CS 261-020
Data Structures

Lecture 13

Heapsort

Maps and Hash Tables

2/29/24, Thursday

Oregon State University

Odds and Ends

- Assignment 4 due Sunday midnight via TEACH
- Quiz 4 unlock after today's lecture, due Sunday midnight via Canvas

Removing from an array-based Heap

value

- Recall: in min heap, always remove the node with the lowest priority (i.e., root)
- Remove an element from the array representation of the heap follows this procedure:
  1. Remember the value of the first element in the array (to be returned later).
  2. Replace ~~the value of the~~ first element in the array with ~~the value of the~~ last element and remove the last element.
  3. If the array is not empty (i.e. it started with more than one element), compute the indices of the children of the replacement element (2 * i + 1 and 2 * i + 2).
     - If both of these elements fall beyond the bounds of the array, stop here. priority
  4. Compare the value of the replacement element with the minimum ~~value~~ of its two children (or possibly one child).

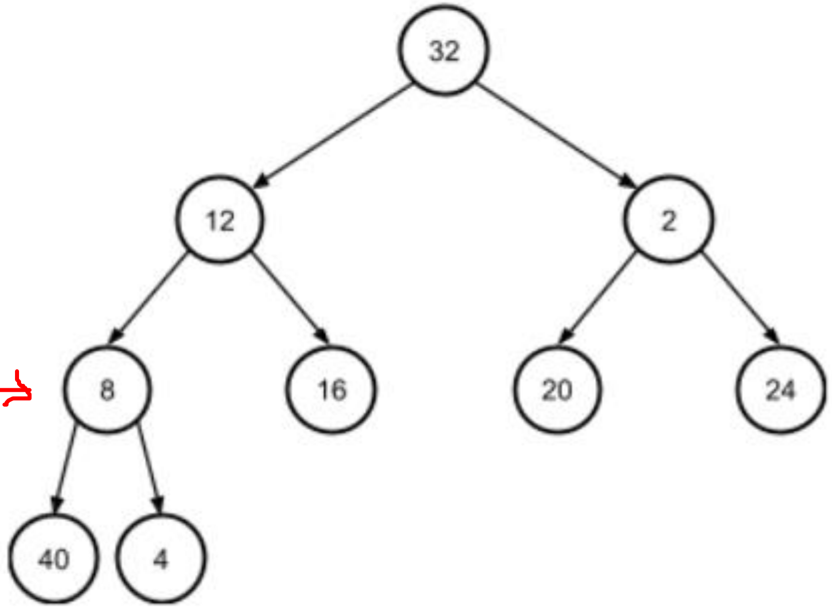
     priority priority
  5. If the replacement element's ~~value~~ is greater than its minimum child's ~~value~~, swap those two elements in the array and repeat from step 3

Min

Building a heap from an arbitrary array

- Percolate down the first non-leaf element, then the subtree rooted at that element's original position will be a proper heap
  - first non-leaf element (from the back of the array) is at n / 2 - 1

0	1	2	3	4	5	6	7	8
32	12	2	8	16	20	24	40	4



Building a heap from an arbitrary array

- Time Complexity:
  - perform n / 2 downward percolation operations.
  - Each of these operations is O(log n).
  - This means the total complexity is O(n log n).

- Space Complexity:
  - No additional space needed and no recursive calls: O(1)

Lecture Topics:

- Heapsort

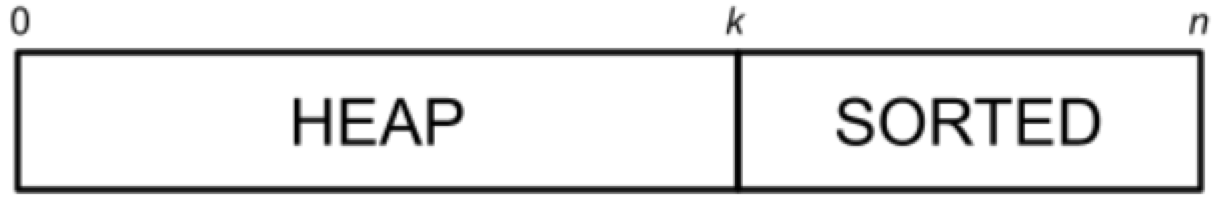
- Hash Tables
- Hash Functions
- Hash Collisions

Heap Sort

- Given the heap and its operations, we can implement an efficient (O(n log n)), in-place sorting algorithm called heapsort.
- First, build a heap out of the array
- Then, sort:
  - Keep a running counter k that is initialized to one less than the size of the array (i.e. the last element).
  - Swap the first element in the array (the min) with the last element (the kth element).
    - The array itself remains the same size, and we decrement k.
  - Percolate the replacement value down to its correct place in the array, stop at the kth element.
    - Thus, the heap is effectively shrinking by 1 at each iteration
- Repeat this procedure until k reaches the beginning of the array

Heap Sort

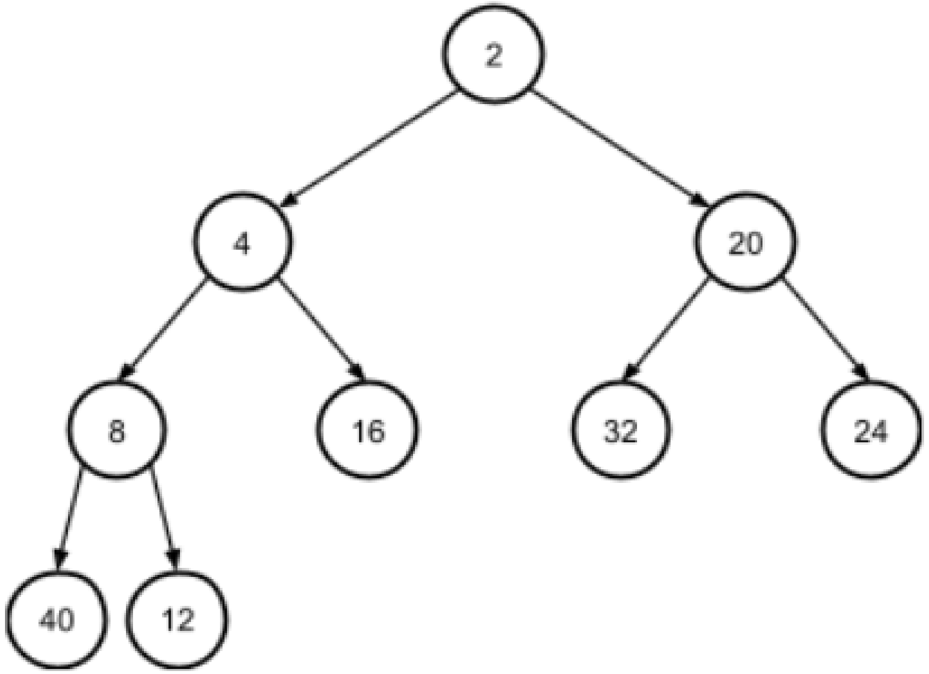
- As this sorting procedure runs, it maintains two properties:
  - The elements of the array beyond k are sorted, with the minimum element at the end of the array.
  - The array through element k always forms a heap, with the minimum remaining value at the beginning of the array



