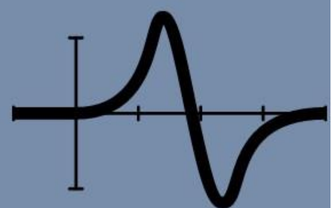
Bpress
Penerbitan Universitas Bakrie

DATA STRUCTURE
Vol. 1

Written by:
YUSUF LESTANTO



DATA STRUCTURE

Vol. 1

Written by

Yusuf Lestanto

DATA STRUCTURE Vol. 1

Copyright ©2023 Yusuf Lestanto
vi + 37 pages, 21 x 29,7 cm

Cover designed by Nuri

First published in Jakarta in August 2023
By **Universitas Bakrie Press**

Jl. H. R. Rasuna Said No.2, RT.2/RW.5, Karet,
Kecamatan Setiabudi, Kuningan,
Daerah Khusus Ibukota Jakarta 12940
https://ubakriepress.bakrie.ac.id/
email: ubakriepress@bakrie.ac.id

All Right Reserved

No part of this book may be used or reproduced in any manner without written permission, except in the case of brief quotations in critical articles or reviews.

PREFACE

In the name of Allah SWT, who has given me guidance and enlightenment resulting in the preparation of this Data Structure teaching material.

Data structure is one of the fastest growing courses in the branch of computer science / Informatics Engineering and is almost always applied to help other fields of science. By studying data structures, students can organize data using data structure concepts and are able to implement them into programs. This data structure teaching material consists of five chapters, namely: array, arraylists, single-, double-linked lists, and stacks. The material in this teaching material is delivered systematically and is accompanied by program examples on each subject and is structured using simple sentences that are definitely representative of the author's experience in teaching data structure courses.

Finally, the author would like to wish you a happy reading, and for all criticism and suggestions, the author would like to thank you.

Jakarta, September 24, 2023

Author

Contents

PREFACE **iii**

1 Introduction **1**
1.1 Learning Objective . . . 1
1.2 Introduction . . . 1
1.3 Definition of Data . . . 1
1.4 Data and Information . . . 2
1.5 Data Structure Purpose . . . 3
1.6 Data Type . . . 4
1.7 Data Set . . . 5
1.8 Algorithms and Data Structures . . . 6
1.9 Data Structure Operations . . . 7
1.10 Complexity Analysis . . . 8

2 Array, ArrayList **10**
2.1 Learning objectives . . . 10
2.2 Array . . . 10
2.3 ArrayList . . . 13
2.4 Exercises . . . 14

3 Single Linkedlist **15**
3.1 Learning objectives . . . 15
3.2 Single linked list . . . 15
3.2.1 Method append . . . 17
3.2.2 Method remove last . . . 18
3.2.3 Method remove first . . . 19
3.2.4 Method prepend . . . 19
3.2.5 Method insert . . . 20
3.2.6 Other Methods . . . 21
3.3 Exercises . . . 21

4 Double Linkedlist **23**
4.1 Learning Objective . . . 23
4.2 Double linked list . . . 23
4.2.1 Method append . . . 26
4.2.2 Other Methods . . . 27
4.3 Exercise . . . 29

5 Stack **31**
5.1 Learning Objective 31
5.2 Stack 31
5.2.1 Method push 34
5.2.2 Method pop 34
5.3 Exercise 35

List of Figures

3.1 Single linked list [3] 15

4.1 Double linked list 23

5.1 Various type of stacks [3] 31

Chapter 1

Introduction

1.1 Learning Objective

The learning objectives of the data structure are as follows:

- Understand the different types of data structures and their applications.
- To be able to implement data structures in Java.
- To be able to write efficient algorithms that use data structures.

1.2 Introduction

Data structures are the fundamental building blocks of computer programming. They provide the means to store and access data in an efficient and organized manner. The basic concepts of data structures include the principles of organization, storage, and retrieval of data. These principles are used to create data structures that are used to store and access data in a computer program. Examples of data structures include arrays, linked lists, trees, and graphs. Each of these data structures has its own set of rules and principles that govern how data is stored and accessed. By understanding the basic concepts of data structures, a programmer can create efficient and effective programs.

1.3 Definition of Data

The concept of data structure refers to the way in which raw facts are organized and stored in a computer program or information system. This includes the arrangement of values, information, or other elements in a specific format. Data structures can be

numbers, text, characters, images, or any other type of value that is organized in a particular way.

Data can be thought of as ”raw materials” that are organized, stored, and processed in a specific format to enable efficient access and manipulation. Data structures are used to arrange and group data in an appropriate manner to facilitate the resolution of various programming tasks and issues.

1.4 Data and Information

Raw facts and figures are referred to as data, while the meaning derived from them is known as information. In the field of information management and data processing, these two terms are often used interchangeably, but they have distinct meanings. Data is composed of unprocessed facts and figures, while information is the interpretation of the data:

- Data

  Data is a collection of unprocessed facts, such as figures, characters, images or other symbols, without specific meanings or contexts. For example, a series of numbers 5, 10, 15, 20 and 25 is data without additional information about their significance.

