Data Structures

Bhashkar Paudyal

June 11, 2025

Abstract

This is a course taught by **Dr. Manoj Shakya**.

Contents

Listings

List of Figures

Chapter 1

Introduction to Algorithms and Algorithm Analysis

Lecture 1: First Lecture

Note (Objectives).
- Agimtotic analysis??

16 Apr. 09:39

1.1 Introduction to Algorithm

Definition 1.1.1 (Algorithm). An algorithm is a step-by-step procedure or set of rules desighned to solve a problem or perform a task.

Algorithms are everywhere

- Sorting a list of names alphabetically.
- Searching for a word in a dictionary.

1.2 Components of Algorithm

- Problem
- Solution
- Steps
- characterstics

1.3 Characterstics of an Algorithm

Finiteness The algorithm should terminate.

Definiteness Every step should be well-defined (non-ambiguos)

Input and Input Size The no. of entities the algorithm handles. It is denoted by "n".

Example. In a searching algorithm,the algorithm needs a sequence of entities and the entity to search.

Output The outcome of running the whole algorithm.

Effectiveness (not to be confused with efficient) Does the algorithm solve the problem it was meant to solve? Section 1.2

Efficiency Sometimes algorithms can be useless even if the algorithm is effective in real world scenarios.

Example. See the travelling salesman problem

Problem 1.3.1 (Travelling Salesman Problem).

1.4 Analysis of an Algorithm

Problem 1.4.1 (Why to Analyze Algorithm?). Some sorting algoritms are:

- Quick
- Merge
- Bubble
- Selection

How to know which sorting algorithm is fastest?
How to know which sorting algorithm takes the least amount of memory?

Answer. Analyze the algorithm!! for **Efficiency**
Basis of measuring efficiency:

1. Time Complexity

2. Space Complexity

⊛

1.5 Time Complexity

We evalute the time complexity by calculating the number of operations in a algorithm denoted by "n".

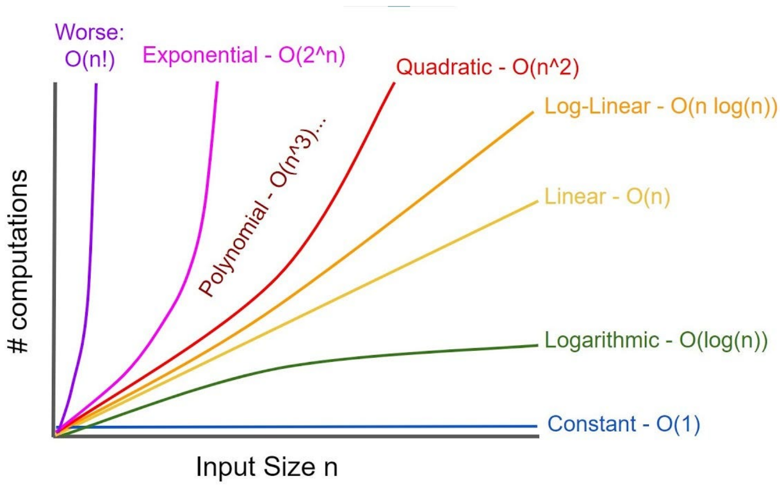

Figure 1.1: Time Complexity of Algorithms when $n \to \infty$

The time complexity of algorithms are measured in O(expression) big O notation, Ω notation and θ notation.

Notations	Case
Ω	Best
θ	Average
O	Worst

Table 1.1: Notations for time complexity

Chapter 2

Linear and Binary Search

2.1 Linear Search

Problem 2.1.1 (Linear Search). For a given list of items, find a target item and locate where it is.

Algorithm 2.1: Linear Search Algorithm

Data: arr: list of items
target: item to locate
Result: index of target if found (first instance) else -1

```
begin
    for i = 0 to len(arr) do
        if arr[i]==target then
            return i
    return -i
```

2.1.1 Time Complexity of Linear Search

Notation	Complexity
Ω	1
O	n

Table 2.1: Time Complexity of Linear Search

2.2 Binary Search

```python
def linear_search(arr, target):
    """
    returns index of target if target is found
    returns -1 if target is not found
    """
    for i in range(len(target)):
        if arr[i]==target:
            return i
    return -1
```

Listing 2.1: Linear Search Implementation

Problem 2.2.1 (Binary Search). For a given list of sorted items, find a target item and locate where it is.

