Hashing - 2

CS 202 – Fundamental Structures of
Computer Science II

Bilkent University

Computer Engineering Department

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.

Open Addressing

- The cells that are tried successively can be expressed formally as:
  - ❑ $h_0(x)$, $h_1(x)$, $h_2(x)$, …
    - $h_0(x)$ is the initial cells that causes a collision.
    - $h_1(x)$, $h_2(x)$, …are alternative cells.
  - ❑ $h_i(x)$ = (hash(x) + f(i)) mode TableSize
    - f(i) is *collision resolution strategy* (function).
    - f(0) = 0.

Open Addressing

- There are varies methods as open addressing schemes:
  - Linear Probing
    - hash(x) = hash(x) + f(i) = i, where i >= 0
  - Quadratic Probing
    - hash(x) = hash(x) + f(i) = i^2, where i >= 0
  - Double Hashing
    - hash(x) = $hash_1(x)$ + i · $hash_2(x)$, where i >= 0

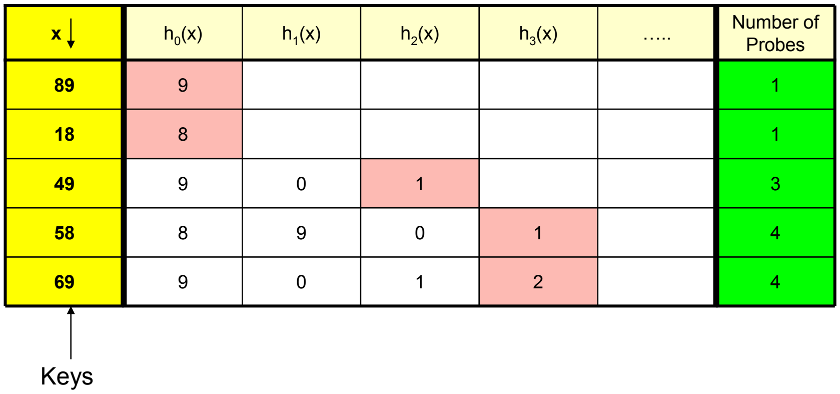
Linear Probing

- 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 hash(x) = x mod 10.
    - Collision resolution strategy is f(i) = i;

Example

Cell number	Empty Table	After inserting 89	After inserting 18	After inserting 49	After inserting 58	After inserting 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

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



x ↓	$h_0(x)$	$h_1(x)$	$h_2(x)$	$h_3(x)$	…..	Number of Probes
89	9					1
18	8					1
49	9	0	1			3
58	8	9	0	1		4
69	9	0	1	2		4

Keys

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 Unsuccesful Searches

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

for Succesful Searches

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

λ is load factor

Collision Resolution Analysis

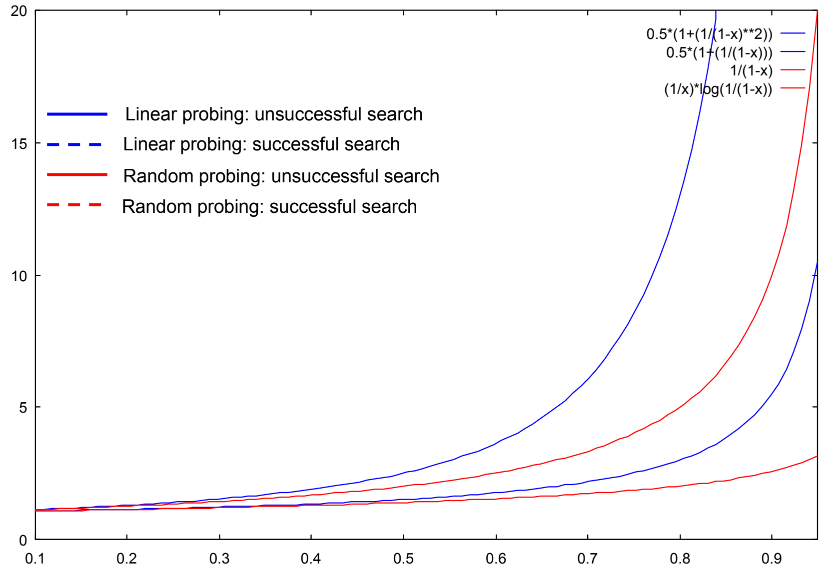
- Assume collision resolution is random.
  - f(i) = a random number between *0* and *TableSize-1*
- Load factor is λ (fraction of cells that are full)
- Fraction of cells that are empty is 1-λ
- Then expected number of cells to probe for unsuccessful search is: 1/ (1-λ)

