DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 24

1. HOW HASH MAPS WORK
2. RUST: HASH MAPS WITH CUSTOM TYPES
3. REPRESENTING GRAPHS

1. HOW HASH MAPS WORK

2. RUST: HASH MAPS WITH CUSTOM TYPES

3. REPRESENTING GRAPHS

LAST TIME: HASH MAPS `HashMap<K,V>`

- Mapping
  - from a set of keys (type `K`)
  - to set of values (type `V`)
- Python: dictionaries
- C++: `unordered_map<K,V>`
- Java: `Hashtable<K,T>`

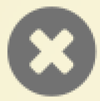
LAST TIME: HASH MAPS `HashMap<K,V>`

- Mapping
  - from a set of keys (type `K`)
  - to set of values (type `V`)
- Python: dictionaries
- C++: `unordered_map<K,V>`
- Java: `Hashtable<K,T>`

In [2]:

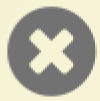
```
use std::collections::HashMap;

let mut crispy_crêpes_café = HashMap::new();
crispy_crêpes_café.insert(String::from("Nutella Crêpe"),5.85);
crispy_crêpes_café.insert(String::from("Strawberries and Nutella Crêpe"),8.75);
crispy_crêpes_café.insert(String::from("Roma Tomato, Pesto and Spinach Crêpe"),8.90);
crispy_crêpes_café.insert(String::from("Three Mashroom Crêpe"),8.90);

fn on_the_menu(cafe: &HashMap<String,f64>, s:String) {
    print!("{}: ",s);
    match cafe.get(&s) {
        None => println!("not on the menu"),
        Some(price) => println!("${:.2}",price),
    }
}
on_the_menu(&crispy_crêpes_café, String::from("Four Mashroom Crêpe"));
on_the_menu(&crispy_crêpes_café, String::from("Three Mashroom Crêpe"));
```

```
Four Mashroom Crêpe: not on the menu
Three Mashroom Crêpe: $8.90
```



STORAGE

- Array representing B buckets
- *Hash function* $h : K \to \{0, 1, \dots, B-1\}$
  - maps keys in the collection to buckets

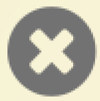


STORAGE

- Array representing B buckets
- *Hash function* $h : K \to \{0, 1, \dots , B - 1\}$
  - maps keys in the collection to buckets

GENERAL IDEAS

- Store keys (and associated values) in buckets
- Searching: go over the entire bucket



STORAGE

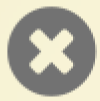
- Array representing B buckets
- *Hash function* $h : K \to \{0, 1, \ldots, B-1\}$
  - maps keys in the collection to buckets

GENERAL IDEAS

- Store keys (and associated values) in buckets
- Searching: go over the entire bucket

COLLISION: TWO KEYS MAPPED TO THE SAME BUCKET

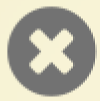
- Make hash function h very random $\Rightarrow$ few collisions
- What to do if two keys in the same bucket



HANDLING COLLISIONS

CHAINING

- Keep collection for items in the same bucket
  - (traditional:) linked list
  - vector
- Search through the collection to find key



HANDLING COLLISIONS

CHAINING

- Keep collection for items in the same bucket
  - (traditional:) linked list
  - vector
- Search through the collection to find key

OPEN ADDRESSING (SIMPLEST VERSION)

- Each array entry: (key,value)

Inserting:

- entry $h(k)$ busy: try $h(k)+1$, $h(k)+2$, etc.
- insert into first empty

?

HANDLING COLLISIONS

CHAINING

- Keep collection for items in the same bucket
  - (traditional:) linked list
  - vector
- Search through the collection to find key

OPEN ADDRESSING (SIMPLEST VERSION)

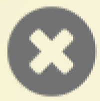
- Each array entry: (key,value)

Inserting:

- entry $h(k)$ busy: try $h(k)+1$, $h(k)+2$, etc.
- insert into first empty

Searching:

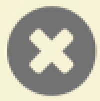
- try $h(k)$, $h(k)+1$, $h(k)+2$, etc.
- stop when found or empty entry



