TECHNICAL REPORT WRITING

Applications of Single Linked List

NAME: RUPAK SARKAR

ROLL NO.: 14271024036

STREAM: MASTER OF COMPUTER APPLICATION

SUBJECT NAME: DATA STRUCTURES WITH PYTHON

SUBJECT CODE: MCAN-201

COLLEGE NAME: MEGHNAD SAHA INSTITUTE OF TECHNOLOGY

UNIVERSITY NAME: MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY

ABSTRACT

A **Single Linked List (SLL)** is a dynamic data structure widely used in computer science for efficient memory management and data handling. It is essential in implementing **stacks, queues, symbol tables, and graph adjacency lists**. SLLs also play a key role in **undo operations, polynomial arithmetic, and navigation through datasets**, making them useful in software applications and embedded systems. This report explores these applications, highlighting the advantages and limitations of SLLs in various computational scenarios.

CONTENTS

INTRODUCTION

A **Single Linked List (SLL)** is a linear data structure consisting of nodes, where each node contains data and a reference to the next node in the sequence. Unlike arrays, SLLs offer dynamic memory allocation, allowing efficient insertions and deletions without the need for contiguous memory blocks. This makes them particularly useful in applications requiring frequent modifications to data structures.

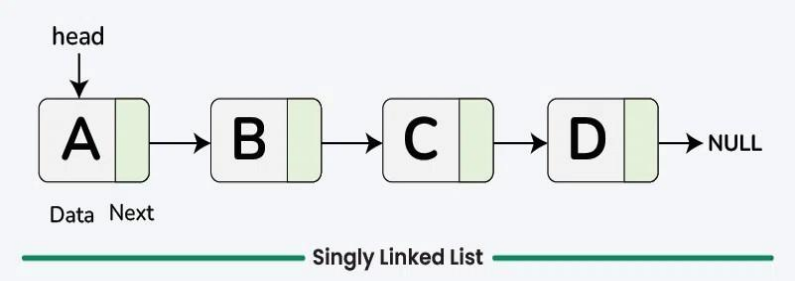
SLLs are widely used in various computing fields, including **data management, memory allocation, and algorithm implementation**. They serve as the foundation for **stacks, queues, symbol tables, and graph representations**, playing a crucial role in optimizing data access and manipulation. Additionally, they are employed in applications such as **text editors (undo operations), polynomial arithmetic, and embedded systems** where efficient memory utilization is essential.

This report explores the practical applications of Single Linked Lists, analyzing their advantages, limitations, and real-world implementations in computing.

TECHNICAL REPORT

Introduction to Single Linked List.

A Single Linked List (SLL) is a linear data structure composed of nodes, where each node contains data and a reference to the next node in the sequence. Unlike arrays, SLLs do not require contiguous memory allocation, making them more flexible for dynamic memory management. They allow efficient insertions and deletions, particularly at the beginning or middle of the list, without requiring costly shifting operations. However, accessing elements in an SLL is slower than in arrays, as traversal must occur sequentially from the head node. Due to these characteristics, SLLs are widely used in applications such as implementing stacks, queues, symbol tables, and graph representations. They are also found in text editors (undo functionality), polynomial arithmetic, memory management, and embedded systems, where efficient memory utilization and dynamic data handling are essential. Despite their advantages, SLLs have limitations, such as increased overhead due to pointer storage and slower random access. Nonetheless, they remain a fundamental data structure in computer science, enabling efficient data organization and manipulation in various applications.



Applications of Single Linked List –

A **Single Linked List (SLL)** is widely used in computer science due to its dynamic memory allocation and efficient insertion and deletion operations. Some of its key applications include:

1. **Implementation of Stacks and Queues**
    - SLLs serve as the foundation for **stacks (LIFO)** and **queues (FIFO)**, which are essential in memory management, expression evaluation, and scheduling algorithms.
2. **Dynamic Memory Allocation**
    - Operating systems use linked lists to manage **free memory blocks** and allocate memory dynamically, improving memory utilization.
3. **Graph Representations**
    - Adjacency lists in graph theory utilize SLLs to efficiently store and traverse graph nodes, optimizing operations like BFS and DFS.

4. **Undo Functionality in Text Editors**
    - Text editors maintain a history of changes using linked lists, enabling **undo and redo** operations efficiently.
5. **Polynomial Arithmetic**
    - SLLs represent polynomials dynamically, allowing efficient storage and computation of polynomial expressions.
6. **Symbol Tables in Compilers**
    - Compilers use linked lists to store variable names, function identifiers, and symbols for faster lookup and retrieval.
7. **Efficient Navigation in Data Structures**
    - Applications like **music players, image viewers, and web browsers** use SLLs to navigate sequentially through data.
8. **Implementation of Hash Tables**
    - Collision handling in **hash tables** is managed using separate chaining with linked lists.
9. **Memory-Efficient Data Handling in Embedded Systems**
    - Due to their low memory overhead, SLLs are preferred in **embedded systems and IoT devices** where resources are limited.

Single Linked Lists remain a fundamental data structure with extensive applications in **software development, operating systems, and real-time systems**, ensuring efficient data management and manipulation.

Q1) Create a Single Linked List and do the following operations:
a) Count the total number of nodes.
b) Sort the linked list.
c) Reverse the linked List.

