1. List-Stack-Queue

1.1. Summary:

List	Using **Array**	Notes
Insert	O(N)	traverse + elements after the node move backward
Remove	O(N)	traverse + elements after the node move forward
Get	O(1)	get element by index
Find	O(N)	traverse the whole array to find the target
Size	O(1)	that's it
Empty or not	O(1)	that's it

List	Using **doubly Linked List**	Notes
Insert	O(N)	traverse to the N'th node, or O(1) if at front/end (best)*
Remove	O(N)	traverse to the N'th node, or O(1) if at front/end (best)* Note: Once you find the element, it's just O(1)
Get	O(N)	traverse to the N'th node
Find	O(N)	traverse the whole linked list
Size	O(N)	traverse to count, or O(1) with a counter
Empty or not	O(1)	that's it

*If the inserting and removing are **at the front/end** and the linked list is **doubly linked**, the time complexity is O(1) because we just link the sentinel to the 2nd or the last 2nd element.

What's more: understand **binary search** in the linked list **(suppose it has been sorted)**, the time complexity of it is O(logN).

Stack	Using **Array**	Notes
Push (Insert)	O(1)	add at the last element
Pop (Remove)	O(1)	remove the last element
Get the top	O(1)	return the last element
Size	O(1)	that's it

Stack	Using **doubly Linked List**	Notes
Push (Insert)	O(1)	add at the last element
Pop (Remove)	O(1)	remove the last element
Get the top	O(1)	return the last element
Size	O(N)	traverse to count, or O(1) with a counter

List is last-in-first-out. The push and pop operations are same for both array and linked list implementations, as the operation only add/remove the last element. The get the top operation is also the same because we just get the last element. The only different is getting the size.

Queue	Using **Array**	Notes
Enq (Insert)	O(1)	add at the last element
Deq (Remove)	O(N)	remove the first, and move others forward
Get the front	O(1)	return the first element
Return size	O(1)	that’s it

Queue	Using **doubly Linked List**	Notes
Enq (Insert)	O(1)	add at the last element
Deq (Remove)	O(1)	remove the first element
Get the front	O(1)	return the first element
Return size	O(N)	traverse to count, or O(1) with a counter

Queue is first-in-first-out, so the enq operation simply adds an element at the end, which are both O(1). However, the deq operation removes the first element, so the linked list implementation takes O(1) and the array implementation takes O(n) because it needs to move every element forward after removing the first element. The get the top operation is also the same because we just get the first element. Getting the size is different.

1.2. Example Questions:

1.2.1. List

-With linked cells, the get operation: O(N)

-With array, the get operation: O(1)

-With array, the remove operation: O(N)

-With linked cells, the add operation worst: O(N)
-With array, the add operation best: O(1)

1.2.2. Stack
-With array, the push operation worst: O(1)
-With linked cells, the push operation best: O(1)

1.2.3. Queue
-With array, the deque operation worst: O(N)

2. Trees

2.1. Summary

2.1.1. Basic Terms

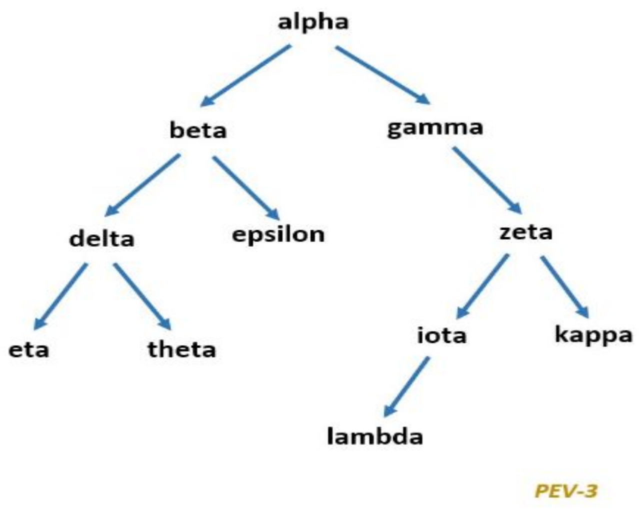
-Tree height: the height of the root
-Depth of a node: length of path from **root** to the node
-Height of a node: length of **longest path** from its **leaf** to the node
-Root has depth 0, leaf has height 0 (It's 0, not 1!!!)
-n-ary tree: the max number of children of any node determines "N"

2.1.2. Tree Traversal **(all traversals are O(N)!!!)**

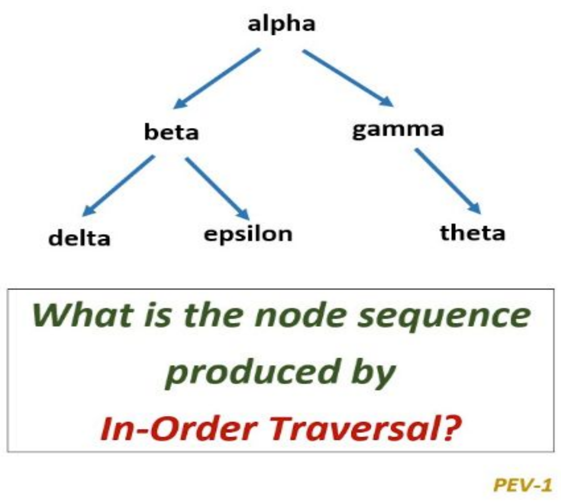
-Depth-First
- -PreOrder: root-L-R
- -PostOrder: L-R-root
- -InOrder: L-root-R

