Data Structures and Algorithms

UBCO Master of Data Science – DATA 532

Recap...

- Looked at variety of ADT data structures
  - Stacks
  - Queues
  - Linked lists

What should you be able to do after today's lecture

- Understand sorting algorithm
- Understand the complexity of different sorting algorithms
- Understand the notion of hashing
- Understand the idea of collision in hashing and how to avoid it

The Sorting problem…

The Sorting Problem

Input:

- A sequence of n numbers $a_1, a_2, \ldots, a_n$

Output:

- A permutation (reordering) $a_1', a_2', \ldots, a_n'$ of the input sequence such that $a_1' \leq a_2' \leq \cdots \leq a_n'$

Why Study Sorting Algorithms?

Various algorithms are better suited to some of these situations

- Binary Search
- A program that renders graphical objects which are layered on top of each other need to sort the objects according to an "above" relation so that it can draw these objects from bottom to top

There are a variety of situations that we can encounter

- Do we have randomly ordered keys?

Stability

A STABLE sort preserves relative order of records with equal keys

Sorted on first key:

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Sort file on second key:

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker
Aaron	4	A	664-480-0023	097 Little

Records with key value 3 are not in order on first key!!

10

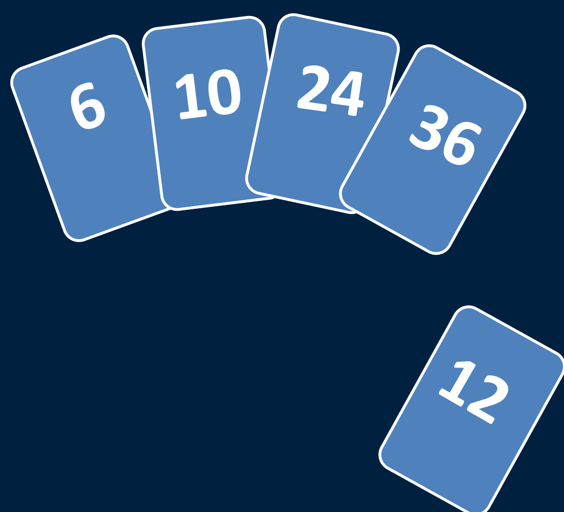
Insertion Sort

Idea: like sorting a hand of playing cards

- Start with an empty left hand and the cards facing down on the table.
- Remove one card at a time from the table, and insert it into the correct position in the left hand
  - compare it with each of the cards already in the hand, from right to left
- The cards held in the left hand are sorted
  - these cards were originally the top cards of the pile on the table

