Assignment - I

* Application of Data Structure

→ Applications of Data structures are summarized in following points:

1. Data structure are essential for organizing, storing and manipulating data efficiently in computer programs.
2. Arrays are widely used for storing collections of elements and are used in various algorithms, such as sorting and searching.
3. Linked lists are dynamic data structures used for efficient insertion and deletion operations. They are used in implementation of stacks, queues and other data structures.
4. Stacks are employed in applications like function call management, expression evaluation and backtracking algorithms.
5. Queues are used for process scheduling, breadth first search algorithms and simulation systems.
6. Trees are hierarchial structures used in file systems, decision trees and representing hierarchial relationships.
7. Graphs are used to model relationships and connections, finding applications in social networks, routing algorithms and network analysis.

8. Hash table provide efficient key-value pair storage and retrieval, finding use in database indexing, caching mechanisms, and symbol tables.

9. Heaps are useful for priority queues, where the highest or lowest priority element needs to be accessed efficiently.

10. Trees are employed in text search algorithms, auto-completion features and spell checking applications.

11. Graphical data structures are used for representing and manipulating graphical objects such as vectors, polygons and images in computer graphics, image processing and geometric algorithms.

12. Data structures play a crucial role in optimizing perfomance, memory usage and area of ease of implementation in various applications.

* Characteristics of Data Structure

→ Characteristics of Data structure are summarized as follows:

1. Organization:
   Data structures provide a way to organize and structure data effectively.

2. Memory management:
   They manage memory allocation and deallocation for efficient storage.

3. Access Methods:
Data structures define methods for accessing and manipulating data elements.

4. Efficiency:
They are designed to optimize time and space complexity for operations.

5. Perfomance:
Different data structures have varying perfomance characteristics.

6. Data integrity:
Data structures enforce rules to maint the validity and consistency of data.

7. Modularity and Reusability:
They can be modular and reusable, promoting code reuse.

8. Scalability:
Data structures should handle large data volumes and increased workloads efficientl

9. Abstraction:
They provide a simplified interface whi hiding implementation details.

10. Suitability for specific applications:
Different data structures are optimized for specific operations and data access patterns.

Assignment II

* Operations on Data structure

→ Operations on data structure refer to the various actions that can be performed on the data stored within them. Different data structures support different operations, and the choice of data structure depends on the specific requirements and constraints of the problem at hand.

Here is brief overview of some common operations performed on data structures:

1. Access:

The operation involves retrieving or accessing the data stored within a data structure. For example; accessing an element at a specific position in an array or retrieving the value associated with a particular key in a dictionary.

2. Insertion:

Insertion involves adding new data into a data structure. This can include appending an element at the end of an array, inserting an element at a specific position in a linked list, or adding a key value pair to a hash table.

3. Deletion:

Deletion refers to removing data from a data structure. For instance,

removing an element from an array, deleting a node from a linked list, or removing a key value pair from a hash table.

4. Search:

Searching involves finding a specific element or value within a data structure. This can be done by iterating over the elements in an array or linked list, searching through a tree structure or using search algorithms like binary search for sorted arrays.

5. Update:

Updating involves ~~finding~~ modifying the data stored within a data structure. For example: changing the value of an existing element in an array or linked list, updating the properties of a node in a tree, or modifying the value associated with a key in a hash table.

6. Traversal:

Traversal refers to visiting or processing each element in a data structure in a specific order. This is commonly done using iteration or recursion. For instance, traversing an array from the first element to the last, visiting each node in a tree in a particular order (eg: in-order, pre-order, post-order) or

iterating over the elements in a linked list.

7. Sorting: Sorting involves arranging the element in a data structure in a specific order. This can be done using various sorting algorithms such as bubble sort, insertion sort, selection sort, merge sort, quick sort or heapsort

8. Merging:
Merging involves combining two or more data structures into a single data structure. For example, merging two sorted arrays into a single sorted array, merging two sorted linked lists into a single sorted linked list, or merging two balanced binary search trees into a single balanced binary search tree.