-Breadth-First

2.2. Example Questions



The height of the tree: 4 (root alpha has height of 4 from lambda to itself)
The height of node "beta": 2 (eta:0, delta:1, beta: 2)
The depth of node "beta": 1 (root:0, beta: 1)



In-Order Traveral: delta, beta, epsilon, alpha, gamma, theta
Breadth-First Traversal: alpha, beta, gamma, delta, epilon, theta

2.3. Tree representations

2.3.1. Linked cells

2.3.2. Array

3. BST

3.1. Summary

-A special case of binary tree, in which for each node, all nodes in the **left** subtree **is smaller** (if it has) and in the **right** subtree **is larger** (if it has)

-A BST could have two extreme structures: **linear** or **balanced/complete**

-relates to the question of given a height, how many max/min nodes?

-e.g. max # nodes in BST with height 15: 2^16 - 1 (complete BT)

-e.g. min # nodes in BST with height 15: 16 (linear BT)

-Get familiar with these operations

-contains/get, insert, find min/max, remove

-general idea of contain/get, insert, find min/max is to use the rule that the left child is smaller and the right child is larger, then recursively compare to find the target node.

-remove is special: consider different situations: leaf node, node with one child, node with two children

-(a) leaf node: just remove it!

-(b) node with one child: just make its parent link to its child!

-(c) node with two children: find **minimum** node in the **Right** subtree and replace the node with the minimum node found.Then delete the minimum node found (this requires recursion).

- Sorting with BST: **O(N*logN)**

-Build a BST and then use in-order traveral to print the results.

-**Avg:** O(N)-(inserts) * O(logN)-(average of insert) + O(N)-(traversal)

-Better than sort with a list (O(N^2))

-**Worst:** O(N) * O(N)-(worst of insert) + O(N) = (O(N^2))

-So it's not always better than sorting with a list.

BST	Worst	Average	Notes
Insert	O(N)	O(logN)	The difference is between “linear” BST and “balanced” BST
Remove	O(N)	O(logN)	
FindMin	O(N)	O(logN)	
FindMax	O(N)	O(logN)	
Contains	O(N)	O(logN)	
Get	O(N)	O(logN)	
Empty or not	O(1)		that’s it
Size	O(1)		Keep Conter

3.2. Example Questions

-For any set of keys, there is only one BST: **False.**
-For a BST with N nodes, the average of post-order traversal is: O(N)-always!
-Other time complexity questions refer to the table
-BST sorting is always better than sorting with a list: Not always!

4. Priority Queue

4.1. Summary

-A PQ can be implemented in linked list, arrary or binary heap.

Priority Queue	Doubly Linked List	Sorted List	Binary Heap
enq	O(1) add at the tail	O(n) find the rightest	O(logN)
deq	O(n) traverse to find	O(1) get the head	O(logN)
front	O(n)	O(1)	O(1)

Operations in PQ:
-enq: puts an item with its priority into the queue
-deq: removes the highest priority item from the queue
-front: returns the highest priority item

4.2. Example Questions

-Most questions are about time complexity, refer to the table above.
-Two tricky questions from class: PQ as **a normal linked cell list**, the worst time for “enq” is O(N) and for “front” is O(1). Personally I suppose the “normal linked cell list” means a sorted list, not a normal doubly linked list. (Not 100% sure with these two questions)
-PQ as **a balanced BST**, the worst time for “enq” and “front” are both O(logN),just as insert and find min/max in BST. And worst is O(N).

5. Binary Heap

5.1. Summary

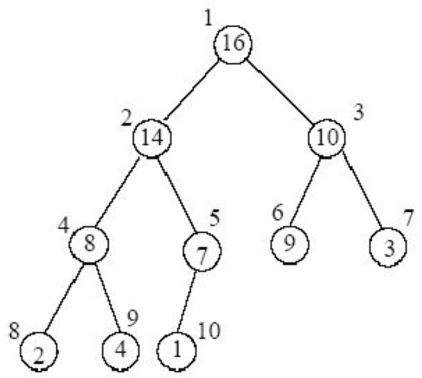
-Binary Heap != Priority Queue, just an implementation.

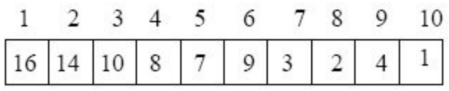
-Two properties

-Structure: it is a complete (not full) binary tree (as left as possible)

-Heap-order: every child >= parent (min BHEAP), but **the order of children do not matter!** Each subtree of a BHEAP is also a BHEAP.