- Information

  Information occurs when data is processed, organized, and given a context so that it becomes relevant for the recipient. Information is the result of interpreting data and providing a deeper knowledge or understanding. In the previous example, if the number data is arranged in ascending order, the resulting information is ”a number of ascending order numbers from 5 to 25”.

The main difference between data and information is that data are only raw facts without meaning, whereas information gives the data meaning and understanding. The process of data transformation into information includes data analysis, organization and interpretation. Consequently, information is more valuable than data because it provides insights and knowledge that can be used to make decisions and action.

1.5 Data Structure Purpose

The objective of the data structure is to organize and store data efficiently so that data can be accessed, searched, and processed quickly and precisely. The main objectives of data structures are as follows:

1. Efficiency

   Data structures are designed to increase efficiency in data management and data processing operations. Using the right data structure can minimize the time and resources needed for certain operations.

2. Memory Storage

   Data structure helps to manage memory effectively. By properly organizing data, memory waste is avoided, which is essential in limited computer environments.

3. Data search and update

   The data structure facilitates high-speed data search and update operations. For example, using indexes on data structures such as tables or hash tables, the search process can be conducted at the same time without visiting each element individually.

4. Data Structure Storage

   Data Structures help to store data in structured format, so that data can be accessed and manipulated more easily and safely.

5. Algorithm complexity

   Data structure also plays a role in determining algorithm complexity. The selection of the appropriate data structure can have a significant impact on algorithm performance.

6. Organization of data in a specific case

   Each data structure has different uses depending on the case of use. The purpose of a data structure is to provide the appropriate set of operations to handle the specific tasks required by a program.

7. The management of organized data

   data structures help to maintain organized and structured data. With good data management, ensuring consistency and reliability of the data becomes easier.

To achieve these objectives, data structures help to create efficient, reliable and easy-to-maintain programs that deal with various problems and challenges associated with data in computing.

1.6 Data Type

In data structures, the term "data type" refers to the type of value or information that can be stored and processed by a particular data structure. Each data type has certain characteristics and restrictions that describe how data is organized, stored, and accessed. Choosing the appropriate data type is very important to ensure the integrity of the data and the effective operation of the data structure. Some common data types in data structures:

1. Integer

   This data type is used to store integers such as -1, 0, 1, 100, etc. in a data structure.

2. Float and Double

   This type of data is used to store decimal numbers such as 3.14, 2.718, etc. Float has a lower accuracy than Double.

3. Character (Char)

   This data type is used to store a single character such as letters, numbers, or symbols. Each character is represented by an ASCII or Unicode value.

4. Boolean

   This data type has only two values: true or false. Used to store logical states.

5. String

   Although it is not a primitive data type, strings are an important data type in data structures. Used to store text or character sets.

6. Custom Data Types

   Some programming languages allow users to create customised data types, which can consist of multiple primitive data types or other objects.

Understanding the types of data is very important in programming and data structure use. The selection of the appropriate data type to describe the data to be stored

and processed will help to avoid errors and improve the efficiency of program operations.

1.7 Data Set

Data sets in data structures refer to a set or collection of similar or related values or entities arranged in a particular format. Data structures help organize and group these data sets so that they can be accessed, searched and manipulated more efficiently. Data sets may contain the same value or multiple values depending on the data structure used. Examples of data sets in some common data structures:

1. Array

   It represents a collection of data elements of the same type, indexed sequentially. For example, the **[1, 2, 3, 4, 5]** array is a set of five integer data.

2. Linked List

   It is a collection of connected nodes. Each node contains data and a link to the next node. For example, a linked list contains fruit names such as "(apple)–>(orange)–>(mango)".

3. Stack

   A data collection that is accessible according to the **Last In First Out** (LIFO) principle. The data entered in the last place is the first to be extracted. For example, the stack contains the values of the operator in the mathematical expression.

4. Queue

   It is a data collection that is accessed according to the principle of First In First Out (FIFO). The first data entered is the first data to be retrieved. For example, the queue contains the queue of customers in a service system.

5. Tree

   A set of nodes organized in a tree structure. Each node has data and connections to the child node. For example, a family tree with its members as data at each node.

6. Hash Table

   It represents a collection of pairs of "key values". The key serves as an index to

access the associated value. For example, a hash table stores contact information with the name as key and the telephone number as a value.

7. Graph

   It is a collection of vertices and edges that connect these vertices. A graph is used to represent relationships or relationships between entities. For example, a social graph with people as nodes and edges that connect friendships.

Data sets in data structures help organize information in a structured manner, making it easier to process, analyze, and manipulate data. The right data structure is selected depending on the data type that you want to represent and the operations you want to perform.

