



EECS 560

DATA STRUCTURES

MODULE III: HASH TABLE

DISCLAIMER

- This document is intended to be used for studying EECS 560 Data Structures, only by students who are currently enrolled in the course.
- This document is a copyright of Dr. Cuncong Zhong. Distribution of this document, or use it for any purpose other than what is stated above, is considered as a copyright infringement. Dr. Cuncong Zhong reserves the right to take necessary legal action.
- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from *Data Structures and Algorithm Analysis in C++*, 4th edition, by Mark Allen Weiss.

HASH TABLE

- Hash table is an ADT that supports fast (ideally $O(1)$ time) search of an item and retrieve its related information
 - For example, we have registered all your favorite video games
 - We will be using a data structure to address the following need: given the name of the student, retrieve her/his favorite video game

| Name | Game |
|------|------------------|
| Alex | Warcraft |
| Mary | Minecraft |
| Lisa | Candy Crush Saga |
| Joy | Angry Bird |
| Tom | Call of Duty |

HASH TABLE

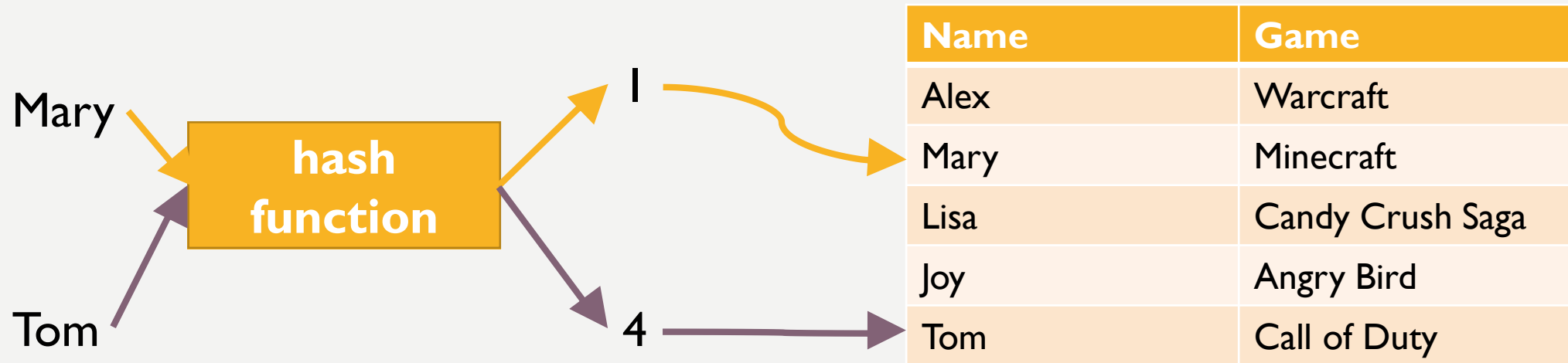
- The simplest solution is to go through the array, and find the name of the student to retrieve the name of the game
- However, in the worst case it will take $O(n)$ time, if we have n students
- Or, if we have spent extra time to sort the records alphabetically, we can use binary search to look for the name, and it will take $O(\log(n))$ time. It is faster but still not constant.

HASH TABLE

- In the context of hash table
 - The item we use to perform the search is called “key” (i.e., the name of the student). We assume all keys are unique (e.g., no two students have identical names).
 - The information we are searching for is called “value” (i.e., the name of the video game). Duplicated values are allowed (e.g., two students may love the same video game, that’s OK).
 - Each hash table record is in fact a key-value pair.
 - The records are stored in a fix-sized array. That is, the number of key-value pairs we can store in a given hash table is fixed (unless we perform some resizing operations).

HASH TABLE

- Instead of directly searching for the key, the hash table ADT introduces a function, called **hash function**, to map each key to the position of its corresponding record in the hash table.
- Computing the hash function should only take $O(1)$ time, given a fix-sized key size. In this case the entire retrieval process will take $O(1)$ time.



HASH TABLE

- There are three fundamental challenges in implementing a hash table:
 - **The choice of hash function:** Ideally, we should choose hash functions that can distribute the records throughout the array as evenly as possible. If two records are assigned to the same position of the array (called hash collision, or simply collision), extra efforts are needed to retrieve the correct information (discussed later).
 - **The handling of collision:** Practically, there is no perfect hash function that can completely avoid collision. We expect ways to handle collision with high efficiency and accuracy.
 - **The handling of array overflow:** The hash table is implemented using a fix-sized array. When more records are added and causes the overflow, we will need to resize and rehash all elements.

