Advance C Library Manual

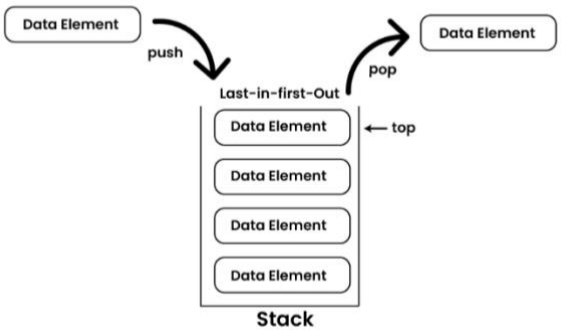
Welcome to the Advance C Library Manual! This manual serves as a comprehensive guide to various data structures implemented in C programming language, providing you with detailed information about their functionalities, usage, and examples.

Data structures are fundamental components of computer science and programming. They allow us to efficiently organize and manipulate data, facilitating tasks such as storing, searching, sorting, and retrieving information. By understanding and utilizing different data structures, programmers can develop more efficient and scalable solutions to a wide range of computational problems.

In this manual, we cover a variety of commonly used data structures, including vectors, ArrayLists, linked lists, hash maps, binary trees, doubly linked lists, and graphs. Each section provides a detailed overview of the data structure, its operations, and examples demonstrating how to use them effectively in your C programs.

1. Stack

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle, meaning that the most recently added element is the first one to be removed. It's like a stack of plates where you can only add or remove the top plate. Stacks are commonly used in algorithms for backtracking, expression evaluation, and function call management.



Create:

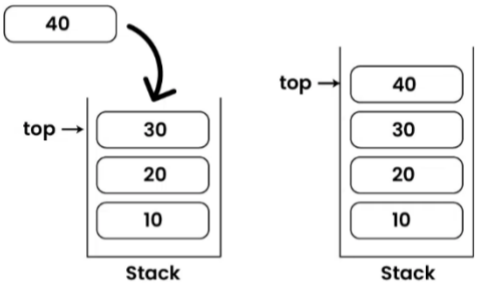
stack_create

- Syntax: Stack* stack_create(size_t capacity)
- Description: Creates a new stack with a specified capacity.
- Parameters:
  capacity: Initial capacity of the stack.
- Example:
  Stack* stack = stack_create(10);

Push:

stack_push

- Syntax: void stack_push(Stack* stack, int element)
- Description: Pushes an element onto the stack.
- Parameters:
  stack: Pointer to the stack.



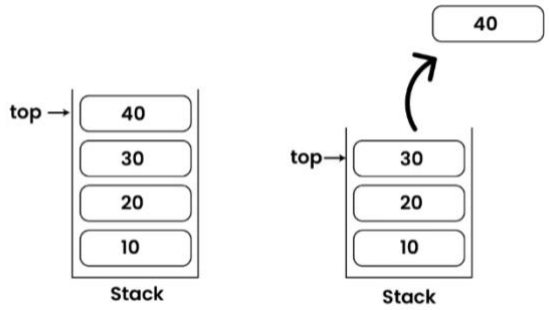
element: Element to be pushed onto the stack.

- Example:
  stack_push(stack, 42);

Pop:

stack_pop

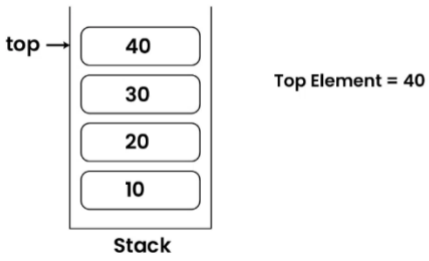
- Syntax: int stack_pop(Stack* stack)
- Description: Pops an element from the stack.
- Parameters:
  stack: Pointer to the stack.
- Returns:
  Element popped from the stack.
- Example:
  int element = stack_pop(stack);



Peek:

stack_peek

- Syntax: int stack_peek(const Stack* stack)
- Description: Retrieves the top element of the stack without popping it.
- Parameters:
  stack: Pointer to the stack.
- Returns:
  Top element of the stack.
- Example:
  int top_element = stack_peek(stack);