1.8 Algorithms and Data Structures

In programming and information processing, the connections between algorithms and data structures are very close. Algorithms are a series of steps and instructions designed to solve problems or achieve specific objectives, and data structure is the way data is organized and stored in programs. These two concepts are combined to create effective and effective programmes. Some of the interrelationships between algorithms and data structures are as follows:

1. Selection of the Right Data Structure

   Algorithms and data structures interact to solve problems. Choosing the correct data structure is very important, because it affects the algorithm's performance. An effective data structure that fits the characteristics of the problem will increase the speed and efficiency of the algorithm's operations.

2. Algorithm Complexity

   Data structure plays an important role in determining algorithm complexity, i.e. execution time and memory usage. Using the right data structure, optimize algorithms' complexity and reduce computational burdens.

3. Data Search and Processing

   Algorithms require access and manipulation of data to complete their work. The appropriate data structure facilitates the operation of searching, inserting, deleting, and updating data.

4. Operation Efficiency

   The choice of a suitable data structure will provide more efficient algorithms for performing certain operations, such as sorting, searching, and filtering data.

5. Algorithm Implementation

   The data structure is the basis for the implementation of algorithms. Some algorithms require specific data structures as the basis for implementation, and without proper data structures, algorithms cannot work efficiently.

6. Performance Optimization

   The right combination of algorithms and data structures can optimize the overall performance of the program. Better performance means that the program runs faster and requires fewer resources.

In program development, it is important to understand the interaction between algorithms and data structures in order to choose the appropriate data structures and implement algorithms effectively. Using the right data structure and the right algorithms will achieve optimal results and ensure the best performance of the program.

1.9 Data Structure Operations

Data structure operations include various actions or manipulations that can be performed on data in a data structure. Each type of data structure has unique operations for accessing, storing, and manipulating data. Here are some common operations that are often found in different data structures:

1. Insertion

   Insert a new element into the data structure at a specified location. This operation can be performed at the beginning, end or middle of the data structure depending on the type of structure and needs.

2. Deletion

   Remove certain elements from the data structure. Like additions, deletions can also occur in different positions, including at the beginning, end, or middle of the data structure.

3. Searching

   The search engine searches for elements in the data structure based on certain

criteria. Searches can be performed with indexes (in arrays), keys (in hash tables) or other criteria.

4. Sorting

   The data structure element is sorted according to specific criteria. This sorting operation ensures that the data is arranged in the proper order and facilitates search and processing.

5. Accessing

   Access to specific elements of a data structure. This is usually done using an index, link, or key indicating the position of the element you want to access.

6. Modification

   Change the value or content of elements in the data structure. This may include replacing values, editing data or other appropriate changes.

7. Traversal

   Repeat or iterate all elements of the data structure. This operation allows the processing or display of the entire data.

8. Merge

   Merge two data structures into a new data structure.

9. Reversal

   Reverse the order of elements in the data structure.

10. Union and Intersection

    Special operations for data structures containing sets, such as sets or graph data structures.

Each data structure has special operations according to the characteristics and purposes of its use. Understanding these operations is essential to fully exploit the potential of data structures and effectively manage data.

1.10 Complexity Analysis

Complexity analysis of data structures is a process of understanding and predicting how efficient or slow an algorithm or process is on a data structure as the data volume

increases. This analysis provides a sense of the efficiency of algorithms or data structures and how much resources (e.g. time and memory) the algorithm will use when working with larger data.

Analisis kompleksitas biasanya mencakup dua aspek penting:

- Execution Time (Time Complexity)

  Execution time measures how quickly or slowly an algorithm or operation on a data structure will operate when the processed data is growing. The execution time is measured in the O symbol (large O symbol). The Big O symbol gives an estimate of the speed at which the algorithm's execution time increases as input size increases (e.g., the number of elements in the data structure). Examples of big O symbols include O(1) (constant), O(log n), O(n), $O(n2)$ (quadratic), etc.

- Memory Usage

  Memory Usage measures how much memory an algorithm or operation on a data structure uses to operate with larger data. Like execution time, the complexity of the space is also measured in the Big O symbol. Algorithms with lower spatial complexity are more efficient in terms of memory consumption.

The analysis of data structure complexity is very important because it helps to select the right algorithms and data structure to handle specific tasks or problems. When working with large amounts of data, low-complex algorithms or data structures are more desirable because they are faster and more efficient in resource use. Complexity analysis also helps identify potential bottlenecks and weaknesses in the program to enable appropriate optimization.

Understanding complexity analysis enables developers to design and implement algorithms and data structures with optimal performance, ensuring that programs work efficiently and react appropriately when dealing with large data sets.

Chapter 2

Array, ArrayList

2.1 Learning objectives

The purpose of learning arrays in data structures is:

- Learn how to store data in a structured order using arrays.
- Understand how to perform search and update operations on array data.
- Understand the concept of array indexing and how to use indexes to access data elements.

2.2 Array

A array is a data structure that stores a set of the same type of data. Arrays can be compared to arrays composed of boxes, each box can store data. The array data can be accessed using an index, a number that indicates the location of the array data.

