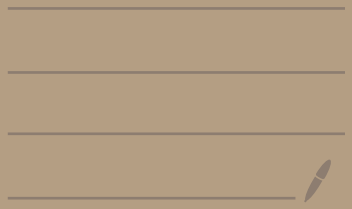
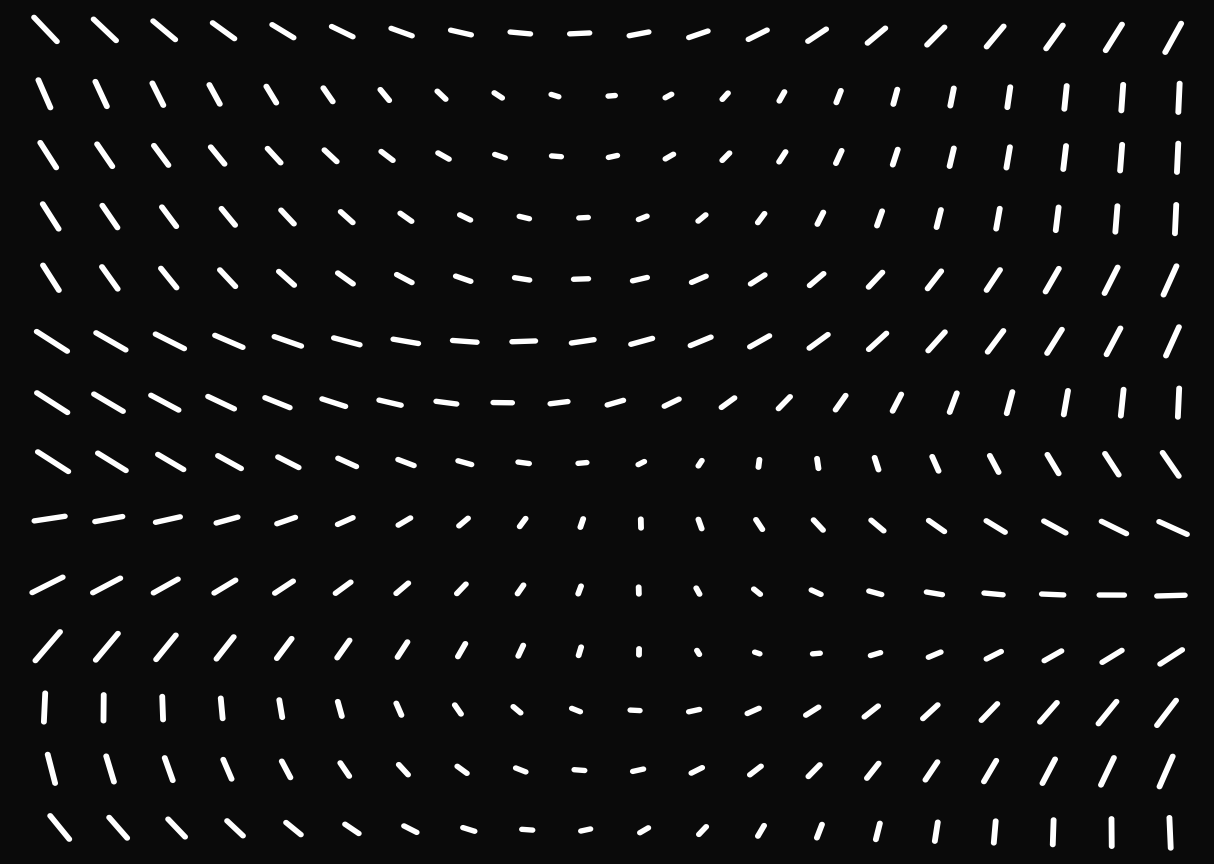
CISC 235



Lecture Notes



Lecture 1: Data Structures and Algorithm Complexity

What is a data structure

- data structure is a way to store, organize and manage data effeciently
- implemented as an Abstract Datatype

Organizing Data: Bookshelf Analogy

- Random order, Easy to insert, hard to Search
- Alphabetical order, Harder to Insert, faster to Search
- catcgorized + Alphabetical, Best for both

Algorithm Complexity & Effeciency

- Time Complexity → How fast an algorithm runs
- Space Complexity → How much memory an algorithm uses
- Experimental → Run program, measure time
- Theoretical → Use Big-O Notation

Linear Search vs. Binary Search

- Linear Search: Checking every book one by one
  - Best case: First book
  - worst case: Not in list (O(N))
  - Average case: N/2 steps

Bad for large inputs

- Binary Search
  - Splits the list in half, Compare, halve again
  - worst case: O(logCn))
  - Bineary Search is exponentially faster than linear Search

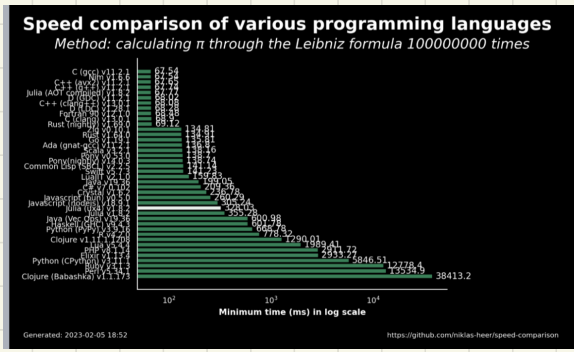
Books (N)	Linear Search (Worst Case)	Binary Search (Worst Case)
16	16 steps	4 steps
1,024	1,024 steps	10 steps
131,072	131,072 steps	17 steps

- Algorithms + Data Structures, work together to make effecient algorithms.

Lecture 2: Data structure and Program Complexity

Computational Complexity → Helps us choose best algorithm

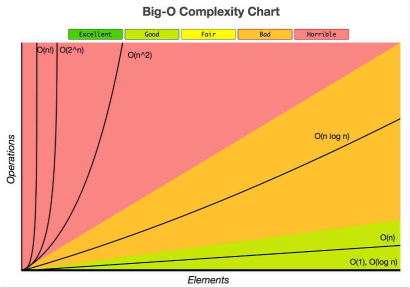
- Is the measure of how much time and space an algorithm requires
- To measure complexity, use theoretical analysis (Big O Notation) and use experimental analysis.



Algorithm analysis

- understanding resources that the algorithm takes
- Time complexity (run time)
- Space complexity (memory)

Note: Typically low level, compiled languages tend to run faster, higher level languages are usually slower.



$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	"n log n" or linearithmic
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential

Big-O, Big-Omega, Big theta

$O(g)$: upper bound on how quickly the function grows (worst scenario)
$\Omega(g)$: Lower bound on how quickly the function grows (best scenario)
$\Theta(g)$: Tight bound on how quickly the function grows (best and worst)

Note: To prove big theta prove big O and big-omega first.

Growth Rate

- Look at most dominant n term
- growth factor is calculated by dividing the functions value at n by its value at n/2. This shows how much function increases when input size doubles.

Counting Operations

A1:

```
total = 0      → 1
for i = 1 to n:
  total = total + i   → 2n
                → 2n + 2
```

$T_{A1}(n) = 1 + (2n+2) + 2n$

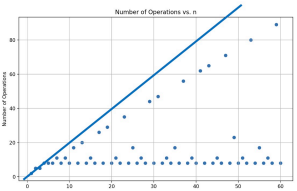
$T_{A1}(n) = 4n + 3$

Emperical Data, blue dots.

$f(n) \leq 2n$ for $n \geq 3$

C is Scaling factors

Let's plot how f(n) grows when increasing n



<- If you try to calculate the number of operations for n ranges from 1 to 60

Is 1 prime? False. Number of operation
Is 2 prime? True. Number of operation
Is 3 prime? True. Number of operation
Is 4 prime? False. Number of operation
Is 5 prime? True. Number of operation
Is 6 prime? False. Number of operation
Is 7 prime? True. Number of operation
Is 8 prime? False. Number of operation
Is 9 prime? False. Number of operation
Is 10 prime? False. Number of operatio
...

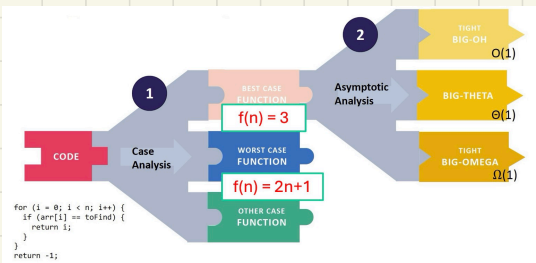
Is the runtime O(1) or O(n)?

How about c = 2 and n_0 = 3, f (n) ≤ 2 n for n ≥ 3

? Could n_0 = 4? 100?
Could c = 3? 100?

27

Perform Case Analysis when Comparing Functions



* Appendix Tricks Proofs

Lecture 3 - List

Linear data structures: List, Stack, Queue
Non-linear ds : trees, graphs

Abstract data type (ADT) - Concept that defines what operations can be performed on the ds, For example a List is an ADT, but linkedList / Array are implementation.

