Hashing - 2

Outline

- Collision Resolution Techniques
  - Separate Chaining – (we have seen this)
  - Open Addressing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
  - Rehashing
- Extendible Hashing
Open Addressing

- Separate chaining method was using linked lists.
  - Requires implementation of a second data structures
  - For some languages, creating new nodes (for linked lists) is expensive and slows down the system.

- In open addressing:
  - All items are stored in the hash table itself.
  - If a collision occurs, alternative cells are tried until an empty cell is found.

The cells that are tried successively can be expressed formally as:

- $h_0(x), h_1(x), h_2(x), \ldots$
  - $h_0(x)$ is the initial cells that causes a collision.
  - $h_1(x), h_2(x), \ldots$ are alternative cells.

- $h_i(x) = (\text{hash}(x) + f(i)) \mod \text{TableSize}$
  - $f(i)$ is collision resolution strategy (function).
  - $f(0) = 0$. 
Open Addressing

- There are various methods as open addressing schemes:
  - Linear Probing
    - $\text{hash}(x) = \text{hash}(x) + f(i) = i$, where $i \geq 0$
  - Quadratic Probing
    - $\text{hash}(x) = \text{hash}(x) + f(i) = i^2$, where $i \geq 0$
  - Double Hashing
    - $\text{hash}(x) = \text{hash}_1(x) + i \cdot \text{hash}_2(x)$, where $i \geq 0$

---

Linear Probing

- In linear probing, $f$ is a linear function of $i$.
- Typically $f(i) = i$.
- When a collision occurs, cells are tried sequentially in search of an empty cell.
  - Wrap around when end of array is reached.
- Example:
  - Insert items: 89, 18, 49, 58, 69 into an empty hash table.
  - Table size is 10.
  - Hash function is $\text{hash}(x) = x \mod 10$.
  - Collision resolution strategy is $f(i) = i$;
Example

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

Primary cluster cells: 8, 9, 0, 1, 2

<table>
<thead>
<tr>
<th>Keys</th>
<th>h₀(x)</th>
<th>h₁(x)</th>
<th>h₂(x)</th>
<th>h₃(x)</th>
<th>...</th>
<th>Number of Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>89</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>58</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>69</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Primary Clustering

- Blocks of occupied cells (a cluster) are starting forming
- A key that is hashed into the cluster, will requires several attempts to resolve the collision. After several attempts it will add up to the cluster, making the cluster bigger.
- This is called primary clustering.

Performance

Expected Number of Probes
for Insertions and Unsuccessful Searches

\[ \frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)^2} \right) \]

for Successful Searches

\[ \frac{1}{2} \left( 1 + \frac{1}{1 - \lambda} \right) \]

\( \lambda \) is load factor
Collision Resolution Analysis

- Assume collision resolution is random.
  - \( f(i) = \) a random number between \( 0 \) and TableSize-1
- Load factor is \( \lambda \) (fraction of cells that are full)
- Fraction of cells that are empty is \( 1-\lambda \)
- Then expected number of cells to probe for unsuccessful search is: \( 1/ (1-\lambda) \)

Cost of average successful search

- The cost of a successful search of item \( x \) is:
  - Equal to the Cost of inserting that item \( x \) (that was done previously).
  - When we insert items, load factor increasing, hence the insertion cost of later items is bigger
  - Compute average cost of \( N \) items from the insertion cost of \( N \) items.

\[
\frac{1}{\lambda} \int_{x=0}^{\lambda} \frac{1}{1-x} \, dx = \frac{1}{\lambda} \ln\left(\frac{1}{1-\lambda}\right)
\]

For empty table, the load factor is : 0
After the last element that is inserted, the load factor is : \( \lambda \)
Therefore, the load factor is changing from 0 to \( \lambda \)
Linear Probing

- As a rule of thumb:
  - Linear probing is a bad idea if the load factor is expected to grow beyond 0.5
  - Rehashing should be used to grow the hash table if the load factor is more than 0.5 and linear hashing is wanted to be used.

- Comments
  - Linear probing causes primary clustering
  - Simple collision resolution function to evaluate.
Quadratic Probing

- Eliminates primary clustering
- Collision resolution function is a quadratic function
  - \( f(i) = i^2 \)
- Causes secondary clustering
- Rule of thumbs for using quadratic probing
  - TableSize should be prime
  - Load factor should be less than 0.5, otherwise table needs to rehashed.

Example

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

Primary clusters eliminated.
### Quadratic Probing

- There is no guarantee to find an empty cell is table is more than half full.
- If table is less than half full, it is guaranteed that we can find an empty cell by quadratic probing where we can insert a colliding item.
  - Table size must be prime to have this condition hold.
Theorem

- If quadratic probing is used, and the table size is prime, than a new element can always be inserted if the table is at least half empty.

Proof:
- Let the table size (T) be a prime number greater than 3.
- We will first show that:
  - For a given key x, that need to be inserted, the first \( k = \text{upper}(T/2) \) alternative locations are all distinct.
  - Namely, \( h_1(x), h_2(x), h_3(x), \ldots, h_{k-1}(x) \) are all distinct.

Let \( i \) and \( j \) be two probes so that \( i \neq j \)
Suppose that the probes map to the same location:
\[
\text{hash}(x) + i^2 = \text{hash}(x) + j^2 \pmod{\ T}
\]
\[
i^2 = j^2 \pmod{\ T}
\]
\[
i^2 - j^2 = 0 \pmod{\ T}
\]
\[
(i - j)(i + j) = 0 \pmod{\ T}
\]

Since \( T \) is prime, either \((i - j)\) or \((i + j)\) should be equal to zero.

