CS 5002: Discrete Structures **Fall 2018**

Lecture 7: October 18, 2018 [1]

Instructors: Adrienne Slaughter, Tamara Bonaci

Intro to Linear Data Structures

Readings for this week:
Rosen, Chapter 2.1, 2.2, 2.5, 2.6
Sets, Set Operations, Cardinality of Sets, Matrices

7.1 Overview

Objective: Introduce data structures.

1. How do computers work? (very high level)
2. What is a data structure?
3. Intro to a piece of data we'll use for lecture: a deck of playing cards
4. How we might want to order cards
   - Stack (ie, shuffled deck ready to be dealt)
   - Queue (ordered in hand, ready to play?)
   - "Random" (maybe in some order, but want to find the best at points in time)
5. Lists
6. Storing of lists
   - Contiguous memory
   - Non-contiguous
7. Array
8. Linked List

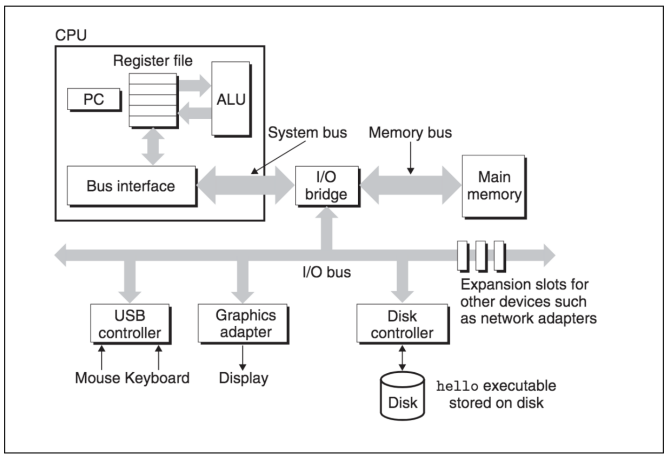
Kinds of problems we can do:

- Given this kind of thing, what kind of data structure?
- Given a compound/crazy data structure, what would it be good for? What are its pros and cons?

7.2 Introduction:

Today we're talking about data structures. Your reading introduced you to arrays, linked lists, stacks, and queues; we'll introduce a map, and talk about them.

To really understand why data structures are important, we need to know just a little bit about how a computer works. Figure 7.2 shows a diagram of a computer system architecture.



The brain of the computer is the CPU, which contains the program, or set of instructions, that's running. The CPU can only act on data in memory. However, data in memory is volatile: it doesn't last. In order to make sure we don't lose data, we have to store it on the disk drive, which allows us to ***persist*** the data. When a CPU needs to do something with data on the disk, it's first copied from the disk into main memory. You might ask why doesn't everything just live in memory: the short answer is that it's expensive. There are tradeoffs between cost ($$$), size (how many pieces of data the memory can hold), and speed. If it gets too big, in order to stay fast, it's expensive. So, we get around that by only storing the data we need in memory when we need it.

You can think of memory like a chest of drawers, and each drawer has an address (or name, or some other way to distinguish one from another). You can put one thing in each drawer. To put something new into a drawer, you have to take whatever is in the drawer out first. In fact, as far as computers go, you can think of it like this: when you run a program, the variables you use each live in one drawer. When the program is over, the computer takes everything out of all the drawers you used and throws it away. If you wanted to keep any of those variables, you have to be sure to take it out and put it somewhere safe (that is, copy it onto the disk drive).

This is hopefully a useful model to help us start thinking about different data structures and why they matter.

7.3 Collections of Items

When we're writing programs, we're usually dealing with collections of items. Maybe we have a bunch of points that describe a polygon, or maybe we are implementing chess, and have the board and the pieces. Or cards, for a card game.

In all of these cases, we have things— items, elements, nodes— that we are using, and need to manipulate. The most basic container or collection of items we have is called the ***list***.

With each program and list of elements, there are a number of things we may want to do with it:

1. ***Get*** or access the kth element: `get(theList, whichElem)`
2. ***Insert*** a node before the kth element: `insert(theList, newElem, whichNode)`
3. ***Delete*** or remove the kth element: `delete(theList, whichElem)` or `delete(myList, 5)` to delete element number 5.

