Probing Hash Tables

- Disadvantages of separate chaining:
  - Requires the use of another data structure, linked lists.
  - Slow when allocating new elements.
- Open Addressing: when a collision occurs try alternative cells until an empty cell is found.
- $h_0(x), h_1(x), h_2(x), \ldots$
- $h_i(x) = (\ hash(x) + f(i)\ ) \bmod TableSize; \quad f(0) = 0$
- f = collusion resolution strategy:
  - Linear probing $f(i) = i$
  - Quadratic probing $f(i) = i^2$
  - Double Hashing $f(i) = i * hash_2(x)$
- Needs a big table with load factor < 0.5

Linear Probing

- h(x) = (x + i) mod TableSize
- Example: TableSize = 10, insert: 89, 18, 49, 58, 69

  => primary clustering (clusters of occupied cells start forming)

	Initial	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Linear Probing

- As the hash table begins to fill up the probability that a record can be inserted in its home cell decreases.
- Problem: How to estimate the time for insertion (unsuccessful search)?
- Assume probe sequence follows a random permutation.
- Probability to find a cell occupied = N/M =λ
- Probability to find both home cell and next cell in the probe sequence occupied:

$$\frac{N}{M}\frac{(N-1)}{(M-1)}$$

- Probability of i collisions:

$$\frac{N}{M}\frac{(N-1)}{(M-1)}\cdots\frac{(N-i+1)}{(M-i+1)}\approx\left(\frac{N}{M}\right)^i$$

Linear Probing

- Expected number of probes:

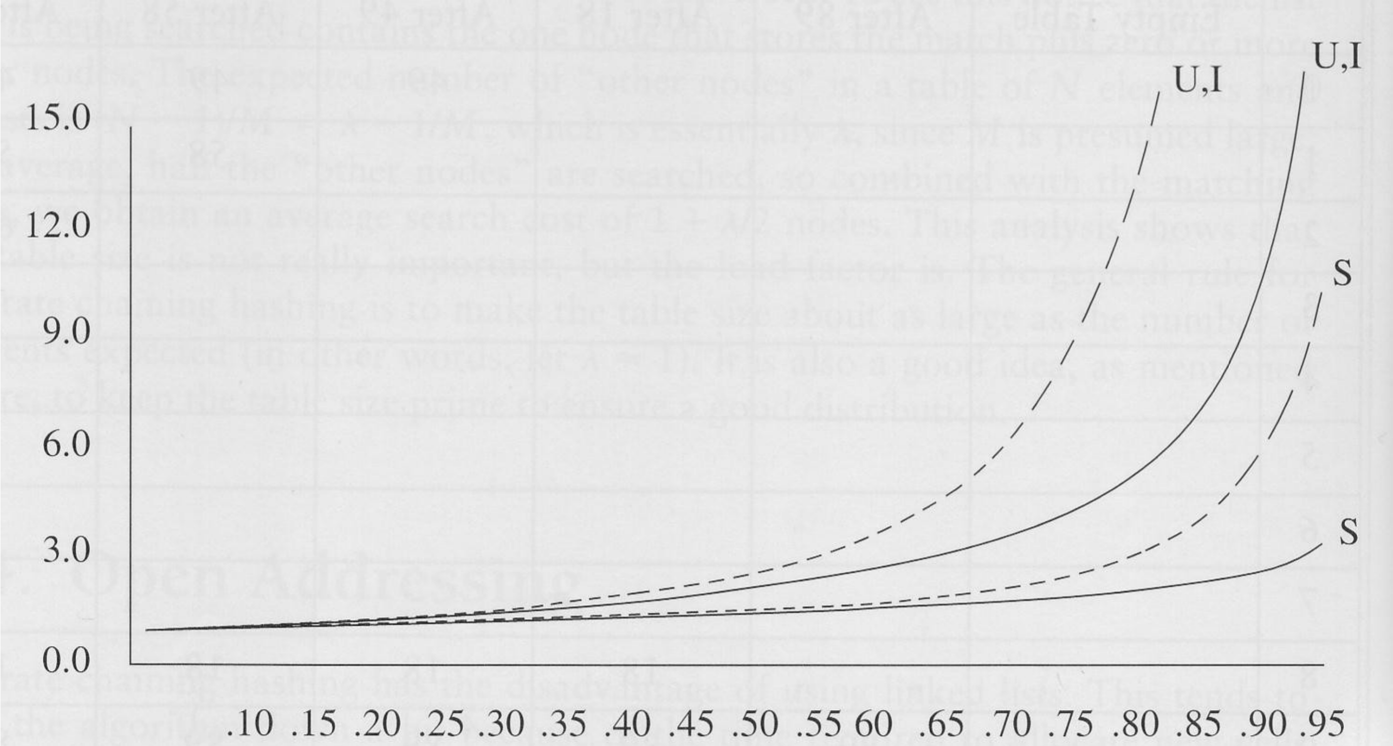
$$1+\sum_{i=1}^{\infty}\left(\frac{N}{M}\right)^{i}=\frac{1}{1-\lambda}$$

- Estimate of insertion time = average over all insertion times:

$$I(\lambda)=\frac{1}{\lambda}\int_{0}^{\lambda}\frac{1}{1-x}dx=\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$

- Lower bound on the expected running time for insertion in the average case.

Linear Probing



U = unsuccessful search, I = insertion, S = successful search
Dashed – linear probing, Solid line – random probing

Linear probing can be a bad idea if the table is expected to be more than half full

Quadratic Probing

- $h(x) = (x + i^2) \bmod TableSize$
- Eliminates the primary clustering problem
- Example: TableSize = 10, insert: 89, 18, 49, 58, 69
- Problem: if we delete 89 then a next search will fail => use lazy deletion
- Secondary clustering => elements that hash to the same cell will probe the same alternative cells => not so bad in practice.