Algorithm 2.2: Binary Search Algorithm

Data: arr: list of items
target: item to locate
Result: index of target if found (first instance) else -1

```
begin
    low, high ← 0, len(arr)-1;
    mid ← (low+high)//2;
    while low ≤ high do
        if arr[mid]<target then
            high = mid;
            mid = (low+high)//2;
        else if arr[mid]>target then
            low = mid;
            mid = (low+high)//2;
        else
            return mid;
    return -1;
```

2.2.1 Time Complexity of Binary Search

Notation	Complexity
Ω	1
O	$\log_2 n$

Table 2.2: Time Complexity of Binary Search

```python
def binary_search(arr, target):
    """Searches for a target in a sorted list

    :arr: sorted list of items
    :target: target item to locate
    :returns: index of target if found else -1

    """
    low, high = 0, len(arr)-1
    mid = (low+high)//2
    while low<=high:
        if arr[mid]< target:
            high = mid
            mid = (low+high)//2

        elif arr[mid]> target:
            low = mid
            mid = (low+high)//2
        else
            return mid
    return -1
```

Listing 2.2: Binary Search Implementation

Chapter 3

Stack

3.1 Properties of stack

Stack is an **Abstract Data Type**.

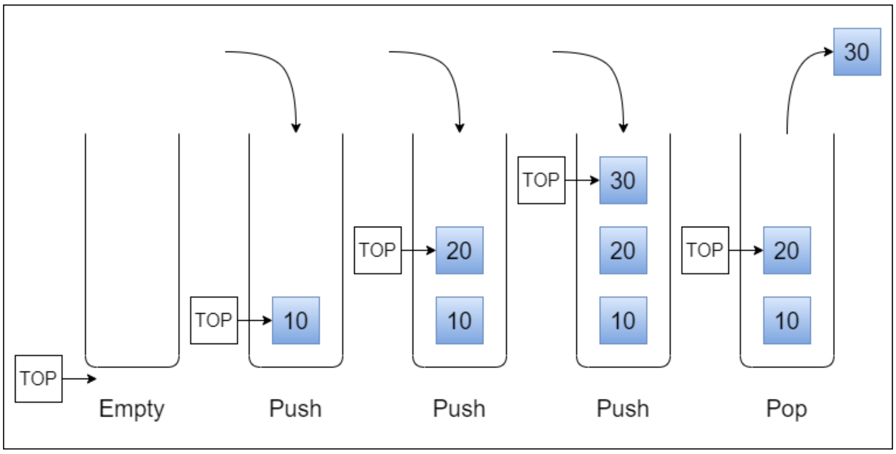

Figure 3.1: Visualization of Stack

3.2 Properties of Stack

1. **Last In First Out** → In the Figure 3.1, recently added item is at the top of the stack and it is removed first.

2. One end of a stack- towards the top, is open.

3.3 Applications of Stack

1. Undo Operation in a word-processor

2. Checking if a number is palindrome
3. Reversing a sequence
4. Factorial
5. Recursion
6. Function Calls in Programming Languages

3.4 Operations of Stack

Push: New Item is added on top of the stack.
`push(item)`
Pop: The recently addeed item is removed from the stack.
`pop()->item`
Peek: See what is recently added.
`peek()->item`
Is Empty: Check if the stack has no element.
`isempty()->bool`
Is Full: (In term of Memory) has the stack used up all the allocoted memory to it?
`isfull()->bool`
Size: # of elements in the stack
`size()->int`

3.5 Stack Implementation

```
class Stack:
    _s = []

    def is_empty(self) -> bool:
        return len(self._s) == 0

    def push(self, item):
        self._s.append(item)

    def pop(self):
        if self.is_empty():
            raise IndexError("Pop from empty stack")   # Underflow
        return self._s.pop()

    def peek(self):
        if self.is_empty():
            raise IndexError("Pop from empty stack")   # Overflow
        return self._s[-1]

    def size(self) -> int:
        return len(self._s)

    def __str__(self) -> str:
        return str(self._s)
```

