Chapter 5
Hashing

Introduction

- hashing performs basic operations, such as insertion, deletion, and finds in __________ average time
- better than other ADTs we’ve seen so far

Hashing

- a hash table is merely an __________ of some fixed size
- hashing converts ______________ into locations in a hash table
- searching on the key becomes something like array lookup
- hashing is typically a many-to-one map: multiple keys are mapped to the same array index
- mapping multiple keys to the same position results in a __________ that must be resolved
- two parts to hashing:
  - a hash function, which transforms keys into array indices
  - a collision resolution procedure

Hashing Functions

- let $K$ be the set of search keys
- hash functions map $K$ into the set of $M$ __________ in the hash table
  \[ h: K \rightarrow \{0, 1, \ldots, M - 1\} \]
- ideally, $h$ distributes $K$ __________ over the slots of the hash table, to minimize collisions
- if we are hashing $N$ items, we want the number of items hashed to each location to be close to $N/M$
- example: Library of Congress Classification System
  - hash function if we look at the first part of the call numbers (e.g., E470, PN1995)
  - collision resolution involves going to the stacks and looking through the books
  - almost all of CS is hashed to QA75 and QA76 (BAD)
- suppose we are storing a set of nonnegative integers
- given \( M \), we can obtain hash values between 0 and 1 with the hash function
  \[ h(k) = k \mod M \]
- fast operation, but we need to be careful when choosing \( M \)
- example: if \( M = 2^p \), \( h(k) \) is just the \( p \) lowest-order bits of \( k \)
- are all the hash values equally likely?
- choosing \( M \) to be a \( \_\_\_\_\_\_\_\_\_ \) not too close to a power of 2 works well in practice

- we can also use the hash function below for floating point numbers if we interpret the bits as an \( \_\_\_\_\_\_\_\_\_\_\_ \)
  \[ h(k) = k \mod M \]
- two ways to do this in C, assuming \texttt{long int} and \texttt{double} types have the same length
- first method uses C \( \_\_\_\_\_\_\_\_\_\_\_ \) to accomplish this task

\[
\begin{align*}
\text{unsigned long } & k; \quad \text{double } x; \\
& k = (\text{unsigned long }*) &x; \\
\text{long int } & \text{hash} = k \mod M;
\end{align*}
\]

- we can hash strings by combining a hash of each \( \_\_\_\_\_\_\_\_\_\_\_ \)
- second uses a \( \_\_\_\_\_\_\_\_\_\_\_ \), which is a variable that can hold objects of different types and sizes
  \[
  \begin{array}{l}
  \text{union } \\
  \begin{array}{l}
  \text{long int } k; \\
  \text{double } x;
  \end{array}
  \end{array}
  \]
  \[
  u;
  u.x = 3.1416;
  \begin{array}{l}
  \text{long int } & \text{hash} = u.k \mod M;
  \end{array}
  \]

- \( R \) is an additional parameter we get to choose
- if \( R \) is larger than any character value, then this approach is what you would obtain if you treated the string as a base-\( R \)
- K&R suggest a slightly simpler hash function, corresponding to $R = 31$

```c
char *s;
unsigned hash;
for (hash = 0; *s != '\0'; s++) {
    hash = 31 * hash + *s;
}
hash = hash % M;
```

- Weiss suggests $R = 37$

- we can use the idea for strings if our search key has __________ parts, say, street, city, state:

```c
hash = ((street * R + city) % M) * R + state) % M;
```

- same ideas apply to hashing vectors

---

- the choice of parameters can have a __________ effect on the results of hashing
- compare the text's string hashing algorithm for different pairs of $R$ and $M$
- plot ________________ of the number of words hashed to each hash table location; we use the American dictionary from the aspell program as data (305,089 words)

- example: $R = 31$, $M = 1024$
- good: words are ________________
Hash Functions

- example: $R = 32, M = 1024$
- very bad

Collision Resolution

- hash table collision
  - occurs when elements hash to the ________________ in the table
  - various ________________ for dealing with collision
    - separate chaining
    - open addressing
    - linear probing
    - other methods
- separate chaining
  - keep a list of all elements that _______ to the same location
  - each location in the hash table is a __________________________
  - example: first 10 squares

- observations
  - insert, search, delete in lists
    - all proportional to __________ of linked list
    - insert
      - new elements can be inserted at ________ of list
      - duplicates can increment __________
    - other structures could be used instead of lists
      - binary search tree
      - another hash table
    - linked lists good if table is __________ and hash function is good

