**BINARY SEARCH:**

Binary search is a search algorithm used to find the position of a target value within a sorted array or list. It works by repeatedly dividing in half the portion of the list that could contain the target value, until it narrows down the possible locations to just one.

**ALGORITHM PROCEDURE FOR BINARY SEARCH:**

1. **Initialize**: Set two pointers, low and high, to the first and last index of the sorted array, respectively.
2. **Loop**: While the low pointer is less than or equal to the high pointer:
   * Calculate the middle index: mid = (low + high) // 2.
   * Check if the middle element of the array is equal to the target value:
     + If it is, return the index mid.
     + If it's not, compare the target value with the middle element:
       - If the target is greater than the middle element, update low = mid + 1.
       - If the target is less than the middle element, update high = mid - 1.
3. **Exit Loop**: If the loop exits without finding the target value, return -1 to indicate that the target is not present in the array.

This algorithm relies on the fact that the array is sorted. By repeatedly dividing the search space in half, it efficiently reduces the number of elements that need to be searched. This results in a time complexity of O(log n), where n is the number of elements in the array.

Top of Form



**Python program to implement Binary search:**

import matplotlib

import matplotlib.pyplot as plt

matplotlib.use('TKAgg')

import math

import time

def binarysearch(a,key):

low=0

high=len(a)-1

while low<=high:

mid=(high+low)//2

if a[mid]==key:

return mid

elif key<a[mid]:

high=mid-1

else:

low=mid+1

return-1

start=time.time()

a=[]

s=int(input("enter the array size:"))

while True:

if len(a)<s:

arr=int(input("enter the array elements:"))

a.apppend(arr)

else:

break

print("the array elements are:",a)

k=int(input("enter the key elements to search:"))

r=binarysearch(a,k)

if r<-1:

print("search unsuccessful")

else:

print("search successfull,found at location:",r+1)

x=list(range(1,1000))

plt.plot(x,[math.log(y,2)for y in x])

plt.title("binary search time complexity is 0(logn)")

plt.xlabel("input")

plt.ylabel("time")

plt.show()

**BINARYSEARCH ALGORITHM:**

* Binarysearch works only on sorted array.
* This technique searches the given elements by repatedly dividing the array into half
* The idea of binary search is to reduce the time complexity
* The algorithm of binarysearch is given below

***ALGORITHM:***

Algorithm Binary\_search(A[0,1…..n-1],key)

