Data Structures
Lesson 8

BSc in Computer Science
University of New York, Tirana

Assoc. Prof. Marenglen Biba

Hash Tables

- The implementation of hash tables is frequently called hashing, and it performs insertions, deletions, and finds in constant average time.
- Unlike with the binary search tree, the average-case running time of hash table operations is based on statistical properties rather than the expectation of random-looking input.
- This improvement is obtained at the expense of a loss of ordering information among the elements: Operations such as findMin and findMax and the printing of an entire table in sorted order in linear time are not supported.

Hash Tables

- The hash table supports the retrieval or deletion of any named item.
- We want to be able to support the basic operations in constant time, as for the stack and queue.

- When the size of the set increases, searches in the set should take longer.
  - However, that is not necessarily the case.

Hash Table

- Suppose that all the items we are dealing with are small nonnegative integers, ranging from 0 to 65,535.
- We can use a simple array to implement each operation as follows.
- First, we initialize an array **a** that is indexed from 0 to 65,535 with all 0s.
- To perform insert(i), we execute a[i]++. Note that a[i] represents the number of times that i has been inserted.
- To perform find(i), we verify that a[i] is not 0.
- To perform remove(i), we make sure that a[i] is positive and then execute a[i]--.
- The time for each operation is clearly constant; even the overhead of the array initialization is a constant amount of work (65,536 assignments).

Hash Tables

- There are two problems with this solution.
- First, suppose that we have 32-bit integers instead of 16-bit integers.
  - Then the array **a** must hold 4 billion items, which is impractical.
- Second, if the items are not integers but instead are strings (or something even more generic), they cannot be used to index an array.

Strings

- The second problem is not really a problem at all.
- Just as a number 1234 is a collection of digits 1, 2, 3, and 4, the string "junk" is a collection of characters 'j', ‘u' , 'n', and 'k'.
- Note that the number 1234 is just $1 * 10^3 + 2 * 10^2 + 3 * 10^1 + 4 * 10^0$.
- Recall that an ASCII character can typically be represented in 7 bits as a number between 0 and 127.
- Because a character is basically a small integer, we can interpret a string as an integer.
- One possible representation is $'j' * 128^3 + 'u' * 128^2 + 'u' * 128^1 + 'k' * 128^0$.
- This approach allows the simple array implementation discussed previously.

Representation

- The problem with this strategy is that the integer representation described generates huge integers:
  - The representation for "junk" yields 224,229,227, and longer strings generate much larger representations.
- This result brings us back to the first problem: How do we avoid using an absurdly large array?

Avoiding large arrays: Mapping

- How to avoid using a large array?
- We do so by using a function that maps large numbers (or strings interpreted as numbers) into smaller, more manageable numbers.
- A function that maps an item into a small index is known as a hash function.
- If x is an arbitrary (nonnegative) integer, then x % tableSize generates a number between 0 and tableSize-1 suitable for indexing into an array of size tableSize.
- If s is a string, we can convert s to a large integer x by using the method suggested previously and then apply the mod operator (%) to get a suitable index.

Collisions

- The use of the hash function introduces a complication: Two or more different items can hash out to the same position, causing a collision.
- This situation can never be avoided because there are many more items than positions.
- However, many methods are available for quickly resolving a collision.
- We investigate three of the simplest: linear probing, quadratic probing, and separate chaining.
- Each method is simple to implement, but each yields a different performance, depending on how full the array is.

Hash function

- Computing the hash function for strings has a subtle complication:
  - The conversion of the String s to x generates an integer that is almost certainly larger than the machine can store conveniently — because $128^4 = 2^{28}$.
- This integer size is a factor of 8 from the largest int.
- Consequently, we cannot expect to compute the hash function by directly computing powers of 128.
- Instead, we use the following observation.

Hash function

- A general polynomial

$$A_3X^3 + A_2X^2 + A_1X^1 + A_0X^0 \qquad \textbf{(20.1)}$$

can be evaluated as

$$(((A_3)X + A_2)X + A_1)X + A_0 \qquad \textbf{(20.2)}$$

Hash function

- Note that in Equation 20.2, we avoid computation of the polynomial directly, which is good for three reasons.
- First, it avoids a large intermediate result, which, as we have shown, overflows.
- Second, the calculation in the equation involves only three multiplications and three additions:
  - an N-degree polynomial is computed in N multiplications and additions.
- These operations compare favorably with the computation in Equation 20.1.
- Third, the calculation proceeds left to right (A3 corresponds to 'j', A2 to ' u', and so on, and X is 128).

