

# Programming Challenges - Week 2

## Data Structures and Strings

Claus Aranha

caranha@cs.tsukuba.ac.jp

College of Information Sciences

April 21, 2014

# Some Notes

## Comments

Don't forget to add a short comment at the start of the code talking about your program. It counts for the final grade.

Specially if you had to submit 5, 10 times before getting the "Accepted" result. Explain what went wrong, and how you managed to fix it.

# Outline for Today

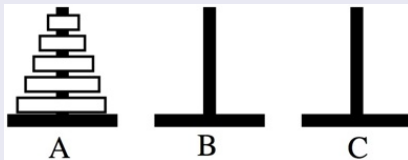
- Chapters 2 and 3 from skiena's book;
- Review of Data Structures;
- Best Practices for choosing them;
- Examples for string use and matching;

# Data Structures

## Data structures are the heart of a program

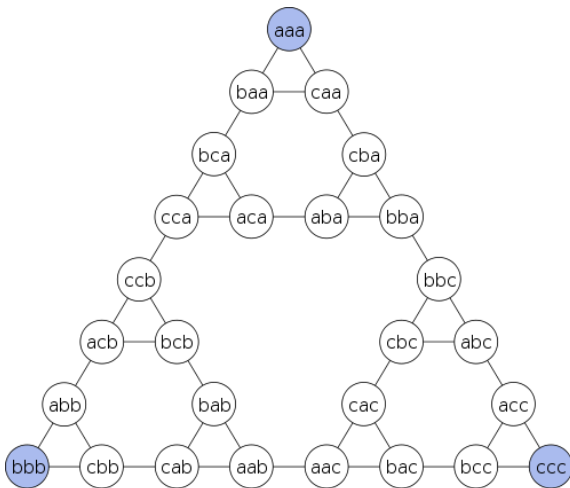
- Using the correct data type can make a problem much easier;
- Using the incorrect data type can make a problem much harder;

## The towers of Hanoi



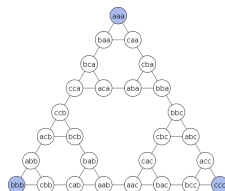
QUIZ: How do you represent the data in this problem?

# An easy way to visualize the Towers of Hanoi



# Explaining the Tower of Hanoi Data Structure

- Each node identifies an state in the problem;
- Each character in the string represents one disk and its position;
- We can have at most 3 state transitions at each state (can you prove it?)
- To solve the Towers of Hanoi problem, we find the path between the start and end states.
- (just beware of state explosion)



# Know your data structures!

There are many types of data structures – You have probably studied all of these before. Did you study [how to implement them](#) or [how to use them](#)?

## Implementation Study – what you learn?

Implement each data structure using the primitives of your language.

- What is the time/memory complexity of a data structure?
- What kind of bugs can come from a data structure?
- How to modify a data structure to fit an specific purpose?

## Use Study – what you learn?

# Know your data structures!

There are many types of data structures – You have probably studied all of these before. Did you study [how to implement them](#) or [how to use them](#)?

Implementation Study – what you learn?

Use Study – what you learn?

Apply data structures to various programs using your language's API.

- How to dish out a complex data structure quickly;
- Which are more or less appropriate to certain problems;
- Similarities and differences between them;



# Using the computer's head, not yours

## Time Complexity

- How fast this algorithm calculates?
- Important for avoiding “Time Limit Exceeded”.
- Depends on the computer's capacity.
- Faster computers are fast even with slow code.

## Code Complexity

- How fast can you debug/modify this code?
- Important for avoiding “Spirit Limit Exceeded”.
- Depends on the programmer capacity.
- Experienced programmers understand more complex code.

# Avoiding Code Complexity

- Don't reinvent the wheel: learn the API for your favorite language;
- Before using or implementing a complex method, check the size of the problem;

# Avoiding Code Complexity

- Don't reinvent the wheel: learn the API for your favorite language;

For Example: stl and string for C++; java.util.\* for Java; string.h and stdlib.h for C.

- Before using or implementing a complex method, check the size of the problem;

# Avoiding Code Complexity

- Don't reinvent the wheel: learn the API for your favorite language;
- Before using or implementing a complex method, check the size of the problem;  
e.g. You want to find the highest number in a collection. If the size of the collection is smaller than 100, a sequential search might make no difference

# Data Structure Review

Let's quickly overview some popular Data structures – you probably know all of them.