ADT - has Data Properties and Operations

→ Data properties include: items one after another, specific position, can have dups

Basic Operations

- Add elements
- delete elements
- Access elements by position
- Traverse through elements

- Python list has continuous memory locations
- Python list is an array of pointers, to memory
- Python dynamically resizes lists

Useful list methods

- .append(x)
- .insert(i, x)
- .remove(x)

Big O - Complexity of List OPS

NON mutated

len(data)	O(1)
data[i]	O(1)
.count	O(n)
.index(value)	O(K+1)
val in data	O(n)
data1 + data2	O(n1+n2)

mutated

data[i] = val	O(1)
data.append(val)	O(1) *
data.insert(val)	O(n)
data.pop(val)	O(1)
.reverse()	O(n)
.sort()	n log n

Non mutating Operation - Does not modify original list
Mutating operation - Modifies original list

LinkedLists

- Collection of nodes that stores data and pointer
- Do not use continigous memory
- First and last nodes are head / tail

* must keep reference to head

Fast inserts / Deletes
Slow accesing / Searching

Common Operations

Traverse	O(n)
insert	O(1)
remove	O(1)
Check empty	O(1)
get size	O(n)

Lecture 4: Stacks and queues

Stacks

- LIFO, Last in, First out

Operations	T.C
push(e)	O(1)
pop()	O(1)
top()	O(1)
is_empty()	O(1)
len(s)	O(1)

- reverse a word
- undo functions

Implementing a Stack using Linked List

- Stack follows LIFO
- The head is used at the top of the stack
- push happens at the head O(1) time
- pop happens from head at O(1) time

Infix, Prefix and Postfix Notation

- Infix: Operators between operands → A + B
- Prefix: Operators before operands → +AB
- Postfix: Operators after operands → AB +

General notes:

- The order of operands does not change
- recent high precedence operator will be used first

Infix → Postfix

1) identify precedence
2) convert by switching operation order

Ex) A + B * C → A + BC * → ABC * +

(A - B) * (C + D) → AB - (AB -)(CD +) *
↘ CD + AB - CD + *

Using a stack, you read in all operands first then, pop operators in precedence level order

Practise

Ex1

infix to postfix

A + B * C - D
BC *
A + BC *
ABC * +
ABC * + D -

2) (A + B) * (C - D)
AB +
CD -
AB + CD - *

3) A * B + C / D - E
AB *
CD /
AB * CD / + E -

Postfix to infix

1) AB + C -
((A + B) - C)

2) ABC + *
A * (B + C)

3) ABC * + D -
(A + (B * C)) - D

4) AB + CD - *
(A + B) * (C - D)

5) AB * CD / E - +
((A * B) + (C / D) - E)

Lecture 5 - Queue implementation

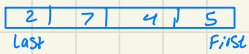
- Queue is a FIFO data structure
- Elements are added to the back, enqueue and removed from the front dequeue.

Note: Also an ADT, can be implemented with linked lists

Common Operations

enqueue() - O(1)
dequeue() - O(1)
first() - O(1) -> gets first in line
is_empty() - O(1)
size() - O(1)

Array-based Queue

- 

dequeue -> 5, 4, 7, 2.
The dequeue operation requires O(n) time because you shift everybody up a spot.

Linked-List Based Queue

- Uses a linked list for effecient insertions and deletions
- enqueue() and dequeue are both O(1)
- Due to, only the pointer needing to be re-referenced

Implementing Queues with Two Stacks

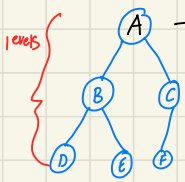
- You can use two queues to mimic stack operations
  - Pushing adds new elements to queue 1
  - Popping moves elements from queue 1 -> queue 2

- You either make push costly or pop costly by making it O(n).

Lecture 6 - Tree

- A tree is a special type of non-linear data structures (linked data structure)
- useful in databases, Ai, etc

Basic Tree Concepts



→ Root node (most top node)

→ leaf nodes (0 children)

Key Defs:

Nodes - building block of tree
Roooot - top most node
children - directly under a node
leaf node - node with 0 kids

Types of Trees

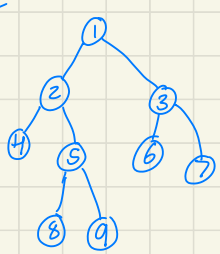
1.) General tree - each node can have any number of children
2.) Binary tree - Each node has at most 2 children
3.) BST (Binary Search tree) - Left child < parent, Right child > parent
4.) Balanced Trees (AVL, Red-Black trees) - self balancing to keep operations effecient

Note: Trees must have a unique
Singular path from root to any node

More Key Terminology

- Edge, connection between two nodes
- path, seqvence of edges
- Height of node, number of edges on longest path down to a leaf
- Height of tree, height of the root node
- Depth - number of edges from node to the root
- Level - number of connections between node and root +1

Ex



root: 1
leafs: 8, 9, 4, 6, 7
6 is level 3
depth of 6 is 2
Height of tree is 3
Note: height is max level - 1
Level of 9 is 4

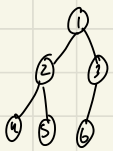
lecture 7 - Binary Trees

Binary Trees

- A binary tree is hiarchical data structure
- Each node has max 2 children: left and right
- Nodes can be linked using pointers in Python

Types of Binary Trees

- Full Binary trees, every node has either 0 or 2 children
- Complete Binary Tree, all level's are completely filled, except maybe last, Should still have left to right rule
  - ↳ if its not full, it should read left to right.

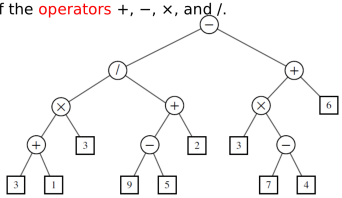


√ valid Complete tree

x invalid incomplete tree

Binary, Arithmetic Expression Tree

An arithmetic expression can be represented by a binary tree whose leaves are associated with variables or constants, and whose internal nodes are associated with one of the operators +, −, ×, and /.



((((3 +1)×3)/((9 −5)+2))−((3×(7−4)) +6))

5

Operations on Binary Trees

- Traversing (visiting all the nodes)
- Searching (Find node)
- Adding (add node)
- removing (delete node, dont break structure)
- Pruning (remove subtree)
- grafting (attach subtree)

4 ways to traverse Binary Tree

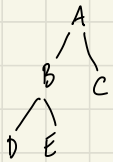
a) Pre-Order Traversal

- Vist root → left → right

Pre-order: A B D E C

b) In order Traversal

- visit left → Root → Right
- Usefull for sorting (BST)



output: D B E A C

Note: Every traversal operations runs in O(n) time.

c) Post order Traversal

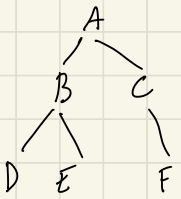
- Visit left → right → root
- used for deleting tree

output: D E B C A

Start bottom left, visit root last

D) Level-order Traversal (BFS)

- visit nodes level by level from left to right, top down
- uses queue instead of recursion



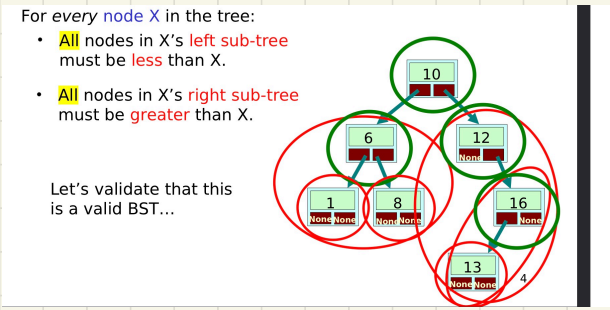
output: A B C D E F

Top-down

Lecture 8 – BST Operations

Binary Search Tree Overview

- Each node has at most two children
- left subtree contains nodes with values smaller than node
- right subtree is greater than its parent
- many BST's excludes duplicates to maintain efficiency.



Operations on a Binary Search Tree

- Search
- insert
- delete
- Traverse

BST Traversal & In order Traversal

- to print elements in sorted order, we use in-order traversal

1.) visit left subtree
2.) Process current node
3.) visit right subtree

Note: would print correctly in alphabetical order

Searching in a BST

- Start at the root
- if v == root.value, found
- if v is less than current nodes value, go left
- if v is greater than current nodes value, go right
- if None object, not found

Best case: (Balanced BST): O(logn)
Worst case: (unbalanced BST): O(n)

- Balanced BST, tree where both sides are approx. the same

Note: to print elements in sorted order, we use in order traversal
left subtree, value, right subtree

Inserting a new value to BST

- Start at root
- if V < node.value → go left
- if V > node.value → go right
- Find empty spot and insert V.

Best case: O(log n) → Because binary search is used to sort
Worse case: O(n)

Deleting Node in BST

- has 3 cases

case 1: Node has no children

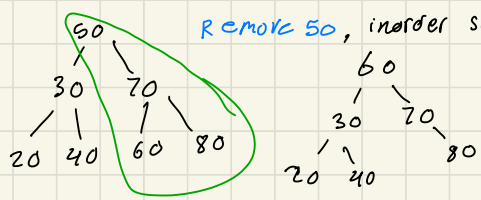
- Just remove node

Case 2: Node has 1 Child

- Replace the node with its child

Case 3: 2 children

- Find inorder successor (smallest node in right tree)
- replace nodes value with succesor and delete



Remove 50, inorder successor = 60

Finding Min & Max BST

- min value: leftmost node
- max value; rightmost node

Time complexity: O(log n), O(n)
↓ most times | worst case

Pruning BST

- keep values within range
- Pruning removes nodes, but depending on the conditions

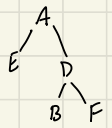
Lecture 9 - BST's Examples

Recursive vs. Iterative

- Recursive, shorter easier to understand but uses extra memory in the call stack
- Iterative, uses while loop instead of recursion, no additional memory usage but more code.

Saving Tree: Pre order

Root → left → right



AEDBF

Also store each nodes value with extra info

(A,2), (E,0), (D,2)

How many children each node has

Expression tree:

infix: $(((3+1)\times 3)/((9-5)+2)) - (((3\times(7-4))+6)$

Saving & Restoring BST's

- Serialization is useful for:
  - Storing BST in file
  - Transmitting across network
  - Reconstructing BST
- Use tree traversal techniques to save the structure.

