Experiment No 1

Problem Statement: Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client's telephone number. Make use of two collision handling techniques.

Objectives:
1. To understand concept of Hashing
2. To understand to find record quickly using hash function.
3. To understand collision handling techniques.

Software Requirements: Open Source Python

Input: Name and telephone number of N users

Output: Specific User data

Theory:

Hashing: Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

Example: h(large_value) = large_value % m

Here, h() is the required hash function and 'm' is the size of the hash table. For large values, hash functions produce value in a given range.

Collision Handling

If we know the keys beforehand, then we have can have perfect hashing. In perfect hashing, we do not have any collisions. However, If we do not know the keys, then we can use the following methods to avoid collisions:

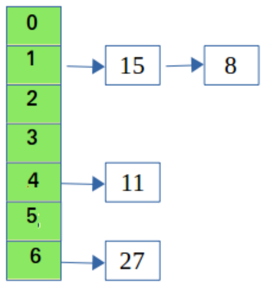
1. Chaining   2. Open Addressing (Linear Probing, Quadratic Probing, Double Hashing)

Chaining

While hashing, the hashing function may lead to a collision that is two or more keys are mapped to the same value. Chain hashing avoids collision. The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let's say hash table with 7 buckets (0, 1, 2, 3, 4, 5, 6)

Keys arrive in the Order (15, 11 , 27 , 8)



Algorithm

1. Declare an array of a linked list with the hash table size.

2. Initialize an array of a linked list to NULL.

3. Find hash key.

4. If chain[key] == NULL

   Make chain[key] points to the key node.

5. Otherwise(collision),

   Insert the key node at the end of the chain[key].

Open addressing

Linear Probing:
In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

The function used for rehashing is as follows: rehash(key) = (n+1)%table-size.

Algorithm:

1. *Calculate the hash key. i.e. key = data % size*
2. *Check, if hashTable[key] is empty*
   - *store the value directly by hashTable[key] = data*
3. *If the hash index already has some value then*
   - *check for next index using key = (key+1) % size*
4. *Check, if the next index is available hashTable[key] then store the value. Otherwise try for next index.*
5. *Do the above process till we find the space.*

Quadratic Probing

Quadratic probing is a method with the help of which we can solve the problem of clustering that was discussed above. This method is also known as the mid-square method. In this method, we look for the i^2'th slot in the i^{th} iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.

Double Hashing: The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the i^{th} rotation.

Conclusion: In this way we have implemented Hash table for quick lookup using Python.

Experiment No 2

Problem Statement: Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with / without replacement.

Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys

must be unique Standard Operations: Insert(key, value), Find(key), Delete(key)

Objectives:
1. To understand Dictionary (ADT)
2. To understand concept of hashing
3. To understand concept & features like searching using hash function

Software Requirements: Open Source Python

Input: No. of. elements with key and value pair

Output: Create dictionary using hash table and search the elements in table.

Theory:

Dictionary ADT

Dictionary (map, association list) is a data structure, which is generally an association of

unique keys with some values. One may bind a value to a key, delete a key (and naturally an

associated value) and lookup for a value by the key. Values are not required to be unique. Simple

usage example is an explanatory dictionary. In the example, words are keys and explanations are

values.

Dictionary Operations

 Dictionary create()

creates empty dictionary

 boolean isEmpty(Dictionary d)

tells whether the dictionary d is empty

 put(Dictionary d, Key k, Value v)

associates key k with a value v; if key k already presents in the dictionary old value is replaced by v

 Value get(Dictionary d, Key k)

returns a value, associated with key kor null, if dictionary contains no such key

 remove(Dictionary d, Key k)

removes key k and associated value

 destroy(Dictionary d)

destroys dictionary d

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.

Basic Operations of hash table

Following are the basic primary operations of a hash table.

 Search − Searches an element in a hash table.

 Insert − inserts an element in a hash table.

 delete − Deletes an element from a hash table.

1. DataItem

Define a data item having some data and key, based on which the search is to be conducted in a

hash table.

```
struct DataItem {
 int data;
 int key;
};
```

2. Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
 return key % SIZE;
}
```

3. Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code

Example

```
struct DataItem *search(int key) {
 //get the hash
 int hashIndex = hashCode(key);
 //move in array until an empty
 while(hashArray[hashIndex] != NULL) {
```

```
 if(hashArray[hashIndex]->key == key)
 return hashArray[hashIndex];
 //go to next cell
 ++hashIndex;
 //wrap around the table
 hashIndex %= SIZE;
 }
 return NULL;
}
```

4. Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Example

```
void insert(int key,int data) {
 struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
 item->data = data;
 item->key = key;
 //get the hash
 int hashIndex = hashCode(key);
 //move in array until an empty or deleted cell
 while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
 //go to next cell
 ++hashIndex;
 //wrap around the table
 hashIndex %= SIZE;
 }
 hashArray[hashIndex] = item;
}
```

5. Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the

index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Example

```
struct DataItem* delete(struct DataItem* item) {
 int key = item->key;
 //get the hash
 int hashIndex = hashCode(key);
 //move in array until an empty
 while(hashArray[hashIndex] !=NULL) {
 if(hashArray[hashIndex]->key == key) {
 struct DataItem* temp = hashArray[hashIndex];
 //assign a dummy item at deleted position
 hashArray[hashIndex] = dummyItem;
 return temp;
 }
 //go to next cell
 ++hashIndex;
 //wrap around the table
 hashIndex %= SIZE;
 }
 return NULL;

