CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

22: HashTables, Undirected Graphs

Alexandra Papoutsaki
she/her/hers

Tom Yeh
he/him/his

Assignment 6 and 7 due today

Lecture 22: Hash tables

- Hash functions
- Separate chaining
- Open addressing

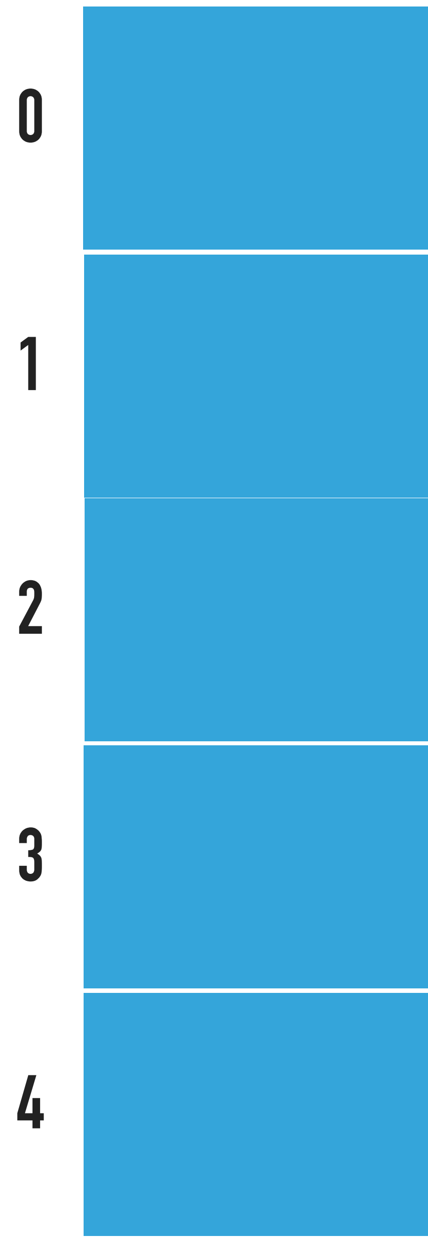
Some slides adopted from Algorithms 4th Edition or COS226

Summary for symbol table operations - can we do better?

	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	$2 \log n$	$2 \log n$	$2 \log n$	$1 \log n$	$1 \log n$	$1 \log n$

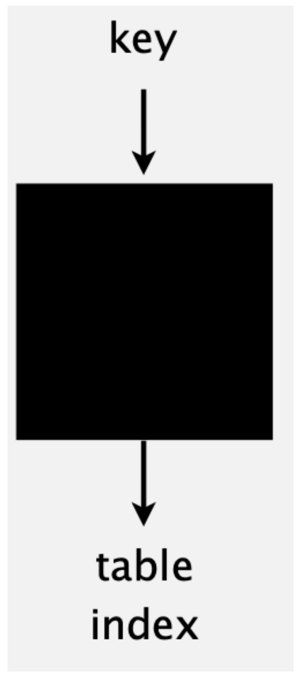
Dictionaries with O(1) search using hashing

- Goal: Build a key-indexed array to model dictionaries for efficient ($O(1)$) search.
  - Index is a function of the key
- Hash function: Method for computing array index from key.
  - 2 steps: hash code, hash value (index)
- Issues:
  - Computing the hash function.
  - Equality test: checking whether two keys are equal.
  - Collision resolution Trade off between time and space
- Space-time tradeoff: key as unique index vs all items to 1 index



Computing hash function

- Ideal scenario: Scramble the keys uniformly to produce a 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.
    - Problematic in practical applications.
- Examples: Dictionary where keys are social security numbers.
  - Keys are phone numbers, names, schools
- Practical challenge: Need different approach for each key type.



Hashing in Java

- All Java classes inherit a method `hashCode()`, which returns an 32-bit integer.
- Requirement: If `x.equals(y)` then `x.hashCode()==y.hashCode()`.
- Ideally: If `!x.equals(y)` then `x.hashCode()!=y.hashCode()`.
- Default implementation: Memory address of x.
  - Need to override both `equals()` and `hashCode()` for custom types.
  - Already customized for us for standard data types: `Integer`, `Double`, String.

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 `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 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.

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

Modular hashing

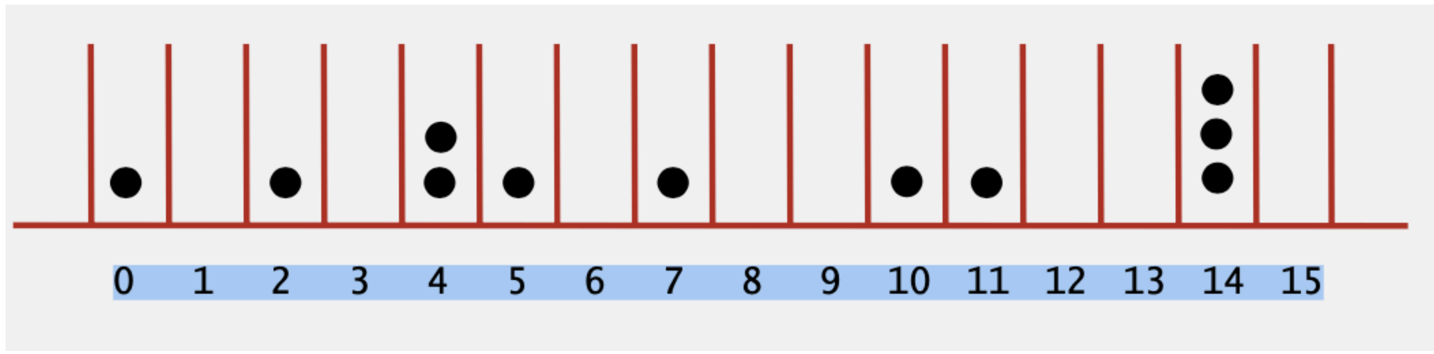
- Hash code: a 32-bit int can be negative (between -2^{31} and $2^{31}-1$)
- Hash function: need an int for the index (between 0 and $m-1$), where m is the hash table size
- The class that implements the dictionary of size m should implement a hash function. Examples:

  -
    ```
    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;
    }
    ```
    - Close

  -
    ```
    private int hash (Key key){
        return (key.hashCode() & 0x7fffffff) % m; // make hashcode positive
    }
    ```
    -

Uniform hashing assumption



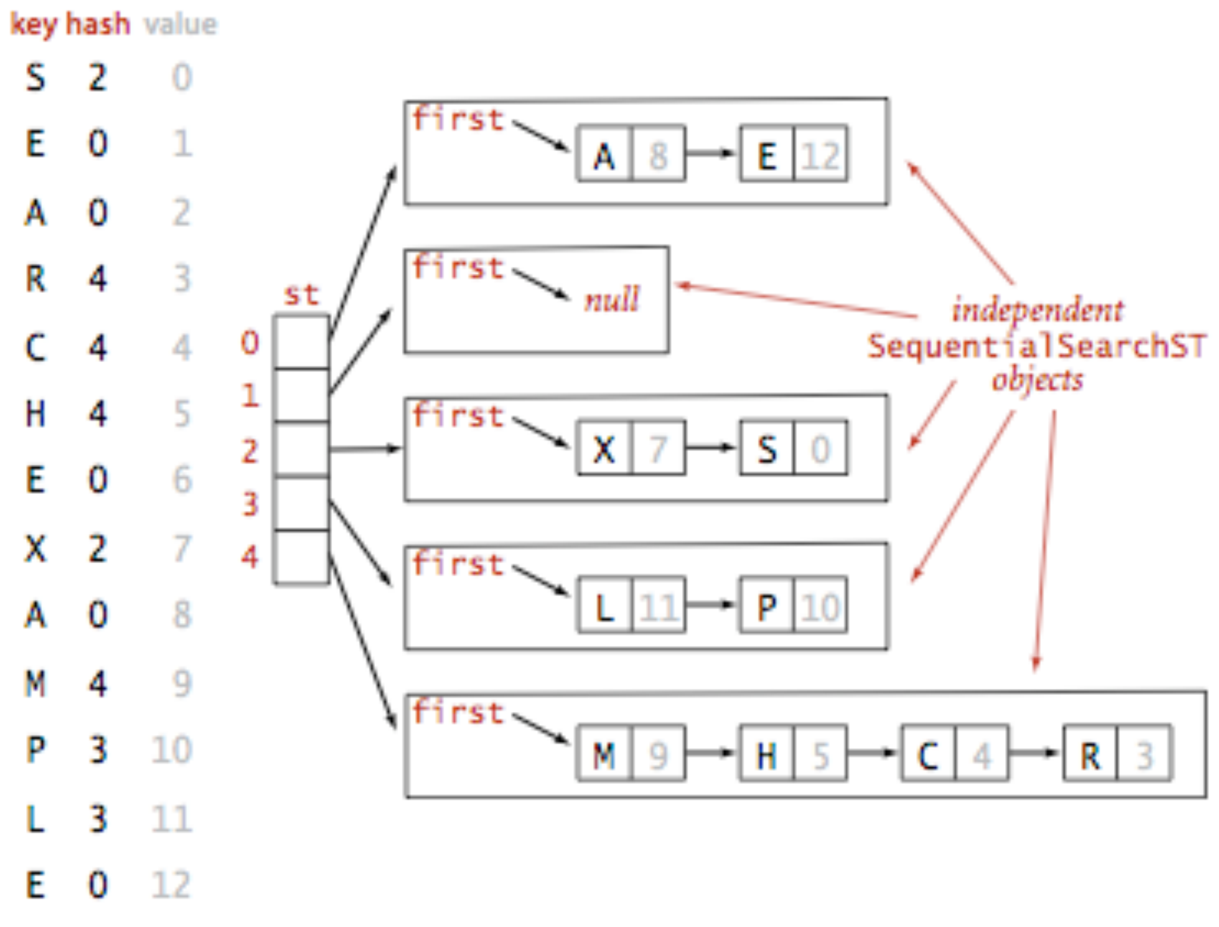
- Uniform hashing assumption: Each key is equally likely to hash to an integer between 0 and $m - 1$.
- Mathematical model: balls & bins. Toss n balls uniformly at random into m bins.
- Good news: load balancing
  - When $n >> m$, the number of balls in each bin is "likely close" to n/m.
  - Need to handle collisions

Lecture 26-27: Hash tables

- Hash functions
- Separate chaining
- Open addressing

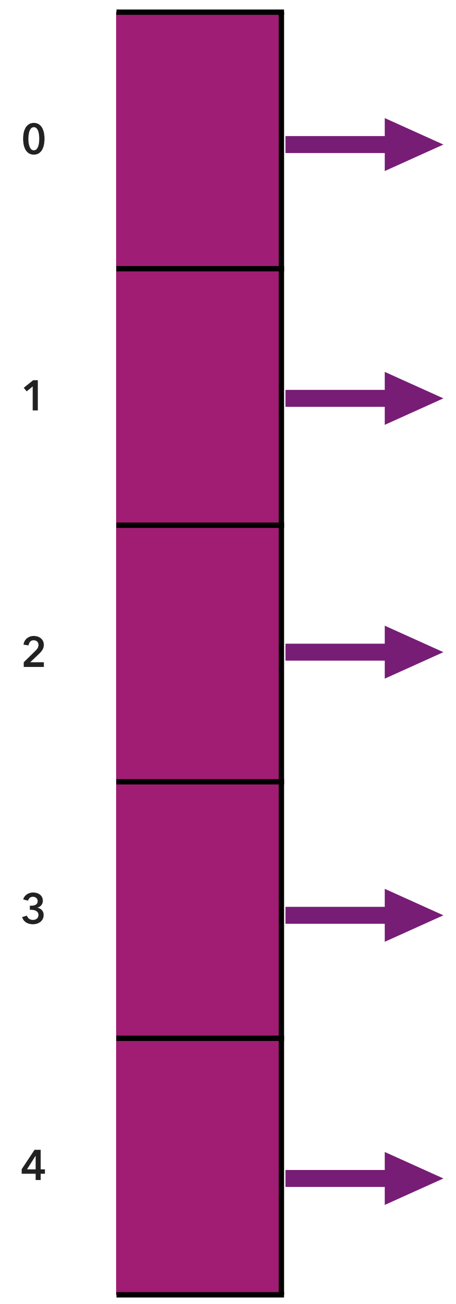
Separate/External Chaining (Closed Addressing)

- 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

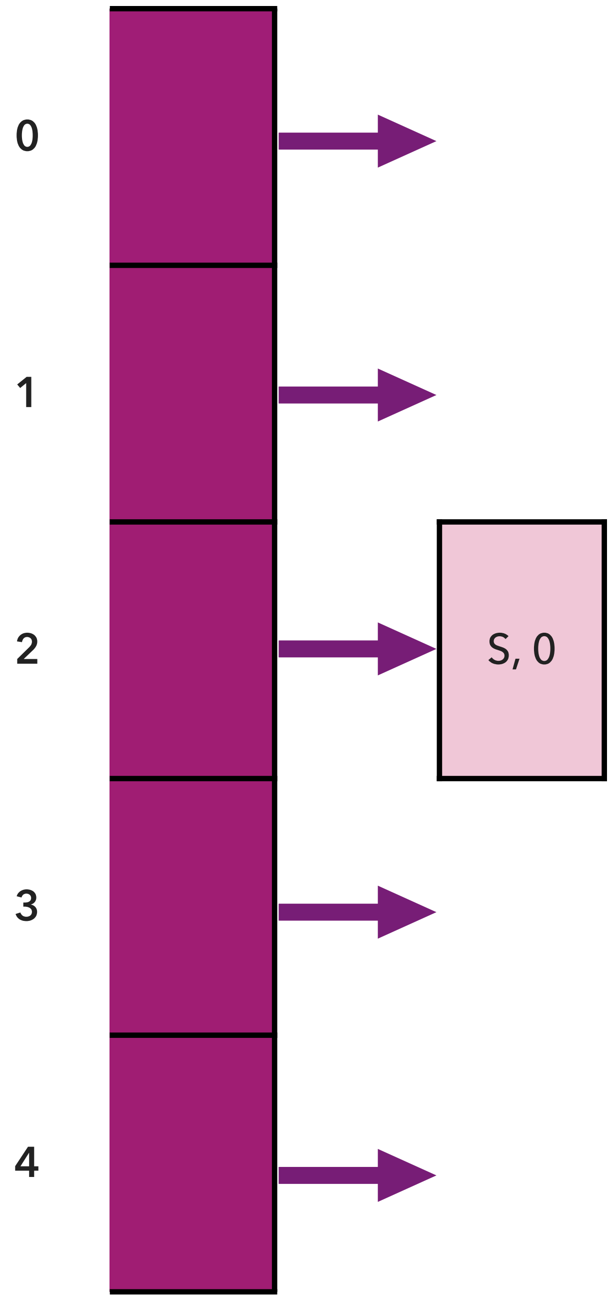
Separate Chaining Example



Next step: Insert (S, 0)