4. ***Combine*** two lists into one: `combine(theFirstList, theSecondList)`
5. ***Split*** a list into two or more: `split(theList, whereToSplit)` or `split(myList, 5)` to split the list into a section of the first 5, and then the rest.
6. Make a ***copy*** of a list: `newList = copy(oldList)`
7. Determine the number of elements (***length***) in a list: `length(theList)`
8. ***Sort*** the elements into ascending/descending order, depending on certain attributes of the elements: `sort(theList, ascending)`
9. Search the list for a element with a particular value in a specified field: `find(theList, queenOfHearts)`

A computer program rarely needs to do all of these things all the time; there are usually 1-2 operations that are considered primary and will be utilized far more than the other operations. The way we structure or store our data elements has an impact on how easy or efficient the above operations are. Therefore, we choose a data structure and organization based on the specific needs of the data.

Let's consider a deck of playing cards: we frequently shuffle the cards at the start of a game, and then from then on the cards stay in the same order. A hand is dealt from the top of the deck, one by one, while all the cards stay in the same order.

Question: Which operations is this? Remove/delete the first node. (that is, $k = 1$)

Another example: when I come across a book I want to read, I put it on my table. When I'm done with my current book, I put it on my bookshelf, then take the next book from the pile on my table. This is a special kind of ordering, called a ***LIFO*** structure: Last In, First Out. In this case, I always read the top book, but I do fear I never get to the bottom!

A better approach might be this: Instead of a table to stack my books, I'll dedicate a shelf to "books I want to read". When I get a new book, put it on the right side of the shelf. When I'm ready for the next book, I take it from the left side. What does that get me? **A chance to get to the end!** This is an example of a ***FIFO*** structure: First In, First Out. This way, I'll be sure to get through the books I want to, in the order I find them and add them to the shelf.

Okay, so I've tried this shelf approach, but I find that sometimes I'm so excited that I really want to read the book I just put on the shelf, so I go ahead and take that one instead. That makes my shelf a mix or combination of LIFO and FIFO.

The first data structure we call a ***stack***, and the second we call a ***queue***. The third structure, as long as I restrict myself to only taking from the left or right (beginning or end), is called a ***deque***, or a "double-ended" queue.

Container Data Structures

These data structures are sometimes called a ***container data structure***. A container data structure is a structure that permits storage and retrieval of data items, independent of content.

This means we can put items in and take them out, and it doesn't matter the actual content of the item.

Containers can be characterized by how items are inserted and retrieved:

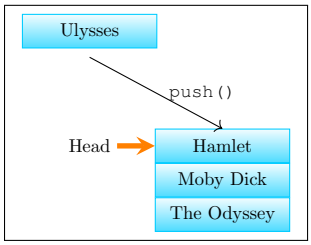
- **LIFO:** Last In, First Out.
- **FIFO:** First In, First Out.

In the next few sections, we'll go over each of these data structures and introduce terminology and operations specific to each. We'll see how these different data structures are each better for some list operations that we identified earlier, while making other operations more difficult. As I've noted, since most programs prioritize one or two operations, we want those programs to use data structures that make those operations easy, while avoiding those that make the required operations more difficult.

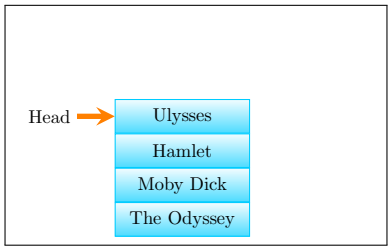
7.4 Stack

Our ***stack*** is a FIFO data structure. It's similar to a stack of books or plates: you start stacking them up, and then you use the one on the top of the pile first.

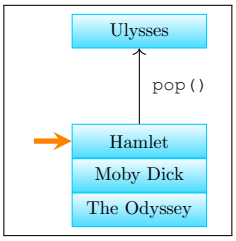
What is a Stack?



Add a new element to an existing stack.



The head pointer is moved up to the new element.



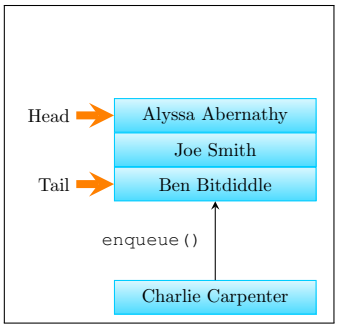
When we pop an element from the stack, we take the top element off.

