HASHING

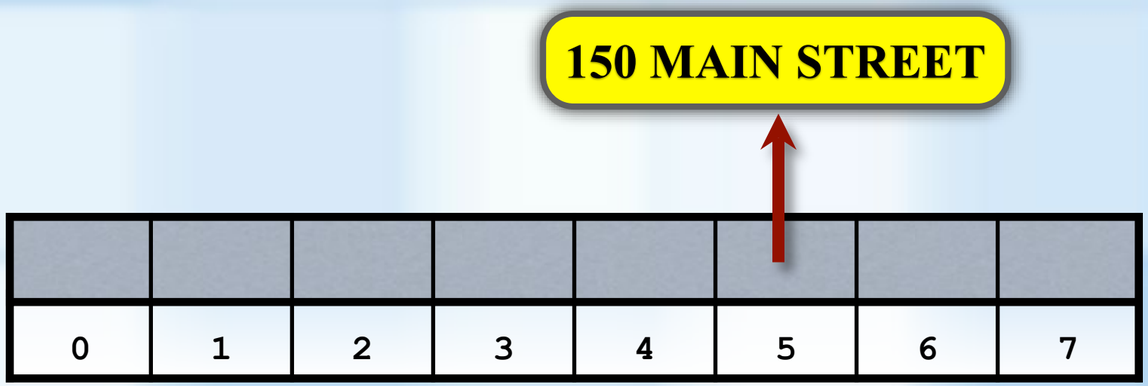
HASHING

- **Can result in O(1) search times**
- **Cannot provide traversals of search keys is sorted order**
- **Determines index of an entry using only search key**
  - Does not search for item
- **Hash Table**
  - Array storing values indexed by a hash function
- **Hash Function**
  - Takes search key and produces index of element in the hash

```
Algorithm add(key, value)
    index = hash(key)
    hashTable[index] = value

Algorithm getValue(key)
    index = hash(key)
    return hashTable[index]
```

```
hash(555-1234) => 5
```



Hash Functions

- **Perfect Hash Function**
  - Maps each search key into a different integer that is suitable as an index to the hash table
- **Sparse Hash Table**
  - Only a few elements in hash table are in use
- **Typical Hash Function**
  - Convert the search key into hash code
  - Compress hash code into range of hash table
- **Collision**
  - Hash function returns the same hash code for two different search keys

- *Our small town has 700 residents*
- *All telephone numbers are part of the 555 exchange*
- *Simple hash for our telephone numbers:*
  - *return the 4-digit extension*

```
hash(555-1234) => 1234
```

- *Requires a hash table of 10,000 elements*
  - *Hash value range : 0000 - 9999*
- *Table will be sparse*
  - *9,300 elements will not be used*

```
Algorithm getHashIndex(phoneNumber)
// Returns an index to an array
// of tableSize locations.
i = last four digits of phoneNumber return
i % tableSize
```

```
getHashIndex(555-0721) => 21
getHashIndex(555-2121) => 21
```

Pearson

Hash Functions

- **A good hash function**
  - Minimize collisions
  - Be fast to compute

To reduce the chance of a collision, choose a hash function that distributes entries uniformly throughout the hash table.

HashedDictionary.h

```
// Create a hash function type called hashFunction that hashes "KeyType"
// First we create an an unordered_map object for our KeyType and ItemType
std::unordered_map<KeyType, ItemType> mapper;

// Then we call the hash_function method to return the hash function
// for the KeyType and assign it to 'hashFunction'
typename std::unordered_map<KeyType, ItemType>::hasher hashFunction = mapper.hash_function();
```

HashedDictionary.cpp

```
int HashedDictionary<KeyType, ItemType>::getHashIndex(const KeyType& itemKey) const
{
   // static_cast needed since the hasFunction returns an unsigned long
   return static_cast<int>(hashFunction(itemKey) % hashTableSize);
} // end getHashIndex
```

Hash Functions

- **A good hash function**
  - Minimize collisions
  - Be fast to compute
- **Hash Codes for Strings**
  - Convert characters to Unicode integers
  - Multiply characters based on position

```
Algorithm hashCode(someString)
    int hash = 0;
    int n = someString.length();
    for (int i = 0; i < n; i++)
          hash = g * hash + someString[i];
    // The Java String class uses g = 31
```

