Algorithms in Systems Engineering
ISE 172

Lecture 12

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Chapter 5
- References
  - CLRS Chapter 11
  - D.E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (Third Edition), 1998.
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Resolving Collisions

- There are two primary of methods of resolving collisions.
- Chaining: Form a linked list of all the elements that hash to the same value.
  - Easy to implement.
  - The table never "fills up" (better for extremely dynamic tables)
  - May use more memory overall.
  - Easy to insert and delete.
- Open Addressing: If the hashed address is already used, use a simple rule to systematically look for an alternate.
  - Very efficient if implemented correctly.
  - When the table is nearly full, basic operations become very expensive.
  - Deleting items can be very difficult, if not impossible.
  - Once the table fills up, no more items can be added until items are deleted or the table is reallocated (expensive).

Analysis of a Hash Table with Chaining

- Insertion is always constant time, as long as we don't check for duplication.
- Deletion is also constant time if the lists are doubly linked.
- Searching takes time proportional to the length of the list.
- How long the lists grow on average depends on two factors:
  - how well the hash function performs, and
  - the ratio of the number of items in the table to its size (called the *load factor*).

Length of the Linked Lists

- We will assume *simple uniform hashing*, i.e., that any given key is equally likely to hash to any address.
- Let the load factor be α.
- Under these assumptions, the average number of comparisons per search is $\Theta(1+\alpha)$, the average chain length plus the time to compute the hash value.
- If the table size is chosen to be proportional to the maximum number of elements, then this is just $O(1)$.
- This result is true for both search hits and misses.
- Note that we are still searching each list sequentially, so the net effect is to improve the performance of sequential search by a factor of M.
- If it is possible to order the keys, we could consider keeping the lists in order, or making them binary trees to further improve performance.

Related Results

- It can be shown that the probability that the maximum length of any lists is within a constant multiple of the load factor is very close to one.
- The probability that a given list has more than $t\alpha$ items on it is less than

$$\left(\frac{\alpha e}{t}\right) e^{-\alpha}$$

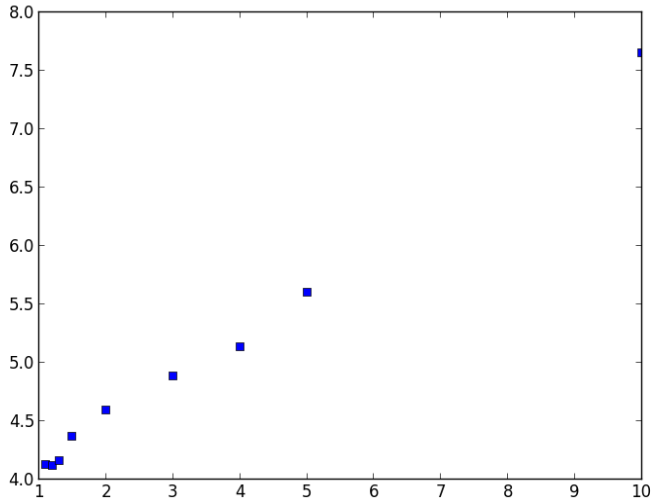
- In other words, if the load factor is 20, the probability of encountering a list with more than 40 items on it is $.0000016$.
- A related result tells that the number of empty lists is about $e^{-\alpha}$.
- Furthermore, the average number of items inserted before the first collision occurs is approximately $1.25\sqrt{M}$.
- This last result solves the classic *birthday problem*.
- We can also derive that the average number of items that must be inserted before every list has at least one item is approximately $M \ln M$.
- This result solves the classic *coupon collector's problem*.

