

1. Now that we have seen a number of ADTs and data structures in class, we finally have enough to start to see the correspondence between real-world problems and these structures. To review, here are our ADTs and structures so far:

<https://rhodes.app.box.com/file/749539374218?s=bsroxqbo9m2hvr50f4nvnknlhb73w7dx> (we have covered all of these except priority queues and graphs).

For this problem, you will analyze a real-world situation and pick a corresponding ADT and data structure to apply to it. Here's an example problem to get you warmed up:

Suppose you work for a railroad company and they want you to write programs to manage their trains. If you wanted to write a program to manage the order of the cars in a train, you might choose a **List** ADT, implemented with a **linked list** of strings, where each item in the list was a car. In this way, you could add cars to the front or back of a train (and with a little more effort, in the middle). Using an **arraylist** implementation would also be appropriate.

Suppose now each train car has a unique integer identification number, and you need to store the contents of each car in a program. You might use a **Map ADT (either BST or hash table)**, where the keys are the train car ID numbers and the values are strings with the contents.

Here are your situations. For each of the following situations, tell me an appropriate ADT, and the specific data structure you would use for this situation. For instance, you could tell me that you would use a List ADT implemented by an ArrayList, or a List ADT implemented by a linked list. Also tell me the specific data types your data structure would contain (e.g., for a BST/hash table, tell me the types of the keys and values).

- a. You want to write a program to decode morse code to English. (Morse code is a code where every letter of the alphabet has a corresponding sequence of dots and dashes). You need a quick way to take these dots and dashes and look up the English letter. What ADT/implementation should be used?
  - b. Suppose you wish to be able to *encode* morse code as well as decode it. Does this change/amend your answer to the first question?
  - c. Continuing from above, suppose you receive a long sequence of morse code (dots and dashes). What ADT/implementation should be used to store this sequence?
2. During class we talked about the idea of a *hash function*: a function that is designed to take an argument and return a *hash code*, which is an integer that can be used to identify which index in a hash table an item should be stored at. In this problem we will see where those functions come from and how they might be designed from scratch.

Suppose we are designing a program to store information about people's birthdays, with the desire that we can type in a birthday and get back the person who has that birthday (to make life easier, assume for the moment that nobody shares a birthday).

So we want to create a Map where the keys are birthdays and the values are names. We therefore

must design a hash function that takes a birthday and returns an integer. Assume birthdays are represented by a month (1-12), a day of the month (1-31), and a 4-digit year. Here are some options for how this hash function might be created:

Option 1: the hash code is year + month + day.

Option 2: the hash code is year \* month \* day.

Option 3: the hash code is (1000 \* year) + (40 \* month) + day.

- a. Suppose we wish to insert the following birthdays and so we must generate hash codes for them:

February 10, 2000

May 7, 2000

October 2, 2000

[ Double check you know which month goes with which integer 1-12 😊 ]

Under option 1, option 2, and option 3, write down the hash codes that would be used for each birthday.

- b. Using your answers to part (a), explain which of the three options for hash codes would do the best job at minimizing collisions. Explain why the other two options cause more collisions than the one you picked.

*Note that in this problem, you are not creating the hash table. I haven't told you the size of the table, nor the names of the people with the birthdays in question (a), so you can't actually create the table here. This problem is just about designing a good hash function.*

3. Suppose we return to our example from class of designing a Map for a veterinary clinic that stores dogs' names mapped to their ages (So a key is a dog's name, and the value is their age). Here are the dogs and their ages, along with the hash codes that we are using. (In other words, the specifics of the hash function itself for this problem do not matter; you just need the integer that the hash function generates.)

Dog	Age	Hash code
Toto	3	26
Lady	4	42
Tramp	2	5
Dino	5	44
Fifi	7	92
Pluto	6	59
Goofy	5	40
Daisy	4	36
Millie	8	12

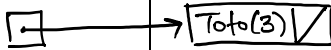
Suppose we create a hash table with 11 slots (so we have an ArrayList of size 11, with indices 0-10). This hash table will use **linear probing** to resolve collisions. Show the table after inserting the nine dogs (and their ages) in the order given above. Remember, to find the appropriate index for a dog (and its age), take the dog's hash code, take the remainder of that code with the size of the table, and that's

the index you use. If there is a collision, follow the linear probing strategy (keep adding 1 to the index, and wrap around to zero if you walk off the end of the table).

You can use the table below or write it out separately. I've already inserted the first dog/age pair for you. Remember, because this is linear probing, only one dog/age pair can fit in each array slot.

Index	Dog (Age)
0	
1	
2	
3	
4	Toto (3)
5	
6	
7	
8	
9	
10	

4. Repeat the previous problem again, but now show the table when you use **chaining**. In this situation, recall that the hash table stores pointers to linked lists, and each element of the linked list can store an individual dog/age pair. I've already inserted the first dog for you:

Index	Pointer to LL
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

5. Using the mergesort algorithm discussed in class, show how mergesort would work on the following array:

6    1    3    4    2    5

Specifically, show each call to the `mergesort()` function and its `array` argument, as well as each call to the `merge()` function and its `leftArray` and `rightArray` arguments. For each call to `mergesort()`, show the array at the end of the function as well.

Do not just draw the "simplified" picture where it appears the two recursive calls to `mergesort` happen simultaneously. Instead, draw this as a "recursion tree" where I can see the **order** that the calls happen in.

Here's a little bit to get you started:

- `mergesort([6 1 3 4 2 5])`
  - split into 6 1 3 / 4 2 5
  - `mergesort([6 1 3])`
  - etc...

6. Suppose we run mergesort on array with  $n$  items, where we guarantee  $n$  is a power of 2.
  - a. How many total calls (as a function of  $n$ ) are there to the mergesort() function (including the initial call)? Give this as an exact answer, not a big-oh value.
  - b. How many total calls (as a function of  $n$ ) are there to the merge() function? Give this as an exact answer, not a big-oh value.
  
7. Suppose we invent a new version of mergesort called **mergesort3** which instead of splitting the array in half, it divides the array in 3 parts (thirds), calls mergesort3 on each individual third of the array, and then merges the three sorted arrays together.
  - a. Write down a recursive  $T(n)$  formula for mergesort3. Hint: this is almost the same as the recursive  $T(n)$  for regular mergesort, but with tiny differences.
  - b. What is the big-oh of this function? Show your work.