GROWING COLLECTION: AMORTIZATION

Example: if number of keys $\geq 0.75B$

- Double B
- Pick new hash function
- Move the information



GROWING COLLECTION: AMORTIZATION

Example: if number of keys $\geq 0.75B$

- Double B
- Pick new hash function
- Move the information

ADVERSARIAL DATA

- Could create lots of collisions
- Potential basis for *denial of service attacks*

1. HOW HASH MAPS WORK

2. RUST: HASH MAPS WITH CUSTOM TYPES

3. REPRESENTING GRAPHS

HASHING WITH CUSTOM TYPES IN RUST

Required for hashing:

1. check if $k_1, k_2 \in$ `K` equal
2. compute a hash function for elements of `K`

In [3]:
```
struct Point {
    x:f64,
    y:f64,
}

let point = Point{x:2.3,y:-1.4};

let mut elevation = HashMap::new();

elevation.insert(point,2.3);
```

```
elevation.insert(point,2.3);
                 ^^^^^ the trait `Eq` is not implemente
d for `Point`
elevation.insert(point,2.3);
          ^^^^^^ required by a bound introduced by this
call
the trait bound `Point: Eq` is not satisfied


elevation.insert(point,2.3);
                 ^^^^^ the trait `Hash` is not implemen
ted for `Point`
elevation.insert(point,2.3);
```

HASHING WITH CUSTOM TYPES IN RUST

Required for hashing:

1. check if $k_1, k_2 \in$ `K` equal
2. compute a hash function for elements of `K`

In [3]:
```
struct Point {
    x:f64,
    y:f64,
}

let point = Point{x:2.3,y:-1.4};

let mut elevation = HashMap::new();

elevation.insert(point,2.3);
```

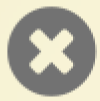
```
elevation.insert(point,2.3);
                 ^^^^^ the trait `Eq` is not implemente
d for `Point`
elevation.insert(point,2.3);
          ^^^^^^ required by a bound introduced by this
call
the trait bound `Point: Eq` is not satisfied


elevation.insert(point,2.3);
                 ^^^^^ the trait `Hash` is not implemen
ted for `Point`
elevation.insert(point,2.3);
```

Need two traits:

- `Eq`
- `Hash`

Default implementation:

In [4]:
```
#[derive(Hash,Eq,PartialEq)]
struct DistanceKM(i64);

let mut tired = HashMap::new();

tired.insert(DistanceKM(30),true);
```



`HashSet<K>`

- No value associated with keys
- Just a set of items
- Same implementation

HashSet<K>

- No value associated with keys
- Just a set of items
- Same implementation

In [5]:

```
use std::collections::HashSet;

// create
let mut covid = HashSet::new();

// insert values
for i in 2019..=2022 {
    covid.insert(i);
};
```

`HashSet<K>`

- No value associated with keys
- Just a set of items
- Same implementation

In [5]:
```
use std::collections::HashSet;

// create
let mut covid = HashSet::new();

// insert values
for i in 2019..=2022 {
    covid.insert(i);
};
```

In [6]:
```
// iterate over values in the set
for year in &covid {
    print!("{} ",year);
}
println!();
```

```
2022 2021 2019 2020
```

?

HashSet<K>

- No value associated with keys
- Just a set of items
- Same implementation

In [5]:
```
use std::collections::HashSet;

// create
let mut covid = HashSet::new();

// insert values
for i in 2019..=2022 {
    covid.insert(i);
};
```

In [6]:
```
// iterate over values in the set
for year in &covid {
    print!("{} ",year);
}
println!();
```

```
2022 2021 2019 2020
```

In [7]:
```
covid.get(&2015)
```

Out[7]:
```
None
```

`HashSet<K>`

- No value associated with keys
- Just a set of items
- Same implementation

In [5]:
```rust
use std::collections::HashSet;

// create
let mut covid = HashSet::new();

// insert values
for i in 2019..=2022 {
    covid.insert(i);
};
```

In [6]:
```rust
// iterate over values in the set
for year in &covid {
    print!("{} ",year);
}
println!();
```

```
2022 2021 2019 2020
```