If you see an “unfamiliar” data structure, raise your hand for a more detailed explanation.

# Arrays and Linked Lists

Implementation infrastructure for all the other types

## Arrays

- Allow random access to any index;
- Low code complexity;

## Linked List

- Easy to insert/remove items in the middle;
- No predefined size limits;

# Stacks

The simplest data structure.

- Last-in, First-out policy;
- Good on problems where item order is not important;
- Also very useful when a **Nested Structure** is necessary:  
(Recursion, Counting Parenthesis, Depth First Search)

## Common Interface:

- **Push** – add a new item to the stack;
- **Pop** – remove the top item from the stack;
- **Peek or Top** – return the top item, but do not remove it;
- **Full/Empty** – return the status of the data structure;

# Queues

A “fairer” stack, everyone gets their turn.

- First-In, First-Out policy;
- Has some implementation issues:
- Uses: Deck of cards, Ordered Tasks, Breadth-First Search;

## Common Interface:

Same as stacks (Pop/push/peek), just remember the order policy;



# Queues

A “fairer” stack, everyone gets their turn.

- First-In, First-Out policy;
- Has some implementation issues:

## Stack Implementation

Either you have to move all the elements every time you pop an item from the queue, or you have to do a “circular array” implementation, keeping track of the first and last items.

- Uses: Deck of cards, Ordered Tasks, Breadth-First Search;

## Common Interface:

Same as stacks (Pop/push/peek), just remember the order policy;

# Queues

A “fairer” stack, everyone gets their turn.

- First-In, First-Out policy;
- Has some implementation issues:

## Linked List Implementation

You need to keep a pointer to the start of the list, as well as to the end of the list.

- Uses: Deck of cards, Ordered Tasks, Breadth-First Search;

## Common Interface:

Same as stacks (Pop/push/peek), just remember the order policy;

# POP QUIZ!

How do you implement a Queue using Stacks?

# POP QUIZ!

How do you implement a Queue using Stacks?

- Hint: You need two stacks;

# POP QUIZ!

How do you implement a Queue using Stacks?

- Hint: You need two stacks;
- Hint: You need an Input Stack and an Output Stack;

# POP QUIZ!

## How do you implement a Queue using Stacks?

- Hint: You need two stacks;
- Hint: You need an Input Stack and an Output Stack;
- Solution:
  - Push all incoming items into the Input Stack;
  - Whenever you need to pop an item, pop from the Output Stack;
  - If the output stack is empty, pop every item from the input stack and push it into the output stack;

# Dictionaries

Data Structure focused on Content, not Position

- Many different implementations.
- Each different implementation may be more costly to insert, delete or search;
- Some examples: Sorted Arrays, Binary Trees, Hash tables, etc;

## Common Interface:

- **Insert** – Insert data into the dictionary;
- **Delete(x)** – Delete item with data  $x$  from the dictionary;
- **Search(x)** – Retrieve an item with content  $x$  from the dictionary;

# Priority Queue

A mix of a dictionary/queue, sorted by a value

- Different kinds of data, tied together by some orderable value;
- Some examples:
- Implementations: Heap, Sorted Array (with extra data for the sort value);

## Common Interface:

- **Insert, Delete** – same as dictionary;
- **Maximum** – Get the maximum value in the Priority Queue;
- **ExtractMax** – Get and remove the element with the maximum value;



# Priority Queue

A mix of a dictionary/queue, sorted by a value

- Different kinds of data, tied together by some orderable value;
- Some examples:
  - (1) Schedule: Which event is next to happen? Prioritize by date to happen;
- Implementations: Heap, Sorted Array (with extra data for the sort value);

## Common Interface:

- **Insert, Delete** – same as dictionary;
- **Maximum** – Get the maximum value in the Priority Queue;
- **ExtractMax** – Get and remove the element with the maximum value;

# Priority Queue

A mix of a dictionary/queue, sorted by a value

- Different kinds of data, tied together by some orderable value;
- Some examples:
  - (2) Simulation: Similar to scheduling, simulations order agents by the next action to be taken;
- Implementations: Heap, Sorted Array (with extra data for the sort value);

## Common Interface:

- **Insert, Delete** – same as dictionary;
- **Maximum** – Get the maximum value in the Priority Queue;
- **ExtractMax** – Get and remove the element with the maximum value;

