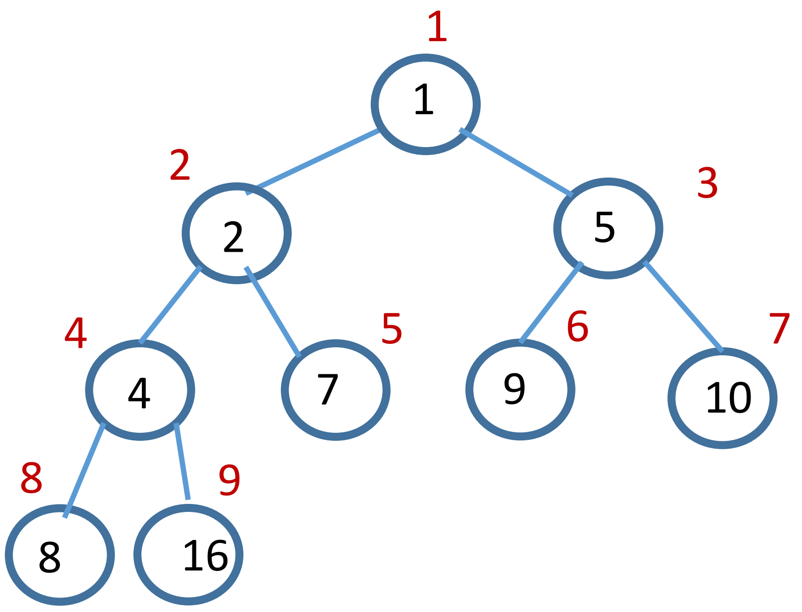
COMP 2611, DATA STRUCTURES
LECTURE 15

MIN-HEAPS

PRIORITY QUEUES: MAX-PRIORITY QUEUES

HASHING TECHNIQUES

A Min-Heap



0	1	2	3	4	5	6	7	8	9
	1	2	5	4	7	9	10	8	16

Priority Queues

- Application of a heap: an efficient priority queue
- Priority queues come in two forms: max-priority queues and min-priority queues
- A priority queue is a data structure for maintaining a set *S* of elements, each one with an associated value called a *key*. A max-priority queue supports the following operations:

```
insert (S, x)
maximum (S)
extractMax (S)
increaseKey (S, x, k)
```

Priority Queues: Implementation

```
struct MaxPriorityQueue {
  MaxHeap * heap;
};
```

```
insert (S, x)              void insert (MaxPriorityQueue * mpq, int key);
maximum (S)                int maximum (MaxPriorityQueue * mpq);
extractMax (S)             int extractMax (MaxPriorityQueue * mpq);
increaseKey (S, x, k)      void increaseKey(MaxPriorityQueue * mpq, int i, int key);
```

Operations on Data Structures

Search

Search	Data structure
$O(n)$	Arrays
$O(n)$	Linked Lists
$O(n)$	Binary Trees
$O(\log_2 n)$	Binary Search Trees
$O(\log_2 n)$	Heaps

