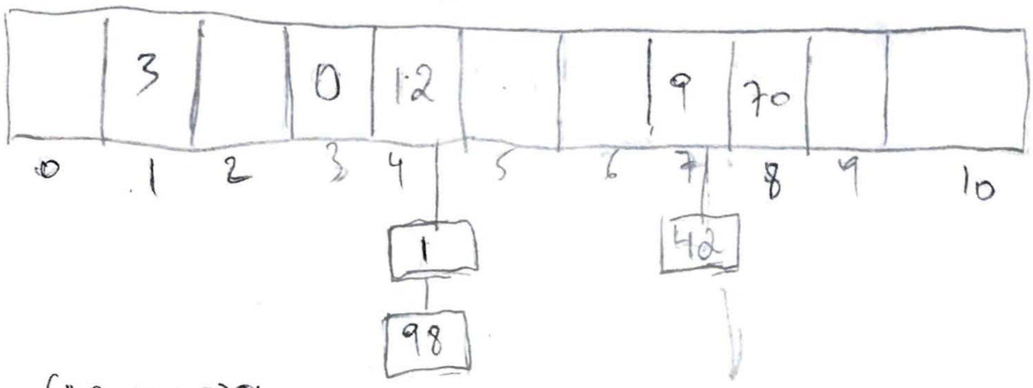
(1)




key 12 = (12 · 12 · 3) % 11 = 4

key 9 = (9 · 9 + 3) % 11 = 84 % 11 = 7

key 1 = (1 · 1 + 3) % 11 = 4 % 11 = 4

key 0 = (0 · 0 + 3) % 11 = 3

key 42 = (42 · 42 + 3) % 11 = 7

key 98 = (98 · 98 + 3) % 11 = 4

key 70 = (70 · 70 + 3) % 11 = 8

key 3 = (3 · 3 + 3) % 11 = 1

Linear Probing · Probe(i) = (i + 1) % Tablesize

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | | 0 | 12 | 1 | 98 | 9 | 42 | 70 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Quadratic Probing Probe(i) = (1 · i + 5) % Table size

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 42 | | 0 | 12 | 3 | | 9 | 70 | 1 | 98 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

(2) 1500 because the number of total entries is not given. the largest table size is the best pick

(3) 53491/106963 = 0.5 entries per bucket

(4) insert (x) → O(1)
Rehash () → O(n)
Remove (x) → O(1)
Contains (x) → O(1)

(5) No, because you only need to insert, search, and delete functo

(6)
```
int hashit (int key, int TS){
    return key % TS;
}
int hashit (String key, int TS){
  Char t[];
  t = key.toCharArray();
  int length = key.length();
  int x = 0, y = 0
  while (x < length){
    y = y + t[x];
      x++;
  }
  return y % Ts;
}
```

(7) Program will slow down because table size will be doubled, making operations based on size take longer.

(8) Push(x) $\rightarrow O(1)$
top() $\rightarrow O(1)$
Pop() $\rightarrow O(\log n)$
buildHeap(vector<int> {1...n}) $\rightarrow O(n)$

(9) priority Queue can be used in a printer, smaller print jobs can be taken care of quickly and make the whole process more efficient.

(10)

Heap

Parent: $i/2$

Children: $2i$, $2i+1$

D-Heap

Parent: $(i-1)$

Children: $d(i-1)+2, d(i-1)+3, \ldots d(i-1)+d+1$

(11)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 10 | | | | | | | | |
| 10 | 12 | | | | | | | |
| 10 | 12 | 1 | | | | | | |
| 1 | 12 | 10 | | | | | | |
| 1 | 12 | 10 | 14 | | | | | |
| 1 | 12 | 10 | 14 | 6 | | | | |
| 1 | 6 | 10 | 14 | 12 | | | | |
| 1 | 6 | 10 | 14 | 12 | 5 | | | |
| 1 | 6 | 5 | 14 | 12 | 10 | | | |
| 1 | 6 | 5 | 14 | 12 | 10 | 15 | | |
| 1 | 6 | 5 | 14 | 12 | 10 | 15 | 3 | |
| 1 | 3 | 5 | 6 | 12 | 10 | 15 | 14 | |
| 1 | 3 | 5 | 6 | 12 | 10 | 15 | 14 | 11 |

12

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 12 | 10 | 15 | 14 | 11 |

13

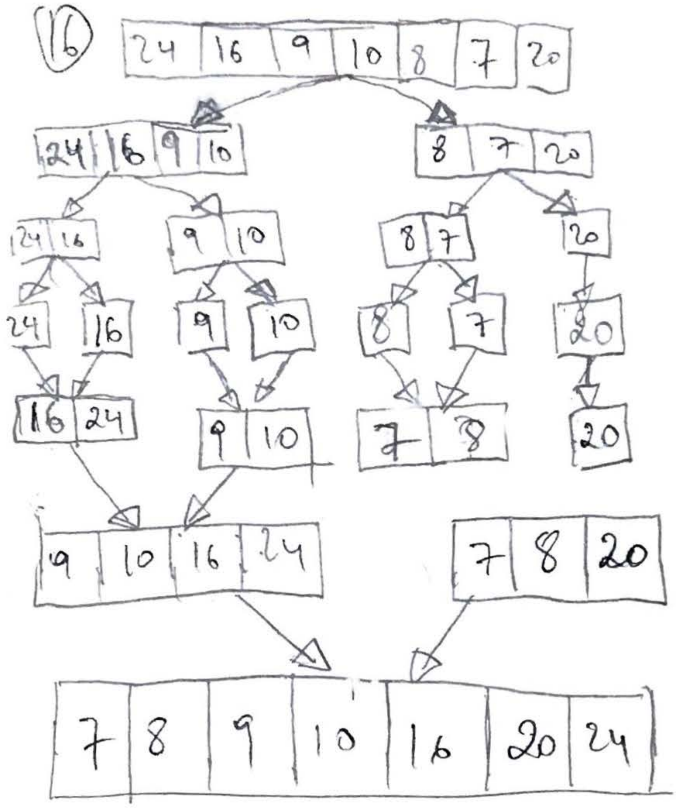
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 6 | 5 | 11 | 12 | 10 | 15 | 14 |
| 5 | 6 | 10 | 11 | 12 | 14 | 15 | |
| 6 | 11 | 10 | 15 | 12 | 14 | | |

