SORTING

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

21: HashTables

Alexandra Papoutsaki
she/her/hers

Tom Yeh
he/him/his

BBICS Mentor (Black and Brown in Computer Science)

About Me

Naomi (Nay) Amuzie, she/her

Hometown: Houston, TX

Senior at Pomona majoring in CS and minoring in Africana Studies

Session

3-5pm on Fridays

Safe space to work among other Black, Hispanic, and Indigenous people of color

Work on weekly assignment

Work on interview practice problems based on weekly lectures

Pomona CS watches Dune

LAEMMLE THEATER

Tuesday, 11/9 6:45 PM
450 W 2nd St,
Claremont, CA 91711

PRE-REGISTER AND ARRIVE PROMPTLY FOR YOUR TICKET!!!

We have 50 tickets available; more are welcome but will have to pay themselves. Fill out this form to reserve a ticket (first preference given to CS students):
https://forms.gle/P27QNejocuts3gdz7

Or use QR code on right, Clickable link on the CS slack

Assignment 6 Extension

- You can take a 1 week extension on Assignment 6
  - Due 11/16
  - Note: Assignment 7 will be due at the same time

Lecture 21: Hash tables

- Hash functions
- Separate chaining
- Open addressing

Some slides adopted from Algorithms 4th Edition or COS226

Summary for symbol table operations

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
Sequential search (unordered array)	n	n	n	$n/2$	n	$n/2$
Binary search (ordered array)	$\log n$	n	n	$\log n$	$n/2$	$n/2$
BST	n	n	n	$1.39 \log n$	$1.39 \log n$	?
2-3 search tree	$c \log n$	$c \log n$	$c \log n$	$c \log n$	$c \log n$	$c \log n$
Red-black BSTs	$2 \log n$	$2 \log n$	$2 \log n$	$1 \log n$	$1 \log n$	$1 \log n$

Basic plan for implementing dictionaries using hashing

- Goal: Build a key-indexed array (table or hash table or hash map) to model dictionaries (or symbol tables) for efficient ($O(1)$ search).
- Hash function: Method for computing array index (hash value) from key.
  - hash("Texas") = 2 ???
  - hash("California") = 2
- Issues:
  - Computing the hash function.
  - Method for checking whether two keys are equal.
  - How to handle collisions when two keys hash to same index.
- Trade off between time and space

Computing hash function

- Ideal scenario: Take any key and uniformly "scramble" it to produce a symbol table/dictionary index.
- Requirements:
  - Consistent - equal keys must produce the same hash value.
  - Efficient - quick computation of hash value.
  - Uniform distribution - every index is equally likely for each key.
- Although thoroughly researched, still problematic in practical applications.
- Examples: Dictionary where keys are social security numbers.
  - Bad: if we choose the first three digits (geographical region and time).
  - Better: if we choose the last three digits.
  - Best: use all data.
- Practical challenge: Need different approach for each key type.

Hashing in Java

- All Java classes inherit a method `hashCode()`, which returns an integer.
- Requirement: If `x.equals(y)` then it should be `x.hashCode()==y.hashCode()`.
- Ideally: If `!x.equals(y)` then it should be `x.hashCode()!=y.hashCode()`.
- Default implementation: Memory address of x.
  - Need to override both `equals()` and `hashCode()` for custom types.
  - Already done for us for standard data types: `Integer`, `Double`, etc.

Equality test in Java

- Requirement: For any objects x, y, and z.
  - Reflexive: `x.equals(x)` is `true`.
  - Symmetric: `x.equals(y)` iff `y.equals(x)`.
  - Transitive: if `x.equals(y)` and `y.equals(z)` then `x.equals(z)`.
  - Non-null: if `x.equals(null)` is `false`.
- If you don't override it, the default implementation checks whether x and y refer to the same object in memory.

Java implementations of `equals()` for user-defined types

```
public class Date {
      private int month;
      private int day;
      private int year;
      …
      public boolean equals(Object y) {
          if (y == this) return true;    // same memory location
          if (y == null) return false;   // compare with null
          if (y.getClass() != this.getClass()) return false;
          Date that = (Date) y;          // same object type
          return (this.day == that.day &&
                  this.month == that.month &&
                  this.year == that.year);  // compare 3 ints
      }
}
```

General equality test recipe in Java

- Optimization for reference equality.
  - `if (y == this) return true;`
- Check against null.
  - `if (y == null) return false;`
- Check that two objects are of the same type.
  - `if (y.getClass() != this.getClass()) return false;`
- Cast them.
  - `Date that = (Date) y;`
- Compare each significant field.
  - `return (this.day == that.day && this.month == that.month && this.year == that.year);`
  - If a field is a primitive type, use `==`.
  - If a field is an object, use `equals()`.
  - If field is an array of primitives, use `Arrays.equals()`.
  - If field is an array of objects, use `Arrays.deepEquals()`.

Java implementations of hashCode()

```
public final class Integer {
      private final int value;
      …
      public int hashCode() {
            return (value);       // just return the value
      }
}
```

```
public final class Boolean {
      private final boolean value;
      …
      public int hashCode() {
            if(value)    return 1231;   // return 2 values (true/false)
            else return 1237;
      }
}
```

Java implementations of hashCode() for user-defined types

```
public class Date {
        private int month;
        private int day;
        private int year;
        …
        public int hashCode() {
            int hash = 1;
            hash = 31*hash + ((Integer) month).hashCode();
            hash = 31*hash + ((Integer) day).hashCode();
            hash = 31*hash + ((Integer) year).hashCode();
            return hash;
            //could be also written as
            //return Objects.hash(month, day, year);
        }
}
```

31x+y rule

General hash code recipe in Java

- Combine each significant field using the 31x+y rule.
- Shortcut 1: use `Objects.hash()` for all fields (except arrays).
- Shortcut 2: use `Arrays.hashCode()` for primitive arrays.
- Shortcut 3: use `Arrays.deepHashCode()` for object arrays.

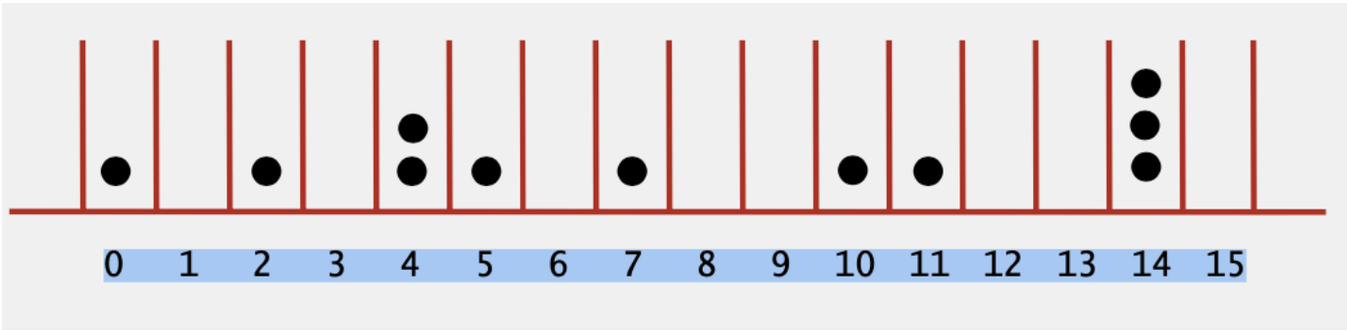
Modular hashing

- Hash code: a 32-bit int between -2^{31} and $2^{31} - 1$
- Hash function: an int between 0 and $m - 1$, where m is the hash table size (typically a prime number or power of 2).
- The class that implements the dictionary of size m should implement a hash function. Examples:
  -
```
private int hash (Key key){
    return key.hashCode() % m;
}
```
    - Bug! Might map to negative number.

-
```
private int hash (Key key){
    return Math.abs(key.hashCode()) % m;
}
```
  - Very unlikely bug. For a hash code of -2^{31}, `Math.abs` will return a negative number!
  - Largest positive number representable with 32 bits is $2^{31} - 1$, abs(-2^{31}) = -2^{31}

-
```
private int hash (Key key){
    return (key.hashCode() & 0x7fffffff) % m;
}
```
  - Correct. Bitwise AND with 0 followed by 31 1s gives us the positive components of the integer.
  - You will learn bit-wise operators in CS181OR