{

Low🡨0

High🡨n-1

While(low🡨high)do

{

Mid🡨(high+low)//2

If(key==a[mid])then

Return mid

Else if (key<a[mid])then

High🡨mid-1

Else

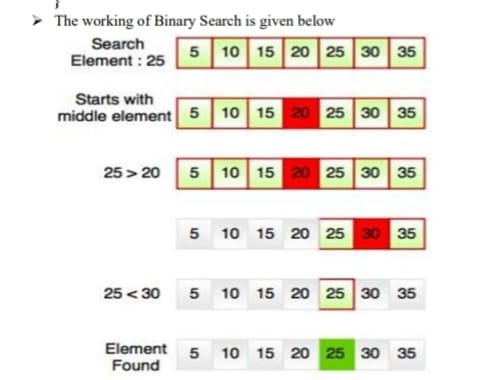
Low🡨mid+1

}

Return -1

}

**HOW DOES BINARYSEARCH WORK IN MEMORY:**

****

**APPLICATION OF BINARYSEARCH:**

1. **Searching in Sorted Arrays**: Binary search is commonly used to search for a target element in a sorted array. This application is efficient and has a time complexity of O(log n), where n is the number of elements in the array.
2. **Finding an Element in a Database**: In databases, binary search can be used to quickly locate records based on a sorted key. For example, if a database is indexed by a unique ID, binary search can efficiently locate the record corresponding to a given ID.
3. **Dictionary Lookup**: In Python, dictionaries are implemented using hash tables, which do not support binary search directly. However, if keys are stored in sorted order, binary search can be used to efficiently find a key or range of keys.
4. **Identifying Peaks in Data**: Binary search can be used to identify peaks in data sets, such as finding the maximum or minimum value in a unimodal function. By comparing neighboring elements using binary search, peaks can be located efficiently.
5. **Searching in Trees and Graphs**: Binary search trees (BSTs) are a type of binary tree where the left child of a node contains only values less than the node's value, and the right child contains only values greater than the node's value. Binary search can be used to efficiently search, insert, and delete elements in BSTs.
6. **Text Search in Sorted Dictionaries**: If a dictionary is sorted by keys, binary search can be used to find words or phrases efficiently. This is particularly useful in applications such as spell checkers or autocomplete features.
7. **Range Queries**: Binary search can be used to perform range queries, such as finding all elements within a given range in a sorted array or dictionary. By performing two binary searches to find the starting and ending indices of the range, all elements within the range can be efficiently located.

**STACK:**

In Python, a stack is a data structure that follows the Last In, First Out (LIFO) principle. It is similar to a stack of plates where the last plate placed on top is the first one to be removed. In a stack, elements are added or removed from only one end, typically referred to as the "top" of the stack.

**EXPLANATION OF METHODS**

* \_\_init\_\_(self): Initializes an empty stack using a list to store elements.
* is\_empty(self): Returns True if the stack is empty, False otherwise.
* push(self, item): Adds an item to the top of the stack.
* pop(self): Removes and returns the item at the top of the stack. Raises an IndexError if the stack is empty.
* peek(self): Returns the item at the top of the stack without removing it. Raises an IndexError if the stack is empty.
* size(self): Returns the number of elements in the stack.

This demonstrates the basic functionality of a stack in Python. It's important to note that Python's built-in list can also be used as a stack, as it supports the append() and pop() operations. However, using a custom stack class provides encapsulation and additional methods for stack manipulation.

**PYTHON PROGRAM TO IMPLEMENT STACK DATA**

**STRUCTURE:**

s=[ ]

def push():

if len(s)==size:

print(“stack is full”)

else:

item=input(“enter the elements”)

s.append(item)

def pop()

if len(s)==0:

print(“stack is empty”)

else:

item=s[-1]

del(s[-1])

print(“the term deleted is:”,item)

def display():

if len(s)==0:

print(“stack is empty”)

else:

print(“the data elements are:”,s)

size=int(input(“enter the stack size:”))

while True:

c=int(input(“insert-1 delete-1 display-3 exit-4 make your choice:”))

if c==1:

push()

elif c==2:

pop()

elif c==3:

display()

else:

break

**ALGORITHM AND PROCEDURE:**

1. **Initialize an Empty Stack**:
   * Create an empty list to represent the stack.
   * Set the stack size to 0.
2. **Push Operation**:
   * To push an element onto the stack:
     + Append the element to the end of the list.
     + Increment the stack size.
3. **Pop Operation**:
   * To pop an element from the stack:
     + If the stack is empty, return an error or raise an exception.
     + Otherwise, remove the last element from the list.
     + Decrement the stack size.
     + Return the removed element.
4. **Peek Operation**:
   * To peek at the top element of the stack:
     + If the stack is empty, return an error or raise an exception.
     + Otherwise, return the last element from the list without removing it.
5. **Check If Stack is Empty**:
   * To check if the stack is empty:
     + Return True if the stack size is 0, indicating an empty stack.
     + Return False otherwise.
6. **Get Stack Size**:
   * To get the current size of the stack:
     + Return the stack size.

This algorithm describes the basic operations and their implementations for a stack data structure. It can be implemented in various programming languages, such as Python, C++, Java, etc., using arrays, linked lists, or other underlying data structures.