Heap Sort Example

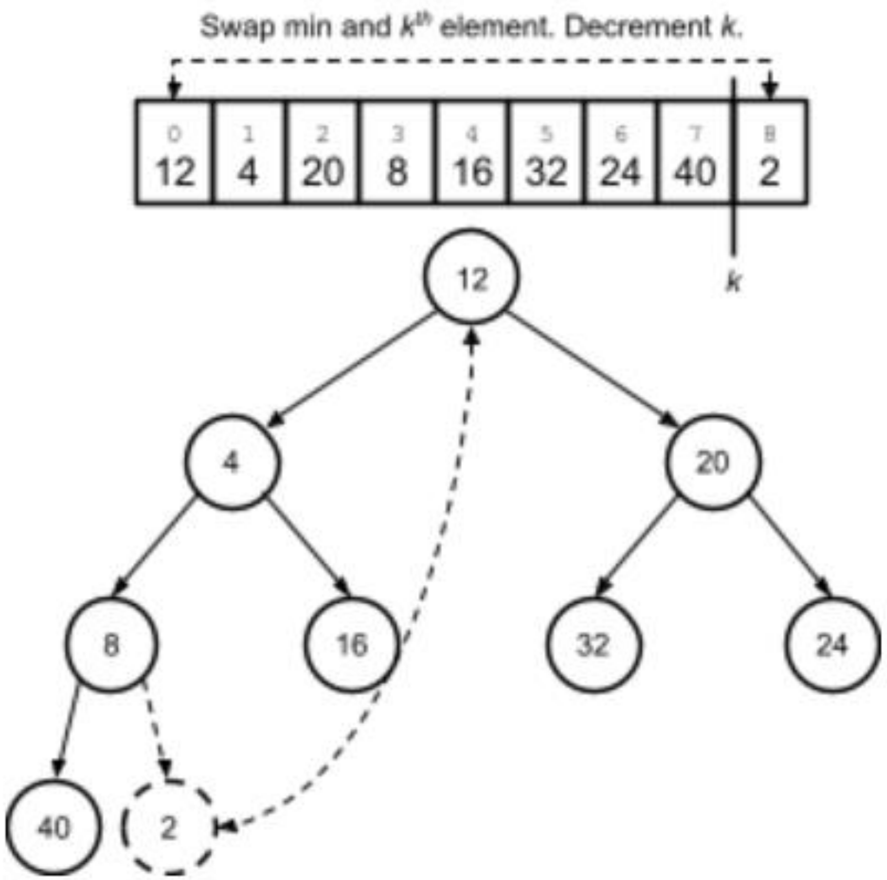
- Apply Heapsort to the following heap array (descending order):

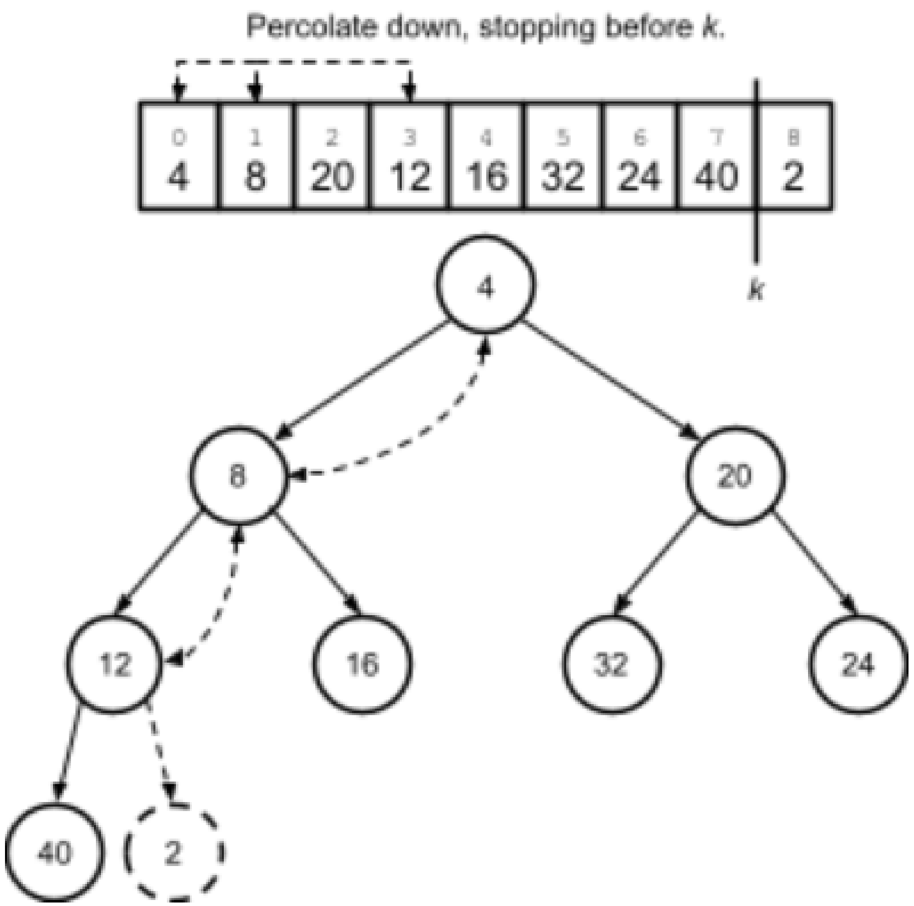
0	1	2	3	4	5	6	7	8
2	4	20	8	16	32	24	40	12



