1. gcc hello.c −o hello quiz4: quiz4.h quiz4.c quiz4\_test.c. gcc quiz4.c quiz4\_test.c −o quiz4
2. scanf(“%d”, &a); scanf(“%s”, s); char 1byte int 2byte float 4 byte double 8byte
3. array: Contiguous chunk of memory;Must hold the same type in every slot;  Must specify the size when you create it ;YOU must make sure you don’t run off the end ;A ‘string’ is just an array of characters with a special character at the end(\0 ) ;function signature: **return-value-type function-name(parameter list) {statements}**
4. **function prototype**: tells C what to expect from your function. The **function definition** has a well-defined structure and implements the function**. Function scope;**File scope;Block scope ;Function-prototype scope
5. Naming an array, stating its type, and specifying the number of elements in the array is called **declare** the array. An array declaration reserves space for the array. Array is a group of memory locations related by the fact that they have the same name and same type. **Declare**: int c[13]; initializing: use a loop/ int n[3] = {52, 13, -5}; / Use a single value: int n[3] = {0};
6. void printArray(int \*a, int len) = void printArray(int a[], int len);multi-dimensional arrays: When passing to a function, \*have\* to specify all dimensions past the first. int b[2][2] = {{1, 2}, {3, 4}}; printDiags(int\*[3] box){}
7. ‘\n’ (newline), ‘\0’ (null), ’\t’ (tab) ; string: Represented by an array of chars ending with a NULL character (‘\0’)
8. When you initialize a string, you get back a pointer to a char array ;Unlike regular arrays, you can use the existence of the \0 to find the end of the array ;An array of strings turns out to be an array of pointers to arrays of chars;Because an array is just a pointer to the first element of the array, an array of strings is a pointer to a pointer. **Declare:** char color[] = “blue”;char \*colorPtr = “blue”; char color[] = {‘b’, ‘l’, ‘u’, ‘e’, ‘\0’}
9. char \*strcpy(char \*s1, const char \*s2): Copy s2 into the array of s1, returns the value of s1;; char \*strcat(char \*s1, const char \*s2): Append string s2 to the array s1 (overwriting the original \0 of s1); return the val of s1.
10. int strcmp(const char \*s1, const char \*s2) 0: ==; -: left < right; +: left > right ;; char \*strchr(const char \*s, int c): Finds first occurrence of a char c in s, returns a pointer to c in s. If not found, returns NULL. **<string.h>**
11. char \*foo = “blue”; char \*\*str = &foo; int bar = 0; int\* barPtr = &bar;
12. struct card foo1; struct card foo2 = foo1; struct card \*cPtr = &foo1; initialize: struct card a = {“queen”, “king”};
13. access member by name: a.suit; by pointer: struct card\* aPtr = &a; a->suit; typedef struct card Card = typedef struct{};Card
14. enum suit {SPADES=1, HEARTS, CCLUBS};
15. cd lect2 emacs p1.c& Starting a new problem/flashcard :•Ctrl-x Ctrl-f (new file) • p3.c (filename) • Ctrl-x b (*change buffer*)• Ctrl-k (*a few times, to get the first few lines*) • Ctrl-x b (*go back to my new buffer*)• Ctrl-y (*yank the info into the new buffer*) compile and running: • M-x shell-command <enter> • gcc p2.c <enter>• M-x shell-command <enter> • ./a.out • M-! (M + Shift + 1) <enter> make your code look good: intending a region• Ctrl-<space>• Ctrl-P/Ctrl-N, etc (selecting your region) • M-x indent-region
16. **Valgrind:**  testing whether your code has any memory leaks Valgrind –leak-check=full ./test

**stack:** struct stackOfBooks{char\* booknames[SIZE];int bookAtTop;} ;stackOfBooks**\* create(){** stackOfBooks\* newStack =(stackOfBooks\*)malloc(sizeof(stackOfBooks));newStack->bookAtTop = -1;return newStack;}stackOfBooks\* **push**(char\* newBook, stackOfBooks\* bookStack){bookStack->bookAtTop++; bookStack->booknames[WHAT GOES HERE] = newBook;return bookStack;}char\* **peek**(stackOfBooks\* bookStack){return bookStack->booknames[WHAT GOES HERE];}

**Queue: 1. Array** struct Queue {  int front, rear, size; int\* array; };struct Queue\* **createQueue**() { struct Queue\* queue = (struct Queue\*) malloc(sizeof(struct Queue)); queue->front = queue->size = 0;  queue->array = (int\*) malloc(queue->capacity \* sizeof(int));  return queue; } void **enqueue**(struct Queue\* queue, int item) { if (isFull(queue)) return;  queue->rear = (queue->rear + 1); queue->array[queue->rear] = item; queue->size = queue->size + 1; }

int **dequeue**(struct Queue\* queue) { if (isEmpty(queue)) return INT\_MIN; int item = queue->array[queue->front]; queue->front = (queue->front + 1); queue->size = queue->size - 1; return item; } int **front**(struct Queue\* queue) { if (isEmpty(queue)) return INT\_MIN; return queue->array[queue->front]; } nt **rear**(struct Queue\* queue) { if (isEmpty(queue)) return INT\_MIN; return queue->array[queue->rear]; } 2**. Linked list**: struct QNode { int key; struct QNode \*next; }; struct Queue { struct QNode \*front, \*rear; }; struct QNode\* newNode(int k) { struct QNode \*temp = (struct QNode\*)malloc(sizeof(struct QNode)); temp->key = k; temp->next = NULL;  return temp;

struct Queue \***createQueue**() { struct Queue \*q = (struct Queue\*)malloc(sizeof(struct Queue)); q->front = q->rear = NULL; return q; } void **enQueue**(struct Queue \*q, int k) { struct QNode \*temp = newNode(k); if (q->rear == NULL) { q->front = q->rear = temp; return; } q->rear->next = temp; q->rear = temp; } struct QNode \***deQueue**(struct Queue \*q) { if (q->front == NULL) return NULL; struct QNode \*temp = q->front; q->front = q->front->next; if (q->front == NULL) q->rear = NULL; return temp; }

