CSCI203

Week 5 – Lecture B

Faster Searching

- Say we have a set of data consisting of pairs:
  - E.g. Name and Telephone Number.
- How do we find the number associated with a given name?
  - What is the best data structure to use?
  - Assume that there are n pairs of data.
- Linear list:
  - Look at every entry.
  - $\Theta(n)$.
- BST:
  - Traverse the tree from the root.
  - $\Theta(\log n)$.
  - $O(n)$ worst case…
    - …so use a balanced tree.

Even Faster

- Can we do better than $\Theta(\log n)$?
- How about $\Theta(1)$?
  - Constant time searching.
- To do this we use a dictionary:
  - Map;
  - Hash table.
- This is a data structure that allows you to determine:
  - Whether a key is present;
  - If a key is present what its associated data is.

Operations on Dictionaries

- Given a dictionary **`D`**, with contents consisting of pairs of the form **`<key: value>`**, we require the following operations to be defined.
  - Insert:
    - **`D[key]=value`**
  - Delete:
    - **`delete(D[key])`**
  - Search:
    - **`value=D[key]`**, **`value == nil`** if **`key`** has not been stored.
  - Note that the dictionary behaves like an array with non-integer index.

Ubiquity

- Dictionaries form a part of every modern computer language:
  - C++:
    - ```
      Std::map<key_type, value_type> dictionary_name;
      ```
  - Java:
    - ```
      Map dictionary_name = new Hashtable();
      ```
    - ```
      Map dictionary_name = new HashMap();
      ```
    - ```
      Map dictionary_name = new LinkedHashMap();
      ```
  - Python:
    - Dictionary data type – created by reference.
    - E.g.
      ```
      en_fr = {"red" : "rouge", "green" : "vert",
               "blue" : "bleu", "yellow":"jaune"}
      ```

Motivation

- Dictionaries are used in many applications:
  - Databases;
    - Fast access to record given key
  - Compilers;
    - Maintenance of symbol table
  - Network routers;
    - Looking up IP address
  - String matching
    - Genetic analysis.
  - Security
    - Password checking.

Implementation

- There are several ways to implement the dictionary data type:
- Let us start with the simplest (and, in most cases, worst) approach:
  - The Direct Access Table:

Implementation 0: The Direct Access Table

- This is simply a big array where the index of the array is the key and the contents of the array is the value.
- Only works if keys are integers
  - E.g. key = phone number, value = name.
- So, should we use it in this case?
  - Typical phone number:
    - +61 2 4221 5576
    - 11 digits – one hundred billion possible entries
    - 20 characters per name
    - Two terabytes of storage
      - For 100,000,000,000 numbers
      - For 100 numbers!
  - If $\mathfrak{U}$ is the universe of keys $n = |\mathfrak{U}|$.

0	**value$_0$**
1	**value$_1$**
2	**-**
3	**-**
4	**value$_4$**
...	**...**
n	**value$_n$**
***n*+1**	**-**

Fixing the Problems

- Problem 1: Keys must be (non-negative) integers.
- Solution: define a function **`prehash(key): integer`**
  - This function, when given a key of whatever type we need to store returns a non-negative integer value.
  - So **`D[key]=value`** becomes **`T[prehash(key)]=value`**.
    - **`T`** is the direct access table we are using to implement the dictionary, **`D`**.
- Hold on!
  - That was too easy!
  - What exactly does **`prehash()`** do?

Implementing **`prehash()`**

- In theory:
  - Every piece of data in a computer is a sequence of bits
  - Every sequence of bits can be interpreted as a non-negative integer.
  - Problem solved!
  - Really?
  - Consider 8-character keys:
    - 8 characters = 64 bits
    - Does this mean we need an array with 2^{64} entries?

Implementing **`prehash()`**

- In Practice:
  - There are many different possible prehash functions.
  - Ideally:
    - **`prehash(`*x*`)=prehash(`*y*`)`** $\Leftrightarrow$ ***x*=*y***
    - This is not usually always true, sometimes two different kays may have the same prehash value.
    - For the sake of simplicity we will assume that the above relationship holds.

Fixing the Problems

- Problem 2:
  - Direct access tables are huge!
    - Phone numbers:- 2^{11} records
    - 8-letter words:- 2^{64} records
  - Clearly this is a BAD THING™
  - The problem here is the size of the universe of possible keys $|\mathfrak{U}|$.
- Solution: Hashing.
  - Reduce the (huge) universe of all possible keys down to a manageable size, m.
  - Our table will be of size m.
  - We have a hash function h so that $0 \leq h(key) < m$ for all valid keys.

