Data Structures & Algorithms

Fall 2023-2024

Instructor: Eng. Dania Al-Said

Teacher Assistant: Eng. Lina Al-Badawi

Student: Hasan Abu-Freiha - 22120021

Contents

Part 1

Exploring Data Structures

Arrays & Array Lists

Arrays and Array-Lists are fundamental data structures in programming, often used to store collections of elements. They have similarities in their use but differ significantly in their implementation and operation. Let's explore each of these data structures in detail.

Array

An array is a basic data structure available in most programming languages. It is a collection of elements, all the same type, stored in contiguous memory locations.

Characteristics:

Fixed Size: Once declared, the size of an array cannot be changed. Memory allocation is static and done at compile time.

Homogeneous Elements:

All elements in an array must be of the same data type.

Random Access:

Arrays provide constant-time (O(1)) access to elements using an index.

Efficiency:

Due to continuous memory allocation, arrays are very efficient in terms of memory and performance, especially for fixed-size collections.

Memory Allocation:

Being a static data structure, arrays can sometimes lead to wasted memory space if not fully utilized or, conversely, to insufficient space for data growth.

Example Use-Cases:

Storing a fixed number of elements of the same type, like days of the week.

For algorithms that require fast access to elements, like binary search.

Array-List

Array-List, also known as a dynamic array or resizable array, is a more flexible version of the traditional array. It's available in many high-level languages like Java and C# as part of standard libraries.

Characteristics:

Dynamic Resizing:

Unlike arrays, Array-Lists can change their size during runtime. Elements can be added or removed, and the data structure will resize itself.

Homogeneous Elements:

Like arrays, Array-Lists typically store elements of the same type. However, in some languages (like Java), one can use Array-Lists to store objects of different types (thanks to polymorphism).

Random Access:

Array-Lists still provide constant-time access to elements.

Additional Overhead:

Due to their dynamic nature, Array-Lists incur additional memory and performance overhead. This is due to the underlying mechanisms used to resize the collection.

Convenience Methods:

Array-Lists often come with built-in methods for common tasks like adding, removing, and searching elements, making them more convenient to use than raw arrays.

Example Use-Cases:

When the number of elements is not known in advance or is likely to change frequently.

For applications that require an array-like structure with more flexibility and convenience, like managing a list of users in an application.

Key Differences

Size Flexibility:

Arrays are fixed in size, while Array-Lists can dynamically grow or shrink.

Performance:

Arrays offer better performance for direct access and manipulation, but Array-Lists provide more flexibility and ease of use with built-in methods.

Memory Usage:

Arrays can be more memory-efficient for fixed-size collections, while Array-Lists can lead to overhead due to resizing operations.

Linked Lists

Linked lists are fundamental data structures used extensively in computer programming. They are collections of elements, called nodes, each linking to the next node in the sequence. This structure allows for efficient insertion and deletion of elements from any position in the list, making them a versatile alternative to arrays.

Types of Linked Lists

Singly Linked List:

Each node contains data and a reference (or pointer) to the next node in the list. The last node points to a null reference, indicating the end of the list.

Doubly Linked List:

Nodes contain two references (or pointers): one to the next node and another to the previous node. This allows traversal in both directions but requires extra memory for the additional reference.

Circular Linked List:

Like a singly or doubly linked list, but the last node points back to the first node, forming a circle.

Characteristics

Dynamic Size:

Unlike arrays, linked lists are dynamic in nature and can grow or shrink in size as needed.

Efficient Insertions and Deletions:

Adding or removing nodes is generally more efficient than in an array, as it doesn't require shifting elements.

Sequential Access:

Unlike arrays, linked lists do not provide direct access to their elements. Traversal is required to access a particular node, making their access time linear (O(n)).

Memory Utilization:

Each node in a linked list requires extra memory for storing the reference to the next (and possibly previous) node, in addition to the data.

No Memory Wastage:

Linked lists efficiently utilize memory as they allocate memory as needed, unlike arrays which may allocate more memory than required.

Operations

Traversal:

Accessing each node sequentially to find a node or to perform an operation on each node.

Insertion:

Nodes can be added anywhere in the list. Common operations include inserting at the beginning, end, or after/before a given node.

Deletion:

Nodes can be removed from any position in the list with proper adjustment of the adjacent node references.

Searching:

Finding a node with a given value requires traversal from the beginning of the list until the node is found or the end is reached.

Use Cases

Ideal for applications where frequent insertion and deletion are required.

Used in implementing stacks, queues, and other abstract data structures.

Forms the basis for more complex data structures like hash tables, adjacency lists in graphs.

Queue

The queue is a fundamental data structure used in computer science and programming. It operates on the principle of "First-In, First-Out" (FIFO), much like a line at a ticket counter where the first person in line is the first to be served. This structure is used in various areas of computing, including operating systems, networking, and algorithm design.

Characteristics of a Queue:

FIFO Order:

Elements are processed in the order they arrive. The first element added to the queue will be the first one to be removed.

Dynamic Size:

Queues can grow and shrink as elements are enqueued and dequeued.

Efficient Operations:

Enqueue and dequeue operations are typically O(1), meaning they have a constant time complexity.

Operations:

Enqueue: Adds an element to the back of the queue.

Dequeue: Removes an element from the front of the queue.

Peek/Front: Returns the element at the front of the queue without removing it.

Types of Queues:

Simple Queue: Standard FIFO implementation.

Circular Queue:

Utilizes a circular structure, where the last position is connected back to the first. This is efficient in terms of memory utilization, especially in a fixed-size array implementation.

Priority Queue:

Elements are ordered based on priority instead of arrival order. The element with the highest priority is served first.

Double-Ended Queue (Deque):

Allows insertions and deletions from both ends of the queue.

Implementation:

Queues can be implemented using various underlying data structures:

Arrays/Lists: Simple and straightforward but may require shifting elements when elements are dequeued.

Linked List: More efficient as it avoids shifting elements. Enqueueing is done at the tail, and dequeuing is done at the head.

Applications of Queues:

Order Processing Systems: In any scenario where requests need to be handled in the order they arrive.

Resource Scheduling: In operating systems for managing processes and task scheduling based on order.

Buffering: In applications where a buffer of requests or data packets is maintained, like in printers or network request handling.

Breadth-First Search (BFS): In graph algorithms, a queue is used to hold vertices or nodes for processing.

Stack

The stack is a fundamental data structure in computer science, used extensively in programming and system implementation. It operates on the principle of "Last-In, First-Out" (LIFO), much like a stack of plates where the last plate put on the stack is the first one to be taken off.

Characteristics of a Stack:

LIFO Order: The last element added to the stack is the first to be removed. This property is the defining characteristic of a stack.

Dynamic Size: The stack can grow and shrink dynamically with push and pop operations.

Efficient Operations: Pushing and popping are typically O(1) operations, providing fast and efficient access to the top element.

Operations:

Push: Adds an element to the top of the stack.

Pop: Removes the element from the top of the stack.

Peek/Top: Returns the element at the top of the stack without removing it.

Implementation:

Stacks can be implemented using various underlying data structures, each with its trade-offs:

Array-Based: Fixed-size or dynamic arrays can be used to implement stacks. Array-based stacks provide fast access but might require resizing if the stack grows beyond the array size.

Linked List: A more flexible implementation that grows dynamically as elements are added. Each element (node) contains the data and a reference to the next element in the stack.

Applications of Stacks:

Function Calls/Recursion: Stacks are used to keep track of function calls in programming languages. Each function call pushes a new frame onto the stack, and returning from a function pops the frame.

Undo Mechanisms: Many applications use stacks to keep track of operations or changes, allowing for an undo feature.

Syntax Parsing: Compilers use stacks for syntax checking and parsing expressions (e.g., checking balanced parentheses).

Backtracking Algorithms: In algorithms where you need to backtrack (like depth-first search), a stack can keep track of the path taken.

Key Advantages:

Simplicity: Stacks are simple to understand and implement.

Memory Efficiency: They are memory-efficient, as there is no need to allocate more memory than necessary.

Data Access: Provides fast access to the last-added element, which is useful in numerous algorithms and systems.

Data structures Implementation

Array

```c
int arrayOfIntegers[1337];
arrayOfIntegers[4] = 5;
```

Array-List

```c
/*---{ Array list }---*/
typedef struct ArrayList{
    int count;
    int numberOfElementsInserted;
    int* array;
    void (*insert)(struct ArrayList* self, int data);
    void (*insertAt)(struct ArrayList* self, int index, int data);
    void (*remove)(struct ArrayList* self, int index);
    int (*fetch)(struct ArrayList* self, int index, int def);
    void (*del)(struct ArrayList* self);
}ArrayList, *pArrayList;
void __resize(pArrayList self, int newSize) {
    if (self == NULL || newSize < 0) return;
    int *newArray = realloc(self->array, newSize * sizeof(int));
    if (newArray == NULL) return;
    if (newSize > self->count) {
        for (int i = self->count; i < newSize; i++) {
            newArray[i] = INT_MIN;
        }
    }
    self->array = newArray;
    self->count = newSize;
}
void __insert(pArrayList self, int data){
    for(int i = 0; i < self->count; i++){
        if(self->array[i] == INT_MIN) {
            self->array[i] = data;
            self->numberOfElementsInserted++;
            return;
        }
    }
    int oldCount = self->count;
    __resize(self, self->count*2);
    self->numberOfElementsInserted++;
    self->array[oldCount] = data;
}
void __insertAt(pArrayList self, int index, int data) {
    if (self == NULL || index < 0 || index >= self->count) return;
    if (self->array[index] == INT_MIN) {
        self->numberOfElementsInserted++;
    }
    self->array[index] = data;
}
void __remove(pArrayList self, int index){
    if(self == NULL || index < 0 || index >= self->count) return;
    for (int i = index; i < self->count - 1; i++) {
        self->array[i] = self->array[i + 1];
    }
    self->array[self->count - 1] = INT_MIN;
    self->numberOfElementsInserted--;
    if(self->numberOfElementsInserted < (self->count/2)+1) __resize(self, (self->count/2)+1);
}
int __fetch(pArrayList self, int index, int def){
    if(self == NULL || index < 0 || index >= self->count) return def;
    return self->array[index];
}
void __del(pArrayList self){
    free((void*)self->array);
    free((void*)self);
}
pArrayList newArrayList(){
    pArrayList new = (pArrayList)malloc(sizeof(*new));
    new->count = 10;
    new->numberOfElementsInserted = 0;
    new->array = (int*)malloc(new->count*sizeof(int));
    for(int i = 0; i < new->count; i++) new->array[i] = INT_MIN;
    new->insert = &__insert;
    new->insertAt = &__insertAt;
    new->remove = &__remove;
    new->fetch = &__fetch;
    new->del = &__del;
    return new;
}
```

Singly Linked List (Simple)

```c
/*---{ Singly Linked List }---*/
typedef struct Node{
    struct Node* next;
    int data;
}Node, *pNode;
pNode newNode(int data){
    pNode _newNode = (pNode)malloc(sizeof(*_newNode));
    _newNode->next = NULL;
    _newNode->data = data;
    return _newNode;
}
typedef struct LinkedList{
    pNode head;
    pNode tail;
    int count;
    void (*insert)(struct LinkedList* self, int data);
    void (*remove)(struct LinkedList* self, int index);
    void (*traverse)(struct LinkedList* self);
    int (*search)(struct LinkedList* self, int index, int def);
}LinkedList, *pLinkedList;

void __insert(pLinkedList self, int data){
    pNode toBeAdded = newNode(data);
    if(self->head == NULL){
        self->head = toBeAdded;
        self->tail = toBeAdded;
        self->count++;
        return;
    }
    self->tail->next = toBeAdded;
    self->tail = toBeAdded;
    self->count++;
}
void __remove(pLinkedList self, int index){
    if (self == NULL || self->head == NULL || index < 0 || index >= self->count) return;

    if (index == 0) {
        pNode temp = self->head;
        self->head = self->head->next;
        if (self->tail == temp) {
            self->tail = NULL;
        }
        free(temp);
        self->count--;
        return;
    }

    pNode current = self->head;
    for (int i = 0; i < index - 1; i++) {
        current = current->next;
    }

    pNode toBeRemoved = current->next;
    current->next = toBeRemoved->next;


    if (self->tail == toBeRemoved) {
        self->tail = current;
    }

    free(toBeRemoved);
    self->count--;
}
void __traverse(pLinkedList self){
    for(pNode current = self->head; current != NULL; current=current->next){
        printf("%d ", current->data);
    }
}
int __search(pLinkedList self, int index, int def){
    if (self == NULL || self->head == NULL || index < 0 || index >= self->count) return def;
    pNode toBeFound = self->head;
    for(int count = 0; count < index; count++){
        toBeFound = toBeFound->next;
    }
    return toBeFound->data;
}
pLinkedList newLinkedList(){
    pLinkedList _newLinkedList = (pLinkedList)malloc(sizeof(*_newLinkedList));
    _newLinkedList->head = NULL;
    _newLinkedList->tail = NULL;
    _newLinkedList->count = 0;
    _newLinkedList->insert = &__insert;
    _newLinkedList->remove = &__remove;
    _newLinkedList->traverse = &__traverse;
    _newLinkedList->search = &__search;
    return _newLinkedList;
}
```

Queue

```c
/*---{ Queue }---*/
typedef struct Node {
    struct Node *next;
    int data;
} Node, *pNode;

pNode newNode(int data) {
    pNode newNode = (pNode) malloc(sizeof(*newNode));
    newNode->next = NULL;
    newNode->data = data;
    return newNode;
}

typedef struct Queue {
    pNode head;
    pNode tail;
    int count;

    void (*enqueue)(struct Queue *self, int data);

    int (*dequeue)(struct Queue *self);

    void (*traverse)(struct Queue *self);

    int (*search)(struct Queue *self, int index, int def);
} *pQueue;

void __enqueue(pQueue self, int data) {
    pNode toBeAdded = newNode(data);
    if (self->head == NULL) {
        self->head = toBeAdded;
        self->tail = toBeAdded;
        self->count++;
        return;
    }
    self->tail->next = toBeAdded;
    self->tail = toBeAdded;
    self->count++;
}

int __dequeue(pQueue self) {
    if (self == NULL || self->head == NULL) return INT_MIN;

    pNode temp = self->head;
    self->head = self->head->next;
    if (self->tail == temp) {
        self->tail = NULL;
    }
    int toBeReturned = temp->data;
    free(temp);
    self->count--;
    return toBeReturned;
}

void __traverse(pQueue self) {
    for (pNode current = self->head; current != NULL; current = current->next) {
        printf("%d ", current->data);
    }
}

int __search(pQueue self, int index, int def) {
    if (self == NULL || self->head == NULL || index < 0 || index >= self->count) return def;

    pNode toBeFound = self->head;
    for (int count = 0; count < index; count++) {
        toBeFound = toBeFound->next;
    }
    return toBeFound->data;
}

pQueue newQueue() {
    pQueue _newQueue = (pQueue) malloc(sizeof(*_newQueue));
    _newQueue->head = NULL;
    _newQueue->tail = NULL;
    _newQueue->count = 0;

    _newQueue->enqueue = &__enqueue;
    _newQueue->dequeue = &__dequeue;
    _newQueue->traverse = &__traverse;
    _newQueue->search = &__search;

    return _newQueue;
}
```

Stack

```c
/*---{ Stack }---*/
typedef struct Node {
    struct Node *next;
    int data;
} Node, *pNode;

pNode newNode(int data) {
    pNode newNode = (pNode) malloc(sizeof(*newNode));
    newNode->next = NULL;
    newNode->data = data;
    return newNode;
}

typedef struct Stack {
    pNode head;
    int count;

    void (*push)(struct Stack *self, int data);

    int (*pop)(struct Stack *self);

    void (*traverse)(struct Stack *self);
} *pStack;

void __push(pStack self, int data) {
    pNode toBeAdded = newNode(data);
    if (self->head == NULL) {
        self->head = toBeAdded;
        self->count++;
        return;
    }
    toBeAdded->next = self->head;
    self->head = toBeAdded;
    self->count++;
}

int __pop(pStack self) {
    if (self == NULL || self->head == NULL) return INT_MIN;

    pNode temp = self->head;
    self->head = self->head->next;
    int toBeReturned = temp->data;
    free(temp);
    self->count--;
    return toBeReturned;
}

void __traverse(pStack self) {
    for (pNode current = self->head; current != NULL; current = current->next) {
        printf("%d ", current->data);
    }
}


pStack newStack() {
    pStack _newStack = (pStack) malloc(sizeof(*_newStack));
    _newStack->head = NULL;
    _newStack->count = 0;

    _newStack->push = &__push;
    _newStack->pop = &__pop;
    _newStack->traverse = &__traverse;

    return _newStack;
}
```

Design Specification

Queue

A data Structure Based on a First-In-First-Out (FIFO) concept, where the first item inserted is the first to get removed.

Core Definitions:

Queue Entry (Node), contains:

Data – contains the data held in each node.

Next pointer – Holds the location of the next node in the queue.

Previous pointer – Holds the location of the previous node in the queue.

Queue (Main Structure), contains:

Count – Holds the number of elements (Nodes) in the queue.

Head pointer – Points to the first node in the queue.

Tail pointer – Points to the last node in the queue.

Queue Iterator (Helper) – Used to iterate the queue, contains:

Current – Points to the current node in holding.

Next – A function used to get the node after the one in holding.

Del – A function to free the object from memory.

Queue Implementation (Wrapper for the queue) – Used to encapsulate the queue structure:

DumpQueue – A function to print diagnostic data for the queue.

DelQueue – A function Free the queue object from memory.

iterQueue – A function to iterate the queue.

updateData – A function to update/add data at a given index.

getQueue – A function to get the data at a given index.

getHead – A function to get the data in the first node of the queue.

getHeadIndex – A helper function to retrieve the index of the first node in the queue.

putHead – A function to add a node at the beginning of the queue.

getTail – A function to retrieve the data in the last node of the queue.

enqueueData – A function to enqueue data to the queue.

dequeueData – A function to dequeue data from the queue.

deleteData – A function to delete the data at a given index in the queue.

sizeQueue – A function that returns the number of elements in the queue.

Main functions and their time complexity:

DumpQueue – O(n)

DelQueue – O(n)

iterQueue – O(n)

updateData – O(n)

getQueue – O(n)

getHead – O(1)

getHeadIndex – O(1)

putHead – O(1).

getTail – O(1)

enqueueData – O(1)

dequeueData – O(1)

deleteData – O(n)

sizeQueue – O(1)

Private functions (Helper functions)

__newQueueEntry: A constructor for new nodes.

__newQueueIter: A constructor for the queue iterator.

newQueue: A constructor the queue structure.

__QueueFind: A function to find a node at a given index.

__updateDataOrdered:

A helper function that adds a node to its respected place depending on the node's index.

Queue Operations Explained

DumpQueue (__QueueDump) – Prints out diagnostic data about the queue:

```
void __QueueDump(pQueue self){
    if(self == NULL){
        return;
    }
    printf("Nodes count: %d\n", self->Queue->__count);
    int i = 0;
    for(pQueueEntry cur = self->Queue->__head; cur != NULL; cur = cur->__next, i++){
        printf("Node 0x%p\n\t\\___ Index: %d\n\t\\___ Data: %s\n", cur, i, cur->data);
    }
}
```

This function provides a diagnostic tool to visualize the current state of a queue data structure. It is primarily used for debugging and verification purposes to ensure that the queue is behaving as expected.

Parameters:

self: A pointer to the queue object (pQueue) whose contents are to be displayed.

Functionality:

Null Check: The function first checks if the provided queue pointer is NULL. If it is, the function returns immediately, ensuring that it doesn't attempt to access a null pointer.

Display Queue Size: It prints the total number of nodes (__count) in the queue. This gives an overview of how many elements are currently stored in the queue.

Iterate and Display Nodes:

The function iterates through each node in the queue, starting from the head (__head) and following the next pointers (__next) until it reaches the end of the queue (indicated by NULL).

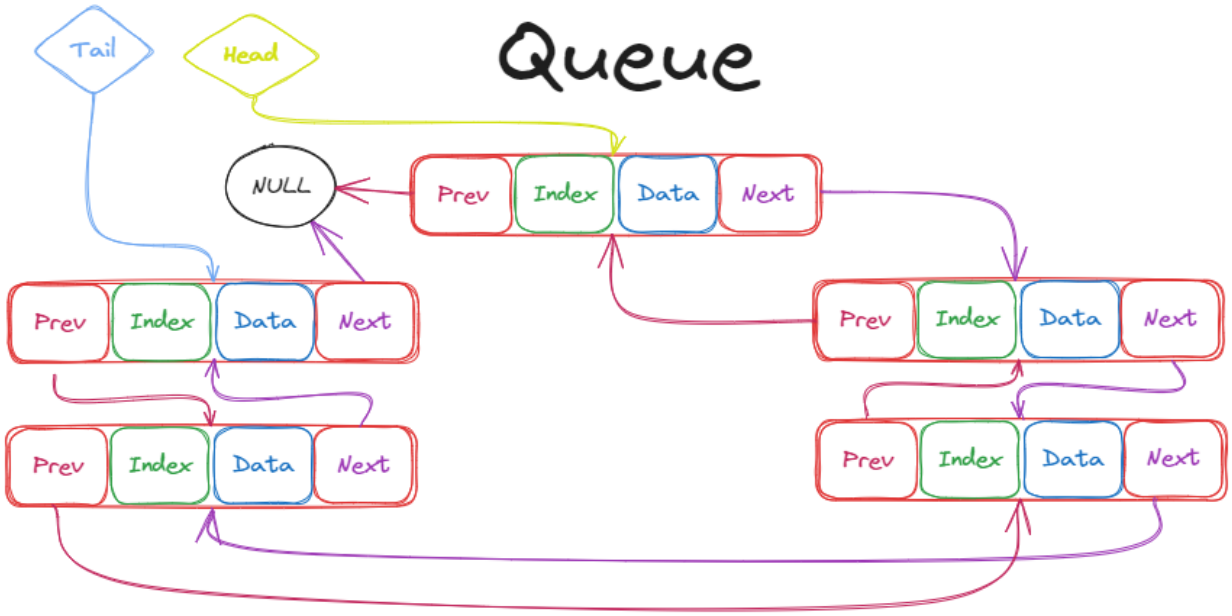
For each node, it prints:

The memory address of the node, providing a reference to its location in memory.

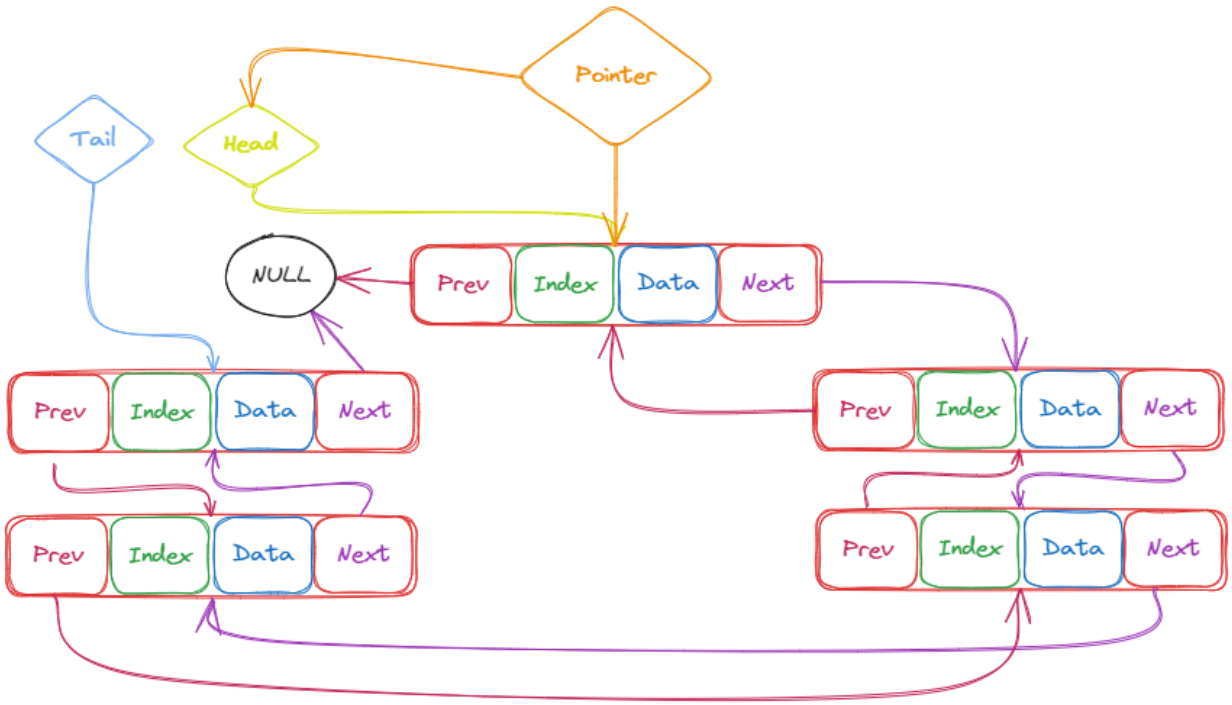
The index of the node.

The data stored in the node, which is expected to be a string.

