Data Structures

Module 1:

Array :→ Contiguous area of memory consisting of equal sized elements

Constant time [O(1)] → Read and write access

Can have 1D, 2-D ... nD array.

We can have row major or column major arrays

Arrays are great if we want to add or remove at the end, but very expensive to add or remove otherwise.

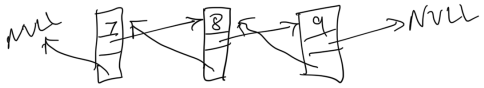
Linked List: It is an array of nodes. The first node is connected to the next and so on. We can add, remove anywhere. Here add and remove in the beginning is less expensive than at the end. It is easier by storing tail.

This can be made [illegible]

But it makes access to end easier, not before that.

Single linked list

With single linked list, we can add after, remove after. But we can't add before, remove before less expensively with double linked list.



List elements are not contiguous.

Stack → abstract data type which adds key to the collection and returns the most recent key. Like a stack of books.

Applications → balanced parenthesis.

We can implement stacks with arrays as well as linked list. With arrays, we have constant space. With linked list, we need extra space to store pointers as well. So if we know what would be the maximum size of stack, we can use arrays.

Stacks are LIFO [Last in First Out]

Queue → Abstract data structure which adds key to the collection and remove the least recent key.

These are FIFO [First in, first out]

These are used in server operations.

These can be implemented with array and linked list, but linked list is preferred.

Trees → These are used for geography, arithmetic operations, sentences etc.

There are different types of trees. Binary Search Tree (BST) is a tree where values in left subtree are smaller and values in right subtree are larger than the root node.

A tree is empty (or) a node with a key and list of child trees.

Trees are recursively defined.

There are two main ways to walk a tree.

(1) Depth-first → eg. inorder. [left → root → right]

Inorder is only for binary tree. Preorder and postorder are for all trees. Preorder → root → child.

(2) Breadth-first → we use queue. For a node, we dequeue it, and enqueue its child and print the node. This gives level-order-travel.

Module 2:

Problem with arrays are that they are static, i.e. we cannot change their size during run time. We can only set their size at compile time.

One solution would be to dynamically allocate the array during run-time.

```
int* arr = new int[size];
```

But the problem with this is that we always [illegible] array we

don't know the 'size' of the array we want. Here 'dynamic array' are used. They only store the pointer to the array. And once their size has reach, it creates a new array of greater size and copy the previous array and delete the previous array.

Dynamic Array: An abstract data type with the following operations (at a minimum):
(1). Get(i) (2) set(i,val) (3) PushBack(val)
(4) Remove(i)

We store
(i) arr: dynamically-allocated array
(ii) capacity: size of the dynamically-stored array.
(iii) size: number of elements currently stored in array.

These are implemented in many programming languages.
C++ → vector ; Java → ArrayList ; Python → list

Amortised Analysis:

Sometimes, while calculating worst-case time we may overstate it. So we calculate average cost of 'n' operation. This is called amortized cost.

$$\text{Amortized cost} = \frac{\text{Cost}(n \text{ operations})}{n}$$

(1) Aggregate Method:
It directly uses definition of amortized cost.
Assuming, that we double the size of array.
$C_i \rightarrow$ cost of i^{th} insersion.

$$C_i = 1 + \begin{cases} i-1 & \text{if } i-1 \text{ is power of 2} \\ 0 \end{cases}$$

$$\text{Amortized cost } (AC) = \frac{\sum C_i}{n}$$

$$= \frac{\sum_{i=1}^{n} 1 + \sum_{j=1}^{\lfloor \log_2(n-1) \rfloor} 2^j}{n}$$

$$\sum_{j=1}^{\lfloor \log_2(n-1) \rfloor} 2^j = 2 + 2^2 \cdots 2^{\lfloor \log_2(n-1) \rfloor}$$

$$= 2\left[1 + 2 + 2^2 \cdots 2^{\lfloor \log_2(n-1) \rfloor - 1}\right]$$

$$= 2\left[1\left(2^{\lfloor \log_2(n-1) \rfloor} - 1\right)\right]$$

$$= 2 \times 2^{\lfloor \log_2(n-1) \rfloor} - 2 \leq 2 * 2^{\log_2 n} - 2$$

$$\leq 2n - 2$$

$$\leq O(n)$$

So,

$$AC = \frac{n + O(n)}{n} = \frac{O(n)}{n} = O(1)$$

That is amortized cost is $O(1)$ for all operations but worst case cost is $O(n)$.

