-0)!} = 1$$ node

At depth 1 there are $\binom{4}{1} = \frac{4!}{1!(4-1)!} = \frac{4!}{1\times3} = 1\times2\times3\times4 = 4$ nodes.

At depth 2 there are $\binom{4}{2} = \frac{4!}{2!(4-2)!} = \frac{4!}{2!2!} = 6$ nodes.

- A Binomial tree $B_k$, consists of a root with children as $B_0$, $B_1$, $B_2$ ........... $B_{k-1}$.

Ex: A Binomial tree $B_4$ consists of a root with children as $B_0$, $B_1$, $B_2$, $B_3$. 
**Note:** The degree and depth of a binomial tree with at most n nodes is at most log(n). Each tree “doubles” the previous.

**BINOMIAL QUEUE OPERATIONS**

Various operations that can be performed on Binomial Queue are:

1. Merging of two Binomial Queues. (Union or Meld)
2. Insertion of an element into a Binomial Queue.
3. Deletion of an element from a Binomial Queue.
4. Find Minimum element from Binomial Queue.

**Merging of Two Binomial Queues**

The merge is performed by essentially adding the two queues together.

**Procedure**

**Input:** Two Binomial Queue $H_1$ and $H_2$.

**Output:** A Binomial Queue $H_3$ in which the merged result is stored.

1. $H_3$ can contain only one $B_k$ for each $k$.
2. If a Binomial tree $B_k$ of height $k$ does not exist either in $H_1$ or $H_2$ then don’t add $B_k$ to $H_3$.
3. If only one of $H_1$ and $H_2$ contain a $B_k$ add it to $H_3$.
4. If $H_1$ and $H_2$ both contain a $B_k$ merge them and add $B_{k+1}$ to $H_3$.
5. Now if $H_1$, $H_2$ or both contain a $B_{k+1}$ merge until there are Zero or one of them is remained.

- Merging operation is similar to Binary Addition
  - Adding bits corresponds to merging trees.
  
  To compute value of bit ‘k’ for $H_3$
  
  - Add bit ‘k’ from $H_1$ and $H_2$ and carry obtained from position $k-1$.
  - May generate carry bit to position $k+1$.

  **Ex.:** $0+0=0$ i.e., if there is no $B_k$ in $H_1$ or $H_2$ so it is not added to $H_3$.
  
  $1+0=1$ (or) $0+1=1$ i.e., if there is only one $B_k$ either in $H_1$ or $H_2$ then add it to $H_3$.
  
  $1+1=10$ i.e., if there are two $B_k$’s one in $H_1$ and other in $H_2$ then $B_{k+1}$ is created by merging $B_k$ and it will be considered as carry.
STEP 1: $B_0$ does not exist in $H_1$. But it is present in $H_2$. So add it to $H_3$.

STEP 2: $B_1$ exists in both $H_1$ and $H_2$. So merge them, so that we will get $B_2$.

STEP 3: $B_2$ exists both in $H_1$ and $H_2$ and we obtained one more $B_2$ from step 2. Now there are total 3 $B_2$'s. From these 3 trees we add 1 to result and other 2 are merged. Tree with root 23 is added to the result and trees with root 12 and 14 are merged.

Example 2: Merge the following Binomial queues $H_1$ and $H_2$
1) Two queues $H_1$ and $H_2$ both have $B_0$’s so merge them, so that we will get $B_1$.

2) In $H_2$ one $B_1$ is present and in above step we obtained a $B_1$. So merge them.

3) Total we have 3 $B_2$’s one in $H_1$ and other in $H_2$, and one more is obtained in previous step as carry. From these three trees add carry to merged result $H_3$ and remaining two (which is in $H_1$ and $H_2$) merge.

4) In $H_1$, $B_3$ is present and in above step one more $B_3$ is obtained, so merge the two $B_3$’s to get a $B_4$.
• Merging two binomial trees takes constant time.
• There are \( O(\log n) \) Binomial trees, so the merge takes \( O(\log n) \) time in worst case.
TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
**INSERTION:**
- Insertion is a special case of merging, since we merely create a one-node tree and perform a merge.
- The worst case time complexity of this operation is $O(\log n)$.

Create a binomial heap by inserting all the elements from 1 to 11.
IMPLEMENTATION OF BINOMIAL QUEUES

A binomial queue can be implemented as an array of linked lists. Each node in a tree is represented as a linked list with 3 fields (i.e., data field, child field, sibling field).

```c
struct Binnode
{
    int ele;
    struct Binnode  *child;
    struct Binnode  *sibling;
};
struct collection
{
    int currentsize;
    Bintree Tree[MAXTREE];
};
typedef struct Binnode *position;
typedef struct collection *BinQueue;
typedef struct Binnode *BinTree;
```
Binomial Queue is an array Tree [MAXTREE], which stores the addresses of roots of every Binomial tree in the Binomial Queue.

**Example:**

```
H_3:  13
     /   \
    23    12
     /     / \
   51    24  24
     /     /   / \
   65  24  14  16
      /     /     / \
     65  26 16  18
```

**This queue is stored:**

```
0    1    2    3    4    5    6    7
  NULL 13  23  51  24  65  12  24
       
```

Every Binomial tree in queue is represented as follows:

```
12  
   / \
  21 24
     /   / \
    65 26 16
       /     / \
      18  18
```

From the 3rd cell children of ‘12’ it considers only the 14 (which has more height) as its child and ‘24’ is considered as 14’s sibling. ‘21’ is considered as 24’s sibling.

**Merging Two Binomial trees of same height:**

```
T_1
  12
   / \
  24 23
     /   \
   23
```
```
T_2
  14
   / \
  24 16
     /   / \
    26 16 18
       /     / \
      18
```

```
12
+/  \\
  14
   /   \
   24 23
      /   \
     21
```

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
Among $T_1$ and $T_2$ select the minimum as root and other as child for root. If the root has already a child make this as the sibling to the new child i.e., among 12, 14 12 is minimum so make it as root, ‘14’ as child of ‘12’. The previous child of 12 (i.e., 24) make as sibling to ‘14’.

**Routine to Merge 2 Binomial Trees of equal size**

```c
BinTree combineTrees(BinTree T1, BinTree T2) {
    if(T1->ele > T2->ele)
        return combineTrees(T2,T1);
    T2->sibling=T1->child;
    T1->child=T2;
    return T1;
}
```

**Algorithm to Merge two Binomial Queues or Priority Queues**

- The following algorithm combines two Binomial Queues $H_1$ and $H_2$.
- It places the result in $H_1$ and making $H_2$ Empty.
- $T_1$ and $T_2$ are the trees in $H_1$ and $H_2$ respectively and carry is the tree carried from previous step.
- !! $T_1$, is 1 if $T_1$ exists and is 0 otherwise.
- !! $T_2$, is 1 if $T_2$ exists and is 0 otherwise.
- !! carry, is 1 if carry exists and is 0 otherwise.
- Depending on each of eight possible cases, the tree that results for rank ‘i’ and the ‘carry’ tree of rank i+1 is formed.
- This process proceeds from rank 0 to the last rank in the resulting Binomial Queue.

**Routine to Merge Two Binomial Queues or Priority Queues**

```c
BinQueue Merge(BinQueue H1, BinQueue H2) {
    BinTree T1,T2,carry=NULL;
    int i,j;
    if (H1->currentsize+H2->currentsize > capacity)
        print "Merge would exceed capacity";
    H1->currentsize = H1->currentsize+H2->currentsize;
    for(i=0,j=1;i<=H1->currentsize;i++,j=j*2)
    {
        T1=H1->Tree[i];
        T2=H2->Tree[i];
        switch (!!T1+2*!!T2+4*!!carry) {
            case 0: case 1:break;
            case 2: H1->Tree[i]=T2;
                    H2->Tree[i]=NULL;
                    break;
            case 4: H1->Tree[i]=carry;
                    carry=NULL;
                    break;
            case 5: break;
            case 6: break;
            case 7: break;
        }
    }
}```
case 3:carry=combineTrees(T1,T2);
    H1->Tree[i]=NULL;
    H2->Tree[i]=NULL;
    break;

case 5:carry=combineTrees(T1,carry);
    H1->Tree[i]=NULL;
    break;

case 6:carry=combineTrees(T2,carry);
    H2->Tree[i]=NULL;
    break;

case 7:H1->Tree[i]=carry;
    carry=combineTrees(T1,T2);
    H2->Tree[i]=NULL;
    break;
}
}
return H1;

DELETION OF AN ELEMENT FROM BINOMIAL QUEUE

- In deletion operation the smallest element is deleted from Binomial queue.
- Deletion is performed by first finding the Binomial tree with the smallest root.
- Let the tree with smallest root be B_k, and let the original priority queue be H.
- Form a new Binomial queue H_1, by removing the binomial tree B_k from the forest of trees in H.
- Form a new Binomial queue H_II, by removing the root of B_k creating binomial trees B_0, B_1, B_2, B_3, ... B_{k-1}.
- Now merge H_1 and H_II. This is the final result of deleting minimum element from min heap binomial queues.

Example:

Apply Delete operation to H.
Step 1) Find the tree with smallest root i.e., 12.
Step 2) Create H_1 by removing the Binomial tree with root 12 from H.

Step 3) Create H_II by removing the root of B_3 from H. So add B_0, B_1, B_2 to H_II

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
The resultant Binomial Queue after deleting the minimum element:

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
IMPLEMENTATION OF DELETION ALGORITHM

