1. Dictionary

* 1) Input description: A set of n records, each identified by one or more key fields.
* 2) Problem description: Build and maintain a data structure to efficiently locate,
* insert, and delete the record associated with any query key q.
* 3) Implementations
* Unsorted lists and arrays: for small size
* Hash table: size 100 and 10,000,000. Need to handle collision
* Binary search tree: need maintain balance
* B-tree: very large data sets, and store in database
* 2. Priority Queues (queue for breath)
* 1) Input description: A set of records with numerically or otherwise totally-ordered keys.
* 2) Problem description: Build and maintain a data structure for providing quick access to the smallest or largest key in the set
* 3) Discussion: Priority queues are useful data structures in simulations, particularly for maintaining a set of future events ordered by time
* 4) Implementations:
* sorted array: make insertion slow(but deletion fast)
* binary heaps: insertion and deletion O(lgn)
* bounded height priority queue: known the key range(1 to n)
* binary search tree: insert and deletion O(lgn) and can use as dictionary
* 3. Suffix Trees
* 1) input: A reference string S
* 2) Problem description: Build a data structure to quickly find all places where an arbitrary query string q occurs in S
* 3) Discussion: Suffix trees and arrays are phenomenally useful data structures for solving string problems elegantly and efficiently (O(n) time)
* 4) Implementations:
* Trie: where each edge represents one character, and the root represents the null string. Thus, each path from the root represents a string, described by the characters labeling the edges traversed
* Suffix Trie: trie of the n suffixes of an n-character string S (s[0:n], s[1:n],…. s[n,n]

5) Usage:

* Trie: test string in string set, test prefix
* Suffix Tree: test substring in a string(all substring questions for single string)
  + Find all occurrences of q as a substring of S
  + Longest substring common to a set of strings
  + Find the longest palindrome in S (比较A[i] 和 A[n-i-1], A[i] 和 A[n-i])

4. Graph Data Structures

* 1) Input description: A graph G.
* 2) Problem description: Represent the graph G using a flexible, efficient data structure.
* 3) Discussion: If your graph is extremely large, it may become necessary to switch to a hierarchical representation, where the vertices are clustered into subgraphs that are compressed into single vertices
* - The first breaks the graph into components in a natural or application-specific way
* - The other approach runs a graph partition algorithm in 16.6
* 4) Implementations
* adjacency matrix:
* adjacency lists (Bipartite incidence structures)

5. Set Data Structures

* 1) Input description: A universe of items U = {u1, . . . , un} on which is defined a collection of subsets S = {S1, . . . , Sm}.
* 2) Problem description: Represent each subset so as to efficiently (1) test whether ui ∈ Sj , (2) compute the union or intersection of Si and Sj , and (3) insert or delete members of S.

3) Representation

* Bit vectors (Element insertion and deletion simply flips the appropriate bit. Intersection and union are done by “and-ing” or “or- ing” the bits together )
* Containers or dictionaries (a linear- time traversal through both subsets identifies all duplicates, set size is small)