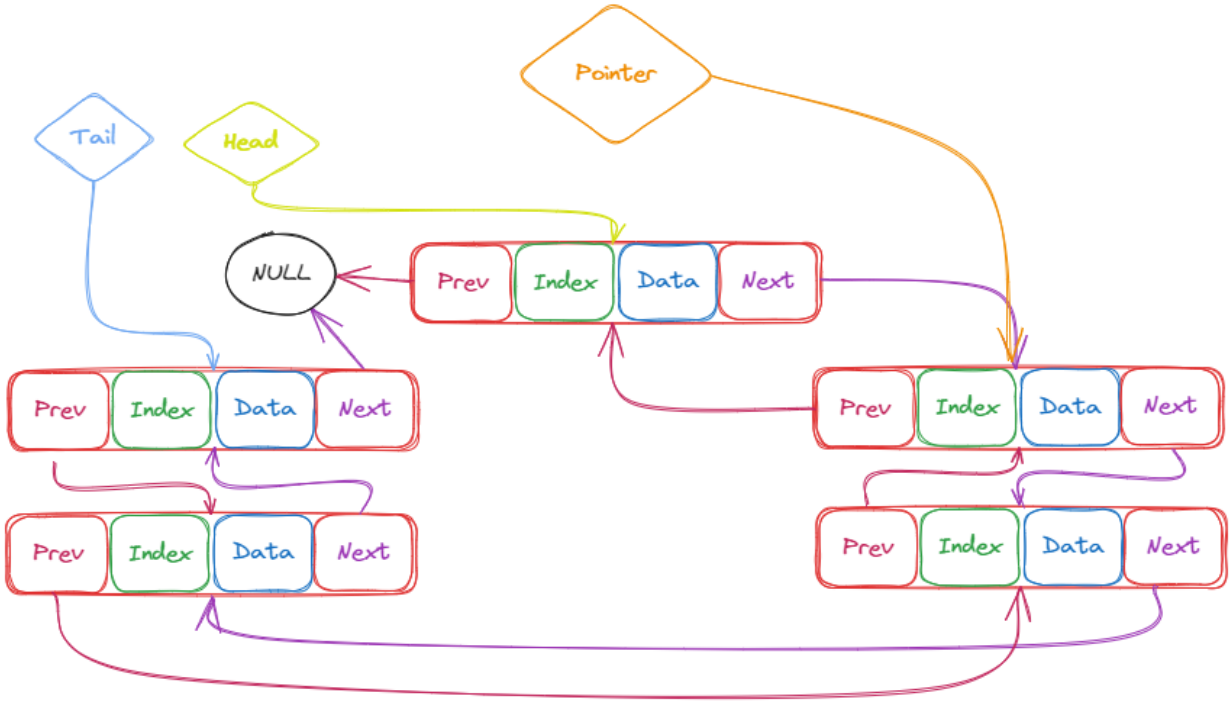
DumpQueue Illustration:



Making a pointer to the head of the queue, which will point to the first node in the queue:



Traversing the queue by printing out the current node's information then going to the next node.



Example output:

```
Nodes count: 5
Node 0x00000182c85c1630
        \___ Index: 0
        \___ Data: this is a test
Node 0x00000182c85c17c0
        \___ Index: 1
        \___ Data: this is a test x2
<SNIP>
```

DelQueue (__delQueue) – Frees the queue out of memory:

```
void __delQueue(pQueue self){
    DEBUG_INFO("Entered __delQueue function\n");
    pQueueEntry current, next;
    current = self->Queue->__head;
    DEBUG_OKAY("Got a pointer to the head\n\t\\___ Deleting nodes...\n");
    while(current != NULL){
        next = current->__next;
        free((void*) current->data);
        free((void*)current);
        current = next;
    }
    DEBUG_OKAY("Deleted the nodes\n\t\\___ Deleting the queue object.\n");
    free((void*)self);
    self= NULL;
    DEBUG_INFO("Exiting __delQueue function...\n");
}
```

This function is responsible for deallocating the memory occupied by a queue and its constituent elements, ensuring proper memory management, and preventing memory leaks.

Parameters:

self: A pointer to the queue object (pQueue) to be deleted.

Functionality:

Initialization:

Initializes two pointers, current and next, to help traverse the queue.

Sets current to point to the head of the queue (self->Queue->__head).

Traversal and Deallocation:

Iterates over each node in the queue, starting from the head.

For each node (current):

next is set to the next node in the queue (current->__next).

The data associated with the current node is deallocated using free((void*) current->data).

The current node itself is deallocated using free((void*)current).

Moves to the next node by setting current to next.

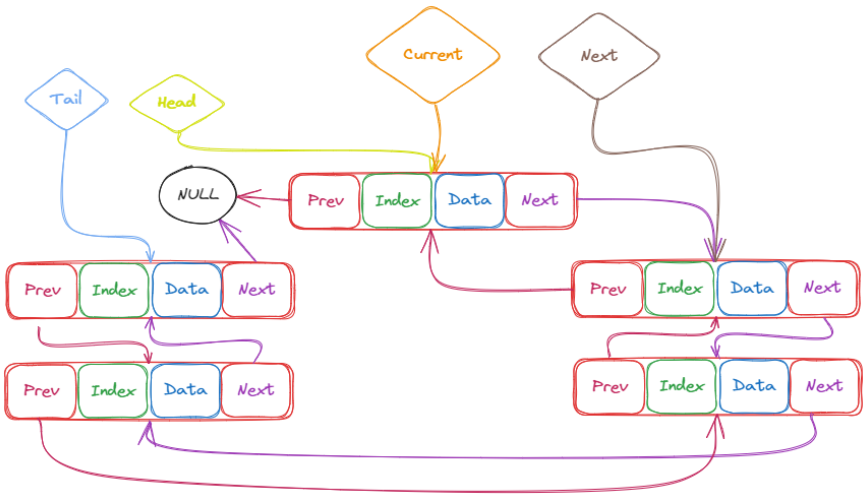
This loop continues until all nodes in the queue have been visited and deallocated.

Queue Object Deallocation:

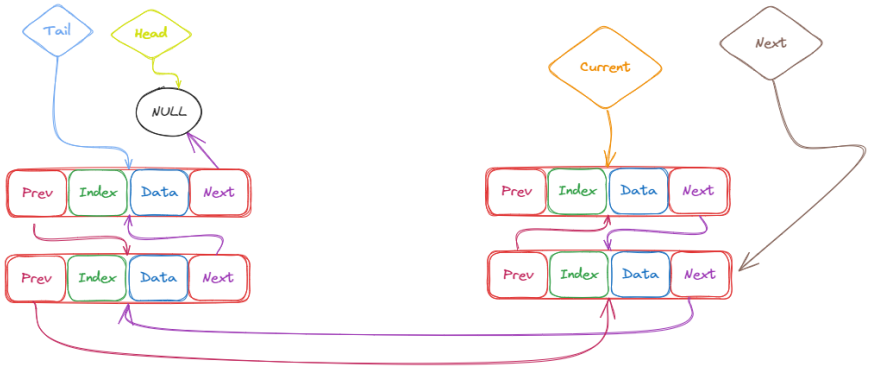
After all nodes have been deallocated, the function deallocates the queue object itself using free((void*)self).

Sets the self pointer to NULL to prevent dangling references.

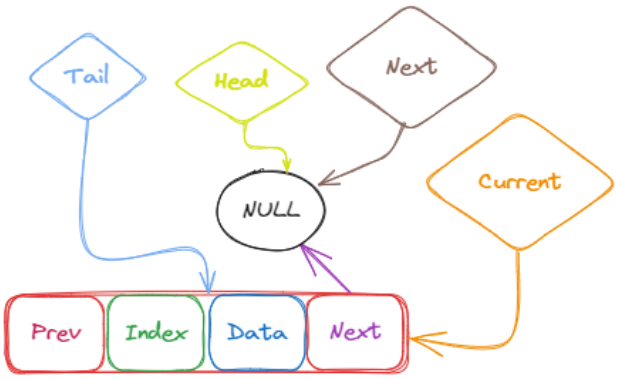
DelQueue Illustration:



After deleting the first node



Upon getting to the last node



Enqueue (__putTail) – Enqueues data at the end of the queue:

```
void __putTail(pQueue self, char* data){
    DEBUG_INFO("Entered __putTail function\n");
    pQueueEntry newEntry = __newQueueEntry(data);
    self->Queue->__count++;
    if(self->Queue->__head == NULL){
        self->Queue->__head = newEntry;
        self->Queue->__tail = newEntry;
        DEBUG_OKAY("The Queue is empty\n\t\\___ Linked the head and tail to the new node\n");
        return ;
    }
    self->Queue->__tail->__next = newEntry;
    newEntry->__prev = self->Queue->__tail;
    self->Queue->__tail = newEntry;
    DEBUG_OKAY("Linked the new node to the tail\n");
    DEBUG_INFO("Exiting the __putTail function...\n");
}
```

This function appends a new element to the end of a queue.

Parameters:

self: A pointer to the queue object (pQueue) where the new element will be added.

data: A pointer to a string (char*) representing the data to be stored in the new queue node.

Functionality:

Node Initialization:

Creates a new queue entry (pQueueEntry) using the given data. This is achieved by calling the __newQueueEntry function.

Increment Node Count:

Increments the __count field of the queue, indicating a new node is being added.

Handling an Empty Queue:

Checks if the queue is empty by verifying if the __head pointer of the queue is NULL.

If the queue is empty, the new node becomes both the head and tail of the queue.

Appending to a Non-Empty Queue:

The index of the new node is set to one more than the index of the current tail node.

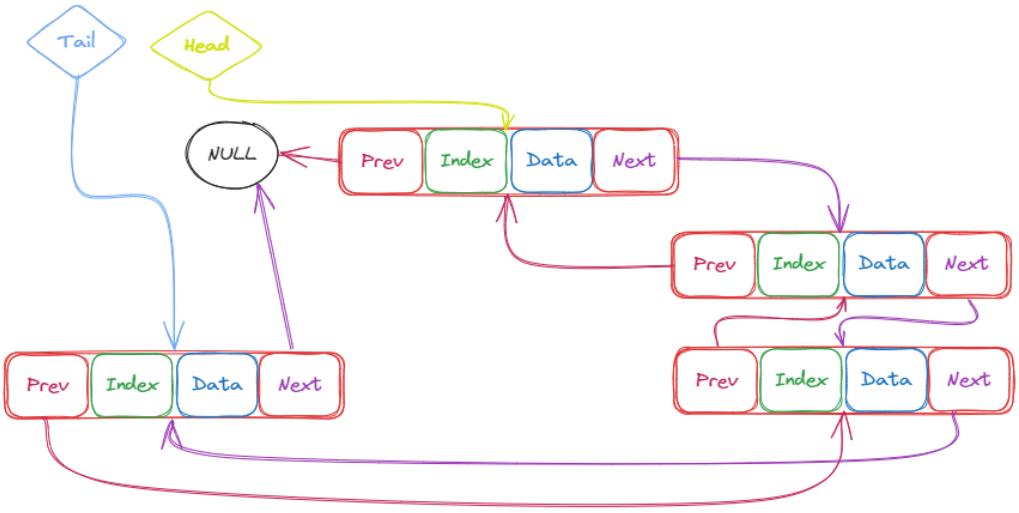
The new node is linked to the current tail node:

The __next pointer of the current tail node is set to the new node.

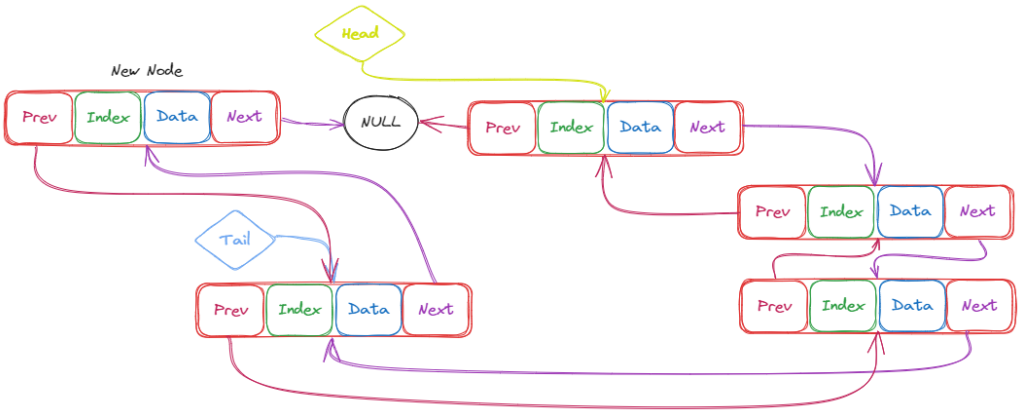
The __prev pointer of the new node is set to the current tail node.

The new node becomes the new tail of the queue.

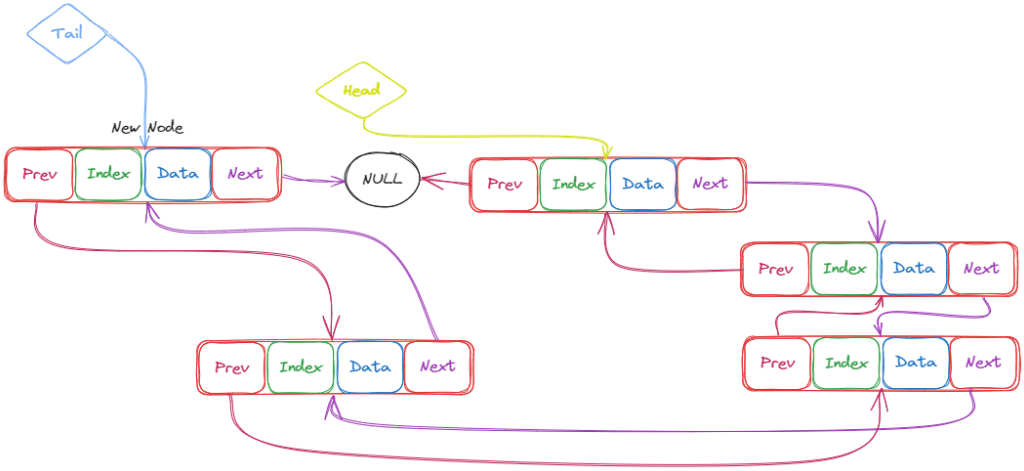
Enqueue Illustration:



Linking the newNode to the Tail of the queue:



Pointing the tail to the new node:



Dequeue (__delHead) – Dequeues the head of the queue

```
char* __delHead(pQueue self, char* def){
    DEBUG_INFO("Entered __delHead function (dequeue)\n");
    if(self->Queue->__head == NULL) {
        DEBUG_ERROR("Queue is empty\n");
        return def;
    }
    char* retData = __getHead(self, def);
    if(strcmp(retData, def) == 0) return def;
    self->Queue->__count--;
    pQueueEntry toBeDeleted = self->Queue->__head;
    if(self->Queue->__head->__next != NULL) {
        self->Queue->__head->__next->__prev = NULL;
        self->Queue->__head = self->Queue->__head->__next;
    } else {
        self->Queue->__head = NULL;
        self->Queue->__tail = NULL;
    }
    free((void*)toBeDeleted);
    DEBUG_OKAY("retrieving the data inside the head...\n");
    DEBUG_INFO("Exiting __delHead (dequeue) function...\n");
    return retData;
}
```

This function removes – Dequeues – the head (front) element from the queue and returns its data.

Parameters:

self: A pointer to the queue object (pQueue) from which the head element will be removed.

def: A pointer to a default string value to be returned if the queue is empty or in case of an error.

Functionality:

Empty Queue Check:

Checks if the queue is empty by verifying if the __head pointer is NULL.

If the queue is empty, returns the default value def.

Retrieve Head Data:

Retrieves the data from the head of the queue by calling the __getHead function.

If the retrieved data is equal to the default value, returns def.

Decrement Node Count:

Decreases the __count field of the queue, indicating a node is being removed.

Handle Node Removal:

Points a temporary toBeDeleted pointer to the current head of the queue.

Adjusts the queue's head pointer to the next node in the queue.

If there's only one node in the queue, it sets both the head and tail pointers to NULL.

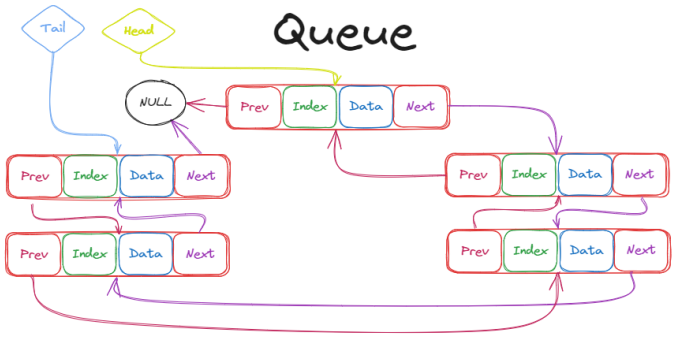
Free Memory:

Frees the memory allocated for the head node that is being removed.

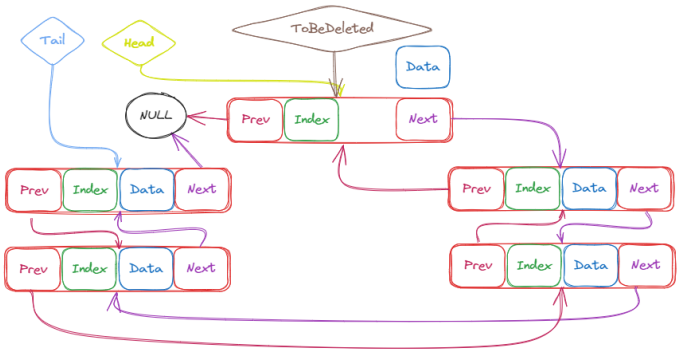
Return Value:

Returns the data (retData) from the deleted head node.

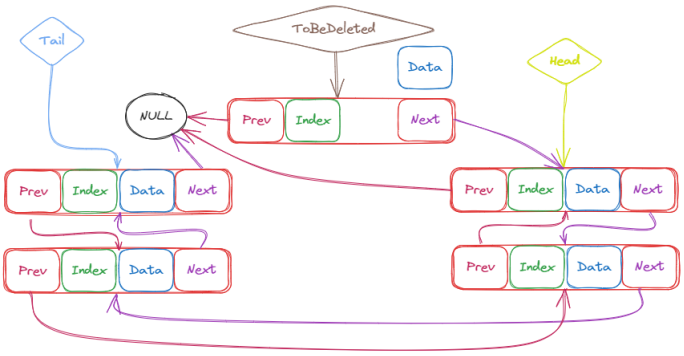
Dequeue Illustration



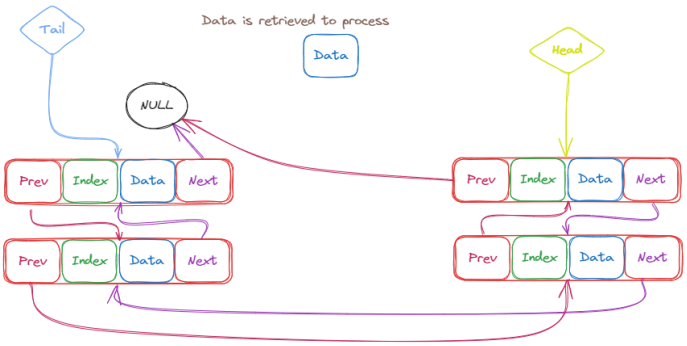
Retrieving the data and setting the “toBeDeleted” pointer to the current head:



Setting the new head to the next node:



Deleting the node and retrieving its data:



Stack

A data structure based on a Last-In-First-Out (LIFO) concept, where the last item inserted will be the first to be removed.

Core Definitions:

Action (Original), contains:

ActionDone: The type of action done (Insert, Remove, Modify)

Index: The index at which the action occurred.

Data: The data which was affected by the action.

Next: A pointer to the next action in the stack.

ActionStack (Original), contains:

Top: A pointer to the last action pushed on the stack.

Capacity: The maximum capacity for the undo stack.

Count: The number of actions pushed on the stack.

Action (Wrapper), contains:

Action: A pointer to the original action class.

GetData: A function to retrieve the data from a given action.

GetIndex: A function to retrieve the index from a given action.

GetAction: A function to retrieve the type of action from a given action.

DelAction: A function to delete (free) the action object from the memory.

ActionStack (Wrapper), contains:

ActionStack: A pointer to the original action stack class.

GetTopIndex: A function to get the index of the last pushed action on the stack.

GetTopData: A function to get the data of the last pushed action on the stack.

GetTopAction: A function to get the action type of the last pushed action on the stack.

Push: A function to push an action on to the stack.

Pop: A function to pop the action at the top of stack and retrieve it.

DelStack: A function to delete (free) the stack object from the memory.

Main Functions and their time complexity:

Action – GetData: O(1)

Action – GetIndex: O(1)

Action – GetAction: O(1)

Action – DelAction: O(1)

ActionStack – GetTopIndex: O(1)

ActionStack – GetTopData: O(1)

ActionStack – GetTopAction: O(1)

ActionStack – Push: O(1)

ActionStack – Pop: O(1)

ActionStack – DelStack: O(n)

Helper Functions:

newAction: A constructor for the action.

newActionStack: A constructor for the ActionStack.

Stack Operations Explained

Push (__push) – Pushes an action onto the action stack:

```
void __push(pActionStack self, pAction doneAction) {
    DEBUG_INFO("Entered __push function\n");
    if(self->__actionStack->__count >= self->__actionStack->__capacity){
        DEBUG_INFO("Action Stack FULL...\n\t\\__ Printing it's contents...\n");
#ifdef DEBUG
        __printActionStack(self);
#endif
        DEBUG_INFO("Freeing the actionStack...\n");
        for(pAction cur = self->pop(self); cur != NULL;cur=self->pop(self)){
            pAction temp = cur;
            free((void*)temp);
        }
        DEBUG_INFO("Printing the stack after freeing it...\n");
#ifdef DEBUG
        __printActionStack(self);
#endif
    }

    if(self->__actionStack->__top == NULL){
        self->__actionStack->__top = doneAction;
        self->__actionStack->__count++;
        return;
    }
    doneAction->action->__next = self->__actionStack->__top;
    self->__actionStack->__top = doneAction;
    self->__actionStack->__count++;
    DEBUG_INFO("Exiting __push function...\n");
}
```

This function pushes an action onto the top of an action stack, effectively adding it as the most recent action in a LIFO (Last-In, First-Out) sequence.

Parameters:

self: A pointer to the action stack object (pActionStack) where the action will be added.

doneAction: A pointer to the action (pAction) to be pushed onto the stack.

Functionality:

Capacity Check:

Checks if the stack has reached its maximum capacity by comparing the current count of actions (__count) with the stack's capacity (__capacity).

If the stack is full, it enters a cleanup mode where all actions in the stack are popped and freed to clear the stack.

Pushing Action to an Empty Stack:

If the stack's top (__top) is NULL (indicating an empty stack), the doneAction becomes the new top of the stack.

Increments the stack's action count (__count).

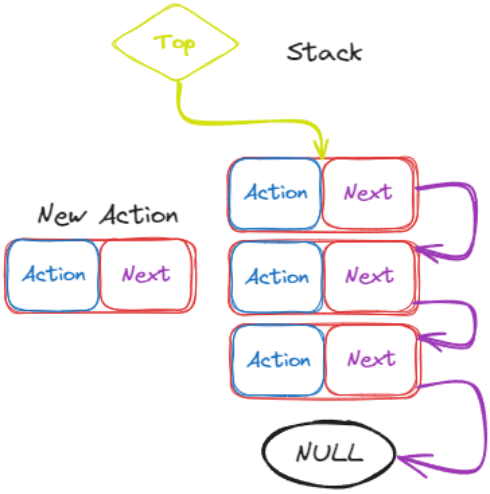
Pushing Action to a Non-Empty Stack:

If the stack is not empty, link the doneAction to the current top of the stack by setting doneAction->action->__next to the current top.

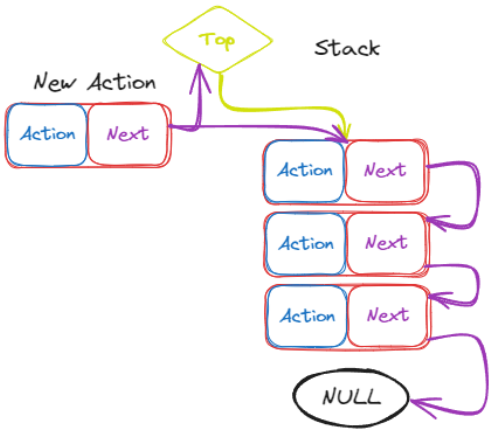
Updates the stack's top to the new doneAction.

Increments the stack's action count.

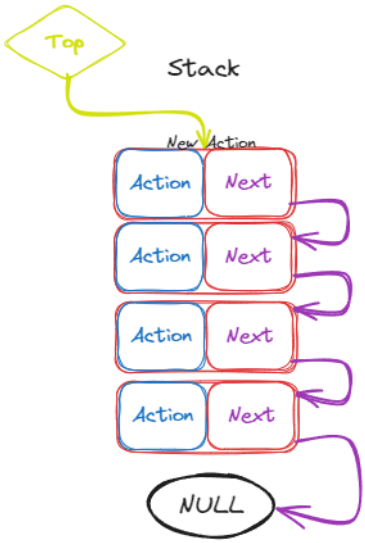
Push Illustration:



Since the stack is not empty, we will be setting the next pointer of the new action to the top node:



Setting the top pointer to our new node:



Pop (__pop) – Pops an action from the action stack:

```
pAction __pop(pActionStack self) {
    DEBUG_INFO("Entered __pop function\n");
    if (self->__actionStack->__top == NULL) return NULL;
    pAction poppedAction = self->__actionStack->__top;
    self->__actionStack->__top = self->__actionStack->__top->action->__next;
    poppedAction->action->__next = NULL;
    self->__actionStack->__count--;
    DEBUG_INFO("Exiting __pop  function...\n");
    return poppedAction;
}
```

This function removes the topmost action from an action stack and returns it. It's the typical pop operation in a LIFO (Last-In, First-Out) stack data structure.

Parameters:

self: A pointer to the action stack object (pActionStack) from which the top action will be popped.

Functionality:

Empty Stack Check:

Checks if the stack is empty by verifying if the __top pointer is NULL.

If the stack is empty, returns NULL, indicating there are no actions to pop.

Pop Action:

Retrieves the top action from the stack (poppedAction) by pointing to the current top action (self->__actionStack->__top).

Updates the stack's top pointer (__top) to the next action in the stack (self->__actionStack->__top->action->__next), effectively removing the top action from the stack.

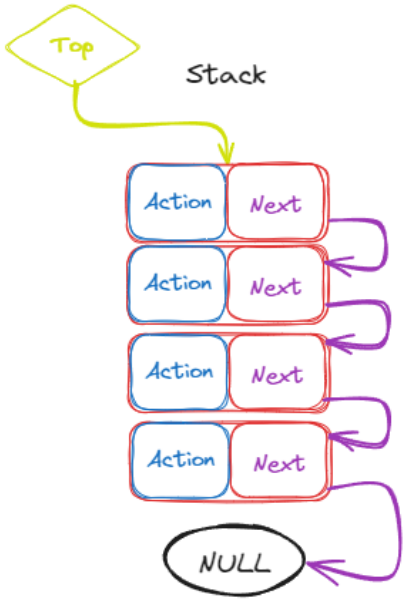
Sets the __next pointer of the popped action to NULL to disconnect it from the stack.