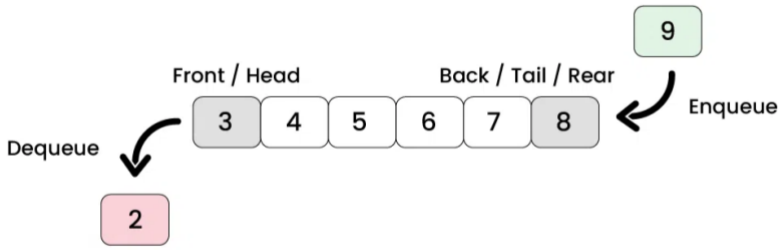
Destroy:

stack_destroy

- Syntax: void stack_destroy(Stack* stack)
- Description: Destroys the stack and frees memory.
- Parameters:
  stack: Pointer to the stack.
- Example:
  stack_destroy(stack);

2. Queue

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, meaning that the first element added to the queue will be the first one to be removed. It's like a queue of people waiting for a service where the first person who joins the queue is the first one to be served. Queues are commonly used in algorithms for job scheduling, breadth-first search, and task management.



Create:

queue_create

- Syntax: Queue* queue_create(size_t capacity)
- Description: Creates a new queue with a specified capacity.
- Parameters:
  capacity: Initial capacity of the queue.
- Example:
  Queue* queue = queue_create(10);

Enqueue:

queue_enqueue

- Syntax: void queue_enqueue(Queue* queue, int element)
- Description: Enqueues an element into the queue.
- Parameters:
  queue: Pointer to the queue.
  element: Element to be enqueued.
- Example:
  queue_enqueue(queue, 42);

Dequeue:

queue_dequeue

- Syntax: int queue_dequeue(Queue* queue)
- Description: Dequeues an element from the queue.
- Parameters:
  queue: Pointer to the queue.
- Returns:
  Element dequeued from the queue.
- Example:

int element = queue_dequeue(queue);

Front:

queue_front

- Syntax: int queue_front(const Queue* queue)
- Description: Retrieves the front element of the queue without dequeuing it.
- Parameters:
  queue: Pointer to the queue.
- Returns:
  Front element of the queue.
- Example:
  int front_element = queue_front(queue);

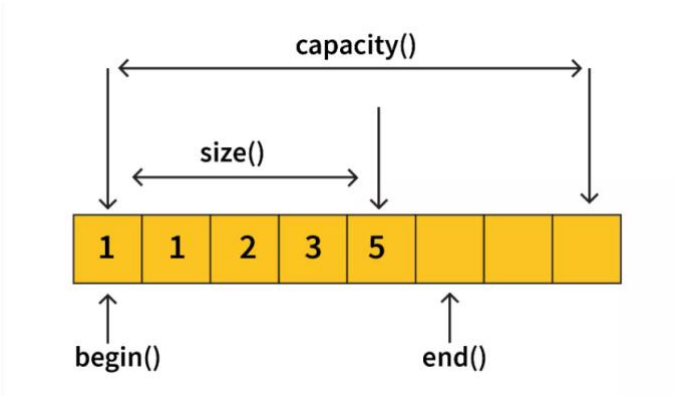
Destroy:

queue_destroy

- Syntax: void queue_destroy(Queue* queue)
- Description: Destroys the queue and frees memory.
- Parameters:
  queue: Pointer to the queue.
- Example:
  queue_destroy(queue);

3. Vector

A vector is a dynamic array that can resize itself automatically when elements are added or removed. It provides constant-time access to elements and efficient insertion and deletion at the end of the array.



Here's a breakdown of its operations and their functions:

Create: Creates a new vector with an initial capacity.

vector_create

- Syntax: Vector* vector_create()
- Description: Creates a new empty Vector.
- Parameters: None
- Example:
  Vector* vec = vector_create();

Destroy: Destroys the vector and frees up memory.

vector_destroy

- Syntax: void vector_destroy(Vector* vec)
- Description: Destroys the Vector and frees up memory.
- Parameters:
  vec: Pointer to the Vector to be destroyed.
- Example:
  vector_destroy(vec);

Add: Adds an element to the end of the vector. If the vector is full, it automatically resizes itself to accommodate the new element.

vector_add

- Syntax: int vector_add(Vector* vec, int element)
- Description: Adds an element to the Vector at the end.
- Parameters:
  vec: Pointer to the Vector.
  element: Element to be added to the Vector.
- Example:
  vector_add(vec, 42);