**LOGIC FOR STACK:**

class Stack:

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

self.items = []

def is\_empty(self):

return self.items == []

def push(self, item):

self.items.append(item)

def pop(self):

if not self.is\_empty():

return self.items.pop()

else:

print("Stack is empty")

return None

def peek(self):

if not self.is\_empty():

return self.items[-1]

else:

print("Stack is empty")

return None

def size(self):

return len(self.items)

# Example usage:

stack = Stack()

stack.push(1)

stack.push(2)

stack.push(3)

print("Stack:", stack.items) # Output: [1, 2, 3]

print("Pop item:", stack.pop()) # Output: 3

print("Stack after pop:", stack.items) # Output: [1, 2]

print("Peek item:", stack.peek()) # Output: 2

print("Size of stack:", stack.size())

**HOW DOES STACK WORK IN MEMORY:**

1. **Memory Allocation**: When a program starts, the operating system allocates a region of memory for the stack. This region is usually located at the top of the program's memory space and grows downward.
2. **Stack Pointer (SP)**: The stack is managed using a stack pointer, which is a special register or memory location that keeps track of the top of the stack. Initially, the stack pointer points to the bottom of the stack.
3. **Push Operation**: When an item is pushed onto the stack, it is placed at the memory address pointed to by the stack pointer, and the stack pointer is decremented to point to the new top of the stack.
4. **Pop Operation**: When an item is popped from the stack, the item at the memory address pointed to by the stack pointer is removed, and the stack pointer is incremented to point to the next item in the stack.
5. **Overflow and Underflow**: If the stack grows beyond the allocated memory space, it results in a stack overflow. Similarly, if an attempt is made to pop from an empty stack, it results in a stack underflow.
6. **Local Variables and Function Calls**: In many programming languages, local variables and function call information (such as return addresses and parameters) are stored on the stack. When a function is called, its local variables and other information are pushed onto the stack, and when the function returns, this information is popped off the stack.
7. **Stack Frames**: Each function call typically creates a new stack frame (also known as activation record or call frame) on the stack to store its local variables and other information. When the function returns, its stack frame is removed from the stack.

**APPLICATION OF STACK:**

1. **Function Call Stack**: In Python (and in many other programming languages), the call stack is implemented using a stack data structure. When a function is called, its local variables and other information are pushed onto the call stack. When the function returns, its stack frame is popped off the stack. This mechanism allows for nested function calls and ensures that function calls are resolved in the reverse order of their invocation.
2. **Expression Evaluation**: Stacks can be used to evaluate arithmetic expressions, such as infix, postfix, or prefix expressions. For example, converting an infix expression to postfix notation involves using a stack to keep track of operators and operands according to their precedence and associativity.
3. **Syntax Parsing**: Stacks are often used in syntax parsing and parsing algorithms such as recursive descent parsers and LR parsers. They can help in tracking the structure of expressions, statements, or code blocks during parsing.
4. **Undo Mechanisms**: Stacks can be used to implement undo mechanisms in applications where users need to revert to previous states or actions. Each action can be pushed onto the stack, and the undo operation pops the most recent action and reverts the state accordingly.
5. **Backtracking Algorithms**: Stacks are essential in backtracking algorithms such as depth-first search (DFS) and backtracking search. These algorithms use stacks to maintain the current path or state while exploring the search space and backtrack when necessary.
6. **Memory Management**: Stacks can be used for memory management in certain scenarios, such as managing temporary data or allocating memory for recursive function calls.
7. **Algorithm Implementations**: Stacks are used in various algorithm implementations, including depth-first search, tree traversal algorithms (e.g., inorder, preorder, and postorder traversal), finding connected components in a graph, and more.

