CS 106 INTRODUCTION TO DATA STRUCTURES

SPRING 2020

PROF. SARA MATHIESON

HAVERFORD COLLEGE

ADMIN

- **Lab 6 due ~~Sunday~~ Tuesday**

- **Welcome prospective students!**

- **May need to do random breakout rooms since many people are not signed into Zoom**

Remind me to RECORD

REVISED TA/OFFICE HOURS

Sunday 7-9pm (Juvia)

Monday 8-midnight (Steve)

Tuesday 11:30-12:30pm (Lizzie)

Tuesday 4:30-6pm (Sara)

Wednesday 8-midnight (Steve)

Today/Tomorrow

Thursday 11:30-12:30pm (Lizzie)

Thursday 9-11pm (Will)

Friday 8-10pm (Gareth)

Saturday 4-6pm (Will)

Saturday 8-10pm (Gareth)

APR 7 OUTLINE

- Heap sort recap and example
- Motivation for hash maps
- Hash functions
- Implementing a hash table

APR 7 OUTLINE

- **Heap sort recap and example**
- **Motivation for hash maps**
- **Hash functions**
- **Implementing a hash table**

IN-PLACE SORTING

In-place sorting algorithm: we do not create a new data structure, we instead sort the elements within their existing data structure

- Cons: destroys the original order, which may have been important
- Pros: very efficient in terms of space

IN-PLACE SORTING

In-place sorting algorithm: we do not create a new data structure, we instead sort the elements within their existing data structure

- Cons: destroys the original order, which may have been important
- Pros: very efficient in terms of space

Out-of-place sorting algorithm: returns a new data structure with the original data sorted

- Cons: space inefficient
- Pros: preserves original order

IN-PLACE SORTING

In-place sorting algorithm: we do not create a new data structure, we instead sort the elements within their existing data structure

- Cons: destroys the original order, which may have been important
- Pros: very efficient in terms of space

Out-of-place sorting algorithm: returns a new data structure with the original data sorted

- Cons: space inefficient
- Pros: preserves original order

Heap Sort can be implemented either way, but we will cover the in-place version now

HEAP SORT (IN PLACE WITH ARRAY)

Phase I: unsorted array -> heap

for i = 0, 1, ... n-1:

bubble up element at index i until arr[0...i] form a heap

HEAP SORT (IN PLACE WITH ARRAY)

Phase I: unsorted array -> heap

for i = 0, 1, ... n-1:

bubble up element at index i until arr[0...i] form a heap

Phase II: heap -> sorted array

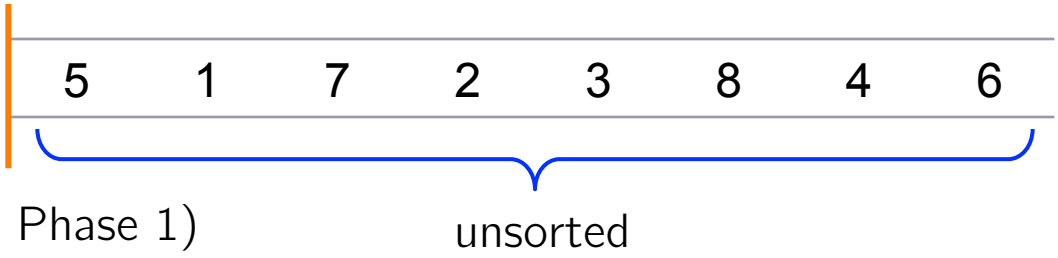
for i = n-1, n-2, ... 0:

swap(0, i) // 0 is the root index

bubble down so arr[0...i] are still a heap

HEAP SORT EXAMPLE: PHASE I

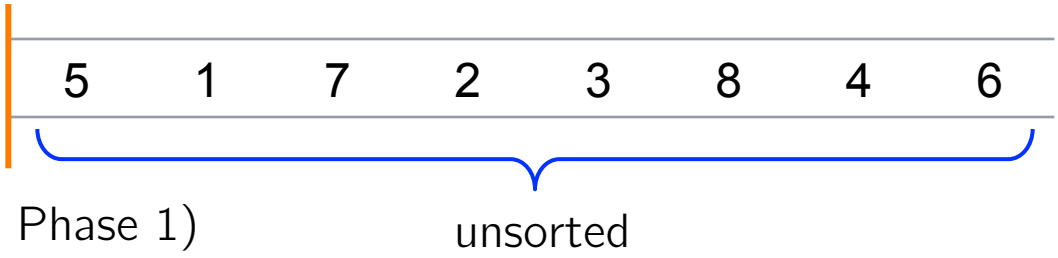
Phase I: unsorted array -> heap



(Below are two different stages in Phase 1)

