**UNIT-V**

Brief introduction to hashing and hash functions:

Hashing is a fundamental concept in computer science used for various purposes such as data retrieval, cryptography, and data integrity verification. At its core, hashing involves taking an input (or 'message') and returning a fixed-size string of bytes, which represents a unique 'digest' of the input data. This process is performed by a mathematical function called a hash function.

Hash Functions:

A hash function takes an input (or 'message') and produces a fixed-size string of bytes, typically of a much smaller size than the input. The output is often referred to as the 'hash value', 'digest', or 'hash code'. Some key properties of a good hash function include:

1. \*\*Deterministic\*\*: For the same input, a hash function should always produce the same output.

2. \*\*Fast\*\*: Hash functions should be computationally efficient to calculate.

3. \*\*Uniform Distribution\*\*: Ideally, hash values should be uniformly distributed across the output space to minimize collisions (where two different inputs produce the same hash value).

4. \*\*Irreversibility\*\*: It should be computationally infeasible to reverse-engineer the original input from the hash value (one-way property).

5. \*\*Small Changes Yield Large Differences\*\*: Even a small change in the input should result in a significantly different hash value (avalanche effect).

Applications of Hash Functions:

1. \*\*Data Retrieval and Storage\*\*: Hash functions are used in hash tables and hash maps for fast data retrieval. They enable constant-time lookup, insertion, and deletion operations.

2. \*\*Cryptographic Applications\*\*: In cryptography, hash functions are used for data integrity verification, password hashing, digital signatures, and various other security mechanisms.

3. \*\*Unique Identifier Generation\*\*: Hash functions can generate unique identifiers for data records, files, or objects.

4. \*\*Caching\*\*: Hash functions play a crucial role in caching systems for efficiently mapping data to cache slots and distributing data across cache nodes in distributed systems.

Hashing and hash functions are foundational concepts in computer science and are widely used across various fields for their efficiency, security, and reliability.

Collision resolution techniques:

Collision resolution techniques are employed in hash tables to handle situations where two different keys hash to the same index, causing a collision. Here are some common collision resolution techniques:

1. \*\*Chaining\*\*:

* In chaining, each bucket in the hash table contains a linked list or another data structure (such as a dynamic array) that stores all the key-value pairs that hash to that bucket.
* When a collision occurs, the new key-value pair is simply appended to the list in the corresponding bucket.
* Chaining is relatively simple to implement and handles collisions well, but it may suffer from increased memory overhead due to the additional storage for the linked lists.

2. \*\*Open Addressing\*\*:

* + In open addressing, collisions are resolved by finding an alternative location within the hash table itself.
  + When a collision occurs, the algorithm systematically probes the table for an empty slot (using a sequence of probing functions).
  + Common probing techniques include linear probing (moving to the next slot), quadratic probing (using a quadratic function to probe), and double hashing (using a second hash function to calculate the next position).
  + Open addressing can have better cache performance and memory usage compared to chaining, but it requires careful implementation to avoid clustering and performance degradation.

3. \*\*Robin Hood Hashing\*\*:

* + Robin Hood hashing is a variation of linear probing that attempts to maintain a more balanced distribution of elements in the hash table.
  + When inserting a new key-value pair, if it encounters an existing element with a shorter probe distance (i.e., it is "richer"), it swaps positions with the richer element, allowing the new element to occupy a more favorable position in the probe sequence.
  + This technique aims to minimize the variance in probe lengths, resulting in more predictable performance.

4. \*\*Cuckoo Hashing\*\*:

* + Cuckoo hashing uses multiple hash functions and two or more hash tables to resolve collisions.
  + When a collision occurs, the new key-value pair is moved to an alternative location determined by another hash function in a different table.
  + This process continues recursively until no collisions remain or a maximum number of iterations is reached.
  + Cuckoo hashing guarantees constant lookup time and provides good average-case performance, but it requires careful tuning of parameters and may suffer from performance degradation in worst-case scenarios.

Certainly, let's delve deeper into both Chaining and Open Addressing collision resolution techniques:

# Open Addressing Collision Handling technique in Hashing

* A well-known search method is hashing.
* When the new key's hash value matches an already-occupied bucket in the hash table, there is a collision.