Get: Retrieves the element at a specified index in the vector.

vector_get

- Syntax: int vector_get(const Vector* vec, size_t index)
- Description: Gets the element at the specified index in the Vector.
- Parameters:
  vec: Pointer to the Vector.
  index: Index of the element to retrieve.
- Example:
  int element = vector_get(vec, 0);

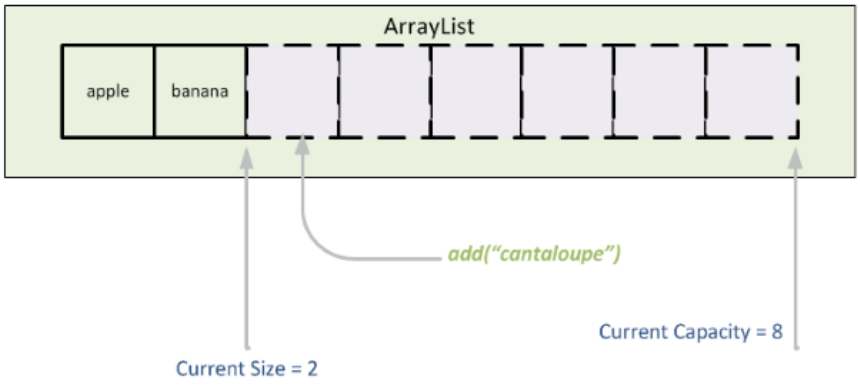
Remove: Removes the element at a specified index in the vector. It may involve shifting elements to fill the gap.

vector_remove

- Syntax: int vector_remove(Vector* vec, size_t index)
- Description: Removes the element at the specified index in the vector. It may involve shifting elements to fill the gap.
- Parameters:
  vec: Pointer to the Vector from which an element needs to be removed.
  index: Index of the element to be removed.
- Example:
  vector_remove(vec, 2);

4. ArrayList

An ArrayList is similar to a vector but provides better performance for random access. It dynamically resizes itself and allows efficient insertion and deletion at any position in the array. Here's a look at its operations:



Create: Creates a new ArrayList with an initial capacity.

arraylist_create

- Syntax: ArrayList* arraylist_create()
- Description: Creates a new empty ArrayList.
- Parameters: None
- Example:
  ArrayList* list = arraylist_create();

Destroy: Destroys the ArrayList and frees up memory.

arraylist_destroy

- Syntax: void arraylist_destroy(ArrayList* list)

- Description: Destroys the ArrayList and frees up memory.
- Parameters:
  list: Pointer to the ArrayList to be destroyed.
- Example:
  arraylist_destroy(list);

Add: Adds an element to the end of ArrayList. If needed, the ArrayList automatically resizes itself to accommodate the new element.

arraylist_add

- Syntax: int arraylist_add(ArrayList* list, int element)
- Description: Adds an element to the end of ArrayList.
- Parameters:
  list: Pointer to the ArrayList.
  element: Element to be added to the ArrayList.
- Example:
  arraylist_add(list, 20);

Get: Retrieves the element at a specified index in the ArrayList.

arraylist_get

- Syntax: int arraylist_get(const ArrayList* list, size_t index)
- Description: Gets the element at the specified index in the ArrayList.
- Parameters:
  list: Pointer to the ArrayList.
  index: Index of the element to retrieve.
- Example:
  int element = arraylist_get(list, 0);

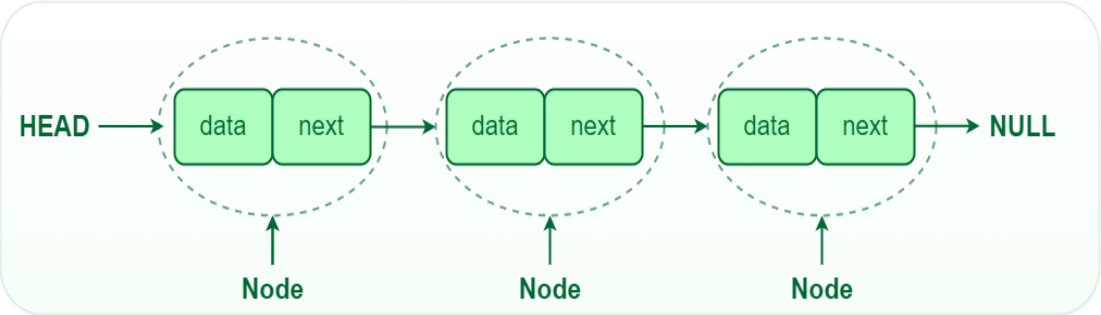
Remove: Removes the element at a specified index in the ArrayList. It may involve shifting elements to fill the gap.