Uniform hashing assumption



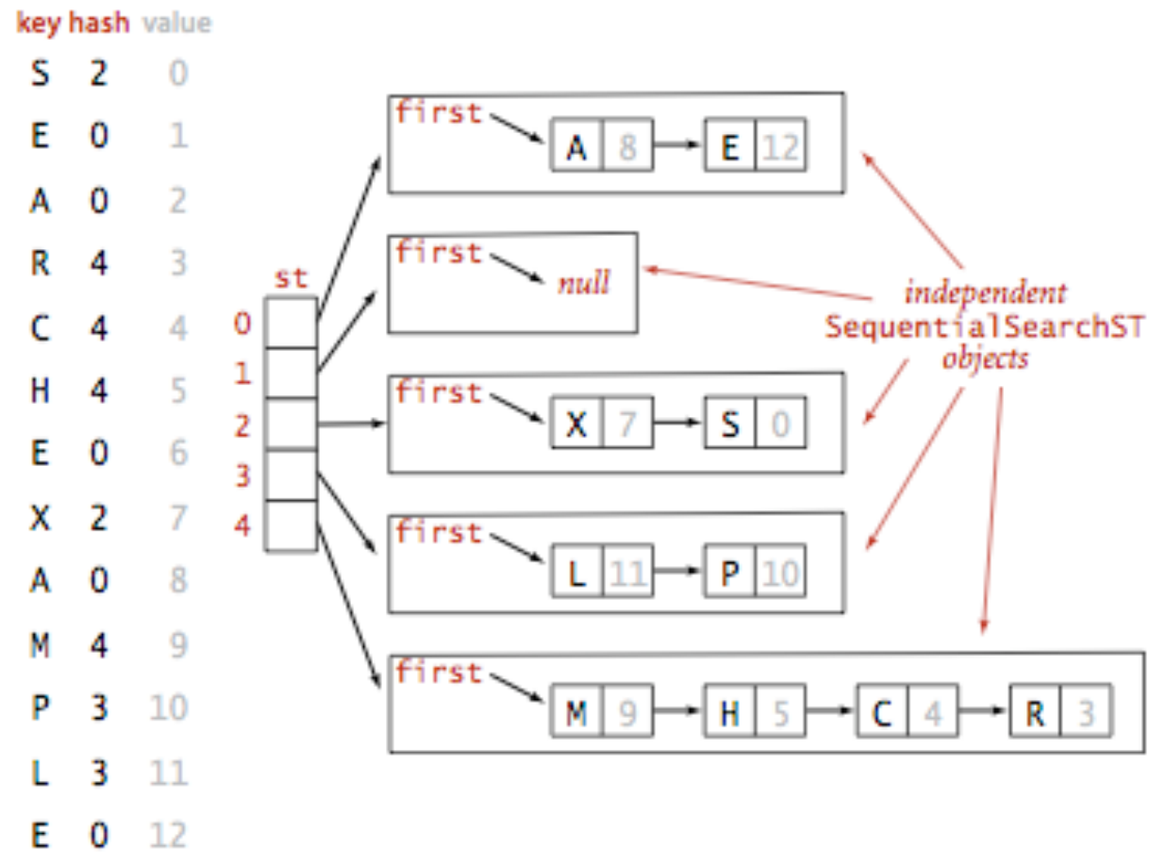
- Uniform hashing assumption: Each key is equally likely to hash to an integer between 0 and $m - 1$.
- Mathematical model: balls & bins. Toss n balls uniformly at random into m bins.
- Bad news: Expect two balls in the same bin after ~$\sqrt{(\pi m/2)}$ tosses.
  - Birthday problem: In a random group of 23 or more people, more likely than not that two people will share the same birthday.
- Good news: load balancing
  - When $n >> m$, the number of balls in each bin is "likely close" to n/m.

Lecture 26-27: Hash tables

- Hash functions
- Separate chaining
- Open addressing

Separate/External Chaining (Closed Addressing)

- Use an array of $m < n$ distinct lists [H.P. Luhn, IBM 1953].
  - Hash: Map key to integer i between 0 and $m - 1$.
- Insert: Put at front of i-th chain (if not already there).
- Search: Need to only search the i-th chain.

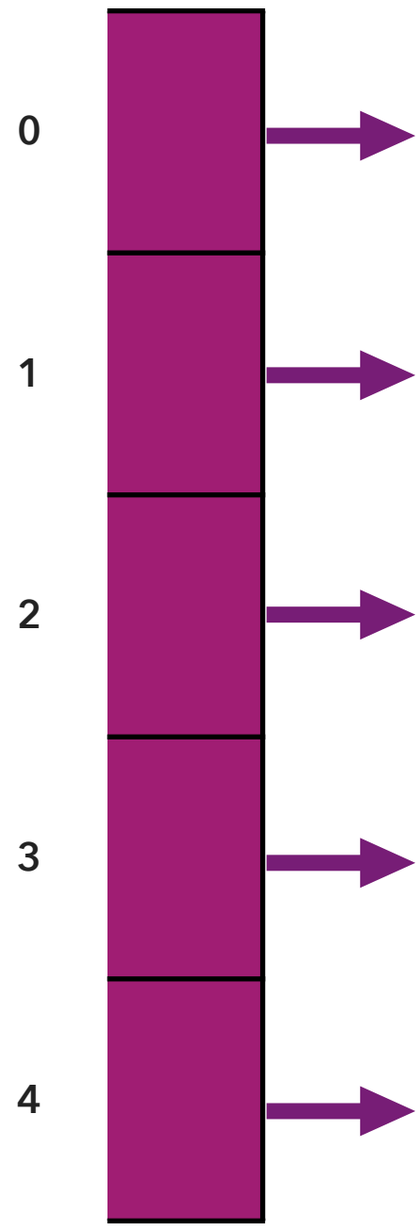


Hashing with separate chaining for standard indexing client

Separate Chaining Example

- Let's assume we implement a dictionary using hashing and separate chaining for collisions.
- The size of the table is 5, that is $m = 5$.
- We will hash the keys S, E, A, R, C, H, E, X, A, M, P, L, E where I will provide you with their hash values.
- Every time we hash a key, we go to the chain attached to that index and traverse the linked list.
  - If we find a node with the same key we want to insert, we just update its corresponding value.
  - If no node contains our key, we insert the key-value pair at the head of the chain.

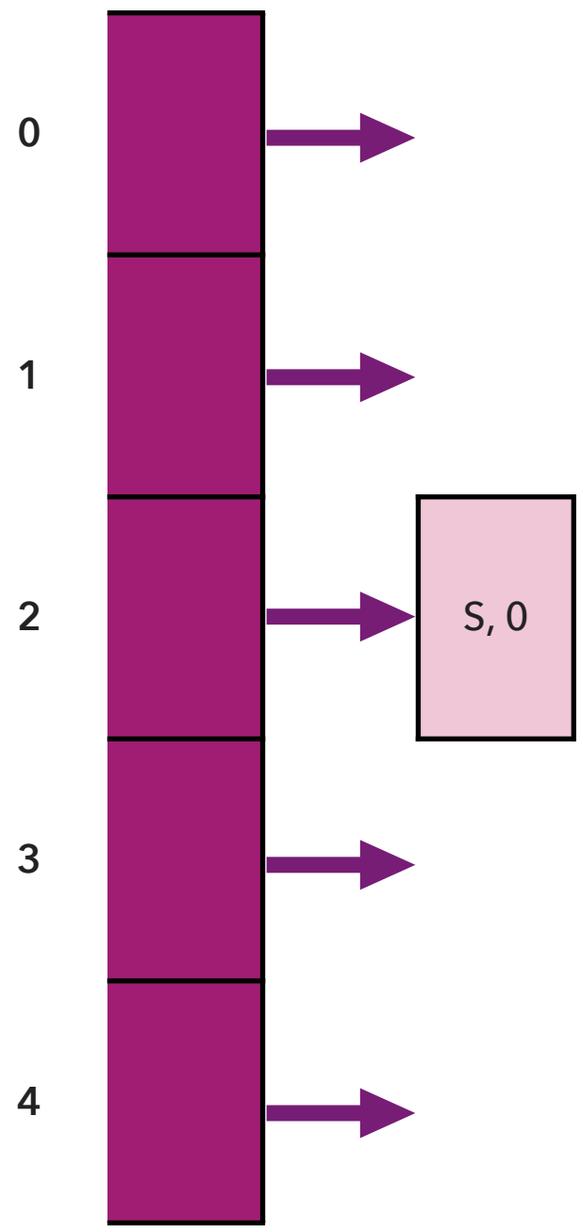
Separate Chaining Example



Next step: Insert (S, 0)

Separate Chaining Example

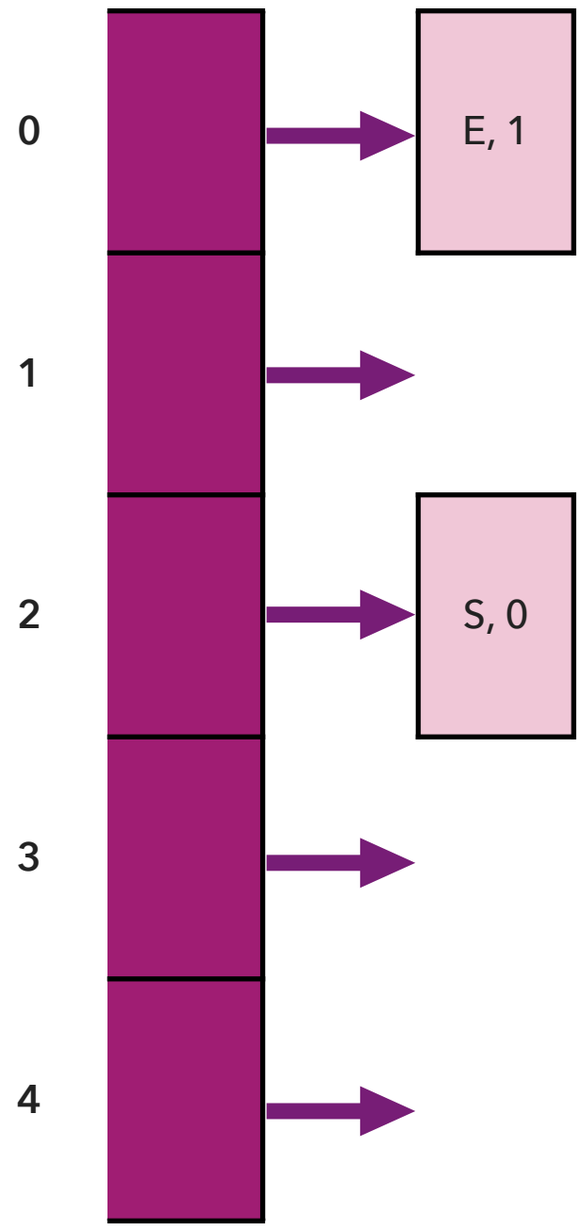
Key	Hash	Value
S	2	0



Next step: Insert (E, 1)

