DATA STRUCTURES CHEAT SHEET

1. Arrays: In C, arrays are used to store a predetermined number of identically typed elements.

Function: Store elements of the same type in contiguous memory locations.

Key Operations: Access elements by index, iterate through elements, modify elements.

Code :

1. #include <stdio.h>
2. #define SIZE 5
4. **int** main() {
5. **int** arr[SIZE] = {2, 4, 6, 8, 10};
6. **int** i;
8. **for** (i = 0; i < SIZE; i++) {
9. printf("%d ", arr[i]);
10. }
12. **return** 0;
13. }

2. Linked Lists: Linked lists in C are used to store a collection of elements in a dynamic way. Linked lists are useful for implementing dynamic data structures that can be easily modified.

Function: Store elements where each element points to the next, forming a chain.

Key Operations: Insertion and deletion of elements, traversal, searching.

Code:

1. #include <stdio.h>
2. #include <stdlib.h>
4. **struct** Node {
5. **int** data;
6. **struct** Node \*next;
7. };
9. **void** printList(**struct** Node \*node) {
10. **while** (node != NULL) {
11. printf("%d ", node->data);
12. node = node->next;
13. }
14. }
16. **int** main() {
17. **struct** Node\* head = NULL;
18. **struct** Node\* second = NULL;
19. **struct** Node\* third = NULL;
21. head = (**struct** Node\*)malloc(**sizeof**(**struct** Node));
22. second = (**struct** Node\*)malloc(**sizeof**(**struct** Node));
23. third = (**struct** Node\*)malloc(**sizeof**(**struct** Node));
25. head->data = 1;
26. head->next = second;
28. second->data = 2;
29. second->next = third;
31. third->data = 3;
32. third->next = NULL;
34. printList(head);
36. **return** 0;
37. }

3. Stacks: Last-in, first-out (LIFO) order is used to store a group of elements in stacks in the C programming language. For implementing algorithms that need a hierarchical ordering, stacks are helpful.

Function: Follow Last In, First Out (LIFO) principle for storing elements.

Key Operations: Push (insert), pop (remove), peek (retrieve top element), check if empty.

Code :

1. #include <stdio.h>
2. #include <stdlib.h>
4. #define MAX\_SIZE 100
6. // define the structure for the stack
7. **struct** stack {
8. **int** items[MAX\_SIZE];
9. **int** top;
10. };
12. // function to create an empty stack
13. **struct** stack\* createStack() {
14. **struct** stack\* s = malloc(**sizeof**(**struct** stack));
15. s->top = -1;
16. **return** s;
17. }
19. // function to check if the stack is empty
20. **int** isEmpty(**struct** stack\* s) {
21. **if** (s->top == -1)
22. **return** 1;
23. **else**
24. **return** 0;
25. }
27. // function to check if the stack is full
28. **int** isFull(**struct** stack\* s) {
29. **if** (s->top == MAX\_SIZE - 1)
30. **return** 1;
31. **else**
32. **return** 0;
33. }
35. // function to add an element to the stack
36. **void** push(**struct** stack\* s, **int** value) {
37. **if** (isFull(s))
38. printf("Stack is full!");
39. **else** {
40. s->top++;
41. s->items[s->top] = value;
42. }
43. }
45. // function to remove an element from the stack
46. **int** pop(**struct** stack\* s) {
47. **int** item;
48. **if** (isEmpty(s)) {
49. printf("Stack is empty");
50. item = -1;
51. } **else** {
52. item = s->items[s->top];
53. s->top--;
54. }
55. **return** item;
56. }
58. **int** main() {
59. **struct** stack\* s = createStack();
61. push(s, 1);
62. push(s, 2);
63. push(s, 3);
65. printf("Popped item: %d\n", pop(s));
66. printf("Popped item: %d\n", pop(s));
67. printf("Popped item: %d\n", pop(s));
69. **return** 0;
70. }

4. Queues: Queues in C are used to store a collection of elements in a first-in, first-out (FIFO) order. For implementing algorithms that need a processing order, queues are helpful.

Function: Follow First In, First Out (FIFO) principle for storing elements.

Key Operations: Enqueue (insert), dequeue (remove), front (retrieve front element), check if empty.

Code:

1. #include <stdio.h>
2. #include <stdlib.h>
4. #define MAX\_SIZE 100
6. // define the structure for the queue
7. **struct** queue {
8. **int** items[MAX\_SIZE];
9. **int** front;
10. **int** rear;
11. };
13. // function to create an empty queue
14. **struct** queue\* createQueue() {
15. **struct** queue\* q = malloc(**sizeof**(**struct** queue));
16. q->front = -1;
17. q->rear = -1;
18. **return** q;
19. }
21. // function to check if the queue is empty
22. **int** isEmpty(**struct** queue\* q) {
23. **if** (q->rear == -1)
24. **return** 1;
25. **else**
26. **return** 0;
27. }
29. // function to add an element to the queue
30. **void** enqueue(**struct** queue\* q, **int** value) {
31. **if** (q->rear == MAX\_SIZE - 1)
32. printf("Queue is full!");
33. **else** {
34. **if** (q->front == -1)
35. q->front = 0;
36. q->rear++;
37. q->items[q->rear] = value;
38. }
39. }
41. // function to remove an element from the queue
42. **int** dequeue(**struct** queue\* q) {
43. **int** item;
44. **if** (isEmpty(q)) {
45. printf("Queue is empty");
46. item = -1;
47. } **else** {
48. item = q->items[q->front];
49. q->front++;
50. **if** (q->front > q->rear) {
51. q->front = q->rear = -1;
52. }
53. }
54. **return** item;
55. }
57. **int** main() {
58. **struct** queue\* q = createQueue();
60. enqueue(q, 1);
61. enqueue(q, 2);
62. enqueue(q, 3);
64. printf("Dequeued item: %d\n", dequeue(q));
65. printf("Dequeued item: %d\n", dequeue(q));
66. printf("Dequeued item: %d\n", dequeue(q));
68. **return** 0;

5. Trees: A tree is a hierarchical data structure in which every node has at least one child and at least one parent. The bottom nodes are referred to as leaves, and the uppermost node is known as the root. Uncommon tree data structures include binary trees.

Function: Hierarchical structure with a root node and children nodes, used for hierarchical data.

Types: Binary Trees, Binary Search Trees (BST), AVL Trees, Red-Black Trees.

Key Operations: Insertion, deletion, traversal (in-order, pre-order, post-order), search.

6. Graphs: A graph is made up of nodes, or vertices, and the connecting edges. We know that graphs can be cyclic or acyclic, directed or undirected and also weighted or unweighted.

Function: Represent networks with nodes (vertices) and edges (connections).

Types: Directed, undirected, weighted, un weighted.

Key Operations: Add/Remove vertices and edges, traversal (DFS, BFS), search, shortest path algorithms (Dijkstra’s, Floyd-Warshall).

8. Heaps: A heap is a specialized tree-based data structure that satisfies the heap property. In a min-heap, the parent node is always smaller than its children, and in a max-heap, the parent node is always larger than its children.

Function: A complete binary tree used to implement priority queues.

Types: Min-Heap, Max-Heap.

Key Operations: Insert, delete (usually delete-min or delete-max), heapify, extract-min/extract-max.