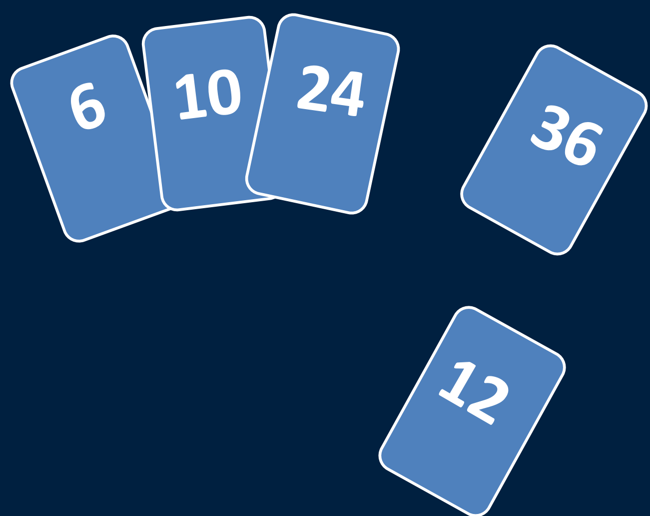
Insertion Sort

Idea: like sorting a hand of playing cards

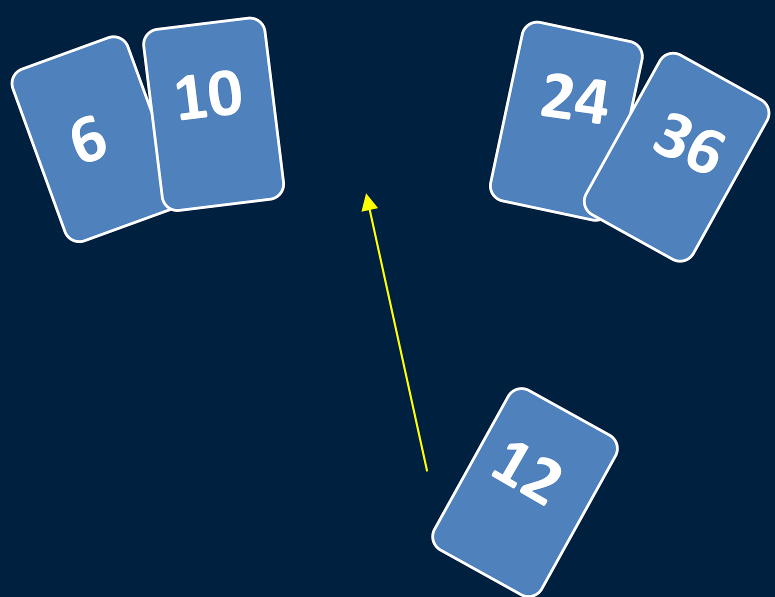


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



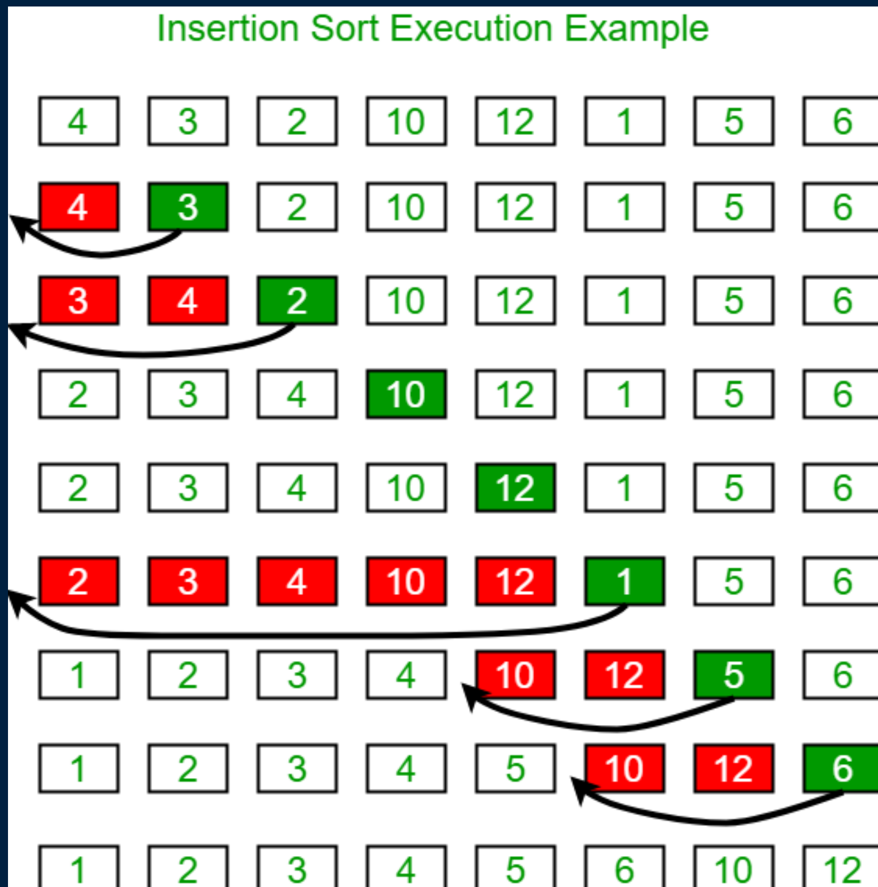
Insertion Sort

input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

Insertion Sort



INSERTION-SORT

Alg.: INSERTION-SORT*(A)*

0	1	2	3	4	5	6	7
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8

for j ← 1 **to** n

  do key ← *A*[j]