Separate Chaining Example

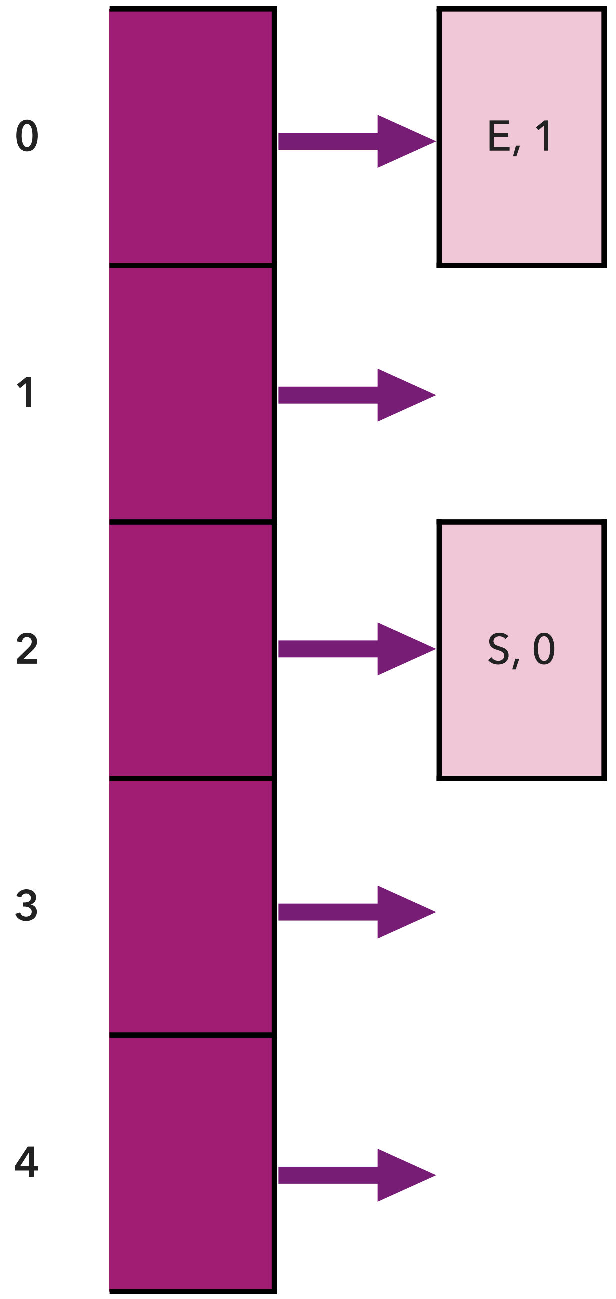
Key	Hash	Value
S	2	0



Next step: Insert (E, 1)

Separate Chaining Example

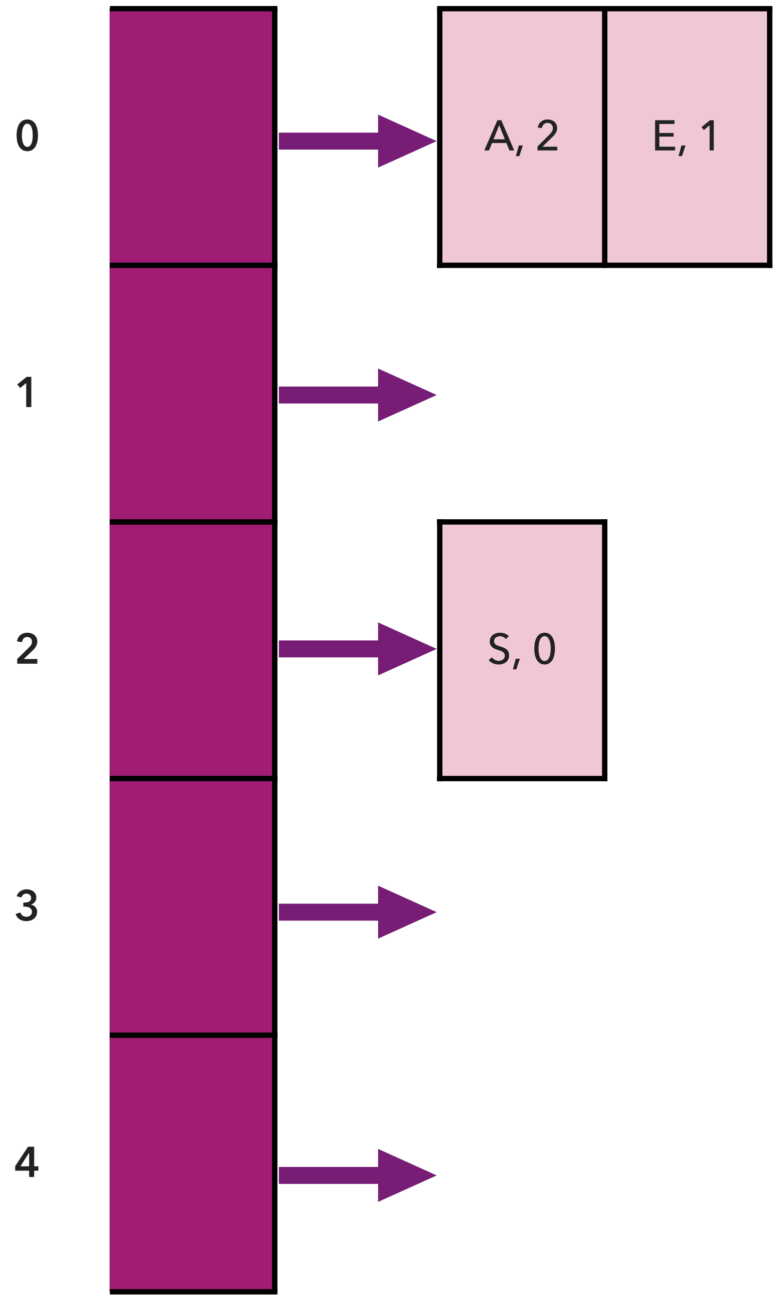
Key	Hash	Value
S	2	0
E	0	1



Next step: Insert (A, 2)

Separate Chaining Example

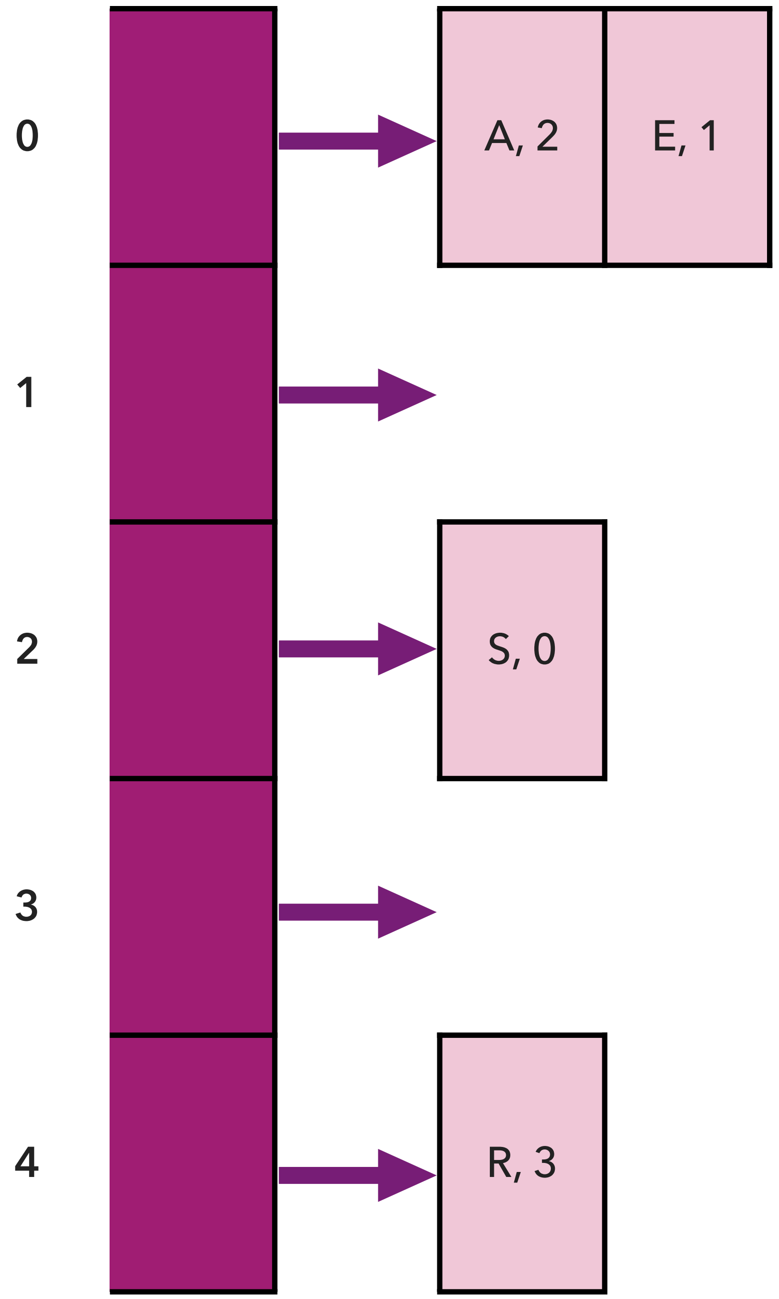
Key	Hash	Value
S	2	0
E	0	1
A	0	2



Next step: Insert (R, 3)

Separate Chaining Example

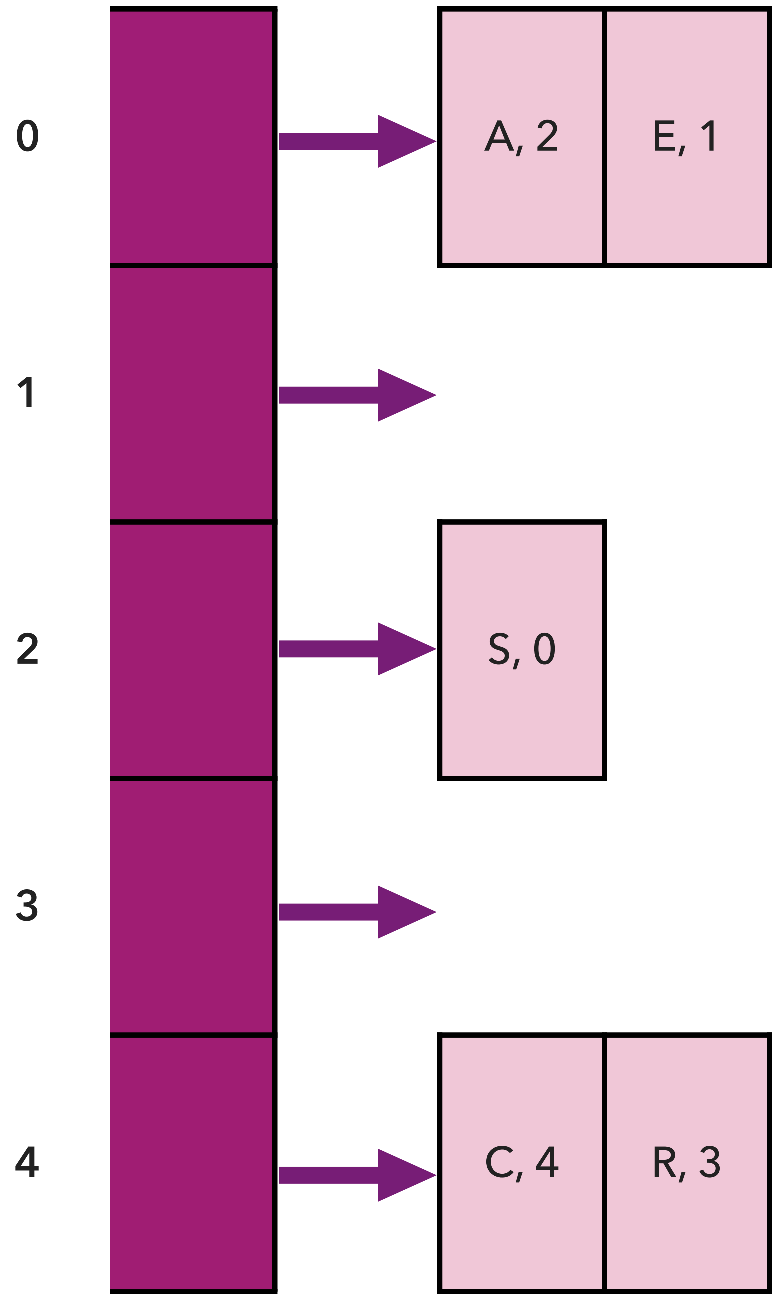
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3



Next step: Insert (C, 4)