Listing 3.1: Implementation of a stack

3.6 Application of Stack: Algebriac Expressions

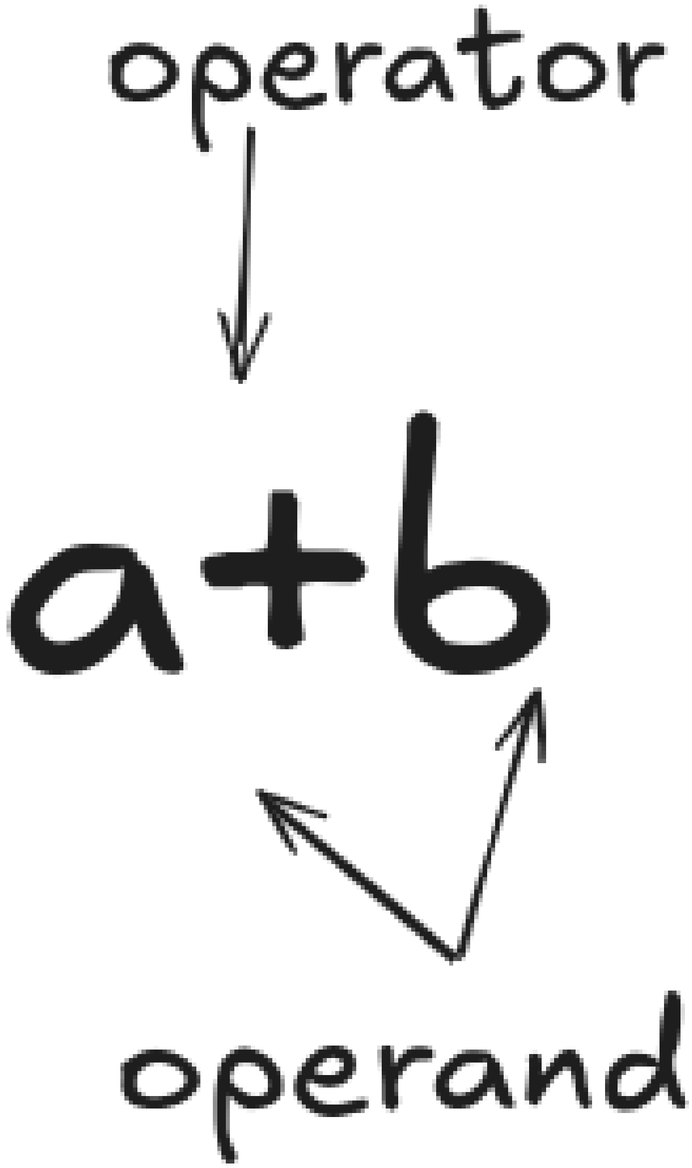

Figure 3.2: Operations: Operands and Operator

3.6.1 Postfix

Operands are written first then operator is written.

Example. 52+

3.6.2 Prefix

Operator are written first then operands are written.

Example. +52

3.6.3 Infix

Example. $5 + 2$

Operator is written in between the operands.

3.6.4 Conversion

Take expression $a + b$. First step is to bracket the expression: $(a + b)$

Infix to Prefix We move the operator to before the bracket: $+ab$

Infix to Postfix We move the operator to after the bracket: $ab+$

Example $(a + b * c)$**.** Apply the BODMAS rule to put brackets:

$$(a + (b * c))$$

To prefix: $+a * bc$
To postfix: $abc * +$

Example $((a + b) * c - d + e)$**.** Apply the BODMAS rule to put brackets:

$$(((a + b) * c) - (d + e))$$

To prefix: $- * +abc + de$
To postfix: $ab + c * de + -$

Lecture 4: Fourth Lecture

26 May. 14:18

Infix to Postfix Conversion Implementation

Consider the relative precedence of operations according to **BODMAS** rule.