Heap Sort Example

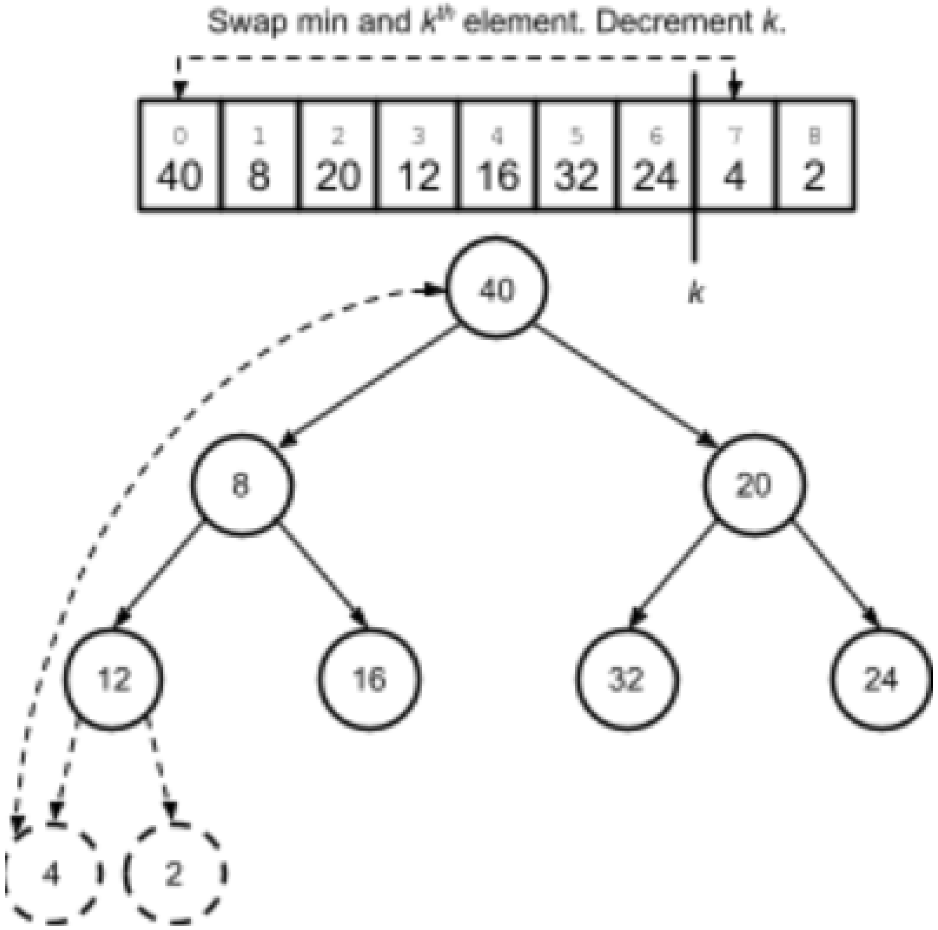
- Apply Heapsort to the following heap array (descending order):

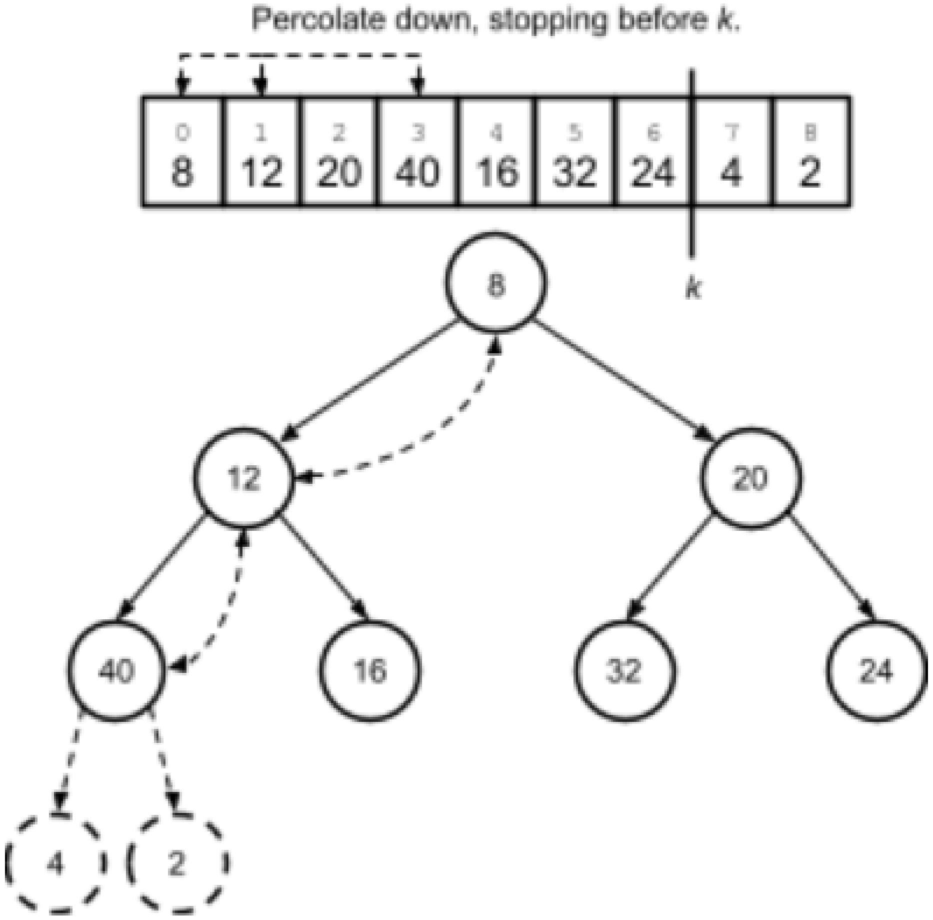




Heap Sort Example

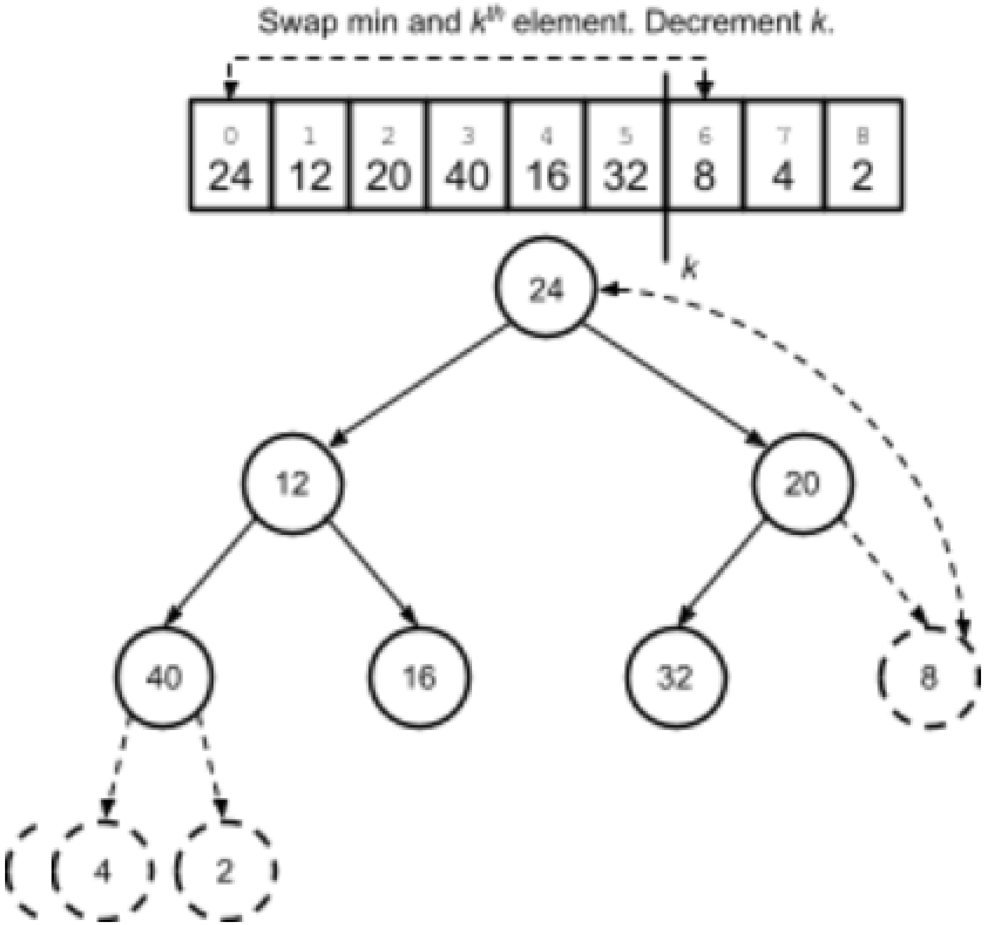
- Apply Heapsort to the following heap array (descending order):