# Priority Queue

A mix of a dictionary/queue, sorted by a value

- Different kinds of data, tied together by some orderable value;
- Some examples:
  - (3) Geometry: Scan algorithms order points based on their angles to some location;
- Implementations: Heap, Sorted Array (with extra data for the sort value);

## Common Interface:

- **Insert, Delete** – same as dictionary;
- **Maximum** – Get the maximum value in the Priority Queue;
- **ExtractMax** – Get and remove the element with the maximum value;

# Sets

## A List of Unique Items

- Order is usually not important;
- Elements **NOT** on the Set are as important as the elements in the set;
- Implementations: Dictionaries (restrict size!), Bit Arrays, etc;


### Common Interface:

- **member(x)** – does item x belongs to the set?
- **union and intersection** – same as the mathematical operations on sets;
- **insert and delete** – remember that the items are unique!

# Storing Text

To store text in a computer, we have to transform it into a [code](#) that can be stored into the memory. These codes are displayed using associated [Glyphs](#).

USASCII code chart



Column Row	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	\	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
10	LF	SUB	*	:	J	Z	j	z
11	VT	ESC	+	;	K	[	k	{
12	FF	FS	,	<	L	\	l	
13	CR	GS	-	=	M	]	m	}
14	SO	RS	.	>	N	^	n	~
15	SI	US	/	?	O	_	o	DEL

# The ASCII Code

USASCII code chart

				Character							
Row	Col	b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	b <sub>0</sub>		
										0	1
0	0	0	0	0	0	0	0	0	0	NUL	DLE
0	0	0	0	0	0	0	1	0	0	SP	@
0	0	0	0	0	0	0	1	1	0	A	Q
0	0	0	0	0	0	0	1	1	1	0	1
0	0	0	0	0	0	1	0	0	0	2	B
0	0	0	0	0	0	1	0	0	1	3	C
0	0	0	0	0	0	1	0	1	0	4	D
0	0	0	0	0	0	1	0	1	1	5	E
0	0	0	0	0	0	1	1	0	0	6	F
0	0	0	0	0	0	1	1	0	1	7	G
0	0	0	0	0	0	1	1	1	0	8	H
0	0	0	0	0	0	1	1	1	1	9	I
0	0	0	0	0	1	0	0	0	0	10	J
0	0	0	0	0	1	0	0	0	1	11	K
0	0	0	0	0	1	0	0	1	0	12	L
0	0	0	0	0	1	0	0	1	1	13	M
0	0	0	0	0	1	0	1	0	0	14	N
0	0	0	0	0	1	0	1	1	0	15	O
0	0	0	0	0	1	0	1	1	1	16	P
0	0	0	0	0	1	1	0	0	0	17	Q
0	0	0	0	0	1	1	0	0	1	18	R
0	0	0	0	0	1	1	0	1	0	19	S
0	0	0	0	0	1	1	0	1	1	20	T
0	0	0	0	0	1	1	1	0	0	21	U
0	0	0	0	0	1	1	1	0	1	22	V
0	0	0	0	0	1	1	1	1	0	23	W
0	0	0	0	0	1	1	1	1	1	24	X
0	0	0	0	1	0	0	0	0	0	25	Y
0	0	0	0	1	0	0	0	0	1	26	Z
0	0	0	0	1	0	0	0	1	0	27	[
0	0	0	0	1	0	0	0	1	1	28	\
0	0	0	0	1	0	0	1	0	0	29	]
0	0	0	0	1	0	0	1	0	1	30	^
0	0	0	0	1	0	0	1	1	0	31	_
0	0	0	0	1	0	0	1	1	1	32	`
0	0	0	0	1	0	1	0	0	0	33	a
0	0	0	0	1	0	1	0	0	1	34	b
0	0	0	0	1	0	1	0	1	0	35	c
0	0	0	0	1	0	1	0	1	1	36	d
0	0	0	0	1	0	1	1	0	0	37	e
0	0	0	0	1	0	1	1	0	1	38	f
0	0	0	0	1	0	1	1	1	0	39	g
0	0	0	0	1	0	1	1	1	1	40	h
0	0	0	0	1	0	1	1	1	1	41	i
0	0	0	0	1	0	1	1	1	1	42	j
0	0	0	0	1	0	1	1	1	1	43	k
0	0	0	0	1	0	1	1	1	1	44	l
0	0	0	0	1	0	1	1	1	1	45	m
0	0	0	0	1	0	1	1	1	1	46	n
0	0	0	0	1	0	1	1	1	1	47	o
0	0	0	0	1	0	1	1	1	1	48	p
0	0	0	0	1	0	1	1	1	1	49	q
0	0	0	0	1	0	1	1	1	1	50	r
0	0	0	0	1	0	1	1	1	1	51	s
0	0	0	0	1	0	1	1	1	1	52	t
0	0	0	0	1	0	1	1	1	1	53	u
0	0	0	0	1	0	1	1	1	1	54	v
0	0	0	0	1	0	1	1	1	1	55	w
0	0	0	0	1	0	1	1	1	1	56	x
0	0	0	0	1	0	1	1	1	1	57	y
0	0	0	0	1	0	1	1	1	1	58	z
0	0	0	0	1	0	1	1	1	1	59	{
0	0	0	0	1	0	1	1	1	1	60	
0	0	0	0	1	0	1	1	1	1	61	}
0	0	0	0	1	0	1	1	1	1	62	~
0	0	0	0	1	0	1	1	1	1	63	DEL