1. **Initialize** an empty stack for operators, S, and an empty output string (or list) for the postfix expression, P.
2. **Scan** the infix expression E from left to right, one symbol at a time.
3. **For each symbol t in E:**
   a) If t is an **operand** (letter or digit), append t to P.
   b) If $t =$ `(`, push t onto S.
   c) If $t =$ `)`, pop and append all operators from S to P until a left parenthesis `(` is encountered. Pop and discard the left parenthesis.
   d) If t is an **operator** (`+`, `-`, `*`, `/`, `^`):
      i) While S is not empty, and the operator at the top of S has greater or equal precedence than t (and is not a left parenthesis), pop from S and append to P.
      ii) Push t onto S.
4. **After scanning** the entire infix expression, pop and append any remaining operators from S to P.
5. The output P is the postfix expression.

We define functions to handle precedence prehand and functions for handling expressions.

```python
def operate(a, b, o):
    if o == "+":
        return a + b
    if o == "-":
        return a - b
    if o == "*":
        return a * b
    if o == "/":
        return a / b


def is_operator(char: str):
    return char in "+-/*^"


def precedence(op):
    if op in "+-":
        return 1
    if op in "*/":
        return 2
    if op == "^":
        return 3
    return 0
```

Listing 3.2: Expressions Handling

Algorithm 3.1: Infix to Postfix Conversion

Input: Infix expression E

Output: Postfix expression P

```
Initialize an empty stack S;
Initialize an empty output string P;
for each token t in E (from left to right) do
    if t is an operand then
        Append t to P;
    else
        if t = ( then
            Push t onto S;
        else
            if t = ) then
                while top of S ≠ ( do
                    Pop from S and append to P;
                Pop ( from S;
            else
                while S is not empty and precedence(t) ≤ precedence(top of S) and
                  top of S ≠ ( do
                    Pop from S and append to P;
                Push t onto S;
while S is not empty do
    Pop from S and append to P;
return P;
```

```python
from expressions import is_operator, precedence
from stack import Stack


def infix_to_postfix(exp):
    stack = Stack()
    output = ""

    for char in exp:
        if char.isalnum():
            output += char
        elif char == "(":
            stack.push(char)
        elif char == ")":
            while not stack.is_empty() and stack.peek() != "(":
                output += stack.pop()
            stack.pop()
        elif is_operator(char):
            while (
                not stack.is_empty()
                and stack.peek() != "("
                and precedence(char) <= precedence(stack.peek())
            ):
                output += stack.pop()
            stack.push(char)

    while not stack.is_empty():
        output += stack.pop()

    return output
```

Listing 3.3: Infix to Postfix Conversion

3.6.5 Postfix Evaluation

1. Scan the expression from left to right until we encounter any operator.
2. Perform the operation
3. replace the expression with its computed value.
4. Repeat the steps from **1** to **3** until no more operators exist.

Example. Postfix Expression: $34 * 25 * +$

Algorithm 3.2: Postfix Evaluation

Data: exp: Expression to evaluate

Result: Result of evaluation of the expression

```
begin
    stack ← empty stack;
    for char in exp do
        if char is operator then
            a ← stack.pop();
            b←stack.pop();
            o ← char;
            push a operated(o) on b to stack;
        else
            push char to stack;
    return pop stack
```

Input	Stack	Action
$34*25*+$	empty	Push 3
$4*25*+$	3	Push 4
$*25*+$	3,4	Pop 3, 4. Perform $3*4$. Push 12
$25*+$	12	Push(2)
$5*+$	12, 2	Push 5
$*+$	12, 2, 5	Pop 2, 5. Perform $2*5$, Push 10
$+$	12, 10	Pop 12, 10. Perform $12+10$, Push 22
	22	22 is the evaluation.

Postfix Evaluation Implementation

We have previously made a Stack. We have used the helper functions made previously. Then the function for evaluating postfix expression is defined.

```python
from stack import Stack
from expressions import operate, is_operator


def eval_postfix(exp: str):
    stack = Stack()
    for char in exp:
        if is_operator(char):
            stack.push(operate(stack.pop(), stack.pop(), char))
        else:
            stack.push(int(char))
    return stack.pop()
```