Hashing

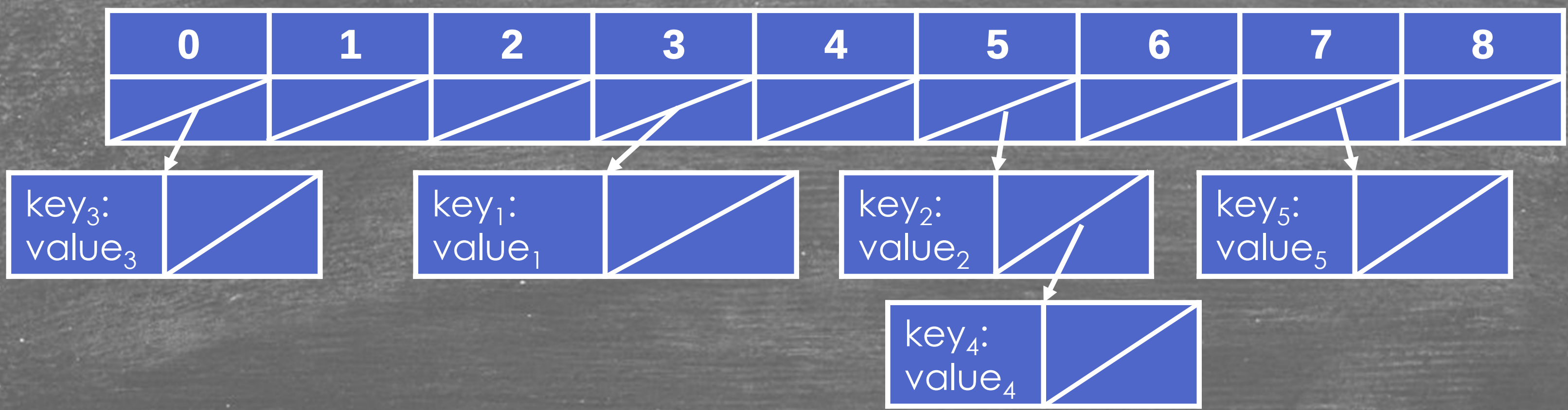
- Ideally, if we have n keys with associated values, we would like $m \in \Theta(n)$.
  - $m = 2n$, $m = 3n$.
- This presents a problem:
  - Although $m > n$, the number of keys we are storing, it is far smaller that the number of *possible* keys.
  - There will always be circumstances where $key_1 \neq key_2$ but $h(key_1) = h(key_2)$.
  - This leads to a collision:
    - Two different keys with the same hash value;
    - Two different keys with the same location in the table.
  - How do we fix this?

Chaining

- One simple solution to the collision problem is *chaining*.
- If two keys hash to the same value store both the records in the same location:
  - As a list!
  - Yes, I really said, as a list.
  - A linked list.
  - A *dynamic* data structure.

An Example: Hashing With Chaining

- Consider the following data:
  - <key$_1$:value$_1$, key$_2$:value$_2$, key$_3$:value$_3$, key$_4$:value$_4$, key$_5$:value$_5$)
- With hash values:
  - h(key$_1$) = 3, h(key$_2$) = 5, h(key$_3$) = 0, h(key$_4$) = 5, h(key$_5$) = 7, m=9.
- Our hash table looks like this.



Worst Case

- What if h(key) has the same value for all the keys in our set?
- Our hash table has just become a complicated way of storing a single linked list!
- Access to a given key:value pair is no $\Theta(n)$.
- So should we give up on hashing?
  - No!
  - In practice this does not happen.

Hash Functions

- We still have not looked at possible hash functions.
- The following are simple approaches which often work reasonably well:

1. The Division method:
   - $h(k) = k \bmod m$
   - Good if m is prime and is not close to a power of 2 or a power of 10.
2. The Multiplication method:
   - $h(k) = a \times k \bmod 2^w >> (w - r)$
   - Here:
     - a is an arbitrary w-bit integer; (a should be odd and not close to a power of 2)
     - w is the word length of the machine you are using;
     - >> is the right-shift operator;
     - $m = 2^r$.

Hash Functions continued

3. Universal Hashing:
   - $h(k) = (a \times k + b \bmod p) \bmod m$
   - Here:
     - p is a prime number, bigger than $|\mathfrak{U}|$, the number of all possible keys.
       - Yes, p is BIG!
     - a and b are random integers between 0 and p-1.
- This is an excellent hash function.
- The worst case probability of two keys colliding is $1/m$.
- This means that, even if a and b are poorly chosen, this hash function will always work well.
- The problems of finding a large prime and performing arithmetic on big integers will be left for now.