(14)

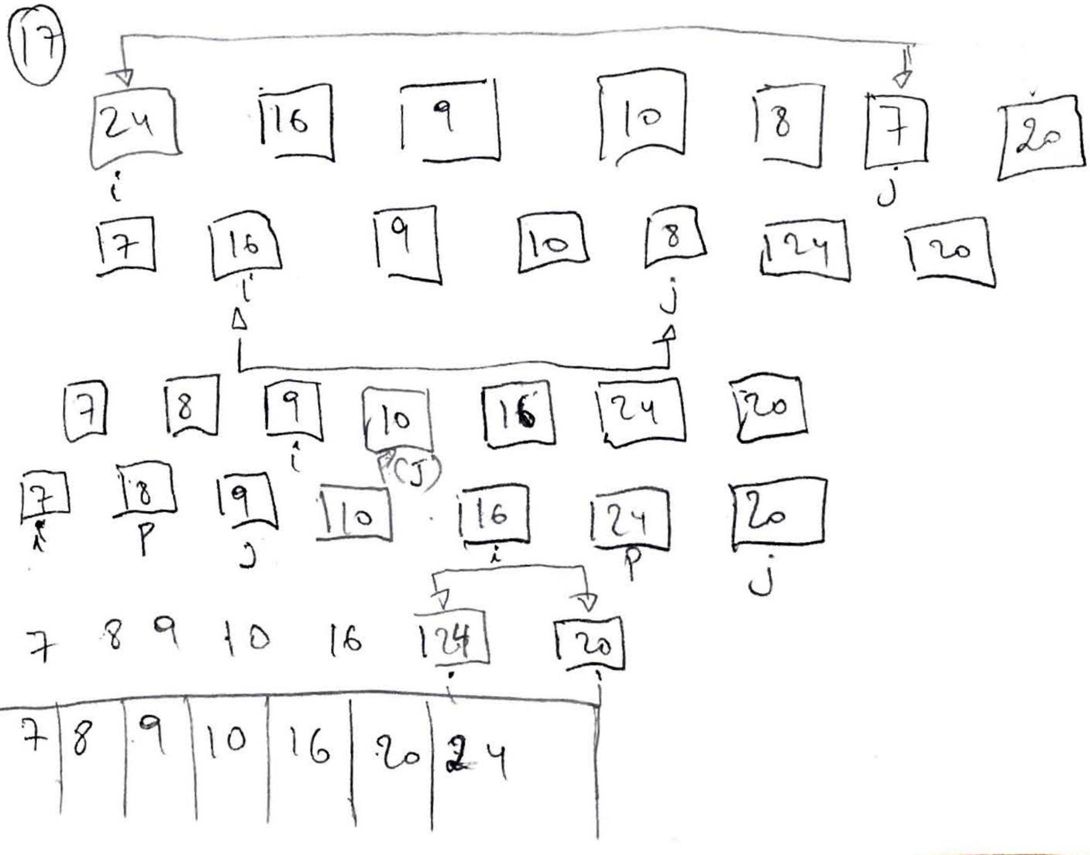
| Algorithm | Average complexity | Stable (yes/no) |
|---|---|---|
| Bubble Sort | $O(n^2)$ | yes |
| Insertion sort | $O(n^2)$ | yes |
| Heap Sort | $O(n \log n)$ | No |
| Merge Sort | $O(n \log n)$ | Yes |
| Radix Sort | $O(nk)$ | yes |
| Quick sort | $O(n \log n)$ | No |

(15) Quicksort is an IN-PLACE Algorithm while ~~M~~ merge sort is not. Quick sort does not require any additional memory while sorting but, mergesort requires extra memory in order of $O(n)$
Quicksort is not a stable sorting algorithm while Merge is. merge sort preserves the relative order of the elements with equal values but, quicksort does not guarantee that.
Quicksort is a bit faster than merge sort and uses no extra memory.

16


24 16 9 10 8 7 20
24 16 9 10
8 7 20
24 16
9 10
8 7
20
24
16
9
10
8
7
20
16 24
9 10
7 8
20
9 10 16 24
7 8 20
7 8 9 10 16 20 24

17


24 16 9 10 8 7 20
i
j
7 16 9 10 8 24 20
i
j
7 8 9 10 16 24 20
i
(j)
7 8 9 10 16 24 20
i
P
j
i
P
j
7 8 9 10 16 24 20
i
j
7 8 9 10 16 20 24