Algorithms and Architecture I

Hashing & Hash Tables

Introduction

- A hash table is an effective data structure for implementing *dictionaries* (insert, search, and delete operations).
- Worst case access time is *O(n)* but in practice the expected time is *O(1)*.
- Idea:
  - use direct addressing of arrays
  - *compute* an index from the key (i.e. hash value)
  - handle collisions with lists.

Direct-Address Tables

- Works well for a small set of (different) keys.
- Direct-address table (i.e. array) where each *slot* corresponds to a key.
- **search(T,x)**: **return** T[k]
- **insert(T,x)**: T[key(x)]:=x
- **delete(T,x)**: T[key(x)]:=NIL
- Problem with the range of the keys.

Hash Tables

- How to store if the set of keys is large?
- Use a hash function to map keys to slots
  - but collisions are possible
  - this is solved by chaining
- **insert(T,x)**:
  insert x at the head of T[h(key(x))]
- **search(T,k)**:
  search element with key k in T[h(k)]
- **delete(T,x)**: delete x from T[h(key(x))]

Hash Functions

- What makes a good hash functions?
- BTW: so far we are cheating because we know in advance the set to store.
- If we know in advance the keys then it is possible to construct a perfect hash function and hash tables so that it works well.
- But what if we don't know the keys? or even the number of elements to be stored?

In Practice

- If you know almost nothing on the elements to be stored, i.e., size, number, etc...
- Need for a fast good hash function, maybe several ones.
- Need for dynamic hash tables.
- Good examples at: http://burtleburtle.net/bob/hash/
- Good if you can have the size of the hash table a power of 2.