COS 226
Midterm Review
Spring 2016

guna@cs.princeton.edu

Time and location:

- The midterm is during lecture on
  - **Wednesday March 9th from 11-12:20pm.**
- The exam will start and end promptly, so please do arrive on time.
- The midterm room is

  - The midterm rooms are
    - Precepts P01, P02, P02A, P03, P03A, P04, P05: McCosh10 (traditional lecture room)
    - Precepts P06, P07, P07A : MCCOH46 (new room)
    - Flipped Lecture/P99: Jadwin A10 (flipped lecture room)

- Failure to go to the right room can result in a serious deduction on the exam. There will be no makeup exams except under extraordinary circumstances, which must be accompanied by the recommendation of a Dean.

Rules

- Closed book, closed note.
- You may bring one 8.5-by-11 sheet (one side) with notes in your own handwriting to the exam.
- No electronic devices (including calculators, laptops, and cell phones).

Material covered

- *Algorithms, 4th edition, Sections 1.3–1.5, Chapter 2, and Chapter 3.*
- *Lectures 1–10*
- *Programming assignments 1–4.*

concepts (so far)

List of algorithms and data structures:

quick-find	quick-union	weighted quick-union	
resizing arrays	linked lists	stacks	queues
insertion sort	selection sort	Knuth shuffle	
mergesort	bottom-up mergesort		
quicksort	3-way quicksort	quickselect	
binary heaps	heapsort		
sequential search	binary search	BSTs	
2-3 trees	left-leaning red-black BSTs		

Hashing – linear probing, separate chaining

Analysis of algorithms

useful formulas

$1 + 2 + \ldots + N.$

$$\sum_{i=1}^{N} i \sim \int_{x=1}^{N} x\,dx \sim \frac{1}{2} N^2$$

$1^k + 2^k + \ldots + N^k.$

$$\sum_{i=1}^{N} i^k \sim \int_{x=1}^{N} x^k dx \sim \frac{1}{k+1} N^{k+1}$$

$1 + 1/2 + 1/3 + \ldots + 1/N.$

$$\sum_{i=1}^{N} \frac{1}{i} \sim \int_{x=1}^{N} \frac{1}{x} dx = \ln N$$

3-sum triple loop.

$$\sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=j}^{N} 1 \sim \int_{x=1}^{N} \int_{y=x}^{N} \int_{z=y}^{N} dz\,dy\,dx \sim \frac{1}{6} N^3$$

$1 + ½ + ¼ + ⅛ + \ldots$

$$\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2$$

$$\sum_{i=1}^{n} r^{i-1} = \left(\frac{1-r^n}{1-r}\right)$$

Union-Find

- A data structure that supports two operations, union and find
- Applications – Connectivity
- Three models

	Union	Find	space
Quick-union			
Quick-Find			
Weighted Quick-Union			

Weighted quick-union

Core ideas of weighted union-find

- The tree height must be logarithmic
- when trees are combined smaller tree is attached to larger tree
- only instance that height of a tree increase is when two trees are the same height

Circle the letters corresponding to arrays that *cannot* possibly occur during the execution of weighted quick union.

	i:	0	1	2	3	4	5	6	7	8	9
A.	a[i]:	1	2	3	0	1	1	1	4	4	5
B.	a[i]:	9	0	0	0	0	0	9	9	9	9
C.	a[i]:	1	2	3	4	5	6	7	8	9	9

Provide an order in which elements in A are possibly added to union-find. Do the same for B and C.

Sorting

Sorting questions

- Which operations is performed the most in general? Compares or exchanges?
- Which sorting algorithms have the best case order of growth O(N)?
- Which sorting algorithms have the worst case order of growth O(N)?
- Which sorting algorithms have the average/worst case order of growth $O(N^2)$?
- Which sorting algorithms have the best/average/worst case order of growth O(N log N)?
- Which sorting algorithms are stable?
- If names array (first name, last name) are sorted in “standard order”, what comparators should we use and in what order?
- Which sorting algorithms are in-place?
- Which sorting algorithms do the least amount of exchanges?

invariants

Insertion sort

Selection sort

Merge-sort(top down)