arraylist_remove

- Syntax: int arraylist_remove(ArrayList* list, size_t index)
- Description: This function removes the element at the specified index in the ArrayList. It shifts the elements after the removed element to fill the gap.
- Parameters:
  list: Pointer to the ArrayList from which the element will be removed.
  index: Index of the element to be removed.
- Example:
  arraylist_remove(list, 1);

5. LinkedList

A LinkedList is a collection of nodes where each node contains a data element and a reference to the next node in the sequence. It provides efficient insertion and deletion operations but slower random access compared to arrays. Here are its operations:



Create: Creates a new empty LinkedList.

linkedlist_create

- Syntax: LinkedList* linkedlist_create()
- Description: Creates a new empty linked list.
- Parameters: None
- Example:
  LinkedList* list = linkedlist_create();

Destroy: Destroys the LinkedList and frees up memory.

linkedlist_destroy

- Syntax: void linkedlist_destroy(LinkedList* list)
- Description: Destroys the linked list and frees up memory.
- Parameters:
  list: Pointer to the linked list to be destroyed.
- Example:
  linkedlist_destroy(list);

Add: Adds an element to the end of the LinkedList.

linkedlist_add

- Syntax: int linkedlist_add(LinkedList* list, int element)
- Description: Adds an element to the linked list.
- Parameters:
  list: Pointer to the linked list.

element: Element to be added to beginning of the linked list.

- Example:
  linkedlist_add(list, 42);

Get: Retrieves the element at a specified index in the LinkedList by traversing the list from the beginning.

linkedlist_get

- Syntax: int linkedlist_get(const LinkedList* list, size_t index)
- Description: Gets the element at the specified index in the linked list.
- Parameters:
  list: Pointer to the linked list.
  index: Index of the element to retrieve.
- Example:
  int element = linkedlist_get(list, 0);

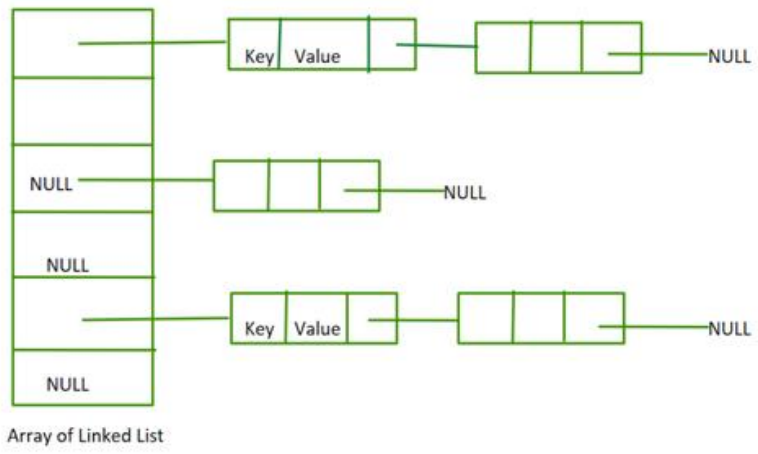
Remove: Removes the element at a specified index in the LinkedList by adjusting the references of neighboring nodes.

Other functions:

linkedlist_sort

- Syntax: void linkedlist_sort(LinkedList* list)
- Description: Sorts the linked list in ascending order using insertion sort.
- Parameters:
  list: Pointer to the linked list.
- Example:
  linkedlist_sort(list);

6. HashMap



Array of Linked List

A HashMap is a data structure that stores key-value pairs. It uses a hash function to map keys to indices in an array, allowing constant-time retrieval of values based on their keys. Here are its operations:

Create: Creates a new HashMap with an initial capacity.

hashmap_create

- Syntax: HashMap* hashmap_create(size_t capacity)
- Description: Creates a new HashMap with the specified capacity.
- Parameters:
  capacity: The initial capacity of the HashMap.
- Example:
  HashMap* map = hashmap_create(10);

Destroy: Destroys the HashMap and frees up memory.