Search

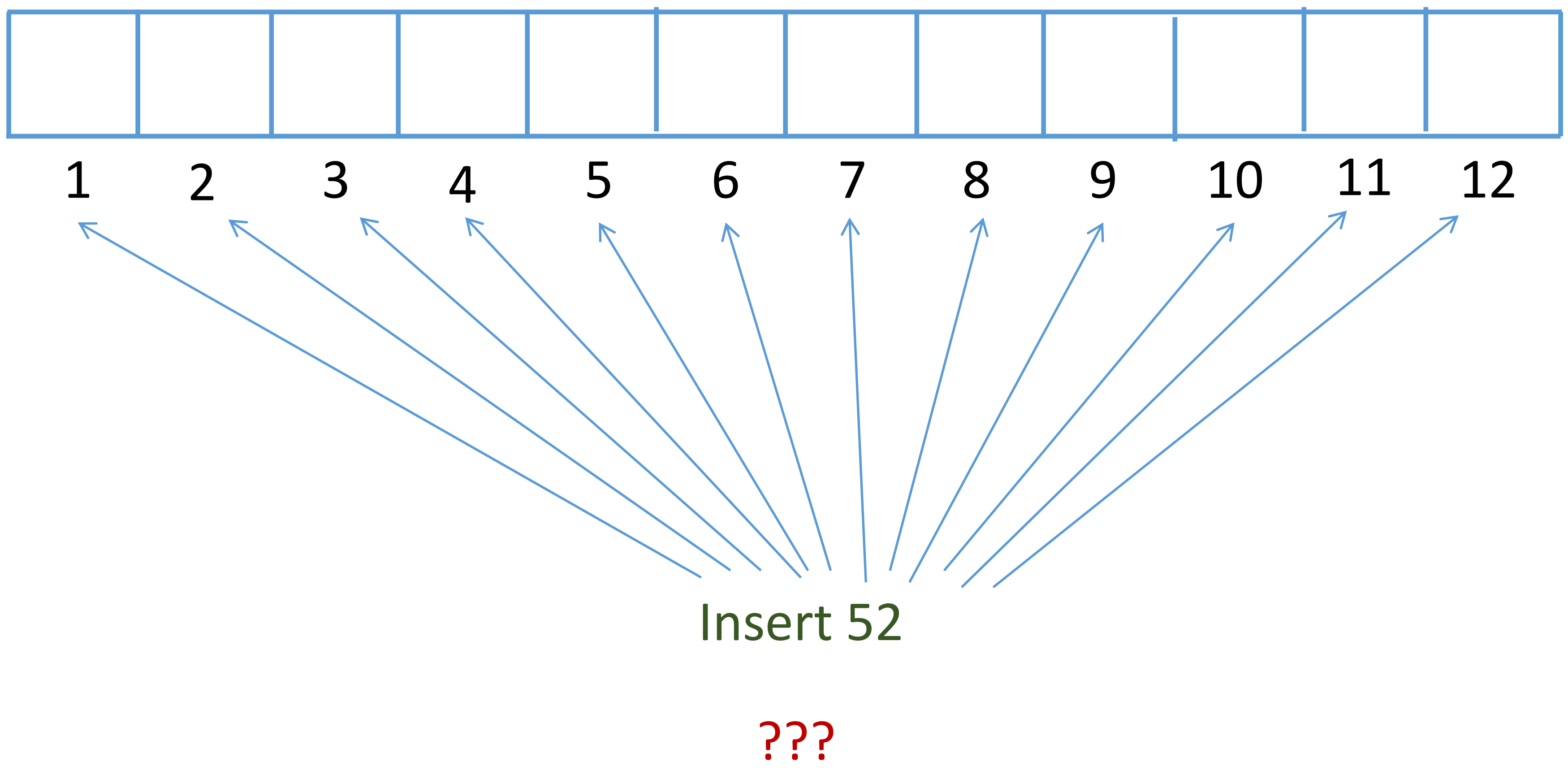
Insert

Delete

Get ordered list of keys

Is it possible to have a data structure where searching for a key takes O(1) or constant time?

Hashing



Hashing

- Use *hash* function, *h*, to convert *key* to a valid location in the array. When this is done, the array is called a *hashtable*.
- For example, h = key % 12 + 1
- Where will 52 hash to?
- 52 will hash to, 52 % 12 + 1 = 5. Location 5 is empty, so 52 is inserted in location 5.

				52							
1	2	3	4	5	6	7	8	9	10	11	12

Empty Locations

- Initialize the elements of the hashtable with a value that indicates empty. If the keys are positive integers, 0 can be used to indicate empty.

Hashing

$33 \rightarrow 33 \% 12 + 1 = 10$

$84 \rightarrow 84 \% 12 + 1 = 1$

$43 \rightarrow 43 \% 12 + 1 = 8$

- Insert 33, 84, and 43.
- Insert 16.
- 16 % 12 + 1 = 5. But, location 5 already has 52.
- This is referred to as a *collision*.
- Where to insert 16?