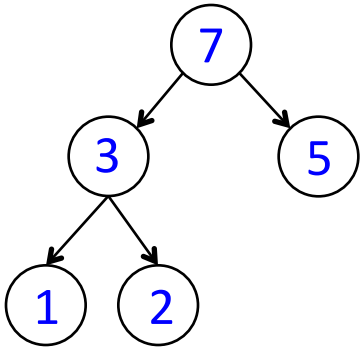
HEAP SORT EXAMPLE: PHASE I

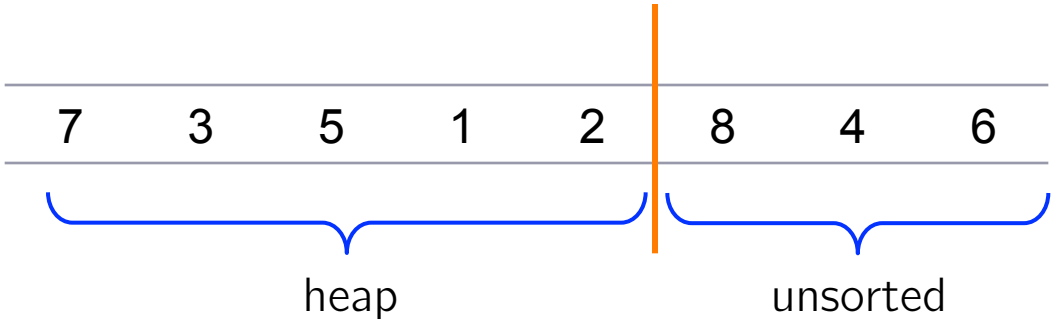
Phase I: unsorted array -> heap



(Below are two different stages in Phase 1)

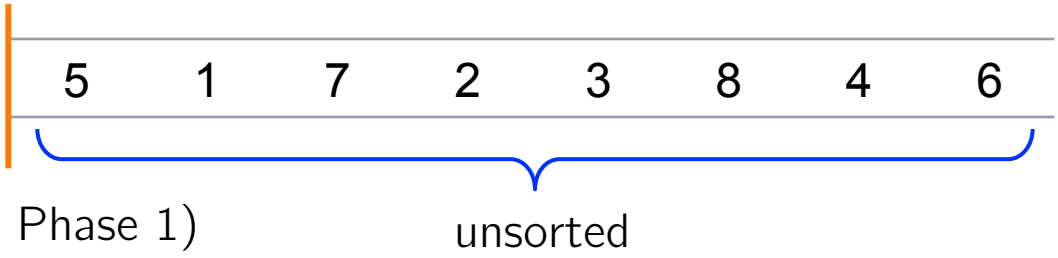
After processing i=4:





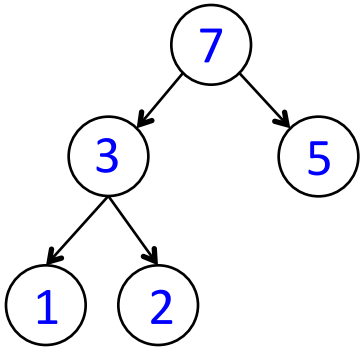
HEAP SORT EXAMPLE: PHASE I

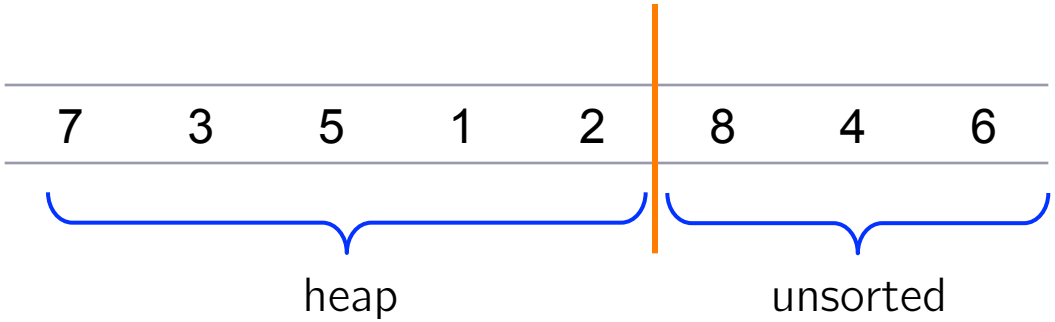
Phase I: unsorted array -> heap



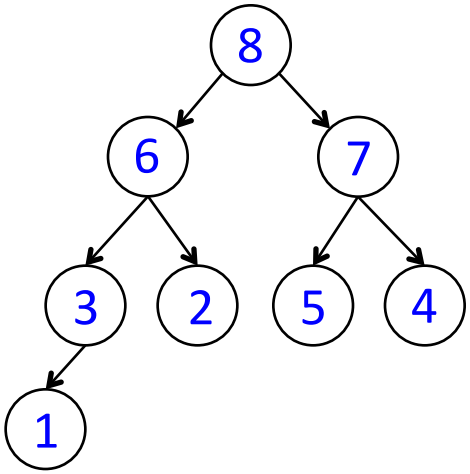
(Below are two different stages in Phase 1)