Code:

```
class Node:
    def __init__(self,data):
        self.info=data
        self.link=None
class Single_Linked_List:
    def __init__(self):
        self.head=None
        self.tmp=None
    def create(self,item):
        new_node=Node(item)
```

```
        if self.head is None:
            self.head=new_node
            self.tmp=self.head
        else:
            self.tmp.link=new_node
            self.tmp=self.tmp.link
    def count_node(self):
        count=0
        tmp=self.head
        while tmp:
            count+=1
            tmp=tmp.link
        return count
    def sort_list(self):
        if self.head is None:
            return
        current=self.head
        while current is not None:
            index=current.link
            while index is not None:
                if current.info>index.info:
                    current.info,index.info=index.info,current.info
                index=index.link
            current=current.link
    def reverse_list(self):
        prev=None
        current=self.head
        while current is not None:
            next_node=current.link
            current.link=prev
            prev=current
            current=next_node
        self.head=prev
    def display(self):
        tmp=self.head
        print("\nElements in the Linked List are: ")
```

```
        while tmp is not None:
            print(tmp.info,end=" ")
            tmp=tmp.link

if __name__=="__main__":
    sl=Single_Linked_List()
    n=int(input("\nHow many number you want to insert in the Linked List:"))
    for i in range(n):
        x=int(input("\nEnter the value of node %d:"%(i+1)))
        sl.create(x)
    sl.display()
    print("\nTotal number of nodes:", sl.count_node())
    print("\nSorting the Linked List: ")
    sl.sort_list()
    sl.display()
    print(" ")
    print("\nReversing the Linked List: ")
    sl.reverse_list()
    sl.display()
```

Output:

```
How many number you want to insert in the Linked List: 3

Enter the value of node 1: 20

Enter the value of node 2: 30

Enter the value of node 3: 10

Elements in the Linked List are:
20 30 10
Total number of nodes: 3

Sorting the Linked List:

Elements in the Linked List are:
10 20 30

Reversing the Linked List:

Elements in the Linked List are:
30 20 10
```

Q2) Create two different single linked lists and merge them together in a third linked list.

Code:

```
class Node:
    def __init__(self, data):
        self.info = data
        self.link = None


class Single_Linked_List:
    def __init__(self):
        self.head = None
        self.tmp = None
    def create(self, item):
        new_node = Node(item)
        if self.head is None:
            self.head = new_node
            self.tmp = self.head
        else:
            self.tmp.link = new_node
            self.tmp = self.tmp.link
    def insert_at_end(self, item):
        new_node = Node(item)
        if self.head is None:
            self.head = new_node
            return
        tmp = self.head
        while tmp.link:
            tmp = tmp.link
        tmp.link = new_node
    def display(self):
        pt = self.head
```

```
        print("\nElements in the Linked List are:")
        while pt is not None:
            print(pt.info, end=" ")
            pt = pt.link
        print()
    def merge_lists(self, list2):
        merged_list = Single_Linked_List()
        tmp1 = self.head
        tmp2 = list2.head
        while tmp1 is not None:
            merged_list.insert_at_end(tmp1.info)
            tmp1 = tmp1.link
        while tmp2 is not None:
            merged_list.insert_at_end(tmp2.info)
            tmp2 = tmp2.link
        return merged_list

if __name__ == "__main__":
    sl1 = Single_Linked_List()
    n1 = int(input("\nHow many numbers you want to insert in the First Linked List: "))
    for i in range(n1):
        x = int(input("\nEnter the value of node %d: " % (i + 1)))
        sl1.create(x)
    sl2 = Single_Linked_List()
    n2 = int(input("\nHow many numbers you want to insert in the Second Linked List: "))
    for i in range(n2):
        x = int(input("\nEnter the value of node %d: " % (i + 1)))
        sl2.create(x)
    print("\nFirst Linked List:")
    sl1.display()
    print("\nSecond Linked List:")
```

```
sl2.display()
merged_list = sl1.merge_lists(sl2)
print("\nMerged Linked List:")
merged_list.display()
```

Output:

```
How many numbers you want to insert in the First Linked List:  1

Enter the value of node 1:  10

How many numbers you want to insert in the Second Linked List:  1

Enter the value of node 1:  100

First Linked List:

Elements in the Linked List are:
10

Second Linked List:

Elements in the Linked List are:
100

Merged Linked List:

Elements in the Linked List are:
10 100
```

CONCLUSION

Single Linked Lists (SLLs) are a fundamental data structure that provides efficient dynamic memory management, making them highly useful in various computational applications. Their ability to allow fast insertions and deletions without requiring contiguous memory allocation makes them ideal for **implementing stacks, queues, graph adjacency lists, and symbol tables**. Additionally, SLLs play a crucial role in **text editors (undo functionality), polynomial arithmetic, and efficient data navigation**.

In this report, we explored the concept of SLLs, their real-world applications, and implemented essential operations such as **counting nodes, sorting, reversing, and merging linked lists**. These operations demonstrate the versatility and practical significance of SLLs in managing and manipulating data structures. Despite some limitations, such as sequential access time and extra memory for pointers, SLLs remain a powerful and widely used structure in programming and software development.

ACKNOWLEDGEMENT

I would like to acknowledge all those without whom this report would not have been successful. Firstly, I would wish to thank my teacher Mr. Saubhik Goswami who guided me throughout the report and gave his immense support. He made us understand how to successfully complete this report and without him, the report would not have been complete.

I would also like to thank my parents who have always been there whenever needed. Once again, thanks to everyone for making this report successful.