Tirgul 7

Hash Tables

Motivation

Find an efficient implementation of a dynamic collection of elements with unique keys
Supported Operations: Insert, Search and Delete.
The keys belong to a universal group of keys, U = {1 . . . M}.

Direct Address Tables – An array of size m. An Element with key i is mapped to cell i.
Time-complexity: O(1)

What might be the problem?

If the range of the keys is much larger than the number of elements a lot of space is wasted
- might be too large to be practical.

Hash Tables

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.
What property should a good hash function have?

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

An example of a hash function:

$$h(k) = k \mod m$$

Hash choices

- Assume we want to hash strings for a country club managing software, each string contains full name (e.g. “Parker Anthony”).

- Suggested hash function : the ascii number of the first symbol in the string (e.g. h(“Parker Anthony”) = ascii of ‘h’ = 80.

- Is this a good hash function?
- No, since it is likely that many families register together. This means that we will have many collisions even with few entries.

- We would like to have something “random looking”.
- The exact meaning of this will be explained next week, however, it is important to think of this point when choosing a hash function.

How to choose hash functions

Can we choose a hash function that will spread the keys uniformly between the cells?

Unfortunately, we don’t know in advance what keys will be received, so there can always be a ‘bad input’.

We can try to find a hash function that usually maps a small number of keys to the same cell.

Two common methods:

- The division method
- The multiplication method

The division method

- Denote by *m* the size of the table.
  The division method is : $h(k) = k \bmod m$
- What is the value of *m* we should choose?
- For example:
  |U|=2000, if we want each search to take three operations (on the average), then choose a number close to 2000/3 ≈ 666.
- For technical reasons it is preferable to choose m as a prime (for a prime p Z_p is a field).
- In our example we choose m = 701.

The multiplication method

- The multiplication method:
  - Multiply k by a constant 0<A<1.
  - Take the fractional part of kA
  - Multiply by m.
  - Formally, $h(k) = \lfloor m(kA \bmod 1) \rfloor$
- The multiplication method does not depends as much on m since A helps randomizing the hash function.
- Which values of A are good choices? Which are bad choices?

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 $A \approx (\sqrt{5}-1)/2 = 0.6180339887...$ 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.
- How can we convert a string into a number?
- 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

$$((\text{int})'k')\times 256^2 + ((\text{int})'e')\times 256^1 + ((\text{int})'y')\times 256^0$$

Translating long strings to numbers

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

- How can this problem be solved when using the division method?
- We can write the integer value of “word” as (((w* 256 + o)*256 + r)*256 + d)

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

Translating long strings to numbers

- The expression we reach is:
  - ((((((w*256+o)mod m)*256)+r)mod m)*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.

The Birthdays Paradox

- There are 28 people in the room. Assuming that birthdays are distributed uniformly over the year, how many pairs of people who share birthdays will you expect?
- Surprisingly, the expected number of such pairs is approximately 1.
- Analysis: There are n people. The probability that two people share birthdays is 1/365.
- The number of pairs is $\binom{n}{2} = \frac{n(n-1)}{2}$

- So – the expected number of pairs with the same birthday is $\binom{n}{2} = \frac{n(n-1)}{2*365}$

Collisions

- Can several keys have the same entry?
  - Yes, since U is much larger than m.
- **Collision** - several elements are mapped into the same cell.
- Similar to the birthdays paradox analysis.

- When are collisions more likely to happen?
- When the hash table is almost full.

We define the "load factor" as $\alpha = n/m$.
  - n - the number of keys in the hash table
  - m - the size of the table

- How can we solve collisions?

Chaining

- 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.

Chaining

- This complexity is calculated under the assumption of uniform hashing.

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

Analysis:

Insert - O(1)

Unsuccessful search - calculating the hash value – O(1), Scanning the linked-list – expected O(α). **Total:** O(1 + α).

Successful search – Let x be the i'th element inserted to the hash table.

When searching for x, the expected number of scanned elements is (i-1)/m.

For any element - the expected number of scanned elements is O(α). **Total:** O(1 + α).

Open addressing

- 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.

Open addressing

- 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? How can it be solved?

Open addressing

- 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.

What functions can we use to implement open addressing?

- We will discuss two ways :
  - linear probing
  - double hashing

Open addressing

- Linear probing - $h(k,i)=(h(k)+i)$ mod m
  - 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.
  - How can we change the hash function to avoid clusters?

Open addressing

- Double hashing –

$$h(k,i)=(h_1(k)+ih_2(k)) \bmod m$$

  - Better than linear probing.
  - It's best if for all k we have that $h_2(k)$ does not have a common divisor with m.
  - m^2 That way different probe sequences!

Performance

- Insertion and unsuccessful search : 1/(1- α) probes on average.
- Intuition: We always check at least one cell. In probability α this cell is occupied. In probability ≈ α² (n/m * (n-1)/m) the next cell is also occupied.
- Expected probes number ≈ 1 + α + α² + α³ +… = 1/(1- α).

- A successful search: (1/ α)ln(1/(1- α)) probes on average
- Idea: When searching for x, the probes number is the same as when the element was inserted (depends on the number of elements that were inserted before x). How to get to the formula look in CLR.

- 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

- Lets look at an example:
- First assume we use linear probing
- insert the numbers: 4, 28, 6, 38, 26 with m=11

  and h(k)=k mod m.

  [ ] [ ] [ ] [ ] [4] [38] [28] [6] [26] [ ] [ ]

  0 1 2 3 4 5 6 7 8 9 10
- Note the clustering effect on 26!

Example for Open Addressing

- Now lets try double hashing, again m=11
- Use:

  $h_1(k)$=k mod m

  $h_2(k)$=1 + k mod (m-1) .

  h(k)= ($h_1(k)$ + i*$h_2(k)$) mod m

- Insert numbers: 4, 28, 6, 38, 26

  [26] [ ] [6] [ ] [4] [38] [28] [ ] [ ] [ ] [ ]

  0 1 2 3 4 5 6 7 8 9 10
- The clustering effect disapeard.

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.

DAST 2004

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.