**BINARY SEARCH TREE:**

A Binary Search Tree (BST) is a binary tree data structure in which each node has at most two children (referred to as the left child and the right child), and the key (value) of each node follows a specific ordering property:

**ALGORITHM:**

1. The key in the left child node is less than the key in its parent node.
2. The key in the right child node is greater than the key in its parent node.
3. TreeNode represents a single node in the binary search tree, containing a key (value) and references to its left and right children.
4. BinarySearchTree is the binary search tree class containing methods for insertion, searching, and inorder traversal.
5. insert method inserts a new key into the binary search tree.
6. \_insert\_recursive is a helper method for recursive insertion.
7. search method searches for a key in the binary search tree.
8. \_search\_recursive is a helper method for recursive search.
9. inorder\_traversal method performs an inorder traversal of the binary search tree, returning the keys in sorted order.
10. \_inorder\_traversal\_recursive is a helper method for recursive inorder traversal.

This is a basic implementation of a binary search tree in Python. It provides functionality for insertion, searching, andinorder traversal.

**PYTHON PROGRAM TO IMPLEMENT BINARY**

**SEARCH TREE**

class Node:

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

self.data=value

self.left=None

self.right=None

class BinarysearchTree:

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

self.root=None

def insert(self,value):

newNode=Node(value)

if self.root is None:

self.root=newNode

else:

curNode=self.root

while curNode is not None:

if value<curNode.data:

curNode.left=newNode

break

else:

curNode=curNode.left

else:

if curNode.right is None:

curNode.right=newNode

break

else:

curNode=curNode.right

def preorder(self,rt):

print(rt.data,end=””)

if rt.left is not None:

self.preorder(rt.left)

if rt.right is not None:

self.preorder(rt.right)

def postorder(self,rt):

if rt.left is not None:

self.postorder(rt.left)

if rt.right is not None:

self.postorder(rt.right)

print(rt.data,end=””)

def inorder(self,rt):

if rt.left is not None:

self.inorder(rt.left)

print(rt.data,end=””)

if rt.right is not None:

self.inorder(rt.right)

bst=BinarySearchTree()

ls=[ ]

s=int(input(“enter the size”))

while True:

if len(ls)<s:

a=int(input(“enter the elements:”))

ls.appens(a)

else:

break

for I in ls:

bst.insert(i)

print(“\n preorder traversal is:”bst.preorder(bst.root))

print(“\n postorder traversal is:”,bst.postorder(bst.root))

print(“\ inorder traversal is:’,bst.inorder(bst.root))

**LOGIC FOR BINARY SEARCH TREE:**

class TreeNode:

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

self.key = key

self.left = None

self.right = None

class BinaryTree:

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

self.root = None

def insert(self, key):

if self.root is None:

self.root = TreeNode(key)

else:

self.\_insert\_recursive(self.root, key)

def \_insert\_recursive(self, node, key):

if key < node.key:

if node.left is None:

node.left = TreeNode(key)

else:

self.\_insert\_recursive(node.left, key)

elif key > node.key:

if node.right is None:

node.right = TreeNode(key)

else:

self.\_insert\_recursive(node.right, key)

def search(self, key):

return self.\_search\_recursive(self.root, key)

def \_search\_recursive(self, node, key):

if node is None or node.key == key:

return node

if key < node.key:

return self.\_search\_recursive(node.left, key)

return self.\_search\_recursive(node.right, key)

def inorder\_traversal(self):

result = []

self.\_inorder\_traversal\_recursive(self.root, result)

return result

def \_inorder\_traversal\_recursive(self, node, result):

if node:

self.\_inorder\_traversal\_recursive(node.left, result)

result.append(node.key)

self.\_inorder\_traversal\_recursive(node.right, result)