Separate Chaining Example

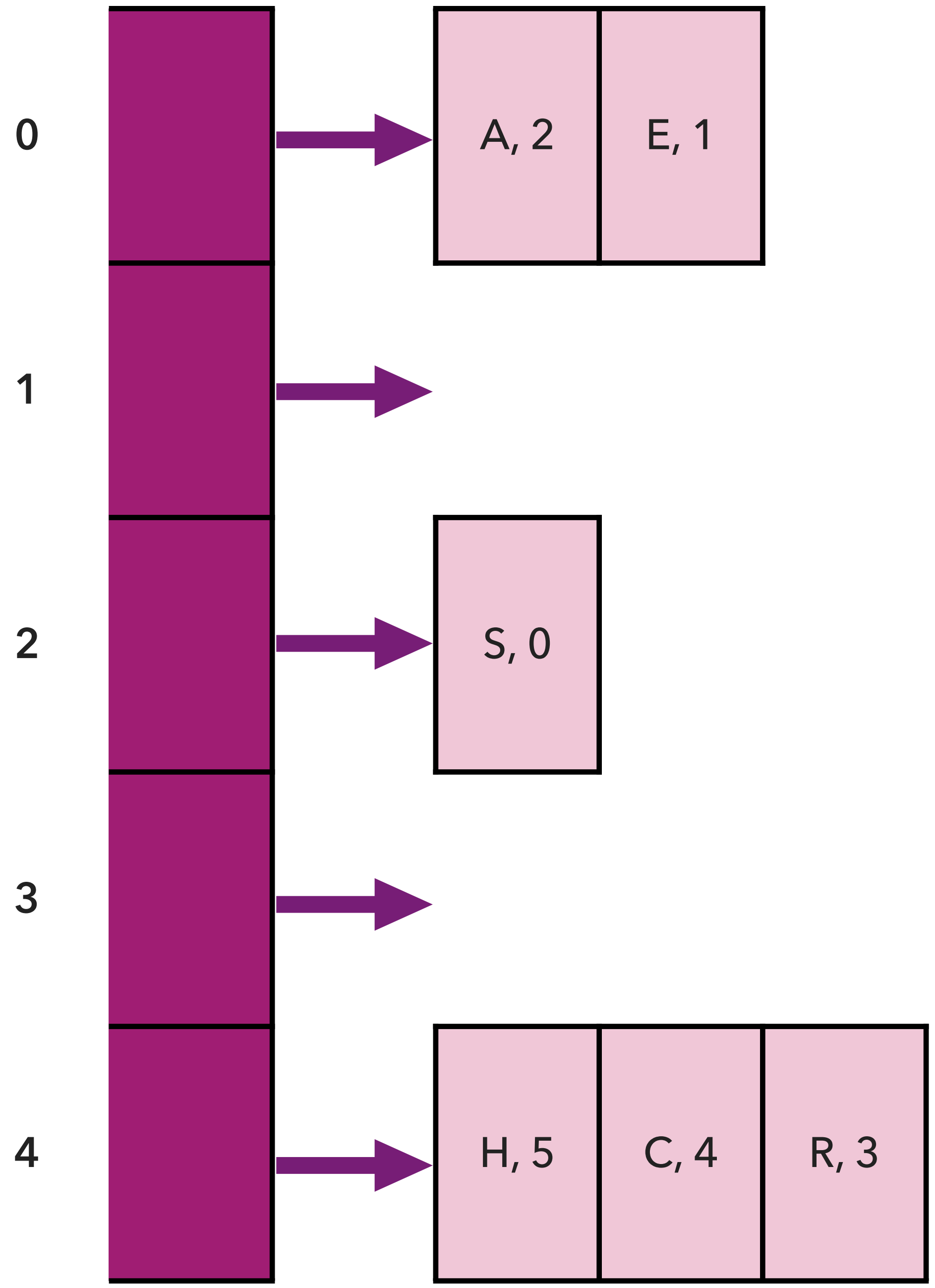
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4



Next step: Insert (H, 5)

Separate Chaining Example

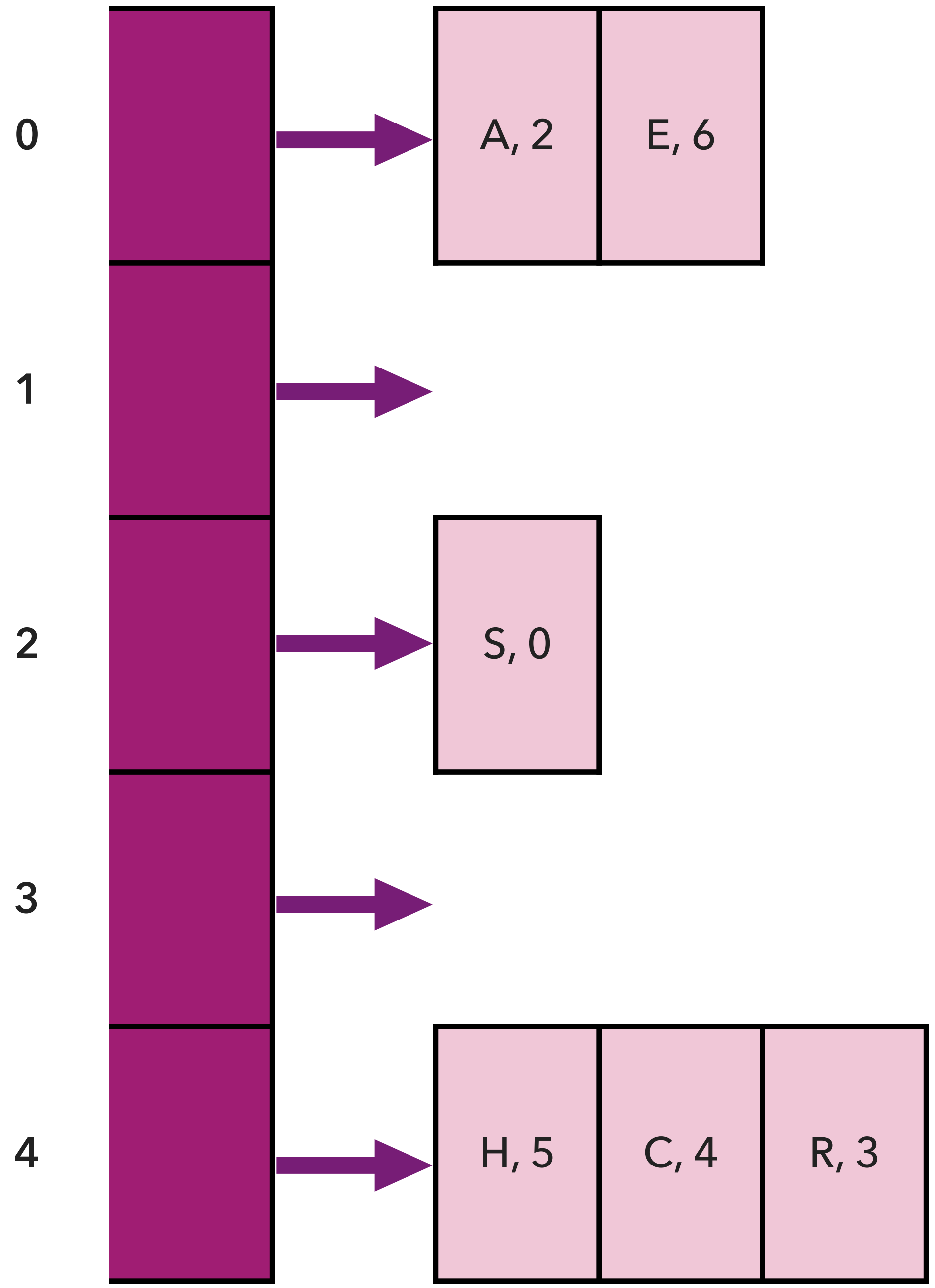
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5



Next step: Insert (E, 6)

Separate Chaining Example

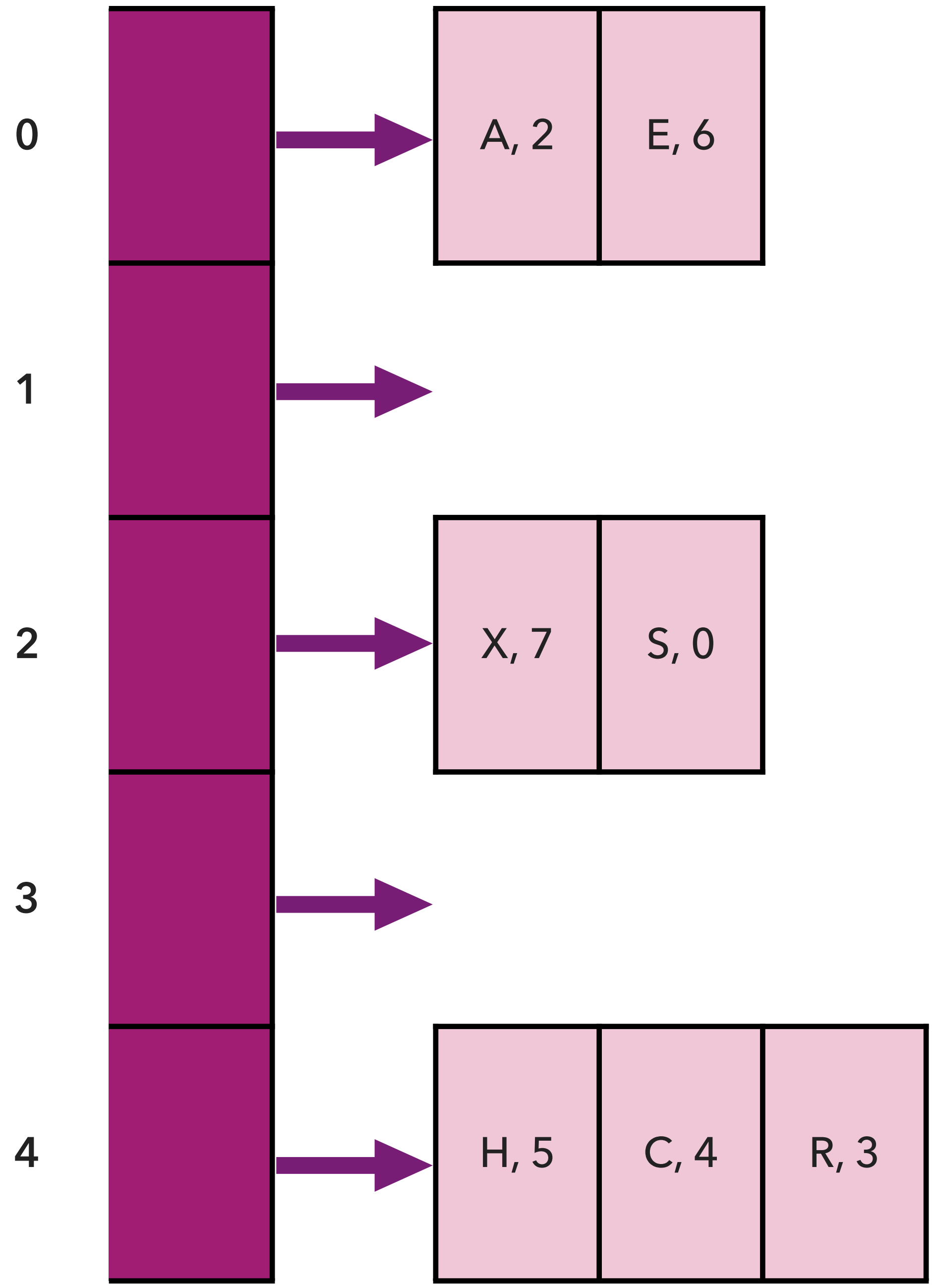
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6



Next step: Insert (X, 7)

Separate Chaining Example

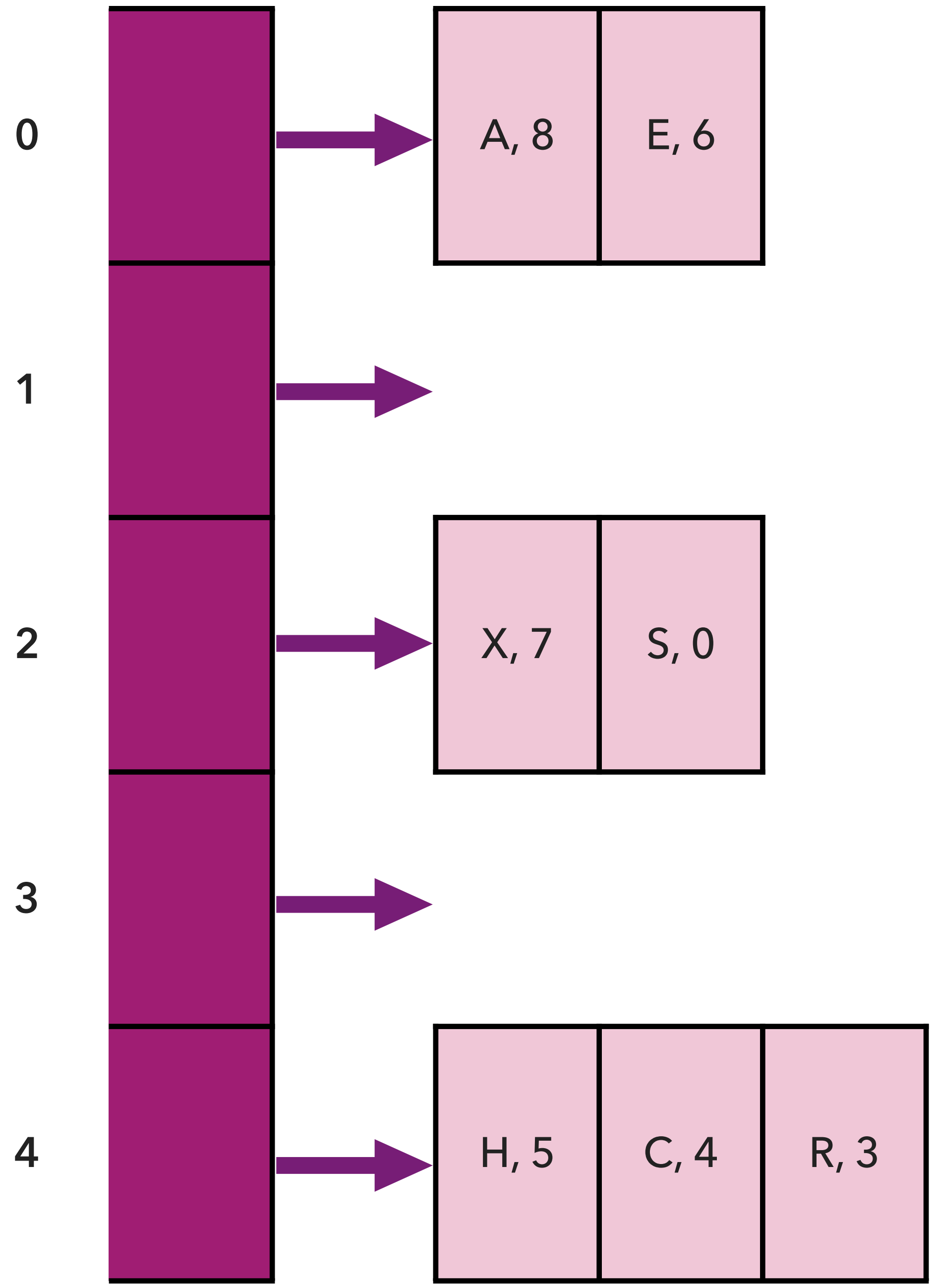
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7