HASH TABLE: HASH FUNCTION

- The main objectives of designing a hash function:
 - evenly distribute the records
 - easy to compute (associate with a smaller constant even in $O(1)$ time)
- Practically, the hash function expects the key as a numerical value
 - if it is not, we can convert it into a numerical value (e.g., using ASCII to convert string, or using binary to represent other more complicated objects)

HASH TABLE: HASH FUNCTION

- Simple hash function: taking the modular of the array size

```
1  int hash( const string & key, int tableSize )
2  {
3      int hashVal = 0;
4
5      for( char ch : key )
6          hashVal += ch;
7
8      return hashVal % tableSize;
9  }
```

- Caveats:
 - cannot fully distribute all records if the number of possible records is smaller than the table size, yet will cause collision if we use a small table size
 - try setting the table size to a prime number to avoid unexpected collision

HASH TABLE: HASH FUNCTION

- Solution: extending the range we can map the key

```
1  /**
2   * A hash routine for string objects.
3   */
4  unsigned int hash( const string & key, int tableSize )
5  {
6      unsigned int hashVal = 0;
7
8      for( char ch : key )
9          hashVal = 37 * hashVal + ch;
10
11     return hashVal % tableSize;
12 }
```

HASH TABLE: HASH FUNCTION

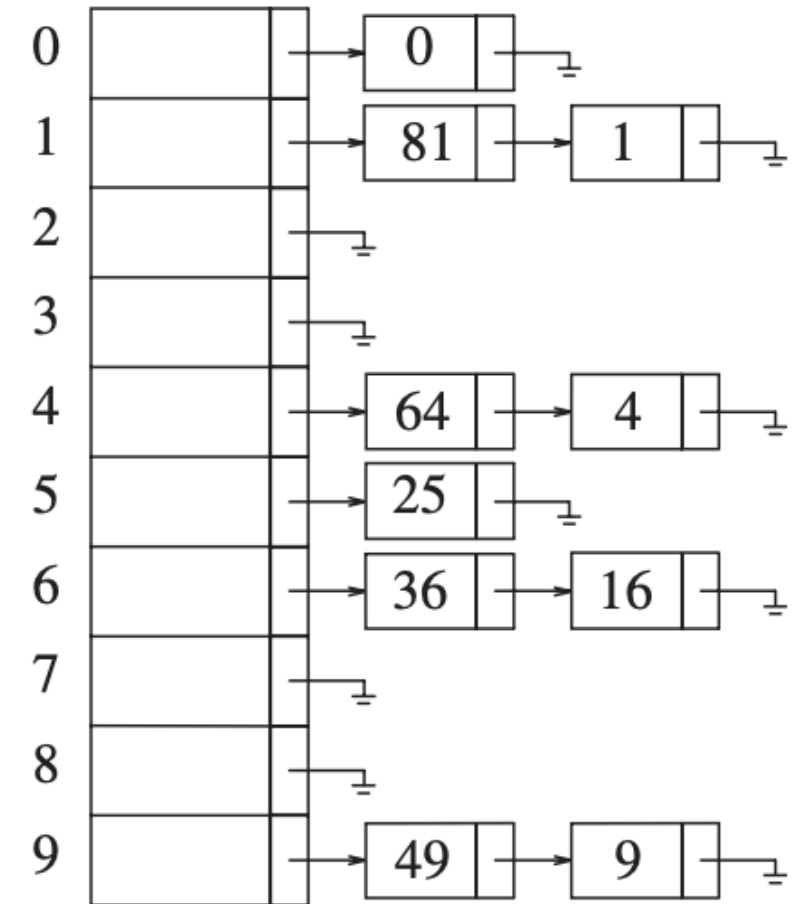
- Designing a good hash function is an art.
- A good hash function considers the features of the keys (such as the distribution of the values to avoid collision).

HASH TABLE: HANDLING COLLISION

- Collision happens when we have two different records that are hashed into the same position of the array
- The solution is intuitive:
 - either we append the record into the current position (separate chaining)
 - or we hash the record to another position that is empty (hashing without chaining)
- Either choice would increase the worst-case time complexity for inserting a record to and retrieving a record from the hash table.

