Data Structures

Dictionaries

- Similar to dictionary of words, facilitates looking up things
- Elements are assigned with keys, keys are used to insert, look up or remove elements
- Dictionary stores items which are key-element pairs (k, e) (key may be the element)
- A symbol table maintained by a compiler
  - Name of an entity is the key
  - Properties of an entity constitute the element part
- Customer records of a bank
  - Account number is the key
  - Other information (personal ad financial) related to a customer is the element part

Dictionaries

- Different data structures to realize dictionaries:
  - Arrays, linked lists
  - Hash tables
  - AVL trees
  - Binary trees
  - B-trees
  - Red/black trees

Dictionaries

- The operations should be supported are:
  - Insertion
  - Removal
  - Searching elements
- Types of dictionaries:
  - Ordered and unordered
  - In both cases key serves as an identifier

Dictionaries

- It can store multiple items with the same key (should not be the case for some applications)
- If keys are unique, then the key is viewed as the address for that item in the memory
- The methods supported by the dictionary ADT are:
  - findElement(k): if there exists an item with key k, returns the element of such an item, and NO_SUCH_KEY otherwise
  - InsertItem((k,e): inserts item (k,e)
  - removeItem(k): if there exists an item with key k, removes such an item and returns the element of the removed item, and NO_SUCH_KEY otherwise
  - NO_SUCH_KEY is known as sentinel

Dictionaries

- Supporting methods:
  - size()
  - isEmpty()
  - elements()
  - keys()
  - findAllElements(k)
  - removeAllElements(k)

Dictionaries

- Consider a telephone service provider
- Has a large number of customers
- How do the company maintain the data of customers so that individual records can be accessed efficiently?
- Data is in the form of (key, customer_details)
- Key is the phone number, which has 10 digits
- n is the number of customers

Dictionary: unordered sequence

- We push the data into the dictionary arbitrarily
- Search operation is O(n)
- Removal is takes O(n)
- Insertion takes O(1)

Dictionary: ordered sequence

- The customer recorded are stored as an ordered sequence (say in the order of keys)
- Search takes O(log n)
- Insertion takes O(n)
- Deletion takes O(n)

Dictionary: direct addressing

- Have an array of 10^{10} cells
- Insert item (k, e) in the cell with index k
- All operations are supported in constant time
- What is the big disadvantage?
- Another issue?

Dictionary: hash table

- Define a table of size n
- Store the item (k, e) in the cell k mod n
- All operations are supported in O(1) time
- Issue?

Hash table

- An efficient way to implement a directory is using hash table
- Worst-case running time is O(n) and expected running time is O(1) (n is the number of items in the dictionary)
- It consists of two major components:
  - a bucket array
  - a hash function
- Bucket array:
  - an array A of size N
  - Each cell of A is thought of as a bucket (container of key-element pairs)
  - N is the capacity
  - An element with key “k” is inserted into bucket A[h(k)]
  - Cells associated with keys not present in the dictionary are holding sentinel, NO_SUCH_KEY (assumption)

Hash table

- Bucket array:
  - If keys are not unique, then we have a “collision”
  - If each cell is capable to hold only one element, then collision is an issue (can be dealt with)
- If the keys are unique and $N \geq n$, then what is the time complexity of operations?
- What about space complexity?
- Define hash table as a combination of a bucket array and a good mapping function from keys to the integers in the range $[0, N-1]$

Hash table: Hash functions

- The second part of hash table structure is a "hash function (h)" which maps keys to integers in the range [0, N-1], N is the bucket array capacity
- "h" can be applied to arbitrary keys
- h(k) serves as the index into the bucket array, store (k, e) in the bucket A[h(k)]
- A key k should get mapped to the same value each time
- Characteristics of a good hash function: minimize collisions, easy and fast, uniform hashing
- Uniform hashing: each key is equally likely to hash to any of N buckets, independent of where any other key has hashed to

Hash table: Hash functions

- Evaluation of hash function consists of two operations:
  - mapping keys to integers (called hash code)
  - mapping hash code to an integer in the range of bucket array indices (compression map)

Hash table: Hash functions

- First action:
  - Takes an arbitrary key k and maps it to an integer (hash code or value for k)
  - This integer need not be in the range [0, N-1] (could be negative)
  - The hash codes should avoid collisions as much as possible
- Hash codes:
  - cast a long integer representation of a key down to an integer (may lead to a high number of collisions)
  - Sum integer representation of higher order bits and lower order bits
  - This approach can be extended to any kind of key x whose binary representation can be viewed as a k-tuple (x_0, x_1, . . ., x_{k-1}) of integers
  - Character strings can be interpreted as tuple of integers using ASCII character set
  - The hash code of x is: $\sum_{i=0}^{k-1} x_{\mathrm{i}}$