SORTING

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

21: HashTables

Tom Yeh
he/him/his

Class News

- Midterm scheduled for this Thursday 4/8/22
  - Option 1 - keep existing schedule for 4/8/22
  - Option 2 - postpone until Tuesday 4/12

Representation

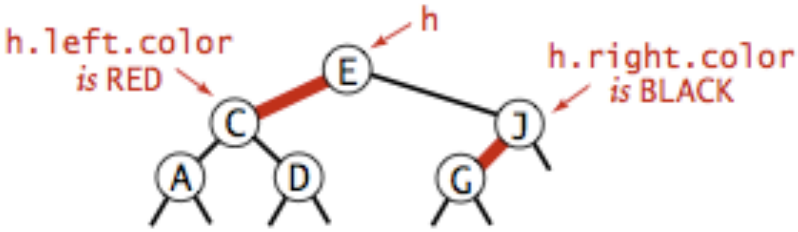
- There is no representation of the links in BST
  - Each node is pointed to by one node, its parent.
  - We can use this to encode the color of the links in nodes.
- True if the link from the parent is red and false if it is black. Null links are black.

```
private static final boolean RED   = true;
private static final boolean BLACK = false;

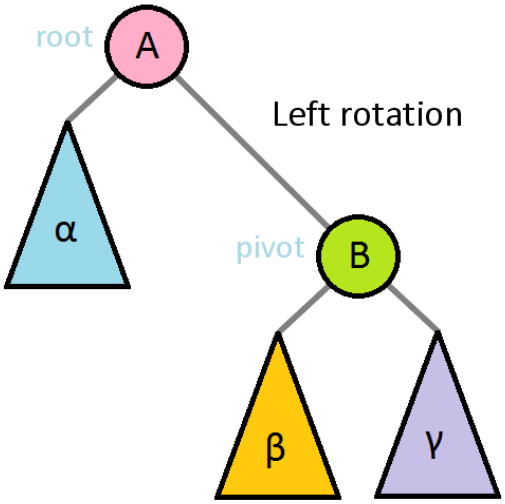
private Node root;     // root of the BST

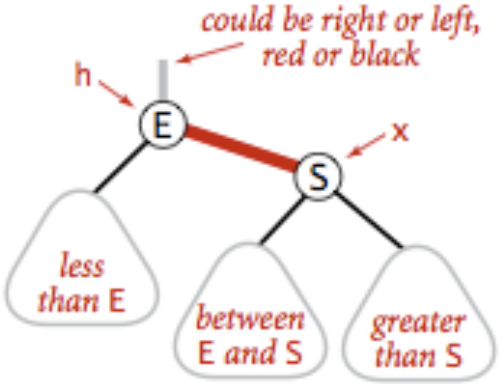
// BST helper node data type
private class Node {
    private Key key;           // key
    private Value val;         // associated data
    private Node left, right;  // links to left and righ
    private boolean color;     // color of parent link
    private int size;          // subtree count

private boolean isRed(Node x) {
    if (x == null) return false;
    return x.color == RED;
}
```

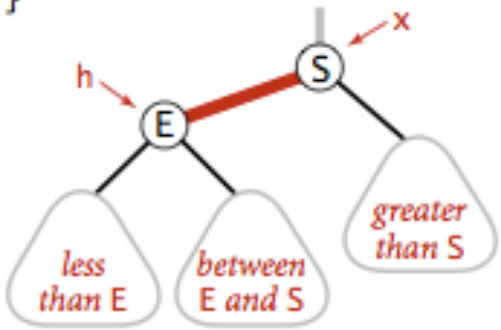


Left rotation: Orient a (temporarily) right-leaning red link to lean left



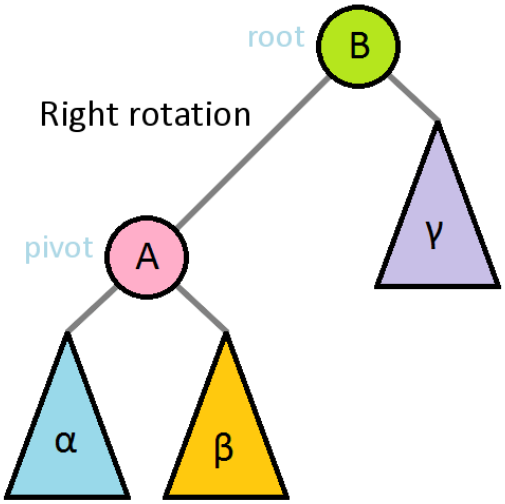


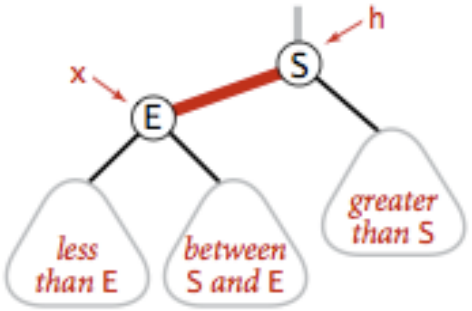

```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



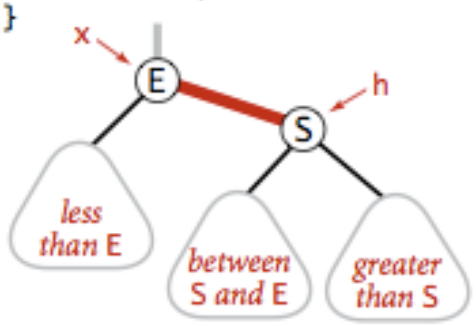
Left rotate (right link of h)

Right rotation: Orient a left-leaning red link to a (temporarily) lean right



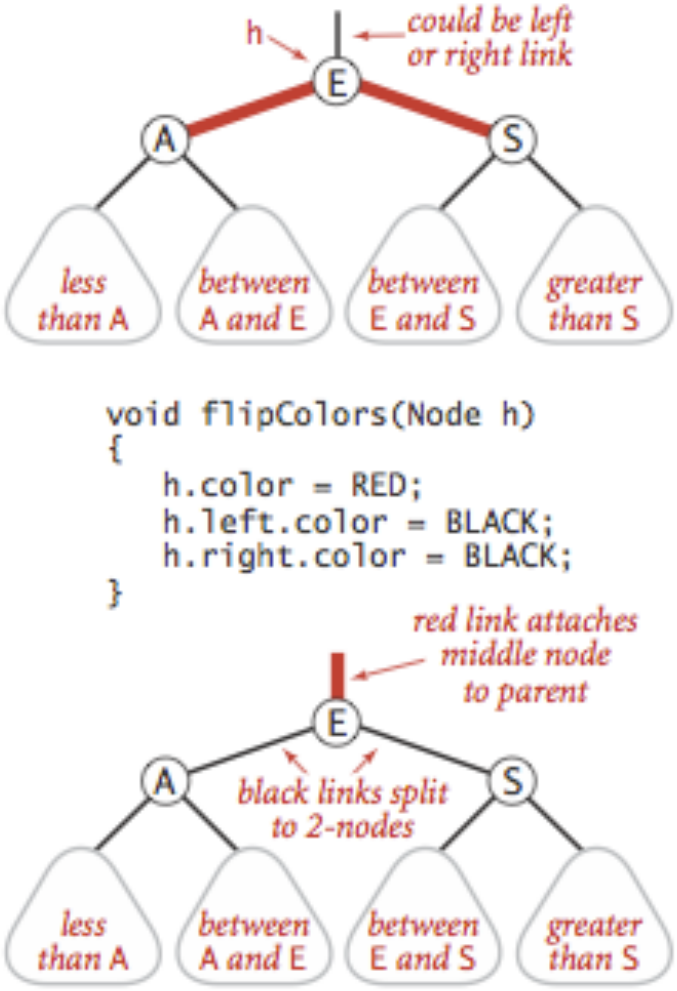



```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



Right rotate (left link of h)

