Lab Experiment No. 3

Objective

To compare Arrays with Linked Lists and analyze their performance in different scenarios.

Theory

Arrays and Linked Lists are two fundamental data structures used in computer science to store and manage collections of data. While both serve similar purposes, they differ significantly in their implementation, performance, and use cases.

3.1 Arrays

- **Definition**: A collection of elements stored in contiguous memory locations.
- **Characteristics**:
    - Fixed size.
    - Random access using indices.
    - Homogeneous elements.
- **Operations**:
    - Access: O(1)
    - Insertion/Deletion: O(n) (due to shifting)
    - Search: O(n) (linear search) or O(log n) (binary search if sorted).

3.2 Linked Lists

- **Definition**: A collection of nodes, where each node contains data and a pointer to the next node.
- **Characteristics**:
    - Dynamic size.
    - Non-contiguous memory allocation.
    - Sequential access.
- **Types**:
    - Singly Linked List
    - Doubly Linked List
    - Circular Linked List
- **Operations**:
    - Access: O(n)
    - Insertion/Deletion: O(1) (if the position is known)
    - Search: O(n).

Algorithm & Implementation

Performance Analysis

Scenario 1: Frequent Access

- **Arrays**: Ideal for scenarios requiring frequent access to elements. Random access in O(1) time makes arrays highly efficient.
- **Linked Lists**: Poor performance for frequent access, as traversal is required (O(n)).

Scenario 2: Frequent Insertions/Deletions

- **Arrays**: Inefficient for frequent insertions/deletions, as elements need to be shifted, resulting in O(n) time complexity.
- **Linked Lists**: Efficient for frequent insertions/deletions, especially when the position is known (O(1)).

Scenario 3: Memory Usage

- **Arrays**: Memory-efficient for storing data, as they only store elements.
- **Linked Lists**: Require additional memory for pointers, leading to higher memory overhead.

Scenario 4: Dynamic Data

- **Arrays**: Not suitable for dynamic data, as resizing is expensive.
- **Linked Lists**: Ideal for dynamic data, as nodes can be easily added or removed.

Code

```
#include<iostream>
using namespace std;

int N = 10;

struct studentNode{
   int roll;
   string name,stream;
   studentNode* next;

   studentNode(int roll, string name, string
stream){
      this->roll = roll;
      this->name = name;
      this->stream = stream;
      this->next=NULL;
      ++record_count;
      cout<<"Want to enter more records: (y/n) ";
      cin>>next;
   }while (next!='n');
printf("Entered Student Detials: \n");
   printf("S.No \tRoll No\t Name\t Stream\n");
   for(int i=0;i<record_count;++i){

cout<<i+1<<"\t"<<roll[i]<<"\t"<<name[i]<<"\t"<<
stream[i]<<endl;
   }

   // print all student name in O(n)
   cout<<"All Student name list: (in O(n))"<<endl;
   for(int i=0;i<record_count;++i){
      cout<<i+1<<"\t"<<name[i]<<endl;
```

```
    }
};

int main(){
    int roll[N], record_count=0, _roll;
    string name[N], stream[N], _name,_stream;
    char next;
    studentNode* st;
    cout<<"Enter details below: \n 1. Student Roll
No \n 2. Student Name \n 3. Student Stream
\nEnter records and each entry detials seperated by
space:\n";

    do{
        cin>>_roll>>_name>>_stream;
        //
cin>>roll[record_count]>>name[record_count]>>s
tream[record_count++];
        if(record_count<=N){
            roll[record_count]=_roll;
            name[record_count]=_name;
            stream[record_count]=_stream;
        }else cout<<"Array was of fixed size and
we've exhausted the space.\nLinked list dynamic
so still accepting entries...";
        if(record_count)
            st->next = new
studentNode(_roll,_name,_stream);
        else
            st=new studentNode(_roll,_name,_stream);
    }

    // print any student name in O(1)
    cout<<"Name of Student record by S. No.: (in
O(1) array access) ";
    do{
        int idx;cin>>idx;
        cout<<idx<<"\t"<<name[--idx]<<endl;
        cout<<"Want to view more records: (y/n) ";
        cin>>next;
    }while (next!='n');

    // print any student name in O(n)
    cout<<"Name of Student record by S. No.: (in
O(n) Linked List access) ";
    do{
        int idx;cin>>idx;
        studentNode* temp = st;
        while(temp->next!=NULL){
            if(temp->roll==idx)
                break;
            temp=temp->next;
        }
        cout<<idx<<"\t"<<temp->name<<endl;
        cout<<"Want to view more records: (y/n) ";
        cin>>next;
    }while (next!='n');

    return 0;
}
```

Sample Output

```
Enter details below:
 1. Student Roll No
 2. Student Name
 3. Student Stream
Enter records and each entry details separated by space:
1 Ram CSA
Want to enter more records: (y/n) y
2 Som CSE
Want to enter more records: (y/n) n
Entered Student Detials:
S.No    Roll No  Name    Stream
1       1        Ram     CSA
2       2        Som     CSE
All Student name list: (in O(n))
```

1 Ram
2 Som
Name of Student record by S. No.: (in O(1) array access) 1
1 Ram
Want to view more records: (y/n) n
Name of Student record by S. No.: (in O(n) Linked List access) 2
2 Som
Want to view more records: (y/n) n

Complexity Analysis

Operation	**Arrays**	**Linked Lists**
Access	O(1)	O(n)
Insertion		
- At beginning	O(n) (shifting required)	O(1)
- At middle	O(n) (shifting required)	O(n) (traversal) + O(1)
- At end	O(1) (if space is available)	O(n) (traversal) + O(1)
Deletion		
- At beginning	O(n) (shifting required)	O(1)
- At middle	O(n) (shifting required)	O(n) (traversal) + O(1)
- At end	O(1)	O(n) (traversal) + O(1)
Searching	O(n) (linear search)	O(n)
Traversal	O(n)	O(n)
Memory Overhead	Low (only stores data)	High (stores data + pointers)

Conclusion

Arrays and linked lists are both essential data structures, each with its strengths and weaknesses. Arrays are ideal for static data and frequent access, while linked lists are better suited for dynamic data and frequent modifications. Understanding their performance characteristics helps in selecting the appropriate data structure for a given problem.