Ian Piper
CSCI203

Algorithms and Data Structures

Week 6 – Lecture A

Hashing – Picking *m*

Hashing: Picking *m*

- As we saw last week we want m, the number of slots in the dictionary, to be $\Theta(n)$, where n is the number of entries in the dictionary.
- Remember: operations on a dictionary are $O(1+n/m)$, so if n grows too large we get less and less efficient.
- The problem we face is that, often, we do not know how many records we will need to store.
  - If m is too small, the dictionary becomes inefficient.
  - If m is too large, we waste storage (memory or disc).
- How do we get the right value for m?
- Let's say we want $m \geq n$ at all times.

Lucky Guess?

- If we have no knowledge of the ultimate size of *n*, what can we do?
  - Guess.
  - Pick *m* based on an optimistic assessment of the likely size of *n*.
  - No idea?
    - Pick your favourite small number ☺.
    - *m* = 8, say.
  - Now what?
    - What if *n* turns out to be greater than 8?
  - Make *m* bigger.
    - How much bigger?

Changing *m*

- Hang on a sec.
- If we change *m* we have problems:
  - Our hash array is too small.
  - Our hashed keys will be wrong.
    - They depend on the value of *m*.
- Does this mean that we have to recreate the hash table from scratch?
  - It sure does.
- Isn't this a BAD THING™?

Growing a Hash Table

Growing a Hash Table

- What exactly has to happen if we change m?
  - Let's say the new table size is m'.
- We now need a new array with m' elements.
  - We also need to move all of the existing elements from the old table to the new one.
- Build a new hash function h'.
  - Remember, the hash function depends on m.
- Insert the existing data into the new table.
  - This involves re-hashing every key.
- So, the first question is:
  - How much do we grow m?

$m' = ?$

- Each time we grow the table we perform $\Theta(m+n+m')$ operations.
  - This is $\Theta(n)$.
- Let's look at some options:
  - $m' = m+1$.
    - What is the cost of n insertions?
      - $\Theta(1)$ for the first m insertions.
      - $\Theta(m')$ for each insertion after that.
    - Overall $\Theta(n^2)$

$m' = ?$

- $m' = 2m$
  - $\Theta(1)$ for the first m insertions.
  - $\Theta(m)$ for the next insertion.
  - $\Theta(1)$ for the next $m-1$ insertions.
  - $\Theta(2m)$ for the next insertion.
  - $\Theta(1)$ for the next $2m-1$ insertions.
- Overall $\Theta(n+(n/2)+(n/4)+\ldots) = \Theta(2n) = \Theta(n)$
- The cost of expanding the table gets spread over the extra elements we are making room for.
- This is known as *Amortized* cost.
- Note: an amortized cost of $\Theta(1)$ per operation does not mean that every operation has this cost.
  - Just that this is the average cost per operation.

Amortized Cost

Amortized Cost

- We say an operation has a cost of "T(k) Amortized" if k operations take a total of $k \times T(k)$ time.
- Table doubling takes $\Theta(n)$ operations for n insertions so the amortized cost is $\Theta(1)$.
- This is, actually, a GOOD THING™.
- Note: we can use table doubling to implement any solution where we do not know the size of the data structure in advance and it grows in a "well behaved" way.
- Table doubling minimizes the cost associated with dynamic data structures.

Deletions

- What about deletions?
  - Each deletion is still $\Theta(1)$.
  - They simply increase the number of operations (insertions and deletions) we can perform between doublings.
- What if it's all deletions?
  - In this case the table becomes progressively less and less full.
  - Solution: Shrink the table.
- How, exactly?

Shrinking a Hash Table

Shrinking Tables

- What should our strategy for reducing the size of the table be?
- How about "if $n < m/2$ make $m' = m/2$"?
  - What if the next operation is an insertion?
    - Double the table size!
      - Then a deletion?
        - Halve the table!
          - Insertion?
            - Double…
    - We now have $\Theta(n)$ operations for each change in the data.
- Instead use "if $n < m/4$ make $m' = m/2$".

Constant Time?

- Although, in our example, T(n) is in $\Theta(1)$; for some operations the actual cost is in $\Theta(n)$.
- What does this mean if we have a real-time application?
  - Every so often we get an insertion /deletion that takes a really looooong time.
  - Can we remedy this so that **every** operation is in $\Theta(n)$?
- The answer is Yes!
  - We simply adopt the following strategy…
  - …when a table starts to become full—perform the table doubling in the background.
  - Keep two sets of the data until you either actually need the double size or until the panic is over.