- how long are the linked lists in a hash table?
  - ________ value: \( N/M \) where \( N \) is the number of keys and \( M \) is the size of the table
  - is it reasonable to assume the hash table would exhibit this behavior?
    - load factor \( \lambda = N/M \)
    - average length of a list = \( \lambda \)
    - time to search: ________ time to evaluate the hash function + time to __________ the list
      - unsuccessful search: 1 + \( \lambda \)
      - successful search: 1 + (\( \lambda/2 \))
Separate Chaining

-declaration of hash structure

```cpp
template <typename HashedObj>
class HashTable {
public:
  explicit HashTable(int size = 101);
  bool contains(const HashedObj &x) const;
  void makeEmpty();
  bool insert(const HashedObj &x);
  bool insert(HashedObj &x);
  bool remove(const HashedObj &x);
private:
  vector<list<HashedObj>> theLists; // the array of lists
  int currentSize;
  void rehash();
  size_t myhash(const HashedObj &x) const;
};
```

-hash member function

```cpp
size_t myhash(const HashedObj &x) const
{
    static hash<HashedObj> h;
    return h(x) % theLists.size();
}
```

-routines for separate chaining

```cpp
bool insert(const HashedObj &x) {
    auto &whichList = theLists[myhash(x)];
    if (find(begin(whichList), end(whichList), x) == end(whichList))
        return false;
    whiteList.push_back(x);
    // Rehash; see Section 5.6
    if (++currentSize > theLists.size())
        rehash();
    return true;
}
```

-routines for separate chaining
Open Addressing

- linked lists incur extra costs
  - time to ________________ for new cells
  - effort and complexity of defining second data structure
- a different collision strategy involves placing colliding keys in nearby _________ slots
  - if a collision occurs, try _________ cells until an empty one is found
  - bigger table size needed with \( M > N \)
  - load factor should be below \( \lambda = 0.5 \)
- three common strategies
  - linear probing
  - quadratic probing
  - double hashing

Linear Probing

- linear probing insert operation
  - when \( k \) is hashed, if slot \( h(k) \) is open, place \( k \) there
  - if there is a collision, then start looking for an empty slot starting with location \( h(k) + 1 \) in the hash table, and proceed _________ through \( h(k) + 2, \ldots, m - 1, 0, 1, 2, \ldots, h(k) - 1 \) wrapping around the hash table, looking for an empty slot
  - search operation is similar
  - checking whether a table entry is vacant (or is one we seek) is called a __________

Linear Probing

- example: add 89, 18, 49, 58, 69 with \( h(k) = k \% 10 \) and \( f(i) = i \)

<table>
<thead>
<tr>
<th>Empty Table</th>
<th>After 89</th>
<th>After 18</th>
<th>After 49</th>
<th>After 58</th>
<th>After 69</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>18</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>49</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>38</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>69</td>
<td>69</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>89</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- as long as the table is _________, a vacant cell can be found
  - but time to locate an empty cell can become large
  - blocks of occupied cells results in primary __________
- deleting entries leaves __________
  - some entries may no longer be found
  - may require moving many other entries

- expected number of probes
  - for search hits: \( \sim \frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)} \right) \)
  - for insertion and search misses: \( \sim \frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)^2} \right) \)
  - for \( \lambda = 0.5 \), these values are 3/2 and 5/2, respectively
Linear Probing

- performance of linear probing (dashed) vs. more random collision resolution
- adequate up to $\lambda = 0.5$
- Successful, Unsuccessful, Insertion

![Linear Probing Graph]

Quadratic Probing

- quadratic probing
  - eliminates ________________
  - collision function is quadratic
  - example: add 89, 18, 49, 58, 69 with $h(k) = k \mod 10$ and $f(i) = i^2$

<table>
<thead>
<tr>
<th>Empty Table</th>
<th>After 89</th>
<th>After 18</th>
<th>After 49</th>
<th>After 58</th>
<th>After 69</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>49</td>
<td>49</td>
<td>49</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>1</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
</tr>
</tbody>
</table>

![Quadratic Probing Table]

Quadratic Probing

- in linear probing, letting table get nearly _______ greatly hurts performance
- quadratic probing
  - no ____________ of finding an empty cell once the table gets larger than half full
  - at most, ________ of the table can be used to resolve collisions
  - if table is half empty and the table size is prime, then we are always guaranteed to accommodate a new element
  - could end up with situation where all keys map to the same table location