Next step: Insert (A, 8)

Separate Chaining Example

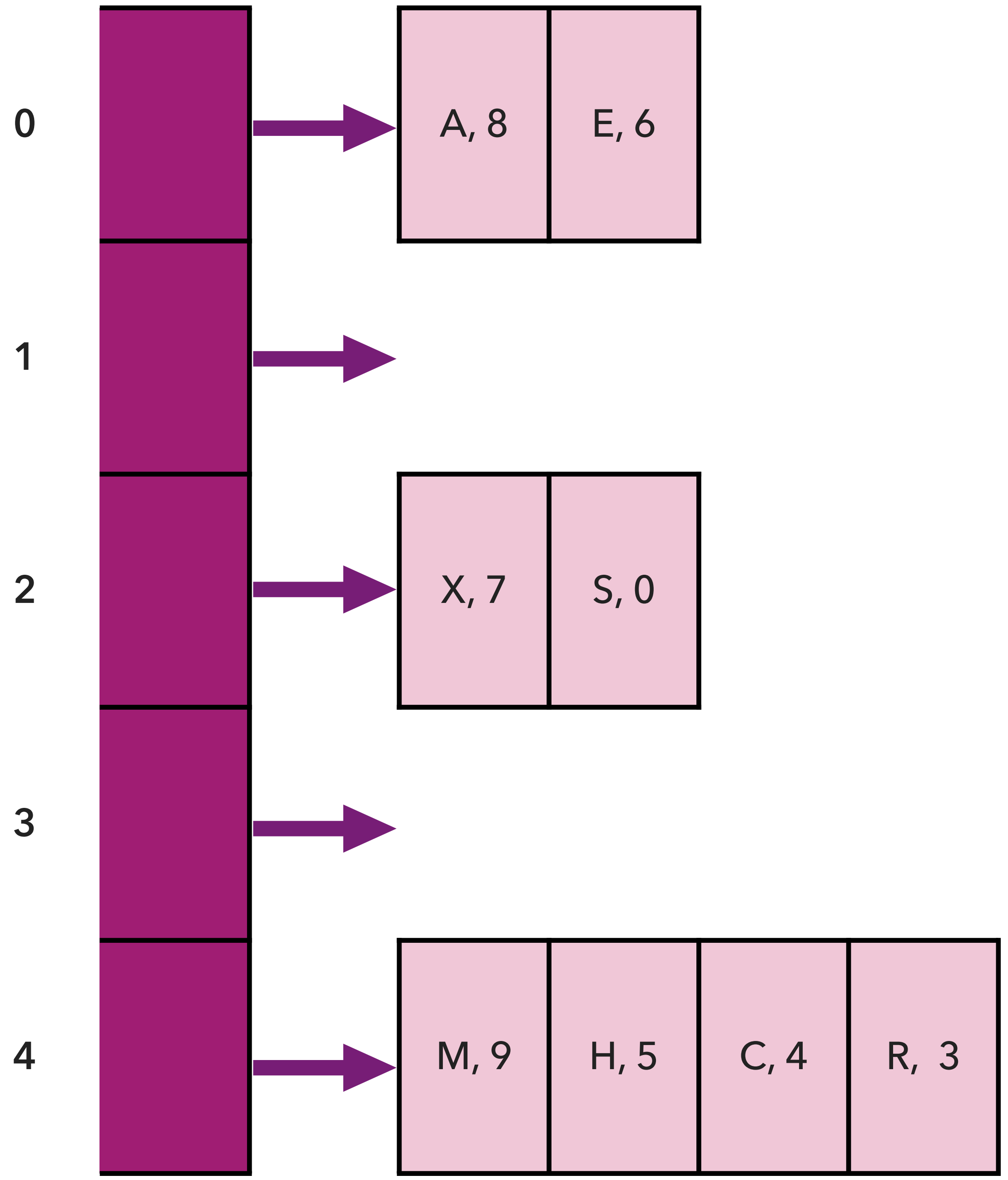
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8



Next step: Insert (M, 9)

Separate Chaining Example

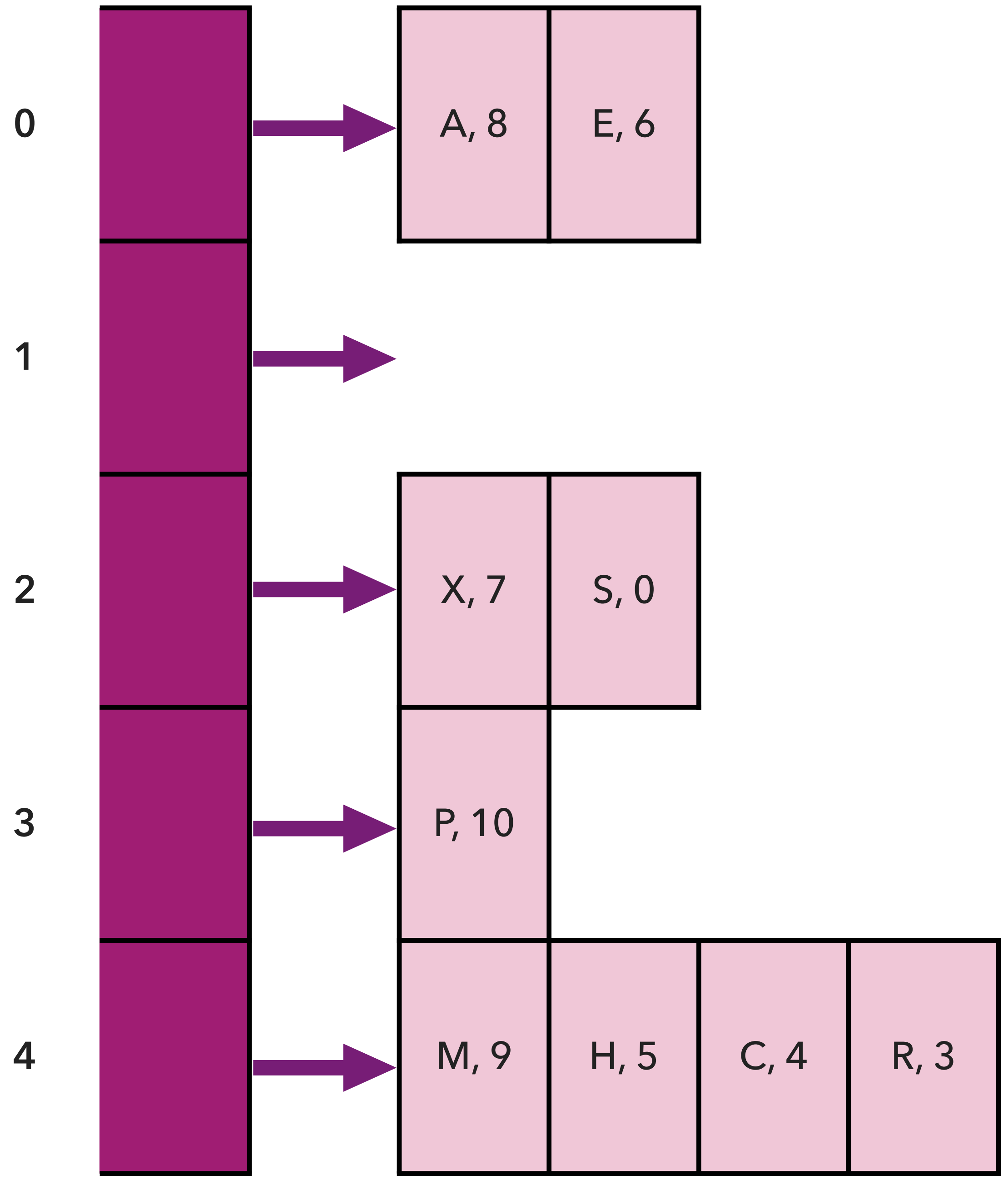
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9



Next step: Insert (P, 10)

Separate Chaining Example

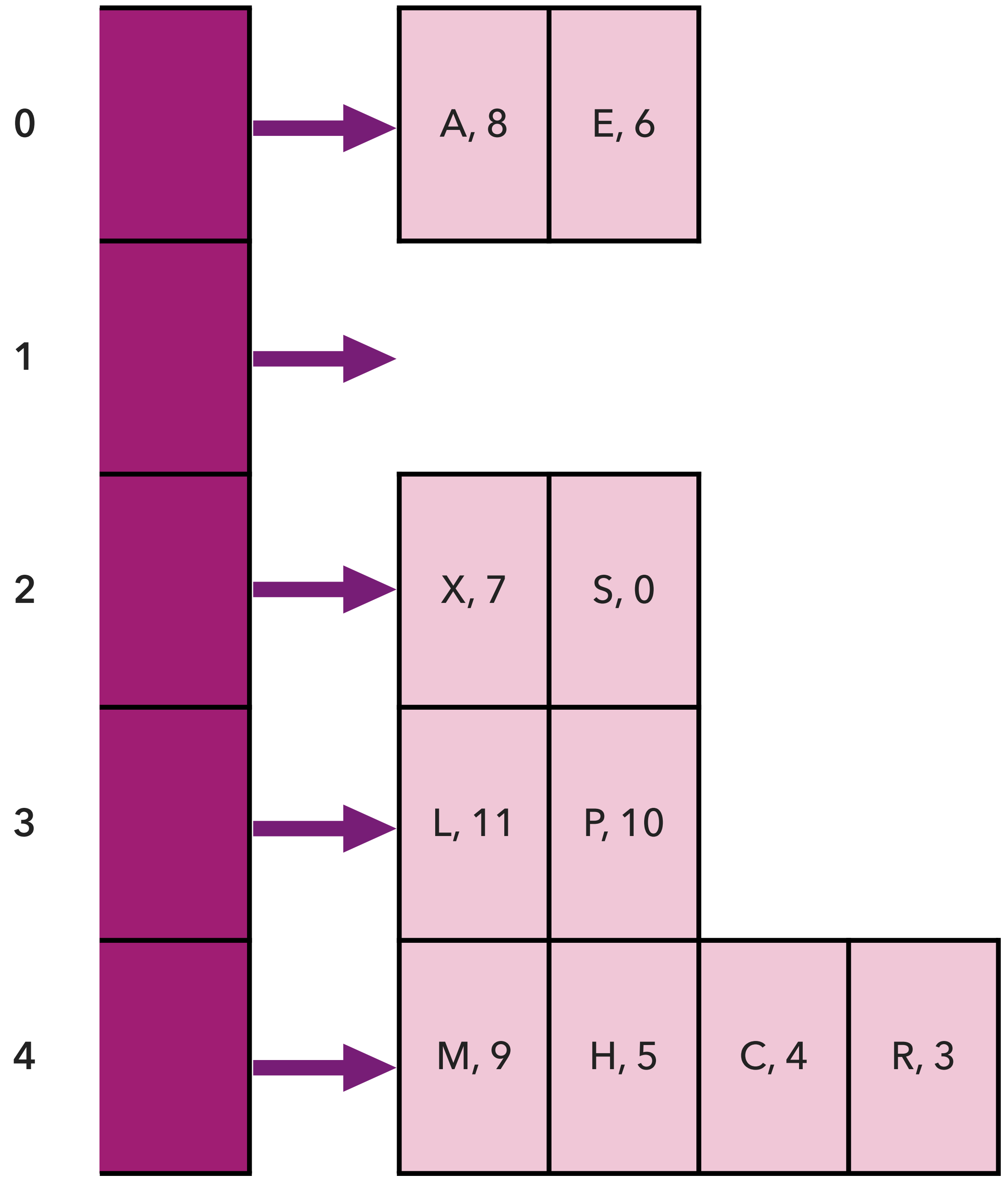
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10



Next step: Insert (L, 11)

Separate Chaining Example

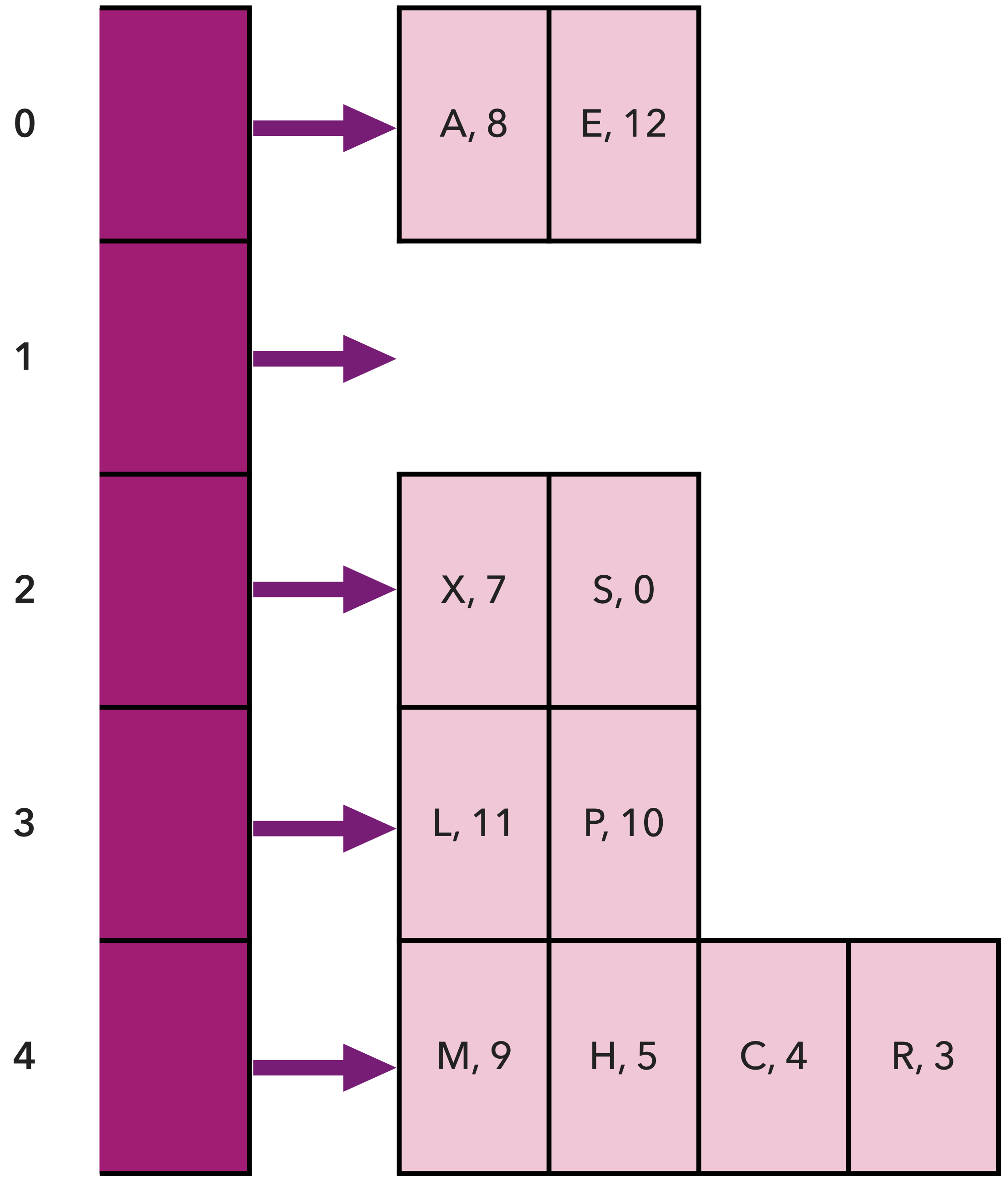
Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11



