Name: Kanupriya Sharma
Subject: DS

Batch: C2-1
Roll no: C074

Experiment - 1

Aim: To implement stack using menu driven approach.

Theory: Stack is an important data structure which stores its elements in an ordered manner. A stack is a linear data structure in which the elements in a stack are added and removed only from one end which is called the top. Hence, a stack is called LIFO (Last in first Out) data structure as the element that was inserted last is the first one to be taken out.

The three basic operations of stack are as follows:

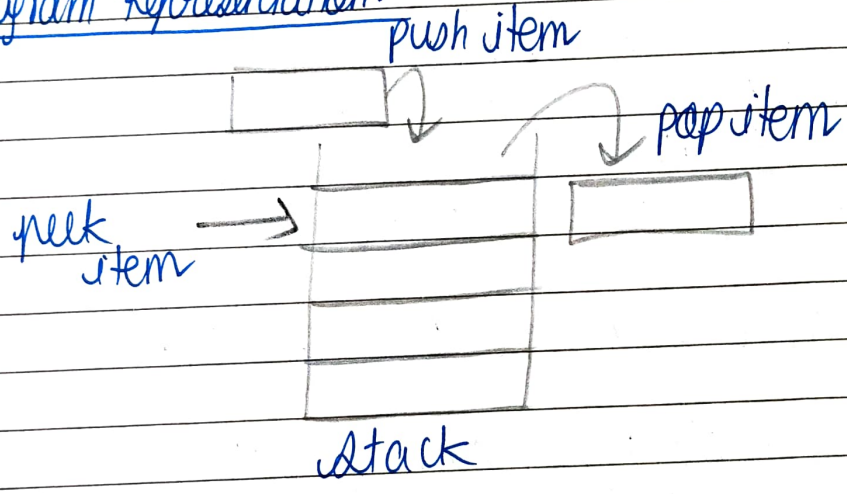
1) Push operation: The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. However, before inserting the value we must first check if the stack is full or not. If full then no more insertions can be done.

2) Pop operation: The pop operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if the

stack is empty or not. If empty then no deletion can be done.

3) Peek operation: Peek operation returns the value of the topmost element of the stack without deleting it from the stack. However, the peek operation first checks if the stack is empty or not, if empty then no peek operation can be performed.

Diagram Representation:



Conclusion: So, through this experiment I got to learn that stacks are powerful data structure. They are used to manage function calls. When a function is called its content is pushed into stack. When the function finishes (its content is popped. Many software applications (like text editors) also use stack to keep track of users actions.

Experiment: 2

Name: Kanupriya Sharma
Batch: C2-1
Subject: DS
Roll no: C074

Aim: To convert infix expression to a postfix expression using a stack.

Theory: Infix expressions are the standard way of writing expressions that are readable for humans but difficult for computers to evaluate directly due to operator precedence and associativity rules.

Key concepts: 1. Infix expression: An expression where operator are between operands. Eg: A + B

2. Postfix expression: An expression where operators ~~follow~~ is after operands. Eg: AB+

3. Prefix expression: An expressions where operator is before operands. Eg: +AB

4. Stack: A data structure that follows last in, first out (LIFO) principle. It used to hold operators and intermediate results during the conversion process.

Infix to Postfix Algorithm

1. Initialize an empty stack for storing operators and an empty string for the postfix result.

2. Scan the expression from left to right.

→ If the character is an operand (a number/letter), append it directly to the postfix result.

→ If the character is an operater (+, -, *, /, %) pop from the stack to the postfix result until the stack is empty or the top of the stack has ~~the~~ an operator with lower precedence or a left parenthesis. Push the current operator onto the stack.

→ If the operater is a left parenthesis, '(' push it onto the stack.

→ If the character is a right parenthesis pop from the stack to the postfix result until a left parenthesis is at the top of stack. Remove the left parenthesis from the stack.

3. Pop all the operater from the stack and append them to your postfix result.

Operater precedence and associativity

→ Operator have : ~~@~~ ^ > * / > + / -
this is precedence levels.

Associativity: Except ^ all the operators have left to right associativity.