1	2	3	4	5	6	7	8	9	10	11	12
84				52	16		43		33		

Hashing

- Insert 59 and 31.

- 59 % 12 + 1 = 12. Location 12 is empty. So, 59 is placed in location 12.

- 31 % 12 + 1 = 8. But, location 8 already has 43. Collision!

- We try the next location, 9. It is empty so 31 is placed in location 9,

84				52	16		43	31	33		59
1	2	3	4	5	6	7	8	9	10	11	12

Hashing

- Insert 23.

- 23 % 12 + 1 = 12. Location 12 is occupied. So, we try the next location.

- Treat the table as circular so the next location is 1. But, location 1 is occupied. So, we try the next location, location 2.

- Location 2 is empty so 23 is placed there.

84	23			52	16		43	31	33		59
1	2	3	4	5	6	7	8	9	10	11	12

Hashing

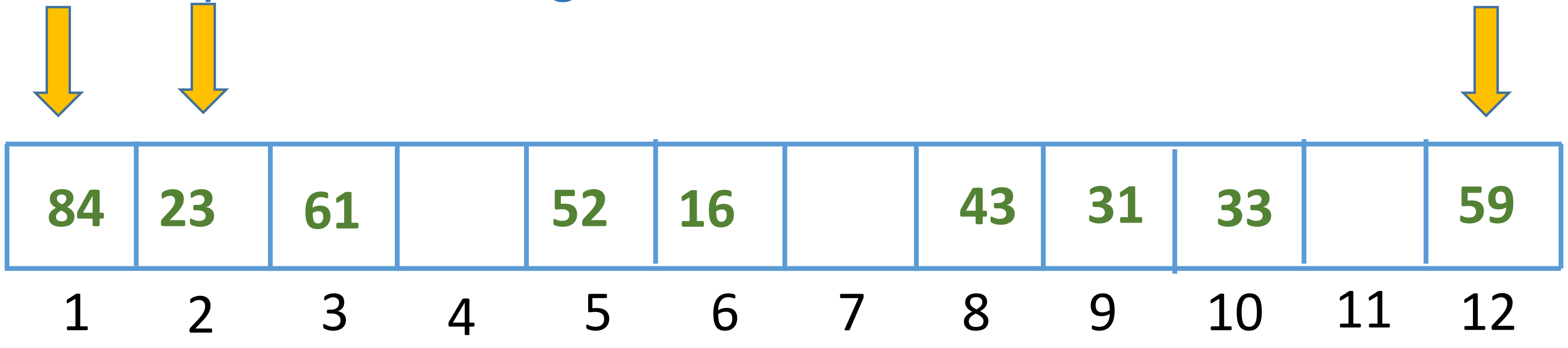
- Insert 61.
- 61 % 12 + 1 = 2. Location 2 is occupied. So, we try the next location.
- Location 3 is empty so 61 is placed there.

84	23	61		52	16		43	31	33		59
1	2	3	4	5	6	7	8	9	10	11	12

Searching for a Key

- Let’s search for 23.
- 23 % 12 + 1 = 12. Location 12 is occupied but not by 23.
- We try the next location, 1. Location 1 is occupied but not by 23
- We try the next location 2. Location 2 contains the key we are looking for.

> How would we know if the table does NOT contain the key?



Deleting a Key

- Once a key is found in the hashtable it can be deleted by assigning a value that signifies "deleted". For example, if the keys are positive integers, -1 can be placed in a deleted location.

- For example, let's delete 84.

84	23	61		52	16		43	31	33		59
1	2	3	4	5	6	7	8	9	10	11	12

Dealing with a Deleted Key

- When searching for a key (e.g., 23), a deleted key is treated as if it is a valid key.
- When inserting a key, it can be inserted in the first empty location or in the first deleted location.
- For example, 71 can be inserted in Location 1.

1	2	3	4	5	6	7	8	9	10	11	12
-1	23	61		52	16		43	31	33		59