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

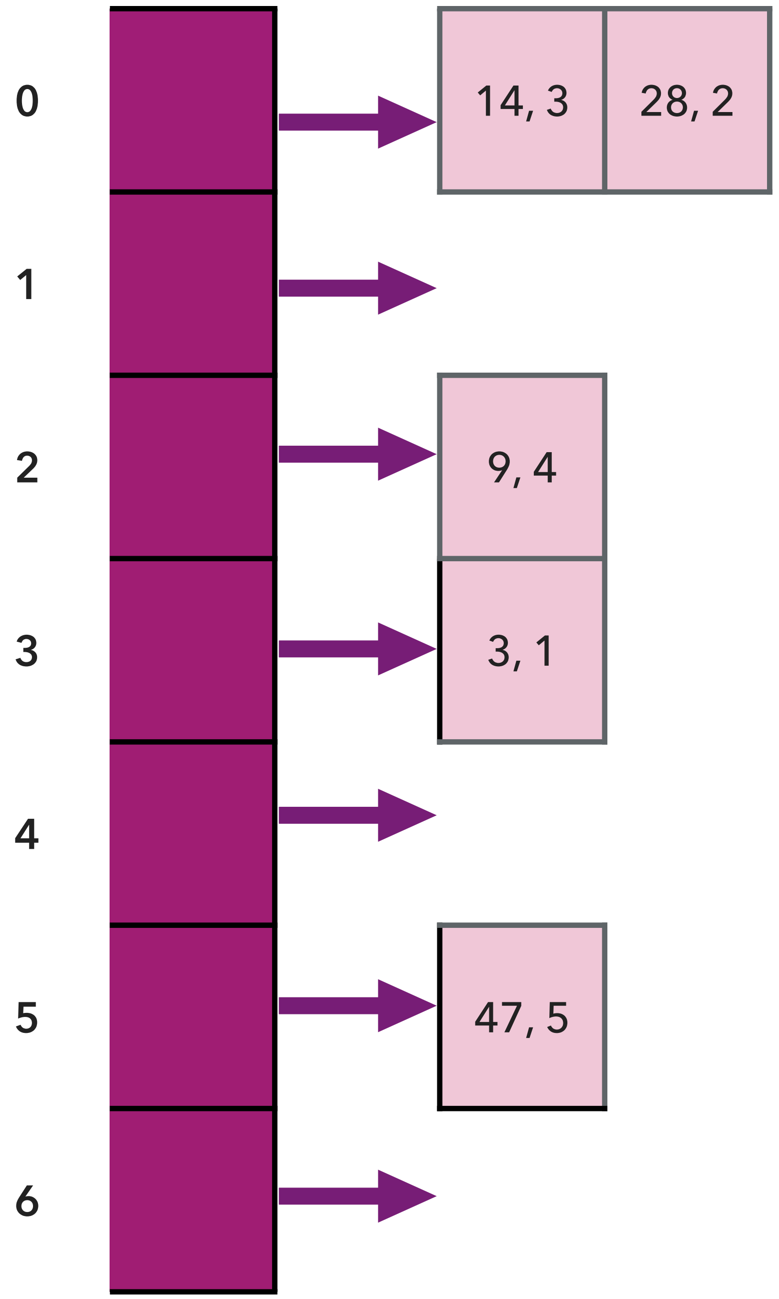


Practice Time

- 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, hash code is just the key, 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) {                               // hash is a private method to return hashcode
        int i = hash(key);                                    // compute hash value - bitwise & and mod
        for (Node x = st[i]; x != null; x = x.next) {          // traverse linked list - start at hash index i
            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)) {                            // start at hash index i
                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, with n keys and a table of size, the length of each chain is ~n/m.
- Consequence: Number of probes for search/insert is O(1) and proportional to n/m
  - 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). Look at example

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.

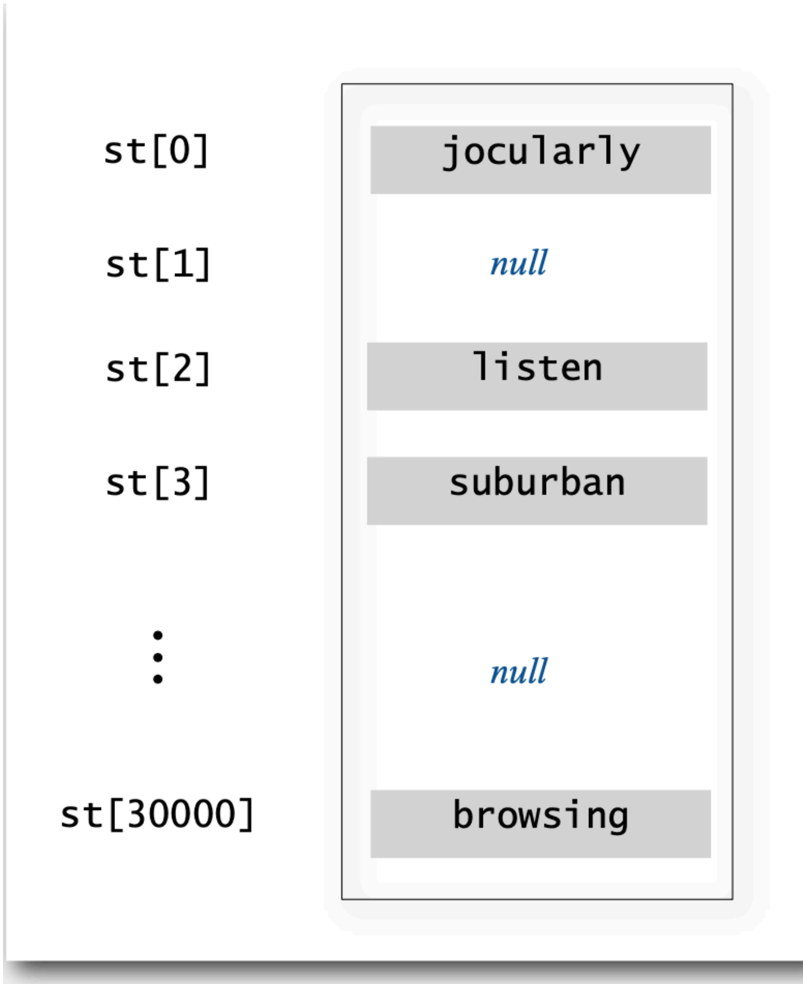
implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	compareTo()
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	compareTo()
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	equals() hashCode()

Lecture 26-27: Hash tables

- Hash functions
- Separate chaining
- Open addressing

Linear Probing

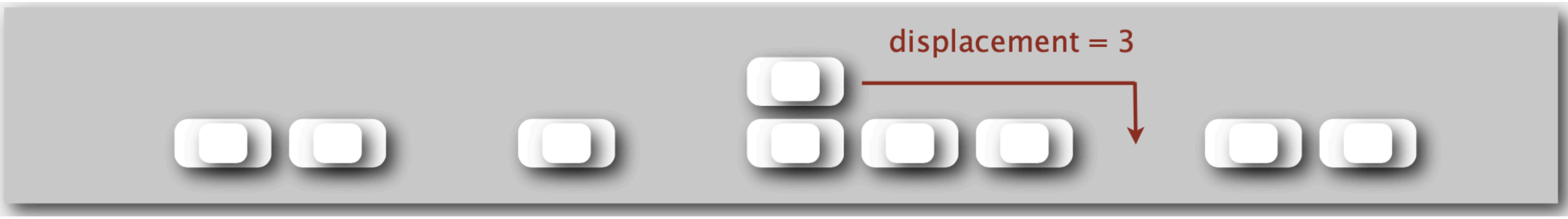
- Alternate collision resolution when table size > number of keys ($m > n$).
- Maintain keys and values in two parallel arrays.
- When a new key collides, find next empty slot and put it there.
- If the array is full, the search would not terminate.



linear probing (M = 30001, N = 15000)

Linear Probing

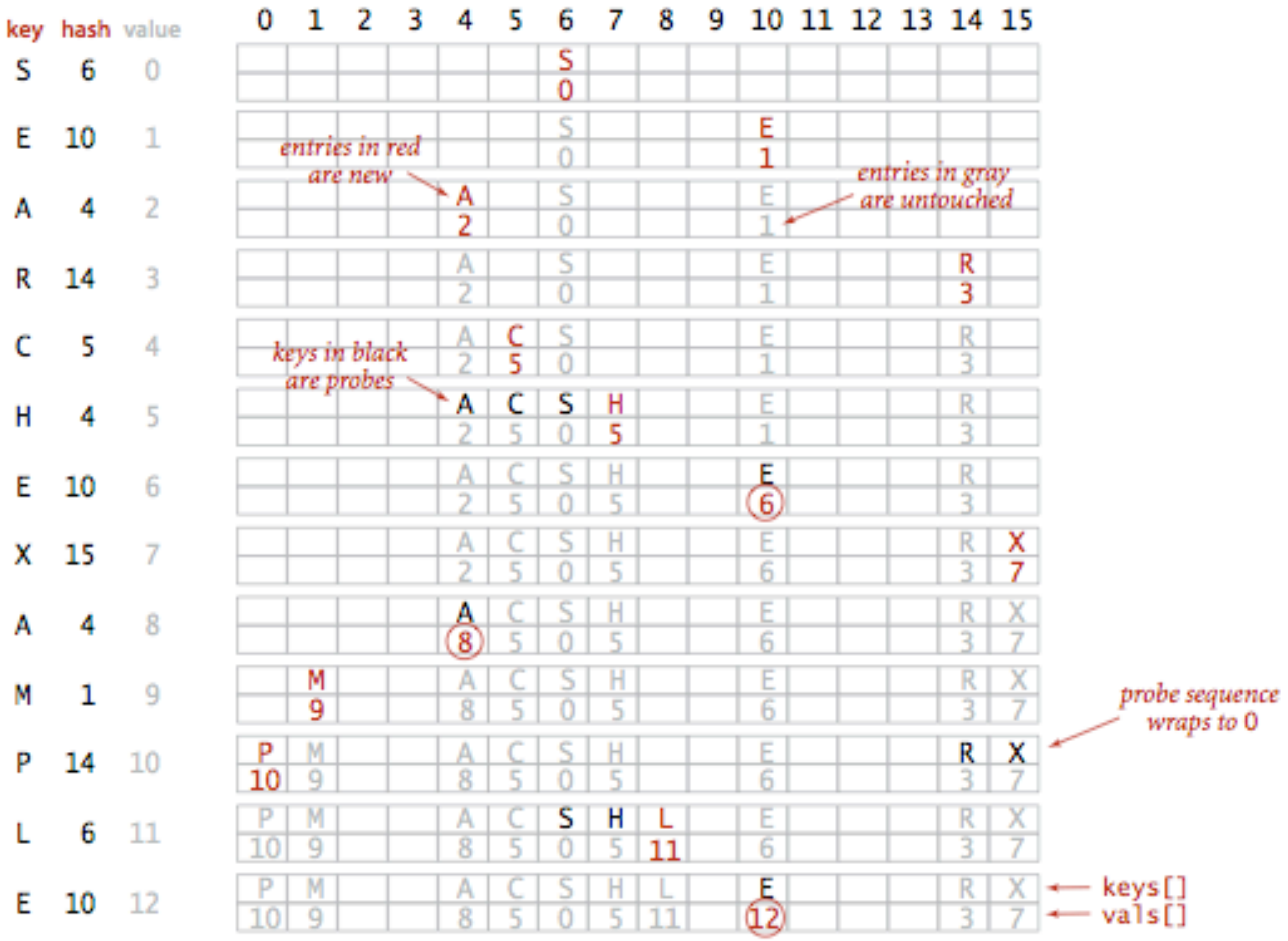
- Hash: Map key to integer i between 0 and $m - 1$.
- Insert: Put at index i if free. If not, try $i + 1$, $i + 2$, etc.
- Search: Search table index i. If occupied but no match, try $i + 1$, $i + 2$, etc
  - If you find a gap then you know that it does not exist.
- Table size m **must** be greater than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

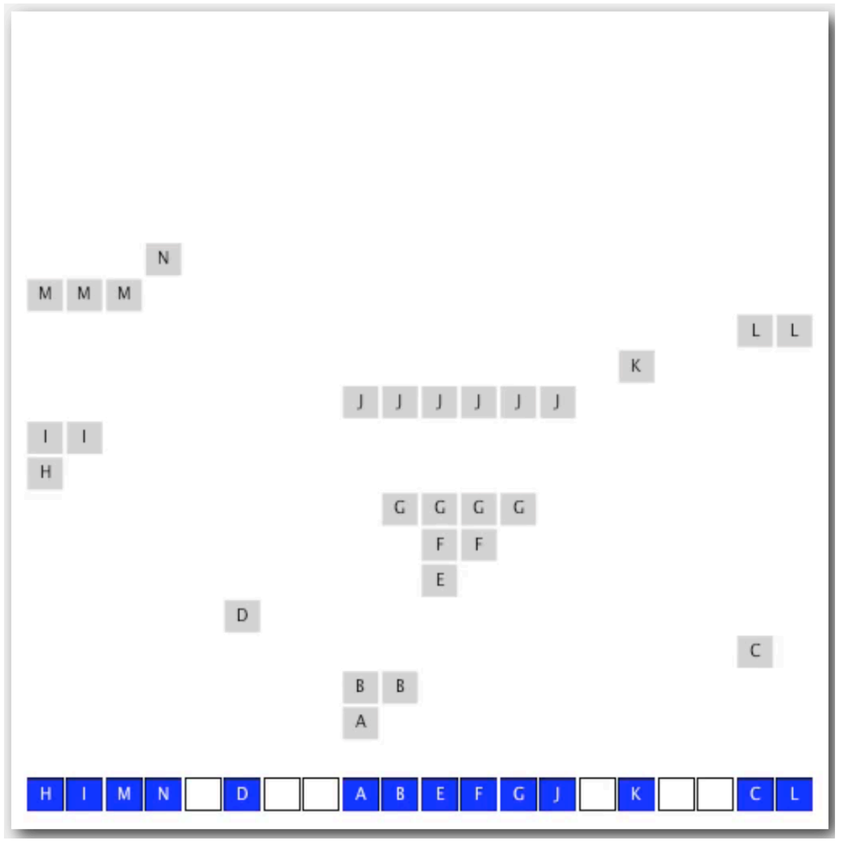
Keys	28	14	9	3		47	
Values	2	3	2	1		5	
Indices	0	1	2	3	4	5	6

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.
  - What happens to time complexity for search and insert?



