UNIT 10 HASHING

Structure

10.0 INTRODUCTION

Hashing is a key technique in information retrieval. Hashing transforms the input data into a small set of keys that can be efficiently stored and retrieved. Hashing provides constant time and highly efficient information retrieval capability irrespective of total search space.

10.1 OBJECTIVES

After going through this unit, you should be able to

- the drivers and motivations for hashing,
- various hashing techniques,
- understand index mapping
- collision avoiding techniques,
- understand load factor and rehashing

10.2 MAIN DRIVERS AND MOTIVATIONS FOR HASHING

As part of searching and information retrieval we use hashing for mainly below given reasons:

- Provide constant time data retrieval and insertion
- Manage the data related to large set of input keys efficiently
- Provide cost efficient hash key computations

10.3 INDEX MAPPING

In order to store and retrieve huge set of data, we can think of using a large sized array and store/retrieve the data from the large array. We can plan to use the data value as the key for the array. As the array indexing takes only O(1) time, the method guarantees a constant and fast performance.

Index mapping or trivial hashing method assumes a large sized array and the input keys as index for the array to retrieve the value.

Let's look at an example for this index mapping method. Let's design a large array based on the index mapping method. We use the array to store the name of the user and we plan to lookup the user details using their 4-digit employee id. As depicted in Figure 1, we create a large array to accommodate all the employee ids and use the employee id as an index to get the user details from the array.

Index mapping approach can be used when we know or when we can predict all the input keys. For instance, if we were to store the details for months of a year we know that we can have maximum of 12 months and hence we can design the hash table with size 12. Similarly, for input keys such as days of month, countries, states within a country where we can predict the maximum values, we can use the index mapping method.

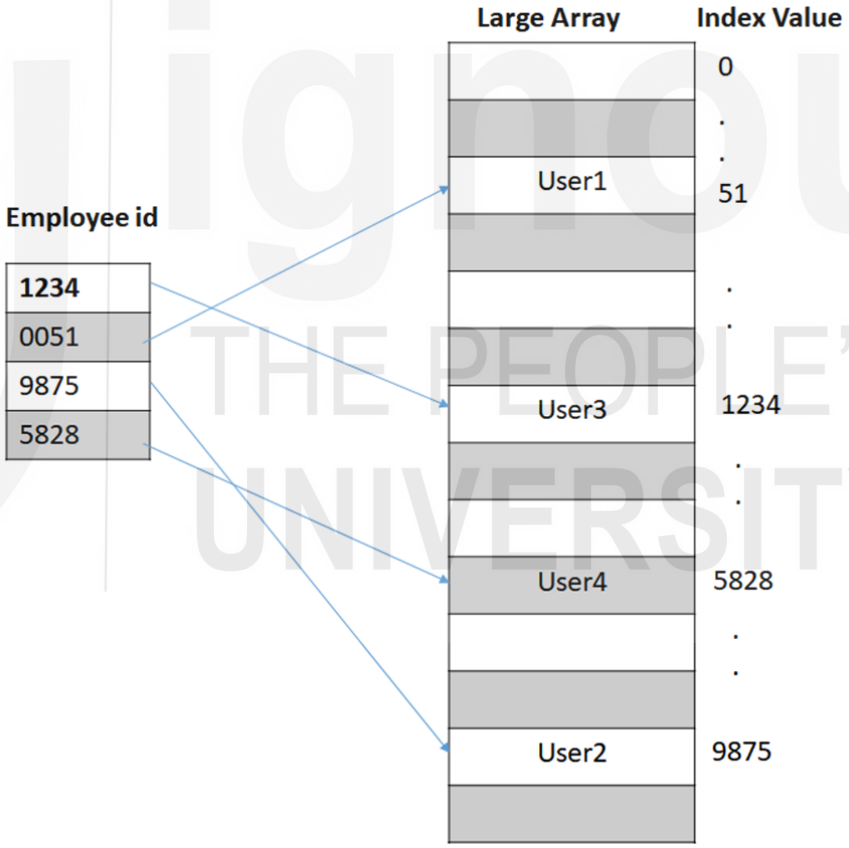

Figure 1 Index mapping approach

10.3.1 Challenges with Index Mapping

We can observe two main drawbacks with the index mapping approach. Firstly, this approach needs only non-negative integer values as the index keys and secondly as we can see from Figure 1, the array size poses performance and scalability challenges. The array has to be sized to accommodate the largest key value and the values are unevenly distributed leading to wastage of space.

To overcome the two limitations mentioned above, we need a conversion function that converts all data types (string, image, alphanumeric etc.) into a non-negative integer

value. Additionally, the conversion function also maps the input values to smaller set of keys that can be used as index for a smaller sized array. We call the conversion function as "hash function".

Let's use a hash function that takes the input values (employee id) and converts the value into a key that forms of the index for the hash table storing the employee details. We have used the below given hash function:

$h(x) = x \bmod 10$

where h(x) is the hash value, x is the input key. The mod operation provides the remainder value for the key. As a result of this hash function we now have the hash table with optimal size as depicted in Figure 2.

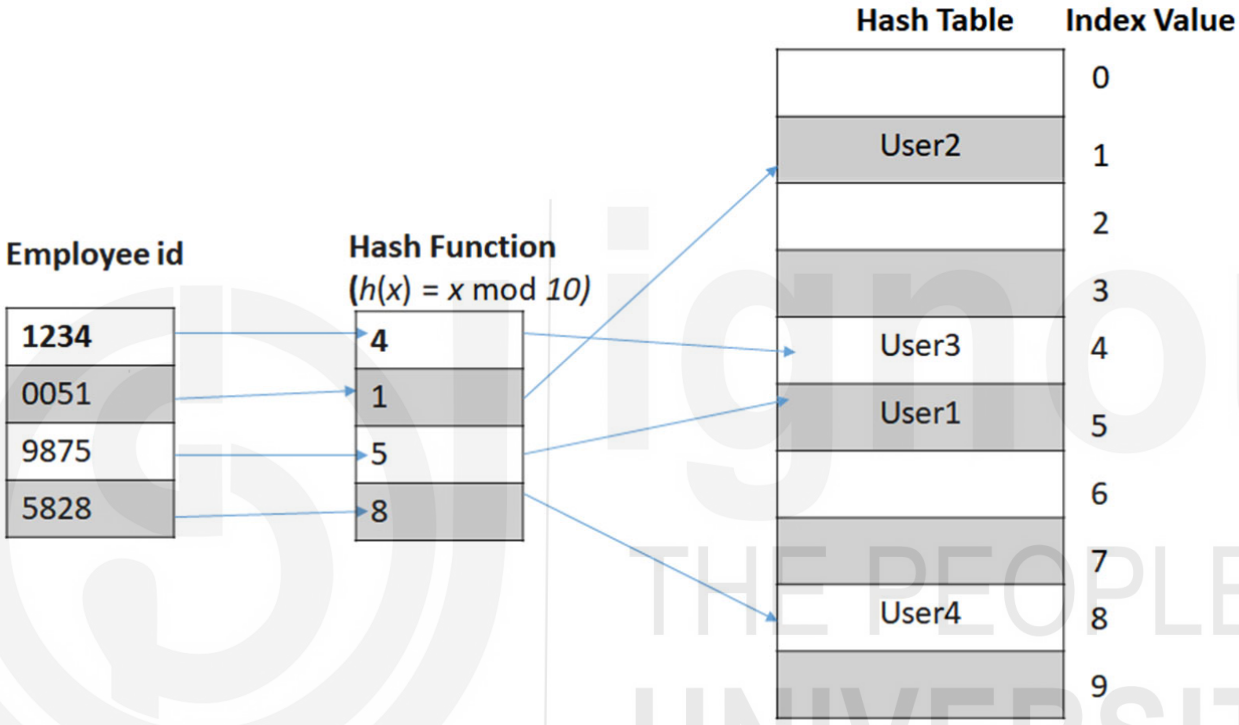

Figure 2 Hash Function and Hash Table

As we can see from Figure 2, instead of creating a huge hash table of size 9875, we have now managed to store the elements within an array of size 10. The hash function has converted the input into smaller set of keys that are used as index for the hash table.

Let us re-look at two challenges we saw in our earlier direct access method. The examples we discussed in Figure 2 use integer values as input keys. If we use non-integer values such as images or strings, we need to convert it first into a non-negative integer value. For instance, using the corresponding ascii values for each of its character, we can get a numeric value for a string. We can then use the hash function to create a fingerprint for the numeric value to store the corresponding details in the right sized hash table.