hashmap_destroy

- Syntax: void hashmap_destroy(HashMap* map)
- Description: Destroys the HashMap and frees up memory.
- Parameters:
  map: Pointer to the HashMap to be destroyed.
- Example:
  hashmap_destroy(map);

Put: Inserts a key-value pair into the HashMap. If the key already exists, its associated value is updated.

hashmap_put

- Syntax: int hashmap_put(HashMap* map, int key, int value)
- Description: Puts a key-value pair into the HashMap.
- Parameters:
  map: Pointer to the HashMap.
  key: Key of the entry.
  value: Value associated with the key.
- Example:
  hashmap_put(map, 42, 100);

Get: Retrieves the value associated with a specified key in the HashMap.

hashmap_get

- Syntax: int hashmap_get(const HashMap* map, int key)
- Description: Gets the value associated with the specified key in the HashMap.
- Parameters:

map: Pointer to the HashMap.
key: Key of the entry to retrieve the value for.

- Example:
  int value = hashmap_get(map, 42);

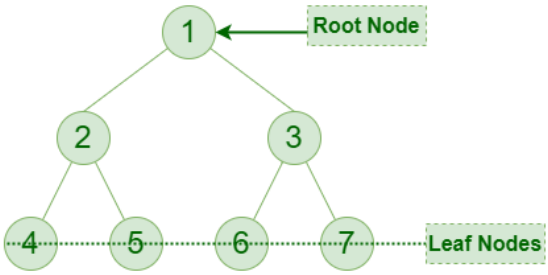
Remove: Removes the key-value pair associated with a specified key from the HashMap.

hashmap_remove

- Syntax: int hashmap_remove(HashMap* map, int key)
- Description: This function removes the key-value pair associated with the specified key from the HashMap.
- Parameters:
  map: Pointer to the HashMap from which the key-value pair will be removed.
  key: The key of the pair to be removed.
- Example:
  hashmap_remove(map, 54);

7. Binary Tree

A Binary Tree is a hierarchical data structure where each node has at most two children: a left child and a right child. It provides efficient searching, insertion, and deletion operations. Here are its operations:



Create: Creates a new empty Binary Tree.

binarytree_create

- Syntax: BinaryTree* binarytree_create()
- Description: Creates a new binary tree.
- Parameters: None
- Example:
  BinaryTree* tree = binarytree_create();

Destroy: Destroys the Binary Tree and frees up memory.

binarytree_destroy

- Syntax: void binarytree_destroy(BinaryTree* tree)
- Description: Destroys the binary tree and frees up memory.
- Parameters:
  tree: Pointer to the binary tree to be destroyed.
- Example:
  binarytree_destroy(tree);

Insert: Inserts a new node with the specified data into the Binary Tree according to certain rules (e.g., smaller values to the left, larger values to the right.

binarytree_insert

- Syntax: void binarytree_insert(BinaryTree* tree, int data)
- Description: Inserts a node with the given data into the binary tree.
- Parameters:
  tree: Pointer to the binary tree.
  data: The data to be inserted into the binary tree.
- Example:
  binarytree_insert(tree, 42);

insert_recursive

- Syntax: TreeNode* insert_recursive(TreeNode* node, int data)
- Description: Recursively inserts a new node into the binary tree.
- Parameters:
  node: Pointer to the current node in the binary tree.
  data: The data to be inserted into the binary tree.
- Example:
  binarytree_insert(tree, 10);

Traverse (Inorder, Preorder, Postorder): Traverses the BinaryTree in different orders, providing different sequences of node visits.

binarytree_traverse_inorder

- Syntax: void binarytree_traverse_inorder(const BinaryTree* tree)
- Description: Traverses the binary tree in-order.
- Parameters:
  tree: Pointer to the binary tree.
- Example:
  binarytree_traverse_inorder(tree);

binarytree_traverse_preorder

- Syntax: void binarytree_traverse_preorder(const BinaryTree* tree)
- Description: Traverses the binary tree pre-order.
- Parameters:
  tree: Pointer to the binary tree.
- Example:
  binarytree_traverse_preorder(tree);

binarytree_traverse_postorder

- Syntax: void binarytree_traverse_postorder(const BinaryTree* tree)
- Description: Traverses the binary tree post-order.
- Parameters:
  tree: Pointer to the binary tree.
- Example:
  binarytree_traverse_postorder(tree);

