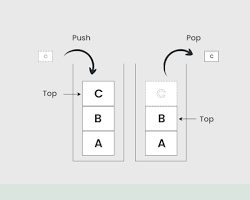
**1. Stack**

* **Definition:** LIFO (Last In, First Out) data structure, typically implemented using arrays or linked lists.
* **Operations:**
  + **Push:** Add an element to the top of the stack (constant time)
  + **Pop:** Remove the top element from the stack (constant time)
  + **Peek:** Get the value of the top element without removing it (constant time)
  + **IsEmpty:** Check if the stack is empty (constant time)
* **C Code (using an array):**

#include<stdio.h>

int stack[100], choice, n, top = -1, x, i;

void push(void);

void pop(void);

void display(void);

int main() {

printf("\n Enter the size of STACK[MAX=100]:");

scanf("%d", &n);

printf("\n\t STACK OPERATIONS USING ARRAY");

printf("\n\t--------------------------------");

printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");

do {

printf("\n Enter the Choice:");

scanf("%d", &choice);

switch(choice) {

case 1:

push();

break;

case 2:

pop();

break;

case 3:

display();

break;

case 4:

printf("\n\t EXIT POINT ");

break;

default:

printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");

}

} while(choice != 4);

return 0;

}

void push() {

if(top >= n - 1) {

printf("\n\tSTACK is over flow");

} else {

printf(" Enter a value to be pushed:");

scanf("%d", &x);

top++;

stack[top] = x;

}

}

void pop() {

if(top <= -1) {

printf("\n\t Stack is under flow");

} else {

printf("\n\t The popped elements is %d", stack[top]);

top--;

}

}

void display() {

if(top >= 0) {

printf("\n The elements in STACK \n");

for(i = top; i >= 0; i--)

printf("\n%d", stack[i]);

printf("\n Press Next Choice");

} else {

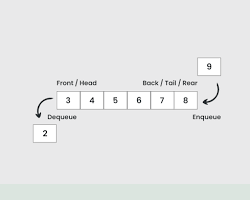
printf("\n The STACK is empty");

}

}

**2. Queue**

* **Definition:** FIFO (First In, First Out) data structure, typically implemented using arrays or linked lists.
* **Operations:**
  + **Enqueue:** Add an element to the back of the queue (constant time)
  + **Dequeue:** Remove the element from the front of the queue (constant time)
  + **Peek:** Get the value of the front element without removing it (constant time)
  + **IsEmpty:** Check if the queue is empty (constant time)

**CODE:**

**a.enqueue() or insertion():**

void insertion()

{

if(rear==max)

printf("\n Queue is Full");

else

{

printf("\n Enter no %d:",j++);

scanf("%d",&queue[rear++]);

}

}

**b.dequeue() or deletion():**

void deletion()

{

if(front==rear)

{

printf("\n Queue is empty");

}

else

{

printf("\n Deleted Element is

%d",queue[front++]);

x++ }

**c.dispaly()**

void display()

{

if(front==rear)

{

printf("\n Queue is empty");

}

else

{

for(i=front; i<rear; i++)

{

printf("%d",queue[i]);

printf("\n");

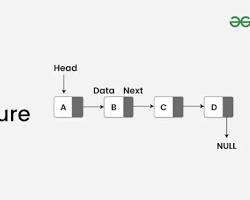
}

}

}

**3. Linked List**

* **Definition:** A collection of nodes, each containing data and a pointer to the next node. Offers dynamic resizing at runtime.
* **Operations:**
  + **Access:** Requires traversing the list to find the element (linear time on average)
  + **Insertion:** Can be done at the beginning, end, or in the middle (relatively efficient)
  + **Deletion:** Can be done by finding the node and relinking pointers (relatively efficient)
  + **Traversal:** Iterate through nodes by following pointers (linear time)

**CODE:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the linked list

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to insert a node at the beginning of the linked list

void insertAtBeginning(struct Node\*\* head\_ref, int data) {

struct Node\* newNode = createNode(data);

newNode->next = \*head\_ref;

\*head\_ref = newNode;

}

// Function to insert a node at the end of the linked list

void insertAtEnd(struct Node\*\* head\_ref, int data) {

struct Node\* newNode = createNode(data);

struct Node\* current = \*head\_ref;

if (\*head\_ref == NULL) {

\*head\_ref = newNode;

} else {

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

}

// Function to delete a node with a given key from the linked list

void deleteNode(struct Node\*\* head\_ref, int key) {

struct Node\* temp = \*head\_ref;

struct Node\* prev = NULL;