10.3.2 Hash Function

As we saw in the previous example, a hash function reduces a large non-negative integer into a smaller hash value which is used as an index into the hash table for searching the value.
The efficiency of a hash function is determined by following characteristics:

- **Computing efficiency** – The hash function should compute the hash value quickly and efficiently even for large key values
- **Uniform distribution** – The hash function should be able to distribute the keys evenly in the hash table.
- **Deterministic** – The hash function should consistently create the same key for a given value.
- **Minimal Collisions** – The hash function should minimize the key collisions in the hash table.

10.3.3 Simple Hash Example

Let us look at implementation of main functions using a modulo-based hash function.

We assume no collisions for this sample code in Java.

```
// the function returns the stored value for the input key
public V getValue(int inputKey) {
        int hashvalue = getHashValue(inputKey);
        return this.hashArray[hashvalue].value;
}

//the function adds the value for the given input key
public void addValue(int inputKey V inputValue) {
        int hashvalue = getHashValue(inputKey);
        this.hashArray[hashvalue].value = inputValue ;
}

//the function computes the hash value using mod logic
public int getHashValue(int inputKey) {
        return inputKey % this.hashArray.length;
}

//the function removes the input value from the hashArray
public void removeValue(int inputKey) {
        int hashvalue = getHashValue(inputKey);
        this.hashArray[hashvalue].value = null;
}
```

10.4 COLLISION RESOLUTION

For large set of input keys, we might end up having same hash value for two different input values. For instance, let us consider our simple hash function $h(x) = x \bmod 10$

As the hash function provides the remainder value, if we have two input keys 24 and 4874, the hash value will be 4. These cases cause collision as both the input keys 24 and 4874 compete for same slot in the hash table.

We discuss three key collision resolution techniques in subsequent sections.

10.4.1 Separate Chaining

In separate chaining method, we chain the values to the same spot in the hash table. We use a data structure like linked list that can store multiple values.

The example Figure 3 depicts collision handling using the modulo based hash function. The input values 0051 and 821 result in the same hash value of 1. In the hash table we chain the values for user 2 and user 5 for the same slot.

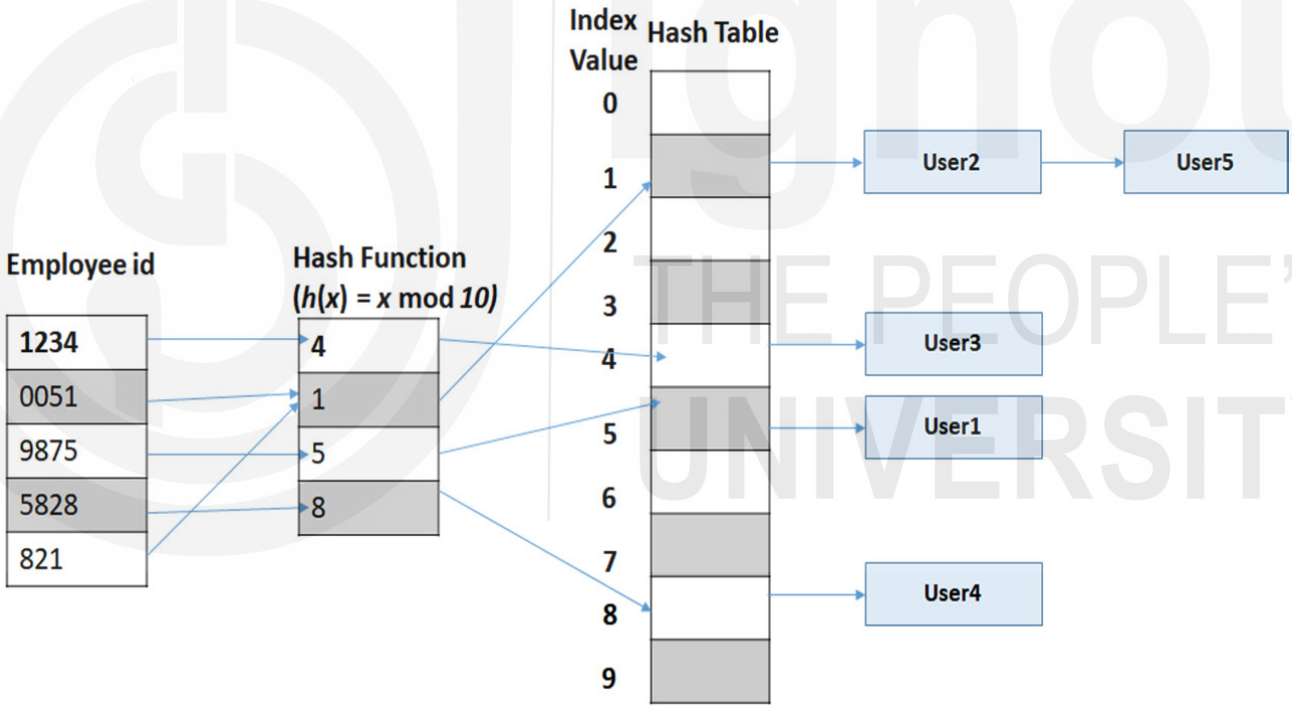

Figure 3 Separate Chaining

We are chaining the data values user2 and user5 for the slot 0 in the hash table.

Insert Operation

Given below are the steps for the insert operation using separate chaining method to insert a key k:

1. Compute the hash value of the key k using the hash function.
2. Use the computed hash value as the index in the hash table to find the slot for the key k.
3. Add the key k to the linked list at the slot.

Search operation

Given below are the steps for the search operation for key k:

1. Compute the hash value of the key k using the hash function.
2. Use the computed hash value as the index in the hash table to find the slot for searching the key k.
3. Check for all elements in the linked list at the slot for a match with key k.

☞ Check Your Progress – 1

1. _____ provides direct mapping of keys to the hash table index.
2. The two key challenges with index mapping are _____
3. The main characteristics of a hash function are _____.
4. The time complexity of index mapping is _____.
5. _____ data structure can be used to chain multiple values to the same spot in the hash table.
6. Hash value should always be non-negative. True/False

10.4.2 Open Addressing

In the open addressing collision addressing strategy, we search for the next available slot in the hash table when the natural slot is not available. We search for the open or unused locations in the hash table.

There are mainly two variants of open addressing – linear probing and quadratic probing. In the linear probing we sequentially iterate to the next available spot.

We define the linear probe by the following equation for i^{th} iteration:

$h(x,i) = (h(x)+i) \bmod m$ *where m is the hash table size*

We need to accordingly change the other hash functions such as getHashValue, putHashValue and deleteHashValue.

- For getHashValue we need to start from the initial slot (hashTable[h(x)]) we need to check the subsequent slots till we find the matching key.
- For putHashValue we need to start with the initial slot (hashTable[h(x)]) till we find an empty slot or till the hashtable is full.
- For deleteHashValue we can place null into the slot or use an availability marker (occupied, available).

The linear probing leads to a situation known as "primary clustering" wherein the consecutive slots form "cluster" of keys in the hash table. As the cluster size grows, it impacts the efficiency of the probing for placing the next key.

Quadratic probing makes larger jumps to avoid the primary clustering. We define the quadratic probe by the following equation for i^{th} iteration:

$h(x,i) = (h(x)+i^2) \bmod m$

As we can see in the equation, for every iteration the quadratic probing makes larger jumps. In Quadratic probing we encounter "secondary clustering" problem.

Let us look at an example for linear probing to avoid the collision. Let us consider the following set of keys [56, 1072, 97, 84, 60] and the hash table size of 5. When we apply mod based hash function and start placing the keys in the appropriate slots we get the placement as depicted in Figure 4.

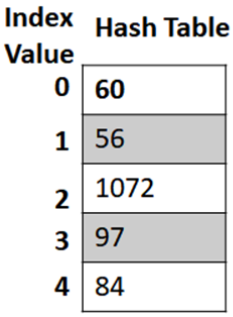

Figure 4 Linear Probing

The value 56 goes to position 1 due to the mod value of (56 mod 5) operation. Similarly, 1072 assumes position 2. However, when we try to place the next element 97 we end up with a collision at slot 2. So, we find the next empty slot at slot 3 and place 97 there. Rest of the elements 84 and 60 go to the positions 4 and 0 respectively based on their mod values.

10.4.3 Double Hashing

We can avoid the challenges with primary clustering and secondary clustering using the double hashing strategy. Double hashing uses a second hash function to resolve the collisions. The second hash function is different from the primary hash function and uses the key to yield non-zero value.

