CSE 100: HASHING, BOGGLE
Probability of Collisions

- If you have a hash table with M slots and N keys to insert in it, then the probability of at least 1 collision is:

\[
P_{N,M}(\text{collision}) = 1 - P_{N,M}(\text{no collision})
\]

\[
= 1 - \prod_{i=1}^{N} P_{N,M}(\text{ith key no collision})
\]
Hashtable collisions and the "birthday paradox"

• Suppose there are 365 slots in the hash table: $M=365$

• What is the probability that there will be a collision when inserting $N$ keys?
  • For $N = 10$, $\text{prob}_{N,M}(\text{collision}) = 12\%$
  • For $N = 20$, $\text{prob}_{N,M}(\text{collision}) = 41\%$
  • For $N = 30$, $\text{prob}_{N,M}(\text{collision}) = 71\%$
  • For $N = 40$, $\text{prob}_{N,M}(\text{collision}) = 89\%$
  • For $N = 50$, $\text{prob}_{N,M}(\text{collision}) = 97\%$
  • For $N = 60$, $\text{prob}_{N,M}(\text{collision}) = 99+\%$

• So, among 60 randomly selected people, it is almost certain that at least one pair of them have the same birthday

• In general: collisions are likely to happen, unless the hash table is quite sparsely filled

• So, if you want to use hashing, can’t use perfect hashing because you don’t know the keys in advance, and don’t want to waste huge amounts of storage space, you have to have a strategy for dealing with collisions
Making hashing work

• Important issues in implementing hashing are:
  • Deciding on the hash function
  • Deciding on the size of the hash table
  • Deciding on the collision resolution strategy

• With a good hashtable design, $O(1)$ average-case insert and find operation time costs can be achieved, with $O(1)$ space cost per key stored

• This makes hashtables a very useful and very commonly used data structure
Hash functions: desiderata

Important considerations in designing a hash function for use with a hash table
• It is fast to compute (must be $O(1)$)
• It distributes keys evenly
• It is consistent with the equality testing function (i.e. two keys that are equal will have the same hash value)

Designing a good hash function is not easy!
We’ll look at a few, but there’s much more to explore.
Simple (and effective?) hash function for integers: \( H(K) = K \mod M \)

- When is the function \( H(K) = K \mod M \) where \( K \) is the key and \( M \) is the table size a good hash function for integer keys?
  A. Always
  B. When \( M \) is prime
  C. When values of \( K \) are evenly distributed
  D. In the case of either B or C
  E. Never
Simple (and effective?) hash function for integers: \( H(K) = K \mod M \)

- When is the function \( H(K) = K \mod M \) where \( K \) is the key and \( M \) is the table size a good hash function for integer keys?
  
  When \( M \) is prime OR When values of \( K \) are evenly distributed

Think of an example of a non-prime \( M \) and a distribution of keys that would cause poor behavior from this function.
Simple (and effective?) hash function for integers: \( H(K) = K \mod M \)

- When is the function \( H(K) = K \mod M \) where \( K \) is the key and \( M \) is the table size a good hash function for integer keys?

  When \( M \) is prime OR When values of \( K \) are evenly distributed

Because you can never count on evenly distributed keys, always use prime-size table with this hash function
Hash table size

- By "size" of the hash table we mean how many slots or buckets it has.
- Choice of hash table size depends in part on choice of hash function, and collision resolution strategy.
- But a good general “rule of thumb” is:
  - The hash table should be an array with length about 1.3 times the maximum number of keys that will actually be in the table, and
  - Size of hash table array should be a prime number (especially with the simple hash function we looked at).
- So, let \( M \) = the next prime larger than 1.3 times the number of keys you will want to store in the table, and create the table as an array of length \( M \).
- (If you underestimate the number of keys, you may have to create a larger table and rehash the entries when it gets too full; if you overestimate the number of keys, you will be wasting some space.)
Hash functions for strings

• It is common to want to use string-valued keys in hash tables

• What is a good hash function for strings?