Color flip: Recolor to split a (temporary) 4-node

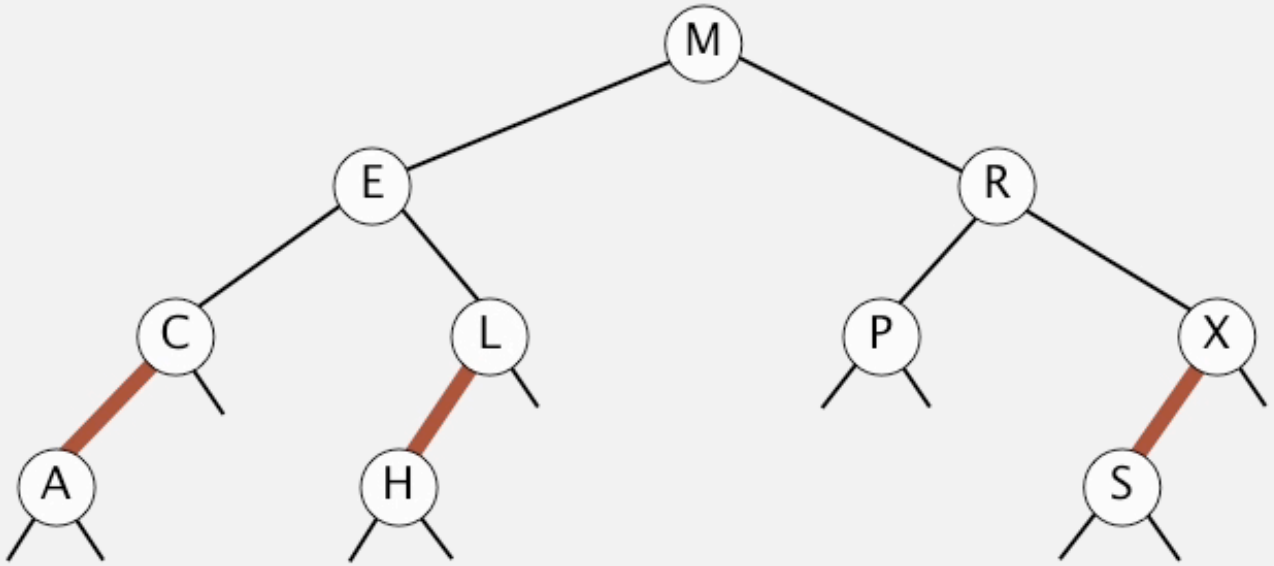



```
void flipColors(Node h)
{
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

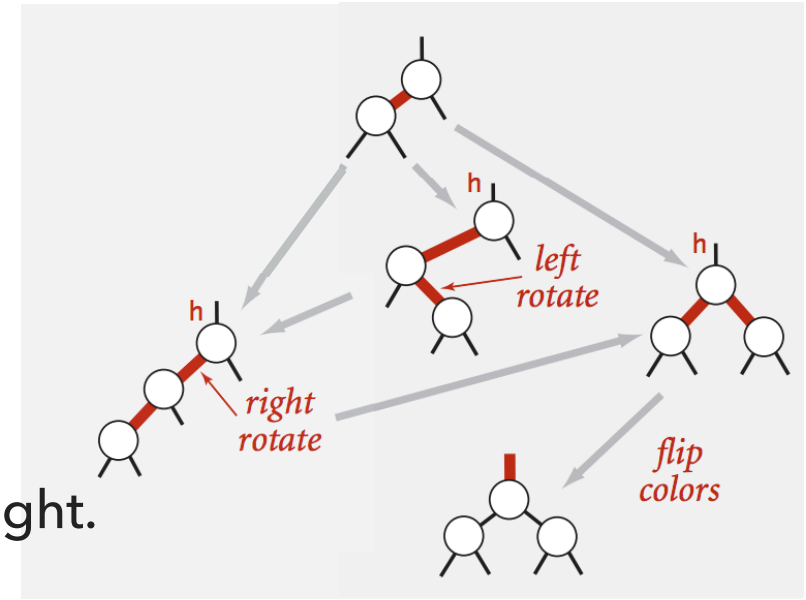
Flipping colors to split a 4-node

Red-black BST construction demo

red-black BST



Implementation



- Only three cases:
  - Right child red; left child black: rotate left.
  - Left child red; left-left grandchild red: rotate right.
  - Both children red: flip colors.

```
// insert the key-value pair in the subtree rooted at h
private Node put(Node h, Key key, Value val) {
    if (h == null) return new Node(key, val, RED, 1);  // Insert at bottom and color red

    int cmp = key.compareTo(h.key);                    // Compare as before to traverse tree
    if      (cmp < 0) h.left  = put(h.left,  key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else              h.val   = val;

    if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h); // Fix any right-leaning links
    if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);// 2 left red links
    if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);     // 4-node
    h.size = size(h.left) + size(h.right) + 1;

    return h;
}
```

Visualization of insertion into a LLRB tree

- 255 insertions in ascending order.

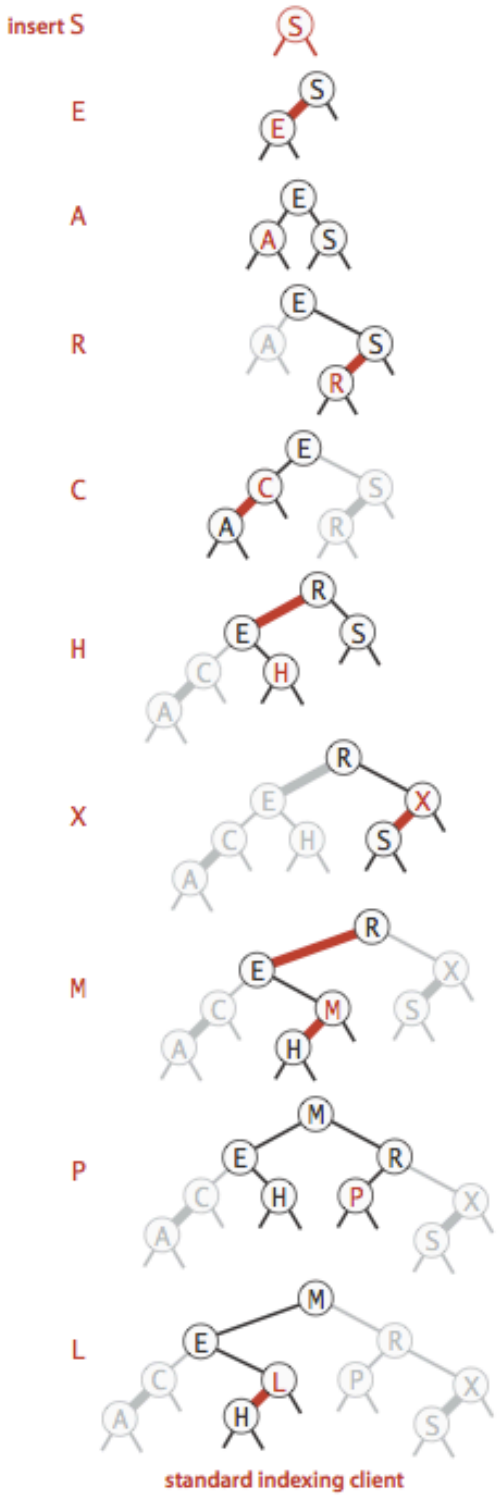
Visualization of insertion into a LLRB tree

- 255 insertions in descending order.

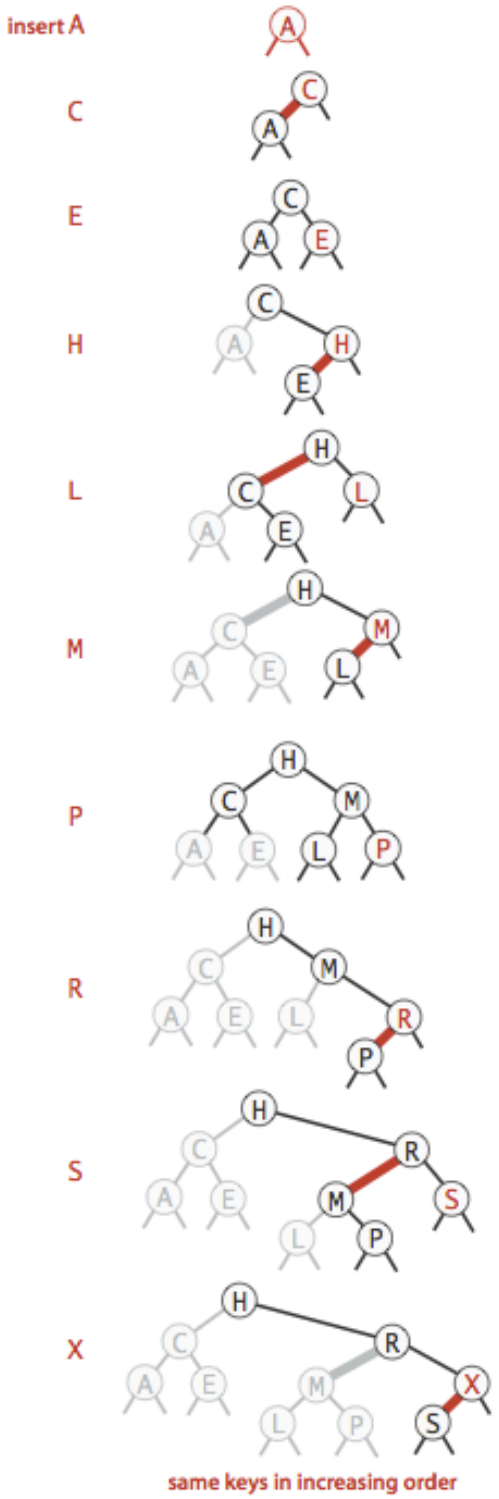
Visualization of insertion into a LLRB tree

- 255 insertions in random order.

Examples



standard indexing client



same keys in increasing order

Lecture 28-29: Left-leaning Red-Black Trees

- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

Balance in LLRB trees

- Height of LLRB trees is $\leq 2 \log n$ in the worst case. Can you think of the worst case?
- Worst case is a 2-3 tree that is all 2-nodes except that the left-most path is made up of 3-nodes.
- All ordered operations (min, max, floor, ceiling) etc. are also $O(\log n)$.

Summary for symbol table operations

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
BST	n	n	n	$1.39 \log n$	$1.39 \log n$	$\sqrt{n}$
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	$2 \log n$	$2 \log n$	$2 \log n$	$1 \log n$	$1 \log n$	$1 \log n$

Summary for symbol table operations

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
Sequential search (unordered	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	$2 \log n$	$2 \log n$	$2 \log n$	$1 \log n$	$1 \log n$	$1 \log n$

Lecture 28-29: Left-leaning Red-Black Trees

- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

Red-black trees

- Why red-black? Invented at Xerox PARC, had a laser printer and red and black had the best contrast...
- Left-leaning red-black trees [Sedgewick, 2008]
  - Inspired by difficulties in proper implementation of RB BSTs.
- RB BSTs have been involved in lawsuit because of improper implementation.
  - Telephone service outage due to exceeding height bound
  - Telephone company sues database provider

Balanced trees in the wild

- Red-black trees are widely used as system symbol tables.
  - e.g., Java: `java.util.TreeMap` and `java.util.TreeSet`.
- Other balanced BSTs: AVL, splay, randomized.
- 2-3 search trees are a subset of b-trees.
  - See book for more.
  - B-trees are widely used for file systems and databases.

Lecture 28-29: Left-leaning Red-Black Trees

- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

Readings:

- Textbook: Chapter 3.3 (Pages 432-447)
- Website:
  - https://algs4.cs.princeton.edu/33balanced/

Practice Problems:

- 3.3.9-3.3.22

Lecture 20: Midterm Topics

- Sorting
- Heaps/Priority Queues
- Dictionaries
- Misc
- Practice Problems
  - End of lecture slides
  - Go over in midterm review during lab

Sorting

- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort

Sorting

- Given an array of n items, sort them in non-descending order based on a comparable key.
- Cost model counts comparisons (calls to `less()`) and exchanges (calls to `exch()`) (or array accesses).
- Not in place: If linear extra memory is required.
- Stable: If duplicate elements stay in the same order that they appear in the input.
- Practice: https://visualgo.net/en/sorting (minus quick sort).

Dictionaries

- Binary search trees
  - Search
  - Insertion
  - Deletion
- 2-3 search trees
- Left-leaning Red Black search trees

Dictionaries

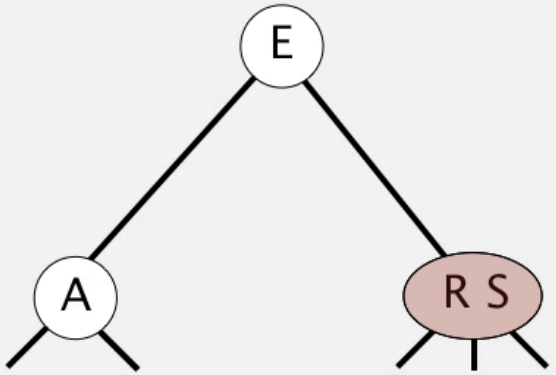
- Binary search trees
- 2-3 search trees

2-3 search trees

- Balanced (every path from root to leaf has same length) search tree that follow the symmetric order. Contain 2 nodes (one key and two children) or 3 nodes (two keys and three children).
- Search and insertion of keys (and values) is $O(\log n)$.
- A pain to implement.
- Practice: https://www.cs.usfca.edu/~galles/visualization/BTree.html (max-degree 3).

2-3 tree demo: construction

insert R



Misc

- Comparable/Comparator Interfaces
- Iterable/Iterator Interfaces
- BT Traversals

Comparable Interface

- Interface with a single method that we need to implement:
  `public int compareTo(T that)`
- Implement it so that `v.compareTo(w)`:
  - Returns >0 if v is greater than w.
  - Returns <0 if v is smaller than w.
  - Returns 0 if v is equal to w.
- Corresponds to natural ordering.

Comparator Interface

- Sometimes the natural ordering is not the type of ordering we want.
- Comparator is an interface which allows us to dictate what kind of ordering we want by implementing the method:
  `public int compare(T this, T that)`
- Implement it so that `compare(v, w)`:
  - Returns >0 if v is greater than w.
  - Returns <0 if v is smaller than w.
  - Returns 0 if v is equal to w.