Lecture 10: AVL Trees (Balanced BST's)

- AVL Tree's are self balancing BST's
- When inserting a value into BST, it may become unbalanced leading to inefficient operations $O(\log n) > O(n)$ (better / worse)
- AVL Tree's ensure trees are balanced

What is an AVL Tree

- AVL Tree is a BST that automatically balances using rotations
- Key properties: every node keeps balance factor of -1, 0 or 1.
- Balance Factor: BF(T) = Height of Right Subtree - Height left subtree
  - BF(T) > 1 → right heavy (rotation needed)
    BF(T) < 1 → left heavy (rotation needed)
    BF(T) $\in \{-1, 0, 1\}$, tree is balanced

Height(x) = max(left tree, right tree) +1 (note: height for empty tree is -1)

AVL Tree Properties

- Maintains height balance: ensure tree's height remains $O(\log n)$
- Balance Factor is always -1, 0, 1
- Self-balancing, after insertions/deletions, the tree performs rotations to restore balance
- Height of an AVL Tree's with n nodes is $O(\log n)$
- Recursive formula for minimum number of nodes: $n(h) = 1 + n(h-1) + n(h-2)$

AVL Tree Rotations

- When the tree becomes unbalanced, rotates restores balance

Types of Rotation

1) Single Rotation (LL or RR)
   - LL - perform right rotation
   - RR - perform left rotation

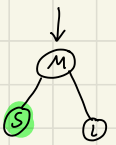
2) Double Rotation (LR or RL)
   LR - perform Left rotation, then right
   RL - perform right rotation, then left

Visualizations for Rotations

Left-Left or Right Right

Happens when:

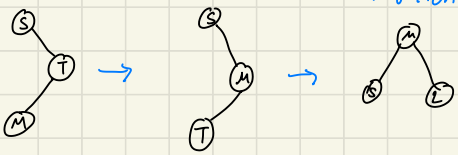
Perform left rotation on Smallest



Left-Right / Right-left rotation

Happens when:

Requires double rotation



Right rotation at T
Left rotation at M

AVL Tree Insertion Algorithm

- Perform BST Insertion
- Update height of ancestor nodes
- Check balance factor for each
- Perform rotations, if needed

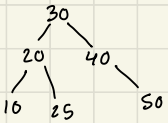
- Search: O(log N)
- Insertion: O(log N)
- Rotation: O(1)

In AVL Trees, you check for balance after every insertion / deletion.

Note: Deleting from a AVL Tree and Standard BST Tree is the same. Broken into 3 cases, but check if tree is still balanced after every deletion.

Practise

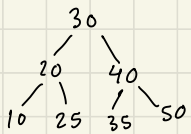
1)



	H,	BF
30 :	2	1 − 1 = 0
20 :	1	1 − 1 = 0
40 :	1	1 − 0 = 1
10 :	0	0 − 0 = 0
25 :	0	0 − 0 = 0
50 :	0	0 − 0 = 0

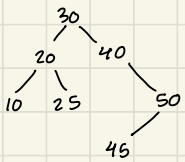
2) Inserting 35 and check balance

Can see its balanced but just to check



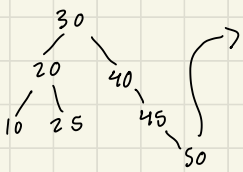
	H	
30	2	0
20	1	0
40	1	0
10	0	0
25	0	0
35	0	0
50	0	0

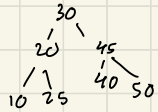
3) Insert 45, cause inbalance



	H	BF
30	3	1
20	1	0
40	2	2
10	0	0
25	0	0
50	1	−1
45	0	0

→ right left case required

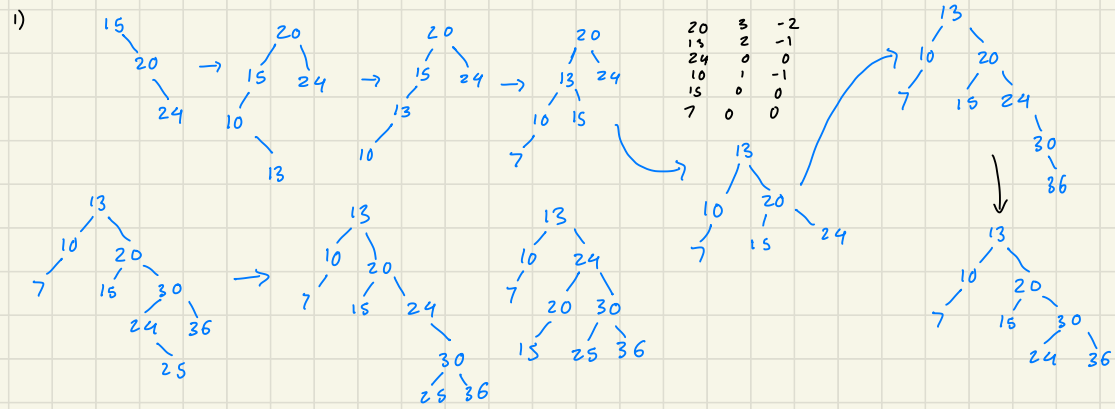




Full AVL Tree Insertion

- Insert 15, 20, 24, 10, 13, 7, 30, 36, 25 into an empty AVL tree

1)



Lecture 12: Priority Queue and Heap

Topic 5:

- Priority queue vs queue?
- Priority queue vs Heap?
- Types of Heap
- Max Heap Implementation

ADT Priority Queue

- Queue follows FIFO Principal. USefull in fair scenarios.
- Some cases require exceptions to FIFO
- Priority Queue is a special type of queue where each item has a priority rating.
- Item with highest priority gets taken out first.

Priority Queue Operations

1) Insert new item to queue
2) Get value of highest priority item
3) Remove the highest-priority item from queue

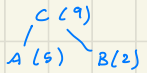
Note: If two elements have same priority, they are processed in their order of arrival (FIFO)

Implementation

- Can group by Priority, High-Medium-Low
- If we have huge number of priorities, we use Heap Data structure (more efficient way to manage)
- Heap allows fast access to highest priority item, using binary tree.

Ex

A = priority 5
B = priority 2
C = priority 9



- Removing item from Heap removes highest priority item first.

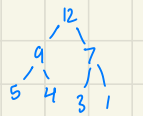
Heaps and Complete Binary Trees

- Heap is special type of complete binary tree
- all levels except the last one are completely filled
- Last level filled left to right

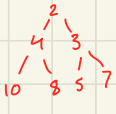
Types of Heaps: Miniheap vs Maxheap

1) Maxheap - largest value is always at root of tree
2) Miniheap - smallest value is always at the root of tree

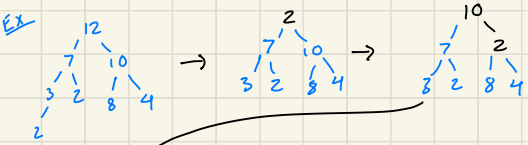
Maxheap



Miniheap



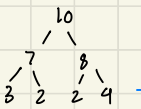
Ex



Extracting Maximum (Heapify Down)

- remove largest element:

1) Replace root with last node in heap
2) Remove last node
3) sift down the new root (swap with largest child)



- Heapify Down - used to remove max value
- Note: In max_heapify(a, heapSize, i): Left side = 2*i, Right side = 2*i+1
  (Left side: index; Right side: index)

Code version

Challenges in Using an Array for Heaps

- Finding bottom most right node
- Finding bottom most left most open spot
- Locating nodes parents for swaps
- Use array, not linked list

Extract Item from Heap

1. If the len(heap) == 0 (it's an empty tree), return error.
2. Otherwise, heap[0] holds the biggest value. Remember it for later.
3. If the len(heap) == 1 (that was the only node) then delete the only value and return the saved value.
4. Copy the value from the right-most, bottom-most node to the root node: heap[0] = heap[-1]
5. Delete the right-most node in the bottom-most row: del heap[-1]
6. Repeatedly swap the just-moved value with the larger of its two children:
   Starting with i=0, compare and swap:
   heap[i] with heap[2*i+1] and heap[2*i+2]
7. Return the saved value to the user.

heap

0	12
1	7
2	10
3	3
4	2
5	8
6	4
7	2
8	1
9	
10	
11	
12	
13	
...	

Storing Heap in Array

- using level order traversal
- root node can be 0/1
- left index = 2i // 2i+1
- right index = 2i+1 // 2i+2
- parent of node is index (i-1)//2

Heapify UP

- Inserting into heap
- Step 1: Add new elements at available index
- Step 2: Compare with its parent then swap (if greater)
- Step 3: Repeat until heap is sorted

Code version

Adding a Node to a Maxheap

1. Insert a new node in the bottom-most, left-most open slot:
   heap.append(value)
2. Compare the new value heap[-1] with its parent's value: heap[(len(heap)-1 - 1)//2]
3. If the new value is greater than its parent's value, then swap them.
   (we don't need to care other nodes, why?)
4. Repeat steps 2-3 until the new value rises to its proper place or we reach the top of the array.

0	10
1	7
2	8
3	3
4	
...	

26

Time Complexity of Heap Operations

- Insertion (Heapify up) - $O(\log n)$
- Extraction (Heapify down) - $O(\log n)$
- Reason: Heap is binary tree, height of balanced tree is $\log_2(n)$

Searching in Heap

- BST allows efficient searching because of total ordering
- Heaps dont have full ordering, just ensures parent is greater/smaller than children
- Searching in heap takes $O(n)$ time.

Heapsort

- Insert all N numbers into a maxheap (heapify up process)
- while elements remain:
  - remove max value (root)
  - place in last open slot in array
- Naive heapsort can take $2x\ n\log n = O(n\log n)$
- Optimized way:
  1.) Build heap in $O(n)$ using heapify down process
  2.) Extract elements normally ($O(n\log n)$)

  Reduces to $n\log n$ but better constant factor

Max-Heap

- type of binary heap
- How to convert randomly arranged input into max-heap
  1.) start from last non leaf node (bottom most, right most)
  2.) visualize array as tree
  3.) Convert each subtree to maxheap
  4.) move upwards to root
  5.) Keep shifting values down

To convert an unsorted array into a Max heap, we use Heapify.

last non-leaf node - index $\lfloor n/2 - 1 \rfloor$

- locating child at index $(N/2 - 1)$, allows us to skip 50% of subtrees

```
for curNode = (N/2 - 1) to 0:
    heapify(curNode)
```

Building maxheap: $O(n)$
Heap extraction(sorting): $O(\log n)$

Sorting → $O(n \log n)$

Lecture 13 - Hash Tables

- Hash table is a data structure that implements a MAP (associative array) ADT.
- Allows mapping keys to values and gives fast lookups / insertions.
  - Search average = O(1), worst = O(n)
- Dictionary: generic way to map keys to values
- Hash table: implementation of a dictionary using hash function.