```c
int DeleteMin(BinQueue H)
{
    int i, j, MinTree, MinItem;
    BinQueue DeletedQueue;
    position DeletedTree, oldroot;
    if (IsEmpty(H))
    {
        printf("Empty Binomial Queue").
        return -1;
    }
    MinItem = Infinity;
    for (i = 0; i < MaxTrees; i++)
    {
        if (H->Tree[i] && H->Tree[i]->ele < MinItem)
        {
            MinItem = H->Tree[i]->ele;
            MinTree = i;
        }
    }
    DeletedTree = H->Tree[MinTree];
    oldroot = DeletedTree;
    DeletedTree = DeletedTree->child;
    free(oldroot);
    DeletedQueue = Initialize();
    DeletedQueue->currentsize = (1 << MinTree) - 1;
    for (j = MinTree - 1; j >= 0; j--)
    {
        DeletedQueue->Tree[j] = DeletedTree;
        DeletedTree = DeletedTree->sibling;
        DeletedQueue->Tree[j] -> sibling = NULL;
    }
    H->Tree[MinTree] = NULL;
    H->currentsize = H->currentsize - (DeletedQueue->currentsize + 1);
    Merge(H, DeletedQueue);
    return MinItem;
}
```
In the above algorithm, if(Is_empty(H)) statement checks whether the Binomial Queue is empty or not. If it is empty then deletion is not possible.

- for(i=0,...) loop finds the tree with minimum root.
- Next H^{1} is created by removing the root and create B_{0}, B_{1}....B_{k-1} by the for(j=1,...) loop. These are stored in DeletedQueue. (DeletedQueue->Tree[0], DeletedQueue->Tree[1],...)
- H^{1} is created by H->Tree[MinTree]=NULL; statement.
- Both H and H^{1} are merged by calling the function Merge(H,...).

**LAZY Binomial Queues**

- Binomial queue in which merge is done lazily.
- Here to merge two Binomial Queues, we simply concatenate the two lists of Binomial trees.
- In the resulting forest, there may be several trees of same size.
- Because of the lazy merge, merge and insert are both worst case O(1) time.

**DeleteMin**

- It converts lazy Binomial Queue into a standard Binomial Queue.
- Do DeleteMin as in standard queue.

**Priority Queues Applications**

1) Implementing scheduler in OS, and distributed systems.
2) Representing event lists in discrete event simulation.
3) Implementing numerous graph algorithms efficiently.
4) Selecting K^{th} largest and K^{th} smallest elements in lists.
5) Sorting applications.
6) A* Search.
7) Huffman encoding.
8) Network bandwidth management.
### Time Complexity of Binomial Heap Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binomial Heap</th>
<th>Amortized Binomial Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make-Heap</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Insert</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Delete min (or max)</td>
<td>O(log n)</td>
<td>-</td>
</tr>
<tr>
<td>Merge (Union Or Meld)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Extract-Min</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>
UNIT – IV

GRAPHS

Graph is a non-linear data structure. In tree structure, there is a hierarchical relationship between parent and children, i.e., one parent and many children. On the other hand, in graph, relationship is less restricted. Here relationship is from many parents to many children.

Some examples of graph structures:
1) Air lines
2) Konisberg’s bridge
3) Flowchart of a program etc.

GRAPH TERMINOLOGIES

GRAPH: A graph G consists of two sets G = (V, E)
i) A set V, called set of all vertices (or nodes)
ii) A set E, called set of all edges (or arcs). This set E is the set of pair of elements from V.

DIGRAPH: A digraph is also called a directed graph. It is a graph G, such that G = (V, E), where V is the set of all vertices and E is the set of ordered pair of elements from V. Here if any ordered pair (v_i, v_j) is in E then there is an edge directed from v_i to v_j (indicated by arrow sign). In case of undirected graph, pair (v_i, v_j) is unordered i.e., (v_i, v_j) and (v_j, v_i) are the same edges, but in case of digraph they correspond to two different edges.

WEIGHTED GRAPH: A graph or digraph is termed as weighted graph if all edges in it are labeled with some weights. The weights may refer distance, time or cost.

ADJACENT VERTICES: A vertex v_i is adjacent to another vertex v_j, if there is an edge from v_i to v_j.

COMPLETE GRAPH: A graph (digraph) G is said to be complete if each vertex v_i is adjacent to every other vertex v_j in G, i.e., there are edges from every vertex to all other vertices.

CYCLE: If there is a path containing one or more edges which starts from a vertex v_i and terminates into the same vertex then the path is known as a cycle.

ACYCLIC GRAPH: If a graph (digraph) does not have any cycle then it is called acyclic graph.

CONNECTED GRAPH: In a graph G, two vertices v_i and v_j are said to be connected if there is a path from v_i to v_j (or v_j to v_i). A graph is said to be connected if for every pair of distinct vertices v_i, v_j in G, there is a path.

DEGREE OF VERTEX: The number of edges connected with vertex v_i is called the degree of vertex v_i and is denoted by degree (v_i).

For a digraph there are two degrees: in degree and out degree. Number of edges incident into v_i is the indegree of v_i, and is denoted as indegree (v_i). Number of edges leaving from v_i is the outdegree of v_i and is denoted as outdegree (v_i).

REPRESENTATION OF GRAPHS

A graph can be represented in many ways. Some of these representations are:

1) Linked representation
2) Sequential (Adjacency Matrix) representation.

Let us consider the below graph structures, to illustrate the above representations.
LINKED REPRESENTATION: (ADJACENCY LIST REPRESENTATION)

This is space-saving way of graph representation. In this representation, two types of node structures are used.

<table>
<thead>
<tr>
<th>NODE-LABEL</th>
<th>ADJ-LIST</th>
<th>WEIGHT</th>
<th>NODE-LABEL</th>
<th>ADJ-LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node structure for non-weighted graph</td>
<td>Node structure for weighted graph</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now let us see this representation for the above four graphs:

Representation of graph G3

Representation of graph G4 (Weighted Graph)

Observe that in the linked list representation of graphs, the number of lists depends on the number of vertices in the graph. The header node in each list maintains a list of all adjacent vertices of a node.
SEQUSENTIAL REPRESENTATION: (ADJACENCY MATRIX REPRESENTATION)

Matrix representation is the most useful way of representing any graph. This representation uses a square matrix of order n x n, n being the number of vertices in the graph. A general representation is shown below.

Entries in the matrix are as follows:

\[ a_{ij} = \begin{cases} 1, & \text{if there is an edge from } v_i \text{ to } v_j \\ 0, & \text{otherwise} \end{cases} \]

This matrix is known as adjacency matrix because an entry stores the information whether two vertices are adjacent or not. The matrix is also termed as bit matrix or Boolean matrix as the entries are either 0 or 1.

Adjacency matrix is also useful to store multigraph as well as weighted graph. In case of multigraph, instead of entry 1, the entry will be number of edges between two vertices. In case of weighted graphs the entries are weights of the edges between the vertices instead of 1. Figure below shows the adjacency matrix representation of graphs G₁, G₂, G₃ and G₄.

\[
\begin{array}{cccc}
    & v_1 & v_2 & v_3 \\
    v_1 & 0 & 1 & 0 \\
v_2 & 1 & 0 & 0 \\
v_3 & 0 & 0 & 1 \\
\end{array}
\]

A is an adjacency matrix of a graph G and if \( A = A^T \) then G is a simple undirected graph.

- For a simple digraph \( A \neq A^T \).
- If the graph is simple then all the diagonal entries are 0.
- G is a digraph and A is its adjacency matrix, then the diagonal entries of \( A.A^T \) gives the outdegree of all vertices in G and the diagonal entries of \( A^T.A \) give the indegree of all vertices.
- A is an adjacency matrix of a digraph \( A^L=[a_{ij}^L] \) gives the number of paths of length L from \( v_i \) to \( v_j \), where \( A^L \) is the \( L^{th} \) power matrix of A.

PATH MATRIX: Let G be a simple graph with n vertices \( v_1, v_2, v_3, \ldots, v_n \). An n x n matrix \( P = [p_{ij}] \) whose entries are defined as:

\[ p_{ij} = \begin{cases} 1, & \text{if there exists a path from } v_i \text{ to } v_j \\ 0, & \text{otherwise} \end{cases} \]

then P is called path matrix or reachability matrix of graph G.
In a simple digraph the length of any elementary path is <=n-1, where n is the no. of vertices in the graph. Similarly the length of any elementary cycle does not exceed n.

- A vertex \(v_i\) contains a cycle if the \(i, i\) entry in the path matrix \(P\) is 1.
- A graph is strongly connected if for all \(v_i, v_j \in G\), both the \(i, j\) entry and \(j, i\) entry in the path matrix are 1.

**APPLICATIONS OF GRAPHS**

The major graph applications are:

1) Shortest path problem
2) Topological sorting problem
3) Connectivity of a graph
4) Spanning trees
5) Binary decision diagrams
6) Euler's path and Hamiltonian path

**MINIMUM SPANNING TREES**

Given a connected and undirected graph, a **spanning tree** of that graph is a sub graph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A **minimum spanning tree** (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. A minimum spanning tree has \((V - 1)\) edges where \(V\) is the number of vertices in the given graph and contains all vertices of graph. If graph ‘\(G\)’ is not connected then there is no spanning tree of ‘\(G\)’. Similarly there will not be a unique spanning tree of a graph ‘\(G\)’. For example consider the graph below:

![Graph](image1)

![Spanning tree 1](image2)

![Spanning tree 2](image3)

Spanning trees has the properties that for any pair of vertices there exists only one path between them. Intersection of any edge to spanning tree will form a unique cycle.

A spanning tree ‘\(T\)’ of graph ‘\(G\)’, where the sum of weights of all edges in ‘\(T\)’ is minimum is called the ‘Minimum Spanning Tree’ of ‘\(G\)’. Minimum spanning tree represents the cheapest way of connecting all the edges of a graph ‘\(G\)’. Note that the minimum spanning tree of a graph need not be necessarily unique.

There are two well known algorithms to compute the minimum spanning trees of a given weighted, undirected graph. They are:

i) Prim’s algorithm and ii) Kruskal’s algorithm.

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
PRIM’S ALGORITHM

One way to compute minimum spanning tree is to grow the tree in successive stages. In each stage, one node is picked as the root and we add an edge and thus an associated vertex to the tree.

At any point in the algorithm, we can see that we have a set of vertices that have been already included in the tree, the rest of the vertices have not. The algorithm, at each stage, finds a new vertex to add to the tree by choosing the edge \((u,v)\) such that the cost of \((u,v)\) is the smallest among all the edges where \(u\) is in tree and \(v\) is not.

**Algorithm PRIM(G)**

/* \(G=(V,E)\) is a weighted, connected, undirected graph and \(E^l\) is the set of edges which are to be extracted to obtain the minimum cost spanning tree*/

\[E^l=\phi; \] /* Initialize \(E^l\) to a null set */

Select a minimum cost edge \((u,v)\) from \(E\);

\[V^l=(u)\]

While \(V^l\neq V\) do

Let \((u,v)\) be the lowest cost edge such that \(u\) is in \(V^l\) and \(V\) is in \(V\setminus V^l\)

Add edge \((u,v)\) to set \(E^l\);

Add \(v\) to set \(V^l\);

End while

End PRIM

**Example:**

Below figure shows how this algorithm would build the minimum spanning tree for the above graph, starting from \(v_1\). Initially, \(v_1\) is in the tree as a root with no edges. Each step adds one edge and one vertex to the tree.
The running time is $O(|V|^2)$ without heaps, which is optimal for dense graphs, and $O(|E|\log|V|)$ using binary heaps, which is good for sparse graphs.

**KRUSKAL'S ALGORITHM**

Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

**Description**

- create a forest $T$ (a set of trees), where each vertex in the graph is a separate tree
- create a set $S$ containing all the edges in the graph
- while $S$ is nonempty and $T$ is not yet spanning
  - remove an edge with minimum weight from $S$
  - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree, otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

**Algorithm KRUSKAL(G)**

/* $G=(V,E)$ is a weighted, connected, undirected graph and $E^1$ is the set of edges which are to be extracted to obtain the minimum cost spanning tree*/

1. $T = \emptyset$
2. for each $v \in V$:
   3. MAKE-SET($v$)
4. for each $(u, v)$ ordered by weight($u$, $v$), increasing:
   5. if FIND-SET($u$) $\neq$ FIND-SET($v$):
      6. $T = T \cup \{(u, v)\}$
     7. UNION($u$, $v$)
8. return $T$
9. End

This greedy strategy is continually to select the edges in order of smallest weights and accept an edge if it does not cause a cycle. The action of the algorithm on the above graph in the preceding example is shown in figure below.

Kruskal's algorithm maintains a forest – a collection of trees. Initially, there are $|V|$ single-node trees. Adding an edge merges two trees into one. When the algorithm terminates, there is only one tree, and this is the minimum spanning tree. Below figure shows the order in which edges are added to the forest.
The algorithm terminates when enough edges are accepted. It turns out to be simple to decide whether edge \((u,v)\) should be accepted or rejected. The appropriate data structure is the Union/Find algorithm.

Each vertex is initially in its own set. If \(u\) and \(v\) are in the same set, the edge is rejected, because since they are already connected, adding \((u,v)\) would form a cycle. Otherwise, the edge is accepted, and a Union is performed on the two sets containing \(u\) and \(v\).

The worst-case running time of this algorithm is \(O(|E|\log|E|)\), which is dominated by the heap operations. Notice that since \(|E|=O(|V|^2)\), this running time is actually \(O(|E|\log|V|)\).

**Applications of Minimum Spanning Tree:** MST is fundamental problem with diverse applications.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths, LDPC codes for error correction, image registration with Renyi entropy, learning salient features for real-time face verification, reducing data storage in sequencing amino acids in a protein, model locality of particle interactions in turbulent fluid flows, autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.

**SHORTEST PATH PROBLEM**

This problem of a graph is about finding a path between two vertices in such a way that this path will satisfy some criteria of optimization. For example, for a unweighted graph, the number of edges will be minimum and for weighted graph, the sum of weights on all edges in the path will be minimum.

This problem has variety of solutions. A few important algorithms are:

1. Warshall’s algorithm (Transitive Closure of a Graph)
2. Floyd’s algorithm
3. Dijkstra’s algorithm

**WARSHALL’S ALGORITHM**

By this algorithm we can determine whether there is a path from one vertex \(v_i\) to another vertex \(v_j\) either directly or one or more intermediate vertices. In other words, we can test the reachability of all pairs of vertices in a graph.
The path matrix can be computed from the adjacent matrix A of a graph. This computation involves the computation of $A^2$, $A^3$, ..., $A^n$ and then $B_n$. This method is not efficient at all. Warshall’s algorithm can be used to compute the path matrix in a more elegant manner.

This algorithm treats the entries in the adjacency matrix as bit entries and performs AND (\(^\land\)) and OR (\(\lor\)) the Boolean operations on them. The heart of the algorithm is a trio of loops, which operates very much like loops in the algorithm for matrix multiplication.

**Algorithm WARSHALL(A)**

**INPUT:** A directed graph G whose adjacency matrix is ‘A’ and vertices are labeled 1, 2, 3, ………, N; N being the number of vertices in the graph.

**OUTPUT:** The path matrix P

**Data structure:** Matrix representation of graph with pointer as A

/* Initialization of path matrix P with the adjacency matrix A*/

1. For i=1 to N do
   a. For j=1 to N do
      i. For k=1 to N do
         P[i][j]=A[i][j]
      ii. End For
   b. End For
2. End For

/* Updation of path matrix P*/

3. For k=1 to N do
   a. For i=1 to N do
      1. For j=1 to N do
         P[i][j]=P[i][j] \(\lor\) (P[i][k] \(\land\) P[k][j])
      2. End For
   b. End For
4. End For
5. Return (P)
6. Stop

In step1 we initialize the path matrix P with the adjacency matrix A. This signifies that initially the path matrix maintains P[i][j] entries whether there is a direct path from $v_i$ to $v_j$. In step3, the outermost loop will execute N times. For each k, it decides whether there is a path from $v_i$ to $v_j$ (for all $i, j = 1, 2, ...., N$) either directly or via k, i.e., from $v_i$ to $v_k$ and then from $v_k$ to $v_j$. It therefore sets the P[i][j] entries to 0 or 1 accordingly.

*Figure: Illustration of WARSHALL’S algorithm*
From the graph it is observed that there is a path from \( v_1 \) to \( v_4 \) via \( v_2 \). During the execution of step3 in Warshall’s algorithm and when \( k=2 \), we see that:

\[
P[1][4]=P[1][4] \lor (P[1][2] \land P[2][4]) = 0 \lor (1 \land 1) = 1
\]

thus enumerating the path from \( v_1 \) to \( v_4 \) via \( v_2 \). The final path matrix is obtained as shown in figure (b). From this, it is seen that there is no path from the vertex \( v_4 \) to any other vertex. Also, we cannot reach to vertex \( v_1 \) from any other vertex in the graph. Other possible reachabilities can also be seen from this path matrix. Here, the vertex \( v_1 \) acts as a ‘Source Vertex’ (one can reach to any vertex from it) and vertex \( v_4 \) acts as a ‘Sink Vertex’ (i.e., if we reach at this vertex then it cannot be moved to any other vertex). There is no cycle in the graph as all the diagonal entries in the path matrix are zero.

**TRANSITIVE CLOSURE OF A GRAPH**

Given a graph, find out if a vertex \( j \) is reachable from another vertex \( i \) for all vertex pairs \((i, j)\) in the given graph. Here reachable mean that there is a path from vertex \( i \) to \( j \). The reach-ability matrix is called transitive closure of a graph.

**FLOYD ALGORITHM**

The Floyd Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

**Example:**

**Input:**

\[
\text{graph}[][] = \begin{bmatrix}
0 & 5 & \text{INF} & 10 \\
\text{INF} & 0 & 3 & \text{INF} \\
\text{INF} & \text{INF} & 0 & 1 \\
\text{INF} & \text{INF} & \text{INF} & 0
\end{bmatrix}
\]

which represents the following graph:

```
0 ----> (3)
|          /|
|          |
\(5 \)
|          |
\(\text{v1}\) ----> (2)
```

Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

**Output:**

Shortest distance matrix

\[
\begin{bmatrix}
0 & 5 & 8 & 9 \\
\text{INF} & 0 & 3 & 4 \\
\text{INF} & \text{INF} & 0 & 1 \\
\text{INF} & \text{INF} & \text{INF} & 0
\end{bmatrix}
\]

**Floyd Algorithm**

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to pick one by one all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number \( k \) as an intermediate vertex, we already have considered vertices \( \{0, 1, 2, \ldots k-1\} \) as intermediate vertices. For every pair \((i, j)\) of source and destination vertices respectively, there are two possible cases:

1. \( k \) is not an intermediate vertex in shortest path from \( i \) to \( j \). We keep the value of dist[\( i \)][\( j \)] as it is.
2. \( k \) is an intermediate vertex in shortest path from \( i \) to \( j \). We update the value of dist[\( i \)][\( j \)] as dist[\( i \)][\( k \)] + dist[\( k \)][\( j \)].

The following figure shows the above optimal substructure property in all-pairs shortest path problem.
**Algorithm FLOYD(W)**

**INPUT:** A directed graph G whose Weight matrix is W and vertices are labeled 1, 2, 3, \ldots, N; N being the number of vertices in the graph

**OUTPUT:** The length matrix Q

**Data structure:** Matrix representation of graph with pointer as W

/* Initialization of Length matrix Q with the Weight matrix W*/
1. For i=1 to N do
   1. For j=1 to N do
      1. If W[i][j]=0 then
         Q[i][j]= ∞
      2. Else
         Q[i][j]=W[i][j]
   2. End For
2. End For

/* Updation of Length matrix P*/
3. For k=1 to N do
   1. For i=1 to N do
      1. For j=1 to N do
         Q[i][j]= MIN(Q[i][j], Q[i][k]+Q[k][j])
      2. End For
   2. End For
4. End For
5. Return (P)
6. Stop

**DIJKSTRA'S ALGORITHM**

This algorithm is used to find the shortest path from a single source. In this problem, there is a distinct vertex called Source Vertex and it requires finding the shortest path from this source vertex to all the other vertices. Let us consider a simple graph as shown in figure. The different shortest paths, assuming v₁ as the source vertex are listed as below:

![Graph Image](image_url)

Assume that all the vertices in the graph are labeled as 1,2,\ldots, N and the graph is represented through an adjacency matrix. Dijkstra's algorithm requires three arrays as below:

- LENGTH[1\ldots N] = array of distances
- PATH[1\ldots N] = array of vertices
- SET[1\ldots N] = array of Boolean tags

The shortest distance from the Source vertex to any vertex is stored in LENGTH[i], while PATH[i] contains the nearest predecessor(s) of the vertex i in the path determining the shortest distance from the source vertex to the vertex i. In other words, the array PATH keeps a track of shortest path from any vertex to the source vertex. The Boolean array SET is used during the execution of the algorithm. SET[i] = 1 means the shortest distance and path from the source vertex to the vertex i is already enumerated.
Suppose the source vertex be S. The algorithm consists of two major parts: an initialization part followed by an iteration part. In initialization part, the above mentioned three arrays are initialized. In the iteration part, the vertices will be included one by one in the set of vertices for which the shorter distance from S is enumerated.

Algorithm DIJKSTRA(S)
Input: Adj, the pointer to the graph. S the source vertex. N the number of vertices.
Output: LENGTH, an array of distances from S to all other vertices. PATH, an array of string of vertices giving the track of all shortest paths.
Data Structure: Matrix representation of graph with pointer as Adj

Steps:
/*INITIALIZATION*/
1. For i =1 to N do
   SET[i] = 0 //Initialization of SET array
2. End For
3. For i =1 to N do
   (1). If Adj[S][i] =0 then //There is no direct path from S to vertex i
      i. LENGTH[i] =∞ //Empty path
   (2). Else
      i. LENGTH[i] =Adj[S][i]
      ii. PATH[i] =S  //Source vertex is immediate predecessor to vertex i
   (3). End If
3. End For
5. SET[S] =1
6. LENGTH[S] =0 // Source vertex is implicitly enumerated with length as 0
/*ITERATION*/
7. complete =FALSE //It is a flag for controlling the iteration
8. WHILE( not complete ) do
   (1). j =SEARCH_MIN( LENGTH, SET)  //Find a vertex j which has minimum distance among those vertices not enumerated for shortest path
   (2). SET[j] =1 // vertex is enumerated
   (3). For i =1 to N do // For each i not enumerated
      i. If SET[i] =1 then //If i is already enumerated then goto next vertex
         i =i+1
      ii. Else
         a. If Adj[i][j] ≠ 0 then
            i. If ((LENGTH[j]+Adj[i][j])< LENGTH[i]) then
               (a). LENGTH[i] =LENGTH[j]+Adj[i][j]
               (b). PATH[i]=j  //vertex j becomes immediate predecessor of i
            (ii). End If
         b. End If
      iii. End If
   (4). End For
   /* To test whether all the vertices are enumerated or not*/
(5). Complete =TRUE
(6). For i=1 to N do
   i. If SET[i] =0 then
      a. complete= FALSE
      b. Break // Break the loop
   ii. Else
      i= i+1
   iii. End If
(7). End For
9. End While
10. Return (LENGTH, PATH)
11. Stop

In this algorithm the procedure SEARCH_MIN (step 8.(1)) will return the label of the vertex which has minimum distance (by consulting the array LENGTH) and which is not yet enumerated in the shortest path (by consulting the array SET).

**Example 1**: Let us trace the Dijkstra’s algorithm for the graph shown below:

Following information can be concluded from LENGTH & PATH arrays:

<table>
<thead>
<tr>
<th>Path From (Source vertex)</th>
<th>To</th>
<th>Length</th>
<th>Shortest Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1-2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1-2-3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1-2-4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1-2-3-5</td>
</tr>
</tbody>
</table>
TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

🔗 Compiled Books: https://www.tutorialsduniya.com/compiled-books
🔗 Programs: https://www.tutorialsduniya.com/programs
🔗 Question Papers: https://www.tutorialsduniya.com/question-papers
🔗 Python Notes: https://www.tutorialsduniya.com/python
🔗 Java Notes: https://www.tutorialsduniya.com/java
🔗 JavaScript Notes: https://www.tutorialsduniya.com/javascript
🔗 JSP Notes: https://www.tutorialsduniya.com/jsp
🔗 Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
🔗 OR Notes: https://www.tutorialsduniya.com/operational-research
Example 2:

![Directed Graph G](image)

- **Table 1:** Initial configuration of table used in Dijkstra's algorithm.
- **Table 2:** After $v_2$ is declared known.
- **Table 3:** After $v_1$ is declared known.
- **Table 4:** After $v_5$ and then $v_3$ are declared known.
- **Table 5:** After $v_4$ is declared known.
- **Table 6:** After $v_7$ is declared known.

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
After $v_6$ is declared known and algorithm terminates

**Figure** Stages of Dijkstra’s algorithm
UNIT – 5

LOWER BOUND ON COMPLEXITY FOR SORTING METHODS

Result 1: The worst case complexity of any sorting algorithm that only uses key comparisons is: \( \Omega(n \log n) \)

Result 2: The average case complexity of any sorting algorithm that only uses key comparisons is: \( \Omega(n \log n) \).

The above results are proved using a decision tree which is a binary tree in which the nodes represents the status of the algorithm after making some comparisons.

DECISION TREE

- A decision tree is an abstraction used to prove lower boundary.
- In this context, a decision tree is a binary tree.
- Each node represents a set of possible orderings, consistent with comparisons that have been made, among the elements.
- The results of the comparisons are three edged.

Ex: Consider a node ‘x’ in a decision tree and let ‘y’ be it’s left child and ‘z’ it’s right child.

Basically ‘y’ represents a state consisting of the information known at ‘x’ plus the fact that the key ‘k1’ is less than key ‘k2’.

Ex: The following decision tree represents an algorithm that sorts the 3 elements a, b and c.

- The initial state of the algorithm is at the root. No comparisons have been done so all orderings are legal.
- The first comparison that this particular algorithm performs compares ‘a’ and ‘b’. The two results lead to two possible states. If a<b, then only 3 possibilities remain.
- If the algorithm reaches node 2, then it will compare ‘a’ and ‘c’. If a>c the algorithm enters state 5.
- Since there is only one ordering that is consistent, the algorithm can terminate and report that it has completed the sort.
Every algorithm that sorts by using any comparisons can be represented by a decision tree.

The number of comparisons used by the sorting algorithm is equal to the depth of the deepest leaf. Here, the algorithm uses three comparisons in the worst case.

Internal nodes represent the comparison made between input items.

External nodes or leaf nodes represent the results of the algorithm.

To prove the lower bounds, all that needs to be shown, some basic tree properties.

Lemma 1: Let ‘T’ be a binary tree of depth ‘d’, then ‘T’ has at most $2^d$ leaves.

Lemma 2: A binary tree with ‘L’ leaves must have depth at least $\lceil \log L \rceil$.

**Theorem:** Any sorting algorithm that uses only comparisons between elements requires at least $\lceil \log n! \rceil$ comparisons in worst case.

$$\log(n!) = \log(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2.1)$$

$$= \log(n) + \log(n-1) + \log(n-2) + \ldots + \log2 + \log1$$

$$\geq n/2 \log n/2$$

$$\approx \Omega(n \log n).$$
**Theorem:** Any sorting algorithm that uses only comparisons between elements requires $(\log n!)$ comparisons in average case.

- The average case complexity of any sorting algorithm is $\Omega(n \log n)$.
- We assume Tree ‘T’ with depth $< \log k$.
- The $k$ represents total number of leaves.

Suppose tree T is of the form as shown in the following figure:

![Tree Diagram](image)

The average depth

$$\text{depth} \geq \frac{k_1}{(k_1+k_2) \log k_1} + \frac{k_2}{(k_1+k_2) \log k_2} + 1$$

$$= \frac{k_1}{k} \log k_1 + \frac{k_2}{k} \log k_2 + \left[\frac{k_1}{k} + \frac{k_2}{k}\right]$$

$$= \frac{1}{k} (k_1 \log_2 k_1 + k_2 \log_2 k_2)$$

$$\text{depth} \geq k \log k.$$

This contradicts our assumption that depth $< \log k$.

- Hence average case efficiency for the tree with $n!$ leaves can be $\Omega(n \log n)$ average case complexity.
- Hence the lower bound on worst case complexity: $\Omega(n \log n)$
- Hence the lower bound on average case complexity: $\Omega(n \log n)$

**HEAP SORT**

Heap sort is implemented by two operations.

1) **Build Heap**  
2) **Delete Max**

a) **Build Heap:** This operation creates a heap from the input. So the value at every parent node is greater than that of values at its children node for max heap.

b) **Delete Max:** This operation is applied ‘N’ times for ‘N’ elements.

- Every time it deletes the maximum number which is at root, and places that at the last portion of array.
- So that after ‘N’ deletions we will get sorted sequence of ‘N’ elements.
  (i.e., after DeleteMax, the heap shrinks by 1. Thus the cell that was last in the heap can be used to store the element that was just deleted. As an example, suppose we have heap with 6 elements, the first delete max
produces A1. Now the heap has only 5 elements, so we can place A1 in position 6. The next deletemax produces A2, since the heap will now have only 4 elements, we can place A2 in position 5).

Using this strategy, after the last deletemax the array will contain the elements in increasing order.

After creating heap in linear time, we perform N-1 deletemax by swapping the last element in the heap with the first element, decrementing the heap size, and percolating down.

When the algorithm terminates the array contains the elements in ascending order.

```c
#define lchild (i) (2*(i)+1)
void percdown(int a[], int i, int n)
{
    int c, temp;
    for (temp = a[i]; lchild(i) < n; i = c)
    {
        c = lchild[i];
        if (c != n-1 && a[c+1] > a[c])
            c++;
        if (temp < a[c])
            a[i] = a[c];
        else
            break;
    }
    a[i] = temp;
}
void Heapsort(int a[], int n)
{
    for (i = n/2; i > 0; i--) //Build Heap
        percdown(a, i, n);
    for (i = n-1; i > 0; i--) //DeleteMax
    {
        swap(a[0], a[i]);
        percdown(a, 0, i);
    }
}
```

In the above algorithm, the build heap uses at most 2N comparisons. In the second phase, the i\textsuperscript{th} DeleteMax uses at most 2[log i] comparisons, for a total of at most 2NlogN-O(N) comparisons are used by heap sort.

Ex: Input 31, 41, 59, 26, 53, 58, 97. Arrange the elements in sorted order by using heap sort.

Buildheap creates the heap as follows:
Delete 97 and readjust the heap, place 97 in last position, i.e., a[6].


Delete 58 and readjust the heap, place 58 in a[4] position.

Delete 53 and readjust the heap, Place 53 in a[3] position.

Delete 41 and readjust the heap, place 41 in a[2] position.
Delete 31 and readjust the heap, place 31 in a[1] position.

Delete 26 and place in a[0]

The above list is in sorted order.

**QUICK SORT**

- Quick sort is sorting algorithms that uses the divide and conquer strategy. In this method division is dynamically carried.
- The three steps of Quick Sort are as follows.
  i. Divide: Rearrange the elements and split the array into the sub arrays and an element in between. Each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element.
  ii. Conquer: Recursively sort the two sub arrays.
  iii. Combine: Combine all the sorted elements in a group to form a list of sorted elements.

**Procedure:**

1. Assume the first element as pivot element.
   - Assume the index of next element to pivot as 'i'.
   - Assume the index of last element in the list as 'j'.

Now (a) compare pivot with a[i]
   1. if a[i]<pivot, move 'i' to next position by i++;
   
   If a[i] >pivot then stop the moving here, go to the end of the list.

Now (b) compare pivot with a[j]
   1. if a[j]>pivot, move 'j' to the previous position by j--.
   
   If a[j]<pivot then stop moving,
(c) Now exchange a[i] and a[j] and do i++, j--

(d) Repeat this process until i>j, when i>j then exchange a[j] and pivot elements.

**Ex:**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>65</td>
<td>70</td>
<td>75</td>
<td>80</td>
<td>85</td>
<td>60</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

Assume ‘65’ as pivot; i=2, j=9.

Here a[i]>pivot so stop moving. Go to next end

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>70</td>
<td>75</td>
<td>80</td>
<td>85</td>
<td>60</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

Here a[j]<pivot, so stop moving. Now swap a[i],a[j] and do i++, j--

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>45</td>
<td>75</td>
<td>80</td>
<td>85</td>
<td>60</td>
<td>55</td>
<td>50</td>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>

a[j]<pivot, exchange a[i], a[j], do i++, j--

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>45</td>
<td>50</td>
<td>80</td>
<td>85</td>
<td>60</td>
<td>55</td>
<td>75</td>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>

a[j]<pivot exchange a[i], a[j] do i++, j--

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>45</td>
<td>50</td>
<td>55</td>
<td>85</td>
<td>60</td>
<td>80</td>
<td>75</td>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>

a[j]<pivot exchange a[i],a[j] do i++, j--

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>45</td>
<td>50</td>
<td>55</td>
<td>60</td>
<td>85</td>
<td>80</td>
<td>75</td>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>

Now i>j, so exchange pivot element & a[j]

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>45</td>
<td>50</td>
<td>55</td>
<td>65</td>
<td>85</td>
<td>80</td>
<td>75</td>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>

At this instance ‘65’ is placed at 5<sup>th</sup> position and all elements left of 65 are less than 65 and all elements right of 65 are greater than 65. So 65 is placed in sorted order. Now consider the left part as 1 sub array and sort it and right part as another sub array and sort it.

1 2 3 4

60 45 50 55

Here p=1, i=2, j=4

60 45 50 55
\[a[2] < p \text{ so do } i++\]
\[60 \ 45 \ 50 \ 55\]
\[a[3] < p \text{ so do } i++\]
\[60 \ 45 \ 50 \ 55\]
\[a[4] < \text{pivot so do } i++\]

now \(i=5, j=4\) so \(i>j\).

now exchange \(a[j]\), pivot
\[55 \ 45 \ 50 \ 60\]

Now the list is
\[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9\]
\[55 \ 45 \ 50 \ 60 \ 65 \ 85 \ 80 \ 75 \ 70\]

At this instant 60, 65 both are in sorted order i.e., left side of them are less than them, right side of them are greater than them. So consider sub arrays and sort them recursively.

After sorting, the resultant list is
\[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9\]
\[45 \ 50 \ 55 \ 60 \ 65 \ 70 \ 75 \ 80 \ 85\]

**Algorithm:**

Algorithm Qsort (p, q)
{
    if (p<q) then
    {
        j=partition (a, p, q+1)
        Qsort (p, j-1);
        Qsort (j+1, q);
    }
}

Algorithm partition (a, m, p)
{
    v=a[m], i=m; j=p;
    repeat
    {
        repeat
        { i=i+1; until (a[i] ≥v);
        repeat
        { j=j-1; until (a[j] ≤v);
        if (i<j) then
        {