Separate Chaining Example

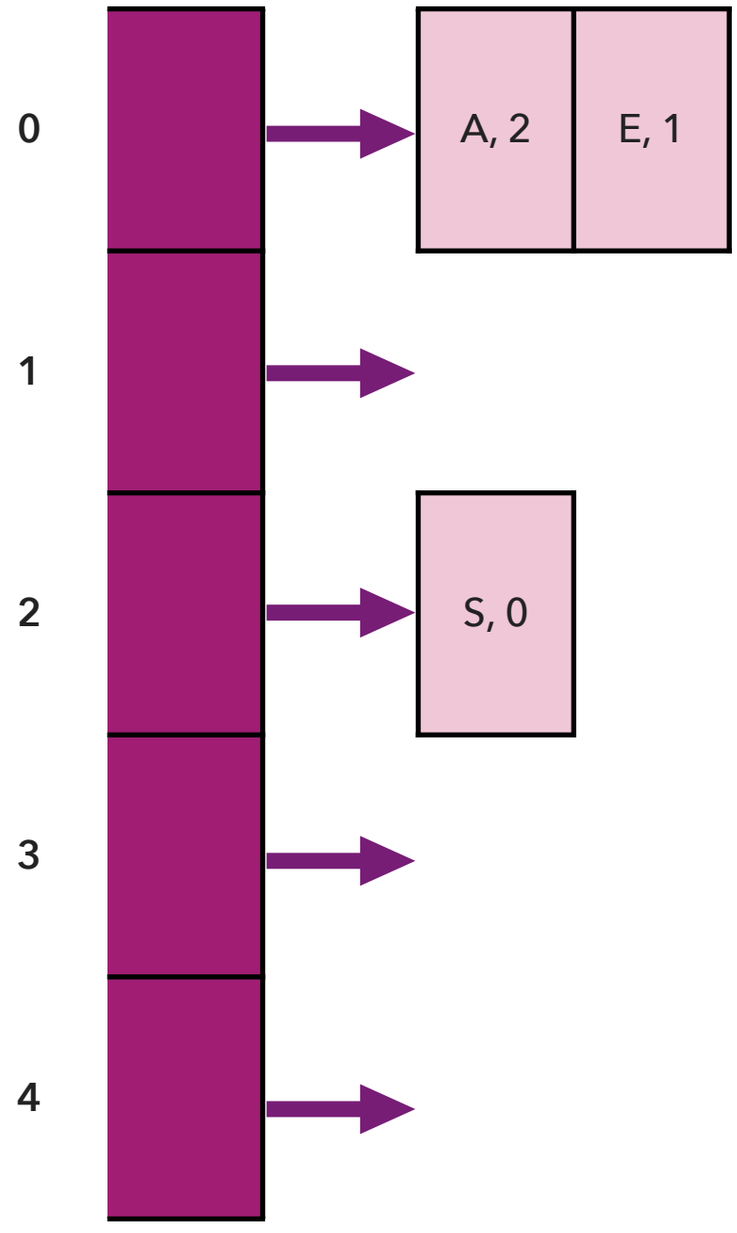
Key	Hash	Value
S	2	0
E	0	1



Next step: Insert (A, 2)

Separate Chaining Example

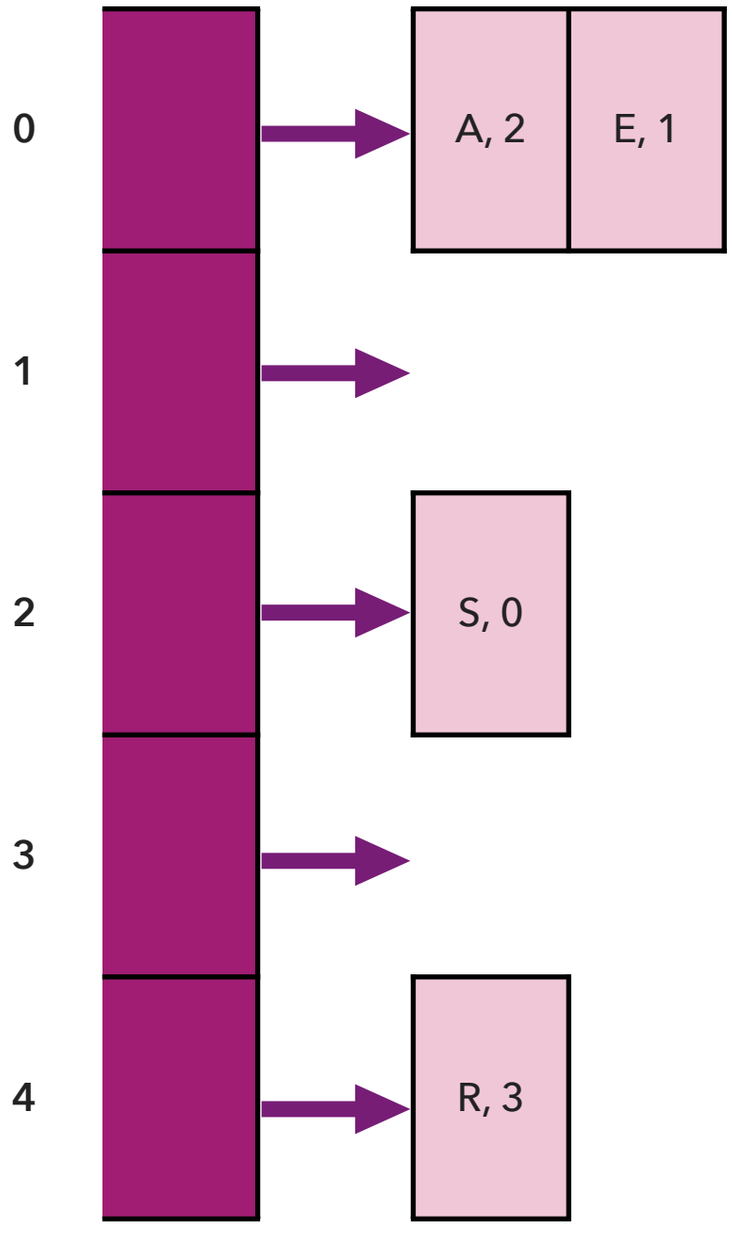
Key	Hash	Value
S	2	0
E	0	1
A	0	2



Next step: Insert (R, 3)

Separate Chaining Example

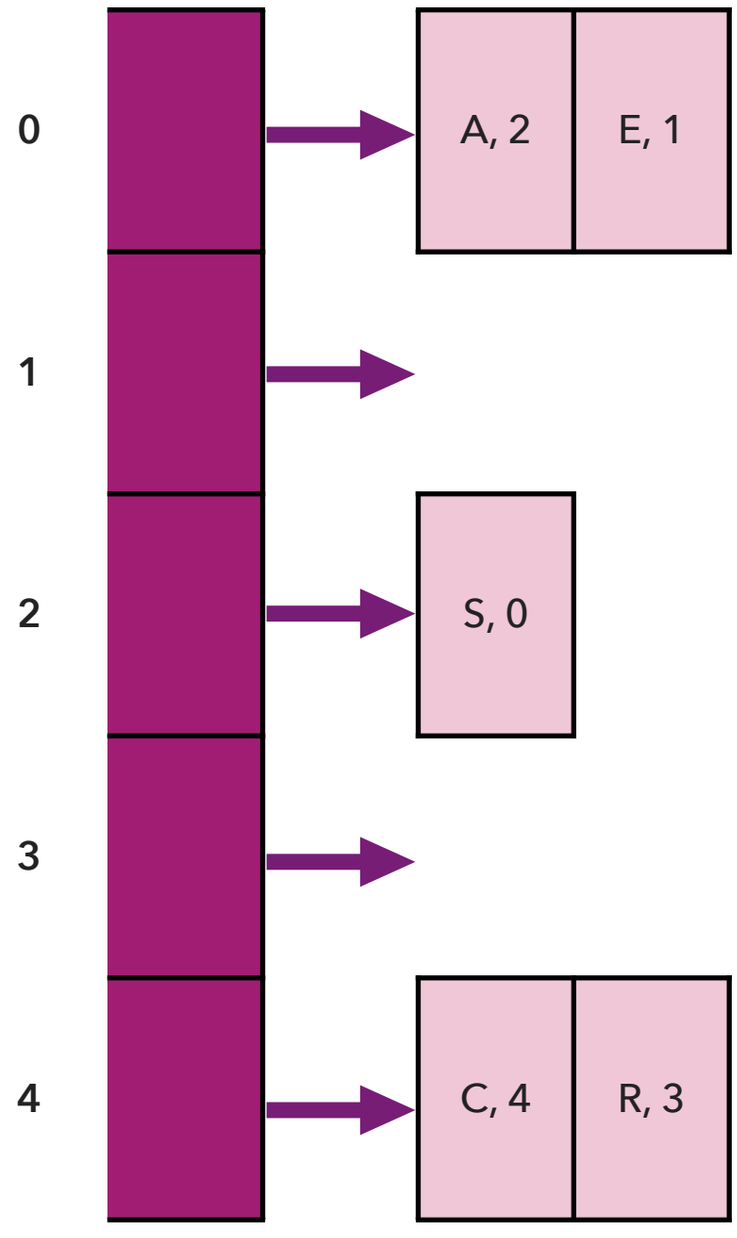
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3



Next step: Insert (C, 4)

