Linked List and Space complexity

Bachelor of Technology
Computer Science and Engineering

Submitted By

ARKAPRATIM GHOSH (13000121058)

MARCH 2023

Techno Main
EM-4/1, Sector-V, Salt Lake
Kolkata- 700091
West Bengal
India

TABLE OF CONTENTS

1. Introduction

Linked lists are a popular data structure used in computer programming to store collections of elements, with each element consisting of two parts - the actual data and a reference to the next element in the list. One important consideration when working with linked lists is space complexity, which refers to the amount of memory required to store the data structure.

Space complexity is an important consideration for any program, as it can impact the overall performance and efficiency of the code. In the case of linked lists, the space complexity is generally higher than that of arrays, due to the additional memory required to store the pointers that link the elements together. Specifically, each element in a singly linked list requires two memory allocations - one for the data and one for the pointer to the next element. In a doubly linked list, each element requires three memory allocations - one for the data, one for the pointer to the next element, and one for the pointer to the previous element.

In addition to the memory required for the elements themselves, linked lists can also require additional memory for the list header or tail pointer, which keeps track of the beginning or end of the list. This can add to the overall space complexity of the data structure.

While linked lists generally have a higher space complexity than arrays, they have other advantages that may make them more suitable for certain applications. For example, linked lists can dynamically grow and shrink in size, making them ideal for situations where the number of elements in the collection may change frequently. Additionally, linked lists can be more efficient than arrays for certain operations, such as inserting or deleting elements in the middle of the list.

Overall, when working with linked lists, it is important to consider the space complexity of the data structure in relation to the requirements of the program.

Space complexity is a measure of the amount of memory space used by an algorithm or a program during its execution. It is an important metric in computer science, as it determines how much memory resources are required to run a particular program or algorithm. Space complexity is usually measured in terms of the amount of memory used in the worst case scenario.

2. Body

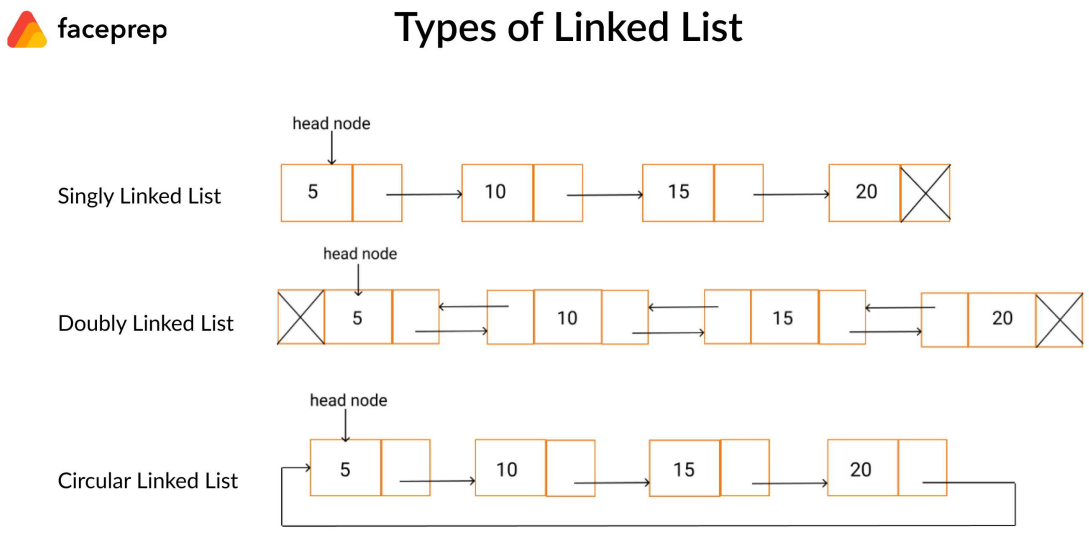
A linked list is a data structure commonly used in computer programming that stores a collection of elements called nodes. Each node contains two parts: one is the actual data stored in the node, and the other is a reference to the next node in the list. This reference is often called a "link" or a "pointer", and it allows for efficient traversal of the list from one node to the next.

Unlike arrays, which have a fixed size and require contiguous memory allocation, linked lists can dynamically grow and shrink in size as nodes are added or removed. This makes linked lists a flexible and efficient data structure for certain applications, particularly those that involve frequent insertions or deletions of elements.

Linked lists come in several different flavors, including singly linked lists, doubly linked lists, and circular linked lists. Singly linked lists have nodes that only point to the next node in the list, while doubly linked lists have nodes that point to both the next node and the previous node in the list. Circular linked lists form a loop, with the last node in the list pointing back to the first node.

Linked lists have a number of advantages over other data structures in certain situations. For example, they can be used to implement stacks and queues, and they can also be used to represent graphs and trees. However, they can also have some disadvantages, such as slower access times than arrays for random access operations and higher memory overhead due to the need for extra pointers.

Overall, linked lists are a powerful and versatile tool in the programmer's toolkit, and understanding how they work is an important part of mastering data structures and algorithms.