```c
k=a[i];
a [i] =a[j];
a [j]=k;
}
} until (i≥j);
a [m] =a[j];
a [j]=v;
return j;
}

Explanation:
Quick sort algorithm has two functions. They are

1) Qsort       2) partition

Qsort: This function uses 2 variables p & q. They are starting and ending indexes of list. This function again calls Qsort recursively to solve the left sub array and right sub array individually.

Partition: It uses the variable ‘v’ to store the pivot element, ‘i’ is the index of next element of pivot, and ‘j’ is the index of last element in list.

The repeat-until loops are executed if the condition is false. When the condition becomes true, then they stop the execution of loops.

So first repeat-until loop checks whether a[i]≥v, if it is true then it stops ‘i’ there, otherwise it moves ‘i’ to next position by i=i+1

Second, repeat-until loop checks whether a[j]≤v if it is true then it stops ‘j’ there, otherwise it moves ‘j’ to previous position by j=j-1

when (i<j) is true then a[i] & a[j] are exchanged.
when (i>j) is true then a[j] & v (i.e., pivot element) are exchanged.

TIME COMPLEXITY
The running time of Quick sort depends on whether partition is balanced or unbalanced.

A good partition splits an array up into two equal sized arrays. A bad partition, on the other hand splits an array up into two arrays of very different sizes. The worst partition puts only one element in one array and the remaining elements in another array. If the partitioning is balanced, the quick sort runs fast. On the other hand, if partitioning is unbalanced, the quick sort runs very slowly.

Best Case: The best case of quick sort will happen if each partitioning stage divides the array exactly in half. If the procedure ‘partition’ produces two arrays of size n/2, the recurrence relation is

\[ T(n) = T(n/2) + T(n/2) + \Theta(n) \]
\[ = 2T(n/2) + \Theta(n) \]
\[ T(n) = O(n \log n). \]

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
**Worst Case:** The worst case occurs if given array a[1:n] is already sorted. The partition function call always returns ‘j’. So successive calls to partition will split arrays of length n,n-1,n-2,...,2 and running time proportional to
\[ n+(n-1)+(n-2)+...+2=((n+2)(n-1)/2)=O(n^2). \]
The worst case can also occur if a[1:n] starts out in reverse order.

**Average Case:**
Let T(n) be an average case time complexity and ‘n’ be the total no. of elements.
\[ T(n)=(n+1)+1/n(\sum [T(k-1)+T(n-k)]) - 1 \]
In partition function, the first call takes \((n+1)\) comparisons.
If we multiply both sides of equation 1 by \(n\) we will get
\[ nT(n)=n[(n+1)+1/n \sum [T(k-1)+T(n-k)]] \]
\[ 1 \leq k \leq n \]
\[ nT(n)=n(n+1)+ \sum T(k-1)+T(n-k) \]
\[ 1 \leq k \leq n \]
i.e., \( nT(n)=n(n+1)+2[T(0)+T(1)+.........+T(n-1)] - 2 \)
Substituting \(n=n-1\) in above equation
\[ (n-1)T(n-1)=(n-1)n+2[T(0)+T(1)+.........+T(n-2)] - 3 \]
Note that \(T(0)=T(1)=0\)
Subtracting equation 3 from equation we will get,
\[ nT(n)-(n-1)T(n-1)=2n+2T(n-1) \]
this can also be written as
\[ T(n)/(n+1)=T[(n-1)/n]+[2/(n+1)] \]
W can repeatedly write for
\[ T(n)/(n+1)=T(n-2)/(n-1)+2/n+2/(n+1) \]
\[ =T(n-3)/(n-2)+2/(n-1)+2/n+2/(n+1) \]
\[ ... \]
i.e., \( T(n)/(n+1) = T(1)/2+2 \sum 1/k \quad = 2 \sum 1/k \quad [T(1)=0] \)
\[ 3 \leq k \leq n+1 \quad 3 \leq k \leq n+1 \]
RADIX SORT

- In a radix sort we decompose the sort key using some radix ‘r’.
- For decimal numbers the radix r is 10.
- This sort is based on the values of actual digits in the positional representations of the numbers of the numbers being sorted.

Eg.: The number 367 in decimal notation is written with a 3 in hundreds position, a 6 in the ten’s position, and a 7 in the one’s position.

- First, we will sort all the elements based on their 1’s place
- Next, we will sort all the elements based on their 10’s place
- Next, we will sort all the elements based on their 100’s place
- This we will continue up to all the digits of all the numbers are sorted.
- For this we will take 10 bins (0-9) for decimal numbers because the radix for the decimal numbers is 10.
- When the elements are sorted based on 1’s place, we consider the LSB digits. Suppose the digit in 1’s place is ‘6’ then we place the number in the 6\(^{th}\) bin, if LSB digit for a number is 5, then we place the number in 5\(^{th}\) bin, we apply the procedure for all the numbers. Then we collect all the numbers from 0\(^{th}\) bin to 9\(^{th}\) bin.
- Next we consider 10’s place digits for all numbers and apply the same procedure.
- Next we consider 100’s place digit for all numbers and apply same procedure.
- After all, finally the elements were placed in sorted order.

**Eg:** 179, 208, 306, 93, 859, 984, 55, 9, 271, 33

(1) Sort all the numbers based on 1’s place digit.

```
   0   1   2   3   4   5   6   7   8   9
   271 93   984 55  306 208 179 859
   33
```

- 179, 859, 9 were placed in 9\(^{th}\) bin because for all these numbers the 1’s place digit is 9.
- Now remove all elements from bin 0 to bin 9.
  271, 93, 33, 984, 55, 306, 208, 179, 859, 9.

(2) Sort all the above numbers based on 10’s place digit.

```
   306 208
   9
   33
   55 859
   271 179
   984
   9
```

- 179, 859, 9 were placed in 9\(^{th}\) bin because for all these numbers the 1’s place digit is 9.
Here the 10’s place digit of 306, 208, 9 is 0 so all these numbers are placed into bin 0.

Remove all elements from bin 0 to bin 9:
306, 208, 9, 33, 55, 859, 271, 179, 984, 93

(3) Sort all the numbers based on 100’s place digit

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>179</td>
<td>208</td>
<td>306</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>859</td>
</tr>
<tr>
<td>33</td>
<td>271</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>984</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9, 33, 55, 93, 179, 208, 271, 306, 859, 984.

**Algorithm for Radix Sort**

```c
int radix(int a[], int link[], int d, int r, int n)
{
    int front[r], rear[r];
    int i, bin, current, first, last;
    first = 1;
    for(i = 1; i < n; i++)
        link[i] = i + 1;
    link[n] = 0;
    for (i = d - 1; i >= 0; i--)
    {
        for(bin = 0; bin < r; bin++)
            front[bin] = 0;
        for(current = first; current != 0; current = link[current])
        {
            bin = digit(a[current], i, r);
            if(front[bin] == 0)
                front[bin] = current;
            else
                link[rear[bin]] = current;
            rear[bin] = current;
        }
        for(bin = 0; !front[bin]; bin++);
        first = front[bin]; last = rear[bin];
        for(bin++; bin < r; bin++)
            if (front[bin])
            {
                link[last] = front[bin];
                last = rear[bin];
            }
        link[last] = 0;
    }
}```
In the above algorithm, we will take the input list into an array a[], ‘n’ is the no. of elements of array a[], ‘r’ is a radix value. For decimal numbers it is 10.

digit(a[i],j,r) is a function which extracts \( j^{th} \) digit of a[i].

Initially we create chain of records starting at 1 by using first ‘for’ statement. So the result is (link[i]=i+1)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[10]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

179\rightarrow208\rightarrow306\rightarrow93\rightarrow859\rightarrow984\rightarrow55\rightarrow9\rightarrow271\rightarrow33

for(i=d-1;i>=0;i-- ) this is executed for i=2,i=1,i=0 if the maximum no. of digits in a number is 3.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>front</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>rear</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

By executing for(current=first; current; current=link[current]) all the numbers will be placed in appropriate bins based on theirs 1’s place digit.

digit( ) is a function which extracts the appropriate digit.

for i=2 it extracts, 1’s place digit in all numbers.
for i=1 it extracts, 10’s place digit in all numbers.
for i=0 it extracts, 100’s place digit in all numbers.

\[
\text{bin}=\text{digit(a[current],i,r)}
\]

In this for loop the values in front array and, rear array are changed.

for current=1, \[
\begin{align*}
\text{c=2} & \quad \text{bin}=\text{digit(a[1],2,10)}=9 \\
\text{c=3} & \quad \text{bin}=\text{digit(a[2],2,10)}=8 \\
\text{c=4} & \quad \text{bin}=\text{digit(a[3],2,10)}=6 \\
\text{c=5} & \quad \text{bin}=\text{digit(a[4],2,10)}=3 \\
\text{c=6} & \quad \text{bin}=\text{digit(a[5],2,10)}=9 \\
\text{c=7} & \quad \text{bin}=\text{digit(a[6],2,10)}=4 \\
\text{c=8} & \quad \text{bin}=\text{digit(a[7],2,10)}=5 \\
\end{align*}
\]
c=8 \quad \text{bin} = \text{digit}(a[8], 2, 10) = 9
\hline
\text{c=9} \quad \text{bin} = \text{digit}(a[9], 2, 10) = 1
\hline
\text{c=10} \quad \text{bin} = \text{digit}(a[10], 2, 10) = 3
\hline
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
\text{Link} & 5 & 3 & 4 & 10 & 8 & 7 & 8 & 9 & 10 & 0 \\
\end{array}
\hline
\text{for(bin=0;!front[bin];bin++) \; \text{this finds the first nonempty location in \text{front} array.}}
\text{link[rear[4]]=6, link[1]=5}
\text{link[rear[9]]=8, link[5]=8}
\text{link[rear[3]]=10, link[4]=10}
\hline
\text{first=\text{front}[1]=9}
\hline
\text{last=\text{rear}[1]=9}
\hline
\text{for(bin++;b<10;bin++)}
\begin{array}{cccccccccccc}
\text{if(front[2])} & \text{if(front[3])} & \text{if(front[4])} & \text{if(front[5])} & \text{if(front[6])} \\
& \text{last=10} & \text{last=6} & \text{last=7} & \text{last=3} \\
\hline
\text{if(front[0])} & \text{if(front[8])} & \text{if(front[9])} \\
& \text{link[3]=2} & \text{link[2]=1} \\
& \text{last=2} & \text{last=8} \\
\end{array}
\hline
\text{After the iteration link[last]=link[8]=0}
\hline
\text{After the first iteration the link array is changed as follows:}
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
\text{Link} & 5 & 1 & 2 & 10 & 8 & 7 & 3 & 0 & 4 & 6 \\
\end{array}
\hline
\text{first=9. \; 9->4->10->6->7->3->2->1->5->8->0}
\hline
\text{271\rightarrow93\rightarrow33\rightarrow984\rightarrow55\rightarrow306\rightarrow208\rightarrow179\rightarrow859\rightarrow9}
\hline
\text{Now the above sequence is executed for i=1. All numbers in 10's place are sorted.}
\text{This is repeated for i=0. Finally the sorted sequence is obtained.}
MERGE SORT

Merge sort is a sorting algorithm that sorts the given list using divide-and-conquer strategy. Merge sort includes three steps.

(i) The sequence of 'n' elements a[1]...a[n] is divided into two parts a[1]...a[n/2] and a[n/2+1]...a[n].
(ii) Each set is individually sorted.
(iii) The resulting sorted sequences are merged to produce a single of 'n' elements.

**EX:** a[1:10]=(310,285,179,625,351,423,861,254,450,520)
Dividing the list $a[1:10]$ into individual lists, 10 lists are obtained. Each list contains one element. Each list is in sorted order. Now consider list1, list2 and merge them.

**Algorithm for Merge Sort**

