DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 23

1. MEMORY MANAGEMENT IN VECTORS

2. HASH MAPS

1. MEMORY MANAGEMENT IN VECTORS

2. HASH MAPS

LAST TIME: VECTORS `Vec<T>`

- Dynamic-length array/list
- Allowed operations:
  - access item at specific location
  - `push`: add something to the end
  - `pop`: remove an element from the end
- Python: list
- C++: `vector<T>`
- Java: `ArrayList<T>` / `Vector<T>`

LAST TIME: VECTORS `Vec<T>`

- Dynamic-length array/list
- Allowed operations:
  - access item at specific location
  - `push`: add something to the end
  - `pop`: remove an element from the end
- Python: list
- C++: `vector<T>`
- Java: `ArrayList<T>` / `Vector<T>`

HOW TO IMPLEMENT THIS EFFICIENTLY?

SELECT IMPLEMENTATION DETAILS

CHALLENGES

- Size changes: allocate on the heap?
- What to do if a new element added?
    - Allocate a larger array and copy everything?
    - Linked list?

SELECT IMPLEMENTATION DETAILS

CHALLENGES

- Size changes: allocate on the heap?
- What to do if a new element added?
    - Allocate a larger array and copy everything?
    - Linked list?

SOLUTION

- Allocate more space than needed!
- When out of space:
    - Increase storage size by, say, 100%
    - Copy everything

SELECT IMPLEMENTATION DETAILS

CHALLENGES

- Size changes: allocate on the heap?
- What to do if a new element added?
    - Allocate a larger array and copy everything?
    - Linked list?

SOLUTION

- Allocate more space than needed!
- When out of space:
    - Increase storage size by, say, 100%
    - Copy everything

UNDER THE HOOD

Variable of type `Vec<T>` contains:

- pointer to allocated memory
- size: the current number of items
- capacity: how many items could currently fit

Important: size $\leq$ capacity

EXAMPLE

Method `capacity()` reports current storage size

In [2]:
```
// print out the current size and capacity
fn info<T>(vector:&Vec<T>) {
    println!("size = {}, capacity = {}",vector.len(),vector.capacity());
}
```

EXAMPLE

Method `capacity()` reports current storage size

In [2]:
```
// print out the current size and capacity
fn info<T>(vector:&Vec<T>) {
    println!("size = {}, capacity = {}",vector.len(),vector.capacity());
}
```

In [3]:
```
let mut v = Vec::new();
let mut capacity = v.capacity();
info(&v);
for i in 1..=1000 {
    v.push(i);
    if v.capacity() != capacity {
        capacity = v.capacity();
        info(&v);
    }
};
```

```
size = 0, capacity = 0
size = 1, capacity = 4
size = 5, capacity = 8
size = 9, capacity = 16
size = 17, capacity = 32
size = 33, capacity = 64
size = 65, capacity = 128
size = 129, capacity = 256
size = 257, capacity = 512
size = 513, capacity = 1024
```

EXAMPLE

Method `capacity()` reports current storage size

In [2]:
```
// print out the current size and capacity
fn info<T>(vector:&Vec<T>) {
    println!("size = {}, capacity = {}",vector.len(),vector.capacity());
}
```

In [3]:
```
let mut v = Vec::new();
let mut capacity = v.capacity();
info(&v);
for i in 1..=1000 {
    v.push(i);
    if v.capacity() != capacity {
        capacity = v.capacity();
        info(&v);
    }
};
```

```
size = 0, capacity = 0
size = 1, capacity = 4
size = 5, capacity = 8
size = 9, capacity = 16
size = 17, capacity = 32
size = 33, capacity = 64
size = 65, capacity = 128
size = 129, capacity = 256
size = 257, capacity = 512
size = 513, capacity = 1024
```

In [4]:
```
info(&v);
while let Some(_) = v.pop() {}
info(&v);
```

```
size = 1000, capacity = 1024
size = 0, capacity = 1024
```

EXAMPLE (CONTINUED)

In [5]:
```
// shrinking the size manually
info(&v);

for i in 1..=13 {
    v.push(i);
}

info(&v);

v.shrink_to_fit();

info(&v);
// note: size and capacity not guaranteed
//       to be the same
```

```
size = 0, capacity = 1024
size = 13, capacity = 1024
size = 13, capacity = 13
```

EXAMPLE (CONTINUED)

In [5]:
```
// shrinking the size manually
info(&v);

for i in 1..=13 {
    v.push(i);
}

info(&v);

v.shrink_to_fit();

info(&v);
// note: size and capacity not guaranteed
//       to be the same
```

```
size = 0, capacity = 1024
size = 13, capacity = 1024
size = 13, capacity = 13
```

In [6]:
```
// creating vector with specific capacity
let mut v2 : Vec<i32> = Vec::with_capacity(1234);
info(&v2);

// avoids reallocation if you know how many items
// to expect
```

```
size = 0, capacity = 1234
```

