arr[i] = *(a+i)
Start + (i x # bytes)

Notes

XOR

Array & pointers = equivalent in C

Sorted array = fastest = BS

Endian: LSB in
Big = High address
Little = Low

C arithmetic promotes lower type to higher type

Scope / printf (with format specifiers)

sizeof() = # of bytes

0 = false, everything else (1) = **true**

== ⇒ equality, = assignment

cannot return array type ← Functions are defined once, declared / called many times

Call-by-value in C → parameters copy are passed, value DOESN'T change

& = address of

Static variables = inside function (persist value), outside = used in multiple func in file

name of array = pointer to element 0

Lecture 6: e^x, $\sqrt{x}$, sin(x), Taylor series, FP arithmetic = errors / approximations

String = array of characters that ends in '\0'

(Lecture 8) Pointer = variable that holds memory address (or NULL) points to location of obj in memory

have addresses ← Pointer Arithmetic (++/--) → prev/next address (can +/- numbers) no AD

Stacks = LIFO (capacity can increase = dynamic)

Queue = FIFO (priority queue = elements of higher are DQed before (lower)

return void * ← Dynamic memory allocation = allocated on heap @ run time

malloc() → allocate spaces, calloc() → allocate + set to 0, realloc() → realloc ptr to point at bytes on heap

free (void * ptr) → Frees memory (seg fault → acces mem. you can't)

tail recursion = iterative ← Recursion → function calls itself (func. call creates stack frame

Formal: Graph has set of vertices and set of edges

Adjacency Matrix: n×n matrix (non-zero = edge) → $O(n^2)$ space

Topological ordering

Adjacency List: Column array for nodes (LL of edges)

ex. slide → (Lecture 13)

BFS → uses queue, explore reachable vertices and repeat

DFS → uses recursion (stack), search as far as possible then back up

Hamiltonian path = (proof) path that visits each vertex ONCE → go back to origin

Makefile → build exec from source code (DAG) topological

$(var) - value, variable assignments ⇒ = lazy, := immediate, ?= conditional, += concat

Makefile can recursively call another

$@ = target name, $^ all dependencies, $? dep. more recent than target, $< = 1st

Shell: command expansion, * = expansion, % = pattern match placeholder

Lempel-Ziv (L278)

Entropy = measure of uncertainty ($H = -\sum_{i=1}^{n} p_i \log_2(p_i)$)

(Lecture 15) = Huffman coding → histogram → PQ → H Tree → code table →

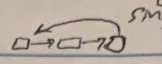
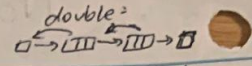
Sets: unordered collections characterized by elements they contain

Insert: val | (1 << (x % size(val)) / Remove: w & ~(1 << (x % size

Bit vectors: set: a[i/size] |= (1 << (i % size))

clear: a[i/size] &= (1 << (i % size))

get: return (a[i/size] >> (i % size)) & 1

extra memory than arrays  

Linked Lists: cannot randomly access elements / Double: each points to prev. / next

Sentinel nodes = dummy NULL

Inserting into D-LL: pro: clean up logic of insert / con: need 2 extra nodes

Traversals: if(root):

Preorder:	Inorder:	Postorder:	Level order:
do something	print left	print left	same as BFS
print left	do something	print right	visit by level
print right	print right	do something	use queue

ignore duplicates

BST: ordered → key less than val → left subtree & key greater than val → right

Balanced: height of left differs at most 1 from right

File = sequence of bytes root → user files + hierarchical directory

Cryptography = make message unreadable except for recipient

message —E→ ciphertext —D→ message

RSA: factoring large numbers (anyone has public key, private = recipient)

KNOW GMP STUFF: $n = p \times q$, $\phi(n) = p-1(q-1)$, $\gcd(e, \phi(n)) = 1$, $ex \, d = 1 \bmod \phi(n)$, $E(m) = m^e \pmod n$, $D = c^d \pmod n$

Debug: scan-build, valgrind

static = compile-time, dynamic = run-time / lldb

Compiling: Preprocessor → Compiler → Assembler → Linker → Executable

(Lecture 23) Memory = Stack / heap, data / text; assembly code; object code; libraries; Loader

JUMP Regex: matches patterns (+ 1 or more, * 0 or more, . = any [char], () group, {#} # of matches)

NFA → state transition not uniquely determined by curr. state / input

DFA = math. model that accepts set of words → one string → result

BASH, $PATH, variables = store data ($(var) = val)) Scripts

Function, array = list of strings, * = matches any string

awk = pattern matching

tests in [[ ]], str comparing = </>, in [[ ]] for ints use >/<

File descriptors = stdin = 0, stdout = 1, stderr = 2, | = connect stdout of process to stdin of other

load: how much HT is filled

Hashing → take key and make number → index of value

Hash Table = unordered collection of key-value pairs h(key) = index [∈ [0, m-1]]

Collisions = 2 pieces of data have same hash value

Bloom Filter → to see if element is in set, hash → set bits, false positive maybe in set

2 questions on security, 2-3 on multithreading

Processes:

Memory hierarchy → Register → L1 → L2 → L3 → DRAM → DISK

1 process at a time! (unless multi-core)

Process can create child process

VM (virtual memory) → OS → more memory than exists → recent: physical; less: disk

Process states

Creation of process: system initialization / execution of process creation system call

Voluntary / Involuntary exit