```
public static Comparator<ClassName> reverseComparator(){
   return (ClassName a, ClassName b)->{return -a.compareTo(b)};
 }
```

Misc

- Comparable/Comparator Interfaces
- Iterable/Iterator Interfaces
- BT Traversals

`Iterable<T>` Interface

- Interface with a single method that we need to implement: `Iterator<T> iterator()`
- Class becomes iterable, that is it can be traversed with a for-each loop.
-
```
for (String student: students){
        System.out.println(student);
}
```

`Iterator<T>` Interface

- Interface with two methods that we need to implement: `boolean hasNext()` and `T next()`.
- `hasNext()` checks whether there is any element we have not seen yet.
- `next()` returns the next available element.
- Always check if there are any available elements before returning the next one.
- Typically a comparable class, has an inner class that implements Iterator. Outer class's `iterator` method returns an instance of inner class.
- Can also be implemented in a standalone class where collection to iterate over is passed in the constructor.

Misc

- Comparable/Comparator Interfaces
- Iterable/Iterator Interfaces
- BT Traversals

BT traversals

- Pre-order: mark root visited, left subtree, right subtree.
- In-order: left subtree, mark root visited, right subtree.
- Post-order: left subtree, right subtree, mark root visited.
- Level-order: start at root, mark each node as visited level by level, from left to right.

Practice Problems

- Problem 1 - Sorting
- Problem 2 - Heaps
- Problem 3 - Tree traversals
- Problem 4 - Binary Trees
- Problem 5 - Binary Search Trees
- Problem 6 - Iterators

Problem 1 - Sorting

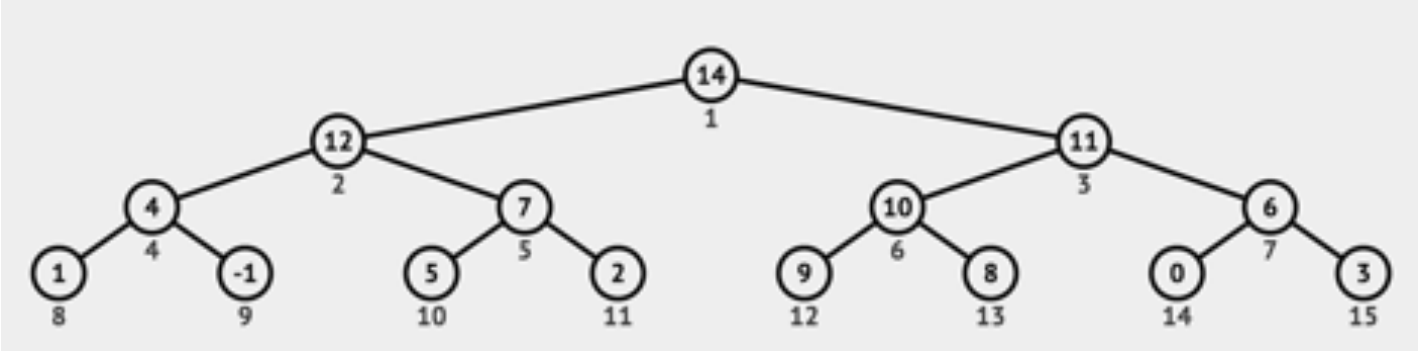
- In the next slide, you can find a table whose first row (last column 0) contains an array of 18 unsorted numbers between 1 and 50. The last row (last column 6) contains the numbers in sorted order. The other rows show the array in some intermediate state during one of these five sorting algorithms:
  - 1-Selection sort
  - 2-Insertion sort
  - 3-Mergesort
  - 4-Quicksort (no initial shuffling, one partition only)
  - 5-Heapsort
- Match each algorithm with the right row by writing its number (1-5) in the last column.

Problem 1 - Sorting

12	11	35	46	20	43	42	47	44	32	16	10	40	18	41	21	28	15	**0**
11	12	20	35	42	43	46	47	44	32	16	10	40	18	41	21	28	15	
10	11	12	46	20	43	42	47	44	32	16	35	40	18	41	21	28	15	
10	11	12	15	16	43	42	47	44	32	20	35	40	18	41	21	28	46	
43	32	42	28	20	40	41	21	15	11	16	10	35	18	12	44	46	47	
11	12	20	35	46	43	42	47	44	32	16	10	40	18	41	21	28	15	
10	11	12	15	16	18	20	21	28	32	35	40	41	42	43	44	46	47	**6**

Problem 2 - Heaps

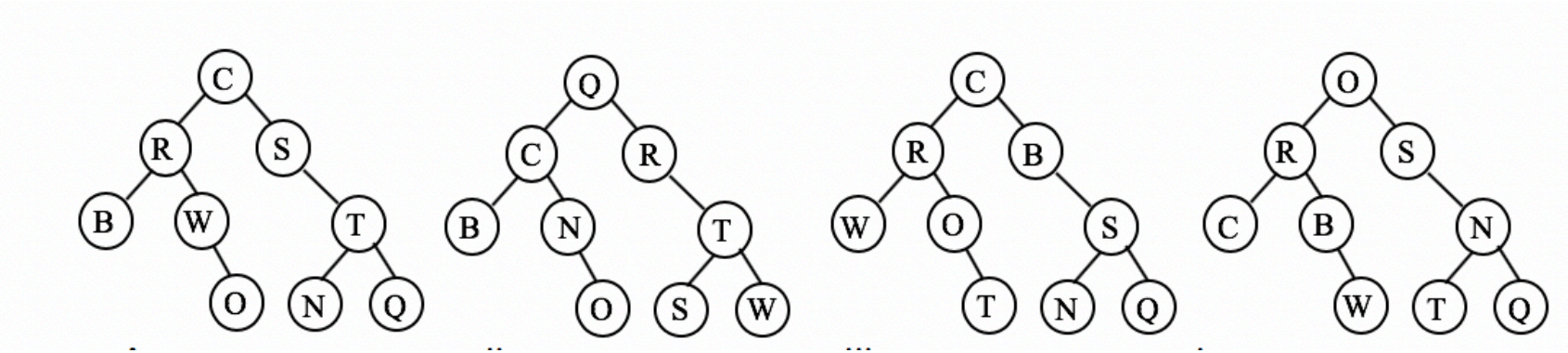
- Consider the following max-heap:



- Draw the heap after you insert key 13.
- Suppose you delete the maximum key from the original heap. Draw the heap after you delete 14.

Problem 3 - Tree Traversals

- Circle the correct binary tree(s) that would produce both of the following traversals:
  - Pre-order: C R B W O S T N Q
  - In-order: B R W O C S N T Q



Problem 4 - Binary Trees

- You are extending the functionality of the `BinaryTree` class that represents binary trees with the goal of counting the number of leaves. Remember that `BinaryTree` has a pointer to a `root` `Node` and the inner class `Node` has two pointers, `left` and `left` to the root nodes that correspond to its left and right subtrees.
- You are given the following public method:

```
public int sumLeafTree()
   return sumLeafTree(root);
}
```

- Please fill in the body of the following recursive method

```
private int sumLeafTree(Node x){…}
```

Problem 5 - Binary Search Trees

- You are extending the functionality of the BST class that represents binary search trees with the goal of counting the number of nodes whose keys fall within a given `[low, high]` range. That is you want to count how many nodes have keys that are equal or larger than `low` and equal or smaller than `high`. Remember that BST has a pointer to a `root` Node and the inner class Node has two pointers, `left` and `left` to the root nodes that correspond to its left and right subtrees and a `Comparable Key key` (please ignore the value).
- You are given the following public method:

```
public int countRange(Key low, Key high)
  return countRange(root, Key low, Key high);
}
```

- Please fill in the body of the following recursive method

```
private int countRange(Node x, Key low, Key high){…}
```

Problem 6 - Iterators

- A programmer discovers that they frequently need only the odd numbers in an arraylist of integers. As a result, they decided to write a class `OddIterator` that implements the `Iterator` interface. Please help them implement the constructor and the `hasNext()` and `next()` methods so that they can retrieve the odd values, one at a time. For example, if the arraylist contains the elements [7, 4, 1, 3, 0], the iterator should return the values 7, 1, and 3. You are given the following public class:

```
public class OddIterator implements Iterator<Integer> {

// The array whose odd values are to be enumerated
private ArrayList<Integer> myArrayList;

//any other instance variables you might need

//An iterator over the odd values of myArrayList
public OddIterator(ArrayList<Integer> myArrayList){…}

//runs in O(n) time
public boolean hasNext(){…}

//runs in O(1) time
public Integer next(){…}
}
```

Lecture 21: Hash tables

- Hash functions
- Separate chaining
- Open addressing

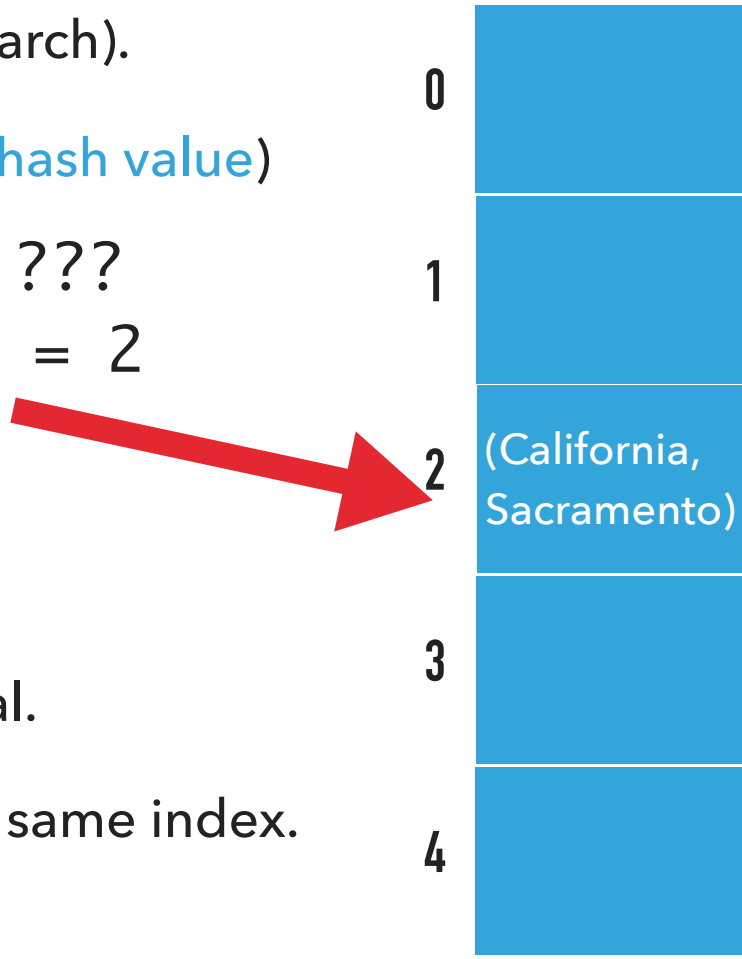
Some slides adopted from Algorithms 4th Edition or COS226

Summary for symbol table operations

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
Sequential search (unordered	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	$2 \log n$	$2 \log n$	$2 \log n$	$1 \log n$	$1 \log n$	$1 \log n$

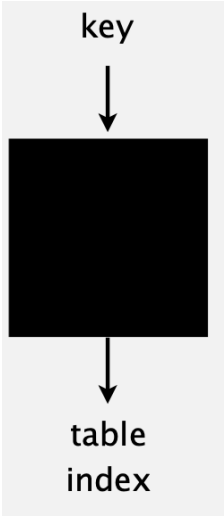
Basic plan for implementing dictionaries using hashing

- 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

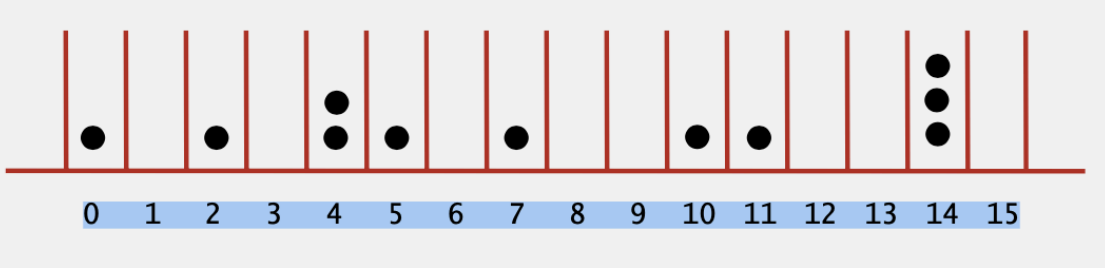
General hash code recipe in Java

- 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



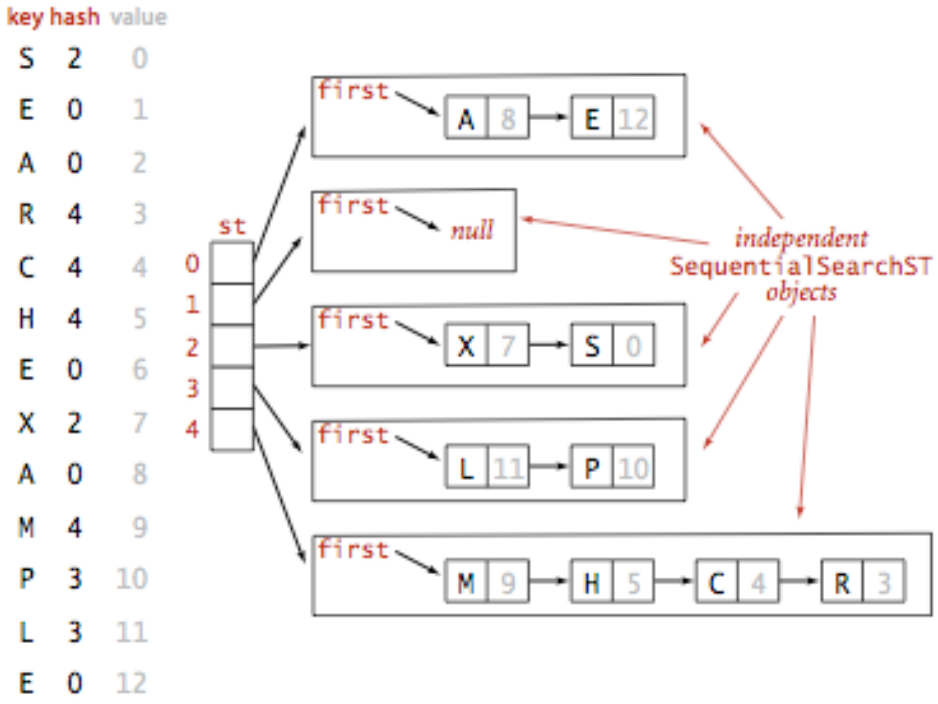
- Uniform hashing assumption: Each key is equally likely to hash to an integer between 0 and $m - 1$.
- Mathematical model: balls & bins. Toss n balls uniformly at random into m bins.
- Bad news: Expect two balls in the same bin after ~$\sqrt{(\pi m/2)}$ tosses.