Single Linked List

```
#include <stdio.h>
#include <stdlib.h>
// define the structure of a node in the linked list
typedef struct Node {
   int data;
   struct Node *next;
} Node;
// function to create a new node with the given data
Node* createNode(int data) {
   Node* newNode = (Node*) malloc(sizeof(Node));
   newNode->data = data;
   newNode->next = NULL;
   return newNode;
}
// function to add a new node to the end of the linked list
void addNode(Node** head, int data) {
   Node* newNode = createNode(data);
   if (*head == NULL) {
      *head = newNode;
   } else {
      Node* temp = *head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
   }
}
// function to print the linked list
void printList(Node* head) {
   Node* temp = head;
   while (temp != NULL) {
      printf("%d ", temp->data);
      temp = temp->next;
   }
   printf("\n");
}
// function to delete a node from the linked list
void deleteNode(Node** head, int data) {
   Node* temp = *head;
```

```
    Node* prev = NULL;
    while (temp != NULL && temp->data != data) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Node with data %d not found.\n", data);
        return;
    }
    if (prev == NULL) {
        *head = temp->next;
    } else {
        prev->next = temp->next;
    }
    free(temp);
}
int main() {
    Node* head = NULL;
    // add nodes to the list
    addNode(&head, 1);
    addNode(&head, 2);
    addNode(&head, 3);
    addNode(&head, 4);
    // print the list
    printf("Original list: ");
    printList(head);
    // delete a node from the list
    deleteNode(&head, 2);
    // print the list again
    printf("List after deletion: ");
    printList(head);
    return 0;
}
```

In this code, a `Node` structure is defined with two fields: `data` and `next`. The `data` field stores the value of the node, and the `next` field is a pointer to the next node in the list.

The `createNode` function creates a new node with the given data, and returns a pointer to the new node.

The `addNode` function adds a new node to the end of the list. If the list is empty, it sets the head pointer to the new node. Otherwise, it iterates over the list to find the last node, and adds the new node after it.
The `printList` function prints the values of all nodes in the list.
The `deleteNode` function deletes a node with the given data from the list. It iterates over the list to find the node with the given data, and deletes it by adjusting the `next` pointer of the previous node.
Finally, the `main` function creates a list, prints it, deletes a node from it, and prints it again to demonstrate the functionality of the list.

Double Linked List

```
#include <stdio.h>
#include <stdlib.h>
// define the structure of a node in the doubly linked list
typedef struct Node {
   int data;
   struct Node* prev;
   struct Node* next;
} Node;
// function to create a new node with the given data
Node* createNode(int data) {
   Node* newNode = (Node*) malloc(sizeof(Node));
   newNode->data = data;
   newNode->prev = NULL;
   newNode->next = NULL;
   return newNode;
}
// function to add a new node to the beginning of the doubly linked list
void addToBeginning(Node** head, int data) {
   Node* newNode = createNode(data);
   if (*head == NULL) {
      *head = newNode;
   } else {
      newNode->next = *head;
      (*head)->prev = newNode;
      *head = newNode;
   }
}
// function to add a new node to the end of the doubly linked list
```

```
void addToEnd(Node** head, int data) {
   Node* newNode = createNode(data);
   if (*head == NULL) {
      *head = newNode;
   } else {
      Node* temp = *head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
      newNode->prev = temp;
   }
}
// function to print the doubly linked list from beginning to end
void printList(Node* head) {
   Node* temp = head;
   while (temp != NULL) {
      printf("%d ", temp->data);
      temp = temp->next;
   }
   printf("\n");
}
// function to print the doubly linked list from end to beginning
void printListReverse(Node* head) {
   Node* temp = head;
   while (temp->next != NULL) {
      temp = temp->next;
   }
   while (temp != NULL) {
      printf("%d ", temp->data);
      temp = temp->prev;
   }
   printf("\n");
}
// function to delete a node from the doubly linked list
void deleteNode(Node** head, Node* nodeToDelete) {
   if (*head == NULL || nodeToDelete == NULL) {
      return;
   }
   if (*head == nodeToDelete) {
```

```
        *head = nodeToDelete->next;
    }
    if (nodeToDelete->next != NULL) {
        nodeToDelete->next->prev = nodeToDelete->prev;
    }
    if (nodeToDelete->prev != NULL) {
        nodeToDelete->prev->next = nodeToDelete->next;
    }
    free(nodeToDelete);
}
int main() {
    Node* head = NULL;
    // add nodes to the list
    addToEnd(&head, 1);
    addToBeginning(&head, 0);
    // print the list
    printf("List from beginning to end: ");
    printList(head);
    printf("List from end to beginning: ");
    printListReverse(head);
    // delete a node from the list
    deleteNode(&head, head->next);
    // print the list again
    printf("List after deletion: ");
    printList(head);
    return 0;
}
```

In this code, a `Node` structure is defined with three fields: `data`, `prev`, and `next`. The `data` field stores the value of the node, the `prev` field is a pointer to the previous node in the list, and the `next` field is a pointer to the next node in the list.

The space complexity of a linked list is O(n), where n is the number of nodes in the list. Each node in the list requires space to store its data and two pointers, one for the previous node and one for the next node. As the number of nodes in the list increases, the amount of space required to store the list also increases linearly.

In addition to the nodes themselves, there may also be additional space required to store a pointer to the head or tail of the list, as well as any auxiliary data structures used to manipulate the list. Overall, the space complexity of a linked list is generally more efficient than an array for large datasets that may require frequent insertions or deletions, but may

not be as efficient as an array for datasets that require random access or a fixed amount of memory.

3. Conclusion

In conclusion, a linked list is a popular data structure that consists of a series of nodes linked together by pointers. It is commonly used to store and manage large amounts of data in a dynamic and efficient way, particularly when frequent insertions and deletions are required. The space complexity of a linked list is O(n), where n is the number of nodes in the list. As the number of nodes increases, so does the amount of space required to store the list. Despite this, linked lists remain a valuable tool for managing large datasets, and are widely used in various applications, including operating systems, databases, and web development. Understanding the space complexity of a linked list is important for developing efficient algorithms and data structures that can handle large amounts of data while minimizing the use of memory resources.

4. References

1. DATA STRUCTURES THROUGH C IN DEPTH ~ SK and DIPALI SRIVASTAVA
2. ALGORITHMS DESIGN AND ANALYSIS ~ UDIT AGARWAL
3. www.wikipedia.com