Quadratic Probing

- quadratic probing
  - collisions will probe the same alternative cells
  - ________________ clustering
  - causes less than half an extra probe per search
Double Hashing

- double hashing
  - \( f(i) = i \cdot \text{hash}_2(x) \)
  - apply a second hash function to \( x \) and probe across
  - function must never evaluate to _____
  - make sure all cells can be probed

Rehashing

- table may get _______________
  - run time of operations may take too long
  - insertions may ______ for quadratic resolution
  - too many removals may be intermixed with insertions
- solution: build a new table _______________ (with a new hash function)
  - go through original hash table to compute a hash value for each (non-deleted) element
  - insert it into the new table

Double Hashing example

- \( \text{hash}_2(x) = R - (x \mod R) \) with \( R = 7 \)
- \( R \) is a prime smaller than table size
- insert 89, 18, 49, 58, 69

<table>
<thead>
<tr>
<th>Empty Table</th>
<th>After 89</th>
<th>After 49</th>
<th>After 58</th>
<th>After 69</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>69</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>58</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>49</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Rehashing

- example: insert 13, 15, 24, 6 into a hash table of size 7
  - with \( h(k) = k \mod 7 \)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>15</td>
<td>24</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- example (cont.)
  - insert 23
  - table will be over 70% full; therefore, a new table is created

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>15</td>
<td>23</td>
<td>24</td>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rehashing

- example (cont.)
  - new table is size 17
  - new hash function \( h(k) = k \mod 17 \)
  - all old elements are inserted into new table

Rehashing

- rehashing run time \( O(N) \) since \( N \) elements and to rehash the entire table of size roughly \( 2N \)
  - must have been \( N/2 \) insertions since last rehash
- rehashing may run OK if in ______________
  - if interactive session, rehashing operation could produce a slowdown
- rehashing can be implemented with ______________
  - could rehash as soon as the table is half full
  - could rehash only when an insertion fails
  - could rehash only when a certain __________ is reached
  - may be best, as performance degrades as load factor increases
Hash Tables with Worst-Case $O(1)$ Access

- hash tables so far
  - $O(1)$ average case for insertions, searches, and deletions
  - separate chaining: worst case $\Theta(\log N / \log \log N)$
    - some queries will take nearly logarithmic time
  - worst-case $O(1)$ time would be better
    - important for applications such as lookup tables for routers and memory caches
    - if $N$ is known in advance, and elements can be \dots, worst-case $O(1)$ time is achievable

perfect hashing
- assume all $N$ items known \dots
- separate chaining
  - if the number of lists continually increases, the lists will become shorter and shorter
  - with enough lists, high probability of \dots
  - two problems
    - number of lists might be unreasonably \dots
    - the hashing might still be unfortunate
      - $M$ can be made large enough to have probability $\frac{1}{2}$ of no collisions
      - if collision detected, clear table and try again with a different hash function (at most done 2 times)

perfect hashing (cont.)
- how large must $M$ be?
  - theoretically, $M$ should be $N^2$, which is \dots
- solution: use $N$ lists
  - resolve collisions by using hash tables instead of linked lists
    - each of these lists can have $n^2$ elements
  - each secondary hash table will use a different hash function until it is \dots
    - can also perform similar operation for primary hash table
  - total size of secondary hash tables is at most $2N$
Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing
  - $\Theta(\log N/\log \log N)$ bound known for a long time
  - researchers surprised in 1990s to learn that if one of two
tables were ________ chosen as items were inserted,
the size of the largest list would be $\Theta(\log \log N)$, which is
significantly smaller
  - main idea: use 2 tables
    - neither more than _______ full
    - use a separate hash function for each
    - item will be stored in one of these two locations
    - collisions resolved by _______________ elements

Hash Tables with Worst-Case $O(1)$ Access (cont.)

- example: 6 items; 2 tables of size 5; each table has
randomly chosen hash function

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Table 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>G</td>
</tr>
</tbody>
</table>

- A can be placed at position 0 in Table 1 or position 2 in
Table 2
- a search therefore requires at most 2 table accesses in
this example
- item deletion is trivial

Hash Tables with Worst-Case $O(1)$ Access (cont.)