In [7]:
```rust
covid.get(&2015)
```

Out[7]: `None`

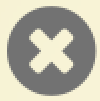
In [8]:
```rust
covid.get(&2021)
```

Out[8]: `Some(2021)`

1. HOW HASH MAPS WORK

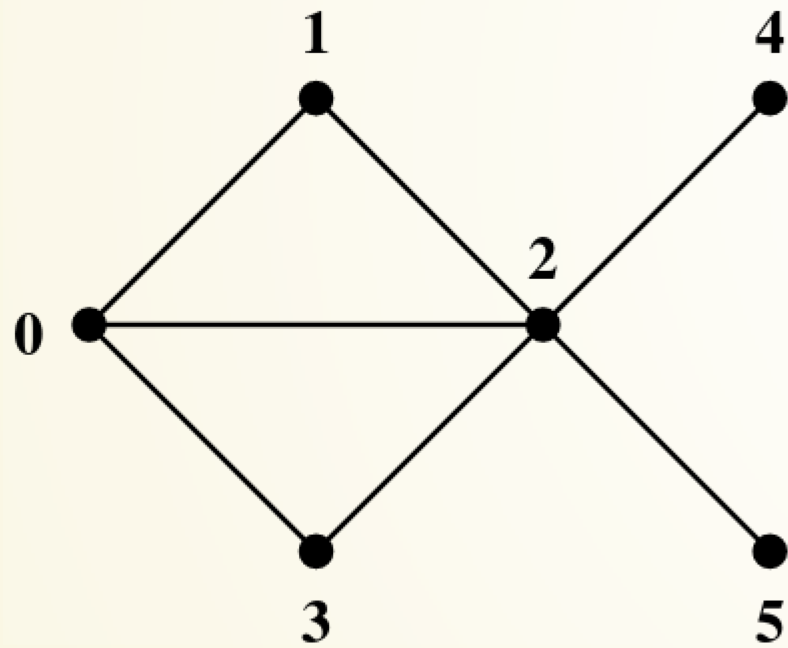
2. RUST: HASH MAPS WITH CUSTOM TYPES

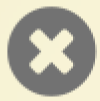
3. REPRESENTING GRAPHS



GRAPH REPRESENTATIONS: VARIOUS OPTIONS

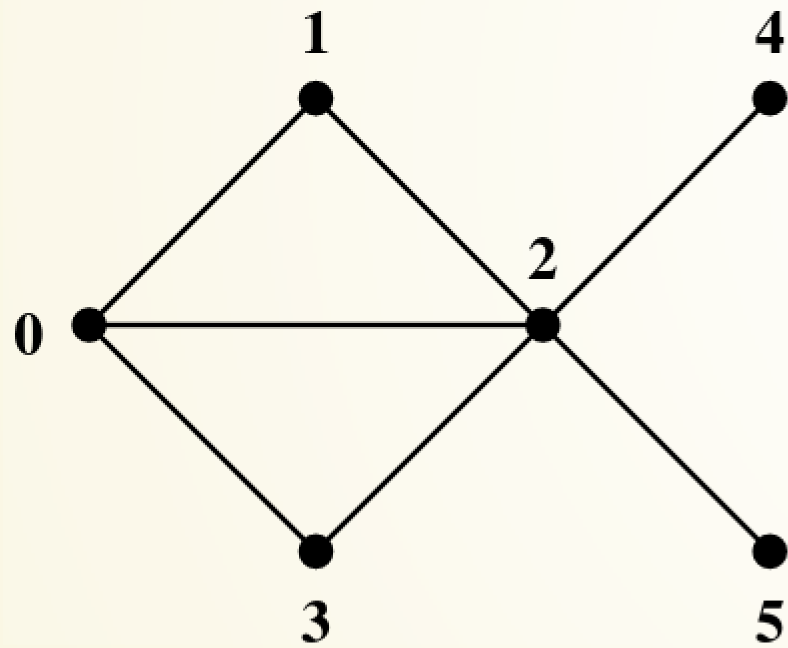
- What information we want to access
- What efficiency required





GRAPH REPRESENTATIONS: VARIOUS OPTIONS

- What information we want to access
- What efficiency required



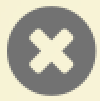
Today:

- Adjacency lists
- Adjacency matrix

Focus on undirected graphs:

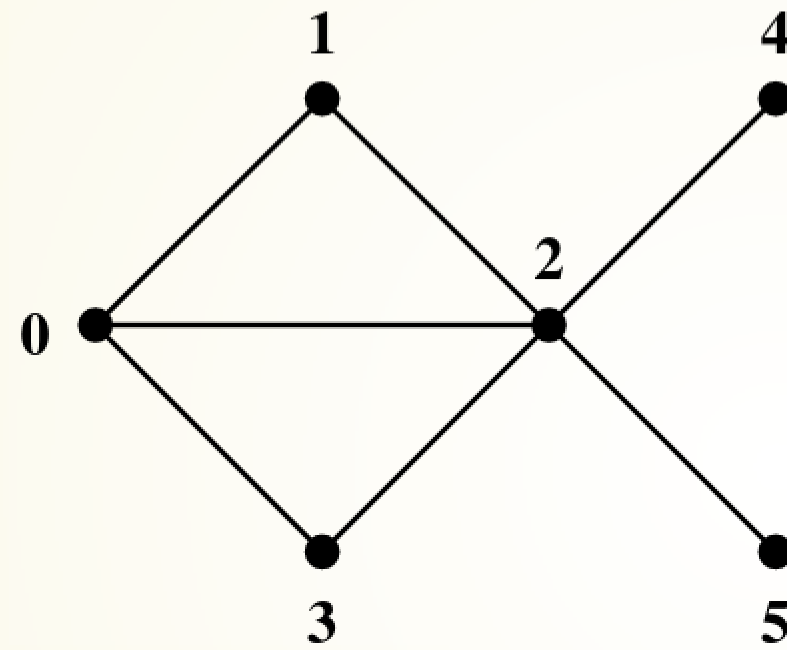
- easy to adjust for directed

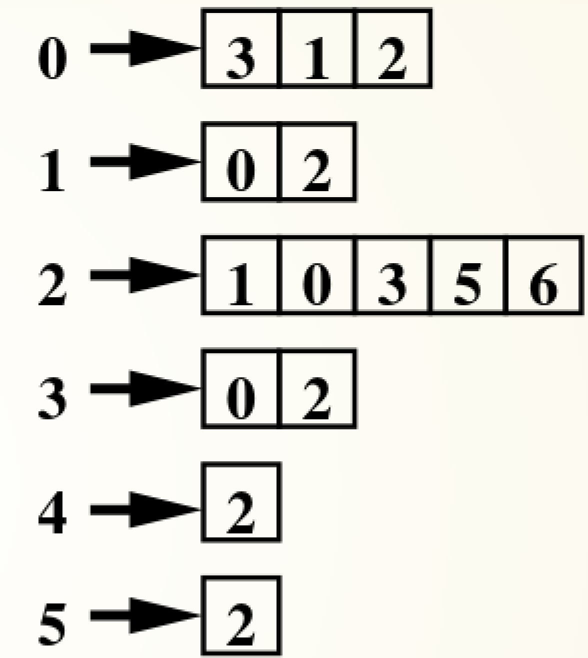
?