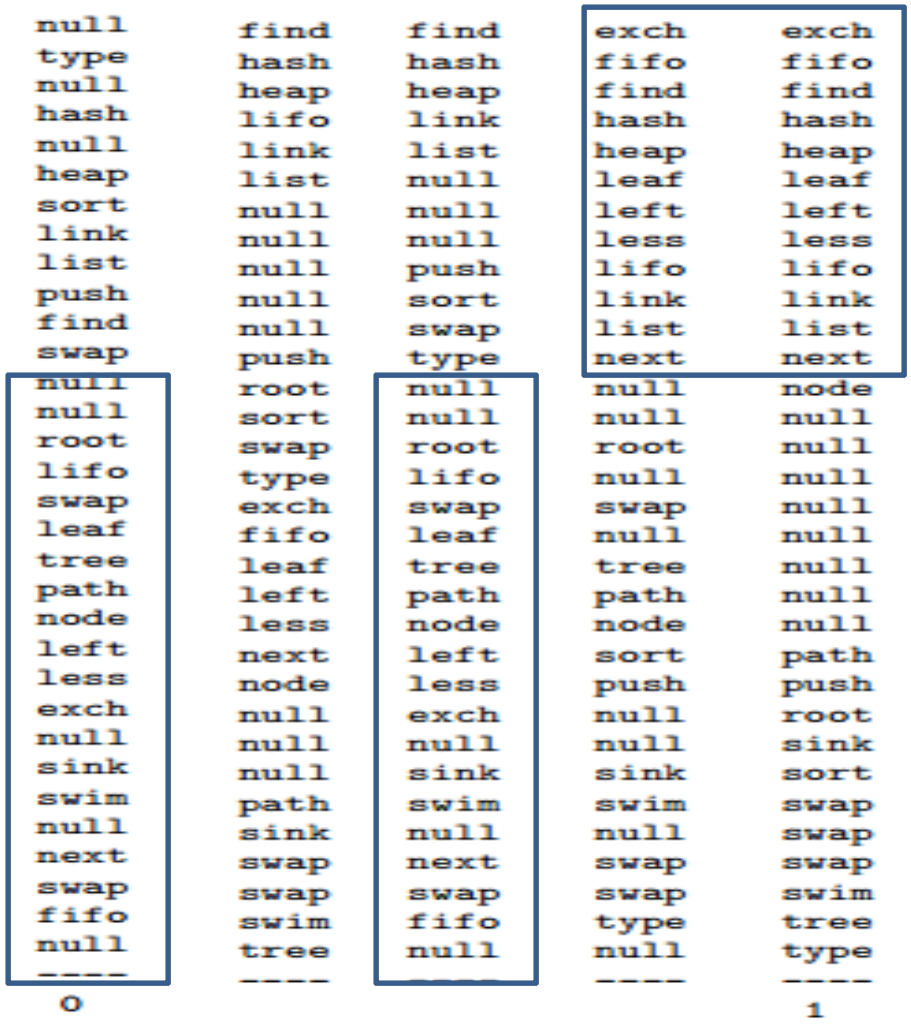
Merge-sort(bottom-up)

Quicksort (2-way)

Quicksort (3-way)

Heapsort

Insertion, selection or mergesort?



Further invariants:

2. Insertion: A[i+1…N-1] is the original

3. Selection: A[0…i] are the smallest in the array and are in the final position

4. Mergesort: A[0..i] is sorted and $I = 2^k$ for some k

2-way and 3-way quicksort

The pivot (first entry) is placed in the final position

2-way - All elements to the left are <= and right are >=

3-way - All elements equal to pivot are in the middle. Left is < and right is >

- How can one recognize 2-way from 3-way if first entry has no duplicates?
  - **2-way** does not swap a lot and do one swap of the pivot at the end. That is, all elements <= pivot must stay in their original position and all elements >= must stay in their original position
  - **3-way** takes the pivot with it and making swaps. All elements < have moved to the left as pivot moves forward

2-way or 3-way?