   Insert *A*[j] into the sorted sequence *A*[1 . . j -1]

   i ← j - 1

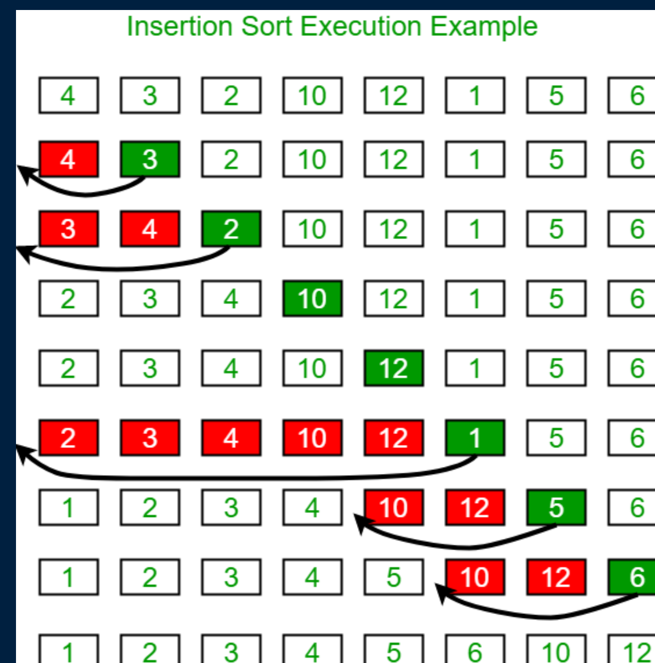
   while i >= 0 and *A*[i] > key

    do *A*[i + 1] ← *A*[i]

     i ← i - 1

   A[i + 1] ← key

Insertion sort – sorts the elements in place



Insertion Sort: Time Complexity

Best case O(n)

Worst case O(n^2)

Average case O(n^2)

*This makes insertion sort impractical for sorting large arrays.

Insertion Sort - Summary

Advantages

- Good running time for “almost sorted” arrays $\Theta(n)$
- One of the fastest algorithms for sorting very small arrays (around ten), even faster than quicksort (recursive).

Since sorting can often reduce the complexity of a problem,

Disadvantages

- $\Theta(n^2)$ running time in worst and average case

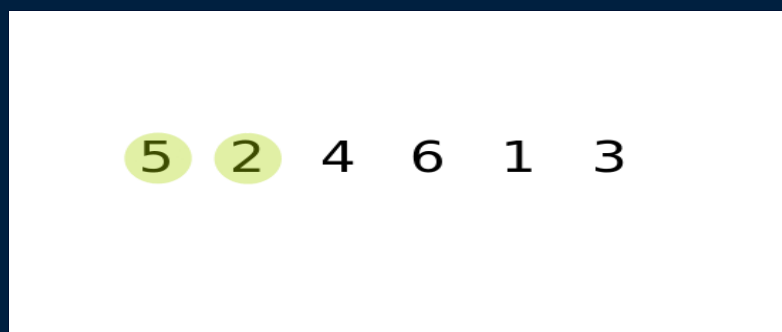
Bubble Sort

Idea:

- Repeatedly pass through the array
- Swaps adjacent elements that are out of order

input array

5 2 4 6 1 3



Easier to implement, but slower than Insertion sort

Example

8	4	6	9	2	3	1
i = 1						j

8	4	6	9	2	1	3
i = 1					j	

8	4	6	9	1	2	3
i = 1				j		

8	4	6	1	9	2	3
i = 1			j			

8	4	1	6	9	2	3
i = 1		j				

8	1	4	6	9	2	3
i = 1	j					

1	8	4	6	9	2	3
i = 1	j					

1	8	4	6	9	2	3
	i = 2					j

1	2	8	4	6	9	3
		i = 3				j

1	2	3	8	4	6	9
			i = 4			j

1	2	3	4	8	6	9
				i = 5		j

1	2	3	4	6	8	9
					i = 6	j

1	2	3	4	6	8	9
						i = 7
						j

Bubble Sort

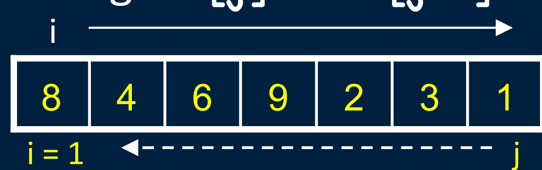
Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A]

  do for j ← length[A] **downto** i + 1

    do if A[j] < A[j -1]

      then exchange A[j] ↔ A[j-1]