• The basic approach is to use the characters in the string to compute an integer, and then take the integer mod the size of the table

• How to compute an integer from a string?
  • You could just take the last two 16-bit chars (or last four 8-bit characters) of the string and form a 32-bit int
  • But then all strings ending in the same 2 (or 4) chars would hash to the same location; this could be very bad
  • It would be better to have the hash function depend on all the chars in the string

• There is no recognized single "best" hash function for strings. Let’s look at some possible ones
String hash function #1

- This hash function adds up the integer values of the chars in the string (\textit{then need to take the result mod the size of the table}): 

\begin{verbatim}
int hash(std::string const & key) {
    int hashVal = 0,
    len=key.length();
    for(int i=0; i<len; i++) {
        hashVal += key[i];
    }
    return hashVal;
}
\end{verbatim}

What is wrong with this hash function for storing words of 8-characters? 
A. Nothing 
B. It is too complex (takes too long) to compute 
C. It will lead to collisions between words with similar endings 
D. It will not distribute keys well in a large table 
E. It will never distribute keys well in any table
String hash function #1

- This hash function adds up the integer values of the chars in the string (then need to take the result mod the size of the table):

```cpp
int hash(std::string const & key) {
    int hashVal = 0, len = key.length();
    for(int i=0; i<len; i++) {
        hashVal += key[i];
    }
    return hashVal;
}
```

- This function is simple to compute, but it often doesn’t work very well in practice:

- Suppose the keys are strings of 8 ASCII capital letters and spaces

- There are $27^8$ possible keys; however, ASCII codes for these characters are in the range 65-95, and so the sums of 8 char values will be in the range 520 - 760

- In a large table ($M>1000$), only a small fraction of the slots would ever be mapped to by this hash function! For a small table ($M<100$), it may be okay
String hash function #2: Java code

- `java.lang.String`’s `hashCode()` method uses a polynomial with \( x=31 \) (though it goes through the String’s chars in reverse order), and uses Horner’s rule to compute it:

```java
class String implements java.io.Serializable, Comparable {
   /** The value is used for character storage. */
   private char value[];
   /** The offset is the first index of the storage that is used. */
   private int offset;
   /** The count is the number of characters in the String. */
   private int count;
   public int hashCode() {
      int h = 0;
      int off = offset;
      char val[] = value;
      int len = count;

      for (int i = 0; i < len; i++)
         h = 31*h + val[off++];

      return h;
   }

   H(s) = \sum_{i=0}^{n} s.charAt(i) * x^i
```

\[ H(s) = \sum_{i=0}^{n} s.charAt(i) * x^i \]
Collisions

• Since the hash function is O(1), a hash table has the potential for very fast find performance (the best possible!), but...

• ... since the hash function is mapping from a large set (the set of all possible keys) to a smaller set (the set of hash table locations) there is the possibility of collisions: two different keys wanting to be at the same table location
Collision resolution strategies

• Unless we are doing "perfect hashing" we have to have a collision resolution strategy, to deal with collisions in the table.

• The strategy has to permit find, insert, and delete operations that work correctly!

• Collision resolution strategies we will look at are:
  • Linear probing (from your reading)
  • Random hashing
  • Separate chaining (from your reading)
Linear probing: inserting a key

- When inserting a key $K$ in a table of size $M$, with hash function $H(K)$
  1. Set $\text{indx} = H(K)$
  2. If table location $\text{indx}$ already contains the key, no need to insert it. Done!
  3. Else if table location $\text{indx}$ is empty, insert key there. Done!
  4. Else collision. Set $\text{indx} = (\text{indx} + 1) \mod M$.
  5. If $\text{indx} == H(K)$, table is full! (Throw an exception, or enlarge table.) Else go to 2.

$M = 7, \ H(K) = K \mod M$
insert these keys 701 (1), 145 (5), 217 (0), 19 (5), 13 (6), 749 (0)
in this table, using linear probing:

<table>
<thead>
<tr>
<th>index:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

Linear probing: searching for a key

- If keys are inserted in the table using linear probing, linear probing will find them!