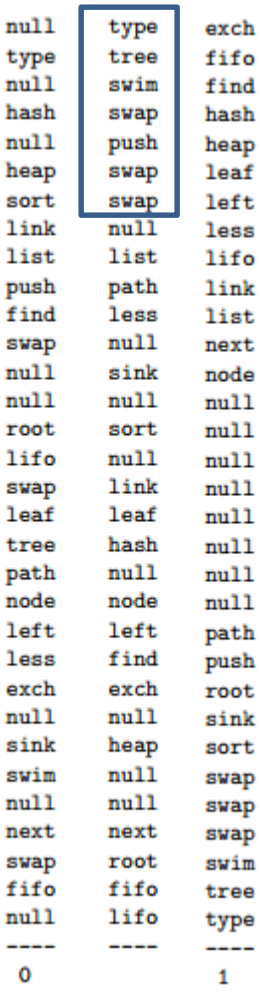
null	hash	fifo	exch
type	heap	next	fifo
null	fifo	null	find
hash	link	hash	hash
null	list	null	heap
heap	next	heap	leaf
sort	find	exch	left
link	lifo	link	less
list	exch	list	lifo
push	leaf	less	link
find	less	find	list
swap	left	left	next
null	node	node	node
null	null	leaf	null
root	null	lifo	null
lifo	null	null	null
swap	null	swap	null
leaf	null	null	null
tree	null	tree	null
path	null	path	null
node	null	null	null
left	path	swap	path
less	tree	push	push
exch	swap	sort	root
null	sink	null	sink
sink	swim	sink	sort
swim	root	swim	swap
null	swap	null	swap
next	swap	type	swap
swap	push	swap	swim
fifo	sort	null	tree
null	type	root	type
----	----	----	----
0			1

5. 2-way quicksort

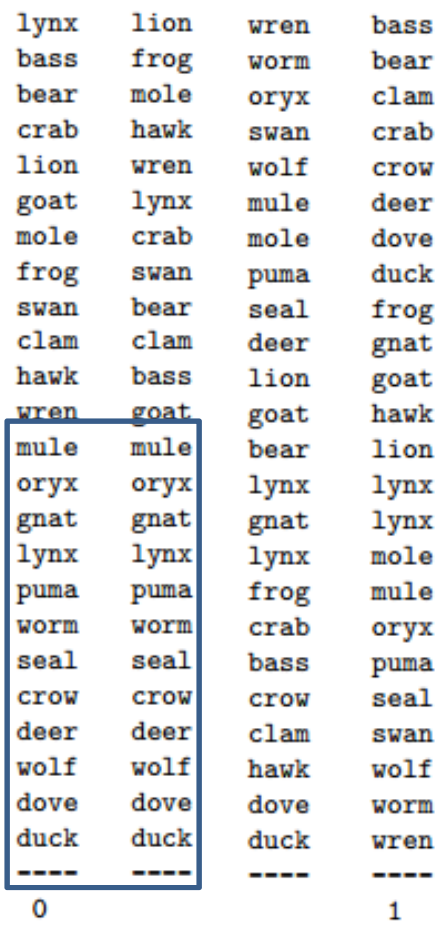
6. 3-way quicksort

navy	corn	mist	bark
plum	mist	coal	blue
coal	coal	jade	cafe
jade	jade	blue	coal
blue	blue	cafe	corn
pink	cafe	herb	dusk
rose	herb	gray	gray
gray	gray	leaf	herb
teal	leaf	dusk	jade
ruby	dusk	mint	leaf
mint	mint	lime	lime
lime	lime	bark	mint
silk	bark	corn	mist
corn	navy	navy	navy
bark	silk	wine	palm
wine	wine	silk	pine
dusk	ruby	ruby	pink
leaf	teal	teal	plum
herb	rose	sage	rose
sage	sage	rose	ruby
cafe	pink	pink	sage
mist	plum	pine	silk
pine	pine	paln	teal
palm	palm	plun	wine
----	----	----	----
0			1

Heapsort and Knuth Shuffle



null	type	exch
type	tree	fifo
null	swim	find
hash	swap	hash
null	push	heap
heap	swap	leaf
sort	swap	left
link	null	less
list	list	lifo
push	path	link
find	less	list
swap	null	next
null	sink	node
null	null	null
root	sort	null
lifo	null	null
swap	link	null
leaf	leaf	null
tree	hash	null
path	null	null
node	node	null
left	left	path
less	find	push
exch	exch	root
null	null	sink
sink	heap	sort
swim	null	swap
null	null	swap
next	next	swap
swap	root	swim
fifo	fifo	tree
null	lifo	type
----	----	----
0		1



lynx	lion	wren	bass
bass	frog	worm	bear
bear	mole	oryx	clam
crab	hawk	swan	crab
lion	wren	wolf	crow
goat	lynx	mule	deer
mole	crab	mole	dove
frog	swan	puma	duck
swan	bear	seal	frog
clam	clam	deer	gnat
hawk	bass	lion	goat
wren	goat	goat	hawk
mule	mule	bear	lion
oryx	oryx	lynx	lynx
gnat	gnat	gnat	lynx
lynx	lynx	lynx	mole
puma	puma	frog	mule
worm	worm	crab	oryx
seal	seal	bass	puma
crow	crow	crow	seal
deer	deer	clam	swan
wolf	wolf	hawk	wolf
dove	dove	dove	worm
duck	duck	duck	wren
----	----	----	----
0			1