Separate Chaining Example

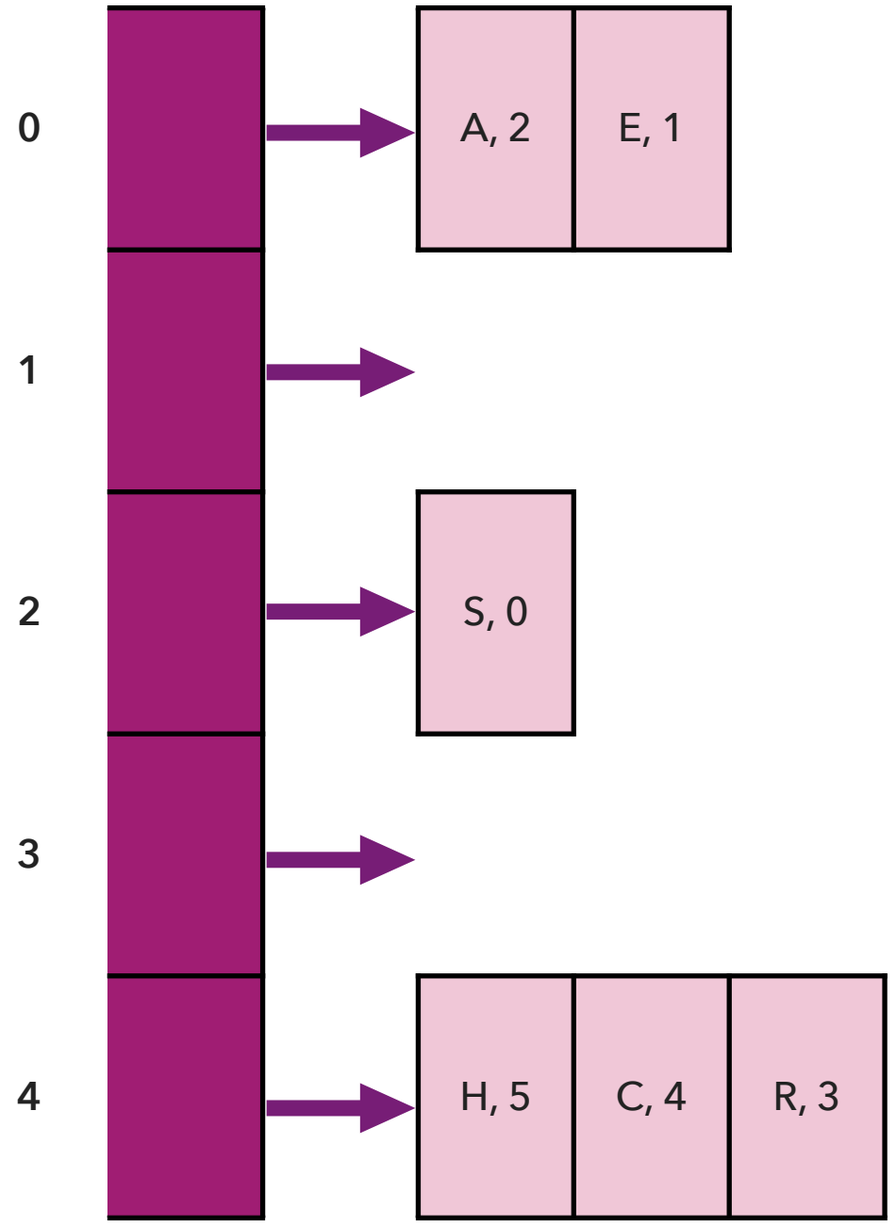
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4



Next step: Insert (H, 5)

Separate Chaining Example

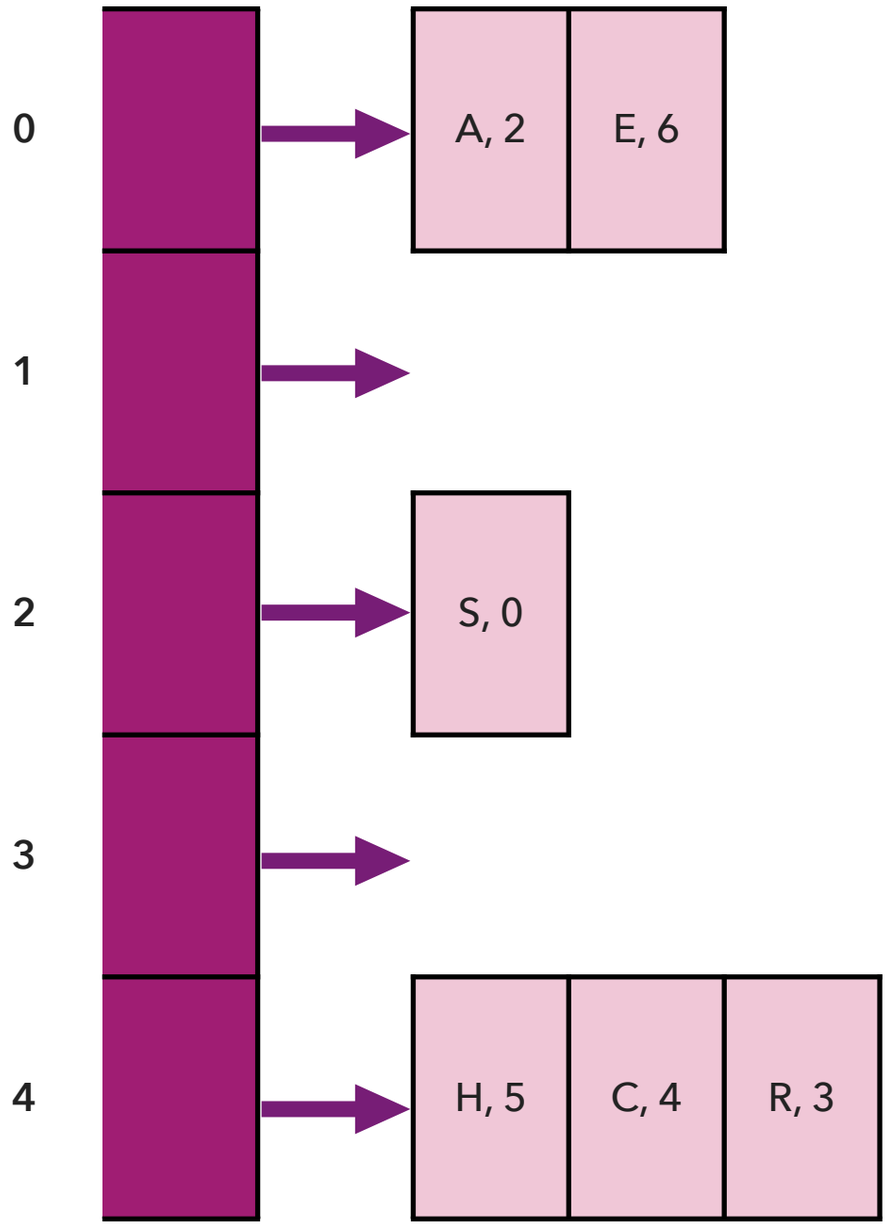
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5



Next step: Insert (E, 6)

Separate Chaining Example

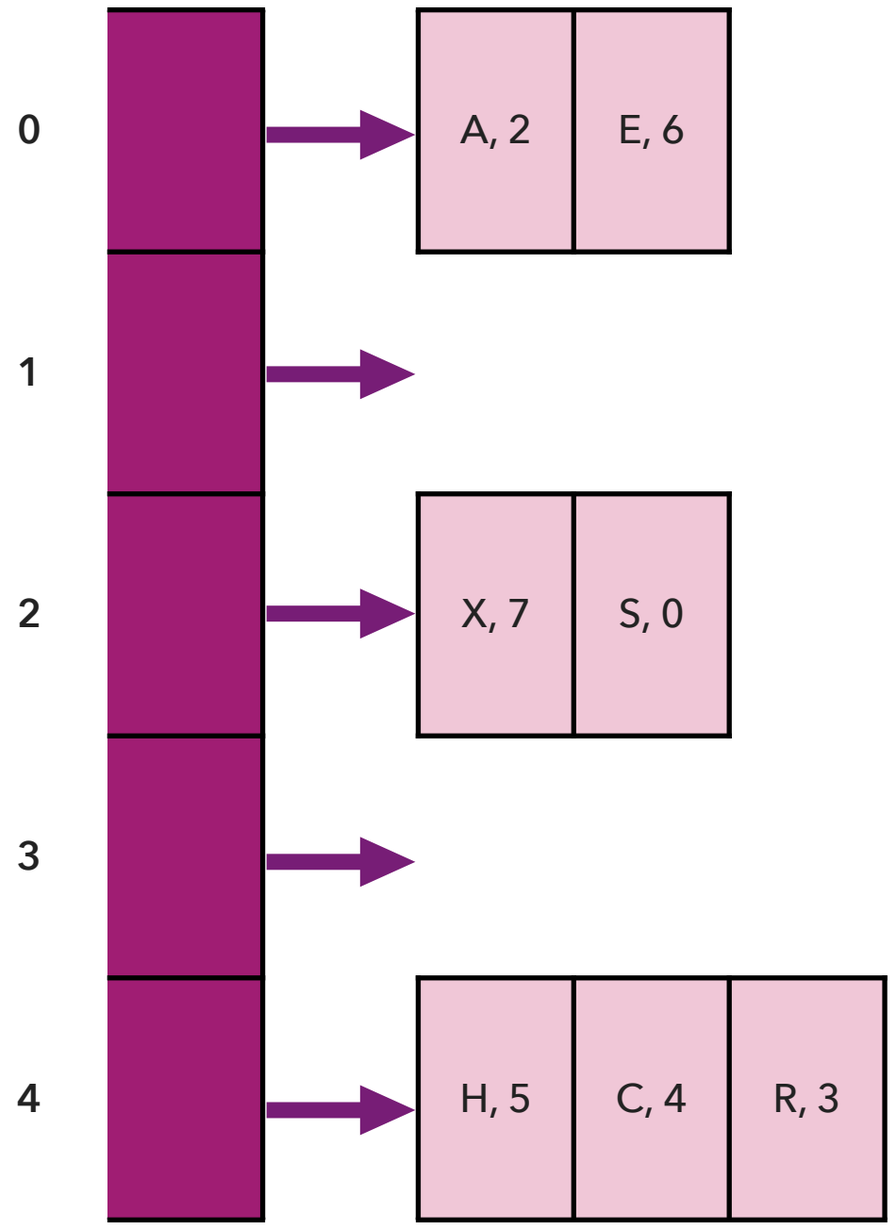
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6



