CS2040C Tut 7

Hash Table - Collision Resolution
Introduction to Binary Search Tree

Collision Resolutions

Open Addressing

Closed Addressing

Open vs Closed Addressing

Closed Addressing

Where the object will be slotted in is *completely* dependent on the hash function.

Open Addressing

Where the object will be slotted in depends on *other objects in the Hash Table*.

(The address can *vary*)

(Self-read)

Closed Addressing

Imagine there is a street with houses.

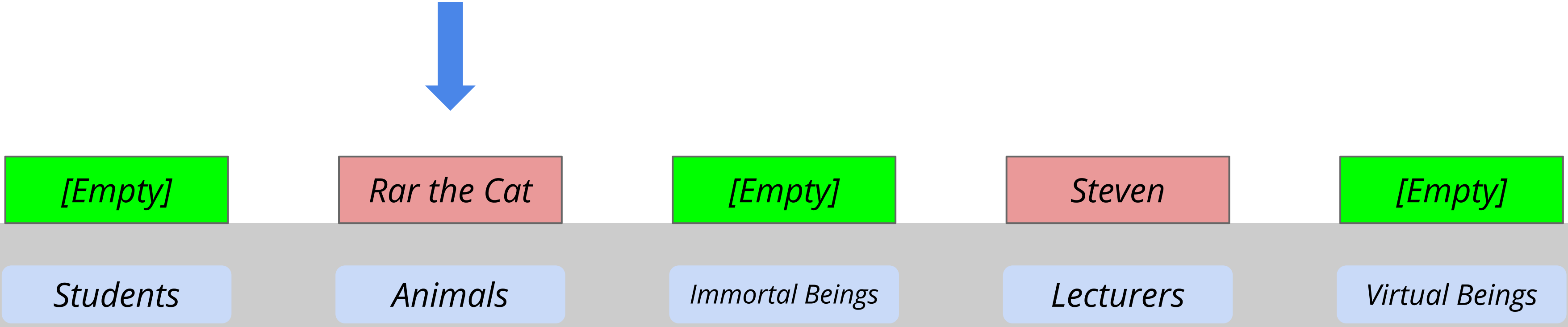
Some are empty, some are occupied.

[Empty]	*Rar the Cat*	*[Empty]*	*Steven*	*[Empty]*

Closed Addressing

A new person “Jacq the Dino” comes along.

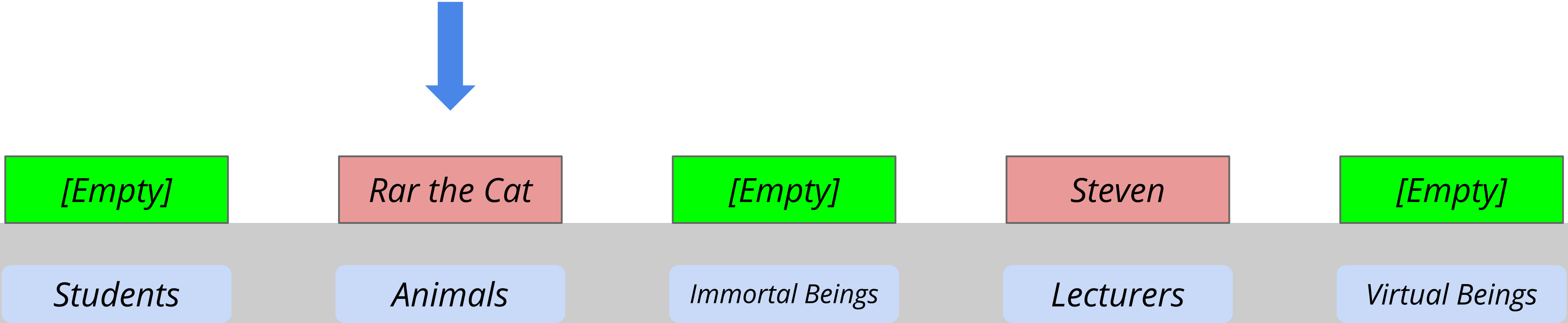
We use our hash function to determine where to place her.



(Self-read)

Closed Addressing

However, the house at the chosen location, is already occupied. :(



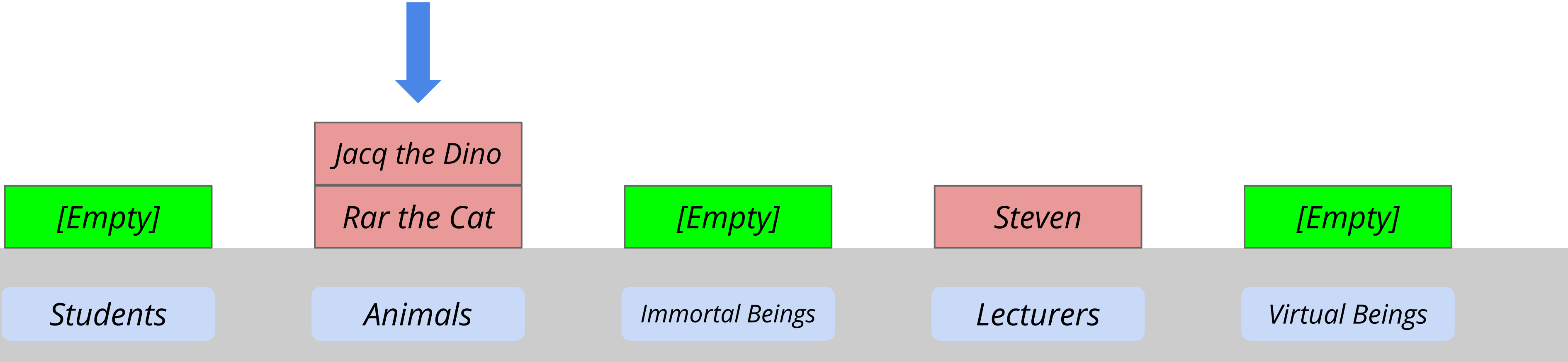
(Self-read)

Closed Addressing

In **closed** addressing, instead of finding a new house...

We simply ‘build a new floor’ at the same position...

and place the element there.



(Self-read)

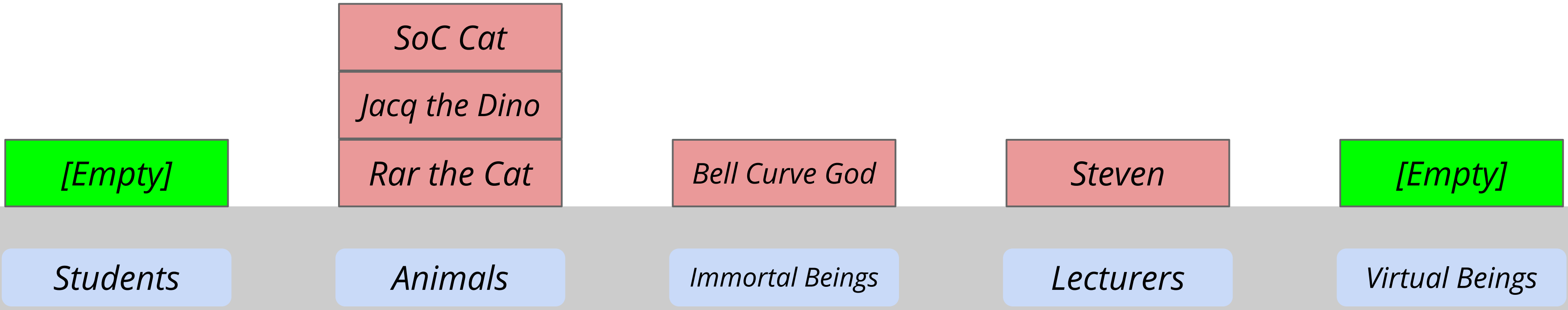
Closed Addressing

As we add more and more people, there are

Some positions with more people and

some positions with nobody.

However, we know they are *always in the position they are hashed to.*



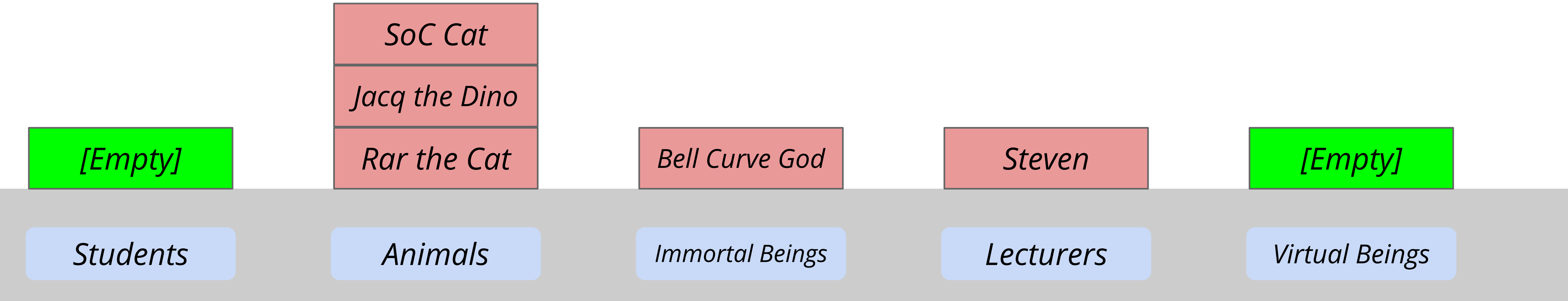
(Self-read)

Closed Addressing

Implementation

We can use an *array of* vectors (or Singly/Doubly LL).

Each position will be a *vector (resizeable array)* so that it can accommodate more people if needed.