Decrements the action count (__count) of the stack.

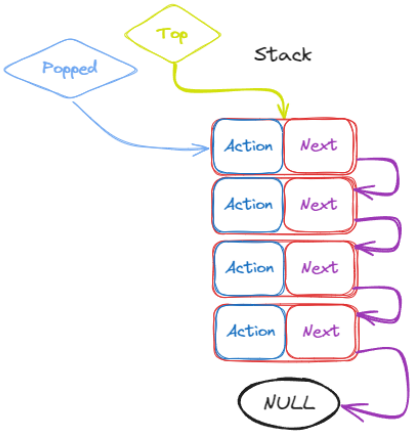
Return Value:

Returns the popped action (poppedAction), which was previously at the top of the stack.

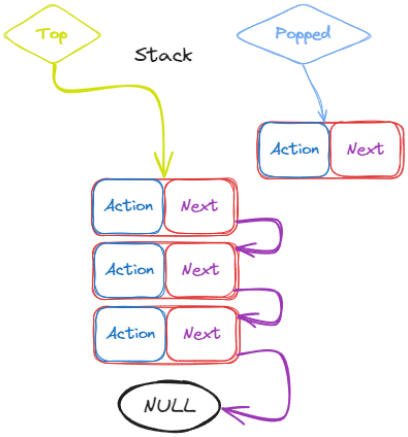
Pop Illustration



Point to the popped action:

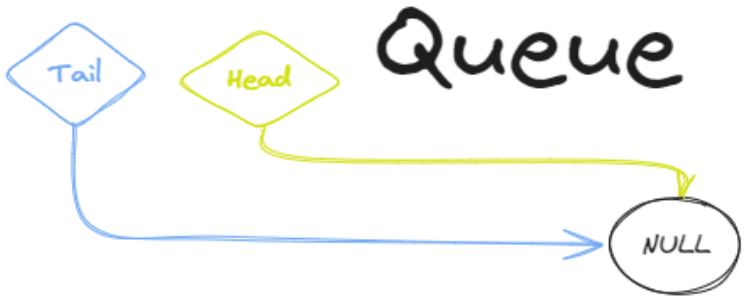


Separate it from the stack and return it:

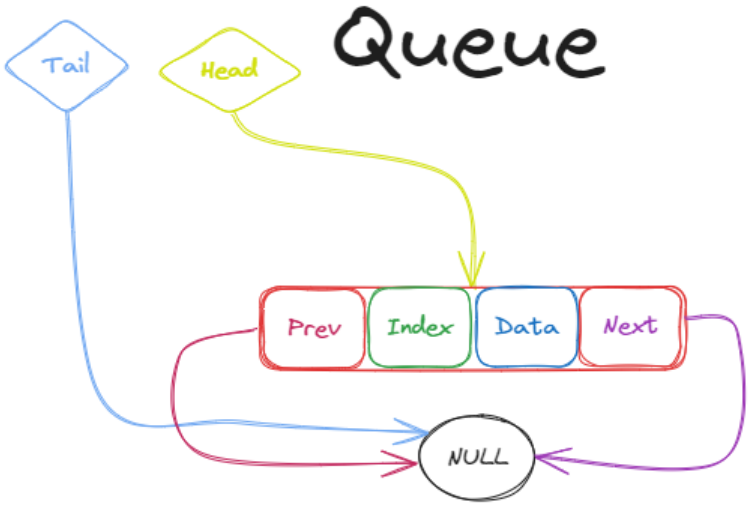


First-In-First-Out Illustration

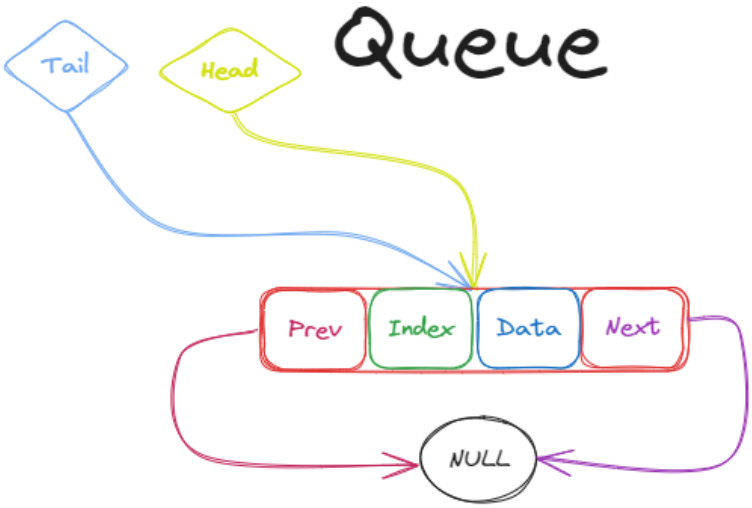
An empty queue:



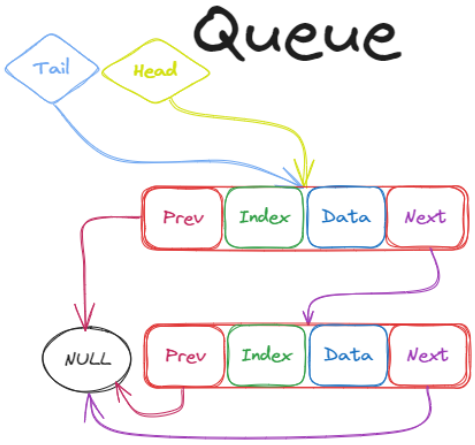
Enqueueing the first node – Linking the head: (The new node is initialized to point to NULL)



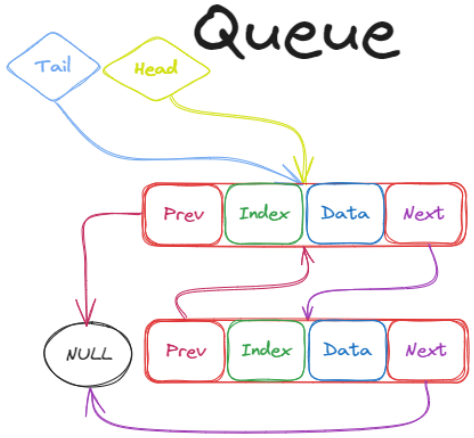
Enqueueing the first node – Linking the tail:



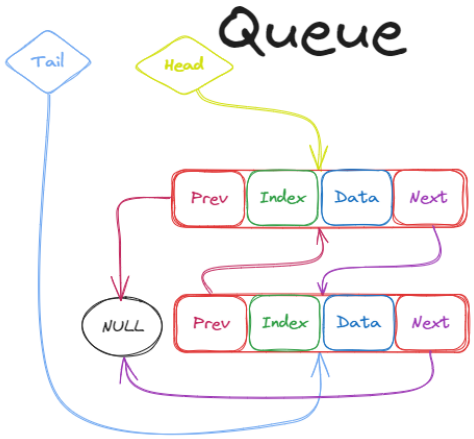
Enqueueing a second node – Linking the “Next” pointer of the tail to the new node:



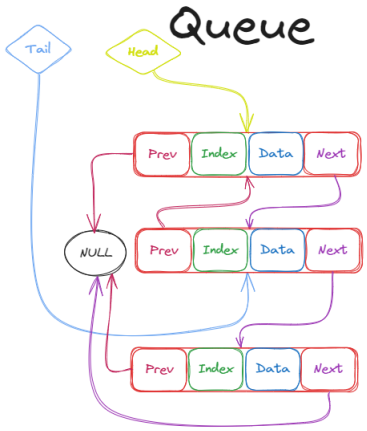
Enqueueing the second node – Linking the “prev” pointer of the new node to the tail:



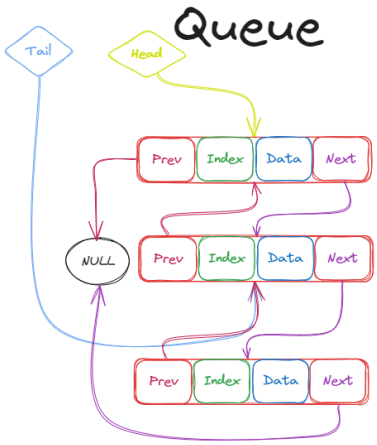
Enqueueing the second node – Linking the Tail pointer to point to the new node:



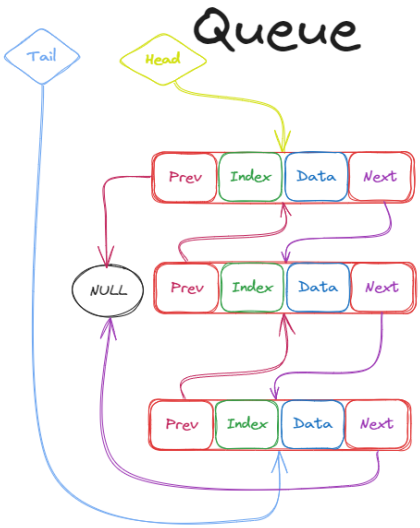
Enqueueing a third node – Linking the “Next” pointer of the tail to the new node:



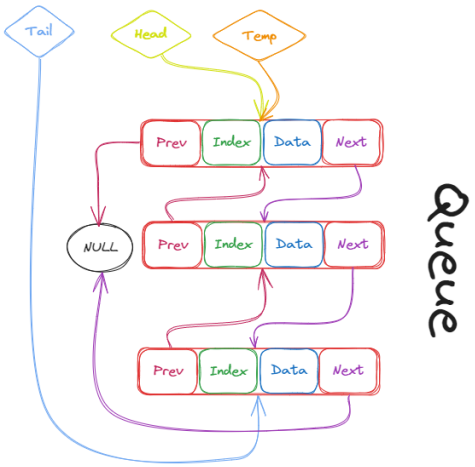
Enqueueing a third node – Linking the “prev” pointer of the new node to the tail:



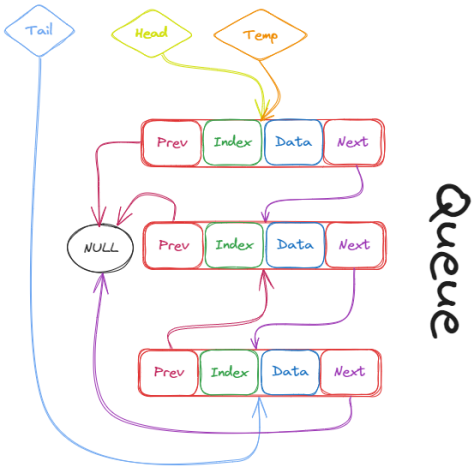
Enqueueing a third node – Linking the “Tail” to the new node:



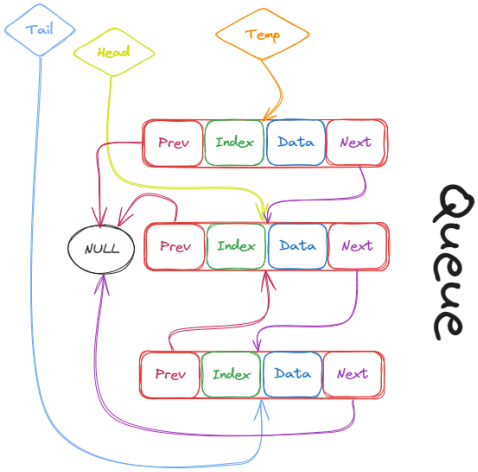
Dequeuing from the queue – Setting a temporary pointer to the head:



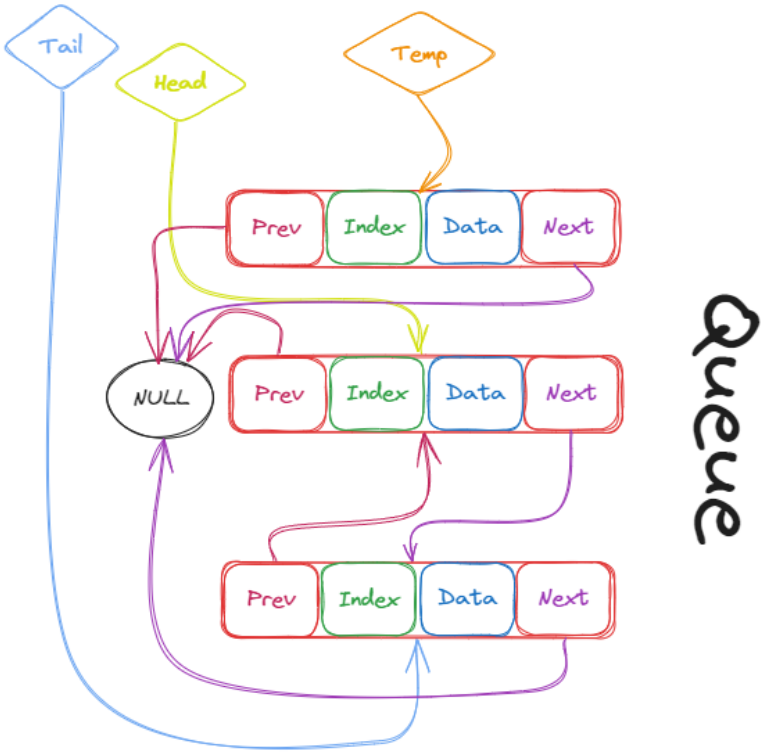
Dequeuing from the queue – Nullifying the “prev” pointer of the next node after the head:



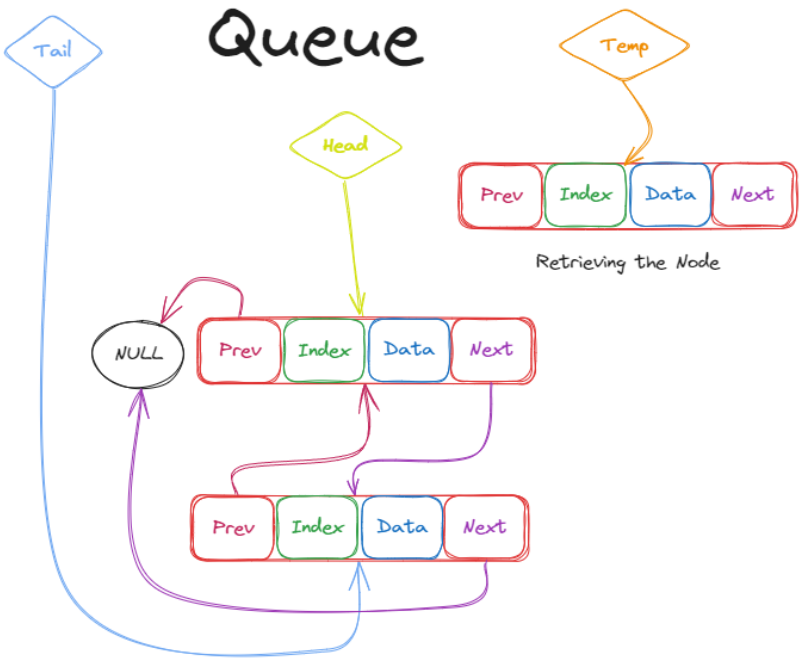
Dequeuing from the queue – Setting the head to the “next” pointer of the temp pointer:



Dequeuing from the queue – Nullifying the “next” pointer of the temp pointer:



Dequeuing from the queue – Retrieving the node pointed at by Temp:



Last-In-First-Out (Stack) Implementation

The abstract data type (ADT) for a software stack is a collection of elements organized according to the Last-In, First-Out (LIFO) principle. This means that the most recently added element is the first one to be removed. In the code provided, the stack ADT is specifically used to manage a sequence of actions, often for undo/redo functionality in software applications.

Characteristics:

- LIFO (Last-In-First-Out) Principle:
  - Elements are added (pushed) and removed (popped) from the top of the stack.
- Single End for Operations:
  - All operations (push, pop) occur at one end of the stack, commonly referred to as the "top" of the stack.
- Size Flexibility:
  - A stack can grow and shrink dynamically as elements are added and removed.
- Peek Operation:
  - Optionally, a stack can allow inspection of the top element without removing it.

Operations:

- Push: Adds an element to the top of the stack.
- Pop: Removes and returns the element from the top of the stack.
- Top/Peek: Returns the element at the top of the stack without removing it (optional).
- IsEmpty: Checks if the stack is empty.
- Size: Returns the number of elements in the stack (optional).

Implementation in the Provided Code:

In the provided code, the stack ADT is implemented using a dynamically allocated linked structure to manage actions, such as undoable commands in an application. Each element in the stack represents an action and contains relevant data for that action.

Action Structure (pAction):

Represents a stack element. Each action has information about the operation performed and a link to the next action in the stack.

Stack Management (pActionStack):

Push (__push):

Adds a new action to the top of the stack. If the stack's capacity is reached, it handles overflow by clearing the stack.

Pop (__pop):

Removes and returns the action at the top of the stack. It handles the empty stack case gracefully by returning NULL.

Top/Peek:

While not explicitly defined, the top action can be accessed directly via the stack's __top attribute.

Usage in Undo/Redo Functionality:

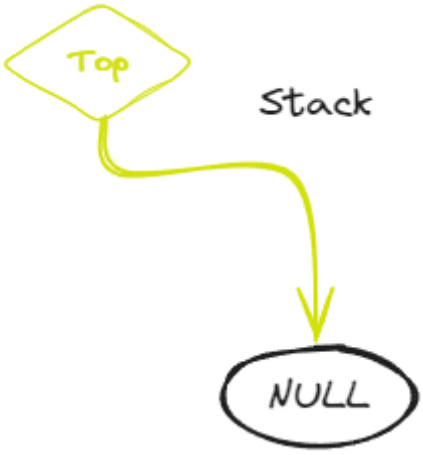
The stack is used to keep track of user actions in the application. When a user performs an action, it is pushed onto the stack.

To undo an action, the application pops the top action from the stack and reverses its effect.

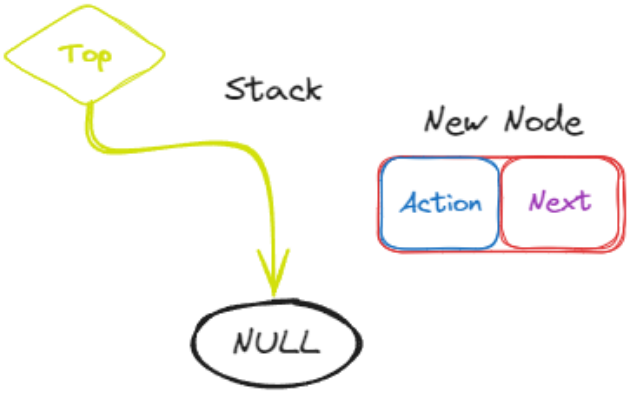
This implementation allows for easy tracking and reversal of recent actions, which is a key aspect of user-friendly application design.

Last-In-First-Out (Stack) Illustration

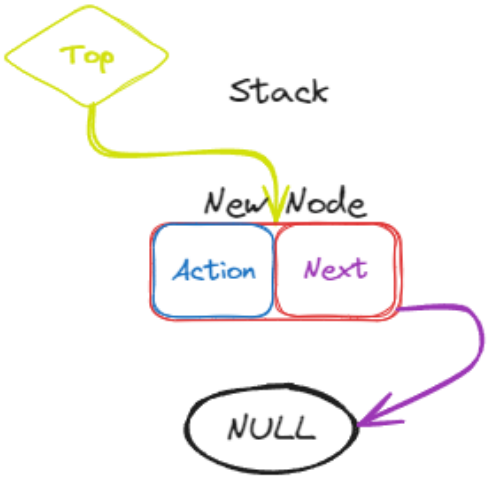
An empty stack:



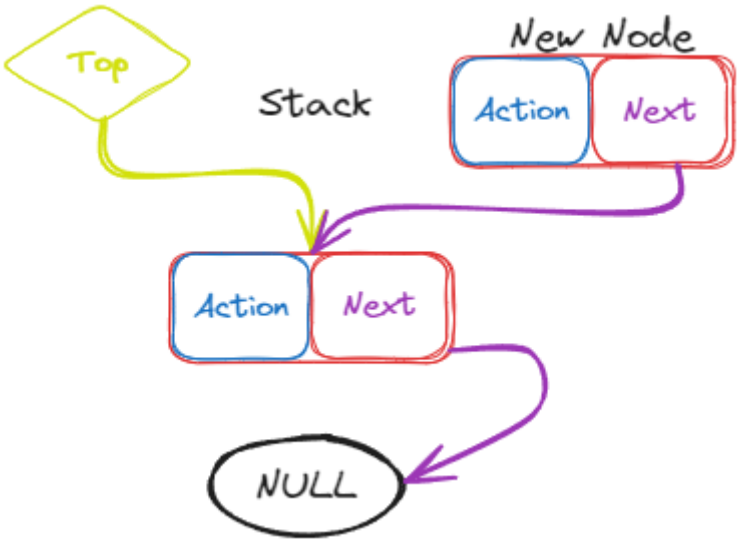
Making a new node:



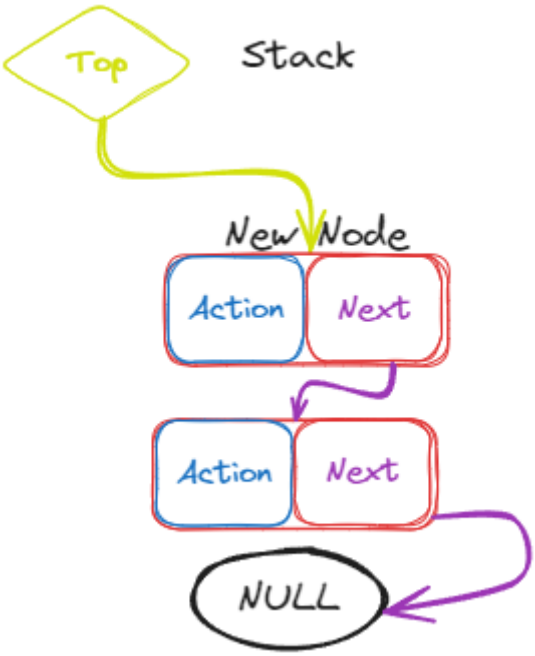
Pushing a new node onto the stack – Linking the “top” pointer to the new node:



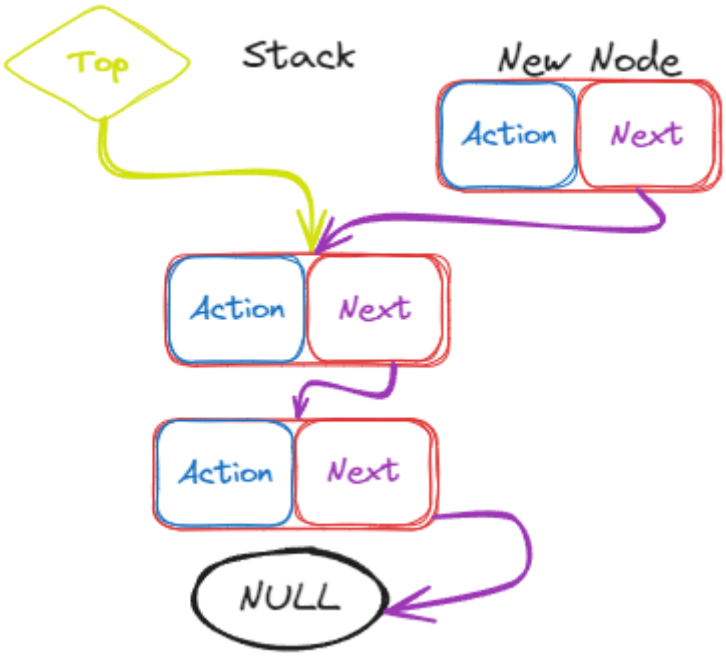
Pushing the second node onto the stack – Linking new node “next” pointer to the “Top” pointer:



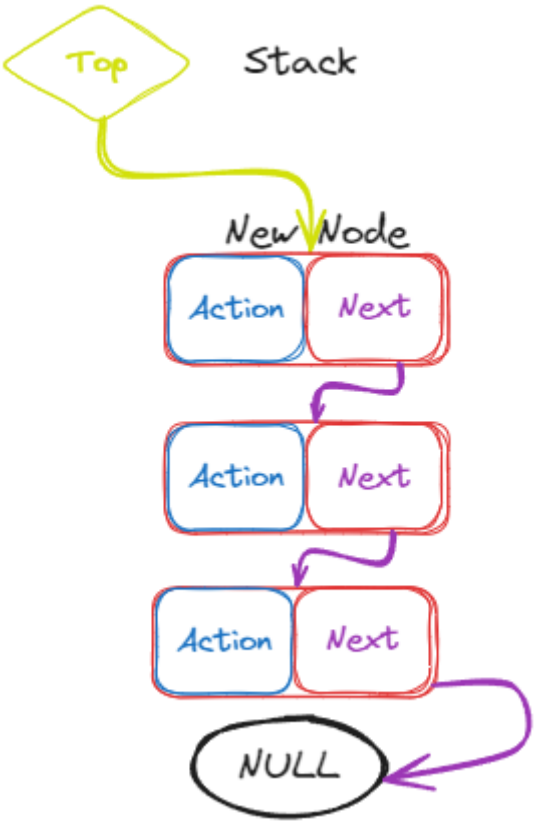
Pushing the second node onto the stack – Linking the “Top” pointer to the new node:



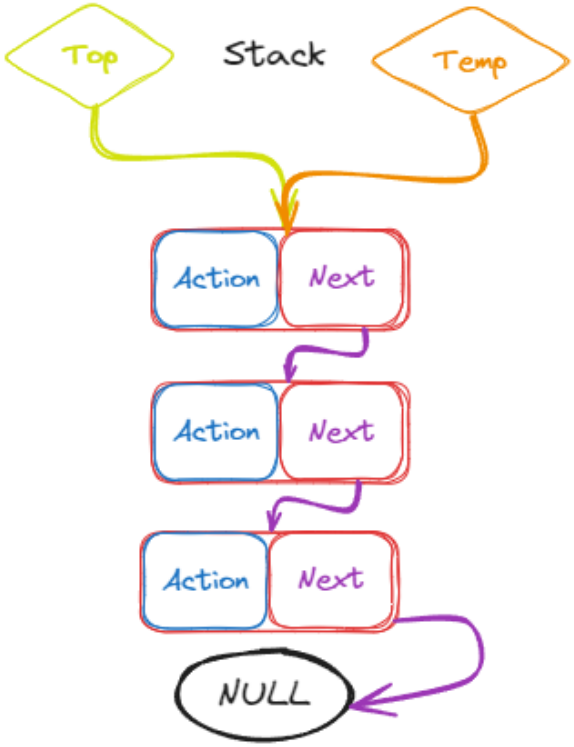
Pushing a third node onto the stack – Linking the “Next” pointer to the “Top” Pointer:



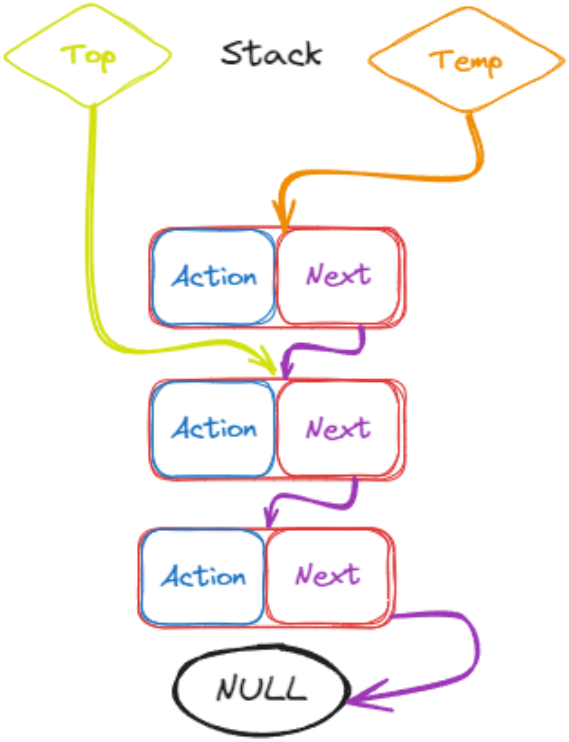
Pushing a third node onto the stack – Linking the “Top” Pointer to the new node:



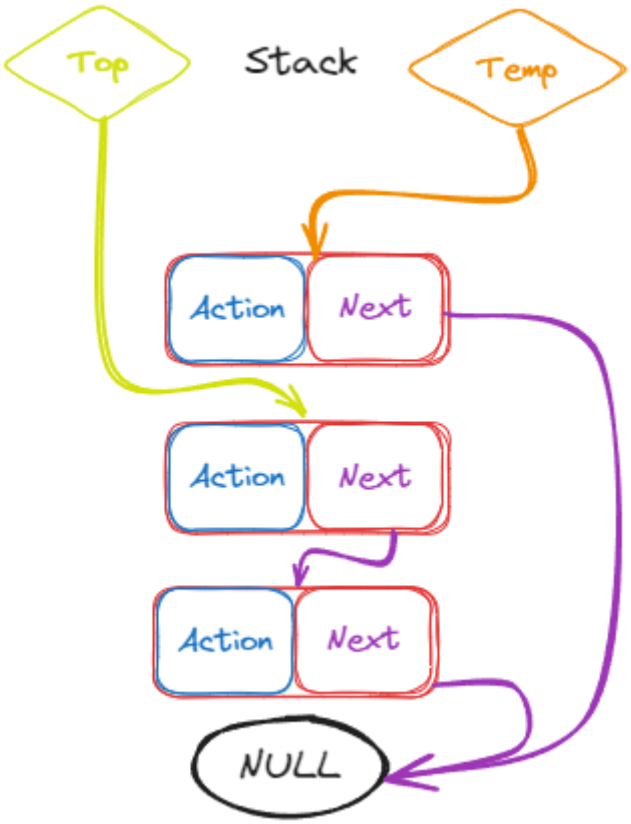
Popping from the stack – Linking a “Temp” pointer to the “Top” pointer:



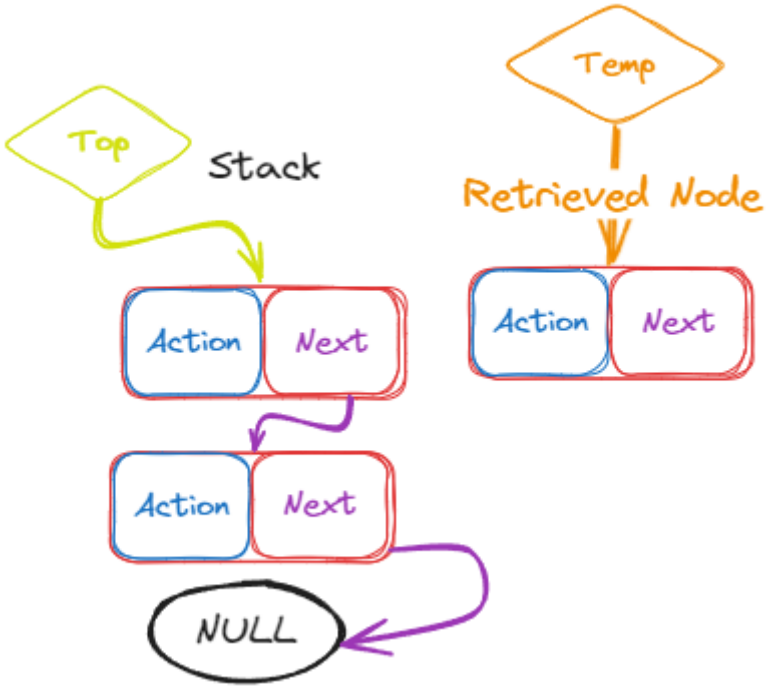
Popping from the stack – Linking the “Top” pointer to the “Next” pointer of the “Top” pointer:



Popping from the stack – Nullifying the “Next” pointer of the “Temp” pointer:



Popping from the stack – Retrieving the popped Node:



How Abstract Data Types (ADTs) help with software development

Stack ADT for Undo Functionality

Problem Solved:

The Stack ADT is used to track user actions in reverse order. Each action performed by the user is "pushed" onto the stack. When the user invokes the undo functionality, the most recent action is "popped" from the stack and reversed.

How It Helps:

LIFO Order: The Last-In, First-Out order of the stack is ideal for undo functionalities. It ensures that the most recent action is always on top and can be easily accessed and reversed.

Efficiency: The stack allows for O(1) time complexity for both pushing (adding) and popping (removing) actions, making it a highly efficient data structure for this purpose.

Simplicity: The stack provides a straightforward and intuitive way to manage the order and reversal of actions without complex logic.

Impact of Not Using Stack:

Without a stack, implementing an undo feature would become significantly more complex. Alternative data structures might not naturally support the LIFO order, leading to additional logic and potential for errors. Efficiency in accessing the most recent actions could also be compromised.

Queue ADT for Tracking Sentences

Problem Solved:

The Queue ADT is used to store and manage sentences entered by the user in the order they were entered. This structure is likely used for processing or displaying the sentences sequentially.

How It Helps:

FIFO Order: The First-In, First-Out nature of the queue ensures that sentences are processed in the exact order they were entered. This is crucial for maintaining the chronological sequence of user input.

Consistent Order Management: The queue provides an automatic and error-free way to manage the order of sentences, reducing the complexity of handling this manually.

Efficiency in Processing: With O(1) time complexity for enqueue (adding) and dequeue (removing) operations, the queue offers an efficient way to handle sentence inputs.

Impact of Not Using Queue:

Without a queue, maintaining the correct order of sentences could become cumbersome and error-prone. Alternative data structures might require additional logic to ensure FIFO processing, potentially leading to increased complexity and decreased efficiency.

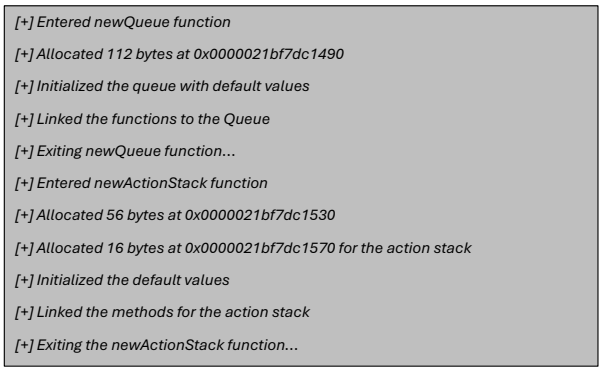
Code Implementation

The code has been provided in a single .zip file, and uploaded on a GitHub repo at the following link:

GitHub Repo

Testing & Debugging

Initializing the structures



Inserting the first text

```
$>insert this is the first item
 [+] Entered __inputSplitter function
[+] Allocating memory for the split user input...
[+] Allocated 8 bytes at 0x000001f6047f1590
[+] Allocating a temporary buffer for the user input...
[+] Allocated 31 bytes at 0x000001f6047f15b0
[+] Initializing the token pointer...
[+] Got the token pointer
    \___ Location: 0x000001f6047f15b0
    \___ Data: insert
[+] Allocating memory for the 1 user input...
[+] Allocated 7 bytes at 0x000001f6047f15e0 for the 1 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x000001f6047f15b7
    \___ Data: this
[+] Allocating memory for the 2 user input...
[+] Allocated 5 bytes at 0x000001f6047f1600 for the 2 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x000001f6047f15bc
    \___ Data: is the first item
[+] Allocating memory for the 3 user input...
[+] Allocated 18 bytes at 0x000001f6047f1620 for the 3 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x0000000000000000
    \___ Data: (null)
[+] Freeing 1 from 0x000001f6047f15b0
[+] Exiting __inputSplitter function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered __putTail function
[+] Entering __newQueueEntry function
[+] Allocated 32 bytes for the new node at 0x000001f6047f1640
[+] Allocated 23 bytes for the new node data at 0x000001f6047f1670
[+] Set up the new node at 0x000001f6047f1640
[+] Exiting __newQueueEntry function...
[+] The Queue is empty
    \___ Linked the head and tail to the new node
[+] Entered newAction function
[+] Allocated 40 bytes at 0x000001f6047f1690
[+] Copied the action data to the Action
[+] Linked the action methods to the object
[+] Exiting newAction function...
[+] Entered __push function
[+] Action stack is empty
    \___ Assigning the new action to the top...
[+] Exiting __push function...
```

Inserting a second text

```
$>insert this is the second text
 [+] Entered __inputSplitter function
[+] Allocating memory for the split user input...
[+] Allocated 8 bytes at 0x000001f6047f15b0
[+] Allocating a temporary buffer for the user input...
[+] Allocated 32 bytes at 0x000001f6047f1700
[+] Initializing the token pointer...
[+] Got the token pointer
    \___ Location: 0x000001f6047f1700
    \___ Data: insert
[+] Allocating memory for the 1 user input...
[+] Allocated 7 bytes at 0x000001f6047f1730 for the 1 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x000001f6047f1707
    \___ Data: this
[+] Allocating memory for the 2 user input...
[+] Allocated 5 bytes at 0x000001f6047f1750 for the 2 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x000001f6047f170c
    \___ Data: is the second text
[+] Allocating memory for the 3 user input...
[+] Allocated 19 bytes at 0x000001f6047f1770 for the 3 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x0000000000000000
    \___ Data: (null)
[+] Freeing 1 from 0x000001f6047f1700
[+] Exiting __inputSplitter function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered __putTail function
[+] Entering __newQueueEntry function
[+] Allocated 32 bytes for the new node at 0x000001f6047f1790
[+] Allocated 24 bytes for the new node data at 0x000001f6047f17c0
[+] Set up the new node at 0x000001f6047f1790
[+] Exiting __newQueueEntry function...
[+] Linked the new node to the tail
[+] Exiting the __putTail function...
[+] Entered newAction function
[+] Allocated 40 bytes at 0x000001f6047f17e0
[+] Copied the action data to the Action
[+] Linked the action methods to the object
[+] Exiting newAction function...
[+] Entered __push function
[+] Linking the action to the previous actions...
[+] Exiting __push function...
```

Printing all the contents

```
$>printdata
 [+] Entered __inputSplitter function
[+] Allocating memory for the split user input...
[+] Allocated 8 bytes at 0x00000202d1441ad0
[+] Allocating a temporary buffer for the user input...
[+] Allocated 10 bytes at 0x00000202d1441bd0
[+] Initializing the token pointer...
[+] Got the token pointer
    \___ Location: 0x00000202d1441bd0
    \___ Data: printdata
[+] Allocating memory for the 1 user input...
[+] Allocated 10 bytes at 0x00000202d1441af0 for the 1 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x0000000000000000
    \___ Data: (null)
[+] Freeing 1 from 0x00000202d1441bd0
[+] Exiting __inputSplitter function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered the __textMenu function
    \___ Command: printdata
[+] Entered __QueueIterate function
[+] Entered __newQueueIter Function
[+] New Iterator at 0x00000202d1441b10
    \___ Size: 24
[+] Linked the Iterator to the queue head
    \___ Iterator: 0x00000202d1441b10
    \___ Queue: Head 0x00000202d1441640
[+] Linked the next function to the iterator
    \___ Iterator: 0x00000202d1441b10
    \___ Function: 0x00007ff7d5c216b0
[+] Linked the delete function to the iterator
    \___ Iterator: 0x00000202d1441b10
    \___ Function: 0x00007ff7d5c21700
[+] Exiting __newQueueIter Function...
[+] Entering __QueueIterNext Function
[+] Retrieved the queue head for the iterator at 0x00000202d1441640
[+] Exiting __QueueIterNext Function...
Index: 0, Data: this is the first item
[+] Entering __QueueIterNext Function
[+] Retrieved the queue head for the iterator at 0x00000202d1441820
[+] Exiting __QueueIterNext Function...
Index: 1, Data: this is the second texr
[+] Entering __QueueIterNext Function
[+] Retrieved the queue head for the iterator at 0x0000000000000000
[+] Exiting __QueueIterNext Function...
[+] Entring __QueueIterDel Function
[+] Freeing the Iterator at 0x00000202d1441b10
    \___ Size: 24
[+] Freed 24 bytes from the Iterator at 0x00000202d1441b10
[+] Exiting __QueueIterDel function...
[+] Deleted the Iterator
[+] Exiting __QueueIterate function...
$>
```

Modifying the text at index 2

```
$>modify 1 this is the modified text at index 1
 [+] Entered __inputSplitter function
[+] Allocating memory for the split user input...
[+] Allocated 8 bytes at 0x000001ef20ff1960
[+] Allocating a temporary buffer for the user input...
[+] Allocated 46 bytes at 0x000001ef20ff1c60
[+] Initializing the token pointer...
[+] Got the token pointer
    \___ Location: 0x000001ef20ff1c60
    \___ Data: modify
[+] Allocating memory for the 1 user input...
[+] Allocated 7 bytes at 0x000001ef20ff1ca0 for the 1 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x000001ef20ff1c67
    \___ Data: 1
[+] Allocating memory for the 2 user input...
[+] Allocated 2 bytes at 0x000001ef20ff1700 for the 2 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x000001ef20ff1c69
    \___ Data: this is the modified text at index 1
[+] Allocating memory for the 3 user input...
[+] Allocated 37 bytes at 0x000001ef20ff63a0 for the 3 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x0000000000000000
    \___ Data: (null)
[+] Freeing 1 from 0x000001ef20ff1c60
[+] Exiting __inputSplitter function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered the __textMenu function
    \___ Command: modify
[+] Entered __updateData function
[+] Queue: 0x000001ef20ff1490
    \___ Index: 1
    \___ Data: this is the modified text at index 1
[+] Entered __QueueFind Function
[+] Got a link to the Queue Head at 0x000001ef20ff1640
[+] Found the needed node
[+] Exiting __updateData function...
[+] Entered newAction function
[+] Allocated 40 bytes at 0x000001ef20ff63d0
[+] Copied the action data to the Action
[+] Linked the action methods to the object
[+] Exiting newAction function...
[+] Entered __push function
[+] Linking the action to the previous actions...
[+] Exiting __push function...
```

Printing all the contents (post-modification)

```
$>printdata
 [+] Entered __inputSplitter function
[+] Allocating memory for the split user input...
[+] Allocated 8 bytes at 0x000001ef20ff1ba0
[+] Allocating a temporary buffer for the user input...
[+] Allocated 10 bytes at 0x000001ef20ff1940
[+] Initializing the token pointer...
[+] Got the token pointer
    \___ Location: 0x000001ef20ff1940
    \___ Data: printdata
[+] Allocating memory for the 1 user input...
[+] Allocated 10 bytes at 0x000001ef20ff19e0 for the 1 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x0000000000000000
    \___ Data: (null)
[+] Freeing 1 from 0x000001ef20ff1940
[+] Exiting __inputSplitter function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered the __textMenu function
    \___ Command: printdata
[+] Entered __QueueIterate function
[+] Entered __newQueueIter Function
[+] New Iterator at 0x000001ef20ff18e0
    \___ Size: 24
[+] Linked the Iterator to the queue head
    \___ Iterator: 0x000001ef20ff18e0
    \___ Queue: Head 0x000001ef20ff1640
[+] Linked the next function to the iterator
    \___ Iterator: 0x000001ef20ff18e0
    \___ Function: 0x00007ff6a11a16b0
[+] Linked the delete function to the iterator
    \___ Iterator: 0x000001ef20ff18e0
    \___ Function: 0x00007ff6a11a1700
[+] Exiting __newQueueIter Function...
[+] Entering __QueueIterNext Function
[+] Retrieved the queue head for the iterator at 0x000001ef20ff1640
[+] Exiting __QueueIterNext Function...
Index: 0, Data: this is the first item
[+] Entering __QueueIterNext Function
[+] Retrieved the queue head for the iterator at 0x000001ef20ff1790
[+] Exiting __QueueIterNext Function...
Index: 1, Data: this is the modified text at index 1
[+] Entering __QueueIterNext Function
[+] Retrieved the queue head for the iterator at 0x0000000000000000
[+] Exiting __QueueIterNext Function...
[+] Entring __QueueIterDel Function
[+] Freeing the Iterator at 0x000001ef20ff18e0
    \___ Size: 24
[+] Freed 24 bytes from the Iterator at 0x000001ef20ff18e0
[+] Exiting __QueueIterDel function...
[+] Deleted the Iterator
[+] Exiting __QueueIterate function...
```

Dequeuing data

```
$>remove def
 [+] Entered __inputSplitter function
[+] Allocating memory for the split user input...
[+] Allocated 8 bytes at 0x000002380e0c1b20
[+] Allocating a temporary buffer for the user input...
[+] Allocated 11 bytes at 0x000002380e0c1ae0
[+] Initializing the token pointer...
[+] Got the token pointer
    \___ Location: 0x000002380e0c1ae0
    \___ Data: remove
[+] Allocating memory for the 1 user input...
[+] Allocated 7 bytes at 0x000002380e0c17c0 for the 1 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x000002380e0c1ae7
    \___ Data: def
[+] Allocating memory for the 2 user input...
[+] Allocated 4 bytes at 0x000002380e0c6430 for the 2 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x0000000000000000
    \___ Data: (null)
[+] Freeing 1 from 0x000002380e0c1ae0
[+] Exiting __inputSplitter function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered the __textMenu function
    \___ Command: remove
[+] Entered __delHead function (dequeue)
[+] Entered __getHead function
[+] Retrieving data inside the head...
[+] Exiting __getHead function...
[+] retrieving the data inside the head...
[+] Exiting __delHead (dequeue) function...
[+] Entered newAction function
[+] Allocated 40 bytes at 0x000002380e0c1640
[+] Copied the action data to the Action
[+] Linked the action methods to the object
[+] Exiting newAction function...
[+] Entered __push function
[+] Linking the action to the previous actions...
[+] Exiting __push function...
this is the first item
```

Printing all the contents (post-dequeuing)

```
$>printdata
 [+] Entered __inputSplitter function
[+] Allocating memory for the split user input...
[+] Allocated 8 bytes at 0x000002380e0c1b40
[+] Allocating a temporary buffer for the user input...
[+] Allocated 10 bytes at 0x000002380e0c1b60
[+] Initializing the token pointer...
[+] Got the token pointer
    \___ Location: 0x000002380e0c1b60
    \___ Data: printdata
[+] Allocating memory for the 1 user input...
[+] Allocated 10 bytes at 0x000002380e0c1920 for the 1 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x0000000000000000
    \___ Data: (null)
[+] Freeing 1 from 0x000002380e0c1b60
[+] Exiting __inputSplitter function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered the __textMenu function
    \___ Command: printdata
[+] Entered __QueueIterate function
[+] Entered __newQueueIter Function
[+] New Iterator at 0x000002380e0c1a60
    \___ Size: 24
[+] Linked the Iterator to the queue head
    \___ Iterator: 0x000002380e0c1a60
    \___ Queue: Head 0x000002380e0c1790
[+] Linked the next function to the iterator
    \___ Iterator: 0x000002380e0c1a60
    \___ Function: 0x00007ff6e21116b0
[+] Linked the delete function to the iterator
    \___ Iterator: 0x000002380e0c1a60
    \___ Function: 0x00007ff6e2111700
[+] Exiting __newQueueIter Function...
[+] Entering __QueueIterNext Function
[+] Retrieved the queue head for the iterator at 0x000002380e0c1790
[+] Exiting __QueueIterNext Function...
Index: 1, Data: this is the modified text at index 1
[+] Entering __QueueIterNext Function
[+] Retrieved the queue head for the iterator at 0x0000000000000000
[+] Exiting __QueueIterNext Function...
[+] Entring __QueueIterDel Function
[+] Freeing the Iterator at 0x000002380e0c1a60
    \___ Size: 24
[+] Freed 24 bytes from the Iterator at 0x000002380e0c1a60
[+] Exiting __QueueIterDel function...
[+] Deleted the Iterator
[+] Exiting __QueueIterate function...
```

Undoing the dequeue action

```
$>undo
 [+] Entered __inputSplitter function
[+] Allocating memory for the split user input...
[+] Allocated 8 bytes at 0x000002380e0c1a60
[+] Allocating a temporary buffer for the user input...
[+] Allocated 5 bytes at 0x000002380e0c1670
[+] Initializing the token pointer...
[+] Got the token pointer
    \___ Location: 0x000002380e0c1670
    \___ Data: undo
[+] Allocating memory for the 1 user input...
[+] Allocated 5 bytes at 0x000002380e0c6450 for the 1 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x0000000000000000
    \___ Data: (null)
[+] Freeing 1 from 0x000002380e0c1670
[+] Exiting __inputSplitter function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered the __textMenu function
    \___ Command: undo
[+] Entered __pop function
[+] Exiting __pop  function...
[+] Entered __updateData function
[+] Queue: 0x000002380e0c1490
    \___ Index: 0
    \___ Data: this is the first item
[+] Entered __QueueFind Function
[+] Got a link to the Queue Head at 0x000002380e0c1790
[+] Exiting __QueueFind Function...
[+] Calling __updateDataOrdered...
[+] Entered __updateDataOrdered function
[+] Queue: 0x000002380e0c1490
    \___ Index: 0
    \___ Data: this is the first item
[+] Entering __newQueueEntry function
[+] Allocated 32 bytes for the new node at 0x000002380e0c6470
[+] Allocated 23 bytes for the new node data at 0x000002380e0c19c0
[+] Set up the new node at 0x000002380e0c6470
[+] Exiting __newQueueEntry function...
[+] Incremented the queue count
[+] Entered __putHead function
[+] Entering __newQueueEntry function
[+] Allocated 32 bytes for the new node at 0x000002380e0c64a0
[+] Allocated 23 bytes for the new node data at 0x000002380e0c1aa0
[+] Set up the new node at 0x000002380e0c64a0
[+] Exiting __newQueueEntry function...
[+] Linked the new Node
[+] Exiting __putHead function...
[+] Added the new Node to the head
[+] Exiting __updateData function...
```

Printing all the contents (post-undo)

```
$>printdata
 [+] Entered __inputSplitter function
[+] Allocating memory for the split user input...
[+] Allocated 8 bytes at 0x000002380e0c1980
[+] Allocating a temporary buffer for the user input...
[+] Allocated 10 bytes at 0x000002380e0c1c00
[+] Initializing the token pointer...
[+] Got the token pointer
    \___ Location: 0x000002380e0c1c00
    \___ Data: printdata
[+] Allocating memory for the 1 user input...
[+] Allocated 10 bytes at 0x000002380e0c18c0 for the 1 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x0000000000000000
    \___ Data: (null)
[+] Freeing 1 from 0x000002380e0c1c00
[+] Exiting __inputSplitter function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered the __textMenu function
    \___ Command: printdata
[+] Entered __QueueIterate function
[+] Entered __newQueueIter Function
[+] New Iterator at 0x000002380e0c19a0
    \___ Size: 24
[+] Linked the Iterator to the queue head
    \___ Iterator: 0x000002380e0c19a0
    \___ Queue: Head 0x000002380e0c64a0
[+] Linked the next function to the iterator
    \___ Iterator: 0x000002380e0c19a0
    \___ Function: 0x00007ff6e21116b0
[+] Linked the delete function to the iterator
    \___ Iterator: 0x000002380e0c19a0
    \___ Function: 0x00007ff6e2111700
[+] Exiting __newQueueIter Function...
[+] Entering __QueueIterNext Function
[+] Retrieved the queue head for the iterator at 0x000002380e0c64a0
[+] Exiting __QueueIterNext Function...
Index: 0, Data: this is the first item
[+] Entering __QueueIterNext Function
[+] Retrieved the queue head for the iterator at 0x000002380e0c1790
[+] Exiting __QueueIterNext Function...
Index: 1, Data: this is the modified text at index 1
[+] Entering __QueueIterNext Function
[+] Retrieved the queue head for the iterator at 0x0000000000000000
[+] Exiting __QueueIterNext Function...
[+] Entring __QueueIterDel Function
[+] Freeing the Iterator at 0x000002380e0c19a0
    \___ Size: 24
[+] Freed 24 bytes from the Iterator at 0x000002380e0c19a0
[+] Exiting __QueueIterDel function...
[+] Deleted the Iterator
[+] Exiting __QueueIterate function...
```

Exiting the application

```
$>exit
 [+] Entered __inputSplitter function
[+] Allocating memory for the split user input...
[+] Allocated 8 bytes at 0x000002d52a061700
[+] Allocating a temporary buffer for the user input...
[+] Allocated 5 bytes at 0x000002d52a061720
[+] Initializing the token pointer...
[+] Got the token pointer
    \___ Location: 0x000002d52a061720
    \___ Data: exit
[+] Allocating memory for the 1 user input...
[+] Allocated 5 bytes at 0x000002d52a061890 for the 1 user input
[+] Copying the data from the token to the user input array...
[+] Getting the new token pointer...
[+] Got the token pointer
    \___ Location: 0x0000000000000000
    \___ Data: (null)
[+] Freeing 1 from 0x000002d52a061720
[+] Exiting __inputSplitter function...
[+] Entered __toLowerString function
[+] Exiting __toLowerString function...
[+] Entered __delQueue function
[+] Got a pointer to the head
    \___ Deleting nodes...
[+] Deleted the nodes
    \___ Deleting the queue object.
[+] Exiting __delQueue function...
[+] Entered __delStack
[+] Found More than one action
    \___ Deleting...
[+] Entered __delAction
[+] Exiting __delAction...
[+] Exiting __delStack...

Process finished with exit code 0
```

Encapsulation & Information Hiding

Encapsulation

Encapsulation in OOP involves bundling the data (variables) and the methods (functions) that operate on the data into a single unit or class. In C, this is typically achieved using structures (struct) to bundle data and function pointers for methods.