Listing 3.4: Postfix Evaluation

Chapter 4

Queue

4.1 Properties of queue

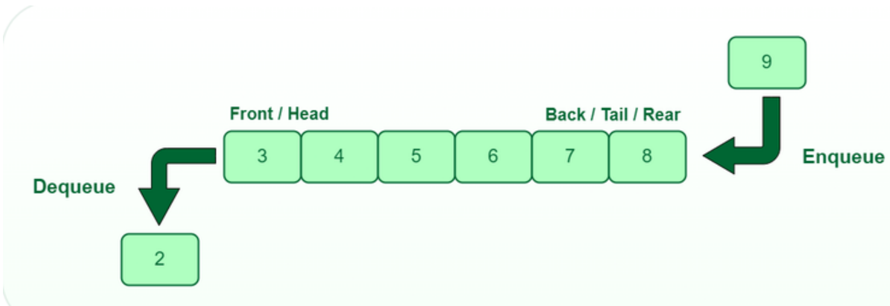

Figure 4.1: Queue Visualization

1. **First In First Out**
2. Both end of a queue is open.

4.2 Application of Queue

- Hospital, Canteen queue
- **CPU scheduling** for processes
- Railway System for simulating train.
- Simulation

4.3 Operations on Queue

- **deque**: Take out element which came first.

- **enqueue**: Insert element at the last of the queue.
- **isEmpty**: Check if the queue is empty.
- **isFull**: is the queue full in terms of memory.
- **peek**: See what element will be dequed.

4.4 Queue Implementation

The following is the implementation of queue data structure in **Python**.

```python
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, element):
        """Add an element to the end of the queue."""
        self.queue.append(element)

    def dequeue(self):
        """Remove and return the front element of the queue.
        Returns a message if the queue is empty."""
        if self.isEmpty():
            raise ValueError("Queue is empty")
        return self.queue.pop(0)

    def peek(self, end=False):
        """Return the front element without removing it.
        Returns a message if the queue is empty."""
        if self.isEmpty():
            raise ValueError("Queue is empty")
        return self[0] if end else self[-1]

    def isEmpty(self):
        """Return True if the queue is empty, else False."""
        return len(self.queue) == 0

    def size(self):
        """Return the number of elements in the queue."""
        return len(self.queue)

    def __getitem__(self, index: int):
        return self.queue[index]
```

Listing 4.1: Queue Implementation

4.5 Circular Queue

A circular queue is an extended version of a standard queue where the last element is connected to the first, forming a circular structure. This design efficiently utilizes the queue's storage space by allowing elements to wrap around from the end to the beginning of the underlying array.

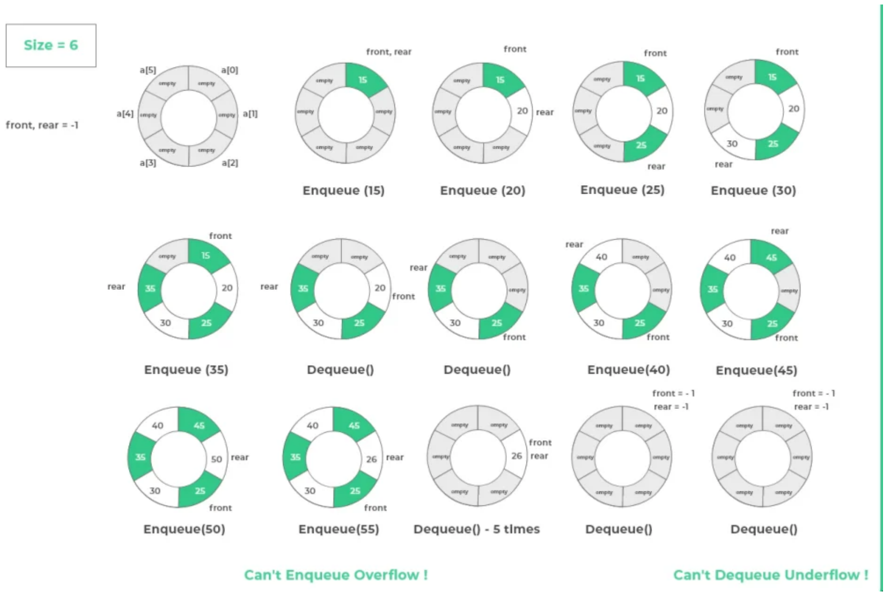

Figure 4.2: Circular Queue

4.5.1 Key Characteristics and Components

A circular queue typically uses:

- An underlying array (or list in Python) of a fixed size (**maxSize** or **k** or **capacity**) to store the queue elements.
- Two pointers:
  - **FRONT** (or **head**): Tracks the first element of the queue.
  - **REAR** (or **tail**): Tracks the last element of the queue.
- **Initialization**: Initially, both **FRONT** and **REAR** are often set to -1 to indicate an empty queue.