(Self-read)

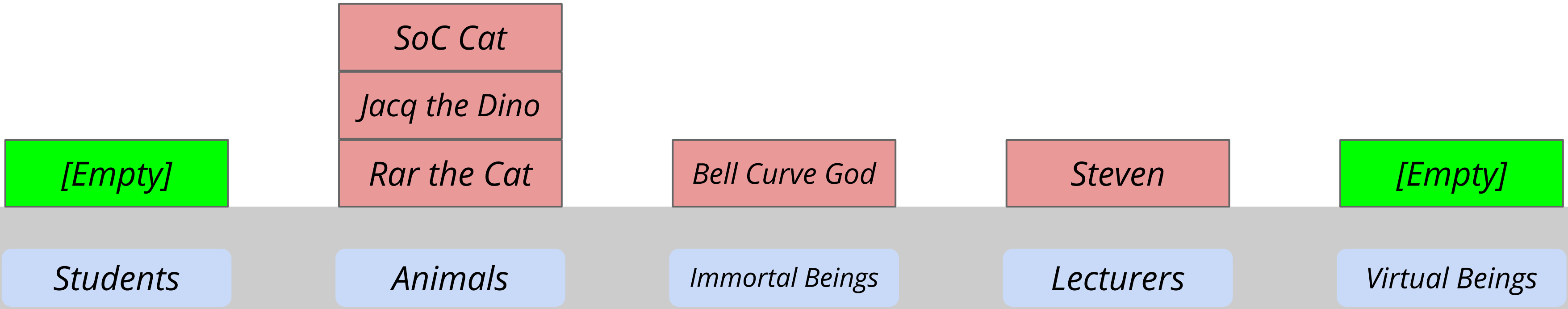
Closed Addressing

Implementation

Recall that finding an element in a vector is O(**α**).

Recall that deleting any element in a vector is also O(**α**).

We will try to make **α** as small as we can.



(Self-read)

Open Addressing

Imagine again there is a street with houses.

Some are empty, some are occupied.

And you have that same ‘hash function’ from just now too.

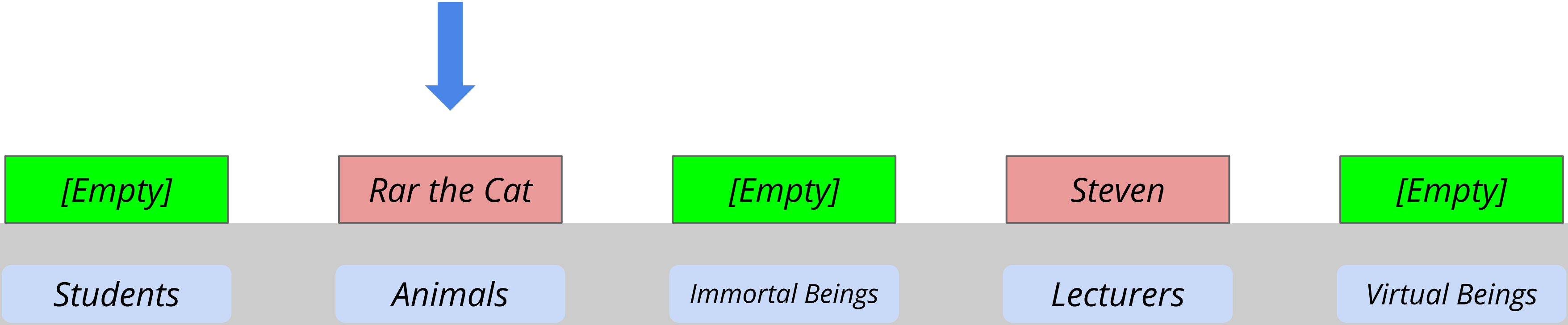
[Empty]	*Rar the Cat*	*[Empty]*	*Steven*	*[Empty]*
Students	*Animals*	*Immortal Beings*	*Lecturers*	*Virtual Beings*

(Self-read)

Open Addressing

Similarly, a new person “Jacq the Dino” comes along.

We determine where she should go, depending on the hash function.



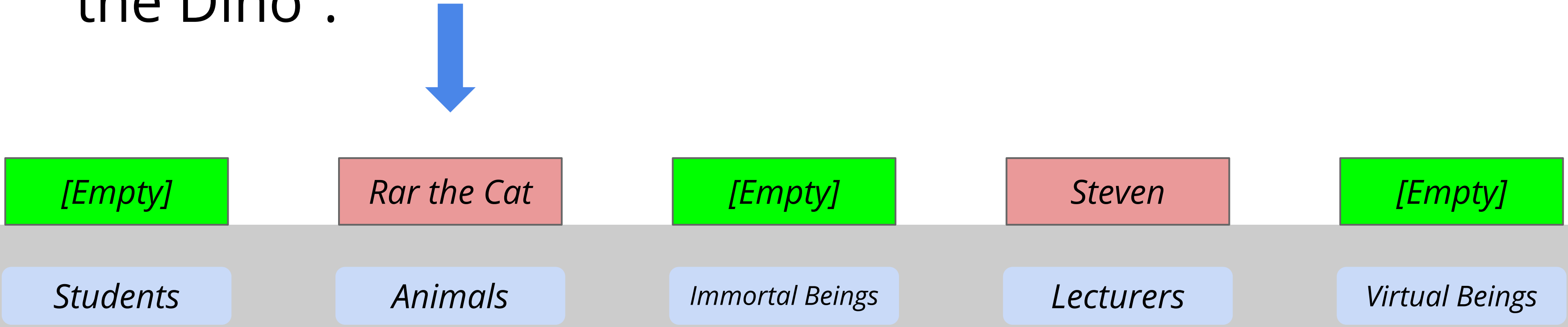
(Self-read)

Open Addressing

However, the house is already occupied :(

In open addressing, we cannot build another floor.

We thus need to find another (vacant) house for “Jacq the Dino”.

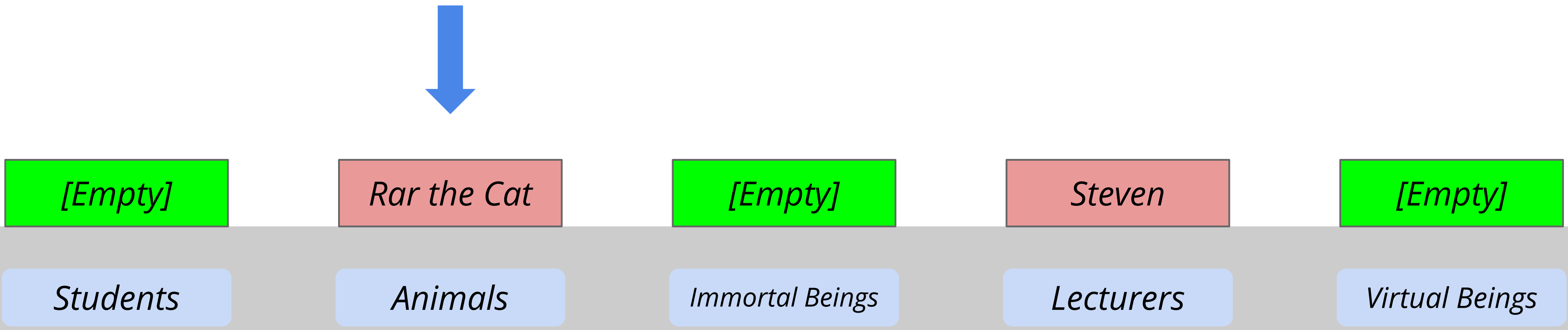


(Self-read)

Open Addressing

There are several ways to do so:

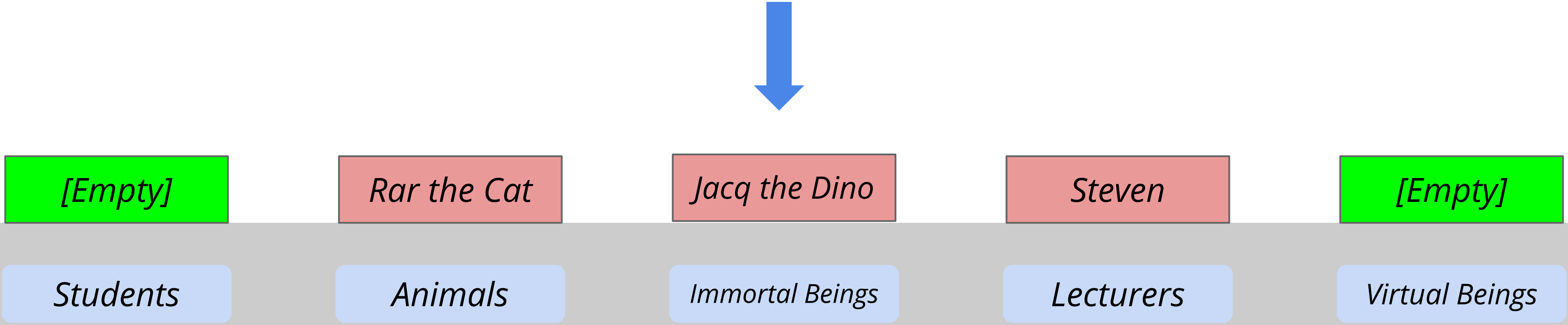
- Go down the road *one by one* and find the first vacant house on the right. (**Linear Probing** aka LP)
- Restart from the front if needed.



(Self-read)

Open Addressing - Linear Probing

If we used Linear Probing, “Jacq the Dino” will end up here!

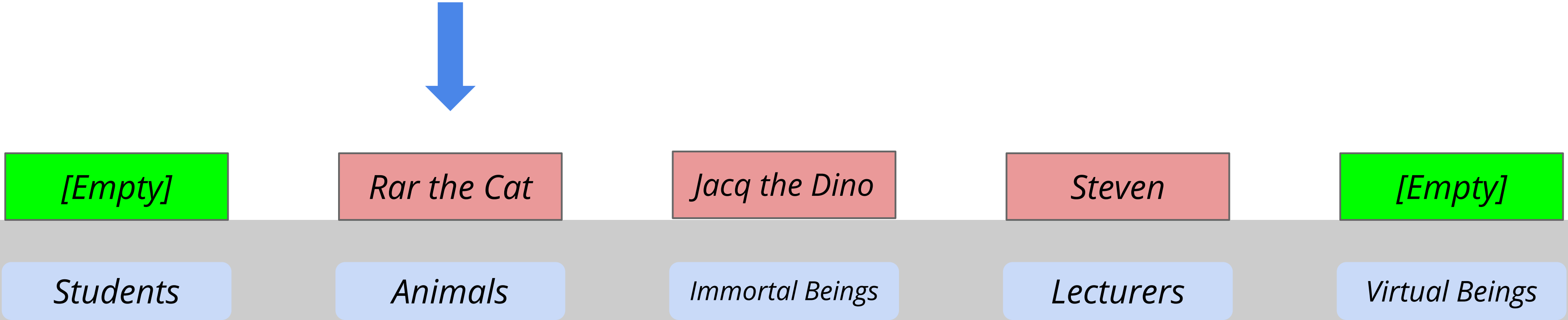


(Self-read)

Open Addressing - Linear Probing

Now lets see when “SoC Cat” comes.