- When searching for a key $K$ in a table of size $N$, with hash function $H(K)$:
  
  1. Set $indx = H(K)$
  2. If table location $indx$ contains the key, return FOUND.
  3. Else if table location $indx$ is empty, return NOT FOUND.
  4. Else set $indx = (indx + 1) \mod M$.
  5. If $indx == H(K)$, return NOT FOUND. Else go to 2.

- Question: How to delete a key from a table that is using linear probing?
  - Could you do "lazy deletion", and just mark the deleted key’s slot as empty? Why or why not?
Random hashing

- Random hashing avoids clustering by making the probe sequence depend on the key

- With random hashing, the probe sequence is generated by the output of a pseudorandom number generator seeded by the key (possibly together with another seed component that is the same for every key, but is different for different tables)

- The insert algorithm for random hashing is then:

1. Create RNG seeded with K. Set indx = RNG.next() mod M.
2. If table location indx already contains the key, no need to insert it. Done!
3. Else if table location indx is empty, insert key there. Done!
4. Else collision. Set indx = RNG.next() mod M.
5. If all M locations have been probed, give up. Else, go to 2.

- Random hashing is easy to analyze, but because of the "expense" of random number generation, it is not often used. There is another method called double hashing that works just as well.
Resolving Collisions: Double hashing

- A sequence of possible positions to insert an element are produced using two hash functions

- $h_1(x)$: to determine the position to insert in the array, $h_2(x)$: the offset from that position

701 (1, 2), 145 (5,4), 217 (0,3), 19 (5, 3), 13 (6,2), 749 (0,2)

in this table:

<table>
<thead>
<tr>
<th>217</th>
<th>701</th>
<th></th>
<th></th>
<th></th>
<th>145</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

index:
Open addressing vs. separate chaining

- Linear probing and random hashing are appropriate if the keys are kept as entries in the hashtable itself...
  - doing that is called "open addressing"
  - it is also called "closed hashing"

- Another idea: Entries in the hashtable are just pointers to the head of a linked list ("chain"); elements of the linked list contain the keys...
  - this is called "separate chaining"
  - it is also called "open hashing"

- Collision resolution becomes easy with separate chaining: just insert a key in its linked list if it is not already there
  - (It is possible to use fancier data structures than linked lists for this; but linked lists work very well in the average case)
Resolving Collisions: Separate Chaining

• using the hash function \( H(K) = K \mod M \), insert these integer keys:

\[
701 \ (1), \ 145 \ (5), \ 217 \ (0), \ 19 \ (5), \ 13 \ (6), \ 749 \ (0)
\]

in this table:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

Is there an upper bound to the number of elements that we can insert into a hashtable with separate chaining?

A. Yes, because of space constraints in the array
B. No, but inserting too many elements can affect the run time of insert
C. No, but inserting too many elements can affect the run time of find
D. Both B and C
Analysis of open-addressing hashing

• What is the *worst-case* time to find a single element in a hash table with N elements in it?
  A. O(1)
  B. O(log(N))
  C. O(N)
  D. O(Nlog(N))
  E. O(N^2)
Analysis of open-addressing hashing

- A useful parameter when analyzing hash table Find or Insert performance is the load factor

  \[ \alpha = \frac{N}{M} \]

  where \( M \) = size of the table
  \( N \) = number of keys that have been inserted in the table

- The load factor is a measure of how full the table is

- Given a load factor \( \alpha \), we would like to know the time costs, in the best, average, and worst case of
  - new-key insert and unsuccessful find (these are the same)
  - successful find

  We will not derive these, but rather look at a graph that plots the relationship between \( \alpha \) and expected number of probes.
Dependence of average performance on load

Take away 1: performance depends on load factor, not directly on number of elements or size of table.
Take away 2: things start to go badly when the load factor exceeds 70%
Average case costs with separate chaining

What you need to know:

• Separate chaining performance also depends only on load factor, and not the number of elements or size of table
• In practice it performs extremely well, even with relatively high loads
The game of Boggle:
- Two players, one board (MxN) dies
- Each die face contains a string
- Player that constructs the maximum number of unique words is the winner

Rules of the game:
- Words are constructed by traversing a sequence of adjacent dies (an acyclic path starting at any dice)
- Two dice are adjacent if they are next to each other horizontally, vertically, or diagonally.
- Die can only be used once in a word (acyclic path)

Checkpoint:
Implement
- setBoard(): Given a 2D array of strings populate the board
- isOnBoard(): Given a word, check if it is on the board
This is the boggle board… how would you represent it in your code?