Stack Operations (Abstract)

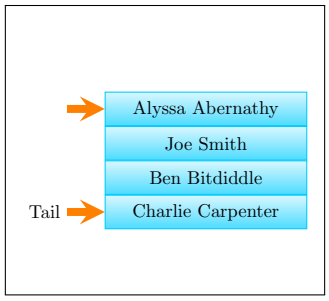
There are a number of typical or standard operations for a stack:

- `pop()` : Get the top item from the stack, removing that item
- `push()` : Put a new item on the stack
- `peek()` : Look to see what the top element is
- `create()` : Create the stack
- `destroy()` : Destroy the stack

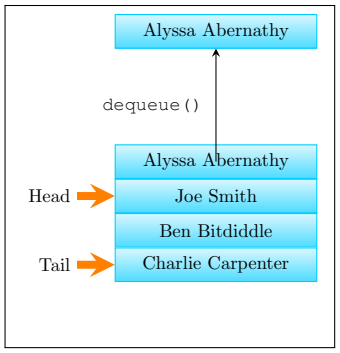
7.5 Queue



When we add an element to a queue, it goes at the end (tail).



The tail pointer is updated.



When we remove an element from a queue, we remove it from the front.

Queue Operations (Abstract)

- `enqueue()` : Put an element at the back of the queue

- `dequeue()` : Get the element that is at the front of the queue
- `front()` : Look at the element at the front
- `back()` : Look at the element at the back/last inserted
- `create()` : Create the queue
- `destroy()` : Destroy the queue

7.6 Deque (Optional)

Sometimes you'll see a deque, or a ***double-ended queue***. A double-ended queue lets you put things in the queue at either the front or the back, and take things out via the front or the back.

Deque Operations (Abstract)

- `insertFront()` : Put an element at the front of the deque
- `insertBack()` : Put an element at the back of the deque
- `removeFront()` : Get the element at the front of the deque
- `front()` : Look at the element at the front
- `back()` : Look at the element at the back/last inserted
- `create()` : Create the queue
- `destroy()` : Destroy the queue

7.7 Exercises/Examples

An *input-restricted deque* is a list that allows items to be inserted only at one end, but can be removed from either end. An input-restricted deque could act as either a stack or a queue. Could an *output-restricted deque* also act as either a stack or queue?

Imagine we have 4 railroad cars on the input side of a track organized like a stack. We perform this sequence of operations:

- PUSH(CAR 1)
- PUSH(CAR 2)
- POP(CAR 2)
- PUSH(CAR 3)
- PUSH(CAR 4)
- POP(CAR 4)
- POP(CAR 3)
- POP(CAR 1)

When we do this, we change the order of the cars from 1-2-3-4 to 2-4-3-1.

Example: If we have 6 cars, numbered 1-6, can they be changed to the order 3-2-5-6-4-1, using just a stack? If so, show the order of operations.

Example: Can they be permuted into 1-5-4-6-2-3 using a stack? If so, show the order of operations.

Example: If we have 6 cars, numbered 1-6, can we use a queue to change the order to 3-2-5-6-4-1? If so, show the order of operations.

Example: Can they be permuted into 1-5-4-6-2-3 using a queue? If so, show the order of operations.

7.8 Deeper Example: Playing Cards

Now that we know about a stacks and queues, we'll introduce a more complex example. In fact, we'll start talking about what all of this looks like in code.

I'm using Python pseudo-code for the rest of these notes.

We're using playing cards. I'll define a playing card as the following class:

```
class Card:
  address = '\0'
  suit = 'Hearts'
  value = 'A'
  isFaceUp = false
  nextCard = '\0'
```

Terminology:

- LOC(A): which "drawer" A is in; the address of A.
- CONTENTS(d11): the "thing" in drawer 11.
- `d11`: drawer 11 (for today, we'll use 'd' to indicate we're referring to a drawer, or a specific location in memory.
- `Card`: The name of a "thing" that we're manipulating. We're going to call it an ***object***, and a particular value of that object an ***instance***. (A real-world equivalent: A playing card is a card, it has a suit and face value; a card is an object, the Queen of Hearts is an instance) Also generally referred to as a data element, node, item, etc.
- `queenOfHearts`: An instance of a "thing": In this case, the Queen of Hearts is a specific card.
- `A.fieldName`: the value of fieldName in "thing" A.

Our cards:

What to do with our cards:

- Count the number of cards in the deck
- Alg A: Take the top card off, turn up the next one.
  - TopCard =

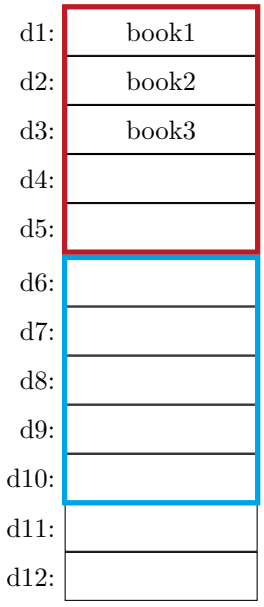
What are the steps to undo algorithm A?