Hash Tables

- Stores key value pairs
- Uses hash function to compute index (hash) for given key
- average time of O(1) for insertions, deletions, and lookups.

Key Value Database

- Key value database is a type of NoSQL database, stores data as collection of key value pairs
- Each entry has a key and a value pair
- flexible, efficient

MAP ADT

- data structure stores key - value pairs
- get(key K)
- put (key K, value v)
- remove (key K)
- size()
- isEmpty()

Hash Table Advantage

- Searching sorted list or BST takes O(log n) time, Hash tables are near instant retrieval

Problem with Direct Adressing

- waste of memory

- Hash function maps a large key space to smaller set of slots

Large space transforms → smaller space
values distributed to fewer slots

```
def hashFunc(idNum):
    array_size = 100 000
    bucket = idNum % array_size
    return bucket
```

modulus operator ensures remainder

Collision - occurs when two different values map to the same bucket.

↓ To handle collision:

1) Chaining (linked list at each slot) (open hash table)
   each bucket holds a list of values [a1, a2, a3]

2) Linear probing, if 5224 is taken check 5225 (closed hash table)
   - avoids wasting space but causes clustering

3) Double hashing, use another hash function for new slot

Closed Hashing vs Open Hashing

Closed:

- Stores value in array
- if slot is full find next open, Linear Probe

1) compute hash index
2) if slot is empty, store value
3) if slot is taken, next

open:

- Each slot holds linked list
- most flexible but require memory for pointers

Issues with Linear Probing

- 1) primary clustering (if many values are same area)
- 2) wrap around (reach end, continue searching from idx 0)
- 3) wasted slots

comeback Slide 21/33

Lecture 14 - Hash Tables

Primary Clustering in Linear Probing

- When many values hash to similiar locations, form clusters
- new keys must probe through these clusters (making search and insertion slower)
- Keys cluster, causing long runs of probing

Avoiding Primary Clustering

- Spread out probes so that they don't land in straight line. (2 better alternatives)

Quadratic Probing

- Instead of moving to next slot (i+1) like linear probing, quadratic probing jumps farther away by using a quadratic function
  - Formula: $h(K,i) = (h'(K) + c_1 i + c_2 i^2) \mod m$ → Note when $c_1 = 0, c_2 = 1$, $h(K,i) = (h'(K) + i^2)$
    - $h'(K)$ is initial hash function
    - i is probe number
    - c_1, c_2 constants
    - → If slot is filled, it goes $1^2, 2^2, 3^2, 4^2 \ldots$
  - Quadratic probing spreads (1, 4, 9, 16...)

Double Hashing

- Uses second hash function to decide how far to jump
- Formula: $h(K,i) = (h_1(K) + i h_2(K)) \mod m$
  - $h_1(K)$ - first hash function
  - $h_2(K)$ - second hash function (each key, unique probe sequence)
- Best alternative

Quadratic Probing Problem (Secondary Clustering)

- Quadratic Probing reduces clustering, but not eliminate fully
- If two different keys have same initial hash index, they follow exact same probe sequence, which is why double hash is better (to get diff index)
- Same initial hash still cluster, less severe than Linear Probing

Fixing Secondary Clustering

- Solution: Double Hashing, instead of following probe sequence, second hash function
  $h(K,i) = (h_1(K) + i * h_2(K)) \bmod m$
  - Each key gets unique probing sequence, avoids secondary clustering
  - More randomized placement → faster search / insert
  - Secondary clustering is better than primary clustering but not ideal

EX $h(K,i) = (h'(K) + i^2) \bmod m$

c1 = 1, c2 = 1

m = 12

$h(K) = 0$ Probing sequence: 0, 2, 6, 0, 2, 6, 0, 2, 6, ...

Problem:
- only contains even addresses
- miss many availible empty slots
- Probing pattern is restricted by i^2.

Double Hash - How it works

1) Compute first hash: $h_1(K) = K \bmod m$ → Starting position
2) Compute second hash: $h_2(K) = 1 + (K \bmod (m-1))$ → Step size
3) Each probe step is determined by key itself

- Ensures no primary / secondary clustering

- Why $h_2(K)$ and m must be relatively prime.

Ex: if $m=10$, $h_2(K)=2$, we only probe even indices

if m is a power of 2, ensure $h_2(K)$ is always odd

- if x is odd, return 2, else $x+1$

Hash Table Efficiency

- Efficiency depends on:
  1) Type of hash table
  2) Load Factor (how full table is)
  3) Number of collisions (more collision, slower)

Search/Insert: O(1) - Best
O(1) - Average
O(n) - worst (everything collides)

Lecture 15 - Hash Table

Open Hash Table Efficiency

- worst case: all elements end up in same bucket
  - Hash table turns into a linked list, $O(n)$ - search / insert

Simple Uniform Hashing

- Every key K has an equal probability of being assigned to any of the m buckets

$$Pr(h(K) = j) = \frac{1}{m}$$

Each bucket has roughly equal share of keys

Good Hash Function

- hash function is good, when keys are uniformly spread across m buckets
- Number of elements per bucket: $\alpha = \frac{N}{M}$ (N: # elem in table, M: # of buckets, α: Load Factor) (Lower α is better)

Unsuccessful Search Hash Tables

- Search entire linked list at hash index
- Expected number of checks = length of list
- $\alpha = N/M$
- $\Theta(1 + \alpha)$ - Expected time for unsuccesfull

Successful Search Hash Table

- Find correct bucket, scan linked list
- check about half items before finding target
- $\Theta(1 + \frac{\alpha}{2}) = \Theta(1 + \alpha)$

Empty Hash table

- max steps to insert value = 1
- Finding item is $O(1)$

Full Hash Table

- Probe many slots before finding empty one
- worst case: $O(n)$ time complexity!

Load Factor (α):

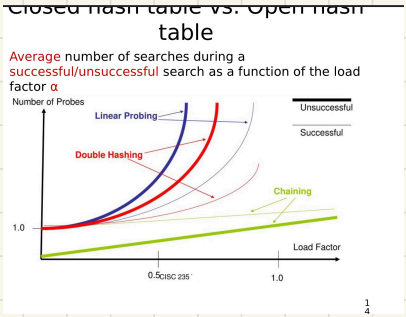
- measure of how full table is

$$\alpha = \frac{50\,000}{100\,000} \} \frac{\text{elems}}{\text{buckets}}$$

- $\alpha > 1$ means elems > buckets only in Open hash table
- (high α) = Slower search
- (low α) = Faster search

Closed H.T Efficiency

- Prob of occupied bucket is α
- Prob of empty bucket is $1-\alpha$
- Expected Steps Formula: $\frac{1}{1-\alpha}$
- Successful search time: $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$
- insertion complexity: $i+1$ at $\alpha = \frac{i}{m}$



LP - fast when empty, degrades quickly
- Clustering increases search time

DH - better than LP
- less clustering

SC - consistently good performance at higher
- scales better because own linked list
- unsuccessful takes longer

Rehashing - increasing table size, recalculating hash values
. the positions of key will change - O(1) time

Lecture 16 - Hash Function Analysis

- Different Hashing Methods

K mod m (division method)

- take key, compute mod m
- usually used in linear probing, works better when K and m are relatively prime to avoid clustering
- High chance of collision when keys are similar

Sum of Digits Hashing

- Convert key into digits
- Sum digits, compute sum mod m
- High chance of collision (523 and 802) both sum to 10
- works well for small numeric keys

K^2 mod m (Mid Square method)

- more balanced hashing method, avoids patterns
- Square K
- Extract middle digits of squared number
- compute mod m to get hash idx

$h(k)$ = middle digits of (K^2) mod m

Ex: $56^2 = 3136$ → middle digits = 13

13 mod 100 = 13

- Avoids Clustering, more computation

For parent: $(index - 1 \;//\; 2)$

Hashing Strings

- Strings can't be modded, so we use polynomial hashing

1) convert each character into ASCII value
2) multiply each ASCII value by power of prime number
3) Sum all values take mod m

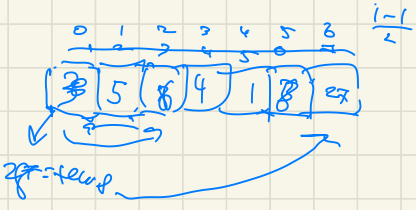
"abc", p=31, $m = 10^9 + 7$

97, 98 x 31

98 x 961 = 95139

97 + 3038 + 95139 = 98274

Final Hash = 98274



Lecture 17 - Graph - Basics

- Properties
- graphs in Python
- graph ADT

General concepts

- graphs have vertices and edges
- Vertices are adjacent if there an edge between them
- Neighbors: Vertices that are adjacent
- Degree is number of edges connected to vertex
- In directed graph: In degree: Edges coming into a vertex
  Out degree: Edges goin out from vertex

More Terminology

- walk: Sequence of vertices where each adjacent pair is connected
- Trail: walk with no repeated edges
- Path: walk with no repeated vertices
- Circuit: closed trail
- Cycle: closed path
- Loop: Edge that starts and ends at same loop

Represention

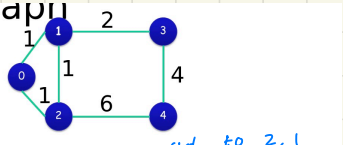
- Adjacency matrix (1, if two cells i,j are edge)
- Adjacency list: Each vertex has a list of neighbors
- Reordering vertices affects the matrix/list but not actual structure

Weighted Graph

- weighted graph has edges with values
- Adjacency matrix contains numeric weights, not just 0's and 1's.
- Adjacency list: Each connection is labelled with weight like (1, 2.1)

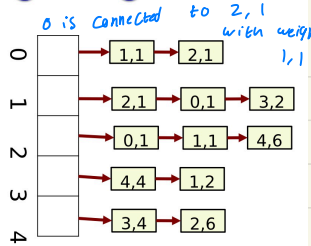
Graph

- Adjacency Matrix
- Adjacency List



	0	1	2	3	4
0	0	1	1	0	0
1	1	0	1	2	0
2	1	1	0	0	6
3	0	2	0	0	4
4	0	0	6	4	0

0 is connected to 2, 1 with weights 1, 1 *



Directed Graph

- directed graph has edges that point in only one direction
- Adjacency matrix only records one direction
- Adjacency list: only outgoing edges listed

Adjacency Matrix vs Adjacency List

- Adjacency Matrix:
  - few vertices, many edges
  - Dense graphs
- Adjacency List:
  - many vertices, few edges
  - Sparse graphs

- Dense graph: many edges between nodes
- Sparse graph: few edges compared to total possible

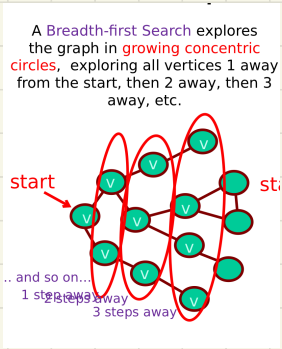
Graph as ADT

- supports a set of operations regardless of how it's implemented
- Common operations:
  - Test (if graph empty)
  - Get (# of edges/vertices)
  - check (if edge exists)
  - insert (vertex, edge between 2 vertices)
  - remove (vertex, edge ....)