9. Create:
Create operation initializes a new instance of a data structure by allocating memory and setting up its components. It prepares the data structure for storing and manipulating data, ensuring it is ready for use.

Assignment III

* Explain about expression format / expression notation in DSA in brief

→ Infix, postfix and postfix notations are different ways to represent mathematical expressions in DSA.

1. Infix notations:

This is the most common notation, where operators are placed between operands. It follows the usual convention.
for eg: "a+b" or "(a+b) - (c * d)".

2. Postfix notation:

In postfix notation, operators are placed after their operands. This eliminates the need for parantheses and makes the expression unambiguous.
for eg: "ab+"

3. Prefix notation:

In prefix notation, operators are placed between their operands. Like postfix notation, it eliminates the need for parantheses and makes the expression unambigious.
for eg: "+ab".

Assignment III

Convert infix to postfix.

A + (B * C - (D / E $ F) * G) * H

Scan character	Stack	Postfix string
A		A
+	+	A
(	+(	A
B	+(	AB
*	+(*	AB
C	+(*	ABC
-	+(-	ABC
(	+(-(	

Assignment III

Convert infix to postfix.

A + (B * C - (D / E $ F) * G) * H

→

Scan character	Stack	Postfix string
A		A
+	+	A
(	+(	A
B	+(	AB
*	+(*	AB
C	+(*	ABC
-	+(-	ABC*

Scan Character	Stack	Postfix Expression
(	+(-(	ABC*
D	+(-(	ABC*D
/	+(-(/	ABC*D/
E	+(-(/	ABC*DE
$	+(-(/$	ABC*DE
F	+(-(/$	ABC*DEF
)	+(-	ABC*DEF$/
*	+(-*	ABC*DEF$/
G	+(-*	ABC*DEF$/G
)	+	ABC*DEF$/G*-
*	+*	ABC*DEF$/G*-
H	+*	ABC*DEF-$/G*-H
ϕ (empty)		ABC*DEF-$/G*-H*+

postfix expression: ABC * DEF- $ /G* - H*+

Assignment IV:

Q. Write recursive algorithm for fibonacci series

→ Algorithm:

Step 1: Start

step 2: If (n == 1 or n == 0)
return n

Step 3: Else
return (fibo(n-1) + fibo(n-2))

Step 4: Exit

Q. Write recursive algorithm and recursive program for sum of natural numbers. Also make recursion tree.

→

Algorithm:

Step 1: Start

Step 2: If (n == 0)
return 0

Step 3: Else
return (n + sum(n-1))

Step 4: Stop

Program:

```
#include<stdio.h>
int sum(int)
int main()
{
    int n, resut;
    printf("Enter no. of terms");
    scanf("%d", &n);
    result = sum(n);
    printf("Sum of natural numbers up to %d is
            %d\n", n, result);
    return 0;
}
int sum(int n)
{
    if(n==1)
    return 1;
```

```
else
return (n + sum(n-1));
}
```

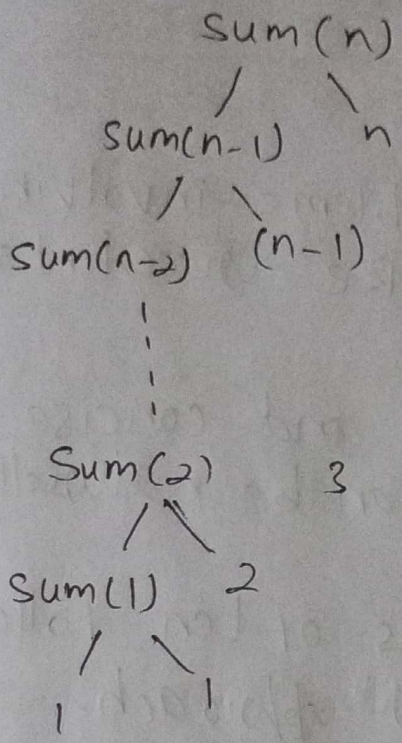


Fig: Recursion tree

Assignment 5:

Q. Recursion vs Iteration with syntax and example.

→

Here is a comparision of recursion and iteration in context of data structure and algorithms.