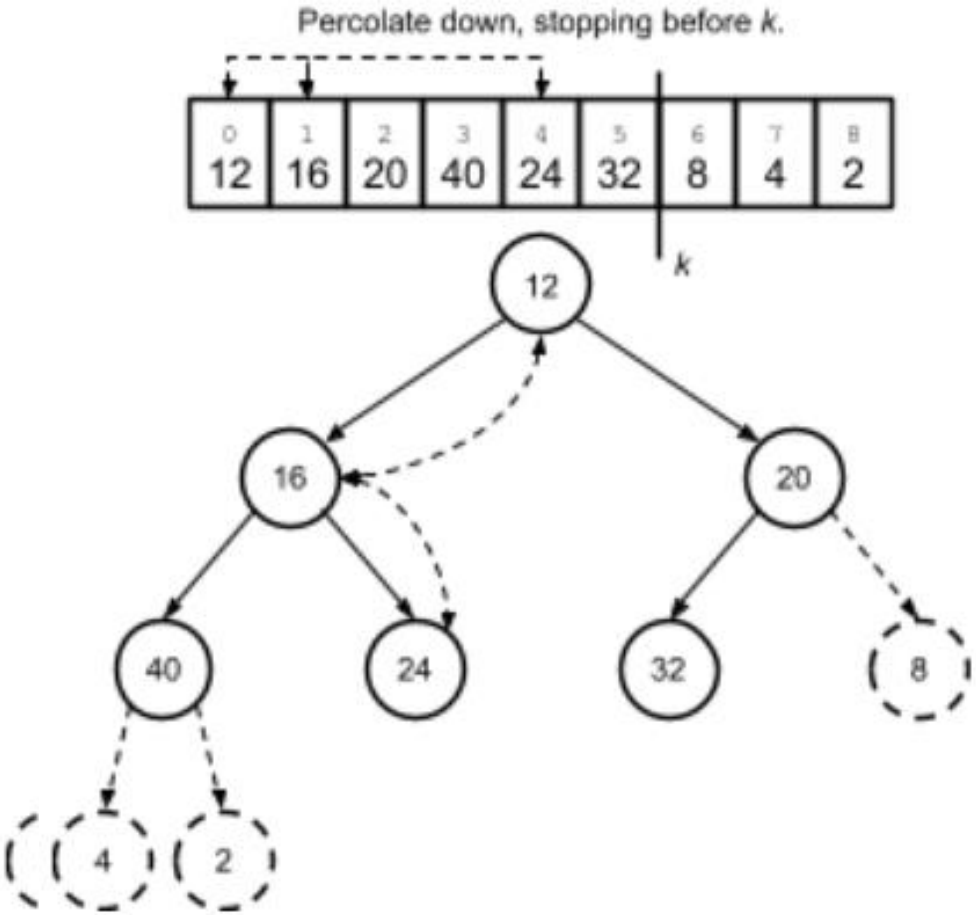




Heap Sort Example

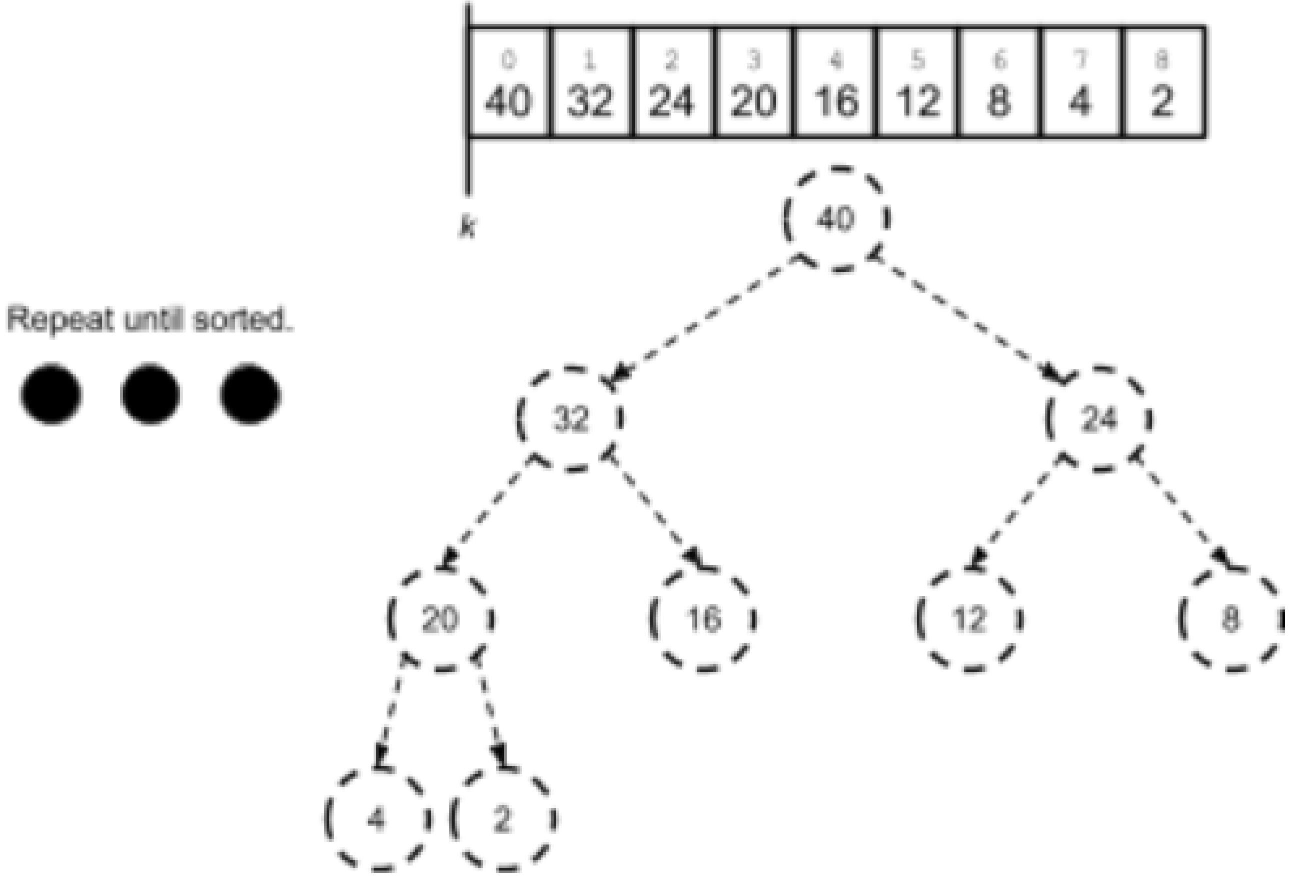
- Apply Heapsort to the following heap array (descending order):