(2) Banker's Method: Here we charge extra for each cheap operation. Then we save the extra charge as tokens in your data structure. And then we use these tokens to pay for expensive operations.
So basically we pay one token for insersion, one token on the element inserted and one buddy token on 2^{nd} prior element $[(i-2)^{th}]$.

(3) Physicist's Method: Here we define a potential function (ϕ) which maps the state of data structure.

$\phi(h_0) = 0$ 'h' → data structure
0 → initial time

$\phi(h_t) \geq 0$

So amortized cost for operation t:
$C_t + \phi(h_t) - \phi(h_{t-1})$

We need ϕ such that:
(i) If C_t is small, ϕ should increase.
(ii) If C_t is large, ϕ should decrease by the same scale.

$$\text{Amortized cost} = \sum C_i = \sum \left[C_i + \phi(h_i) - \phi(h_{i-1})\right]$$

$$= \sum C_i + \sum \left[\phi(h_i) - \phi(h_{i-1})\right]$$

$$= \sum C_i + \phi(h_n) - \phi(h_0)$$

But we know $\phi(h_0) = 0$

so $AC = \sum C_i + \phi(h_n) \geq \sum C_i$

That means we have come up with lower bound of the amortized cost i.e sum of true cost.

* In aggregate method, we saw that AC is $O(1)$ if we double the capacity. But this does not hold good if we 10 times the capacity.

so AC is $O(n)$, if we [illegible] capacity 10 times once the size is full.

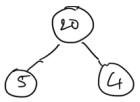
Module 3:

Priority Queue: This is an inbuilt data structure in many languages like C++ (priority_queue), Java (PriorityQueue) and Python (heapq). It is a generalization of queue where each element is assigned a priority and elements come out in order of priority.
It is mainly used to extract max.
There are many algorithm which use priority queue.
For example, Dijsktra's algo, Prim's algo, Huffman's algo, Heap sort.

Naive Implementation: If we use doubly linked list, to find max, we need $O(n)$ time. If we use sorted array, finding max is $O(1)$. But insersion is $O(n)$.

Binary heap: These are of two types binary max heap and binary min heap. Binary max heap is a binary tree where the value of each node is at least the values of its children.
Ex. 

It cannot have duplicate nodes.
[illegible] bottom and



(10) To insert, we add node [illegible] start swapping with its parent until we have binary max heap.

We keep adding to one leaf. So after adding n nodes, we would get height as $O(n)$.

If we have a problematic node, we swap with largest child and hence shift down.

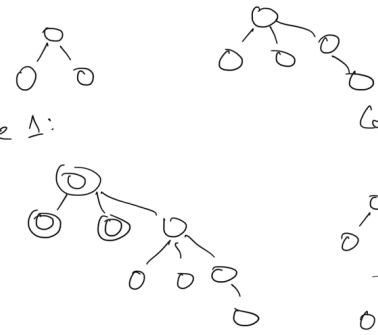
There is a way to keep the binary tree complete in the priority queue. The advantage is the height is minimum. And extracting max take $O(h)$ time, so time will also decrease.

Heap Sort: Given an array, we insert in priority queue and then extract Max one by one.
This gives us sorted array. Time $O(n \log n)$.
Also it is an inplace sort algorithm.
Heap sort worst case – $O(n \log n)$
Quick sort average case – $O(n \log n)$
So, we can have an "Intrasort" algo. In this, we start with quick sort and if the time exceeds $c \log n$ for some 'c', then we shift to heap sort which is gauranteed to complete in $O(n \log n)$

Disjoint Sets:

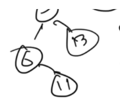
A maze problem could be solved with the help of disjoint set data structure. A disjoint-set data structure supports the following operations.
(i) MakeSet(x) → creates singleton set {x}
(ii) Find(x) → returns ID of x
If x, y lie in same set find(x) = Find(y)
(iii) Union(x,y) → merges x and y
So to reach point B from A, we take a point C and make set of all neighbours. Then for each neighbour, we call union. This creates bigger sets. Finally we check if find(A) = find(B)
We can use array to represent elements of set.
While merging, we get $O(n)$ time.
We can use linked list to get $O(1)$ time by doing tail→next = head. But then running time to find is $O(n)$. Moreover, we need more storage.
All these are naive implementation.
A more efficient way: USE TREES.
A [illegible] to represent a complete

We use a rooted tree [illegible] set. Merging also becomes easier. We use shorter tree to merge.