**Linked list:** struct LinkedList{Node\* firstNode;};**delete**(4):curNode = head; while(curNode != NULL){if (curNode->next**->data** == 4){ curNode->next = curNode->next->next; } curNode = curNode->next;}**Double linked list**: struct Node( char\* data; Node\* prevNode; Node\* nextNode;) linked list is non-contiguous, size differ

list\* **create**(){list\* list = (list\*)malloc(sizeof(list));return list;}

node\* **createNode**(char\* data){node\* node = (node\*)malloc(sizeof(node)); node->data = data;node->prevNode = NULL;node->nextNode = NULL;return node;} void **destroy**(list\* aList){free(aList->firstNode); free(aList);}

void **insert**(list\* aList, node \*newNode){ if(aList->firstNode != NULL){newNode->nextNode = aList->firstNode;

aList->firstNode->prevNode = newNode;}aList->firstNode = newNode;}

void **remove(**node\* aList, node\* aNode){if(aNode->prevNode != NULL){aNode->prevNode->nextNode

aNode->nextNode;} if(aNode->nextNode != NULL){aNode->nextNode->preNode = aNode->preNode;}

**Trees** are connected, acyclic, undirected graphs, Directed acyclic graphs are called DAGs .

**Tree: traversal:** void printTree(Node \*root){printf(“%s\n”, root->data);printTree(root->leftChild); printTree(root->rightChild);} **DFS**: stack s;push(s, root); Node\* curNode; while (!isEmpty(s)){ curNode = pop(s); print(curNode);push(s, curNode->right); push(s, curNode->left);}

**Insert:** Node\* insert(Node\* root, int key, int value) {if (!root) return new Node(key, value);else if (key < root->key)

root->left = insert(root->left, key, value);else //key >= root->key root->right = insert(root->right, key, value);return root;

**Delete:** def deleteNode(self, root, key): if not root: return None;if root.val < key:root.right = self.deleteNode(root.right, key);elif root.val > key:root.left = self.deleteNode(root.left, key);else:if not root.left:return root.right;if not root.right:

return root.left; if not root.right.left:root.right.left = root.left;return root.right;else:par = cur = root.right;while cur.left:

par = cur;cur = cur.left;par.left = cur.right;cur.left = root.left;cur.right = root.right;return cur;return root

**Heap:** void insert(heap \*heap, int val){heap->vals[heap->nextVal++] = val;int new\_ind = heap->nextVal;int par\_ind = (new\_ind - 1)/2;while (heap->vals[par\_ind] > heap->vals[new\_ind]){// Swap them}}

**Graph:** matrix: size |V^2|, Adjacency list space: a|V|+b|E|(a is size of node, b is size of linkedlist element)

Terminology: A DAG represents a partial order and a topological sort produces a total order that is consistent with it

A directed graph is strongly connected if there is a directed path between any two vertices.

**Djikstra’s :** int minDistance(int dist[], bool sptSet[]) { int min = INT\_MAX, min\_index; for (int v = 0; v < V; v++) if (sptSet[v] == false && dist[v] <= min) min = dist[v], min\_index = v; return min\_index; } double **dijkstra(**double \*\*graph, int source, int V, int dest, char\* city[]) {double distance[V];int finalized[V], index[V];int i, j;for (i = 0; i < V; i++) {distance[i] = DBL\_MAX;finalized[i] = 0;}distance[source] = 0;for (i = 0; i < V; i++) {int minIndex = minimumPathVertics(distance, finalized, V);finalized[minIndex] = 1;for (j = 0; j < V; j++) {if (finalized[j] == 0 && graph[minIndex][j] != -1&& distance[minIndex] != DBL\_MAX&& distance[minIndex] + graph[minIndex][j] < distance[j]) {distance[j] = distance[minIndex] + graph[minIndex][j];}}}

**Adacency List:** struct ListNode {int dest;int weight;struct ListNode\* next;};struct List {struct ListNode \*head;};struct Graph {int V;struct List\* array;}; ListNode\* **newListNode(**int dest, int weight) {ListNode\* newNode = (ListNode\*) malloc(sizeof(ListNode));newNode->dest = dest;newNode->weight = weight;newNode->next = NULL;return newNode;} Graph\* **createGraph**(int V) {Graph\* graph = (Graph\*) malloc(sizeof(Graph));graph->V = V;graph->array = (List\*) malloc(V \* sizeof(List));for (int i = 0; i < V; ++i) {graph->array[i].head = NULL;}return graph;}void **addEdge**(Graph\* graph, int src, int dest, int weight) {ListNode\* newNode = newListNode(dest, weight);newNode->next = graph->array[src].head; graph->array[src].head = newNode;} void **destroyGraph**(Graph\* graph) {int V = graph->V;

ListNode\* tmp;for (int i = 0; i < V; i++) {while (graph->array[i].head != NULL) {tmp = graph->array[i].head;

graph->array[i].head = graph->array[i].head->next;free(tmp);}}free(graph->array);free(graph);}

int\*\* time;time = (int \*\*)malloc(NUM\_CITY \* sizeof(int \*));for (i = 0; i < NUM\_CITY; ++i) {int a;time[i] = (int \*)malloc(NUM\_CITY \* sizeof(a));}

;: semicolon **#define** SIZE 10 fourth element: array[3] element 4: array[4] printf(“%.2f”)