HASH TABLE: HANDLING COLLISION (SEPARATE CHAINING)

- Each position of the array is extended from a simple data holder to the head of a linked list
- When collision happens, we append the new record at the end of the linked list
- When we retrieve a record, we go through each element of the linked list ($O(n)$ time worst-case scenario)



HASH TABLE: HANDLING COLLISION (SEPARATE CHAINING)

- insertion implementation

```
bool insert( const HashedObj & x )
{
    auto & whichList = theLists[ myhash( x ) ]; // computes the position using the hash function "myhash"
    if( find( begin( whichList ), end( whichList ), x ) != end( whichList ) )
        return false; // error checking
    whichList.push_back( x ); // insert the element to the end of the linked list

    // Rehash; see Section 5.5
    if( ++currentSize > theLists.size( ) )
        rehash( );

    return true;
}
```

HASH TABLE: HANDLING COLLISION (SEPARATE CHAINING)

- remove operation (we can implement the look-up operation in a similar way)

```
bool remove( const HashedObj & x )
{
    auto & whichList = theLists[ myhash( x ) ]; // computes the position using the hash function "myhash"
    auto itr = find( begin( whichList ), end( whichList ), x );
                                                    // goes through the linked list to search for the key

    if( itr == end( whichList ) )
        return false;

    whichList.erase( itr );
    --currentSize;
    return true;
}
```

// when implementing the look-up operation, we can simply return the value

HASH TABLE: HANDLING COLLISION (SEPARATE CHAINING)

- Disadvantages:
 - it needs an additional implementation of the linked list ADT
 - the linked list implementation requires more space than the array implementation
 - for each data, linked list requires two pointers to link the former and later data blocks
- Can we avoid the linked list?
 - we can use an alternative hash function to find another empty slot when collision happens

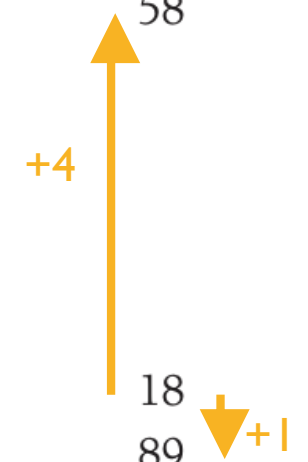
HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- The alternative hash function is based on the primary hash function, but added with another function that depends on the number of trials.
- The quadratic probing hash function
 - $h_i(x) = (\text{hash}(x) + f(i)) \pmod{\text{TableSize}}$
 - $f(i) = i^2$

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- imagine we are inserting 89, 18, 49, 58, 69 into an empty hash table

| | Empty Table | 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 |



HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- Do we always find an empty slot?
- When we have a large hash table, and its size is prime, then we can expect such a nice property.
- **Theorem:** If quadratic probing is used, and the table size is prime, then the new element can always be inserted if the table is at least half empty.

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- **Proof** (using prove by contradiction):
 - Assume that the table size is a prime, and it is larger than 3
 - Consider two different hash trails i and j , where i and j are integers. We have $i \neq j$, and $0 \leq i, j \leq \text{floor}(\text{TableSize}/2)$ (this is because $i^2 > \text{TableSize}$ and $i > \text{TableSize} \geq 3$)
 - Now, if the i th and the j th probing are pointing to the same position (which means we are not able to find an empty slot), we have:
 - $(\text{hash}(x) + i^2)(\text{mod } \text{TableSize}) = (\text{hash}(x) + j^2)(\text{mod } \text{TableSize})$
 - $i^2(\text{mod } \text{TableSize}) = j^2(\text{mod } \text{TableSize})$
 - $(i^2 - j^2)(\text{mod } \text{TableSize}) = 0$
 - $(i + j)(i - j)(\text{mod } \text{TableSize}) = 0$

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- **Proof cont.**

- First, $(i + j)(\text{mod } TableSize)$ cannot be 0 because $i, j \leq \text{floor}(TableSize/2)$
- Second, $(i - j)(\text{mod } TableSize)$ cannot be 0 because $i, j \leq \text{floor}(TableSize/2)$ and $i \neq j$
- Therefore, it is impossible for two different hash probes to point to the same location, if all premises are met

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- implementation: the findPos function for finding the next available position