}
```

Conclusion: This program gives us the knowledge of dictionary(ADT)

Experiment No 3

Problem Statement:

A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method

Objectives:
1. To understand concept of tree data structure

2. To understand concept & features of object oriented programmin

Software Requirements g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Input: Book name & its number of sections and subsections along with name

Output: Formation of tree structure for book and its sections

Theory:

Introduction to Tree:

Definition:

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship

that satisfies the following

• if T is not empty, T has a special tree called the root that has no parent

• each node v of T different than the root has a unique parent node w; each node with parent w is a child of w

Recursive definition

• T is either empty

• or consists of a node r (the root) and a possibly empty set of trees whose roots are the children of r

Tree is a widely-used data structure that emulates a tree structure with a set of linked

nodes.

A node may contain a value or a condition or represent a separate data structure or a tree of its own. Each node in a tree has zero or more child nodes, which are one level lower in the tree hierarchy (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent. A node that has no childs is called a leaf, and that node is of course at the bottommost level of the tree. The height of a node is the length of the longest path to a leaf from that node.

The height of the root is the height of the tree. In other words, the "height" of tree is the "number of levels" in the tree. Or more formaly, the height of a tree is defined as follows:

1. The height of a tree with no elements is 0

2. The height of a tree with 1 element is 1

3. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.

The depth of a node is the length of the path to its root (i.e., its root path). Every child node is always one level lower than his parent. The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root). All other nodes can be reached from it by following edges or links.

Important Terms

Following are the important terms with respect to tree.

 Path − Path refers to the sequence of nodes along the edges of a tree.

 Root − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.

 Parent − Any node except the root node has one edge upward to a node called parent.

 Child − The node below a given node connected by its edge downward is called its child node.

 Leaf − The node which does not have any child node is called the leaf node.

 Subtree − Subtree represents the descendants of a node.

 Visiting − Visiting refers to checking the value of a node when control is on the node.

 Traversing − Traversing means passing through nodes in a specific order.

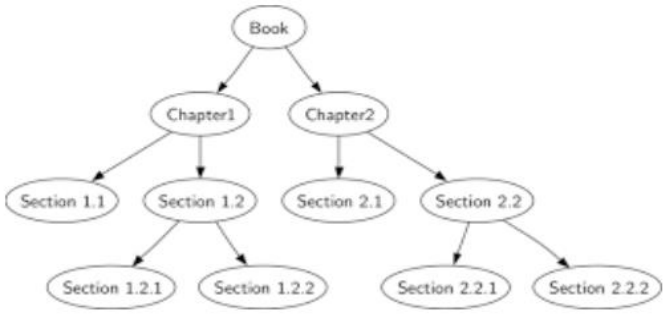
 Levels − Level of a node represents the generation of a node. If the root node is at level

0, then its next child node is at level 1, its grandchild is at level 2, and so on.

 keys − Key represents a value of a node based on which a search operation is to be

carried out for a node.

For this assignment we are considering the tree as follows.



Conclusion: This program gives us the knowledge tree data structure

Experiment No 4

Problem Statement:

Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree - i. Insert new node ii. Find number of nodes in longest path iii. Minimum data value found in the tree iv. Change a tree so that the roles of the left and right pointers are swapped at every node v. Search a value

Objectives:

1. To understand concept of Tree & Binary Tree.
2. To analyze the working of various Tree operations

Software Requirements g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Input: Data with distinct key values

Output: Binary Search tree

Theory:

Tree

Tree represents the nodes connected by edges also a class of graphs that is acyclic is termed as trees. Let us now discuss an important class of graphs called trees and its associated terminology.

Trees are useful in describing any structure that involves hierarchy. Familiar examples of such

structures are family trees, the hierarchy of positions in an organization, and so on.

Binary Tree

A binary tree is made of nodes, where each node contains a "left" reference, a

"right" reference, and a data element. The topmost node in the tree is called the root.

Every node (excluding a root) in a tree is connected by a directed edge from exactly one other

node. This node is called a parent. On the other hand, each node can be connected to arbitrary number

of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data

Algorithm to insert a node :

Step 1 - Search for the node whose child node is to be inserted. This is a node at some level i, and a node is to be inserted at the level i +1 as either its left child or right child. This is the node after which the insertion is to be made.

Step 2 - Link a new node to the node that becomes its parent node, that is, either the Lchild or the Rchild

Deletion

Case 1. Delete a Leaf Node in BST

Case 2. Delete a Node with Single Child in BST

Deleting a single child node is also simple in BST. Copy the child to the node and delete the node.

Case 3. Delete a Node with Both Children in BST

Deleting a node with both children is not so simple. Here we have to delete the node is such a way, that the resulting tree follows the properties of a BST.

The trick is to find the inorder successor of the node. Copy contents of the inorder successor to the node, and delete the inorder successor.

Algorithm

Delete (TREE, ITEM)