The first hash function in the double hashing finds the initial slot for the key and the second hash function determines the size of jumps for the probe. The i^{th} probe is defined as follows

$h(x,i) = (h_1(x)+ i * h_2(x)) \bmod m$ where m is the hash table size

Let us look at an example for the double hashing to understand it better. Let us consider a hash table of size 5 and we use the below given hash functions:
H1(x) = x mod 5
H2(x) = x mod 7

Let us try to insert two elements 60 and 80 into the hash table. We can place the first element 60 at slot 0 based on the hash function. When we try to insert the second element 80, we face a collision at slot 0. For the first iteration we apply the double hashing as follows:
H(80,1) = (0+1*3) mod 5 = 3

Hence, we now place the element 80 in slot 3 to avoid collision as depicted in figure 5.

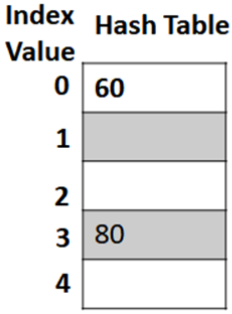

Figure 5 Double Hashing Example

10.5 COMPARISON OF COLLISION RESOLUTION METHODS

Comparison of linear probing, quadratic probing and double hashing is given below:

Collision Resolution Technique	Separate Chaining	Open Addressing - Linear Probing	Open Addressing - Quadratic Probing	Double Hashing
Primary clustering	No	Yes	No	No
Secondary clustering	No	No	Yes	No
Key storage in Hash table	Inside & Outside Hashtable	Inside Hash table	Inside Hash table	Inside Hash table

10.6 LOAD FACTOR AND REHASHING

The hash table provides constant time complexity for operations such as retrieval, insertion and deletion with lesser keys. As the key size grows, we run out of vacant spots in the hash table leading to collision that impacts the time complexity. When the collision happens, we need to re-adjust the hash table size so that we can accommodate additional keys. Load factor defines the threshold when we should re-size the hash table to main the constant time complexity.

Load factor is the ratio of the elements in the hash table to the total size of the hash table. We define load factor as follows:

Load factor = (Number of keys stored in the hash table)/Total size of the hash table.

In open addressing as all the keys are stored within the hash table, the load factor is <=1. In separate chaining method as the keys can be stored outside the hash table, there is a possibility of load factor exceeding the value of 1.

If the load factor is 0.75 then as soon as the hash table reaches 75% of its size, we increase its capacity. For instance, lets consider the hash table of size 10 with load factor of 0.75. We can insert seven hash keys into this hash table without triggering the re-size. As soon as we add the eighth key, the load factor becomes 0.80 that

exceeds the configured threshold triggering the hash table resize. We normally double the hash table size during the resize operation.

10.6.1 Rehashing

when the load factor exceeds the configured value, we increase the size of hash table. Once we do it we should also re-compute the hash values for the existing keys as the size of the hash table has changed. This process is called "rehashing". Rehashing is a costly exercise especially if the key size is huge. Hence it is necessary to carefully select the optimal initial size of the hash table to start with.

Given below are the high-level steps for rehashing:

1. For each new key insert into the hash table, compute the load factor
2. If the load factor exceeds the pre-defined value then increase the hash table size (normally we double the hash table size)
3. Recompute the hash value (rehash) for each of the existing elements in the hash table.

Let us look at the rehashing with an example. We have a hash table of size 4 with load factor of 0.60. Let's start by inserting these elements – 30, 31 and 32. We can insert 30 at slot 2 and 31 at slot 3 and 32 at slot 0. Insertion of 32 triggers the hash table resize as the load factor has breached the threshold of 0.60. As a result, we double the hash table size to 8.

With the new hash table size, we need to recalculate the hash values of the already inserted keys. Key 30 will now be placed in slot 6, key 31 will be placed in slot 7 and key 32 in slot 0.

☞ Check Your Progress – 2

1. The main techniques of open addressing are _____
2. Load factor triggers _____
3. Linear probing leads to _____ clustering
4. _____ probing make large jumps leading to secondary clustering
5. Second hash function in double hashing can result in 0. True/False
6. _____ should be done for the existing keys of the hash table post resizing.

10.7 SUMMARY

In this unit, we started discussing the main motivations for the hashing. Hashing allows us to store and retrieve large data efficiently.
Index mapping uses the input values as direct index into the hash table. Index mapping requires huge hash table size leading to inefficiencies. When we handle large size input values we encounter collision where multiple input values compete for the same spot in the hash table. The main collision resolution techniques are separate chaining and open addressing. In separate chaining we chain the values that get mapped to a spot. We use linear probing and quadratic probing as part of open addressing technique to find the next available spot. We use two hash functions as part of double hashing. Load factor determines the trigger for the hash table resizing and once the hash table is resized, we re-compute the hash values of the existing keys using rehashing.

10.8 SOLUTIONS/ANSWERS

☞ Check Your Progress – 1

1. Index mapping
2. handling non integer keys and large hash table size
3. computing efficiency, uniform distribution, deterministic and minimal collisions
4. O(1)
5. Linked List
6. True

☞ Check Your Progress – 2

1. linear probing, quadratic probing and double hashing
2. resizing of hash table
3. primary
4. quadratic
5. False
6. Rehashing

10.9 FURTHER READINGS

Horowitz, Ellis, Sartaj Sahni, and Susan Anderson-Freed. *Fundamentals of data structures*. Vol. 20. Potomac, MD: Computer science press, 1976.

Cormen, Thomas H., et al. *Introduction to algorithms*. MIT press, 2022.

Lafore, Robert. *Data structures and algorithms in Java*. Sams publishing, 2017.

Karumanchi, Narasimha. *Data structures and algorithms made easy: data structure and algorithmic puzzles*. Narasimha Karumanchi, 2011.

West, Douglas Brent. *Introduction to graph theory*. Vol. 2. Upper Saddle River: Prentice hall, 2001.

https://en.wikipedia.org/wiki/Hash_function#Trivial_hash_function

https://en.wikibooks.org/wiki/A-level_Computing/AQA/Paper_1/Fundamentals_of_data_structures/Hash_tables_and_hashing

https://ieeexplore.ieee.org/book/8039591

UNIT 11 ADVANCED DATA STRUCTURES

Structure

11.0 INTRODUCTION

In this unit we discuss various kinds of trie data structure. For efficient searching the data we use the trie. We shall closely look at various operations such as search, insert, delete, successor and predecessor on the trie data structure. We also look at optimizing the time complexity in the trie data structure.

11.1 OBJECTIVES

After going through this unit, you should be able to

- understand key concepts of scapegoat tries,
- understand tries and binary tries,
- understand key concepts of x-fast tries and y-fast tries

11.2 BRIEF INTRODUCTION TO BINARY SEARCH TREE (BST)

In this unit we discuss the Tries that are type of binary search trees. So, we briefly discuss the binary search tree. Binary search tree is a sorted tree that stores the keys in ordered way. For any given node in the binary search tree, the keys in left side is lesser than keys in the right side.

As the binary search tree is sorted, the data structure is useful for searching and sorting the data. To search for a key in the BST, we start from root node and if the search key is less than root node we check for left node else we check the right node. We recursively search for the key till we exhaust the nodes. To find the successor of the key k, we find the smallest key that is greater than k and for finding the predecessor for key k, we find the largest key that is smaller than k. The complexity of BST for search, insert and delete is O(n) where n is the number of nodes in the tree.

11.3 SCAPEGOAT TRIES

The scapegoat tries are self-balancing binary search tree that provides the worst-case performance of O(logn) for search and O(log n) amortized time for insert and delete operation.

We say that the binary search tree is balanced when the weight of the left nodes matches the weight of the right nodes. We define the α-weight-balanced tree by the below-given equation:

$$\text{Size}(\text{left}[x]) \leq \alpha * \text{size}(x) \text{ and}$$
$$\text{Size}(\text{right}[x]) \leq \alpha * \text{size}(x)$$

Where

- $0.5 \leq \alpha < 1$
- X is the node of the tree and left(x) is the left child of x and right(x) is the right child of x.
- The size (x) is defined as number of keys stored in the sub-tree of x including the key stored at x.