## Open Addressing for Collision Handling

Similar to separate chaining, open addressing is a technique for dealing with collisions. In Open Addressing, the hash table alone houses all of the elements. The size of the table must therefore always be more than or equal to the total number of keys at all times (Note that we can increase table size by copying old data if needed). This strategy is often referred to as closed hashing. The foundation of this entire process is probing. We will comprehend several forms of probing later.

* **Insert (k):** Continue probing until a slot is left open. Put k in the first empty spot you find.
* **Search (k):** Continue probing until either an empty slot is found or the slot's key no longer equals k.
* **Delete (k):** An intriguing delete procedure. The search can fail if we just remove a key. Therefore, deleted key slots are specifically noted as "deleted."

Although an item can be inserted into a deleted slot, the search continues after the slot has been empty.

**NOTE-**

* The "removed" buckets are handled the same as any other empty buckets during insertion.
* When searching, the search does not stop when it comes across a "deleted" bucket.
* Only when the necessary key or an empty bucket are discovered does the quest come to an end.

### Open Addressing

Open addressing is when

* All the keys are kept inside the hash table, unlike separate chaining.
* The hash table contains the only key information.

The methods for open addressing are as follows:

* Linear Probing
* Quadratic Probing
* Double Hashing

The following techniques are used for open addressing:

### (a) Linear probing

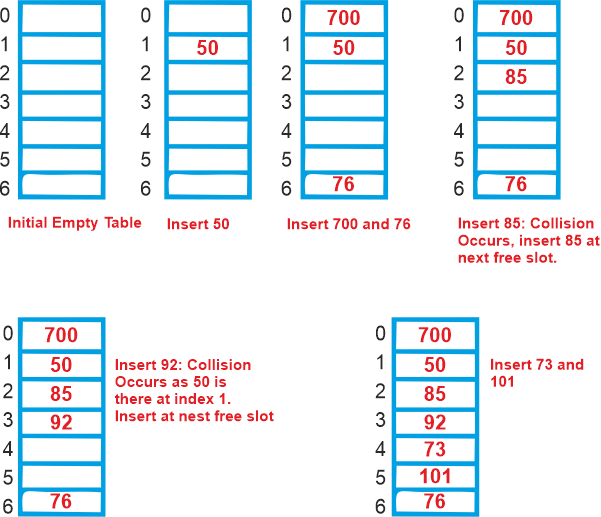
In linear probing, the hash table is systematically examined beginning at the hash's initial point. If the site we receive is already occupied, we look for a different one.

The rehashing function is as follows: table-size = (n+1)% rehash(key). As may be seen in the sample below, the usual space between two probes is 1.

Let S be the size of the table and let hash(x) be the slot index calculated using a hash algorithm.

1. If slot hash (x) % S is full, then we try ( hash (x) + 1 ) % S
2. If ( hash (x) + 1 ) % S is also full, then we try ( hash (x) + 2) % S
3. If ( hash (x) + 2 ) % S is also full, then we try ( hash (x) + 3 ) % S
4. ..................................................
5. ..................................................

Let's use "key mod 7" as a simple hash function with the following keys: 50, 700, 76, 85, 92, 73, 101.



**Linear probing problems:**

* **Primary Clustering:** Primary clustering is one of the issues with linear probing. Many successive items form clusters, making it difficult to locate a free slot or to search for an element.
* **Secondary Clustering:** Secondary clustering is less severe, and two records can only share a collision chain (also known as a probe sequence) if they start out in the same location.

**Advantage-**

* Calculating this is simple.

**Disadvantage-**

* Clustering is the fundamental issue with linear probing.
* Groups are composed of several adjacent pieces.
* After then, searching for an element or an empty bucket takes time.

**Time Complexity:**

The worst time in linear probing to search an element is O ( table size ). This is due to

* even if all other elements are absent and there is only one element.
* The hash table's "deleted" markers then force a full table search.

### (b) Quadratic probing

If you pay close attention, you will notice that the hash value will cause the interval between probes to grow. The above-discussed clustering issue can be resolved with the aid of the quadratic probing technique. The mid-square method is another name for this approach. We search for the i2'th slot in the i'th iteration using this strategy. We always begin where the hash was generated. We check the other slots if only the location is taken.