```plaintext
Algorithm Msort (low, high)
{
    if(low<high) then
    {
        mid:=[(low+high)/2];
        Msort(low, mid);
        Msort(mid+1, high);
        Merge(low, mid, high);
    }
}
```

Algorithm Merge(low, mid, high)
{
    h:=low;
    i:=low;
    j:=mid+1;
    while((h<=mid) and (j<=high)) do
    {
        if(a[h]<=a[j]) then
        {
            b[i]:=a[h];
            h:=h+1;
        }
        else
        {
            b[i]:=a[j];
        }
    }
}
j:=j+1;
}
i:=i+1;
}
if(h>mid) then
for k:=j to high do
{
    b[i]:=a[k];
i:=i+1;
}
else
for k:=h to mid do
{
    b[i]:=a[k];
i:=i+1;
}
for k:=low to high do
    a[k]:=b[k];

(A)

\. In above algorithm Msort, calls two functions:
   (i) Msort( ), recursively
   (ii) Merge( )

\. Msort algorithm accepts 2 inputs variables ‘low’, ‘high’.
   ‘low’ is the starting index of an array i.e., In a[1:10], low is ‘1’
   ‘high’ is the ending index of an array i.e., In a[1:10], high is ‘10’

\. ‘Msort’ finds the middle value as mid=(low+high)/2.
   Next it recursively calls Msort function to split the array into 2 parts. This
   dividing of an array continues until each list contains only one element in it.

(B) Merging

\. ‘Merge’ function is called by ‘Msort’ and it accepts 3 values as input. They
   are ‘low’, ‘high’, ‘mid’.
\n\. It considers two sorted lists and merge them, so that the merged list is again
   in sorted order.

Ex:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>179</td>
<td>285</td>
<td>310</td>
<td>351</td>
<td>652</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>254</td>
<td>423</td>
<td>450</td>
<td>520</td>
<td>861</td>
</tr>
</tbody>
</table>

\. It compares a[h] with a[j]. If a[h] is less than a[j], then, the element a[h] will
   be sent to a new array b[i] and ‘h’ is moved to next location.
\. It a[j] is less than a[h] then the element a[j] will be sent to a new array b[i]
   and ‘j’ is moved to next location.
(a) In the above example $a[h]<a[j]$. So send $a[h]$ into $b[i]$ and move ‘h’ to next position.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>179</td>
<td>254</td>
<td>310</td>
<td>351</td>
<td>652</td>
<td>861</td>
</tr>
</tbody>
</table>

(b) Now $a[h]$ is 285, $a[j]$ is 254. $a[h]<a[j]$ false. So send $a[j]$ into ‘b’ move ‘j’ to next location

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>179</td>
<td>285</td>
<td>310</td>
<td>351</td>
<td>652</td>
<td>861</td>
</tr>
</tbody>
</table>

(c) $a[h]<a[j]$ So sent $a[h]$ into ‘b[3]’
- Change ‘h’ position to next location

(d) $a[h]<a[j]$ So sent it into ‘b[4]’
- Change ‘h’ position
Here a[h]<a[j] so sent to b

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
All the elements of left array[1:5] are copied to the array ‘b’. So the remaining elements of right array can be copied to array ‘b’ without comparing with any elements, i.e., ‘861’ can be copied to array ‘b’.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>179</td>
<td>254</td>
<td>285</td>
<td>310</td>
<td>351</td>
<td>423</td>
<td>450</td>
<td>520</td>
<td>652</td>
<td>861</td>
</tr>
</tbody>
</table>

**Time Complexity of Merge Sort:**

- If the time for the merging operation is proportional to ‘n’, then the computing time for merge sort is described by the recurrence relation.

\[
T(n) = \begin{cases} 
  a & \text{n=1, ‘a’ a constant} \\
  2T(n/2)+cn & \text{n>1, ‘c’ a constant}
\end{cases}
\]

When ‘n’ is a power of 2, \(n=2^k\), we can solve this equation by successive substitutions.

\[
T(n) = 2T(n/2)+cn \\
= 2[2T(n/4)+cn/2]+cn \\
= 4T(n/4)+cn+cn \\
= 4T(n/4)+2cn \\
= 4[2T(n/8)+cn/4]+2cn \\
= 8T(n/8)+3cn \\
= 2^3 T(n/2^3)+3cn
\]

\[
\vdots \\
= 2^kT(n/2k)+kcn \\
= 2^kT(n/n)+kcn \\
= 2^kT(1)+kcn \\
= .n.a+cn \log_2^n
\]

\[
T(n) = O(n \log n).
\]

- The below figure is a tree that represents the sequence of recursive calls, that are produced by ‘Msort’ function; when it is applied to ten elements.
- The pair of values in each node are the values of the parameters ‘low’ and ‘high’.
- It shows that how splitting continues until list containing a single element.
The below figure is a tree that represents the calls to the merge function. Ex: 1,1, and 2 represents the merging of a[1:1] with 2.
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
UNIT – VI

- A file is a repository of data that is stored in a performed storage media, mainly in secondary memory.
- It is difficult to get large amount of data as input to a program through keyboard. The solution to this is to maintain all the data into a file and let the 'C' program read the input from the file.
- Files are not only used for storing personal data, programs are also stored in files.
- The following operations are applied on the file:
  a) Naming a file  
  b) Opening a file  
  c) Reading data from a file  
  d) Write data to a file  
  e) Closing a file

Streams

- A stream is a sequence of data types which is used to read and write data.
- The streams that represent the input data of a program are known as input streams.
- Streams that represent the output data of a program are known as output streams.
- Input streams interpret data from different devices such as keyboard and mouse.
- Output streams write the data to the devices such as printer and monitor.

A stream is linked to a file using an open operation.
A stream is disassociated from a file using close operation.
There are two types of streams: text and binary.

**Low Level Functions for Reading and Writing Files**

- All operating systems provide service points through which programs requires services from the kernel.
- So the statements which needs the help of a kernel to execute it is known as known as system call.
- So many file system calls are there. Some of them are:
  1. open() 2. read() 3. write() 4. fseek() 5. close()
- These low level functions are used to open the files, read data from files, write data to files, change pointer position in a file etc.

**open()**

The open system call is used to open a file. When a process (or) a program wants to read some data from one file or write some data to the file, that file should be opened.

**Example:** If the program ‘x’ wants to read the data from ‘y’ file, before reading the file ‘y’ should be opened.

**General Syntax**

```c
#include<sys/types.h>
#include<fcntl.h>
int open(const char *pathname, int oflag, mode_t mode)
```

In the above syntax, the arguments are described as:

1. **pathname**: The pathname is the name of the file (or) path of the file which you want to open.
2. **oflag**: The second argument oflag is an integer value, that specifies for which purpose we are opening the file. Here the oflag values are 2 types.
   - a) Mandatory flags
   - b) Optional flags

**Mandatory Flags:**

1) O_RDONLY: opens the file for read only
2) O_WRONLY: opens the file for write only
3) O_RDWR: opens the file for read and write

We can also represent O_RDONLY as 0, O_WRONLY as 1, O_RDWR as 2

**Optional Flags:** These flags are specified by bitwise-OR with one of the above flags.

1. O_APPEND - appends data to the end of the file
2. O_CREAT - creates the file if it does not exist
3. O_EXCL - used with O_CREAT flag only, this flag generates error when the file already exists.
4. O_TRUNC - If the file exists, discards the file content and sets the file size to 0 bytes.

**mode**: This argument in open system call is supplied, when the second argument include O_CREAT flag, otherwise this mode value is not passed.
Mode is a permission mode for users.
This mode can be represented in two ways. They are:

a) 3-digit octal integer. Ex: 0644, 0711, 0777 etc.

b) By the symbolic constants which were defined in <sys/stat.h> header file.

<table>
<thead>
<tr>
<th>mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IRUSR</td>
<td>user has read permission</td>
</tr>
<tr>
<td>S_IWUSR</td>
<td>user has write permission</td>
</tr>
<tr>
<td>S_IXUSR</td>
<td>user has execute permission</td>
</tr>
<tr>
<td>S_IRGRP</td>
<td>group have read permission</td>
</tr>
<tr>
<td>S_IWGRP</td>
<td>group have write permission</td>
</tr>
<tr>
<td>S_IXGRP</td>
<td>group have execute permission</td>
</tr>
<tr>
<td>S_IROTH</td>
<td>others have read permission</td>
</tr>
<tr>
<td>S_IWOTH</td>
<td>others have write permission</td>
</tr>
<tr>
<td>S_IXOTH</td>
<td>others have execute permission</td>
</tr>
<tr>
<td>S_IRWXU</td>
<td>all permissions to the owner (read, write, execute)</td>
</tr>
<tr>
<td>S_IRWXG</td>
<td>all permissions to the group</td>
</tr>
<tr>
<td>S_IRWXO</td>
<td>all permissions to the others</td>
</tr>
</tbody>
</table>

- When an open system call is executed successfully, kernel returns a positive integer value, which is known as the file descriptor.
- When the open call is failed it returns a negative integer -1
- Always the file descriptor value is more than 2 (i.e., 3...63) because (0 1 2) these three no's are reserved fd's.
  0 is the fd of standard i/p stream (keyboard)
  1 is the fd of standard o/p stream (monitor)
  2 is the fd of standard error stream (monitor)
- A single file can open 63 files at a time.

Ex: 1) fd=open("x",O_RDONLY);
   2) fd1=open("y",O_WRONLY);
   3) fd2=open("z",O_CREAT|O_WRONLY|O_EXCL,0644);
   4) fd3=open("f2",O_CREAT|O_RDONLY|O_EXCL,0722);

This system call fails and fd3=-1 is stored, why because, when a file is created newly they cannot read any data from that because it is empty. So when the flag is O_CREAT that should be opened for writing purpose.

5) fd4=open("a",O_RDONLY,0422);
Open returns error because when the second argument is O_CREAT then only the 3rd argument should be passed.

6) fd5=open("b",O_CREAT|O_EXCL,0722);
returns error, because we should use one mandatory flag compulsorily as 2nd argument. (i.e., O_RDONLY,O_WRONLY,O_RDWR);

7) fd6=open("c",O_CREAT|O_WRONLY|O_EXCL);
Returns error because when the 2nd second argument includes O_CREAT, 3rd argument i.e., mode value should be passed.
8) fd7 = open("d", O_WRONLY|O_EXCL);

Returns error because O_EXCL should be used with O_CREAT only.

9) fd8 = open("e", 1); Correct

b) **read()**: The read system call is used to read data from a file using a file descriptor

**General syntax**

```c
#include<sys/types.h>
#include<unistd.h>
Size_t read(int fd, void *buf, int nbytes);
```

**The arguments to read system call**

a) **fd**: fd is a file descriptor value which is returned by the open system call, when you opened the file. This specifies from which file the data has to be read.

b) **buf**: buf is a character array, or the address of buf. This buf is necessary to hold the data that is to be read.

c) **nbytes**: This is an integer value. It specifies how many number of bytes user want to read from the file.

**Output**: If the read system call is executed successfully, it returns the no. of bytes of data successfully read and stored in the buf argument otherwise it returns -1.

Ex: 
```c
#include<sys/types.h>
#include<unistd.h>
main()
{
    int fd;
    char a[20], b[1024];
    fd=open("x", O_RDONLY);
    read(fd, a, 20);
    read(fd, b, 1024);
    read(fd, a, 20);
}
```

- the above example opens a file ‘x’ with open call and returns a ‘fd’ that will be passed in read call.
- The first read call reads 20 bytes of data from x and stores into an array ‘a’.
- Second read call reads ‘1024’ bytes of data from 21st position and stores the data into the variable b.
- Third read calls reads 20 bytes of data from 1045th position and stores the data into the variable a.

c) **write()**: The write system call is used to write the data to a file using a file descriptor.

**General syntax:**
The arguments of write system call are described as below:

a) fd: fd is an integer file descriptor value which is returned by open system call when you opened a file. This specifies to which file the data has to be written.
b) buf: buf is a character array which contains data to be written to the file.
c) nbytes: nbytes specifies how many bytes of data are in the buf argument.

**Output**: If the write system call is executed successfully it returns the number of bytes of data successfully written to a file. If write fails it returns a -1 value.

Ex:
```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>