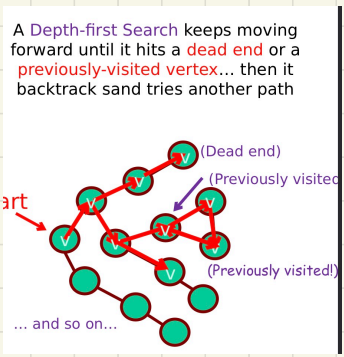
Graph Traversal

- graphs can be traversed in many ways because of cycles and multiple paths
- Key Methods:
  - ↳ Breadth First Search (BFS) level by level
  - ↳ Depth First Search (DFS) as deep as possible, before back tracking

Lecture 18 - Graph BFS/DFS



- Breadth First Search (BFS)
  - ↳ Explores nodes in layers (growing concentric circles)
  - Visiting all nodes 1 step away from start, then 2 steps away
    - Breadth means broad/wide
    - Keeping track of indices in Queue
    - Time Complexity of $O(|V| + |E|)$
    - Shortest path of unweighted graphs



- Depth - first search (DFS)
  - goes as far as possible along one path before backtracking.
  - explores deeply, then tries other paths.
  - Uses Stack to hold visited vertices
  - Pop nodes when visited
    - Time complexity of $O(|V| + |E|)$

- For BFS, you add it to queue, then mark it as seen. Then you checkout the neighbors.
- For DFS, you make sure each subset of nodes are visited before moving on to the sibling.

DFS Code Idea

- Takes in node as input, marks as visited
- For every neighbor, recursively call function
- goes untill all nodes are marked

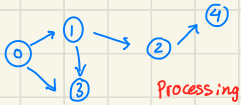
BFS Code Idea

- Start node as input
- mark node as visited
- create queue enqueue node
- while the queue is not empty, enqueue and dequeue all nodes that are seen.

BFS Traversal Demo

- Start from vertex, make queue
- add starting vertex mark it, while nodes left
- take out node, loop through neighbors, if it hasn't been seen add it to queue and mark it

Example



Processing order:

0 → 1 → 3 → 2 → 4

* Always process first node in queue

BFS Runtime Analysis

- n; number of vertices
- m; number of edges

- Each node is enqued and deqed once → O(n)
- Each edge is seen once O(m)
- Time complexity: O(m+n)
  - ↳ If you use adjacency matrix its $O(n^2)$

Adjacency list > Adjacency Matrix

- list stores actual neighbors
- list is much more effecient

DFS Traversal Demo

- goes deep first, then hits deadend then goes to next path

- Can be done using recursion or stack

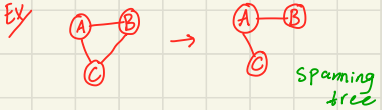
1) Push node on stack, while stack not empty
   Pop top node, if not visited mark as visited, neighbors on stack

- As long as you follow property of DFS, can have diff order outputs
- Runs in O(n+m) time

Disconnected Graphs

- If graphs aren't fully connected, BFS/DFS starting at a node won't visit everything
  - → would need to loop through all nodes

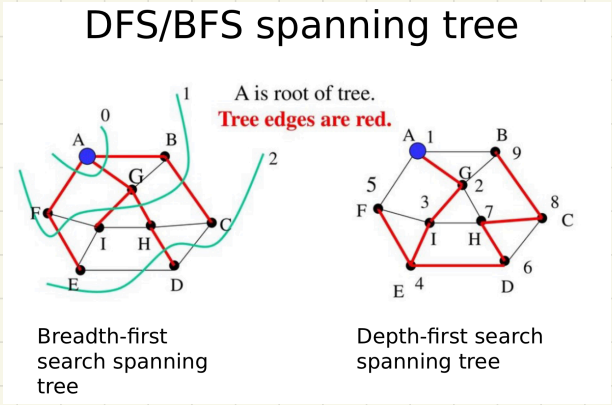
Lecture 19 - Min Spanning Trees

Ex 

What is Minimum Spanning Tree

- Tree: Connected graph with no cycles
- Spanning Tree: Subgraph that includes all vertices and is a tree
- MST (Minimum Spanning tree): Spanning tree with lowest possible total edge weight
- Disconnected Graph can not have spanning tree
- Every connected graph has atleast one Spanning tree

- Undirected : edges have no direction
- directed : edges have a one way direction



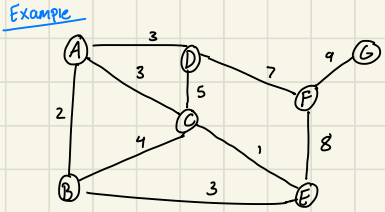
- Min Spanning Tree is not unique
- Min Spanning Tree is undirected connected weighted graph

Kruskals Algorithm

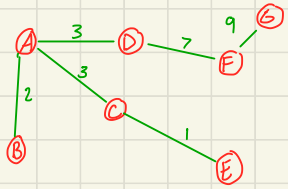
- Used to build the MST edge by edge, starting from smallest

1) Sort all edges by weight
2) Start with an empty tree
3) Loop through edges (lowest → highest)
4) If edge doesn't form cycle, add it
   If it would create cycle, skip it
5) Stop when added n-1 edges

Example



CE = 1
AB = 2
AD = 3
AC = 3
BE = 3
BC = 4
DC = 6
DF = 7
EF = 8
FG = 9



Time complexity: $O(E \log E)$

More Notes on Kruskal's Algorithm

- In code the idea is while the number of vertices is less than n-1
- Pick smallest edge, that doesn't make a cycle
- add it

- Kruskals builds a forest, if two nodes are alr connected dont add more
- Kruskals Algorithm is super efficient because of Union and Find O(1) optimization

Optimizing Kruskals Algorithm

- Basic: brute force used to update group labels $O(n^2)$
- optimized: uses sets and Tree's to track group.
- roots as set label, every node has pointer to its parent, root points to itself
  - ↳ just updates pointers, $O(m \log n)$

Find-set(u): follows parent pointers to find root
Union(u, v); Connects two groups, make one root point to other

Lecture 20 - Graphs - Prims Algorithm

- another way to build a MST
- Unlike Kruskals. Prim's builds from a single vertex, and expands tree one edge at a time, always adding the cheapest edge that connects to tree.

1) Start at any vertex
2) Choose cheapest edge
3) Look at all edges combining them, take cheapest

Basically, go around choosing cheapest connection, picking up all the vertices.

Visual Difference: Kruskal vs Prism

Kruskal

- Starts w edges
- add lowest weight
- union find
- Edge based

Prim

- Starts w vertex
- lowest weight that touches tree
- set of visited nodes
- Vertex based

Problem with Basic Prim

- Scan all edges every time, to find next min valid one
- $O(m \cdot n)$ time $O(m \log n)$ to sort them

↳ Solution: Use a min Heap

- Store candidate edges in min heap, ordered by weight
- Pop smallest edge, check connection to vertex inside tree to outside

Basically: Make a few different sets:

T = already covered nodes
R = Not yet visited
M = Min heap nodes

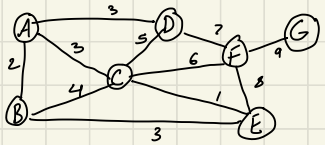
Code Idea:

- For all neighbors in V
  add it to min heap

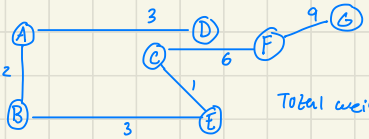
While set has n-1 vertices
Pop from min heap and if
x in tree and y not in tree
then add edge to MST, Y, to tree

Ex 1

Using Prisms Alg to get MST



1) Select A
2) Choose AB
3) Keep Choosing Cheapest Connector



Total weight: 24

Visited: {A, B, E, C, D, F, G}

MST is unique when edge weighting is unique.

3rd Solution to Prims

- Using a Priority array
- Always pick vertex from a with smallest K-value
- manually tracking smallest edge K[v]
- update K[v] when cheaper edge is found

Time Complexity of Prim's using min heap (priority queue)

↳ Total time: $O(m \log n)$
- n ↳ # of vertices
- m ↳ # of edges

- Cheapest edge gets globally optimal MST

- Kruskal's better for sparse graphs
- Prim's better for dense graphs

Lecture 21 - Shortest Paths Dijkstra's Algorithm

4 Types of Shortest Path

- Single pair shortest path (smallest weight)
  - ↳ Point A to B
- Single source: same start to different finishes
- Single destination: Different start same destination
- All pairs: Shortest path between every pair of nodes

Dijkstra's Algorithm

- Solves single source shortest path and shortest path to every other vertex
- takes in 2 inputs, graph G with edge weights and a vertex starting point S.
- Output: an array Dist[ ], showing shortest distance from S to every other node
- Splits vertices into 2: Settled and unsettled
  - ↳ confirmed shortest
  - ↳ still exploring
- similar to Prims

1) From A, update neighbors
2) Pick node w smallest distance, thus far → check its neighbors, update
3) pick next smallest unsettled, repeat

3 Data Structures used by Dijkstra

- Dist[ ] - Stores current best known distances from S
- Done[ ] - keeps track of settled nodes
- Parent[ ] - for path reconstruction
- To find path, use parents in reverse order

Relaxing: testing whether the shortest path can be improved by going through U.