1. let hash (x) be the slot index computed using hash function.
2. If slot hash(x) % S is full, then we try  ( hash (x) + 1\*1 ) % S
3. If ( hash (x) + 1\*1 ) % S is also full, then we try ( hash (x) + 2\*2 ) % S
4. If ( hash (x) + 2\*2 ) % S is also full, then we try ( hash (x) + 3\*3 ) % S
5. ..................................................
6. ..................................................

### c) Double Hash

Another hash function calculates the gaps that exist between the probes. Clustering is optimally reduced by the use of double hashing. This method uses a different hash function to generate the increments for the probing sequence. We search for the slot i\*hash2(x) in the i'th rotation using another hash algorithm, hash2(x).

1. let hash(x) be the slot index computed using hash function.
2. If slot hash(x) % S is full, then we try (hash(x) + 1\*hash2(x)) % S
3. If (hash(x) + 1\*hash2(x)) % S is also full, then we try (hash(x) + 2\*hash2(x)) % S
4. If (hash(x) + 2\*hash2(x)) % S is also full, then we try (hash(x) + 3\*hash2(x)) % S
5. ..................................................
6. ..................................................

**Comparing the first three:**

* The best cache performance is provided by linear probing, although clustering is a problem. Linear probing also has the benefit of being simple to compute.
* Between the two in terms of clustering and cache performance is quadratic probing.
* Although double hashing lacks clustering, it performs poorly in caches. Due to the necessity to compute two hash functions, double hashing takes longer to compute.

|  |  |  |
| --- | --- | --- |
| **S. No.** | **Separate Chaining** | **Open Addressing** |
| **1.** | Chaining is easier to put into practise. | Open Addressing calls for increased processing power. |
| **2.** | Hash tables never run out of space when chaining since we can always add new elements. | Table may fill up when addressing in open fashion. |
| **3.** | Chaining is less susceptible to load or the hash function. | To prevent clustering and load factor, open addressing calls for extra caution. |
| **4.** | When it is unclear how many or how frequently keys might be added or removed, chaining is typically utilised. | When the frequency and quantity of keys are known, open addressing is employed. |
| **5.** | Chaining's cache performance is poor since keys are stored in linked lists. | Since everything is stored in the same table, open addressing improves cache speed. |
| **6.** | Space wastage (Some Parts of hash table in chaining are never used). | A slot can be used in open addressing even if an input doesn't map to it. |
| **7.** | Chaining requires additional room for links. | Links absent in open addressing |

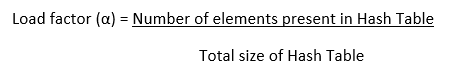
Because we traverse a Linked List by essentially jumping from one node to the next throughout the computer's memory, chaining's cache efficiency is poor. Because of this, the CPU is unable to cache nodes that haven't been visited yet, which is bad for us. However, since data isn't dispersed while using Open Addressing, the CPU can cache information for speedy access if it notices that a particular area of memory is frequently accessed.

Performance of Open Addressing: Similar to Chaining, the performance of hashing can be assessed assuming that each key has an equal likelihood of being hashed to any slot of the table (simple uniform hashing)

1. m = Number of slots in the hash table
2. n = Number of keys to be inserted in the hash table
4. Load factor α = n/m  ( **<** **1** )
6. Expected time to search/insert/delete **<** **1** / ( 1 - α )
8. So Search, Insert and Delete take (1 / ( 1 - α ) ) time

### Load Factor (α)-

Load factor (α) is defined as-



The load factor value in open addressing is always between 0 and 1. This is due to

* In open addressing, the hash table contains all of the keys.
* As a result, the table's size is always more than or at least equal to the number of keys it stores.