Conclusion: The experiment demonstrates that using a stack to convert an infix expression to a postfix expression simplifies the process of expression evaluation. The stack effectively manages the operators and their precedence, allowing for the correct ordering of operators in the postfix expression. This conversion is crucial for creating interpreters and compilers that need to evaluate expressions efficiently. The implementation of this algorithm in C highlights the practical application of data structures in solving real world problems in computer science.

Experiment-3

Name: Kanupriya Sharma
Subject: DS

Batch: CO-1
Roll no: CO74

Aim: To implement and perform operations on a Singly Linked list.

Theory: A linked list is a dynamic data structure that consists of a sequence of elements called nodes, where each node contains:

i) Data: The value to be stored
ii) Pointer (next): A pointer to the next node in the sequence.

Unlike arrays, which have a fixed size and store elements in contiguous memory locations, linked list store nodes in arbitary memory locations. This allows a linked list to grow and shrink.

Types: i) Singly linked list: Each node points to the next node and the last node points to NULL.
ii) Doubly linked list: Each has two pointers - one to the next node and one to the previous node.
iii) Circular linked list: The last node points back to the first node, forming a circular structure.

Operations on Singly linked list:

1. Insertion: a) At the beginning: A new node can be inserted at the start of the list. The new node's pointer will

point to the current head and the head will be updated to point to new node.

b) At the end: A new node can be added to the end of list. The pointer of current last node is updated to point the new node.

c) At a specific position: The list can be traversed to a specific position and the new node can be inserted by adjusting the pointers of adjacent nodes.

2. Deletion: a) By value: To delete the node with a specific value, the list is traversed to find the node. Once found, the previous node's pointer is updated to skip over the node to be deleted, effectively removing it from the list.

b) At the beginning: Deleting the head node involves pointing the head to the next node in list.

c) At end: The second-last node's pointer is set NULL

3. Search: Searching for a node involves traversing the linked list from head, node by node, until the required value is found or the end of list is reached.

Conclusion: The experiment successfully demonstrated the implementation of singly linked list. And we are able to insert nodes dynamically, delete nodes without disrupting the structure and search nodes efficiently by traversing through the list.

Experiment-5

Experiment-4

Name: Kanupriya Sharma

Batch: C2-1
Roll no: C074

Aim: To implement stack, queue using linked list.

Theory: 1) Stack using linked list: A stack is a collection of elements with two primary operations push, pop and follows LIFO rule that is last in first out.

Now, using linked list the a stack can grow or shrink in size dynamically as elements are added or removed without the need for predefined size constraints.

Implementation: Each node in the stack consist of two components; data and next pointer. The data stores the value of the element and next pointer points to the nextnode of stack.
The top pointer refers to the top of element of stack.

Operations: 1) Push operation: A new node is created and its next pointer is assigned to the current top. The top is then updated to this new node.

2) Pop operation: The element at top is removed. The top is updated to point to the next node and the removed node is freed from memory.

3) Peek operation: Returns the data of the top element without removing it.

Advantages: Memory is allocated only when needed, avoiding wastage. No limit on stack size.

2. Queue using linked list: A queue is a linear data structure that allows insertion at one end (rear) and deletion at other (front).

Implementation: Each node in the queue has:
data: to stores the value of element
next pointer: points to the next node of queue.
The front pointer refers to the front element while the rear pointer refers to the last element

Operations: 1) Enque: A new node is created and added at rear. The rear pointer is updated to this new node.
2) Dequeu: The element at front is removed and front is updated to next node and is freed from memory.
3) Peek: Returns the rear data.

Advantages: Efficient insertion, deletion. Dynamically sized

Conclusion: It demonstrates the successful implementation of stack, queue. It ~~over~~ offers various advantages over normal normal array such as dynamic memory allocation, ease of insertion, deletion. Eliminates the drawbacks of fixed size associated with array.

Experiment-5

Name: Kanupriya Sharma
Roll no: C074
Batch: C2-1
Subject: DS

Aim: To implement and understand polynomial addition using linked list.

Theory: Polynomials can be expressed as:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

Where $a_n, a_{n-1}, \dots a_0$ are the coefficients of the polynomial terms.
$x^n, x^{n-1}, \dots x^0$ are the powers of the variable x.