Examples:

- Linked List, Queue, Stack:
  - Data: head, tail, and count are encapsulated within structures.
  - Methods: Functions like insert, remove, traverse, search (for linked lists), enqueue, dequeue (for queues), and push, pop (for stacks) are encapsulated as function pointers within the structures.
- Array List:
  - Data: The dynamic array and its size (count) are encapsulated within the structure.
  - Methods: Functions like insert, remove, fetch, resize, and delete are included as function pointers.

Advantages:

- Modularity: Each data structure is a self-contained module with its own data and operations.
- Ease of Use: Users of these data structures don't need to know the implementation details to use them effectively.
- Maintainability: Changes to the implementation of a data structure (e.g., changing the underlying linked list in a queue) do not affect the code that uses it.

Information Hiding

Information hiding is about restricting access to the internal details of a module (data structure), exposing only what is necessary for the user. In C, this is achieved by defining structures and their associated functions in such a way that the user doesn't need to know the internal workings.

Examples:

- The internal node structure (pNode) in the linked list, queue, and stack is not exposed to the user.
- Internal resizing logic in the array list is hidden from the user.

Advantages:

- Security: Internal data is protected from unintended external interference.
- Simplicity for the User: Users interact with a simple interface, making it easier to work with complex data structures.
- Abstraction: The user does not need to understand the complex underlying implementation to use the data structure.
- Controlled Access: The internal state of the data structure can only be modified in controlled and predictable ways.

Overall Impact of Using OOP Concepts in C

While C does not support OOP natively like C++ or Java, the use of structures and function pointers to implement encapsulation and information hiding brings many OOP benefits to C programming. These include cleaner, more maintainable code, and a separation between the interface (how you use the data structure) and the implementation (how the data structure is internally structured and managed). This approach leads to more robust and reusable code, where data structures can be easily swapped out or updated without affecting the rest of the program.

Encapsulation & Information Hiding Implementation

Queue

As we can see in “Queue.h”, all the underlying implementation has been encapsulated and hidden in a separate file “Queue.c”, only the means to interact with the queue have been exposed.

```c
#include <stdbool.h>



typedef struct Queue *pOGQueue;
typedef struct QueueEntry QueueEntry, *pQueueEntry;
// Public Class to Encapsulate the original one - Interface
typedef struct QueueImplementation{
    pOGQueue Queue;

    // Dump the queue for debugging purposes
    void (*dumpQueue)(struct QueueImplementation* self);
    // Free the queue from the memory
    void (*delQueue)(struct QueueImplementation* self);
    // Iterate the queue printing out all the elements in FIFO order
    void (*iterQueue)(struct QueueImplementation* self);
    // Update data at the given index in the queue
    int (*updateData)(struct QueueImplementation* self, int index, char* data, bool inc);
    // Get the data at the given index
    char* (*getQueue)(struct QueueImplementation* self, int index, char* def);
    // Get the data at the front of the queue
    char* (*getHead)(struct QueueImplementation* self, char* def);
    // Get the Head's index
    int (*getHeadIndex)(struct QueueImplementation* self);
    // Insert data at the beginning of the queue
    int (*putHead)(struct QueueImplementation* self, char* data, bool inc);
    // Get the data at the end of the queue
    char* (*getTail)(struct QueueImplementation* self, char* def);
    // Add data to the queue
    int (*enqueueData)(struct QueueImplementation* self, char* data);
    // Get the data at the end of the queue then free it (Don't forget to free the dequeued data)
    char* (*dequeueData)(struct QueueImplementation* self, char* def);
    // Delete data at a given index
    void (*deleteData)(struct QueueImplementation* self, int index);
    // Get the size of the queue aka the number of nodes
    int (*sizeQueue)(struct QueueImplementation* self);
}*pQueue, *pQueueImpl;

// Initialize the queue
pQueue newQueue();
```

Action Stack

Observing the “ActionUndo.h”, we can see that, like the Queue, the underlying structure for the action stack has been encapsulated and hidden “ActionUndo.c” , leaving only the methods to interact with the structure exposed.

```c
// A holder for the values of ops
typedef enum actionDone {INSERT, MODIFY, DELETE} ActionDone, *pActionDone;
// Defining a struct from our original Action Class to encapsulate
typedef struct ActionOG *pActionOG;
// Defining the encapsulated Action Class
typedef struct Action{
    pActionOG action;
    char* (*getData)(struct Action* self);
    int (*getIndex)(struct Action* self);
    enum actionDone (*getAction)(struct Action* self);
    void (*delAction)(struct Action* self);
}*pAction;
// Defining the original ActionStack class to encapsulate
typedef struct ActionStack *pActionStackOG;
// Defining the encapsulated ActionStack
typedef struct ActionStackImpl{
    pActionStackOG __actionStack;
    int (*getTopIndex)(struct ActionStackImpl* self);
    char* (*getTopData)(struct ActionStackImpl* self, char* def);
    enum actionDone (*getTopAction)(struct ActionStackImpl* self);
    void (*push)(struct ActionStackImpl* self, pAction action);
    pAction (*pop)(struct ActionStackImpl* self);
    void (*delStack)(struct ActionStackImpl* self);
} ActionStack, *pActionStack;
// Constructor for the encapsulated ActionStack
pActionStack newActionStack();
// Constructor for the encapsulated Action
pAction newAction(enum actionDone action, int index, char* data);
```

Algorithms

An algorithm is a step-by-step procedure or formula for solving a problem. In computing, an algorithm is a set of instructions designed to perform a specific task. This could be anything from a simple calculation to a complex operation like data sorting or pattern searching. The key characteristics of an algorithm include:

Well-defined Inputs and Outputs: An algorithm has zero or more inputs, which are taken from a specified set, and it produces one or more outputs, which are also from a specified set.

Clear and Unambiguous: Each step of the algorithm must be precisely defined; the actions to be carried out in each step must be clearly and unambiguously specified.

Finiteness: An algorithm must always terminate after a finite number of steps.

Effectiveness: Each operation in the algorithm must be sufficiently basic that it can be carried out, in principle, by a person using only a pencil and paper. This does not preclude the possibility of the operations being extremely complex or involving large amounts of computation.

Generality: The algorithm should be applicable for all problems of the desired nature and scope.

Algorithm Complexity

Algorithm complexity refers to the amount of computational resources (like time and space) that an algorithm requires to complete its task. It is typically categorized into two types:

Time Complexity: This measures how the execution time of an algorithm increases with the size of the input data. It's often expressed using Big O notation, which provides an upper limit (Worst Case) on the time taken in terms of the size of the input.

Space Complexity: This measures the amount of memory space required by an algorithm as a function of the size of the input data.

Trade-Offs

The relationship between space and time complexity in algorithms is often characterized by a trade-off. This trade-off can be understood in the context of optimizing one aspect at the expense of the other. Here are key points that illustrate this relationship:

Trade-Off:

> In many cases, you can optimize an algorithm to use less time by allowing it to use more space, and vice versa. This is a common scenario in algorithm design where a balance needs to be struck based on the requirements and constraints of the specific application or system.

Space-Time Complexity Trade-Off Examples:

> Caching/Memorization: Increases space complexity to reduce time complexity. By storing the results of expensive function calls and reusing them when the same inputs occur again, you save time but use more memory.
>
> Data Structures: Different data structures offer various trade-offs. For example, a hash table provides fast data retrieval (lower time complexity) but uses more memory (higher space complexity) compared to a linked list.

Constraints and Optimization:

> The choice of whether to optimize for time or space often depends on system constraints. For instance, in memory-constrained environments like embedded systems, minimizing space complexity may be more important, while in time-sensitive applications like real-time systems, minimizing time complexity is crucial.

Big O Notation:

> Big O notation is used to describe the upper bound of time or space complexity. An algorithm might be efficient in terms of time (e.g., O(log n)) but costly in terms of space (e.g., O(n^2)), or the other way around.

Algorithmic Paradigms:

> Different algorithmic paradigms may emphasize different aspects of the space-time trade-off. For example, dynamic programming often trades increased space usage for reduced computation time.

Modern Computing Trends:

> With the increase in computing power and memory capacities in modern systems, there's often a greater emphasis on reducing time complexity over space complexity, unless working in a highly memory-constrained environment.

No Universal Best Solution:

> It's important to note that there's no universally best solution. The optimal balance between space and time complexity is highly dependent on the specific use case, requirements, and constraints of the application.

How Algorithm Complexity Affects Performance

Efficiency: The complexity of an algorithm directly impacts its efficiency. Lower complexity often means that the algorithm can handle larger input sizes more efficiently.

Scalability: Algorithms with lower time and space complexity are more scalable, meaning they can handle large datasets or complex problems without a significant decrease in performance.

Resource Utilization: An algorithm with high time complexity might require more computational power and thus more energy, which could be a concern in resource-constrained environments like mobile devices. Similarly, high space complexity can lead to increased memory usage, affecting the overall system performance.

Practical Usability: Sometimes, an algorithm with better theoretical efficiency (lower complexity) may not be the best choice in practice due to factors like ease of implementation, constant factors hidden in Big O notation, or specific case optimizations.

Application Suitability: The choice of an algorithm often depends on the specific requirements and constraints of the problem it's being used to solve. For instance, an algorithm that is faster (lower time complexity) but uses more memory (higher space complexity) might be suitable for desktop computers but not for mobile devices.

Comparing ADTs with normal data types

Abstraction Level:

ADTs:

ADTs operate at a higher level of abstraction. They define operations on data without specifying how these operations are implemented. Examples include stacks, queues, lists, and graphs.

ADTs focus more on what operations are performed rather than how they are implemented, allowing for different underlying data structures.

Primitive Data Types:

Primitive data types, like integers, floats, characters, and Booleans, represent basic data and are built into the language.

They provide a foundation for constructing more complex data structures and ADTs.

Functionality and Operations:

ADTs:

ADTs come with a set of operations. For instance, a stack has push, pop, and peek; a queue has enqueue and dequeue.

The operations of ADTs are generally more complex and can include various algorithms.

Primitive Data Types:

Operations on primitive types are straightforward and fundamental, like addition, subtraction, comparison, etc.

They do not inherently support complex operations or algorithms.

Complexity and Implementation:

ADTs:

ADTs can be complex to implement as they require an understanding of algorithms and data structure concepts.

The same ADT can have multiple implementations. For example, a stack can be implemented using arrays or linked lists.

Primitive Data Types:

Primitive types are simple and predefined in the language, so no user-defined implementation is needed.

Their complexity is low, making them easy to understand and use.

Memory Usage:

ADTs:

- Memory usage can vary significantly based on the ADT's implementation.
- ADTs can be more memory-intensive, especially for complex structures like trees or graphs.

Primitive Data Types:

- Memory usage is minimal and well-defined. For instance, an integer in Java always uses 4 bytes.

Performance:

ADTs:

- Performance can vary based on the implementation and the operations performed.
- Some ADTs can be optimized for specific operations, like a heap for quickly finding the minimum or maximum element.

Primitive Data Types:

- Generally, have fast performance for basic operations due to their simplicity and the direct support of the underlying hardware.

Usage and Application:

ADTs:

- Used for solving complex problems in software development where data needs to be organized in specific ways (like FIFO or LIFO).
- Suitable for applications that require complex data manipulation and storage, like managing a printer queue (queue ADT) or undo functionality in an editor (stack ADT).

Primitive Data Types:

- Used for straightforward data representation and calculations.
- Ideal for basic data manipulation where complex data structures are not necessary.

Conclusion:

While primitive data types are the building blocks for any data manipulation in programming, ADTs provide a framework for managing and organizing data in more complex ways. The choice between using ADTs and primitive types depends on the problem at hand. ADTs offer powerful tools for complex data management and algorithms, while primitive data types are suited for basic operations and data representations.

Implemented Code Complexity

Queue

__updateData: This function updates a specific element in the queue.

Time Complexity: O(n) – In the worst case, it may need to traverse the entire queue to find the element to update.

Space Complexity: O(n) – Space is proportional to the size of the input.

```
void __updateData(pQueue self, int index, char* data) {
    DEBUG_INFO("Entered __updateData function\n");
    DEBUG_INFO("Queue: 0x%p\n\t\___ Index: %d\n\t\___ Data: %s\n",self ,index, data);
    if (!self || !data) return ;
    pQueueEntry currentNode = __QueueFind(self, index);
    size_t newDataSize = strlen(data);
    if (currentNode != NULL) {
        DEBUG_INFO("Found the needed node\n");
        pQueueEntry newNode = __newQueueEntry(data);
        currentNode->__prev = newNode;
        newNode->__next = currentNode;
        DEBUG_INFO("Exiting __updateData function...\n");
        return ;
    }
    DEBUG_INFO("Calling __updateDataOrdered...\n");
    __updateDataOrdered(self, index, data);
    DEBUG_INFO("Exiting __updateData function...\n");
}
pQueueEntry __QueueFind(pQueue self, int index){
    DEBUG_INFO("Entered __QueueFind Function\n");
    pQueueEntry current = self->Queue->__head;
    int count = 0;
    DEBUG_OKAY("Got a link to the Queue Head at 0x%p\n", current);
    while(current != NULL && count < index){
        count++;
        current = current->__next;
    }
    return current;
    DEBUG_INFO("Exiting __QueueFind Function...\n");
}
void __updateDataOrdered(pQueue self, int index, char* data) {
    DEBUG_INFO("Entered __updateDataOrdered function\n");
    DEBUG_INFO("Queue: 0x%p\n\t\___ Index: %d\n\t\___ Data: %s\n",self ,index, data);
    if (!self || !data) return ;
    pQueueEntry newNode = __newQueueEntry(data);
    self->Queue->__count++;
    DEBUG_INFO("Incremented the queue count\n");
    if(index == 0){
        __putHead(self, data);
        DEBUG_OKAY("Added the new Node to the head\n");
        return ;
    }
    if(self->Queue->__head == NULL){
        self->Queue->__head = newNode;
        self->Queue->__tail = newNode;
        return ;
    }
    DEBUG_INFO("Finding the right place it insert the node...\n");
    int count = 0;
    pQueueEntry cur = self->Queue->__head;
    while (cur->__next != NULL && count < index) {
        cur = cur->__next;
    }
    DEBUG_INFO("Found the right place!!\n\t\___ cur: 0x%p\n\t\___ cur->next: 0x%p\n\t\___ newNode: 0x%p\n\t\___ head:
0x%p\n\t\___ Tail: 0x%p\n",
                cur, cur->__next, newNode, self->Queue->__head, self->Queue->__tail
                );
    DEBUG_INFO("Setting the newNode->next to  the current->next...\n");
    newNode->__next = cur->__next;
    DEBUG_INFO("Values of\n\t\___ newNode->next: 0x%p\n\t\___ current->next: 0x%p\n", newNode->__next, cur->__next);
    if (cur->__next != NULL) {
        DEBUG_INFO("Linking the node in front of the current one..\n");
        cur->__next->__prev = newNode;

    } else {
        DEBUG_INFO("No node in front of the current one\n\t\___ Linking the tail\n");
        self->Queue->__tail = newNode;
        DEBUG_INFO("Linked the tail\n");
    }
    newNode->__prev = cur;
    DEBUG_INFO("Linked the newNode previous pointer to the current node\n");
    cur->__next = newNode;
    DEBUG_INFO("Linked the current node next pointer to the newNode\n");
    DEBUG_OKAY("Linked the new node in it's right place\n\t\___ Current Node: 0x%p\n\t\___ Next Node: 0x%p\n\t\___
Previous Node: 0x%p\n",
                newNode, newNode->__next, newNode->__prev);
    DEBUG_INFO("Exiting __updateDataOrdered function...\n");
}
```

__sizeQueue: This function returns the size of the queue.

Time Complexity: O(1) – This operation is constant time as it likely just returns a stored count.

Space Complexity: O(1) – No extra space is needed.

```
// Get the size of the queue
int __sizeQueue(pQueue self){
    return self->Queue->__count;
}
```

__getQueue: This function to fetch queue data based on an index.

Time Complexity: O(n) – In the worst case, it needs to traverse the queue to the given index.

Space Complexity: O(1) – No additional space is required.

```
// Get the data at the given index - TC O(n)
char* __getQueue(pQueue self, int index, char* def){
    DEBUG_INFO("Entered __getQueue function\n");
    pQueueEntry toFind = __QueueFind(self, index);
    if(toFind == NULL) {
        DEBUG_ERROR("Didn't find the required node\n");
        DEBUG_INFO("Exiting __getQueue function...\n");
        return def;
    }
    DEBUG_OKAY("Found the required node\n");
    DEBUG_INFO("Exiting __getQueue function...\n");
    return toFind->data;
}
```

__getHead: This function retrieves the data inside the head.

Time Complexity: O(1) – Accessing or modifying the head is a constant time operation.

Space Complexity: O(1) – No extra space is needed.

```
char* __getHead(pQueue self, char* def){
    DEBUG_INFO("Entered __getHead function\n");
    if(self->Queue->__head == NULL) {
        DEBUG_ERROR("Queue head is NULL\n");
        DEBUG_INFO("Exiting __getHead function...\n");
        return def;
    }
    DEBUG_OKAY("Retrieving data inside the head...\n");
    DEBUG_INFO("Exiting __getHead function...\n");
    return self->Queue->__head->data;
}
```

__putHead: This function adds a node to the beginning of the queue.

Time Complexity: O(1) – Adding a node requires only linking the existing head to it.

Space Complexity: O(n) – The space needed is proportional to the input size.

```
// Add a node to the head of the queue - Time Complexity O(1)
void __putHead(pQueue self, char* data){
    DEBUG_INFO("Entered __putHead function\n");
    pQueueEntry newEntry = __newQueueEntry(data);
    self->Queue->__count++;
    if(self->Queue->__head == NULL){
        self->Queue->__head = newEntry;
        self->Queue->__tail = newEntry;
        return ;
    }
    self->Queue->__head->__prev = newEntry;
    newEntry->__next = self->Queue->__head;
    self->Queue->__head = newEntry;
    DEBUG_OKAY("Linked the new Node\n");
    DEBUG_INFO("Exiting __putHead function...\n");
}
```

__getTail: This function retrieves the tail element of the queue.

Time Complexity: O(1) – Direct access to the tail, it's a constant time operation.

Space Complexity: O(1) – No additional space is required.

```
// Get the data in the tail of the queue - TC O(1)
char* __getTail(pQueue self, char* def){
    DEBUG_INFO("Entered __getTail function\n");
    if(self->Queue->__tail == NULL) {
        DEBUG_ERROR("Queue tail is NULL\n");
        DEBUG_INFO("Exiting __getTail function...");
        return def;
    }
    DEBUG_OKAY("Retrieving data inside the tail...\n");
    DEBUG_INFO("Exiting __getTail function...\n");
    return self->Queue->__tail->data;
}
```

__delHead (dequeueData): This function removes the head element of the queue retrieving its data along the process.

Time Complexity: O(1) – Removing the head is typically a constant time operation.

Space Complexity: O(1) – Space is freed up as elements are removed.

```
char* __delHead(pQueue self, char* def){
    DEBUG_INFO("Entered __delHead function (dequeue)\n");

    if(self->Queue->__head == NULL) {
        DEBUG_ERROR("Queue is empty\n");
        return def;
    }
    // Get the data inside the head
    char* retData = __getHead(self, def);
    // Check if it is the default value
    if(strcmp(retData, def) == 0) return def;
    self->Queue->__count--;
    // if not retrieve the OG data
    pQueueEntry toBeDeleted = self->Queue->__head;
    if(self->Queue->__head->__next != NULL) {
        self->Queue->__head->__next->__prev = NULL;
        self->Queue->__head = self->Queue->__head->__next;
    } else {
        // If there's only one node, set both head and tail to NULL
        self->Queue->__head = NULL;
        self->Queue->__tail = NULL;
    }
    // Free the deleted Node from the memory
    free((void*)toBeDeleted);
    DEBUG_OKAY("retrieving the data inside the head...\n");
    DEBUG_INFO("Exiting __delHead (dequeue) function...\n");
    return retData;
}
```

__putTail (enqueueData): This function adds an element to the tail of the queue.

Time Complexity: O(1) – Adding to the tail is a constant time operation if the tail is directly accessible.

Space Complexity: O(n) – The space is proportional to the size of the input.

```
void __putTail(pQueue self, char* data){
    DEBUG_INFO("Entered __putTail function\n");
    // Initialize the new node
    pQueueEntry newEntry = __newQueueEntry(data);

    self->Queue->__count++;
    // If the queue is empty
    if(self->Queue->__head == NULL){
        self->Queue->__head = newEntry;
        self->Queue->__tail = newEntry;
        DEBUG_OKAY("The Queue is empty\n\t\\___ Linked the head and tail to the new node\n");
        return ;
    }
    // If the queue is not empty

    self->Queue->__tail->__next = newEntry;
    newEntry->__prev = self->Queue->__tail;
    self->Queue->__tail = newEntry;
    DEBUG_OKAY("Linked the new node to the tail\n");
    DEBUG_INFO("Exiting the __putTail function...\n");
}
```

__delData: This function deletes a specific element from the queue.

Time Complexity: O(n) – Requires traversal to find and remove the specific element.

Space Complexity: O(1) – Space is released as elements are deleted.

```
void __delData(pQueue self, int index){
    DEBUG_INFO("Entered __delData function\n");
    pQueueEntry toBeDeleted = __QueueFind(self, index);
    if (toBeDeleted == NULL) return;
    DEBUG_OKAY("Found a node\n\t\\___ Deleting...\n");
    // Handle the head of the queue
    if (toBeDeleted == self->Queue->__head) {
        self->Queue->__head = toBeDeleted->__next;
        if (self->Queue->__head != NULL) {
            self->Queue->__head->__prev = NULL;
        }
    }

    // Handle the tail of the queue
    if (toBeDeleted == self->Queue->__tail) {
        self->Queue->__tail = toBeDeleted->__prev;
        if (self->Queue->__tail != NULL) {
            self->Queue->__tail->__next = NULL;
        }
    }

    // Handle a middle node
    if (toBeDeleted->__next != NULL) {
        toBeDeleted->__next->__prev = toBeDeleted->__prev;
    }
    if (toBeDeleted->__prev != NULL) {
        toBeDeleted->__prev->__next = toBeDeleted->__next;
    }

    // Free the node
    free((void*)toBeDeleted->data);
    free((void*)toBeDeleted);
    DEBUG_INFO("Exiting __delData function...\n");
}
```

__QueueIterate: This function iterates over the queue.

Time Complexity: O(n) – Requires traversal of the entire queue.

Space Complexity: O(1) – No extra space needed for iteration.

```
void __QueueIterate(pQueue Queue) {
    DEBUG_INFO("Entered __QueueIterate function\n");
    pQueueIter iter = __newQueueIter(Queue);
    pQueueEntry entry;
    while ((entry = iter->next(iter)) != NULL) {
        printf("Index: %d, Data: %s\n", entry->index, entry->data);
    }

    iter->del(iter);
    DEBUG_OKAY("Deleted the Iterator\n");
    DEBUG_INFO("Exiting __QueueIterate function...\n");
}
```

__QueueDump: This function prints the entire queue.

Time Complexity: O(n) – Involves traversing and processing each element.

Space Complexity: O(1) – No additional space besides the output.

```
void __QueueDump(pQueue self){
    DEBUG_INFO("Entered __QueueDump Function\n");
    if(self == NULL){
        DEBUG_ERROR("Passed in a NULL pointer\n");
        DEBUG_INFO("Exiting __QueueDump...\n");
        return;
    }

    printf("Nodes count: %d\n", self->Queue->__count);
    int i = 0;
    for(pQueueEntry cur = self->Queue->__head; cur != NULL; cur = cur->__next, i++){
        printf("Node 0x%p\n\t\\___ Index: %d\n\t\\___ Data: %s\n", cur, i, cur->data);
    }
    DEBUG_INFO("Exiting __QueueDump...\n");
}
```

__delQueue: This function deallocates the entire queue.

Time Complexity: O(n) – Needs to traverse and free each node in the queue.

Space Complexity: O(1) – Memory is being freed, not allocated.

```
void __delQueue(pQueue self){
    DEBUG_INFO("Entered __delQueue function\n");
    pQueueEntry current, next;
    current = self->Queue->__head;
    DEBUG_OKAY("Got a pointer to the head\n\t\\___ Deleting nodes...\n");
    while(current != NULL){
        next = current->__next;
        free((void*) current->data);
        free((void*)current);
        current = next;
    }
    DEBUG_OKAY("Deleted the nodes\n\t\\___ Deleting the queue object.\n");
    free((void*)self);
    self= NULL;
    DEBUG_INFO("Exiting __delQueue function...\n");
}
```

In general, the operations that involve traversing the queue (like __updateData, __getQueue, __delData) have a linear time complexity, while operations that deal with the ends of the queue (like enqueue, dequeue) are more efficient with constant time complexity. The space complexity for most operations is constant, as they don't require additional space proportional to the size of the queue.