main()
{
    int f1,f2,n;
    char a[20],b[20];
    f1=open("x",O_RDONLY);
    f2=open("y",O_CREAT|O_WRONLY|O_EXCL,0644);
    n=read(f1,a,sizeof a);
    write(f2,a,n);
}
```

Write a c program to implement cp command
```
#include<unistd.h>
#include<stdio.h>
main(int argc,char *argv[])
{
    int fd1,fd2,a;
    char b[120];
    if(argc!=3)
    {
        printf("error in giving input");
        exit(0);
    }
    if((fd1=open(argv[1],O_RDONLY))<0)
    {
        printf("open error");
        exit(0);
    }
    if((fd2=open(argv[2],O_WRONLY|O_CREAT|O_EXCL,0766))<0)
    {
        printf("create error");
        exit(0);
    }
```

while((n=read(fd1,b,sizeof b))
write(fd2,b,n);
}

In the above program, the first
- open call opens file1
- Second open call creates and opens file2
- read call reads the data from file1 to b
- write call writes the data to file2 from b

High Level Functions for Reading & Writing into Files

Opening a File:
- To process a file i.e., before reading and writing the file should be opened.
- A file can be opened by the library function fopen()

General syntax:
```c
FILE *fp
fp=fopen(const char *pathname,char *mode)
```

The fopen() function accepts two arguments:
1) The name or pathname of a file that is to be opened.
2) The second argument is mode. The mode specifies the purpose of opening a file, that is whether the file is opening for reading purpose or writing purpose etc.

Several mode values:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>open a text file for reading</td>
</tr>
<tr>
<td>w</td>
<td>creates and opens a file for writing. If the file is already existed then “w” overwrites the existing file</td>
</tr>
<tr>
<td>a</td>
<td>append to a text file. If the file does not exist it is created</td>
</tr>
<tr>
<td>r+</td>
<td>if the file already exists it opens for both reading and writing otherwise it returns an error.</td>
</tr>
<tr>
<td>w+</td>
<td>create a file and opens it both for writing and reading. If the file already exists, its contents are erased</td>
</tr>
<tr>
<td>a+</td>
<td>opens the file for reading and appending if the file already exists otherwise it creates a file</td>
</tr>
<tr>
<td>rb</td>
<td>it allows you to open a file as a binary file for reading data from it</td>
</tr>
<tr>
<td>wb</td>
<td>It allows to open a file as a binary file for writing data to it</td>
</tr>
<tr>
<td>ab</td>
<td>it allows to open a file as binary file for appending the data at the end of the file</td>
</tr>
<tr>
<td>rb+</td>
<td>it allows to open a file as a binary file for reading as well as writing data to a file</td>
</tr>
<tr>
<td>wb+</td>
<td>it allows to open a file as a binary file for writing as well as reading data from a file</td>
</tr>
<tr>
<td>ab+</td>
<td>it allows to open a file as a binary file for both reading and writing data to a file</td>
</tr>
</tbody>
</table>
Ex:-1 \( fp = \text{fopen("x.txt","r")}; \)

In the above example, the ‘fopen’ library function opens a file x.txt for reading purpose and returns a file pointer ‘fp’.

Ex:-2 \( fp1 = \text{fopen("y.c","w")}; \)

In the above example, fopen() function creates and opens a file “y.c” for writing purpose and returns a file pointer fp1.

- When fopen is executed successfully it returns file pointer ‘fp’ of the data type FILE.
- On error fopen returns a NULL value.
- ‘fp’ is a pointer which contains address of structure FILE (FILE has been defined in header file stdio.h)

When a file is opened the following steps will be performed by fopen()

  a) First, fopen() searches on the disk for the file to be opened
  b) if the file is present, it loads the file from the disk into memory.
     - If the file is not available, it returns NULL value
     - NULL is a macro defined in stdio.h which indicates that you failed to open the file
  c) It sets up a character pointer (which is a part of FILE Structure) which points to the
     first character of the chunk of memory where the file has been loaded.

In c, the fopen().after loading a file from the disk into memory it establishes a connection between the program in main memory and program in disk.

It then links the external file name with the pointer variable declared in the program

**NOTE**: After opening a file to read data from that file and write data to that file, fp(file pointer) is used.

Example:  
```c
#include<stdio.h>
void main()
{
    FILE *fp1;
    fp1=fopen("x.txt","r");
    if(fp1!=NULL)
```

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
{ 
    printf("File x.txt is opened successfully");
    exit(0);
} 
else 
{ 
    printf("File x.txt is not opened");
    exit(0);
} 
fclose(fp1);

**Output:** FILE x.txt is opened successfully.

When the above program is executed, if file x.txt is existed then ‘fp1’ is not a null value otherwise fp1 is a null value.

**Reading & Writing Data from Files**

To read data from file and write data to file we have 2 types of I/O statements. They are:

i) formatted I/O
   ii) unformatted I/O

i) The formatted I/O is again categorized as printf, scanf: which writes data to monitor and reads data from keyboard

   sprintf, sscanf: which reads data from string and writes data to string

   fprintf, fscanf: which reads data from file and writes data to file

ii) Unformatted I/O is again categorized as

1. character at a time I/O: getchar(), getc(), fgetc() 
2. Line at a time: gets(), fgets(), puts(), fputs() 
3. Reading more objects at a time: fread, fwrite

1. **Character at a time I/O:**
   
a) **Input functions:**

The following functions read the data character by character. They are:

i) getc()

ii) fgetc()

iii) getc():

   syntax: #include<stdio.h>

   ```c
   int getc(FILE *fp)
   ```

   a) This function reads a single character from the file ‘fp’.

   b) On success it returns the next character as a unsigned char converted to an int, from the specified i/p stream and increments the file position

   c) If the end of the file is reached, getc() returns EOF
Write a program that reads and display the contents of a text file.

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

void main(int argc, char *argv[])
{
    FILE *fp;
    char ch;
    if((fp=fopen(argv[1],"r"))==NULL)
    {
        printf("cannot open file \n");
        exit(1);
    }
    while((ch=getc(fp))!=EOF)
        printf("%c",ch);
    fclose(fp);
}
```

ii) `fgetc()`:

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

fgetc(FILE *fp)
```

The `fgetc()` function also returns the next character from the specified input stream and increments the file position indicator.

The character is read as an unsigned char that is converted to an integer.

If the end of file is reached, `fgetc()` returns EOF

**Example**: Program for reading data and displays the contents of text file

```c
#include<stdio.h>
void main(int argc, char *argv[])
{
    FILE *fp;
    char ch;
    if((fp=fopen(argv[1],"r"))==NULL)
    {
        printf("cannot open file \n");
        exit(1);
    }
    while(ch=fgetc(fp))!=EOF)
        printf("%c",ch);
    fclose(fp);
}
```

b) **Output Function**: The functions which print the data character at a time. They are...
i) putc()  ii) fputc()

The syntax for the two functions are

```c
#include<stdio.h>
int putc(int ch,FILE *fp);
int fputc(int ch,FILE *fp);
```

All the two returns c if ok, EOF on error.

i) fputc() writes the character 'c' to the specified stream fp at the current file position. Even though 'c' is declared to be an 'int' it is converted by fputc() into an unsigned char.

ii) putc() function is also similar to the fputc() function

iii) But the main difference is putc() can be implemented as macro, whereas fputc() cannot be implemented as macro.

**Example:** Program for reading data from one file and write the data to another file.

```c
#include<stdio.h>
void main(int argc,char *argv[])
{
    FILE *fp1,*fp2;
    char ch;
    if((fp1=fopen(argv[1],"r"))==NULL)
    {
        printf("FILE 1 cannot open");
        exit(1);
    }
    if((fp2=fopen(argv[1],"w"))==NULL)
    {
        printf("n FILE 2 cannot open");
        exit(1);
    }
    while((ch=fgetc(fp1))!=EOF)
    {
        fputc(ch,fp2);
        fclose(fp1);
        fclose(fp2);
    }
}
```

2. **Line at a time I/O:**

a) **Input function:** The fgets() function reads the data as one line at a time

**fgets():**

Syntax:
The `fgets()` function reads the characters from the specified stream `fp` and stores them into character array `buf`.
- `fgets()` function reads up to `n-1` characters from file `fp` and stores them into `buf`.
- After the specified characters have been read a null is stored in the array immediately after the last character read.
- A new line character will be retained and will be part of the array pointed to by `buf`.
- If `fgets()` is success it returns `buf`, otherwise, NULL pointer.

**Ex:** Write a C program to read the data from a file and print it on the monitor.

```c
#include<stdio.h>

int main(int argc, char *argv[]) {
    FILE *fp;
    char str[128];
    if((fp=fopen(argv[1],"r"))==NULL) {
        printf("n can’t open file");
        exit(1);
    }
    while(!feof(fp)) {
        if((fgets(str,126,fp)!=NULL))
            printf("%s",str);
    }
    fclose(fp);
    return 0;
}
```

In this program the `fgets` reads the data from a file `fp` and it prints the data on monitor.

b) **Output Function**: The `fputs()` function prints one line at a time.

`fputs()`:

**Syntax:**
```
#include<stdio.h>

int fputs(const char*str, FILE *fp)
```

- The `fputs()` function writes the contents of the string pointed to by `str` to the specified stream.
The null terminator of ‘str’ is not written to ‘fp’.

The fputs() function returns non-negative value, if it is successful, otherwise EOF an error.

Ex: Write a ‘c’ program to read the data from one file and write it to another file.

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

void main(int argc,char *argv[])
{
    FILE *fp1,*fp2;
    char str[128];
    if((fp1=fopen(argv[1],"r"))==NULL)
    {
        printf("can’t open file1");
        exit(1);
    }
    if((fp2=fopen(argv[2],"r"))==NULL)
    {
        printf("can’t open file2");
        exit(1);
    }
    while(!feof(fp1))
    {
        fgets(str,126,fp1);
        fputs(str,fp2);
    }
}
```

3) Direct I/O: If we want to read or write an entire structure at a time, or an array of elements at a time, then it is not possible with fgets, fputs

Therefore the following two functions are provided to read and write structures of data. They are:

i) fread() ii) fwrite()

i) fread():

```c
#include<stdio.h>
size_t fread(void *ptr,size_t size,size_t nobj,FILE *fp);
```

- Here the first argument is the address of an object, which object you want to read (i.e., either array name or structure variable).
- The second argument specifies the size of an object.
- The third argument specifies the no. of objects you want.
- The fourth argument specifies the stream from which you want to read.
If fread executes successfully, it returns the no. of objects it has read

Ex: In order to read a binary array, we can write the fread function as

```c
float x[10];
fread(&x[4], sizeof(float), 3, fp);
```

- Here in the fread function, first argument is the address of an array x from the fourth element.
- The second argument is the size of float value.
- Third argument is from x[4] position we want to read 3 values i.e., x[4], x[5], x[6]
- The fourth argument is fp, (a file pointer of a file from which we want to read)

ii) fwrite(): This function is used to write the structure of data to a function

Syntax:
```c
#include<stdio.h>
size_t fwrite(void *ptr, size_t size, size_t noobj, FILE *fp);
```

- The first argument is the address of an object, which object you want to write (i.e., either array name or structure variable)
- The second argument specifies the size of an object
- The third argument specifies the no. of objects you want.
- The fourth argument specifies the stream from which you want to write.
- If fwrite executes successfully, it returns the no. of objects it has written.

Ex: In order to write an array, we can write the fread function as

```c
float x[10];
twrite(&x[4], sizeof(float), 3, fp);
```

- Here, in the fwrite function, first argument is the address of an array x, from the fourth element.
- The second argument is the size of float.
- The third argument specifies, from x[4] position fwrite writes 3 values i.e., x[4], x[5], x[6].
- The fourth argument is fp (a file pointer to which we want to write).

Write a c program which illustrates fread and fwrite

```c
#include<stdio.h>
void main()
{
    char ch[10];
    struct student
    {
        char name[20];
        int m, p, c;
    }s;
    FILE *fp;
    fp=fopen("student.dat", "w");
    while(1)
```
The above program reads the data of student i.e., student name, marks in 3 subjects from keyboard and writes, that data to the file ‘student.dat’ and then it again reopens the file for reading purpose, and now reads the data from ‘student.dat’, then writes the data to the monitor.

Formatted I/O:

- In order to print the data according to required format, we have formatted I/O.
  - formatted input functions are:
    - printf(), fprintf(), sprintf()
  - formatted output functions are:
    - scanf(), fscanf(), sscanf()

fscanf(): `fscanf` function is used to read the data from the file.

```
#include<stdio.h>

fscanf(FILE *fp,const char *format[,add1,add2,add3...]);
```

Ex: fscanf(fp,"%d%s%d",&a,s,&b);

In this example, from file pointer ‘fp’ the first word is read into a variable ‘a’, and the second word is read into a variable ‘s’, and the third word is read into the variable ‘b’.

fprintf(): used to print the data

```
#include<stdio.h>

int fprintf(FILE *fp,const char *format[,arg1,arg2,......]);
```
The `fprintf()` function prints the arguments to the stream `fp`

Ex:
```
int a=5,b=7;
fprintf(fp,"%d%d",a,b);
```
The value of a and b are printed on a file 'fp'.

Ex: Write a c program to print the first 3 lines of file1 into file2 by using `fscanf` & `fprintf`.
```
#include<stdio.h>
#include<stdlib.h>

int main(int argc,char *argv[])
{
    FILE *fp1,*fp2;
    int n,i;
    char s1[10],s2[10],s3[10];
    fp1=fopen(argv[1],"r");
    fp2=fopen(argv[2],"w");
    for(i=1;i<=3;i++)
    {
        fscanf(fp1,"\n %s%d%s",s1,&n,s2);
        fprintf(fp2,"\n%s%d%s",s1,n,s2);
    }
    fclose(fp1);
    fclose(fp2);
}
```

`fclose()`: An opened file, must be closed as soon as all operations on it have been completed (i.e., read & write).

`fclose()` flushes all memory area occupied by that opened file.

Syntax: `fclose(filepointer)`: The filepointer value is obtained from `fopen()` function.

Ex:- main()
```
{
    FILE *fp1,*fp2;
    fp1=fopen("x.txt","r");
    fp2=fopen("y.txt","w");
    ....
    ....
    fclose(fp1);
    fclose(fp2);
}
```

This program opens two files “x.txt” in read mode and “y.txt” in write mode. After sometime, it has closed that pointers with `fclose()` statements.
SPECIAL CHARACTERS IN FILES

As you create the file structures described in this text, you may encounter some difficulty with extra, unexpected characters that turn up in your files with characters that disappear and with numeric counts that are inserted into your files. Here are some examples of the kinds of things you might encounter:

- On many computers you may find that a Control-Z (ASCII value of 26) is appended at the end of your files. Some applications use this to indicate end-of-file even if you have not placed it there. This is most likely to happen on MS-DOS systems.

- Some systems adopt a convention of indicating end-of-line in a text file as a pair of characters consisting of a carriage return (CR: ASCII value of 13) and a line feed (LF: ASCII value of 10). Sometimes I/O procedures written for such systems automatically expand single CR characters or LF characters into CR-LF pairs. This unrequested addition of characters can cause a great deal of difficulty. Again, you are most likely to encounter this phenomenon on MS-DOS systems. Using flag “b” in a C file or mode ios::bin in a C++ stream will suppress these changes.