In a linked list:

→ Each node represent a term in polynomial.
→ Each node contains: 1) coefficient: the numerical factor of the term.
2) Exponent: the power of the variable in the term.
3) Pointer: a reference to the next node in the list.

Steps in Polynomial Addition Using Linked Lists:

1. Create Nodes for Each Term:
   - Two polynomials are represented by two separate linked lists.
   - Each node in the linked list holds a coefficient and an exponent.

2. Traverse and Compare Terms: Start from the

head of both linked lists (representing the highest power term).
Compare the exponents of the terms in both polynomials:
a) If exponents are equal, add their coefficients and the result.
b) If one exponent is larger, copy that term to the result.
c) Move to the next node in the corresponding list.

3. Build the Resultant Polynomial:
→ As the lists are traversed, nodes are created in a new linked list for the resultant polynomial.
→ Terms with the same exponent are combined and others are appended as-is.

4. Edge Cases: If one polynomial has terms with higher exponents that the other lacks, these terms are directly added to the resultant polynomial.
→ If one list becomes empty before the other, the remaining terms from the non-empty list are copied to the result.

Example of Polynomial Addition:

Consider two polynomials:

$P_1(x) = 4x^3 + 3x^2 + 2x + 1$

$P_2(x) = 3x^2 + 5x + 6$

Representing these as linked lists:

P1 Linked List:

Node 1: Coefficient = 4, Exponent = 3

Node 2: Coefficient = 3, Exponent = 2

Node 3: Coefficient = 2, Exponent = 1

Node 4: Coefficient = 1, Exponent = 0

P2 Linked List:

Node 1: Coefficient = 3, Exponent = 2

Node 2: Coefficient = 5, Exponent = 1

Node 3: Coefficient = 6, Exponent = 0

Resultant Polynomial:

1. Compare $4x^3$ (P1) and nothing (P2): add $4x^3$ to result.
2. Compare $3x^2$ (P1) and $3x^2$ (P2): add coefficients ($3 + 3 = 6$), giving $6x^2$.
3. Compare $2x^1$ (P1) and $5x^1$ (P2): add coefficients ($2 + 5 = 7$) giving $7x^1$.
4. Compare $1x^0$ (P1) and $6x^0$ (P2): add coefficients ($1 + 6 = 7$) giving $7x^0$.

Result: Presult$(x) = 4x^3 + 6x^2 + 7x + 7$.

Advantages of using linked lists for polynomial operations:

1) Dynamic size: Linked lists allow efficient representation of polynomials without requiring pre-defined array sizes,

more than ideal for polynomials with many or few terms.

3) Sparse representation: Terms with zero or coefficients can be easily omitted, saving memory and avoiding unnecessary operations.

Applications: 1) Signal processing: Polynomials are often used in signal processing to model and manipulate functions.

2) Computer graphics: Polynomial expressions help in transformations and curve fitting algorithms.

Conclusion: Polynomial addition using linked list is a flexible and efficient method for dealing with polynomials of varying sizes and complexities. This approach minimizes memory usage and allows easy manipulation of terms, particularly in cases involving sparse or large polynomials. The use of linked lists also simplifies polynomial operations such as addition, subtraction.

Experiment: 6

Name: Kanupriya Sharma
Subject: DS

Batch: C2-1
Roll no: C074

Aim: Implementation of double ended queue using menu driven approach

Theory: A double ended queue (deque) is an advanced data structure that extends the capabilities of traditional queues and stacks. It allows operation at both ends, making it particularly useful in scenarios where elements need to be accessed or modified from either end.

Advantages: 1. Versatile Operations: Deques support a combination of queue and stack operations. This dual functionality enables more complex data handling.

2. Efficiency: Deques can efficiently manage sequences of data, especially in scenarios where both ends of a sequence require frequent access, such as in sliding window algorithms or managing buffers.

3. Memory Utilization: Unelike fixed sized arrays, deques can grow and shrink dynamically, allowing better memory mangement in applications with unpredictable data loads.

Detailed Operations of a Deque:

1. Insert at front: Add an element to the front of the deque. Useful in scenarios where the least

item needs to be prioritized.