Stack

__getTopAction, __getTopIndex, __getTopData:

Time Complexity: O(1) – These functions perform a constant amount of work regardless of the size of the stack since they are accessing the top element.

Space Complexity: O(1) – They use a fixed amount of space for temporary variables.

```
// Method to get the index of the top action on the stack
int __getTopIndex(pActionStack self) {
    return self->__actionStack->__top->getIndex(self->__actionStack->__top);
}
// Method to get the data of the top action on the stack
char *__getTopData(pActionStack self, char *def) {
    if (self->__actionStack->__top->getData(self->__actionStack->__top) == NULL) return def;
    return self->__actionStack->__top->getData(self->__actionStack->__top);
}
// Method to get the actionType of the top action on the stack
enum actionDone __getTopAction(pActionStack self) {
    return self->__actionStack->__top->getAction(self->__actionStack->__top);
}
```

__push:

Time Complexity: O(1) – This function adds an element to the top of the stack, which is an operation that takes a constant amount of time.

Space Complexity: O(n) – The space used by the function is independent of the number of elements in the stack but proportional to the size of the data entered.

```
// Method to "push" an action onto the ActionStack
void __push(pActionStack self, pAction doneAction) {
    DEBUG_INFO("Entered __push function\n");
    if(self->__actionStack->__count >= self->__actionStack->__capacity){
        DEBUG_INFO("Action Stack FULL...\n\t\\__ Printing it's contents...\n");
#ifdef DEBUG
        __printActionStack(self);
#endif
        DEBUG_INFO("Freeing the actionStack...\n");
        for(pAction cur = self->pop(self); cur != NULL;cur=self->pop(self)){
            pAction temp = cur;
            free((void*)temp);
        }
        DEBUG_INFO("Printing the stack after freeing it...\n");
#ifdef DEBUG
        __printActionStack(self);
#endif
    }

    if(self->__actionStack->__top == NULL){
        DEBUG_INFO("Action stack is empty\n\t\\__ Assigning the new action to the top...\n");
        self->__actionStack->__top = doneAction;
        self->__actionStack->__count++;
        DEBUG_INFO("Exiting __push function...\n");
        return;
    }
    DEBUG_INFO("Linking the action to the previous actions...\n");
    doneAction->action->__next = self->__actionStack->__top;
    self->__actionStack->__top = doneAction;
    self->__actionStack->__count++;
    DEBUG_INFO("Exiting __push function...\n");
}
```

__pop:

Time Complexity: O(1) – This function removes the top element from the stack, which is also a constant time operation.

Space Complexity: O(1) – Uses a fixed amount of space.

```
// Method to "pop" an action from the ActionStack
pAction __pop(pActionStack self) {
    DEBUG_INFO("Entered __pop function\n");
    if (self->__actionStack->__top == NULL) return NULL;
    pAction poppedAction = self->__actionStack->__top;
    self->__actionStack->__top = self->__actionStack->__top->action->__next;
    poppedAction->action->__next = NULL;
    self->__actionStack->__count--;
    DEBUG_INFO("Exiting __pop  function...\n");
    return poppedAction;
}
```

__delStack:

Time Complexity: O(n) – This function deallocates each element in the stack, it would need to iterate through all elements, leading to linear time complexity.

Space Complexity: O(1) – The space used for deletion is independent of the size of the stack.

```
// Method to delete (free) stack from the memory
void __delStack(pActionStack self) {
    DEBUG_INFO("Entered __delStack\n");
    if (self->__actionStack->__top != NULL) {
        pAction current = self->__actionStack->__top;
        pAction stored;
        while (current->action->__next != NULL) {
            DEBUG_INFO("Found More than one action\n\t\\___ Deleting...\n");
            stored = current->action->__next;
            __delAction(current);
            current = stored;
        }
        free((void *) current);
    }
    free((void *) self->__actionStack);
    free((void *) self);
    self = NULL;
    DEBUG_INFO("Exiting __delStack...\n");
}
```

These functions represent typical stack operations, and their complexities are aligned with standard stack behavior. The constant time complexities for push and pop operations are key characteristics of a stack, making it efficient for operations like undo/redo in applications.

Part 2

Recursive Functions in the Stack

We will be observing the behavior of the calling stack by implementing a fibonacci function.

Code:

```c
// Fibonacci sequence
int fibonacci(int n){
    return n < 2 ? n: fibonacci(n-1) + fibonacci(n-2);
}
int main() {
    printf("Fibonacci 10 = %d\n", fibonacci(10));
    return 1;
```

To understand the recursive calling we need to understand the disassembled functions, what we need to know before:

RBP:

Base Pointer Register – Points to the start of the memory stack.

RSP:

Stack Pointer Register – Points to the end of the memory stack.

RAX:

Accumulator Register – General use register, used to store the return value from the fibonacci function.

RCX:

Counter Register – General use register, normally used to count for loops, in this case used to pass the first argument to the functions.

RDX:

Data Register – General use register, usually used to store data, in this case it's used to pass the second argument to the functions.

Main Function:

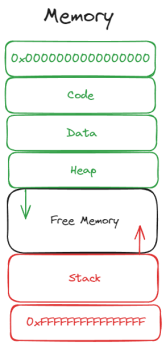
```
main:
; Push the old stack base pointer.
    push   rbp
; Set the base pointer to the end of the stack.
    mov    rbp,rsp
; Expand the stack by subtracting (adding) 32 bytes to the stack - Allocate a stack frame for the function.
    sub    rsp,0x20
; Call the __main function (Compiler function)
    call   0x7ff61f1141c7 <__main>
; Load '10' into the rcx register (first argument to the fib function).
    mov    ecx,0xa
; Call the fibonacci function (Passing 10 to it)
    call   0x7ff61f111634 <fibonacci>
; Store the returned value from the function in the rdx register.
    mov    edx,eax
; Load the format string to the rax register (constant string)
    lea    rax,[rip+0xb96e]
; Load the format string to the rcx (first argument to printf)
    mov    rcx,rax
; Call the printf function
    call   0x7ff61f1115e0 <printf>
; Load 0 to the eax (main function return value)
    mov    eax,0x0
; Resize the stack indicating the end of the function
    add    rsp,0x20
; Retrieve the old base pointer signaling the retrieval of the old stack
    pop    rbp
; Return to control to the calling function indicated by the base pointer
    ret
```

Fibonacci Function:

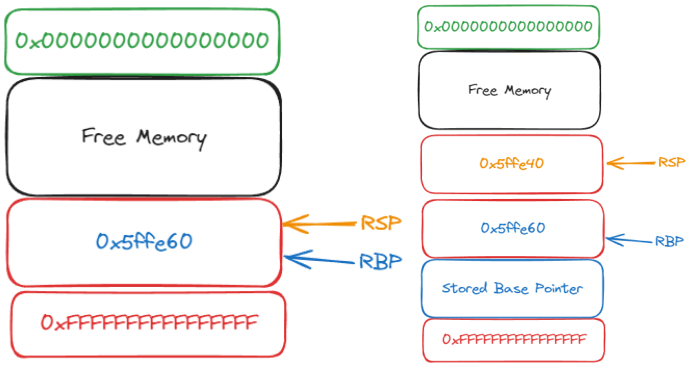
```
fibonacci:
; Push the old stack base pointer.
    push   rbp
; Push the old RBX value - Callee saved register, must be stored before executing functions.
    push   rbx
; Allocate space for the new function stack frame - 40 bytes.
    sub    rsp,0x28
; Setup the base pointer for the new stack frame leaving 8 bytes for padding (Stack alignment).
    lea    rbp,[rsp+0x20]
; Copy the fist argument to the allocated space in the stack frame - (10)
    mov    DWORD PTR [rbp+0x20],ecx
; Check to see if we achieved the base case of the argument value being less than three - (ecx < 3 ?)
    cmp    DWORD PTR [rbp+0x20],0x2
; If it's less than three, we jump to the end of the function - return
    jle    0x7ff65e9d1668 <fibonacci+52>
; If it's not we move the copied value to the eax register - (10)
    mov    eax,DWORD PTR [rbp+0x20]
; We subtract one from the copied value - (10-1 = 9)
    sub    eax,0x1
; We move the new value to the ecx register - the first argument for a function call
    mov    ecx,eax
; We call the fibonacci function on the new value - (9)
    call   0x7ff65e9d1634 <fibonacci>
; We store the value returned by the function call to the rbx register - Store fibonacci(9)
    mov    ebx,eax
; Copy the copied value on the stack to the eax register - (10)
    mov    eax,DWORD PTR [rbp+0x20]
; We subtract two from the copied value - (10-2 = 8)
    sub    eax,0x2
; We load it onto ecx - First argument for a function call
    mov    ecx,eax
; We call the fibonacci function on the new value - (8)
    call   0x7ff65e9d1634 <fibonacci>
; We add the value of rbx (First function call) to the eax register (Second function call)
    add    eax,ebx
; Jump unconditionally to the end of the function - (add   rsp,0x28)
    jmp    0x7ff65e9d166b <fibonacci+55>
; Load the return value of the function to the rax register
    mov    eax,DWORD PTR [rbp+0x20]
; Free the allocated memory from the stack
    add    rsp,0x28
; Retrieve the old value of the rbx register
    pop    rbx
; Retrieve the return address of the caller function
    pop    rbp
; return control to the caller function
    ret
```

Illustration

Main Memory:



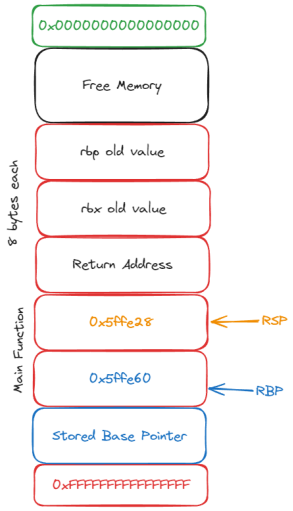
Allocating memory for the new stack frame for the main function:



Passing in the argument for the fibonacci function to the ecx register:

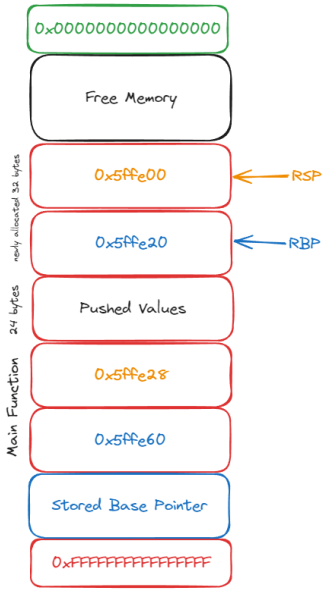
```
Register
rax = 0x0000000000000001 [1]
rbx = 0x0000000000000000 [0]
rcx = 0x000000000000000a [10]
rdx = 0x00000000001e14e0 [1971424]
rsi = 0x0000000000000000 [0]
rdi = 0x0000000000000000 [0]
rsp = 0x00000000005ffe40 [0x5ffe40]
rbp = 0x00000000005ffe60 [0x5ffe60]
```

Calling the fibonacci function (10) – Pushing rbx & rbp :



Calling the fibonacci function (10) – Allocating memory for the stack frame (Pushing) :

(Allocated 8 extra bytes for stack alignment)



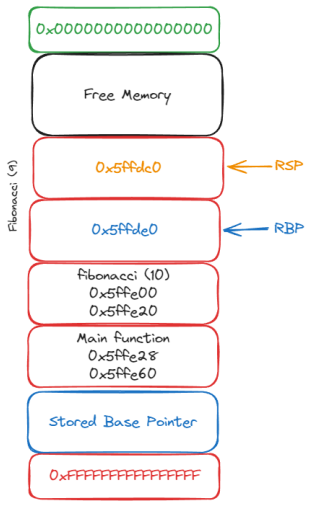
Calling the fibonacci function (10) – Loading the argument and comparing it to the base case :

```
Registers:
 rcx = 0x000000000000000a [10]
Local Variables:
 n = {int} 10
```

Calling the fibonacci function (10) – Loading the argument into eax and subtracting one from it:

```
Registers:
$ecx = 0x0000000a [10]
$eax = 0x00000009 [9]
$rbp = 0x00000000005ffe20 [0x5ffe20]
$rsp = 0x00000000005ffe00 [0x5ffe00]
$rbx = 0x0000000000000000 [0]
Local variables:
n = {int} 10
```

Calling the fibonacci function (9) – Allocating space for the new stack frame (Pushing):



Calling the fibonacci function (9) – Loading the argument and comparing it to the base case:

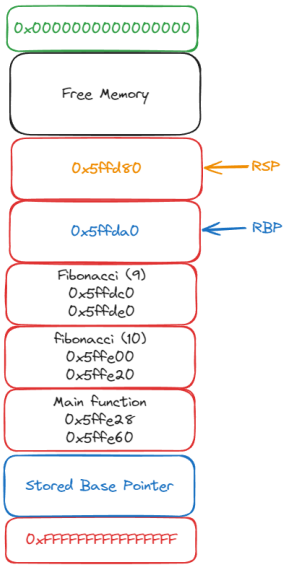
```
Registers
 rax = 0x0000000000000009 [9]
 rcx = 0x0000000000000009 [9]
n = {int} 9
```

Calling the fibonacci function (9) – Loading the argument into eax and subtracting 1 from it:

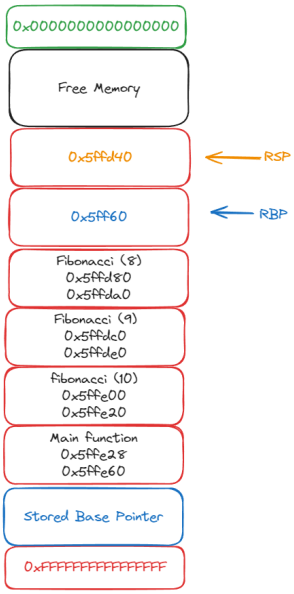
```
Registers
$ecx = 0x00000008 [8]
$eax = 0x00000008 [8]
$rbp = 0x00000000005ffde0 [0x5ffde0]
$rsp = 0x00000000005ffdc0 [0x5ffdc0]
$rbx = 0x0000000000000000 [0]

n = {int} 9
```

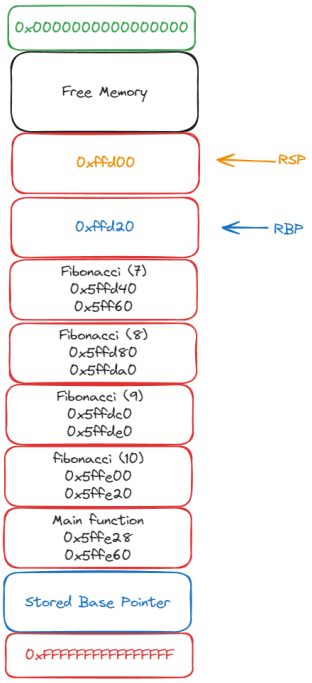
Calling the fibonacci function (8) (Pushing):



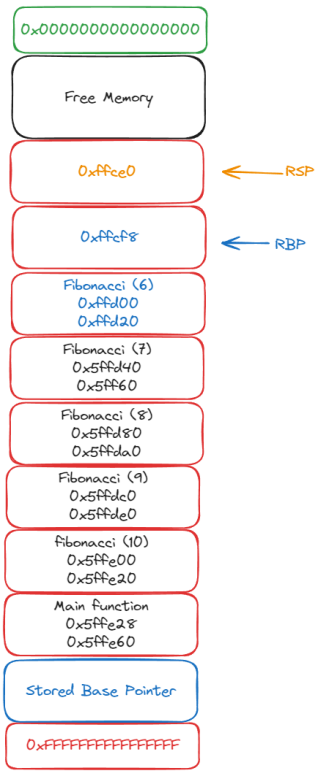
Calling the fibonacci function (7) (Pushing)



Calling the fibonacci function (6) (Pushing)

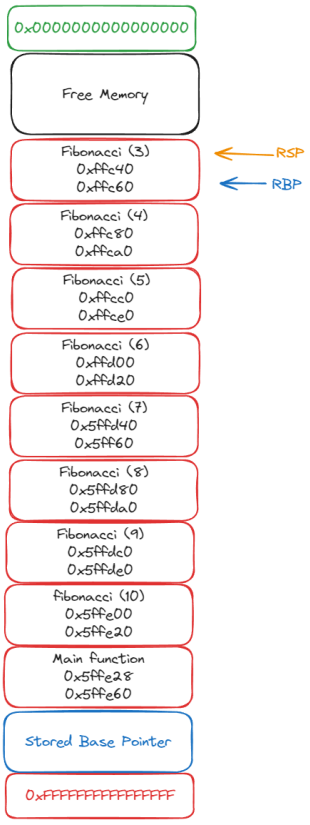


Calling the fibonacci function (5) (Pushing)

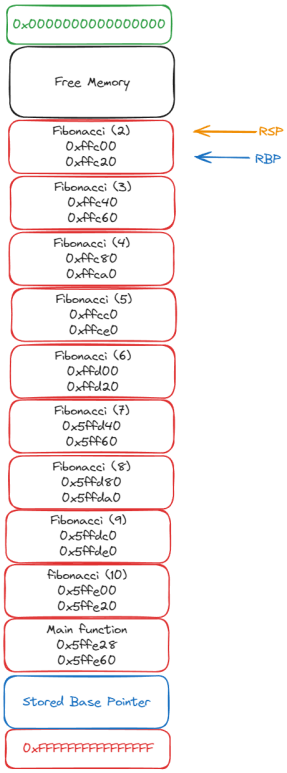


Calling the fibonacci function (4) (Pushing)

Calling the fibonacci function (3) (Pushing)



Calling the fibonacci function (2) (Pushing)



Calling the fibonacci function (2) – Comparing to the base case:

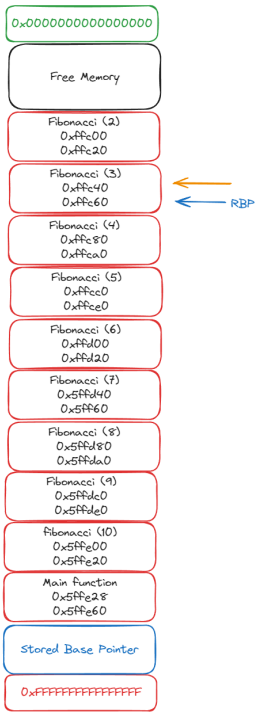
```
Registers
 rcx = 0x0000000000000002 [2]
```

Calling the fibonacci function (2) – Loading the return value to eax:

```
Registers
n = {int} 2
$rsp = 0x00000000005ffc00 [0x5ffc00]
$eax = 0x00000002 [2]
$ecx = 0x00000002 [2]
```

Calling the fibonacci function (2) – popping the stack frame:

By incrementing rsp and retrieving the old value for rbp (popping it from the stack)



Calling the fibonacci function (3) – Copying the retrieved value to rbx to store it:

```
Registers
n = {int} 3
$eax = 0x00000002 [2]
$ebx = 0x00000002 [2]
```

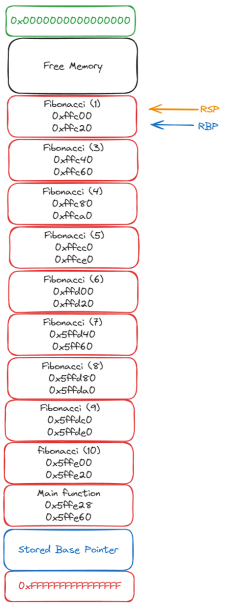
Calling the fibonacci function (3) – Moving the stored value (3) to eax and subtracting 2 from it:

```
Registers
n = {int} 3
$eax = 0x00000001 [1]
```

Calling the fibonacci function (3) – Copying it to ecx to pass it as the first argument:

```
Registers
n = {int} 3
$ecx = 0x00000001 [1]
$eax = 0x00000001 [1]
```

Calling the fibonacci function (1):



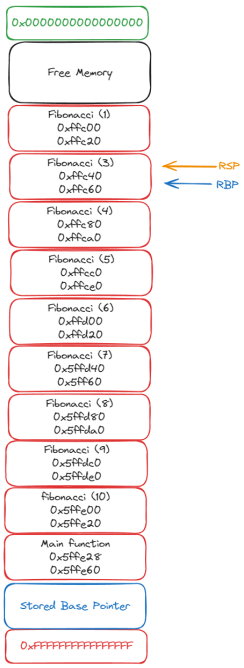
Calling the fibonacci function (1) – Comparing it to the base case:

```
Registers
n = {int} 1
 rcx = 0x0000000000000001 [1]
```

Calling the fibonacci function (1) – Moving the return value to the eax register:

```
Registers
n = {int} 1
$eax = 0x00000001 [1]
$ebx = 0x00000002 [2]
$ecx = 0x00000001 [1]
```

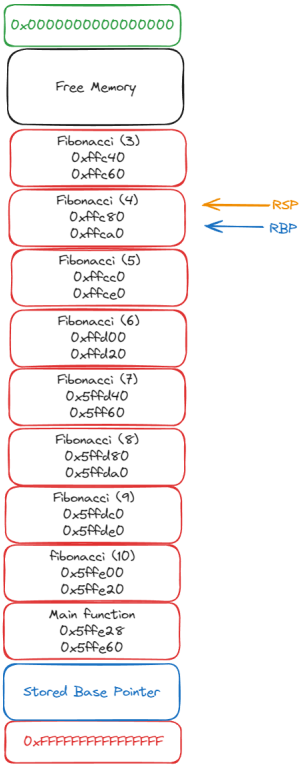
Calling the fibonacci function (1) – Popping the stack frame:



Calling the fibonacci function (3) – Adding the stored value (ebx) to the return value (eax):

```
Registers
n = {int} 3
$eax = 0x00000003 [3]
$ebx = 0x00000002 [2]
$ecx = 0x00000001 [1]
```

Calling the fibonacci function (3) – Popping the stack frame:



Calling the fibonacci function (4) – Adding the return value eax to ebx (0):

```
Registers
n = {int} 4
$eax = 0x00000003 [3]
$ebx = 0x00000003 [3]
```

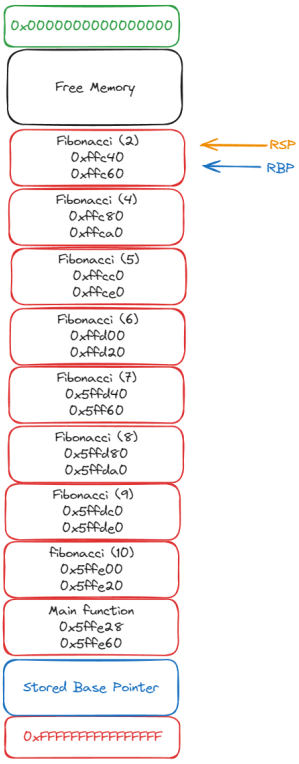
Calling the fibonacci function (4) – moving the stored value (4) to eax and subtracting 2 from it:

```
Registers
n = {int} 4
$ecx = 0x00000001 [1]
$eax = 0x00000002 [2]
$ebx = 0x00000003 [3]
```

Calling the fibonacci function (4) – Copying eax to ecx to pass it as the first argument:

```
Registers
n = {int} 4
$ecx = 0x00000002 [2]
$eax = 0x00000002 [2]
```

Calling the fibonacci function (2) (pushing):



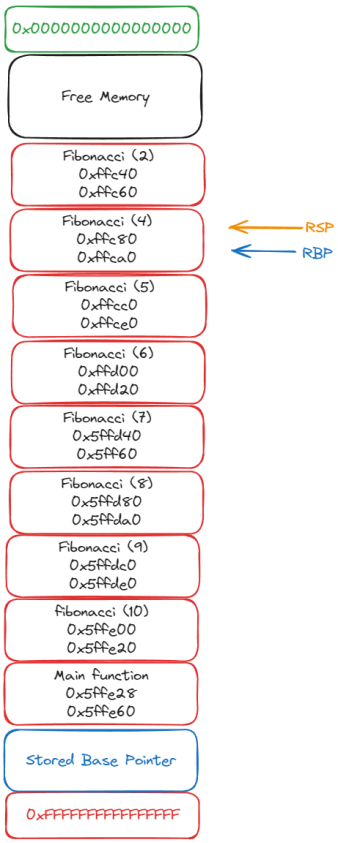
Calling the fibonacci function (2) – Comparing it to the base case:

```
Registers
n = {int} 2
rcx = 0x0000000000000002 [2]
```

Calling the fibonacci function (2) – Moving the return value to the eax register:

```
Registers
n = {int} 2
$eax = 0x00000002 [2]
$ebx = 0x00000003 [3]
$ecx = 0x00000002 [2]
```

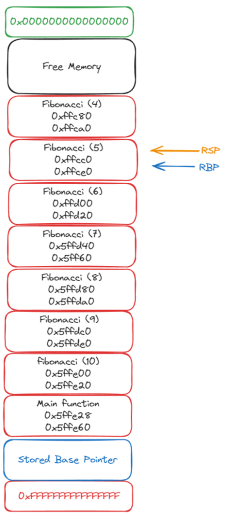
Calling the fibonacci function (2) – Popping the stack frame:



Calling the fibonacci function (4) – Adding the stored value (ebx) to the returned value (eax):

```
Registers
n = {int} 4
$eax = 0x00000005 [5]
$ebx = 0x00000003 [3]
$ecx = 0x00000002 [2]
```

Calling the fibonacci function (4) – Popping the stack frame:



Calling the fibonacci function (5) – Copying the returned value (eax) to the stored value rbx (0):

```
Registers
n = {int} 5
$eax = 0x00000005 [5]
$ebx = 0x00000005 [5]
$ecx = 0x00000002 [2]
```