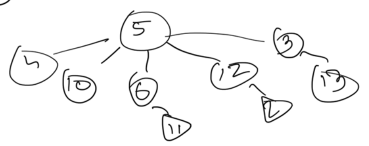


We use an array (rank) to store the height of each tree.
if rank(i) ≥ rank(j)
parent[j] = find(i)
All operations are $O(\log n)$

One more interesting thing is path compression.
Suppose we have a tree. We ran operation to find parent of 6.
It showed us that 3 and 12 both also have 5 as its [illegible]



parent. So why waste the information. We use it and make a new tree



This is called path compression. Here, the running time is $O(m \log^* n)$.

$\log^* n$ → number of times we need to apply $\log_2$ to reach 1

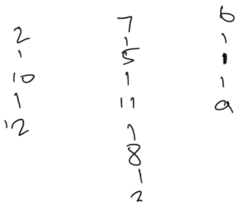
For example

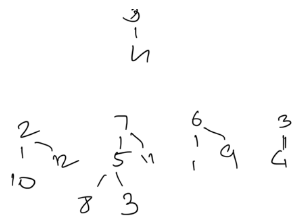
$4 \xrightarrow{\log_2} 2 \xrightarrow{\log_2} 1$

so $\log^* 4 = 2$. Similarly $\log^* 16 = 4$

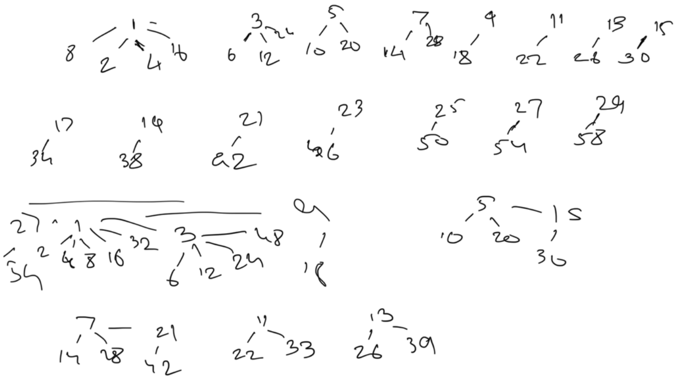
For $n = 2^{65537}$, $\log^* n = 5$. Most of the time we would have $n \le 2^{65537}$.

So amortised cost is $O(\log^* n) \approx O(5) = O(1)$ Constant time





$1 \times 2 \times 1 \times 1 = 2$



Module 4:

Hashing:

This is a very powerful algorithm. It is used in many places. One of the emerging application is blockchain. Cryptocurrencies are based on blockchain. Moreover, it is used in compilers of programming language. For example, there are many keywords in programming languages like "for", "while", "return" etc. So while compiling, the compiler has inbuilt hash table to store the function of these keywords. So this is where hashing is used. Another example, file location. Computer stores human readable file location to actual file location.

One more example is store log access of IP addresses. There are two ways to store.

(1) Direct Addressing → There are 2^{32} bit IP addresses. So we make an array of size 2^{32} and store counter for every IP address. Hashing allows $O(1)$ search and update. But the space taken would be $O(n)$.

So direct addressing works when sample size is small. For example, a normal person would have less than 1000 phone numbers in contacts so we can create a hash function to hash 1000 phone numbers from 0 to 999.

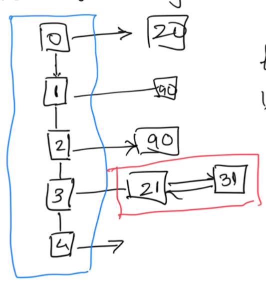
For any set of objects S and any integer $m>0$, a function $h: S \to \{0,1,2 \ldots m-1\}$ is called a hash function. where m is defined as the cardinality of hash function h. Hash function ... input. Moreover, we need should be fast and unique for each input. ... small cardinality. But this is not true if size of sample is greater than size of array. So this is called collision.