Invariants

7. Heapsort :

A[1..N] is a max-heap. In fact A[1..i] is a max-heap for any i

8. Knuth shuffle

A[0..i] is randomized and A[i+1..N-1] is the original

Mystery function

```
public Key mystery(Key key) {
   Node best = mystery(root, key, null);
   if (best == null) return null;
   return best.key;
}

private Node mystery(Node x, Key key, Node best) {
   if (x == null) return best;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) return mystery(x.left, key, x);
   else if (cmp > 0) return mystery(x.right, key, best);
   else              return x;
}
```

Analysis of Algorithms

Analysis of Algorithms

- Performance of an algorithm is measured using
  - comparisons, array accesses, exchanges,
  - memory requirements
- best, worst, average
  - Performance measure based on some specific input type
    - Eg: sorted, random, reverse sorted. Almost sorted

Order of Growth (average/worst)

	insert	delete	search	findMax	findMin	Find kth smallest
Unsorted arrays						
Sorted arrays						
BST						
LLRB						
Binary Heap(PQ)						
Unordered linked list						
ordered linked list (ascending)						
Ordered ST (BST)						
Unordered ST(hash table)						

Classify problems

- Algorithm 1 solves problems of size N by recursively dividing them into 3 sub-problems of size N/3 and combining the results in time c (where c is some constant).

- Algorithm 2 solves problems of size N by solving one sub-problem of size N/2 and performing some processing taking some constant time 1.

- Algorithm 3 solves problems of size N by solving two sub-problems of size N/2 and performing a linear amount (i.e., cN where c is some constant) of extra work.

- Algorithm 4 solves problems of size N by solving a sub-problem of size (N-1) and performing some linear amount of work

Algorithm Analysis

```
int N = Integer.parseInt(args[0]);
String[] genomes = new String[N];
for (int i = 0; i < N; i++) {
      In gfile = new In("genomeFile" + i + ".txt");
      genomes[i] = gfile.readString();
}

for (int i = 1; i < N; i++) {
      for (int j = i; j > 0; j--) {
            if (genomes[j-1].length() > genomes[j].length())
                  exch(genomes, j-1, j);
            else break;
      }
}
```

1. What is the order of growth of the code above?

2. What is the number of array compares in tilde notation?

3. What is number of array reads and writes in tilde notation?

Amortized analysis

- Measure of average performance over a series of operations
  - Some good, few bad
  - Overall good (or not bad)
- Array resizing
  - Resize by 1 as extra space is needed
  - Resize by doubling the size as extra space is needed
  - Resize by tripling the size as extra space is needed

counting memory

- standard data types (int, bool, double)
- object overhead – 16 bytes
- array overhead – 24 bytes
- references – 8 bytes
- Inner class reference – 8 bytes (unless inner class is static)

```
public class TwoThreeTree<Key extends Comparable<Key>, Value> {
   private Node root;

   private class Node {
      private int count;                  // subtree count
      private Key key1, key2;             // the one or two keys
      private Value value1, value2;       // the one or two values
      private Node left, middle, right;   // the two or three subtrees
   }
   ...
}
```

- How much memory is needed for a 2-3 tree object that holds N nodes? Express the answer in tilde notation, big O notation

Priority Queues

- Which operations are supported by a maxPQ? What is the worst case order of growth of each operation?
- What data structures can be used to implement a maxPQ?
- What is the order invariant of a binary heap? What is a shape invariant?
- What is the minimum space requirement to implement a PQ?

PQ short questions

- What is cost of finding a min in a maxPQ?
- What is the cost of finding a random key in a maxPQ?
- What is the cost of creating a sorted list using a minPQ?
- What is/are the major difference/s between a binary heap and a BST?