Certainly! Here's a basic implementation of a hash table along with the common operations:

### Basic Implementation:

class HashTable:

def \_\_init\_\_(self, size):

self.size = size

self.table = [[] for \_ in range(size)]

def \_hash\_function(self, key):

return hash(key) % self.size

def insert(self, key, value):

index = self.\_hash\_function(key)

self.table[index].append((key, value))

def search(self, key):

index = self.\_hash\_function(key)

for k, v in self.table[index]:

if k == key:

return v

return None

def delete(self, key):

index = self.\_hash\_function(key)

for i, (k, v) in enumerate(self.table[index]):

if k == key:

del self.table[index][i]

return True

return False

```

### Operations:

1. \*\*Insertion\*\*:

- Inserting a key-value pair into the hash table involves calculating the hash of the key to determine the index where the pair should be stored. The pair is then appended to the list at that index.

2. \*\*Search\*\*:

- Searching for a key in the hash table requires calculating the hash of the key to find the index. Then, iterating over the list at that index to find the key-value pair with the matching key.

3. \*\*Deletion\*\*:

- Deleting a key from the hash table involves calculating the hash of the key to find the index. Then, iterating over the list at that index to find the key-value pair with the matching key and removing it.

### Usage Example:

# Creating a hash table with size 10

hash\_table = HashTable(10)

# Inserting key-value pairs

hash\_table.insert('apple', 5)

hash\_table.insert('banana', 8)

hash\_table.insert('orange', 10)

# Searching for a key

print(hash\_table.search('banana')) # Output: 8

# Deleting a key

hash\_table.delete('banana')

# Searching for the deleted key

print(hash\_table.search('banana')) # Output: None

```

This implementation provides a basic understanding of how hash tables work and how to perform common operations such as insertion, search, and deletion. However, in real-world scenarios, more advanced implementations may be required to handle collisions, resizing, and other optimizations.

Hashing has numerous applications in computing, including unique identifier generation and caching. Here's how it is utilized in each:

Unique Identifier Generation:

1. \*\*Hash-based Unique Identifiers\*\*: Hash functions are often used to generate unique identifiers for various purposes. These identifiers are derived from the data they represent through a hash function. For example, in databases, hash-based identifiers might be generated for rows in a table or for objects in a distributed system. These identifiers are usually unique for different inputs due to the nature of hash functions.

2. \*\*Password Hashing\*\*: In security applications, passwords are often hashed before being stored in a database. This ensures that even if the database is compromised, the original passwords cannot be easily retrieved. When a user logs in, their password is hashed again and compared to the stored hash.

### Caching:1. \*\*Content Addressable Memory (CAM)\*\*: In caching systems, hash functions are used to map data to cache slots. This allows for quick retrieval of cached data based on a hash of the data's ontent. Content Addressable Memory (CAM) is a specialized hardware or software memory structure where data can be accessed directly based on its content, typically through hashing.

2. \*\*Hash Tables\*\*: Hash tables are a fundamental data structure used in caching systems. They allow for efficient storage and retrieval of key-value pairs. Hash functions are used to map keys to indices in the hash table, providing fast access to cached data. In this context, hashing enables constant-time lookup, insertion, and deletion operations.

3. \*\*Distributed Caching\*\*: In distributed systems, hash functions are used to determine which node should store a particular piece of data. By hashing the data's key, a consistent hash function can be used to map the data to a node in the distributed cache. This ensures that each piece of data is evenly distributed across the cache nodes, reducing hotspots and improving overall performance.

In both unique identifier generation and caching, hashing provides a way to efficiently map data to identifiers or cache locations, enabling fast access and storage of information.

Caching, in computing, involves storing frequently accessed data in a high-speed memory or storage location to reduce access time and improve performance. Hash tables are commonly used in caching systems due to their efficient lookup and retrieval operations. Here are some caching applications where hash tables play a significant role:

1. \*\*Web Caching\*\*:

* + Web servers often use caching to store frequently accessed web pages, images, or other resources. A hash table can be employed to map URLs to their corresponding cached content.
  + When a user requests a web page, the server checks the hash table to see if the page is already cached. If found, the server serves the cached content, reducing latency and server load.

2. \*\*Database Query Caching\*\*:

* + In database systems, query results can be cached to avoid expensive database queries for commonly accessed data.
  + Hash tables can be utilized to store the query parameters as keys and the corresponding results as values. This allows for efficient retrieval of cached results based on the query parameters.

3. \*\*File System Caching\*\*:

* + Operating systems often cache frequently accessed files or file system metadata to improve disk access performance.
  + Hash tables can be employed to map file paths or file identifiers to their corresponding cached data blocks or metadata entries.