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

    while( array[ currentPos ].info != EMPTY &&
           array[ currentPos ].element != x )
    {
        currentPos += offset; // Compute ith probe
        offset += 2;          // increasing the jump size for each probing failure
        if( currentPos >= array.size( ) ) // computing the modular
            currentPos -= array.size( );
    }

    return currentPos;
}
```

HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- implementation: insertion

```
bool insert( const HashedObj & x )
{
    // Insert x as active
    int currentPos = findPos( x ); // calling the findPos function to locate an empty slot
    if( isActive( currentPos ) )
        return false;

    array[ currentPos ].element = x;
    array[ currentPos ].info = ACTIVE;

    // Rehash; see Section 5.5
    if( ++currentSize > array.size( ) / 2 )
        rehash( );

    return true;
}
```


HASH TABLE: HANDLING COLLISION (QUADRATIC PROBING)

- implementation: removal (similarly, look-up)

```
bool remove( const HashedObj & x )
{
    int currentPos = findPos( x );    // calling the findPos function to locate an empty slot
    if( !isActive( currentPos ) )
        return false;

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

HASH TABLE: HANDLING COLLISION

- Other methods share a similar idea, but use different $f(i)$ functions
 - quadratic probing uses $f(i) = i^2$
 - linear probing uses $f(i) = i$
 - double hashing uses $f(i) = i * hash_2(x)$, where $hash_2(x)$ is a different hash function than the primary hash function

HASH TABLE: HANDLING OVERFLOW

- The final issue remains unsolved is how do we handle hash table overflow
 - we know that the hash table has a fixed size, while the number of records we insert is unknown and can be much larger than the pre-set table size
 - even we use separate hashing, not increasing the array size can lead to very long chains, resulting in slower insertion and look-ups
 - furthermore, for quadratic probing, we need to ensure that the table is at least half empty to make sure that we can always find an empty slot
- We should not allocate a huge space for the hash table, because it may use unnecessarily amount of memory
- Solution: we dynamically change the table size, according to the number of records in it

HASH TABLE: HANDLING OVERFLOW

- We can adopt the following strategy:
 - denote the ratio between the number of records in the hash table and the size of the hash table as the **load factor**
 - when we insert, if the load factor is $> 1/2$, we double the size (to a near prime number) of the hash table (called **table doubling**)
 - when we delete, if the load factor is $< 1/8$, we halve the size (to a near prime number) of the hash table (called **table halving**)

HASH TABLE: HANDLING OVERFLOW

- After a resize operation, TableSize has been changed.
- Therefore, it is necessary to revise the hash function (especially if we are taking modular of TableSize).
- This process is called rehashing.

HASH TABLE: HANDLING OVERFLOW (REHASHING)

All records need to be rehashed, which would take $O(n)$ time where n is the number of existing records

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

mod(7)



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

mod(17)

HASH TABLE: HANDLING OVERFLOW (REHASHING)

- Fortunately, not every insertion or deletion will trigger a table resizing operation
 - Note that after a table doubling, the new load factor becomes $1/4$. It means we can delete additional $\text{TableSize}/8$ records or insert additional $\text{TableSize}/4$ records without triggering a resize operation
 - Similarly, after a table halving, the new load factor becomes $1/4$. It means we can delete additional $\text{TableSize}/8$ records or insert additional $\text{TableSize}/4$ records without triggering a resize operation
- On average, we still have $O(1)$ insertion and deletion time complexity with the implementation of the table doubling and table halving mechanism
 - proof will be discussed in EECS 660

THE C++ STL FOR HASH TABLE

- `std::unordered_map` and `std::unordered_set`
 - `std::unordered_map` stores both of the key and value of the records, while `std::unordered_set` only stores the key
 - e.g., `std::unordered_map` can answer questions like “What is the favorite video game of Mary?”, while `std::unordered_set` can only answer the question of “Is Mary one of the students in our class?”
- `std::map` and `std::set`
 - they are similar to the unordered version, but the keys in them are ordered
 - faster when you perform in-order traversal, but slower for each insertion/deletion/look-up
 - not implemented using hash table, but with tree ADTs (will be discussed more later)

SUMMARY

- Hash table allows you to quickly ($O(1)$ time) look-up entries using the keys
- Challenges in designing hash table
 - good hash function that evenly distribute the records
 - handling collisions
 - separate hashing using linked list
 - quadratic probing without linked list
 - handling overflow
 - table doubling and table halving
- C++ STL: `std::unordered_map` and `std::unordered_set`