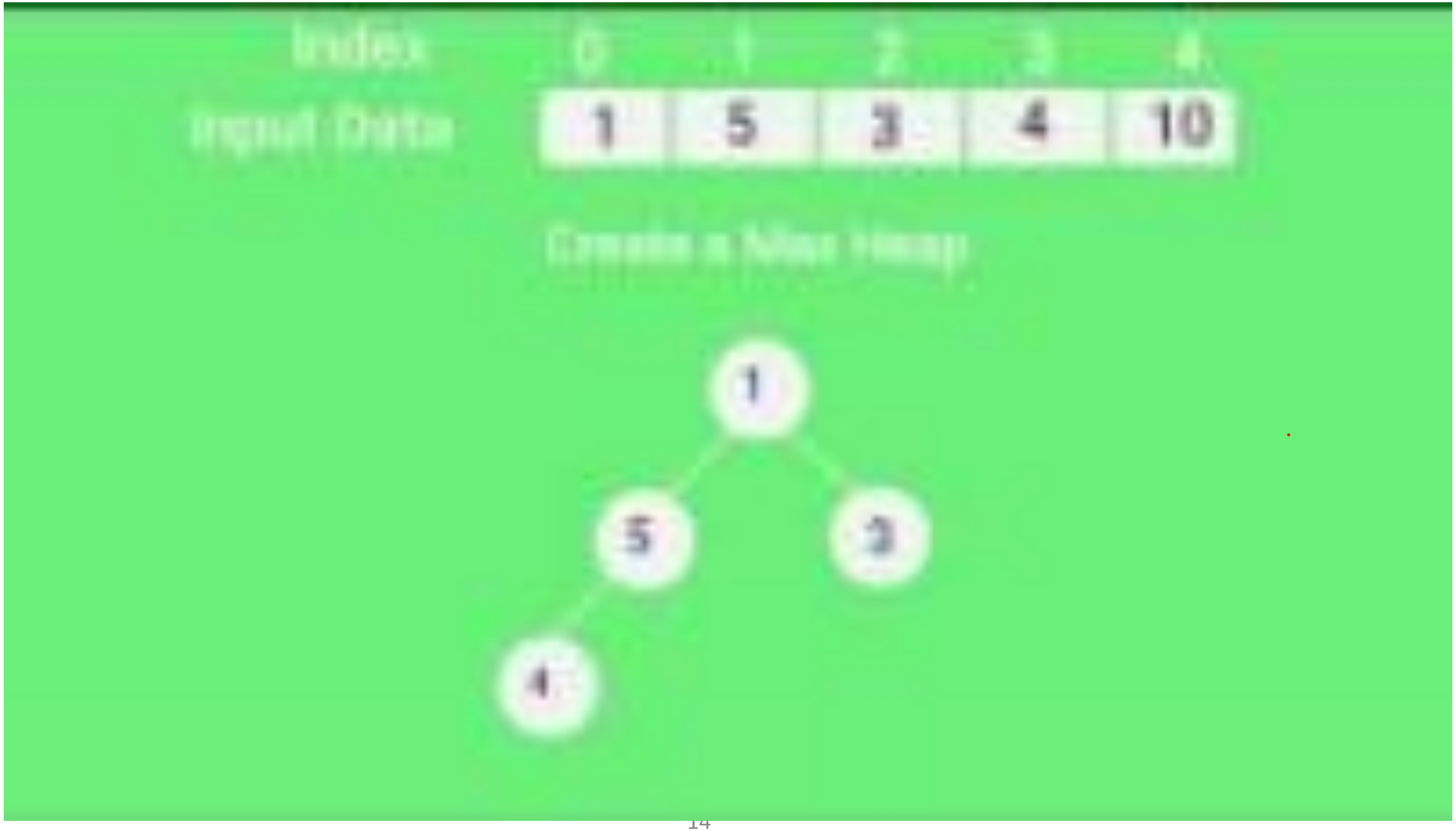


Heap Sort Example

- Apply Heapsort to the following heap array (descending order):



Heap Sort Example



Lecture Topics:

- Heapsort
- Hash Tables
- Hash Functions
- Hash Collisions

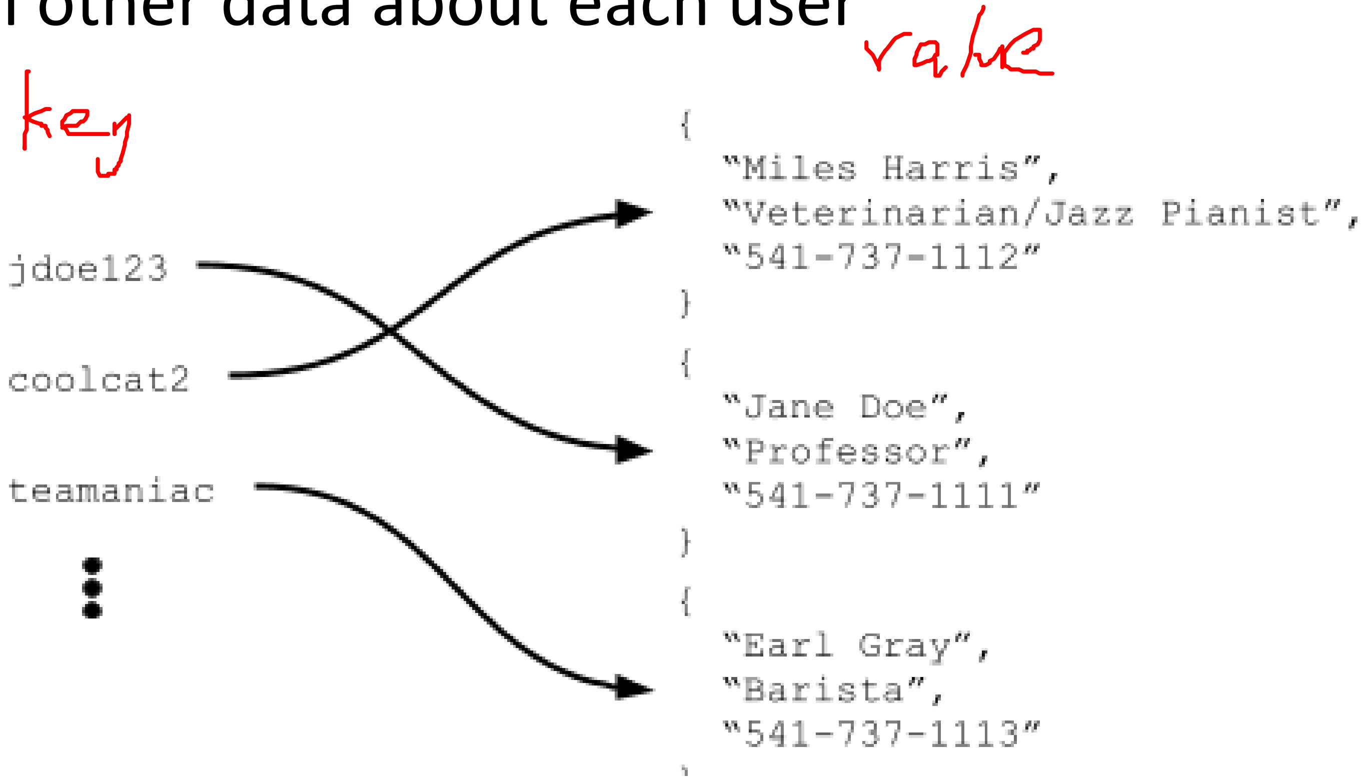
Maps

- Map data type: when insertion and lookup (even removal) are the only operations we need
  - A map is also known as a dictionary or an associative array