Table Size with Chaining

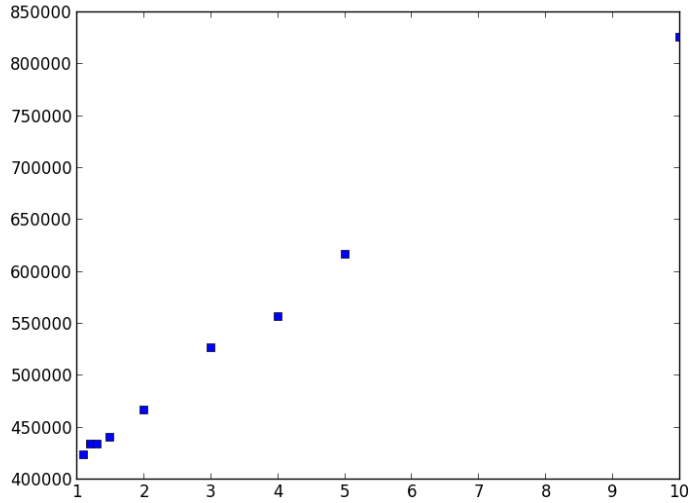
- Choosing the size of the table is a perfect example of a *time-space* tradeoff.
- The bigger the table is, the more efficient it will be.
- On the other hand, bigger tables also mean more wasted space.
- When using chaining, we can afford to have a load factor greater than one.
- A load factor as high as 5 or 10 can work well if memory is limited.

Empirical Performance of Chaining

Here is a graph showing the time and number of comparisons required to insert all the words in Dickens' Tale of Two Cities into a hash table with chaining.



(a) Insertion time



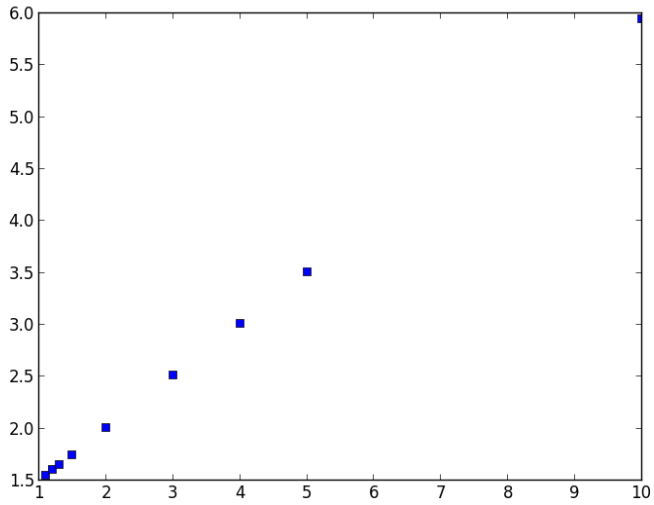
(b) Number of Comparisons

Figure 1: Performance of chaining as a function of load factor

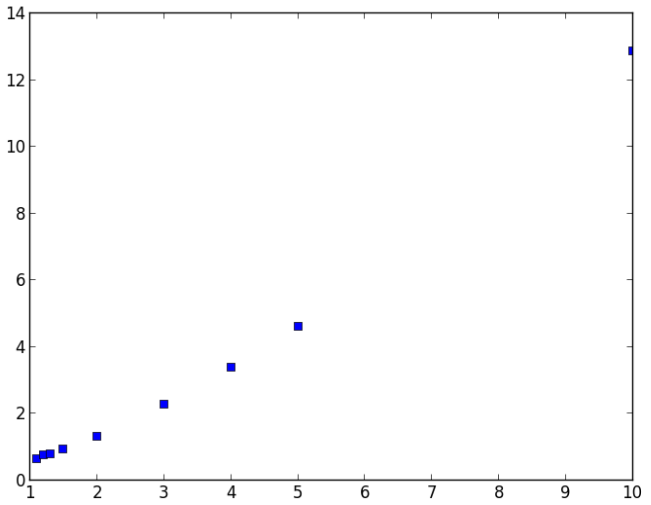
- The hash function used is the pseudo-universal one from Lecture 9.
- Note the agreement with theoretical predictions, indicating that the hash function is indeed effective.