Recursion:

i) Well suited for problems involving recursive data structures like trees, linked lists or graphs.

ii) Can provide elegant and concise solutions for problems that can be naturally divided into subproblems.

iii) Recursive algorithms often follow the "divide and conquer" approach.

iv) Examples: tree traversal (pre-order, in-order, post-order), recursive backtracking (eg: solving mazes) and recursive sorting algorithms (eg: merge sort, quick sort).

Here is a simplified syntax for recursion in C:

1. Base Case(s):
   - Check if the problem can be solved directly.
   - If yes, return the solution.
2. Recursive Case(s):
   - Modify the problem to a smaller version.

- Call the same function with the modified problem.

3. Handle the base case(s):
- If the base case(s) are met, return the solution directly.

4. Handle the recursive case(s):
- Call the function recursively with the modified problem.

Syntax of recursion in C:

```
return_type recursive_function(parameters){
    if(base_case_condition){
        return base_case_solution;
    }
    modified_parameters = modify_parameters(parameters);
    return recursive_function(modified_parameters);
}
```

Iteration:

i) Generally more suitable for problems that can be solved using loops and iteration constructs.
ii) Allows for more control over the flow of execution and step size.
iii) Commonly used for linear data structures like arrays, lists or stacks.
iv) Examples: linear search, binary search, sorting algorithms (eg, bubble sort, insertion sort) and dynamic programming algorithms.

Here is a simplified syntax for iteration in general sense:

1. Initialization: Initialize any necessary loop before the loop.
2. Loop condition: Specify the condition that determines whether the loop should continue executing.

3. Loop body: Include the code that is executed repeatedly within the loop.
4. Update loop variables: Define the necessary operations to update loop variables within each iteration.

Syntax of iterations in C:

```
Initialization;
while(loop condition)
    Loop body;
    Update loop variables;
}
```

Assignment:

6. Application of Queue.

→ Queue's applications are stated and elaborated as follows:

1. Operating Systems:
In operating systems, queues are used for process/task management. The ready queue holds processes waiting to be executed, while the job queue keeps track of submitted processes awaiting scheduling. The CPU scheduling algorithm selects processes from these queues based on arrival time and priority.

2. Printers and I/O Devices:
Queues are often employed to manage print jobs or input/output operations. When multiple print requests are received, they are placed in a queue and the printer processes them one by one, following the order of arrival. Similarly, queues can be used to

handle input/output operations from devices such as keyboards or network interfaces.

3. Web servers:

In web servers, queues are used to handle incoming requests. When a user requests a web page the request is added to a queue and the web server processes the requests in the order they arrived. This ensures fairness and prevents overload.

4. Message Queues:

Message queues are widely used in distributed system and interprocess communication. They enable asynchronous communication between different components or systems. Messages are sent to a queue and then retrieved by the recipient when they are ready to process them.

5. Task scheduling:

Queues are used for managing scheduled tasks or jobs in systems like task schedulers or batch processing systems. The tasks are added to a queue and processed according to their priority or scheduling criteria.

6. Networking:

Queues are used to buffer incoming packets when the network is congested. The packets are stored in queues until they can be transmitted or processed further.

7. Breadth-First search:

In graph algorithms, queues are utilized for breadth first search (BFS). The BFS algorithm

explores all the vertices of a graph at the same depth level before moving to the next level. A queue is used to maintain the order of exploration.

There are many more applications of queue. Some of them are explained above.

Assignment:

Q. Write about advantages and application of priority que

→

The advantages of priority are as follows:

1. Quick access to highest priority element.
2. Efficient insertion and retrieval of elements.
3. Flexibility in prioritizing elements based on different criteria.
4. Suitable for real time systems and event driven simulations.
5. Essential for graph algorithms like Dijkstra's algorithm.
6. Effective for job scheduling and resource allocation
7. Useful in data compression and Huffman coding.
8. Enables load balancing in distributed systems.
9. Enhances event driven programming.
10. Optimizes performance and efficiency in various applications.