- insertion
  - ensure item is not already in one of the tables
  - use first hash function and if first table location is
    ___________, insert there
  - if location in first table is occupied
    - ________ element there and place current item
      in correct position in first table
  - displaced element goes to its alternate hash position
    in the second table

- insert B (displace A)
Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - insert $C$

\[
\begin{array}{c|c}
\text{Table 1} & \text{Table 2} \\
\hline
0 & 0 \\
1 & 1 \\
2 & 2 \\
3 & 3 \\
4 & 4 \\
\end{array}
\begin{array}{c|c}
A & 0, 2 \\
B & 0, 0 \\
C & 1, 4 \\
D & 1, 0 \\
E & 3, 2 \\
F & 3, 0 \\
G & 1, 2 \\
\hline
\end{array}
\]

- insert $D$ (displace $C$) and $E$

\[
\begin{array}{c|c}
\text{Table 1} & \text{Table 2} \\
\hline
0 & 0 \\
1 & 1 \\
2 & 2 \\
3 & 3 \\
4 & 4 \\
\end{array}
\begin{array}{c|c}
A & 0, 2 \\
B & 0, 0 \\
C & 1, 4 \\
D & 1, 0 \\
E & 3, 2 \\
F & 3, 4 \\
G & 1, 2 \\
\hline
\end{array}
\]

- (A displaces $B$)

\[
\begin{array}{c|c}
\text{Table 1} & \text{Table 2} \\
\hline
0 & 0 \\
1 & 1 \\
2 & 2 \\
3 & 3 \\
4 & 4 \\
\end{array}
\begin{array}{c|c}
A & 0, 2 \\
B & 0, 3 \\
C & 1, 4 \\
D & 1, 0 \\
E & 3, 2 \\
F & 3, 4 \\
G & 1, 2 \\
\hline
\end{array}
\]

- (B relocated)

\[
\begin{array}{c|c}
\text{Table 1} & \text{Table 2} \\
\hline
0 & 0 \\
1 & 1 \\
2 & 2 \\
3 & 3 \\
4 & 4 \\
\end{array}
\begin{array}{c|c}
A & 0, 2 \\
B & 0, 0 \\
C & 1, 4 \\
D & 1, 0 \\
E & 3, 2 \\
F & 3, 4 \\
G & 1, 2 \\
\hline
\end{array}
\]

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - insert $G$

\[
\begin{array}{c|c}
\text{Table 1} & \text{Table 2} \\
\hline
0 & 0 \\
1 & 1 \\
2 & 2 \\
3 & 3 \\
4 & 4 \\
\end{array}
\begin{array}{c|c}
A & 0, 2 \\
B & 0, 0 \\
C & 1, 4 \\
D & 1, 0 \\
E & 3, 2 \\
F & 3, 4 \\
G & 1, 2 \\
\hline
\end{array}
\]

- displacements are ______________
  - $G$ $D$ $B$ $A$ $E$ $F$ $C$ $G$
  - can try $G$’s second hash value in second table, but it also results in a displacement cycle

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - cycles
    - if table’s load value < 0.5, probability of a cycle is very _______
    - insertions should require < $O(\log N)$ displacements
    - if a certain number of displacements is reached on an insertion, tables can be ____________ with new hash functions
Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - variations
    - higher number of tables (3 or 4)
    - place item in second hash slot immediately instead of __________ other items
  - allow each cell to store __________ keys
    - space utilization increased

<table>
<thead>
<tr>
<th></th>
<th>1 item per cell</th>
<th>2 items per cell</th>
<th>4 items per cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 hash functions</td>
<td>0.49</td>
<td>0.86</td>
<td>0.93</td>
</tr>
<tr>
<td>3 hash functions</td>
<td>0.91</td>
<td>0.97</td>
<td>0.98</td>
</tr>
<tr>
<td>4 hash functions</td>
<td>0.97</td>
<td>0.99</td>
<td>0.999</td>
</tr>
</tbody>
</table>

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - benefits
    - worst-case __________ lookup and deletion times
    - avoidance of __________
    - potential for __________
  - potential issues
    - extremely sensitive to choice of hash functions
    - time for insertion increases rapidly as load factor approaches 0.5

Hash Tables with Worst-Case $O(1)$ Access

- hopscotch hashing
  - improves on linear probing algorithm
    - linear probing tries cells in sequential order, starting from hash location, which can be long due to primary and secondary clustering
    - instead, hopscotch hashing places a bound on __________ of the probe sequence
      - results in worst-case constant-time lookup
    - can be parallelized
  - hopscotch hashing (cont.)
    - if insertion would place an element too far from its hash location, go backward and __________ other elements
      - evicted elements cannot be placed farther than the maximal length
    - each position in the table contains information about the current element inhabiting it, plus others that __________ to it
- hopscotch hashing (cont.)
  - example: MAX_DIST = 4

