Hash Tables (continued)

Reminder

Examples

Tirgul 9

Hash Table

In a hash table, we allocate an array of size m, which is much smaller than |U| (the set of keys).

We use a hash function h() to determine the entry of each key.

The crucial point: the hash function should “spread” the keys of U equally among all the entries of the array.

The division method:

If we have a table of size m, we can use the hash function $h(k) = k \mod m$.

How to choose hash functions

The crucial point: the hash function should “spread” the keys of U equally among all the entries of the array.

Unfortunately, since we don’t know in advance the keys that we’ll get from U, this can be done only approximately.

Remark: the hash functions usually assume that the keys are numbers. We’ll discuss next class what to do if the keys are not numbers.

The division method

A good choice example:

If we have |U|=2000, and we want each search to take (on average) 3 operations, we can choose the closest primal number to $2000/3$, $m=701$.

The multiplication method

The disadvantage of the division method hash function is:

It depends on the size of the table.

The way we choose m affect the performance of the hash function.

The multiplication method hash function does not depend on m as much as the division method hash function.

The multiplication method

The multiplication method:

Multiply a constant $0<A<1$ with k.

The fractional part of kA is taken, and multiplied by m.

Formally, $h(k) = \lfloor m(kA \mod 1) \rfloor$.

The multiplication method does not depends as much on m since A helps randomizing the hash function.

In this method the are better choices for A of course…
The multiplication method

- A bad choice of $A$, example:
  - if $m = 100$ and $A=1/3$, then
  - for $k=10$, $h(k)=33$;
  - for $k=11$, $h(k)=66$;
  - And for $k=12$, $h(k)=99$.
  - This is not a good choice of $A$, since we’ll have only three values of $h(k)$...

- The optimal choice of $A$ depends on the keys themselves.

- Knuth claims that $A = \left(\sqrt{5} - 1\right) / 2 = 0.6180339887...$ is likely to be a good choice.

What if keys are not numbers?

- The hash functions we showed only work for numbers.

- When keys are not numbers, we should first convert them to numbers.

- A string can be treated as a number in base 256.
  - Each character is a digit between 0 and 255.

- The string “key” will be translated to $((\text{int} k) \times 256^0 + (\text{int} e) \times 256^1 + (\text{int} y) \times 256^2)$

Translating long strings to numbers

- The expression we reach is:
  $(((w \times 256 + o) \mod m) \times 256 + r) \mod m \times 256 + d) \mod m$

- Using the properties of mod, we get the simple alg.:
  ```
  int hash(String s, int m)
  
  int h=s[0]
  for ( i=1 ; i<s.length ; i++)
    h = ((h*256) + s[i]) \mod m
  
  return h
  ```

- Notice that $h$ is always smaller than $m$.

- This will also improve the performance of the algorithm.

Translating long strings to numbers

- The disadvantage of the method is:
  - A long string creates a large number.
  - Strings longer than 4 characters would exceed the capacity of a 32 bit integer.

- We can write the integer value of “word” as $(((w \times 256 + o) \times 256 + r) \times 256 + d)$

- When using the division method the following facts can be used:
  - $(a+b) \mod n = ((a \mod n) + b) \mod n$
  - $(a\times b) \mod n = ((a \mod n) \times b) \mod n$

Collisions

- What happens when several keys have the same entry?
  - Clearly it might happen, since $U$ is much larger than $m$.

- Collision.

- Collisions are more likely to happen when the hash table is almost full.

- We define the “load factor” as $\alpha = n / m$
  - Where $n$ is the number of keys in the hash table,
  - And $m$ is the size of the table.
There are two approaches to handle collisions:
- Chaining.
- Open Addressing.

Chaining:
- Each entry in the table is a linked list.
- The linked list holds all the keys that are mapped to this entry.
- Search operation on a hash table which applies chaining takes $O(1 + \alpha)$ time.

This complexity is calculated under the assumption of uniform hashing.

Notice that in the chaining method, the load factor may be greater than one.

In this method, the table itself holds all the keys.

We change the hash function to receive two parameters:
- The first is the key.
- The second is the probe number.

We first try to locate $h(k,0)$ in the table.

If it fails we try to locate $h(k,1)$ in the table, and so on.

It is required that $\{h(k, 0),...,h(k,m-1)\}$ will be a permutation of $\{0,..,m-1\}$.

After $m-1$ probes we’ll definitely find a place to locate $k$ (unless the table is full).

Notice that here, the load factor must be smaller than one.

There is a problem with deleting keys. What is it?

While searching key $i$ and reaching an empty slot, we don’t know if:
- The key $i$ doesn’t exist in the table.
- Or, key $i$ does exist in the table but at the time key $i$ was inserted this slot was occupied, and we should continue our search.

We will discuss two ways to implement open addressing:
- linear probing
- double hashing

The problem: primary clustering.

If several consecutive slots are occupied, the next free slot has high probability of being occupied.

Search time increases when large clusters are created.

The reason for the primary clustering stems from the fact that there are only $m$ different probe sequences.
Open addressing

- Double hashing:
  \[ h(k,i) = (h_1(k) + ih_2(k)) \mod m \]
  - Better than linear probing.
  - The problem \( h_1(k) \) can not have a common divisor with \( m \) (besides 1).
  - \( m^2 \) different probe sequences!

Performance (without proofs)

- Insertion and unsuccessful search of an element into an open-address hash table requires \( 1/(1-\alpha) \) probes on average.

- A successful search: the average number of probes is
  \[ \frac{1}{\alpha} - \frac{1}{1-\alpha} \]

  For example:
  - If the table is 50% full then a search will take about 1.4 probes on average.
  - If the table 90% full then the search will take about 2.6 probes on average.

Example for Open Addressing

- A computer science geek goes to a sibyl.
- She ask him to scramble the Tarot cards.
- The geek does not trust the sibyl and he decides to apply open addressing as scrambling technique.
- The card numbers: 10, 22, 31, 4, 15, 28, 17, 88.
- He tries Linear probing with \( m=11 \)
  and \( h_1(k) = k \mod m \).

  \[
  \begin{array}{cccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  22 & 88 & 4 & 15 & 28 & 17 & 31 & 10
  \end{array}
  \]

  He gets primary clustering which known to be bad luck...

Example for Open Addressing

- Just before the sibyl looses her patience he tries double hashing with \( m=11 \), \( h_2(k) = 1 + (k \mod (m-1)) \),
  and \( h_1(k) = k \mod m \).

  \[
  \begin{array}{cccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  22 & 17 & 4 & 15 & 28 & 88 & 31 & 10
  \end{array}
  \]

When should hash tables be used

- Hash tables are very useful for implementing dictionaries if we don’t have an order on the elements, or we have order but we need only the standard operations.
- On the other hand, hash tables are less useful if we have order and we need more than just the standard operations.
  - For example, last[], or iterator over all elements, which is problematic if the load factor is very low.

When should hash tables be used

- We should have a good estimate of the number of elements we need to store
  - For example, the huji has about 30,000 students each year, but still it is a dynamic d.b.

- Re-hashing: If we don’t know a-priori the number of elements, we might need to perform re-hashing, increasing the size of the table and re-assigning all elements.