Full...

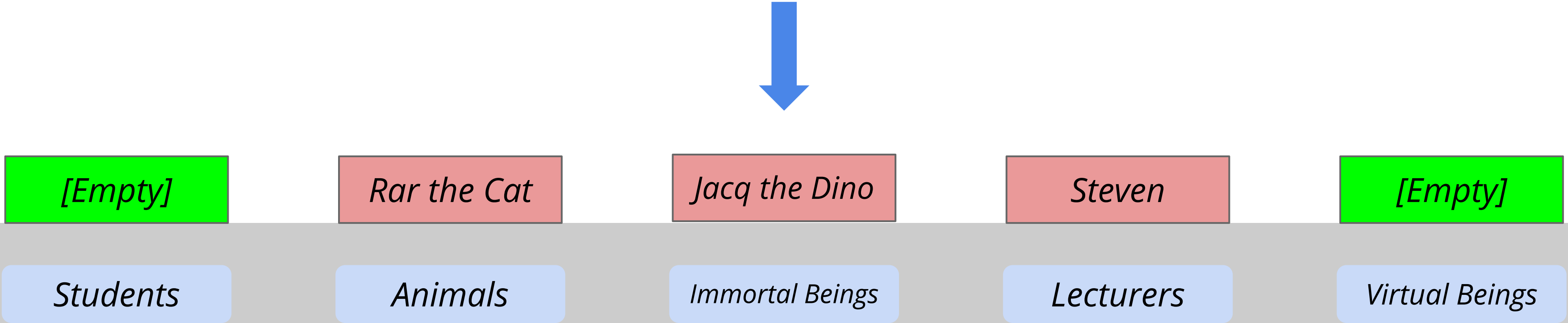


(Self-read)

Open Addressing - Linear Probing

Now lets see when "SoC Cat" comes.

Full… full…

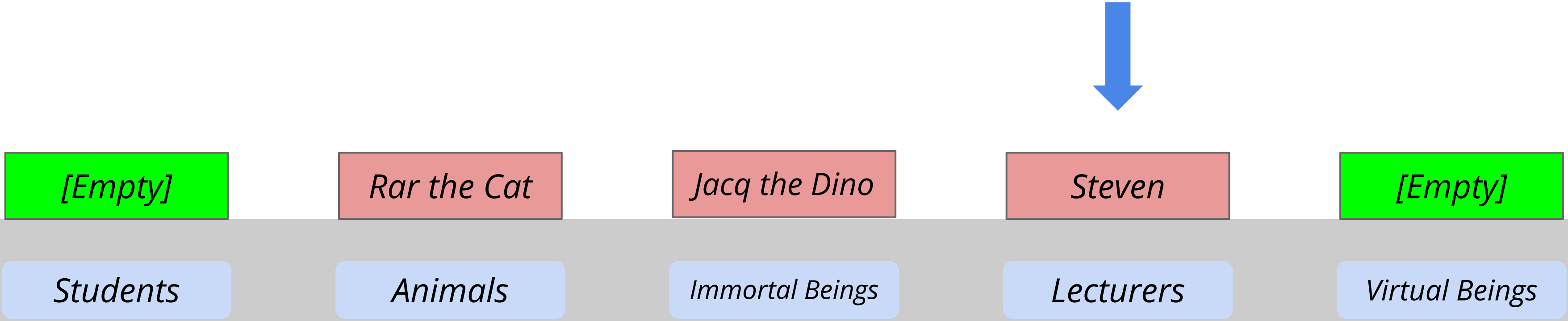


(Self-read)

Open Addressing - Linear Probing

Now lets see when “SoC Cat” comes.

Full… full… *still* full…

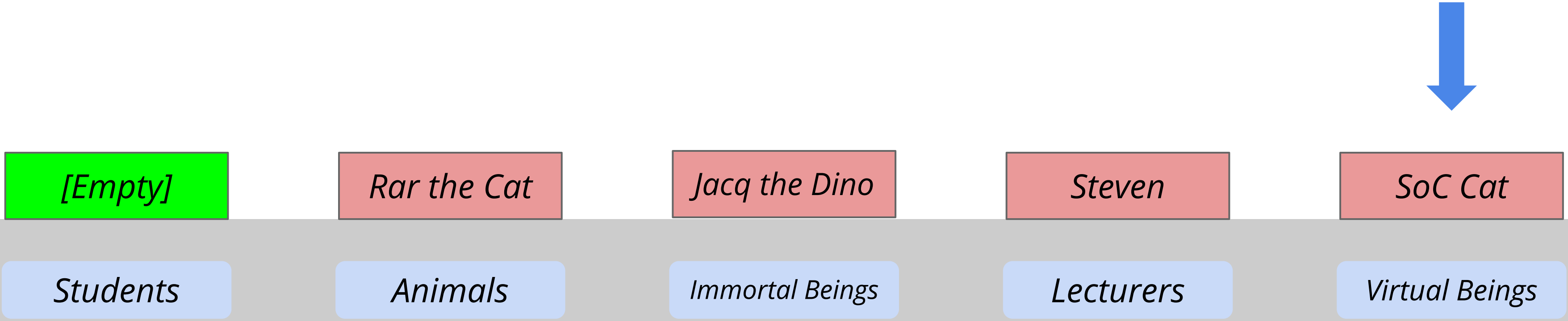


(Self-read)

Open Addressing - Linear Probing

Now lets see when “SoC Cat” comes.

Full… full… *still* full… vacant!



(Self-read)

Open Addressing - Linear Probing

Now when “Bell Curve God” arrives, he will end up there.

After probing a full cycle...

Bell Curve God	Rar the Cat	Jacq the Dino	Steven	SoC Cat
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

(Self-read)

Open Addressing - Linear Probing

As you can see, the hash function *does not have much meaning anymore...*

It only serves to determine the **starting position**.

(This might complicate *some* cases where the key stores important information)

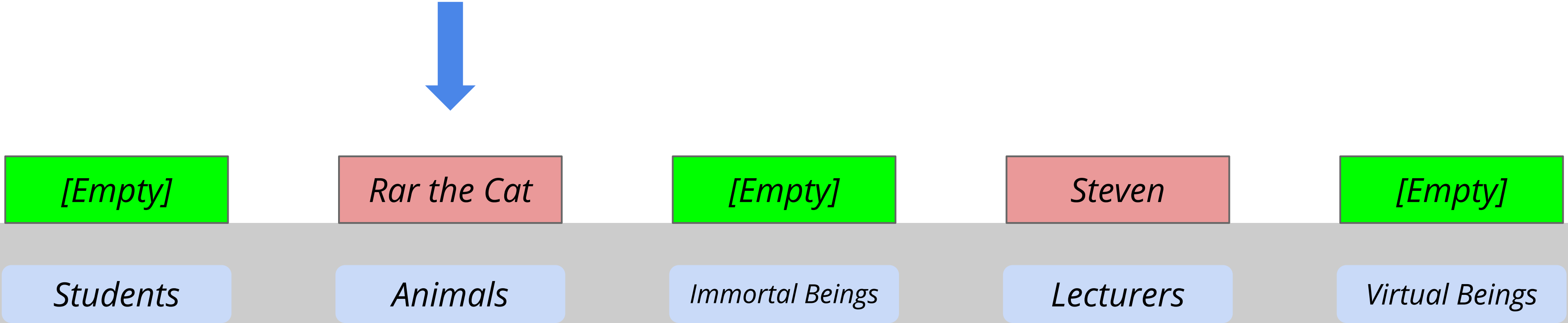
Bell Curve God	Rar the Cat	Jacq the Dino	Steven	SoC Cat
Students	*Animals*	*Immortal Beings*	*Lecturers*	*Virtual Beings*

(Self-read)

Open Addressing

Instead of checking adjacent houses, we can skip some houses based on a quadratic formula

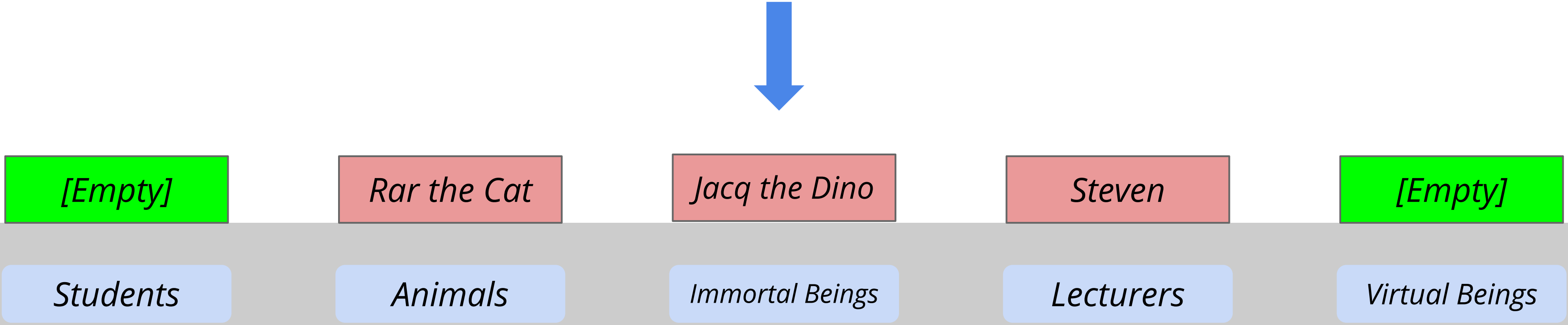
- 1st house, 4th house, 9th house … etc
- **Quadratic Probing** aka QP



(Self-read)