2. Insert at rear: Add an element to the back of the deque. Common in scenarios where items are processed in FIFO manner.
3. Delete from front: Removes the element at the front of the deque. This operation is essential for processing items in a queue like fashion.
4. Delete from rear: Removes the element at the back of the deque. Useful when the last item added is no longer needed, often in applications that need to manage recently added items.
5. Peek front: Retrives the front element.
6. Peek rear: Retrives the rear element.
7. Display: Shows all the elements currently in the deque.

Conclusion: The double ended queue is a highly adaptable and efficient data structure that supports operations from both ends, providing unique advantages in various programming contexts. Its versatility makes it suitable for numerous applications, including task scheduling, undo mechanisms in software and algorithm implementations requiring dynamic data handling. The menu driven approach to interacting with a deque simplifies user engagement, making it an excellent tool for both novice and experienced programmers.

Anupriya

Expt: 7

Name: Kanupriya Sharma
Subject: DS

Batch: C2-1
Roll no: C074

Aim: Implementation of BST using following operations- create, delete, display.

Theory: A Binary Search Tree (BST) is a data structure that facilities efficient searching, insertion and deletion operations. It consist of nodes, where each node contains a key and two children : a left child and a right child. The key in the left child is always less than the parent node's key, while the key in the right child is always greater.

Operations : 1. Create: Start with an empty tree. For each value to be inserted, compare it with the current node. If the value is less, move to the left child; if greater, move to the right child. Repeat until you find an appropriate null position for the new value.

2. Delete: To remove a node from the BST while maintaining its properties. Locate the node to be deleted: i) Node is a leaf (no children): simply remove it.
ii) Node has one child: remove the node and connect its parent to its child.
iii) Node has two children: find the in-order successor (smallest value in the right subtree), replace the node's value with this successor's value and delete the successor node.

3. Display: To visualize the structure of the BST and its values. Typically done using in order traversal, which visits the left subtree, the node itself and then the right subtree. This traversal results in the values being displayed in sorted order, showcasing the BST's properties.

Conclusion: In BST is a fundamental data structure with a logarithmic average time complexity for search, insert, delete operations, making it efficient for dynamic sets of data. Its ability to maintain a sorted order allows for effective range queries and ordered traversals. However, its performance can degrade to linear time complexity in cases of unbalanced trees which is why balancing techniques are often employed in practical applications.

Exp 8

Kanupriya
CO74
C2-1

Aim: Implementation of graph traversal using menu driven program.

Theory: Depth First Search (DFS) and Breadth First Search (BFS) are two graph traversal techniques.

DFS: It is a recursive traversal method that explores as far down a branch as possible before backtracking. It starts at a initial node (source), visiting a node and moving through one adjacent node at a time.

It can be implemented using a stack data structure explicitly or through recursion which implicitly uses the system call stack.

Applications: Ideal for exploring all paths in scenarios like puzzle solving, mazes and path finding in graphs. It is also used in detecting cycles in graphs, analyzing connectivity and generating topological order in directed acyclic graph (DAG).

BFS: It is a iterative method that starts at a source node and explores all its neighbourhood nodes level by level before moving on to nodes at the next level. It uses a queue to keep track of nodes to be explored next.

BFS guarantees the shortest path in a unweighted graph by visiting nodes closest to the source before moving further away.

Applications: BFS is commonly used in finding the shortest path in an unweighted algorithms (like routing algorithms), web crawlers to explore connections and social network to find 6 degree of separation between algorithm.

Conclusion: The experiment successfully implemented menu driven traversal methods for DFS and BFS in C. The results demonstrated that both algorithm could traverse the entire graph, but they differ in their exploration. DFS explores depth wise making it suitable for exploring specific paths. While BFS explores level wise and is more efficient for finding shortest paths in unweighted graph.

Exp: 9

Kanupriya
C074
C2-1

Aim: Write a program to implement Selection, Insertion, quick sort.

Theory: Quick Sort: Algorithm type: Divide and conquer
Mechanism: Quick sort is a divide and conquer sorting algorithm. The basic idea is to select a "pivot" element from the array and partition the other elements into two groups: those smaller than the pivot and those greater than the pivot. After partitioning, the pivot is placed in its correct sorted position. The subarrays are then recursively sorted in the same way.