Hash Tables

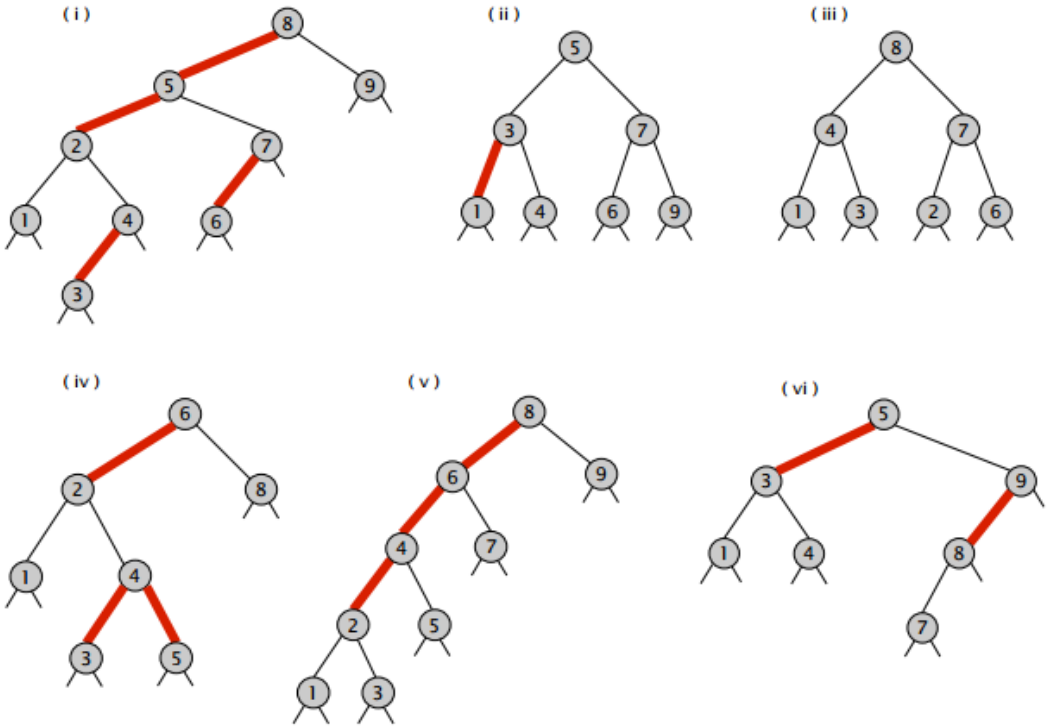
- What are the two types of hashtables?
- How do we assure constant time performance of a hashtable?
- What is the average length of a chain in a good hashtable?

2-3 trees and LLRB's

- Insert 12, 8, 9, 10, 15, 17, 67 into a 2-3 tree

- Show the corresponding LLRB

Legal LLRB's?


(i)
8
5
9
2
7
1
4
6
3
(ii)
5
3
7
1
4
6
9
(iii)
8
4
7
1
3
2
6
(iv)
6
2
8
1
4
3
5
(v)
8
6
9
4
7
2
5
1
3
(vi)
5
3
9
1
4
8
7

Design Problems

Design problems

- Typically challenging
- There can be many possible solutions
  - partial credit awarded
- Usually it is a data structure design to meet certain performance requirements for a given set of operations
  - Example, create a data structure that meets the following performance requirements
    - findMedian in ~1 , insert ~ lg n, delete ~ lg n
  - Example: A leaky queue that can remove from any point, that can insert to end and delete from front, all in logarithmic time or better
- Typical cues to look for
  - *log n* operations are supported by sorted array or balanced BST or some sort of a heap
  - Constant time operations may indicate some use of a uniform hash table
  - Amortized time may indicate, you can have some costly operations once in a while, but on average, it must perform as expected

Guidelines

- Provide ONE clear solution
- Do not write essays
- Do not contradict your early arguments
- No need to write Java code (unless required). Just describe the idea clearly.
- Pay attention to performance requirements
- Pay attention to if you can use extra memory or not, and how much is allowed (constant, linear)

design problem #1

- Design a randomizedArray structure that can insert and delete a random Item from the array. Need to guarantee amortized constant performance
  - Insert(Item item)
  - delete()