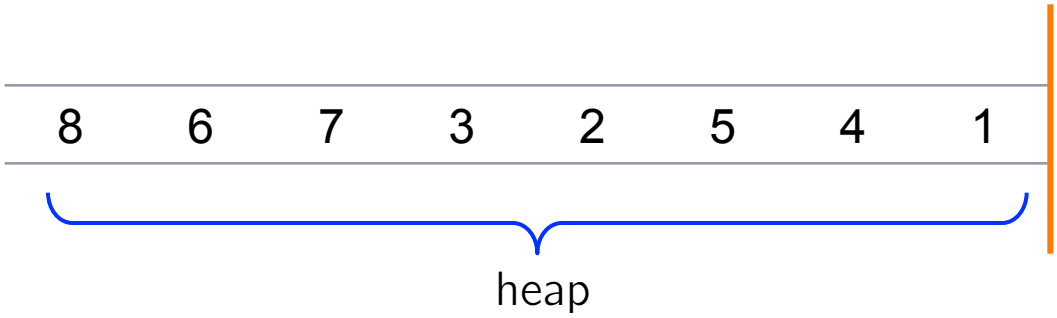
After processing i=4:





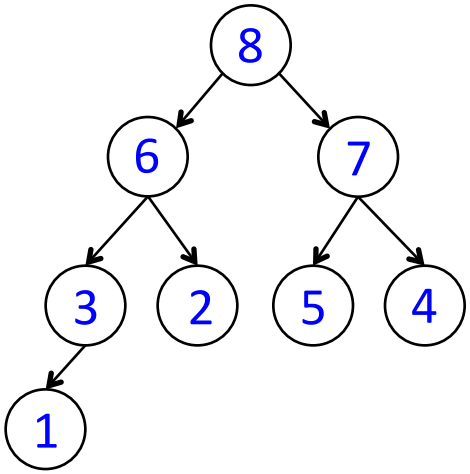
After processing i=n-1 (end of Phase I):





HEAP SORT EXAMPLE: PHASE I

Phase I: unsorted array -> heap

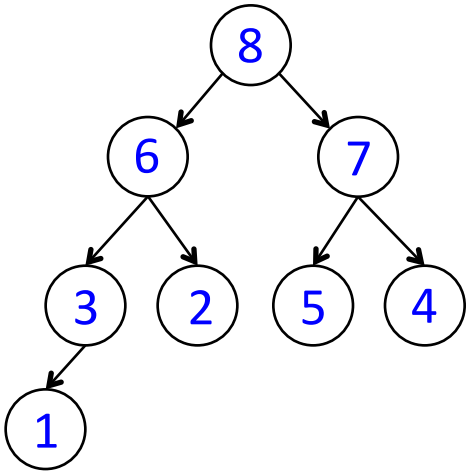


5	1	7	2	3	8	4	6
5	1	7	2	3	8	4	6
7	1	5	2	3	8	4	6
7	2	5	1	3	8	4	6
7	3	5	1	2	8	4	6
8	3	7	1	2	5	4	6
8	3	7	1	2	5	4	6
8	6	7	3	2	5	4	1

HEAP SORT EXAMPLE: PHASE II

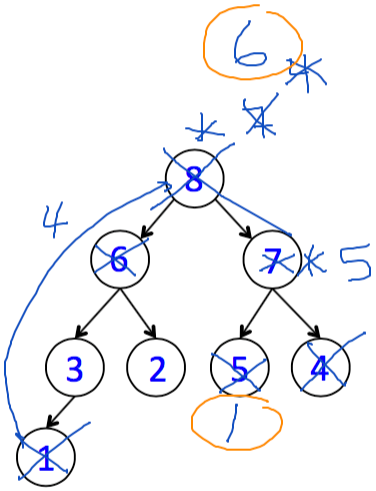
Phase II: heap -> sorted array

8	6	7	3	2	5	4	1



HEAP SORT EXAMPLE: PHASE II

Phase II: heap -> sorted array



~~8~~	~~6~~	~~7~~	3	2	~~5~~	~~4~~	~~1~~
*	4	*			*	7	8
*		5			6		
*							
6							
1							

Sorted

bubble down

next swap ?

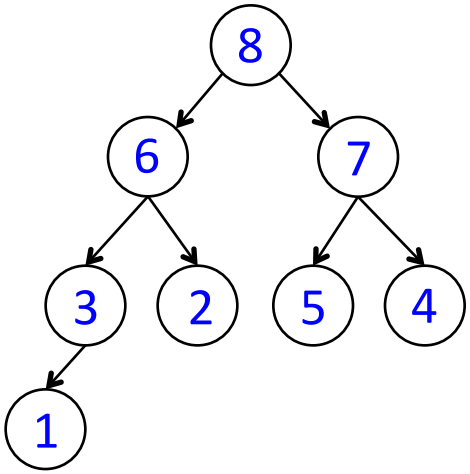
left : 2i+1

right : 2i+2

bubbleDown(end)

index (stop)

RUNTIME KEY IDEA: TREE HEIGHT



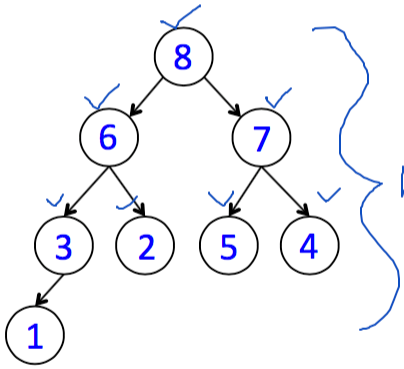
Mute | Stop Video | Manage Participants 32 | Polling | New Share | Pause Share | Annotate | Remote Control | More

