Chapter 4

Priority Queues and Hash Tables

The Main Topics

- Learn about priority queues
- Learn about hash tables

Priority Queues

Priority Queues

The task is to write a software to manage a hospital's emergency room

- When any patient enters the hospital, the staff …
    - … creates a record about that person in a database
    - … keeps track of the patients and decides when each person will receive care

➡️ What data structure should be used to store the records?

- A sorted list that would enable the treatment of patients in alphabetical order by name, or
- A queue that would enable the treatment of patients in the order of arrival

Priority Queues

- The staff should assign some measure of urgency, or *priority*, to the patients waiting for treatment
  - The next available doctor should treat the patient with the highest priority

  ➡️ The data structure should produce this patient on request
- *Priority queue* is the data structure that we need in this situation

Definition of a Priority Queue

- A *priority queue* is a data structure for maintaining a set of elements, each with an associated value called a *priority*
- A *priority value* ($\in \mathbb{N}$) indicates a patient's priority for treatment (or a task's priority for completion)
  - Assuming that *the smallest priority value indicates the highest priority*

Definition of a Priority Queue

- A priority queue provides the following operations:
  - *Test* whether a priority queue is empty
  - *Insert* a new element into the priority queue
  - *Extract* from the priority queue the element with the highest priority
  - (optional) *Get* the element in the priority queue with the highest priority
  - (optional) *Update* the i^{th} element's priority value with the new value

Implementation of Priority Queues

- Using singly-linked list or array
  - The list is arranged in ascending order of elements based on their priority
- Using min-heap
  - This is the most common implementation of priority queue
- Using balanced binary tree

Hashing

Introduction

- In computing, the operations we need to perform for a set of elements most often are:
  - searching for a given element
  - adding a new element
  - deleting an element
- A data structure that supports these three operations is called a *dictionary*
- There are quite a few ways a dictionary can be implemented and one of them is *hashing* technique

Hashing

- Firstly, let's assume that we have to implement a dictionary of n records with keys $k_1, k_2, \dots, k_n$
- *Hashing* is based on the idea of distributing keys among a one-dimensional array $H[0..m-1]$ called a *hash table*
- Given a predefined function h (called *hash function*), the distribution is done by computing the value $h(k_i), \forall i \in [1, n]$
  - $h(k_i) \in [0, m-1]$ is called *hash value*

Example

- If keys are nonnegative integers:

$$h(k) = k \,\%\, m$$

- If keys are letters of some alphabet Σ:

$$h(k) = ord(k) \,\%\, m$$

where $ord(k)$ indicates the k's position in Σ

- If keys are character strings $c_1 c_2 \dots c_n, \forall i \in [1, n]: c_i \in \Sigma$

$$h(k) = \left(\sum_{i=1}^{n} ord(c_i) \right) \%\, m$$

Hash Function

- In general, a hash function needs to satisfy some requirements:
  - It needs to distribute keys among the cells of the hash table *as evenly as possible*
  - It has to be easy to compute
  - It must be a constant-time operation, independent from the number of keys and the size of the hash table

Collisions

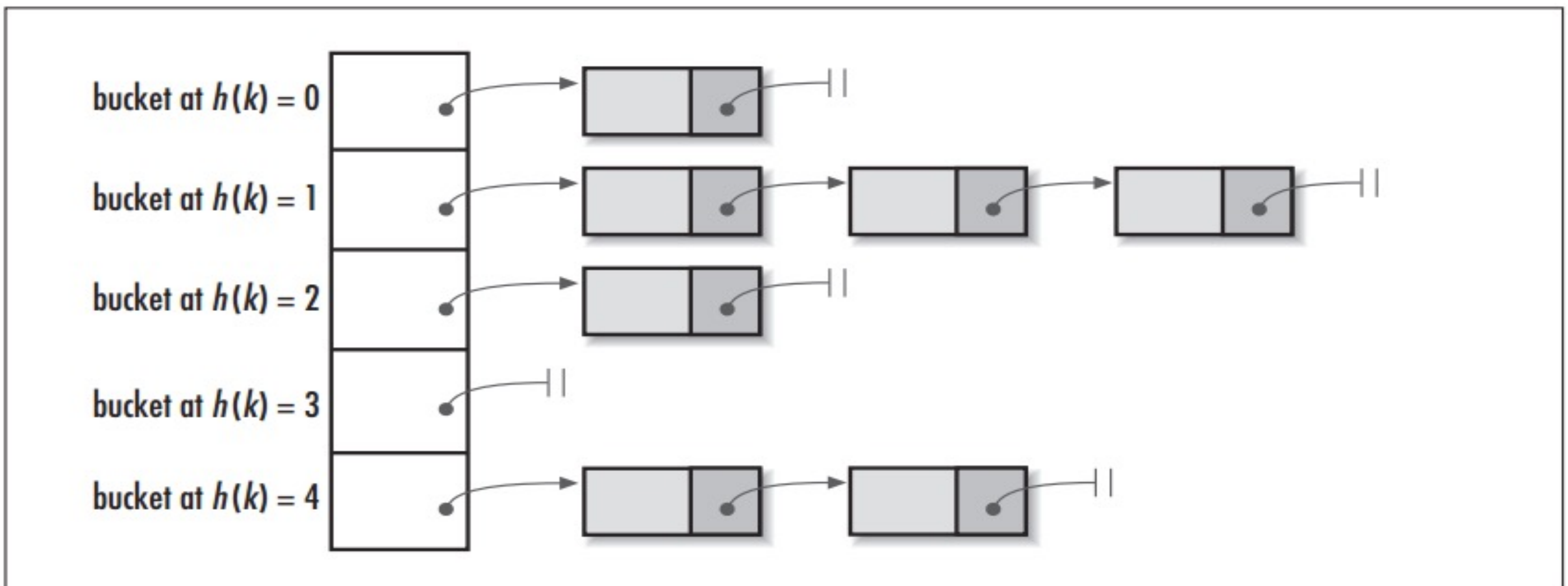
- *Collision* is a phenomenon of two (or more) keys being hashed into the same cell of the hash table
  - If a hash table's size m is smaller than the number of keys n, we will surely get collisions
  - But collisions should be expected even if m is considerably larger than n
- Fortunately, with an appropriately chosen hash table size and a good hash function, this situation happens very rarely