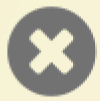


ADJACENCY LISTS

For each vertex, store the list of its neighbors

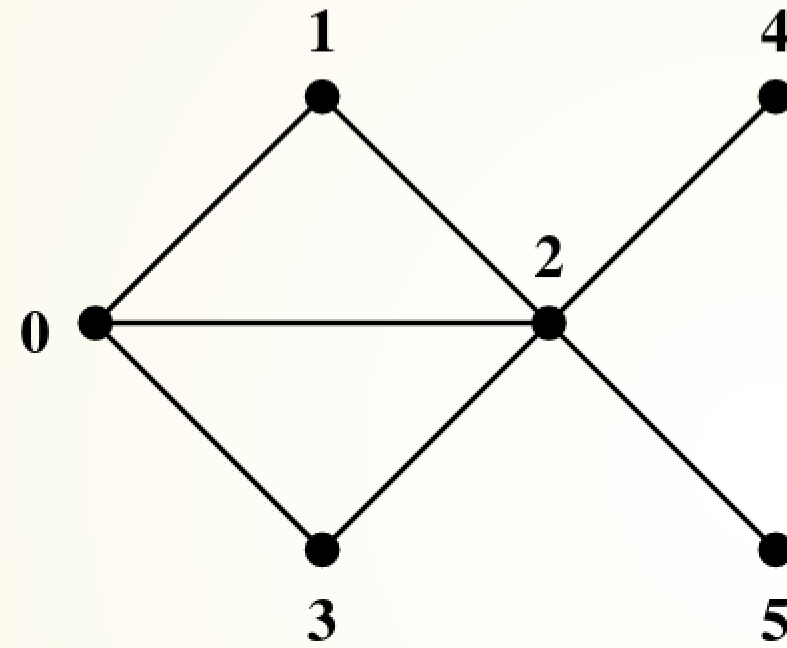


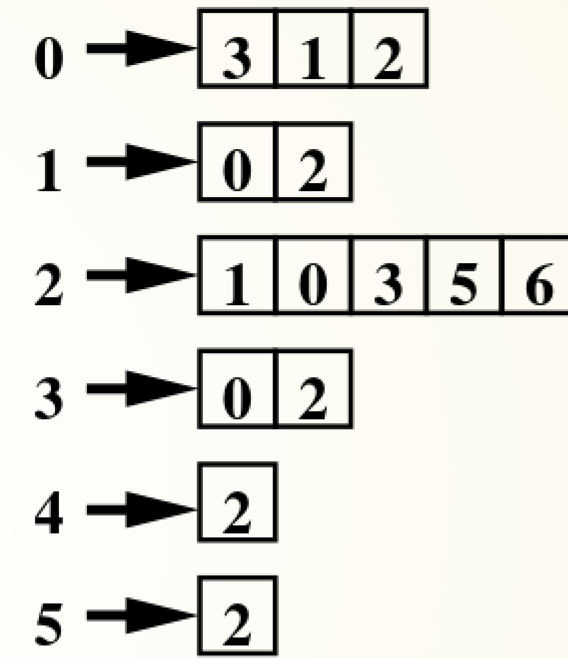




ADJACENCY LISTS

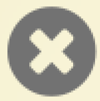
For each vertex, store the list of its neighbors





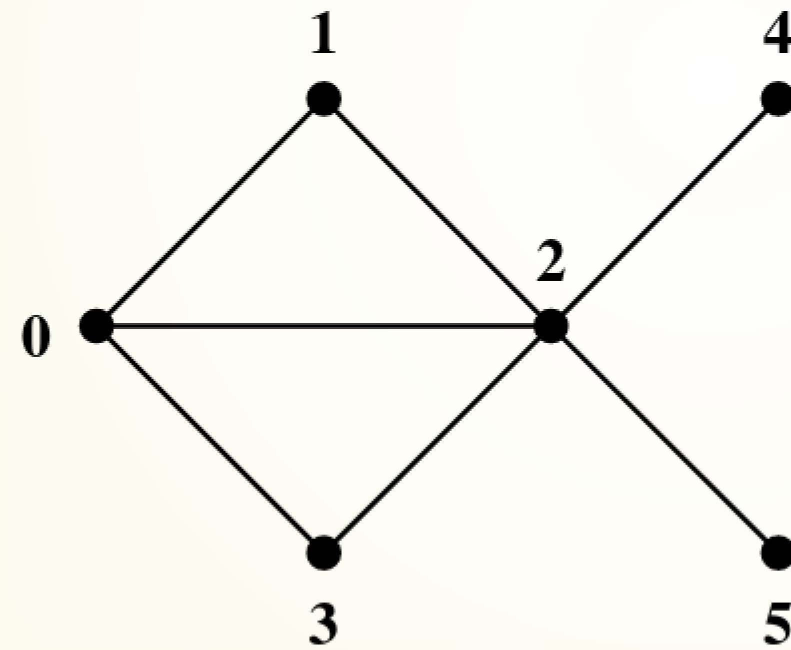
Collection:

- classical approach: linked list
- vectors



ADJACENCY MATRIX

- n vertices
- $n \times n$ matrix
- For each pair of vertices, store a boolean value: edge present or not



	0	1	2	3	4	5
0		1	1	1	0	0
1	1		1	0	0	0
2	1	1		1	1	1
3	1	0	1		0	0
4	0	0	1	0		0
5	0	0	1	0	0	