So formally we say, we need small probability of collision. Here comes chaining.

(2) Chaining → To have small collision probability, we use chaining. Using hash function, if we get collision, we create a list at that point in array.

For example, we have an array of size 5. And we get collision at 3 with two inputs (21 and 31).

So our array would be:



Here our array is in blue and list at collision is in red.

We have used doubly linked list

Hash table is implemented as map or set.

C++ → unordered_set (and) unordered_map

Java → HashSet (and) HashMap

Python → set (and) dict

Map stores key, value pair. Whereas set only store value.

So to decrease space complexity, we need a good hash function.

We could use a random function. But when using same input again, we would get same output.

So hash function should be <u>deterministic</u>.

We do not have a hash function which solves our problem

So we need to use randomization.

We defined a family of set and use randomly.
Let U be the universe (set of all possible keys).
A set of hash functions $\mathcal{H} = \{h : U \to \{0, 1, 2 \dots m-1\}\}$ with cardinality as 'm'. This $\mathcal{H}$ is called universal family if for any two keys $x, y \in U$, $x \neq y$ the probability of collision $P[h(x) = h(y)] \leq \frac{1}{m}$

What does this mean?
It means that a collision $h(x) = h(y)$ for any FIXED pair of different key x and y happens to be no more than $1/m$ of all hash function $h \in \mathcal{H}$. It can give different collision for different keys

Load factor: The ratio n/m between no. of objects and size of hash table is called load factor.
$n/m > 1 \longrightarrow$ atleast one collision.
So average run time $O(1+\alpha)$ where $\alpha = n/m$

If we don't know the size, we could use dynamic hash table. So for example, if $\alpha > 0.9$ we change our hash table. Let T be our hash table

$$\alpha = \frac{T.\text{number of keys}}{T.\text{size}}$$

if $\alpha > 0.9$:
Create T_{new} with size $2 \times T.size$:
Choose h_{new} with cardinality $T_{new}.size$.
T using h_{new}.
Insert all values in new
$T = T_{new}$, $h = h_{new}$.

Example of hash function

$h_p^{a,b}(x) = ((ax+b) \bmod p) \bmod p$. where p is prime number. For any 'x', choose 'p' such that $p > x$. For example, if data set is phone numbers of size 7, p could be 10000019.
→ These are 'p' choices for 'b' and $(p-1)$ choices for 'a'.
So total hash functions $p(p-1)$.

General Case: Hashing phone numbers
(1) Define maximum length L of a phone number.
(2) Convert phone numbers to integers from 0 to $10^L - 1$
(3) Choose prime number $p > 10^L$
(4) Choose hash table size m
(5) Choose random hash function from universal family $\mathcal{H}_p$ (choose random $a \in [1, p-1]$ and $b \in [0, p-1]$).
This gives us $h_p^{a,b}(x) = ((ax+b) \bmod p) \bmod p$. and use chaining.
This maps phone numbers to names.

Hashing names:
Here we need to hash strings. Previously we hash numbers.
Steps:
(1) Convert each character $s[i]$ to integer using ASCII encoding or Unicode etc.
(2) Choose big prime number p
(3) Repeat the same steps as did for phone number.

Polynomial Hashing:

$$\mathcal{P}_p = \left\{ h_p^x(s) = \sum_{i=0}^{|s|-1} s[i] x^i \bmod p \right\}$$

where $p \to$ fixed prime number and $1 \leq x \leq p-1$

Pseudo code:

```
hash ← 0
for i from |s|-1 down to 0:
    hash ← (hash * x + s[i]) mod p
return hash
```

So for s of length 3, $hash = \left(s[0] + s[1]x + s[2]x^2\right) \bmod p$

This is implemented in Java as method hashCode.
It uses $x = 31$ and for technical reason avoids mod p.

Lemma:
For any two different strings s_1 and s_2 of length at most $L+1$, if you choose h from $\mathcal{P}_p$ at random (by selecting a random $x \in [0, p-1]$), the probability of collision $P[h(s_1) = h(s_2)] \leq \frac{L}{p}$
This follows from the fact that the equation:

$$a_0 + a_1 x + a_2 x^2 \dots a_L x^L = 0 \pmod p$$

for prime p has at most L different solutions x.
For very large 'p', we get our hash function, but hash table size is also p. So we need to optimize.