**BINARY SEARCH TREE WORK IN MEMORY:**

1. **Node Allocation**: Each node in the BST is dynamically allocated from the heap memory. The node structure usually contains fields for storing the key (value) and references (pointers) to its left and right children.
2. **Root Node**: The root node of the BST is a special node that serves as the starting point for all operations on the tree. It is stored in a variable or a pointer accessible to the program.
3. **Child Pointers**: Each node in the BST contains pointers to its left and right children (if they exist). These pointers are typically implemented as memory addresses pointing to the respective child nodes.
4. **Ordering Property**: The BST maintains the ordering property, ensuring that the keys in the left subtree of a node are less than the key of the node, and the keys in the right subtree are greater than the key of the node.
5. **Insertion**: When inserting a new key into the BST, the tree is traversed starting from the root node. At each step, the key is compared with the key of the current node, and based on the comparison, traversal continues to the left or right child until an appropriate position for insertion is found.
6. **Deletion**: Deleting a node from the BST involves finding the node to delete, rearranging the pointers to maintain the BST property, and deallocating the memory for the deleted node.
7. **Searching**: Searching for a key in the BST involves traversing the tree starting from the root node and following the appropriate child pointers based on the comparison of keys until the key is found or a leaf node (where the key is not present) is reached.
8. **Memory Management**: The memory for each node in the BST is managed dynamically, allowing nodes to be allocated and deallocated as needed during insertion, deletion, or other operations.

Overall, the BST data structure provides efficient searching, insertion, and deletion operations due to its hierarchical organization and ordering property. Implementing a BST in memory involves managing dynamically allocated memory for nodes and maintaining the ordering property during tree manipulation operations.

**APPLICATION OF BINARY SEARCH TREE:**

1. **Database Indexing**: BSTs are commonly used in database indexing to store and retrieve data efficiently. They allow for quick searching and retrieval of records based on key values.
2. **Symbol Tables**: BSTs are often used to implement symbol tables in compilers, interpreters, and other programming language tools. Symbol tables store identifiers (variables, functions, etc.) along with their associated attributes and facilitate efficient lookup and manipulation.
3. **File System Organization**: BSTs can be used to organize file systems efficiently. In a file system, directories and files can be represented as nodes in a BST, where each node contains information about the file (e.g., name, size, permissions) and pointers to its children (subdirectories or files).
4. **Auto-Completion Systems**: BSTs can be used in auto-completion systems to suggest completions for partially typed words or phrases. Words or phrases can be stored in a BST, and efficient prefix-based searching can be performed to suggest completions.
5. **Range Queries**: BSTs can efficiently support range queries, such as finding all elements within a given range. By performing inorder traversal with additional checks based on the range, BSTs can quickly find elements within a specified range.
6. **Balanced Trees**: Variants of BSTs, such as balanced binary search trees like AVL trees and Red-Black trees, are used in applications where maintaining balance is crucial to ensure efficient performance for all operations. These trees guarantee logarithmic time complexity for search, insert, and delete operations, unlike general BSTs.
7. **Priority Queues**: BSTs can be used to implement priority queues, where each element has a priority associated with it. Elements are stored in the tree based on their priority, and the highest priority element can be efficiently retrieved.
8. **Online Algorithms**: BSTs are used in various online algorithms, such as tracking the median of a stream of data or maintaining sliding windows efficiently.

**MERGE SORT**

Merge sort is a comparison-based sorting algorithm that follows the "divide and conquer" strategy to sort elements in a list or array. It divides the input array into two halves, recursively sorts the two halves, and then merges the sorted halves to produce a single sorted array.

**ALGORITHM:**

Here's a step-by-step explanation of how merge sort works:

1. **Divide**: The unsorted list is divided into two halves recursively until each half contains only one element. This is the base case of the recursion.
2. **Conquer**: After division, each individual half is sorted recursively.
3. **Merge**: The sorted sublists are then merged to produce a single sorted list. This is done by comparing the elements of the two sublists and merging them into a new list in sorted order.

The merge operation is the key to the merge sort algorithm. It takes two sorted arrays and combines them into a single sorted array. This process is repeated recursively until all sublists are merged and the entire list is sorted.

Merge sort is known for its stable sorting nature and efficient time complexity of O(n log n) in the average and worst-case scenarios, making it suitable for sorting large datasets. Additionally, it is a stable sorting algorithm, meaning that the relative order of equal elements is preserved in the sorted output.