	Initial	After 89	After 18	After 49	After 58	After 69
0				49	49	*49*
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	*18*	18	18
9		89	*89*	*89*	*89*	89

Quadratic Probing

- Problem: There is no guarantee of finding an empty cell once the table gets more than half full.

  Worse if TableSize is not prime!
- Theorem: If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.
- Proof:

  TableSize = M > 3, prime number

  Show that the first $\lceil M/2 \rceil$ alternative locations are all distinct.

Proof by contradiction:

Assume that two of these locations $(h(x) + i^2) \bmod M$ and $(h(x) + j^2) \bmod M$ are the same but $i \neq j$.

$0 \le i, j \le \lfloor M/2 \rfloor$

Quadratic Probing

=> $h(x) + i^2 = h(x) + j^2 \pmod{M}$

=> $i^2 = j^2 \pmod{M}$

=> $i^2 - j^2 = 0 \pmod{M}$

=> $(i - j)(i + j) = 0 \pmod{M}$

M is prime => either (i - j) or (i + j) is 0 (mod M).

But $i \neq j$ => first one is not possible

$0 <= i, j <= \lfloor M/2 \rfloor$ => second one is impossible

⇒ The first $\lceil M/2 \rceil$ alternative locations are distinct.

⇒ If at most $\lfloor M/2 \rfloor$ cells are taken then we can always find an empty cell.

Example: M =16 not prime => alternative locations at distances 1, 4, and 9.

Class Interface

```
template <typename HashedObj>
class HashTable
{
 public:
        explicit HashTable( int size = 101 );

        bool contains ( const HashedObj & x ) const;

        void makeEmpty( );
        void insert( const HashedObj & x );
        void remove( const HashedObj & x );

        enum EntryType { ACTIVE, EMPTY, DELETED };

  private:
        struct HashEntry
        {
                        HashedObj element;
                        EntryType info;         //state

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

        vector<HashEntry> array;
        int currentSize;

        bool isActive( int currentPos ) const;
        int findPos( const HashedObj & x ) const;
        void rehash( );
        int myhash ( const HashedObj & x ) const;
};
```

Constructor

```
explicit HashTable( int size = 101 ) :
                                    array( nextPrime( size ) )
{
      makeEmpty( );
}

void makeEmpty( )
{
      currentSize = 0;
      for( int i = 0; i < array.size( ); i++ )
                  array[ i ].info = EMPTY;
}
```

contains

```
bool contains( const HashedObj & x ) const
{
        return isActive( findPos( x ) );
}


int findPos( const HashedObj & x ) const                    //internal
{
        int offset = 1;
        int currentPos = myhash( x );

        while( array[ currentPos ].info != EMPTY &&
        array[ currentPos ].element != x )
        {
                currentPos += offset;         // Compute ith probe
                offset += 2;
                if( currentPos >= array.size( ) )
                        currentPos -= array.size( );
        }
         return currentPos;
}

bool isActive( int currentPos ) const
{
        return array[ currentPos ].info == ACTIVE;
}
```

Insert & remove

```
bool insert( const HashedObj & x )
{
                    // Insert x as active
      int currentPos = findPos( x );
      if( isActive( currentPos ) )
                    return false;

      array[ currentPos ] = HashEntry( x, ACTIVE );
                    // Rehash; see Section 5.5
      if( ++currentSize > array.size( ) / 2 )
                    rehash( );
      return true;
}

bool remove( const HashedObj & x )
{
      int currentPos = findPos( x );
      if( !isActive( currentPos ) )
                    return false;

      array[ currentPos ].info = DELETED;
      return true;
}
```

Double Hashing

- $h(x) = (hash_1(x) + i*hash_2(x)) \bmod TableSize$
- Good function: $hash_2(x) = R – (x \bmod R)$ where $R < TableSize$ and is prime
- Example: TableSize = 10, R = 7 insert: 89, 18, 49, 58, 69, 60
- What if we insert 23 ?
- Simulations show that the expected number of probes is the same as in random probing.

	Initial	After 89	After 18	After 49	After 58	After 69	After 60
0						*69*	69
1							
2							60
3					58	*58*	58
4							
5							
6				49	49	*49*	49
7							
8			18	*18*	18	18	18
9		89	*89*	89	*89*	*89*	89

Rehashing

- Assume linear probing. Insert: 13, 15, 6, 24
- If table too full => slow operations and insertions might fail.
- Solution: build a table twice as big and rehash

0	6
1	15
2	
3	24
4	
5	
6	13

x mod 7

Insert 23 =>

0	6
1	15
2	23
3	24
4	
5	
6	13

rehashing =>
T(N) = O(N)

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

x mod 17

Rehashing

- When to rehash?
  1. As soon as the table is half full.
  2. When an insertion fails.
  3. When a certain load factor is reached.
- Third strategy will be the best.

Rehashing

```
// rehashing for quadratic probing hash table

void rehash( )
{
        vector<HashEntry> oldArray = array;

                        // Create new double-sized, empty table
        array.resize( nextPrime( 2 * oldArray.size( ) ) );
        for( int j = 0; j < array.size( ); j++ )
                        array[ j ].info = EMPTY;

                        // Copy table over
        currentSize = 0;
        for( int i = 0; i < oldArray.size( ); i++ )
                        if( oldArray[ i ].info == ACTIVE )
                                    insert( oldArray[ i ].element );
}
```

Application of hashing

- Compilers: symbol tables – keeps track of the declared variables in the source code.
- Graph problems: map names into integers
- Game programs: transposition table – keeps track of player's position.
- Spelling checker: words can be checked in constant time.