Applications of priority queue are as follows:

1. Task scheduling and task prioritization.

Dijkstra's algorithm for finding shortest points in graph.

3. Event driven simulations and event processing.
4. CPU and resource allocation in operating systems.
5. Data compression and Huffman coding.
6. Network packet scheduling and QoS management.
7. Load balancing in distributed systems.
8. Real time systems and time critical applications.
9. Emergency service dispatch and response prioritization.
10. Efficient management of event driven workflows.

2. Explain in brief about Ascending priority queue and Descending priority queue.

→ Ascending Priority Queue:

In an ascending priority queue, each element is associated with a priority value. The element with the lowest priority value is considered the highest priority. When elements are inserted into the queue, they are placed in a way that ensures the element with the lowest priority value is at the front of the queue. this means that when you want to remove an item from the queue the one with the lowest priority value will be dequed first.

Descending Priority Queue:

In an descending priority queue, each element is associated with a priority value, and the element with the highest priority value is considered as the highest priority. When elements are inserted into the queue they are arranged in a way that ensures the element with the highest priority value is at the front of the queue. This means that when you want to remove an item from the queue the highest priority will be dequed fir

Assignment:

Q. Display and search for SLL.

→

* Algorithm to display data in SLL

1. If (START == NULL)
   a. Display "No Node"
2. Else
   a. TEMP = START
   b. While (TEMP != NULL)
      i. Display TEMP → info
      ii. Display TEMP → next
   c. End while
3. Exit

* Algorithm to search data in SLL.

1. Input data to be searched, x
2. TEMP = START
3. If (START == NULL)
   a. Display "No Node"
4. While (TEMP != NULL)
   a. If (TEMP → info == x)
      i) Display "x is found".
      ii) break
   b. Else
      i) TEMP = TEMP → next
5. End while
6. If (TEMP == NULL)
   a. Display "x is not found".
7. Exit

Q. Delete in SLL

* Deleting first node in SLL

→
1. If (START == NULL)
   a. Display "No Node"
2. Else if (START → next == NULL)
   a. Delete START
   b. START = NULL
3. Else
   a. TEMP = START
   b. START = START → next
   c. Delete TEMP
4. Exit

* Deleting last node in SLL

1. TEMP = START
2. While (TEMP → next → next != NULL)
   a. TEMP = TEMP → next
3. End while
   a. Delete TEMP → next
   b. TEMP → next = NULL

* Deleting node from particular position.

1. Input position POS.
2. i = 1, TEMP = START

3. While (i < POS - 1)
   a. TEMP = TEMP → next
   b. i++
4. End while
5. HOLD = TEMP → next
6. TEMP → next = HOLD → Next
7. Delete HOLD
8. Exit

Assignment:

Q. Advantages of DLL over SLL

→

The advantages of DLL over SLL are as follows:

i) Bidirectional Traversal:

Doubly linked lists allow traversal in both directions (forward and backward) whereas singly linked lists only support forward traversal, requiring additional time for reverse traversal.

ii) Insertion and deletion:

Doubly linked lists faciliate efficient insertion and deletion operations at both the beginning and end of the list as they can easily adjust the links in both directions while singly linked list require traversal from head or tail, resulting in slower operations. for certain cases,

3. Previous Node Access:
Doubly linked lists store a reference to the previous node, enabling easy access to the preceding element which is helpful for various operations and algorithms, whereas singly linked lists lack this direct access, requiring additional traversal to find the previous node.

4. Reversing the List:
Reversing a doubly linked list is a straight forward operation by swapping the next and previous links of each node, while reversing a singly linked list requires extra data structures or significant modifications to the links, making it less efficient.