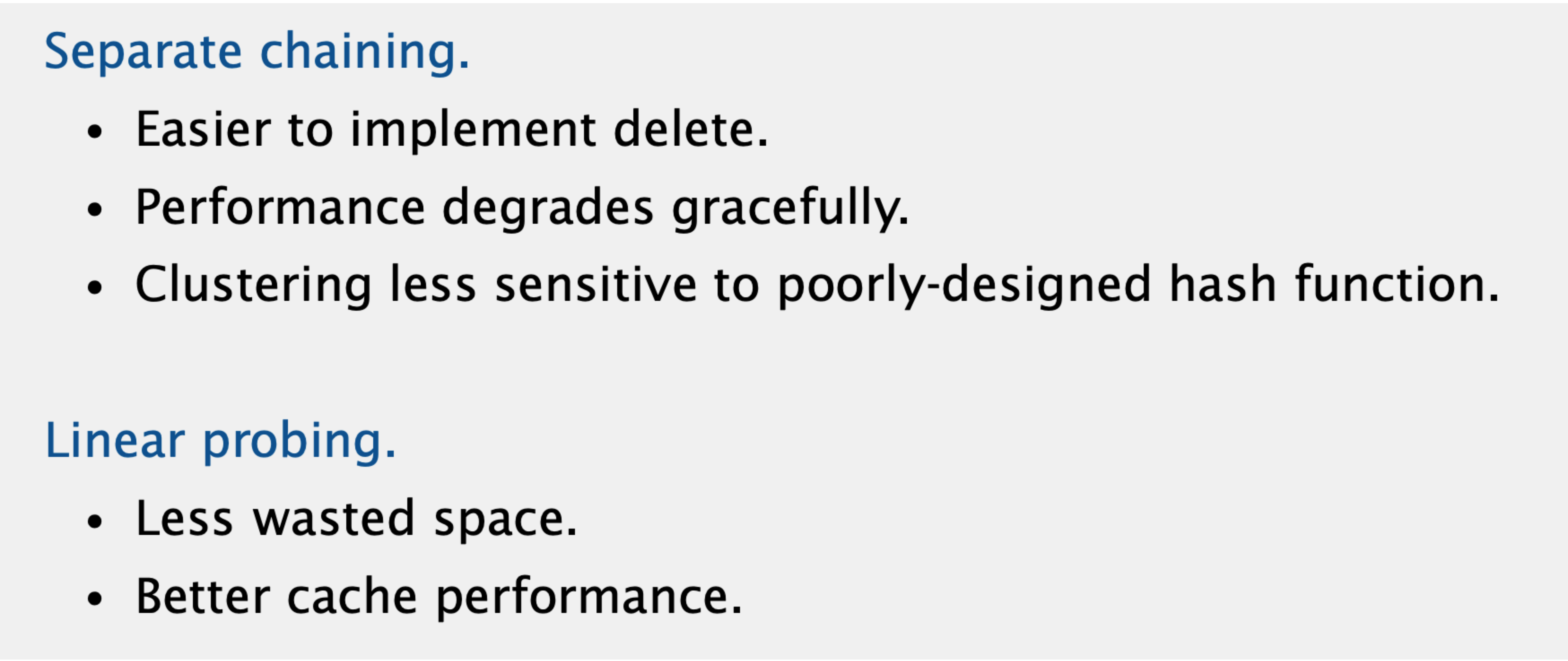
Analysis of Linear Probing

- 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 (alpha is the ratio of n/m)
  - $1/2(1 + \frac{1}{1-a})$ 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. Why?
    - Find, remove, reinsert all subsequent key-value pairs in cluster

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()` `hashCode()`
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()` `hashCode()`



Separate chaining.

- Easier to implement delete.
- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear probing.

- Less wasted space.
- Better cache performance.

- Liner probing - less wasted space if the hashing function is not very good
- Linear probing - better cache performance because of the use of an array instead of chasing pointers

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

Intro to Undirected Graphs

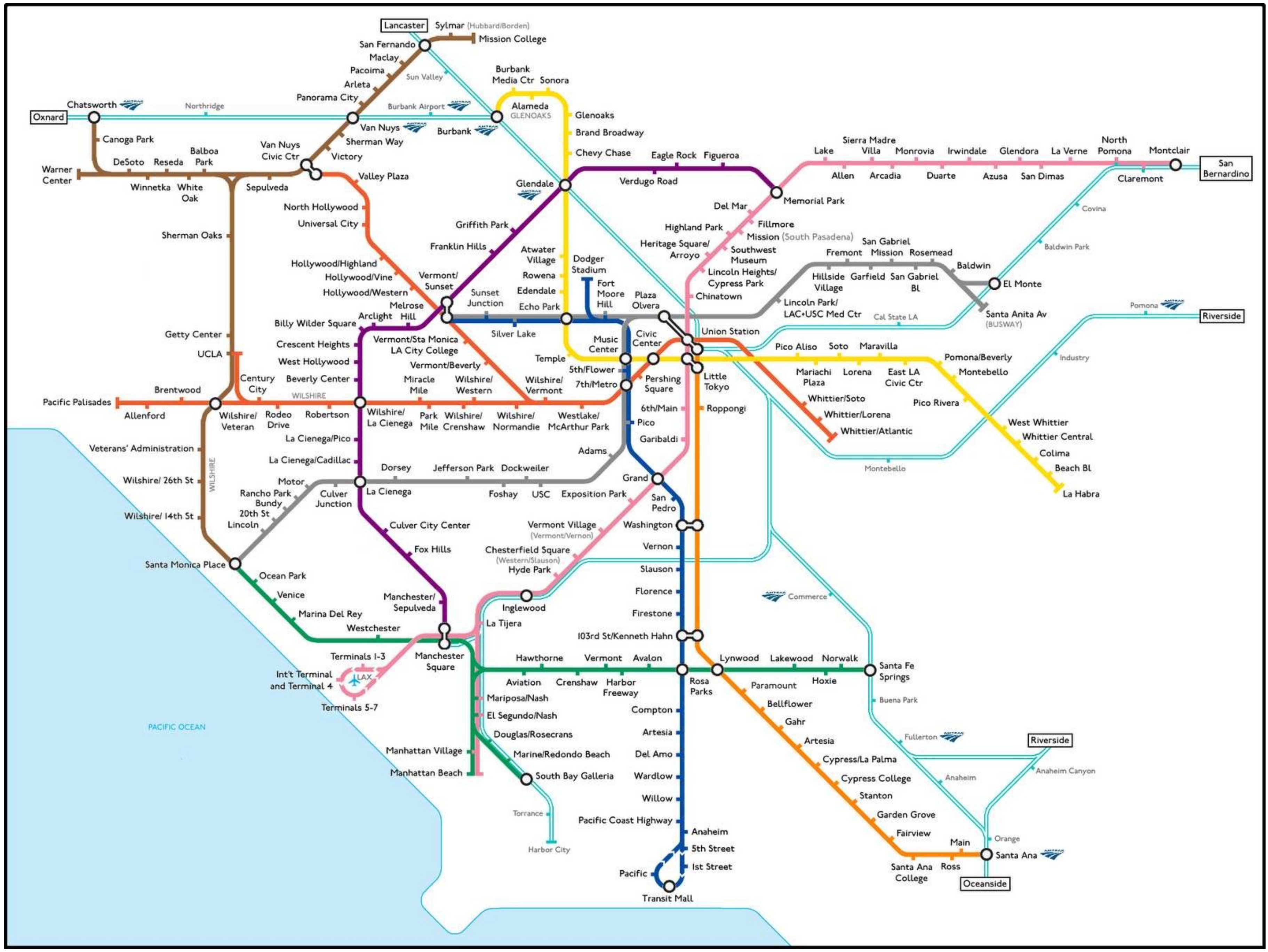
- Undirected Graphs

Graphs

- Graphs: mathematical abstractions that model a set of vertices connected pairwise by edges.
- Why study graphs?
  - Thousands of practical applications.
  - Hundreds of graph algorithms.
  - Interesting and widely applicable abstraction.
  - Core branch of computer science and discrete math.

Example: (Fake) LA subway map

- Vertices: stations. Edges: route.



- Source: LA Weekly

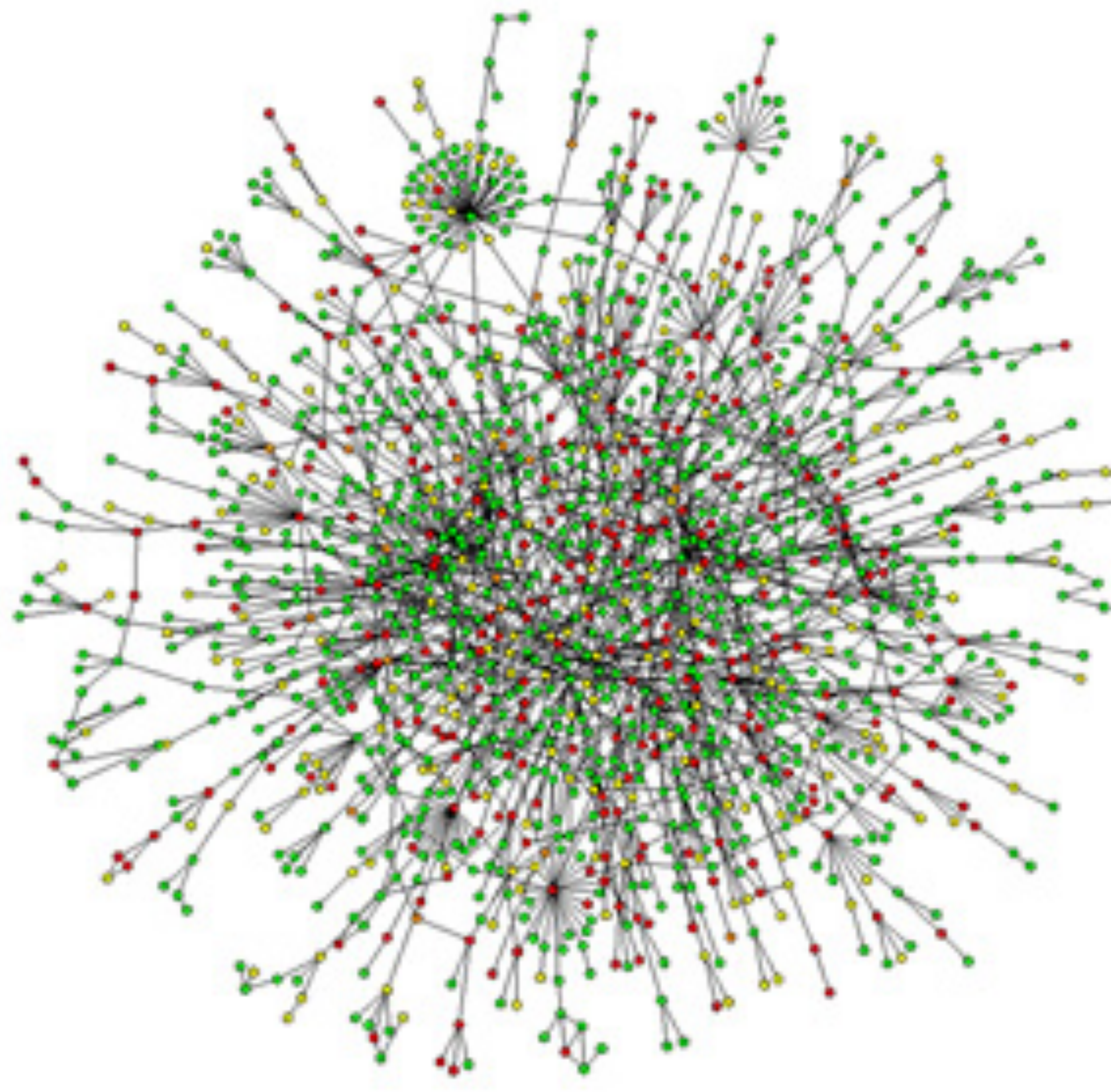
Example: Social networks

- Vertices: people. Edges: “friendships”.
  Source: Paul Butler

facebook
December 2010

Example: Protein-protein networks

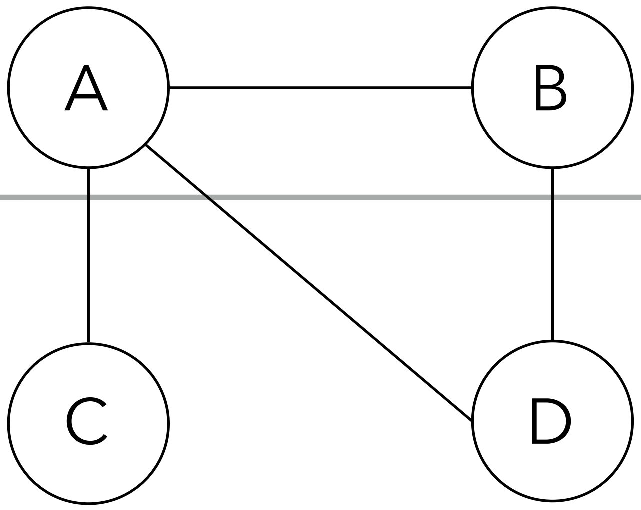
- Vertices: proteins.
- Edges: interactions.