Hash function

- However, an overflow problem persists:
- The result of the calculation is still the same and is likely to be too large.
- But, we need only the result taken mod tableSize.
- By applying the % operator after each multiplication (or addition), we can ensure that the intermediate results remain small.
- The resulting function is as follows: => next slide

Hash function: a first attempt

figure 20.1

A first attempt at a hash function implementation

```
// Acceptable hash function
public static int hash( String key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal = ( hashVal * 128 + key.charAt( i ) )
                                            % tableSize;
    return hashVal;
}
```

Hash function: a first attempt

- An annoying feature of this hash function is that the mod computation is expensive.
- Because overflow is allowed (and its results are consistent on a given platform), we can make the hash function somewhat faster by performing a single mod operation immediately prior to the return.
- Unfortunately, the repeated multiplication by 128 would tend to shift the early characters to the left — out of the answer.
- To alleviate this situation, we multiply by 37 instead of 128, which slows the shifting of early characters. => next slide

A faster hash function

figure 20.2

A faster hash function that takes advantage of overflow

```
1   /**
2    * A hash routine for String objects.
3    * @param key the String to hash.
4    * @param tableSize the size of the hash table.
5    * @return the hash value.
6    */
7   public static int hash( String key, int tableSize )
8   {
9       int hashVal = 0;
10
11      for( int i = 0; i < key.length( ); i++ )
12          hashVal = 37 * hashVal + key.charAt( i );
13
14      hashVal %= tableSize;
15      if( hashVal < 0 )
16          hashVal += tableSize;
17
18      return hashVal;
19  }
```

Note that overflow could introduce negative numbers.
Thus if the mod generates a negative value, we make it positive (lines 15 and 16).
Also note that the result obtained by allowing overflow and doing a final mod is not the same as performing the mod after every step. Thus we have slightly altered the hash function — which is not a problem.

Distribution of keys

- Although speed is an important consideration in designing a hash function, we also want to be sure that it distributes the keys equitably.
- Consequently, we must not take our optimizations too far.
- An example is the hash function shown in Figure 20.3.

A poor hash function

```java
// A poor hash function when tableSize is large
public static int hash( String key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
       hashVal += key.charAt( i );

    return hashVal % tableSize;
}
```

figure 20.3

A bad hash function if `tableSize` is large

It simply adds the characters in the keys and returns the result mod tableSize.
The function is easy to implement and computes a hash value very quickly.
But does not distribute the keys well.

Equitable distribution

- Suppose that tableSize is 10,000.
- Also suppose that all keys are 8 or fewer characters long.
- Because an ASCII char is an integer between 0 and 127, the hash function can assume values only between 0 and 1,016 (127 x 8).
- This restriction certainly does not permit an equitable distribution.
- Any speed gained by the quickness of the hash function calculation is more than offset by the effort taken to resolve a larger than expected number of collisions.

Linear probing

- Now that we have a hash function, we need to decide what to do when a collision occurs.
- Specifically, if X hashes out to a position that is already occupied, where do we place it?
- The simplest possible strategy is linear probing, or searching sequentially in the array until we find an empty cell.
- The search wraps around from the last position to the first, if necessary.
- Figure 20.4 shows the result of inserting the keys 89, 18, 49, 58, and 9 in a hash table when linear probing is used.
- We assume a hash function that returns the key X mod the size of the table.

Linear probing

figure 20.4

Linear probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

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

Linear probing

- So long as the table is large enough, a free cell can always be found.
- However, the time needed to find a free cell can get to be quite long.
- For example, if there is only one free cell left in the table, we may have to search the entire table to find it.
- On average we would expect to have to search half the table to find it, which is far from the constant time per access that we are hoping for.
- But, if the table is kept relatively empty, insertions should not be so costly. We discuss this approach shortly.

Find

- The find algorithm merely follows the same path as the insert algorithm.
- If it reaches an empty slot, the item we are searching for is not found; otherwise, it finds the match eventually.
- For example, to find 58, we start at slot 8 (as indicated by the hash function).
- We see an item, but it is the wrong one, so we try slot 9.
- Again, we have an item, but it is the wrong one, so we try slot 0 and then slot 1 until we find a match.
- A find for 19 would involve trying slots 9, 0, 1, and 2 before finding the empty cell in slot 3. Thus 19 is not found.