```
Step 1: IF TREE = NULL
   Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA
  Delete(TREE->LEFT, ITEM)
  ELSE IF ITEM > TREE -> DATA
   Delete(TREE -> RIGHT, ITEM)
  ELSE IF TREE -> LEFT AND TREE -> RIGHT
  SET TEMP = findLargestNode(TREE -> LEFT)
  SET TREE -> DATA = TEMP -> DATA
   Delete(TREE -> LEFT, TEMP -> DATA)
  ELSE
   SET TEMP = TREE
   IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
   SET TREE = NULL
  ELSE IF TREE -> LEFT != NULL
  SET TREE = TREE -> LEFT
  ELSE
    SET TREE = TREE -> RIGHT
  [END OF IF]
  FREE TEMP
[END OF IF]
Step 2: END
```

Conclusion : Thus we have studied the implementation of various Binary tree operation

Experiment No 5

Problem Statement:

Construct an expression tree from the given prefix expression eg. +--a*bc/def and traverse it using postordertraversal (non recursive)

Objectives:

1. To understand concept of Expression Tree.
2. To analyze the working of various Tree traversals

Software Requirements g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

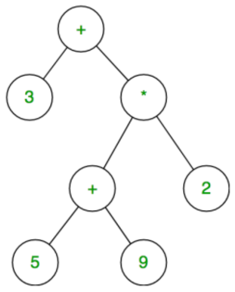
Input : Set of operators and operands for a given expression

Output: Expression tree with post order traversal

Theory:

Expression Tree: The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand

Example: expression tree for 3 + ((5+9)*2) would be:



Construction of Expression Tree:
Now For constructing an expression tree we use a stack. We loop through input expression and do the following for every character.

1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.

In the end, the only element of the stack will be the root of an expression tree.

There are three different types of depth-first traversals, :

 PreOrder traversal - visit the parent first and then left and right children;

 InOrder traversal - visit the left child, then the parent and the right child;

 PostOrder traversal - visit left child, then the right child and then the parent

Algorithm Postorder traversal:

until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

Step 3 − Visit root node.

Conclusion : Thus we have studied the implementation of various Binary tree operations

Experiment No 6

Problem Statement: Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph.

Objectives:

1. To understand directed and undirected graph.
2. To implement program to represent graph using adjacency matrix and list.
3. To perform DFS, BFS operations on Graph

Software Requirements g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Input : Values representing nodes and links between these nodes for a given graph

Output: Adjacency matrix and list representation of given graph and traversal paths using BFS and DFS

Theory:

Graph is a data structure that consists of following two components:

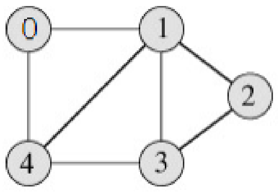
1. A finite set of vertices also called as nodes.

2. A finite set of ordered pair of the form (u, v) called as edge.

The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale. See this for more applications of graph.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

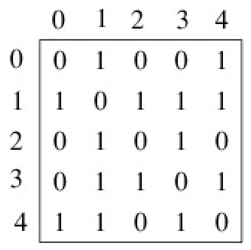
1. Adjacency Matrix 2. Adjacency List

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:

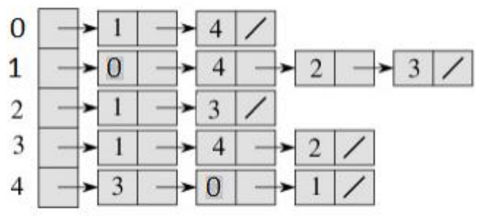
Adjacency Matrix Representation



$$
\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
2 & 0 & 1 & 0 & 1 & 0 \\
3 & 0 & 1 & 1 & 0 & 1 \\
4 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Graph Traversals:

Breadth First Search (BFS) is a fundamental **graph traversal algorithm**. It involves visiting all the connected nodes of a graph in a level-by-level manner.

Algorithm:

1. **Initialization:** Enqueue the starting node into a queue and mark it as visited.
2. **Exploration:** While the queue is not empty:
    - Dequeue a node from the queue and visit it (e.g., print its value).
    - For each unvisited neighbor of the dequeued node:
        - Enqueue the neighbor into the queue.
        - Mark the neighbor as visited.
3. **Termination:** Repeat step 2 until the queue is empty.

Pseudocode:
```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
   remove the head u of Q
   mark and enqueue all (unvisited) neighbours of u
```

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Algorithm:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Pseudocode:
```
DFS(G, u)
   u.visited = true
   for each v ∈ G.Adj[u]
      if v.visited == false
         DFS(G,v)