Cost of average successful searcgh

- 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.

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

For empty table, the load factor is : 0
After the last element that is inserted, the load factor is : λ
Therefore, the load factor is changing from 0 to λ



Load Factor versus Number of Probes

Linear Probing

- As a rule of thumb:
  - Linear probing is bad idea if load factor is expected to grow beyond 0.5
  - Rehashing should be used to grow the hash table if 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

Cell number	Empty Table	After inserting 89	After inserting 18	After inserting 49	After inserting 58	After inserting 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

Primary clusters eliminated.

x ↓	$h_0(x)$	$h_1(x)$	$h_2(x)$	$h_3(x)$	…..	Number of Probes
89	9					1
18	8					1
49	9	0				2
58	8	9	2			3
69	9	0	3			3

Keys

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= upper(T/2) alternative locations are all distinct.
      - Namely, $h_1(x)$, $h_2(x)$, $h_2(x)$,…. $h_{k-1}(x)$ are all distinct.

$hash(x) = hash(x) + i^2, \quad 0 <= i, j <= \lfloor T/2 \rfloor$

Let i and j be two probes so that $i \neq j$

Suppose that the probes map to the same location :

$$h(x) + i^2 = h(x) + j^2 \quad (\text{mod } T)$$
$$i^2 = j^2 \quad (\text{mod } T)$$
$$i^2 - j^2 = 0 \quad (\text{mod } T)$$
$$(i - j)(i + j) = 0 \quad (\text{mod } T)$$

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, i^{th} and j^{th} probes (locations) can not be equal.

Since there are $\lceil T/2 \rceil$ probes that are different and there are at most $\lfloor T/2 \rfloor$ 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

```
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;
       }
```

FindPos

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

```
template <class HashedObj>
    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;
    }
```

Insert

```
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
            rehash( );                       /* double the hash table size.
    }
```

Remove

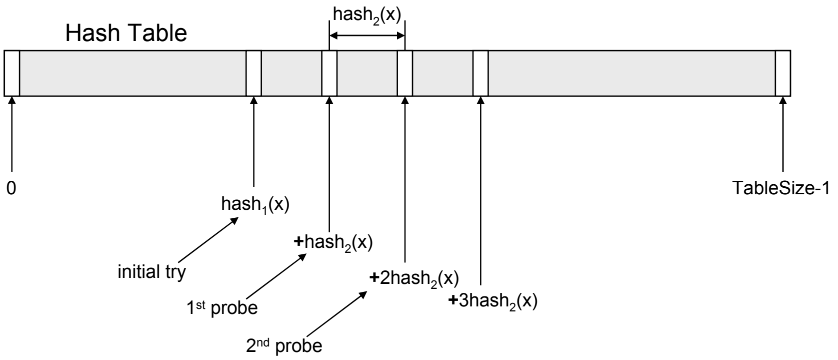
```
 /**
  * Remove item x from the hash table.
  */
template <class HashedObj>
void HashTable<HashedObj>::remove( const HashedObj & x )
{
    int currentPos = findPos( x );
    if( isActive( currentPos ) )  .// item to be deleted found
        array[ currentPos ].info = DELETED;
}
```

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 $i >= 0$.



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.
- $hash_2(x) = R - (x \bmod R)$ would work as a second hash function.
  - R is a prime number here.