Optimization:
(1) First apply random h_x from $\mathcal{P}_p$.
(2) Hash the resulting value again using universal family of integers $h_{a,b}(x)$.

$$h_m(s) = h_{a,b}(h_x(s)) \bmod m$$

Lemma:
For any two different strings s_1 and s_2 of length at most $L+1$ and cardinality m,

$$P[h_m(s_1) = h_m(s_2)] \leq \frac{1}{m} + \frac{L}{p}$$

So if $p > mL \Rightarrow \frac{1}{m} + \frac{L}{p} < \frac{1}{m} + \frac{1}{mL} \leq \frac{2}{m} \to O(\frac{1}{m})$

Search and modification run in $O(1)$ on average!
So phone book can be mapped using two hash functions; one to map phone numbers to names and other vice-versa.

Substring:
Given a text T (website, book, Amazon product page) and a string P (words, phrase, sentence), find all occurrance of P in T. For example, specific term in Wikipedia article, gene in genome or detect files infected by virus by code patterns specific to viruses.
Naive Algo: Using two for loops, one for T and other for P. Time $O(|T||P|) \sim O(n^2)$

Rabin-Karp's Algo:
Use hashing. If $h(p) \neq h(s)$, then definitely $P \neq S$. If $h(P) = h(s)$, there is small probability of collision, so
Then check if $P = S$.
$P[h(P) = h(s)] \leq \frac{|P|}{p}$ $p \to$ prime number.

"False alarm" → $h(P) = h(S)$ but $P \neq S$

$$P[\text{False Alarm}] = P[\text{collision}] \leq \frac{|P|}{p}$$

Time:

$h(P) \rightarrow O(|P|)$

$h(T[i \ldots i+|P|-1]) \rightarrow O(|P|)$

↳ Computed $|T| - |P| + 1$ times

So total time $O(|T||P|)$ same as naive.

Optimization:

$$\text{hash} = \sum s[i] x^i \bmod p$$

Example: T = beach

encoding = beach → 1 4 0 2 7

hash("ach") $= 0 + 2x + 7x^2$

hash("eac") $= 4 + 0x + 2x^2$

We can see that coefficients of $1, x$ and x^2 are the encodings of each letter. So now, we do not need to recalculate hash for each substring to compare with P.

Assuming $H[2] = \text{hash}(\text{"ach"})$ and $H[1] = \text{hash}(\text{"eac"})$

$H[2] = 0 + 2x + 7x^2$ and $H[1] = 4 + 0 + 2x^2$

$= 4 + x[0 + 2x]$

$= 4 + x[0 + 2x + 7x^2] - 7x^3$

$= 4 + x * H[2] - 7x^3$

So previous hash can be calculated in $O(1)$ if we know current hash.

$$H[i+1] = \sum_{j=i+1}^{i+|P|} T[j] x^{j-i-1} \bmod p \quad \text{—(i)}$$

$$H[i] = \sum_{j=i}^{i+|P|-1} T[j] x^{j-i} \bmod p \quad \text{—(ii)}$$

Using the above two equations:

$$H[i] = \left[\sum_{j=i+1}^{i+|P|} T[j] x^{j-i} \right] + T[i] - T[i+|P|] x^{|P|} \bmod p$$

$$H[i] = x \sum_{j=i+1}^{i+|P|} T[j] x^{j-i-1} + " - " " "$$

$$H[i] = x H[i+1] + \left(T[i] - T[i+|P|] x^{|P|} \right) \bmod p$$

$x^{|P|} \bmod p$ → can be computed once before running loop

So running time is $O(|T| + |P|)$ instead of $O(|T||P|)$

Blockchain:

Consider that we have many transactions on our accounts. If a person needs to see his transaction, we cannot show them our complete account. So instead we just show hash values. The person checks this value with their transaction's hash value. A list of transaction hash value (hash chain) is stored in a block (just like a page in notebook). A chain of these blocks make our account's transaction notebook called – Blockchain

It is not necessary that these should be money transactions.