Open Addressing - Quadratic Probing

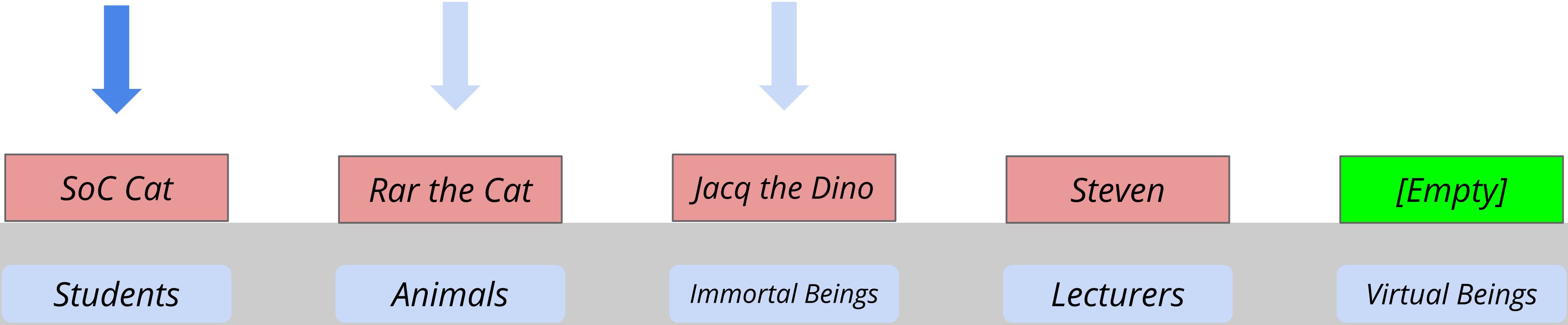
If we used Quadratic Probing, “Jacq the Dino” will still end up here!



(Self-read)

Open Addressing - Quadratic Probing

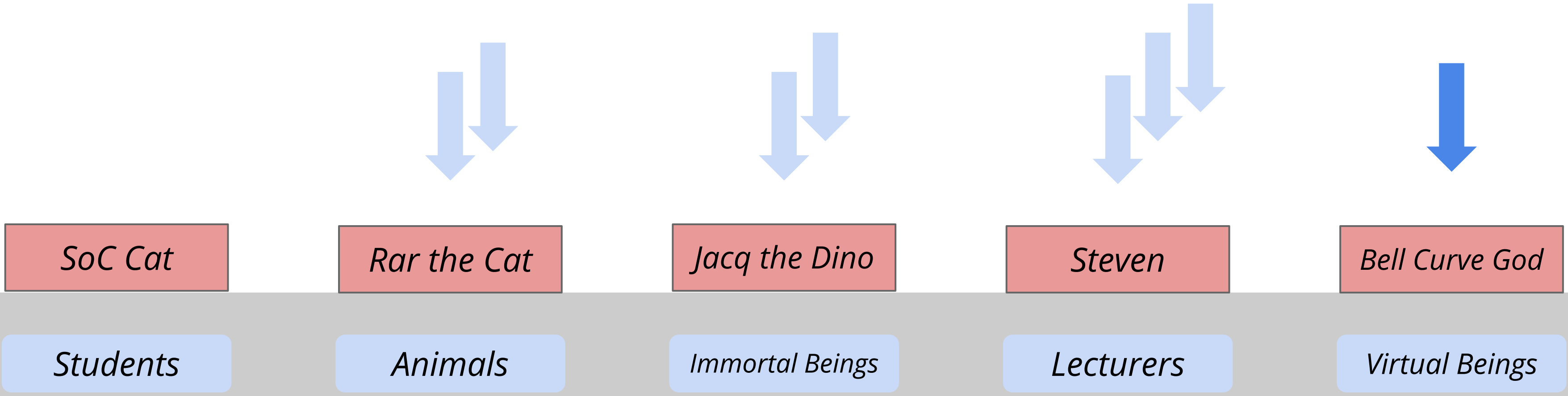
But when "SoC Cat" comes, he will end up... here.



(Self-read)

Open Addressing - Quadratic Probing

But when "Bell Curve God" comes, he will keep checking... and checking .. and end up at the last empty slot after **18** tries....



(Self-read)

Open Addressing - Double Hashing

We can also vary the order to check, based on another hash!

- Let **x** = hash2(*"Jacq the Dino"*) = **3**
- hash2(*"SoC Cat"*) = **2**
- hash2(*"Bell Curve God"*) = **4**

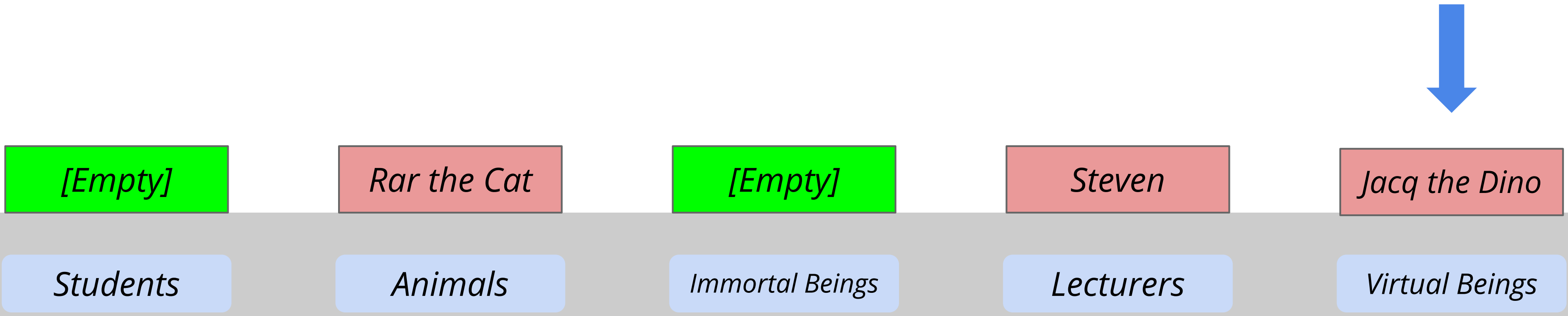
[Empty]	*Rar the Cat*	*[Empty]*	*Steven*	*[Empty]*
Students	*Animals*	*Immortal Beings*	*Lecturers*	*Virtual Beings*

(Self-read)

Open Addressing - Double Hashing

We can also vary the order to check, based on another hash!

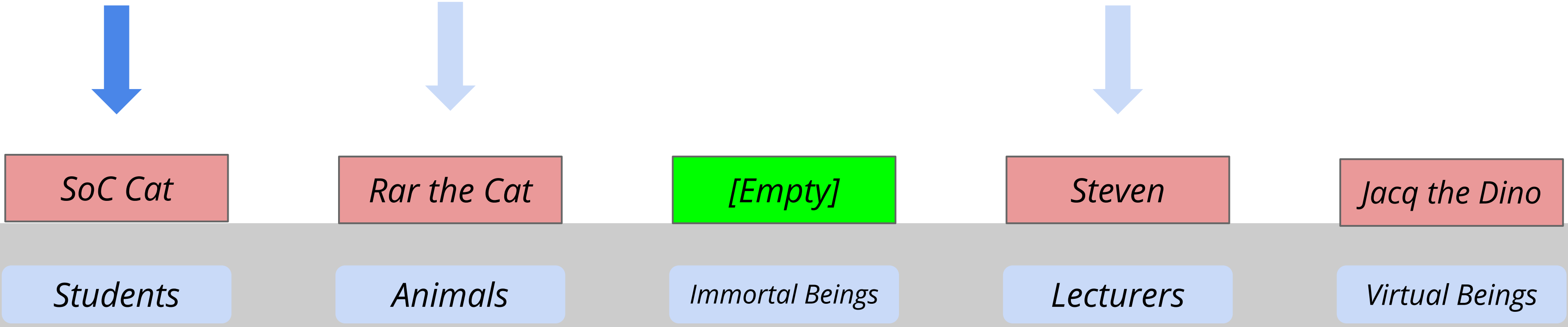
- Then we use the second hash value **x**, to vary our steps!
- Checks: base, base + **x**, base + 2**x**, ... etc



(Self-read)

Open Addressing - Double Hashing

“SoC Cat” with **x** = 2



(Self-read)

Open Addressing - Double Hashing

“Bell Curve God” with **x** = 4

But his base slot is already empty.

SoC Cat	*Rar the Cat*	*Bell Curve God*	*Steven*	*Jacq the Dino*
Students	*Animals*	*Immortal Beings*	*Lecturers*	*Virtual Beings*

(Self-read)

Open Addressing

Implementation

We just need an *array* of elements.

However, the usual practice is to declare much more space than we need. (In general, >= 4 times :D)

Reduce time associated with repeatedly probing.

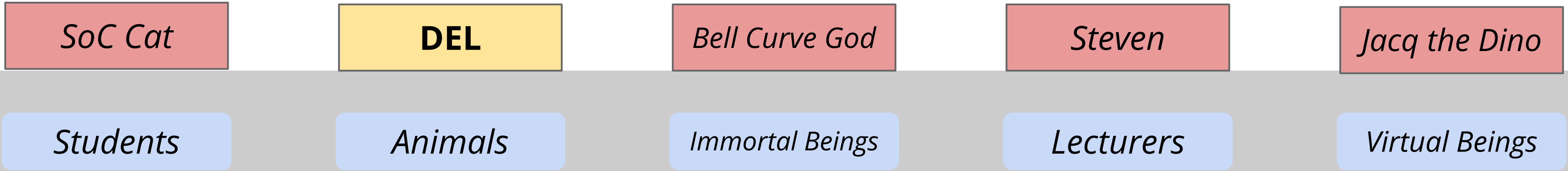
SoC Cat	*Rar the Cat*	*Bell Curve God*	*Steven*	*Jacq the Dino*
Students	*Animals*	*Immortal Beings*	*Lecturers*	*Virtual Beings*

(Self-read)

Open Addressing

Implementation

When deleting elements, we need to flag it as deleted instead.