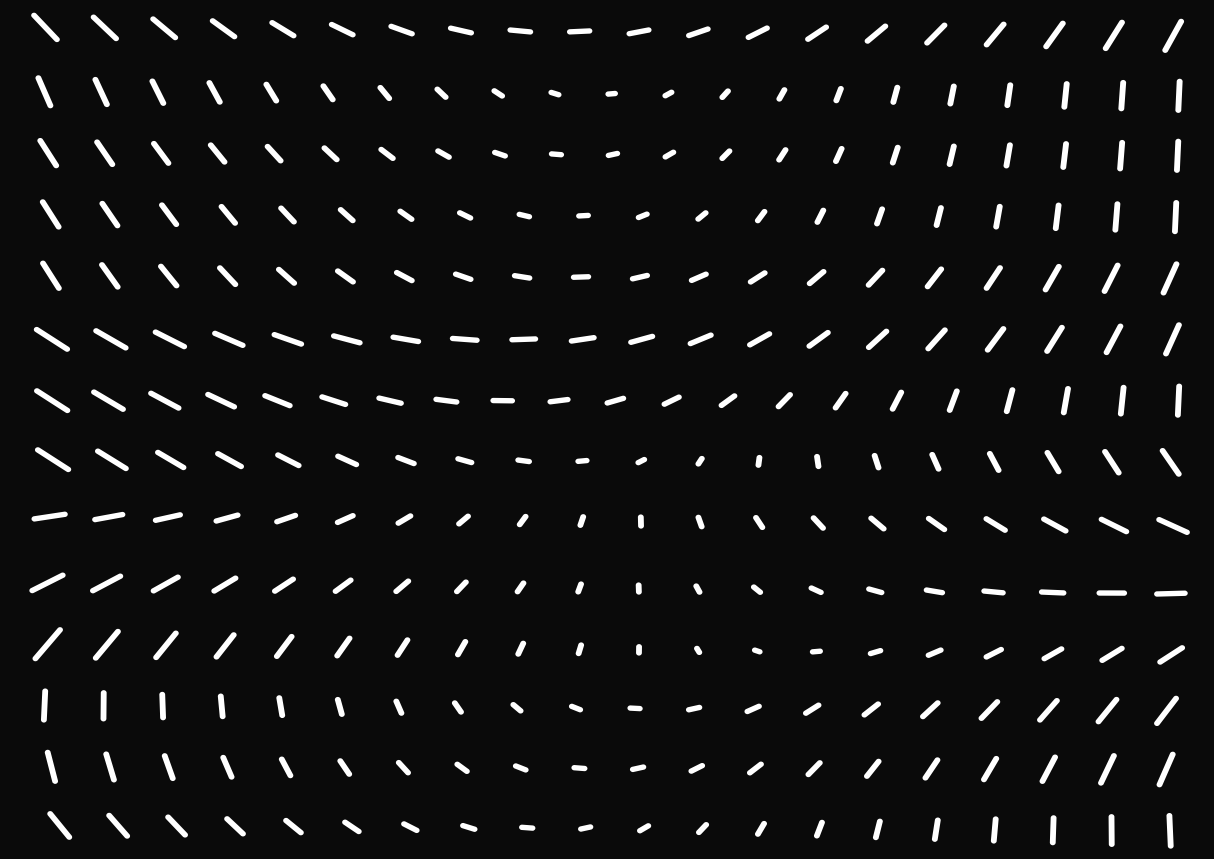
If Dist[v] > Dist[u] + weight(u→v)

Update Dist[v] = Dist[u] + weight(u→v)

Parent[v] = u

runs in O(E log V)
↓
using heaps (log n)

Quiz 1 Review



Practise Quiz 1

1) True
2) True
3) False
4) False
5) 4)
6) A)
7) D)
8) A)

15, 20, 24, 10, 13, 7, 30, 36, 25

```
int mystery(int x, int y) {
    if (y == 0)        } Base case
        return 1;      }
    if (y == 1)        } Base case
        return x;      }
    return x * mystery(x, y-1);
}
```

Base cases implies, $y==0$, return 1, $x^0 = 1$

$y==1$, return x, $x^1 = x$

input (2, 5) $\rightarrow 2^5 = 32$

2 * mystery (2,4) $2^4 = 16$

2 * mystery (2,3) $2^3 = 8$

2 * mystery (2,2) $\rightarrow 2^2 = 4$

2 * mystery (2,1) $\rightarrow 2$

2

Be careful unwinding the recursive function.

another MySt(3,4)

Base case: (3, 1) $\rightarrow$ 3

(3, 2) $\rightarrow$ 3 + 3

(3, 3) $\rightarrow$ 3 + 3 + 3

(3, 4) $\rightarrow$ 3 + 3 + 3 + 3 = 12

Base case: 1 *

Factorial(1): 1 * factorial(2)

factorial(2): 2 * factorial(3)

: 3

4

5

5! = 120

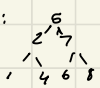
4a) Coding Question

Function takes in an array, level order.
Returns the BST.

Given:

Level order: 5, 2, 7, 1, 4, 6, 8

output:



Idea: Start at root, move down levels

Creating a BST Node

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

Function to get new node

```
def getNode(data):
    newNode = Node(data)
    newNode.data = data
    newNode.left = None
    newNode.right = None
    return newNode
```

Level order structure

```
def Level_order(root, data):
    if (root == None):
        root = getNode(data)
    return root
```

Base case, if tree is empty create root node

Inserting flow

```
if (data <= root.data):
    root.left = Level order(root.left, data)
else:
    root.right = Level order(root.right, data)
return root
```

recursive call

```
def constructBst(arr, n)
    if (n == 0):
        return None
    root = None
    for i in range(0, n)
        root = Levelorder(root, arr[i])
    return root
```

making BST

makes all the roots

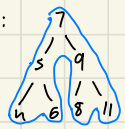
4a) coding

```
def inorderTraversal(root):
    if (root==None):
        return None

    inorderTraversal(root.left)
    inorder traversal(root.right)
```

Practise coding q2

Idea: takes in pre-order traversal and prints tree

Ex:



7, 5, 4, 6, 9, 8, 11

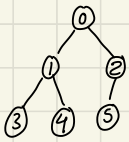
root → left → right

```
def ConstructBSTpreorder(arr, index, min_val, max_val)
    if index[0] >= len(arr)          } stop if index > array
        return None, index

    root_val = arr[index[0]]         -> set root node

    if root_val < min_val or root_val > max_val   -> if the root node is valid
        return None, index

    root = BST_Node(root_val)        - create it
    index[0] += 1

    root.left, index = ConstructBSTpreorder(arr, index, min_val, root_val)   -> recursively call left side
    root.right, index = ConstructBSTpreorder(arr, index, max_val, root_val)  -> recursively call right side

    return root, index
```

- Uses index to track position in array.

Practise 2)

Idea: Function takes pre-order and in order



inorder[ ] = [3, 1, 4, 0, 5, 2] → Left → root → right

Preorder[ ] = [0, 1, 3, 4, 2, 5] → Root → left → right

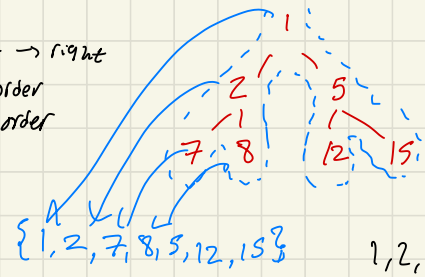
```
if left > right:
    return none

root.left = Function(inorder, preorder, preindex, left)
```

Another Sample quiz

1.) True
2.) False
3.) True
4.) True
5.) D
6.) a) *
7.) b)
8.) b) Root → left → right
Stacks → pre order
queues → level order

1) False -1, from getIndex() means entry was not present
2) False
3) True a)
4) True a)

Head → 2 → 3 → 4



1, 2, 5, 7, 8, 12, 15

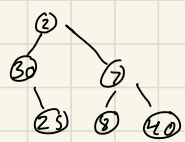
((a+b) * (c-d)) / (e-f) * (g+h))

ab+ cd- ef- gh+

((ab+cd-)*)((ef-gh+)*)/

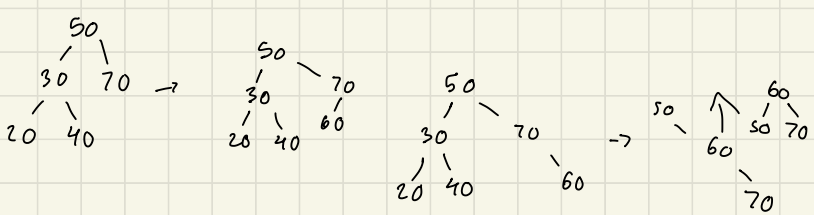
---

in_order = [30, 25, 2, 8, 7, 40]
pre_order = [2, 30, 25, 7, 8, 40]




```
def Construct(in_order, pre_order):
    pre_index = [0]
    left = 0
    right = len(in_order) - 1
    root = Construct(in_order, pre_order, pre_index, left, right)
    return root
```

50, 30, 70, 20, 40, 60, 80



Review for quiz 1: Lecture slides

Data Structures

- way to store, organize, manipulate data effeciently
- Search → Find if element exists
- Insert → add new element efficiently
- Linear Search → O(n)
- Binary Search → O(log N) → reauires a sorted list
- Algorithms, steps to process data efficiently
- Data repelesentation methods: Arrays, Linked lists, trees, graphs

Algorithm Analysis

- Time complexity: Measures the time taken based on input size
- Space Complexity: Measures the amount of memory used

Ex
- Merge sort (nlogn) is fast but needs extra memory
- Quick sort (n log n) but works in place memory
- Key Point: Algorithms may be faster for certain small/large inputs like $A = 37n$, $B = 2n^2$

Experimental vs Theoretical Analysis

- Experimental: run program, affected by hardware, compiler, dont use for big inputs
- Theoretical: uses big-o notation to predict efficiency, use for big inputs
- Abstraction: ignoring unnecessary details: $4n^2 + 3n + 10 \rightarrow O(n^2)$

Big O, Big Ω, Big Θ

- Big O → the upper bound (worst case)
- Big Ω → the lower limit (best case)
- Big Θ → the exact time in both cases (tight bound) → If best/worst case are diff Big Θ does not exist.