- Source: Macmillan Magazines Ltd.

Graph Applications

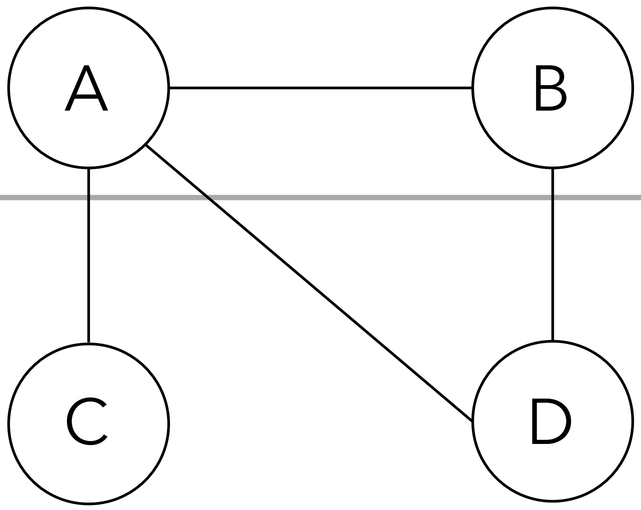
Graph	Vertex	Edge
Communication	Telephone, computer	Cable
Circuit	Gate, register, processor	Wire
Financial	Stock	Transaction
Transportation	Intersection	Street
Game	Board	Legal move
Neural network	Neuron	Synapse
Molecule	Atom	Bond
Schedule	Job	Constraint



Graph Terminology

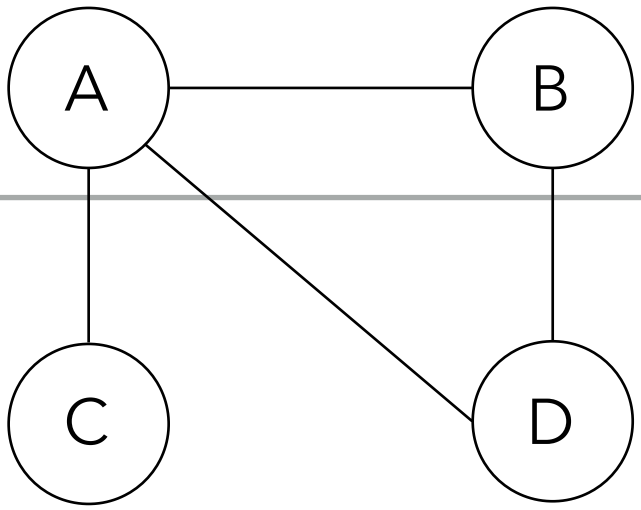
- Graph: set of vertices V connected pairwise by a set of edges E.
  - E.g., V = {A, B, C, D}, E = {{A,B}, {A,C}, {A,D}, {B,D}}.
- Path: sequence of vertices connected by edges, with no repeated edges.
  - A simple path is a path with no repeated vertices.
- Cycle: Path with at least one edge whose first and last vertices are the same.
  - A simple cycle is a cycle with no repeated vertices (other than the first and last).
- The length of a cycle or a path is its number of edges.

Graph Terminology



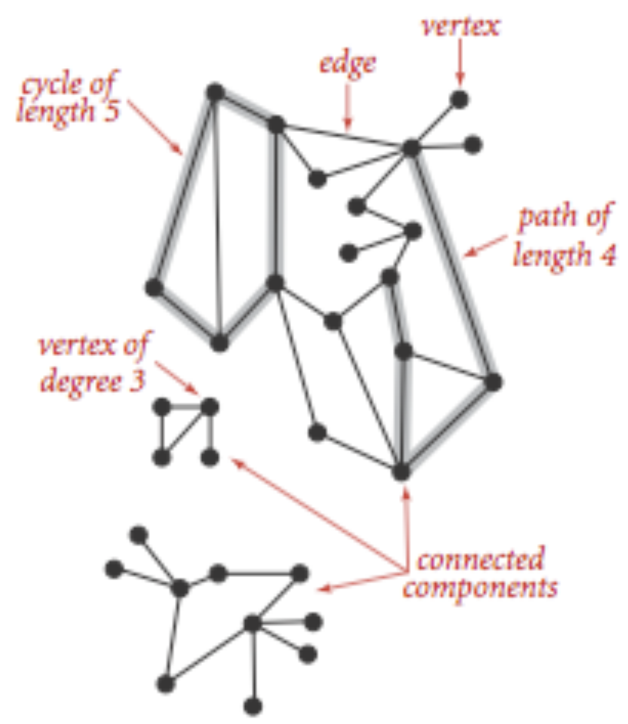
- Self-loop: an edge that connects a vertex to itself.
- Two vertices are connected if there is a path between them.
- Two edges are parallel if they connect the same pair of vertices.
- When an edge connects two vertices, we say that the vertices are adjacent to one another and that the edge is incident on both vertices.
- The degree of a vertex is the number of edges incident on it.
- A subgraph of a graph is a subset of a graph's edges and their associated vertices.

Graph Terminology

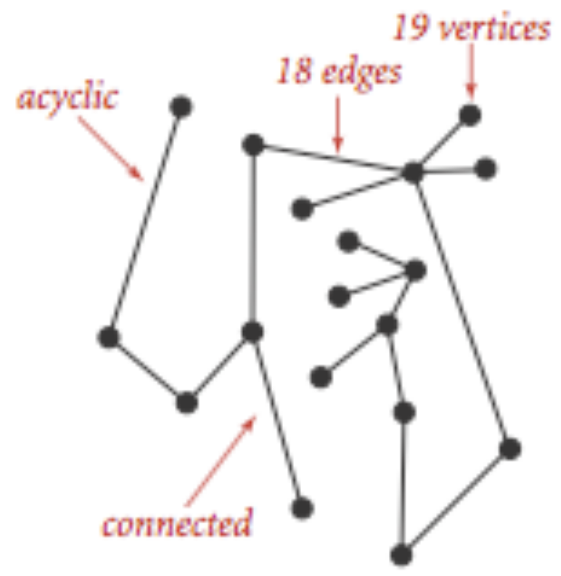


- A graph is connected if there is a path from every vertex to every other vertex.
- A graph that is not connected consists of a set of connected components, which are maximal connected subgraphs.
- An acyclic graph is a graph with no cycles.
- A tree is an acyclic connected graph.
- A forest is a disjoint set of trees.

Graph Terminology



Anatomy of a graph



A tree

Popular graph problems

Problem	Description
s-t path	Is there a path between s and t?
Shortest s-t path	What is the shortest path between s and t?
Cycle	Is there a cycle in the graph?
Euler cycle	Is there a cycle that uses each edge exactly once?
Hamilton cycle	Is there a cycle that uses each vertex exactly once?
Connectivity	Is there a path between every pair of vertices?
Biconnectivity	Is there an vertex whose removal disconnects the graph?

Lecture 33: Intro to Undirected Graphs

- Undirected Graphs

Readings:

- Textbook: Chapter 4.1 (Pages 515-521)
- Website:
    - https://algs4.cs.princeton.edu/41graph/

Lecture 34: Undirected Graphs

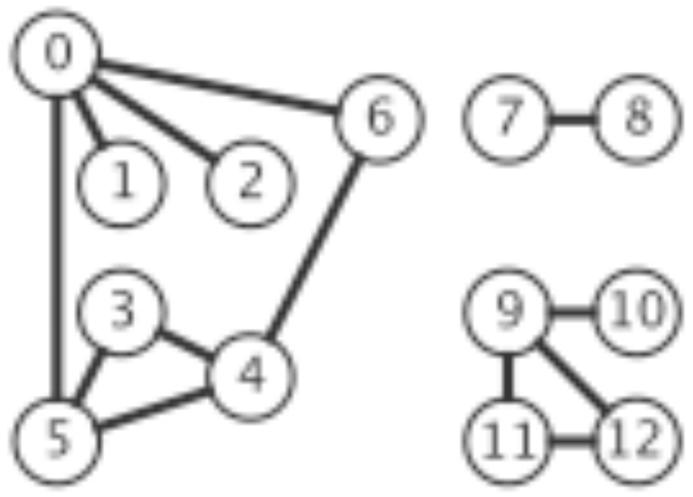
- Graph API
- Depth-First Search
- Breadth-First Search
- Connected Components

Some slides adopted from Algorithms 4th Edition or COS226

Graph representation

- Vertex representation: Here, integers between 0 and V-1.
  - We will use a symbol table to map between names and integers.

```
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```



Basic Graph API

- `public class Graph`
- `Graph(int V)`: create an empty graph with V vertices.
- `void addEdge(int v, int w)`: add an edge v-w.
- `Iterable<Integer> adj(int v)`: return vertices adjacent to v.
- `int V()`: number of vertices.
- `int E()`: number of edges.

Example of how to use the Graph API to process the graph

-
  ```
  public static int degree(Graph g, int v){
          int count = 0;
          for(int w : g.adj(v))
                  count++;
          return count;
  }
  ```

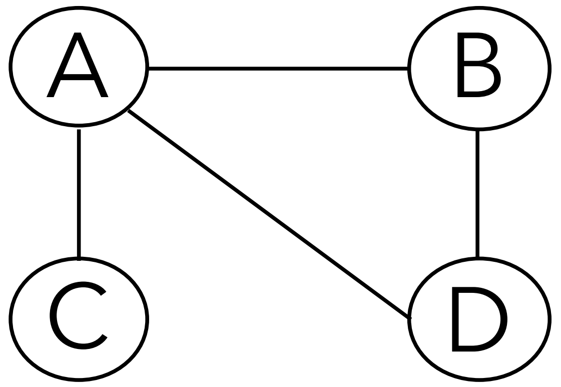
Graph density

- In a simple graph (no parallel edges or loops), if $|V| = n$, then:
  - minimum number of edges is 0 and
  - maximum number of edges is $n(n-1)/2$.
- Dense graph -> edges closer to maximum.
- Sparse graph -> edges closer to minimum.

Graph representation: adjacency matrix

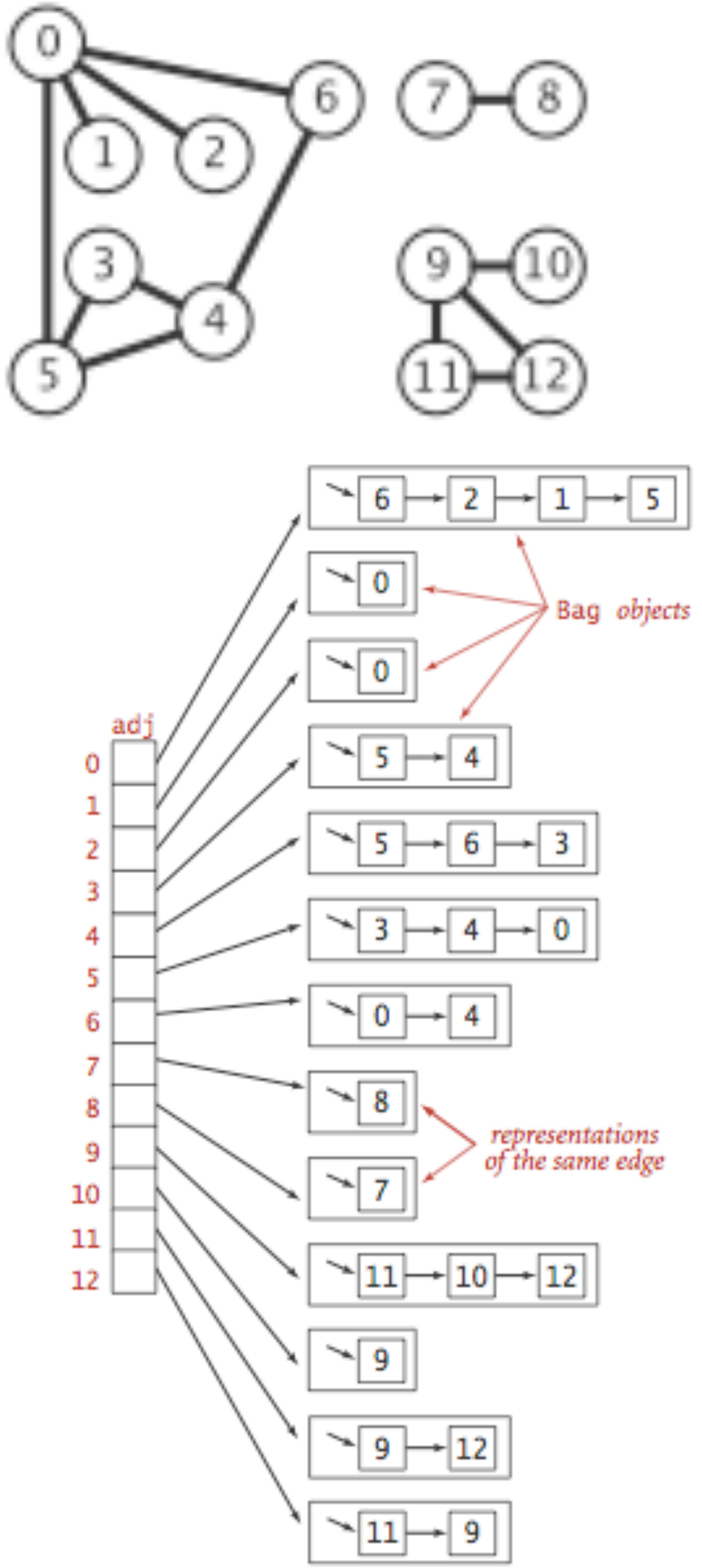
- Maintain a $|V|$-by-$|V|$ boolean array;
  for each edge `v-w`:
  - `adj[v][w] = adj[w][v] = true;` (1).
- Good for dense graphs (edges close to $|V|^2$).
- Constant time for lookup of an edge.
- Constant time for adding an edge.
- $|V|$ time for iterating over vertices adjacent to v.
- Symmetric, therefore wastes space in undirected graphs ($|V|^2$).
- Not widely used in practice.

	A	**B**	**C**	**D**
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0



Graph representation: adjacency list

- Maintain vertex-indexed array of lists.
- Good for sparse graphs (edges proportional to $|V|$) which are much more common in the real world.
- Algorithms based on iterating over vertices adjacent to v.
- Space efficient ($|E| + |V|$).
- Constant time for adding an edge.
- Lookup of an edge or iterating over vertices adjacent to v is $degree(v)$.



Adjacency-list graph representation in Java