(Self-read)

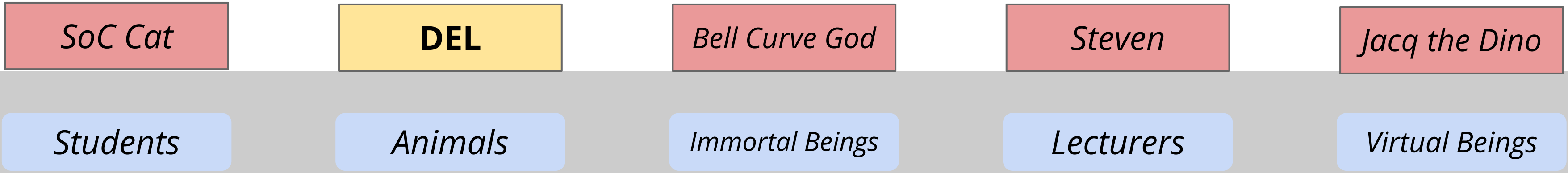
Open Addressing

Implementation

To **find** an element, we follow the same *path* we took when we add the element.

Continue until we find the element...

Or until we see an empty slot.



(Self-read)

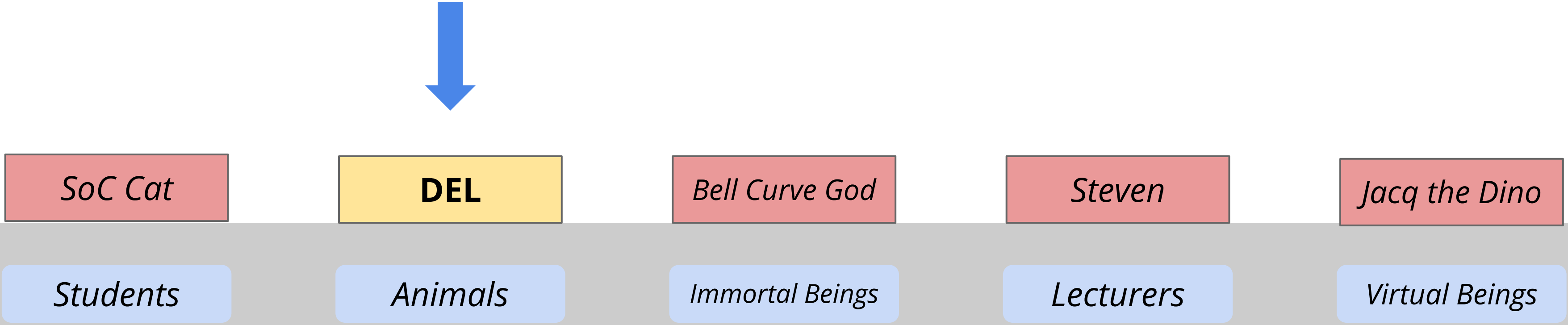
Open Addressing

Implementation

During ***find*** operation,

we cannot treat ‘deleted’ elements as empty.

Why?



(Self-read)

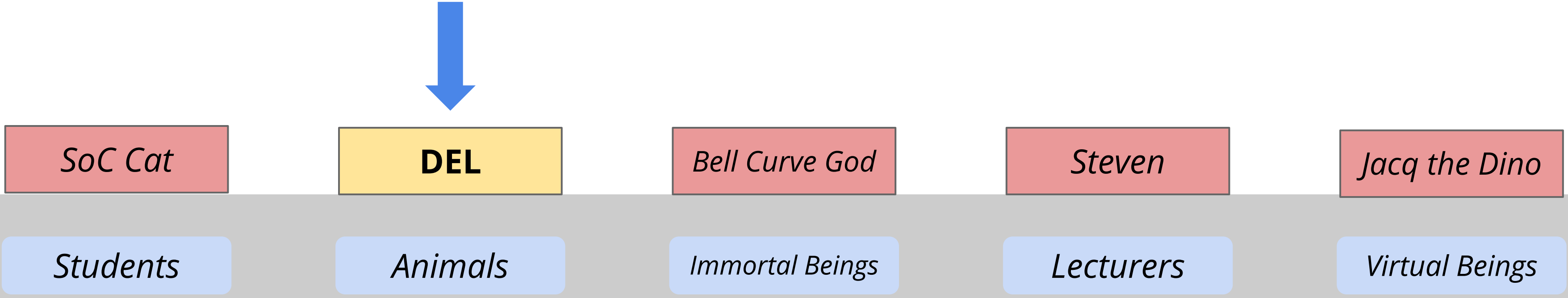
Open Addressing

Implementation

Why?

Imagine if we want to find “SoC Cat” below.

We will stop at the first index if we don’t check beyond the deleted marker.



(Self-read)

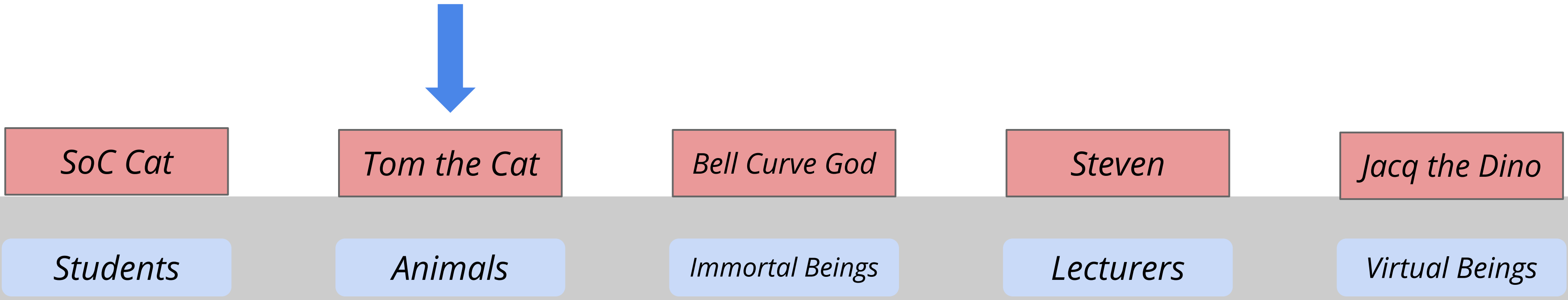
Open Addressing

Implementation

During ***insert*** operation,

We **<u>can</u>** overwrite the deleted markers.

Does not affect our ***find*** operation.



LP vs QP vs DH vs SC

Open Addressing

- Require deleted markers
- Inefficient if there are many deletions and insertions

Separate Chaining

- Unable to fully utilize unused addresses/'slots'

LP vs QP vs DH vs SC

Open Addressing

- Can rehash every 2^K operations (why?)
- More cache friendly (esp LP)
  - Spatial locality

Separate Chaining

- Easy to get obtain elements with same hash

Actual Implementations

C++

- unordered_map uses Separate Chaining
  - Linear space to number of elements inside

Java

- Also uses Separate Chaining
  - Optimized to use Binary Search Tree within each slot if there are too many collisions

Hash Table Applications

Key-Value Mappings

Q2 Part 1

A *mini* population census is to be conducted on every person in your (*not so large*) neighbourhood.

We are only interested in storing every person's **name and age**.

→ **age** is *Integer*, **name** is *String*

Retrieve **age** by **name**.

Retrieve **name(s)** by **age**.

Q2 Part 1

To retrieve **age** by **name**:

<Key, Value> : <**name**, **age**>

h(**name**): Standard string hashing method

https://visualgo.net/en/hashtable?slide=4-7

Q2 Part 1

To retrieve **age** by **name**:

<Key, Value> : <**name**, **age**>

Collision resolution:

- Double Hashing
  - Unlikely to use up all the ‘slots’

Q2 Part 1

To retrieve **name(s)** by **age**:

<Key, Value> : <**age**, **name**>

h(**age**) = **age**

Direct Addressing Table (aka just-an-array)

Q2 Part 1

To retrieve **name(s)** by **age**:

<Key, Value> : <**age**, **name**>

Collision resolution:

- Separate Chaining
  - Already multiple values for the same *key*, no choice

Q2 Part 2

A *much larger* population census is also conducted across the country.

We are only interested in storing every person's **name and age**. → **age** is *Integer*, **name** is *String*

Retrieve **name(s)** of people who are **X** = 17 years or older.

Q2 Part 2

Retrieve **name(s)** of people who are **X** = 17 years or older:

- Age is likely to be from [0, 120]
- We can still use Direct Addressing Table
  - With separate chaining (like Q2 P1)
- Loop through all possible ages ≥ **X**

Q2 Part 3

A *different* population census is conducted across the country.

We are only interested in storing every person's **name and age**. → **age** is *Integer*, **name** is *String*

However, we now want to retrieve **name and age** of people with a given **last name**.

Q2 Part 3

To retrieve **name(s)** by **last name**:

<Key, Value> : <**last name**, **person**>

Person = *pair* of (**name**, **age**)

Collision resolution:

- Double Hashing (unlikely to have *that many* different last names)
- Separate Chaining

Q2 Part 4

A grades management program stores a student’s **index number** and his/her **final marks** in a module. There are 1,000,000 students, each scoring final marks in [0.0, 100.0].

Store **all** the student’s performance.

Print the list of students in **ranking order** that are *more than 65.5.*

Q2 Part 4

Not really...

If

<Key, Value>: <**index number**, **grades**> ...

Then we cannot get list of students in ranking order, without looping through everything, and then sort.

(Self-read)

Q2 Part 4

Not really…

If <Key, Value>: <**grades, index number**> …

To deal with issues with *floating points,* we can round grades to a certain precision (3 d.p.) and converting them to integer. Eg: 98.234 → 98234

Iterating through all possible scores > 65.5 is still quite a lot.

Binary Search Trees

Non-Balanced

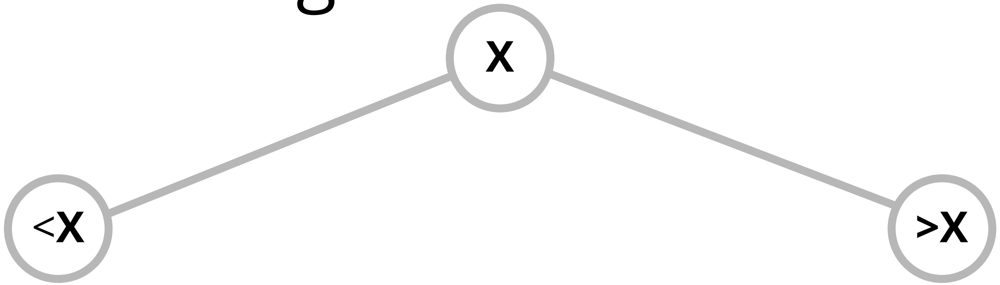
General Properties of BST

Main Idea

For every vertex, **X**:

Left child is smaller than **X**.