- With a map, each data element is actually composed of two parts:
  - The ***key***, which is the value by which we look items up.
  - The ***value***, which is any and all other data associated with the element

Maps

- For example, in a web app, the user data might be represented in a map,
  key = username/email
  value = all other data about each user



In-class activity:

- Given the following words (data pool), what data structure can we use to implement a map that has the best lookup functionality?
- `"yummy"`
  `"delicious"`
  `"incredible"`
  `"fantastic"`
  `"exquisite"`
  `"nonpareil"`

Map Example

Data structures that allow us to implement a map structure:

- Array, storing key/value structs. – binary search
  - This would give us O(n) insertions and lookups (or O(log n) lookups, if we ordered the array by key)

- AVL tree, also storing key/value structs.
  - This would give us O(log n) insertions and lookups

- Can we do it better?
- If we know the index, then insertions and lookups will be O(1)
  - How? By using a **hash table**

Consider this…

- Suppose we want to maintain a set of students (size n), where each student has a unique id from (0 to n-1) and a student name.
- Q: How can we use the student id to find a student in an array?
- Simple! Array of size n, student i will be at index i

- lookup: O(1), insert: O(1), remove: O(1), memory: O(n)

Consider this…

- Suppose we want to maintain a set of students (size n), where each student has a 9-digit id and a student name.
- Q: How can we use the student id to find a student in an array?

- Option 1: An array of size 999 999 999! Student id == index
  - Then, lookup: O(1), insert: O(1), remove: O(1)
  - Problem: memory usage! Lots of unused space

- Alternatively, we can use the **key to compute an index** into a moderate size array.
  - Want: lookup: O(1), insert: O(1), remove: O(1), memory: O(n)

Hash Tables

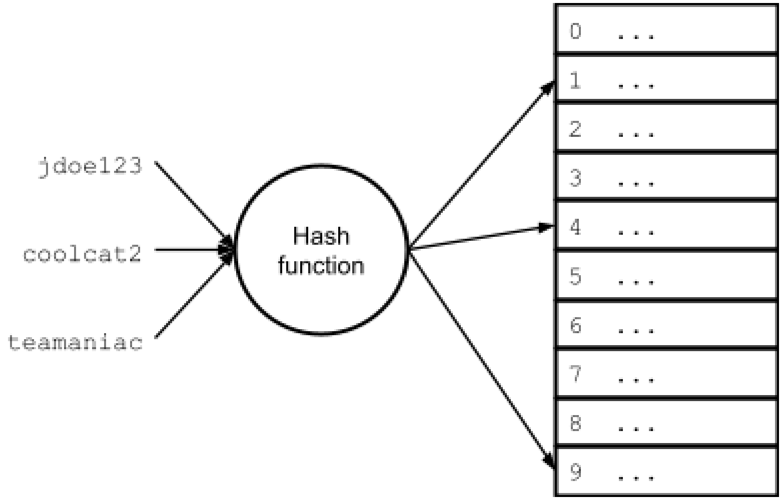
- A hash table is like an array, with a few important differences:
  - Elements can be indexed by values other than integers.
  - More than one element may share an index. (More later)

- The key to implementing a hash table is a hash function, which is a function that takes values key of some type (e.g. string, struct, double, etc.) and maps them to an integer index value

Key → Hash function → integer

Hash Tables

- We can then use this value both to **store** and **retrieve** data out of an actual array:



- Often the hash function computes an index in two steps:

```
hash = hash_function(key)
index = hash % array_size
```

Hash Table Example:

- Use the following hash function to store the words into a hash table.

```
int string_hash(char* str) {
      return (int)(str[0] - 'a') % 6;
}
```

array size

- "yummy"
  "delicious"
  "incredible"
  "fantastic"
  "exquisite"
  "nonpareil"

'y' – 'a' = 24 % 6 = 0
'd' – 'a' = 3 % 6 = 3
'i' – 'a' = 8 % 6 = 2
'f' – 'a' = 5 % 6 = 5
'e' – 'a' = 4 % 6 = 4
'n' – 'a' = 13 % 6 = 1

yummy	nonpareil	incredible	delicious	exquisite	fantastic

Hash Tables

- When choosing or designing a hash function, there are a few properties that are desirable:
  - **Determinism** – a given input should always map to the same hash value.
  - **Uniformity** – the inputs should be mapped as evenly as possible over the output range.
    - A non-uniform function can result in many collisions, where multiple elements are hashed to the same array index. (More later).
  - **Speed** – the function should have low computational burden

Hash Tables

- For example, if we were hashing strings, a simple hash function might sum the ASCII values of the characters, e.g.:

```
"eat" ⇨ 'e' + 'a' + 't' = 101 + 97 + 116 = 314
```

  - An operation like this is known as a folding operation.

- Problems

```
"eat" ⇨ 'e' + 'a' + 't' = 101 + 97 + 116 = 314
"ate" ⇨ 'a' + 't' + 'e' = 97 + 116 + 101 = 314
"tea" ⇨ 't' + 'e' + 'a' = 116 + 101 + 97 = 314
```

Hash Tables

- To fix this, use a shifting operation, which modifies the individual components of a folding operation based on their position.
- E.g. multiply by 2^0, 2^1, 2^2, 2^3, …

```
"eat" ⇨ 'e' + 'a' + 't’ =
1* 101 + 2* 97 + 4* 116 = 759

"ate" ⇨ 'a' + 't' + 'e’ =
1* 97 + 2* 116 + 4* 101 = 733

"tea" ⇨ 't' + 'e' + 'a’ =
1* 116 + 2* 101 + 3* 97 = 609
```

Hash Tables

- An example of a well-known and widely-used hash function, the DJB hash function (for strings):

```
unsigned long hash(unsigned char *str) {
  unsigned long hash = 5381;
  int c;
  while (c = *str++) {
    hash = ((hash << 5) + hash) + c;  // hash * 33 + c
  }
  return hash;
}
```

- This function is simple and fast (though could be faster, e.g. by processing multiple bytes at a time).
- It produces a good distribution

Perfect and Minimally Perfect Hash Functions

- Collision: some keys map to the same index:
  - x != y, but hash(x) == hash(y)

- A perfect hash function is one that results in no collisions.

