**P.V.P Siddhartha Institute of Technology**

**Department of Computer Science and Engineering**

**Subject Code: 20CS3402, Subject Name: Advanced Data Structures, Regulation:PVP20**

**Descriptive Examination -1**

1. A) Apply insertion operation for the given elements using Linear probing collision resolution technique for inserting into the hash table of size 11.

16, 54, 26, 93, 17, 77, 31, 44, 55, 20 CO2, L3, 3M

**Define the Hash Function: 0.5M**

Hash(key)=key mod   11

**Initialize the Hash Table 0.5M**

Create a table of size 11, initially empty:

Index: 0 1 2 3 4 5 6 7 8 9 10

Value: - - - - - - - - - - -

 Insert Elements Using Linear Probing 2M

| **Key** | **Hash Index** | **Collision?** | **Final Index** | **Table Update** |
| --- | --- | --- | --- | --- |
| 16 | 16 % 11 = 5 | No | 5 | [ - - - - - 16 - - - - - ] |
| 54 | 54 % 11 = 10 | No | 10 | [ - - - - - 16 - - - - 54 ] |
| 26 | 26 % 11 = 4 | No | 4 | [ - - - - 26 16 - - - - 54 ] |
| 93 | 93 % 11 = 5 | Yes | 6 | [ - - - - 26 16 93 - - - 54 ] |
| 17 | 17 % 11 = 6 | Yes | 7 | [ - - - - 26 16 93 17 - - 54 ] |
| 77 | 77 % 11 = 0 | No | 0 | [77 - - - 26 16 93 17 - - 54 ] |
| 31 | 31 % 11 = 9 | No | 9 | [77 - - - 26 16 93 17 - 31 54 ] |
| 44 | 44 % 11 = 0 | Yes | 1 | [77 44 - - 26 16 93 17 - 31 54 ] |
| 55 | 55 % 11 = 0 | Yes | 2 | [77 44 55 - 26 16 93 17 - 31 54 ] |
| 20 | 20 % 11 = 9 | Yes | 8 | [77 44 55 - 26 16 93 17 20 31 54 ] |

Final Hash Table (after all insertions):

Index: 0 1 2 3 4 5 6 7 8 9 10

Value: 77 44 55 - 26 16 93 17 20 31 54

 B) Explain Extendible hashing with an example CO1, L2, 2M

**Extendible Hashing** is a **dynamic hashing technique** used in databases and file systems to handle **growing data** efficiently. Unlike static hashing, which may lead to overflow chains when the data exceeds bucket capacity, extendible hashing uses a **directory of pointers to buckets**, and this directory can **grow or shrink** dynamically based on the data.

Main features of Extendible Hashing: The main features in this hashing technique are:

Directories: The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.

Buckets: The buckets are used to hash the actual data

**Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.

**Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.

**Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.

**Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.



* **Step 1 – Analyze Data Elements:**Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer. eg: 49.
* **Step 2 – Convert into binary format:**Convert the data element in Binary form. For string elements, consider the ASCII equivalent integer of the starting character and then convert the integer into binary form. Since we have 49 as our data element, its binary form is 110001.
* **Step 3 – Check Global Depth of the directory.** Suppose the global depth of the Hash-directory is 3.
* **Step 4 – Identify the Directory:**Consider the ‘Global-Depth’ number of LSBs (MSB) in the binary number and match it to the directory id.
Eg. The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110**001** viz. 001.
* **Step 5 – Navigation:**Now, navigate to the bucket pointed by the directory with directory-id 001.
* **Step 6 – Insertion and Overflow Check:**Insert the element and check if the bucket overflows. If an overflow is encountered, go to **step 7** followed by **Step 8**, otherwise, go to **step 9**.
* **Step 7 – Tackling Over Flow Condition during Data Insertion:**Many times, while inserting data in the buckets, it might happen that the Bucket overflows. In such cases, we need to follow an appropriate procedure to avoid mishandling of data.
First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.
	+ **Case1:** If the local depth of the overflowing Bucket is equal to the global depth, then Directory Expansion, as well as Bucket Split, needs to be performed. Then increment the global depth and the local depth value by 1. And, assign appropriate pointers.
	Directory expansion will double the number of directories present in the hash structure.
	+ **Case2:** In case the local depth is less than the global depth, then only Bucket Split takes place. Then increment only the local depth value by 1. And, assign appropriate pointers.

Example:



1. A) Define Priority queue and mention any three applications where priority queues are used. CO1, L2 1M

A priority queue is a special type of abstract data structure where each element is associated with a priority, and elements are served based on their priority.

Higher priority elements are dequeued before lower priority ones.

If two elements have the same priority, they are served according to their order in the queue (typically FIFO).

Applications:

1. **CPU Scheduling in Operating Systems:**

1. **Dijkstra’s Algorithm (Shortest Path Finding):**
2. **Huffman Coding (Data Compression):**

B) Build insertion algorithm for min heap CO3, L3, 2M

MinHeapInsert(heap[], element):

 1. heapSize ← heapSize + 1

 2. heap[heapSize] ← element // Insert at the end

 3. i ← heapSize

 4. while i > 1 and heap[i] < heap[i // 2]: // Compare with parent

 swap heap[i] and heap[i // 2]

 i ← i // 2 // Move up the heap

C) Build a Binomial queue for the following elements one at a time, into an initially empty binomial heap. 21, 12, 65, 24, 16, 18, 13 CO3, L3 2M

1. Construct an AVL tree for the given elements 3, 1, 4, 6, 9, 2, 5, 7 into an initial empty tree. Delete 6, 5, 9 from the build AVL tree. And again insert 13, 22, 55, 79, 65, 70 into the resultant tree. CO3, L3, 5M