Figure 1 provides the scapegoat tree that is α-weight-balanced

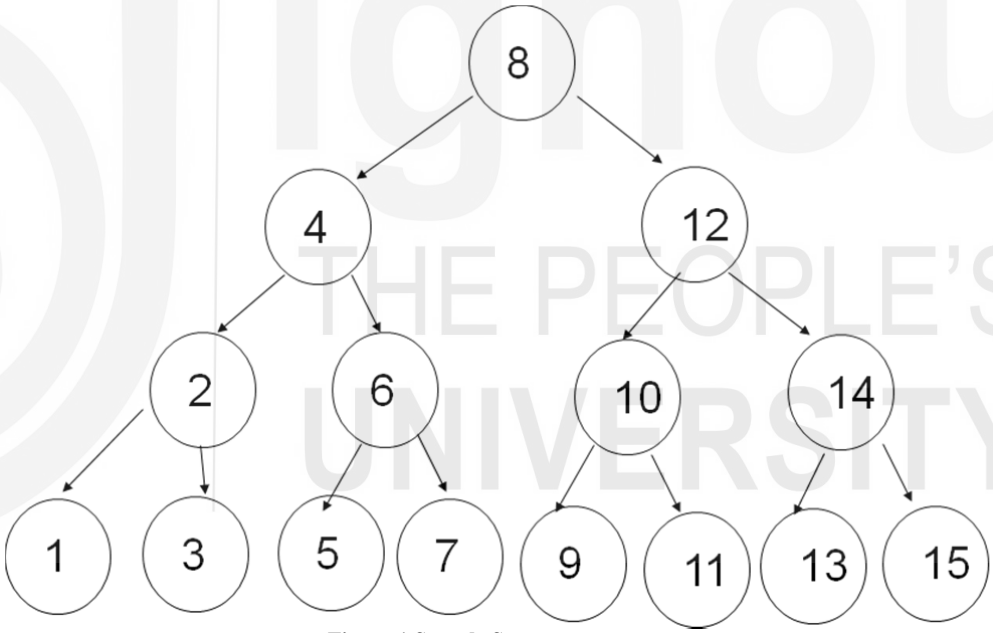

Figure 1 Sample Scapegoat tree

11.3.1 Operations on Scapegoat tries

In this section we define various operations on Scapegoat tries.

Search Operation

The search operation is similar to that of the binary search tree.

Insert Operation

For inserting a new node into the tree, we will find a scapegoat node. Given below are the steps to find the scape goat node for inserting a new value Y into the tree:

1. Create the new node x and insert in natural way

2. Walk up the tree hierarchy to find the scapegoat node that is not α-weight-balanced. If x_i is the node that is not α-weight-balanced, we compute the size of as follows:

   Size $(X_{i+1}) =$ Size (X_i) + size (brother(X_i)) +1

   Where brother(x_i) is the other child of X_i's parent or null if another child does not exist

Let us look at an insert example where we try to insert element 4.5 into a tree as depicted in Figure 2.

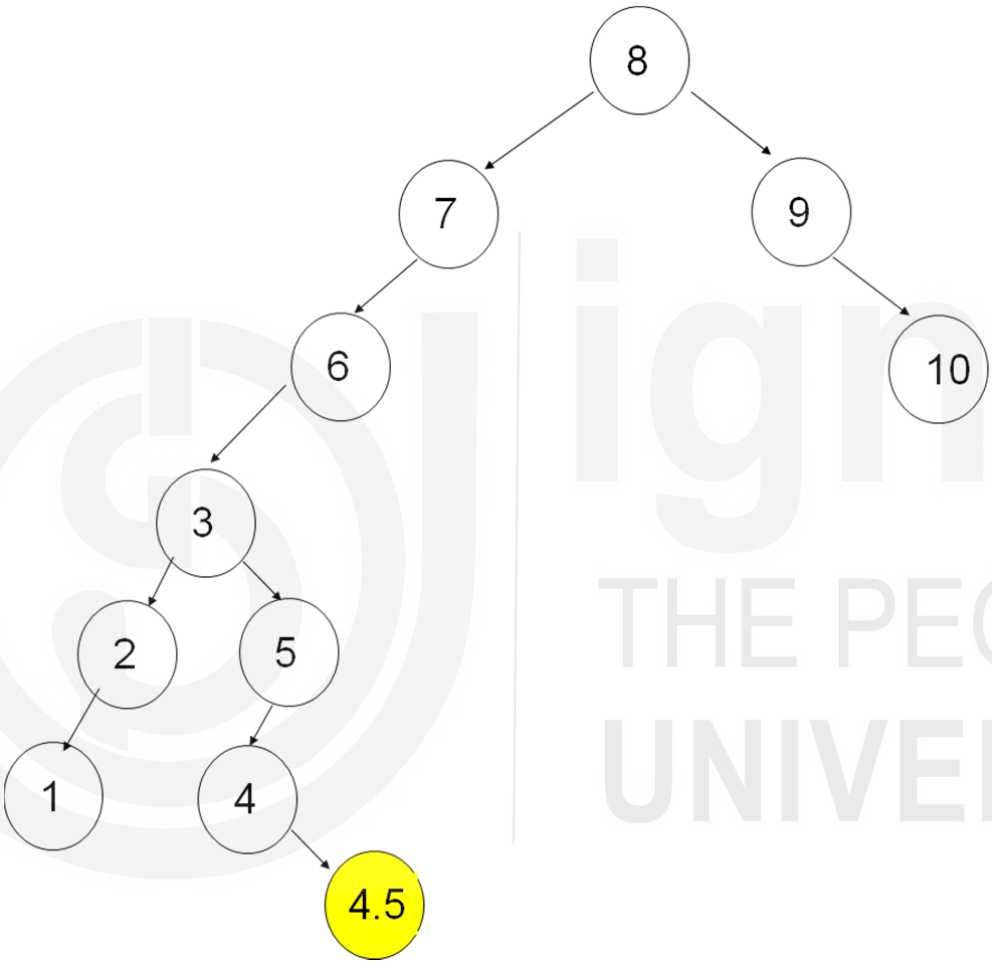

Figure 2 Inserting new node to Scapegoat tree

If we traverse the tree to find the scapegoat node that is not load balanced we reach, node with key 6. So, we balance the tree at node 6. The balanced tree is depicted in Figure 3.

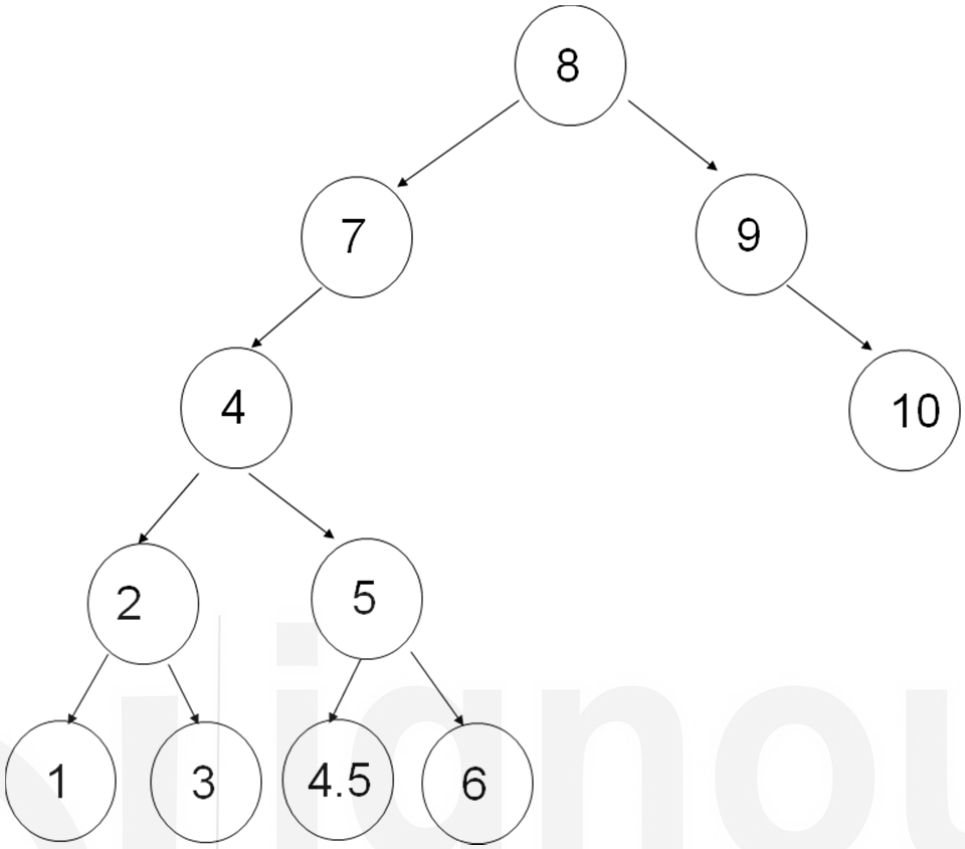

Figure 3 New scapegoat tree rebalanced after inserting

Delete operation

Given below are the steps for deleting the node x:

1. If the node x is a leaf node, we can delete it.
2. If the node x has single left child or single right child, we replace x with its only child.
3. If the node x has both left child and right child, we find the lowest value node in the right sub-tree and replace x with it.