Empirical Performance of Chaining

Here is a graph showing the mean and variance of the number of comparisons required to insert each word in the previous experiment.



(a) Insertion time



(b) Number of Comparisons

Figure 2: Performance of chaining as a function of load factor

Note here that there is an increase in variance as load factor increases, but is almost linear.

Open Addressing

- In *open addressing*, all the elements are stored directly in the hash table.
- If an address is already being used, then we systematically move to another address in a predetermined sequence until we find an empty slot.
- Hence, we can think of the hash function as producing not just a single address, but a sequence of addresses $h(x, 0), h(x, 1), \ldots, h(x, M-1)$.
- Ideally, the sequence produced should include every address in the table.
- The effect is essentially the same as chaining except that we compute the pointers instead of storing them.
- The price we pay is that as the table fills up, the operations get more expensive.
- It is also much more difficult to delete items.

Linear Probing

- In *linear probing*, we simply try the addresses in sequence until an empty slot is found.
- In other words, if h' is an ordinary hash function, then the corresponding sequence for linear probing would be

$$h(x,i) = (h'(x) + i) \mod M, i = 0, \ldots, M-1.$$

- Items are inserted in the first empty slot with an address greater than or equal to the hashed address (wrapping around at the end of the table).
- To search, start at the hashed address and continue to search each succeeding address until encountering a match or an empty slot.
- Deleting is more difficult
  - We cannot just simply remove the item to be deleted.
  - One solution is to replace the item with a sentinel that doesn't match any key and can be replaced by another item later on.
  - Another solution is to rehash all items between the deleted item and the next empty space.

Analysis of Linear Probing

- The average cost of linear probing depends on how the items cluster together in the table.
- A *cluster* is a contiguous group of occupied memory addresses.
- Consider a table with half the memory locations filled.
  - If every other location is filled, then the number of comparisons per search is either 1 or 2, with an average of 1.5.
  - If the first half of the table is filled and the second half is empty, then the average search time is $1 + (\sum_{i=1}^{n} i)/(2n) \approx n/4$.
- Generalizing, we see that search time is approximately proportional to the sum of squares of the lengths of the clusters.

Further Analysis of Linear Probing

- The average cost for a search miss is

$$1 + (\sum_{i=1}^{l} t_i(t_i + 1))/(2M)$$

  where l is the number of clusters and t_i is the size of cluster i.

- This quantity can be approximated in the case of linear probing.
- On average, the time for a search hit is approximately

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

  and the time for a search miss is approximately

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

- These approximations lose their accuracy if α is close to 1, but we shouldn't allow this to happen anyway.

Clustering in Linear Probing

- We have just seen why large clusters are a problem in open addressing schemes.
- Linear probing is particularly susceptible to this problem.
- This is because an empty slot preceded by i full slots has an increased probability, $(i+1)/M$, of being filled.
- One way of combating this problem is to use *quadratic probing*, which means that

$$h(x,i) = (h'(x) + c_1 i + c_2 i^2) \mod M, i = 0, \ldots, M-1$$

- This alleviates the clustering problem by skipping slots.
- We can choose c_1 and c_2 such that this sequence generates all possible addresses.

Double Hashing

- An even better idea is to use *double hashing*.
- Under a double hashing scheme, we use two hash functions to generate the sequence as follows.

$$h(x,i) = (h_1(x) + ih_2(x)) \mod M, i = 0, \ldots, M-1$$

- The value of $h_2(x)$ must never be zero and should be relatively prime to M for the sequence to include all possible addresses.
- The easiest way to assure this is to choose M to be prime.
- Each pair $(h_1(x), h_2(x))$ results in a different sequence, yielding M^2 possible sequences, as opposed to M in linear and quadratic probing.
- This results in behavior that is very close to ideal.
- Unfortunately, we can't delete items by rehashing, as in linear probing.
- To delete, we must use a sentinel.