Arrays can have different dimensions. One-dimensional arrays have an index, two-dimensional arrays have two indexes, and so on. One-dimensional arrays are the arrays we see in everyday life, such as numbers and letters. Two-dimensional arrays are like tables with rows and columns.

Arrays are an important data structure in programming. Arrays can be used to store different types of data, such as numbers, characters, strings, and objects. Arrays can also be used for various applications such as data sorting, search and manipulation. Some of the advantages and disadvantages of arrays are as follows.
Advantages of arrays:

- Arrays can store multiple data in a single variable.

- Arrays can be accessed quickly and easily.
- Arrays can be used for a wide range of applications.

Disadvantages of arrays:

- When a array is declared, the size of the array must be specified.
- Arrays cannot be resized when declared.
- Arrays can occupy a lot of memory.

Here are some examples of using arrays:

1. Keep a list of students' names.
2. Saving students' grades.
3. Maintain a list of goods in the store.
4. Maintain a list of company customers

Arrays are very important data structures in programming. Arrays can be used to store different types of data and are used for different applications.

In Java programming, arrays are data structures used to store a set of the same data elements in a sequence order and index them with non-negative numbers (indices). Arrays are one of Java's primitive data types, usually used to manage large amounts of data. Some information on arrays in Java programming:

1. Array Declaration

   In order to declare arrays, we have to specify the data type and the array size of the elements to be declared arrays. For example, if we want to declare a array of integers with five elements, we can use the following syntax.

```
1       int[] myArray = new int[5];
```

2. Element Assignment and Access

   The elements in the array can assign values using the index. The index begins with 0. For example, to fill a table with integer values, you can do as follows:

```
myArray[0] = 10;
myArray[1] = 20;
myArray[2] = 30;
myArray[3] = 40;
myArray[4] = 50;
```

And to access the elements in the array:

```
System.out.println("First element: " + myArray[0]);
System.out.println("Second element: " + myArray[1]);
```

3. Array Length

   The array length (number of elements) can be accessed by using the length property. For example:

```
int length = myArray.length;
System.out.println("Array length: " + length);
```

4. Multidimensional Arrays

   Java also supports multidimensional arrays. Example of two-dimensional array:

```
int[][] twoDArray = new int[3][4];
twoDArray[0][0] = 1;
twoDArray[0][1] = 2;
```

5. Array initialization

   Arrays can be initialized directly on declaration. For example:

```
int[] myArray = {10, 20, 30, 40, 50};
```

6. Size Limitation

   Java arrays have a fixed size specified in the declaration. If you want to increase or decrease the size of the array, you must create an appropriate new array.

Note that array indexes must always be within the valid range (0 to array length minus 1) to avoid the exception **ArrayIndexOutOfBoundsException**.

Arrays are very useful for managing large amounts of sequential data. However, the array is fixed in size and is not flexible. If you need a dynamic data structure, you can use a collection like **ArrayList**.

2.3 ArrayList

ArrayList is a class belonging to the Java Collections Framework. ArrayList is an implementation of the **List** interface that provides flexible ways of storing and managing dynamic data elements. This means that you can add, remove, or access elements to an array list without thinking about their initial size or capacity, because the arraylist will automatically resize. Some of the main features of Java's programming language ArrayList are as follows:

1. Dynamic

   ArrayList is a dynamic data structure that allows you to dynamically add and remove elements without worrying about manual allocations or sizes.

2. Index

   Elements in an array list are indexed from index 0 to index "size()-1". You can access the elements by using indexes.

3. Duplication

   The array list allows you to store multiple elements, which means that you can have multiple elements of the same value.

4. Generics

   By using generic parameters, you can define an array list to store specific data types. For example, to create array lists to store strings, integers, or other custom object data types.

5. Method dan Operasi

   ArrayList provides various methods and operations for manipulating elements, such as add(), remove(), get(), size(), contain(), clear(), and many others.

Example of using ArrayList in Java:

```
1 import java.util.ArrayList;
2
3 public class ArrayListExample {
4     public static void main(String[] args) {
5         // Creates an ArrayList to store String type
6         ArrayList<String> names = new ArrayList<>();
```

```
7
8         // Adding elements to ArrayList
9         names.add("John");
10        names.add("Jane");
11        names.add("Alice");
12
13        // Access elements using the index
14        System.out.println("First name: " + names.get(0));
15
16        // Removes elements by value
17        names.remove("Jane");
18
19        // ArrayList size after deleting
20        System.out.println("ArrayList size: " + names.size());
21
22        // Use a for-each loop to access elements
23        for (String name : names) {
24            System.out.println("Name: " + name);
25        }
26    }
27 }
```

Output:

First name: John

ArrayList size: 2

Name: John

Nama: Alice

2.4 Exercises

1. Write a program to store elements in array and print them.
2. Write a Java program to create a new arraylist, add some elements (strings) and print the collection.