☞ Check Your Progress – 1

1. The scapegoat tries are _____ binary search tree.
2. For inserting the node we find the _____ node
3. The worst-case performance of scapegoat search operation is _____
4. In the binary search tree, the left node value is _____ than right node value
5. The complexity of BST for search, insert and delete is ____

11.4 TRIES

Tries are tree-based data structure that are used to search for string-based keys in a given set. We use individual characters of a string as edges that connect the nodes of the tree. Given below are the main characteristics of tries:

1. The child nodes of a node are associated with a common prefix.
2. The key is denoted by the position of the node
3. String search can be done by using the depth-first search.

There are many real-world use cases that involve integer-based information retrieval. For instance, id number-based information retrieval, account number-based search all need integer-based lookup. For such use cases we can use bitwise tries for faster information retrieval. Tries are also used in web search engines, for text search use cases.

The average time complexity for search, insert and delete operations for trie data structure is O(n).

Figure 4 depicts a sample Trie

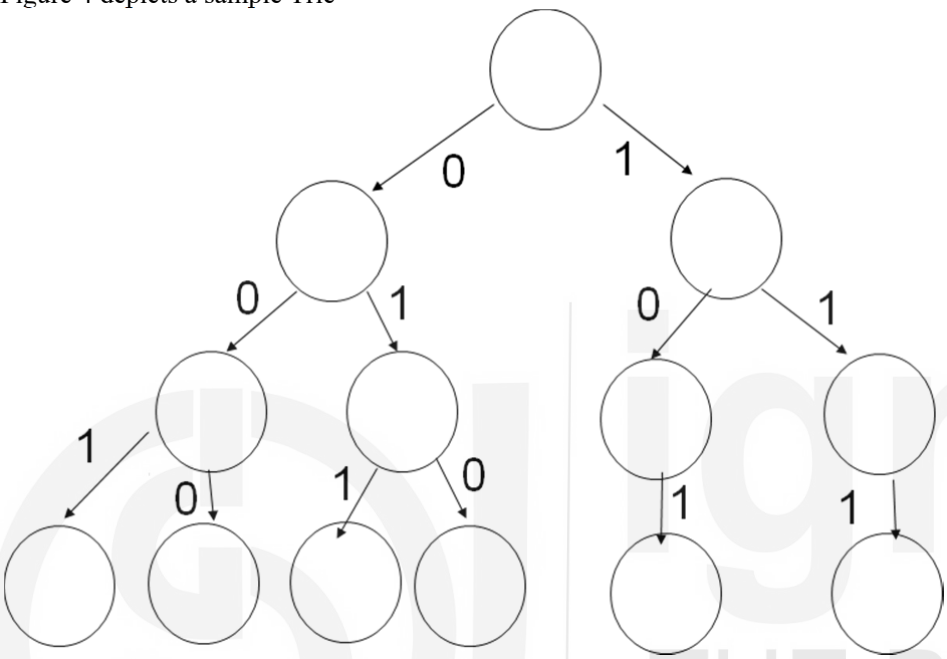

Figure 4 Sample Trie

11.4.1 Binary tries

The binary tries use bits as keys that can be used to represent integer. We can use the bit representation of individual characters of a string which can then be used to traverse the binary Tries. In the binary trie only the leaf node holds the key. We compare the bits of the tree to find the key.

Search operation in Binary trie

Given below are steps for searching for a key in binary trie:

1. Start the search with root node
2. Initialize i=0 and compare the i^{th} bit of key.
3. If the key bit value is 0 check the left node, else search the right node.
4. If the current node is leaf and if all the bits of the key match the path, return success else return failure
5. If the current node is not leaf increment the counter i and repeat the steps 3 through 5.

Insert operation in Binary Trie

We insert the keys based on their bit values. The initial 0 bit will go to left and 1 bit to the right. We follow the process till all the bits in all keys are inserted.
Given below are the steps for insert operation:

1. Search for key k in the tree.

2. When the search failure happens on a non-leaf node, we simply link the non-leaf node to the new node
3. If the search failure happens on leaf node, we create a new internal node to accommodate the new key and existing key.

Delete operation in Binary Trie

Given below are the steps for deleting the node x:

1. Search for node x in the tree. When the leaf node x is reached, it is deleted.
2. We traverse back in the search path for x deleting the nodes until we reach a node u that has child not in the search path for x.
3. We traverse back to the root node deleting the descendent pointer to node x.

Figure 5 is an example of inserting the following keys:

S 01001
M 11001
U 00110

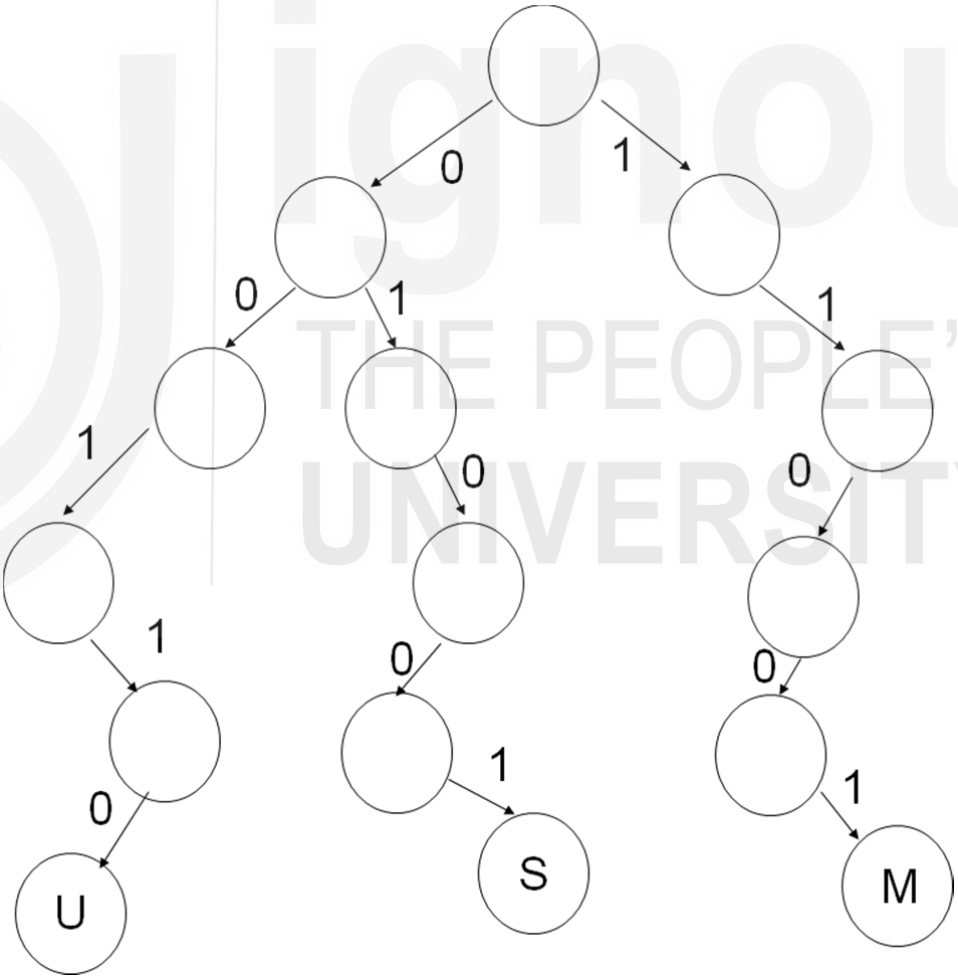

Figure 5 Binary Trie with keys inserted

As we can notice in Figure 5, the keys in left subtree are smaller when compared to keys in the right subtree for any given node.

11.5 X-FAST TRIES

X-fast tries are bitwise tries that stores the integers from a bounded domain. The complexity of successor operation is O(log log M) where M is the domain's maximum value; find operation is O(1); insert and delete operation is O(log M) where M is the domain's maximum value. The subtree uses a common prefix from its parent node. The leaf node represents values from 0 to M-1 where M is the domain's maximum value.