- Users of larger systems, such as VMS, may find that they have just the opposite problem. Certain file formats under VMS remove carriage return characters from your file without asking you, replacing them with a count of the characters in what the system has perceived as a line of text.

These are just a few examples of the kinds of uninvited modifications that record management systems or that I/O support packages might make to your files. You will find that they are usually associated with the concepts of a line of text or the end of a file. In general, these modifications to your files are an attempt to make your life easier by doing things for you automatically. This might, in fact, work out for those who want to do nothing more than store some text in a file. Unfortunately, however, programmers building sophisticated file structures must sometimes spend a lot of time finding ways to disable this automatic assistance so they can have complete control over what they are building. Forewarned is forearmed: readers who encounter these kinds of difficulties as they build the file structures described in this text can take some comfort from the knowledge that the experience they gain in disabling automatic assistance will serve them well, over and over, in the future.
FIELD AND RECORD ORGANISATION:

Field: The basic unit of data is the field which contain a single data value

Record: Collection of fields is called as record.

(a) Field structures: There are many ways of adding structures to files to maintain the identity of fields. Four of the most common methods are:

1) Fix the length of fields.
2) Begin each field with a length indicator
3) Separate the fields with delimiters
4) Use a “keyword =value” expression to identify fields

(1) Fix the Length of Fields: In this method, the length of each field is fixed. We can define a ‘struct’ in ‘C’ to hold these fixed fields.

Ex: struct person
{
    char last[11];
    char first[11];
    char address[16];
    char city[16];
    char state[3];
    char zip[10];
};

In this example each field is a character array that can be a string value of some maximum size. The size of array is one larger than the longest string it can hold.

Suppose i/p is

Mary Ames      Alan Manson
123 Maple      90 East gate
Still water OK 74075   Ada OK 74820

Using this kind of fixed field length structure the output like as

<table>
<thead>
<tr>
<th>Ames</th>
<th>Mary</th>
<th>123 Maple</th>
<th>Still water</th>
<th>OK 74075</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manson</td>
<td>Alan</td>
<td>90 East gate</td>
<td>Ada</td>
<td>OK 74820</td>
</tr>
</tbody>
</table>

Disadvantages:

1) Adding all the padding required to bring the fields upto a fixed length makes the file much larger.
2) In some cases, the data is too long to fit into the allocated amount of space.

(2) Begin Each Field with a Length Indicator:

- In this method for every field, length is stored just ahead of the field.
- If the fields are not too long (<256 bytes) then it is possible to store the length in a single byte at the start of each byte.
We refer to these fields as length based.

Ex:

<table>
<thead>
<tr>
<th>04 Ames</th>
<th>04 Mary</th>
<th>09 123 Maple</th>
<th>10 Still water</th>
<th>02 OK</th>
<th>05 74075</th>
</tr>
</thead>
<tbody>
<tr>
<td>06 Manson</td>
<td>04 Alan</td>
<td>11 90 East gate</td>
<td>03 Ada</td>
<td>02 OK</td>
<td>05 74820</td>
</tr>
</tbody>
</table>

Begin each field with a length indicator

(3) **Separate the Fields with Delimiters:**

- Here the fields are separated with delimiters.
- Any special character or sequence of characters can be used as delimiters.
- These special characters are inserted after each field.

Ex:

Ames| Mary|123 Maple|Still water|OK|74075|
Manson|Alan|90 East gate|Ada|OK|74820|

(4) **Use a “keyword = value” Expression to Identify Fields:**

- In this method, a field provides information about itself

Ex: last = Ames   first = Mary   address = 123 Maple   city = Still water   State = OK   zip = 74075

Reading a stream of fields:

- Stream.getline () is used to read the fields.
- The arguments to getline() are a character array to hold the string, a maximum length and a delimiter.

(b) **RECORD STRUCTURES**

A record can be defined as a set of fields that belong together when the file is viewed in terms of a higher level of organization.

Following are some of the most often used methods for organizing the records of a file:

(1) Require that the records be a predictable no. of bytes in length
(2) Require that the records be a predictable no. of fields in length
(3) Begin each record with a length indicator
(4) Use an index to keep track of addresses
(5) Place a delimiter at the end of each record

(1) **Make Records A Predictable No. of Bytes (Fixed Length) Record:**

- A fixed length record file is one in which each record contains the same no. of bytes.
- The ‘C’ structure person, discussed before is an example for fixed length record.
- In this we have a fixed no. of fields, each with a predetermined length that combine to make a fixed length.
Fixed length records are frequently used as containers to hold variable numbers of variable length fields.

It is also possible to mix fixed and variable length fields with a record.

The following figure illustrates how variable length fields might be placed in a fixed length record.

<table>
<thead>
<tr>
<th>Ames</th>
<th>Mary</th>
<th>123 Maple</th>
<th>Still water</th>
<th>OK</th>
<th>74075</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manson</td>
<td>Alan</td>
<td>90 east gate</td>
<td>Ada</td>
<td>OK</td>
<td>74820</td>
</tr>
</tbody>
</table>

(2) **Make Records a Predictable No. of Fields:**
According to this method each record contains a fixed no. of fields.

Ex: In the following figure each record contains 6 fields.

Ames| Mary|123 Maple|Still water|OK|74075|Manson|Alan|90 East gate|Ada|OK|74820|

Even though the line has more fields, only the 6 fields are considered as a record.

(3) **Begin Each Record with a Length Indicator:**
- In this method each record is preceded by an integer that indicates how many bytes there are in the rest of record.
- This is a commonly used method for handling a variable length record.

(4) **Use An Index To Keep Track Of Address:**
- In this method we can use an index to keep a byte offset for each record in the original file
- The byte offsets allow us to find the beginning of each successive record and compute the length of each record
- We look up the position of a record in the index then seek to the record in the data file

(5) **Place A Delimiter At The End Of Each Record:**
- A common choice of record delimiter for files that contain readable text is the new line character: \n
- We can use any special characters as the end of record
- In the following figure we use a # character as a record delimiter
MANAGING FIXED-LENGTH, FIXED-FIELD BUFFER:

Class `FixedLengthBuffer` is the subclass of `IOBuffer` that supports read and write of fixed-length records. For this class, each record is of the same size. Instead of storing the record size explicitly in the file along with the record, the write method just writes the fixed-size record. The read method must know the size in order to read the record correctly. Each `FixedLengthBuffer` object has a protected field that records the record size.

Class `FixedFieldBuffer`, as shown in Fig. 4.17 and files `fixfld.h` and `fixfld.cpp`, supports a fixed set of fixed-length fields. One difficulty with this strategy is that the unpack method has to know the length of all of the fields. To make it convenient to keep track of the

```cpp
class FixedFieldBuffer: public FixedLengthBuffer
public:
    FixedFieldBuffer (int maxFields, int recordSize = 1000);
    FixedFieldBuffer (int maxFields, int * fieldSize);
    int AddField (int fieldSize); // define the next field
    int Pack (const void * field, int size = -1);
    int Unpack (void * field, int maxBytes = -1);
    int NumberOfFields () const; // return number of defined fields
protected:
    int * FieldSize; // array to hold field sizes
    int MaxFields; // maximum number of fields
    int NumFields; // actual number of defined fields

Figure 4.17 Class FixedFieldBuffer.
```
field lengths, class FixedFieldBuffer keeps track of the field sizes. The protected member FieldSize holds the field sizes in an integer array. The AddField method is used to specify field sizes. In the case of using a FixedFieldBuffer to hold objects of class Person, the InitBuffer method can be used to fully initialize the buffer:

```c
int Person::InitBuffer (FixedFieldBuffer & Buffer)
// initialize a FixedFieldBuffer to be used for Persons
{
    int result;
    result = Buffer . AddField (10); // LastName [11];
    result = result && Buffer . AddField (10); // FirstName [11];
    result = result && Buffer . AddField (15); // Address [16];
    result = result && Buffer . AddField (15); // City [16];
    result = result && Buffer . AddField (2); // State [3];
    result = result && Buffer . AddField (9); // ZipCode [10];
    return result;
}
```

Starting with a buffer with no fields, InitBuffer adds the fields one at a time, each with its own size. The following code prepares a buffer for use in reading and writing objects of class Person:

```c
FixedFieldBuffer Buffer(6, 61); // 6 fields, 61 bytes total
MaryAmeInitBuffer (Buffer);
```

Unpacking FixedFieldBuffer objects has to be done carefully. The object has to include information about the state of the unpacking. The member NextByte records the next character of the buffer to be unpacked, just as in all of the IObuffer classes. FixedFieldBuffer has additional member NextField to record the next field to be unpacked. The method FixedFieldBuffer::Unpack is implemented as follows:

```c
int FixedFieldBuffer :: Unpack (void * field, int maxBytes)
{
    if (NextField == NumFields || Packing)
        // buffer is full or not in unpacking mode
        return -1;
    int start = NextByte; // first byte to be unpacked
    int packSize = FieldSize[NextField]; // bytes to be unpacked
    memcpy (field, &Buffer[start], packSize); // move the bytes
    NextByte += packSize; // advance NextByte to following char
    NextField ++; // advance NextField
    if (NextField == NumFields) Clear (); // all fields unpacked
    return packSize;
}
```
PATTERN MATCHING ALGORITHMS

➢ In pattern matching problem, we are given a text string ‘T’ of length ‘n’ and a pattern string ‘p’ of length ‘m’, and want to bind whether ‘p’ is a substring of ‘T’.
➢ The notion of a “match” is that there is a substring of ‘T’. Starting at some index ‘I’ that matches ‘p’, character by character, so that \( t[i]=p[0], t[i+1]=p[1], \ldots, t[i+m-1]=p[m-1] \), i.e., \( p=t[i\ldots i+m-1] \).
➢ The output from a pattern matching algorithm is an integer including the starting index in ‘T’ of a substring matching ‘p’.

* PATTERN MATCHING ALGORITHMS

Three pattern matching algorithms are presented here.

i. Brute force pattern matching.
ii. The Boyer-Moore algorithm.
iii. The Knuth-Morris-Pratt algorithm.

BRUTE-FORCE PATTERN MATCHING

➢ In this approach, we simply test all the possible placements of ‘p’ relative to ‘T’.
➢ First, \( p[0]\ldots p[m-1] \) is compared with \( t[0]\ldots t[m-1] \) if there is a mismatch, then \( p[0]\ldots p[m-1] \) is compared with \( t[1] \) if there is a mismatch, again \( p[0]\ldots p[m-1] \) is compared against \( t[2], t[3] \)........
➢ This procedure is repeated until the pattern is found in the text or until the end of the text.

Ex: - T

\[
\begin{array}{cccccccc}
\text{a} & \text{g} & \text{t} & \text{g} & \text{g} & \text{a} & \text{g} & \text{a} & \text{g} & \text{t} & \text{t} & \text{a} & \text{a} & \text{g}
\end{array}
\]

- P

\[
\begin{array}{cccc}
\text{a} & \text{g} & \text{a} & \text{g}
\end{array}
\]

- \[
\begin{array}{cccc}
\text{a} & \text{g} & \text{a} & \text{g}
\end{array}
\]

- \[
\begin{array}{cccc}
\text{a} & \text{g} & \text{a} & \text{g}
\end{array}
\]

- \[
\begin{array}{cccc}
\text{a} & \text{g} & \text{a} & \text{g}
\end{array}
\]

- \[
\begin{array}{cccc}
\text{a} & \text{g} & \text{a} & \text{g}
\end{array}
\]

- \[
\begin{array}{cccc}
\text{a} & \text{g} & \text{a} & \text{g}
\end{array}
\]

- \[
\begin{array}{cccc}
\text{a} & \text{g} & \text{a} & \text{g}
\end{array}
\]

- \[
\begin{array}{cccc}
\text{a} & \text{g} & \text{a} & \text{g}
\end{array}
\]

- \[
\begin{array}{cccc}
\text{a} & \text{g} & \text{a} & \text{g}
\end{array}
\]

- \[
\begin{array}{cccc}
\text{a} & \text{g} & \text{a} & \text{g}
\end{array}
\]

- \[
\begin{array}{cccc}
\text{a} & \text{g} & \text{a} & \text{g}
\end{array}
\]

-> this pattern is found

- Here first \( t[0] \) is compared with \( p[0] \), \( t[1] \) with \( p[1] \), but \( t[1] \) not equal to \( p[2] \). So start comparison of \( p[0] \) from \( t[1] \). Again \( t[1] \) not equal to \( p[0] \). So again start comparison from \( t[2] \) with \( p[0] \). Repeat this until the pattern is found or end of the text is reached.
Algorithm brute force match \((T, P)\):

\[
\begin{align*}
N &= \text{length}(T) \\
M &= \text{length}(P) \\
\text{For } i &= 0 \text{ to } n-m \\
&\quad \{ \\
&\quad \quad j = 0 \\
&\quad \quad \text{While } (j < m \text{ and } T[i+j] = P[j]) \\
&\quad \quad \quad j = j+1 \\
&\quad \quad \text{If } j = m \text{ then} \\
&\quad \quad \quad \text{Return } i; \\
&\quad \} \\
\text{Return } \text{"there is no substring of \text{"T\text{" matching \text{"P\text{").} \\
\}
\]

- The above algorithm accepts two input values text \((T)\), pattern \((P)\). \(n\) is the length of the text \(T\). \(m\) is the length of the pattern \(P\).
- It consists of two nested loops, with the outer (for) loop indexing through all possible starting indices of the pattern in the text, and the inner loop indexing through each character of the pattern, comparing it to its potentially corresponding character in the text.
- The outer (for) loop is executed at most \(n-m+1\) times, and the inner loop (while) is executed at most \(m\) times.
- Thus the running time of the brute-force method is \(O((n-m+1)m) \Rightarrow O(nm)\).

**BOYER-MOORE ALGORITHM**

- Boyer-Moore (BM) pattern matching algorithm, can sometimes avoid comparisons between \(p\) and sizable fraction of the characters in \(T\).
- Boyer-Moore algorithm uses two time-saving heuristics.