A. As a linked list
B. As a 2D array
C. As a tree
D. Graph
E. A vector
Graphs in PA4 (HINT!!)

This is also a graph... where are the edges? (fill them in)
Graphs in PA4 (HINT!!)

This is also a graph… where are the edges? (fill them in)
Graphs in PA4 (HINT!!)

/**
 * Vector representing the boggle board.
 */
std::vector<BoardPos> board;
//Elements are in row major order

If row is the number of rows and col is the number of columns, how would you access the element in the ith row and jth column of the 2D array using the vector representation
A. board[i+j]
B. board[i*col+j]
C. board[j*col+i]
D. board[i*row+j]
E. board[j*row+i]
Why do we have the visited field?
Which of the following algorithms best applies to our problem i.e searching for a word on the board?

A. BFS
B. DFS
C. Dijkstra’s
D. Prims

std::vector<int> BogglePlayer::isOnBoard(const std::string &word_to_check)
What do we get from running (vanilla) DFS on this graph, starting with source node C₄?

A. All possible words (valid and invalid) starting with C
B. A sequence containing all the letters on the board starting with C
C. A sequence containing some of the letters on the board starting with C
D. B or C
What is the simplest change you can make to DFS to get us one step closer to our solution?
isOnBoard(G,v, word ) ( v is the vertex where the search starts, word is the word we are looking for )

Stack S := {}; ( start with an empty stack ), pos:=0
for each vertex u, set visited[u] := false;
push S, v;
while (S is not empty) do
  u := pop S;
  if (NOT u.visited AND u.letter ==word[pos])
    pos:=pos+1
    u.visited := true;
  for each unvisited neighbour w of u
    push S, w;
  end if
end while
END DFS()

With the modified algorithm, can I find the word CAP (if the source vertex is C 5)
A. Yes
B. No
C. Not exactly, but I have made progress
isOnBoard(G,v, word ) ( v is the vertex where the search starts, word is the word we are looking for )

Stack S := {}; ( start with an empty stack ), pos:=0
for each vertex u, set visited[u] := false;
push S, v;
while (S is not empty) do
    u := pop S;
    if (NOT u.visited AND u.letter == word[pos])
        u.visited := true;
        pos:=pos+1
    end if
    for each unvisited neighbour w of u
        push S, w;
    end for
end while
END DFS()

Am I done? What happens if I search for “CANIBAL”?
A. We can always find it without any problem
B. We may or may not find it
isOnBoard(G,v, word) ( v is the vertex where the search starts, 
word is the word we are looking for )

Stack S := {}; (start with an empty stack), pos:=0
for each vertex u, set visited[u] := false;
push S, v;
while (S is not empty) do
    u := pop S;
    if (NOT u.visited AND u.letter ==word[pos])
        u.visited := true;
        pos:=pos+1
        if pos==length(word)
            return
        for each unvisited neighbour w of u
            push S, w;
    end if
end while
END DFS()
isOnBoard(G,v, word ) ( v is the vertex where the search starts, word is the word we are looking for )
Stack S := {}; ( start with an empty stack ), pos:=0
for each vertex u, set visited[u] := false;
push S, v;
while (S is not empty) do
  u := pop S;
  if u.revisit == true
    u.revisit == false
    u.visited := false;
pos:=pos-1
  else if (NOT u.visited AND u.letter == word[pos])
    u.visited := true;
pos:=pos+1
    if pos==length(word)
      return
    u.revisit := true;
push S, u
  for each unvisited neighbour w of u
    push S, w;
end if
end while
END DFS()