ID: 483-963-860 | Stop Share

Mouse | Select | Text | Draw | Stamp | Spotlight | Eraser | Format | Undo | Redo | Clear | Save

RUNTIME KEY IDEA: TREE HEIGHT

n = # nodes



height? $= x$

$$2^0 + 2^1 + 2^2 + 2^3 + \cdots 2^x = n$$

$$2^1 + 2^2 + \cdots 2^x + 2^{x+1} = 2n$$

$$2^{x+1} - 1 = n$$

$$x \approx \log(n)$$

heap sort: $O(n \log n)$

APR 7 OUTLINE

- Heap sort recap and example
- **Motivation for hash maps**
- Hash functions
- Implementing a hash table

MAP

A searchable collection of key-value pairs

Multiple entries with the same key are not allowed

Also known as dictionaries, hash tables, etc

EXAMPLES OF MAPS

- A book in a library has a physical position (like an index in an array)
- It would be fast to find if you knew this position, but...

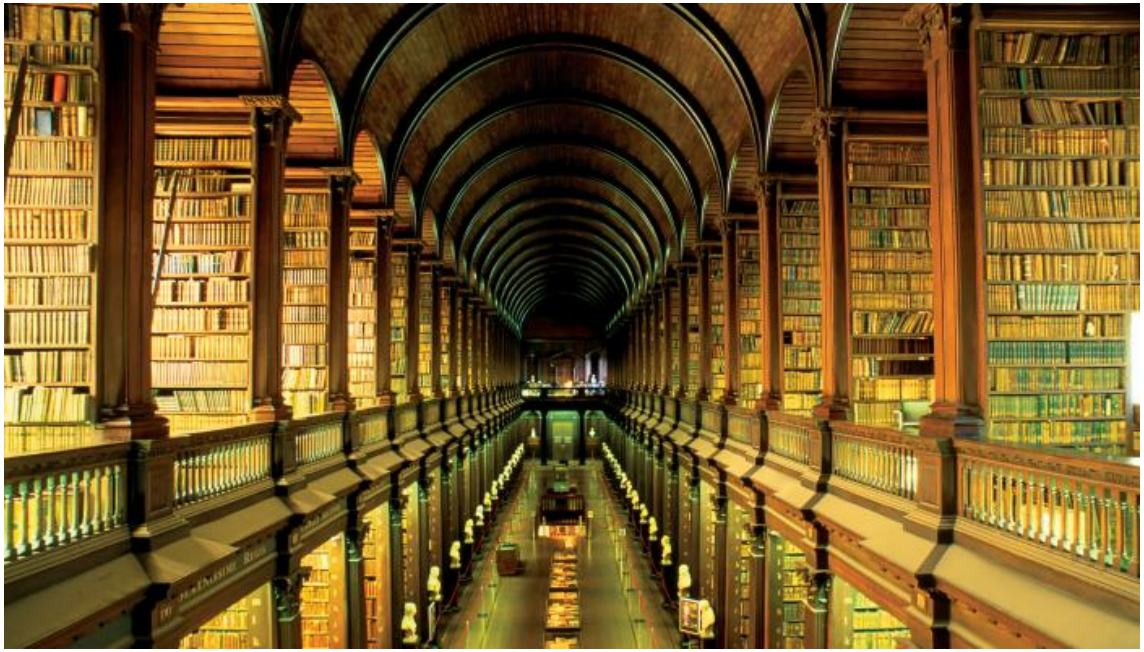


Image from: http://aetherforce.com/who-really-burned-the-library-of-alexandria-by-preston-chesser/

EXAMPLES OF MAPS

- A book in a library has a physical position (like an index in an array)
- It would be fast to find if you knew this position, but...
- You shouldn't need to already know that when you walk into the library – you can look it up!