Design problem #2

- Design a data structure that can support the following operations
  - Insert(Comparable item) in log N
  - delete() in log N
  - findMedian in constant time

design problem #3

An ExtrinsicMaxPQ is a priority queue that allows the programmer to specify the priority of an object independent of the intrinsic properties of that object. This is unlike the MaxPQ from class, which assumed the objects were comparable and used the compare method to establish priority. You may assume the Items are comparable.

```
public class ExtrinsicMaxPQ<Item extends Comparable<Item>> {
      ExtinsicMaxPQ()                    //do not implement
      void put(Item x, int priority)
      Item delMax()
}
```

If an Item already exists in the priority queue, then its priority is changed instead of adding another item. All operations should complete in **amortized logarithmic time in the worst case**. Your ExtrinsicMaxPQ should use **memory proportional to the number of items**. **For a <u>small</u> amount of partial credit, you may assume that no Item's priority is ever changed (i.e. no item is inserted twice).**

Example:

```
put("cat", 12)      // cat is inserted with priority 12
put("dog", 10)
put("swimp is a raccoon who enjoys fries and does not like to eat dirt", 11)
put("dog", 15)      // dog's priority is changed to 15
delMax()            // deletes dog, which has priority 15, cat is now max
put("fish", 20)     // fish is inserted, and is now max
put("fish", 11)     // fish's priority is reduced to 11, cat is again max
delMax()            // removes cat (priority 12), either swimp or fish now max
delMax()            // removing either swimp or fish is OK, both priority 11
```

Design Problem #4

`public class MoveToFront<Item>`

`MoveToFront()`	*create an empty move-to-front data structure*
`void add(Item item)`	*add the item at the front (index 0) of the sequence (thereby increasing the index of every other item)*
`Item itemAtIndex(int i)`	*the item at index i*
`void mtf(int i)`	*move the item at index i to index 0 (thereby increasing the index of items 0 through i – 1)*

All operations should take time proportional to $\log N$ *in the worst case, where* N *is the number of items in the data structure.*

Design problem #5

MIN-MAX Priority Queue

Standard priority queues come in two flavors: either a minimum priority queue (in which elements can be removed in descending order) or a maximum priority queue (in ascending order); and generally all operations can be implemented in logarithmic time.

Here, we would like to design a priority queue that allows for deletion, at any given time, *either* of the current minimum or the current maximum. In other words, the API should support the following operations:

```
public class MinMaxQueue<Item>
{
  MinMaxQueue()           // creates an empty priority queue

  void insert(Item item)  // add an item to the priority queue
  Item delMin()           // delete the minimum from the PQ and return it
  Item delMax()           // delete the maximum from the PQ and return it

  int size()              // number of elements in the PQ
  boolean isEmpty()       // returns true if the size is zero
}
```

Design an implementation in which all operations are supported in logarithmic worst-case time.

Design Problem #6

LRU cache.

An *LRU cache* is a data structures that stores up to N *distinct* keys. If the data structure is full when a key not already in the cache is added, the LRU cache first removes the key that was *least recently cached*.

Design a data structure that supports the following API:

`public class LRU<Key>`	
`LRU(int N)`	*create an empty LRU cache with capacity N*
`void cache(Key key)`	*if there are N keys in the cache and the given key is not already in the cache, (i) remove the key that was least recently used as an argument to* `cache()` *and (ii) add the given key to the LRU cache*
`boolean inCache(Key key)`	*is the key in the LRU cache?*

The operations `cache()` and `inCache()` should take constant time on average under the uniform hashing assumption.

Key choices for design problems

- Choice of data structure
  - LLRB
    - insert, delete, rank, update, find, max, min, rank (logarithmic time)
  - Hash table
    - insert, find, update (constant time with uniform hashing assumption)
  - Heap
    - delMax, insert (logarithmic)
  - symbol table
    - ordered (same as LLRB), unordered (same as hash table)
  - LIFO Stack and FIFO queue
    - inserts and deletes in amortized constant time with resizable array implementation

Possible/impossible questions

- We can build a heap in linear time. Is it possible to build a BST in linear time?
- is it possible to find the max or min of any list in log N time?
- Is it possible to create a collection where an item can be stored or found in constant time
- Is it possible to design a max heap where find findMax, insert and delMax can be done in constant time?

Possible/impossible questions

- is it possible to implement a FIFO queue using a single array, still have amortized constant time for enque and deque?

- Is it possible to solve the 3-sum problem in n log n time?