7.9 Data Structure Implementation: Storage Issues

Up until now, we've talked about our most basic data structures. The next thing to talk about is the storage of these data structures. What does this mean? Remember, earlier I said that memory is like a set of drawers (see 7.1. Thinking about the storage of these data structures is like thinking about how to put the elements in these data structures in the drawers.

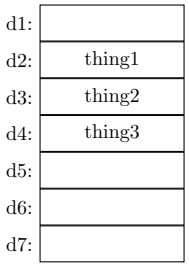
Let's say I have a bunch of data items. If I want to put them in storage (i.e., the set of drawers), I might start at the top, putting an element in the first drawer, then the next element in the second drawer, and so on. If I have n data elements, I'll put them in the first n drawers in the dresser. When I'm looking one data element, I know exactly where the next element is (it's in drawer $n+1$). I also know exactly what element is before it (it's in drawer $n-1$). This is pretty convenient!

Figure 7.2: Our memory storage. Here, the red line indicates that we set aside 5 blocks for kids books, and the blue indicates that we set aside 5 blocks for other books.



Let's look at a real-world equivalent: Believe it or not, every once in a while I get rid of some books. So now, I've got a big pile of books to get rid of. The problem is, some of them are kids' books, and while all the rest are technical books. I want to put all the kids' books together, and all the tech books together, all in storage– in my "dresser". We've already established that we can put a bunch of like elements together in memory; what happens when we have two kinds of elements?

Figure 7.1: A representation of our memory storage. Each slot has an address and can hold one thing.

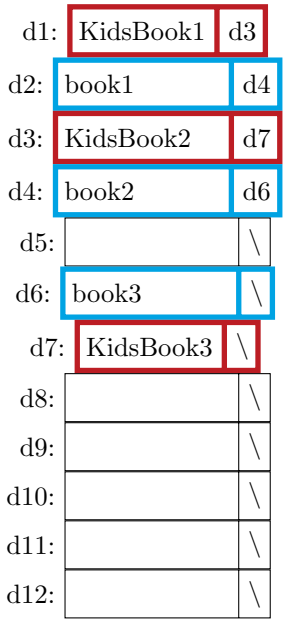


As I go through this pile of books, I pick up one book, determine if it's a kids book or not, and put it in the right place. Well, to know where the right place is, I have to ahead of time decide which group of drawers I'm setting aside for kids books, and what group of drawers I'm setting aside for the other books.

We really have to do something like this:

- Guess how many kids books I have, n
- Set the first n drawers aside for kids books.
- Estimate how many other books I have, x
- Set the $n+1$ through $n+1+x$ drawers (slots) aside for other books.
- Pick up a book
  - If the book is a kids book, put it in the next empty slot in the "kids book" area.

Figure 7.3: Our memory storage. Here, the red line indicates that we set aside 5 blocks for kids books, and the blue indicates that we set aside 5 blocks for other books.



– Otherwise, put it in the next empty slot in the "other" books area.

TODO: psuedocode?

What could go wrong?

This is fine as long as I have no more books than I plan for. What happens if I have 10 kids books though? I fill up all the kids books slots, and then have to put some more somewhere else.

Instead of just putting a book in a slot, let's instead put a book in a slot, and then also store where the next is:

d1:	book1	d2
d2:	book2	\

Here, book1 is in slot d1, and book2 is in slot d2. But book1 keeps track of where the next book is, by storing the address of the next book along with it.

This allows us to do something like this:

TODO: introduce topic of overflow?

The distinction between these two ways to store our data is ***contiguous*** versus linked. There are two data structures that are used to implement all other data structures, with the main distinctions being the difference of contiguous versus linked. Those are the ***array*** and linked list.

Contiguous vs Linked Structures

- **Contiguously-Allocated:** Composed of a single chunk of memory. Examples: arrays, matrices, heaps, hash tables.
- **Linked data structure:** Composed of distinct chunks of memory connected by pointers. Examples: Lists, trees, graphs adjacency lists.

7.9.1 Arrays

An array is simply a contiguous chunk of memory set aside for storage. Generally arrays can only contain one type of data (e.g., an int, or a float, or a char). When you declare an array, you need to specify how big it is, and that's all you get. The nice thing is that it's one chunk in memory, so if you know where one element is, you know where the previous and next one are.