Deletion

- Standard deletion cannot be performed because, as with a binary search tree, an item in the hash table not only represents itself, but it also connects other items by serving as a placeholder during collision resolution.
- Thus, if we removed 89 from the hash table, virtually all the remaining find operations would fail.
- Consequently, we implement lazy deletion, or marking items as deleted rather than physically removing them from the table.
- This information is recorded in an extra data member. Each item is either active or deleted.

Naive analysis of linear probing

To estimate the performance of linear probing, we make two assumptions:

1. The hash table is large
2. Each probe in the hash table is independent of the previous probe.

(Probe: to search into or examine thoroughly)

Naive analysis of linear probing

- Assumption 1 is reasonable; otherwise, we would not be bothering with a hash table.
- Assumption 2 says that, if the fraction of the table that is full is λ, each time we examine a cell the probability that it is occupied is also λ, independent of any previous probes.
- Independence is an important statistical property that greatly simplifies the analysis of random events.
- Unfortunately, as we have discussed, the assumption of independence is not only unjustified, but it also is erroneous.
- Thus the naive analysis that we perform is incorrect.
- Even so, it is helpful because it tells us what we can hope to achieve if we are more careful about how collisions are resolved.

Naive analysis of linear probing

- As mentioned earlier, the performance of the hash table depends on how full the table is. Its fullness is given by the load factor.

definition: The *load factor*, λ, of a probing hash table is the fraction of the table that is full. The load factor ranges from 0 (empty) to 1 (completely full).

Naive analysis of linear probing

We can now give a simple but incorrect analysis of linear probing in:

- Theorem 20.1:
- If independence of probes is assumed, the average number of cells examined in an insertion using linear probing is $1/(1 - \lambda)$.

- For a table with a load factor of λ, the probability of any cell's being empty is $1 - \lambda$.
- Consequently, the expected number of independent trials required to find an empty cell is $1/(1 - \lambda)$.

Independence does not hold

- Unfortunately, independence of probes does not hold.

Primary clustering

- In primary clustering, large blocks of occupied cells are formed.
- Any key that hashes into this cluster requires excessive attempts to resolve the collision, and then it adds to the size of the cluster.
- Not only do items that collide because of identical hash functions cause degenerate performance, but also an item that collides with an alternative location for another item causes poor performance.
- The mathematical analysis required to take this phenomenon into account is complex but has been solved, yielding:

Theorem 20.2.

- The average number of cells examined in an insertion using linear probing is roughly $(1 + 1/(1-\lambda)^2)/2$.

Find operation

Theorem 20.3.

- The average number of cells examined in an unsuccessful search using linear probing is roughly $(1 + 1/(1- \lambda)^2)/2$.
- The average number of cells examined in a successful search is approximately $(1 + 1/(1 - \lambda))/2$.

Primary clustering

- Primary clustering not only makes the average probe sequence longer, but it also makes a long probe sequence more likely.
- The main problem with primary clustering therefore is that performance degrades severely for insertion at high load factors.

Reducing the number of probes

- To reduce the number of probes, we need a collision resolution scheme that avoids primary clustering.
- Note, however, that, if the table is half empty, removing the effects of primary clustering would save only half a probe on average for an insertion or unsuccessful search and one-tenth a probe on average for a successful search.
- Even though we might expect to reduce the probability of getting a somewhat lengthier probe sequence, linear probing is not a terrible strategy.
- Because it is so easy to implement, any method we use to remove primary clustering must be of comparable complexity. Otherwise, we expend too much time in saving only a fraction of a probe. One such method is quadratic probing.

Quadratic probing

- Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing by examining certain cells away from the original probe point.
- Its name is derived from the use of the formula $F(i) = i^2$ to resolve collisions.
- Specifically, if the hash function evaluates to H and a search in cell H is inconclusive, we try cells $H + 1^2$, $H + 2^2$, $H + 3^2$,..., $H + i^2$ (employing wraparound) in sequence.
- This strategy differs from the linear probing strategy of searching H+1, H+2, H+3, ..., H+ i.