// If the key is found at the head

if (temp != NULL && temp->data == key) {

\*head\_ref = temp->next;

free(temp);

return;

}

// Search for the key to be deleted, keep track of the previous node as well

while (temp != NULL && temp->data != key) {

prev = temp;

temp = temp->next;

}

// If the key was not found

if (temp == NULL) {

printf("Key not found in the linked list.\n");

return;

}

// Unlink the node from the linked list

prev->next = temp->next;

free(temp);

}

// Function to display the linked list

void display(struct Node\* head) {

struct Node\* current = head;

printf("Linked List: ");

while (current != NULL) {

printf("%d ", current->data);

current = current->next;

}

printf("\n");

}

// Function to free the memory allocated for the linked list

void freeList(struct Node\*\* head\_ref) {

struct Node\* current = \*head\_ref;

struct Node\* next;

while (current != NULL) {

next = current->next;

free(current);

current = next;

}

\*head\_ref = NULL;

}

int main() {

struct Node\* head = NULL;

// Inserting elements at the beginning

insertAtBeginning(&head, 3);

insertAtBeginning(&head, 2);

insertAtBeginning(&head, 1);

display(head);

// Inserting elements at the end

insertAtEnd(&head, 4);

insertAtEnd(&head, 5);

display(head);

// Deleting a node

deleteNode(&head, 3);

display(head);

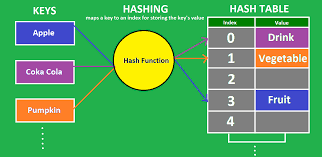
// Freeing the memory allocated for the linked list

freeList(&head);

return 0;

}

**4.HASH TABLE:**

* **Definition:** A data structure that implements an associative array, mapping unique keys to values. It uses a hashing function to efficiently locate the value associated with a particular key.
* **Operations:**
  + **Insert:** Add a key-value pair to the hash table (average constant time)
  + **Search:** Find the value associated with a given key (average constant time)
  + **Delete:** Remove a key-value pair from the hash table (average constant time)
* **C Code (basic implementation):**

C

#include <stdio.h>

#include <stdlib.h>

#define TABLE\_SIZE 10

struct HashTable {

int key;

char\* value;

};

struct HashTable hashTable[TABLE\_SIZE];

void initializeHashTable() {

for (int i = 0; i < TABLE\_SIZE; i++) {

hashTable[i].key = -1; // -1 indicates unused slot

hashTable[i].value = NULL;

}

}

int hashFunction(int key) {

return key % TABLE\_SIZE; // Simple modulo hashing

}

void insert(int key, char\* value) {

int index = hashFunction(key);

// Handle collisions (linear probing)

while (hashTable[index].key != -1) {

index++;

index %= TABLE\_SIZE; // Wrap around if necessary

}

hashTable[index].key = key;

hashTable[index].value = value;

}

char\* search(int key) {

int index = hashFunction(key);

while (hashTable[index].key != -1) {

if (hashTable[index].key == key) {

return hashTable[index].value;

}

index++;

index %= TABLE\_SIZE;

}

return NULL; // Key not found

}

int main() {

initializeHashTable();

insert(1, "apple");

insert(5, "banana");

insert(9, "cherry");

char\* value = search(5);

if (value != NULL) {

printf("Found value for key 5: %s\n", value);

} else {

printf("Key 5 not found\n");

}

return 0;

}

**5. Map**

The map data structure, also known as dictionaries or associative arrays, is a fundamental data structure that allows efficient storage and retrieval of key-value pairs. It is implemented using linked lists or trees, with the internal implementation depending on the programming language or library being used. Maps are typically implemented as associative arrays or hash tables, using a hash function to compute a unique index for each key-value pair.

**APPLICATIONS:**

1. **Stack**: Web browser history utilizes a stack data structure, enabling the back button functionality by pushing visited web pages onto the stack and popping them off when navigating backward.
2. **Queue**: A print queue manages print jobs in a first-in-first-out manner, ensuring fairness and orderliness in processing multiple print requests sent to a printer.
3. **Linked List**: Train cars in a railroad system are often organized as a linked list, facilitating efficient addition and removal of cars to and from the train while maintaining sequential order.
4. **Map**: A GPS navigation system employs a map data structure to efficiently store and retrieve information about geographical locations, enabling real-time route planning and navigation for users.
5. **Hash Table**: Symbol tables in compilers utilize hash tables to store identifiers and their associated attributes, providing fast access to variable names and their corresponding values during the compilation process.