Right child is larger than **X**.



General Properties of BST

A BST is also a *binary tree*.

All vertex has at most 2 children vertices.

A BST is **not** <u>always</u> a *complete* binary tree.

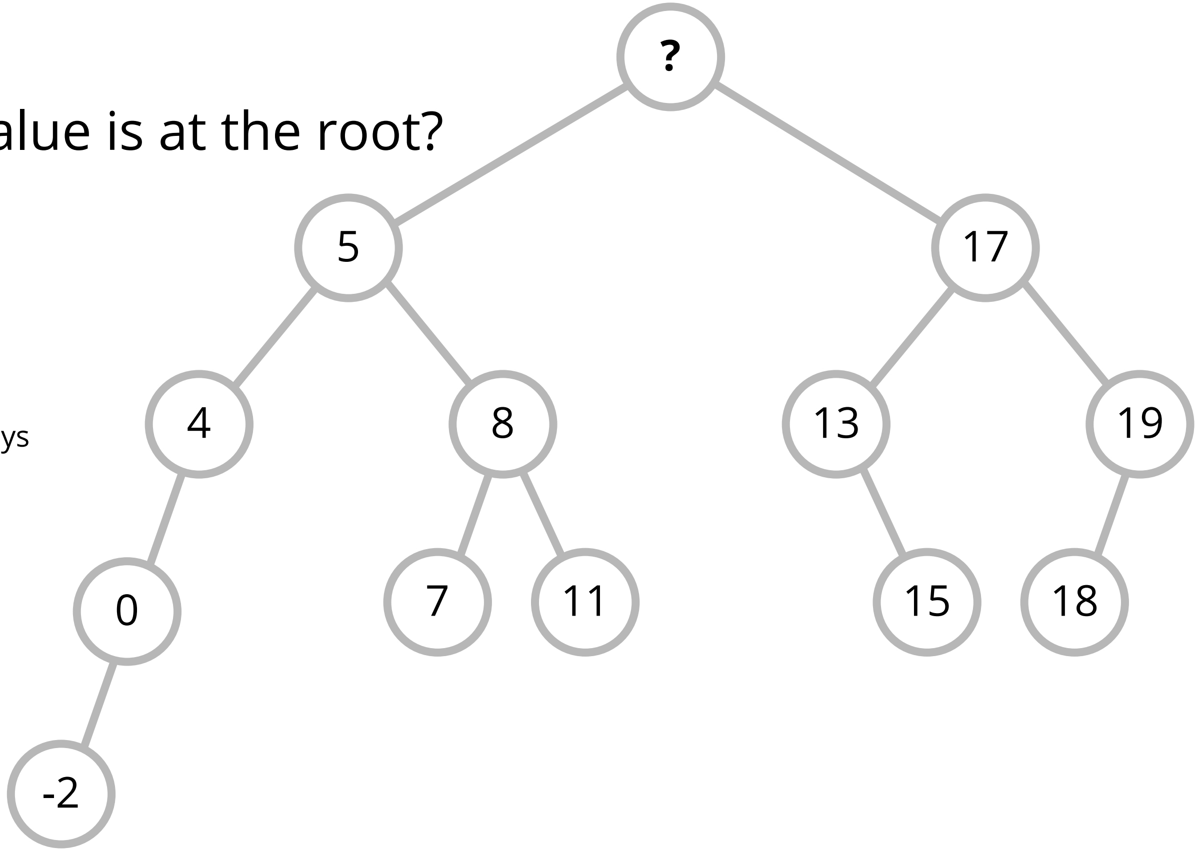
Is a **b**BST <u>always</u> a *complete* binary tree?

BST

What value is at the root?

A) 10
B) 11
C) 12
D) 13

PS: Integer keys

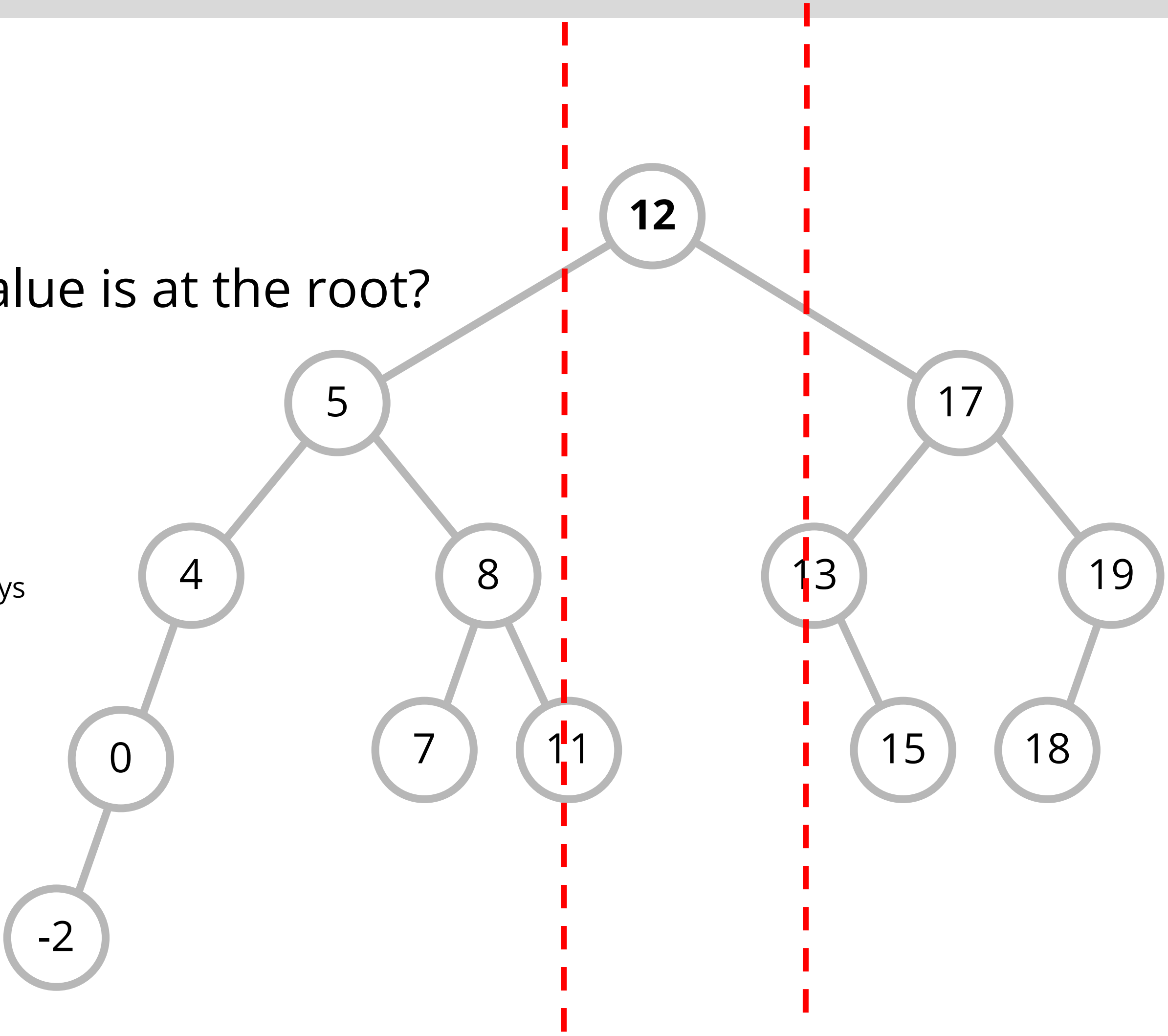


BST

What value is at the root?

A) 10
B) 11
C) 12
D) 13

PS: Integer keys



General Properties of BST

- **Unique** key
  - How would you handle duplicate keys?
  - Store frequency
    - Other ways exist (separate chaining?)

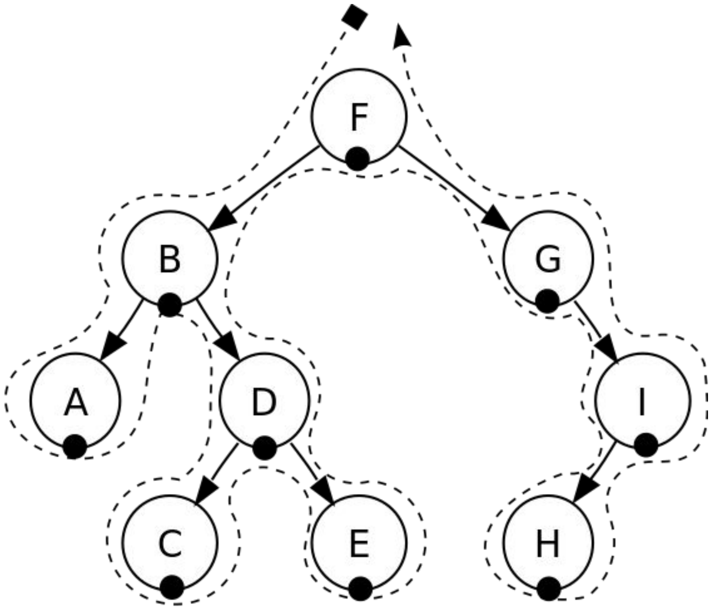
- **In-order** traversal is in order

Tree Traversals [Credits: SG IOI Training 2017]