They are:

1. **Looking-glass Heuristic**: When testing a possible placement of \(P\) against \(T\), begin the comparisons from the end of \(P\) and move backward to the front of \(P\).

2. **Character-jump Heuristic**: If there is a match of \(p[j]\) with \(t[j]\), then move \(j, i\) to previous locations and compare \(p[j-1]\) with \(t[i-1]\), if again there is a match then move \(i, j\) to previous locations i.e., \(p[j-2]\) with \(t[i-2]\) and compare. Repeat this until we get a mismatch or \(j\) becomes zero.

   - If \(j\) is zero and every character of \(p[j].....p[0]\) is matched with \(t[i].....t[i-j]\) then declare that the pattern was found.
   - If there is a mismatch, then it could be handled as follows:
During the testing of a possible placement of ‘p’ against T, a mismatch of text character t[i]=c with corresponding pattern character p[j] is handled as follows:-
   a) If ‘c’ is not available anywhere in ‘p’, then shift ‘p’ completely past t[i].
   b) Otherwise, shift ‘p’ until an occurrence of character ‘p’ in and the character in ‘p’ gets aligned with a character ‘c’ in t[i], is possible with a function called as last(c) function.

Last(c) can be defined as
   a) If ‘c’ is in ‘p’, then last(c) is the index of the last (right-most) occurrence of ‘c’ in ‘p’.
   b) Otherwise last(c) = -1.

The jump statement illustrated in figure.

**Case 1:**
```
   . . . . . . a . . . . . .
   . a . . b .
   . a . . b .
```

Where we use the notion l = last (t[i]).
If 1+l <=j, then shift the pattern by j-l units.

**Case 2:**
```
   . . . . . . a . . . . . .
   . a . . b .
   . a . . b .
```

If j < 1+l, then shift the pattern by one unit.

**Case 3:**
```
   a c a b a
   a c a b a
```

Here ‘y’ is not in pattern so shift ‘P’ completely past ‘y’.

Algorithm BM Match (T, P)
{
 Compute function last ()
 N=length (T);
M = length (P);
J = m - 1;
While (i < n) do 
{
    If (P[j] = T[i]) 
    {
        If (j = 0) then 
            Return i;
        Else 
            i = i - 1;
            j = j - 1;
    } 
    Else 
    {
        i = i + m - min (j, 1 + last T[i])
        j = m - 1;
    }
} 
Return “there is no sub string of ‘T’ matching ‘P’”

➔ The Boyer-Moore algorithm takes two input values i.e., text string ‘T’, pattern string ‘P’. ‘n’ is the no of characters in ‘T’. ‘M’ is the characters in ‘P’.
➔ The algorithm first computes last(c) value for every character ‘c’ in the pattern. It starts comparing from p[m-1] and continues to p.
➔ The worst case running time of BM algorithm is O (nm + 1Σ1). In the last () function takes time O (m + 1Σ1) and the actual search for the pattern takes O (nm) time in the worst case.

Table:

<table>
<thead>
<tr>
<th>Text</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
The pattern was found at 11\textsuperscript{th} location in the given text.

Last (): last or right most occurrence of `x` in pattern.

Text 'T'

1) \text{P[5]} is compared with \text{T[6]} is both are not matched. Check the condition \(i+l \leq j\) 1+4 \leq 5. So shift pattern j-l moves 5-4=1.

2) \text{P[4]}, \text{P[5]} are matched with corresponding text. But \text{P[3]} is not matched with \text{T[5]}. 
   \(1+l (a) =1+4 \leq 3\) false. \(j < 1+l (a) =3<1+4\) true. So shift pattern by 1 move

3) \text{P[5]} and \text{T[8]} are not matched \(1+l (a) \leq 5 \Rightarrow 1+4 \leq 5\) true shift `p` by \(j-l (5-4=1)\) shift.

4) \text{P[5]} and \text{T[9]} are not matched. \(l (d) =-1\) so shift pattern to past of d. 5-(-10)=6 apply 6 shifts.

5) \text{P[5]} and \text{T[15]} not matched. \(1+l (a) \leq j \Rightarrow 1+4 \leq 5\) true, so shift pattern by \(j=l \Rightarrow 5-4=\text{shift}\).

6) \text{P[5]} is matched with \text{T[16]}. 
   \text{P[4]} is matched with \text{T[13]}. 
   \text{P[3]} is matched with \text{T[14]}. 
   \text{P[0]} is matched with \text{T[11]}. 

So the pattern is found at 11\textsuperscript{th} position.
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Computer System Architecture
- Programming in Java
- Discrete Structures
- Data Structures
- Operating Systems
- Computer Networks
- Algorithms
- Android Programming
- Database Management Systems
- PHP Programming
- Software Engineering
- JavaScript
- Theory of Computation
- Java Server Pages
- Operational Research
- Python
- System Programming
- Microprocessor
- Data Mining
- Artificial Intelligence
- Computer Graphics
- Machine Learning
- Data Science

 Compiled Books: https://www.tutorialsduniya.com/compiled-books
 Programs: https://www.tutorialsduniya.com/programs
 Question Papers: https://www.tutorialsduniya.com/question-papers
 Python Notes: https://www.tutorialsduniya.com/python
 Java Notes: https://www.tutorialsduniya.com/java
 JavaScript Notes: https://www.tutorialsduniya.com/javascript
 JSP Notes: https://www.tutorialsduniya.com/jsp
 Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
 OR Notes: https://www.tutorialsduniya.com/operational-research
**KNUTH-MORRIS-PRATT ALGORITHM**

- Knuth-Morris-Pratt (KMP) is a pattern matching algorithm.
- The main idea of the KMP algorithm is to pre-process the pattern string ‘p’ so as to compute a failure function ‘f’.

**FAILURE FUNCTION**

- The failure function $f(j)$ is defined as the length of the longest prefix ‘P’ that is suffix of $P[1.......j]$.
- The failure function ‘f’ indicates the proper shift of ‘p’. so that, to the largest extent possible, we can reuse previously performed comparisons.
- The importance of this failure function is that it “encodes” repeated substrings inside the pattern itself.

**E.g:** The pattern is abacabd.

When the pattern is matched against text T, assume that every character of the pattern was matched except the last one ‘d’.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>d</td>
</tr>
</tbody>
</table>

A b a c a b d.

In this case we can move the pattern such that o 1 positions of pattern occupies the 4, 5 positions and we can avoid the compare here (i.e., $p[0]$, $p[1]$ and continue comparisons from $p[2]$ and soon).

- For the pattern $a b a c a b d$

Find out the failure function.

- To evaluate failure function (or) prefix table find out the prefix list and suffix list of every possible string in the pattern (i.e., from string length ‘0’ to string length m-1).
- Then if there is a common prefix and suffix is there for the string then find out the length of prefix or suffix. it becomes as the failure function $f(j)$. 

Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
<table>
<thead>
<tr>
<th>Prefix</th>
<th>Suffix</th>
<th>F (j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P [0]</td>
<td>a</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ab</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>aba</td>
<td>a, ab</td>
</tr>
<tr>
<td></td>
<td>abac</td>
<td>a, ab, aba</td>
</tr>
<tr>
<td></td>
<td>abaca</td>
<td>a, ab, aba, abac</td>
</tr>
<tr>
<td></td>
<td>abacab</td>
<td>a, ab, aba, abac, abaca</td>
</tr>
<tr>
<td></td>
<td>abacabd</td>
<td>a, ab, aba, abac, abaca, abacab</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Suffix</th>
<th>F (j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P [0]</td>
<td>a</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ab</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>aba</td>
<td>a, ab</td>
</tr>
<tr>
<td></td>
<td>abaa</td>
<td>a, ab, aba</td>
</tr>
<tr>
<td></td>
<td>abaab</td>
<td>a, ab, aba, abaa</td>
</tr>
<tr>
<td></td>
<td>abaaba</td>
<td>a, ab, aba, abaa, abaab</td>
</tr>
<tr>
<td></td>
<td>abaa</td>
<td>a, ab, aba, abaa, abaab</td>
</tr>
</tbody>
</table>
The KMP algorithm matches pattern 'p' against text 'T' starting from location p[0] and continues up to p[m-1], where 'm' is the length of the pattern.

When p[j] is compared with t[i], if there is a match then it increments the current indices (i.e., both 'i' and 'j').

If there is a mismatch, then the algorithm consults the failure function table to determine the new index in 'p' where we need to continue checking 'p' against 'T'.

Otherwise (there is a mismatch and we are at the beginning of p), we simply increment the index for 'T'. (And keep the index variable for 'p' at its beginning).

i.e., if j>0 and there is a mismatch then j is changed then j=f(j-1).

If j=0 then increment the index of t by L (i=i+1).

'J' position remains in the same way.

We repeat this process until we find a match of 'p' in 'T' or the index for 't' reaches n, the length 'T' (indicating that we did not find the pattern 'p' in 'T').

```plaintext
M=length(P);
N=length(T);
F=KMP Failure Function(P)
I=0
J=0
While(i<n) do
{
"
```
If\(p[j] = T[j]\) then

\[
\begin{align*}
\text{If}(j=m-1) & \text{ then} \\
& \text{Return } i-m+1; \\
& I=i+1; \\
& J=J+1;
\end{align*}
\]

Else if \(j>0\) then

\(J=f(j-1);\)

Else

\(I=i+1;\)

\}

Return “there is no substring of ‘T’ matching ‘p’;”

The main pattern of the KMP algorithm is the while-loop, which performs a comparison between a character in ‘T’ and a character in ‘p’ each iteration.

Depending upon the outcome of this comparison, the algorithm either moves on to the next characters in ‘T’ and ‘p’, consults the failure function for a new candidate character in ‘p’, or starts over with the next index in ‘T’.

Any comparisons that are skipped are actually unnecessary, for the failure function guarantees that all ignored comparisons are redundant.

Algorithm KMP Failure Function (P)

\{
\begin{align*}
I &= 1; \\
J &= 0; \\
F(0) &= 0; \\
\text{While } (I, m) \text{ do} \\
& \{ \\
& \text{If } (p[j] = p[i]) \text{ then} \\
& \{ \\
& \quad F(i) = j+1; \\
& \quad I = i+1; \\
& \quad J = J+1; \\
& \} \\
& \text{Else if } j > 0 \text{ then} \\
& \quad J = f(j-1); \\
& \text{Else} \\
& \quad \{ \\
& \quad \quad F(i) = 0; \\
& \quad \quad I = i+1; \\
& \quad \} \\
& \}
\}
\}
In this algorithm the pattern is compared to itself. Each time we have two characters that match, we set \( f(i) = j+1 \).

- Note that, since we have \( i > j \) throughout the execution of the algorithm, \( f(j-1) \) is always defined when we need to use it.
- The KMP Failure function runs in \( O(m) \) time.
- The KMP match runs in \( O(n) \) time.

Total time required for KMP match is \( O(m+n) \).

![Pattern Comparison Diagram]

1. Compare \((p[0], T[0]) \) (\( p[1], T[1] \)) all are same. But \( p[5], t[5] \) are not matched. Here \( 'j' \) is 5. so change \( j \) value to \( j = f(5-1) = f(4) = 1 \). Now compare \( t[5] \) with \( p[1] \).
2. Here \( p[0], t[4] \) not compared. Comparison directly starts from \( p[1] \). \( T[5] = p[1] \). here \( j > 0 \), \( j = f(1-1) = f(0) = 0 \).
4. \( T[9] \neq p[0] \). But \( j = 0 \), so increment \( I \) by 1. Now compare \( t[10] \) with \( p[0] \).

The pattern available at \( T[10] \).

**TRIES:**

- A trie is a tree-based data structure for storing strings in order to support fast pattern matching.
- The main application for tries is in information retrieval.
- The name trie came from the word retrieval.
- The primary operations that tries support are
  i. Pattern matching
  ii. Prefix matching.
**Standard Tries:**

Let's be a set of strings from alphabet \( \Sigma \), such that no string in 's' is a prefix of another string.

A standard trie for 's' is an ordered tree 'T' with the following properties:

- Each node of 'T', except the root, is labelled with a character of \( \Sigma \).
- The ordering of the children of an internal node of 'T' is determined by a canonical ordering of the alphabet \( \Sigma \).
- 'T' has 'S' external nodes, each associated with a string of 'S', such that the concatenation of the labels of the nodes on the path from the root to an external node 'V' of 'T' yields the string of s associated with 'V'.
- Thus, a trie 'T' represents the strings of 'S' with paths from the root to the external nodes. No string in 'S' is a prefix of another string. This ensures that each string of 'S' is uniquely associated with an external node of 'T'.

![Trie Diagram]

\[ S = \{ \text{bear, bell, bid, bull, buy, sell, stack, stop} \}. \]

**Advantages of Tries:**

1. In tries the keys are searched using common prefixes. Hence it is faster. The lookup of the keys depends upon the height case of binary search tree.

2. Tries take less space when they contain large number of short strings. As nodes are shared between the keys.

3. Tries help with longest prefix matching, when we want to find the key.

**Applications of Tries:**

1. Tries has an ability to insert, delete or search for the entries. Hence they are used in building dictionaries such as entries for telephone numbers, English words.

2. Tries are also used in spell-checking softwares.
**Digital Search Tree:**

- A digital search tree is a binary tree in which each node contains one element.
- The element-to-node assignment is determined by the binary representation of the element keys.
- Number the bits in the binary representation of a key from left to right beginning at one.
  Ex: bit one of 1000 is 1, and bits two, three and four are 0.
- All keys in the left subtree of a node at level ‘I’ have bit ‘I’ equal to zero whereas those in the right subtree of nodes at this level have bit i=1.
  Ex: construct a digital binary search tree with keys 1000, 0010, 1001, 0001, 1100 and 0000.

```
A       00001
S       10011
E       00101
R       10010
C       00011
H       01000
I       01001
N       01110
G       00111
X       11000
M       01101
P       10000
```

Ex: build the digital search tree
Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.
TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research