The left child adds 0 to the prefix and the right child adds 1 to the prefix.

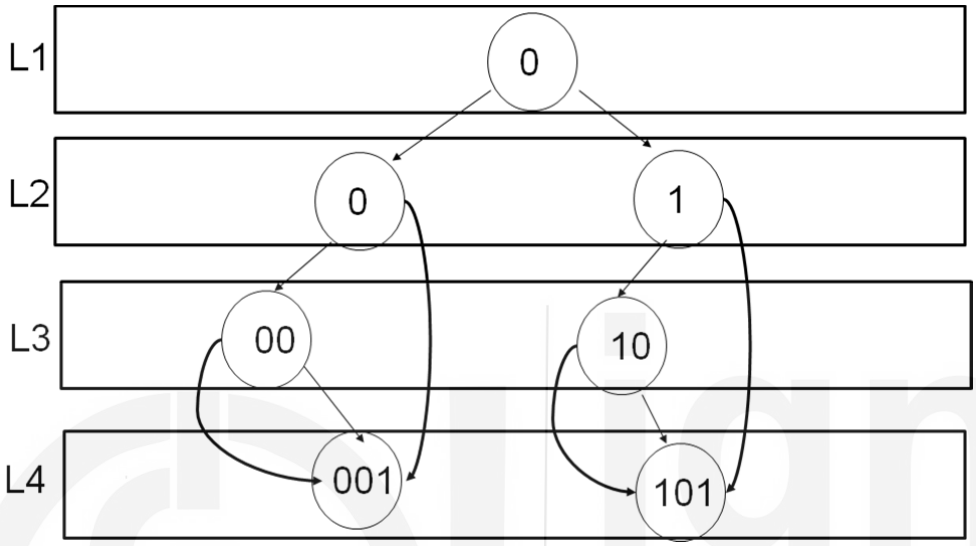

Figure 6 X-fast trie

We have depicted a four-level x-fast trie in Figure 5. Each level of the x-fast trie is implemented as hash table. The internal nodes are represented only if their subtree has one or more leaf nodes. If the internal node does not have a left child, then the left pointer points to the smallest leaf node in the right subtree which is called descendant pointer.

The tree depicted in Figure 5 has domain size of 8 from 0 to 7. We store the nodes 0 (001) and 5 (101).

11.5.1 Operations on X-fast tries

In this section we shall look at the main operations on X-fast tries

Find operation

For finding value of key k, we can look up the hash table indexing the leaf nodes. This is a constant time operation.

Successor and Predecessor

To find the successor of key k, we follow these steps:

1. Using the prefix of key, we perform the binary search by querying the hash table to find the lowest ancestor of k.
2. Once we get a match we use the descendant pointer of k
3. For the predecessor use the previous leaf and for successor use the next leaf.

Insert Operation

To insert key k with value v, we follow these steps:

1. Find the predecessor and success or k
2. Create a new leaf node between predecessor and successor of K and point it to v.
3. Start traversing the tree from the root to leaf and create the missing internal nodes and adding the descendant pointers wherever necessary.

Delete Operation

To delete the key k, we follow these steps:

1. Remove the k from the hash table indexing the leaf nodes.
2. Delete the k node and link its predecessor and successor.
3. Traverse the tree from the root and delete the internal nodes that have k as only nodes within its subtrees and update the descendant pointers appropriately.

11.6 Y-FAST TRIES

Y-fast tries are bitwise tries that store integers from bounded context. They are the improved version of X-fast tries that optimize the memory. Similar to the X-fast tries, all subtrees will have a common prefix of its parent.

The complexity of successor operation is O(log log M) where M is the domain's maximum value; find operation is O(1); insert and delete operation is O(log M) where M is the domain's maximum value.

The Y-fast trie consists of two data structures – X-fast trie at the top and the balanced binary search tree in the bottom. We have depicted the Y-fast Trie in Figure 6.

We chose a representative r from the binary search tree and store it in the X-fast trie. The r value should be smaller than its successor in the X-fast trie.
One of the main differences between X-fast trie and Y-fast trie is that in the Y-fast trie, the X-fast trie operates on the representatives of the binary search tree. The leaf node point to the binary search tree.

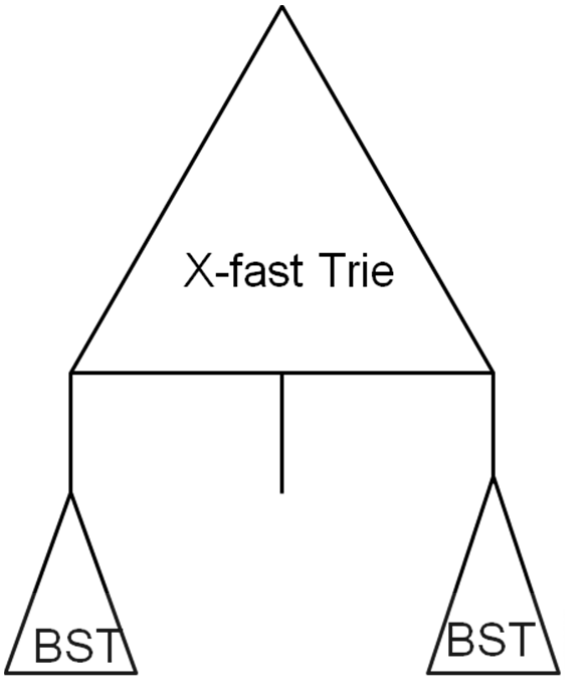

Figure 7 Y-fast Trie

11.6.1 Operations on Y-fast tries

In this section we shall look at the main operations on Y-fast tries.

Find operation

Given below are the steps to find the key k:

1. Find the successor and predecessor of k in the X-fast trie. We can do this by finding the smallest representative r that is greater than k and largest representative r^1 that is smaller than k in the X-fast trie.
2. We carry out the binary search for k in the binary search trees represented by r and r^1

Successor and Predecessor

1. To find successor of k, search for smallest representative r in the X-fast trie.
2. Find the predecessor of r in X-fast trie.
3. Search for the binary trees pointed by these two representatives for getting the successor of k.

Insert Operation

To insert the key k, given below are the steps:

1. Insert the key to the tree that has the successor of k.
2. Point the node to v
3. If the tree contains more than 2 log U elements (where U is the maximum value of the domain), then remove it representative from the X-fast trie. Split the binary search tree into two and add the representatives from each of the binary search tree into the X-fast trie.

Delete operation

Given below are the steps for deleting the key k:

1. Delete the key k from the tree

2. If the elements in the tree drop below (log U)/4 (where U is the maximum value of the domain), merge the tree with its predecessor or successor. Remove the representatives of the merged trees from the X-fast trie.
3. Add the representative of the new tree into X-fast trie.
4. If the newly-formed tree contains more than 2 log U elements (where U is the maximum value of the domain), then remove it representative from the X-fast trie. Split the binary search tree into two and add the representatives from each of the binary search tree into the X-fast trie.

☞ Check Your Progress – 2

1. The child nodes of a node in a Trie are associated with a _____
2. The average time complexity for search, insert and delete operations for trie data structure is _____
3. The binary tries use ____ as keys that can be used to represent integer
4. X-fast tries are bitwise tries that stores the integers from a _____
5. Each level of the x-fast trie is implemented as _____
6. Y-fast tries are the improved version of _____

11.7 SUMMARY

In this unit, we started discussing the key aspects of binary search tree. Binary search tree is a sorted tree that stores the keys in ordered way. The scapegoat tries are self-balancing binary search tree. When the weight of the left nodes matches the weight of the right nodes then we say that the binary search tree is balanced. To insert the data we find the scapegoat node in scapegoat trees and balance the tree around it. Tries are tree-based data structure that are used to search for string-based keys in a given set. The average time complexity for search, insert and delete operations for trie data structure is O(n). The binary tries use bits as keys that can be used to represent integer. X-fast tries are bitwise tries that stores the integers from a bounded domain.

11.8 SOLUTIONS/ANSWERS

☞ Check Your Progress – 1

1. Self-balancing
2. scapegoat
3. O(logn)
4. Lesser
5. O(n)

☞ Check Your Progress – 2

1. common prefix
2. O(n)
3. bits
4. bounded domain
5. hash table

6. X-fast tries

11.9 FURTHER READINGS

Horowitz, Ellis, Sartaj Sahni, and Susan Anderson-Freed. *Fundamentals of data structures*. Vol. 20. Potomac, MD: Computer science press, 1976.

Cormen, Thomas H., et al. *Introduction to algorithms*. MIT press, 2022.

Lafore, Robert. *Data structures and algorithms in Java*. Sams publishing, 2017.

Karumanchi, Narasimha. *Data structures and algorithms made easy: data structure and algorithmic puzzles*. Narasimha Karumanchi, 2011.

West, Douglas Brent. *Introduction to graph theory*. Vol. 2. Upper Saddle River: Prentice hall, 2001.

UNIT 12 FILE STRUCTURES

Structure

12.0 INTRODUCTION

The data is stored and organized through file structure. Software applications and platforms store the data in files. Operating systems manage critical configuration through file and even databases also store the data in files such as transaction logs. Files essentially serves as the key building blocks for data organization. We can enforce governance such as data integrity checks, security rules, access controls, sharing controls for files.
In this unit we discuss various aspects of file such as file organization, file types and related concepts.

12.1 OBJECTIVES

After going through this unit, you should be able to

- understand key concepts of files,
- understand the file organization concepts,
- understand sequential files and direct file organization,
- Understand indexed file organization and indexed sequential files

12.2 BASIC CONCEPTS OF FILE

A file is a collection of data records. The key operations of the file are as follows:

- File creation when the file gets created initially.
- File update when we update the file with new data records or modify existing data records.

- File deletion when we delete the file
- File merging that combines the content of multiple files into a single file.
- File searching that involves searching for a specific data record in a file.

A data record in a file encapsulates the data of a single entity. For instance, an employee data record captures details of a single employee such as employee id, employee name, employee address, employee phone number and employee date of join.

Each data record is uniquely identified by a key. For instance, in the employee data record, employee id is the key.

The table 1 provides the employee data records in an organization:

Employee Number	Employee Name	Employee Address	Department	Date of Join
45278	Employee 1	Kanpur	Sales	01-Jan-2020
167352	Employee 2	Mysore	Development	01-Mar-2020

12.3 FILE ORGANIZATION

File organization defines the way the data record is stored and retrieved from the files. File organization provides the data record relationship specifying how the data records are mapped to the disk blocks. We can either store fixed length record in the files or use flexible file structure to support data records of varying length.

There are multiple ways of file organization such as sequential, direct, indexed sequential and others. We can select the file organization type based on the use case, information retrieval performance, ease of information management and others.

Given below are the most commonly used file organizations:

- Sequential files – The data records are stored in a pre-defined order. For instance, the data records are sequenced based on the key values. In this file organization type we need to sequentially access the record.
- Relative files – The data record is stored at a fixed location that can be accessed using a static key value. As we can access the data record using a fixed value, we can retrieve or insert the data record randomly in this file organization type.
- Direct files – Similar to the relative file that supports non-integer key value.
- Indexed sequential files – In this file organization we add an index to provide the random access to the sequential files.
- Indexed files – The data records are indexed to improve the performance of the data retrieval.

Figure 1 provides various file organizations that are most commonly used.

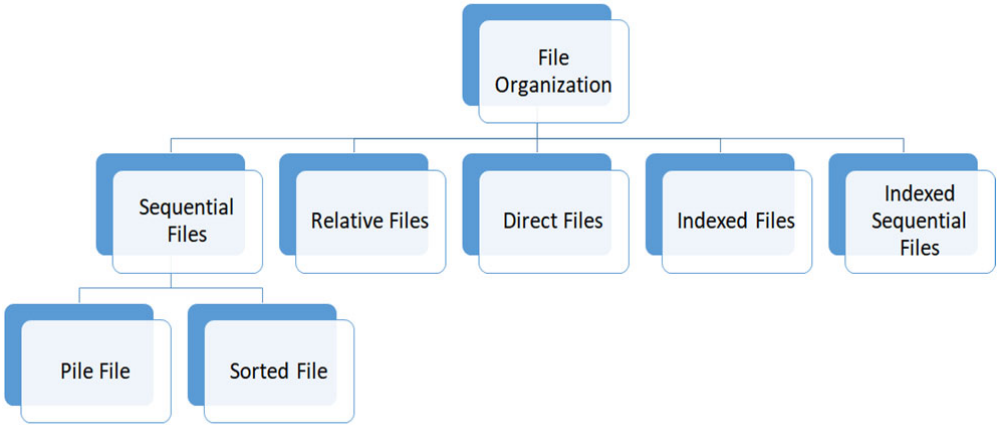

Figure 1 Types of File organizations

12.3.1 Key drivers for file organization

Given below are the main drivers for the file organization:

- File organization helps in optimal design of file records so that they can be retrieved as quickly as possible.
- Improve the performance of file operations such as insert, update and delete.
- Ensure the data consistency and avoid duplicates during file operations.
- Provide optimal storage for managing the files.

12.4 SEQUENTIAL FILES

We write the data records in a specific sequence in sequential files. The data is also accessed in the order it is written to the physical device. We usually order the data records based on the key value. In Sequential files the reading happens from the beginning of the file and write happens at the end of the file.

The legacy storage systems such as tape storage use sequential files.

12.4.1 Operations on sequential files

We discuss the main operations on the sequential files:

Insert operation

As the sequential files are organized based on their keys, when we insert we should find the exact position of the new key value and insert the key over there.

If there are multiple insertions, we can batch them to optimize the performance optimization. We can collect all the new inserts into transaction log and then sort the new keys in order in the transaction log. We can then merge the entries in the transaction log into the main file in batch mode.

Delete Operation

In the delete operation, we need to reclaim the space from the deleted record. Here as well we can use the concept of transaction log for performance optimization. We can add the delete marker for the key position in the transaction log and later we can merge the changes to the main file in batch mode. During merge operation, we drop the data records in the main file based on the marker.

Update Operation

For performing the update operation, we need to add the delete marker for the corresponding key and then insert the new data record for the corresponding key in the transaction log. Like always we merge the transaction log to the main file

Retrieval Operation

We read the data record based on its key. For stale data reads, we can directly read the data record for the key from the main file. To maintain read consistency, we need to check If there are any pending updates in the transaction log and if so we need to merge the changes (deletion/insertion).

12.4.2 Types of sequential files

There are mainly two types of sequential files – Pile files and sorted files.

Pile Files

In Pile files, the data records are inserted in the order of their arrival. For instance, let's consider the below given incoming Data records –

DR1, DR3, DR5, DR2

The data records are inserted as given in Figure 2:

Figure 2 Pile File

For searching the records, the file is searched from the beginning till the desired record is found. For deleting a record, we search for the record and add a deletion marker for the found record.
For inserting a new record, it will be done at the end of the file. The new record DR4 is to be inserted as shown in Figure 3

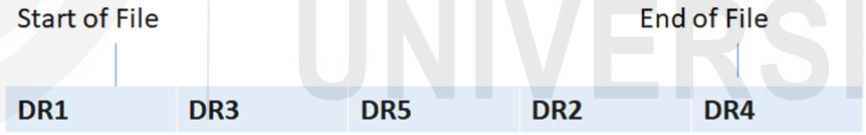

Figure 3 Pile File Insertion

Sort Files

In case of sort files, the data records are sorted based on the key and are inserted into the file.

For instance, let's consider the below given incoming Data records –

DR1, DR3, DR5, DR2

The data records are inserted as given in Figure 4:

Figure 4 Sort File

While inserting a new record the key of the new record is sorted along with the existing records and then inserted. For instance when we try to insert the data record DR4, the sorted insert is depicted in Figure 5

Figure 5 Sort file insert

12.4.3 Advantages and disadvantages of sequential files

In this section we discuss the main advantages and disadvantages of the sequential files.

Advantages

The main advantages of sequential files are as follows:

1. Simple and easy to implement
2. Provides best usage of the storage space.
3. Can be used for data archival needs.
4. The underlying storage systems provide low-cost storage option.
5. Can be stored in inexpensive device like tape device.

Disadvantages

The main disadvantages of the sequential files are as follows:

1. The update operation is costlier and time consuming.
2. The access speed is slow due to the sequential access.
3. It is not possible to directly access a given data record key
4. All records should have uniform structure.
5. The insert time of sorted file is high.
6. Overall processing is slow.

12.4.4 Use cases for sequential files

We generally use sequential files for long-term archival of data and for batch processing operations. We also use sequential access for storing large amount of data that tolerates slow retrieval time.

☞ Check Your Progress – 1

1. A _____ in a file encapsulates the data of a single entity
2. Each data record is uniquely identified by a ____.
3. In _____ files, the data records are inserted in the order of their arrival.
4. For optimizing the performance, we get all the updates into a _____ and later merge it into main file.
5. The two types of sequential files are _____ and _____.
6. Define a data record for a typical university's student enrolment system.