  - Birthday problem: In a random group 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.

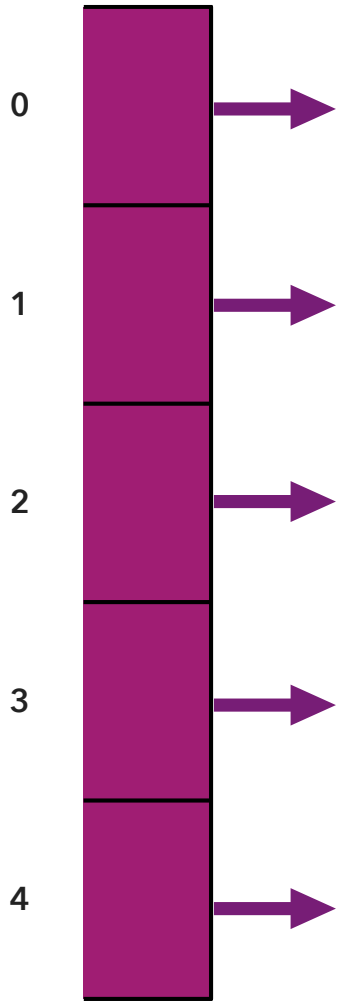


Hashing with separate chaining for standard indexing client

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.

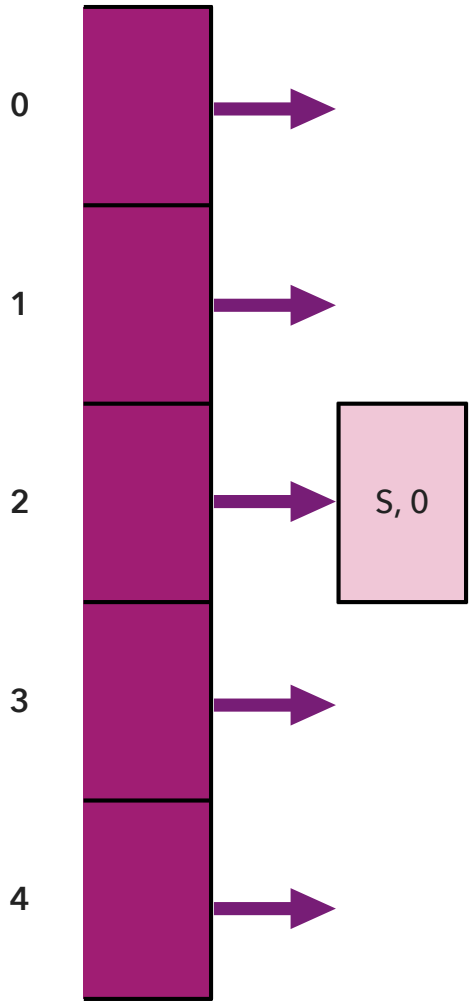
Separate Chaining Example



Next step: Insert (S, 0)

Separate Chaining Example

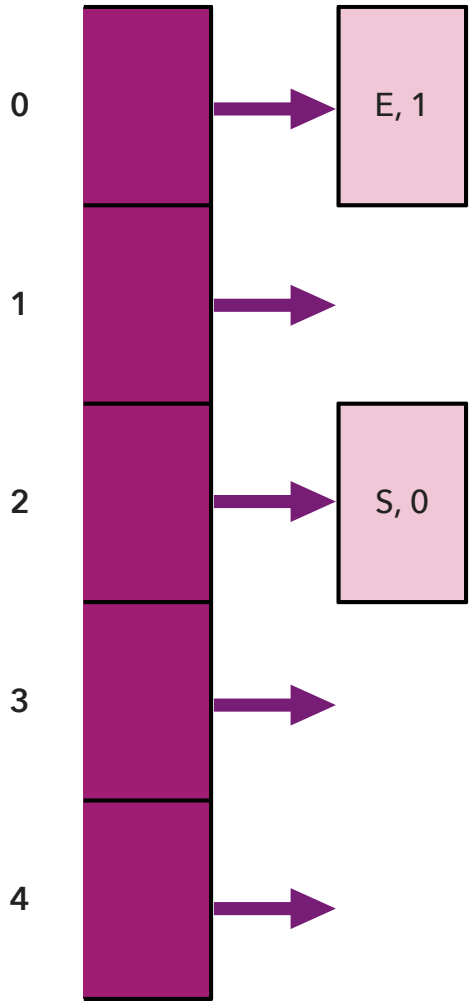
Key	Hash	Value
S	2	0



Next step: Insert (E, 1)

Separate Chaining Example

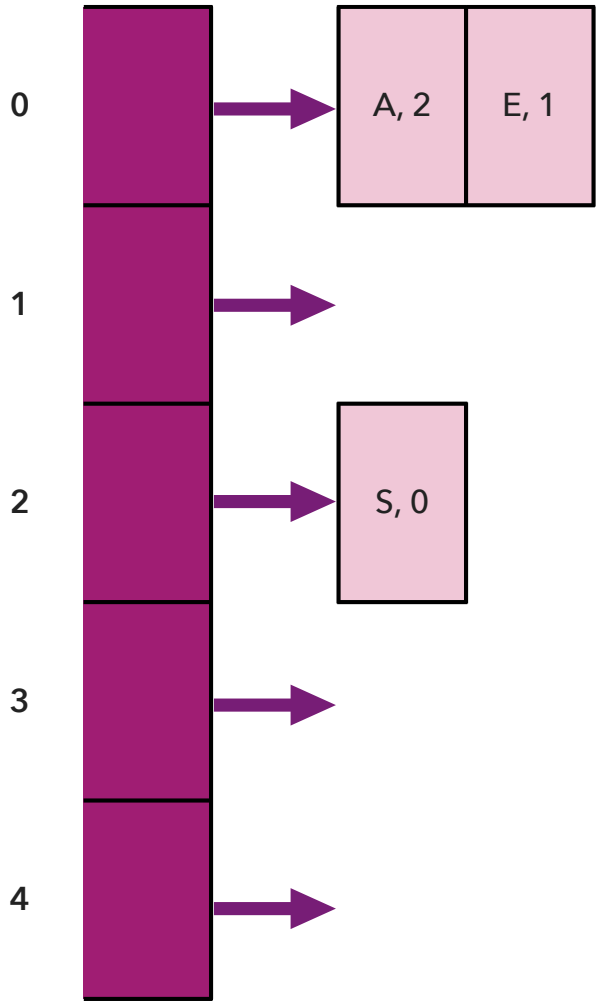
Key	Hash	Value
S	2	0
E	0	1



Next step: Insert (A, 2)

Separate Chaining Example

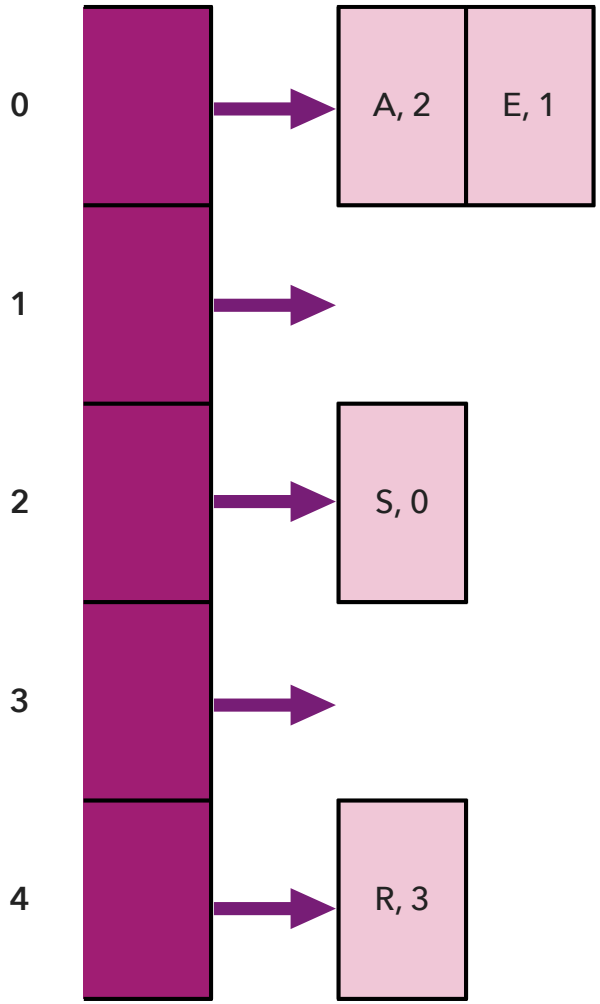
Key	Hash	Value
S	2	0
E	0	1
A	0	2



Next step: Insert (R, 3)

Separate Chaining Example

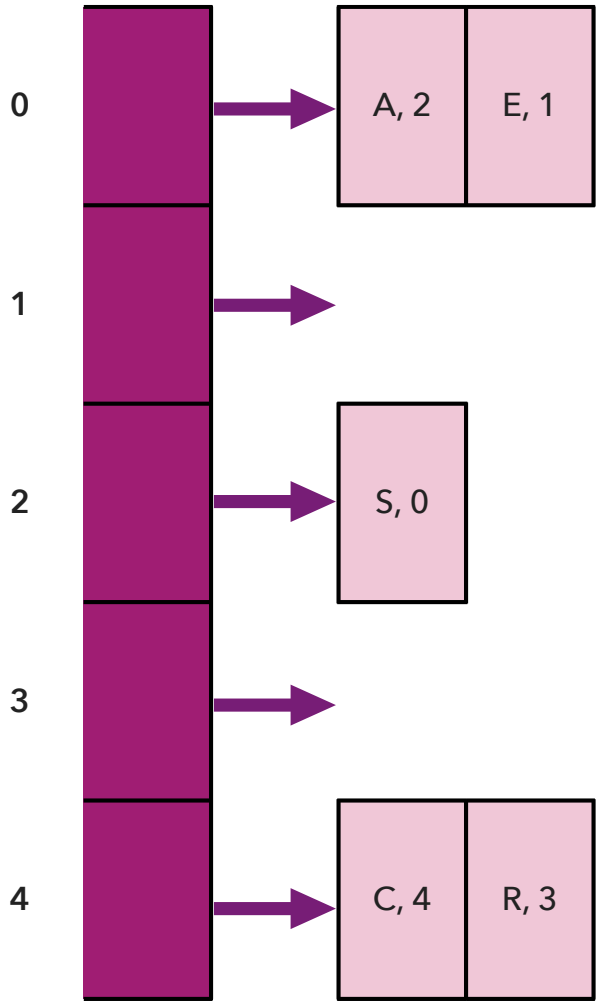
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3



Next step: Insert (C, 4)

Separate Chaining Example

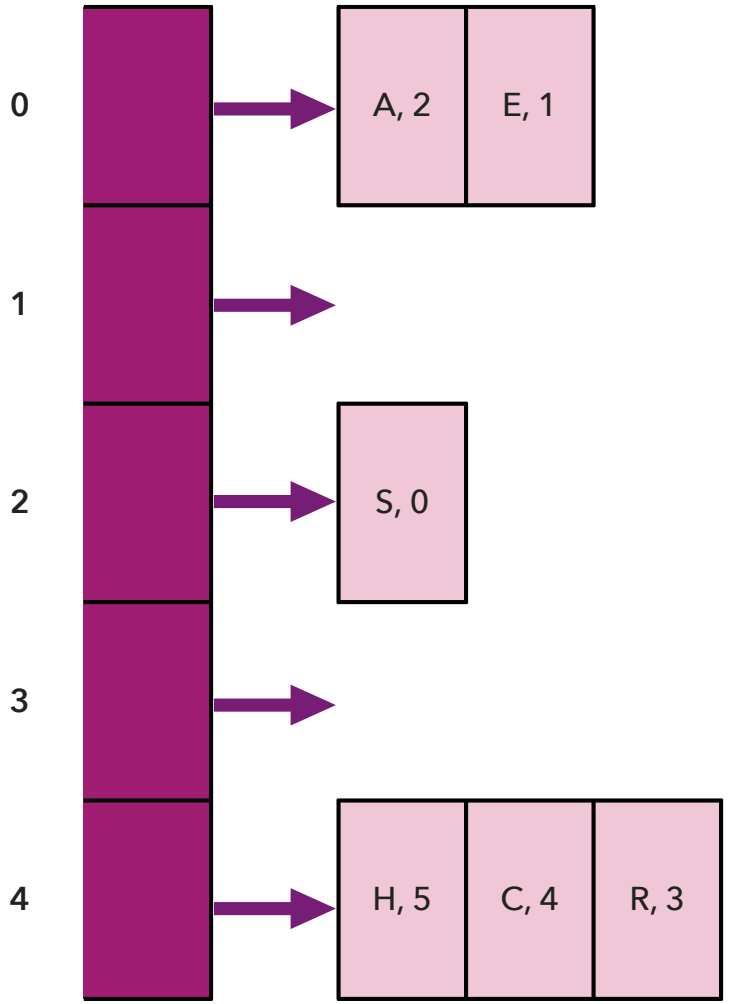
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4



Next step: Insert (H, 5)

Separate Chaining Example

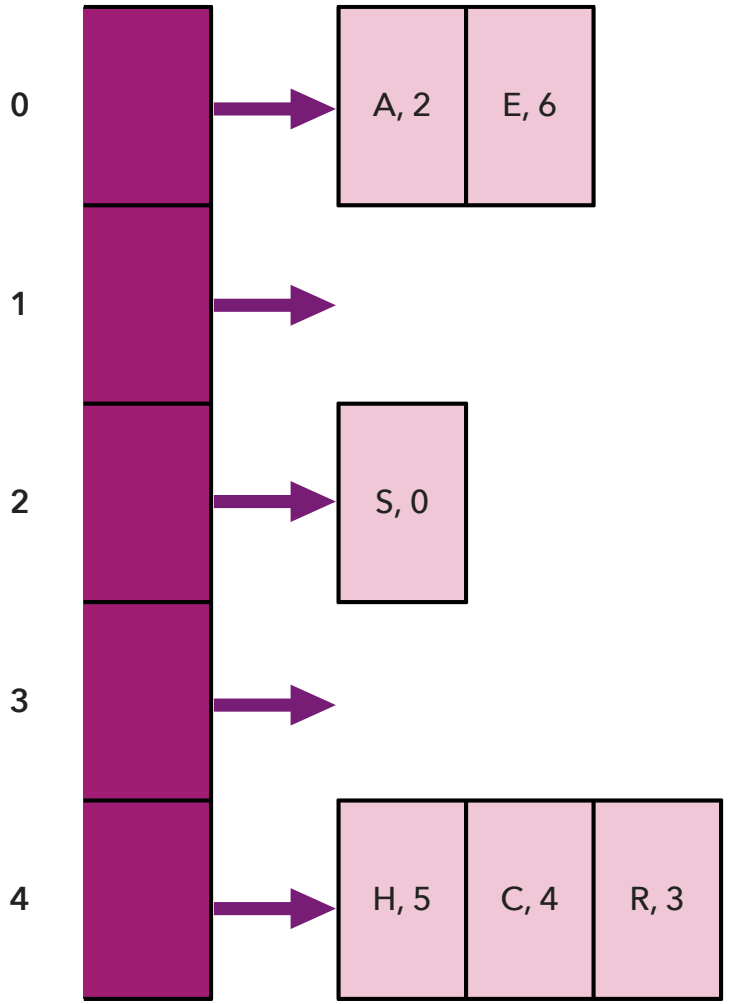
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5