Hashing With Chaining Considered Bad

- There is still one small issue with this method.
  - We have a hybrid data structure—an array of linked lists.
- A second approach uses just a simple array.
- Clearly, we still have a potential problem with collision—two keys which hash to the same value.
- We resolve this with a technique known as *Open Addressing*.

Open Addressing

An alternative to chaining

Open Addressing

- We wish to hash n items into an array with m slots.
- We may only store one item per slot.
- Clearly, $m \geq n$.
- We insert an item into the table using an iterative technique known as *probing*.

Probing

- This process works as follows: (for insertion)
- ```
  Set hash function to starting value, h0
  repeat
       calculate probe=hash(key)
       if table(probe) contains data then
              go to the next hash function
       else
              store the item in table(probe)
       fi
  until we have stored the item
  ```
- This means we must have a sequence of hash functions, h0, h1, h2…
- … or a hash function which produces a sequence of values.
  ```

The Hash Function

- Our new hash function requires two arguments:
  - The key;
  - The iteration count.
- Thus: **`probe=OpenHash(key, count)`**
- Here:
  - **`key`** is a valid element of `U`, the universe of keys;
  - **`count`** is a non-negative integer.
- As usual, $0 \leq$ **`probe`** $< m$-1.

- In addition, we want our hash function to have the following property:
- For any arbitrary key k the sequence of m probes:
  - $h(k, 0), h(k, 1), h(k, 2), \ldots, h(k, m-1)$;
- Must be a permutation of the integers:
  - $0, 1, 2, \ldots, m-1$.
- This property guarantees that we must eventually find a vacant slot to insert the item into.
- Clearly, the sequence of probes must be different for different keys.
- We can see this with an example.

Example: Insertion with Open Addressing

- Consider the following table:

k	h(0,k)	h(1,k)	h(2,k)	h(3,k)	h(4,k)	h(5,k)	h(6,k)	h(7,k)	h(8,k)	h(9,k)
899	9	8	5	6	0	7	8	2	4	1
950	5	7	4	9	2	3	1	6	8	0
12	3	8	7	2	5	9	1	6	0	4
367	7	1	2	3	4	5	6	8	9	0
359	2	1	9	5	6	7	3	8	0	4
980	4	7	1	8	9	3	0	5	2	6
229	0	8	2	7	1	6	3	9	4	5
598	8	6	3	5	0	7	9	1	4	2
838	6	2	6	7	1	3	8	2	0	2
549	9	8	4	6	7	5	0	1	2	3

- Let us insert the keys into our hash table in order

k	h(0,k)	h(1,k)	h(2,k)	h(3,k)	h(4,k)	h(5,k)	h(6,k)	h(7,k)	h(8,k)	h(9,k)
899	9	8	5	6	0	7	8	2	4	1
950	5	7	4	9	2	3	1	6	8	0
12	3	8	7	2	5	9	1	6	0	4
367	7	1	2	3	4	5	6	8	9	0
359	2	1	9	5	6	7	3	8	0	4
980	3	7	1	8	9	4	0	5	2	6
229	0	8	2	7	1	6	3	9	4	5
598	8	6	3	5	0	7	9	1	4	2
838	6	2	4	7	1	3	8	2	0	2
549	9	8	4	6	7	5	0	1	2	3

0	1	2	3	4	5	6	7	8	9
229	980	359	12	549	950	838	367	598	899

Search with Open Addressing

- The procedure used to search using open addressing is similar to insertion.
- ```
  count=0
  repeat
       probe=hash(key, count)
       if table(probe)==key then
             return item
       else
             count++
       fi
  until table(probe)==empty or count==n
  return not found
  ```
- This is pretty straightforward.
  ```

Deletion with Open Addressing

- When we get to deletion we have a new problem.
- ```
  count=0
  repeat
      probe=hash(key, count)
      if table(probe)==key then
            delete item
            return
      else
            count++
      fi
  until table(probe)==empty or count==n
  return not found
  ```
- How, exactly, do we delete the item?
  ```

Deletion…

- If we simply replace the item with our empty value we will have an issue:
  - What if the key we next search for is after the probe corresponding to the deleted key's location.
  - If, in our previous example, we delete 899, where h(899,0)=9, and then search for 549, where the sequence of hash values are 9, 8, 4…
    - We test D(9) and discover it has the value **`empty`**.
    - We conclude that 549 is not in the table.
    - Wrong! It is in D(4).
- To fix this we need a second special value, **`deleted`**.

Deletion concluded

- Our deletion process becomes:
  - ```
    count=0
    repeat
       probe=hash(key, count)
       if table(probe)==key then
             table(probe)==deleted
             return
       else
             count++
       fi
    until table(probe)==empty or count==n
    return not found
    ```
- This fixes search but introduces a problem with insertion.
    ```

Insertion Revisited.