Collision Resolutions

- Every hashing scheme must have a *collision resolution mechanism*
- This mechanism is different in the two principal versions of hashing:
  - *open hashing* (aka *separate chaining*)
  - *closed hashing* (aka *open addressing*)

Open Hashing (Separate Chaining)

- In open hashing, keys are stored on linked lists (called *buckets*) attached to cells of a hash table
- Each bucket contains all the keys hashed to its cell



Search Operation

- Let's assume that k is the search key
- Firstly, we compute the hash value $h(k)$
- If the bucket at index $h(k)$ is not empty, it may contain the search key

➡️ Traversing the bucket to find an occurrence of k

Insertion Operation

- Let's assume that a key k needs to be inserted into the hash table
- At first, the key k will be searched for in the hash table
- If this is an unsuccessful search, the key will be added to the bucket at index $h(k)$ in the hash table

Deletion Operation

- Let's assume that a key k needs to be removed from the hash table
- Firstly, the key k will be searched for in the hash table
- If this is a successful search, the node containing the key in the bucket at index $h(k)$ will be deleted

Analysis of Open Hashing

- The efficiency depends on the lengths of the buckets, which, in turn, depend on the table size and the quality of the hash function
- If n keys are distributed among m cells of the hash table evenly, the length of each bucket will be about

$$\alpha = \frac{n}{m}$$

  - α is called the *load factor* of the hash table
  - Its value should be not far from 1

Closed Hashing (Open Addressing)

- In closed hashing, all keys are stored in the hash table itself without the use of linked lists
  - The table size m must be at least as large as the number of records n
- There are different strategies that can be employed for collision resolution
- The simplest strategy – called *linear probing* – is mentioned in this course

Search Operation

Firstly, the hash value $h(k)$ of the search key k is computed

if the cell at index $h(k)$ is empty, the search is unsuccessful

otherwise, k is compared with the key in the cell

  if they are equal, the search is successful

  otherwise, k is compared with the key in the next cell

➡️ This process is repeated until either a matching key or an empty cell is encountered

Note: If the end of the hash table is reached, the search is wrapped to the beginning of the table

Insertion Operation

- Let k be the key that needs to be inserted into the hash table
- Firstly, the search operation is used to locate an empty cell
- Then, k will be installed there

Rehashing

- The operations will fail if the hash table is full
- When the table gets close to being full, it will be rehashed

Rehashing: The current table is scanned, and all its keys are relocated into a larger table

Deletion Operation

- Let k be the key that needs to be deleted from the hash table

Firstly, the search operation is used to locate the cell containing the key k

if it is a successful search, k will be deleted from the cell

otherwise ?

k is in the table but the search is unsuccessful

Deletion Operation

- Let k be the key that needs to be deleted from the hash table

Firstly, the search operation is used to locate the cell containing the key k

if it is a successful search, k will be deleted from the cell

otherwise ?

➡️ To mark previously occupied locations by a special symbol to distinguish them from locations that have not been occupied

Analysis of Closed Hashing

- The analysis of linear probing is a much more difficult problem than that of open hashing
- The cost when the table is not too full is typically close to one cell access