Next step: Insert (E, 6)

Separate Chaining Example

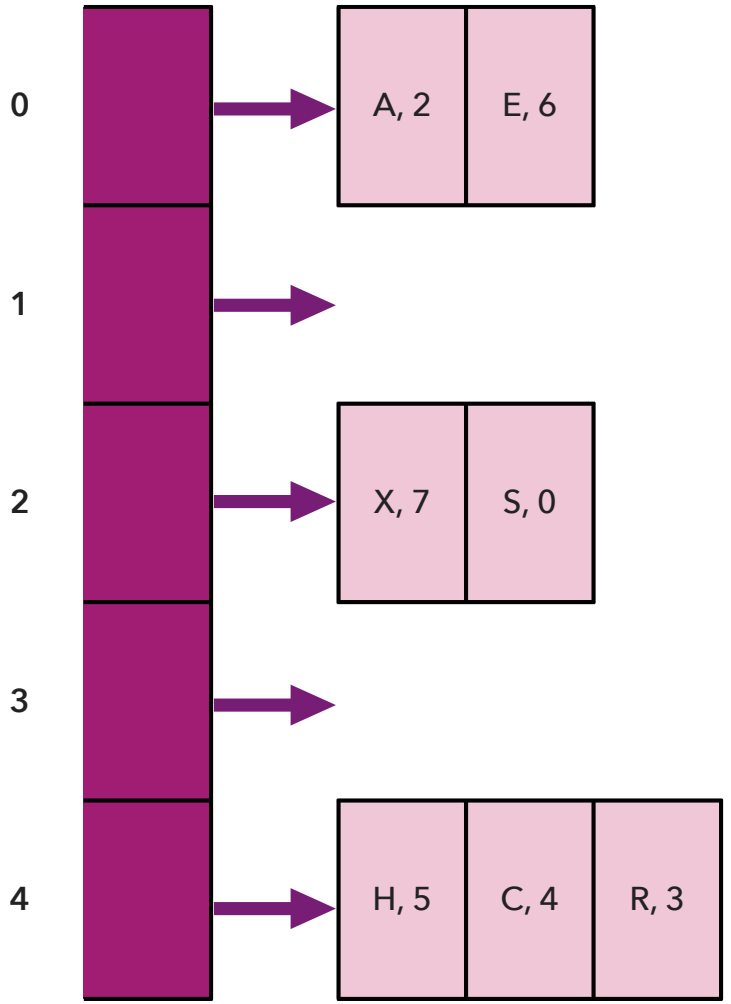
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6



Next step: Insert (X, 7)

Separate Chaining Example

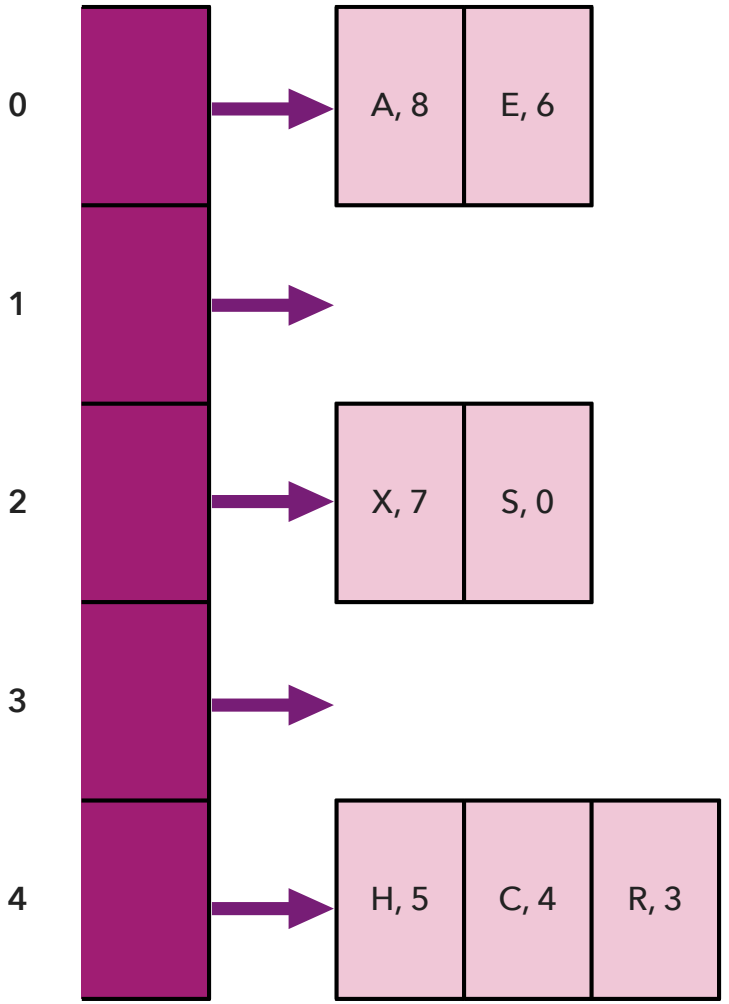
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7



Next step: Insert (A, 8)

Separate Chaining Example

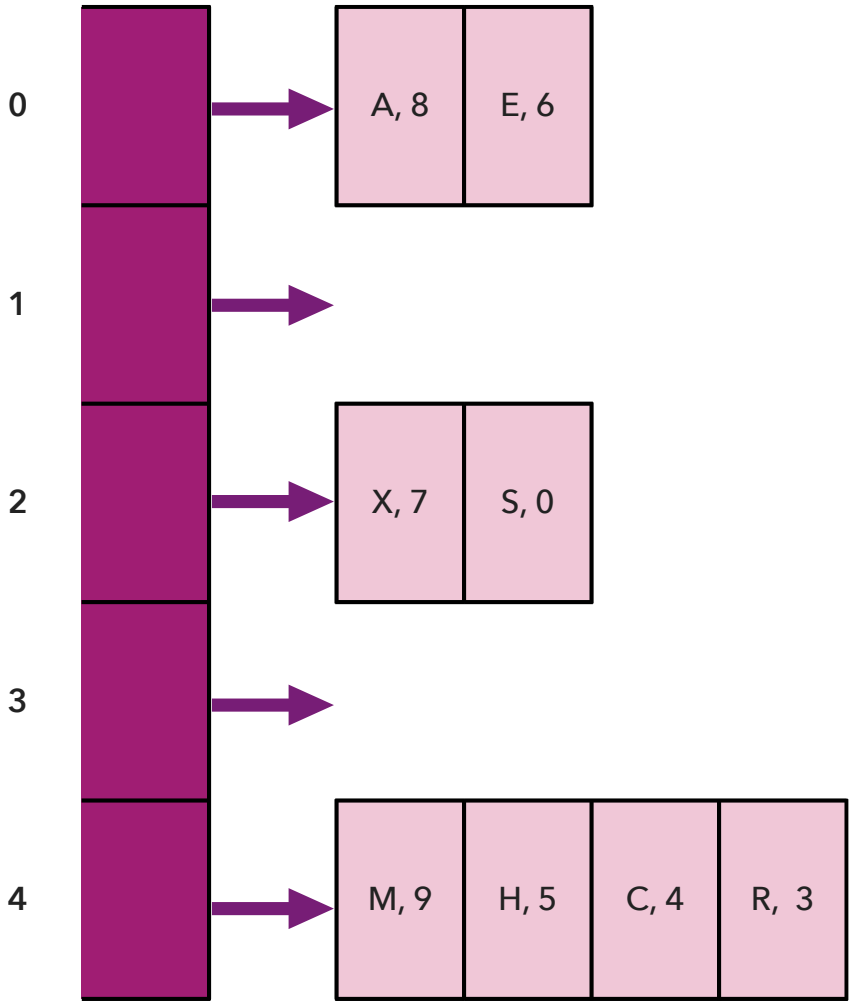
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8



Next step: Insert (M, 9)

Separate Chaining Example

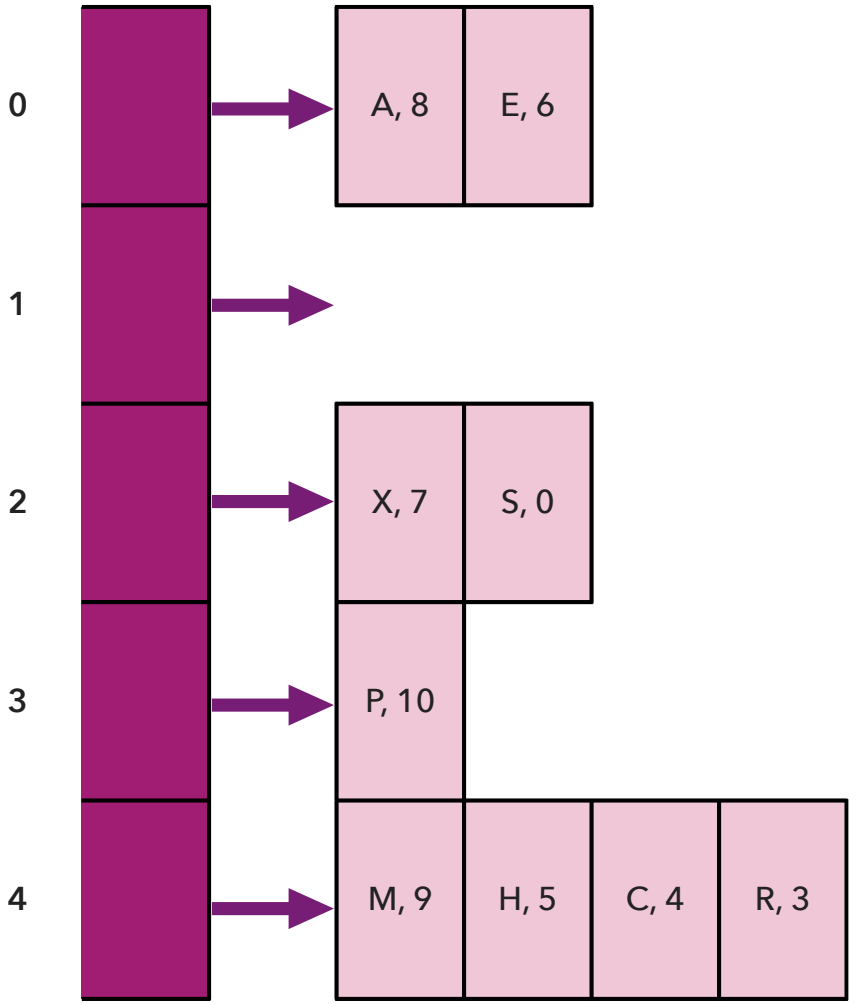
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9



Next step: Insert (P, 10)

Separate Chaining Example

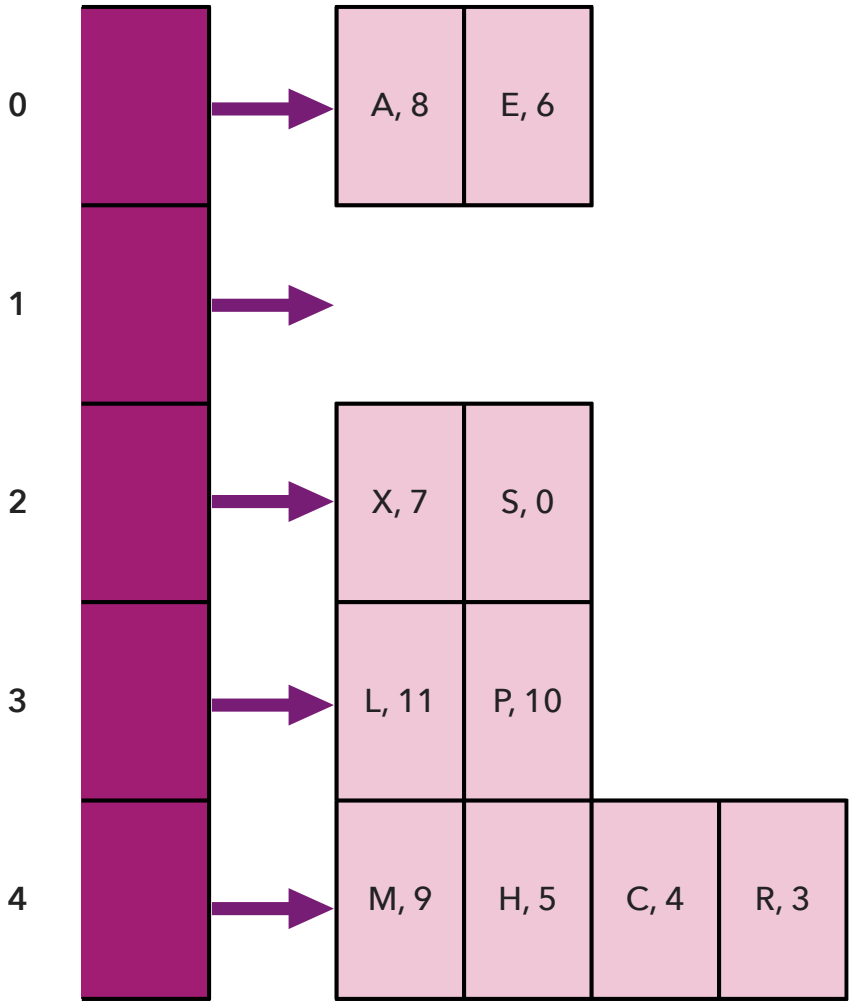
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10



Next step: Insert (L, 11)

Separate Chaining Example

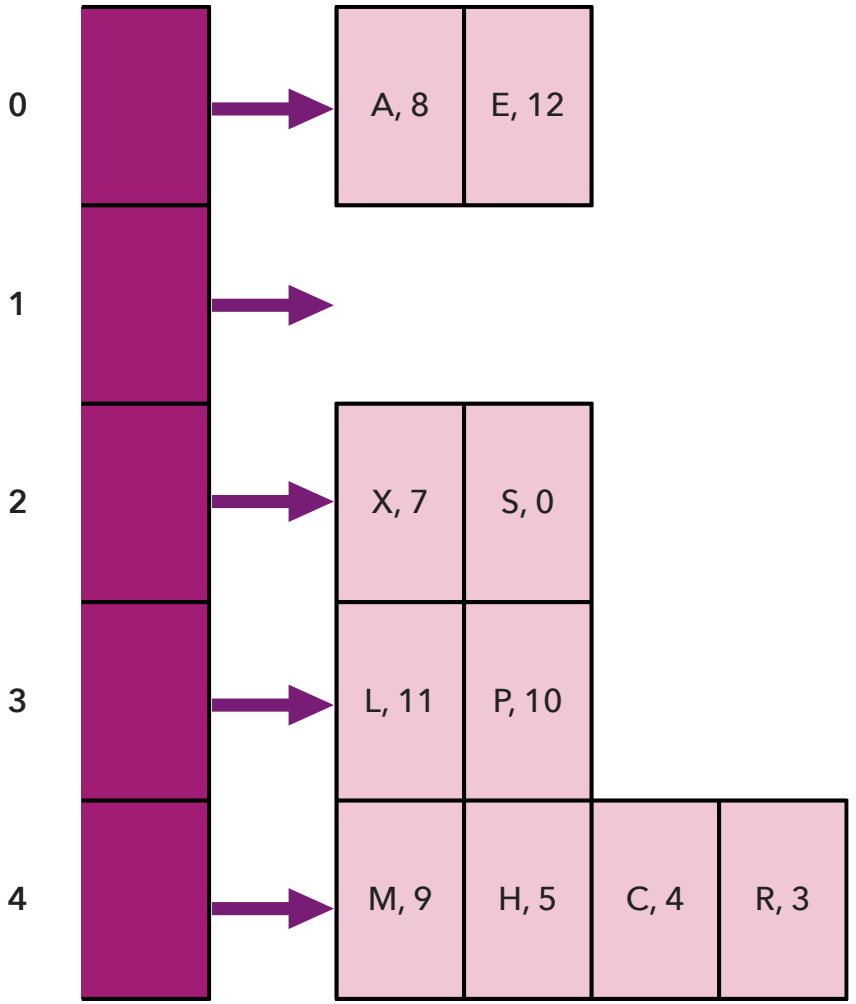
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11



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

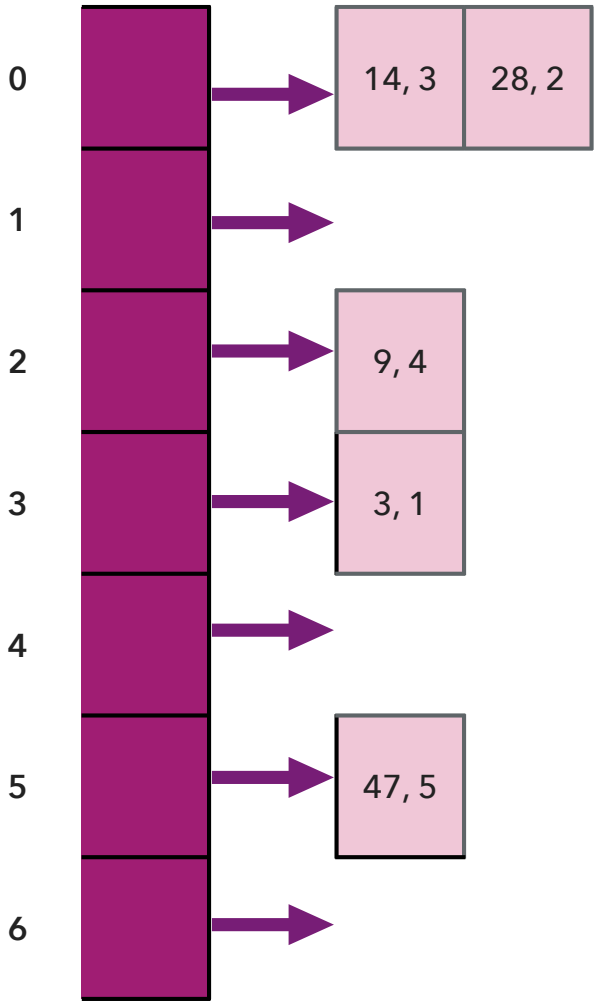


Practice Time