Operations

The primary operations of a circular queue include:

Enqueue (Adding an element)

Dequeue (Removing an element)

Front (Peek)

isFull

isEmpty

4.6 Priority Queue

Each element has different priority. The item with highest priority is dequeued first. Other than that, everything is same as queue.

Chapter 5

Deque

It's Just a list which can add elements to front and back.

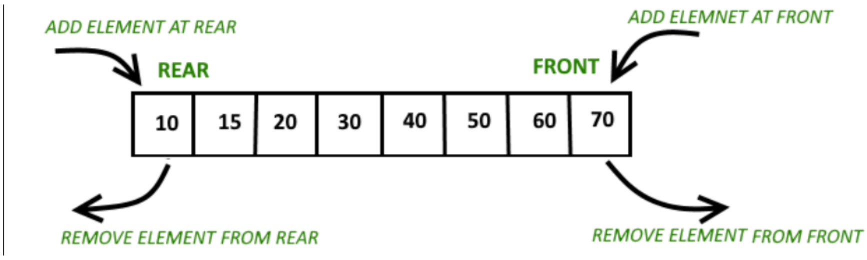

Figure 5.1: Deque Visualization

```
from collections import deque

a = deque()
a.appendleft(5)      # [5]
a.appendleft(3)      # [3,5]
print(a.popleft())   # [5]
a.append(1)          # [5, 1]
a.append(2)          # [5, 1, 2]
print(a.pop())       # [5, 1]
```