Chapter 3

Single Linkedlist

3.1 Learning objectives

The learning objectives of *single linkedlist* in data structures are:

- Learning a single linked list helps to understand the concept of a linked list-based data structure.
- Understand the main purpose of the linked list is to enable dynamic data management. The size of the link list can change when the program is running.
- Understand the complexity of basic operations, such as adding, deleting and searching for elements in a linked list.

3.2 Single linked list

Single linkedlist is a one-way interconnected linear data structure with pointers to each node and the next node. Each node in the single linked list is called **node**. Each node has two components: data and the next pointer, which shows the next node on the list. The first node of the list is called **Head**, and the last node is called **Tail**. The last Node in the list contains the pointer to **null**. Each node can be accessed linearly by passing the list from head to tail, as seen at figure 3.1.

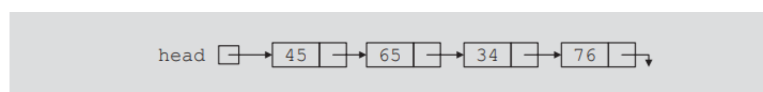


Figure 3.1: Single linked list [3]

In the Figure 3.1, node (45) is the head of the list and node (76) is the tail. Each node is linked in such a way that node (45) is linked to node (65) and next to node (34). The following node (34) points to the node (76). Node (76) refers to *null* because it is the last node on the list.

Here is an example of implementing *single linkedlist* in Java programming:

```
1  // Class for Linkedlist
2  class LinkedList {
3     private Node head;
4     private Node tail;
5     private int length;
6
7     // inner class
8     class Node {
9        int data;
10       Node next;
11
12       public Node(int data) {
13          this.data = data;
14          this.next = null;
15       }
16    }
17
18    public LinkedList(int data) {
19       Node newNode = new Node(data);
20       head = newNode;
21       tail = newNode;
22       length = 1;
23    }
24
25    // Method for adding elements to the end of a link list.
26    public void add(int data) {
27       Node newNode = new Node(data);
28       if (length == 0) {
29          head = newNode;
30          tail = newNode;
31       } else {
32          tail.next = newNode;
```

```
33            tail = newNode;
34        }
35        length++;
36    }
37
38    // Method for printing the contents of the linked list.
39    public void print() {
40        Node current = head;
41        while (current != null) {
42            System.out.print(current.data + " ");
43            current = current.next;
44        }
45    }
46 }
47
48 public class Main {
49    public static void main(String[] args) {
50        LinkedList myList = new LinkedList();
51        myList.add(10);
52        myList.add(20);
53        myList.add(30);
54        myList.print();
55    }
56 }
```

Output: 10 20 30

In the above example, create a single linklist of three nodes and print the data values of each node from beginning to end.

In the above implementation, the Node class is used to store data and node references to the next node. The LinkedList class offers an *add* method to add elements at the end of the LinkedList and an *print* method to print the contents of the LinkedList. The last node in the link list represents the end of the list and represents the value *null*.

3.2.1 Method append

Steps to add a new node at the end of the list in the following:.

- Create new Node.
- It first checks, whether the head is equal to null which means the list is empty.
- If the list is empty, both head and tail will point to the newly added node.
- If the list is not empty, the new node will be added to end of the list such that tail's next will point to the newly added node. This new node will become the new tail of the list.

```
public void append(int value) {
    Node newNode = new Node(value);
    if (length == 0) {
        head = newNode;
        tail = newNode;
    } else {
        tail.next = newNode;
        tail = newNode;
    }
    length++;
}
```

3.2.2 Method remove last

The method works by first checking if the link list is empty. If so, the function is returned. Otherwise, the function creates two points, current and previous. The current pointer indicates the current node on the link list, and the previous pointer indicates the previous node. The function then iterates through the linked list, moving the current pointer to the next node every time. The previous pointer is always a node behind the current pointer. When the current pointer reaches the last node in the linked list, the previous pointer points to the second to the last node. Then the function sets the next pointer of the previous node to *null* and effectively removes the last node from the list.

```
public Node removeLast() {
    if (length == 0) return null;

    Node temp = head;
    Node pre = head;
```

```
6     while(temp.next != null) {
7        pre = temp;
8        temp = temp.next;
9     }
10    tail = pre;
11    tail.next = null;
12    length--;
13    if (length == 0) {
14       head = null;
15       tail = null;
16    }
17    return temp;
18 }
```

3.2.3 Method remove first

The removeFirst() method removes the head of the linked list. It first checks if the head is null. If so, the list is empty, and the method does not perform any function. Otherwise, you set the head variable to the next node on the list. This effectively removes the head node from the list.

```
1        public Node removeFirst() {
2           if (length == 0) return null;
3           Node temp = head;
4           head = head.next;
5           temp.next = null;
6           length--;
7           if (length == 0) {
8              tail = null;
9           }
10          return temp;
11       }
```

3.2.4 Method prepend

To add a node to a Java linked list, follow the following steps:

1. Create a new node with the data you want to add.

2. Set the next property of the new node to the current head of the link list.

3. Set the head of the linked list to the new node.

Here is an example of how to add a node to a Java linked list:

```java
public void prepend(int value) {
    Node newNode = new Node(value);
    if (length == 0) {
        head = newNode;
        tail = newNode;
    } else {
        newNode.next = head;
        head = newNode;
    }
    length++;
}
```

3.2.5 Method insert

The following is a step by step guide to inserting a node into the Java programming language's linked list:

1. Create a new node. The new node will have two fields: data and next. The data field stores the data to be inserted in the linked list, and the next field indicates the next node in the linked list.