Steps:
1. Select a pivot element (it can be random or based on a fixed strategy such as picking the first, last or middle element).
2. Partition the array around the pivot such that all elements less than the pivot are on left and all elements greater than pivot are on right.
3. Recursively apply the same process to the left and right sub arrays.

Performance: Best case: $O(n \log n)$: Occurs when the pivot divides the array into nearly equal parts.
Average case: $O(n \log n)$: Typical case where the pivot selection is random or results in balanced partitions.
Worst case: $O(n^2)$: Occurs when the pivot is poorly chosen. (eg: the largest, smallest element), causing

unbalanced partitions (eg: if the array is sorted).

- Space complexity: O(log n): Due to the recursive stack, where each recursive call processes one subarray.
- Stability: Quick sort is not stable because equal elements may not retain their relative order.

Advantages: Very efficient on large databases. In place sorting algorithm, meaning it requires only a small auxiliary stack for recursion.

Disadvantages: Worst case time complexity is $O(n^2)$. Not stable, which could be important for certain applications.

Insertion Sort: Algorithm type: Iterative, Comparison

Mechanism: Insertion sort works similarly to how you might sort playing cards in your hands. The algorithm builds a sorted portion of the array incrementally, one element at a time. It takes the current position in the sorted position by comparing it to the elements before it.

Steps:
1. Start with second element (since a single element array is already sorted).
2. Compare it with the previous element and insert it in the correct position.
3. Move to the next element and repeat the process until the entire array is sorted.

Performance: Best Case: O(n): When the array is already sorted or nearly sorted, only one comparison is needed for each element.
Average, Worst Case: $O(n^2)$: In these cases, each element has to be compared and shifted to its correct position, resulting in quadratic number of comparisons, shifts.
Space complexity: O(1): No extra space.
Stability: Insertion sort is stable, meaning equal elements retain their relative order.

Advantages: Simple to implement. Very efficient for small datasets or nearly sorted data.

Disadvantages: Inefficient on large datasets due to the $O(n^2)$ time complexity. Not suitable for large or completely unordered arrays.

Selection Sort: Algorithm type: Iterative, Comparison based.

Mechanism: Selection sort works by repeatedly finding the smallest (or largest) element from the unsorted part of array and swapping it with the first unsorted element. This process continues until all elements are sorted.

Steps:
1. Start at the first element of array.
2. Find the smallest (largest) element in unsorted part of the array.
3. Swap the element with 1st unsorted element.
4. Move the boundary between the sorted, unsorted portions of array one step forward and repeat the.

process until the entire array is sorted.

Performance: Best, Average, Worst Case: $O(n^2)$: Regardless of the initial order of elements, the algorithm always performs $n(n-1)/2$ comparisons, making the time complexity quadratic.

Space complexity: $O(1)$: No additional memory

Stability: Not stable, as equal elements might not retain their relative order after sorting.

Advantages: Simple to understand and implement. $O(1)$ space.
Disadvantages: Inefficient on large datasets due to $O(n^2)$.

Conclusion: Quick sort, insertion sort, selection sort each have their own strengths and weaknesses. Quick sort is the most efficient for large datasets, especially when optimized, but it can degrade with bad pivot choices. Insertion sort is ideal for small or nearly sorted datasets due to its simplicity and $O(n)$ best case. Selection sort is easy to implement but is rarely used in practice for large datasets due to its $O(n^2)$ time complexity.

Experiment-10

Kanupriya
CO74
C2-1

Aim: Write a menu driven program in C to implement hashing techniques.

Theory: Hashing is an essential data organization and retrieval method where a key is transformed into a unique address (hash value) using a hash function. This address is used to store the data in hash table. The primary objective is to achieve quick access to data by minimizing the number of comparisons.

Characteristics of hashing:

a) Deterministic: The same input key will always produce the same hash value.

b) Efficient: A well designed hash function ensures even distribution of keys and minimizes collisions.

c) Dynamic: Hash tables can resize themselves when they grow beyond a certain load factor to maintain efficiency.

d) Versatile: Useful for diverse application, including cryptography, searching, network data management.

Types of hashing function:

1. Division method:
The hash value is calculated as:

h(key) = key % table-size
Simple and efficient when the table size is prime to reduce clustering.