- 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: $m \sim 1/5n$ -> 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

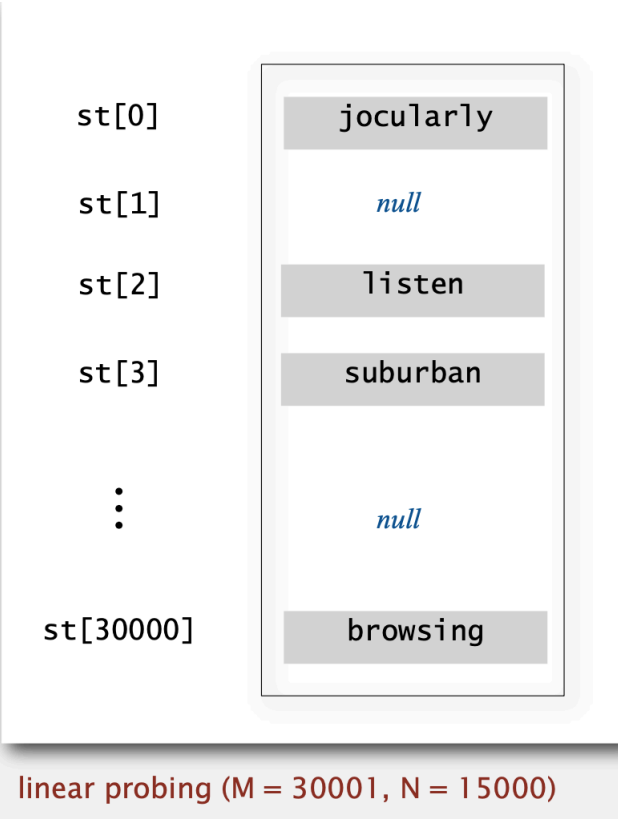
- Deletion: Easy! Hash key, find its chain, search for a node that contains it and remove it.
- Ordered operations: not supported. Instead, look into (balanced) BSTs.
- Fastest and most widely used dictionary implementation for applications where **key order** is not important.

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)

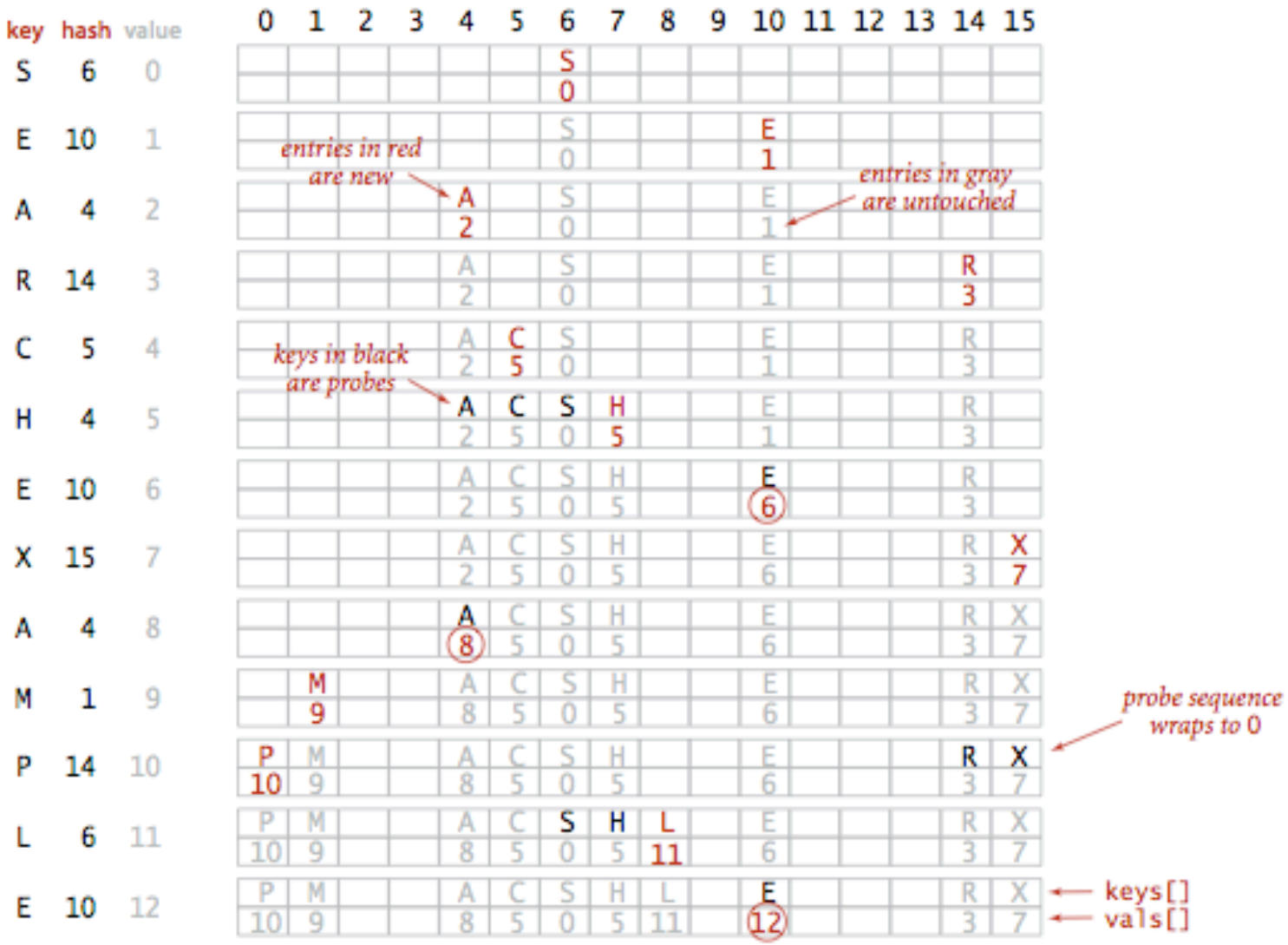
Linear Probing

- 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> {

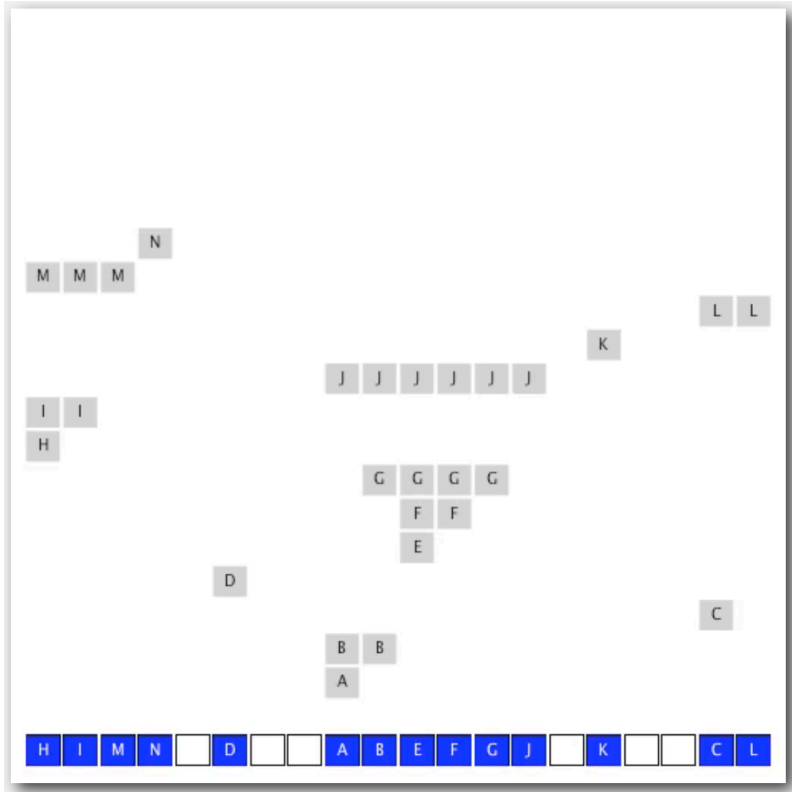
    private int m = 32768;  // hash table size
    private Value[] Vals = (Value[]) new Object[m]; // parallel arrays
    private Key[] Vals = (Key[]) new Object[m];

    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 keys where $n = \alpha m$ 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], α < 1 what happens when alpha is 1/2? Close to 1 say 0.9?
- 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 reduces 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