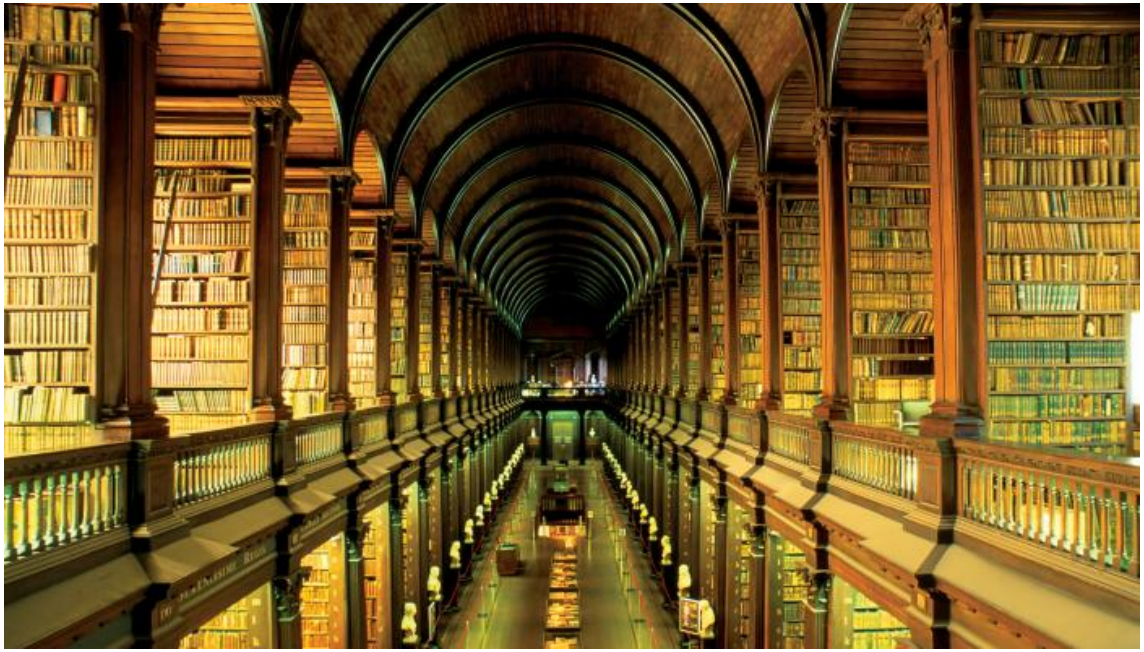


Image from: http://aetherforce.com/who-really-burned-the-library-of-alexandria-by-preston-chesser/

EXAMPLES OF MAPS

- A book in a library has a physical position (like an index in an array)
- It would be fast to find if you knew this position, but...
- You shouldn't need to already know that when you walk into the library – you can look it up!
- Look up a book in the library by title
  - Key: title
  - Value: physical book

Key idea: hash function maps key to position

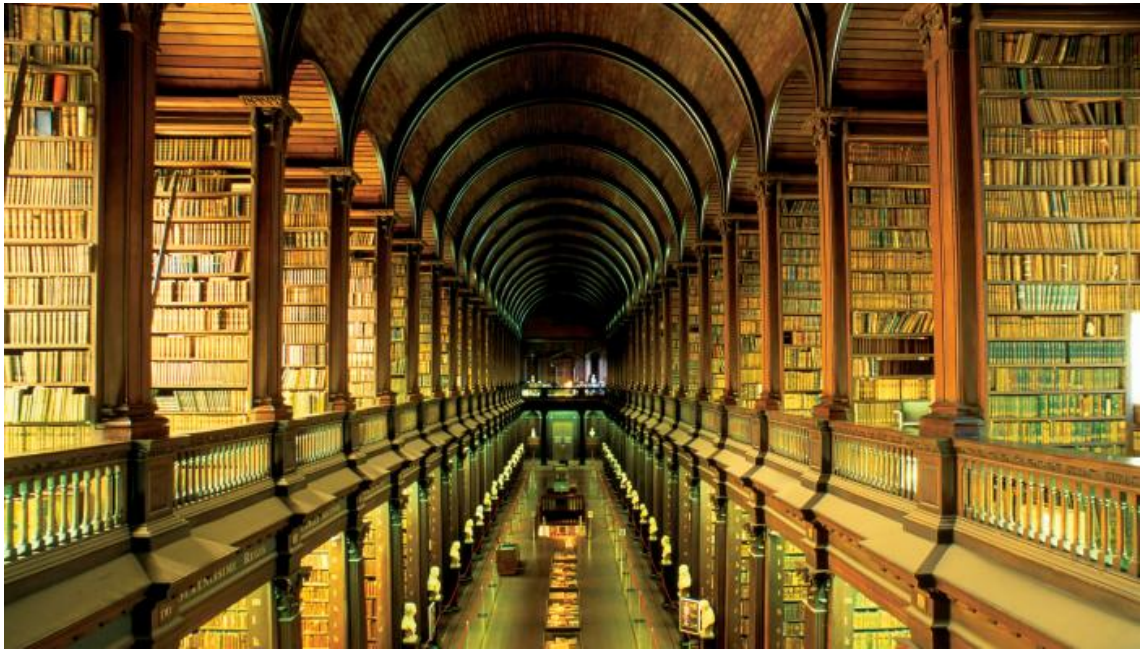


Image from: http://aetherforce.com/who-really-burned-the-library-of-alexandria-by-preston-chesser/

EXAMPLES OF MAPS

- **Look up students by name or ID number**
  - Key: student name or ID number
  - Value: class schedule, transcript, dorm, dean, etc

