Dynamic Memory Allocation (DMA)

- allocated at run time from region called a heap
  - dynamic = on the fly allocation during run time
    - CTA = compile run time = memory from variable allocated by computer at compile time
      - requires exact size/type of storage at compile time
    - DMA: calculated & allocates exact memory needed at run time
  - DMA: allocates memory when needed + however much needed & variables can be accessed beyond current scope
- calloc(), malloc(), and realloc() in stdlib.h
  - must use free() → otherwise cause memory leak
- Heap = unmanaged, anonymous memory region
  - slow to read/write bc ptrs are needed
- void* malloc(size_t size) → returns pointer to [size] bytes of uninitialized memory allocation
  ↳ EX) int *arr = malloc(10 * sizeof(int));
  // allocates memory for array of 10 ints
- void* calloc(size_t nmemb, size_t size)
  - $\boxed{nmemb}$ = # of objects & $\boxed{size}$ = size of each object
  - returns ptr to [nmemb]*[size] bytes of allocated memory
  - slower than malloc but contents of allocated memory are known (zeroed out)
  ↳ EX) int* arr = calloc(10, sizeof(int)) //like malloc EX

- for matrix n×n allocation, use matrix_delete() for free
- void *realloc (void *ptr, size_t size)
  - reallocates [ptr] to newly point [size] bytes of memory
  - if [size] > size of originally allocated memory, the contents of the returned ptr contains everything from the original block (extra memory = uninitialized)
    - ↳ if [size] < original's size, contents of returned ptr contain exactly beginning [size] bytes from og bl
- free() = deallocating/freeing memory
  - ↳ void free(void *ptr) → deallocates memory space pointed to by [ptr]
  - memory leaks happen when alloc mem isn't freed
  - segmentation fault/core dump when program tries to access mem location it can't access (not allowed to)
  - ptr that have to be freed should = NULL
    - ↳ EX) int * arr = ...
      free(arr)
      arr = NULL
- gdb, infer, & valgrind find memory leaks/seg faults
- valgrind = collection of dynamic analysis tools → memcheck most useful for detection, reading, writing

Static vs Dynamic Analyzers

- static analyzers like infer analyze source code before run
  - compared to set(s) of coding rules for bugs
- dynamic analyzers like valgrind track errors in program execution (good for checking if program executes like it's supposed to)
  - only analyzes in execution (so anything outside execution can't be checked)

- infer checks for NULL ptr exceptions, resource leakage, race conditions, and missing lock guards

- recursion's a function, call requires creating stack frame (takes time & space)

- all tail recursive functions can be written as iteration

stack before func call

	parameter area	] caller
SP →	stack grows ↓	

stack after func call

	parameter area	] caller
	saved link register (LR)	callee
	saved frame ptr (old R7)	callee
R7 →	save registers	callee
	local storage	callee
SP →	stack grows ↓	

- use recursion when natural to express algorithm like that
  ↳ EX) binary search of ordered array in O(n log n) → fastest
    ↳ if empty, not there; if key smaller, look left; if key larger look right

binary_search()
string()
new_code() } examples of code in slides

recursion → use for dividing space up
- places we searched
- places we haven't searched
- if we get stuck, can try different path

Ex) Knight's Tour

- 8 Queens: print board
  is position safe?
  search for solution

- recursion is natural, good for search problems, not inherently inefficient, should use where it makes sense

Graphs

- graph = network routing (from graph theory)
  - nodes = computers, edges = connections
  social networks
- graph = $G = \langle V, E \rangle$ vertices & edges
- V = set of vertices = $\{v_1, v_2, v_3 \ldots\}$
- E = set of edges = $\{\langle v_i, v_j \rangle, \langle v_p, v_q \rangle, \ldots \langle v_s, v_t \rangle\}$

- edges can have directions or no directions (undirected)
- weight can represent capacity, strength, cost, etc
  ↳ edges contain weights

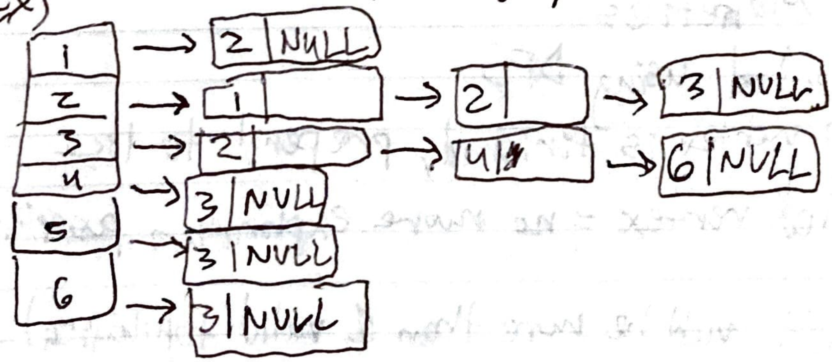
- Adjacency Matrix = $n \times n$, weight $\neq 0$, binary (edges present or absent)
- Adjacency List (AL)
  - column array for nodes
  - linked list of edges from each node
  - may contain weights

- non-zero M_{ij} entry means edge $n_i \to n_j$ exists
- undirected matrix if symmetric around diagonal
- entry specifies existence of edge & its weights
- requires $O(n^2)$ space (can be improved)

[graph example in C on pg 11 Lecture 13)

- AL:
  - each node represented as entry in column vector
  - linked elements have destination node & weight of edge
  - efficient for sparse graphs

EX)



[AL C code on pg 13 Lecture 13)

Single-Source Shortest Path (SSSP)
- assume Graph = $\langle V, E \rangle$ & source vertex $s \in V$
- want shortest path from s to any $v \in V$
- use Bellman-Ford or Dijkstra's algorithms

- Dijkstra's Algorithm at end of file

- Hamiltonian Path
  - undirected or directed graph, visit each vertex only once
    - must start from origin vertex & end at origin

- Eulerian Path
  - undirected or directed graph, visit each edge only once
    - must start & end at origin like Hamiltonian