Quadratic Probing

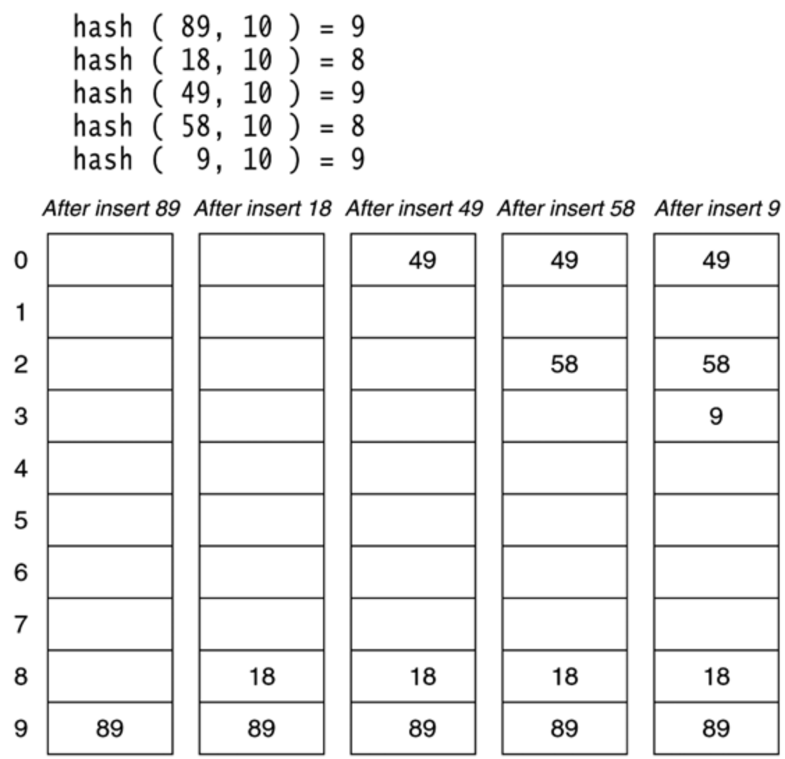


figure 20.6

A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).

58 collides at position 8. The cell at position 9 (which is one away) is tried, but another collision occurs.

A vacant cell is found at the next cell tried, which is $2^2 = 4$ positions away from the original hash position. Thus 58 is placed in cell 2.

Quadratic Probing: Issue 1

- In linear probing, each probe tries a different cell.
  - Does quadratic probing guarantee that, when a cell is tried, we have not already tried it during the course of the current access?
  - Does quadratic probing guarantee that, when we are inserting X and the table is not full, X will be inserted?

Quadratic Probing: Issue 2 and 3

- Linear probing is easily implemented. Quadratic probing appears to require multiplication and mod operations.
  - Does this apparent added complexity make quadratic probing impractical?

- What happens (in both linear probing and quadratic probing) if the load factor gets too high?
  - Can we dynamically expand the table, as is typically done with other array-based data structures?

Table size

- Fortunately, the news is relatively good on all cases.
- If the table size is prime and the load factor never exceeds 0.5, we can always place a new item X and no cell is probed twice during an access.
- However, for these guarantees to hold, we need to ensure that the table size is a prime number.

Theorem 20.4.

- 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. Furthermore, in the course of the insertion, no cell is probed twice.

Finding prime numbers

```java
/**
 * Method to find a prime number at least as large as n.
 * @param n the starting number (must be positive).
 * @return a prime number larger than or equal to n.
 */
private static int nextPrime( int n )
{
    if( n % 2 == 0 )
        n++;

    for( ; !isPrime( n ); n += 2 )
        ;

    return n;
}
```

figure 20.7

A routine used in quadratic probing to find a prime greater than or equal to N

Efficiency

- The second important consideration is efficiency.
- Recall that, for a load factor of 0.5, removing primary clustering saves only 0.5 probe for an average insertion and 0.1 probe for an average successful search.
- We do get some additional benefits: Encountering a long probe sequence is significantly less likely.

Efficiency

- The formula for quadratic probing suggests that this calculation appears to be much too expensive to be practical.
- However, we can use the following trick, as explained in
- Theorem 20.5.
- Quadratic probing can be implemented without expensive multiplications and divisions.

Dynamic expansion

- The final detail to consider is dynamic expansion.
- If the load factor exceeds 0.5, we want to double the size of the hash table. This approach raises a few issues.
- First, how hard will it be to find another prime number?
- The answer is that prime numbers are easy to find. We expect to have to test only O(logN) numbers until we find a number that is prime.
- Consequently, the routine shown in Figure 20.7 is very fast.
- The primality test takes at most $O(N^{1/2})$ time, so the search for a prime number takes at most O(N log N) time.
- This cost is much less than the O(N) cost of transferring the contents of the old table to the new.