- A minimally perfect hash function is one that results in no collisions for a table size that exactly equals the number of elements.

n

Perfect and Minimally Perfect Hash Functions

- For example, consider this collection of strings:

```
"yummy"
"delicious"
"incredible"
"fantastic"
"exquisite"
"nonpareil"
```

- The following function is minimally perfect

```
int string_hash(char* str) {
      return (int)(str[0] - 'a') % 6;
}
```

- Specifically, we have all the values 0 through 5 covered

```
string_hash("yummy") → 0          // 'y' - 'a' = 24
string_hash("delicious") → 3      // 'd' - 'a' = 3
string_hash("incredible") → 2     // 'i' - 'a' = 8
string_hash("fantastic") → 5      // 'f' - 'a' = 5
string_hash("exquisite") → 4      // 'e' - 'a' = 4
string_hash("nonpareil") → 1      // 'n' - 'a' = 13
```

Perfect and Minimally Perfect Hash Functions

- In practice, we don’t usually have such a nicely arranged situation, so it’s rare that our hash function will be minimally perfect.
  - For example, even with perfectly uniform random distribution of elements and a hash table with a capacity of 1 million elements, there is a 95% probability of a collision with only 2450 elements

- This means that, most likely, we’ll need to be able to deal with collisions

Collision Example:

- Hash function to store the words into a hash table.
- ```
  int string_hash(char* str) {
      return (int)(str[0] - 'a') % 6;
  }
  ```
- "yummy"
  "delicious"
  "incredible"
  "fantastic"
  "exquisite"
  “date"