**PYTHON PROGRAM TO IMPLEMENT MERGE SORT:**

Import matplotlib

Import matplotlib.pyplot as plt

Matplotlib.use(‘TKAgg’)

Import math

Def margesort(arr):

If len(arr)>1:

Mid=int(len(arr)/2)

L=arr[:mid]

R=arr[mid:]

mergesort(L)

mergesort(R)

i=0

j=0

k=0

while i<len(L) and j<len(R):

ifL[i]<=R[j]:

arr[k]=L[i]

i=i+1

k=k+1

elif L[i]>R[j]:

arr[K]=R[j]

k=k+1

j=j+1

arr=[ ]

s=int(input(“enter the array size:”))

while True:

if len(arr)<s:

a=int(input(“enter the array elements:”))

arr.append(a)

else:

break

print(“before sorting:”,arr)

mergesort(arr)

print(“after sorting:”,arr)

x=list(range(1,1000))

plt.plot(x,[y\*math.log(y,2)for y in x])

plt.title(“merge sort Time complexity is 0(nlogn)”)

plt.xlabel(“input”)

plt.ylabel(“time”)

plt.show()

**LOGIC FOR MERGE SORT:**

explanation of the merge sort logic:

* **Divide**:
  + Split the array into two halves.
  + Recursively apply merge sort to each half until each half contains only one element.
* **Conquer**:
  + Sort each half using the merge sort algorithm.
* **Merge**:
  + Compare the elements of the two sorted halves.
  + Place the smaller (or larger, depending on the sorting order) element into the result array.
  + Repeat this process until all elements from both halves are merged into the result array.

This process continues recursively until all sublists are sorted and merged, resulting in a single sorted array.

**MERGE SORT WORK IN MEMORY:**

Merge sort has a notable characteristic regarding its memory usage, especially when compared to other sorting algorithms like quicksort. Let's break down how merge sort utilizes memory:

1. **Recursive Nature**: Merge sort is a recursive algorithm. When it divides the array into halves, it continues this process until each half contains only one element. This recursive division process results in a series of function calls being placed on the call stack. Each call stack frame contains information about the sub-array being processed.
2. **Temporary Arrays**: Merge sort typically uses additional memory to store temporary arrays during the merging phase. When merging two sorted sub-arrays, it needs a temporary array to hold the merged result before copying it back to the original array. The size of these temporary arrays depends on the size of the input array being sorted.
3. **Space Complexity**: The space complexity of merge sort is O(n), where n is the number of elements in the input array. This is because, in the worst-case scenario, merge sort requires additional memory equal to the size of the input array for the temporary arrays used during the merging phase. However, merge sort has a smaller constant factor in its space complexity compared to other algorithms like quicksort, making it more memory-efficient in practice.
4. **In-place Merge Sort**: While the typical implementation of merge sort uses additional memory for temporary arrays, it is possible to implement an in-place version of merge sort. In this approach, merging is done directly within the original array without using additional memory for temporary arrays. However, implementing an in-place merge sort is more complex and may not be as efficient in practice due to increased overhead.