- We note that we can insert a new item into the dictionary in two circumstances:
  - **`D(i)==empty`**
  - **`D(i)==deleted`**
- We modify our insert process as follows:
  - ```
    count=0
    repeat
       probe=hash(key, count)
       if table(probe)==empty or table(probe)==deleted then
              store item in table(probe)
              return
       else
              count++
       fi
    until count==n
    return no room
    ```
- Now we can insert into the first vacant slot, empty or deleted, that we find in the table.
    ```

Search Revisited

- Because **`empty`** and **`deleted`** are different, we do not have to modify our search procedure.
- The search will skip over deleted records because they do not match the key but will still terminate when it reaches an empty record.

Hash Functions

Open Addressing Hash Functions

- One question remains.
- Can we find a function h(*k*, *i*) which is:
  - Easy to compute;
  - Produces a permutation of {0, 1, …, *m*-1} as *i* varies over {0, 1, …, *m*-1}?
- Let us examine two possible strategies.

Strategy I: Linear Probing

- In this approach we simply take a standard hash function, h(k) and compute the probe p(k, i) as follows:
  - p(k, i)=(h(k)+i) mod m
- In other words, we simply look at sequential entries in the dictionary starting at the entry corresponding to h(k).
  - This is certainly easy to compute.
  - It does satisfy the permutation.
- Is it any good?
  - No!
  - It produces sets of consecutive occupied slots.
  - Clustering.
- The bigger the cluster, the more likely it is to be hit..
  - …and it gets even bigger!

Strategy II: Double Hashing

- In this strategy we have two standard hash functions, $h_1(k)$ and $h_2(k)$.
- We compute $p(k)$, our probe value as follows:
  - $p(k, i) = (h_1(k) + i \times h_2(k)) \bmod m$.
- Do we still satisfy our requirements?
  - This is still easy to compute.
  - Do we always get a permutation?
    - No.
    - Unless we are clever in how we define h_2.

Choosing h_2

- We need $h_2(k)$ to be relatively prime to m.
- I.e. $h_2(k)$ and m must have no common factors except 1.
- This is easy in many cases.
- If we select m to be a power of 2; say $m = 2^r$ then all we need is for $h_2(k)$ to always be an odd number.
- For example, if we have a standard hash function $h'(k)$, we can create $h_2(k)$ as follows:
  - $h_2(k)=(2h'(k)+1) \bmod m$

Table Doubling.

- Once again, we need to expand the dictionary whenever it becomes too full.
- What does "too full" mean in this case?
- We define the occupancy of a table, α, to be the ratio of *n*, the number of entries to *m*, the number of slots.
  - $\alpha = n/m$
  - $0 \leq \alpha \leq 1$
- We can show that the average cost of an operation on a table with occupancy α is in $\Theta(1/(1-\alpha))$.
- In practice we want this value to be reasonably close to 1 so we double as soon as α exceeds 0.5 or thereabouts.
- This keeps operations between $\Theta(1)$ and $\Theta(2)$.

An Important Note on α

- When calculating the occupancy value, α, we must count slots with a value of **`deleted`** as containing data.
- This is because some operations, notably searching, treat deleted records as still containing data.
- Slots containing **`deleted`** may be removed in two ways:
  - Being overwritten with valid data as a result of an insert operation;
  - Being cleaned up when the table is expanded.
- If we did not count **`deleted`** records in calculating α we could have a notionally empty table in which every slot was **`deleted`**.
- Search (and delete) in this table would be $\Theta(m)$, not $\Theta(1)$, as we might expect.

Chaining vs. Open Addressing

- So, which is the better scheme?
- Open Adressing:
  - Uses less memory—no need for pointers;
  - Is faster—provided α is kept below 0.5;
  - Is a little harder to implement and understand.
  - Is clean—one data structure, the array.
- Chaining:
  - Uses more memory;
  - Is faster—if we are not careful with open addressing.
  - Is a little easier to implement and understand.
  - Is a bit messy—arrays of linked lists.
- I know where my vote goes!