# Algorithms in Systems Engineering
# ISE 172

## Lecture 9

Dr. Ted Ralphs

# References for Today's Lecture

- Required reading
  - Chapter 5
- References
  - CLRS Section 11.1, Chapter 12
  - D.E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (Third Edition), 1998.
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

# Symbol Tables and Dictionaries

- We have now discussed in detail the ADT for a list and have seen two implementations, as well as other data structures built on lists.
- We now consider a different kind of list data structure that supports different kinds of operations.
- A *symbol table* is a data structure for storing a list of items, each with a key and satellite data.
- This data structure supports the following basic operations.
  - Construct a symbol table.
  - Search for an item (or items) having a specified key.
  - Insert an item.
  - Remove a specified item.
  - Count the number of items.
  - Print the list of items.
- Symbol tables are also called *dictionaries* because of the obvious comparison with looking up entries in a dictionary.
- Note that the keys may not have an ordering.

# Additional Operations on Symbol Tables

- If the items can be ordered, e.g., by `__lt__` and `__eq__`, we may support the following additional operations.
  - Sort the items (print them in sorted order).
  - Return the maximum or minimum item.
  - Select the $k^{\text{th}}$ item.
  - Return the successor or predecessor of a given item.
- We may also want to be able to join two symbol tables into one.
- These operations may or may not be supported in various implementations.

# Applications of Symbol Tables

- What are some applications of symbol tables?

# Dictionary ADT

```
class Dictionary:
    def __init__()
    def __getitem__(key)
    def __setitem__(key, value)
    def __delitem__(key)
    def __contains__(key)
    def __len__()
    def get()
    def keys()
    def values()
    def items()
    def sort()
```

# Implementation

- Consider a list of items whose keys are small positive integers.
- Assuming no duplicate keys, we can implement such a symbol table using an unordered list.

```
class Dictionary:
    def __init__(self, maxKey):
        self.list = [None for i in range(maxKey)]
        self.maxKey = maxKey
    def append(self, key, value):
        self.list[key] = value
    def remove(key):
        self.list[key] = None
    def __contains__(key)
        if self.list[key] = None:
            return False
        else:
            return True
```

# Implementation (cont.)

```
def __getitem__(self, key):
    for i in self.list:
        if i != None:
            key -= 1
            if key == 0:
                return self.list[i]

def sort(self):
    pass

def __len__(self):
    count = 0
    for i in self.list
    if i != None:
         count += 1
    return count
```

# Arbitrary Keys

- Note that with arrays, most operations are constant time.
- What if the keys are not integers or have arbitrary value?
- We could still use an array or a linear linked list to store the items.
- However, some of the operations would become inefficient.
  - If we keep the items in order, searching would be efficient (binary search), but inserting would be inefficient.
  - If we kept the items unordered, inserting would be efficient, but searching would be inefficient (sequential search).
- A *hash table* is a more efficient data structure for implementing symbol tables where the keys are an arbitrary data type.

# Hash Tables

- We now consider data structure for storing a dictionary that support only the operations
  - insert,
  - delete, and
  - search.
- Most data structures for storing dictionaries depend on using comparison and exchange to order the items.
- This limits the efficiency of certain operations.
- 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 earlier.
- 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 $h : U \to 0, \ldots, M-1$ 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 to store items in the table.
- Note that this implementation no longer allows sorting of the items.
- Questions:
  - What hash function should we use?
  - What do we do if two items result in the same hash value (a *collision*)?

# Choosing a Hash Function

- What makes a good hash function?
- A good hash function minimizes collisions and is easy to compute.
- For a "random" key, we would like the probability of each hash value to be "equally likely."
- This assures that the items are distributed evenly throughout the hash table.
- This is not as easy to accomplish as it sounds!
- It depends on what the distribution of possible key values is.
- We may not know the distribution ahead of time.

# Significant Bits

- Two obvious hash functions are to simply consider either the first (most significant) or last (least significant) $k$ bits of the key.
- How do we compute this hash function?
  - 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?
  - Multiply the key by a number $A$ between 0 and 1.
  - Multiply the fractional part of the answer by $M$ and round off.
  - This can be written as $h(x) = \lfloor M(Ax \mod 1) \rfloor$.
  - If $A = 1/2^k$ and $M = 2^l$, then the result is bits $k$ through $k + l$ (very easy to compute).
- 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 $A = (\sqrt{5} - 1)/2$ (the *golden ratio*) seems to work well.
- Another variation on the theme is to take $h(x) = \lfloor Ax \rfloor \mod M$.

# Converting the Key to an Integer

# Converting the Key to an Integer

- To end up with a valid table address, we must convert the key into a natural number at some point.
- Example: Converting a string to an integer

# Converting the Key to an Integer

- To end up with a valid table address, we must convert the key into a natural number at some point.
- Example: Converting a string to an integer
  - To convert a standard 7-bit ASCII string, interpret it as an unsigned integer base 128.
  - The word `now` converts to

$$110 \cdot 128^2 + 111 \cdot 128^1 + 119 \cdot 128^0 = 1816567$$

  - The Python `ord()` function can be used to convert characters to their ASCII codes.
  - Note that using this method can result in very large numbers!
- To convert floating point numbers to integers, we can simply multiply by a large number.

# Hashing Strings

- As mentioned previously, hashing strings can be problematic because a relatively small string can convert to a huge integer.
- Example: The string "`averylongkey`" has 25 digits when converted to an integer!
- This is too large to be represented in most computers.
- With modular hash functions, we don't need to explicitly calculate the integer equivalent to obtain the hash value.
- We can calculate the result piece by piece using Horner's method.

```
def hash(str, M)
   h, a = 0, 128
   for c in str
      h = (a*h + ord(c)) % M;
   return h
```

# Universal Hashing

- A *universal hash function* is one in which the probability of a collision between any two keys is provably $1/M$.
- Implementing universal hash functions necessarily involves some randomization.
- Here are two approaches.
  - Choose the hash function randomly from a class selected to yield the desired property.
  - Randomize the hash function itself.
- These two methods amount to the same thing.
- For this to work, the randomization has to be independent of the keys.
- Generally, universal hashing isn't worth the additional computation required, but we will look at two simple universal hash function.

# A (Pseduo) Universal Hash Function for Strings

- Consider our earlier modular hash function for strings.
- One way to randomize this hash function is to randomize the value of the constant `a`.
- We will use an inexpensive pseudo-random number generator for this purpose.
- Here is our new hash function.

```
def hash(str, M)
   h, a, b = 0, 31415, 27183
   for c in str:
      h = (a*h + ord(c)) % M
      a = a+b % (M-1)
   if h < 0:
       return h + M
   else:
       return h
```

- This idea can be extended to integers by multiplying each byte by a random coefficient in much the same fashion.

# Other Applications of Hashing

- Ensuring integrity of transferred files
- Cryptography
- Search
- File matching
- Syncing

# The Rsync Algorithm

- Used to efficiently sync files over a network.
- Details are a bit involved, but the basic idea is this.
  - Break file B into blocks and apply a hash function to each block.
  - Send these signatures across the network.
  - Compute similar signatures for file A and only send the blocks for which the hash is different.
- The real algorithms uses two different hash functions and accounts for the possibility of not detecting differences when only using one hash function.

# The `hashlib` Module

The `hashlib` module contains implementations of many of the most commonly used hash functions.

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.digest_size
16
>>> m.block_size
64
```