4. \*\*API Response Caching\*\*:

* + APIs frequently cache responses to reduce latency and improve throughput.
  + Hash tables can be used to cache API responses based on request parameters or request URIs. This allows for quick retrieval of cached responses for identical or similar requests.

5. \*\*Content Delivery Networks (CDNs)\*\*:

* + CDNs cache content such as images, videos, and static files at multiple edge locations to reduce latency and improve content delivery speed.
  + Hash tables are employed to map content identifiers (e.g., URLs or file paths) to cached content stored at different edge locations. This enables efficient content retrieval based on client requests.

6. \*\*DNS Caching\*\*:

* + DNS servers cache DNS query responses to reduce the time required to resolve domain names to IP addresses.
  + Hash tables can be used to store domain names as keys and their corresponding IP addresses as values. This allows for quick resolution of frequently accessed domain names without querying authoritative DNS servers.

In all these caching applications, hash tables provide efficient data storage and retrieval, enabling quick access to cached content based on keys or identifiers. They play a crucial role in improving system performance and reducing latency in various computing environments.

Operations of Deque:

A deque, short for "double-ended queue," is a data structure that supports insertion and deletion at both the front and the back ends. Here are the common operations supported by deques:

A deque, short for "double-ended queue," is a data structure that supports insertion and deletion at both the front and the back ends. Here are the common operations supported by deques:

1. **Insertion Operations:**

a. **Append/Insert at the Back:** Add an element to the back end of the deque.

b. **Append/Insert at the Front:** Add an element to the front end of the deque.

1. **Deletion Operations:**

a. **Pop/Delete from the Back:** Remove and return the element at the back end of the deque.

b. **Pop/Delete from the Front:** Remove and return the element at the front end of the deque.

1. **Access Operations:**

a. **Peek at the Back:** Return the element at the back end of the deque without removing it.

b. **Peek at the Front:** Return the element at the front end of the deque without removing it.

1. **Size and Empty Check Operations:**

a. **Size:** Return the number of elements currently stored in the deque.

b. **Empty Check:** Check if the deque is empty.

1. **Clear Operation:**

a. **Clear:** Remove all elements from the deque, leaving it empty.

These operations provide flexibility in managing data from both ends of the deque, making it suitable for various applications such as implementing queues, stacks, and certain algorithms like breadth-first search.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

// Structure to represent a deque

typedef struct {

int data[MAX\_SIZE];

int front, rear;

} Deque;

// Function to initialize a deque

void initDeque(Deque \*deque) {

deque->front = -1;

deque->rear = -1;

}

// Function to check if the deque is empty

int isEmpty(Deque \*deque) {

return (deque->front == -1 && deque->rear == -1);

}

// Function to check if the deque is full

int isFull(Deque \*deque) {

return (deque->rear + 1) % MAX\_SIZE == deque->front;

}

// Function to add an element to the front of the deque

void addFront(Deque \*deque, int value) {

if (isFull(deque)) {

printf("Deque is full.\n");

return;

}

if (isEmpty(deque)) {

deque->front = 0;

deque->rear = 0;

} else {

deque->front = (deque->front - 1 + MAX\_SIZE) % MAX\_SIZE;

}

deque->data[deque->front] = value;

}

// Function to add an element to the rear of the deque

void addRear(Deque \*deque, int value) {

if (isFull(deque)) {

printf("Deque is full.\n");

return;

}

if (isEmpty(deque)) {

deque->front = 0;

deque->rear = 0;

} else {

deque->rear = (deque->rear + 1) % MAX\_SIZE;

}

deque->data[deque->rear] = value;

}

// Function to remove an element from the front of the deque

int removeFront(Deque \*deque) {

if (isEmpty(deque)) {

printf("Deque is empty.\n");

return -1;

}

int value = deque->data[deque->front];

if (deque->front == deque->rear) {

deque->front = -1;

deque->rear = -1;

} else {

deque->front = (deque->front + 1) % MAX\_SIZE;

}

return value;

}

// Function to remove an element from the rear of the deque

int removeRear(Deque \*deque) {

if (isEmpty(deque)) {

printf("Deque is empty.\n");

return -1;

}

int value = deque->data[deque->rear];

if (deque->front == deque->rear) {

deque->front = -1;

deque->rear = -1;