Next step: Insert (X, 7)

Separate Chaining Example

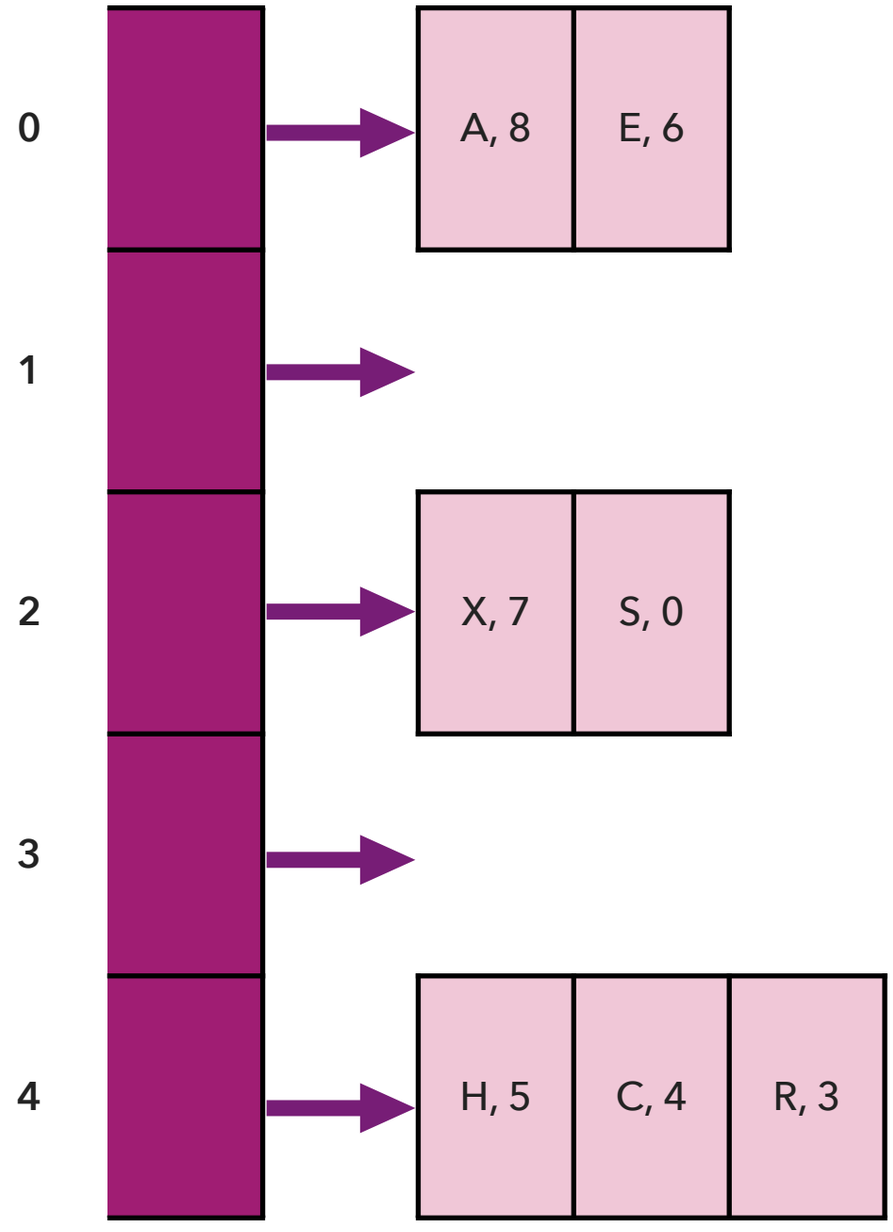
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7



Next step: Insert (A, 8)

Separate Chaining Example

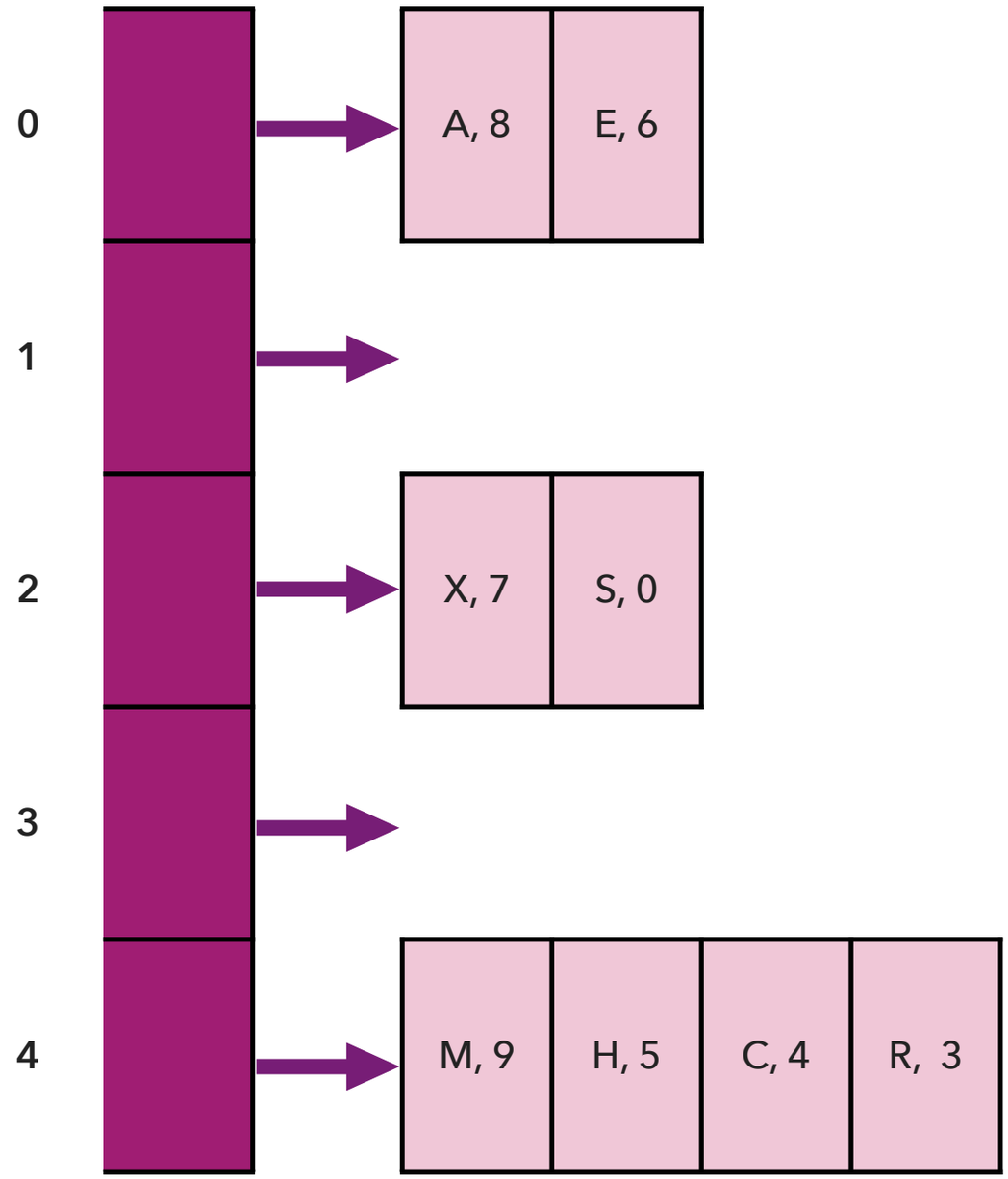
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8



Next step: Insert (M, 9)

Separate Chaining Example

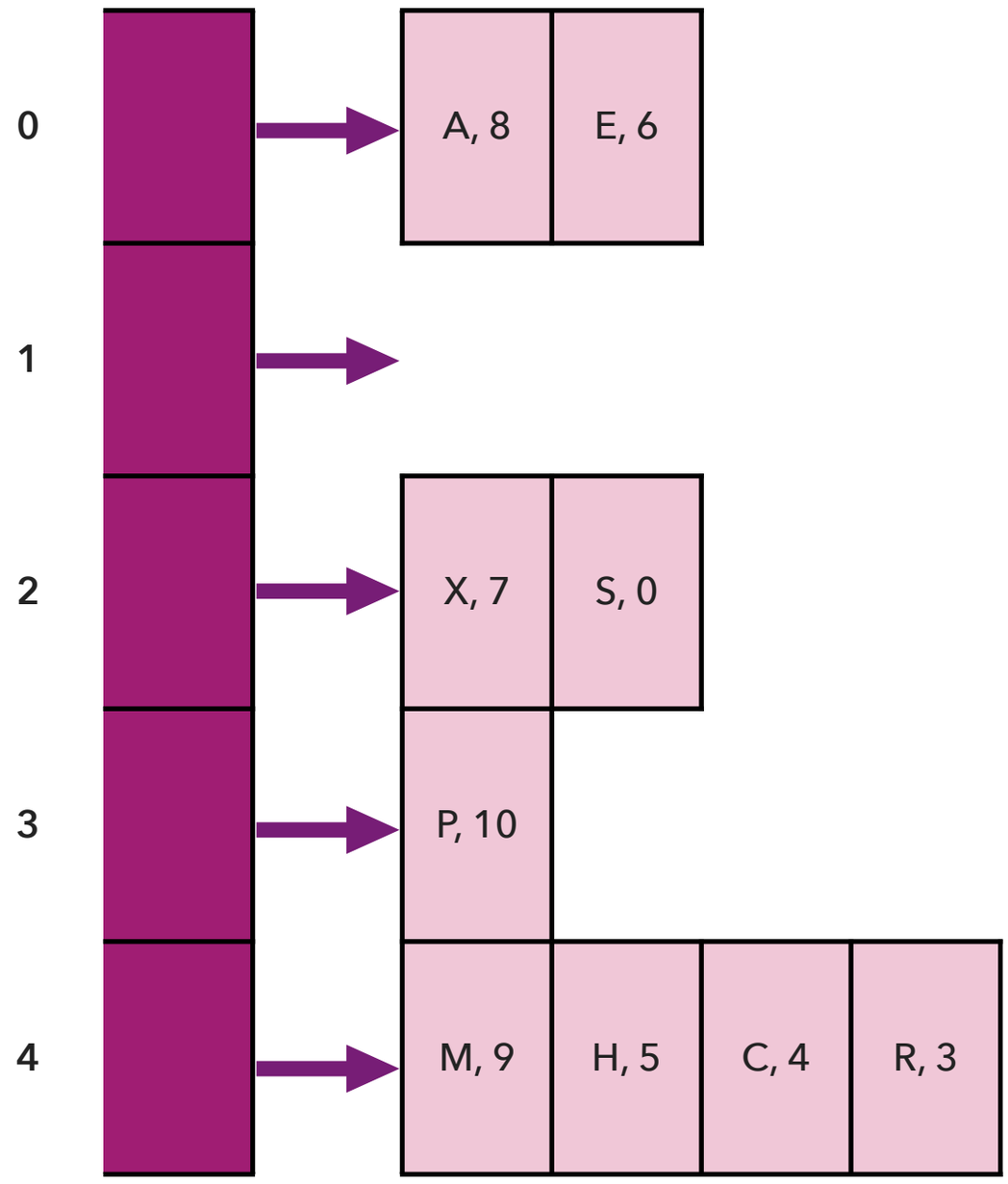
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9