Improved Bubble Sort

Best case is O(n) based on the improved implementation below

Average and worst case are still O(n^2)

An Improved Alternative Implementation

```
procedure bubbleSort( A : list of sortable items ) defined as:
  n := length( A )
  do
    swapped := false
    for each i in 0 to n - 1 inclusive do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
        swapped := true
      end if
    end for
    n := n - 1
  while swapped
end procedure
```

Bubble Sort: Time Complexity

Best case O(n)

Worst case O(n^2)

Average case O(n^2)

*This makes insertion sort impractical for sorting large arrays.

Selection Sort

Idea:

- Find the smallest element in the array
- Exchange it with the element in the first position
- Find the second smallest element and exchange it with the element in the second position
- Continue until the array is sorted

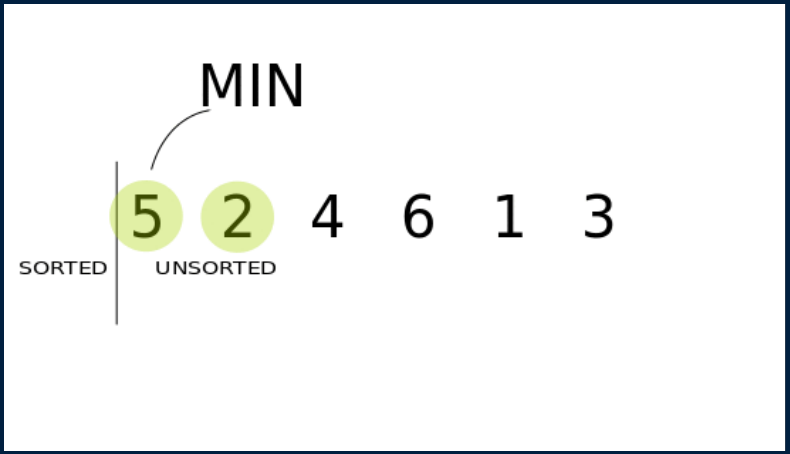
Disadvantage:

- Running time depends only slightly on the amount of order in the file

Selection Sort

input array

5 2 4 6 1 3



Selection Sort

Alg.: SELECTION-SORT(*A*)

1	2	3	4	....		n
8	4	6	9	2	3	1

```
n ← length[A]
for j ← 1 to n - 1
    smallest ← j
    for i ← j + 1 to n
        if A[i] < A[smallest]
            then smallest ← i
    exchange A[j] ↔ A[smallest]
```

Selection Sort: Time Complexity

Best case O(n^2)

Worst case O(n^2)

Average case O(n^2)

Comparison of different sorting algorithms

Comparison sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends	Partitioning	Quicksort can be done in place with O(log(n)) stack space, but the sort is unstable. Naïve variants use an O(n) space array to store the partition. An O(n) space implementation can be stable.
Selection sort	n^2	n^2	n^2	1	No	Selection	Its stability depends on the implementation. Used to sort this table in Safari or other Webkit web browser [3].
Insertion sort	n	n^2	n^2	1	Yes	Insertion	Average case is also $\mathcal{O}(n+d)$, where d is the number of inversions
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends	Yes	Merging	Used to sort this table in Firefox [2].
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size

Hashing

Can we do it?