Example

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

Example

Cell number	Empty Table	After inserting 89	After inserting 18	After inserting 49	After inserting 58	After inserting 69
0						69
1						
2					58	58
3						
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Primary and secondary clusters eliminated.

x ↓	$h_0(x)$	$h_1(x)$	$h_2(x)$	$h_3(x)$	…..	Number of Probes
89	9					1
18	8					1
49	9	6				2
58	8	3				2
69	9	0				2

Keys

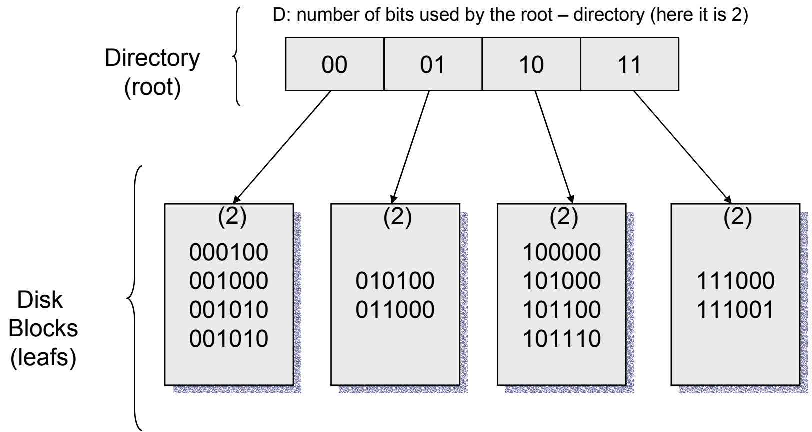
Double Hashing

- 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.



D: number of bits used by the root – directory (here it is 2)

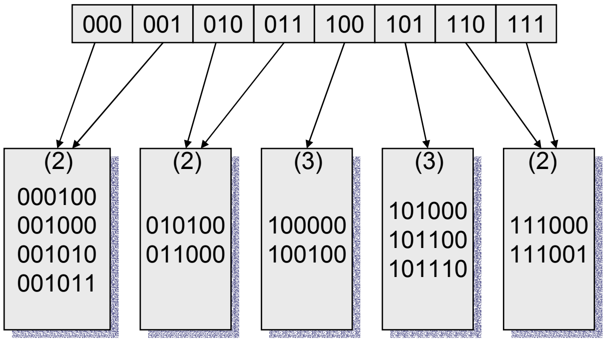
Directory (root)

00	01	10	11

Disk Blocks (leafs)

(2)	(2)	(2)	(2)
000100 001000 001010 001010	010100 011000	100000 101000 101100 101110	111000 111001

d_L: number of bits of a leaf that are common



000 001 010 011 100 101 110 111

(2)
000100
001000
001010
001011

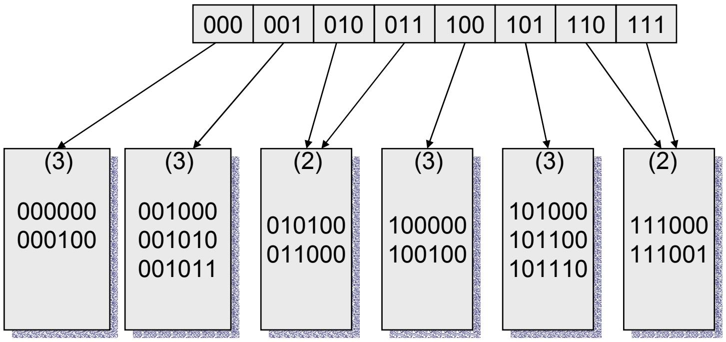
(2)
010100
011000

(3)
100000
100100

(3)
101000
101100
101110

(2)
111000
111001

After insertion of 100100 and <u>leaf and directory split</u>



000 001 010 011 100 101 110 111

(3)
000000
000100

(3)
001000
001010
001011

(2)
010100
011000

(3)
100000
100100

(3)
101000
101100
101110

(2)
111000
111001

After insertion of 000000 and <u>leaf split</u>