Asymptotic Behavior of Polynomials

Polynomial: $P(n) = 3n^2 + 2n^2 + 5n + 7$

Dominant term is $3n^2$

a) If $k \geq d$, then $p(n) = O(n^k)$

Saying, if constant k is the highest degree, that is your tight bound on $O(n^k)$

$P(n) = n^3$, $O(n^3)$ is tight bound, $O(n^4)$ is loose bounds

b) Omega Notation

- if $k \leq d$, then $P(n) = \Omega n^k$

$P(n) = n^3$, grows atleast at n^3, tightest bound is $\Omega(n^3)$

c) Theta notation $\Theta(n^k)$

$k = d$ then $P(n) = \Theta(n^k)$

Exact bounds grows like n^3.

Finding number of Operations

```
def Operation(x):
    n = len(x)          -> O(1)
    A = []              -> O(1)
    for i in range(n):  -> O(n)
        a = 0           -> O(n)
        for j in range(i+1): -> O(n^2)
            a += x[j]   -> O(n^2)
        A.append(a/(i+1)) -> O(n)
    return A            -> O(1)
```

∴ Big O is n^2
Big Ω is n^2
Big Θ is n^2

Array-Based List vs Linked List

Array based list (Dynamic Array)

- Uses fixd size array, resizes dynamically
- Stored next to each other in memory
- Fast access O(1)
- Slow insert/delete O(n)
- resizing requires copying to new array O(n)

LinkedList:

- Each node has data and a pointer to the next node.
- Fast insert/delete (at head/tail)
- Flexible size
- Slow access to nodes O(n)

Array-based List vs. LinkedList

Operation	Array-based List	LinkedList
Insert at head	O(n)	O(1)
Insert at end	O(1)	O(1) if has tail reference, O(n) if not
Insert particular	O(n)	O(n)
Delete at head	O(n)	O(1)
Delete at end	O(1)	O(n)
Delete at middle	O(n) (access + shift)	O(n) (search + delete)
Search	O(n)/O(logn)	O(n)
What is the element at a given position?	O(1)	O(n)

Stacks and Queues

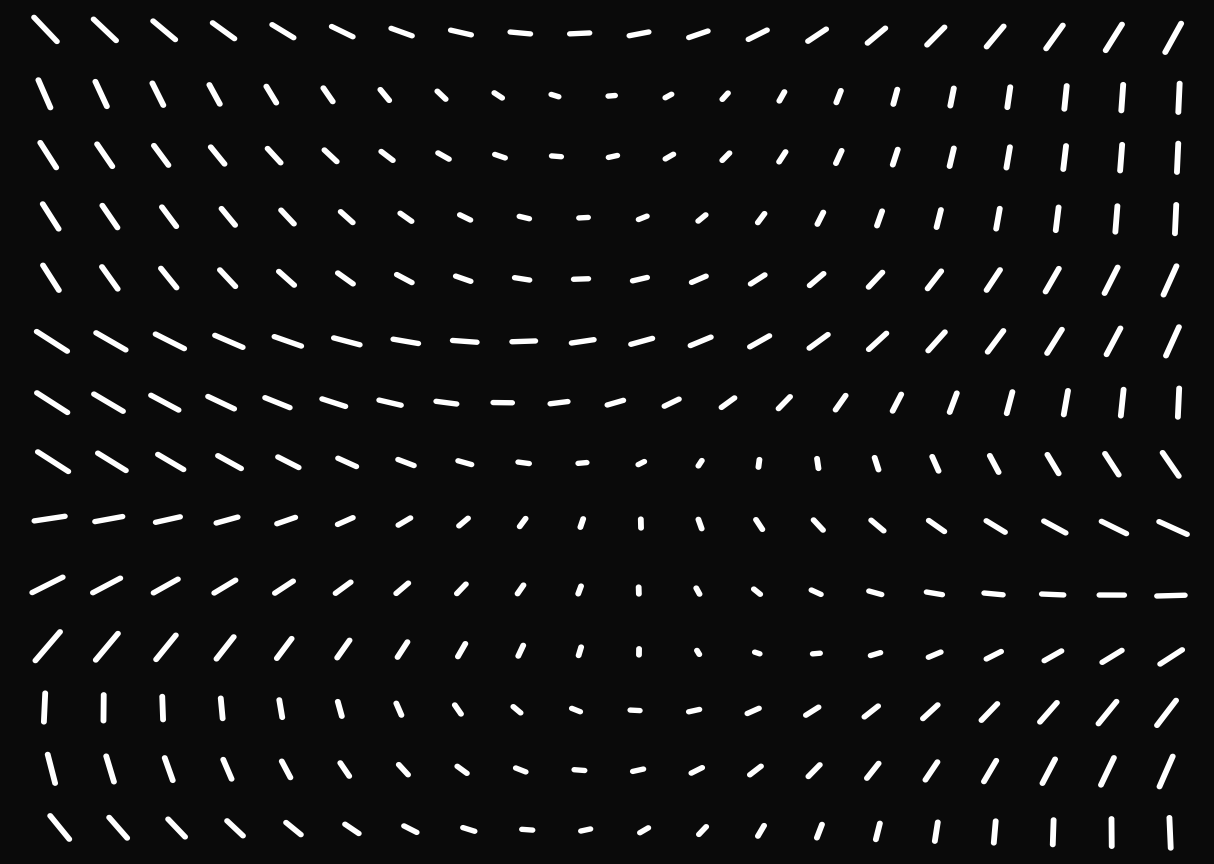
Stacks

- Last in, first out
- Operations: Push, Pop, Peek
- Used for back tracking,
- function calls (recursion), undo operations
- implemented with arrays / linked lists

Queue

- First in, first out
- Operations: enqueue (add), dequeue (remove)

Quiz 2 Review



Quiz 2 Review - Lecture Slides

- Priority Queues
- Heaps
- Hash Tables
- Hash Functions

Quiz 2 practise quiz #1

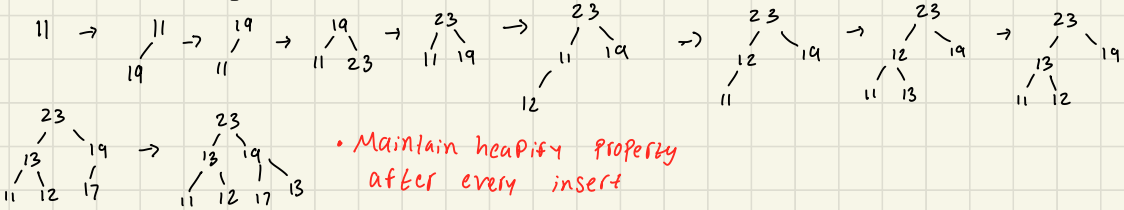
True / False

- min/max heap, Parent node index: Parent index = $\left(\frac{\text{Child index} - 1}{2}\right)$
- Converting array to heap is faster than Searching / inserting array into empty heap - True
  - inserting one by one to empty heap take O(nlogn)
  - Heapifying array all at once (using Heapify approach) O(n)
  - Load factor can be larger than 1 for Chain based collision handling
  - Open addressing must be ≤ 1
  - Large hash table, reduces collisions but memory is inefficient.
  - Ideal Hash Tables: Start with small size dynamically resize, when $\alpha > x$. Resize dynamically

Short Answers

Constructing max heap, into initially empty heap: (one by one)

[11, 19, 23, 12, 13, 17, 13]

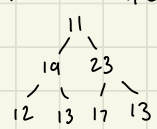


- Maintain heapify property after every insert

other method: Heapify method (Bottom UP)

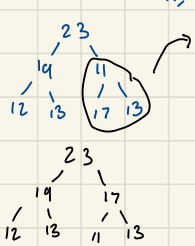
- Given an unsorted array: [11, 19, 23, 12, 13, 17, 13]
- Start at last non leaf node, work bottom up

1) Treat as complete BST:



Check index = (6//2 - 1) = 2
index 2 is 23, which is good

2) i - 1 = 1, value 19, all good, decrement again
i - 1 = 0, value 11, must swap Higher #



Code Implementation:

heapify: (arr, n, i)

largest = i
left = 2 i + 1
right 2 i + 2

if less child exists and greater than root
largest = left

if right child exist and is greater than current largest:
largest = right

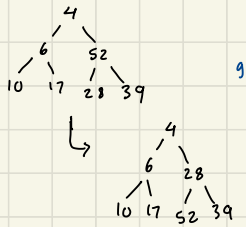
if largest is not root, swap and heapifying

arr[i], arr[largest] = arr[largest], arr[i]

heapify(arr, n, largest)

2) Describing steps to heapify: [4, 6, 52, 10, 17, 28, 39] into min heap

1) create complete BST



Start at index 7
index = (7//2 - 1)
= 2
go to 52
idx = 1, is 6
6 is fine
This is min heap

1) Deal with 52 subtree
2) Then 6 subtree, then 4 subtree, its valid
3) min heap done

3) Insert keys to closed Hash tables
- linear probing
- Array: [7, 8, 30, 11, 18, 9, 14]
- hash function h(k) = k * 3 mod 7
- $\alpha = 0.7$, 7 elems, size = 10

1) Insert 7 keys

H(k)	0	1	2	3	4	5	6	7	8	9
k	7	14		8		11	30	18	9	

2) Check how many collisions each look or
- number of probes to find each key

7	8	30	11	18	14	14
1	1	1	1	3	3	2

$= \frac{12}{7}$

Avg Probes = $\frac{12}{7}$

4) Size 10 hashtable
- double hashing

$h_1(k) = k \bmod 10$

$h_2(k) = 1 + k \bmod 9$

Values: 9, 5, 18, 14, 28, 30

1) use primary hash function to get initial positions in the table:

$h_1(k) = k \bmod 10$

	0	1	2	3	4	5	6	7	8	9
					14	5			8	9

2) • Collision at 28 → apply second hash

$h_2(28) = 1 + (28 \bmod 9)$

Step size → $= 2$

• new address = $h_1(28) + i \times h_2(28) \bmod 10$ (i = # of collisions)

$= (8 + 1 * 2) \bmod 10$

$= 10 \bmod 10 = 0$

(10 comes from size of table)

↳ Insert 28 into index 0.