Consider the problem of searching an array for a given value

- If the array is not sorted, the search requires O(n) time
  - If the value isn't there, we need to search all n elements
  - If the value is there, we search n/2 elements on average
- If the array is sorted, we can do a binary search
  - A binary search requires O(log n) time
  - About equally fast whether the element is found or not
- It doesn't seem like we could do much better
  - How about an O(1), that is, constant time search?
  - We can do it *if* the array is organized in a particular way

Hashing

Suppose we were to come up with a “magic function” that, given a value to search for, would tell us exactly where in the array to look

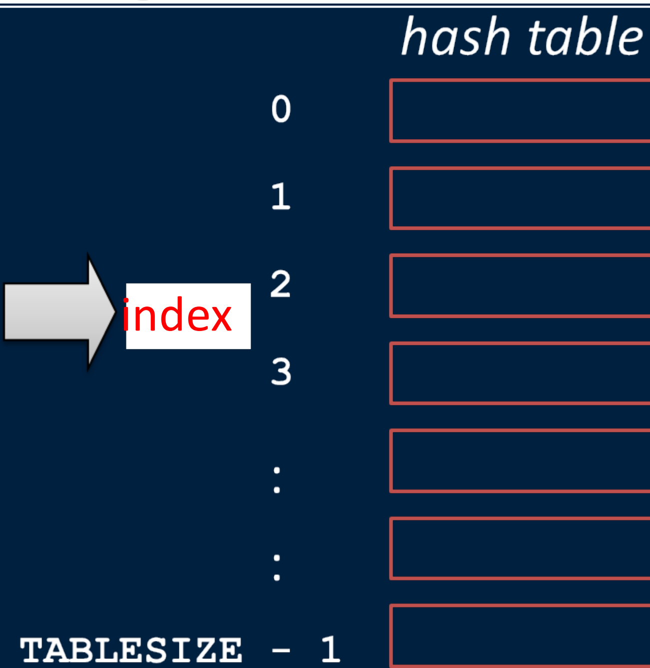
- If it’s in that location, it’s in the array
- If it’s not in that location, it’s not in the array

This function would have no other purpose

If we look at the function’s inputs and outputs, they probably won’t “make sense”

This function is called a hash function because it “makes hash” of its inputs

- A hash function converts an arbitrary key (string or integer) into an integer within a particular range.



Example:
A 9-digit student number can be converted to a 5-digit index by selecting just the final five digits. For example:

009516733 -> 16733
009128356 -> 28356

Alternatively, we could select the third, fifth, sixth, eighth, and ninth digits:

00**9**5**16**7**33** -> 91633
00**9**1**28**3**56** -> 92856

Example (ideal) hash function

Suppose our hash function gave us the following values:

```
hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("cantaloupe") = 7
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2
```

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

Finding the hash function

How can we come up with this magic function?

In general, we cannot--there is no such magic function ☹

- In a few specific cases, where all the possible values are known in advance, it has been possible to compute a perfect hash function

What is the next best thing? (we don't know all the data in advance)

- A perfect hash function would tell us exactly where to look
- In general, the best we can do is a function that tells us where to *start* looking!

Example: Modulo Arithmetic

- Use the modulos operator % to compute the remainder between the key and the size of the array.
- Say we had a table with room for only 10 students. If the table size is B, then (key % B) maps the key into an integer in the range [0, B-1].
- Modulo arithmetic is the basis for most hash functions.
- Still---this strategy cannot prevent collisions. We must deal with them...

008987230 % 10	0
200113231 % 10	1
200323622 % 10	2
…	
200712435 % 10	5
…	
200334439 % 10	9

Example imperfect hash function

Suppose our hash function gave us the following values:

- hash("apple") = 5
  hash("watermelon") = 3
  hash("grapes") = 8
  hash("cantaloupe") = 7
  hash("kiwi") = 0
  hash("strawberry") = 9
  hash("mango") = 6
  hash("banana") = 2
  hash("honeydew") = 6

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

- Now what?

Collisions

When two values hash to the same array location, this is called a collision

Collisions are normally treated as “first come, first served”—the first value that hashes to the location gets it

We have to find something to do with the second and subsequent values that hash to this same location

Handling collisions

What can we do when two different values attempt to occupy the same place in an array?