In-order traversal

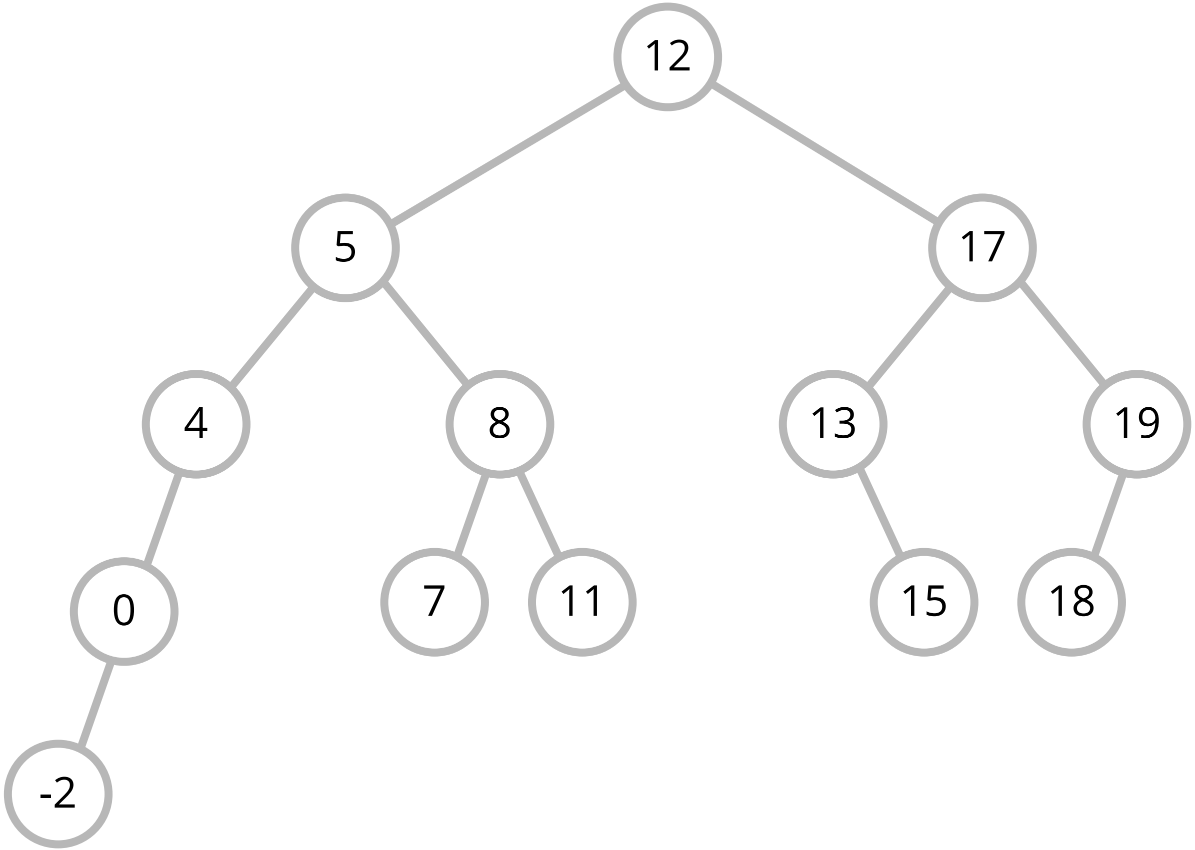
- Perform operations on the vertex after completing the left subtree, but before commencing the right subtree.

In-order Traversal [Credits: SG IOI Training 2017, Wikipedia]

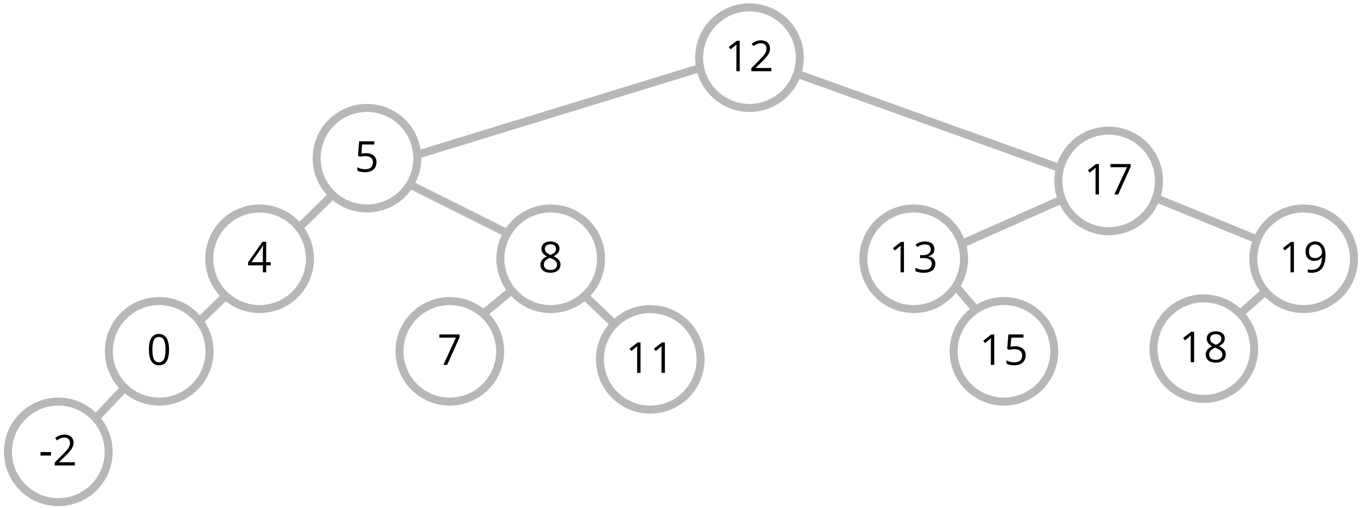


A B C D E F G H I

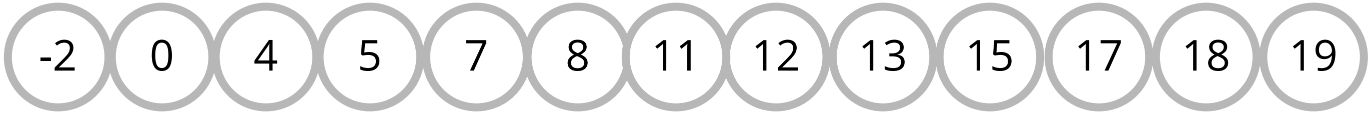
BST



BST



BST



BST

– **Search(v) / Insert(v) / Remove(v)**

Start from the root of the tree, “follow the path”:

- If **v** is smaller than node, go left
- If **v** is larger than node, go right

Time complexity: **O(H)**

https://visualgo.net/en/bst

Remove(v)

When a node with 2 children is to be removed:

- It is replaced by its ***successor***
- Remove the ***successor*** from its original place

Remove(v)

Qn 1: Can it be replaced by its ***predecessor***?

Remove(v)

Qn 1: Can it be replaced by its ***predecessor***?

Yes, this works as well.

Remove(v)

Qn 2: What if there is no ***successor***?

Remove(v)

Qn 2: What if there is no ***successor***?

Then it will be the maximum element of the BST.

The maximum element of the BST will **not** have 2 children.

Remove(v)

Qn 3: What if the ***successor*** has 2 children?

Remove(v)

Qn 3: What if the ***successor*** has 2 children?

In short, this is not possible.

Remove(v)

Qn 3: What if the ***successor*** has 2 children?

That means the successor has a left child and a right child.

The left child must be less than the ***successor***.

Then it must be the deleted vertex itself!

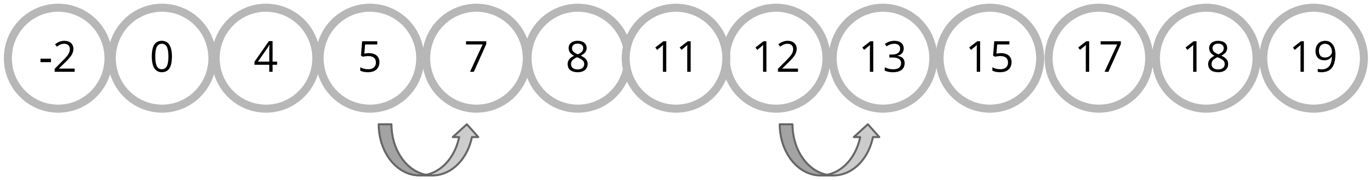
Then the deleted vertex will have only left child.

BST

- Find **successor**
  - Next *highest* key
  - *Next* vertex of in-order traversal.

BST

- Find **successor**
  - Next *highest* key
  - *Next* vertex of in-order traversal.