3) Inserting 30, $h_1(30) = 0$, collision

$h_2(30) = 1 + (30 \bmod 9) = 4$, collision

$h(30) + i * h_2(30) \bmod 10 = 4$ collision

$(0 + 2 * 4) \bmod 10 = 8$ collision

$(0 + 3 * 4) \bmod 10 = 12 \bmod 10 = 2$

0	1	2	3	4	5	6	7	8	9
28		30		14	5			18	9

- Linear Probing: new address = $h_1(key) + i$ mod table size
- Quadratic Probing: new address = $h_1(key) + i^2$ mod table size
- Double hashing: new address = $h_1(key) + i \times h_2(key)$ mod table size

Coding Questions

- Evaluate effectiveness of given hash function

Good Hash Function Satisfies 4 Characteristics

1) Incorperating all information in the key
2) Uniform Mapping to Addresses
3) Fast Computation
4) Discontinuity

Specific to Question

1) function uses both name and priority, but "abC" and "cab" have same hash value
2) multiplication by priority causes clustering
3) This function is fast only in O(n) time
4) Small changes in name task A vs task B, results in small ascii changes

∴ Not a great function.
Better way is weigh based on characters

---

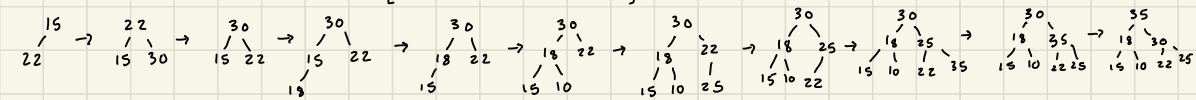
2) given array
returns K^{th} smallest number
leverage heaps

1) use max heap of size K to store K smallest elements seen so far
2) K^{th} smallest element is always at root. If elem is smaller then replace root and heapify
3) process all elements

Complexity: O(n log n)

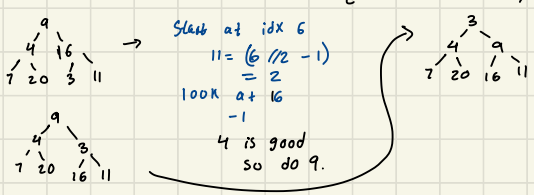
Quiz 2 – Practise Questions

1) max heap one by one, using: [15, 22, 30, 18, 10, 25, 35]



Maintained Heap Property:

2) heapify array to min heap: [9, 4, 16, 7, 20, 3, 11]



3) Insert [10, 15, 23, 8, 20, 25] to closed hash table

0	1	2	3	4	5	6
	15	23	10	8	25	20

h(k) = k mod 7

4) • open hash table, size = 10, double hashing

$h_1(k) = k \bmod 10$

$h_2(k) = 1 + (k \bmod 9)$

Array: [9, 5, 18, 14, 28, 30]

Table:

0	1	2	3	4	5	6	7	8	9
28		30		14	5			18	9

1) First collision at 28: 28 mod 10 = 8, 8 is already taken by 18. Apply second hash: $h_2(28) = 1 + 1 = 2$

new address: $h(k) + i\,h_2(k) \bmod 10 \rightarrow 8 + 1(2) \bmod 10 = 0$

2) Second collision at 30: 30 mod 10 = 0, 0 is taken

Apply $h_2(30) = 1 + 30 \bmod 9 = 4$

new address: $0 + i(4) \bmod 10 = 4$, already taken

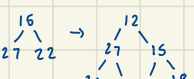
$0 + 2(4) \bmod 10 = 8$, already taken

$0 + 3(4) \bmod 10 = 12$, 2 not taken

---

1) min heap, one by one way:

array = [15, 27, 22, 30, 29, 12, 18]

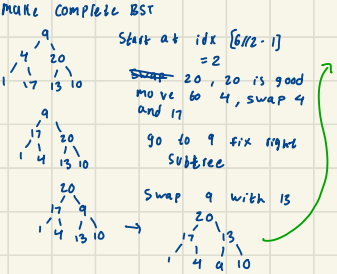


array min heaped: [12, 27, 15, 30, 29, 22, 18]

2) Describe steps heapify to max heap

array: [9, 4, 20, 1, 17, 13, 10]

1) make complete BST



3) Keys = [12, 18, 14, 24, 30, 42, 19]

$h(K) = K \bmod 11$

$h(K, i) = (h'(K) + i^2) \bmod 11$

0	1	2	3	4	5	6	7	8	9	10
	12	24	14			19	18	30	42	

1) For 18 $= 7 + i^2$

$8 \bmod 11 = 3$

2) For 14 $\bmod 11 = 3$, 3 is taken

$= 3 + 1^2 \bmod 11$

$= 4 \bmod 11 = 4$

3) For 30 $\bmod 11 = 8$,

$= 8 + 1^2$ $9 \bmod 11 = 9$

$= 8 + 2^2 \bmod 11 = 1$

$= 8 + 3^2 \bmod 11 =$

4) For 42 $\bmod 11 =$

$9 + i^2 \bmod 11 = 10$

$9 + 2^2 = 2$

$9 + 3^2$

Search complexity: $\frac{7}{11} \simeq 0.636$

Avg probes needed: $\frac{9}{7} \simeq 1.3$

---

4) Keys: [20, 32, 54, 76, 43, 35, 91]

$h_1(K) = K \bmod 10$

$h_2(K) = 1 + K \bmod 7$

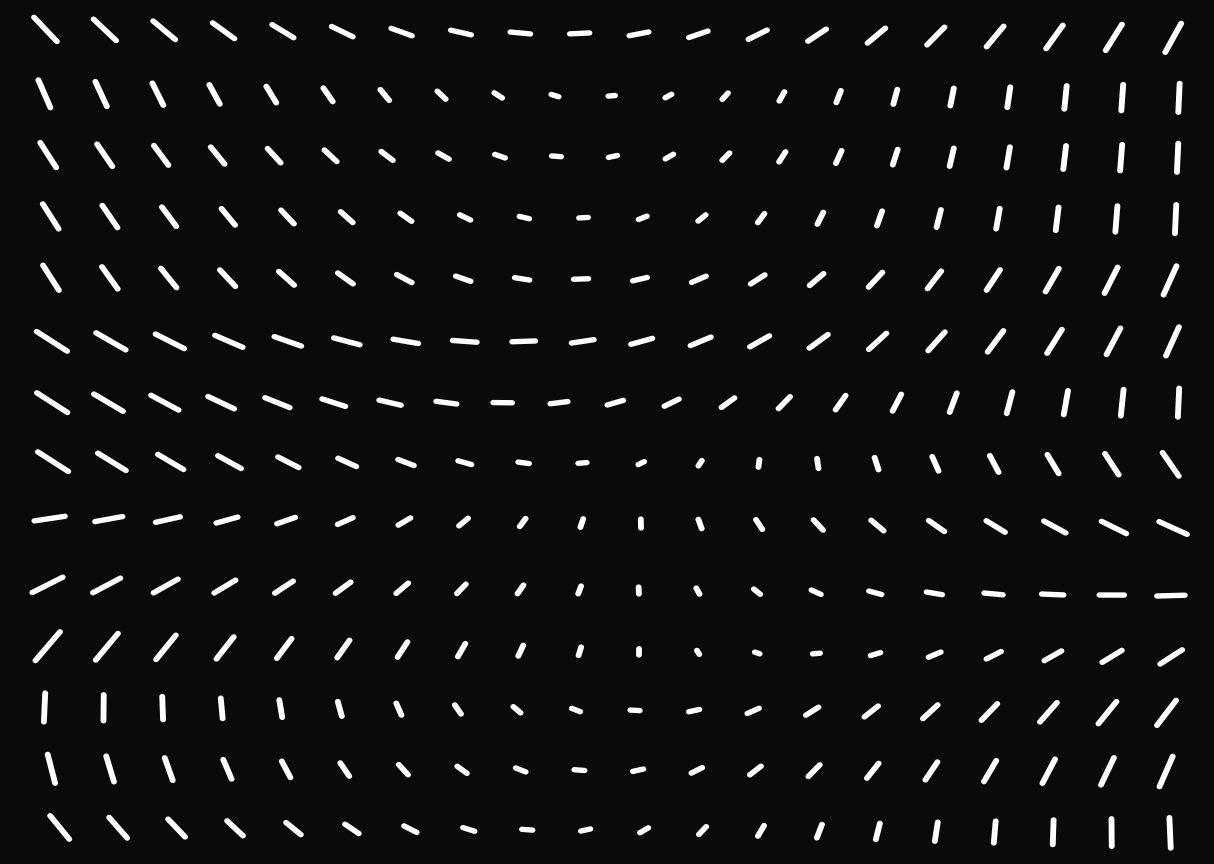
new address: $h_1'(K) + i \cdot h_2(K)$

0	1	2	3	4	5	6	7	8	9
20	10	32	43	54	35	76			

Analyzing Hash Functions

- incorperating all characters, meaningfully
- Uniform mapping (minimizing clustering)
  - Sum = Sum * C + x (makes order matter)
  - Mid Square is bad for small telephone numbers, insurance numbers
  - Regular K mod m
  - Multiplication method multiply by V, take decimal, $V = \frac{\sqrt{5} - 1}{2}$, same V values
  - Hashing Strings, making it a number, $a = a * c + ord(x)$
    - ↳ To make better multiply by Constant

Exam Review



Please message/Email me for more info!