traverse_inorder_recursive

- Syntax: void traverse_inorder_recursive(const TreeNode* node)
- Description: Recursively performs in-order traversal of the binary tree.
- Parameters:
  node: Pointer to the current node in the binary tree.
- Example:
  binarytree_traverse_inorder(tree);

traverse_preorder_recursive

- Syntax: void traverse_preorder_recursive(const TreeNode* node)
- Description: Recursively performs pre-order traversal of the binary tree.
- Parameters:
  node: Pointer to the current node in the binary tree.
- Example:
  binarytree_traverse_preorder(tree);

traverse_postorder_recursive

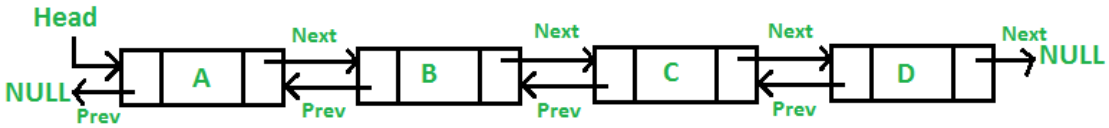
- Syntax: void traverse_postorder_recursive(const TreeNode* node)
- Description: Recursively performs post-order traversal of the binary tree.
- Parameters:
  node: Pointer to the current node in the binary tree.
- Example:
  binarytree_traverse_postorder(tree);

Other functions:

destroy_tree_nodes

- Syntax: void destroy_tree_nodes(TreeNode* node)
- Description: Recursively destroys the binary tree nodes.
- Parameters:
  node: Pointer to the root node of the binary tree.
- Example:
  destroy_tree_nodes(tree->root);

8. Doubly Linked List



A Doubly Linked List is a collection of nodes where each node contains a data element and references to both the previous and next nodes in the sequence. It allows traversal in both forward and backward directions.

Create: Creates a new empty Doubly Linked List.

doublylinkedlist_create

- Syntax: DoublyLinkedList* doublylinkedlist_create()
- Description: Creates a new empty doubly linked list.
- Parameters: None.
- Example:
  DoublyLinkedList* list = doublylinkedlist_create();

Destroy: Destroys the Doubly Linked List and frees up memory.

doublylinkedlist_destroy

- Syntax: void doublylinkedlist_destroy(DoublyLinkedList* list)
- Description: Destroys the doubly linked list and frees up memory.
- Parameters:
  list: Pointer to the doubly linked list to be destroyed.
- Example:
  doublylinkedlist_destroy(list);

Add: Adds a new node with the given data to the end of the Doubly Linked List.

doublylinkedlist_add

- Syntax: void doublylinkedlist_add(DoublyLinkedList* list, int data)
- Description: Adds a new node with the given data to the end of the doubly linked list.
- Parameters:
  list: Pointer to the doubly linked list.
  data: The data to be added to the doubly linked list.
- Example:
  doublylinkedlist_add(list, 42);

Traverse Forward: Traverses the Doubly Linked List in the forward direction, printing each element.

doublylinkedlist_traverse_forward

- Syntax: void doublylinkedlist_traverse_forward(const DoublyLinkedList* list)
- Description: Traverses the doubly linked list in the forward direction and prints its elements.
- Parameters:
  list: Pointer to the doubly linked list.
- Example:
  doublylinkedlist_traverse_forward(list);

Traverse Backward: Traverses the Doubly Linked List in the backward direction, printing each element.

doublylinkedlist_traverse_backward

- Syntax: void doublylinkedlist_traverse_backward(const DoublyLinkedList* list)
- Description: Traverses the doubly linked list in the backward direction and prints its elements.
- Parameters:
  list: Pointer to the doubly linked list.
- Example:
  doublylinkedlist_traverse_backward(list);

Remove: Removes the node at the specified index from the Doubly Linked List.

void doublylinkedlist_remove_at(DoublyLinkedList* list, size_t index)

- Description: Removes the node at the specified index from the Doubly Linked List. If the index is out of bounds, it prints an error message and returns.
- Parameters:
  list: Pointer to the Doubly Linked List from which a node will be removed.

index: Index of the node to be removed.