- **Solution #1:** Search from there for an empty location
  - Can stop searching when we find the value *or* an empty location
  - Search must be end-around
- **Solution #2:** Use a second hash function
  - ...and a third, and a fourth, and a fifth, ...
- **Solution #3:** Use the array location as the header of a linked list of values that hash to this location

All these solutions work, provided:

- We use the same technique to *add* things to the array as we use to *search* for things in the array

Solution #1: Linear Probing

Linear probing

- Simplest way to resolve a collision
- Search array, starting from collision spot, for the first available position

At the start of an insertion, the hashing function is run to compute the **home index** of the item

- If cell at home index is not available, move index to the next to probe for an available cell
- When search reaches last position of array, probing wraps to the beginning to continue from the first position

For retrievals, stop probing process when current array cell is empty or it contains the target item

Insertion, I

Suppose you want to add seagull to this hash table

Also suppose:

- hashCode(seagull) = 143
- table[143] is not empty
- table[143] != seagull
- table[144] is not empty
- table[144] != seagull
- table[145] is empty

Therefore, put seagull at location 145

Deletion

However, we have to be careful about deletion. Assume we wish to delete 5604 from table (a)

If we simply set that cell again to empty we have table (b)

If we now do a search for 3305, we begin the search at index 5. Immediately we encounter an empty cell which tells us that 3305 is not present... So we prematurely ended the search

0	9909
1	
2	
3	
4	2204
5	5604
6	3305
7	
8	
9	0609

(a)

0	9909
1	
2	
3	
4	2204
5	
6	3305
7	
8	
9	0609

(b)

We can resolve this problem by marking deleted cells as <deleted> so that our search algorithm knows to bypass them.

In fact, it is necessary to record whether a cell is in one of three states: empty, occupied, or deleted.

Occupied cells store both the key and the associated data (e.g. the student's record).

An issue with this particular collision resolution scheme (linear probing) is that it often leads to **clustering.** Table items tend to cluster together in groups. If you perform an operation (insertion, removal, or search) that hits such a cluster then the cost of that operation may become significant.

0	9909
1	
2	
3	
4	2204
5	<deleted>
6	3305
7	
8	
9	0609

Searching, I

Suppose you want to look up seagull in this hash table

Also suppose:

- hashCode(seagull) = 143
- table[143] is not empty
- table[143] != seagull
- table[144] is not empty
- table[144] != seagull
- table[145] is not empty
- table[145] == seagull !

We found seagull at location 145

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Searching, II

Suppose you want to look up COW in this hash table

Also suppose:

- hashCode(cow) = 144
- table[144] is not empty
- table[144] != cow
- table[145] is not empty
- table[145] != cow
- table[146] is empty

If COW were in the table, we should have found it by now

Therefore, it isn't here

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Insertion, II

Suppose you want to add hawk to this hash table

Also suppose

- hashCode(hawk) = 143
- table[143] is not empty
- table[143] != hawk
- table[144] is not empty
- table[144] == hawk

hawk is already in the table, so do nothing

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Insertion, III

Suppose:

- You want to add cardinal to this hash table
- hashCode(cardinal) = 147
- The last location is 148
- 147 and 148 are occupied

Solution:

- Treat the table as circular; after 148 comes 0
- Hence, cardinal goes in location 0 (or 1, or 2, or ...)

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl

Solution #2: Rehashing

In the event of a collision, another approach is to rehash: compute another hash function

- Since we may need to rehash many times, we need an easily computable sequence of functions

Simple example: in the case of hashing Strings, we might take the previous hash code and add the length of the String to it

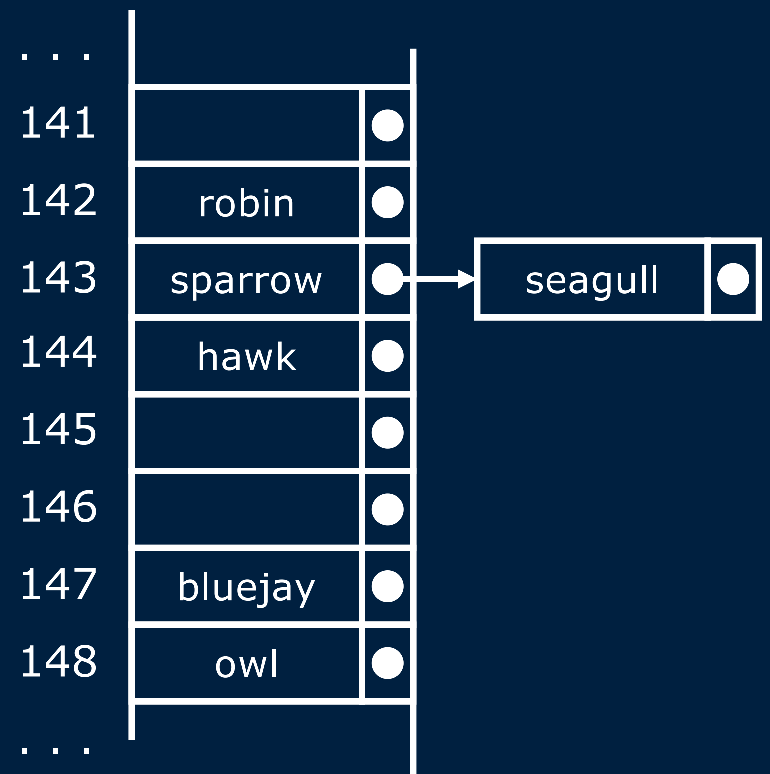
- Probably better if the length of the string was not a component in computing the original hash function