12.5 DIRECT FILE ORGANIZATION

In the direct file organization, we directly access the file by its key. To enable this direct access, we need to map the key to the address where the data record is stored. Direct access is also known as random access. The file records are stored in direct access storage device (DASD) like hard disk, CD, magnetic disk. We randomly place the record throughout the hard disk.

We have depicted the direct file organization in Figure 6

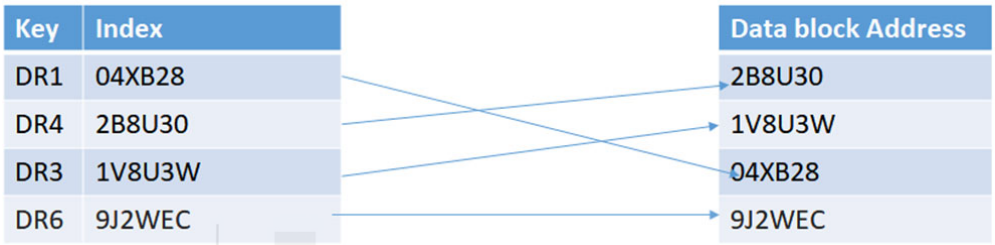

Figure 6 Direct File Organization

We use hash function to convert the key into the address. Naturally direct file access is faster compared to sequential files.

In direct file organization, the records are stored at a known addressed as depicted in Figure 6.

12.5.1 Hash Function and Collision Resolution

The most popular hash function is the modulo based function that computes the remainder value as the hash value based on the key. The hash value is then used to store the data record.

When there are huge set of keys, multiple keys will end up with same hash value leading to collision. In such cases, we use the collision resolution technique. The separate chaining technique chains the values to the same slot and open addressing technique probes for next available slot for placing the value.

12.5.2 Advantages and disadvantages of Direct file organization

We shall look at the main advantages and disadvantages of the direct file organization.

Advantages

Given below are the main advantages of direct file organization:

1. As we can directly access the data, the retrieval is fast that helps in transaction management systems such as relational database systems.
2. The insert, update and delete operations are faster.
3. The direct file organization is efficient in storage and retrieval of large data.
4. Key-based search is faster.
5. Can be used for real-time transactions that needs optimal performance.
6. Concurrent processing is possible

Disadvantages

Given below are the main disadvantages of direct file organization:

1. The required storage technology is costlier.
2. The usage of storage space is sub optimal.
3. Insert, delete and update needs update of the index table.

12.6 INDEXED SEQUENTIAL FILE ORGANIZATION

Indexed sequential file organization supports direct access of keys and also sequential access by keys.

We use the indexed sequential file organization for hierarchical data organization. For instance, lets consider a use case for retrieving the data record for the states. We use the indexed/direct file for getting the country. The country data record points to the sequential file storing the records of its constituent states. Indexed sequential files can be stored only on random access device such as magnetic disk.

To implement the indexed sequential file, we decouple the index and data files. The index file is structured hierarchically in tree structure whereas the data file stores the information sequentially.

We use two types of indexes static index and dynamic index.

We have depicted the depicted the indexed sequential file in Figure 7.

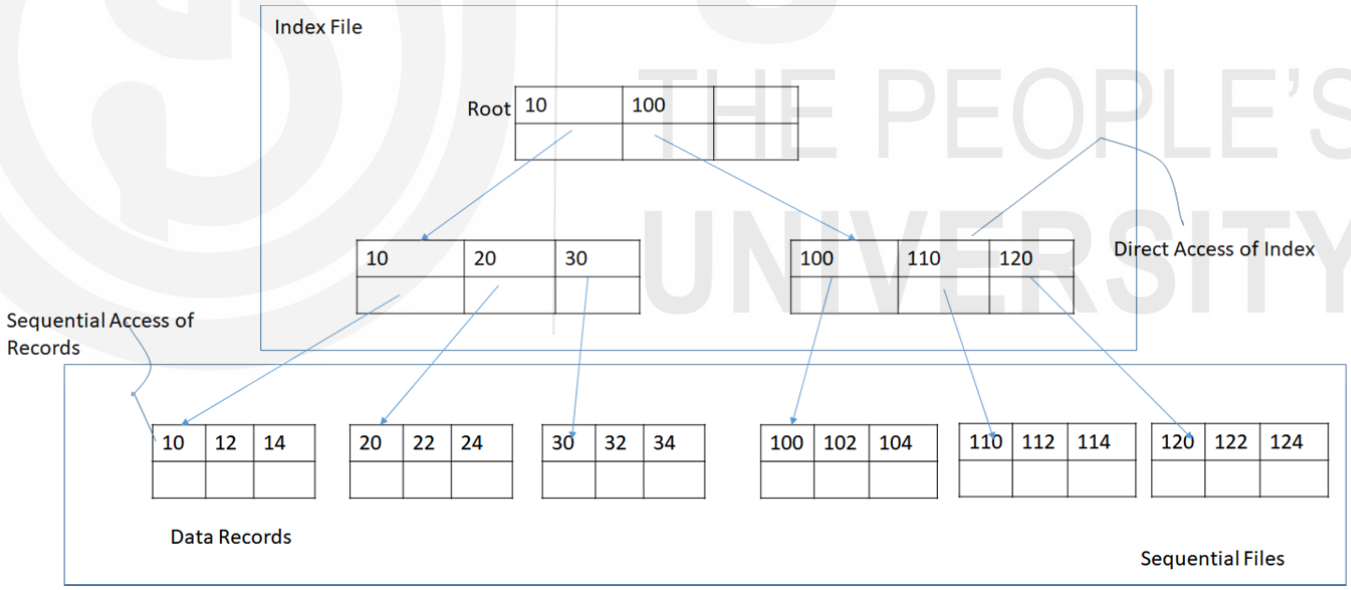

Figure 7 Indexed sequential file

As depicted in Figure 7, the indexed sequential file is depicted as a two-level hierarchy. The first level holds the index to each of the three records in the data file. The second level holds the sequential set of records.

The first level is accessed directly where we get the keys from the index file. In the second level we store the data record sequentially. As the data records are stored sequentially, the key in the first level just points to the first record in the second level from where it can be accessed sequentially.

When we have to insert the record, we update the data file based on the sequence and update the index accordingly. Static index and dynamic index are two main types of indexes. In static indexes the update to the records in the data file does not change the index structure whereas in the dynamic index, the updates to the data file changes the index structure.

12.6.1 Advantages and disadvantages of Indexed sequential file organization

We shall look at the main advantages and disadvantages of the indexed sequential file organization

Advantages

Given below are the main advantages of indexed sequential file organization:

- As the records are directly accessed, the access speed is high
- Record insert is very fast.

Disadvantages

Given below are the main disadvantages of indexed sequential file organization:

- Usage of storage space is sub optimal
- The implementation is costly due to the required of costly hardware
- Extra storage for index file is required.

☞ Check Your Progress – 2

1. In the direct file organization, we directly access the file by its ___
2. The function that converts the key to address is called _____
3. The first level in the indexed sequential acess is done _____ and second level is accessed _____
4. The data records in direct file organization are stored in _____storage device
5. _____ file organization for managing hierarchical data

12.7 SUMMARY

In this unit we mainly discussed about file organization and various types of file organization. A file is a collection of data records. The key operations of the file are File creation, file update, file deletion, file merging, file searching. File organization defines the way the data record is stored and retrieved from the files. We write the data records in a specific sequence in sequential files. There are mainly two types of sequential files - Pile Files and sort files. In Pile files, the data records are inserted in the order of their arrival. In case of sort files, the data records are sorted based on the key and are inserted into the file. In the direct file organization, we directly access the file by its key. Indexed sequential file organization supports direct access of keys and also sequential access by keys.

12.8 SOLUTIONS/ANSWERS

☞ Check Your Progress – 1

1. Data record.

2. Key
3. Pile files
4. Transaction log
5. Pile files and sort files
6. A sample student enrollment structure is as follows:

   Student Number, Student Name, Date of Enrollment, Address, Programme

☞ Check Your Progress – 2

1. Key
2. Hash function
3. Directly and sequentially
4. Direct access
5. Indexed sequential

12.9 FURTHER READINGS

References

https://en.wikipedia.org/wiki/File_system
https://en.wikipedia.org/wiki/Sequential_access
https://en.wikipedia.org/wiki/Indexed_file
https://en.wikipedia.org/wiki/ISAM