2. Initialize the next field of the new node to null. This indicates that the new node is the last node on the linked list.

3. If the link list is empty, the heading of the link list is set to a new node. As a result, the new node is the only node on the linked list.

4. Otherwise, pass through the linked list until you reach the node before inserting a new node.

5. The next field of the previous node is set to the new node. As a result, the new node becomes the next node in the connection list.

6. Set the next field of the new node to the initial node in the position in which you inserted the new node. This ensures that the linked list is always linked together.

Here is an example of how nodes can be added to a Java link list:

```
public boolean insert(int index, int value) {
    if (index < 0 || index > length) return false;
    if (index == 0) {
        prepend(value);
        return true;
    }
    if (index == length) {
        append(value);
        return true;
    }
    Node newNode = new Node(value);
    Node temp = get(index - 1);
    newNode.next = temp.next;
    temp.next = newNode;
    length++;
    return true;
}
```

3.2.6 Other Methods

Other methods to access single linked list are listed below:

- Getter and Setter

  This method gets ot sets value at a given index.

- Remove

  This method removes node at a given index.

- Reverse

  This method reverses order of list.

3.3 Exercises

```
import java.util.*;
public class LinkedListDemo{
    public static void main(String args[]){
        // create a linked list
```

```
5         LinkedList ll = new LinkedList();
6         // add elements to the linked list
7         ll.append("F");
8         ll.append("B");
9         ll.append("D");
10        ll.append("E");
11        ll.append("C");
12        ll.append("Z");
13        ll.prepend("A");
14        ll.append(1,"A2");
15        System.out.println("Original contents of ll: "+ ll);
16
17        // remove elements from the linked list
18        ll.remove("F");
19        ll.remove(2);
20        System.out.println("Contents of ll after deletion: " + ll);
21
22        // remove first and last elements
23        ll.removeFirst();
24        ll.removeLast()
25        System.out.println("ll after deleting first and last: "
26        + ll);
27
28        // get and set a value
29        Object val = ll.get(2);
30        ll.set(2,(String) val +" Changed");
31        System.out.println("ll after change: "+ ll);
32    }
33 }
```

Tasks:

1. Analysis of above source code.

2. Write the outputs of statement at line 31.

Chapter 4

Double Linkedlist

4.1 Learning Objective

The learning objectives of the double linking list are as follows:

- Understand the structure of a double linked list.
- Be able to insert and delete nodes in a double linked list.
- Be able to traverse a double linked list in both forward and backward directions.
- Be able to find the first and last nodes in a double linked list.
- Be able to reverse the order of a double linked list.
- Be able to merge two double linked lists.

4.2 Double linked list

A double-link list is a link list where each node has the next pointer and the back pointer. In other words, each node contains the next node's address (with the exception of the last node), and each node contains the previous node's address (with the exception of the first node). (See Figure 4.1).

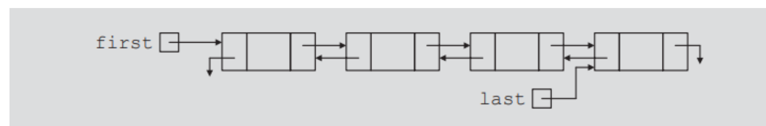


Figure 4.1: Double linked list

A double linked list can be traversed in any direction. In other words, a list that starts with the first node can be traversed or a list that starts with the last node can be traversed if the pointer is given to the last node.
Here are some of the advantages of using double linked lists:

- They can be traversed in both forward and backward directions.
- Insertion and deletion of nodes is relatively efficient, even in the middle of the list.
- They can be easily reversed.
- They can be merged with other double linked lists.

Here are some of the disadvantages of using double linked lists:

- They take up more space than singly linked lists, because each node has two pointers.
- They are not as efficient for some operations, such as finding the middle node in the list.

