M7 - Algorithms and Data Structures

1 Introduction

1.4 Fundamental Data Structures

A data structure can be defined as a particular scheme of organizing related data items.

Sets and Dictionaries

A **set** can be described as an unordered collection (possibly empty) of distinct items called elements of the set

Sets can be implemented in computer applications in two ways:

1. The first considers only sets that are subsets of some large set U, called the universal set.

If set U has n elements, then any subset S of U can be represented by a bit string of size n, called a **bit vector**, in which the ith element is 1 if and only if the ith element of U is included in set S.

This way of representing sets makes it possible to implement the standard set operations very fast; but at the expense of potentially using a large amount of storage.

U = {1, 2, 3, 4, 5, 6, 7, 8, 9}, then S = {2, 3, 5, 7} is represented by the bit string 011010100

2. **(MORE COMMOM)** is to use the list structure to indicate the set's elements. (only for finite sets)

Operations:
void clear(Dictionary d);
void insert(Dictionary d, Key k, E e);
E **remove(Dictionary d, Key k);**
E removeAny(Dictionary d);
E find(Dictionary d, Key k);
int size(Dictionary d);

A data structure that implements these three operations is called the **dictionary**.

an efficient implementation of a dictionary has to strike a compromise between the efficiency of searching and the efficiencies of the other two operations

2.1. a set cannot contain identical elements; a list can.

This requirement for uniqueness is sometimes circumvented by the introduction of a **multiset, or bag**, an unordered collection of items that are not necessarily distinct.

Note, however, the two principal points of distinction between sets and lists:

2.2. a set is an unordered collection of items; therefore, changing the order of its elements does not change the set. A list, defined as an ordered collection of items, is exactly the opposite.

It is also worth mentioning that if a set is represented by a list, depending on the application at hand, it might be worth maintaining the list in a sorted order.

7 Space and Time Trade-Offs

input enhancement: the idea is to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward.

prestructuring: simply uses extra space to facilitate faster and/or more flexible access to the data

dynamic programming: based on recording solutions to overlapping subproblems of a given problem in a table from which a solution to the problem in question is then obtained.

7.3 Hashing

Hashing is based on the idea of distributing keys among a one-dimensional array H[0..m − 1] called a **hash table**.

for each of the keys, the value of some predefined function h called the **hash function**

This function assigns an integer between 0 and m − 1, called the **hash address**, to a key.

Obviously, if we choose a hash table's size m to be smaller than the number of keys n, we will get **collisions**

every hashing scheme must have a collision resolution mechanism

This mechanism is different in the two principal versions of hashing: **open hashing** (also called **separate chaining**) and **closed hashing** (also called **open addressing**).

Open Hashing (Separate Chaining)

keys are stored in **linked lists** attached to cells of a hash table. Each list contains all the keys hashed to its cell.

Indeed, they are almost identical to **searching** sequentially in a linked list; what we have gained by hashing is a reduction in average list size by a factor of m, the size of the hash table.

Having it too small would imply a lot of empty lists and hence inefficient use of space; having it too large would mean longer linked lists and hence longer search times.

the efficiency of these operations is identical to that of searching, and they are all Θ(1) in the average case

The two other dictionary operations —**insertion and deletion**—are almost identical to searching.

Insertions are normally done at the end of a list

Deletion is performed by searching for a key to be deleted and then removing it from its list

But if we do have the load factor around 1, we have an amazingly efficient scheme that makes it possible to search for a given key for, on average, the price of one or two comparisons!

If the hash function distributes n keys among m cells of the hash table about evenly, each list will be about n/m keys long. The ratio **α = n/m**, called the **load factor**

Closed Hashing (Open Adressing)

all keys are stored in the hash table itself without the use of linked lists.

this implies that the table size m must be at least as large as the number of keys n.

Different strategies can be employed for collision resolution:

linear probing: checks the cell following the one where the collision occurs. If that cell is empty, the new key is installed there; if the next cell is already occupied, the availability of that cell's immediate successor is checked, and so on. (It is treated as a circular array)

as the hash table gets closer to being full, the performance of linear probing deteriorates because of a phenomenon called clustering. A **cluster** in linear probing is a sequence of contiguously occupied cells

double hashing: we use another hash function, s(K), to determine a fixed increment for the probing sequence to be used after a collision at location l = h(K): (l + s(K)) mod m, (l + 2s(K)) mod m...

Some partial results and considerable practical experience with the method suggest that with good hashing functions—both primary and secondary—**double hashing is superior to linear probing.**

Some functions recommended in the literature are s(k) = m − 2 − k mod (m − 2) and s(k) = 8 − (k mod 8) for small tables and s(k) = k mod 97 + 1 for larger ones

But its performance also deteriorates when the table gets close to being full

A natural solution in such a situation is **rehashing**: the current table is scanned, and all its keys are relocated into a larger table.