} else {

deque->rear = (deque->rear - 1 + MAX\_SIZE) % MAX\_SIZE;

}

return value;

}

// Function to get the element at the front of the deque

int getFront(Deque \*deque) {

if (isEmpty(deque)) {

printf("Deque is empty.\n");

return -1;

}

return deque->data[deque->front];

}

// Function to get the element at the rear of the deque

int getRear(Deque \*deque) {

if (isEmpty(deque)) {

printf("Deque is empty.\n");

return -1;

}

return deque->data[deque->rear];

}

int main() {

Deque deque;

initDeque(&deque);

addRear(&deque, 10);

addRear(&deque, 20);

addFront(&deque, 5);

printf("Front element: %d\n", getFront(&deque));

printf("Rear element: %d\n", getRear(&deque));

removeFront(&deque);

printf("Front element after removal: %d\n", getFront(&deque));

return 0;

}

A double-ended queue (deque) is a versatile data structure that allows elements to be added or removed from both ends efficiently. Here are several common applications of deques:

1. \*\*Palindrome Checker\*\*:

- Deques can be used to check if a string is a palindrome by comparing elements from both ends towards the center.

2. \*\*Sliding Window Problems\*\*:

- Deques are useful for maintaining a sliding window of elements, especially for calculating maximum or minimum values within the window in an efficient manner.

3. \*\*Job Scheduling\*\*:

- In operating systems or job scheduling systems, deques can be used to manage tasks where tasks can be added or removed from either end of the queue.

4. \*\*Undo/Redo Functionality\*\*:

- Deques can be used to implement undo and redo features in applications like text editors, where you can add new states to the front and remove states from the back.

5. \*\*Breadth-First Search (BFS)\*\*:

- In graph algorithms like BFS, deques can be used to keep track of the nodes to be explored, allowing efficient addition of nodes to the front and removal from the back.

6. \*\*Expression Evaluation\*\*:

- Deques can be used in the evaluation of expressions, especially in postfix or prefix notation, where elements need to be processed from both ends.

7. \*\*Load Balancing\*\*:

- Deques can manage tasks across multiple processors or servers, allowing tasks to be dynamically added or removed from either end based on load.

8. \*\*Real-Time Data Processing\*\*:

- In real-time systems, deques can handle streams of data where old data is removed from the front and new data is added to the back, maintaining a constant-size window of recent data.

9. \*\*Cache Implementation\*\*:

- Deques can be used in the implementation of caches, such as Least Recently Used (LRU) cache, where elements are moved to the front on access and removed from the back when the cache is full.

10. \*\*Simulation\*\*:

- In simulations, deques can manage events that need to be processed in a specific order, allowing events to be added or removed from either end.

11. \*\*Priority Queues\*\*:

- While not a traditional use, deques can be adapted to implement double-ended priority queues where elements can be inserted and removed based on priority from both ends.

These applications leverage the efficient insertion and deletion operations at both ends of the deque, making it a powerful and flexible data structure for a wide range of scenarios.

Back tracking using stack:

Here's how to implement backtracking using a stack in C. We'll use the N-Queens problem as an example.

N-Queens Problem with Stack in C

In this problem, we need to place N queens on an N×N chessboard such that no two queens threaten each other.

Step-by-Step Implementation

1. \*\*Define the State\*\*:

- A state can be represented by a structure holding the positions of the queens placed so far and the current row.

2. \*\*Stack Implementation\*\*:

- Use a stack to manage the states during the exploration of solutions.

3. \*\*Backtracking with the Stack\*\*:

- Push the initial state (an empty board) onto the stack.

- Pop states from the stack, check for solutions, generate valid next states, and push them onto the stack.