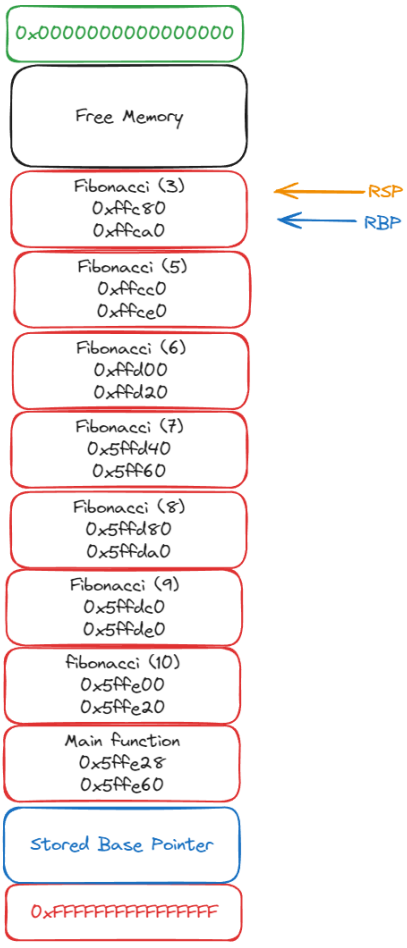
Calling the fibonacci function (5) – Copying the stored value (5) to eax and subtracting 2 from it:

```
Registers
n = {int} 5
$ecx = 0x00000002 [2]
$eax = 0x00000003 [3]
$ebx = 0x00000005 [5]
```

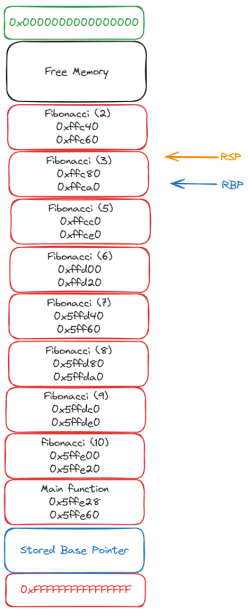
Calling the fibonacci function (5) – Copying the eax to ecx to pass it as the first argument:

```
Registers
n = {int} 5
$ecx = 0x00000003 [3]
$eax = 0x00000003 [3]
$ebx = 0x00000005 [5]
```

Calling the fibonacci function (3) – Pushing the stack frame:



Calling the fibonacci function (2) – Pushing and popping the stack frame:



Calling the fibonacci function (3) – Copying the return value (eax) to the stored value (rbx):

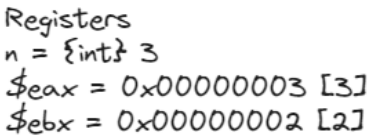
```
Registers
n = {int} 3
$eax = 0x00000002 [2]
$ebx = 0x00000002 [2]
$ecx = 0x00000002 [2]
```

Calling the fibonacci function (3) – Copying the stored value (3) to eax and subtracting 2 from it:

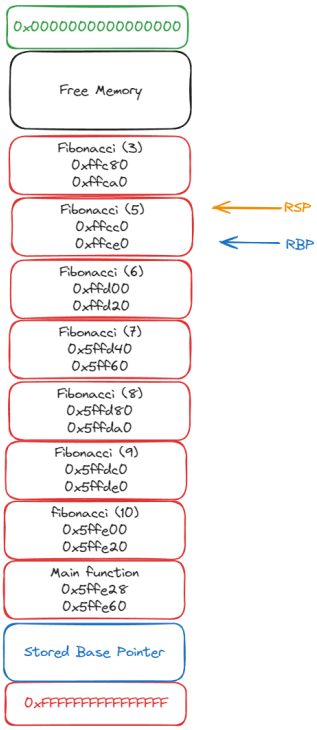
```
Registers
n = {int} 3
$ecx = 0x00000002 [2]
$eax = 0x00000001 [1]
$ebx = 0x00000002 [2]
```

Calling the fibonacci function (1) – Pushing and Popping the stack frame:

Calling the fibonacci function (3) – Adding the stored value (rbx) to the returned value (eax):



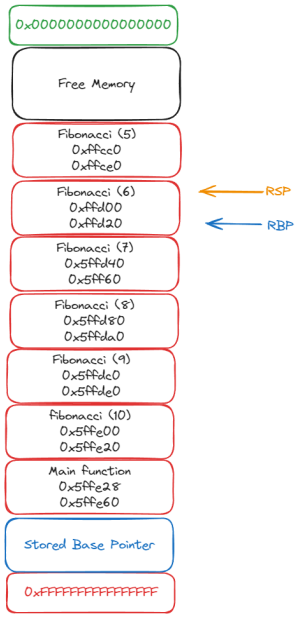
Calling the fibonacci function (3) – Popping the stack frame:



Calling the fibonacci function (5) – Adding the stored value (rbx) to the returned value (eax):

```
Registers
n = {int} 5
$eax = 0x00000008 [8]
$ebx = 0x00000005 [5]
```

Calling the fibonacci function (5) – Popping the stack frame:



Calling the fibonacci function (6) – Copying the returned value (eax) to the stored value (ebx):

```
Registers
n = {int} 6
$eax = 0x00000008 [8]
$ebx = 0x00000008 [8]
```

Calling the fibonacci function (6) – Copying the stored value (6) to eax and subtracting 2 from it:

```
Registers
n = {int} 6
$ecx = 0x00000001 [1]
$eax = 0x00000004 [4]
```

Calling the fibonacci function (6) – Copying eax to ecx to pass it as the first argument:

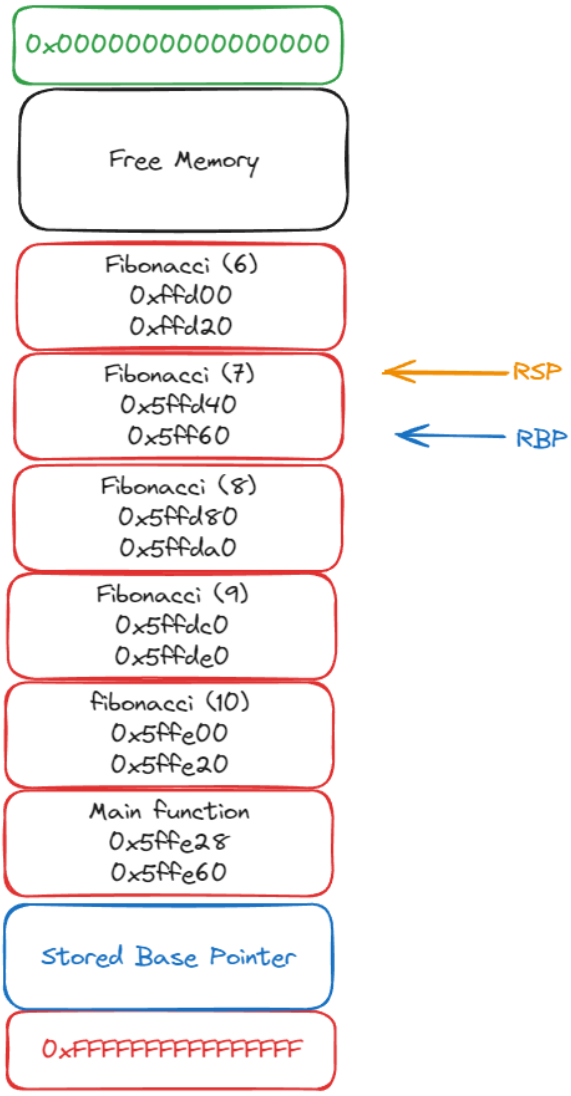
```
Registers
n = {int} 6
$ecx = 0x00000004 [4]
$eax = 0x00000004 [4]
```

Calling the fibonacci function (4) – Same Procedure as mentioned above

Calling the fibonacci function (6) – Copying the stored value (rbx) to the returned value (eax):

```
Registers
n = {int} 6
$eax = 0x0000000d [13]
$ebx = 0x00000008 [8]
```

Calling the fibonacci function (6) – Popping the stack frame:



Calling the fibonacci function (7) – Copying the stored value (7) to eax and subtracting 2 from it:

```
Registers
n = {int} 7
$ecx = 0x00000002 [2]
$eax = 0x00000005 [5]
```

Calling the fibonacci function (7) – Copying the stored eax to ecx to pass it as the first argument:

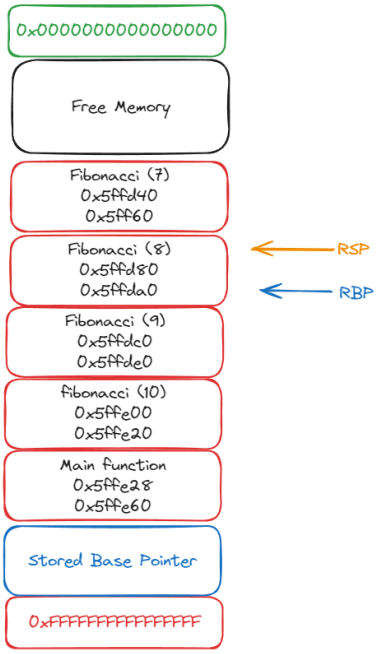
```
Registers
n = {int} 7
$ecx = 0x00000005 [5]
$eax = 0x00000005 [5]
```

Calling the fibonacci function (5) – Same Procedure as mentioned above

Calling the fibonacci function (7) – Adding the stored value (ebx) to the returned value (eax):

```
Registers
n = {int} 7
$eax = 0x00000015 [21]
$ebx = 0x0000000d [13]
```

Calling the fibonacci function (7) – Popping the stack frame:



Calling the fibonacci function (8) – copying the return value (eax) to the stored value (ebx):

```
Registers
n = {int} 8
$eax = 0x00000015 [21]
$ebx = 0x00000015 [21]
```

Calling the fibonacci function (8) – Copying the stored value (8) to eax and subtracting 2 from it:

```
Registers
n = {int} 8
$ecx = 0x00000001 [1]
$eax = 0x00000006 [6]
```

Calling the fibonacci function (8) – Copying eax to ecx to pass it as a function argument:

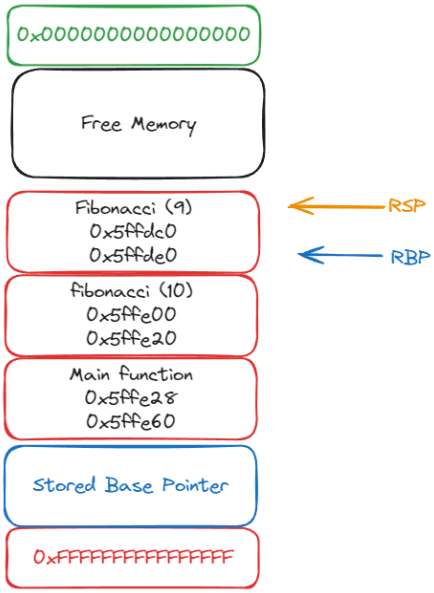
```
Registers
n = {int} 8
$ecx = 0x00000006 [6]
$eax = 0x00000006 [6]
```

Calling the fibonacci function (6) – Same Procedure as mentioned above

Calling the fibonacci function (8) – Adding the stored value (ebx) to the returned value (eax):

```
Registers
n = {int} 8
$eax = 0x00000022 [34]
$ebx = 0x00000015 [21]
```

Calling the fibonacci function (8) – Popping the stack frame:



Calling the fibonacci function (9) – Copying the returned value (eax) to the stored value (rbx):

```
Registers
n = {int} 9
$eax = 0x00000022 [34]
$ebx = 0x00000022 [34]
$ecx = 0x00000002 [2]
```

Calling the fibonacci function (9) – Copying the stored value (9) to eax and subtracting 2 from it:

```
Registers
n = {int} 9
$ecx = 0x00000002 [2]
$eax = 0x00000007 [7]
```

Calling the fibonacci function (9) – Copying eax to ecx to pass it as a function argument:

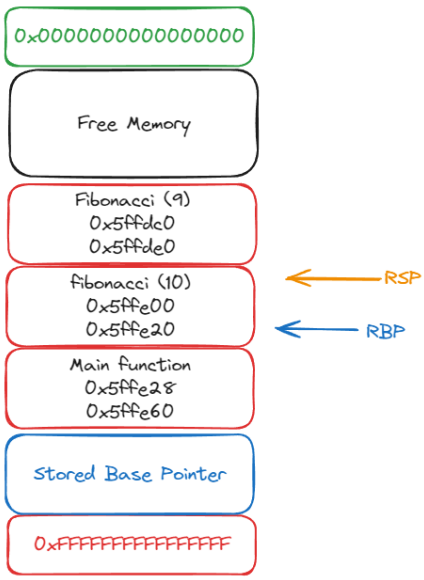
```
Registers
n = {int} 9
$ecx = 0x00000007 [7]
$eax = 0x00000007 [7]
```

Calling the fibonacci function (7) – Same Procedure as mentioned above

Calling the fibonacci function (9) – Copying the stored value (ebx) to the returned value (eax):

```
Registers
n = {int} 9
$eax = 0x00000037 [55]
$ebx = 0x00000022 [34]
```

Calling fibonacci function (9) – Popping the stack frame:



Calling the fibonacci function (10) – Copying the returned value (eax) to the stored value (rbx):

```
Registers
n = {int} 10
$ebx = 0x00000000 [0]
$eax = 0x00000037 [55]
```

Calling the fibonacci function (10) – Copying the stored value (10) to eax and subtracting 2 from it:

```
Registers
n = {int} 10
$ecx = 0x00000001 [1]
$eax = 0x00000008 [8]
```

Calling the fibonacci function (10) – Copying eax to ecx to pass it as a function argument:

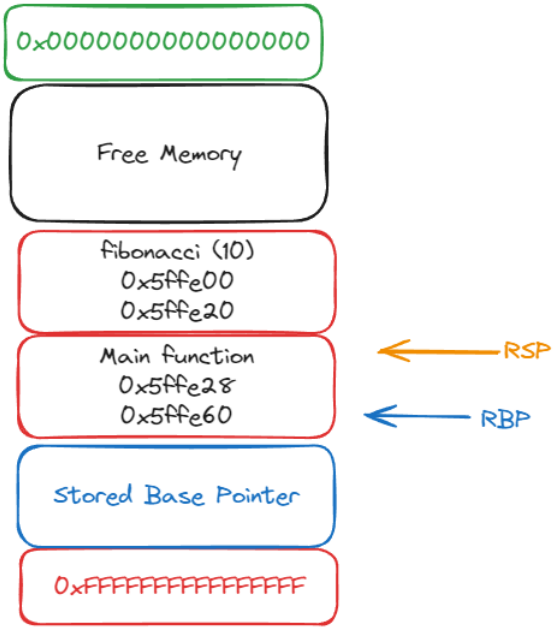
```
Registers
n = {int} 10
$ecx = 0x00000008 [8]
$eax = 0x00000008 [8]
```

Calling the fibonacci function (8) – Same Procedure as mentioned above

Calling the fibonacci function (10) – Copying the stored value (rbx) to the returned value (eax):

```
Registers
n = {int} 10
$eax = 0x00000059 [89]
$ebx = 0x00000037 [55]
$ecx = 0x00000002 [2]
```

Calling the fibonacci function (10) – Popping the stack frame:



Main function (Preparing to call printf) – Loading the returned value (eax) to the edx register

The edx register serves as the second argument to a function call

```
Registers
$edx = 0x00000059 [89]
$eax = 0x00000059 [89]
```

Main function (Preparing to call printf) – Loading the format string address to rax:

```
Registers
$rcx = 0x0000000000000002 [2]
$rax = 0x00007ff6ac5cd000 [140697430446080]
```

Main function (Preparing to call printf) – Copying the value of rax to rcx (function argument):

```
Registers
$rcx = 0x00007ff6ac5cd000 [140697430446080]
$rax = 0x00007ff6ac5cd000 [140697430446080]
```

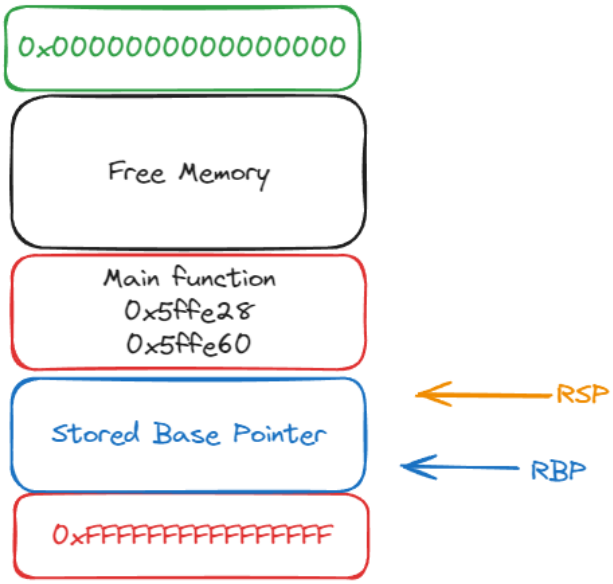
Main function – Calling the printf function and examining the return value:

```
Registers
$eax = 0x00000012 [18]
```

Main function – Loading in the return value:

```
Registers
$eax = 0x00000000 [0]
```

Main function – Popping the stack frame:



Sorting Algorithms

Merge Sort

Time Complexity:

Best, Average, and Worst Cases: O(n log n)

Merge Sort consistently divides the array in half and takes linear time to merge two halves.

Space Complexity:

Worst Case: O(n)

Merge Sort requires additional space for the temporary arrays used during the merger process.

Efficiency:

Highly efficient for large data sets due to its consistent O(n log n) time complexity.

Offers significant performance benefits over less efficient algorithms, especially as the size of the data set increases.

The divide-and-conquer approach helps in parallelizing the algorithm, making it suitable for multi-threaded processing.

Stability and Usage:

Merge Sort is stable; it preserves the order of equal elements.

It is often preferred for sorting linked lists and for external sorting (e.g., with large data sets that do not fit into memory).

Bubble Sort

Time Complexity:

Best Case (already sorted): O(n)

Average and Worst Cases (random order or reverse order): O(n^2)

Bubble Sort repeatedly compares and swaps adjacent elements, which is inefficient for larger arrays.

Space Complexity:

Worst Case: O(1)

Bubble Sort is an in-place sorting algorithm, requiring no additional space beyond the input array.

Efficiency:

Inefficient for large data sets due to its quadratic time complexity.

Performance drastically decreases as the size of the data set increases.

Stability and Usage:

Bubble Sort is stable.

It is simple to implement but rarely used in practice due to poor efficiency. It's typically used for educational purposes or in cases where simplicity is more critical than performance and the data set is known to be small.

Comparative Analysis

Time Complexity:

Merge Sort is far superior to Bubble Sort in terms of time complexity. Its O(n log n) performance is consistent for all cases, while Bubble Sort suffers from O(n^2) in average and worst cases.

Space Complexity:

While Merge Sort uses additional space (O(n)), Bubble Sort has the advantage of being an in-place sort with O(1) space complexity.

Overall Efficiency:

Merge Sort is significantly more efficient than Bubble Sort for medium to large data sets.

Bubble Sort may only be competitive with Merge Sort for very small arrays, or when the array is already nearly sorted.

Use Cases:

Merge Sort is suited for complex, large-scale sorting tasks where stability is needed.

Bubble Sort is generally avoided for practical applications due to its inefficiency, except for educational purposes or extremely small or nearly sorted data sets.

In conclusion, while Merge Sort is a clear choice for most practical applications due to its superior time efficiency and scalability, Bubble Sort's simplicity might only make it suitable for specific scenarios with minimal sorting needs.

Code Implementation and Comparison

Bubble Sort:

```
void bubbleSort(int* array, int length){
    for(int i = 0; i < length; i++){
        for(int j = 1 ; j < length-i; j++){
            if(array[j] < array[j-1]) swap(&array[j], &array[j-1]);
        }
    }
}
```

Merge Sort:

```
void mergeSort(int *array, int length) {
    sort(array, 0, length-1);
}

void merge(int *array, int left, int mid, int right) {
    int firstArraySize = mid - left + 1;
    int secondArraySize = right-mid;

    int firstArray[firstArraySize];
    int secondArray[secondArraySize];

    for(int i = 0; i < firstArraySize; i++) firstArray[i] = array[left + i];
    for(int i = 0; i < secondArraySize; i++) secondArray[i] = array[mid + 1 + i];

    int firstArrayIndex = 0;
    int secondArrayIndex = 0;
    int mainArrayIndex = left;

    while(firstArrayIndex < firstArraySize && secondArrayIndex < secondArraySize){
        if(firstArray[firstArrayIndex] <= secondArray[secondArrayIndex]){
            array[mainArrayIndex] = firstArray[firstArrayIndex];
            firstArrayIndex++;
        }else{
            array[mainArrayIndex] = secondArray[secondArrayIndex];
            secondArrayIndex++;
        }
        mainArrayIndex++;
    }
    while(firstArrayIndex < firstArraySize){
        array[mainArrayIndex] = firstArray[firstArrayIndex];
        firstArrayIndex++;
        mainArrayIndex++;
    }
    while(secondArrayIndex < secondArraySize){
        array[mainArrayIndex] = secondArray[secondArrayIndex];
        secondArrayIndex++;
        mainArrayIndex++;
    }

}

void sort(int *array, int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;

    sort(array, left, mid);
    sort(array, mid + 1, right);

    merge(array, left, mid, right);
}
```

Time comparison – 100 entries

```
Bubble Sort: 0.000014 second
Merge Sort: 0.000005 second
```

Time comparison – 1000 entries

```
Bubble Sort: 0.001152 second
Merge Sort: 0.000070 second
```

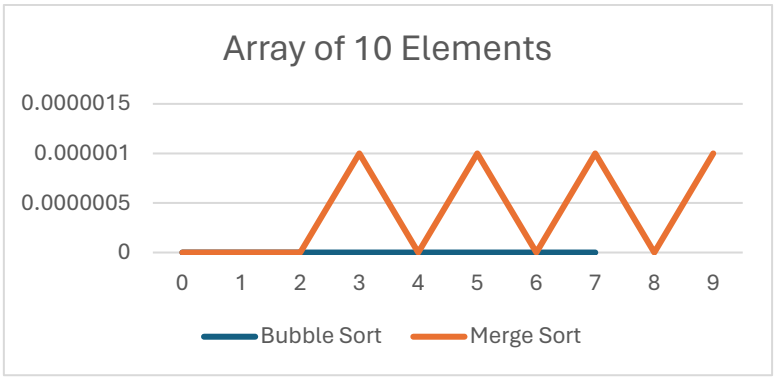
Time comparison – 10000 entries

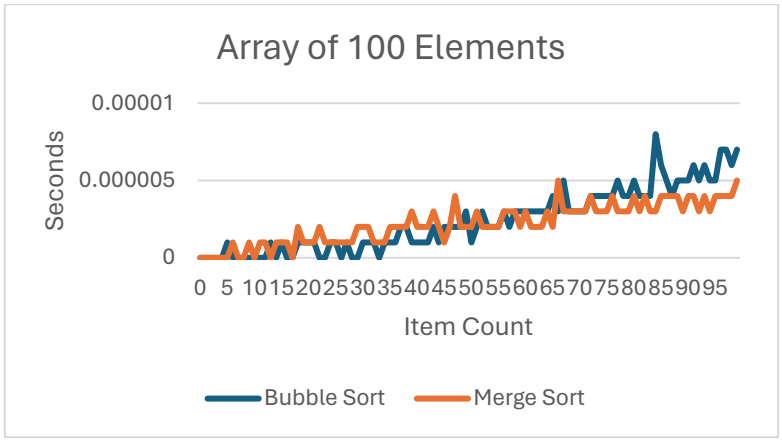
```
Bubble Sort: 0.112366 second
Merge Sort: 0.000611 second
```

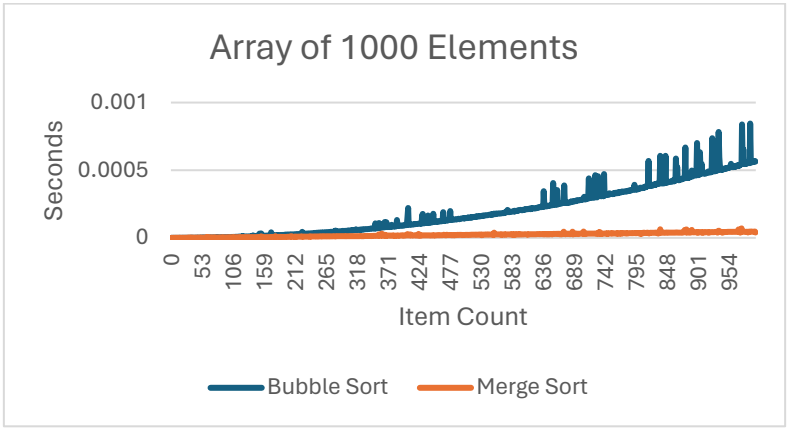
Time comparison – 100000 entries

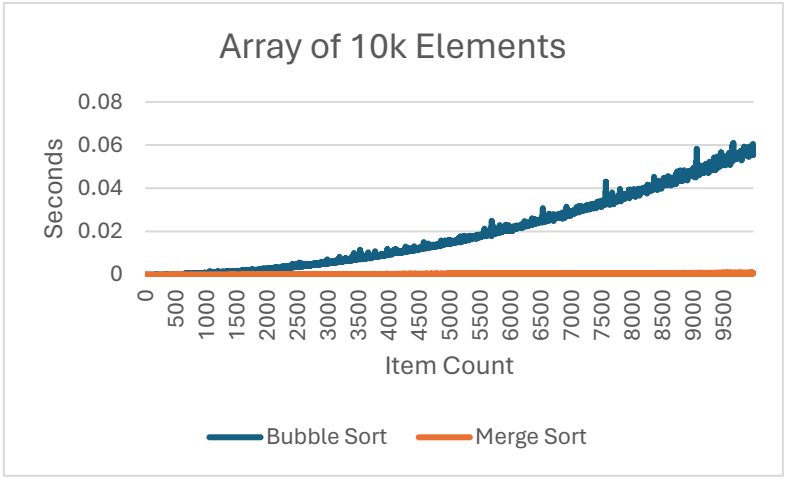
```
Bubble Sort: 18.562691 second
Merge Sort: 0.006243 second
```

Comparison Graphs & Analysis









Conclusion

The comparative performance analysis of Bubble Sort and Merge Sort over varying sizes of data sets reveals distinct trends in efficiency and scalability:

Performance on Small Data Sets (0 - 100 Elements):

For very small arrays (up to 10 elements), Merge Sort shows some speed fluctuations, whereas Bubble Sort maintains a stable performance. This can be attributed to the inherent overhead of Merge Sort in managing smaller arrays.