Rehashing

- Once we have allocated a larger array, do we just copy everything over?
- The answer is most definitely no.
- The new array implies a new hash function, so we cannot use the old array positions.
- Thus we have to examine each element in the old table, compute its new hash value, and insert it in the new hash table.
- This process is called rehashing.
- Rehashing is easily implemented in Java.

figure 20.8

The class skeleton for a quadratic probing hash table

```
package weiss.util;

public class HashSet<AnyType> extends AbstractCollection<AnyType>
                              implements Set<AnyType>
{
    private class HashSetIterator implements Iterator<AnyType>
      { /* Figure 20.17 */ }
    private static class HashEntry implements java.io.Serializable
      { /* Figure 20.9 */ }

    public HashSet( )
      { /* Figure 20.10 */ }
    public HashSet( Collection<? extends AnyType> other )
      { /* Figure 20.10 */ }

    public int size( )
      { return currentSize; }
    public Iterator iterator( )
      { return new HashSetIterator( ); }

    public boolean contains( Object x )
      { /* Figure 20.11 */ }
    private static boolean isActive( HashEntry [ ] arr, int pos )
      { /* Figure 20.12 */ }
    public AnyType getMatch( AnyType x )
      { /* Figure 20.11 */ }

    public boolean remove( Object x )
      { /* Figure 20.13 */ }
    public void clear( )
      { /* Figure 20.13 */ }
    public boolean add( AnyType x )
      { /* Figure 20.14 */ }
    private void rehash( )
      { /* Figure 20.15 */ }
    private int findPos( Object x )
      { /* Figure 20.16 */ }

    private void allocateArray( int arraySize )
      { array = new HashEntry[ arraySize ]; }
    private static int nextPrime( int n )
      { /* Figure 20.7 */ }
    private static boolean isPrime( int n )
      { See online code  */ }

    private int currentSize = 0;
    private int occupied = 0;
    private int modCount = 0;
    private HashEntry [ ] array;
}
```

```java
private static class HashEntry implements java.io.Serializable
{
    public Object  element;   // the element
    public boolean isActive;  // false if marked deleted

    public HashEntry( Object e )
    {
        this( e, true );
    }

    public HashEntry( Object e, boolean i )
    {
        element  = e;
        isActive = i;
    }
}
```

figure 20.9

The `HashEntry` nested class

```java
private static final int DEFAULT_TABLE_SIZE = 101;

/**
 * Construct an empty HashSet.
 */
public HashSet( )
{
    allocateArray( DEFAULT_TABLE_SIZE );
    clear( );
}

/**
 * Construct a HashSet from any collection.
 */
public HashSet( Collection<? extends AnyType> other )
{
    allocateArray( nextPrime( other.size( ) * 2 ) );
    clear( );

    for( AnyType val : other )
        add( val );
}
```

figure 20.10

Hash table initialization

figure 20.11

The searching routines for a quadratic probing hash table

```
/**
 * This method is not part of standard Java.
 * Like contains, it checks if x is in the set.
 * If it is, it returns the reference to the matching
 * object; otherwise it returns null.
 * @param x the object to search for.
 * @return if contains(x) is false, the return value is null;
 * otherwise, the return value is the object that causes
 * contains(x) to return true.
 */
public AnyType getMatch( AnyType x )
{
    int currentPos = findPos( x );

    if( isActive( array, currentPos ) )
        return (AnyType) array[ currentPos ].element;
    return null;
}

/**
 * Tests if some item is in this collection.
 * @param x any object.
 * @return true if this collection contains an item equal to x.
 */
public boolean contains( Object x )
{
    return isActive( array, findPos( x ) );
}
```

```java
/**
 * Tests if item in pos is active.
 * @param pos a position in the hash table.
 * @param arr the HashEntry array (can be oldArray during rehash).
 * @return true if this position is active.
 */
private static boolean isActive( HashEntry [ ] arr, int pos )
{
    return arr[ pos ] != null && arr[ pos ].isActive;
}
```