Listing 5.1: Deque Implementation in Python

Chapter 6

Linked List

6.1 Introduction

6.2 Types of Linked List

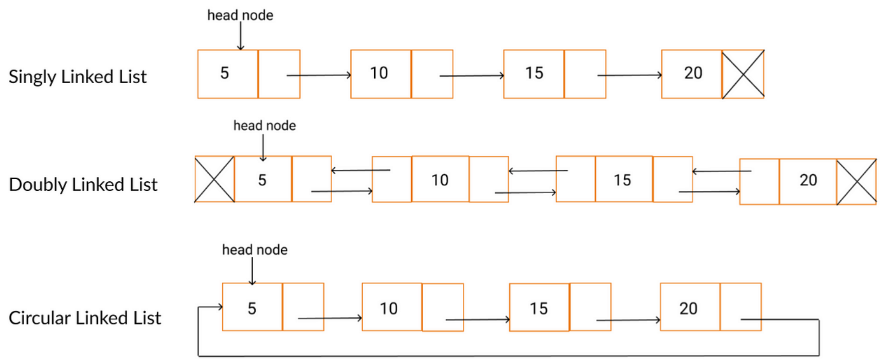

Figure 6.1: Different Types of Linked List

6.3 Singly Linked List

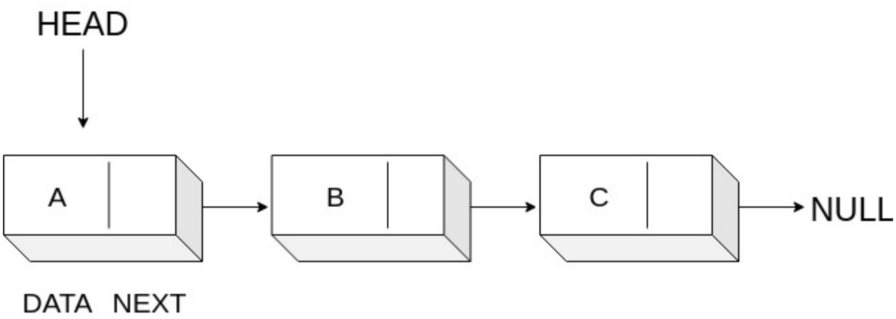

Figure 6.2: Singly Linked List Visualization

6.3.1 Implementation of Singly Linked List

```python
from typing import Any, Iterable, Self


class Node:
    value: Any
    next: Self

    def __init__(self, value, next=None) -> None:
        self.value = value
        self.next = None

    def __str__(self) -> str:
        return str(self.value)

    def __repr__(self) -> str:
        return self.value


class LinkedList:
    current: Node
    head: Node
    length: int = 0
    tail: Node

    def __init__(self, lst: Iterable = []) -> None:
        self.head = None
        self.tail = None
        for e in lst:
            self.append(e)

    def append(self, value):
```

```python
        node = Node(value)
        if not self.head:
            self.head = node
            self.tail = self.head
            self.length += 1
            return
        self.tail.next = node
        self.tail = self.tail.next
        self.length += 1

    def __iter__(self):
        self.current = self.head
        return self

    def __next__(self):
        if not self.current:
            raise StopIteration
        value = self.current.value
        self.current = self.current.next
        return value

    def __str__(self) -> str:
        return " -> ".join([str(e) for e in self])

    def __repr__(self) -> str:
        return self.__str__()

    def __getitem__(self, index: int):
        curr = self.head
        j = 0
        while curr:
            if index == j:
                return curr.value
            curr = curr.next
        raise IndexError("Out of Bounds")

    def __len__(self):
        return self.length


if __name__ == "__main__":
    lst = LinkedList(["hello", "jello", "bello"])
    print(lst)
```

Listing 6.1: Singly Linked List with Tail Implementation

6.4 Doubly Linked List

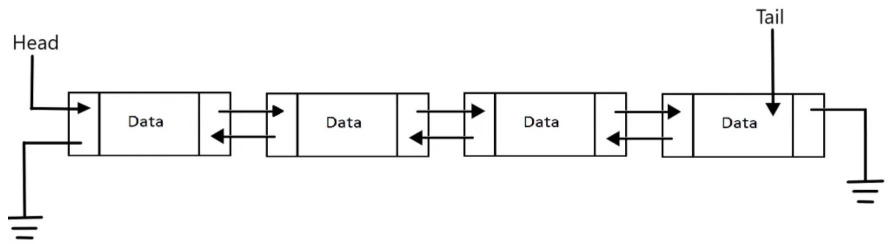

Figure 6.3: Doubly Linked List Visualization

6.4.1 Implementation of Doubly Linked List

```
from typing import Any, Iterable, Self


class Node:
    value: Any
    next: Self
    prev: Self

    def __init__(self, value, next=None, prev=None) -> None:
        self.value = value
        self.next = None
        self.prev = prev

    def __str__(self) -> str:
        return str(self.value)

    def __repr__(self) -> str:
        return self.value



class LinkedList:
    current: Node
    head: Node
    length: int = 0
    tail: Node

    def __init__(self, lst: Iterable = []) -> None:
        self.head = None
```

```
        self.tail = None
        for e in lst:
            self.append(e)

    def append(self, value):
        if not self.head:
            self.head = Node(value)
            self.tail = self.head
            self.length += 1
            return
        self.tail.next = Node(value, prev=self.tail.prev)
        self.tail = self.tail.next
        self.length += 1

    def insert_after(self, index, value):
        if not self.head:
            self.head = Node(value)
            self.tail = self.head
            self.length += 1
            return
        if self.length == index+1:
            self.append(value)
            return
        node = Node(value)
        a = self[index]
        node.next = a.next
        node.prev = a
        self[index].next = node
        self.length += 1

    def delete(self, index):
        a = self[index]
        print(a.prev)
        a.prev.next = a.next
        return a

    def __iter__(self):
        self.current = self.head
        return self

    def __next__(self):
        if not self.current:
            raise StopIteration
        value = self.current.value
        self.current = self.current.next
        return value

    def __str__(self) -> str:
        elements = []
        for e in self:
```

```
            elements.append(str(e))
        return " -> ".join(elements)

    def __repr__(self) -> str:
        return self.__str__()

    def __getitem__(self, index: int):
        curr = self.head
        for i in range(index):
            curr = curr.next
        return curr

    def __len__(self):
        return self.length


lst = LinkedList([1, 2, 3, 4, 5])
print(lst[2].prev)
```

Listing 6.2: Doubly Linked List with Tail Implementation

6.5 Circular Linked List

Figure 6.4: Circular Linked List Visualization

Chapter 7

Tree

Appendix