**APPLICATION OF MERGE SORT:**

1. **Sorting Lists/Arrays**: The most basic application is sorting a list or array of elements. Merge sort can efficiently handle sorting large lists or arrays, making it suitable for tasks like sorting user data, organizing files, or ordering numerical data.
2. **Sorting Data from Files**: Merge sort can be used to sort data read from files, especially when dealing with large datasets that cannot fit entirely into memory. It is commonly used in external sorting algorithms where data is read from disk in chunks and sorted using merge sort.
3. **Database Operations**: Merge sort is often used in database systems for sorting query results, indexing data, or organizing records. It provides predictable performance and stability, making it suitable for various database operations.
4. **Parallel Processing**: Merge sort can be parallelized to take advantage of multiple CPU cores or distributed computing environments. In Python, libraries like multiprocessing or concurrent.futures can be used to parallelize merge sort for sorting large datasets more efficiently.
5. **Data Analysis**: Merge sort can be used in data analysis tasks where sorting is a critical step. For example, it can be used to sort data before performing aggregation, filtering, or joining operations in data processing pipelines.
6. **Searching and Ranking Algorithms**: Merge sort can be used as a key component in searching and ranking algorithms, such as binary search or ranking algorithms used in information retrieval systems. Sorted data structures like binary search trees or priority queues often rely on merge sort for efficient sorting operations.
7. **Merge Sort Variants**: Variants of merge sort, such as bottom-up merge sort or natural merge sort, can be applied to specific scenarios where memory usage or performance constraints need to be addressed differently.

**RABIN KARP:**

The Rabin-Karp algorithm is a string searching algorithm that efficiently finds occurrences of a given pattern within a text. It's particularly useful for finding multiple occurrences of the same pattern in a large text or for finding one pattern in multiple texts simultaneously.

**RABIN-KARP ALGORITHM WORKS:**

1. **Hashing**: The algorithm begins by converting both the pattern and substrings of the text into hash values. It uses a rolling hash function to calculate these hash values incrementally. This rolling hash function efficiently updates the hash value of each substring by considering the previous substring and the newly added character.
2. **Comparison**: Once the hash value of the pattern and a substring match, the algorithm compares the characters of the pattern and the substring directly to confirm the match. If the hash values match but the characters do not, it's considered a spurious hit, and the algorithm moves on to the next substring.
3. **Rolling Hash**: To efficiently compare each substring with the pattern, the algorithm uses a rolling hash function. This function updates the hash value of each substring based on the hash value of the previous substring and the newly added character. By doing so, it avoids recalculating the hash value from scratch for each substring.
4. **Handling Collisions**: Since the hash function may produce collisions (i.e., different strings mapping to the same hash value), the algorithm may produce false positives. To address this, the Rabin-Karp algorithm uses a technique called "Monte Carlo" or "Las Vegas" to reduce the likelihood of false positives. This involves verifying potential matches by comparing the characters directly.

**PYTHON PROGRAM TO IMPLEMENT RABIN KARP**

def rabin\_karp(pattern, text):

# Define prime number for hashing

prime = 101

pattern\_length = len(pattern)

text\_length = len(text)

pattern\_hash = calculate\_hash(pattern, pattern\_length, prime)

text\_hash = calculate\_hash(text[:pattern\_length], pattern\_length, prime)

matches = []

for i in range(text\_length - pattern\_length + 1):

if pattern\_hash == text\_hash and pattern == text[i:i+pattern\_length]:

matches.append(i)

if i < text\_length - pattern\_length:

# Update rolling hash

text\_hash = recalculate\_hash(text\_hash, text[i], text[i+pattern\_length], pattern\_length, prime)

return matches

def calculate\_hash(s, length, prime):

hash\_val = 0

for char in s:

hash\_val = (hash\_val \* prime + ord(char)) % prime

return hash\_val

def recalculate\_hash(old\_hash, old\_char, new\_char, length, prime):

new\_hash = (old\_hash - ord(old\_char) \* pow(prime, length - 1)) % prime

new\_hash = (new\_hash \* prime + ord(new\_char)) % prime

return new\_hash

# Example usage:

text = "ABABCABABCDABCABCDABCDABCABDE"

pattern = "ABCD"

print("Pattern found at indices:", rabin\_karp(pattern, text))