init() {
   For each u ∈ G
      u.visited = false
    For each u ∈ G
      DFS(G, u)
}
```

Conclusion: Implemented program for graph presentation in adjacency matrix and list and graph traversing techniques

Experiment No 7

Problem Statement:

There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph. Check whether the graph is connected or not. Justify the storage representation used.

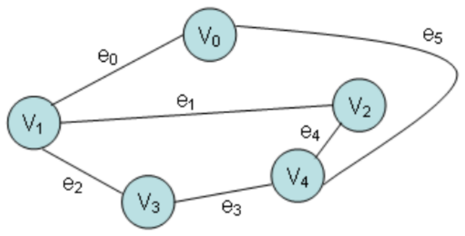
Objectives:
1. To understand concept of Graph data structure
2. To understand concept of representation of graph.
Software Requirements g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Input : 1.Number of cities.

2.Time required to travel from one city to another
Output Create Adjacency matrix to represent path between various cities.

Theory:

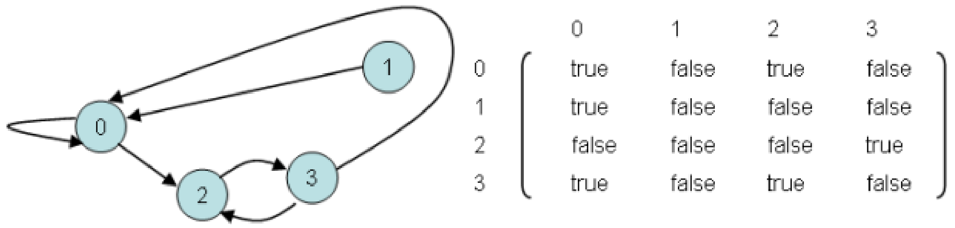
Graphs are the most general data structure. They are also commonly used data structures.
Graph definitions:
 A non-linear data structure consisting of nodes and links between nodes.
Undirected graph definition:
 An undirected graph is a set of nodes and a set of links between the nodes.
 Each node is called a vertex, each link is called an edge, and each edge connects two vertices.
 The order of the two connected vertices is unimportant.
 An undirected graph is a finite set of vertices together with a finite set of edges. Both sets might be empty, which is called the empty graph.



Graph Implementation:

Different kinds of graphs require different kinds of implementations, but the fundamental concepts of all graph implementations are similar. We'll look at several representations for one particular kind of graph: directed graphs in which loops are allowed.

Representing Graphs with an Adjacency Matrix



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & \text{true} & \text{false} & \text{true} & \text{false} \\ 1 & \text{true} & \text{false} & \text{false} & \text{false} \\ 2 & \text{false} & \text{false} & \text{false} & \text{true} \\ 3 & \text{true} & \text{false} & \text{true} & \text{false} \end{array}$$

Fig: Graph and adjacency matrix

Definition:

- An adjacency matrix is a square grid of true/false values that represent the edges of a

graph.

- If the graph contains n vertices, then the grid contains n rows and n columns.

- For two vertex numbers i and j, the component at row i and column j is true if there is an edge from vertex i to vertex j; otherwise, the component is false.

We can use a two-dimensional array to store an adjacency matrix:

boolean[][] adjacent = new boolean[4][4];

Once the adjacency matrix has been set, an application can examine locations of the matrix to determine which edges are present and which are missing.

Representing Graphs with Edge Lists

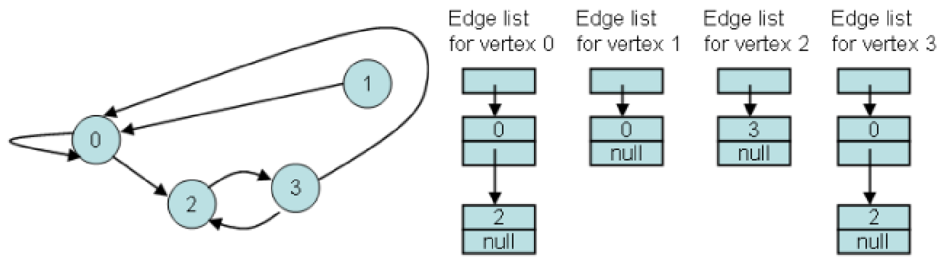


Fig: Graph and adjacency list for each node

Definition:

- A directed graph with n vertices can be represented by n different linked lists.

- List number i provides the connections for vertex i.

 For each entry j in list number i, there is an edge from i to j.

Loops and multiple edges could be allowed.

Representing Graphs with Edge Sets

To represent a graph with n vertices, we can declare an array of n sets of integers. For example:

IntSet[] connections = new IntSet[10]; // 10 vertices

A set such as connections[i] contains the vertex numbers of all the vertices to which vertex i is

connected.

Conclusion: This program gives us the knowledge of adjacency matrix graph

Experiment No 8

Problem Statement:

Given sequence k = k1 < … < kn of n sorted keys, with a search probability pi for each key ki. Build the Binary search tree that has the least search cost given the access probability for each key

Objectives:
1. To understand concept of OBST.
2. To understand concept & features like extended binary search tree.

Software Requirements g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

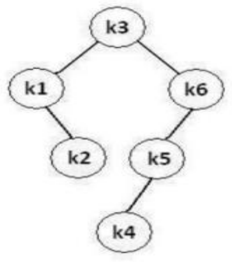
Input : 1.No.of Element. 2. key values 3. Key Probability

Output Create binary search tree having optimal searching cost

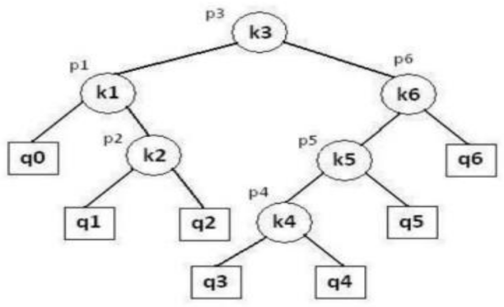
Theory:

An optimal binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum. For the purpose of a better presentation of optimal binary search trees, we will consider "extended binary search trees", which have the keys stored at their internal nodes. Suppose "n" keys k1, k2, … k n are stored at the internal nodes of a binary search tree. It is assumed that the keys are given in sorted order, so that k1< k2 < … < kn.

An extended binary search tree is obtained from the binary search tree by adding successor nodes to each of its terminal nodes as indicated in the following figure by squares:



Binary search tree



Extended binary search tree

In the extended tree:

 The squares represent terminal nodes. These terminal nodes represent unsuccessful searches of the tree for key values. The searches did not end successfully, that is, because they represent key values that are not actually stored in the tree;

 The round nodes represent internal nodes; these are the actual keys stored in the tree;

 Assuming that the relative frequency with which each key value is accessed is known, weights can be assigned to each node of the extended tree (p1 … p6). They represent the relative frequencies of searches terminating at each node, that is, they mark the successful searches.

 If the user searches a particular key in the tree, 2 cases can occur:

 1 – the key is found, so the corresponding weight „p‟ is incremented;

 2 – the key is not found, so the corresponding „q‟ value is incremented.

GENERALIZATION:

The terminal node in the extended tree that is the left successor of k1 can be interpreted as representing all key values that are not stored and are less than k1. Similarly, the terminal node in the extended tree that is the right successor of kn, represents all key values not stored in the tree that are greater than kn. The terminal node that is successes between ki and ki-1 in an inorder traversal represent all key values not stored that lie between ki and ki - 1.

ALGORITHMS

We have the following procedure for determining R(i, j) and C(i, j) with 0 <= i <= j <= n:

```
PROCEDURE COMPUTE_ROOT(n, p, q; R, C)
begin
    for i = 0 to n do
        C (i, i) ←0
        W (i, i) ←q(i)
    for m = 0 to n do
        for i = 0 to (n – m) do
            j ←i + m
            W (i, j) ←W (i, j – 1) + p (j) + q (j)
            *find C (i, j) and R (i, j) which minimize the
            tree cost
end
```

The following function builds an optimal binary search tree

```
FUNCTION CONSTRUCT(R, i, j)
begin
```

```
*build a new internal node N labeled (i, j)
k ←R (i, j)
f i = k then
*build a new leaf node N‟ labeled (i, i)
else
*N‟ ←CONSTRUCT(R, i, k)
*N‟ is the left child of node N
if k = (j – 1) then
*build a new leaf node N‟‟ labeled (j, j)
else
*N‟‟ ←CONSTRUCT(R, k + 1, j)
*N‟‟ is the right child of node N
return N
end
```

COMPLEXITY ANALYSIS:

The algorithm requires O (n2) time and O (n2) storage. Therefore, as „n‟ increases it will run out of storage even before it runs out of time. The storage needed can be reduced by almost half by implementing the two-dimensional arrays as one-dimensional array

Conclusion: This program gives us the knowledge OBST, Extended binary search tree.

Experiment No 9

Problem Statement:

A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword

Objectives:

1. To understand concept of height balanced tree data structure.
2. To understand procedure to create height balanced tree

Software Requirements g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Input : Dictionary word and its meaning.

Output : Allow Add, delete operations on dictionary and also display data in sorted order

Theory:

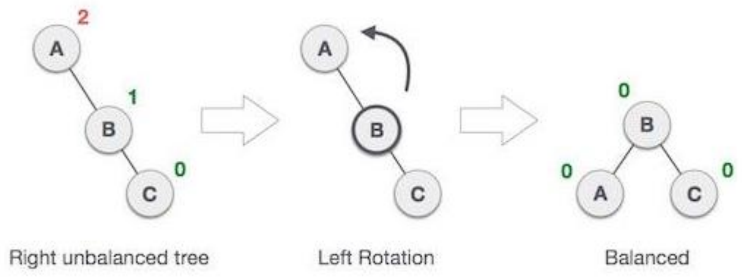
An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR as its left and right sub trees. The T is height balance if and only if Its balance factor is 0, 1, -1.

AVL (Adelson- Velskii and Landis) Tree: A balance binary search tree. The best search time, that is O (log N) search times. An AVL tree is defined to be a well-balanced binary search tree in which each of its nodes has the AVL property. The AVL property is that the heights of the left and right sub-trees of a node are either equal or if they differ only by 1. This difference is called the **Balance Factor**

AVL Rotations
To balance itself, an AVL tree may perform the following four kinds of rotations −
 Left rotation
 Right rotation
 Left-Right rotation
 Right-Left rotation

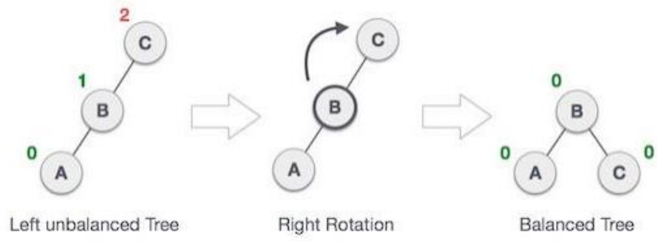
The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2

Left Rotation
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

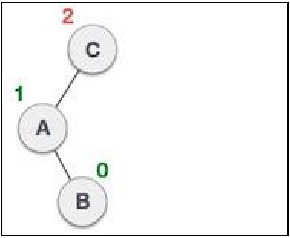
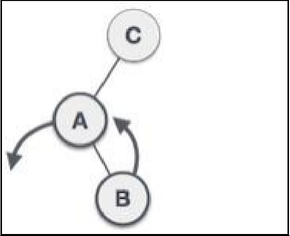
Right Rotation
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation

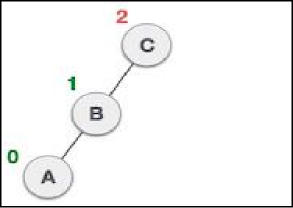
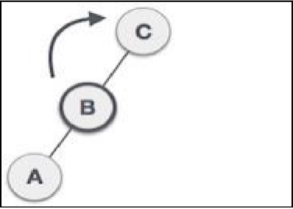
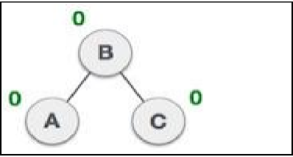


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

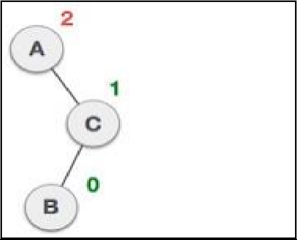
Left-Right Rotation
Double rotations are slightly complex version of already explained versions of rotations.
To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left

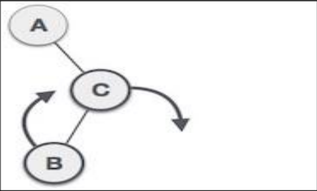
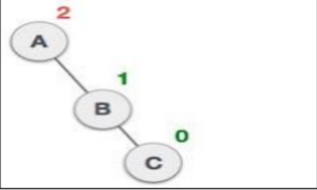
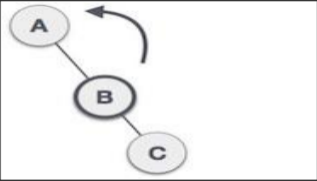
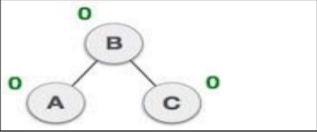
rotation followed by right rotation.

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.

	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.

	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

Algorithm AVL TREE:

Insert:-