Analyzing Double Hashing

- When collisions are resolved by double hashing, the average time for search hits can be approximated by

$$\frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$$

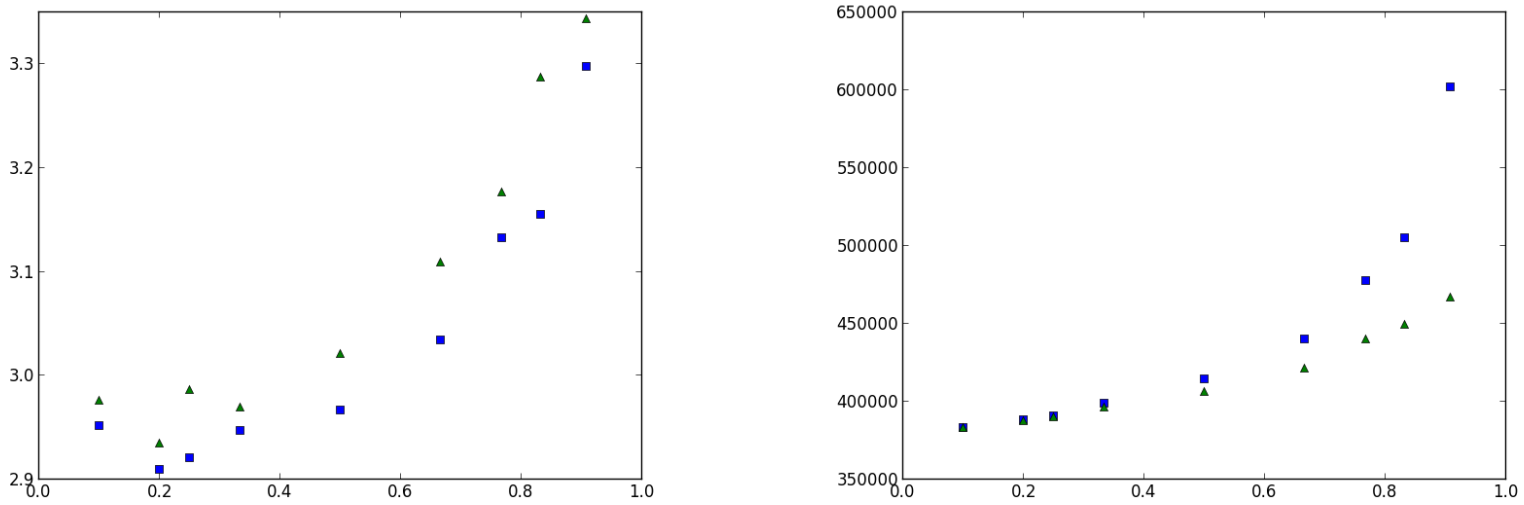
and the time for search misses is approximately

$$\frac{1}{1-\alpha}$$

- This is a big improvement over linear probing.
- Double hashing allows us to achieve the same performance with a much smaller table.

Empirical Performance of Open Addressing

Here is a graph showing the time and number of comparisons required to insert all the words in Dickens' Tale of Two Cities into a hash table with open addressing.



(a) Insertion time

(b) Number of Comparisons

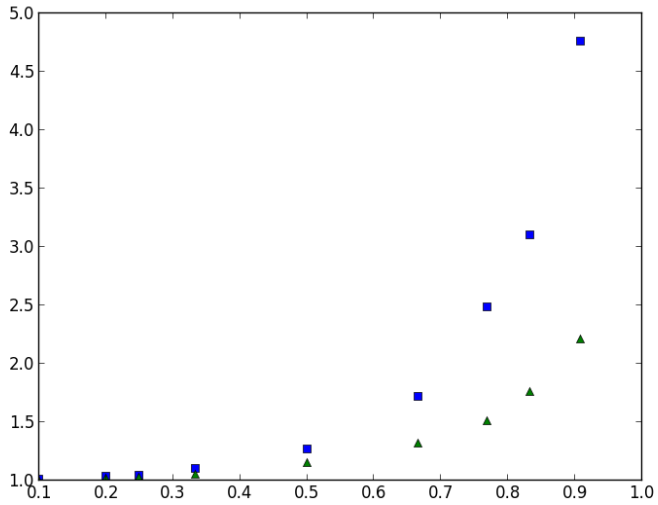
Figure 3: Performance of chaining implementation as a function of load factor

Empirical Performance of Open Addressing (cont.)