figure 20.12

The `isActive` method for a quadratic probing hash table

```java
/**
 * Removes an item from this collection.
 * @param x any object.
 * @return true if this item was removed from the collection.
 */
public boolean remove( Object x )
{
    int currentPos = findPos( x );
    if( !isActive( array, currentPos ) )
        return false;

    array[ currentPos ].isActive = false;
    currentSize--;
    modCount++;

    if( currentSize < array.length / 8 )
        rehash( );

    return true;
}

/**
 * Change the size of this collection to zero.
 */
public void clear( )
{
    currentSize = occupied = 0;
    modCount++;
    for( int i = 0; i < array.length; i++ )
        array[ i ] = null;
}
```

figure 20.13

The `remove` and `clear` routines for a quadratic probing hash table

Small number of elements: rehash

figure 20.14

The add routine for a quadratic probing hash table

```java
/**
 * Adds an item to this collection.
 * @param x any object.
 * @return true if this item was added to the collection.
 */
public boolean add( AnyType x )
{
    int currentPos = findPos( x );
    if( isActive( array, currentPos ) )
        return false;

    array[ currentPos ] = new HashEntry( x, true );
    currentSize++;
    occupied++;
    modCount++;

    if( occupied > array.length / 2 )
        rehash( );

    return true;
}
```

Load factor > 0.5: rehash

figure 20.15

The rehash method for a quadratic probing hash table

```
/**
 * Private routine to perform rehashing.
 * Can be called by both add and remove.
 */
private void rehash( )
{
    HashEntry [ ] oldArray = array;

        // Create a new, empty table
    allocateArray( nextPrime( 4 * size( ) ) );
    currentSize = 0;
    occupied = 0;

        // Copy table over
    for( int i = 0; i < oldArray.length; i++ )
        if( isActive( oldArray, i ) )
            add( (AnyType) oldArray[ i ].element );
}
```

A new, empty hash table that will have a 0.25 load factor when rehash terminates.

```java
/**
 * Method that performs quadratic probing resolution.
 * @param x the item to search for.
 * @return the position where the search terminates.
 */
private int findPos( Object x )
{
    int offset = 1;
    int currentPos = ( x == null ) ?
                     0 : Math.abs( x.hashCode( ) % array.length );

    while( array[ currentPos ] != null )
    {
        if( x == null )
        {
            if( array[ currentPos ].element == null )
                break;
        }
        else if( x.equals( array[ currentPos ].element ) )
            break;

        currentPos += offset;                 // Compute ith probe
        offset += 2;
        if( currentPos >= array.length )      // Implement the mod
            currentPos -= array.length;
    }

    return currentPos;
}
```

figure 20.16

The routine that finally deals with quadratic probing

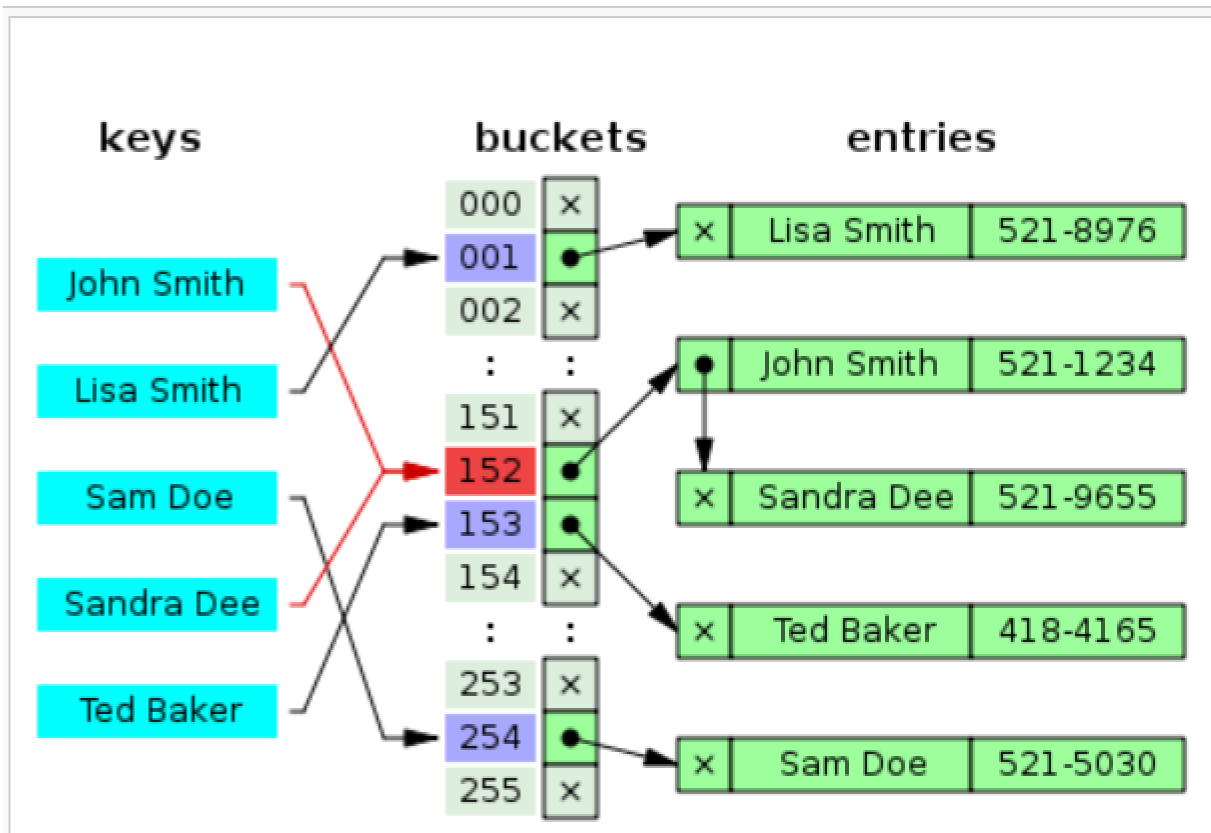
Implement the methodology described in Theorem 20.5, using two additions.