Rehashing is a fairly uncommon approach, and we won't pursue it any further here

Solution #3: Bucket hashing/Chaining

The previous solutions used open hashing: all entries went into a “flat” (unstructured) array

Another solution is to make each array location the header of a linked list of values that hash to that location



Chaining

Items are stored in an array of lists (**chains**)

- Each item's key locates the **bucket** (index) of the chain in which the item resides or is to be inserted

Retrieval and removal each perform these steps:

- Compute the item's home index in the array
- Search the list at that index for the item

To insert an item:

- Compute the item's home index in the array
- If cell is empty, create a node with item and assign the node to cell; else (collision), insert item in chain

64

Hashing with Non-Numeric Keys

Try returning the sum of the ASCII values in the string

This method has effect of producing same keys for **anagrams**

- Strings that contain same characters, but in different order

First letters of many words in English are unevenly distributed

- This might have the effect of weighting or biasing the sums generated

Hashing with Non-Numeric Keys (continued)

One solution:

- If length of string is greater than a certain threshold
  - Drop first character from string before computing sum
  - Can also subtract the ASCII value of the last character

Python also includes a standard **`hash`** function for use in hashing applications

- Function can receive any Python object as an argument and returns a unique integer

Consider searching a hash table with separate chaining:
Unsuccessful search:

- **Best Case:** The hash function gives us the array index. However, when we look at that index in the array, the linked list it contains is empty! Therefore we know that the item we seek is not to be found. The cost: ***O(1)***.
- **Average Case:** On average, each linked list is of length ***L***. Assuming ***L > 0***, at the hashed index value there is a nonempty list. We must (fruitlessly) search this entire list for the desired item. This requires examining about ***L*** nodes.
- **Worst Case:** All ***n*** items are in one linked list, and this is the list that you have to search! The cost: ***O(n)***.

Successful search:

- **Best Case:** Item is first in the linked list. The cost: ***O(1)***.
- **Average Case:** Item is half-way through the linked list. We must examine about ***L/2*** nodes.
- **Worst Case:** As for unsuccessful search, except the item sought now exists and is the last one in this big linked list. The cost: ***O(n)***.

If the load factor, ***L***, is kept close to one then the average cases above are both ***O(1)***!

So why do we bother with any other data structure if hash tables give us ***O(1)*** performance?

- To achieve a low load factor we need a large array ***B > n***. So hashtables can be wasteful of space (i.e. memory).
- The worst case for hashtables is much worse than the worst case for balanced binary search trees (***O(log n)***).
- The above average case analysis assumed uniform distribution of keys---which may be an unrealistic assumption.
- Hashtables are unsorted by nature. To access the data in a sorted manner all keys would have to be first extracted, put into an array (or tree), and then sorted

Hashing in Python

Python also includes a standard `hash` function for use in hashing applications

- Function can receive any Python object as an argument and returns a unique integer

Take home messages…

- We looked at hashing
- We learnt the idea of sorting algorithms
  - Insertion Sort
  - Bubble Sort
  - Selection Sort

Next Class: Recursion


UBC
THE UNIVERSITY OF BRITISH COLUMBIA