Hashing ~ 01

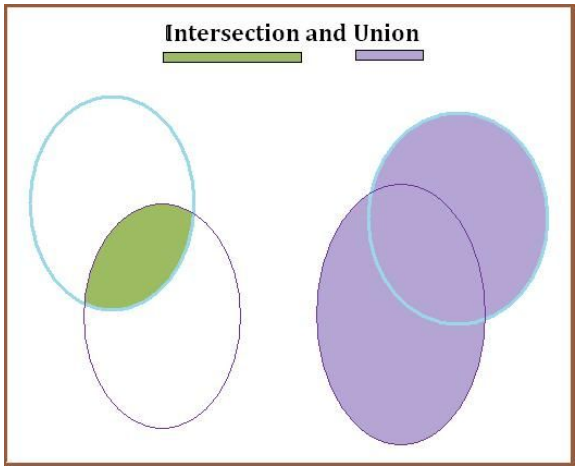
CSE 122 ~ Algorithms & ADT

Sets & Dictionaries

- Sets are drawn from a single universe
  - Ex: the universe of all ordered pairs, all strings, all integers
- Membership is the key property of a set
- The Universe may be infinite but sets for representation are assumed to be finite
- **No duplicates allowed**
- Many computer algorithms ask the question is $x \in S$ abstractly
  - Is this value in the lookup table?
  - Is this person in the database?

Operations on a Set

- **member:** does *x* belong to *S*
- **union:** every element in both sets
- **intersection:** the set that contains all elements of A that are also in B, or all elements of B that are also in A, but no other elements.



Operations on a Set

- **difference:** the set of all *x* in *S* not in *T* (S - T)
- **makeEmptySet:** delete all elements from a set
- **isEmptySet:** check to see if a set is empty
- **size(S):** return how many elements are in a set
- **insert:** insert *x* into S
- **delete:** delete *x* from S
- **equal:** Is S = T?
- **iterate:** perform some operation over all members of the set

Operations on a Set

- Usually want to use a pair (K,I) for *insertions*, *deletions*, and *testing membership* where K is some key and I is the information associated with that key
  - Example: Is John Doe in the phone book, returns the following key and value - <John Doe, 555-555-1212>
- For (K,I) pairs require a lookup operation rather than membership
  - Example: Lookup(K,S) - Given a key K return info I such that (K,I) ∈ S; if K doesn't exist in S return NULL
- **Dictionary:** an abstract data type with only *insert*, *delete*, *MakeEmptySet*, *IsEmptySet*, and *Lookup*

Table Lookup

- If we had 500 different records could assign an index between 1 and 500 and use that to search
- Searching for a value in a data structure or algorithm that uses **comparisons** means the best you can do is **O(*logn*)**
- If you could do **key-indexed searches** that uses the key as an array index, this would be **faster** than comparison methods

Why Hashing?

- ➔ Devise an algorithm for printing the first repeated character in a string
- ➔ **Brute force:** $O(n^2)$ - walk through the string with one for loop and then another for loop doing the comparison with the character value to see if it is repeated or not

Hashing: A Different Approach

- ➔ Assuming the string consists of ASCII characters of which there are 128 characters
  - Create an array of size 128 and initialize it to zeros
  - For each of the characters in the string, go to the corresponding position in the array (i.e. its ASCII value) and increment its count
  - Since using an array, it takes constant time to access any location
  - While scanning for characters if count is already 1, you know you have a duplicate character in the string
- ➔ Linear search *O(n)*, binary search *O(logn)*
  - **Can we do better?**

Hashing: A Different Approach

- ➔ Tech uses Banner IDs to give each student/employee a unique ID. 900XXXXXX
- ➔ The 900 contains no information, but could use the 0 - 999,999 values as an index that stores records in an array.
  - This is not very space efficient as you have a million records and 2500 students/employees
- ➔ Could do better just use the first 0 - 9999 values (XXXX), but still 75% of your array is empty
  - Also, is it wise to give IDs in sequential order?

Hashing: A Different Approach

- Need a way to map keys (banner ids) into indexes
- Key K stored at index h(k), not k
  - h(k) is called a **hash** function.
- **Hash** functions are used to calculate the index at which the elements with the key k will be stored.
- The process of mapping the keys to appropriate locations (or indexes) in a hash table is called **hashing**

Hashing: A Different Approach

- ➔ The **goal** of using a hash function is to reduce the range of array indices that have to be handled. Instead of using the Universe (U) of values (i.e. all Banner IDs) you just need K values, reducing the storage space required.
- ➔ Note: some keys may point to the same location. This is called **collision**
- ➔ Two things you will need to hash:
  - Define the hash function
  - A method to handle the collisions