Blockchain in essence is just a diary which is very hard to forge. Typically it is distributed and allows anybody to generate blocks which are verified and then added into the chain.

Merkle Tree:

To check a transaction in a block, we need to check all transaction which takes $O(n)$ time. But merkle tree data structure does in $O(\log n)$.

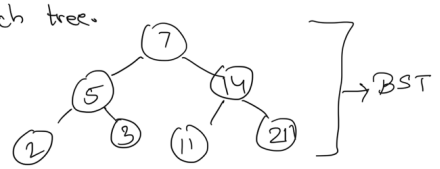
Module 5:

Binary Search Tree:

We need a data structure which allows us to search and check neighbours efficiently. Hash table cannot do this. What if we use array, here we cannot do neighbour search. But what if we do sorted array. In this, insertion and deletion is difficult but we can do binary search. Whereas, if we use linked-list, insertion and deletion is easy but we cannot do binary search.

So we need binary search tree (BST).

We can see sorted array is a better start as we can do binary search. Using the idea of binary search, we have binary search tree.



BST should satisfy one important property.

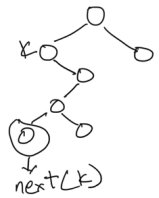
Search Tree Property:

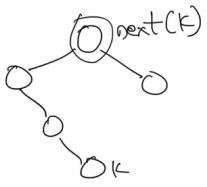
The value of a node should be larger than any key present in the left subtree and smaller than any key present in the right subtree.

Operations:

(i) Find (K): → returns node with key K.
Start with top node. Compare with root node. If K is smaller than root key, check recursively in left subtree, else in right. If equal, return.
If we reached NULL, K not found.

(ii) Next (K): return next largest node.
If 'K' has right child, go to the leftmost child of the right child. Else go to next right ancestor.



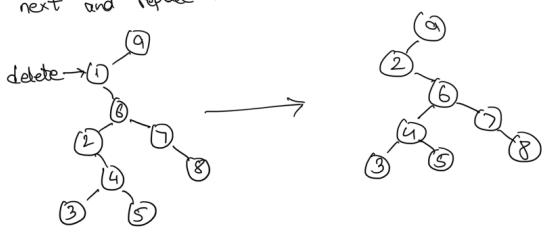


(iii) RangeSearch (x,y): Search nodes with keys from x to y

(iv) Insert (K): insert K in BST at its particular place.
Using find (K), it gives us where to add 'K'. Then insert.

(v) Delete (K): delete the node K.
Change the node with next node. If node is not have right child, use left child. If it does, replace node with

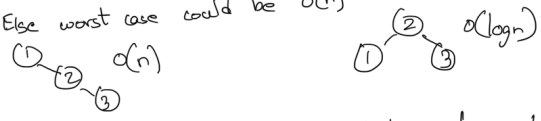
its next and replace its next with its right [illegible]



Time analysis:

(i) Find → o(height) → o(logn) [If we have balanced tree]

Else worst case could be o(n)



So if we want to balance, we need to perform rotations.

AVL Trees: Better way to keep tree balanced.

We add "height" field to the node.

AVL Property: |Height (Left tree) - Height (Right tree)| ≤ 1

New Operations:

(i) Merge: Combine two BST

If one tree has values all smaller than other tree, we just join the two by one node which is largest of the tree with smaller values. Make larger value tree as right child.

(ii) Split: Break tree into two.

One tree with all values less than 'x' and one with larger values. The smaller value tree is just the left subtree [illegible] whole tree of 'x'. The larger value tree is merge of [illegible] without x and its children, and right subtree of x

```
if x < R.Key:
    (R1, R2) ← split (R→left, x)
    R3 ← merge (R2, R→right, R)
    return (R1, R3)

if x > R.key:
    (R1, R2) ← split (R→right, x)
    R3 ← merge (R2, R→left, R)
    return (R1, R3)
```

Applications:

(i) Finding k^{th} smallest element

(ii) Finding median

(iii) Finding 25% percentile.

For finding k^{th} smallest, we store size below, in node.

Another application is to flip array colour. For a block, it can be black and white. After a block 'x', we need to flip colour.

To do this, we create one BST with give colour and one tree with inverted colour. After 'x'th node, we split both trees. Then we merge left tree of first BST and right tree of inverted BST to get the final result.