Next step: Insert (P, 10)

Separate Chaining Example

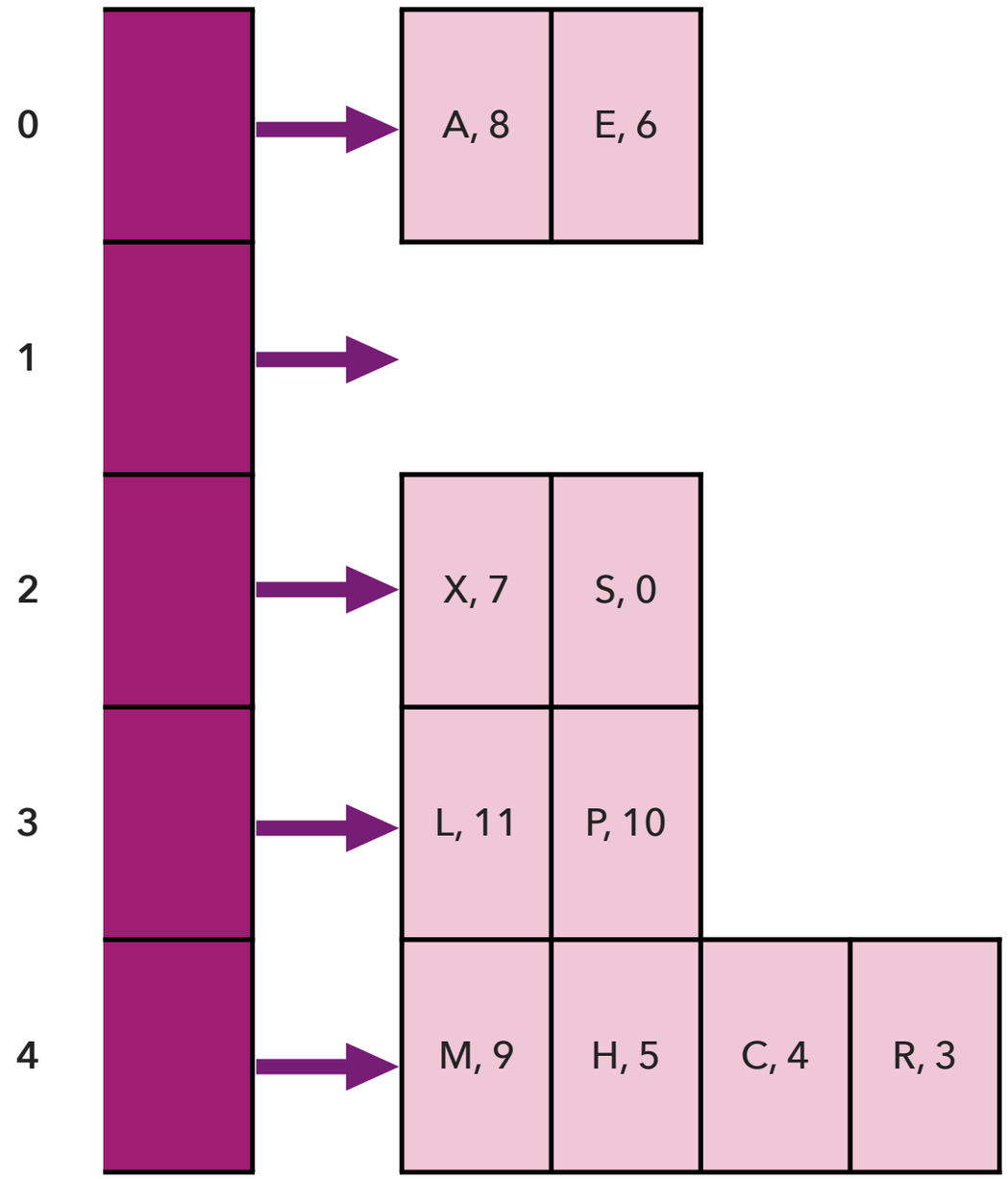
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10



Next step: Insert (L, 11)

Separate Chaining Example

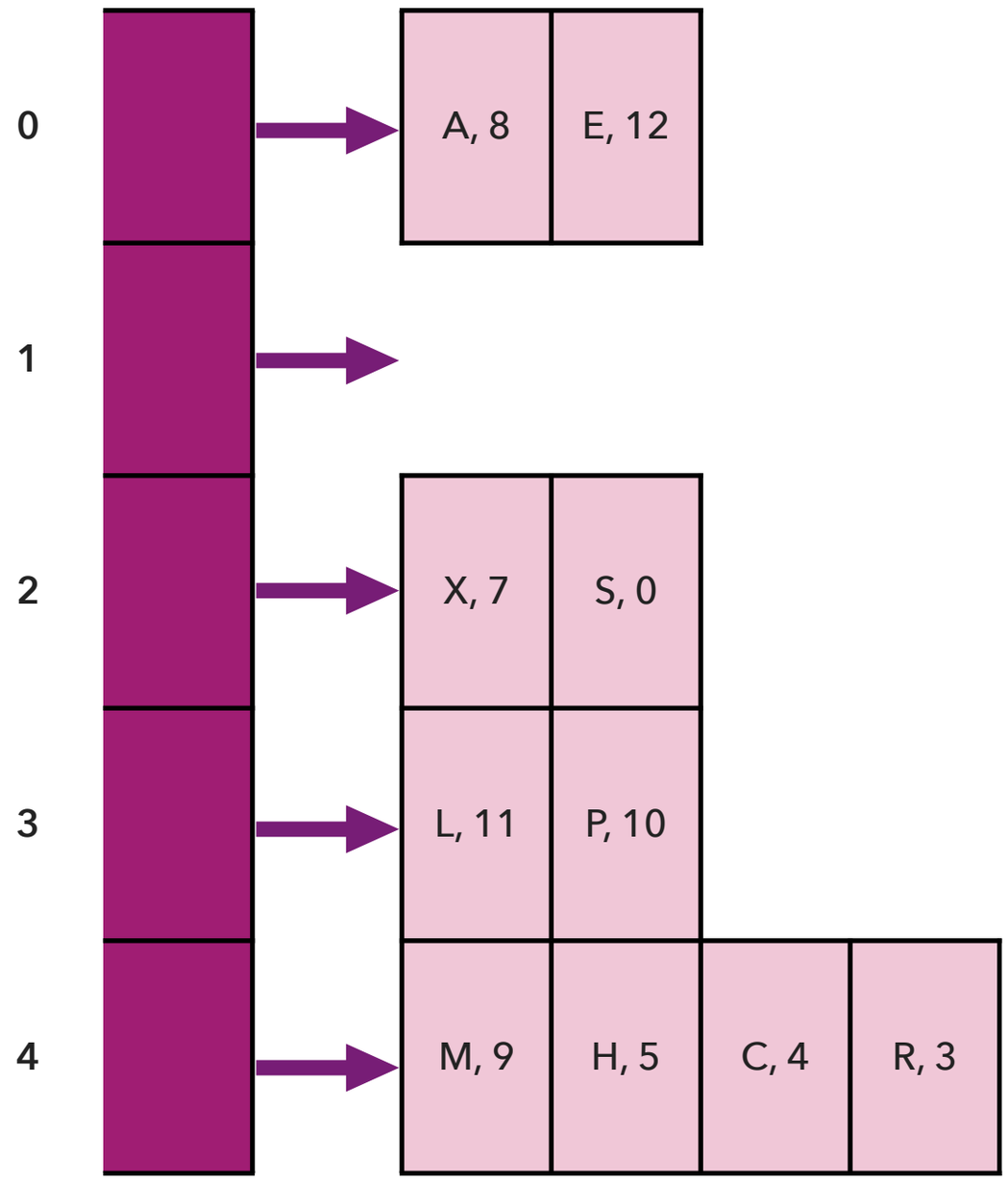
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11



Next step: Insert (E, 12)