  - `byte`, `short`, or `char`
- Manipulate internal representations
  - Perform folding
    - Divide key into parts
    - Recombine and process parts

Hash Functions

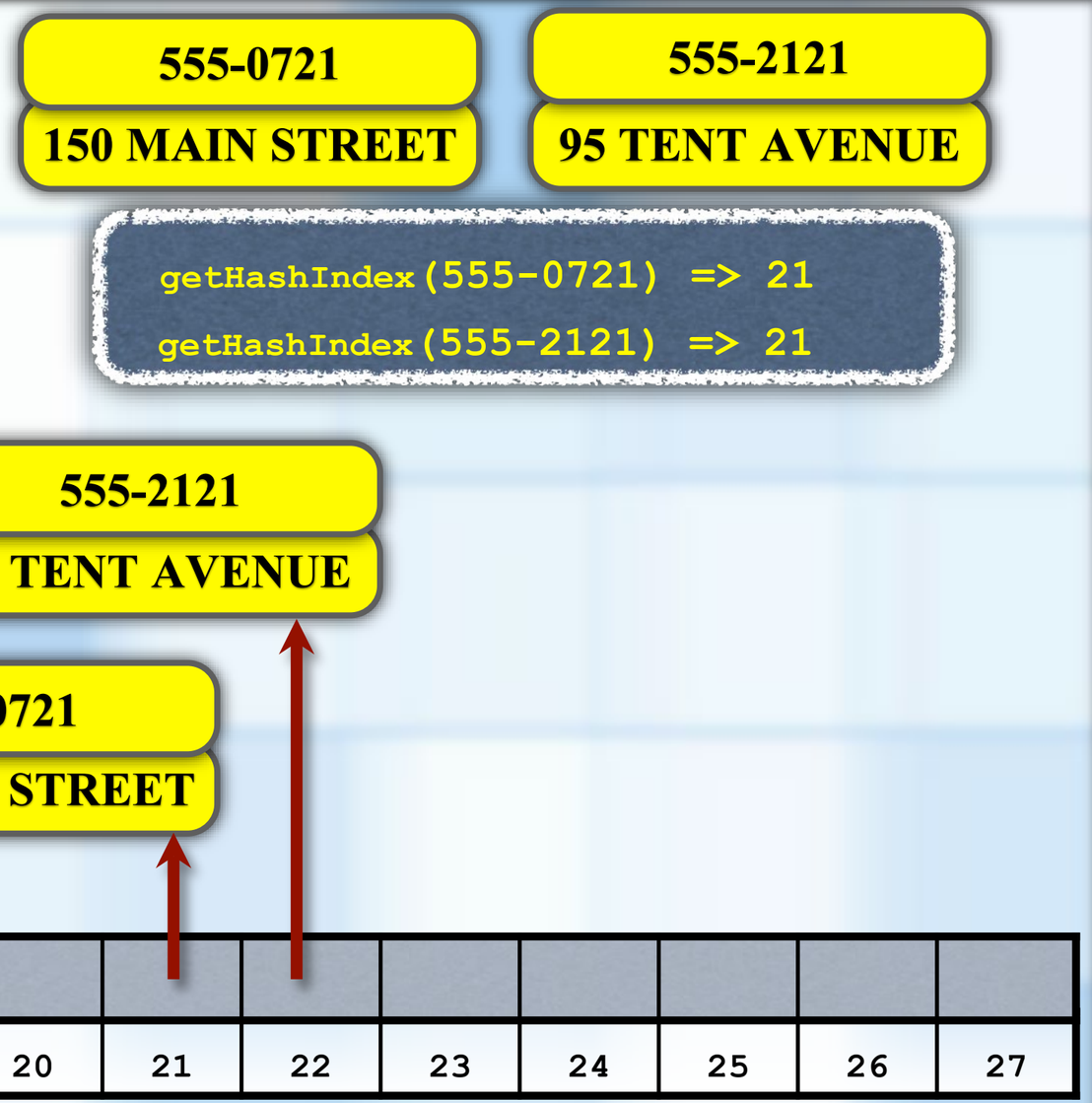
- **Compressing a Hash Code**
  - Use modulo operator `%`
    - `(hashCode) % (hash table size)`
    - `c % n`
  - `n` should be odd
  - Ideally, `n` should be prime

```
int getHashIndex(KeyType searchKey)
{
   int hashIndex = getHashIndex(searchKey) % hashTableSize;
   if (hashIndex < 0)
       hashIndex = hashIndex + hashTableSize;
   return hashIndex;
} // end getHashIndex
```