2. Multiplication Method: Uses a constant value A where (0<A<1) to compute the hash:
h(key) = floor (table-size * A (key * A % 1))

3. Cryptographic hashing: Provides a secure hash function where small changes in the key drastically change the hash value.

Challenges in hashing:

1. Collisions: Occur when two key produce the same hash value. Even the best hash function can produce collisions in practice. Collision handling techniques are crucial to maintain efficiency.

2. Clustering: In linear probing, clusters form as keys are stored consecutively. In quadratic probing, clusters may arise due to repeated patterns in hash values.

3. Load factor: Defines the ratio of the number of keys to the size of hash table. High load factors increases collisons, low load factors waste memory.

4. Hash function design: A poorly designed hash function can result in uneven distribution, increasing search time.

Collision Resolution techniques:

1. Separate Chaining: Each index in the hash table points to a linked list of entries that hash to the same value.
Advantages: Easy to implement and handles high load factors.
Disadvantages: Increased memory usage and slower lookups in long chains.

2. Open Addressing: Collisions are resolved within the hash table by finding alternative slots.
Variants: Linear probing: Checks the next sequential slot until an empty slot is found.
Quadratic probing: Skips slot based on quadratic function to avoid clustering.
Double hashing: Applies a second hash function to calculate the step size.
Advantages: Avoids the overhead of separate structure like linked list.
Disadvantages: Susceptible to performance degradation as the table fills up.

3. Rehashing: When the hash table becomes too full, it is resized, and all keys are inserted using a new hash function or table size.

Advantages of hashing: 1. Speed: Enables fast data retrieval compared to other data structure.
2. Simplicity: Easy to implement
3. Scalability: Handles dynamic data growth through rehashing.
4. Flexibility: Can store diverse data and handle varied use case.

Disadvantages of hashing: ①
1. Memory overhead: Requires more memory than simple data structure.
2. Collision management: A poorly chosen collision resolution strategy can degrade performance.
3. Dependence on hash function: The efficiency of hashing is highly dependent on choice of hash function.

Conclusion: Hashing provides a robust and efficient mechanism for organizing and accessing data, ensuring minimal time complexity under ideal conditions. By employing proper hashing techniques and collision resolution methods, developers can achieve scalable and high performance systems. Despite challenges like collision and memory overhead, hashing remains a cornerstone of computer science with application ranging from database indexing to cryptography.

Aim: To implement searching algorithms.

Theory:

Binary Search: It is a divide and conquer algorithm that operates on sorted arrays or lists. It works by repeatedly dividing the search interval in half. At each step, it compares the target value with the middle element of the array.

→ If target value is equal to the middle element, the search is complete.

→ If the target value is less than the middle element, the search continues on the left left of array.

→ If the target value is greater than the middle element, the search continues on the right half of the array.

The array must be sorted for binary search to work correctly.

Time complexity of O(log n). It drastically reduces the number of comparisons needed compared to linear search.

Linear Search: Linear search is a straightforward searching algorithm where each element of the array is checked one sequentially from the beginning until the target element is found or the entire array has been searched.

→ It starts from the first element, compares it with the target, and moves sequentially to the next element.

→ If the element is found, it returns the index, if not it continues until all elements have been checked.

It works on both sorted and unsorted array.
It has time complexity of O(n) meaning inefficient for large datasets.

Fibonacci Search: It is a variation of binary search that uses fibonacci numbers to divide the search space. Instead of dividing the search interval in half, fibonacci seach uses the closest fibonacci number smaller than or equal to the length of array to divide the array.
→ The algorithm begins by determine two indices based on fibonacci sequence (initially fib(m−2) and fib(m−1) where m is the fibonacci number).
→ The element at these indices is compared to target value and based on comparison the search space is reduced by shifting the indices accordingly.
The array must be sorted. Time Complexity of O(log n).

Conclusion: All three search algorithms aim to locate a target value in dataset. Binary and fibonacci search are highly efficient for sorted data with a logarithmic time complexity making them suitable for larger datasets. Linear search, while simpler is less efficient with a linear time complexity. and is best suited for small, unsorted or datasets. Each algorithm has its own strength and limitation.