date

yummy	nonpareil	incredible	delicious		fantastic
  ```

Collision Resolution

Two mechanisms for resolving hash collisions

- Chaining
- Open addressing

1. Collision Resolution with Chaining

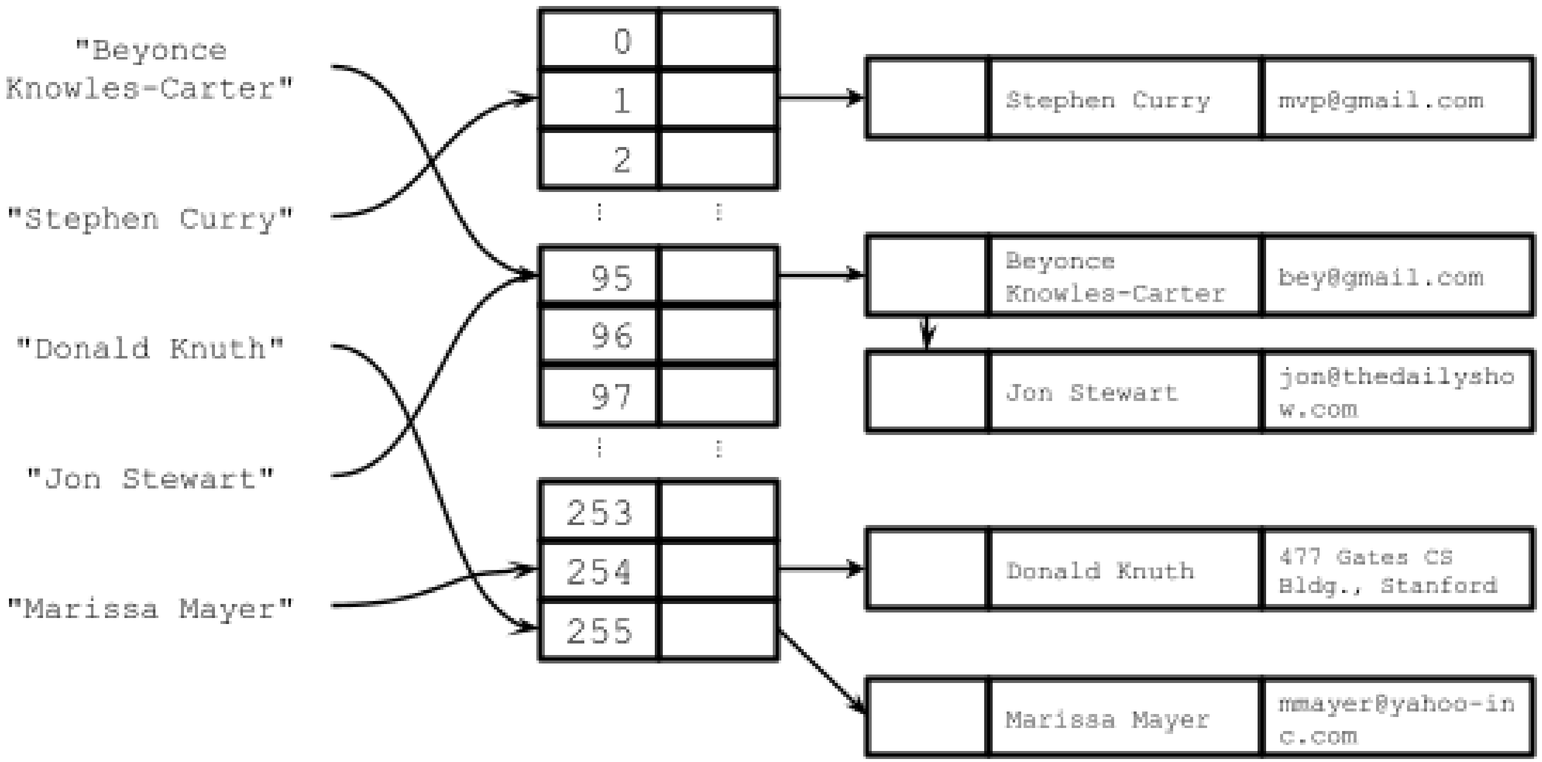
- The chaining method involves storing a collection of elements at each index in the hash table array.
  - Each collection is called a bucket or a chain.

- When a collision occurs, the new element is added to the collection at its corresponding hash index.

- Linked lists are a popular choice for maintaining the buckets themselves.
  - Other data structures could be used, e.g. a dynamic array or a balanced binary tree

1. Collision Resolution with Chaining

- Here's what a hash table with linked list-based chains might look like:



1. Collision Resolution with Chaining

- In a chained hash table,
- To loopup the value for a particular key:
  - Compute the element's bucket using the hash function
  - Search the data structure at that bucket for the element (using the key)
    - E.g. iterate through the items in the linked list.

- To add/remove an element:
  - Compute the element's bucket using the hash function
  - add or remove the element to/from the appropriate bucket's data structure
    - E.g. iterate through the items in the linked list.

2. Collision Resolution with Open Addressing

- The open addressing method: involves probing for an empty spot
- When using open addressing, all hashed elements are stored directly in the hash table array
- To insert an element:
  - Use the hash function to compute an initial index i for the element.
  - If the hash table array at index i is empty, insert the element there and stop.
  - Otherwise, increment i to the next index in the probing sequence (e.g. i + 1) and repeat

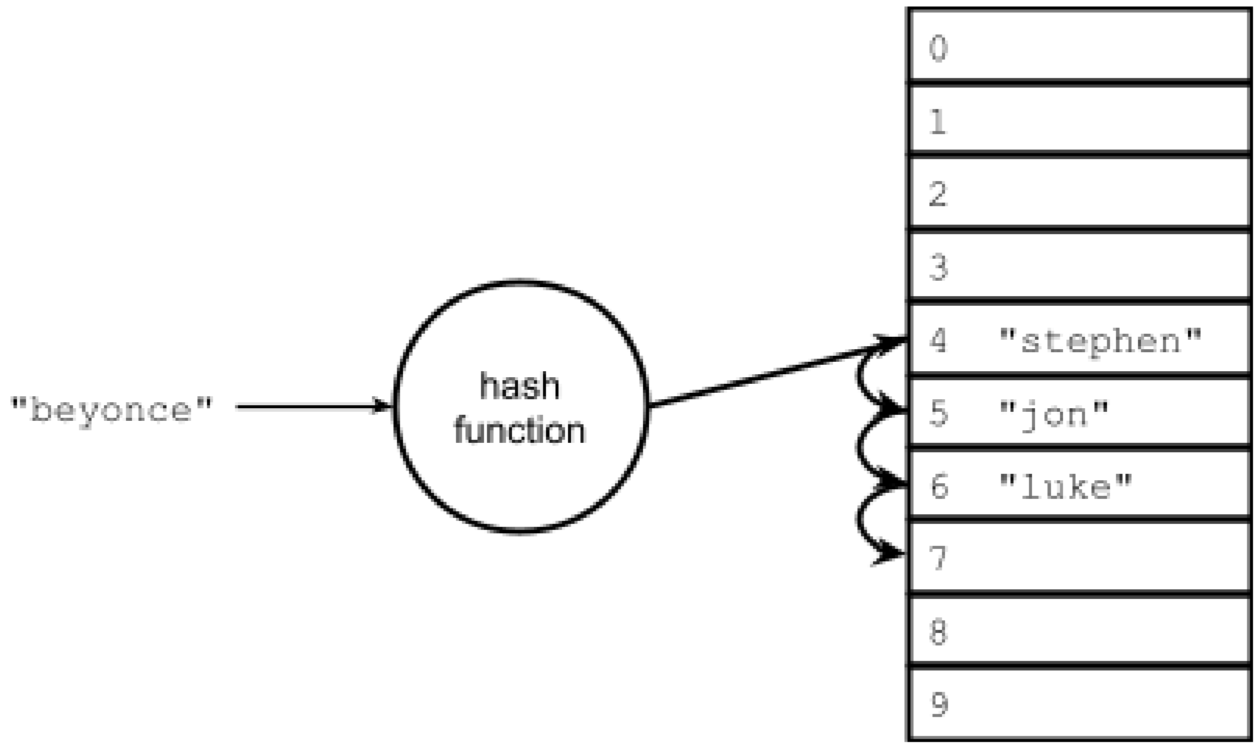
2. Collision Resolution with Open Addressing

- Probing: the process of searching for an empty position.

- There are many different probing schemes:
  - Linear probing: $i = i + 1$
  - Quadratic probing: $i = i + j^2$ $(j = 1, 2, 3, \ldots)$
  - Double hashing: $i = i + j * h_2(key)$ $(j = 1, 2, 3, \ldots)$
    - Here, h_2 is a second, independent hash function.

2. Collision Resolution with Open Addressing

- For example, using linear probing, the key "beyonce" would be inserted at index 7, even though the hash function evaluates to 4 for that key:



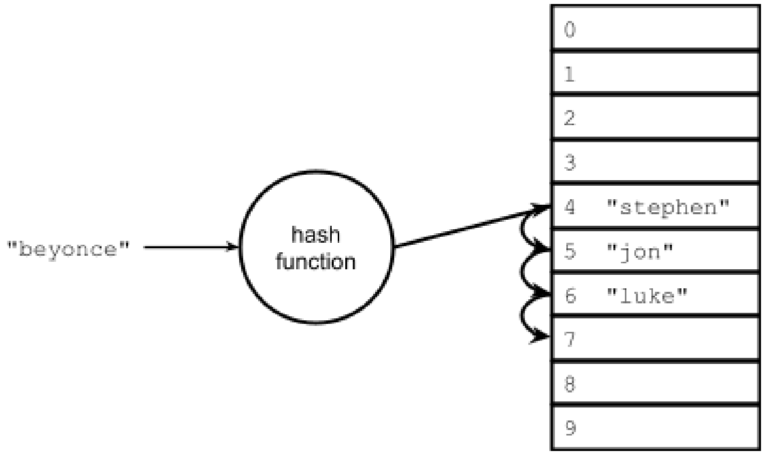
2. Collision Resolution with Open Addressing

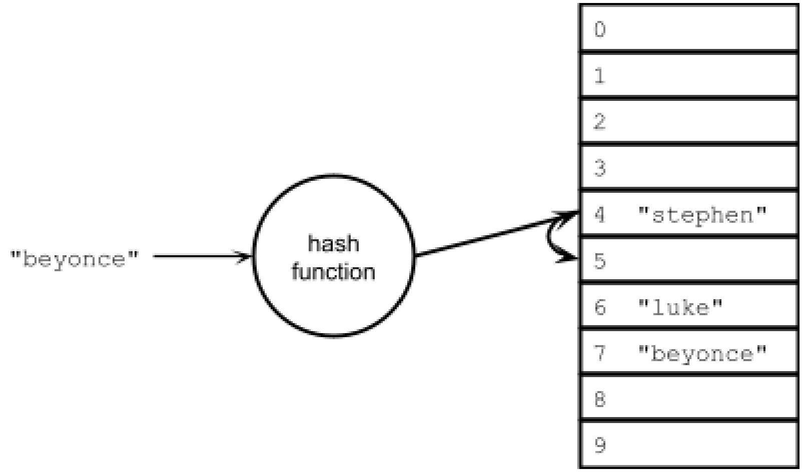
- To search for an element:
  - Use the hash function to compute an initial index i for the element
  - probe until we find either the element or an empty spot
    - If found an empty spot, then the element doesn't exist

- What happens if we reach the end of the array while probing?
  - Simply wrap around to the beginning.

2. Collision Resolution with Open Addressing

- What happens when we remove an element?
  - Search for the element, then remove it
- What about searching after removing?
  - This could disrupt probing for elements after it.
- For example, what if we removed "jon" and then searched for "beyonce"?





2. Collision Resolution with Open Addressing

- To get around this problem, we use a special value known as the tombstone
- Now, when an element is removed, we insert the tombstone value.
  - This value can be replaced when adding a new entry, but it doesn't halt search for an existing element.

- With a tombstone value __TS__ inserted for the removed "jon", the search above for "beyonce" could proceed as normal: