MIT World Peace University

Advanced Data Structures
Second Year B. Tech, Semester 4

Addition of Polynomials using Circular Linked Lists

Assignment No. 1

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

January 24, 2023

Contents

1 Objectives **1**

2 Problem Statement **1**

3 Theory **1**
- 3.1 Brief about Data structure . . . 1
- 3.2 Circular Linked List . . . 1
  - 3.2.1 Representation using Structures . . . 1
  - 3.2.2 Advantages of Circular Linked List . . . 2
- 3.3 Difference between Singly Linked List, Circular Linked List, and Double Linked List . 2
  - 3.3.1 Singly Linked List . . . 2
  - 3.3.2 Circular Linked List . . . 2
  - 3.3.3 Doubly Linked List . . . 3
- 3.4 Various operations on CLL . . . 3

4 Platform **3**

5 Input **4**

6 Output **4**

7 Test Conditions **4**

8 Pseudo Code **4**
- 8.1 Pseudo Code for Creating a Circular Linked List . . . 4
- 8.2 Pseudo Code for Displaying a Circular Linked List . . . 5
- 8.3 Pseudo Code for Addition of 2 Polynomials using Circular Linked List . . . 5
- 8.4 Pseudo Code for Evaluation of a Polynomial using a Circular Linked List . . . 7

9 Code **7**
- 9.1 Program . . . 7
- 9.2 Input and Output . . . 11

10 Conclusion **13**

11 FAQ **14**

1 Objectives

1. To study data structure: Circular Linked List
2. To Study different operations that could be performed on CLL
3. To Study Applications of Circular Linked list

2 Problem Statement

Implement polynomial operations using Circular Linked List: Create, Display, Addition and Evaluation

3 Theory

3.1 Brief about Data structure

Data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

Depending on our requirement and project, it is important to choose the right data structure for our project. For example, if we want to store data sequentially in the memory, then you can go for the Array data structure, while if we have something more complex, we may want to opt for graph, or trees.

They are broadly classified into two types:

1. Linear Data Structures: Like Arrays, Linked Lists, Stacks, Queues, etc.
2. Non-Linear Data Structures: Like Trees, Graphs, etc.

3.2 Circular Linked List

Circular Linked List is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.

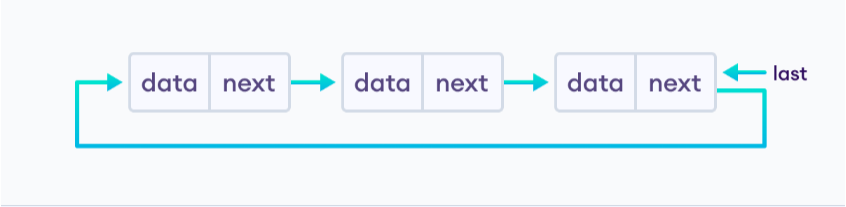


Figure 1: Circular Linked List

3.2.1 Representation using Structures

```
struct Node {
  int data;
  struct Node * next;
};
```

3.2.2 Advantages of Circular Linked List

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Useful for implementation of queue. Unlike linear data structures, we don't need to move all elements after a dequeue operation.
3. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running in a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the rest of the applications execute. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
4. It is used in multiplayer games to give a chance to each player to play the game.
5. Multiple running applications can be placed in a circular linked list on an operating system. The os keeps on iterating over these applications.

3.3 Difference between Singly Linked List, Circular Linked List, and Double Linked List

3.3.1 Singly Linked List

It is the simplest type of linked list. Each node contains two parts: data and pointer to the next node. The last node points to NULL.

1. Each node has two parts: data and pointer to the next node.
2. Each node has only one pointer to the next node.

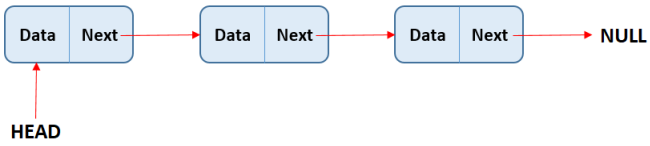


Figure 2:

3.3.2 Circular Linked List

The Major difference between circular linked list and singly linked list is that the last node of the circular linked list points to the first node of the list.

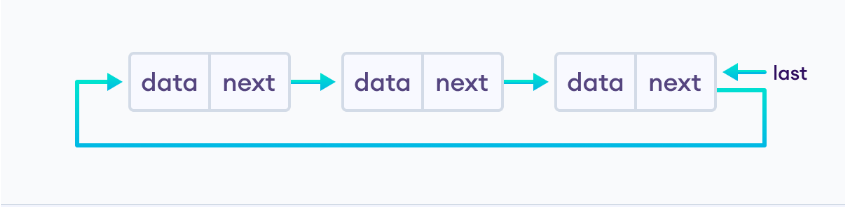


Figure 3:

3.3.3 Doubly Linked List

The major difference between doubly linked list and singly linked list is that the doubly linked list has two pointers: one pointer to the next node and another pointer to the previous node.

Figure 4:

3.4 Various operations on CLL

All the operations that can be performed on any data structure can be performed on a circular linked list. Some of the operations are:

1. Insertion
2. Deletion
3. Traversal
4. Search
5. Sorting
6. Merging
7. Reversing

The ADT is a set of operations that can be performed on the data structure. It is given further in the FAQ section. The Pseudo Code for these operations is given in the following sections.

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers : g++ and gcc on linux for C++

5 Input

1. The Choice for what to do
2. The Coefficients and Exponents of the Polynomials
3. Testcase: Addition of the Following 2 Polynomials

$$3x^2 + 5x + 9 \tag{1}$$
$$4x^6 + 8x \tag{2}$$

6 Output

1. The Resultant Polynomial Represented as a Circular Linked List
2. The Sum of the Given 2 Polynomials.
3. The Menu for what to do.
4. Testcase Output:

$$4x^6 + 8x + 3x^2 + 5x + 9 \tag{3}$$

7 Test Conditions

1. Input at least five nodes.
2. Addition of two polynomials with at least 5 terms.
3. Evaluate polynomial with floating values.

8 Pseudo Code

8.1 Pseudo Code for Creating a Circular Linked List

```
// Create a circular linked list
void create()
{
  struct node *temp, *ptr;
  int i, n;
  printf("Enter the number of nodes: ");
  scanf("%d", &n);
  for (i = 0; i < n; i++)
  {
    temp = (struct node *)malloc(sizeof(struct node));
    printf("Enter the data: ");
    scanf("%d", &temp->data);
    temp->next = NULL;
    if (head == NULL)
    {
      head = temp;
      ptr = temp;
    }
```

```
    else
    {
      ptr->next = temp;
      ptr = temp;
    }
  }
  ptr->next = head;
}
```

8.2 Pseudo Code for Displaying a Circular Linked List

```
// Display a circular linked list

void display(struct Node *head){
  struct Node *ptr = head;
  do{
    printf("%d ", ptr->data);
    ptr = ptr->next;
  }while(ptr != head);
  printf("\n");
}
```

8.3 Pseudo Code for Addition of 2 Polynomials using Circular Linked List

```
struct Node *add_polynomials(struct Node *head1, struct Node *head2)
{
  // Pointers for the result polynomial.
  struct Node *result_head = ASSIGN MEMORY
  result_head->next = result_head;
  struct Node *result_temp = result_head;
  struct Node *result_current;

  // p1 and p2 are the pointers to the first node of the two polynomials.
  struct Node *p1 = head1->next;
  struct Node *p2 = head2->next;

  // In case one of the polynomial exhausts before the other one.
  while (p1 != head1 && p2 != head2)
  {
    // if the exponents are equal, add the coefficients and add the node to the
    result polynomial.
    if (p1->exp == p2->exp)
    {
      // Copy the data of thesum of the nodes to the result polynomial.
      result_current = ASSIGN MEMORY
      result_current->coeff = p1->coeff + p2->coeff;
      result_current->exp = p1->exp;
      result_current->next = result_head;
      result_temp->next = result_current;

      // Increment the result polynomial pointer, and other polynomial pointers.
      result_temp = result_temp->next;
      p1 = p1->next;
      p2 = p2->next;
    }

```

```
32      // If the exponent of the first polynomial is greater than the second one, add
        the node to the result polynomial.
33      else if (p1->exp > p2->exp)
34      {
35        result_current = ASSIGN MEMORY
36        result_current->coeff = p1->coeff;
37        result_current->exp = p1->exp;
38        result_current->next = result_head;
39        result_temp->next = result_current;
40
41        // increment the result polynomial pointer, and p1
42        result_temp = result_temp->next;
43        p1 = p1->next;
44      }
45
46      // If the exponent of the second polynomial is greater than the first one, add
        the node to the result polynomial.
47      else if (p2->exp > p1->exp)
48      {
49        result_current = ASSIGN MEMORY
50        result_current->coeff = p2->coeff;
51        result_current->exp = p2->exp;
52        result_current->next = result_head;
53        result_temp->next = result_current;
54
55        // increment the result polynomial pointer, and p2
56        result_temp = result_temp->next;
57        p2 = p2->next;
58      }
59    }
60
61    // Case when p2 exhausts before p1.
62    if (p1 == head1 && p2 != head2)
63    {
64      result_temp->next = p2;
65
66      // This loop is to make the last node of the result polynomial point to the
       head of the result polynomial.
67      while (result_temp->next != head2)
68      {
69        result_temp = result_temp->next;
70      }
71      result_temp->next = result_head;
72    }
73
74    // Case when p1 exhausts before p2.
75    else if (p1 != head1 && p2 == head2)
76    {
77      result_temp->next = p1;
78      while (result_temp->next != head1)
79      {
80        result_temp = result_temp->next;
81      }
82      result_temp->next = result_head;
83    }
84
85    // Case when both p1 and p2 exhaust.
86    else if (p1 != head1 && p2 != head2)
87    {
```

```
    result_temp->next = p1;
    while (result_temp != head1)
    {
      result_temp = result_temp->next;
    }
    result_temp->next = result_head;

    result_temp->next = p2;
    while (result_temp != head2)
    {
      result_temp = result_temp->next;
    }
    result_temp->next = result_head;
  }

  return result_head;
}
```

8.4 Pseudo Code for Evaluation of a Polynomial using a Circular Linked List

```
// evaluate a polynomial using circular linked list
void evaluate(struct Node *head, int x)
{
  struct Node *temp = head->next;
  int result = 0;

  // Loop to evaluate the polynomial.
  while (temp != head)
  {
    result += temp->coeff * pow(x, temp->exp);
    temp = temp->next;
  }

  printf("The value of the polynomial is %d", result);
}
```

9 Code

9.1 Program

```
// Circular linked List
// Implementing the following polynomial operations using circular linked list.
    Create Dislay and Add.

// Krishnaraj Thadesar
// Jan 22nd 2023
// Assignment 1
// Advanced Data Structures
// Semester 4

#include <stdio.h>
#include <stdlib.h>

// The default way to represent circular linked lists using structures.
struct Node
```

```
{
    int coeff;
    int exp;
    struct Node *next;
};

// Function to accept the polynomial from the user.
void accept_polynomial(struct Node *head)
{
    struct Node *temp = head;
    int choice = 0;
    do
    {
        struct Node *curr = (struct Node *)malloc(sizeof(struct Node));
        printf("\nEnter the Coefficient: ");
        scanf("%d", &curr->coeff);
        printf("\nEnter the Exponent: ");
        scanf("%d", &curr->exp);

        // Main Logic of inserting node at the end and making it point to the head
    .
        curr->next = head;
        temp->next = curr;
        temp = temp->next;

        printf("Do you want to enter more terms? (0 for no, 1 for yes) \n");
        scanf("%d", &choice);
    } while (choice != 0);
}

// Function to display the polynomial.
int display_polynomial(struct Node *head)
{
    // Edge Case of empty list.
    if (head->next == head)
    {
        printf("\nNo terms in the polynomial");
        return -1;
    }

    struct Node *curr = (struct Node *)malloc(sizeof(struct Node));
    curr = head->next;

    while (curr != head)
    {
        printf("%dx^%d", curr->coeff, curr->exp);
        curr = curr->next;
        if (curr != head)
        {
            printf(" + ");
        }
    }
    printf("\n");
}

// Function to add two polynomials, Returns the head of the added polynomial.
// Takes as input the heads of the other 2 polynomials that you want to add.
struct Node *add_polynomials(struct Node *head1, struct Node *head2)
{
```

```
    // Pointers for the result polynomial.
    struct Node *result_head = (struct Node *)malloc(sizeof(struct Node));
    result_head->next = result_head;
    struct Node *result_temp = result_head;
    struct Node *result_current;

    // p1 and p2 are the pointers to the first node of the two polynomials.
    struct Node *p1 = head1->next;
    struct Node *p2 = head2->next;

    // In case one of the polynomial exhausts before the other one.
    while (p1 != head1 && p2 != head2)
    {
        // if the exponents are equal, add the coefficients and add the node to
    the result polynomial.
        if (p1->exp == p2->exp)
        {
            // Copy the data of thesum of the nodes to the result polynomial.
            result_current = (struct Node *)malloc(sizeof(struct Node));
            result_current->coeff = p1->coeff + p2->coeff;
            result_current->exp = p1->exp;
            result_current->next = result_head;
            result_temp->next = result_current;

            // Increment the result polynomial pointer, and other polynomial
    pointers.
            result_temp = result_temp->next;
            p1 = p1->next;
            p2 = p2->next;
        }

        // If the exponent of the first polynomial is greater than the second one,
     add the node to the result polynomial.
        else if (p1->exp > p2->exp)
        {
            result_current = (struct Node *)malloc(sizeof(struct Node));
            result_current->coeff = p1->coeff;
            result_current->exp = p1->exp;
            result_current->next = result_head;
            result_temp->next = result_current;

            // increment the result polynomial pointer, and p1
            result_temp = result_temp->next;
            p1 = p1->next;
        }

        // If the exponent of the second polynomial is greater than the first one,
     add the node to the result polynomial.
        else if (p2->exp > p1->exp)
        {
            result_current = (struct Node *)malloc(sizeof(struct Node));
            result_current->coeff = p2->coeff;
            result_current->exp = p2->exp;
            result_current->next = result_head;
            result_temp->next = result_current;

            // increment the result polynomial pointer, and p2
            result_temp = result_temp->next;
            p2 = p2->next;
```

```
128         }
129     }
130 
131     // Case when p2 exhausts before p1.
132     if (p1 == head1 && p2 != head2)
133     {
134         result_temp->next = p2;
135 
136         // This loop is to make the last node of the result polynomial point to
    the head of the result polynomial.
137         while (result_temp->next != head2)
138         {
139             result_temp = result_temp->next;
140         }
141         result_temp->next = result_head;
142     }
143 
144     // Case when p1 exhausts before p2.
145     else if (p1 != head1 && p2 == head2)
146     {
147         result_temp->next = p1;
148         while (result_temp->next != head1)
149         {
150             result_temp = result_temp->next;
151         }
152         result_temp->next = result_head;
153     }
154 
155     // Case when both p1 and p2 exhaust.
156     else if (p1 != head1 && p2 != head2)
157     {
158         result_temp->next = p1;
159         while (result_temp != head1)
160         {
161             result_temp = result_temp->next;
162         }
163         result_temp->next = result_head;
164 
165         result_temp->next = p2;
166         while (result_temp != head2)
167         {
168             result_temp = result_temp->next;
169         }
170         result_temp->next = result_head;
171     }
172 
173     return result_head;
174 }
175 
176 int main()
177 {
178     int choice = 0;
179     printf("Hello! What do you want to do? \n Remember to enter linked list in the
     descending order of exponents. \n");
180     struct Node *head = (struct Node *)malloc(sizeof(struct Node));
181     struct Node *head2 = (struct Node *)malloc(sizeof(struct Node));
182     struct Node *head3 = (struct Node *)malloc(sizeof(struct Node));
183     struct Node *added;
184 
```

```
    while (choice != 4)
    {
        printf("\n\
1. Create a Polynomial to represent it in a circular linked list. \n\
2. Add 2 Polynomials\n\
3. Display your Polynomial\n\
4. Quit\n");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            accept_polynomial(head);
            display_polynomial(head);
            break;
        case 2:
            printf("Please enter the first polynomial");
            accept_polynomial(head2);
            printf("\nThe First Polynomial you entered is:  \n");
            display_polynomial(head2);

            printf("Please enter the second polynomial");
            accept_polynomial(head3);
            printf("\nThe Second Polynomial you entered is:  \n");
            display_polynomial(head3);

            printf("\nThe Added Polynomial:  \n");
            added = add_polynomials(head2, head3);
            display_polynomial(added);
            break;
        case 3:
            display_polynomial(head);
        case 4:
            break;
        default:
            printf("\nInvalid\n");
            break;
        }
    }

    return 0;
}
```

9.2 Input and Output

```
Hello! What do you want to do?
 Remember to enter linked list in the descending order of exponents.

1. Create a Polynomial to represent it in a circular linked list.
2. Add 2 Polynomials
3. Display your Polynomial
4. Quit
1

Enter the Coefficient: 1

Enter the Exponent: 1
Do you want to enter more terms? (0 for no, 1 for yes)
1

```

```
Enter the Coefficient: 2

Enter the Exponent: 3
Do you want to enter more terms? (0 for no, 1 for yes)
0
1x^1 + 2x^3

1. Create a Polynomial to represent it in a circular linked list.
2. Add 2 Polynomials
3. Display your Polynomial
4. Quit
2
Please enter the first polynomial
Enter the Coefficient: 1

Enter the Exponent: 3
Do you want to enter more terms? (0 for no, 1 for yes)
1

Enter the Coefficient: 2

Enter the Exponent: 2
Do you want to enter more terms? (0 for no, 1 for yes)
1

Enter the Coefficient: 1

Enter the Exponent: 1
Do you want to enter more terms? (0 for no, 1 for yes)
0

The First Polynomial you entered is:
1x^3 + 2x^2 + 1x^1
Please enter the second polynomial
Enter the Coefficient: 2

Enter the Exponent: 3
Do you want to enter more terms? (0 for no, 1 for yes)
1

Enter the Coefficient: 5

Enter the Exponent: 2
Do you want to enter more terms? (0 for no, 1 for yes)
1

Enter the Coefficient: 1

Enter the Exponent: 1
Do you want to enter more terms? (0 for no, 1 for yes)
0

The Second Polynomial you entered is:
2x^3 + 5x^2 + 1x^1

The Added Polynomial:
3x^3 + 7x^2 + 2x^1

1. Create a Polynomial to represent it in a circular linked list.
```

```
2. Add 2 Polynomials
3. Display your Polynomial
4. Quit
4

2
Please enter the first polynomial
Enter the Coefficient: 3

Enter the Exponent: 2
Do you want to enter more terms? (0 for no, 1 for yes)
1

Enter the Coefficient: 5

Enter the Exponent: 1
Do you want to enter more terms? (0 for no, 1 for yes)
1

Enter the Coefficient: 9

Enter the Exponent: 0
Do you want to enter more terms? (0 for no, 1 for yes)
0

The First Polynomial you entered is:
3x^2 + 5x^1 + 9x^0
Please enter the second polynomial
Enter the Coefficient: 4

Enter the Exponent: 6
Do you want to enter more terms? (0 for no, 1 for yes)
1

Enter the Coefficient: 8

Enter the Exponent: 1
Do you want to enter more terms? (0 for no, 1 for yes)
0

The Second Polynomial you entered is:
4x^6 + 8x^1

The Added Polynomial:
4x^6 + 3x^2 + 13x^1 + 9x^0

1. Create a Polynomial to represent it in a circular linked list.
2. Add 2 Polynomials
3. Display your Polynomial
4. Quit
```

10 Conclusion

Thus, implemented different operations on CLL.

11 FAQ

1. **Write an ADT for CLL.**

   ADT for CLL is as follows

```c

// Structure for a node in the circular linked list.
struct Node
{
  int data;
  struct Node *next;
};

```

   Function to create a new node.

```c

struct Node *create_node(int data)
{
  struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
  new_node->data = data;
  new_node->next = NULL;
  return new_node;
}

```

   Function to create a circular linked list.

```c

struct Node *create_circular_linked_list(int data)
{
  struct Node *head = create_node(data);
  head->next = head;
  return head;
}


```

   Function to insert a node at the end of the circular linked list.

```c

struct Node *insert_at_end(struct Node *head, int data)
{
  struct Node *new_node = create_node(data);
  struct Node *temp = head;
  while (temp->next != head)
  {
    temp = temp->next;
  }
  temp->next = new_node;
  new_node->next = head;
  return head;
}


```

   Function to insert a node at a given position in the circular linked list.

```
struct Node *insert_at_position(struct Node *head, int data, int position)
{
  struct Node *new_node = create_node(data);
  struct Node *temp = head;
  int i = 1;
  while (i < position - 1)
  {
    temp = temp->next;
    i++;
  }
  new_node->next = temp->next;
  temp->next = new_node;
  return head;
}
```

Function to delete a node from the beginning of the circular linked list.

```
struct Node *delete_from_beginning(struct Node *head)
{
  struct Node *temp = head;
  while (temp->next != head)
  {
    temp = temp->next;
  }
  temp->next = head->next;
  free(head);
  head = temp->next;
  return head;
}
```

Function to delete a node from a given position in the circular linked list.

```
struct Node *delete_from_position(struct Node *head, int position)
{
  struct Node *temp = head;
  int i = 1;
  while (i < position - 1)
  {
    temp = temp->next;
    i++;
  }
  struct Node *temp2 = temp->next;
  temp->next = temp2->next;
  free(temp2);
  return head;
}
```

Function to display the circular linked list.

```
void display(struct Node *head)
{
  struct Node *temp = head;
  while (temp->next != head)
```

```
    {
      printf("%d ", temp->data);
      temp = temp->next;
    }
    printf("%d ", temp->data);
    printf(" ");
  }

```

Function to search for a node in the circular linked list.

```

  int search(struct Node *head, int data)
  {
    struct Node *temp = head;
    int position = 1;
    while (temp->next != head)
    {
      if (temp->data == data)
      {
        return position;
      }
      temp = temp->next;
      position++;
    }
    if (temp->data == data)
    {
      return position;
    }
    return -1;
  }

```

Function to sort the circular linked list.

```

  struct Node *sort(struct Node *head)
  {
    struct Node *temp = head;
    struct Node *temp2 = NULL;
    int temp_data;
    while (temp->next != head)
    {
      temp2 = temp->next;
      while (temp2 != head)
      {
        if (temp->data > temp2->data)
        {
          temp_data = temp->data;
          temp->data = temp2->data;
          temp2->data = temp_data;
        }
        temp2 = temp2->next;
      }
      temp = temp->next;
    }
    return head;
  }


```

2. **How to perform multiplication of two polynomials?**

   Multiplication of two polynomials is similar to multiplication of two numbers.

   (a) For each term in the first polynomial, multiply it with each term in the second polynomial.

   (b) While multiplying, add the exponents of the two terms. and Multiply the Coefficients.

3. **Write polynomial addition algorithm if terms are not sorted.**

   In case the terms are not sorted, we can either sort both the polynomials, and then proceed to add them in the usual way, or we can follow this algorithm

   (a) For each polynomial in the First Polynomial, check for the presence of a term with the same exponent in the other polynomial by looping over it.

   (b) If the term is present, add the coefficients of the two terms, and store the result. Thereby Incrementing the index of the first polynomial.

   (c) If the term is not present, store the term in the result polynomial. Thereby Incrementing the index of the first polynomial.

MIT World Peace University

Advanced Data Structures
Second Year B. Tech, Semester 4

Creation and Traversal of Binary Trees (Recursive and Non-Recursive)

Assignment No. 2

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

February 5, 2023

Contents

1 Objectives **1**

2 Problem Statement **1**

3 Theory **1**
3.1 Tree . . . 1
3.2 Binary Tree . . . 1
3.2.1 Binary Tree Properties . . . 1
3.2.2 Binary Tree Representation . . . 2
3.2.3 Types of Binary Tree . . . 2
3.2.4 Different definitions related to binary tree. . . . 2
3.3 Different Traversals (Inorder, Preorder and Postorder) . . . 3
3.3.1 Inorder Traversal . . . 3
3.3.2 Algorithm for Inorder Recursive Travesal . . . 3
3.3.3 Algorithm for Inorder Iterative Travesal . . . 3
3.3.4 Preorder Traversal . . . 4
3.3.5 Algorithm for PreOrder Recursive Travesal . . . 4
3.3.6 Algorithm for PreOrder Iterative Travesal . . . 4
3.3.7 Postorder Traversal . . . 4
3.3.8 Algorithm for PostOrder Recursive Travesal . . . 4
3.3.9 Algorithm for PostOrder Iterative Travesal . . . 4

4 Platform **5**

5 Input **5**

6 Output **5**

7 Test Conditions **5**

8 Pseudo Code **5**
8.1 Inorder Traversal - Iterative Approach . . . 5
8.2 Preorder Traversal - Iterative Approach . . . 5
8.3 Postorder Traversal - Iterative Approach . . . 6

9 Time Complexity **6**

10 Code **6**
10.1 Program . . . 6
10.2 Input and Output . . . 11

11 Conclusion **20**

12 FAQ **21**

1 Objectives

1. To study data structure : Tree and Binary Tree
2. To study different traversals in Binary Tree
3. To study recursive and non-recursive approach of programming

2 Problem Statement

Implement binary tree using C++ and perform following operations: Creation of binary tree and traversal (recursive and non- recursive)

3 Theory

3.1 Tree

1. A tree is a non-linear data structure.
2. A tree is a collection of nodes connected by edges.
3. A tree is a hierarchical data structure.
4. A tree is a data structure that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.
5. A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

3.2 Binary Tree

1. A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.
2. A binary tree is a data structure for storing data such as numbers in an organized way.
3. Binary trees can be used to implement several types of abstract data types, such as sets, multisets, and associative arrays.
4. Binary trees are a useful data structure for implementing many common abstract data types, such as priority queues, binary search trees, and heaps.
5. Binary trees can be used to represent expressions involving binary operations.

3.2.1 Binary Tree Properties

1. The maximum number of nodes at level 'l' of a binary tree is 2l.
2. Maximum number of nodes in a binary tree of height 'h' is 2h - 1.

3. In a Binary Tree with N nodes, minimum possible height or minimum number of levels is

$$log_2(N+1)$$

4. A Binary Tree with L leaves has at least

$$\lceil log_2(L+1) \rceil$$

levels

3.2.2 Binary Tree Representation

1. A Binary Tree node contains following parts.
   (a) Data
   (b) Pointer to left child
   (c) Pointer to right child

3.2.3 Types of Binary Tree

1. Full Binary Tree
   (a) A Binary Tree is full if every node has 0 or 2 children.
   (b) Number of leaf nodes is always one more than nodes with two children.
2. Complete Binary Tree
   (a) A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible.
   (b) This property of Binary Tree makes them suitable to be stored in an array.
3. Perfect Binary Tree
   (a) A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.

3.2.4 Different definitions related to binary tree.

1. *Height* of a node is the number of edges on the longest path from the node to a leaf.
2. *Height* of a tree is the height of its root node.
3. *Depth* of a node is the number of edges from the node to the tree's root node.
4. *Level* of a node is defined as 1 + (the number of connections between the node and the root).
5. *Degree* of a node is the number of sub trees of a node.
6. *Degree* of a tree is the maximum degree of the nodes in the tree.
7. *Leaf* is a node with no children.
8. *Internal* node is a node with at least one child.

9. *Sibling* is a group of nodes with the same parent.
10. *Ancestor* is a node reachable by repeated proceeding from parent to parent.
11. *Descendant* is a node reachable by repeated proceeding to a child.
12. *Subtree* is a set of nodes and edges comprised of a parent and all the descendants of that parent.
13. *Forest* is a set of n Greater than or Equal to 0 disjoint trees.

3.3 Different Traversals (Inorder, Preorder and Postorder)

3.3.1 Inorder Traversal

Inorder Traversal is a recursive algorithm for traversing or searching tree data structures. It starts at the tree's root node (or another designated node of a sub-tree), and explores the sub-tree's left branch until it reaches the left-most node (i.e., a node without a left child), which it then visits. It then explores the right branch in the same manner, i.e., it recursively visits the left-most node in that sub-tree, and so on, backtracking to the root node once all nodes have been visited.

3.3.2 Algorithm for Inorder Recursive Travesal

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

3.3.3 Algorithm for Inorder Iterative Travesal

1. Create an empty stack S.
2. Initialize current node as root
3. Push the current node to S and set current = current->left until current is NULL
4. If current is NULL and stack is not empty then
   (a) Pop the top item from stack.
   (b) Print the popped item, set current = poppedItem->right
   (c) Go to step 3.
5. If current is NULL and stack is empty then we are done.

In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

3.3.4 Preorder Traversal

Preorder traversal is a recursive algorithm for traversing or searching tree data structures. It starts at the tree's root node (or another designated node of a sub-tree), visits the left subtree, then the right subtree. Printing the value of the root node after visiting the left and right subtrees. Preorder traversal is a depth-first traversal.

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expressions on an expression tree.

3.3.5 Algorithm for PreOrder Recursive Travesal

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

3.3.6 Algorithm for PreOrder Iterative Travesal

1. Create an empty stack S.
2. Initialize current node as root
3. Push the root node to the stack.
4. Pop the top item from stack and set current node = poppedItem.
5. Print the value of the node.
6. Push the right node if its not null, and then push the left node if its not null.
7. If stack isnt empty, go to step 4.
8. Exit algorithm when stack is empty.

3.3.7 Postorder Traversal

3.3.8 Algorithm for PostOrder Recursive Travesal

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

3.3.9 Algorithm for PostOrder Iterative Travesal

1. Create empty stacks S1 and S2.
2. Add the root node to S1.
3. Pop S1, and push the popped node to S2.
4. Push the left and right node of the popped node to S1.
5. Repeat Step 3 and 4 until S1 is empty.
6. Iterate through S2 and print the value of the node as it is popped.

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers : g++ and gcc on linux for C++

5 Input

1. The Nodes of the Binary Tree

6 Output

1. The traversal of the binary tree in different ways.

7 Test Conditions

1. Input at least 10 nodes.
2. Display all traversals of binary tree with 10 nodes.(recursive and nonrecursive)

8 Pseudo Code

8.1 Inorder Traversal - Iterative Approach

```
    void inorder_iterative(Node){
        if(Node == NULL){
            return;
        }
        stack<Node> s;
        Node current = Node;
        while(current != NULL || s.empty() == false){
            while(current != NULL){
                s.push(current);
                current = current->left;
            }
            current = s.top();
            s.pop();
            cout << current->data << " ";
            current = current->right;
        }
    }
```

8.2 Preorder Traversal - Iterative Approach

```
    void preorder_iterative(Node){
        if(Node == NULL){
            return;
        }
        stack<Node> s;
        s.push(Node);
```

```
        while(s.empty() == false){
            Node current = s.top();
            cout << current->data << " ";
            s.pop();
            if(current->right){
                s.push(current->right);
            }
            if(current->left){
                s.push(current->left);
            }
        }
    }
```

8.3 Postorder Traversal - Iterative Approach

```
    void postorder_iterative(Node){
        if(Node == NULL){
            return;
        }
        stack<Node> s1, s2;
        s1.push(Node);
        while(s1.empty() == false){
            Node current = s1.top();
            s1.pop();
            s2.push(current);
            if(current->left){
                s1.push(current->left);
            }
            if(current->right){
                s1.push(current->right);
            }
        }
        while(s2.empty() == false){
            Node current = s2.top();
            cout << current->data << " ";
            s2.pop();
        }
    }
```

9 Time Complexity

Time Complexities of different Traversals in their Iterative Approaches are as follows:

- Inorder Traversal: O(n)
- Preorder Traversal: O(n)
- Postorder Traversal: O(n)

There is only a single loop involved in all of these traversals, so the time complexity is linear.

10 Code

10.1 Program

```
#include <iostream>
#include <stack>

using namespace std;

class TreeNode
{
    char data[10];
    TreeNode *left;
    TreeNode *right;
    friend class BinaryTree;
};

class BinaryTree
{
public:
    TreeNode *root;
    BinaryTree()
    {
        root = NULL;
    }
    void create_root()
    {
        root = new TreeNode;
        cout << "Enter the data: " << endl;
        cin >> root->data;
        root->left = NULL;
        root->right = NULL;
        create_recursive(root);
    }
    void create_recursive(TreeNode *Node)
    {
        int choice = 0;
        TreeNode *new_node;
        cout << "Enter if you want to enter a left node (1/0): "
             << "for the node -- " << Node->data << "-- ";
        cin >> choice;
        if (choice == 1)
        {
            new_node = new TreeNode;
            cout << "Enter the data: ";
            cin >> new_node->data;
            Node->left = new_node;
            create_recursive(new_node);
        }
        cout << "Enter if you want to enter a right node (1/0): "
             << "for the node -- " << Node->data << "-- ";
        cin >> choice;
        if (choice == 1)
        {
            new_node = new TreeNode;
            cout << "Enter the data: ";
            cin >> new_node->data;
            Node->right = new_node;
            create_recursive(new_node);
        }
    }

    void inorder_recursive(TreeNode *temp)
```

```
60      {
61          if (temp == NULL)
62          {
63              return;
64          }
65          inorder_recursive(temp->left);
66          cout << temp->data << " ";
67          inorder_recursive(temp->right);
68      }
69      void inorder_iterative(TreeNode *temp)
70      {
71          if (!temp)
72          {
73              return;
74          }
75
76          stack<TreeNode *> s;
77          TreeNode *current = temp;
78
79          while (current != NULL || s.empty() == false)
80          {
81              while (current != NULL)
82              {
83                  s.push(current);
84                  current = current->left;
85              }
86              current = s.top();
87              s.pop();
88              cout << current->data << " ";
89              current = current->right;
90          }
91      }
92
93      void preorder_recursive(TreeNode *temp)
94      {
95          if (temp == NULL)
96          {
97              return;
98          }
99          cout << temp->data << " ";
100         preorder_recursive(temp->left);
101         preorder_recursive(temp->right);
102     }
103     void preorder_iterative(TreeNode *temp)
104     {
105         if (!temp)
106         {
107             return;
108         }
109
110         stack<TreeNode *> s;
111         s.push(temp);
112
113         while (s.empty() == false)
114         {
115             TreeNode *current = s.top();
116             cout << current->data << " ";
117             s.pop();
118
```

```
            if (current->right)
            {
                s.push(current->right);
            }
            if (current->left)
            {
                s.push(current->left);
            }
        }
    }

    void postorder_recursive(TreeNode *temp)
    {
        if (temp == NULL)
        {
            return;
        }
        postorder_recursive(temp->left);
        postorder_recursive(temp->right);
        cout << temp->data << " ";
    }
    void postorder_iterative(TreeNode *temp)
    {
        if (!temp)
        {
            return;
        }

        stack<TreeNode *> s1;
        stack<TreeNode *> s2;

        s1.push(temp);

        while (s1.empty() == false)
        {
            TreeNode *current = s1.top();
            s1.pop();
            s2.push(current);

            if (current->left)
            {
                s1.push(current->left);
            }
            if (current->right)
            {
                s1.push(current->right);
            }
        }
        while (s2.empty() == false)
        {
            TreeNode *current = s2.top();
            cout << current->data << " ";
            s2.pop();
        }
    }
};

int main()
{
```

```
178     int choice = 0;
179     BinaryTree main_tree;
180 
181     while (choice != 8)
182     {
183         cout << "\nWhat would like to do? " << endl;
184         cout << "\n\nWelcome to ADS Assignment 2 - Binary Tree Traversals\n\nWhat
    would you like to do? " << endl;
185         cout << "1. Create a Binary Tree"
186              << endl;
187         cout << "2. Traverse the Tree Inorder Recursively"
188              << endl;
189         cout << "3. Traverse the Tree Inorder Iteratively"
190              << endl;
191         cout << "4. Traverse the Tree PreOrder Recursively"
192              << endl;
193         cout << "5. Traverse the Tree PreOrder Iteratively"
194              << endl;
195         cout << "6. Traverse the Tree PostOrder Recursively"
196              << endl;
197         cout << "7. Traverse the Tree PostOrder Iteratively"
198              << endl;
199         cout << "8. Exit" << endl
200              << endl;
201 
202         cin >> choice;
203         switch (choice)
204         {
205         case 1:
206             main_tree.create_root();
207             break;
208         case 2:
209             cout << "Traversing through the binary tree inorder recursively: " <<
    endl;
210             main_tree.inorder_recursive(main_tree.root);
211             break;
212         case 3:
213             cout << "Traversing through the Binary Tree Inorder Iteratively: " <<
    endl;
214             main_tree.inorder_iterative(main_tree.root);
215             break;
216         case 4:
217             cout << "Traversing through the Binary Tree PreOrder Recursively: " <<
     endl;
218             main_tree.preorder_recursive(main_tree.root);
219             break;
220         case 5:
221             cout << "Traversing through the Binary Tree PreOrder Iteratively: " <<
     endl;
222             main_tree.preorder_iterative(main_tree.root);
223             break;
224         case 6:
225             cout << "Traversing through the Binary Tree PostOrder Recursively: "
    << endl;
226             main_tree.postorder_recursive(main_tree.root);
227             break;
228         case 7:
229             cout << "Traversing through the Binary Tree PostOrder Iteratively: "
    << endl;
```

```
            main_tree.postorder_iterative(main_tree.root);
            break;
        case 8:
            cout << "Exiting the program" << endl;
            exit(0);
            break;
        default:
            cout << "Invalid Choice" << endl;
            break;
        }
    }
}
```

10.2 Input and Output

```
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

1
Enter the data:  1
Enter if you want to enter a left node (1/0): for the node -- 1-- 1
Enter the data:  2
Enter if you want to enter a left node (1/0): for the node -- 2-- 1
Enter the data:  3
Enter if you want to enter a left node (1/0): for the node -- 3-- 1
Enter the data:  4
Enter if you want to enter a left node (1/0): for the node -- 4-- 1
Enter the data:  5
Enter if you want to enter a left node (1/0): for the node -- 5-- 0
Enter if you want to enter a right node (1/0): for the node -- 5-- 0
Enter if you want to enter a right node (1/0): for the node -- 4-- 0
Enter if you want to enter a right node (1/0): for the node -- 3-- 0
Enter if you want to enter a right node (1/0): for the node -- 2-- 0
Enter if you want to enter a right node (1/0): for the node -- 1-- 0

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
```

```
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

5
Traversing through the binary tree inorder recursively:
5 4 3 2 1
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

7
Traversing through the Binary Tree PreOrder Recursively:
1 2 3 4 5
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

9
Traversing through the Binary Tree PostOrder Recursively:
5 4 3 2 1

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals
```

```
104
105 What would you like to do?
106 1. Create a Binary Tree
107 2. Insert Elements to the Tree in Auto Ascending Order
108 3. Insert Elements to the Tree Level by Level
109 4. Insert Elements into the Tree Manually
110 5. Traverse the Tree Inorder Recursively
111 6. Traverse the Tree Inorder Iteratively
112 7. Traverse the Tree PreOrder Recursively
113 8. Traverse the Tree PreOrder Iteratively
114 9. Traverse the Tree PostOrder Recursively
115 10. Traverse the Tree PostOrder Iteratively
116 11. Exit
117
118 1
119 Enter the data:  1
120 Enter if you want to enter a left node (1/0): for the node -- 1-- 0
121 Enter if you want to enter a right node (1/0): for the node -- 1-- 1
122 Enter the data:  2
123 Enter if you want to enter a left node (1/0): for the node -- 2-- 0
124 Enter if you want to enter a right node (1/0): for the node -- 2-- 1
125 Enter the data:  3
126 Enter if you want to enter a left node (1/0): for the node -- 3-- 0
127 Enter if you want to enter a right node (1/0): for the node -- 3-- 1
128 Enter the data:  4
129 Enter if you want to enter a left node (1/0): for the node -- 4-- 0
130 Enter if you want to enter a right node (1/0): for the node -- 4-- 1
131 Enter the data:  5
132 Enter if you want to enter a left node (1/0): for the node -- 5-- 0
133 Enter if you want to enter a right node (1/0): for the node -- 5-- 0
134
135 What would like to do?
136
137
138 Welcome to ADS Assignment 2 - Binary Tree Traversals
139
140 What would you like to do?
141 1. Create a Binary Tree
142 2. Insert Elements to the Tree in Auto Ascending Order
143 3. Insert Elements to the Tree Level by Level
144 4. Insert Elements into the Tree Manually
145 5. Traverse the Tree Inorder Recursively
146 6. Traverse the Tree Inorder Iteratively
147 7. Traverse the Tree PreOrder Recursively
148 8. Traverse the Tree PreOrder Iteratively
149 9. Traverse the Tree PostOrder Recursively
150 10. Traverse the Tree PostOrder Iteratively
151 11. Exit
152
153 6
154 Traversing through the Binary Tree Inorder Iteratively:
155 1 2 3 4 5
156 What would like to do?
157
158
159 Welcome to ADS Assignment 2 - Binary Tree Traversals
160
161 What would you like to do?
162 1. Create a Binary Tree
```

```
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

8
Traversing through the Binary Tree PreOrder Iteratively:
1 2 3 4 5
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

10
Traversing through the Binary Tree PostOrder Iteratively:
5 4 3 2 1


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

1
Enter the data:  1
Enter if you want to enter a left node (1/0): for the node -- 1-- 1
Enter the data:  2
Enter if you want to enter a left node (1/0): for the node -- 2-- 1
Enter the data:  4
Enter if you want to enter a left node (1/0): for the node -- 4-- 0
```

```
Enter if you want to enter a right node (1/0): for the node -- 4-- 0
Enter if you want to enter a right node (1/0): for the node -- 2-- 1
Enter the data:  5
Enter if you want to enter a left node (1/0): for the node -- 5-- 0
Enter if you want to enter a right node (1/0): for the node -- 5-- 0
Enter if you want to enter a right node (1/0): for the node -- 1-- 1
Enter the data:  3
Enter if you want to enter a left node (1/0): for the node -- 3-- 1
Enter the data:  6
Enter if you want to enter a left node (1/0): for the node -- 6-- 0
Enter if you want to enter a right node (1/0): for the node -- 6-- 0
Enter if you want to enter a right node (1/0): for the node -- 3-- 1
Enter the data:  7
Enter if you want to enter a left node (1/0): for the node -- 7-- 0
Enter if you want to enter a right node (1/0): for the node -- 7-- 0


What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

5
Traversing through the binary tree inorder recursively:
4 2 5 1 6 3 7
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

7
Traversing through the Binary Tree PreOrder Recursively:
1 2 4 5 3 6 7
```

```
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

9
Traversing through the Binary Tree PostOrder Recursively:
4 5 2 6 7 3 1


What would like to do?
Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

1
Enter the data:
1
Enter if you want to enter a left node (1/0): for the node -- 1-- 1
Enter the data: 2
Enter if you want to enter a left node (1/0): for the node -- 2-- 1
Enter the data: 4
Enter if you want to enter a left node (1/0): for the node -- 4-- 1
Enter the data: 8
Enter if you want to enter a left node (1/0): for the node -- 8-- 0
Enter if you want to enter a right node (1/0): for the node -- 8-- 0
Enter if you want to enter a right node (1/0): for the node -- 4-- 1
Enter the data: 9
Enter if you want to enter a left node (1/0): for the node -- 9-- 0
Enter if you want to enter a right node (1/0): for the node -- 9-- 0
Enter if you want to enter a right node (1/0): for the node -- 2-- 1
Enter the data: 5
Enter if you want to enter a left node (1/0): for the node -- 5-- 0
Enter if you want to enter a right node (1/0): for the node -- 5-- 0
Enter if you want to enter a right node (1/0): for the node -- 1-- 1
```

```
Enter the data: 3
Enter if you want to enter a left node (1/0): for the node -- 3-- 1
Enter the data: 6
Enter if you want to enter a left node (1/0): for the node -- 6-- 0
Enter if you want to enter a right node (1/0): for the node -- 6-- 0
Enter if you want to enter a right node (1/0): for the node -- 3-- 1
Enter the data: 7
Enter if you want to enter a left node (1/0): for the node -- 7-- 0
Enter if you want to enter a right node (1/0): for the node -- 7-- 0

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

5
Traversing through the binary tree inorder recursively:
8 4 9 2 5 1 6 3 7
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

7
Traversing through the Binary Tree PreOrder Recursively:
1 2 4 8 9 5 3 6 7
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
```

```
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

10
Traversing through the Binary Tree PostOrder Iteratively:
8 9 4 5 2 6 7 3 1
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

1
Enter the data:
1
Enter if you want to enter a left node (1/0): for the node -- 1-- 1
Enter the data: 2
Enter if you want to enter a left node (1/0): for the node -- 2-- 1
Enter the data: 5
Enter if you want to enter a left node (1/0): for the node -- 5-- 0
Enter if you want to enter a right node (1/0): for the node -- 5-- 0
Enter if you want to enter a right node (1/0): for the node -- 2-- 1
Enter the data: 3
Enter if you want to enter a left node (1/0): for the node -- 3-- 1
```

```
Enter the data: 4
Enter if you want to enter a left node (1/0): for the node -- 4-- 0
Enter if you want to enter a right node (1/0): for the node -- 4-- 0
Enter if you want to enter a right node (1/0): for the node -- 3-- 0
Enter if you want to enter a right node (1/0): for the node -- 1-- 1
Enter the data: 6
Enter if you want to enter a left node (1/0): for the node -- 6-- 0
Enter if you want to enter a right node (1/0): for the node -- 6-- 0

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

6
Traversing through the Binary Tree Inorder Iteratively:
5 2 4 3 1 6
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

8
Traversing through the Binary Tree PreOrder Iteratively:
1 2 5 3 4 6
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
```

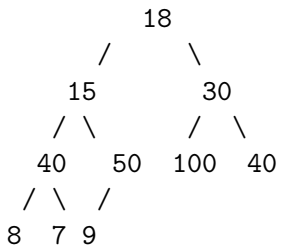
```
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit

9
Traversing through the Binary Tree PostOrder Recursively:
5 4 3 2 6 1
What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Tree
2. Insert Elements to the Tree in Auto Ascending Order
3. Insert Elements to the Tree Level by Level
4. Insert Elements into the Tree Manually
5. Traverse the Tree Inorder Recursively
6. Traverse the Tree Inorder Iteratively
7. Traverse the Tree PreOrder Recursively
8. Traverse the Tree PreOrder Iteratively
9. Traverse the Tree PostOrder Recursively
10. Traverse the Tree PostOrder Iteratively
11. Exit
```

11 Conclusion

Thus, learnt about the different kinds of traversals in binary tree and also learnt about the recursive and non-recursive approach of programming.

12 FAQ

1. Explain any one application of binary tree with suitable example.
2. Explain sequential representation of binary tree with example.
3. Write inorder, preorder and postorder for following tree.



Answer:

(a) Inorder: 8, 40, 7, 15, 9, 50, 18, 100, 30, 100, 40

(b) Preorder: 15, 40, 8, 7, 50, 9, 18, 30, 100, 40, 100

(c) Postorder: 8, 7, 40, 9, 50, 15, 100, 40, 100, 30, 18

MIT World Peace University

Advanced Data Structures
Second Year B. Tech, Semester 4

Implementation of Dictionary using Binary Search Tree

Assignment No. 3

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

February 15, 2023

Contents

1 Objectives **1**

2 Problem Statement **1**

3 Theory **1**
- 3.1 Binary Search Tree . . . 1
- 3.2 Breadth First Traversal . . . 2
- 3.3 Different operations on binary search tree . . . 2
  - 3.3.1 Copy . . . 2
  - 3.3.2 Insert . . . 2
  - 3.3.3 Mirror Image . . . 2
  - 3.3.4 Delete . . . 3

4 Platform **3**

5 Input **3**

6 Output **4**

7 Test Conditions **4**

8 Pseudo Code **4**
- 8.1 Create . . . 4
- 8.2 Display . . . 5
- 8.3 Delete . . . 5
- 8.4 Mirror Image . . . 6
- 8.5 Copy . . . 6

9 Time Complexity **7**
- 9.1 Create . . . 7
- 9.2 Display . . . 7
- 9.3 Delete . . . 7
- 9.4 Mirror Image . . . 7
- 9.5 Copy . . . 7

10 Code **7**
- 10.1 Program . . . 7
- 10.2 Input and Output . . . 16

11 Conclusion **19**

12 FAQ **20**

1 Objectives

1. To study data structure : Binary Search Tree
2. To study breadth first traversal.
3. To study different operations on Binary search Tree.

2 Problem Statement

Implement dictionary using binary search tree where dictionary stores keywords and its meanings. Perform following operations:

1. Insert a keyword
2. Delete a keyword
3. Create mirror image and display level wise
4. Copy the binary Search Tree

3 Theory

3.1 Binary Search Tree

Binary Search Trees are a special type of binary trees where the value of all the nodes in the left sub-tree is less than the value of the root and the value of all the nodes in the right sub-tree is greater than the value of the root.

The left and right sub-trees are also binary search trees. This property of binary search trees makes them useful for searching, as the desired node can be found by repeatedly comparing the input to the value of the root and choosing the appropriate sub-tree, without having to search through the entire tree.

Binary search trees are also useful for sorting, as it is easy to sort the nodes in ascending order by performing an in-order traversal of the tree.

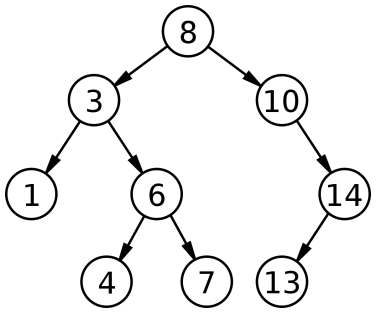


Figure 1: Example of a Binary Search Tree

3.2 Breadth First Traversal

Breadth First Traversal (or Level Order Traversal) is a tree traversal algorithm where we should start traversing the tree from root and traverse the tree level wise i.e. traverse the nodes level by level.

In level order traversal, we visit the nodes level by level from left to right.

3.3 Different operations on binary search tree

3.3.1 Copy

To copy a Binary Search Tree into another Binary Search Tree, we perform a pre-order traversal of the tree. For each node, we create a new node with the same value and then recursively copy the left and right sub-trees of the node.

To copy it Iteratively, we use a queue to store the nodes of the tree. We start by pushing the root node into the queue. We then pop a node from the queue and create a new node with the same value as the popped node. We then push the left and right child of the popped node into the queue and repeat the process until the queue is empty.

3.3.2 Insert

To insert a node in a binary search tree, we start by comparing the value of the node to be inserted with the value of the root node. If the value of the node to be inserted is less than the value of the root node, we move to the left sub-tree and repeat the process. If the value of the node to be inserted is greater than the value of the root node, we move to the right sub-tree and repeat the process.

We repeat this process until we reach a leaf node. The new node is then inserted as the left or right child of the leaf node, depending on the value of the node to be inserted.

3.3.3 Mirror Image

To create a mirror image of a binary search tree, we perform a pre-order traversal of the tree. For each node, we swap the left and right child of the node. We then recursively create the mirror image of the left and right sub-trees of the node.

To create it Iteratively, we use a stack to store the nodes of the tree. We start by pushing the root node into the stack. We then pop a node from the stack and swap the left and right child of the popped node. We then push the left and right child of the popped node into the stack and repeat the process until the stack is empty.

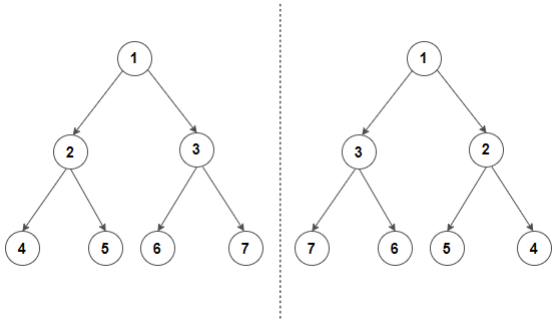


Figure 2: Mirror of a Binary Tree

3.3.4 Delete

There are 3 Cases for deletion of a node in a binary search tree:

1. The node to be deleted is a leaf node. In this case, we simply remove the node from the tree.
2. The node to be deleted has only one child. In this case, we replace the node to be deleted with its child.
3. The node to be deleted has two children. In this case, we find the inorder successor of the node. The inorder successor is the smallest node in the right sub-tree of the node. We replace the value of the node to be deleted with the value of the inorder successor. We then delete the inorder successor. The inorder successor will have at most one child node, so we can use the above two cases to delete it.

   It can also be done using the inorder predecessor. The inorder predecessor is the largest node in the left sub-tree of the node. We replace the value of the node to be deleted with the value of the inorder predecessor. We then delete the inorder predecessor. The inorder predecessor will have at most one child node, so we can use the above two cases to delete it.

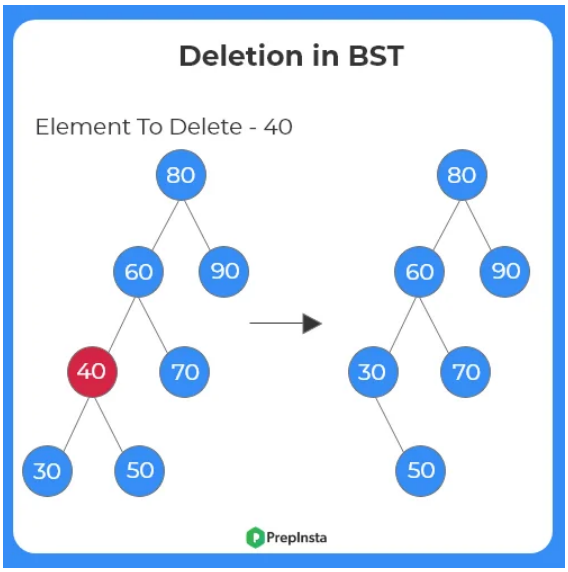


Figure 3: Deleting a node from a Binary Search Tree

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers : g++ and gcc on linux for C++

5 Input

1. Input at least 10 nodes.

2. Display binary search tree levelwise traversals of binary search tree with 10 nodes
3. Display mirror image and copy operations on BST

6 Output

1. The traversal of the binary tree in different ways.

7 Test Conditions

1. Input at least 10 nodes.
2. Display all traversals of binary tree with 10 nodes.(recursive and nonrecursive)

8 Pseudo Code

8.1 Create

```
void create_root()
      root = new WordNode
      display - "Enter the data: " << endl
      Take Input of root->word
      Take Input of root->definition
      root->left = NULL
      root->right = NULL
      create_recursive(root)

 void create_recursive(WordNode *Node)
      int choice = 0
      WordNode *new_node
      display - "Enter if you want to enter a left node (1/0): "
           << "for the node -- " << Node->word << ":  " << Node->definition << "
  -- "
      Take Input of choice
      if (choice == 1)
          new_node = new WordNode
          display - "Enter the data: "
          Take Input of new_node->word
          Take Input of new_node->definition
          Node->left = new_node
          create_recursive(new_node)
      display - "Enter if you want to enter a right node (1/0): "
           << "for the node -- " << Node->word << ":  " << Node->definition << "
  -- "
      Take Input of choice
      if (choice == 1)
          new_node = new WordNode
          display - "Enter the data: "
          Take Input of new_node->word
          Take Input of new_node->definition
          Node->right = new_node
          create_recursive(new_node)
```

8.2 Display

```
    void bfs()
    {
        WordNode *temp = root
        queue<WordNode *> q
        q.push(temp)
        while (!q.empty())
        {
            temp = q.front()
            q.pop()
            display - temp->word << " : " << temp->definition << endl
            if (temp->left != NULL)
            {
                q.push(temp->left)
            }
            if (temp->right != NULL)
            {
                q.push(temp->right)
            }
        }
    }
```

8.3 Delete

```
  void delete_node(WordNode *temp, string word)

        WordNode *parent = NULL;
        while (temp != NULL)
            if (temp->word == word)
                break;
            else
                parent = temp;
                if (strcmp(word.c_str(), temp->word.c_str()) < 0)
                    temp = temp->left;
                else
                    temp = temp->right;
        if (temp == NULL)
            Print - "Word not found" - endl;
            return;
        else
            // no child node.
            if (temp->left == NULL && temp->right == NULL)
                if (parent->left == temp)
                    parent->left = NULL;
                else
                    parent->right = NULL;
                delete temp;

      // 1 Child case right.
            else if (temp->left == NULL)
                if (parent->left == temp)
                    parent->left = temp->right;
                else
                    parent->right = temp->right;
                delete temp;

```

```
    // 1 Child case left.
          else if (temp->right == NULL)
              if (parent->left == temp)
                  parent->left = temp->left;
              else
                  parent->right = temp->left;
              delete temp;
          else
              WordNode *temp1 = temp->right;
              while (temp1->left != NULL)
                  temp1 = temp1->left;
              temp->word = temp1->word;
              temp->definition = temp1->definition;
              delete_node(temp->right, temp1->word);
```

8.4 Mirror Image

```
void mirror_recursive(WordNode *temp)
{
    if (temp == NULL)
    {
        return
    }
    else
    {
        WordNode *temp1
        // Swapping
        temp1 = temp->left
        temp->left = temp->right
        temp->right = temp1

        // Recursively calling the function
        mirror_recursive(temp->left)
        mirror_recursive(temp->right)
    }
}
```

8.5 Copy

```
WordNode *create_copy_recursive(WordNode *temp)
{
    if (temp == NULL)
    {
        return NULL
    }
    else
    {
        WordNode *new_node = new WordNode
        new_node->word = temp->word
        new_node->definition = temp->definition
        new_node->left = create_copy_recursive(temp->left)
        new_node->right = create_copy_recursive(temp->right)
        return new_node
    }
}
```

9 Time Complexity

9.1 Create

- **Time Complexity Worst Case**:

$$\bigcirc(n^2)$$

- **Time Complexity Best Case**:

$$\bigcirc(\log(n))$$

9.2 Display

- **Time Complexity**:

$$\bigcirc(n)$$

9.3 Delete

- **Time Complexity**:

$$\bigcirc(h)$$

$$h$$

is the height of the Binary search tree.

9.4 Mirror Image

- **Time Complexity**:

$$\bigcirc(n)$$

9.5 Copy

- **Time Complexity**:

$$\bigcirc(n)$$

10 Code

10.1 Program

```
#include <iostream>
#include <queue>
#include <stack>
#include <string.h>
using namespace std;

class WordNode
{
    string word;
    string definition;
    WordNode *left;
    WordNode *right;
    friend class BinarySearchTree;
```

```
};

class BinarySearchTree
{
public:
    WordNode *root;
    BinarySearchTree()
    {
        root = NULL;
    }
    void create_root()
    {
        root = new WordNode;
        cout << "Enter the data: " << endl;
        cin >> root->word;
        cin >> root->definition;
        root->left = NULL;
        root->right = NULL;
        create_recursive(root);
    }
    void create_recursive(WordNode *Node)
    {
        int choice = 0;
        WordNode *new_node;
        cout << "Enter if you want to enter a left node (1/0): "
             << "for the node -- " << Node->word << ":  " << Node->definition << "
    -- ";
        cin >> choice;
        if (choice == 1)
        {
            new_node = new WordNode;
            cout << "Enter the data: ";
            cin >> new_node->word;
            cin >> new_node->definition;
            Node->left = new_node;
            create_recursive(new_node);
        }
        cout << "Enter if you want to enter a right node (1/0): "
             << "for the node -- " << Node->word << ":  " << Node->definition << "
    -- ";
        cin >> choice;
        if (choice == 1)
        {
            new_node = new WordNode;
            cout << "Enter the data: ";
            cin >> new_node->word;
            cin >> new_node->definition;
            Node->right = new_node;
            create_recursive(new_node);
        }
    }
    void create_root_and_tree_iteratively()
    {
        WordNode *temp, *current_word;
        int choice = 0;
        root = new WordNode;
        cout << "Enter the Word: ";
        cin >> root->word;
        cout << "Enter the definition of the word: ";
```

```
71          cin >> root->definition;
72          bool flag = false;
73          cout << "Do you want to enter more Nodes? (0/1) " << endl;
74          cin >> choice;
75          while (choice == 1)
76          {
77              temp = root;
78              flag = false;
79              current_word = new WordNode;
80              cout << "Enter the Word: ";
81              cin >> current_word->word;
82              cout << "Enter the definition of the word: ";
83              cin >> current_word->definition;
84
85              while (!flag)
86              {
87                  if (strcmp(current_word->word.c_str(), temp->word.c_str()) <= 0)
88                  {
89                      if (temp->left == NULL)
90                      {
91                          temp->left = current_word;
92                          flag = true;
93                      }
94                      else
95                      {
96                          temp = temp->left;
97                      }
98                  }
99                  else
100                 {
101                     if (temp->right == NULL)
102                     {
103                         temp->right = current_word;
104                         flag = true;
105                     }
106                     else
107                     {
108                         temp = temp->right;
109                     }
110                 }
111             }
112             cout << "Do you want to enter another word? (1/0): ";
113             cin >> choice;
114         }
115     }
116
117     // breadth First Search using queue.
118     void bfs()
119     {
120         WordNode *temp = root;
121         queue<WordNode *> q;
122         q.push(temp);
123         while (!q.empty())
124         {
125             temp = q.front();
126             q.pop();
127             cout << temp->word << " : " << temp->definition << endl;
128             if (temp->left != NULL)
129             {
```

```
130                 q.push(temp->left);
131             }
132             if (temp->right != NULL)
133             {
134                 q.push(temp->right);
135             }
136         }
137     }
138 
139     WordNode *create_copy_recursive(WordNode *temp)
140     {
141         if (temp == NULL)
142         {
143             return NULL;
144         }
145         else
146         {
147             WordNode *new_node = new WordNode;
148             new_node->word = temp->word;
149             new_node->definition = temp->definition;
150             new_node->left = create_copy_recursive(temp->left);
151             new_node->right = create_copy_recursive(temp->right);
152             return new_node;
153         }
154     }
155 
156     WordNode *create_copy_interatively(WordNode *temp)
157     {
158         // We have to create a queue to pop things
159         queue<WordNode *> q;
160         WordNode *copied_tree;
161         WordNode *new_node = new WordNode;
162         new_node->word = temp->word;
163         new_node->definition = temp->definition;
164         q.push(new_node);
165         while (!q.empty())
166         {
167             copied_tree = q.front();
168             q.pop();
169             if (temp->left != NULL)
170             {
171                 WordNode *new_node1 = new WordNode;
172                 new_node1->word = temp->left->word;
173                 new_node1->definition = temp->left->definition;
174                 copied_tree->left = new_node1;
175                 q.push(new_node1);
176             }
177             if (temp->right != NULL)
178             {
179                 WordNode *new_node1 = new WordNode;
180                 new_node1->word = temp->right->word;
181                 new_node1->definition = temp->right->definition;
182                 copied_tree->right = new_node1;
183                 q.push(new_node1);
184             }
185             temp = temp->left;
186         }
187         return copied_tree;
188     }
```

```
189
190     void mirror_recursive(WordNode *temp)
191     {
192         if (temp == NULL)
193         {
194             return;
195         }
196         else
197         {
198             WordNode *temp1;
199             // Swapping
200             temp1 = temp->left;
201             temp->left = temp->right;
202             temp->right = temp1;
203
204             // Recursively calling the function
205             mirror_recursive(temp->left);
206             mirror_recursive(temp->right);
207         }
208     }
209
210     void mirror_iterative(WordNode *node)
211     {
212         WordNode *temp;
213         queue<WordNode *> q;
214         q.push(node);
215         while (!q.empty())
216         {
217             temp = q.front();
218             q.pop();
219             WordNode *temp1;
220
221             // Swapping
222             temp1 = temp->left;
223             temp->left = temp->right;
224             temp->right = temp1;
225
226             if (temp->left != NULL)
227             {
228                 q.push(temp->left);
229             }
230             if (temp->right != NULL)
231             {
232                 q.push(temp->right);
233             }
234         }
235     }
236
237     WordNode *create_mirror_tree_recursive()
238     {
239         WordNode *mirror_tree = create_copy_recursive(root);
240         mirror_recursive(mirror_tree);
241         return mirror_tree;
242     }
243
244     WordNode *create_mirror_tree_iterative()
245     {
246         WordNode *mirror_tree = create_copy_recursive(root);
247         mirror_iterative(mirror_tree);
```

```
248         return mirror_tree;
249     }
250 
251     void delete_node(WordNode *temp, string word)
252     {
253         WordNode *parent = NULL;
254         while (temp != NULL)
255         {
256             if (temp->word == word)
257             {
258                 break;
259             }
260             else
261             {
262                 parent = temp;
263                 if (strcmp(word.c_str(), temp->word.c_str()) < 0)
264                 {
265                     temp = temp->left;
266                 }
267                 else
268                 {
269                     temp = temp->right;
270                 }
271             }
272         }
273         if (temp == NULL)
274         {
275             cout << "Word not found" << endl;
276             return;
277         }
278         else
279         {
280             // no child node.
281             if (temp->left == NULL && temp->right == NULL)
282             {
283                 if (parent->left == temp)
284                 {
285                     parent->left = NULL;
286                 }
287                 else
288                 {
289                     parent->right = NULL;
290                 }
291                 delete temp;
292             }
293             // 1 Child case right.
294             else if (temp->left == NULL)
295             {
296                 if (parent->left == temp)
297                 {
298                     parent->left = temp->right;
299                 }
300                 else
301                 {
302                     parent->right = temp->right;
303                 }
304                 delete temp;
305             }
306             // 1 Child case left.
```

```
307             else if (temp->right == NULL)
308             {
309                 if (parent->left == temp)
310                 {
311                     parent->left = temp->left;
312                 }
313                 else
314                 {
315                     parent->right = temp->left;
316                 }
317                 delete temp;
318             }
319             else
320             {
321                 WordNode *temp1 = temp->right;
322                 while (temp1->left != NULL)
323                 {
324                     temp1 = temp1->left;
325                 }
326                 temp->word = temp1->word;
327                 temp->definition = temp1->definition;
328                 delete_node(temp->right, temp1->word);
329             }
330         }
331     }
332
333     void inorder_iterative(WordNode *temp)
334     {
335         if (!temp)
336         {
337             return;
338         }
339
340         stack<WordNode *> s;
341         WordNode *current = temp;
342
343         while (current != NULL || s.empty() == false)
344         {
345             while (current != NULL)
346             {
347                 s.push(current);
348                 current = current->left;
349             }
350             current = s.top();
351             s.pop();
352             cout << current->word << " : " << current->definition << endl;
353             current = current->right;
354         }
355     }
356     void preorder_iterative(WordNode *temp)
357     {
358         if (!temp)
359         {
360             return;
361         }
362
363         stack<WordNode *> s;
364         s.push(temp);
365
```

```
366         while (s.empty() == false)
367         {
368             WordNode *current = s.top();
369             cout << current->word << " : " << current->definition << endl;
370             s.pop();
371 
372             if (current->right)
373             {
374                 s.push(current->right);
375             }
376             if (current->left)
377             {
378                 s.push(current->left);
379             }
380         }
381     }
382     void postorder_iterative(WordNode *temp)
383     {
384         if (!temp)
385         {
386             return;
387         }
388 
389         stack<WordNode *> s1;
390         stack<WordNode *> s2;
391 
392         s1.push(temp);
393 
394         while (s1.empty() == false)
395         {
396             WordNode *current = s1.top();
397             s1.pop();
398             s2.push(current);
399 
400             if (current->left)
401             {
402                 s1.push(current->left);
403             }
404             if (current->right)
405             {
406                 s1.push(current->right);
407             }
408         }
409         while (s2.empty() == false)
410         {
411             WordNode *current = s2.top();
412             cout << current->word << " : " << current->definition << endl;
413             s2.pop();
414         }
415     }
416 };
417 
418 int main()
419 {
420     int choice = 0;
421     string word;
422     BinarySearchTree main_tree, mirror_tree, copy_tree;
423 
424     while (choice != 10)
```

```
425     {
426         cout << "\nWhat would like to do? " << endl;
427         cout << "\n\nWelcome to ADS Assignment 2 - Binary Tree Traversals\n\nWhat
    would you like to do? " << endl;
428         cout << "1. Create a Binary Search Tree"
429              << endl;
430         cout << "2. Traverse the Tree Inorder Iteratively"
431              << endl;
432         cout << "3. Traverse the Tree PreOrder Iteratively"
433              << endl;
434         cout << "4. Traverse the Tree PostOrder Iteratively"
435              << endl;
436         cout << "5. Traverse it using BFS"
437              << endl;
438         cout << "6. Create a Copy of the tree Recursively and Iteratively"
439              << endl;
440         cout << "7. Create a Mirror of the Tree Recursively"
441              << endl;
442         cout << "8. Create a Mirror of the Tree Iteratively"
443              << endl;
444         cout << "9. Delete a Node from the Tree"
445              << endl;
446         cout << "10. Exit" << endl
447              << endl;
448 
449         cin >> choice;
450         switch (choice)
451         {
452         case 1:
453             main_tree.create_root_and_tree_iteratively();
454             break;
455         case 2:
456             cout << "Traversing through the Binary Tree Inorder Iteratively: " <<
    endl;
457             main_tree.inorder_iterative(main_tree.root);
458             break;
459         case 3:
460             cout << "Traversing through the Binary Tree PreOrder Iteratively: " <<
     endl;
461             main_tree.preorder_iterative(main_tree.root);
462             break;
463         case 4:
464             cout << "Traversing through the Binary Tree PostOrder Iteratively: "
    << endl;
465             main_tree.postorder_iterative(main_tree.root);
466             break;
467         case 5:
468             cout << "Traversing through the Binary Tree using BFS: " << endl;
469             main_tree.bfs();
470             break;
471         case 6:
472             cout << "Creating a copy of the tree" << endl;
473             copy_tree.root = copy_tree.create_copy_recursive(main_tree.root);
474             cout << "Traversing via Breadth First Search: " << endl;
475             copy_tree.bfs();
476             break;
477         case 7:
478             cout << "Creating a mirror of the tree" << endl;
479             mirror_tree.root = main_tree.create_mirror_tree_recursive();
```

```
480             cout << "Traversing via Breadth First Search: " << endl;
481             mirror_tree.bfs();
482             break;
483         case 8:
484             cout << "Creating a mirror of the tree Iteratively" << endl;
485             mirror_tree.root = main_tree.create_mirror_tree_iterative();
486             cout << "Traversing via Breadth First Search: " << endl;
487             mirror_tree.bfs();
488             break;
489         case 9:
490             cout << "Enter the word you want to delete: " << endl;
491             cin >> word;
492             main_tree.delete_node(main_tree.root, word);
493             cout << "Traversing through the Binary Tree Inorder Iteratively: " <<
    endl;
494             main_tree.inorder_iterative(main_tree.root);
495             break;
496         case 10:
497             cout << "Exiting the program" << endl;
498             break;
499         default:
500             cout << "Invalid Choice" << endl;
501             break;
502         }
503     }
504 }
```

10.2 Input and Output

```
1  What would like to do?
2
3
4  Welcome to ADS Assignment 2 - Binary Tree Traversals
5
6  What would you like to do?
7  1. Create a Binary Search Tree
8  2. Traverse the Tree Inorder Iteratively
9  3. Traverse the Tree PreOrder Iteratively
10 4. Traverse the Tree PostOrder Iteratively
11 5. Traverse it using BFS
12 6. Create a Copy of the tree Recursively
13 7. Create a Mirror of the Tree Recursively
14 8. Exit
15
16 1
17 Enter the Word: apple
18 Enter the definition of the word: fruit
19 Do you want to enter more Nodes? (0/1)
20
21 1
22 Enter the Word: banana
23 Enter the definition of the word: fruit
24 Do you want to enter another word? (1/0): 1
25 Enter the Word: keyboard
26 Enter the definition of the word: input
27 Do you want to enter another word? (1/0): 1
28 Enter the Word: pears
29 Enter the definition of the word: fruit
30 Do you want to enter another word? (1/0): 1
```

```
Enter the Word: bottle
Enter the definition of the word: water
Do you want to enter another word? (1/0): 1
Enter the Word: charger
Enter the definition of the word: charging
Do you want to enter another word? (1/0): 1
Enter the Word: monitor
Enter the definition of the word: see
Do you want to enter another word? (1/0): 1
Enter the Word: paper
Enter the definition of the word: 1
Do you want to enter another word? (1/0): 1
Enter the Word: pen
Enter the definition of the word: writing
Do you want to enter another word? (1/0): 1
Enter the Word: phone
Enter the definition of the word: scrolling
Do you want to enter another word? (1/0): 0

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Search Tree
2. Traverse the Tree Inorder Iteratively
3. Traverse the Tree PreOrder Iteratively
4. Traverse the Tree PostOrder Iteratively
5. Traverse it using BFS
6. Create a Copy of the tree Recursively
7. Create a Mirror of the Tree Recursively
8. Exit

2
Traversing through the Binary Tree Inorder Iteratively:
apple : fruit
banana : fruit
bottle : water
charger : charging
keyboard : input
monitor : see
paper : 1
pears : fruit
pen : writing
phone : scrolling

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Search Tree
2. Traverse the Tree Inorder Iteratively
3. Traverse the Tree PreOrder Iteratively
4. Traverse the Tree PostOrder Iteratively
5. Traverse it using BFS
6. Create a Copy of the tree Recursively
```

```
7. Create a Mirror of the Tree Recursively
8. Exit

5
Traversing through the Binary Tree using BFS:
apple : fruit
banana : fruit
keyboard : input
bottle : water
pears : fruit
charger : charging
monitor : see
pen : writing
paper : 1
phone : scrolling

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Search Tree
2. Traverse the Tree Inorder Iteratively
3. Traverse the Tree PreOrder Iteratively
4. Traverse the Tree PostOrder Iteratively
5. Traverse it using BFS
6. Create a Copy of the tree Recursively
7. Create a Mirror of the Tree Recursively
8. Exit

6
Creating a copy of the tree
Traversing through the Binary Tree Inorder Iteratively:
apple : fruit
banana : fruit
bottle : water
charger : charging
keyboard : input
monitor : see
paper : 1
pears : fruit
pen : writing
phone : scrolling
Traversing via Breadth First Search:
apple : fruit
banana : fruit
keyboard : input
bottle : water
pears : fruit
charger : charging
monitor : see
pen : writing
paper : 1
phone : scrolling

What would like to do?


```

```
Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Search Tree
2. Traverse the Tree Inorder Iteratively
3. Traverse the Tree PreOrder Iteratively
4. Traverse the Tree PostOrder Iteratively
5. Traverse it using BFS
6. Create a Copy of the tree Recursively
7. Create a Mirror of the Tree Recursively
8. Exit

7
Creating a mirror of the tree
Traversing through the Binary Tree Inorder Iteratively:
phone : scrolling
pen : writing
pears : fruit
paper : 1
monitor : see
keyboard : input
charger : charging
bottle : water
banana : fruit
apple : fruit
Traversing via Breadth First Search:
apple : fruit
banana : fruit
keyboard : input
pears : fruit
bottle : water
pen : writing
monitor : see
charger : charging
phone : scrolling
paper : 1

What would like to do?


Welcome to ADS Assignment 2 - Binary Tree Traversals

What would you like to do?
1. Create a Binary Search Tree
2. Traverse the Tree Inorder Iteratively
3. Traverse the Tree PreOrder Iteratively
4. Traverse the Tree PostOrder Iteratively
5. Traverse it using BFS
6. Create a Copy of the tree Recursively
7. Create a Mirror of the Tree Recursively
8. Exit

8
Exiting the program
```

11 Conclusion

Thus, implemented Dictionary using Binary search tree.

12 FAQ

1. **Explain application of BST**

   The Applications of Binary Search Tree are:

   (a) Binary Search Tree is used to implement dictionaries.
   (b) Binary Search Tree is used to implement priority queues.
   (c) Binary Search Tree is used to implement disjoint sets.
   (d) Binary Search Tree is used to implement sorting algorithms.
   (e) Binary Search Tree is used to implement expression trees.
   (f) Binary Search Tree is used to implement Huffman coding.
   (g) Binary Search Tree is used to implement B-trees.
   (h) Binary Search Tree is used to implement red-black trees.

2. **Explain with example deletion of a node having two child.**

   If a node has two children, then we need to find the inorder successor of the node. The inorder successor is the smallest in the right subtree or the largest in the left subtree. After finding the inorder successor, we copy the contents of the inorder successor to the node and delete the inorder successor. Note that the inorder predecessor can also be used.

   An Example would be:

```
Let us consider the following BST as an example.
      50
     / \
   30   70
   / \  / \
  20 40 60 80
  /  \  / \
10  25 45 65
```

   Deleting 30 will be done in following steps. 1. Find inorder successor of 30. 2. Copy contents of the inorder successor to 30. 3. Delete the inorder successor. 4. Since inorder successor is 40 which has no left child, we simply make right child of 30 as the new right child of 20.

```
        50
       / \
     40   70
     / \  / \
    20 25 60 80
   /     / \
  10     45 65
```

3. **Define skewed binary tree.**

   A binary tree is said to be skewed if all of its nodes have only one child. A skewed binary tree can be either left or right skewed. A left skewed binary tree is a binary tree in which all the nodes have only left child. A right skewed binary tree is a binary tree in which all the nodes have only right child.

MIT World Peace University

Advanced Data Structures
Second Year B. Tech, Semester 4

Implementation of Threaded Binary Tree and Its Traversals

Assignment No. 4

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

March 26, 2023

Contents

1 Objectives **1**

2 Problem Statement **1**

3 Theory **1**
3.1 Threaded Binary Tree 1
3.1.1 What is Threaded Binary Tree? 1
3.1.2 Advantages of Threaded Binary Tree 1
3.2 Space Utilization in Threaded Binary Tree 2

4 Platform **2**

5 Input **2**

6 Output **2**

7 Test Conditions **2**

8 Pseudo Code **2**
8.1 Create 2
8.2 Inorder Traversal 3
8.3 PreOrder Traversal 4

9 Time Complexity **4**
9.1 Creation of Threaded Binary Tree 4
9.2 Inorder Traversal 4
9.3 Preorder Traversal 4

10 Code **4**
10.1 Program 4
10.2 Input and Output 7

11 Conclusion **9**

12 FAQ **10**

1 Objectives

1. **To study the data Structure** : Threaded Binary Tree
2. To study the advantages of Threaded Binary Tree over Binary Tree

2 Problem Statement

Implement threaded binary tree and perform inorder traversal.

3 Theory

3.1 Threaded Binary Tree

3.1.1 What is Threaded Binary Tree?

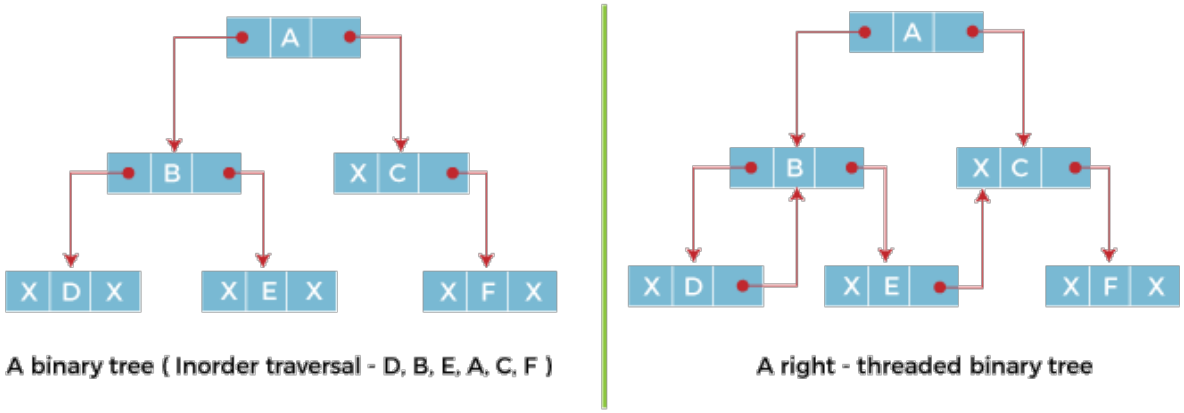


Figure 1: Threaded and a Normal Binary Tree

A Threaded Binary Tree is a binary tree where every node has an extra bit, and this bit is used to indicate whether the right pointer points to the right child or to the inorder successor of the node. In this way, we can traverse the tree without using the stack and without recursion. A Threaded Binary Tree is a binary tree where every node has an extra bit, and this bit is used to indicate whether the right pointer points to the right child or to the inorder successor of the node. In this way, we can traverse the tree without using the stack and without recursion.

3.1.2 Advantages of Threaded Binary Tree

1. Threaded Binary Tree is used to traverse the tree without using the stack and without recursion.
2. Inorder traversal of a binary tree is the most frequently used traversal. Inorder traversal of a threaded binary tree is done without using the stack and without recursion easier than the normal binary tree.

3.2 Space Utilization in Threaded Binary Tree

Space Utilized by a Normal Tree A Normal Binary Tree with N nodes each has data, left and right pointers. So, the space utilized by a normal binary tree is 3N.

Space Utilized by a Threaded Tree

A Threaded Binary Tree with N nodes each has data, left and right pointers and 2 bits. So, the space utilized by a threaded binary tree is 3N+2N = 5N.

But they avoid the usege of a stack during traversal. So, the space utilized by a threaded binary tree is less than the space utilized by a normal binary tree.

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers : g++ and gcc on linux for C++

5 Input

1. Input at least 10 nodes.
2. Display inorder traversal of binary tree with 10 nodes.

6 Output

1. The traversal of the Threaded binary tree in different ways.

7 Test Conditions

1. Input at least 10 nodes.
2. Display all traversals of binary tree with 10 nodes.(recursive and nonrecursive)

8 Pseudo Code

8.1 Create

```
// Allocate memory for root
root = new ThreadedBinaryTreeNode()

// Assign root->leftc and rightc to head
root->left = head
root->right = head

// Accept root data
print "Enter the root data: "
cin >> root->data

```

```
// Assign head lbit to 0
head->left = root
// Assign head->leftc to root
head->isLeftNodeAThread = false

ThreadedBinaryTreeNode *temp, *curr
temp = root
bool flag = true
int again = 0
int choice = 0

print "Do you want to enter another node? (1 or 0)" << endl
cin >> again
while (again != 0)
    temp = root
    flag = true
    print "Enter 1 for Entering a new node to the Left, and 2 for Entering it to
    the right" << endl
    cin >> choice
    while (flag)
        if (choice == 1)
            if (temp->isLeftNodeAThread == true)
                curr = new ThreadedBinaryTreeNode()
                print "Enter the Data: "
                cin >> curr->data
                curr->right = temp
                temp->left = curr
                temp->isLeftNodeAThread = false
                flag = false
            else
                temp = temp->left

        else if (choice == 2)
            if (temp->isRightNodeAThread == true)
                curr = new ThreadedBinaryTreeNode()
                print "Enter the Data: "
                cin >> curr->data
                curr->left = temp
                curr->right = temp->right
                temp->right = curr
                temp->isRightNodeAThread = false
                flag = false
            else
                temp = temp->right

    print "Do you want to enter another node? (1 or 0)" << endl
    cin >> again
```

8.2 Inorder Traversal

```
inorder_traversal()
    ThreadedBinaryTreeNode *temp
    temp = head
    while (true)
        temp = inorder_successor(temp)
        if (temp == head)
            break
        print temp->data << " "
```

```
ThreadedBinaryTreeNode *inorder_successor(ThreadedBinaryTreeNode *temp)
    ThreadedBinaryTreeNode *x = temp->right
    if (!temp->isRightNodeAThread)
        while (x->isLeftNodeAThread == false)
            x = x->left
    return x
```

8.3 PreOrder Traversal

```
void preorder_traversal()
    ThreadedBinaryTreeNode *temp = head->left
    while (temp != head)
        print temp->data << " "
        while (temp->isLeftNodeAThread == false)
            temp = temp->left
            print temp->data << " "
        while (temp->isRightNodeAThread == true)
            temp = temp->right
        temp = temp->right
```

9 Time Complexity

9.1 Creation of Threaded Binary Tree

- **Time Complexity:** O(n)
- **Space Complexity:** O(n)

9.2 Inorder Traversal

- **Time Complexity:** O(n)
- **Space Complexity:** O(1)

9.3 Preorder Traversal

- **Time Complexity:** O(n)
- **Space Complexity:** O(1)

10 Code

10.1 Program

```
#include <iostream>
#include <queue>
#include <stack>
#include <string.h>
using namespace std;

class ThreadedBinaryTreeNode
{
```

```
    string data;
    ThreadedBinaryTreeNode *left;
    ThreadedBinaryTreeNode *right;
    bool isLeftNodeAThread;
    bool isRightNodeAThread;

public:
    ThreadedBinaryTreeNode()
    {
        left = NULL;
        right = NULL;
        isLeftNodeAThread = true;
        isRightNodeAThread = true;
    }
    friend class ThreadedBinaryTree;
};

class ThreadedBinaryTree
{
public:
    ThreadedBinaryTreeNode *head;
    ThreadedBinaryTreeNode *root;
    ThreadedBinaryTree()
    {
        head = new ThreadedBinaryTreeNode();
        head->left = head;
        head->right = head;
        head->isLeftNodeAThread = false;
        head->isRightNodeAThread = false;
        root = NULL;
    }

    void create()
    {
        // Allocate memory for root;
        root = new ThreadedBinaryTreeNode();

        // Assign root->leftc and rightc to head;
        root->left = head;
        root->right = head;

        // Accept root data;
        cout << "Enter the root data: ";
        cin >> root->data;

        // Assign head lbit to 0;
        head->left = root;
        // Assign head->leftc to root;
        head->isLeftNodeAThread = false;

        ThreadedBinaryTreeNode *temp, *curr;
        temp = root;
        bool flag = true;
        int again = 0;
        int choice = 0;

        cout << "Do you want to enter another node? (1 or 0)" << endl;
        cin >> again;
        while (again != 0)
```

```
68          {
69              temp = root;
70              flag = true;
71
72              cout << "Enter 1 for Entering a new node to the Left, and 2 for
    Entering it to the right" << endl;
73              cin >> choice;
74              while (flag)
75              {
76                  if (choice == 1)
77                  {
78                      if (temp->isLeftNodeAThread == true)
79                      {
80                          curr = new ThreadedBinaryTreeNode();
81                          cout << "Enter the Data: ";
82                          cin >> curr->data;
83                          curr->right = temp;
84                          temp->left = curr;
85                          temp->isLeftNodeAThread = false;
86                          flag = false;
87                      }
88                      else
89                      {
90                          temp = temp->left;
91                      }
92                  }
93                  else if (choice == 2)
94                  {
95                      if (temp->isRightNodeAThread == true)
96                      {
97                          curr = new ThreadedBinaryTreeNode();
98                          cout << "Enter the Data: ";
99                          cin >> curr->data;
100                         curr->left = temp;
101                         curr->right = temp->right;
102                         temp->right = curr;
103                         temp->isRightNodeAThread = false;
104                         flag = false;
105                     }
106                     else
107                     {
108                         temp = temp->right;
109                     }
110                 }
111             }
112             cout << "Do you want to enter another node? (1 or 0)" << endl;
113             cin >> again;
114         }
115     }
116     void inorder_traversal()
117     {
118         ThreadedBinaryTreeNode *temp;
119         temp = head;
120         while (true)
121         {
122             temp = inorder_successor(temp);
123             if (temp == head)
124                 break;
125             cout << temp->data << " ";
```

```
        }
    }
    ThreadedBinaryTreeNode *inorder_successor(ThreadedBinaryTreeNode *temp)
    {
        ThreadedBinaryTreeNode *x = temp->right;
        if (!temp->isRightNodeAThread)
        {
            while (x->isLeftNodeAThread == false)
                x = x->left;
        }
        return x;
    }

    void preorder_traversal()
    {
        ThreadedBinaryTreeNode *temp = head->left;
        while (temp != head)
        {
            cout << temp->data << " ";
            while (temp->isLeftNodeAThread == false)
            {
                temp = temp->left;
                cout << temp->data << " ";
            }
            while (temp->isRightNodeAThread == true)
            {
                temp = temp->right;
            }
            temp = temp->right;
        }
    }
};

int main()
{
    ThreadedBinaryTree tree;
    tree.create();
    cout<<"The inorder traversal is:" << endl;
    tree.inorder_traversal();
    cout << endl;
    cout << "The preorder traversal is:" << endl;
    tree.preorder_traversal();
}
```

10.2 Input and Output

```
Enter the root data: 1
Do you want to enter another node? (1 or 0)
1
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
1
Enter the Data: 3
Do you want to enter another node? (1 or 0)
1
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
1
Enter the Data: 5
Do you want to enter another node? (1 or 0)
1
```

```
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
2
Enter the Data: 69
Do you want to enter another node? (1 or 0)
1
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
1
Enter the Data: 544
Do you want to enter another node? (1 or 0)
1
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
1
Enter the Data: 245
Do you want to enter another node? (1 or 0)
1
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
2
Enter the Data: 36
Do you want to enter another node? (1 or 0)
1
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
1
Enter the Data: 41
Do you want to enter another node? (1 or 0)
1
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
1
Enter the Data: 255
Do you want to enter another node? (1 or 0)

1
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
2
Enter the Data: 64
Do you want to enter another node? (1 or 0)
1
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
1
Enter the Data: 21
Do you want to enter another node? (1 or 0)
1
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
1
Enter the Data: 47
Do you want to enter another node? (1 or 0)
1
Enter 1 for Entering a new node to the Left, and 2 for Entering it to the right
2
Enter the Data: 65
Do you want to enter another node? (1 or 0)
0
The inorder traversal is:
47 21 255 41 245 544 5 3 1 69 36 64 65
The preorder traversal is:
1 3 5 544 245 41 255 21 47 69 36 64 65
```

11 Conclusion

Thus, implemented threaded binary tree with inorder traversal.

12 FAQ

1. **Why TBT can be traversed without stack?**

   A threaded binary tree is a binary tree that has additional links between nodes to allow for traversal without the use of a stack or recursion. These extra links, called threads, connect nodes that would otherwise be null pointers in a traditional binary tree.

   There **are two types of threaded binary trees**: inorder threaded binary trees and preorder threaded binary trees.

   (a) In an inorder threaded binary tree, the threads are used to connect nodes to their inorder predecessor and inorder successor. These threads allow for a traversal of the tree in inorder without using a stack or recursion. To traverse the tree in inorder without a stack, you start at the root node and follow the leftmost thread until you reach a node without a left child. Then, you visit that node and follow the thread to its inorder successor. You repeat this process until you have visited all nodes in the tree.

   (b) In a preorder threaded binary tree, the threads are used to connect nodes to their preorder successor. These threads allow for a traversal of the tree in preorder without using a stack or recursion. To traverse the tree in preorder without a stack, you start at the root node and visit each node in the tree in preorder. When you visit a node, you follow the thread to its preorder successor and visit that node next.

   (c) In both cases, the threads provide a way to traverse the tree without using a stack or recursion, which can be useful in situations where memory usage is a concern or where recursion or stack-based algorithms are not allowed. However, constructing the threads requires additional memory and processing time, so the benefits of threaded binary trees depend on the specific use case.

2. **What are the advantages and disadvantages of TBT?**

   Threaded binary trees have several advantages and disadvantages that should be considered when deciding whether to use them in a specific scenario. Here are some of the main advantages and disadvantages:

   Advantages:

   (a) **Traversal without stack or recursion**: As mentioned before, threaded binary trees allow for traversal without using a stack or recursion. This can be useful in situations where memory usage is a concern or where recursion or stack-based algorithms are not allowed.

   (b) **Efficient Inorder and Preorder Traversal**: The use of threads can significantly improve the performance of inorder and preorder traversal. In fact, threaded binary trees can perform these traversals in O(n) time complexity, which is faster than the O(nlogn) complexity of the recursive approach.

   (c) **Space Efficiency**: Compared to traditional binary trees, threaded binary trees require less space to store the threads. This can be useful in situations where memory usage is a concern.

Disadvantages:

(a) **Construction Overhead**: The process of constructing threads can be time-consuming and requires additional memory. In some cases, the overhead of constructing the threads can negate any performance benefits.

(b) **Complexity**: Threaded binary trees can be more complex than traditional binary trees due to the additional links. This can make them harder to understand and maintain.

(c) **Limited Applications**: Threaded binary trees are not suitable for all types of problems. They are typically used in scenarios where traversal performance is critical, but in other cases, traditional binary trees or other data structures may be more appropriate.

3. **Write application of TBT**

(a) **Expression Trees**: Threaded binary trees can be used to represent arithmetic expressions in a compact form. By using threads, the expression tree can be traversed efficiently without using a stack or recursion. This can be useful in compilers and other applications that deal with arithmetic expressions.

(b) **Database Indexing**: Threaded binary trees can be used to implement indexing in databases. By using threads, the binary tree can be traversed efficiently, which can improve the performance of database queries.

(c) **Text Editors**: Threaded binary trees can be used in text editors to efficiently search for words in a document. By using threads, the binary tree can be traversed efficiently, which can improve the speed of searches.

(d) **Spell Checking**: Threaded binary trees can be used in spell checking applications to efficiently search for misspelled words. By using threads, the binary tree can be traversed efficiently, which can improve the speed of spell checking.

(e) **Image Processing**: Threaded binary trees can be used in image processing applications to efficiently process pixels in an image. By using threads, the binary tree can be traversed efficiently, which can improve the speed of image processing algorithms.

MIT World Peace University

Advanced Data Structures
Second Year B. Tech, Semester 4

Implementation of a Graph and its Depth First and Breadth First Traversals

Assignment No. 5

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

April 16, 2023

Contents

1 Objectives **1**

2 Problem Statement **1**

3 Theory **1**
- 3.1 Definition of Graph . . . 1
- 3.2 Types of Graph . . . 1
  - 3.2.1 Directed Graph . . . 1
  - 3.2.2 Undirected Graph . . . 2
  - 3.2.3 Weighted Graph . . . 2
  - 3.2.4 Bipartite Graph . . . 3

4 Platform **3**

5 Input **3**

6 Output **4**

7 Test Conditions **4**

8 Pseudo Code **4**
- 8.1 Creation of the Graph . . . 4
- 8.2 Depth First Search (Recursive) . . . 4
- 8.3 Breadth First Search . . . 5

9 Time Complexity **5**
- 9.1 Creation of Threaded Binary Tree . . . 5
- 9.2 Recursive Depth First Traversal . . . 5
- 9.3 Non Recursive Depth First Traversal . . . 5
- 9.4 Breadth First Traversal . . . 6

10 Code **6**
- 10.1 Program . . . 6
- 10.2 Input and Output . . . 9

11 Conclusion **10**

12 FAQ **11**

1 Objectives

1. *To study data structure Graph and its representation using adjacency list*
2. *To study and implement recursive Depth First Traversal and use of stack data*
3. *structure for recursive Depth First Traversal*
4. *To study and implement Breadth First Traversal*
5. *To study how graph can be used to model real world problems*

2 Problem Statement

Consider a friend's network on Facebook social web site. Model it as a graph to represent each node as a user and a link to represent the friend relationship between them using adjacency list representation and perform DFS and BFS traversals.

3 Theory

3.1 Definition of Graph

A graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as, a Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.

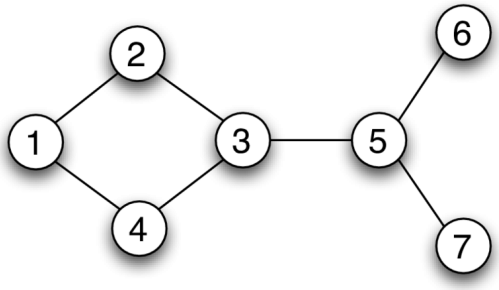


Figure 1: Graph

3.2 Types of Graph

3.2.1 Directed Graph

A directed graph, also known as a digraph, is a type of graph in which edges have a direction. It represents relationships between objects that have a cause-and-effect relationship, such as a food chain. For example, a food chain of lions, zebras, and grass can be represented as a directed graph where the edge points from zebras to lions, lions to grass, and grass to zebras.

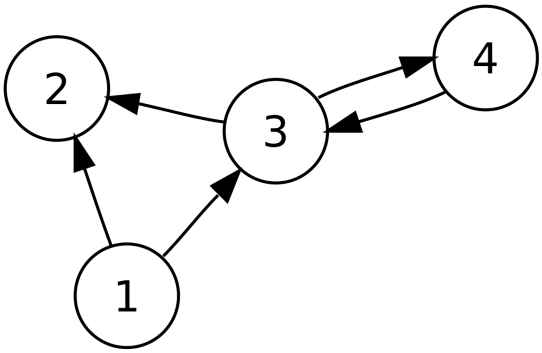


Figure 2: Directed Graph

3.2.2 Undirected Graph

An undirected graph is a type of graph in which edges have no direction. It represents relationships between objects that have a symmetric relationship, such as social networks. For example, a social network of friends can be represented as an undirected graph where each node represents a person, and an edge connects two people if they are friends.

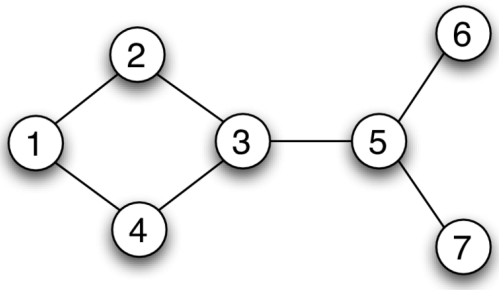


Figure 3: Undirected Graph

3.2.3 Weighted Graph

A weighted graph is a type of graph in which edges have a numerical value. It is used to represent relationships between objects where the relationship has a quantity associated with it, such as distances between cities. For example, a map of cities can be represented as a weighted graph where the nodes represent cities and the edges represent distances between them.

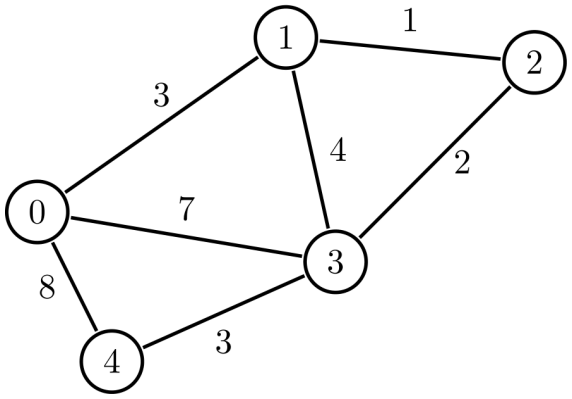


Figure 4: Weighted Graph

3.2.4 Bipartite Graph

A bipartite graph is a type of graph in which the nodes can be divided into two disjoint sets, such that each edge connects a node in one set to a node in the other set. It is used to represent relationships between two different sets of objects, such as employers and employees. For example, a company can be represented as a bipartite graph where one set of nodes represents employers and the other set represents employees, and an edge connects an employer to an employee if the employer employs the employee.

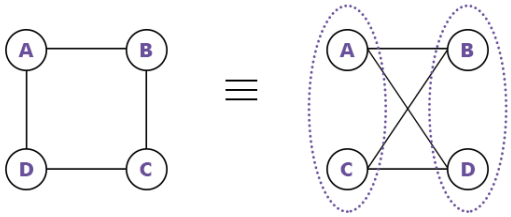


Figure 5: Bipartite Graph

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers : g++ and gcc on linux for C++

5 Input

1. Input at least 5 nodes.
2. Display DFT (recursive and non recursive) and BFT

6 Output

1. The traversal of the Threaded binary tree in different ways.

7 Test Conditions

1. Input at least 10 nodes.
2. Display all traversals of binary tree with 10 nodes.(recursive and nonrecursive)

8 Pseudo Code

8.1 Creation of the Graph

```
1
2  void create_graph()
3  {
4      int src, dst;
5      char ch;
6      do
7          cout << "Enter the source and destination" << endl;
8          cin >> src >> dst;
9          if (src >= nodes || dst >= nodes || src < 0 || dst < 0)
10             cout << "Invalid edge" << endl;
11             continue;
12         if (src == dst)
13             cout << "Invalid edge, self loops not allowed." << endl;
14             continue;
15         AddEdge(src, dst);
16         cout << "Do you want to continue" << endl;
17         cin >> ch;
18     } while (ch == 'y' || ch == 'Y');
19 }
```

8.2 Depth First Search (Recursive)

```
1  void DFS_recursive()
2  {
3      bool *visited = new bool[nodes];
4      for (int i = 0; i < nodes; i++)
5          visited[i] = false;
6      for (int i = 0; i < nodes; i++)
7          if (visited[i] == false)
8              DFS_recursive(i, visited);
9      cout << endl;
10 }
11
12 void DFS_recursive(int src, bool *visited)
13 {
14     visited[src] = true;
15     cout << src << " ";
16     for (auto &adj_node : adjlist[src])
17         if (visited[adj_node] == false)
18             DFS_recursive(adj_node, visited);
19 }
```

8.3 Breadth First Search

```
void breadth_first_traversal()
{
    bool *visited = new bool[nodes];
    for (int i = 0; i < nodes; i++)
        visited[i] = false;
    for (int i = 0; i < nodes; i++)
        if (visited[i] == false)
            breadth_first_traversal(i, visited);
    cout << endl;
}
```

9 Time Complexity

9.1 Creation of Threaded Binary Tree

- **Time Complexity:**

  $$O(V^2)$$

  for adjacency matrix representation
  where V is the number of vertices in the graph, and E is the number of edges in the graph.

- **Time Complexity:**

  $$O(E)$$

  for adjacency list representation
  where V is the number of vertices in the graph

- **Space Complexity:**

  $$O(V^2)$$

  for V being the number of vertices in the graph.

9.2 Recursive Depth First Traversal

- **Time Complexity:**

  $$O(V+E)$$

  for V being the number of vertices in the graph and E being the number of edges in the graph.

- **Space Complexity:**

  $$O(V)$$

  for V being the number of vertices in the graph.

9.3 Non Recursive Depth First Traversal

- **Time Complexity:**

  $$O(V+E)$$

  for V being the number of vertices in the graph and E being the number of edges in the graph.

- **Space Complexity:**

  $$O(V)$$

  for V being the number of vertices in the graph.

9.4 Breadth First Traversal

- **Time Complexity:**

$$O(V+E)$$

for V being the number of vertices in the graph and E being the number of edges in the graph.

- **Space Complexity:**

$$O(V)$$

for V being the number of vertices in the graph.

10 Code

10.1 Program

```
// Creating of a Network in a Graph
#include <iostream>
#include <stack>
#include <queue>
#include <list>

using namespace std;

class Graph
{

private:
    int nodes;
    list<int> *adjlist;

public:
    Graph()
    {
    }

    Graph(int nodes)
    { // Allocate resources
        adjlist = new list<int>[nodes];
        this->nodes = nodes;
    }

    ~Graph()
    { // Free allocated resources
        delete[] adjlist;
    }

    void AddEdge(int src, int dst)
    {
        adjlist[src].push_back(dst);
        adjlist[dst].push_back(src);
    }

    void Iterate(int src)
    {
        cout << src << " : ";
        for (auto &adj_node : adjlist[src])
```

```
42          {
43              cout << adj_node << " ";
44          }
45          cout << endl;
46      }
47
48      void DFS_recursive()
49      {
50          bool *visited = new bool[nodes];
51          for (int i = 0; i < nodes; i++)
52          {
53              visited[i] = false;
54          }
55          for (int i = 0; i < nodes; i++)
56          {
57              if (visited[i] == false)
58              {
59                  DFS_recursive(i, visited);
60              }
61          }
62          cout << endl;
63      }
64
65      void DFS_recursive(int src, bool *visited)
66      {
67          visited[src] = true;
68          cout << src << " ";
69          for (auto &adj_node : adjlist[src])
70          {
71              if (visited[adj_node] == false)
72              {
73                  DFS_recursive(adj_node, visited);
74              }
75          }
76      }
77
78      void breadth_first_traversal()
79      {
80          bool *visited = new bool[nodes];
81          for (int i = 0; i < nodes; i++)
82          {
83              visited[i] = false;
84          }
85          for (int i = 0; i < nodes; i++)
86          {
87              if (visited[i] == false)
88              {
89                  breadth_first_traversal(i, visited);
90              }
91          }
92          cout << endl;
93      }
94
95      void breadth_first_traversal(int src, bool *visited)
96      {
97          queue<int> q;
98          q.push(src);
99          visited[src] = true;
100         while (!q.empty())
```

```
        {
            int node = q.front();
            q.pop();
            cout << node << " ";
            for (auto &adj_node : adjlist[node])
            {
                if (visited[adj_node] == false)
                {
                    q.push(adj_node);
                    visited[adj_node] = true;
                }
            }
        }
    }

    void create_graph()
    {
        int src, dst;
        char ch;
        do
        {
            cout << "Enter the source and destination" << endl;
            cin >> src >> dst;
            if (src >= nodes || dst >= nodes || src < 0 || dst < 0)
            {
                cout << "Invalid edge" << endl;
                continue;
            }
            if (src == dst)
            {
                cout << "Invalid edge, self loops not allowed." << endl;
                continue;
            }
            AddEdge(src, dst);
            cout << "Do you want to continue" << endl;
            cin >> ch;
        } while (ch == 'y' || ch == 'Y');
    }
};

int main()
{
    Graph g(10);

    // g.AddEdge(0, 1);
    // g.AddEdge(0, 2);
    // g.AddEdge(1, 3);
    // g.AddEdge(1, 4);
    // g.AddEdge(2, 3);
    // g.AddEdge(3, 5);
    // g.AddEdge(4, 6);
    // g.AddEdge(5, 6);
    // g.AddEdge(5, 7);
    // g.AddEdge(6, 7);
    // g.AddEdge(6, 8);
    // g.AddEdge(7, 8);
    // g.AddEdge(7, 9);
    // g.AddEdge(8, 9);

```

```
    // cout << "Adjacency list implementation for graph" << endl;

    // g.Iterate(0);
    // g.Iterate(1);
    // g.Iterate(4);

    g.create_graph();

    cout << "Depth First Search Recursive" << endl;
    g.DFS_recursive();
    cout << "Breadth First Search" << endl;
    g.breadth_first_traversal();

    return 0;
}
```

10.2 Input and Output

```
Enter the source and destination
0 1
Do you want to continue
y
Enter the source and destination
0 2
Do you want to continue
y
Enter the source and destination
1 3
Do you want to continue
y
Enter the source and destination
1 4
Do you want to continue
y
Enter the source and destination
2 3
Do you want to continue
y
Enter the source and destination
3 5
Do you want to continue
y
Enter the source and destination
4 6
Do you want to continue
y
Enter the source and destination
5 6
Do you want to continue
y
Enter the source and destination
5 7
Do you want to continue
y
Enter the source and destination
6 7
Do you want to continue
y
Enter the source and destination
```

```
6 8
Do you want to continue
y
Enter the source and destination
7 8
Do you want to continue
y
Enter the source and destination
7 9
Do you want to continue
y
Enter the source and destination
8 9
Do you want to continue
n
Depth First Search Recursive
0 1 3 2 5 6 4 7 8 9
Breadth First Search
0 1 2 3 4 5 6 7 8 9
```

11 Conclusion

Thus, we have represented graph using adjacency list and performed DFT and BFT.

12 FAQ

1. **Explain two applications of graph.**

   Graphs have numerous applications in various fields such as computer science, social sciences, biology, transportation, and more. Here are two examples of applications of graphs:

   - **Social Networks**: Social networks such as Facebook, Twitter, and LinkedIn can be represented as graphs, where each user is a node, and the relationship between them (friendship, follow, connection, etc.) is an edge. Graph algorithms can be used to analyze and understand social networks, such as identifying clusters of friends, finding influential users, detecting fake accounts or spam, and predicting trends or user behavior. For example, graph analysis can help social networks to recommend new friends, suggest content to users, or optimize their algorithms to improve user engagement and retention.

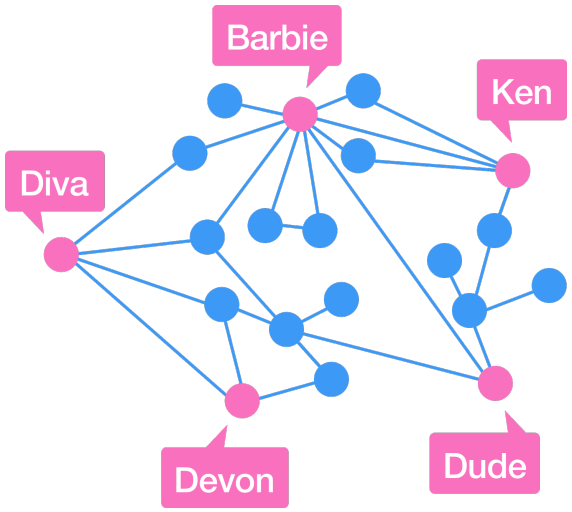


Figure 6: Graph as a Representation for Social Media

   - **Shortest Path Algorithms**: Graphs can be used to model transportation networks such as roads, railways, and flights, where each city or location is a node, and the roads or flights connecting them are edges. Shortest path algorithms such as Dijkstra's algorithm and the A* algorithm can be used to find the shortest path or the fastest route between two locations, taking into account factors such as distance, time, traffic, or cost. These algorithms are widely used in navigation systems, logistics, and transportation planning, and can help optimize the routing and scheduling of vehicles, goods, or passengers, leading to cost savings and improved efficiency.