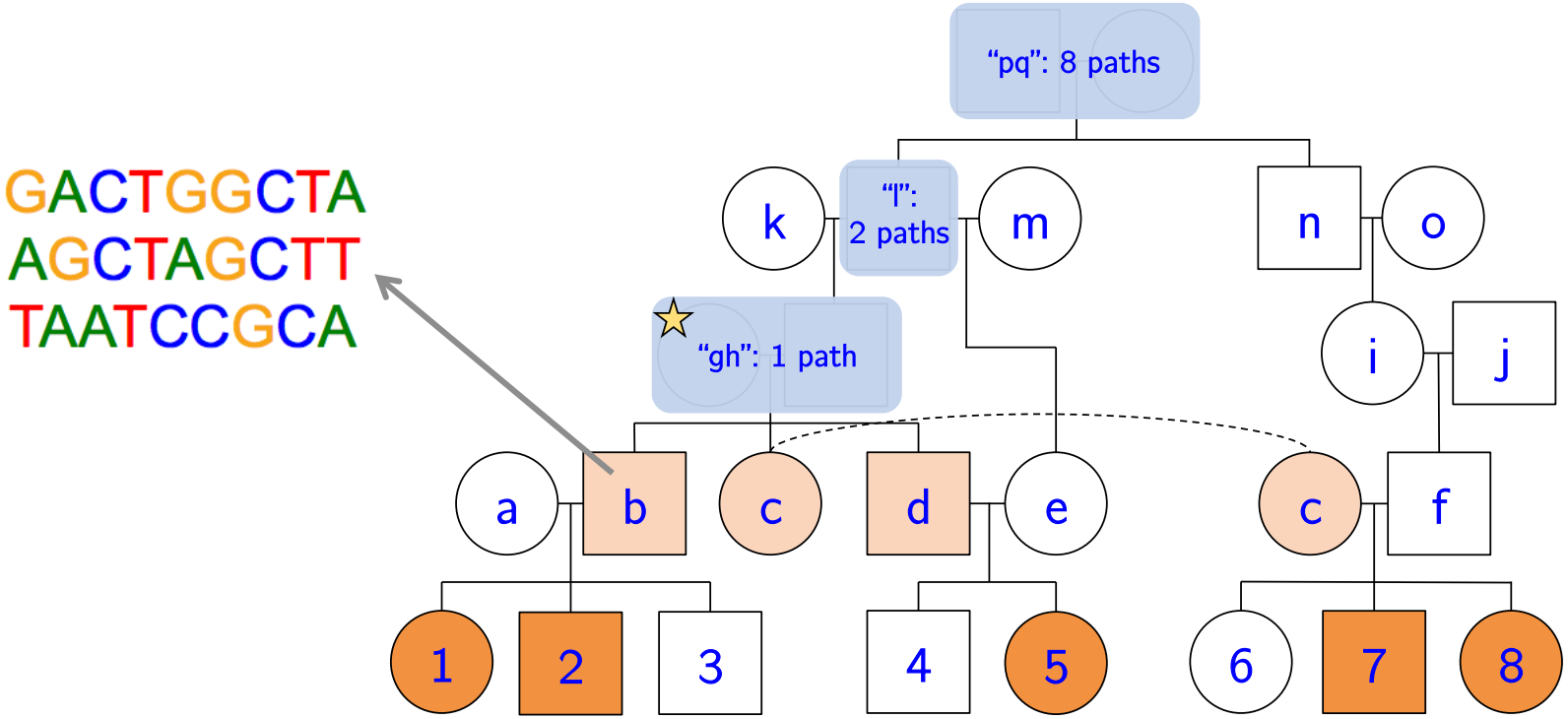
- **Look up by Social Security Number**
  - Key: SSN
  - Value: tax records, voting data, etc

- **User accounts**
  - Key: email or username
  - Value: all account data (tweets, friends, pictures, etc)

BIOLOGY EXAMPLE

Look up individuals in a pedigree or other dataset

- Key: ID number (i.e. “b”)
- Value: DNA data



MAP ADT (MANY WAYS OF DEFINING)

`lookup(k)`: if the map `M` has an entry with key `k`, return its associated value; else, return null

MAP ADT (MANY WAYS OF DEFINING)

`lookup(k)`: if the map `M` has an entry with key `k`, return its associated value; else, return null

`insert(k, v)`: insert entry `(k, v)` into the map `M`; if key `k` is not already in M, then return null; else, replace old value with `v` and return old value associated with `k`

MAP ADT (MANY WAYS OF DEFINING)

`lookup(k)`: if the map `M` has an entry with key `k`, return its associated value; else, return null

`insert(k, v)`: insert entry `(k, v)` into the map `M`; if key `k` is not already in M, then return null; else, replace old value with `v` and return old value associated with `k`

`remove(k)`: if the map `M` has an entry with key `k`, remove it from `M` and return its associated value; else, return null

MAP ADT (MANY WAYS OF DEFINING)

`lookup(k)`: if the map `M` has an entry with key `k`, return its associated value; else, return null

`insert(k, v)`: insert entry `(k, v)` into the map `M`; if key `k` is not already in M, then return null; else, replace old value with `v` and return old value associated with `k`

`remove(k)`: if the map `M` has an entry with key `k`, remove it from `M` and return its associated value; else, return null

`size()`, **`isEmpty()`**

`entrySet()`: return an iterable collection of the entries in `M`

`keySet()`: return an iterable collection of the keys in `M`

`values()`: return an iterable collection of the values in `M`

APR 7 OUTLINE

- Heap sort recap and example
- Motivation for hash maps
- **Hash functions**
- Implementing a hash table

HASH TABLES

Maps are an *abstract data type*. Hash tables are one way (the most common way) to *implement* maps.

We need to know how maps are implemented in order to determine how long it takes to perform each of the main operations:

- **inserting or changing a value at a given key**
- **getting a value at a given key**

HASH TABLES

Suppose we have a function that can map a key to a memory location (a space in our “library”).

To insert / change a key / value pair we:

1. **Calculate the result of the hash function on the key.**
2. **Find that location in memory and insert / change the value.**

We can do the same thing to get a value given a key.