As the array size increases to 100 elements, both algorithms perform similarly in the initial range. However, towards the end of this range (>95 elements), Bubble Sort's performance starts to degrade, indicating the beginning of its inefficiency as the data size grows.

Performance on Medium to Large Data Sets (1,000 - 10,000 Elements):

With 1,000 elements, the performance gap becomes more pronounced. Bubble Sort exhibits a significant decrease in speed, particularly after 318 elements, showcasing its inefficiency with larger data sets. Merge Sort, on the other hand, maintains consistent speed with minor random fluctuations.

This trend is further amplified in the 10,000-element data set. Bubble Sort's speed drops exponentially, highlighting its impracticality for larger arrays. Merge Sort continues to demonstrate a stable and efficient performance, unaffected by the increasing data size. The inefficiency of Bubble Sort is evident from the 2,000-element mark.

Extensive Performance Gap in Very Large Data Sets (100,000 Elements):

The performance disparity is most stark with 100,000 entries. Bubble Sort takes a significantly longer time (over 18 seconds), while Merge Sort completes the sorting in just a fraction of that time (0.006 seconds). This indicates that Bubble Sort becomes increasingly impractical as data sizes grow, while Merge Sort remains highly efficient and scalable.

Overall Analysis:

The data conclusively shows that while Bubble Sort may provide stable performance for very small data sets, its time complexity becomes a major limitation as the array size increases. Merge Sort, with its consistently low time complexity, outperforms Bubble Sort in medium to large data sets, affirming its suitability for more extensive and diverse sorting tasks. The stability and efficiency of Merge Sort make it a superior choice for most practical applications requiring sorting, especially for larger data sets.

Asymptotic analysis (Asymptotics)

Asymptotic analysis Introduction

Asymptotic analysis is a fundamental concept in computer science, particularly in the field of algorithm analysis. It refers to the mathematical study of the behavior of functions as they tend toward infinity or some limiting value. In the context of algorithms, asymptotic analysis is crucial for understanding and predicting the performance and efficiency of algorithms, especially regarding their time and space requirements.

The primary goal of asymptotic analysis is to provide a framework for comparing algorithms in terms of their efficiency and scalability, independent of hardware or software considerations. This analysis focuses on the growth rate of an algorithm's resource requirements (like time or memory) in relation to the size of its input. By abstracting away from specific details and focusing on general trends, asymptotic analysis allows for a more objective and universally applicable evaluation of algorithms.

Key concepts used in asymptotic analysis include:

> Big O Notation (O-notation): This is the most commonly used notation to describe the upper bound of an algorithm's running time or space requirement. It provides a worst-case scenario analysis, giving the maximum amount of resources the algorithm might need.
>
> Big Ω Notation (Omega-notation): This notation describes the lower bound, giving the minimum resource requirement of an algorithm.
>
> Big Θ Notation (Theta-notation): This notation is used when an algorithm's upper and lower bounds are the same, providing a tight bound on its asymptotic growth rate.

Asymptotic analysis is particularly useful because it allows algorithms to be evaluated based on their performance trends as the input size grows infinitely large. This approach helps in identifying the most efficient algorithms for large-scale problems, making it a cornerstone of algorithm design and optimization.

Asymptotic analysis in relation to algorithms and ADTs

Understanding Efficiency and Scalability

Time Complexity Analysis:

Asymptotic analysis provides a clear picture of how the execution time of an algorithm grows with the input size.

In the previous task, Merge Sort's O(n log n) complexity compared to Bubble Sort's O(n^2) highlighted the former's superior scalability and efficiency for large datasets.

This analysis is crucial for understanding whether an algorithm is suitable for practical use, especially when dealing with large volumes of data.

Predicting Performance Trends:

The asymptotic behavior of algorithms helps predict how they will perform as the data scales. For instance, knowing that Bubble Sort deteriorates significantly with larger datasets allows developers to choose more efficient algorithms like Merge Sort for applications that require handling extensive data.

Assessing ADT Suitability

ADT Operations Complexity:

Algorithms are often designed to manipulate specific ADTs (like arrays, linked lists, trees, etc.). Asymptotic analysis shows how the choice of ADT affects the algorithm's performance.

For example, Merge Sort is more efficient on arrays due to its divide-and-conquer approach, which is not as straightforward to implement efficiently on linked lists.

Optimizing Data Structure Choices:

Understanding the asymptotic complexities of various operations (like insertion, deletion, searching) on different ADTs helps in choosing the most suitable data structure for a given problem.

For instance, if frequent insertions and deletions are required, a linked list might be preferred over an array despite the better average search time in sorted arrays.

Decision Making in Algorithm Design

Balancing Trade-offs:

Asymptotic analysis highlights the trade-offs between time and space complexity. An algorithm efficient in time might consume more space and vice versa.

For example, Merge Sort uses additional space for merging, which is a trade-off for its faster time complexity compared to Bubble Sort.

Algorithm Adaptation and Improvement:

By understanding the asymptotic behavior, developers can optimize algorithms or choose adaptive algorithms that change their strategy based on input size or data nature.

Comparative Analysis for Specific Use-Cases:

Asymptotic analysis is not just about finding the fastest algorithm in general but about finding the most suitable one for specific scenarios. For example, Bubble Sort might still be used for small arrays due to its simplicity despite its poor performance with large arrays.

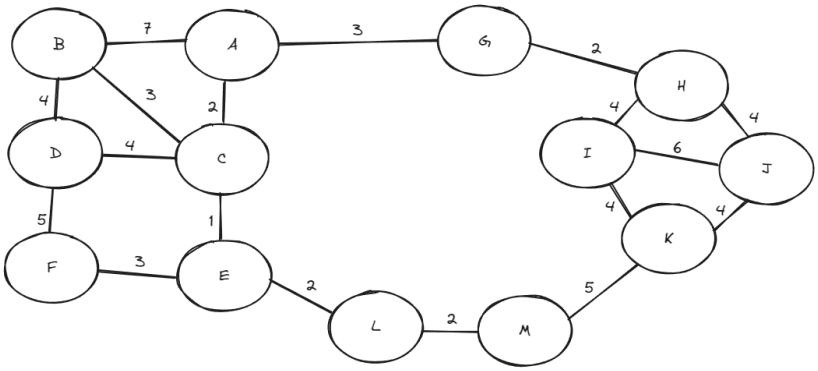
Conclusion

Asymptotic analysis is an invaluable tool in algorithm and ADT assessment. It provides essential insights into the scalability, efficiency, and practicality of algorithms in various scenarios. This analysis guides developers in making informed decisions about algorithm selection and ADT implementation, ensuring optimal performance and resource utilization for their specific applications and constraints.

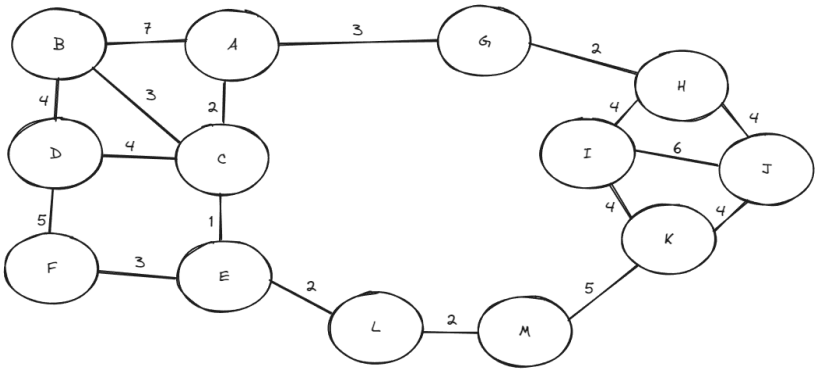
Dijkstra Vs. Bellman-Ford

Starting Graph

We will be starting with an undirected-weighted graph to visualize and compare both algorithms



Dijkstra's Algorithm



Starting Node: A

Ending Node: M

To implement Dijkstra's algorithm on this graph we need to keep track of a couple things:

1. The node's predecessor
2. The visited nodes

Initializing the distances while also keeping track which nodes we visited (V):

A (V)	0	D	∞	G	∞	J	∞	M	∞
B	∞	E	∞	H	∞	K	∞		
C	∞	F	∞	I	∞	L	∞		

Getting the distances for the neighboring nodes (B, C, G):

A	0	D	∞	G	3	J	∞	M	∞
B	7	E	∞	H	∞	K	∞		
C	2	F	∞	I	∞	L	∞		

Keeping Track of the node's predecessors:

A	A	D	Ø	G	A	J	Ø	M	Ø
B	A	E	Ø	H	Ø	K	Ø		
C	A	F	Ø	I	Ø	L	Ø		

Now we pick the closest node to our current one which is C and follow the same procedure:

A(V)	0	D	6	G	3	J	∞	M	∞
B	5	E	3	H	∞	K	∞		
C(V)	2	F	∞	I	∞	L	∞		

What we did is we added the known path weight for our new node (2) to the newly discovered weights of the adjacent nodes, which evidently gave us the weight for the whole route.

We also noticed that we have a new shortest path for the node $B,$ so we proceed to update that as well.

Updating our node's predecessor table:

A	A	D	C	G	A	J	Ø	M	Ø
B	C	E	C	H	Ø	K	Ø		
C	A	F	Ø	I	Ø	L	Ø		

Following we proceed to change our current working node to the second closest one from our starting node (G) and doing the same procedure:

A(V)	0	D	6	G(V)	3	J	∞	M	∞
B	5	E	3	H	5	K	∞		
C(V)	2	F	∞	I	∞	L	∞		

Updating our node's predecessor table:

A	A	D	C	G	A	J	Ø	M	Ø
B	C	E	C	H	G	K	Ø		
C	A	F	Ø	I	Ø	L	Ø		

We proceed then by picking the next closest node (E) and following the same procedure:

A(V)	0	D	6	G(V)	3	J	∞	M	∞
B	5	E(V)	3	H	5	K	∞		
C(V)	2	F	6	I	∞	L	5		

Updating our node's predecessor table:

A	A	D	C	G	A	J	Ø	M	Ø
B	C	E	C	H	G	K	Ø		
C	A	F	E	I	Ø	L	E		

Now we check the next closest one which is (B), B only leads to D and $C,$ comparing the new path from B to our existing path, doesn't lend us a new shortest path, so the table is left unchanged, and we mark B as a visited node.

Proceeding to check the next closest one which is (H) and updating the table accordingly:

A(V)	0	D	6	G(V)	3	J	9	M	∞
B(V)	5	E(V)	3	H(V)	5	K	∞		
C(V)	2	F	6	I	9	L	5		

Updating our node's predecessor table:

A	A	D	C	G	A	J	H	M	Ø
B	C	E	C	H	G	K	Ø		
C	A	F	E	I	H	L	E		

Proceeding to check the next closest one which is (L) and updating the table accordingly:

A(V)	0	D	6	G(V)	3	J	9	M	7
B(V)	5	E(V)	3	H(V)	5	K	∞		
C(V)	2	F	6	I	9	L(V)	5		

Updating our node's predecessor table:

A	A	D	C	G	A	J	H	M	L
B	C	E	C	H	G	K	Ø		
C	A	F	E	I	H	L	E		

We have arrived at our destination and acquired the shortest path to it, now if our objective is to only find the shortest path to a single destination, we can stop here. If we wish to map all the shortest paths to every node in the graph, we shall continue.

Proceeding to check the next closest node which is (D) we can see that it leads to nodes (F) and (B), and the newly discovered paths aren't shorter than our already known ones, so we don't update the table and just mark the node as a visited one.

A(V)	0	D(V)	6	G(V)	3	J	9	M	7
B(V)	5	E(V)	3	H(V)	5	K	∞		
C(V)	2	F	6	I	9	L(V)	5		

Following up to check the next closest node which is (F), we can observe that the node (F) only leads us to the node (D), and the new path isn't shorter than our known one, so we don't update the able and just mark the node as visited.

A(V)	0	D(V)	6	G(V)	3	J	9	M	7
B(V)	5	E(V)	3	H(V)	5	K	∞		
C(V)	2	F(V)	6	I	9	L(V)	5		

Proceeding with the next closest node (M) and doing the same procedure:

A(V)	0	D(V)	6	G(V)	3	J	9	M(V)	7
B(V)	5	E(V)	3	H(V)	5	K	12		
C(V)	2	F(V)	6	I	9	L(V)	5		

Updating our node's predecessor table:

A	A	D	C	G	A	J	H	M	L
B	C	E	C	H	G	K	M		
C	A	F	E	I	H	L	E		

Proceeding to check the next closest node (I) we can see that it leads to the nodes (K) and (J), we can see that the newly acquired path isn't shorter than the already known path, so the table won't change, and we will just mark the node as visited.

A(V)	0	D(V)	6	G(V)	3	J	9	M(V)	7
B(V)	5	E(V)	3	H(V)	5	K	12		
C(V)	2	F(V)	6	I(V)	9	L(V)	5		

Checking the next closest node (J) we can see it also leads to nodes (I) and (K), but the newfound path isn't shorter than the already existing one, so we won't change the paths and just mark it as visited.

A(V)	0	D(V)	6	G(V)	3	J(V)	9	M(V)	7
B(V)	5	E(V)	3	H(V)	5	K	12		
C(V)	2	F(V)	6	I(V)	9	L(V)	5		

Following up on our last node (K) we can see that it leads us to nodes (J) and/or (I) and finally node (M) depending on which route we came from, observing the new paths we can see that they are not shorter than the already existing ones, thus we don't change the paths and just mark the node as visited.

A(V)	0	D(V)	6	G(V)	3	J(V)	9	M(V)	7
B(V)	5	E(V)	3	H(V)	5	K(V)	12		
C(V)	2	F(V)	6	I(V)	9	L(V)	5		

And with that we have mapped our whole graph using Dijkstra's algorithm for shortest paths, with our result being demonstrated in the following tables:

Weights:

A	0	D	6	G	3	J	9	M	7
B	5	E	3	H	5	K	12		
C	2	F	6	I	9	L	5		

Predecessors:

A	A	D	C	G	A	J	H	M	L
B	C	E	C	H	G	K	M		
C	A	F	E	I	H	L	E		

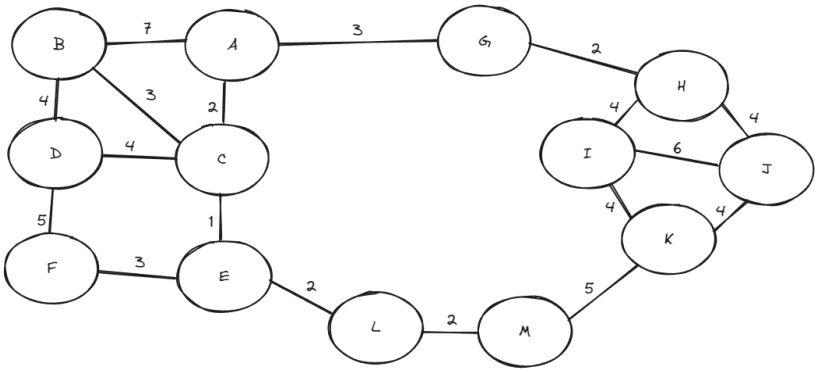
Combined:

NODE	DISTANCE	PREDECESSOR
A	0	A
B	5	C
C	2	A
D	6	C
E	3	C
F	6	E
G	3	A
H	5	G
I	9	H
J	9	H
K	12	M
L	5	E
M	7	L

And our final path starting from node (A) to node (M) is the following:

$$A \overset{2}{\rightarrow} C \overset{1}{\rightarrow} E \overset{2}{\rightarrow} L \overset{2}{\rightarrow} M \;\; SP = 7$$

Bellman Ford's Algorithm



Starting Node: A

Ending Node: M

We can structure our nodes in the following order for ease of demonstration:

Link	A,B	A,C	A,G	B,A	B,C	B,D	C,A	C,B	C,D	C,E
Distance	7	2	3	7	3	4	2	3	4	1
Link	D,B	D,C	D,F	E,C	E,F	E,L	F,D	F,E	G,A	G,H
Distance	4	4	5	1	3	2	5	3	3	2
Link	H,G	H,I	H,J	I,H	I,J	I,K	J,H	J,I	J,K	K,I
Distance	2	4	4	4	6	4	4	6	4	4
Link	K,J	K,M	L,E	L,M	M,K	M,L				
Distance	4	5	2	2	5	2				

Going through our first iteration:

Node	Distance	Predecessor
A	0	
B	~~7~~, 3	~~A~~, C
C	2	A
D	~~11~~, 6	~~B~~, C
E	3	C
F	~~11~~, 6	~~D~~, E
G	3	A
H	5	G
I	9	H
J	9	H
K	~~13~~, 12	~~I / J~~, M
L	5	E
M	~~18~~, 7	~~K~~, L

Going through our second iteration:

Link	A,B	A,C	A,G	B,A	B,C	B,D	C,A	C,B	C,D	C,E
Distance	7	2	3	7	3	4	2	3	4	1
Link	D,B	D,C	D,F	E,C	E,F	E,L	F,D	F,E	G,A	G,H
Distance	4	4	5	1	3	2	5	3	3	2
Link	H,G	H,I	H,J	I,H	I,J	I,K	J,H	J,I	J,K	K,I
Distance	2	4	4	4	6	4	4	6	4	4
Link	K,J	K,M	L,E	L,M	M,K	M,L				
Distance	4	5	2	2	5	2				

Node	Distance	Predecessor
A	0	
B	~~7~~, 3	~~A~~, C
C	2	A
D	~~11~~, 6	~~B~~, C
E	3	C
F	~~11~~, 6	~~D~~, E
G	3	A
H	5	G
I	9	H
J	9	H
K	~~13~~, 12	~~I / J~~, M
L	5	E
M	~~18~~, 7	~~K~~, L

The paths haven't changed during our second iteration, which means that we found the shortest paths, keep in mind that we have iterated in the same order as specified in the structure table, overwriting each path with its shorter equivalent, and thus we have found our shortest path from node (A) to node (M).

$$A \rightarrow C \rightarrow E \rightarrow L \rightarrow M$$

Time & Space Complexity

Dijkstra's Algorithm:

Time Complexity:

Without Priority Queue: $O(V^2)$, where V is the number of vertices. This is because in the worst case, every vertex is extracted from the queue and every extraction operation can go through all vertices to find the vertex with the minimum distance.

With Priority Queue (Binary Heap): $O((V + E) \log V)$, where E is the number of edges. The priority queue (min-heap) optimizes the process of finding the vertex with the minimum distance. Every vertex is added and extracted from the priority queue once.

Space Complexity:

O(V), since the algorithm maintains a set of vertices (or nodes) that have been visited, along with their current shortest distances. A priority queue is also used when the binary heap implementation is applied, which also stores up to V vertices.

Bellman-Ford Algorithm:

Time Complexity:

O(VE), where V is the number of vertices and E is the number of edges. This is because the algorithm relaxes all E edges in each of the V-1 iterations.

Space Complexity:

O(V), as the main storage is for the distances from the source to each vertex and the predecessor of each vertex.

Comparison:

Time Complexity:

Dijkstra's algorithm is generally faster, especially when implemented with a priority queue. It's more efficient for graphs with many edges due to its logarithmic factor as opposed to the linear factor in Bellman-Ford.

Bellman-Ford's O(VE) time complexity makes it slower, especially for dense graphs.

Space Complexity:

Both algorithms have similar space complexities of O(V), primarily storing information about each node.

Use Cases:

Dijkstra's algorithm is preferred for graphs without negative weight edges and where efficiency is a concern.

Bellman-Ford is suitable for graphs that may contain negative weight cycles, as it can detect these cycles. It's a more versatile algorithm but at the cost of higher time complexity.

Graph Type Suitability:

Dijkstra's algorithm doesn't work with graphs containing negative weight edges, as it can lead to incorrect results.

Bellman-Ford works with graphs having negative weights and can handle negative cycles.

While Dijkstra's algorithm is generally more efficient timewise, especially with a priority queue implementation, Bellman-Ford offers a broader applicability for graphs with negative weight edges. Both share similar space complexity.

Practical Considerations

1. Presence of Negative Weight Edges:

    Dijkstra's Algorithm is not suitable for graphs with negative weight edges because it assumes that once a node has been visited, there's no shorter path to it. This assumption fails in the presence of negative weights.

    Bellman-Ford Algorithm can handle negative weight edges and can detect negative weight cycles. If your graph can potentially include negative weights, Bellman-Ford is the go-to choice.

2. Performance and Efficiency:

    Dijkstra's Algorithm is more efficient, especially with a priority queue (using a binary heap, Fibonacci heap, etc.), making it suitable for large graphs with a lot of nodes and edges. Its time complexity is generally better than Bellman-Ford's.

    Bellman-Ford Algorithm has a higher time complexity (O(VE)) and is less efficient, particularly on dense graphs. It's a better choice for smaller graphs or situations where the increased computation time is not a critical issue.

3. Graph Density:

    For sparse graphs (few edges relative to the number of nodes), Dijkstra's algorithm is typically more efficient.

    For dense graphs (edges approaching the maximum possible number, close to V^2), Bellman-Ford's performance disadvantage compared to Dijkstra's might be less pronounced, but Dijkstra's often remains the more efficient choice unless negative weights are involved.

4. Real-time vs Offline Processing:

    If you need real-time route finding or graph traversal (like in GPS systems or network routing), Dijkstra's speed makes it more suitable.

    Bellman-Ford's slower performance might be acceptable in offline scenarios, such as certain types of network analysis or academic research, where negative weight cycles are a concern.

5. Implementation Complexity:

    Dijkstra's Algorithm is generally simpler to implement and understand, especially the version without a priority queue.

    Bellman-Ford Algorithm is a bit more complex due to its handling of negative weights and cycle detection, but it's still straightforward in terms of algorithmic logic.

6. Path Reconstruction:

    Both algorithms can reconstruct the shortest path, but the method of doing so can be simpler in Dijkstra's Algorithm when using a priority queue.

7. Specific Applications:

Dijkstra's Algorithm is widely used in networking for finding the shortest path in routing protocols, and in mapping services for route planning.

Bellman-Ford Algorithm is useful in financial applications for arbitrage detection (which involves negative cycles) and in certain academic or theoretical contexts where negative weight edges are common.

Part 3

How ADTs are the basis for Object Oriented Programming

Definitions

An Abstract Data Type (ADT) is a model for data types where the data type is defined by its behavior (semantics) from the point of view of a user, particularly in terms of possible values, possible operations on data of this type, and the behavior of these operations. Examples include stacks, queues, trees, and graphs.

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

Encapsulation

Both ADTs and OOP are based on the principle of encapsulation. In ADTs, data is hidden from the user, and they can only interact with it through a defined set of operations. Similarly, in OOP, objects encapsulate data and methods, hiding the internal state and requiring interaction through public methods.

Abstraction

ADTs are about abstracting away the details and focusing on what operations are available. OOP takes this a step further by not only providing a set of operations but also associating these operations with the relevant data. This leads to a more natural and understandable structure.

Modularity

ADTs provide a way to structure programs into discrete pieces of functionality. OOP expands on this by allowing for more complex structures like inheritance and polymorphism, enabling more modular and reusable code.

Examples and Justification

Consider a 'Stack' ADT. It defines operations like push(), pop(), and peek() without specifying how these operations are implemented. In OOP, a Stack class would implement these operations, and the internal representation (like using an array or linked list) would be hidden from the user.

Conclusion

ADTs laid the groundwork for the key principles of OOP such as encapsulation, abstraction, and modularity. They provide a conceptual framework that OOP builds upon to create more structured, reusable, and maintainable code. By understanding ADTs, one can appreciate the design and architecture of OOP, which extends these concepts to provide a more powerful and flexible way to organize and manage complex software systems.

Dependent Vs. Independent Data Structures

In the realm of computer science, the concept of implementation-independent data structures refers to data structures whose design and usage are not tied to a specific implementation or programming language. This contrasts with implementation-dependent data structures, which are designed with particular implementation details in mind.

Independent Data Structures

These data structures are defined in terms of their operations, properties, and behaviors rather than their underlying implementation. They are often expressed abstractly, like in Abstract Data Types (ADTs).

Examples: Stack, Queue, List, Tree, Graph. These can be implemented in various ways, such as using arrays, linked lists, or other underlying structures, but their fundamental behaviors and interfaces remain consistent across implementations.

Dependent Data Structures

Implementation-Dependent: These data structures are closely tied to specific implementation details or programming language features. Their usage and efficiency might be highly dependent on how they are implemented.

Examples: Array in C (size and type are fixed), Linked List using pointers (specific to languages that support pointers).

Differences

Flexibility and Portability:

> Independent structures are more flexible and can be adapted or reimplemented in different environments without changing their external behavior.
>
> Dependent structures are less flexible and might not be easily ported to different programming languages or environments.

Abstraction Level:

> Independent data structures operate at a higher level of abstraction, allowing users to focus on what the data structure does, rather than how it does it.
>
> Dependent data structures require understanding the underlying implementation to use them effectively.

Benefits of Using Implementation Independent Data Structures

Ease of Understanding and Use:

> Users of the data structure need not understand the complexities of its implementation. For example, a programmer using a Queue can focus on enqueue and dequeue operations without worrying about the underlying array or linked list implementation.

Reusability and Maintainability:

> Since the implementation details are abstracted away, these data structures can be reused across different projects and applications. Changes to the implementation (like switching from an array to a linked list for a Stack) do not affect the code that uses the stack.

Consistency Across Platforms:

> They provide a consistent interface across different programming languages and platforms. For instance, a Map or Dictionary in Java, C#, and Python may have different underlying implementations but offer similar operations like put, get, and remove.

Conclusion

Implementation-independent data structures are foundational in software development due to their abstraction, flexibility, and portability. They allow programmers to write more generic, reusable, and maintainable code by focusing on the interface and behavior of the data structures rather than their specific implementations.