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11
E	0	12

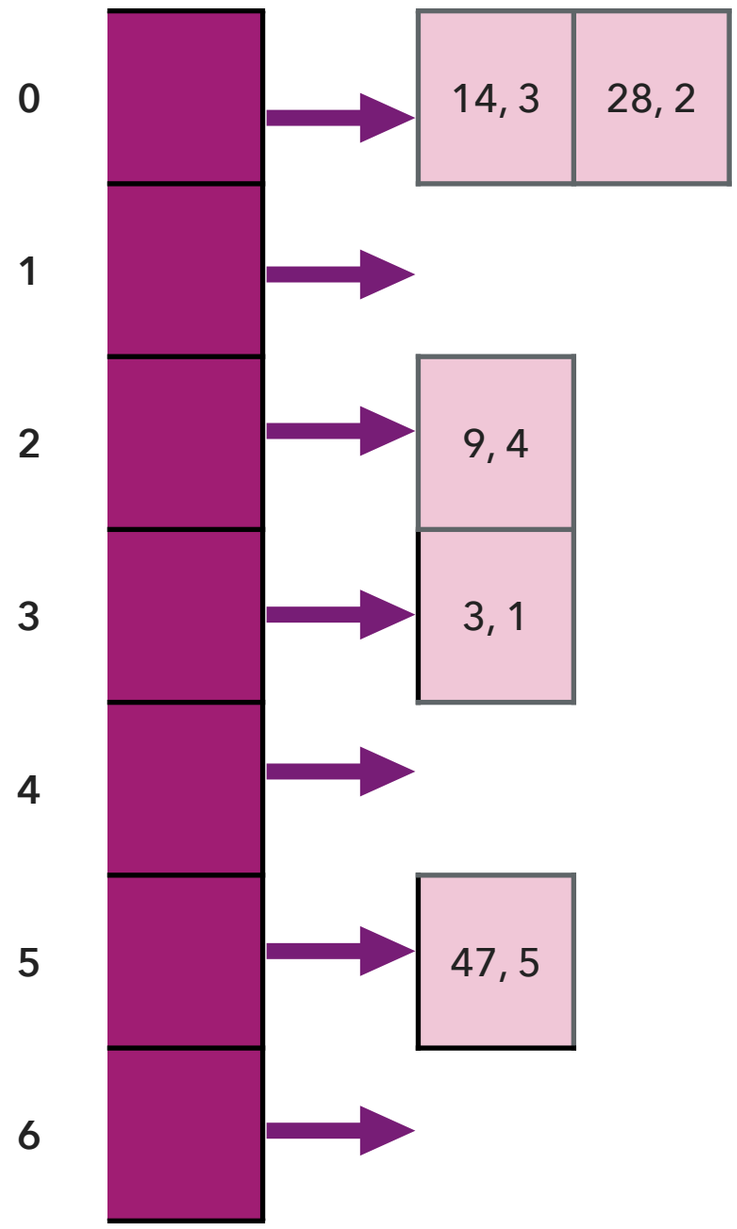


Practice Time

- Assume a dictionary implemented using hashing and separate chaining for handling collisions.
- Let $m = 7$ be the hash table size.
- For simplicity, we will assume that keys are integers and that the hash value for each key k is calculated as $h(k) = k \,\%\, m$.
- Insert the key-value pairs (47, 0), (3, 1), (28, 2), (14, 3), (9,4), (47,5) and show the resulting dictionary.

Answer

Key	Hash	Value
47	5	0
3	3	1
28	0	2
14	0	3
9	2	4
47	5	5



Symbol table with separate chaining implementation

```
public class SeparateChainingLiteHashST<Key, Value> {

    private int m = 128;  // hash table size
    private Node[] st = new Node[m];
    // array of linked-list symbol tables. Node is inner class that holds keys and values of type Object

    public Value get(Key key) {
        int i = hash(key);                                  // compute hash value - bitwise & and mod
        for (Node x = st[i]; x != null; x = x.next)         // traverse linked list
            if (key.equals(x.key)) return (Value) x.val; // return when found
        return null;
    }

    public void put(Key key, Value val) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)         // search for existing node, if found update
            if (key.equals(x.key)) {
                x.val = val;
                return;
        }
        st[i] = new Node(key, val, st[i];                   // create new node at head of linked list
    }                                                       // link to old head of list
```

Analysis of Separate Chaining

- Under uniform hashing assumption, if n keys to hash in a table with size m, the length of each chain is ~n/m.
- Consequence: Number of probes (calls to either `equals()` or `hashCode()`) for search/insert is proportional to n/m (m times faster than sequential search in a single chain).
- m too large -> too many empty chains.
- m too small -> chains too long.
- Typical choice: m~$1/5n$ -> constant time per operation.

Resizing in a separate-chaining hash table

- Goal: Average length of chain n/m = constant lookup.
  - Double hash table size when $n/m \geq 8$.
  - Halve hash table size when $n/m \leq 2$.
- Need to rehash all keys when resizing (hashCode value for key does not change, but hash value changes as it depends on table size).

Parting thoughts about separate-chaining

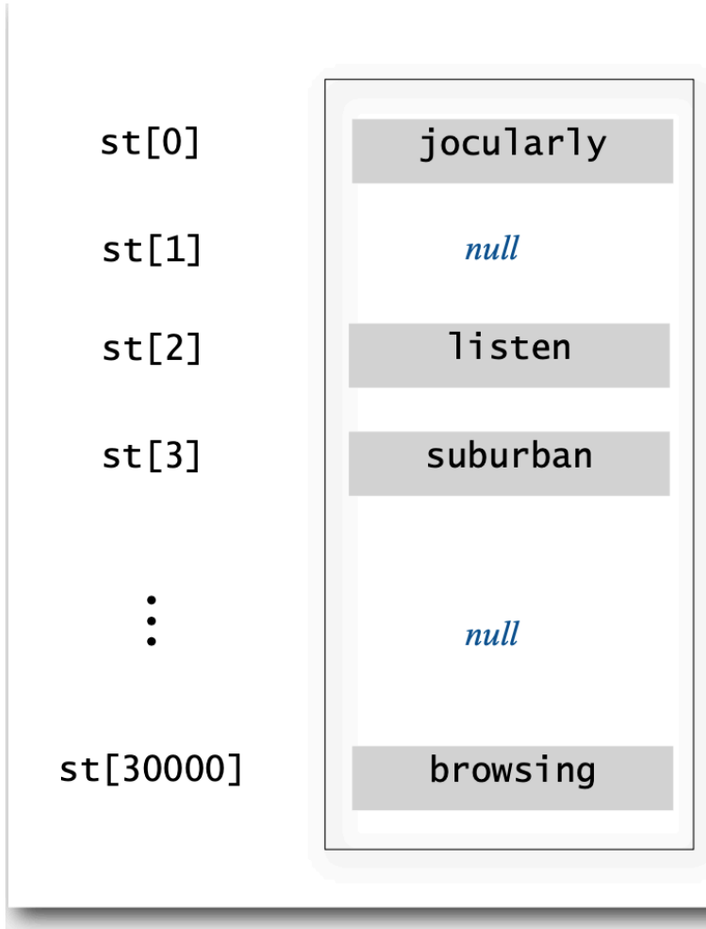
- Deletion: Easy! Hash key, find its chain, search for a node that contains it and remove it.
- Ordered operations: not supported. Instead, look into (balanced) BSTs.
- Fastest and most widely used dictionary implementation for applications where key is not important.

Lecture 26-27: Hash tables

- Hash functions
- Separate chaining
- Open addressing

Linear Probing

- Belongs in the open addressing family.
- Alternate approach to handle collisions when $m > n$.
- Maintain keys and values in two parallel arrays.
- When a new key collides, find next empty slot and put it there.
- If the array is full, the search would not terminate.



linear probing (M = 30001, N = 15000)

