Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Index of arrays → Level order traversal of tree

parent index = current index / 2

left child index = current index * 2

right child index = (current index * 2) + 1

Hash Table General Information

A. Collision Resolution

Open Addressing			Closed addressing
More cache friendly. Require deleted markers and inefficient if there are many deletions and insertions.			Storing of the object is completely dependent on the hash function. - Easy to obtain elements with same hash.
Linear Probing	**Quadratic Probing**	**Double Hashing**	**Separate Chaining**
Index = (base+step*1)%M	Index = (base+step*step)%M	Index = (base+step*secondary)%M	Elements are linked lists
primary cluster -> covers the base address of a key which increase the running time beyond O(1). Long sequences of filled slots increase search and insert time.	Secondary cluster -> clusters formed along the path of probing instead of around the base address. may stuck in an infinite loop searching unless **a**<0.5 and **M** is prime number. increase time taken if collision occurs. may result in a waste of memory.	Scanning forward by the second hash function for the next empty slot. - reduce primary and secondary clustering.	If two keys **x** and **y** both have the same hash value I, both will be appended to the front/back of a doubly linked list i. - Thus remove(v) requires search(v) whose time complexity depends on the load factor **a**[1] search(v) and remove(v) are worst O(1+**a**), best O(1). insert(v) will be O(1) Unable to fully utilize unused slots

Definitions
-M = HT.length = the current hash table size. -base = (key%HT.length). -step = the current probing step starting from 1. -secondary = smaller_prime – key%smaller_prime, which is computed by another hash function.

For n items, in a table of size m, assuming uniform hashing, the expected cost of an operation is:

$$\leq \frac{1}{1-\alpha}$$

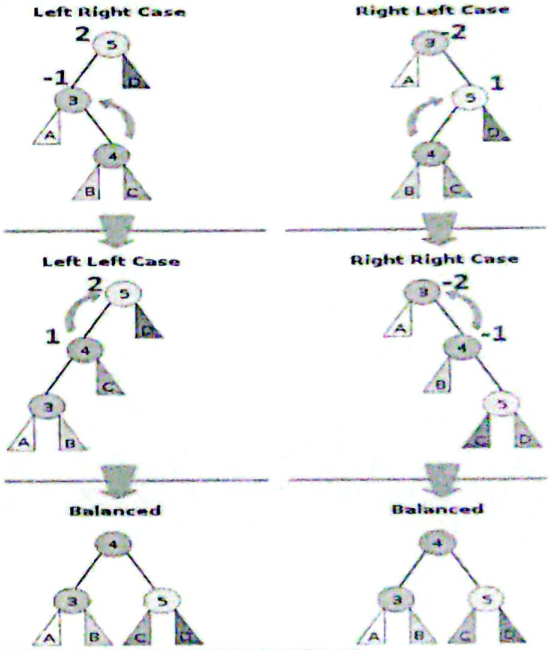
Example: if ($\alpha = 90\%$), then E[# probes] = 10

Algorithm	bubble	select	insert	merge	quick	random quick	count	radix
Average	N^2	N^2	N^2	NlogN	NlogN	NlogN	N	N
Sorted Ascending	N	N^2	N	NlogN	N^2	NlogN	N	N
Sorted Descending	N^2	N^2	N^2	NlogN	N^2	NlogN	N	N
Worst Ascending	N	N^2	N	NlogN	N^2	NlogN	N	N
Worst Descending	N^2	N^2	N^2	NlogN	N^2	NlogN	N	N
Stability[1]	Y	N	Y	Y	N	N	Y	Y
In place[2]	Y	Y	Y	N	Y	Y	N	N
Recursive	N	N	N	Y	Y	Y	N	N
Comparisons	Y	Y	Y					
Div & Conq				Y	Y	Y		

Data Structure	Vector	Singly Linked List	Stack	Queue	Double Linked List	Deque
access i^{th} element	1	N	N	N	N	1
search(num)	N	N	NA	NA	N	NA
peek-front()/top()	1	1	1	1	1	1
peek-back()	1	1	NA	1	1	1
insert(0, num)	N	1	1	NA	1	1
insert(n, num)	1	1	NA	1	1	1
insert(i, num)	i / worst N	N	NA	NA	N	NA
remove head	N	1	1	1	1	1
remove tail	1	N	NA	NA	1	1
remove(i)	i/N	N	NA	NA	N	NA

Binary Search Tree

Query		Update		Others	
Search(v)	O(h) ~ O(N)	Insert(v)	O(h) ~ O(N)	Rank(v)	rank(Min())=1, rank(Max())=N – log(N)
Pred(v)/Succ(v)	O(h)	Remove(v)	O(h) ~ O(N)	Select(k)	Get the element at rank k – log(N)
Inorder traversal	O(logN) / O(N)	Create BST		BST Height	Floor($\log_2 N$) ~ N-1
Min / Max	O(h) ~ O(N)			AVL Height	O($\log_2 N$) or h < 2$\log_2 N$



AVL Tree

1) bf(x) == +2; bf(x.left) == 1; right_rotate(x);
2) bf(x) == -2; bf(x.right) == -1; left_rotate(x);
3) bf(x) == +2; bf(x.left) == -1; left_rotate(x.left); right_rotate(x);
4) bf(x) == -2; bf(x.right) == -1; right_rotate(x.right); left_rotate(x);

Algorithm	Average	Worst
Space	O(N)	O(N)
Search	O(logN)	O(logN)
Insert	O(logN)	O(logN)
Delete	O(logN)	O(logN)
Rank	O(logN)	O(logN)
Select	O(logN)	O(logN)

Insert(v): update the height and balance factor of the traversed vertices and use one of the 4 rotations to balance it at most once.
Remove(v): update the height and balance factor of the traversed vertices and use one of the 4 rotations to balance it up to logN times.

Successor
- If V has right subtree -> get min in the right subtree
- If V has no right subtree -> traverse the ancestor until a right turn

Predecessor
- If V has left subtree -> get max in the left subtree
- If V has no left subtree -> traverse the ancestor until a left turn

Binary Heap		
Algorithm	**Average**	**Worst**
build heap / create	N	NlogN
search	N	N
Insert – bubble sort upwards in O(log N) height	1	logN
top – peek max/min element	1	1
extractmax / pop – bubble sort downwards in O(log N) height	logN	logN
Heap_sort – call extractmax N times	NlogN	NlogN