HASH TABLES – A SKETCH OF THE DETAILS

There are many details we need to get right for this to work:

1. The space in memory needs to be not too big and not too small, and the hash function needs to map into that specific amount of space.
2. Given the limited space, we might map different keys to the same location. We need a plan for this.
3. Hash functions need to be easy to compute and distribute the keys uniformly across the memory, or we'll have problem #2 a lot.

If we do this all correctly, then we can insert, change, and get key / value pairs very quickly!

HASH FUNCTIONS AND TABLES

A hash function h maps a key to integers in a fixed interval $[0, N-1]$

$h(x) = x \% N$ is such a function for integers

$h(x)$ is the hash value of key x

A hash table is an array of size N

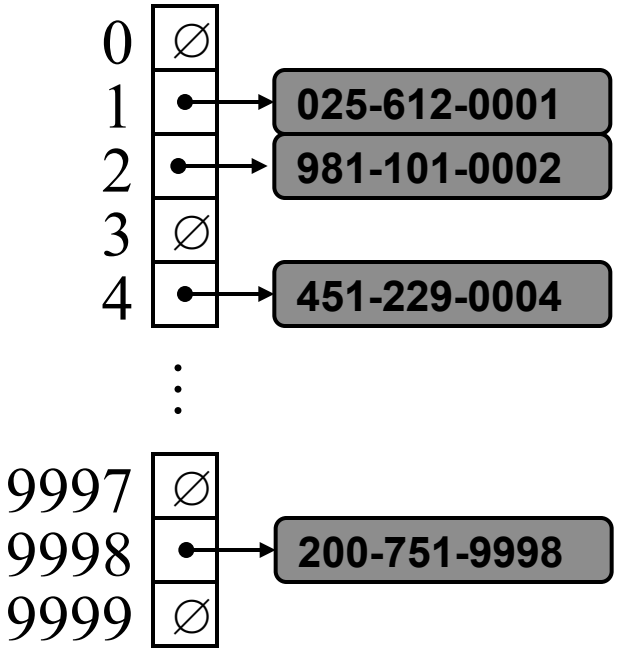
- associated hash function h
- item (k, v) is stored at index $h(k)$

EXAMPLE

A hash table storing entries as (SSN, Name), where SSN is a nine-digit positive integer

Use an array of size $N = 10000$ and the hash function

$h(x) =$ last 4 digits of x



COLLISION HANDLING 1: CHAINING

A hash function does not guarantee one-to-one mapping – no hash function does

One option: when more than one key hashes to the same index, we have a "bucket"

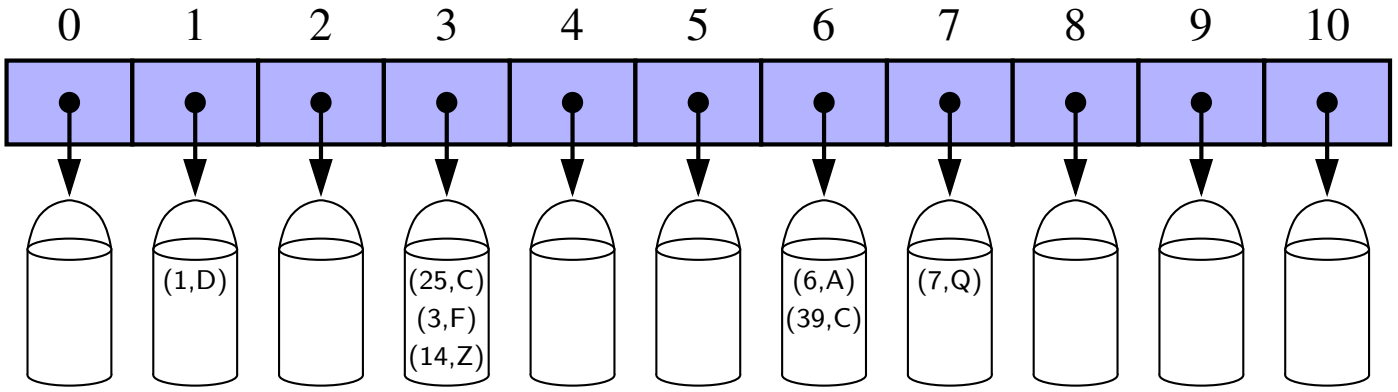
Each index holds a collection of entries

COLLISION HANDLING 1: CHAINING

A hash function does not guarantee one-to-one mapping – no hash function does

One option: when more than one key hashes to the same index, we have a "bucket"

Each index holds a collection of entries

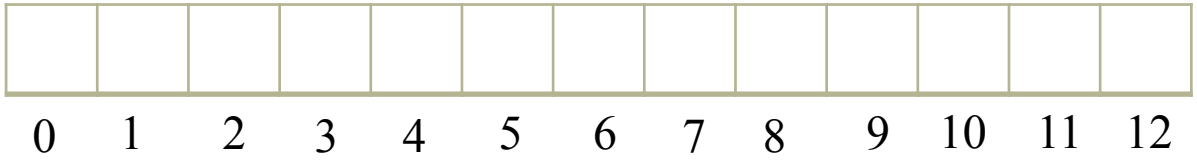


COLLISION HANDLING 2: PROBING

Colliding item is put in a different cell

Example: $h(x)=x\%13$

insert 18, 41, 22, 44, 59, 32, 31, 73



Linear probing: place the colliding item in the next (circularly) available table cell

