References for Today’s Lecture

• Required reading
  – CLRS Chapter 11

• References
Hash Tables

• We now consider data structure for storing a dictionary that support only the operations

• Most data structures for storing dictionaries depend on using comparison and exchange to order the items.

• This limits the efficiency of certain operations (recall the lower bound on the efficiency of comparison-based sorting).

• A hash table is a generalization of an array that takes advantage of our ability to access an arbitrary array element in constant time.

• Using hashing, we determine where to store an item in the table (and how to find it later) without using comparison.

• This allows us to perform all the basic operations extremely efficiently.
Addressing using Hashing

• Recall the array-based implementation of a dictionary from Lecture 9.
• In this implementation, we allocated one memory location for each possible key.
• How can we extend this method to the case where the set $U$ of possible keys is extremely large?
• **Answer**: Use **hashing**.
• A *hash function* is a function $h : U \rightarrow 0, \ldots, M - 1$ that takes a key and converts it into an array index (called the *hash value*).
• Once we have a hash function, we can use the very efficient array-based implementation framework from Lab 9 to store items in the table.
• Note that this implementation no longer allows sorting of the items.
• **Questions**:
Choosing a Hash Function

• What makes a good hash function?
Significant Bits

- Two obvious hash functions are to simply consider either the first (most significant) or last (least significant) $k$ bits of the key.
  - Assume $x$ is a $w$-bit integer.
  - The index formed from the first $k$ bits of $x$ is the result of dividing by $2^{w-k}$ and rounding off, i.e., $h(x) = \lfloor x/2^{w-k} \rfloor$.
  - The index formed from the last $k$ bits of $x$ is the remainder after dividing by $2^k$, i.e., $h(x) = x \mod 2^k$.

- Note that both of these hash functions must be used with a table of size $2^k$.

- These hash functions are very fast to compute (why?).

- However, these are both notoriously bad hash functions, especially for strings (why?).
An Improved Hash Function

- The method of the previous slide can be made to work better simply by changing the size of the hash table.
- To hash a key $x$, take $x \mod M$, where $M$ is the size of the hash table.
- This is called *modular hashing* and is a very popular form of hashing.
- To avoid the problems discussed on the last slide and for reasons that will become clear later, it is best to choose $M$ to be prime.
- Choosing $M$ to be close to a power of two can also cause problems.
- In addition, we want the size of the table to be in a specified range.
- Computing a number satisfying all these requirements can be difficult.
- In practice, such numbers can be looked up in a table.
Other Simple Hash Functions

• Another approach to improving the method of significant bits is to consider the bits in the middle.

• How would we compute this hash function?

• The advantage of this method is that the value of $M$ is not as critical.

• In practice, there are many values of $A$ and $M$ that work well.

• Taking $A = \frac{\sqrt{5} - 1}{2}$ (the golden ratio) seems to work well.

• Another variation on the theme is to take $h(x) = \lfloor Ax \rfloor \mod M$. 
Resolving Collisions

- There are two primary 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
- Deletion
- Searching

How long the lists grow on average depends on two factors:
Length of the Linked Lists

• We will assume \textit{simple uniform hashing}, i.e., that any given key is equally likely to hash to any address.

• Let the load factor be $\alpha$.

• 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 \textit{hits} and \textit{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 0.0000016.

- 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.