Here is the C code

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAXN 20

// Define a structure for the state

typedef struct {

int queens[MAXN]; // Positions of queens, index represents row, value represents column

int row; // Current row to place a queen

} State;

// Stack structure

typedef struct {

State states[MAXN \* MAXN];

int top;

} Stack;

void initStack(Stack \*s) {

s->top = -1;

}

bool isEmpty(Stack \*s) {

return s->top == -1;

}

void push(Stack \*s, State state) {

s->states[++(s->top)] = state;

}

State pop(Stack \*s) {

return s->states[(s->top)--];

}

bool isValid(State \*state, int row, int col) {

for (int i = 0; i < row; i++) {

int queenCol = state->queens[i];

if (queenCol == col || queenCol - i == col - row || queenCol + i == col + row) {

return false;

} }

return true;

}

void printSolution(State \*state, int n) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

if (state->queens[i] == j) {

printf("Q ");

} else {

printf(". ");

}

}

printf("\n");

}

printf("\n");

}

void solveNQueens(int n) {

Stack stack;

initStack(&stack);

State initial;

initial.row = 0;

push(&stack, initial);

while (!isEmpty(&stack)) {

State current = pop(&stack);

if (current.row == n) {

printSolution(&current, n);

continue;

}

for (int col = n - 1; col >= 0; col--) {

if (isValid(&current, current.row, col)) {

State next = current;

next.queens[next.row] = col;

next.row++;

push(&stack, next);

}

}

}

}

int main() {

int n = 4;

printf("Solutions for %d-Queens:\n", n);

solveNQueens(n);

return 0;

}

Applications of stacks in expression evaluation:

Stacks are widely used in expression evaluation due to their LIFO (Last In, First Out) property, which aligns well with the order of operations and nested structures in mathematical expressions. Here are several key applications of stacks in expression evaluation:

### 1. \*\*Infix to Postfix Conversion (Shunting Yard Algorithm)\*\*

The Shunting Yard algorithm, developed by Edsger Dijkstra, uses a stack to convert infix expressions (e.g., `A + B \* C`) to postfix expressions (e.g., `A B C \* +`). This is useful because postfix expressions (Reverse Polish Notation) are easier to evaluate programmatically.

Example Algorithm:

1. Read the tokens (operators and operands) from the infix expression one at a time.

2. Use a stack to hold operators and parentheses.

3. When an operand is read, add it to the output.

4. When an operator is read, pop operators from the stack to the output until an operator with lower precedence is encountered or a left parenthesis is on top.

5. Push the read operator onto the stack.

6. When a left parenthesis is read, push it onto the stack.

7. When a right parenthesis is read, pop from the stack to the output until a left parenthesis is encountered.

8. After reading all tokens, pop any remaining operators from the stack to the output.

2. \*\*Postfix Expression Evaluation\*\*

Once an infix expression is converted to postfix, it can be evaluated efficiently using a stack. The process involves reading the postfix expression from left to right and using the stack to store intermediate results.

Example Algorithm:

1. Read the tokens from the postfix expression one at a time.

2. If the token is an operand, push it onto the stack.

3. If the token is an operator, pop the required number of operands from the stack, apply the operator, and push the result back onto the stack.

4. After reading all tokens, the result is the top value on the stack.

3. \*\*Prefix Expression Evaluation\*\*

Similarly, prefix expressions (Polish Notation) can be evaluated using a stack. The process is similar to postfix evaluation but involves reading the expression from right to left.

Example Algorithm:

1. Read the tokens from the prefix expression from right to left.

2. If the token is an operand, push it onto the stack.

3. If the token is an operator, pop the required number of operands from the stack, apply the operator, and push the result back onto the stack.

4. After reading all tokens, the result is the top value on the stack.

4. \*\*Balancing Parentheses\*\*

Stacks are used to check for balanced parentheses in expressions to ensure they are syntactically correct.

Example Algorithm:

1. Read the expression character by character.

2. Push opening parentheses onto the stack.

3. For closing parentheses, check if the stack is not empty and the top of the stack is the corresponding opening parenthesis. If so, pop the stack.