Since \( i \) is not equal to \( j \), \((i - j)\) can not be zero.
Since \( i \) and \( j \) are greater or equal to zero and they are distinct, \((i + j)\) can not be zero.

Therefore, \( \text{th} \) and \( \text{th} \) probes (locations ) can not be equal.

Since there are \( \left\lfloor T/2 \right\rfloor \) probes that are different and there are at most \( \left\lfloor T/2 \right\rfloor / \text{items in the hash table (table is half - full at most)} \), then we are guaranteed that we fill find an empty cell by used quadratic probing.
### Notes to keep in mind

- Table must be at least half empty
  - Load factor smaller than 0.5
- Table size must be prime
- Deletions should be lazy.
  - The item should not be removed, but just marked as invalid.
- Otherwise, the deleted cell might have caused a collision to go past it.
  - That item is needed to find the next item in probe sequence.

### Hash Table Class with Quadratic Probing

```cpp
template <class HashedObj>
class HashTable
{
    public:
        explicit HashTable( const HashedObj & notFound, int size = 101 );
        HashTable( const HashTable & rhs ) :
            ITEM_NOT_FOUND( rhs.ITEM_NOT_FOUND ),
            array( rhs.array ), currentSize( rhs.currentSize ) { }

        const HashedObj & find( const HashedObj & x ) const;
        void makeEmpty( );
        void insert( const HashedObj & x );
        void remove( const HashedObj & x );
        const HashTable & operator=( const HashTable & rhs );
    enum EntryType { ACTIVE, EMPTY, DELETED };
} 
```
Hash Table Class with Quadratic Probing

private:

struct HashEntry
{
    HashedObj element;
    EntryType info;
    
    HashEntry( const HashedObj & e = HashedObj( ),
               EntryType i = EMPTY )
    : element( e ), info( i ) {} 
}

vector<HashEntry> array;
int currentSize;
const HashedObj ITEM_NOT_FOUND;

bool isActive( int currentPos ) const;

int findPos( const HashedObj & x ) const;

void rehash( );

Find

template <class HashedObj>
const HashedObj & HashTable<HashedObj>::find( const HashedObj & x )
const:
{
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
        return array[ currentPos ].element;
    else
        return ITEM_NOT_FOUND;
}
template <class HashedObj>
int HashTable<HashedObj>::findPos( const HashedObj & x ) const
{
    int collisionNum = 0;
    int currentPos = hash( x, array.size( ) );

    while( array[ currentPos ].info != EMPTY &&
        array[ currentPos ].element != x )  /* search for item */
    {
        currentPos += 2 * (++collisionNum) - 1;  // Compute ith probe
        if( currentPos >= array.size() )
            currentPos = array.size( );
    }

    return currentPos;
}

(i-1)^2 = i^2 - 2i + 1
then i^2 = (i-1)^2 + (2i - 1)
i^th probe is (2i-1) more than the (i-1)^th probe.

template <class HashedObj>
void HashTable<HashedObj>::insert( const HashedObj & x )
{
    // Insert x as active
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
        return;  // return without inserting
    array[ currentPos ] = HashEntry( x, ACTIVE );  // create an active hash entry
    // Rehash; see Section 5.5
    if( ++currentSize > array.size( ) / 2 )  /* load factor greater then 0.5
        /* double the hash table size.
    }
Remove

```cpp
/**
 * Remove item x from the hash table.
 */
template <class HashedObj>
void HashTable<HashedObj>::remove( const HashedObj & x )
{
    int currentPos = findPos( x );
    if( isActive( currentPos ) )  // item to be deleted found
        array[ currentPos ].info = DELETED;
}
```

Quadratic Probing Review

- Causes secondary clustering.
  - Elements that hash to the same position will probe the same alternative cells.
- Load factor should not exceed 0.
- Table size should be a prime number.
Double Hashing

- Two hash functions are used.
  - hash(x) = hash_1(x) + i * hash_2(x), where i \geq 0.

![Hash Table Diagram]

Double Hashing Tips

- Choice of hash_2(x) is very important.
  - A poor choice would not help to resolve collisions.
- hash_2 should never evaluate to zero.
- Table size should be prime.
- hash_2(x) = R - (x mod R) would work as a second hash function.
  - R is a prime number here.
Example

- TableSize is again 10.
- 1\textsuperscript{st} hash function = x mod 10
- 2\textsuperscript{nd} has function = 7 - x mode 7

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

Primary and secondary clusters eliminated.
Double Hashing

- Eliminates primary and secondary clustering
- Two hash functions computed.
  - More cost per operation.
- If table size is not prime, than we can run out of alternative positions much quickly.
Extendible Hashing

- All methods so far assumed that hash table can fit in memory.
- For large amount of data, this may not be true
  - Data items should reside in disk in this case.
- A directory that will ease to reach data items can be kept in memory
  - If it is too big, it too can be stored in disk.

```
N: Number of items to be stored
M: Maximum number of items that can be stored in a disk block.

Directory (root)

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00100</td>
<td>010100</td>
<td>100000</td>
<td>111000</td>
</tr>
<tr>
<td></td>
<td>001000</td>
<td>011000</td>
<td>101000</td>
<td>111001</td>
</tr>
<tr>
<td></td>
<td>001010</td>
<td>011010</td>
<td>101100</td>
<td>111011</td>
</tr>
<tr>
<td></td>
<td>001010</td>
<td>011010</td>
<td>101100</td>
<td>111011</td>
</tr>
</tbody>
</table>

\( d_L \): number of bits of a leaf that are common
```
After insertion of 100100 and leaf and directory split

After insertion of 000000 and leaf split