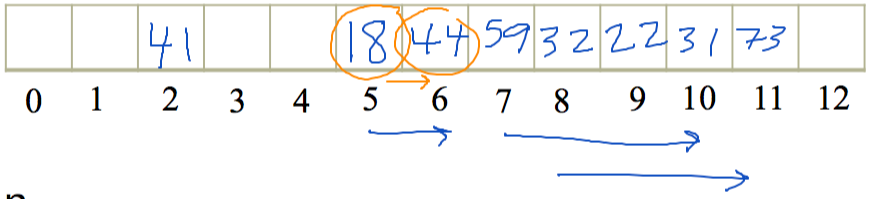
Colliding items cluster together

COLLISION HANDLING 1: PROBING

Colliding item is put in a different cell

Example: $h(x)=x\%13$ (18 → 5)

insert 18, 41, 22, 44, 59, 32, 31, 73



Linear probing: place the colliding item in the next (circularly) available table cell

$x = 44$

$44 \% 13 = 5$

locate

Colliding items cluster together

GROUP EXERCISE

Say we have a very small hash table that contains the ID numbers of TAs, mapped to their name.

```
Table hashTable = new Table(5);
hashTable.insert(31,"Juvia");
hashTable.insert(13,"Steve");
hashTable.insert(100,"Will");
hashTable.insert(75,"Gareth");
hashTable.insert(28,"Lizzie");
```

Draw what the hash table will look like after the above code is executed, assuming we use a hash function that takes the key mod the length of the array.

GROUP EXERCISE

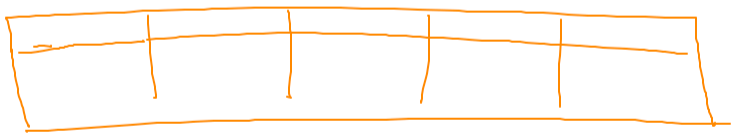
Say we have a very small hash table that contains the ID numbers of TAs, mapped to their name.

```
Table hashTable = new Table(5);
hashTable.insert(31,"Juvia");
hashTable.insert(13,"Steve");
hashTable.insert(100,"Will");
hashTable.insert(75,"Gareth");
hashTable.insert(28,"Lizzie");
```

key = int

value = String

Draw what the hash table will look like after the above code is executed, assuming we use a hash function that takes the key mod the length of the array.



GROUP EXERCISE

Say we have a very small hash table that contains the ID numbers of TAs, mapped to their name.

```
Table hashTable = new Table(5);
hashTable.insert(31,"Juvia");
hashTable.insert(13,"Steve");
hashTable.insert(100,"Will");
hashTable.insert(75,"Gareth");
hashTable.insert(28,"Lizzie");
```

Draw what the hash table will look like after the above code is executed, assuming we use a hash function that takes the key mod the length of the array.

APR 7 OUTLINE

- Heap sort recap and example
- Motivation for hash maps
- Hash functions
- **Implementing a hash table**

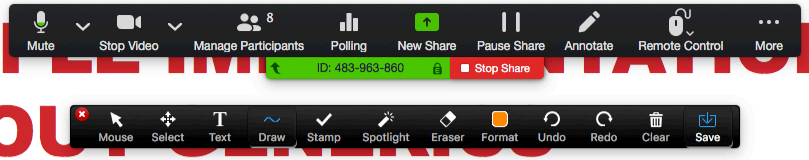
EXAMPLE IMPLEMENTATION WITHOUT GENERICS

```
private class TableRow {
    int key;
    String value;
    TableRow(int key, String value) {
        this.key = key;
        this.value = value;
    }
}
```

EXAMPLE IMPLEMENTATION WITHOUT GENERICS

```
public class Table {
    private TableRow[] rows;

    public Table(int tableSize) {
        rows = new TableRow[tableSize];
    }

    // TODO 1: return the default position (index) where this key is stored
    private int hash(int key) {


    }

    // TODO 2: locates the position (index) where the specified key can be found,
    // or where it should be inserted if it is not already in the table
    private int locate(int key) {




    }
```

EXAMPLE IMPLEMENTATION WITHOUT GENERICS




```
public class Table {
    private TableRow[] rows;

    public Table(int tableSize) {
        rows = new TableRow[tableSize];
    }

    // TODO 1: return the default position (index) where this key is stored
    private int hash(int key) {
          return key % rows.length;

    }

    // TODO 2: locates the position (index) where the specified key can be found,
    // or where it should be inserted if it is not already in the table
    private int locate(int key) {
            int pos = hash(key);
             while (row[pos] != null && row[pos].key != key
                        pos = (pos + 1)  % rows.length;
    }          return pos;
```

linear probing

EXAMPLE IMPLEMENTATION WITHOUT GENERICS

```
// TODO 3: put the specified value in the table under the specified key
public void insert(int key, String value) {




}

// TODO 4: retrieve the value associated with the given key
public String lookup(int key) {



}
```

NEXT TIME!