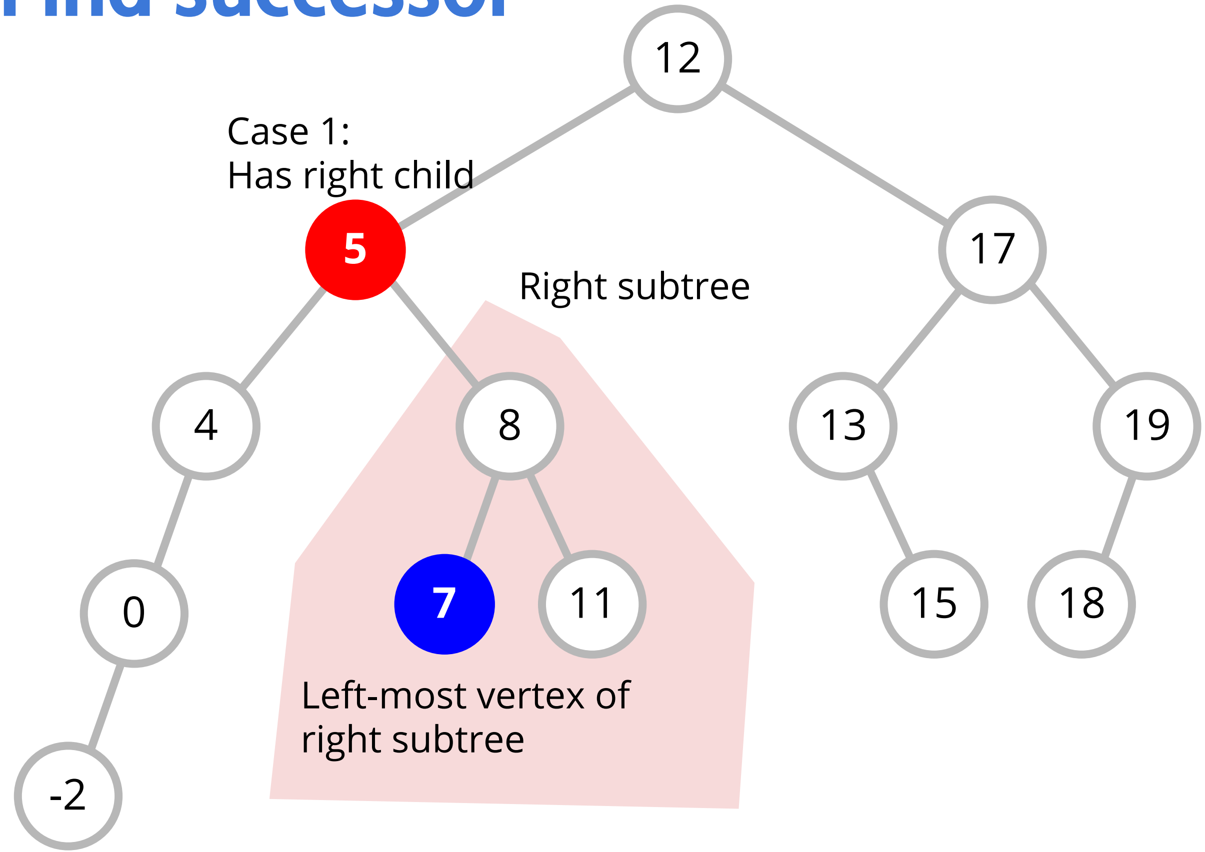
BST: Find successor

2 cases:

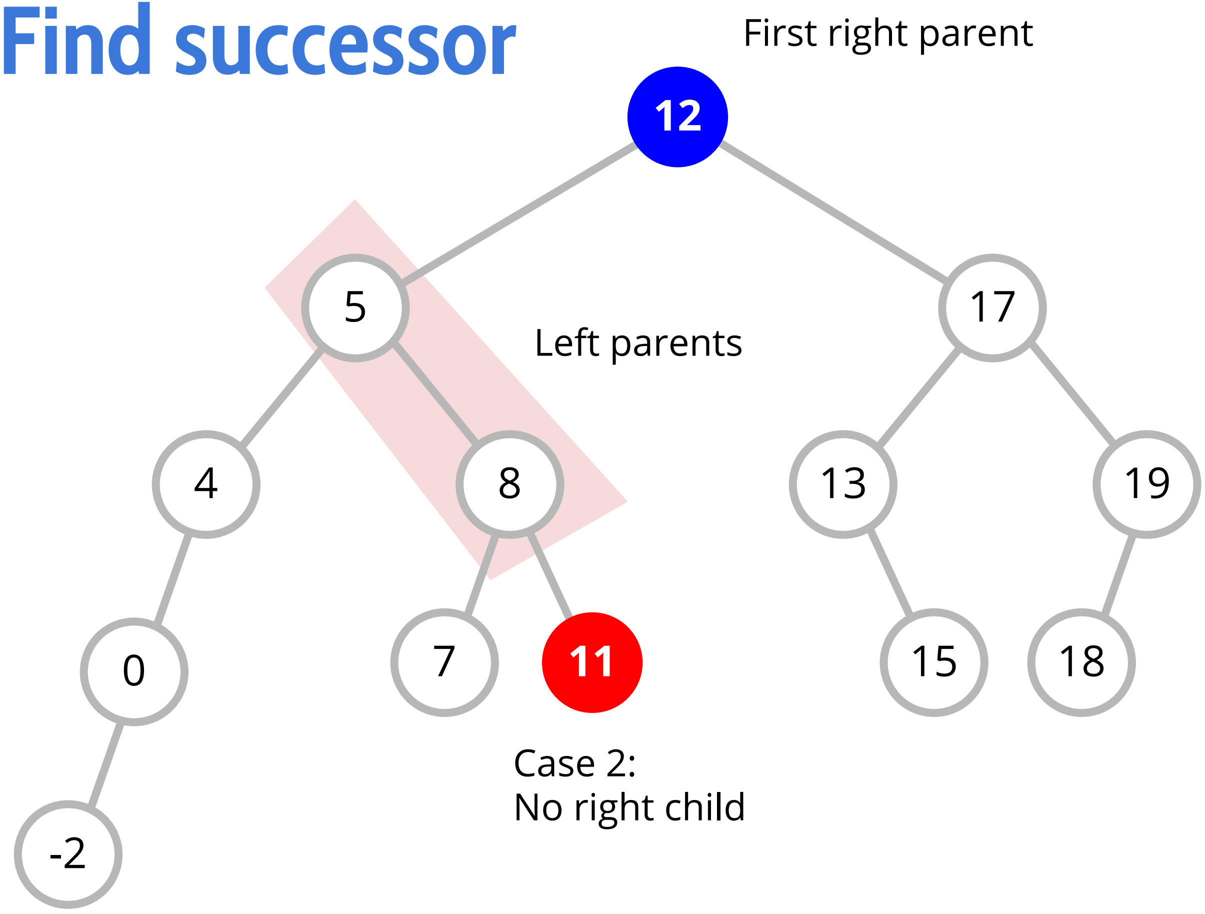
1. Has **right** child
   a. **Left-most** vertex in **right** subtree

2. No **right** child
   a. First **right** parent
   b. What if there isn't a **right** parent?

BST: Find successor

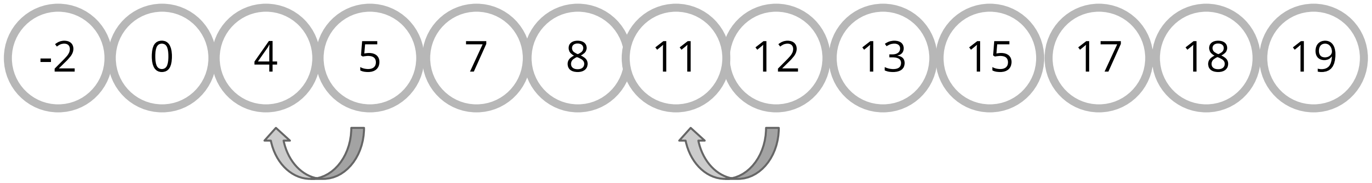


BST: Find successor



BST

- Find **predecessor**
  - Next *lowest* key
  - *Previous* vertex of in-order traversal.
- *Inverse* operation of successor



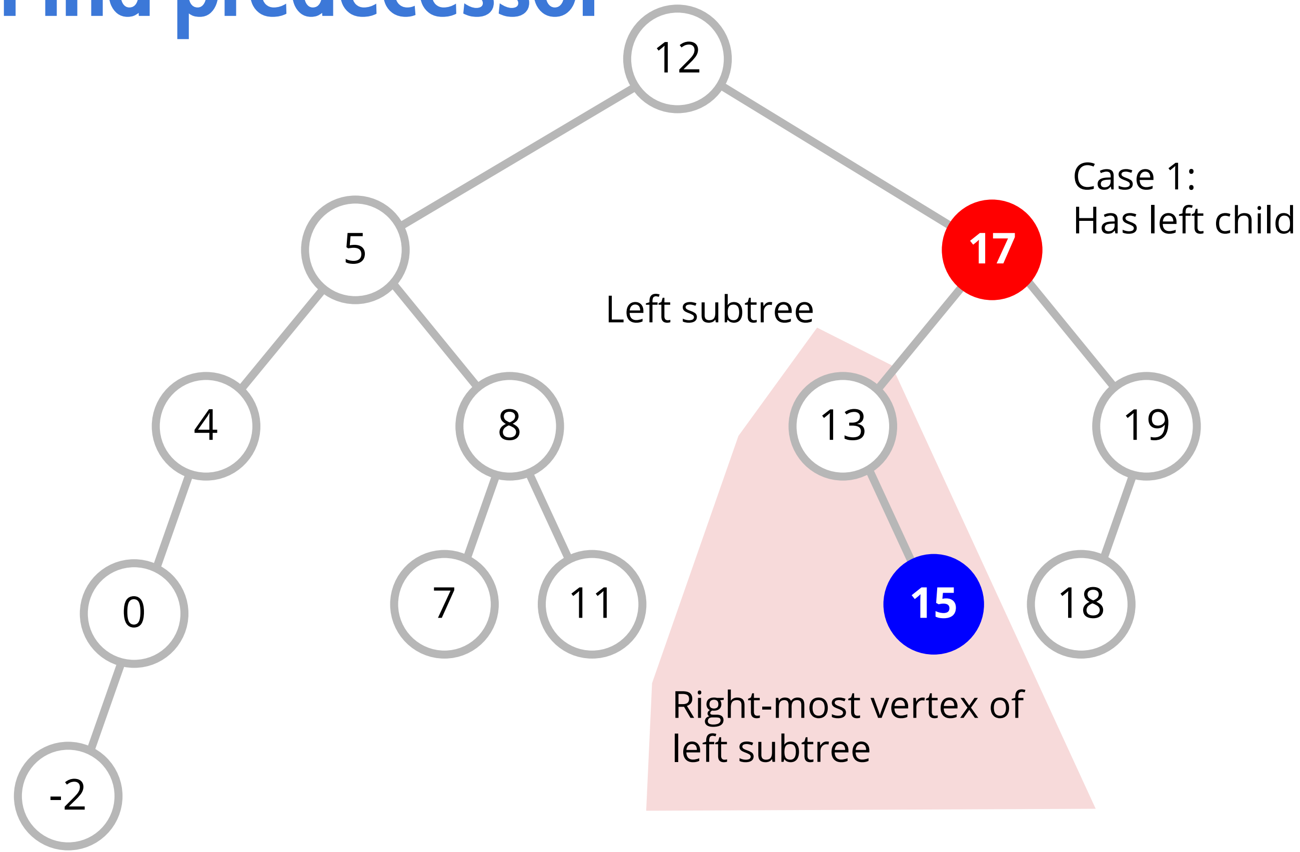
BST: Find predecessor

2 cases:

1. Has **left** child
   a. **Right-most** vertex in **left** subtree

2. No **left** child
   a. First **left** parent

BST: Find predecessor



BST: Find predecessor