SKETCH OF ANALYSIS: AMORTIZATION

- Inserting an element not constant time, $O(1)$

SKETCH OF ANALYSIS: AMORTIZATION

- Inserting an element not constant time, $O(1)$

HOWEVER

- **Assumption:** allocating memory size t takes $O(t)$ or $O(1)$ time
- **Slow operations:** $O(\text{current\_size})$ time
- **Fast operations:** $O(1)$ time
- Slow operation every $\Omega(\text{current\_size})$ fast operations

SKETCH OF ANALYSIS: AMORTIZATION

- Inserting an element not constant time, $O(1)$

HOWEVER

- **Assumption:** allocating memory size t takes $O(t)$ or $O(1)$ time
- **Slow operations:** $O(\text{current\_size})$ time
- **Fast operations:** $O(1)$ time
- Slow operation every $\Omega(\text{current\_size})$ fast operations

- **On average:** $O(1)$ time
- Fast operations pay for slow operations
- **Terminology:** $O(1)$ *amortized* time

SKETCH OF ANALYSIS: AMORTIZATION

- Inserting an element not constant time, $O(1)$

HOWEVER

- **Assumption:** allocating memory size t takes $O(t)$ or $O(1)$ time
- **Slow operations:** $O(\text{current\_size})$ time
- **Fast operations:** $O(1)$ time
- Slow operation every $\Omega(\text{current\_size})$ fast operations

- **On average:** $O(1)$ time
- Fast operations pay for slow operations
- **Terminology:** $O(1)$ *amortized* time

SHRINKING?

- Can be implemented this way too
- Example: shrink by 50% if less than 25% used
- Most implementations don't shrink automatically

1. MEMORY MANAGEMENT IN VECTORS

2. HASH MAPS

COLLECTION `HashMap<K,V>`

Goal: a mapping from elements of `K` to elements of `V`

- elements of `K` called *keys*
- elements of `V` called *values*

COLLECTION `HashMap<K,V>`

Goal: a mapping from elements of `K` to elements of `V`

- elements of `K` called *keys*
- elements of `V` called *values*

In [7]:
```
// creating a hash map and inserting pair

use std::collections::HashMap;

// number of wins in a local Conterstrike league
let mut wins = HashMap::<String,u16>::new();

wins.insert(String::from("Boston University"),24);
wins.insert(String::from("Harvard"),22);
wins.insert(String::from("Boston College"),20);
wins.insert(String::from("Northeastern"),32);
```

COLLECTION `HashMap<K,V>`

Goal: a mapping from elements of `K` to elements of `V`

- elements of `K` called *keys*
- elements of `V` called *values*

In [7]:
```
// creating a hash map and inserting pair

use std::collections::HashMap;

// number of wins in a local Conterstrike league
let mut wins = HashMap::<String,u16>::new();

wins.insert(String::from("Boston University"),24);
wins.insert(String::from("Harvard"),22);
wins.insert(String::from("Boston College"),20);
wins.insert(String::from("Northeastern"),32);
```

Extracting a reference: returns `Option<&V>`

In [8]:
```
wins.get("Boston University")
```

Out[8]: Some(24)

COLLECTION `HashMap<K,V>`

Goal: a mapping from elements of `K` to elements of `V`

- elements of `K` called *keys*
- elements of `V` called *values*

In [7]:
```
// creating a hash map and inserting pair

use std::collections::HashMap;

// number of wins in a local Conterstrike league
let mut wins = HashMap::<String,u16>::new();

wins.insert(String::from("Boston University"),24);
wins.insert(String::from("Harvard"),22);
wins.insert(String::from("Boston College"),20);
wins.insert(String::from("Northeastern"),32);
```

Extracting a reference: returns `Option<&V>`

In [8]:
```
wins.get("Boston University")
```

Out[8]: Some(24)

In [9]:
```
wins.get("MIT")
```

Out[9]: None

Insert if not present:

```
In [10]: wins.entry(String::from("MIT")).or_insert(10);
         wins.get("MIT")

Out[10]: Some(10)
```

Insert if not present:

In [10]:
```
wins.entry(String::from("MIT")).or_insert(10);
wins.get("MIT")
```

Out[10]: Some(10)

Updating:

In [11]:
```
{ // block to limit how long the reference lasts
    let entry = wins.entry(String::from("Boston University")).or_insert(10);
    *entry = 50;
}
wins.insert(String::from("Boston University"),24);
wins.get("Boston University")
```

Out[11]: Some(24)

ITERATING

In [12]:
```
for (k,v) in &wins {
    println!("{}: {}",k,v);
};
```

```
MIT: 10
Harvard: 22
Boston College: 20
Northeastern: 32
Boston University: 24
```

ITERATING

In [12]:
```
for (k,v) in &wins {
    println!("{}: {}",k,v);
};
```

```
MIT: 10
Harvard: 22
Boston College: 20
Northeastern: 32
Boston University: 24
```

In [13]:
```
for (k,v) in &mut wins {
    *v += 1;
};

for (k,v) in &wins {
    println!("{}: {}",k,v);
};
```

```
MIT: 11
Harvard: 23
Boston College: 21
Northeastern: 33
Boston University: 25
```

NEXT TIME

- How do hash tables work?
- Typical graph representations