```
1. If P is NULL, then
I. P = new node
II. P ->element = x
III. P ->left = NULL
IV. P ->right = NULL
V. P ->height = 0
2. else if x>1 => x<P ->element
a.) insert(x, P ->left)
b.) if height of P->left -height of P ->right =2
1. insert(x, P ->left)
2. if height(P ->left) -height(P ->right) =2
if x<P ->left ->element
P =singlerotateleft(P)
else
P =doublerotateleft(P)
3. else
if x<P ->element
a.) insert(x, P -> right)
b.) if height (P -> right) -height (P ->left) =2
if(x<P ->right) ->element
P =singlerotateright(P)
else
P =doublerotateright(P)
```

```
4. else
Print already exits
5. int m, n, d.
6. m = AVL height (P->left)
7. n = AVL height (P->right)
8. d = max(m, n)
9. P->height = d+1
10. Stop
```

RotateWithLeftChild(AvlNode k2)

```
⍰ AvlNode k1 = k2.left;
⍰ k2.left = k1.right;
⍰ k1.right = k2;
⍰ k2.height = max( height( k2.left ), height( k2.right ) ) + 1;
⍰ k1.height = max( height( k1.left ), k2.height ) + 1;
⍰ return k1;
```

RotateWithRightChild(AvlNode k1)

```
⍰ AvlNode k2 = k1.right;
⍰ k1.right = k2.left;
⍰ k2.left = k1;
⍰ k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
⍰ k2.height = max( height( k2.right ), k1.height ) + 1;
⍰ return k2;
```

doubleWithLeftChild(AvlNode k3)

```
⍰ k3.left = rotateWithRightChild( k3.left );
⍰ return rotateWithLeftChild( k3 );
```

doubleWithRightChild(AvlNode k1)

```
⍰ k1.right = rotateWithLeftChild( k1.right );
⍰ return rotateWithRightChild( k1 );
```

Conclusion: This program gives us the knowledge height balanced binary tree.

Experiment No 10

Problem Statement:

Consider a scenario for Hospital to cater services to different kinds of patients as Serious (top priority), b) non-serious (medium priority), c) General Checkup (Least priority). Implement the priority queue to cater services to the patients

Objectives:

1. To understand the concept of priority Queue.
2. How data structures Queue is represented as an ADT.

Software Requirements : g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Input : Name of patients & category of patient

Output : Priority queue cater services to the patients based on priorities.

Theory:

Priority queue is an abstract data type in computer programming that supports the following three operations:

 insertWithPriority: add an element to the queue with an associated priority
 getNext: remove the element from the queue that has the highest priority, and return it (also known as "PopElement(Off)", or "GetMinimum")
 peekAtNext (optional): look at the element with highest priority without removing it.

The rule that determines who goes next is called a queueing discipline. The simplest queueing discipline is called FIFO, for "first-in-first-out." The most general queueing discipline is priority queueing, in which each customer is assigned a priority, and the customer with the highest priority goes first, regardless of the order of arrival. The reason I say this is the most general discipline is that the priority can be based on anything: what time a flight leaves, how many groceries the customer has, or how important the customer is. Of course, not all queueing disciplines are "fair," but fairness is in the eye of the beholder. The Queue ADT and the Priority Queue ADT have the same set of operations and their interfaces are the same. The difference is in the semantics of the operations: A Queue uses the FIFO policy, and a Priority Queue (as the name suggests) uses the priority queueing policy. As with most ADTs, there are a number of ways to implement queues Since a queue is a collection of items, we can use any of the basic mechanisms for storing collections: arrays, lists, or vectors. Our choice among them will be based in part on their performance--- how long it takes to perform the operations we want to perform--- and partly on ease of implementation.

ALGORITHM:

```
ALGORITHM:
Define structure for Queue (Priority, Patient Info, Next Pointer).
```

Empty Queue:

```
Return True if Queue is Empty else False.
isEmpty (Front)
Front is pointer of structure, which is first element of Queue.
Step 1: If Front = = NULL
Step 2: Return 1
Step 3: Return 0
```

Insert Function:

```
Insert Patient in Queue with respect to the Priority.
Front is pointer variable of type Queue, which is 1st node of Queue.
Patient is a pointer variable of type Queue, which hold the information about new patient.
Insert (Front, Queue)
Step 1: If Front = = NULL //Queue Empty
Then Front = Patient;
Step 2: Else if Patient->Priority > Front->Priority Then
```

```
i) Patient->Next = Front;
ii) Front=Patient;
Step 3: Else
 A) Temp=Front;
B) Do Steps a while Temp! = NULL and Patient->Priority <= Temp->Next->Priority
a) Temp=Temp->Next;
c) Patient->Next = Temp->Next;
Temp->Next = Patient;
Step 4: Stop
```

Delete Patient details from Queue after patient get treatment:

Front is pointer variable of type Queue, which is 1st node of Queue.
Delete Node from Front.

```
Delete (Front)
Step 1: Temp = Front;
Step 2: Front = Front->Next;
Step 3: return Temp
```

Display Queue Front:

Front is pointer variable of type Queue, which is 1st node of Queue.

```
Display (Front)
Step 1: Temp = Front;
Step 2: Do Steps while Temp! = NULL
a) Display Temp Data
b) If Priority 1 Then ¯General Checkup‖;
Else If Priority 2 Then Display ¯ Non-serious"; Else
If Priority 3 Then Display "Serious"
Else Display "Unknown";
c)Temp = Temp->Next;
Step 3: Stop.
```

Conclusion: After successful implementation of this assignment, we understood the priority queue as ADT.

Experiment No 11

Problem Statement: Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular employee. If record of student does not exist an appropriate message is displayed. If it is, then the system displays the student details. Use sequential file to main the data.

Objectives:
1. To understand concept of file organization in data structure.
2. To understand concept & features of sequential file organization.

Software Requirements : g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Input : Details of student like roll no, name, address division etc

Output If record of student does not exist an appropriate message is displayed otherwise the student details are displayed.

Theory:

File organization refers to the relationship of the key of the record to the physical location of that record in the computer file. File organization may be either physical file or a logical file.

A physical file is a physical unit, such as magnetic tape or a disk. A logical file on the other hand is a complete set of records for a specific application or purpose. A logical file may occupy a part of physical file or may extend over more than one physical file.

There are various methods of file organizations. These methods may be efficient for certain types of access/selection meanwhile it will turn inefficient for other selections. Hence it is up to the programmer to decide the best suited file organization method depending on his requirement.

Some of the file organizations are

1. Sequential File Organization

2. Heap File Organization

3. Hash/Direct File Organization

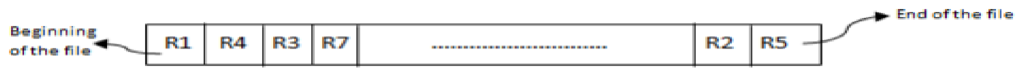
4. Indexed Sequential Access Method

5. B+ Tree File Organization

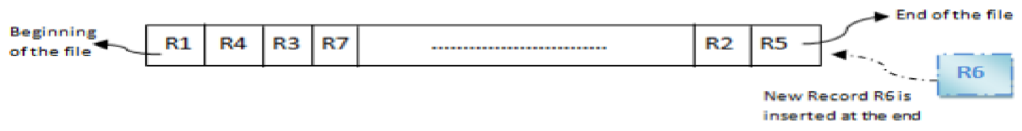
6. Cluster File Organization

Sequential File Organization: It is one of the simple methods of file organization. Here each file/records are stored one after the other in a sequential manner. This can be achieved in two ways:

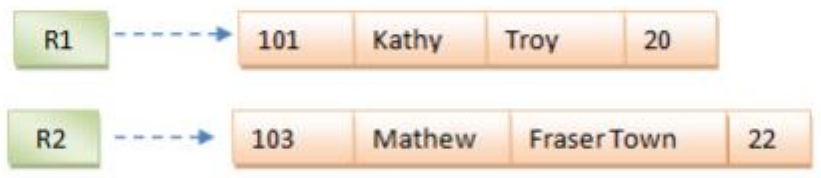
 Records are stored one after the other as they are inserted into the tables. This method is called pile file method. When a new record is inserted, it is placed at the end of the file. In the case of any modification or deletion of record, the record will be searched in the memory blocks. Once it is found, it will be marked for deleting and new block of record is entered.



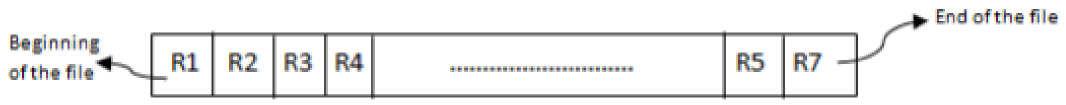
Inserting a new record:



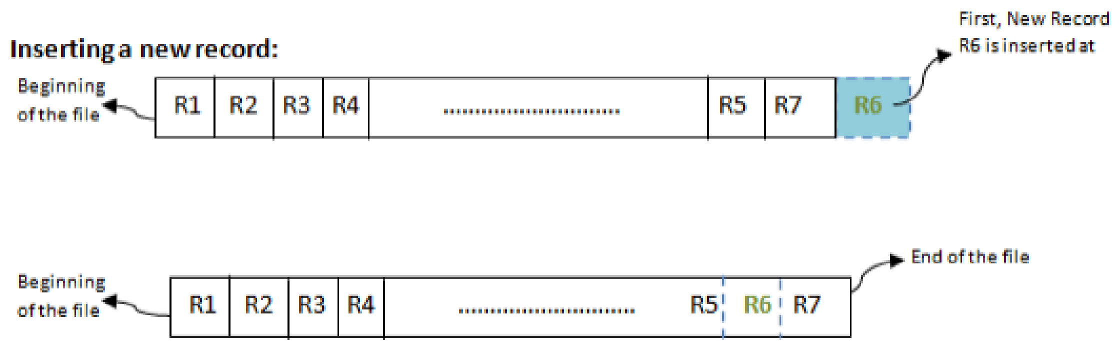
In the diagram above, R1, R2, R3 etc are the records. They contain all the attribute of a row. i.e.; when we say student record, it will have his id, name, address, course, DOB etc. Similarly R1, R2, R3 etc can be considered as one full set of attributes



In the second method, records are sorted (either ascending or descending) each time they are inserted into the system. This method is called sorted file method. Sorting of records may be based on the primary key or on any other columns. Whenever a new record is inserted, it will be inserted at the end of the file and then it will sort – ascending or descending based on key value and placed at the correct position. In the case of update, it will update the record and then sort the file to place the updated record in the right place. Same is the case with delete.



Inserting a new record:



Advantages: • Simple to understand. • Easy to maintain and organize • Loading a record requires only the record key. • Relatively inexpensive I/O media and devices can be used. • Easy to reconstruct the files. • The proportion of file records to be processed is high.

Disadvantages: • Entire file must be processed, to get specific information. • Very low activity rate stored. • Transactions must be stored and placed in sequence prior to processing. • Data redundancy is high, as same data can be stored at different places with different keys. • Impossible to handle random enquiries.

Conclusion: This program gives us the knowledge sequential file organization.

Experiment No 12

Problem Statement: Assume we have two input and two output tapes to perform the sorting. The internal memory can hold and sort m records at a time. Write a program in java for external sorting. Find out time complexity.

Objectives:

Software Requirements : g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Input : To understand the concept and basic of Sorting external memory handling.

• To analyse time complexity of sorting algorithm

Output : External memory handling

Theory:

External Sorting: External Sorting is sorting the lists that are so large that the whole list cannot be contained inthe internal memory of a computer. Assume that the list(or file) to be sorted resides on a disk. The term block refers to the unit ofdata that is read from or written to a disk at one time. A block generally consists of several records. For a disk, there are three factors contributing to read/write time:
a. Seek time: time taken to position the read/write heads to the correct cylinder. This will depend on the number of cylinders across which the heads have to move.

b. Latency time: time until the right sector of the track is under the read/write head.

c. Transmission time: time to transmit the block of data to/from the disk.

Example of External Sorting:2-way Merge Sort, k-way Merge Sort, Polyphase Merge sort etc. External Sorting with Disks: The most popular method for sorting on external storage devices is merge sort. This method consists of essentially two distinct phases. First Phase (Run Generation Phase): Segments of the input file are sorted using a good internal sort method. These sorted segments, known as runs, are written out onto external storage as they are generated. Second Phase (Merge Phase):The runs generated in phase one are merged together following the merge tree pattern , until only one run is left.2- way Merging/ Basic External Sorting Algorithm Assume unsorted data is on disk at start.

Let M=maximum number of records that can be sorted & sorted in internal memory at one time. **Algorithm:**