HASHING

Resolving Collisions

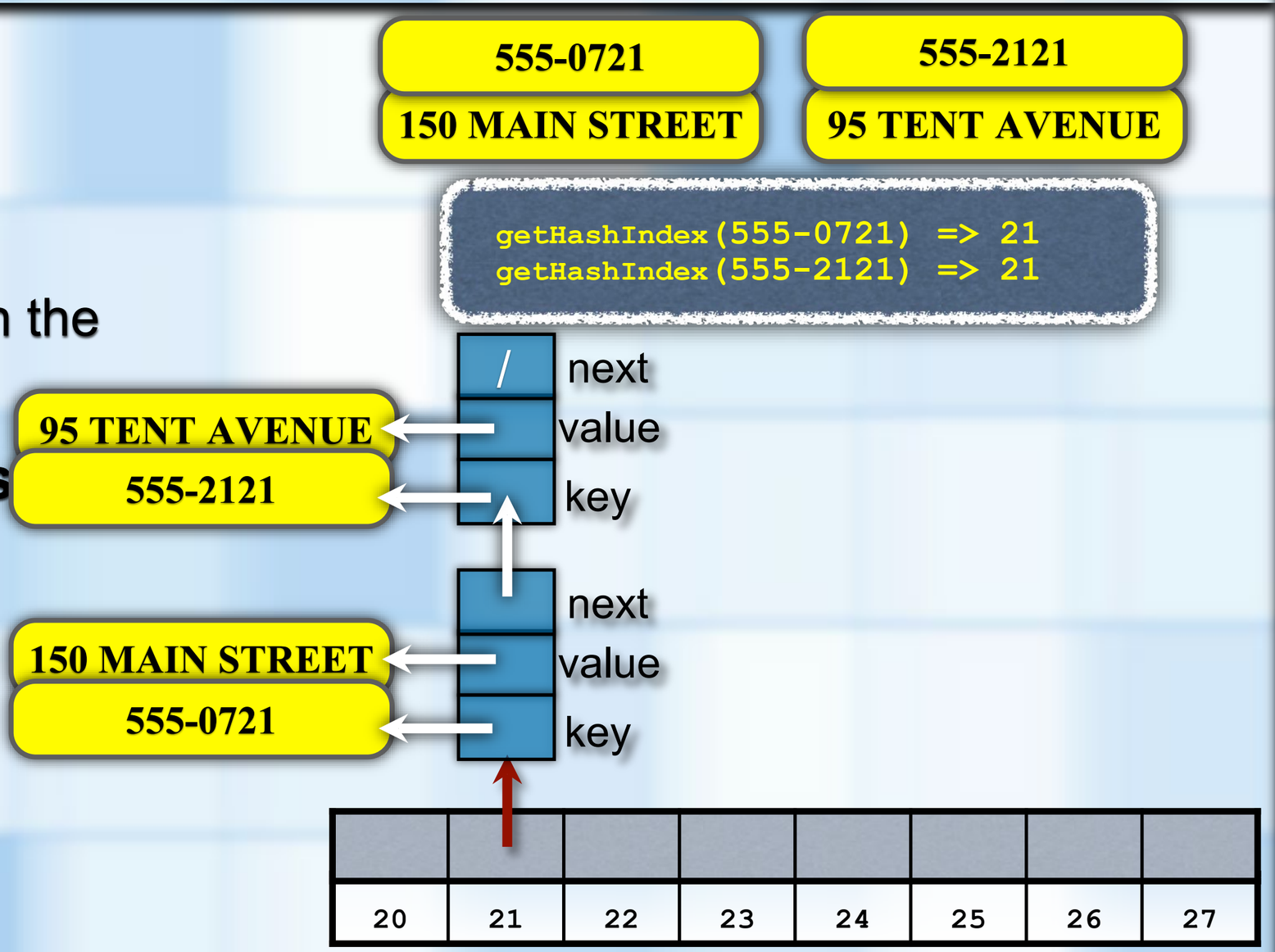
- **Change structure of has table**
  - Each location can represent more than a single value
- **Use another location in the hash table**
  - ***Open Addressing***
  - ***Probing***
    - Locating an open location in hash table
  - ***Linear Probing***
    - Sequentially searching for an open element