5. Implementation of Algorithms:
Doubly linked lists are preferred for certain algorithms that require bidirectional traversal, such as undo/redo functionality or implementing data structures like stacks or queues, whereas singly linked lists may be less suitable due to their unidirectional nature.

Q. Applications of doubly linked list

→ Some applications in common in brief points:

1. Implementing data structures like queues and deque (double ended queues).
2. Undo/redo functionality in text editors and software applications.
3. Browser history navigation, allowing backward and forward movement.

4. Doubly linked list based memory management algorithms, such as the buddy system or linked allocation.
5. Implementing efficient doubly ended priority queues.
6. Circular navigation with a collection like rotating images or circular buffers.
7. Music and media playlist management, enabling easy traversal and manipulation of playlist items.

Assigment

Q. Advantages and Disadvantages of Circular linked list

→

Advantages of Circular Linked List:

1. Circular linked lists allow for efficient insertions and deletions at both the beginning and end of the list
2. Traversing a circular linked list is straight forward due to its cyclic nature, making it easy to iterate through the entire list or implement circular operations.
3. Circular linked lists make efficient use of memory by allowing the last node to point back to the first node, reducing memory waste.
4. They are well suited for implementing circular buffers or queues, where elements are continuously added and removed in a cyclic manner.

Disadvantages of Circular linked list:

1. Implementing and manipulating circular linked lists can be more complex than linear linked lists, requiring special consideration for operations like insertion, deletion and searching.
2. Circular linked list have an additional memory overhead due to the need for an extra pointer in each node to maintain the circular link.
3. Circular linked lists lack direct random access to elements, necessiating traversal from the beginning or a known reference point, which can be inefficient for large lists.
4. Reversing a circular linked list can be challenging due to its cyclic nature, requiring careful swapping and redirection of pointers.

Assignment

Q. Singly linked list as Adt

→ Singly linked list is a common Abstract Data type (ADT) that represents a collection of elements connected through links or pointers. It consists of value and pointer to next node in the list.

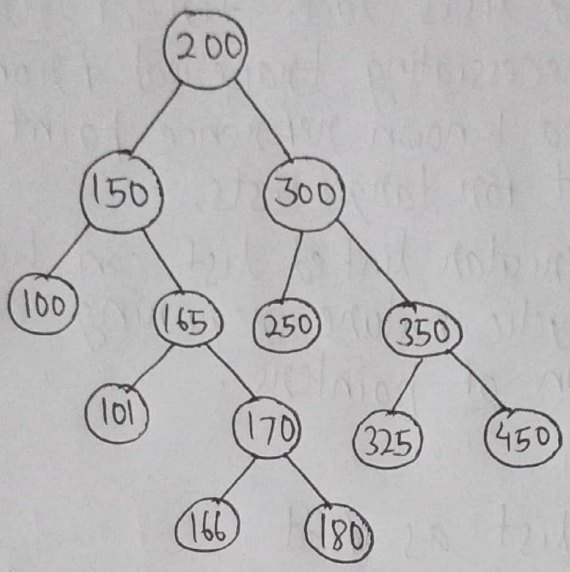
Basic operations propided by singly linked list ADT are:

i) Insertion: Inserting an element at the beginning, end or at specific position in list
ii) Creating: Creating an node
iii) Deletion: Removing an element from the list either from start, end or any position.
iv) Traversal: visiting any node throught the list to access or modify individual elements.
v) searching: Finding the position or presence of specific element in the list.

vi) Merging / Concalenation: Combining two separate linked list into single linked list.

vii) Accessing: Finding the value of an element at a particular position.

Assignment:



⇒

1. Preorder (VLR):

⇒ 200, 150, 100, 165, 101, 170, 166, 180, 300, 250, 350, 325, 450

2. In order (LVR):

⇒ 100, 150, 101, 165, 166, 170, 180, 200, 250, 300, 325, 350, 450

3. Post order (LRV):

⇒ 100, 101, 166, 180, 170, 165, 150, 250, 325, 450, 350, 300, 200