Here is an example of implementing *double linkedlist* in Java programming:

```
// Class for DoubleLinkedList
class DoubleLinkedList {
    private Node head;
    private Node tail;
    private int length;

    // inner class
    class Node {
        int value;
        Node next;
        Node prev;

        public Node(int data) {
            this.value = data;
        }
    }

```

```java
    public DoubleLinkedList(int data) {
        Node newNode = new Node(data);
        head = newNode;
        tail = newNode;
        length = 1;
    }

    // Method for adding elements to the end of a link list.
    public void append(int data) {
        Node newNode = new Node(data);
        if (length == 0) {
            head = newNode;
            tail = newNode;
        } else {
            tail.next = newNode;
            newNode.prev = tail;
            tail = newNode;
        }
        length++;
    }

    // Method for printing the contents of the linked list.
    public void print() {
        Node temp = head;
        while (temp != null) {
            System.out.println(temp.value);
            temp = temp.next;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        DoubleLinkedList dll = new DoubleLinkedList(1);

        dll.getHead();
        dll.getTail();
```

```
55            dll.getLength();
56
57            System.out.println("\nDoubly Linked List:");
58            dll.printList();
59        }
60    }
```

4.2.1 Method append

Each node in the linked list has a field that contains data and a pointer to the next node and to the previous node. For the formation of nodes new, initially next and prev pointers will point to NULL values. Next, the pointer prev will point to the previous node, and the pointer next will point to the next node in the list

- Create a new node.
- It first checks, whether the head is equal to null which means the list is empty.
- If the list is empty, both head and tail will point to the newly added node.
- If the list is not empty, the new node will be added to end of the list such that tail's next will point to the newly added node. This new node will become the new tail of the list.

```
1    public void append(int value) {
2        Node newNode = new Node(value);
3        if (length == 0) {
4            head = newNode;
5            tail = newNode;
6        } else {
7            tail.next = newNode;
8            newNode.prev = tail;
9            tail = newNode;
10       }
11       length++;
12   }
```

4.2.2 Other Methods

Other methods to access double linked list are listed below:

- Remove Last

  This method removes the last node from list

```
public Node removeLast() {
    if(length == 0) return null;
    Node temp = tail;
    if (length == 1) {
        head = null;
        tail = null;
    } else {
        tail = tail.prev;
        tail.next = null;
        temp.prev = null;
    }
    length--;
    return temp;
}
```

- Prepend

  This method adds new node as new head of the list

```
public void prepend(int value) {
    Node newNode = new Node(value);
    if(length == 0) {
        head = newNode;
        tail = newNode;
    } else {
        newNode.next = head;
        head.prev = newNode;
        head = newNode;
    }
    length++;
}
```

- Remove First

  This method removes the head and change the new to the next node.

```
public Node removeFirst() {
    if(length == 0) return null;
    Node temp = head;
    if(length == 1) {
        head = null;
        tail = null;
    } else {
        head = head.next;
        head.prev = null;
        temp.next = null;
    }
    length--;
    return temp;
}
```

- Insert

  This method inserts new node at a given index.

```
public boolean insert(int index, int value) {
    if(index < 0 || index > length) return false;
    if(index == 0) {
        prepend(value);
        return true;
    }
    if(index == length) {
        append(value);
        return true;
    }
    Node newNode = new Node(value);
    Node before = get(index - 1);
    Node after = before.next;
    newNode.prev = before;
    newNode.next = after;
    before.next = newNode;
    after.prev = newNode;
    length++;
```

```
            return true;
        }
```

- Remove

  This method removes node at a given index.

```
        public Node remove(int index) {
            if(index < 0 || index >= length) return null;
            if(index == 0) return removeFirst();
            if(index == length - 1) return removeLast();

            Node temp = get(index);

            temp.next.prev = temp.prev;
            temp.prev.next = temp.next;
            temp.next = null;
            temp.prev = null;

            length--;
            return temp;
        }
```

4.3 Exercise

Based on the source code below, answer the following questions:

```
public class DoubleLinkedListDemo {
    public static void main(String[] args) {
        DoubleLinkedList dll = new DoubleLinkedList();
        dll.printList();
        dll.append(2);

        System.out.println("Add more Nodes");
        dll.prepend(3);
        dll.append(4);
        dll.insert(2, 5);
        dll.insert(3, 6);
        dll.prepend(7);
```

```
13         dll.remove(4);
14     }
15 }
```

1. Analysis of above source code and write down the data in the list.

2. Start from line 14, two more nodes are prepended. Write down the final data in the list

Chapter 5

Stack

5.1 Learning Objective

The learning objectives of the stack are as follows:

- Understand the basic concept of a stack, which is a LIFO (last in first out) data structure.
- Be able to implement a stack using arrays or linked lists.
- Be able to perform the basic stack operations of push, pop, peek, and isEmpty.

5.2 Stack

A stack is a list of homogeneous elements in which elements are added and deleted at one end, called the top of the stack. Data structure in which the elements are added and removed only from a end; a last-in-first-out (LIFO) data structure.