**LOGIC FOR RABIN KARP**

1. **Preprocessing**:
   * Calculate the hash value of the pattern.
   * Calculate the hash value of the first substring of the text with the same length as the pattern.
2. **Iterative Search**:
   * Iterate through the text, comparing hash values of substrings with the hash value of the pattern.
   * If the hash values match, compare the characters of the pattern and substring directly to confirm the match. If they match, record the index where the pattern is found.
   * If the hash values match but the characters do not match, it's considered a spurious hit. Move on to the next substring.
   * Update the hash value of each subsequent substring using a rolling hash function to efficiently calculate hash values for overlapping substrings.
3. **Rolling Hash Function**:
   * A rolling hash function is used to update the hash value of each substring efficiently as we move from one substring to the next.
   * When moving from one substring to the next, instead of recomputing the hash value from scratch, the rolling hash function updates the hash value based on the previous hash value and the characters that are removed and added.
4. **Handling Hash Collisions**:
   * Since hashing can produce collisions (i.e., different strings mapping to the same hash value), the Rabin-Karp algorithm may produce false positives.
   * To reduce the likelihood of false positives, the algorithm may employ additional checks, such as comparing the characters directly if hash values match (Monte Carlo variant) or verifying potential matches only after confirming that hash values match (Las Vegas variant).
5. **Repeat**:
   * Continue this process until the end of the text is reached or all occurrences of the pattern are found.

**RABIN KARP WORKS IN MEMORY:**

1. **Pattern Hashing**: The Rabin-Karp algorithm starts by calculating the hash value of the pattern to be searched. This hash value is typically stored in memory, requiring space proportional to the length of the pattern. However, the space required for storing the hash value is constant relative to the length of the text being searched.
2. **Text Hashing**: During the search process, Rabin-Karp calculates the hash value of each substring of the text. This hash value is calculated incrementally using a rolling hash function, which updates the hash value of each substring based on the hash value of the previous substring and the characters that are removed and added. The rolling hash function avoids the need to store hash values for all substrings separately, reducing the memory overhead.
3. **Temporary Variables**: Rabin-Karp may require additional memory for storing temporary variables, such as loop counters, hash values of substrings, or indices of matches found. However, the memory usage of these temporary variables is typically minimal and constant relative to the size of the input text.
4. **Space Complexity**: The overall space complexity of the Rabin-Karp algorithm is O(m), where m is the length of the pattern. This is because the algorithm requires additional memory proportional to the length of the pattern for storing the hash value. However, the memory usage remains independent of the length of the text being searched.

**APPLICATION OF RABIN KARP**

1. **Text Processing and Searching**: The primary application of Rabin-Karp is in searching for occurrences of a pattern within a text efficiently. It's used in text processing tasks such as searching for keywords or substrings in large documents, parsing log files, or implementing search functionalities in text editors and word processors.
2. **Plagiarism Detection**: Rabin-Karp can be used in plagiarism detection systems to efficiently compare documents and identify similarities between them. By hashing substrings of documents and comparing hash values, Rabin-Karp can quickly identify potential matches or similarities, making it suitable for large-scale plagiarism detection tasks.
3. **Spell Checkers**: Rabin-Karp can be employed in spell checkers to efficiently search for misspelled words within a text. By hashing substrings of words and comparing hash values, Rabin-Karp can quickly identify potential matches or suggestions for correcting misspelled words.
4. **Genomic Sequence Analysis**: In bioinformatics, Rabin-Karp can be used for searching patterns or motifs within genomic sequences efficiently. It's employed in tasks such as DNA sequence alignment, searching for sequence patterns in genetic databases, or identifying common motifs in protein sequences.
5. **Data Compression**: Rabin-Karp can be used in data compression algorithms to efficiently search for repeated patterns within a data stream. By identifying repeated patterns and replacing them with shorter representations or references, Rabin-Karp contributes to the compression of data, reducing storage space and transmission bandwidth requirements.
6. **Network Security**: Rabin-Karp can be utilized in network security applications for searching for known patterns or signatures of malicious code or network attacks within network traffic or log files. It helps in intrusion detection, malware detection, and analyzing network traffic patterns efficiently.

**Bottom of Form**