Resolving Collisions

- **Separate Chaining**
  - Each hash table element is a ***bucket***
  - Place matching hash indices in the same bucket

Search Keys	Where to insert
Unsorted duplicate	Beginning of chain
Unsorted distinct	End of chain
Sorted distinct	In sorted order



Efficiency of Hashing

- **Assuming distinct search keys**
  - Searching for keys is used by
    - `add` method ensures duplicates do not occur
    - `remove` method to find the entry to remove
    - `getValue` method to find the value to return

The successful retrieval of an entry searches the same chain or probe sequence that was searched when the entry was first added to the hash table.

Thus, the cost of a successful retrieval of an entry is the same as the cost of inserting that entry.

- **General observations on dictionaries that use hashing**
  - Successful retrieval or removal has the same efficiency as a successful search
  - An unsuccessful retrieval or removal has the same efficiency as an unsuccessful search
  - A successful addition has the same efficiency as an unsuccessful search
  - An unsuccessful addition has the same efficiency as a successful search

Load Factor

- **Perfect Hash Function**
  - No collisions
  - Dictionary operations are $O(1)$
- **Resolving collisions takes time**
  - As hash table fills, there are more collisions and operations take more time
- **Load Factor**

$$\lambda = \frac{\textit{Number of entries in the dictionary}}{\textit{Number of locations in the hash table}}$$

- **About the Load Factor λ**
  - λ is a measure of the cost of collision resolution
  - λ is never negative
    - Minimum of zero - empty hash table
  - for open addressing, λ does not exceed 1
  - for separate chaining, λ has no maximum value
  - restricting the size of λ improves the performance of hashing

Load Factor

- **Cost of Open Addressing**
  - Search probe sequence
  - Number of comparisons necessary to locate a search key in the hash table
- **Linear Probing**
  - Collisions increase as the hash table fills

The performance of hashing with linear probing degrades significantly as the load factor λ increases.

To maintain reasonable efficiency, the hash table should be less than half full.

That is, keep $\lambda < 0.5$.

Average Number of Comparisons To Search A Probe Sequence

$$\frac{1}{2}\left\{1 + \frac{1}{(1-\lambda)^2}\right\}$$ for an unsuccessful search and

$$\frac{1}{2}\left\{1 + \frac{1}{(1-\lambda)}\right\}$$ for a successful search

λ	Unsuccessful Search	Successful Search
0.1	1.1	1.1
0.3	1.5	1.2
0.5	2.5	1.5
0.7	6.1	2.2
0.9	50.5	5.5

Load Factor

$$\lambda = \frac{Number\ of\ entries\ in\ the\ dicionary}{Number\ of\ chains}$$

- **Cost of Separate Chaining**
  - Number of chains is size of hash table
  - Operations need to search a chain with average length λ
  - Unsuccessful Searches
    - Empty Chain → O(1)
    - Non-empty Chain → λ
      - must search entire chain
  - Successful Searches
    - Always a non-empty chain
    - Must look at 1 + λ/2 entries on average

The average performance of hashing with separate chaining does not degrade significantly as the load factor λ increases.

To maintain reasonable efficiency, you should keep λ < 1.

Average Number of Comparisons To Search A Chain

λ	for an unsuccessful search
1 + λ/2	for a successful search

λ	Unsuccessful Search	Successful Search
0.1	0.1	1.1
0.3	0.3	1.2
0.5	0.5	1.3
0.7	0.7	1.4
0.9	0.9	1.5
1.1	1.1	1.6
1.3	1.3	1.7
1.5	1.5	1.8
1.7	1.7	1.9
1.9	1.9	2.0
2.0	2.0	2.0

EFFICIENCY OF HASHING

- **Comparing Schemes for Collision Resolution**
  - Separate Chaining is more time efficient
    - but requires more memory
  - Hash table can be smaller for separate chaining
  - Open addressing is more likely to lead to ***rehashing***
    - Alternative is to use a large hash table

REHASHING

When the load factor λ becomes too large, and the hash table must be resized to maintain reasonable search times.

To compute the table's new size, first double its present size and then increase the result to the next prime number.

Use the method `add` to add the current entries in the dictionary to the new hash table since

new hash codes must be generated for each search key.