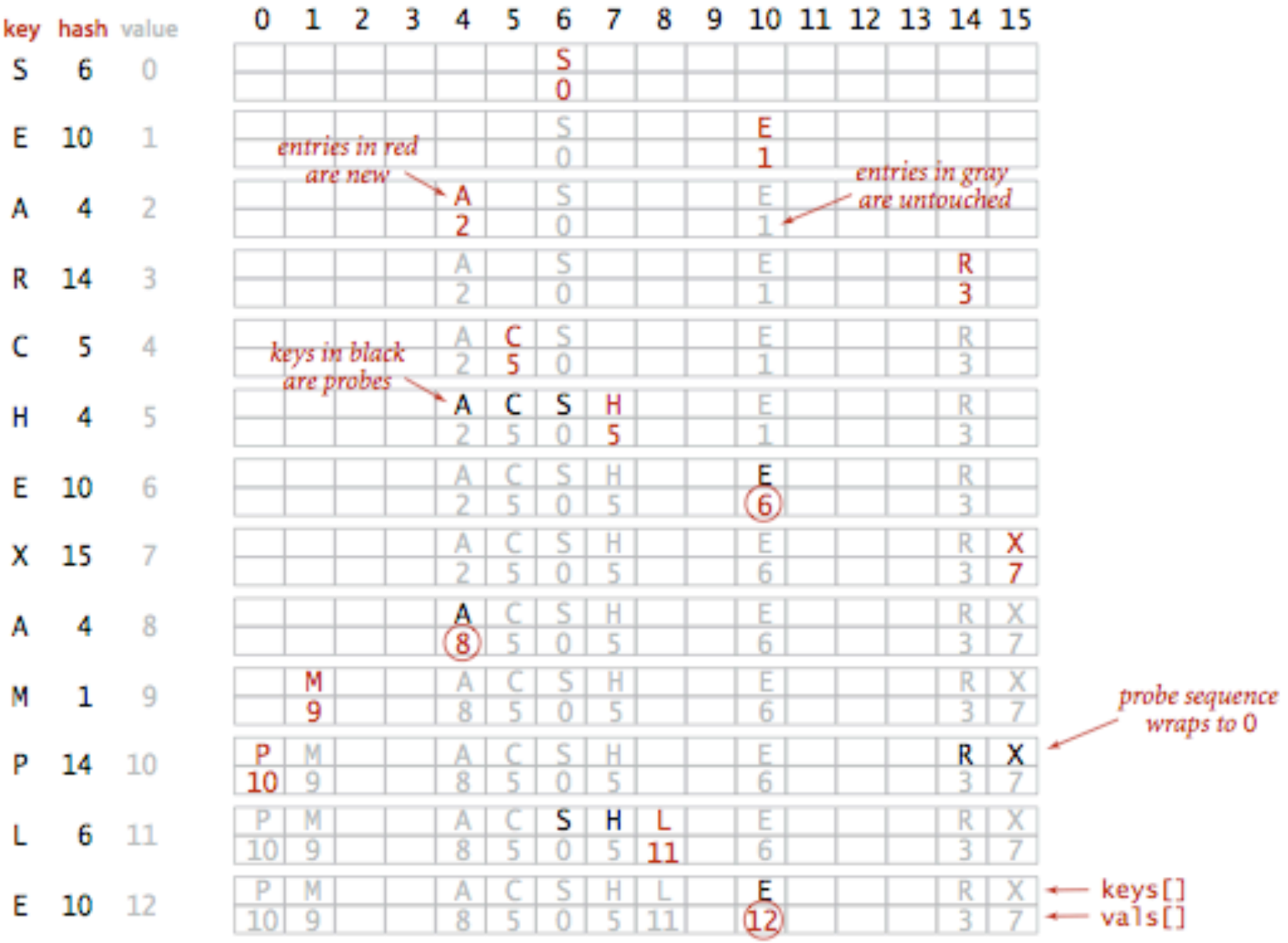
Linear Probing

- Hash: Map key to integer i between 0 and $m - 1$.
- Insert: Put at index i if free. If not, try $i + 1$, $i + 2$, etc.
- Search: Search table index i. If occupied but no match, try $i + 1$, $i + 2$, etc
  - If you find a gap then you know that it does not exist.
- Table size m **must** be greater than the number of key-value pairs n.

http://algs4.cs.princeton.edu

3.4 LINEAR PROBING DEMO

Linear Probing Example



Trace of linear-probing ST implementation for standard indexing client

Practice time

- Assume a dictionary implemented using hashing and linear probing for handling collisions.
- Let $m = 7$ be the hash table size.
- For simplicity, we will assume that keys are integers and that the hash value for each key k is calculated as $h(k) = k \,\%\, m$.
- Insert the key-value pairs (47, 0), (3, 1), (28, 2), (14, 3), (9,4), (47,5) and show the resulting dictionary.

Answer

Key	Hash	Value
47	5	0
3	3	1
28	0	2
14	0	3
9	2	4
47	5	5

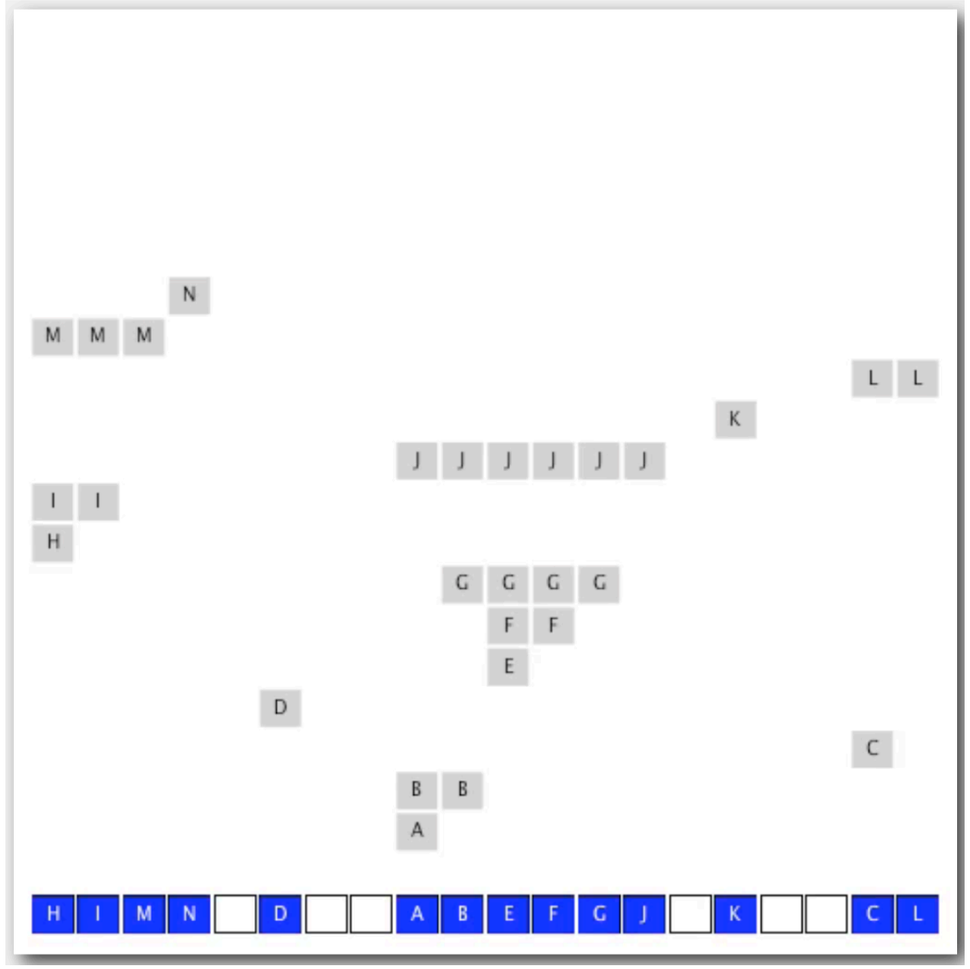
Keys	28	14	9	3		47	
Values	2	3	2	1		5	
Indices	0	1	2	3	4	5	6

Symbol table with linear probing implementation

```
public class LinearProbingHashST<Key, Value> {

    private int m = 32768;  // hash table size
    private Value[] Vals = (Value[]) new Object[m]; // parallel arrays
    private Key[] Vals = (Key[]) new Object[m];

    public Value get(Key key) {
        for (int i = hash(key); keys[i] != null; i = (i+1) % m;) // start at hash
            if (key.equals(keys[i])) return vals[I];         // increment by 1, wrap
        return null;
    }

    public void put(Key key, Value val) {
        int i;
        for (int i = hash(key); keys[i] != null; i = (i+1) % m;)  // start at hash
            if (key.equals(keys[I])){                             // increment by 1, wrap
                break;
        }
        keys[i] = key;
        vals[i] = val;
    }
```

Primary clustering

- Cluster: a contiguous block of keys.
- Observation: new keys likely to hash in middle of big clusters.



Analysis of Linear Probing

- Proposition: Under uniform hashing assumption, the average number of probes in a linear-probing hash table of size m that contains $n = \alpha m$ keys is at most
  - $1/2(1 + \frac{1}{1-a})$ for search hits and
  - $1/2(1 + \frac{1}{(1-a)^2})$ for search misses and insertions.
  - [Knuth 1963]
- Parameters:
  - m too large -> too many empty array entries.
  - m too small -> search time becomes too long.
  - Typical choice for load factor: $\alpha = n/m \sim 1/2$ -> constant time per operation.

Resizing in a linear probing hash table

- Goal: Fullness of array (load factor) $n/m \leq 1/2$.
  - Double hash table size when $n/m \geq 1/2$.
  - Halve hash table size when $n/m \leq 1/8$.
  - Need to rehash all keys when resizing (hash code does not change, but hash value changes as it depends on table size).
  - Deletion not straightforward.

Quadratic Probing

- Another open addressing technique that aims to reduce primary clustering by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

- Modify the probe sequence so that $h(k, i) = (h(k) + c_1 i + c_2 i^2) \% m, c_2 \neq 0$, where i is the i-th time we have had a collision for the given index.

  - When $c_2 = 0$, then quadratic probing degrades to linear probing.

Quadratic probing - Example

- $h(k) = k \,\%\, m$ and $h(k, i) = (h(k) + i^2) \,\%\, m$.
- Assume $m = 13$, and key-value pairs to insert: (17,0), (33,1), (18,2), (20,3), (44,4), (11,5), (19,6), (7,7).

	0	1	2	3	4	5	6	7	8	9	10	11	12	
(17,0)					17									
(33,1)					17			33						
(18,2)					17	18		33						
(20,3)					17	18		33	20					Collision!
(44,4)					17	18	44	33	20					Collision!
(11,5)					17	18	44	33	20			11		
(19,6)					17	18	44	33	20		19	11		Collision!
(7,7)				7	17	18	44	33	20		19	11		Collision!

Summary for dictionary/symbol table operations

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
BST	n	n	n	$\log n$	$\log n$	$\log n$
2-3 search tree	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
Separate chaining	n	n	n	1	1	1
Open addressing	n	n	n	1	1	1

Hash tables vs balanced search trees

- Hash tables:
  - Simpler to code.
  - No effective alternative of unordered keys.
  - Faster for simple keys (a few arithmetic operations versus $\log n$ compares).
- Balanced search trees:
  - Stronger performance guarantee.
  - Support for ordered symbol table operations.
  - Easier to implement `compareTo()` than `hashCode()`.
- Java includes both:
  - Balanced search trees: `java.util.TreeMap`, `java.util.TreeSet`.
  - Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.

Lecture 26-27: Hash tables

- Hash functions
- Separate chaining
- Open addressing

Readings:

- Textbook: Chapter 3.4 (Pages 458-477)
- Website:
  - https://algs4.cs.princeton.edu/34hash/
- Visualization:
  - https://visualgo.net/en/hashtable

Practice Problems:

- 3.4.1-3.4.13