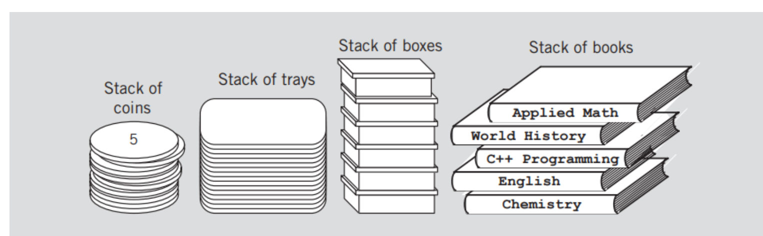


Figure 5.1: Various type of stacks [3]

Figure 5.1 shows that the last element inserted into the stack is the first element to be removed. Stacks can be implemented using arrays or linked lists. The basic operations on a stack are as follows:

- Push: This operation inserts an element into the top of the stack.
- Pop: This operation removes the element from the top of the stack and returns it.
- Top: This operation returns the element at the top of the stack without removing it.

Stacks are used in a variety of applications, including:

- Hanoi Tower Puzzle: The puzzle of the Hanoi Tower can be solved by stacking. It moves a disk set from one peg to another in accordance with a specific rule set.
- Quick Sort and Merge Sort: Quick Sort and Merge Sort are two popular sorting algorithms that use stacks to divide data into smaller sub-problems. This allows algorithms to efficiently sort data.

Here is an example of how a stack can be used to solve the Tower of Hanoi puzzle. The puzzle involves moving a set of disks from one peg to another, following a specific set of rules. The rules are as follows:

1. Only one disk can be moved at a time.
2. A larger disk cannot be placed on top of a smaller disk.
3. The disks must be moved from the first peg to the third peg, with the second peg serving as a temporary holding place.

Here is an example of implementing *Stack* in Java programming:

```
1  // Class for Stack
2  class Stack{
3     private Node top;
4     private int height;
5  
6     // inner class
7     class Node {
8        int value;
9        Node next;
10 
11       public Node(int data) {
```

```
12              this.value = data;
13          }
14      }
15  
16      public Stack(int data) {
17          Node newNode = new Node(value);
18          top = newNode;
19          height = 1;
20      }
21  
22      public void getTop() {
23          if (top == null) {
24              System.out.println("Top: null");
25          } else {
26              System.out.println("Top: " + top.value);
27          }
28      }
29  
30      public void getHeight() {
31          System.out.println("Height: " + height);
32      }
33  
34      // Method for printing the contents of stack.
35      public void print() {
36          Node temp = top;
37          while (temp != null) {
38              System.out.println(temp.value);
39              temp = temp.next;
40          }
41      }
42  }
43  
44  public class Main {
45      public static void main(String[] args) {
46          Stack stack = new Stack(4);
47  
48          stack.getTop();
```

```
49            stack.getHeight();
50
51            System.out.println("Stack:");
52            stack.print();
53        }
54    }
```

Method getTop () gets the top element of the stack is the last element added to the stack, see line 22-28. Because the elements are added and removed from one end (that is, the top), it follows that the item that is added last will be removed first. For this reason, a stack is also called a **Last In First Out** (LIFO) data. Method getTop(), returns the top element of the stack. Prior to this operation, the stack must exist and must not be empty. structure.

5.2.1 Method push

The push method of the stack inserts elements at the top of the stack. The push method usually takes one argument, which is the element to be pushed into the stack. Then the push method increases the stack pointer and stores the elements at the new top of the stack.

```
1  public void push(int data) {
2      Node newNode = new Node(data);
3      if(height == 0) {
4          top = newNode;
5      } else {
6          newNode.next = top;
7          top = newNode;
8      }
9      height++;
10 }
```

5.2.2 Method pop

The pop method in a stack removes element from the top of the stack and returns them. A stack is a linear data structure based on the LIFO principle. This means that the last element inserted in the stack is the first element to be removed. The pop

method first checks if the stack is empty. If the stack is empty, the method returns *null*. Otherwise, the method removes the last element and returns it.

```
public Node pop() {
    if(height == 0) return null;

    Node temp = top;
    top = top.next;
    temp.next = null;

    height--;
    return temp;
}
```

5.3 Exercise

```
public class StackDemo {
    public static void main(String[] args) {
        Stack stack = new Stack(6);
        stack.print();
        stack.push(2);

        System.out.println("Add more data");
        stack.push(3);
        stack.push(4);
        stack.pop();
        stack.push(7);
        stack.push(8);
        stack.pop();
    }
}
```

Tasks:

1. Analysis of above source code and write down the data in the stack.

2. Add new algorithm to empty the stack.

3. Start from line 12, two more nodes are pushed. Write down the final data in the list

Bibliography

[1] Koffman, E. & Wolfgang, P. Data structures: abstraction and design using Java. (John Wiley & Sons, 2021)

[2] Goodrich, M., Tamassia, R. & Goldwasser, M. Data structures and algorithms in Java. (John Wiley & Sons, 2014)

[3] Malik, D. Data structures using C++. (Cengage Learning, 2009)

[4] Main, M. Data structures and other objects using Java. (Addison-Wesley Longman Publishing Co., Inc., 2002)

Bpress
Penerbitan Universitas Bakrie