-Implementation: Array





1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

-Parent: [i/2]
-Lchild: 2*i
-Rchild: 2*i + 1
-So we can infer the tree structure from the array, and vice versa.

BHEAP	Avg	Worst	Notes
Insert	O(1)	O(logN)	1. put the new item at the righest (at the end of the array) 2. compare with its parent node, if it is smaller (suppose minimum BHEAP), swap until not smaller or at the root
Delete	O(logN)	O(logN)	1. remove the root item (because it is the smallest) 2. pull out the last node (at the end of the array) and repeat: - if fit, leave it at the new node - if not, bubble down: swap with smaller child, continue
GetMin	O(1)	O(1)	get the tree root (which is at the slot 1 in the array)
Search	O(N)	O(N)	just traverse the array

-Sorting with BHEAP

-Problem: 1) the array is not sorted 2) build a binary heap and then retrieve the result still has a high time complexity

-If build and then retrieve: insert N & delete min N times, **O(NlogN) for both**

-i.e. the worst case is N (items) * logN(insert) + N (items) * logN(delete)

-Sorting with the new build method: **O(N) + O(logN)**

- -1. Put the values into an array.
- -2. Use the array to generate a binary heap (just fit the structural property, the order does not matter)
- -3. Change the order of nodes (start with the parent of the last node, back to root) then "bubble down" the node: swap the parent node with the smaller child; if needed, continue swapping until leaf.

-Summary of sorting:

- -1. bubble sort: O(N^2)
- -2. BST sorting: average is O(N) + O(logN), worst is O(N^2)
- -3. BHEAP sorting: O(N) + O(logN) with the new build method

5.2. Example Questions

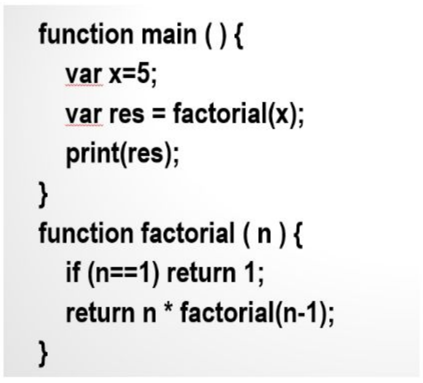
-Check whether a heap is a min/max heap: use the two properties

6. Other topics

6.1. Recursion

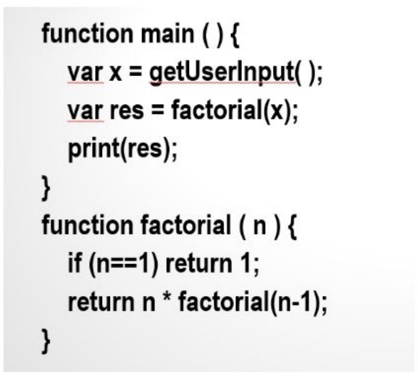
6.2. Runtime stack and heap

Questions are about the amount of run-time stack space:



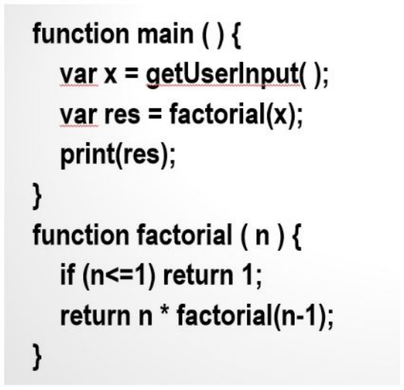
```
function main ( ) {
    var x=5;
    var res = factorial(x);
    print(res);
}
function factorial ( n ) {
    if (n==1) return 1;
    return n * factorial(n-1);
}
```

finite(bounded)



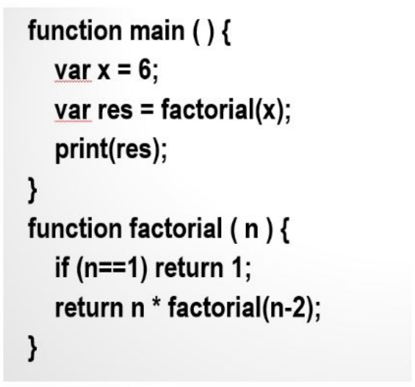
```
function main ( ) {
    var x = getUserInput( );
    var res = factorial(x);
    print(res);
}
function factorial ( n ) {
    if (n==1) return 1;
    return n * factorial(n-1);
}
```

infinite (input may be smaller than 1)



```
function main ( ) {
    var x = getUserInput( );
    var res = factorial(x);
    print(res);
}
function factorial ( n ) {
    if (n<=1) return 1;
    return n * factorial(n-1);
}
```

unbounded but finite (determined by input)



```
function main ( ) {
    var x = 6;
    var res = factorial(x);
    print(res);
}
function factorial ( n ) {
    if (n==1) return 1;
    return n * factorial(n-2);
}
```

infinite (it skips n==1)

Garbage collection in Java: Dynamic memory allocated in the heap, but no longer reachable from a program, is returned to the un-allocated heap.