4. If the stack is empty after processing the entire expression, the parentheses are balanced.

Example Code in C for Postfix Evaluation

Here is an example of how to evaluate a postfix expression using a stack in C:

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define MAXSTACK 100

typedef struct {

int top;

int items[MAXSTACK];

} Stack;

void initStack(Stack \*s) {

s->top = -1;

}

int isEmpty(Stack \*s) {

return s->top == -1;

}

int isFull(Stack \*s) {

return s->top == MAXSTACK - 1;

}

void push(Stack \*s, int value) {

if (!isFull(s)) {

s->items[++(s->top)] = value;

} else {

printf("Stack overflow\n");

exit(1);

}

}

int pop(Stack \*s) {

if (!isEmpty(s)) {

return s->items[(s->top)--];

} else {

printf("Stack underflow\n");

exit(1);

}

}

int evaluatePostfix(char \*expression) {

Stack stack;

initStack(&stack);

char \*token = strtok(expression, " ");

while (token != NULL) {

if (isdigit(token[0])) {

push(&stack, atoi(token));

} else {

int operand2 = pop(&stack);

int operand1 = pop(&stack);

switch (token[0]) {

case '+': push(&stack, operand1 + operand2); break;

case '-': push(&stack, operand1 - operand2); break;

case '\*': push(&stack, operand1 \* operand2); break;

case '/': push(&stack, operand1 / operand2); break;

}

}

token = strtok(NULL, " ");

}

return pop(&stack);

}

int main() {

char expression[] = "3 4 + 2 \* 7 /";

printf("Postfix evaluation: %d\n", evaluatePostfix(expression));

return 0;

}```

Explanation of the Code:

1. \*\*Stack Initialization\*\*:

- `initStack` initializes the stack.

- `isEmpty` and `isFull` check the stack status.

- `push` and `pop` manage the stack operations.

2. \*\*Postfix Evaluation\*\*:

- The `evaluatePostfix` function reads the postfix expression, tokenizes it, and processes each token.

- Operands are pushed onto the stack, and operators pop operands from the stack, apply the operation, and push the result back onto the stack.

, Applications of linked lists:

Linked lists are a fundamental data structure with a variety of applications due to their dynamic nature and efficient insertion and deletion operations. Here are several key applications of linked lists:

### 1. ****Dynamic Memory Allocation****

* **Memory Management**: Linked lists can be used to manage free memory blocks. Each node represents a memory block, and the list helps keep track of which blocks are free and which are allocated.

### 2. ****Implementing Other Data Structures****

* **Stacks and Queues**: Linked lists can be used to implement stacks (LIFO) and queues (FIFO). For a stack, insertion and deletion are done at the head of the list. For a queue, insertion is done at the tail and deletion at the head.
* **Hash Tables**: Linked lists are used in chaining, a technique to handle collisions in hash tables. Each bucket of the hash table points to a linked list of entries that hash to the same index.

### 3. ****Graph Representations****

* **Adjacency Lists**: Linked lists are used to represent graphs through adjacency lists. Each node in the list represents a vertex, and its linked list contains the vertices it is connected to.

### 4. ****Real-Time Applications****

* **Task Scheduling**: Linked lists can be used to manage tasks in real-time systems where tasks need to be inserted or deleted dynamically.
* **Undo Mechanisms**: Applications like text editors use linked lists to implement undo functionality. Each node represents a state of the document, allowing easy traversal backward to previous states.

### 5. ****Database Management Systems****

* **Sparse Matrices**: Linked lists can efficiently represent sparse matrices, where most elements are zero. Only non-zero elements are stored, with linked nodes indicating their row and column positions.

### 6. ****File Systems****

* **Directory Management**: Linked lists are used to manage directories in file systems. Each directory node contains pointers to files and subdirectories, allowing dynamic addition and deletion.
* **Free Space Management**: File systems use linked lists to keep track of free disk space. Each node represents a block of free space.

### 7. ****Text Editors****

* **Buffer Management**: Text editors use linked lists to manage buffers of text. Each line or paragraph can be a node in the list, making insertion and deletion operations efficient.

### 8. ****Operating Systems****

* **Process Scheduling**: Linked lists are used to manage processes in operating systems. Each node represents a process control block (PCB), allowing dynamic addition and removal of processes.
* **Multi-Level Paging**: Linked lists are used in multi-level paging systems to manage page tables dynamically.

### 9. ****Polynomial Arithmetic****

* **Sparse Polynomials**: Linked lists can represent polynomials where each node contains a term (coefficient and exponent). This representation makes it easy to perform polynomial addition, subtraction, and multiplication.

### 10. ****Networking****

* **Packet Buffers**: In network routers and switches, linked lists are used to manage packets in buffers, allowing dynamic insertion and removal as packets are processed and forwarded.