Name and ID:  **Solution**  Section: BAI-3A Marks: 15

**Question#01 [4+4 Marks]**

1. Following is an array to be converted into a binary Min-Heap, using repeated MinHeapify/InsertInHeap calls. Show the contents of this array after the Min-Heap has been constructed.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **…** |
| 11 | 17 | 8 | 21 | 5 | … |

* Delete key. Show the resulting heap and circle the changed positions.
* Insert the key "3". Show the updated heap and circle the changed positions.
* Delete the key again. Show the final heap and circle the changed positions.

Solution

Final Min Heap:

**5**

**/ \**

**11 8**

**/ \**

**21 17**

**Min Heap = [5,11,8,21,17]**

After Deletion:

**8**

**/ \**

**11 17**

**/**

**21**

**Min Heap = [8,11,17,21]**

After inserting 3:

**3**

**/ \**

**8 17**

**/ \**

**21 11**

**Min Heap = [3,8,17,21,11]**

After Deletion:

**8**

**/ \**

**11 17**

**/**

**21**

**Min Heap = [8,11,17,21]**

1. Write a function to decide whether a given integer array satisfy the condition of a Max Heap data structure or not.

Solution

bool isMaxHeap(const vector<int>& arr) {

int n = arr.size();

//Validating Max heap property on all internal nodes

for (int i = 0; i <= (n - 2) / 2; ++i) {

int leftChild = 2 \* i + 1;

int rightChild = 2 \* i + 2;

// Is parent < left child

if (leftChild < n && arr[i] < arr[leftChild]) {

return false;

}

//Is parent < right child

if (rightChild < n && arr[i] < arr[rightChild]) {

return false;

}

}

return true;

}

**Question#02 [2+5 Marks]**

1. Compare the collision resolution strategies Open Addressing vs. Separate Chaining.

Solution

|  |  |
| --- | --- |
| **Open Addressing** | **Separate Chaining** |
| All the keys are stored only inside the hash table. No key present outside the hash table. | Keys stored outside hash table with a linked list, only pointer to the list kept inside the table. |
| Number of keys never exceeds hash table size. | Number of keys can be higher than hash table size. |
| Deletion is difficult. | Deletion is easy. |
| No extra memory other than hash table, cache performance is good. | Extra memory due to linked structures, cache performance is not very good as compared to Open Addressing |
| Bucket is filled when there is no key mapped to these buckets ( probing locations) | Buckets can be empty as there is no key mapped to these. |

1. Taking an initially empty HasHTable of size 10, insert the following keys using hash function

h(key) = key % 7;

(Keys appears in the following order)

**135, 220, 53, 412, 71, 99, 304**

Provide the content of the HashTable when collision resolution strategies used are (i) Separate Chaining and (ii) Quadratic Probing, also state number of collisions in each case.

Solution

|  |  |
| --- | --- |
| Quadratic Probing | Separate Chaining |
| |  |  | | --- | --- | | Index | HashTable[Index] | | 0 |  | | 1 | 71 | | 2 | 135 | | 3 | 220 | | 4 | 53 | | 5 | 99 (actual index 1) | | 6 | 412 | | 7 | 304 (actual index 3) | | 8 |  | | 9 |  |   No. of collision: 4 | |  |  | | --- | --- | | Index | Linked List Chain | | 0 |  | | 1 | 71 -> 99 | | 2 | 135 | | 3 | 220 -> 304 | | 4 | 53 | | 5 |  | | 6 | 412 | | 7 |  | | 8 |  | | 9 |  |   No. of collision: 2 |