Cycle	currentPos	offset
0	1	3
1	4	5
2	9	7
3	16	9
4	25	11
5	36	13
6	49	15
7	64	17

Separate chaining hashing

- A popular and space-efficient alternative to quadratic probing is separate chaining hashing in which an array of linked lists is maintained.
- For an array of linked lists, L_0, L_1, ..., L_{M-1}, the hash function tells us in which list to insert an item X and then, during a find, which list contains X.
- The idea is that, although searching a linked list is a linear operation, if the lists are sufficiently short, the search time will be very fast.

Separate chaining hashing



Separate chaining hashing

- The appeal of separate chaining hashing is that performance is not affected by a moderately increasing load factor; thus rehashing can be avoided.

- For languages that do not allow dynamic array expansion, this consideration is significant.

- Furthermore, the expected number of probes for a search is less than in quadratic probing, particularly for unsuccessful searches.

Implementation of separate chaining hashing

- We can implement separate chaining hashing by using our existing linked list classes.
- However, because the header node adds space overhead and is not really needed, we could elect not to reuse components and instead implement a simple stacklike list.

Hash tables versus binary search trees

- We can also use binary search trees to implement insert and find operations.
- Although the resulting average time bounds are O(logN), binary search trees also support routines that require order and thus are more powerful.
- Using a hash table, we cannot efficiently find the minimum element or extend the table to allow computation of an order statistic.
- We cannot search efficiently for a string unless the exact string is known. A binary search tree could quickly find all items in a certain range, but this capability is not supported by a hash table.
- Furthermore, the O(log N) bound is not necessarily that much more than O(1), especially since no multiplications or divisions are required by search trees.

Hash tables versus binary search trees

- The worst case for hashing generally results from an implementation error, whereas sorted input can make binary search trees perform poorly.
- Balanced search trees are quite expensive to implement.

- *Hence, if no ordering information is required and there is any suspicion that the input might be sorted, hashing is the data structure of choice.*

Hashing applications

- Hashing applications are abundant.
- Compilers use hash tables to keep track of declared variables in source code.
- The data structure is called a symbol table.
- Hash tables are the ideal application for this problem because only insert and find operations are performed. Identifiers are typically short, so the hash function can be computed quickly.
- In this application, most searches are successful.

Hashing applications

- Another common use of hash tables is in game programs.
- As the program searches through different lines of play, it keeps track of positions that it has encountered by computing a hash function based on the position (and storing its move for that position).
- If the same position recurs, usually by a simple transposition of moves, the program can avoid expensive recomputation.
- This general feature of all game-playing programs is called the transposition table.
- Chess games can greatly benefit from this

Hashing applications

- Another use of hashing is in online spelling checkers.
- If misspelling detection (as opposed to correction) is important, an entire dictionary can be prehashed and words can be checked in constant time.
- Hash tables are well suited for this purpose because the words do not have to be alphabetized.
- Printing out misspellings in the order they occurred in the document is acceptable.

Readings

- Chapter 20