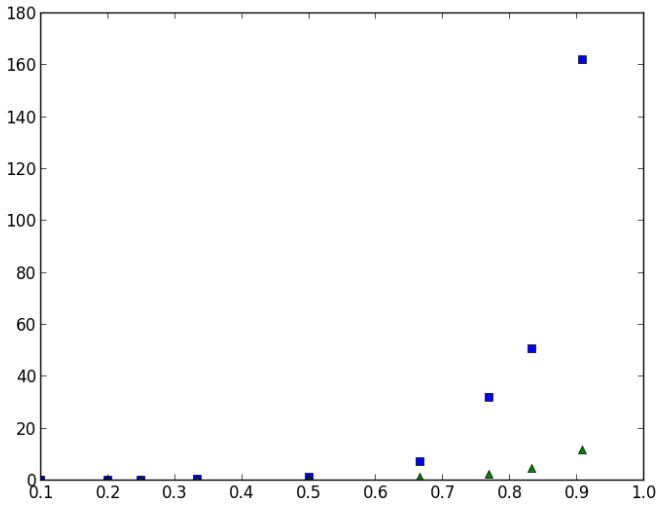
- On the previous slide, the blue squares are linear probing and the green triangles are double hashing.
- The second hash function is a variant of the pseudo-universal one from Lecture 9.
- Note the superior performance of double hasing in terms of required number of comparisons.
- This superior performance does not translate into practical improvements in running time in this case.
- This may be due to the use of an expensive second hash function.

Empirical Performance of Open Addressing

Here is a graph showing the mean and variance of the number of comparisons required to insert each word in the previous experiment.



(a) Insertion time



(b) Number of Comparisons

Figure 4: Performance of open adressing implementations as a function of load factor

Note here that there is an increase in variance as load factor increases is much more dramatic especially with linear probing, as expected.

Worst Case Analysis

- So far, we have only looked at average performance over all possible inputs.
- Particular inputs may not exhibit the nice behavior seen on average.
- As with many algorithms, worst case behavior is easy to find.
- For any hash function, there is always a sequence of inserts that will lead to poor behavior.
- For both open addressing and chaining, a sequence of n inserts could require $\theta(n^2)$ steps.
- A common way to protect against worst-case behavior in algorithms is to *randomize* in case a certain common patern leads to the worst case.
- For this purpose, we can use the universal has functions described in Lecture 9.

Dynamic Hash Tables

- *Dynamic hash tables* attempt to overcome the limitations of open addressing when the number of table items is not known at the outset.
- When the table fills up beyond a certain threshold, we simply allocate a new array and rehash all the existing items.
- This operation is expensive, but it happens infrequently.
- Using a technique called *amortized analysis*, we can show that the average cost of each operation is still approximately constant.
- This may be a good option in some situations.

Python's Hash Table Implementation

- Python uses open adressing with a variant of double hashing.

```
self.mask = newsize - 1
perturb = key_hash
while True:
    i = (i << 2) + i + perturb + 1;
    entry = self.table[i & self.mask]
    if entry.key is None:
        return entry if free is None else free
    if entry.key is key or \
            (entry.hash == key_hash and key == entry.key
        return entry
    elif entry.key is dummy and free is None:
        free = dummy
    perturb >>= PERTURB_SHIFT
```

- The table size is initially 8 and is increased by a factor of 4 whenver it fills up.
- Deletion is by sentinel.