<table>
<thead>
<tr>
<th>Item</th>
<th>Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>C 1000</td>
</tr>
<tr>
<td>7</td>
<td>A 1100</td>
</tr>
<tr>
<td>8</td>
<td>D 0010</td>
</tr>
<tr>
<td>9</td>
<td>B 1000</td>
</tr>
<tr>
<td>10</td>
<td>E 0000</td>
</tr>
<tr>
<td>11</td>
<td>G 1000</td>
</tr>
<tr>
<td>12</td>
<td>F 1000</td>
</tr>
<tr>
<td>13</td>
<td>0000</td>
</tr>
<tr>
<td>14</td>
<td>0000</td>
</tr>
</tbody>
</table>

- each bit string provides 1 bit of information about the current position and the next 3 that follow
  - 1: item hashes to current location; 0: no

- hopscotch hashing (cont.)
  - example: insert H in 9

  - try in position 13, but too far, so try candidates for eviction (10, 11, 12)
  - evict G in 11

- hopscotch hashing (cont.)
  - example: insert I in 6

  - position 14 too far, so try positions 11, 12, 13
  - G can move down one
  - position 13 still too far; F can move down one
universal hashing
- in principle, we can end up with a situation where all of our keys are hashed to the bad in the hash table (bad)
- more realistically, we could choose a hash function that does not distribute the keys
- to avoid this, we can choose the hash function so that it is independent of the keys being stored
- yields provably good performance on average

example: choose a prime \( p \) sufficiently large that every key \( k \) is in the range 0 to \( p - 1 \) (inclusive)
- let \( A = \{0, 1, \ldots, p - 1\} \) and \( B = \{1, \ldots, p - 1\} \)
then the family
\[
h_{a,b}(k) = (ak + b \mod p) \mod M \quad a \in A, b \in B
\]
is a universal class of hash functions

universal hashing (cont.)
- let \( H \) be a finite collection of functions mapping our set of keys \( K \) to the range \( \{0, 1, \ldots, M - 1\} \)
- \( H \) is a collection if for each pair of distinct keys \( k, l \in K \), the number of hash functions \( h \in H \) for which \( h(k) = h(l) \) is at most \( |H|/M \)
- that is, with a randomly selected hash function \( h \in H \), the chance of a between distinct \( k \) and \( l \) is not more than the probability \( (1/M) \) of a collision if \( h(k) \) and \( h(l) \) were chosen randomly and independently from \( \{0,1,\ldots, M - 1\} \)

extendible hashing
- amount of data too large to fit in main consideration is then the number of disk accesses
- assume we need to store \( N \) records and \( M = 4 \) records fit in one disk block
- current problems
- if probing or separate chaining is used, collisions could cause to be examined during a search
- rehashing would be expensive in this case
Hash Tables with Worst-Case $O(1)$ Access

- extendible hashing (cont.)
  - allows search to be performed in _____ disk accesses
  - insertions require a bit more
  - use B-tree
  - as $M$ increases, height of B-tree ________________
    - could make height = 1, but multi-way branching would be extremely high

Hash Tables with Worst-Case $O(1)$ Access

- extendible hashing (cont.)
  - example: 6-bit integers

- root contains 4 pointers determined by first 2 bits
- each leaf has up to 4 elements

Hash Tables with Worst-Case $O(1)$ Access

- extendible hashing (cont.)
  - example: insert 100100
    - place in third leaf, but full
    - split leaf into 2 leaves, determined by 3 bits

Hash Tables with Worst-Case $O(1)$ Access

- extendible hashing (cont.)
  - example: insert 000000
    - first leaf split
Hash Tables with Worst-Case $O(1)$ Access

- extendible hashing (cont.)
  - considerations
    - several directory _________ may be required if the elements in a leaf agree in more than D+1 leading bits
    - number of bits to distinguish bit strings
    - does not work well with ___________ ( > M duplicates: does not work at all)

Hash Tables with Worst-Case $O(1)$ Access

- final points (cont.)
  - hash tables good for
    - symbol table
    - gaming
      - remembering locations to avoid recomputing through transposition table
    - spell checkers

- final points
  - choose hash function carefully
  - watch _______________
    - separate chaining: close to 1
    - probing hashing: 0.5
  - hash tables have some _______________
    - not possible to find min/max
    - not possible to ___________ for a string unless the exact string is known
      - binary search trees can do this, and $O(\log N)$ is only slightly worse than $O(1)$