- Each character corresponds to an integer;
- The codes corresponding to letters and numbers are in an order correspondent to the natural ordering – you can use a code comparison to perform alphabetical comparison.
- There are many other character encodings!

# String Representation

The data structure used to represent strings is usually:

## Array representation

- length value stored at the start of the array;
- delimiter value (usually '0') at the end of the array;

## Linked List representation

- same as the linked list we saw at the beginning of the course;

# String Operations

These are operations we will often want to perform on a string.  
Which operations are easier/more difficult when using an array  
or linked list?

- Set to a value;
- Concatenate strings ("Add");
- Insert substring;
- Delete substring (middle, end);
- Compare strings;
- Find substring;
- Change the order of characters (invert, etc);
- Search in a string;



# Comparing Character/Strings

## “Pattern Matching” algorithms

- Finding **Patterns** in a **Text**
- Naive algorithm: Compare characters one by one

```
int findmatch(char *p, char *t){
    int i, j;
    int plen, tlen;
    plen = strlen(p);
    tlen = strlen(t);
    for (i=0; i<=(tlen-plen); i++) {
        j = 0;
        while ((j<plen)&&(t[l+i]==p[j]))
            j++;
        if (j == plen) return(i)
    }
}
```

# Comparing Character/Strings

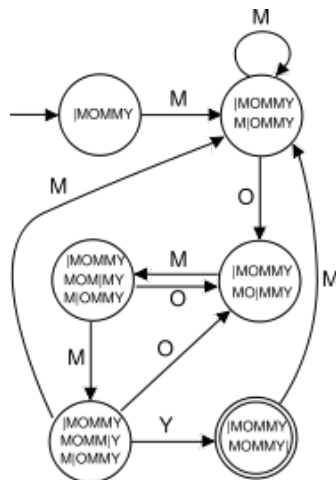
The naive approach can be quite costly! To avoid worst case costs, there are many proposed algorithms for string search

- **State machine based:**  
Build a state machine for the pattern, to avoid backtracking on partial matches. (AAAB x AAAAAAB)
- **Index Based:**  
Build a search tree of relevant substrings in the text;
- **Stub Based:**  
Construct a table of relevant substrings to aid in the search:

# Comparing Character/Strings

## State Machine-based search

- Assemble a grammar/state machine that describes the substring to be found;
- When we find the first letter for the substring, we enter the state machine;
- Finding an unrelated letter tells us if we should restart the state machine, or pick up from a middle point;
- This reduces the need to backtrack, but the cost to make the state machine is high;



# Comparing Character/Strings

Where do we use string comparison?

## Uses of String Searches

- **Bioinformatics**  
GATTACATACCATACACATGAATCACATGA
- **Search Applications**  
fraud analysis, web search, etc

Start  
○○○

Data Structures  
○○○○○○○○○○○○○○○○

Strings  
○○○○○○○○

This Week's Problems  
●○○○

# Jolly Jumpers

Start  
○○○

Data Structures  
○○○○○○○○○○○○○○○○

Strings  
○○○○○○○

This Week's Problems  
○●○○

# Hartals

Start  
○○○

Data Structures  
○○○○○○○○○○○○○○○○

Strings  
○○○○○○○○

This Week's Problems  
○○●○

# Where's Waldorf

Start  
○○○

Data Structures  
○○○○○○○○○○○○○○○○

Strings  
○○○○○○○○

This Week's Problems  
○○○●

# File Fragmentation