One nice thing about arrays is that we have direct access to elements: If I want the 5^{th} element, I can easily get it: `array[4]` (arrays tend to be zero-based).

We've got a couple standard operations:

- `insert`
- `delete`
- `length`
- `find`

Advantages:

- Constant time access
- Space efficiency
- Memory locality

Disadvantages:

- Can't adjust size at runtime
  - Well, you can, but it's expensive. (time-wise)

7.9.2 Linked Lists

```
## Create an array with 3 elements
myCards = ["Q_Hearts", "K_Spades", "3_Diamonds"]

## Refer to the first element
first = myCards[0]

## Determine how many elements there are in the array
numCards = len(myCards)

## Iterate through the array and print out each element
for card in myCards:
  print(card)

## Add a new element to the array
myCards.append("A_Spades")

## Removes an element from the array (by index)
```

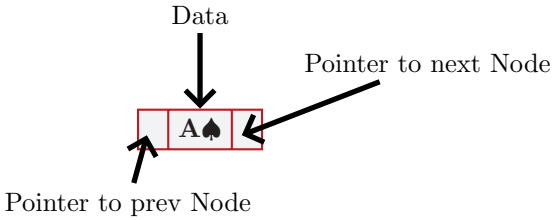


Figure 7.4: A node in a Linked List

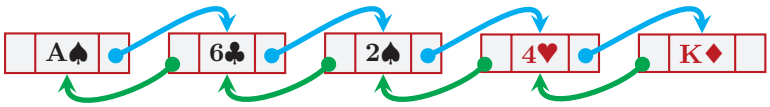


Figure 7.5: A doubly-linked list

```
myCards.pop(2) ## Removes third element

## Removes an element from the array (by value)
myCards.remove("Q_Hearts")
```

```
class LL_node:
  data = aCard
  nextNode
  prevNode

## Create a linked list
list_head = LL_node

## Put the first element in the list
list_head.data = QHearts

## Add an element to the list
def add_elem(head_of_list, new_data):
  cur_node = head_of_list
  while (cur_node.nextNode != '\0'):
    cur_node = cur_node.nextNode
  new_node = LL_node()
  new_node.data = new_data
  new_node.prevNode = cur_node
  cur_node.nextNode = new_node

add_elem(list_head, QSpades)


## Determine how many elements there are in the array
numCards = len(myCards)
def len_list(head_of_list):
  cur_node = head_of_list
  length = 0
  while (cur_node.nextNode != '\0'):
    cur_node = cur_node.nextNode
    length += 1
  return length
```

7.9.3 Comparison: Arrays vs Linked Lists

Comparing

Arrays:

- Linked structures require the overhead of storing pointers.
- Efficient, random access to items
- Better memory locality

Linked Lists:

- It never fills up, unless we've run out of memory
- Insertions and deletions are simpler
- If the items (structs) are large, moving pointers is easier/faster than moving elements

7.10 Evaluating and Comparing Data Structures

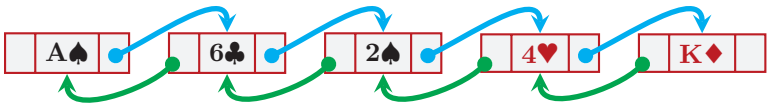
Summary

	Array	Linked List	Stack	Queue	Tree
Type	Linear	Linear	Linear	Linear	Tree/graph
Insert action	Puts	Puts element on top	Puts element on top	Puts element in back	Puts element where specified
Insert Operation	a[n] = i;	insert()	push()	enqueue()	...
Remove Action	a		Returns element at top	Returns element at front	...
Remove operation			pop()	dequeue()	...
Notes			LIFO	FIFO	

7.10.1 Memory

7.10.2 Runtime for typical operations

What's a Linked List?



7.11 Summary

What is a Data Structure?

Data structures are used to structure data we're using. There are two reasons data structures matter: memory and efficiency. We want to know how much memory is used for storing data in one structure versus another, and also how fast or slow the data structure is for what we want to do.

What's coming up?

- Dictionaries
- Trees
- Graphs

Readings for NEXT week:

Rosen, Chapter 4.1, 4.2, 4.3, 4.4

Divisibility and Modular Arithmetic, Integer Representations and Algorithms,

Primes and Greatest Common Divisors, Solving Congruences Solving Congruences