```
Repeat
1. Read M records into main memory & sort internally.
2. Write this sorted sub-list into disk. (This is one"run"). Until data is processed in to runs
Repeat
1. Merge two runs into one sorted run twice as long.
2. Write this single run back onto disk
Until all runs processed into runs twice as long
```

Merge runs again as often as needed until only one large run: The sorted list.

Pseudo Code for merging of M records:

```
open N input files and one output file (* initialize buffers *)
loop from i = 1 to N if end_of_file (file i)
then buffer[i] <- invalid_key else buffer[i] <- first record of file i;
(* merge *) stop <- false repeat
s <- index of smallest buffer element if buffer[s] = invalid_key
then stop <- true else write buffer[s] if end_of_file (file s)
then buffer[s] <- invalid_key
 else buffer[s] <- next record from file s until stop = true
close files
```

Measuring Performance:
Time Complexity: The size of a run and number of initial runs (nr) is dictated by the number of file blocks (b)and the available buffer space (nb
) • Nr = b/nb
• E.g b = 1024 blocks and nb = 5 blocks then 205 initial runs will be needed
The degree of merging (dm) is the number of runs that can be merged together in each pass.
• One buffer block is needed to hold one block from each run being merged
• One buffer block is needed for containing one block of merged result
• Dm is the smaller of (nb– 1) and nr
• Number of passes = logdm(nr)

Analysis This algorithm requires log(N/M) passes with initial run pass. Therefore, at each pass the Nrecords are processed and at last we will get a time complexity as O(N log(N/M).

Conclusion: Implemented external sorting