- Example:
  doublylinkedlist_remove_at(list, 1);

Insert at Index: Inserts a new node with the given data at the specified index in the Doubly Linked List.

doublylinkedlist_insert_at

- Syntax:
- void doublylinkedlist_insert_at(DoublyLinkedList* list, int data, size_t index);
- Description: This function inserts a new node with the given data at the specified index in the Doubly Linked List.
- Parameters:
  list: Pointer to the Doubly Linked List where the node will be inserted.
  data: The data to be stored in the new node.
  index: The index at which the new node will be inserted.
- Example:
  doublylinkedlist_insert_at(list, 15, 1);

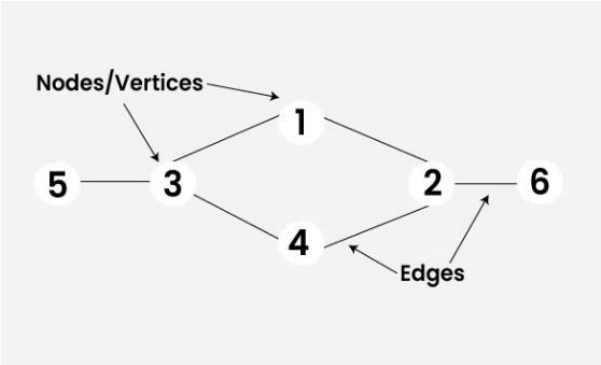
Get at Index: Retrieves the data at the specified index in the Doubly Linked List.

doublylinkedlist_get_at

- Syntax:
- int doublylinkedlist_get_at(const DoublyLinkedList* list, size_t index);
- Description: This function retrieves the data at the specified index in the Doubly Linked List.
- Parameters:
  list: Pointer to the Doubly Linked List.
  index: The index from which to retrieve the data.
- Example:
  int data = doublylinkedlist_get_at(list, 1);

9. Graph

A Graph is a collection of nodes (vertices) connected by edges. It can represent relationships between objects and is used in various applications such as social networks and routing algorithms. Here are its operations:



Create: Creates a new Graph with specified no. of vertices.

graph_create

- Syntax: Graph* graph_create(int num_vertices)
- Description: Creates a new graph with a specified number of vertices.
- Parameters:
  num_vertices: The number of vertices in the graph.
- Example:
  Graph* graph = graph_create(5);

Destroy: Destroys the Graph and frees up memory.

graph_destroy

- Syntax: void graph_destroy(Graph* graph)
- Description: Destroys the graph and frees up memory.
- Parameters:
  graph: Pointer to the graph to be destroyed.
- Example:
  graph_destroy(graph);

Add Vertex: Adds a new vertex to the Graph.

graph_add_vertex

- Syntax: void graph_add_vertex(Graph* graph)
- Description: This function adds a new vertex to the graph by increasing the number of vertices and resizing the adjacency list to accommodate the new vertex.
- Parameters:
  graph: Pointer to the graph structure where the vertex will be added.
- Example:
  Graph* myGraph = graph_create(5); // Create a graph with 5 vertices
  graph_add_vertex(myGraph); // Add a new vertex to the graph

Add Edge: Adds a new edge connecting two vertices in the Graph.

graph_add_edge

- Syntax: void graph_add_edge(Graph* graph, int src, int dest)
- Description: Adds an edge between two vertices in the graph.
- Parameters:
  graph: Pointer to the graph.
  src: Source vertex of the edge.
  dest: Destination vertex of the edge.
- Example:
  graph_add_edge(graph, 0, 1);

Traversal (BFS, DFS): Performs traversal of the Graph to visit all vertices in a certain order (e.g., breadth-first search (BFS) or depth-first search (DFS)).

graph_bfs_traversal

- Syntax: void graph_bfs_traversal(Graph* graph, int start_vertex)
- Description: Performs a breadth-first search traversal of the graph starting from the specified vertex.
- Parameters:
  graph: Pointer to the graph structure.
  start_vertex: The vertex from which the traversal starts.
- Example:
  graph_bfs_traversal(graph, 0);

graph_dfs_traversal

- Syntax: void graph_dfs_traversal(Graph* graph, int start_vertex)
- Description: Performs a depth-first search traversal of the graph starting from the specified vertex.
- Parameters:
  graph: Pointer to the graph structure.
  start_vertex: The vertex from which the traversal starts.
- Example:
  graph_dfs_traversal(graph, 0);