```
public class Graph {

    private final int V;
    private int E;
    private Bag<Integer>[] adj;

    //Initializes an empty graph with V vertices and 0 edges.
    public Graph(int V) {
        this.V = V;
        this.E = 0;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Integer>();
        }
    }

    //Adds the undirected edge v-w to this graph. Parallel edges and self-loops allowed
    public void addEdge(int v, int w) {
        E++;
        adj[v].add(w);
        adj[w].add(v);
    }


    //Returns the vertices adjacent to vertex v.
    public Iterable<Integer> adj(int v) {
       return adj[v];
    }
```

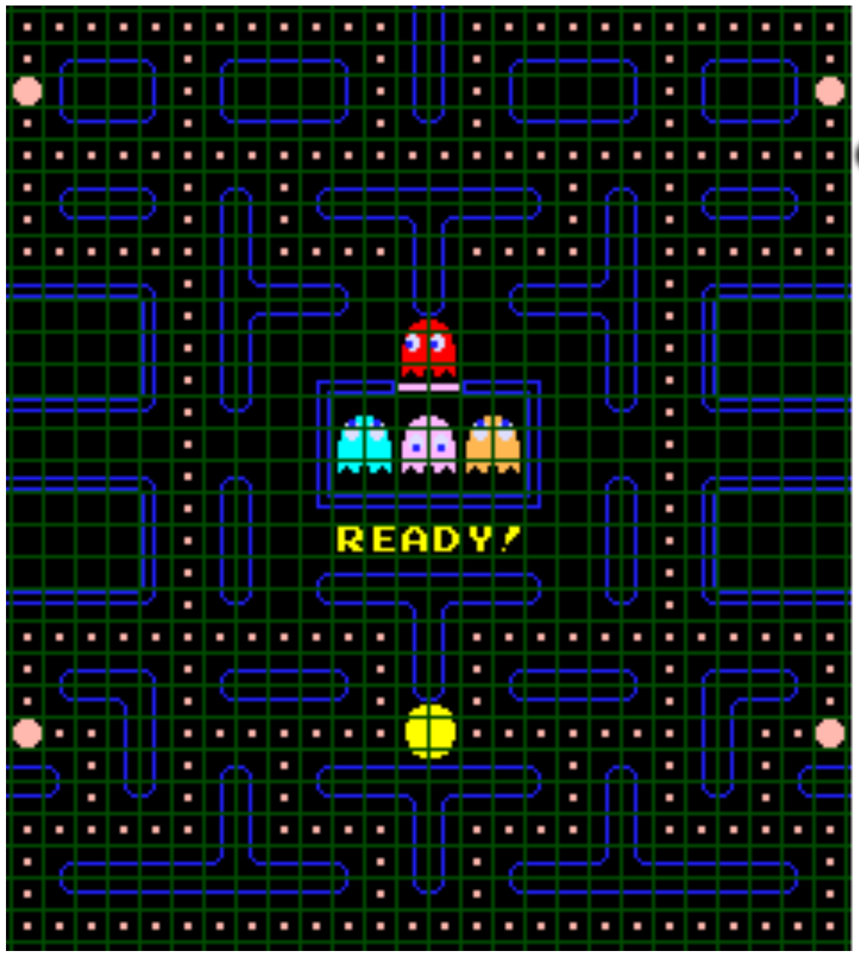
A bag is a collection where removing items is not supported—its purpose is to provide clients with the ability to collect items and then to iterate through the collected items

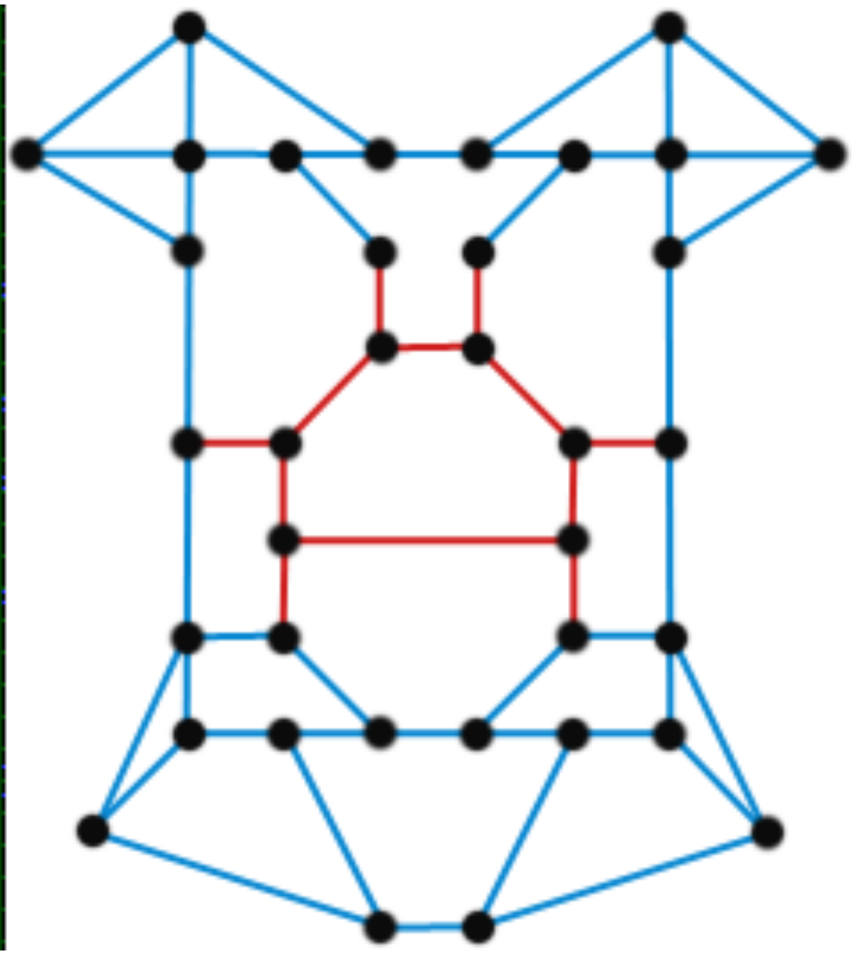
Lecture 34: Undirected Graphs

- Graph API
- Depth-First Search
- Breadth-First Search
- Connected Components

Mazes as graphs

- Vertex = intersection; edge = passage

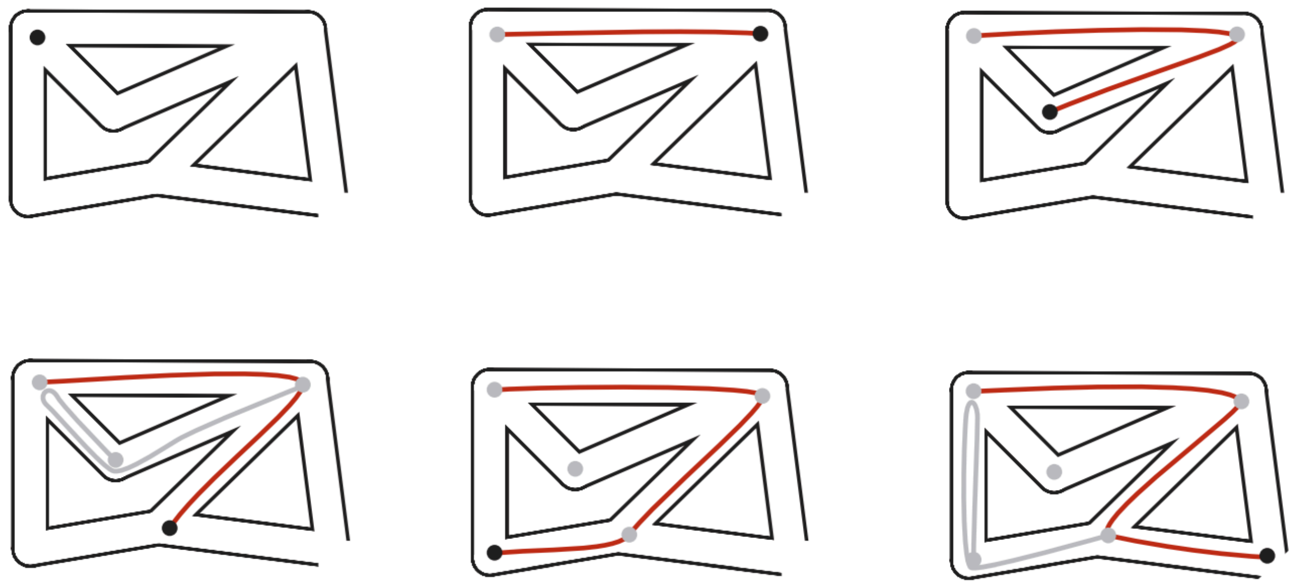




http://oatzy.blogspot.com/2011/09/playing-with-pac-man.html

How to survive a maze: a lesson from a Greek myth

- Theseus escaped from the labyrinth after killing the Minotaur with the following strategy instructed by Ariadne:
  - Unroll a ball of string behind you.
  - Mark each newly discovered intersection.
  - Retrace steps when no unmarked options.
- Also known as the Trémaux algorithm.



Depth-first search

- Goal: Systematically traverse a graph.
- DFS (to visit a vertex v)
  - Mark vertex v.
  - Recursively visit all unmarked vertices w adjacent to v.
- Typical applications:
  - Find all vertices connected to a given vertex.
  - Find a path between two vertices.

http://algs4.cs.princeton.edu

4.1 DEPTH-FIRST SEARCH DEMO

Depth-first search

- Goal: Find all vertices connected to s (and a corresponding path).
- Idea: Mimic maze exploration.
- Algorithm:
    - Use recursion (ball of string).
    - Mark each visited vertex (and keep track of edge taken to visit it).
    - Return (retrace steps) when no unvisited options.

- When started at vertex s, DFS marks all vertices connected to s (and no other).

Depth-first search in Java

```
public class DepthFirstSearch {
     private boolean[] marked;    // marked[v] = is there an s-v path?
     private int[] edgeTo;        // edgeTo[v] = previous vertex on path from s to v

    public DepthFirstSearch(Graph G, int s) {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        dfs(G, s);
    }

    // depth first search from v
    private void dfs(Graph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                edgeTo[w] = v;
                dfs(G, w);
            }
        }
    }
```

Depth-first search Analysis

- DFS marks all vertices connected to `s` in time proportional to $|V| + |E|$ in the worst case.
  - Initializing arrays `marked` and `edgeTo` takes time proportional to $|V|$.
  - Each adjacency-list entry is examined exactly once and there are $2E$ such edges (two for each edge).
- Once we run DFS, we can check if vertex `v` is connected to `s` in constant time. We can also find the `v`-`s` path (if it exists) in time proportional to its length.

Lecture 34: Undirected Graphs

- Graph API
- Depth-First Search
- Breadth-First Search
- Connected Components

Breadth-first search

- BFS (from source vertex s)
  - Put s on a queue and mark it as visited.
  - Repeat until the queue is empty:
    - Dequeue vertex v.
    - Enqueue each of v's unmarked neighbors and mark them.

- Basic idea: BFS traverses vertices in order of distance from s.

http://algs4.cs.princeton.edu

4.1 Breadth-First Search Demo

Breadth-first search in Java

```
public class BreadthFirstPaths {
   private boolean[] marked;  // marked[v] = is there an s-v path
    private int[] edgeTo;      // edgeTo[v] = previous edge on shortest s-v path
    private int[] distTo;      // distTo[v] = number of edges shortest s-v path

    public BreadthFirstPaths(Graph G, int s) {
        marked = new boolean[G.V()];
        distTo = new int[G.V()];
        edgeTo = new int[G.V()];
        bfs(G, s);
   }

   private void bfs(Graph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        distTo[s] = 0;
        marked[s] = true;
        q.enqueue(s);

        while (!q.isEmpty()) {
            int v = q.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    edgeTo[w] = v;
                    distTo[w] = distTo[v] + 1;
                    marked[w] = true;
                    q.enqueue(w);
                }
            }
        }
    }
```

Breadth-first search

- DFS: Put unvisited vertices on a stack.
- BFS: Put unvisited vertices on a queue.
- Shortest path problem: Find path from s to t that uses the fewest number of edges.
  - E.g., calculate the fewest numbers of hops in a communication network.
  - E.g., calculate the Kevin Bacon number or Erdös number.
- BFS computes shortest paths from s to all vertices in a graph in time proportional to $|E| + |V|$
  - The queue always consists of zero or more vertices of distance k from s, followed by zero or more vertices of k+1.

Lecture 34: Undirected Graphs

- Graph API
- Depth-First Search
- Breadth-First Search
- Connected Components

Connectivity queries

- Goal: Preprocess graph to answer questions of the form "is `v` connected to `w`" in constant time.
- `public class CC`
- `CC(Graph G)`: find connected components in `G`.
- `boolean connected(int v, int w)`: are `v` and `w` connected?
- `int count()`: number of connected components.
- `int id(int v)`: component identifier for vertex `v`.

Connected components

- Goal: Partition vertices into connected components.
- Connected Components
    - Initialize all vertices as unmarked.
    - For each unmarked vertex, run DFS to identify all vertices discovered as part of the same component.

http://algs4.cs.princeton.edu

4.1 CONNECTED COMPONENTS DEMO

Connected Components in Java

```
public class CC {
    private boolean[] marked;   // marked[v] = has vertex v been marked?
    private int[] id;           // id[v] = id of connected component containing v
    private int[] size;         // size[id] = number of vertices in given component
    private int count;          // number of connected components

    public CC(Graph G) {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        size = new int[G.V()];
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
                count++;
            }
        }
    }

    private void dfs(Graph G, int v) {
        marked[v] = true;
        id[v] = count;
        size[count]++;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }
```

Lecture 34: Undirected Graphs

- Graph API
- Depth-First Search
- Breadth-First Search
- Connected Components

Readings:

- Textbook: Chapter 4.1 (Pages 522-556)
- Website:
  - https://algs4.cs.princeton.edu/41graph/

Practice Problems:

- 4.1.1-4.1.6, 4.1.9, 4.1.11