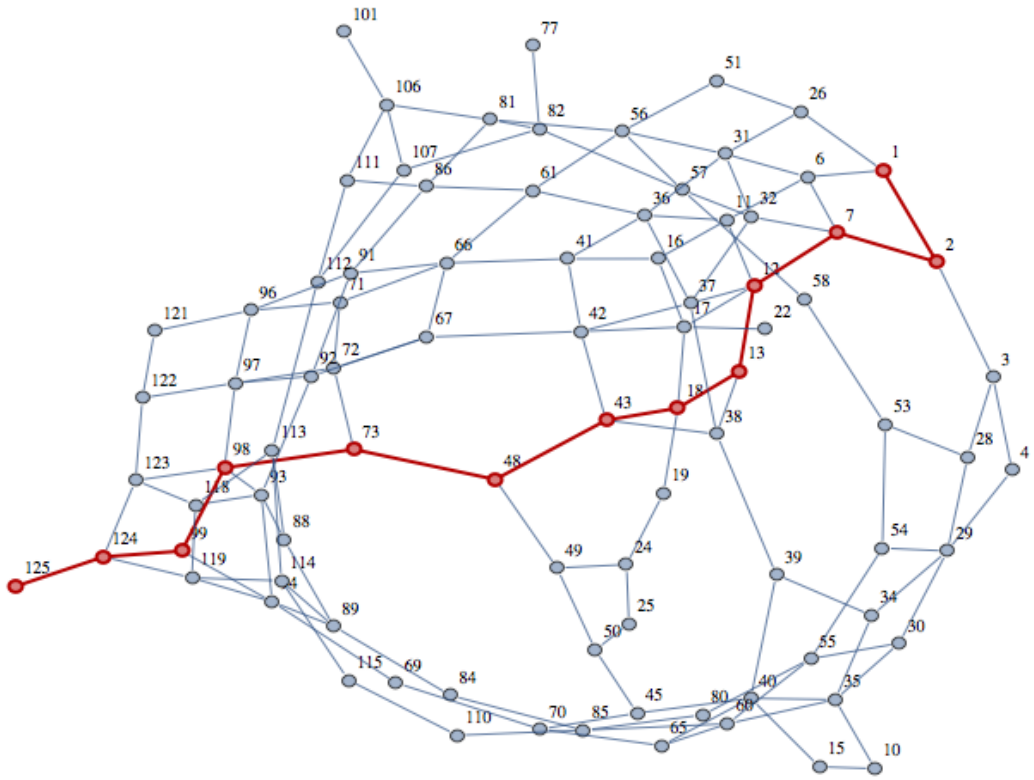


Figure 7: The Shortest Path Problem represented and solved using a Graph

2. **Explain advantages of adjacency list over adjacency matrix.**

   The adjacency list representation of a graph has the following advantages over an adjacency matrix representation:

   - **Space Efficient**: The adjacency list representation is more space efficient than the adjacency matrix representation. This is because the adjacency matrix representation of a graph with V vertices and E edges requires O(V^2) space, while the adjacency list representation requires O(V+E) space.
   - **Faster Adjacent Edge Lookups**: The adjacency list representation allows for faster lookups of adjacent edges of a vertex than the adjacency matrix representation. In the adjacency matrix representation, to find the adjacent edges of a vertex, we have to scan through the entire row of the vertex. In the adjacency list representation, we can simply traverse the linked list of the vertex to find its adjacent edges.
   - **Faster Degree Lookups**: The adjacency list representation allows for faster lookups of the degree of a vertex than the adjacency matrix representation. In the adjacency matrix representation, to find the degree of a vertex, we have to scan through the entire row of the vertex. In the adjacency list representation, we can simply traverse the linked list of the vertex to find its degree.

3. **Why transversal in graph is different than traversal in tree**

- **Graphs can have cycles**: Unlike trees, graphs can contain cycles, which means that a node can be visited multiple times through different paths. This makes traversal more complex, as we need to keep track of the visited nodes to avoid infinite loops or redundant computations. In trees, on the other hand, there are no cycles, so each node is visited exactly once during traversal.
- **Graphs can be disconnected**: Graphs can have disconnected components, which means that some nodes may not be reachable from others. This means that traversal of a graph needs to handle the possibility of having multiple disconnected components, and ensure that all components are visited. In contrast, trees are always connected, so there is no need to worry about disconnected components.
- **Graphs can be directed or undirected**: Graphs can be either directed (where edges have a direction) or undirected (where edges have no direction), which affects how traversal algorithms operate. For example, in a directed graph, traversal algorithms need to take into account the direction of edges to avoid going back to already-visited nodes, while in an undirected graph, traversal can simply use a marking mechanism to track visited nodes.

MIT WORLD PEACE UNIVERSITY

Advanced Data Structures
Second Year B. Tech, Semester 4

MINIMUM COST SPANNING TREE USING *Prim's Algorithm*

ASSIGNMENT NO. 6

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

April 16, 2023

Contents

1 Objectives **1**

2 Problem Statement **1**

3 Theory **1**
- 3.1 Spanning Tree . . . 1
  - 3.1.1 Example of Spanning Tree . . . 1
- 3.2 Weighted Graph . . . 1
  - 3.2.1 Example of Weighted Graph . . . 2
- 3.3 Cost Adjacency Matrix . . . 2
  - 3.3.1 Example of Cost Adjacency Matrix . . . 2
- 3.4 Prim's Algorithm . . . 2
  - 3.4.1 Steps . . . 2
  - 3.4.2 Example of Prim's Algorithm . . . 3

4 Platform **3**

5 Test Conditions **3**

6 Input and Output **4**

7 Pseudo Code **4**
- 7.1 Accepting the Spanning Tree using its adjacency matrix . . . 4
- 7.2 Display of the Spanning Tree using its adjacency matrix . . . 4
- 7.3 Prim's Algorithm . . . 4

8 Time Complexity **5**
- 8.1 Creation of Minimum Cost Spanning Tree using its adjacency matrix . . . 5
- 8.2 Display of Minimum Cost Spanning Tree using its adjacency matrix . . . 5
- 8.3 Prim's Algorithm . . . 6

9 Code **6**
- 9.1 Program . . . 6

10 Conclusion **8**

11 FAQ **9**

1 Objectives

1. To study data structure Graph and its representation using cost adjacency Matrix
2. To study and implement algorithm for minimum cost spanning tree
3. To study and implement Prim's Algorithm for minimum cost spanning tree
4. To study how graph can be used to model real world problems

2 Problem Statement

A business house has several offices in different countries; they want to lease phone lines to connect them with each other and the phone company charges different rent to connect different pairs of cities. Business house wants to connect all its offices with a minimum total cost. Solve the problem using Prim's algorithm.

3 Theory

3.1 Spanning Tree

A spanning tree of a graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

3.1.1 Example of Spanning Tree

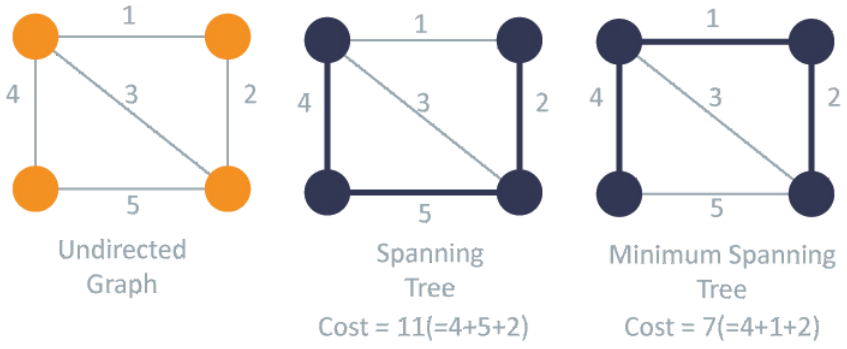


Figure 1: Example of Spanning Tree

3.2 Weighted Graph

A weighted graph is a graph in which each edge is assigned a weight or cost. The weight of an edge is a non-negative real number. The weight of a path is the sum of the weights of its constituent edges. The weight of a cycle is the sum of the weights of its constituent edges. The weight of a graph is the sum of the weights of its edges.

3.2.1 Example of Weighted Graph

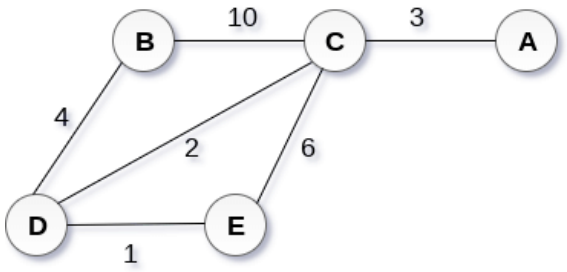


Figure 2: Example of Weighted Graph

3.3 Cost Adjacency Matrix

A cost adjacency matrix is a square matrix that represents the weighted edges of a graph. The matrix is symmetric about the main diagonal. The main diagonal of the matrix is all zeros. The matrix is filled with the weights of the edges. If there is no edge between two vertices, the corresponding entry in the matrix is zero.

3.3.1 Example of Cost Adjacency Matrix

	A	**B**	**C**	**D**	**E**
A	0	-1	3	-1	-1
B	-1	0	10	4	-1
C	3	10	0	2	6
D	-1	4	2	0	1
E	-1	-1	6	1	0

Table 1: Adjacency Matrix of the Graph

3.4 Prim's Algorithm

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

3.4.1 Steps

1. Choose a starting vertex B.
2. Add the vertices that are adjacent to A. the edges that connecting the vertices are shown by dotted lines.
3. Choose the edge with the minimum weight among all. i.e. BD and add it to MST. Add the adjacent vertices of D i.e. C and E.

4. Choose the edge with the minimum weight among all. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C i.e. E and A.

5. Choose the edge with the minimum weight i.e. CA. We can't choose CE as it would cause cycle in the graph.

3.4.2 Example of Prim's Algorithm

This example is based on the above example of weighted graph and its cost adjacency matrix.

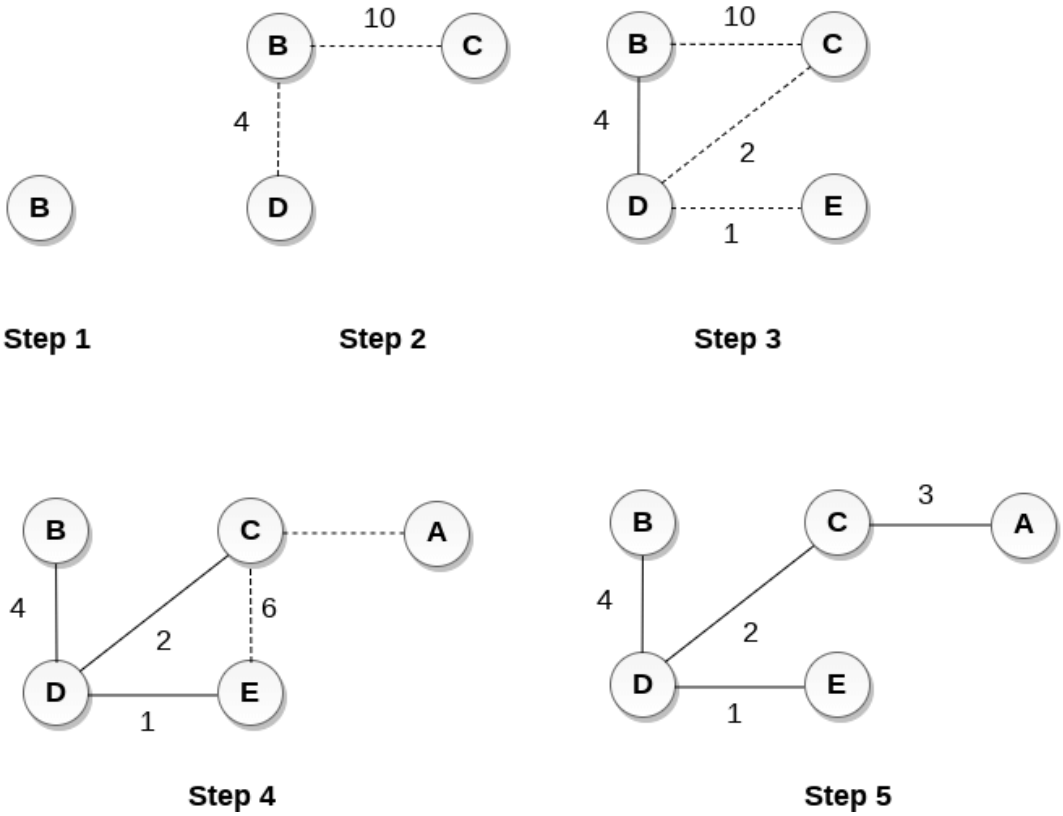


Figure 3: Example of Prim's Algorithm

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers : g++ and gcc on linux for C++

5 Test Conditions

1. Input at least 5 nodes.

2. Display minimum cost spanning tree and minimum cost for its graph.

6 Input and Output

1. The minimum cost of the spanning tree.

7 Pseudo Code

7.1 Accepting the Spanning Tree using its adjacency matrix

```
    // input the adjacency matrix
    cout << "Enter the adjacency matrix: " << endl;

    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            cin >> graph[i][j];
```

7.2 Display of the Spanning Tree using its adjacency matrix

```
// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " \t"
             << graph[i][parent[i]] << " \n";
```

7.3 Prim's Algorithm

```

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
    // Initialize min value
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;
    return min_index;

void primMST(int graph[V][V])
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
```

```
    // Make key 0 so that this vertex is picked as first
    // vertex.
    key[0] = 0;

    // First node is always root of MST
    parent[0] = -1;

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)
            // graph[u][v] is non zero only for adjacent
            // vertices of m mstSet[v] is false for vertices
            // not yet included in MST Update the key only
            // if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];

    // Print the constructed MST
    printMST(parent, graph);
```

8 Time Complexity

Let V be the number of vertices in the graph and E be the number of edges in the graph, then:

8.1 Creation of Minimum Cost Spanning Tree using its adjacency matrix

- **Time Complexity:**

$$O(V^2)$$

- **Space Complexity:**

$$O(V^2)$$

8.2 Display of Minimum Cost Spanning Tree using its adjacency matrix

- **Time Complexity:**

$$O(V^2)$$

- **Space Complexity:**

$$O(V^2)$$

8.3 Prim's Algorithm

- **Time Complexity:**

$$O(V^2 \log V)$$

: We maintain a priority queue, and we also maintain a matrix of size VxV. The algorithm runs in logV time, for each edge. There can be VxV edges in the graph. Hence the time complexity.

- **Space Complexity:**

$$O(V^2)$$

: For storing the graph in the form of an adjacency matrix.

9 Code

9.1 Program

```
// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph

#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " \t"
             << graph[i][parent[i]] << " \n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
```

```
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first
    // vertex.
    key[0] = 0;

    // First node is always root of MST
    parent[0] = -1;

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
    {

        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent
            // vertices of m mstSet[v] is false for vertices
            // not yet included in MST Update the key only
            // if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // Print the constructed MST
    printMST(parent, graph);
}

// Driver's code
int main()
{
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0}};

```

```
    // input the adjacency matrix
    cout << "Enter the adjacency matrix: " << endl;

    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            cin >> graph[i][j];
        }
    }

    cout << " The minimum spanning tree is: " << endl;
    // Print the solution
    primMST(graph);

    return 0;
}

// This code is contributed by rathbhupendra
```

```
Input Adjacency Matrix:
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0
 The minimum spanning tree is:
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

10 Conclusion

Thus, we have represented graph using cost adjacency matrix and implemented Prim's algorithm for MCST.

11 FAQ

1. *Explain two applications of minimum cost spanning tree.*

   Minimum cost spanning trees have a wide range of applications in computer science, operations research, and engineering. Here are two common applications of minimum cost spanning trees:

   (a) **Network design and optimization**:
   Minimum cost spanning trees are often used to design and optimize communication networks, such as the internet or a telephone network.

   For example, a minimum cost spanning tree can be used to find the most cost-effective way to connect a set of cities with a fiber-optic network, or to design a network of sensors to monitor a given area with minimum cost.

   By minimizing the cost of connecting nodes while ensuring that every node is reachable from every other node, minimum cost spanning trees can help improve network performance and reduce operating costs.

   (b) **Circuit design**:
   In circuit design, minimum cost spanning trees are often used to solve the problem of routing wires between electronic components on a printed circuit board.

   By constructing a minimum cost spanning tree that connects all the components, engineers can minimize the length and cost of the wires needed to connect the components, while ensuring that every component is connected to every other component through a path of wires.

   This can lead to more efficient and reliable circuit designs with lower production costs.

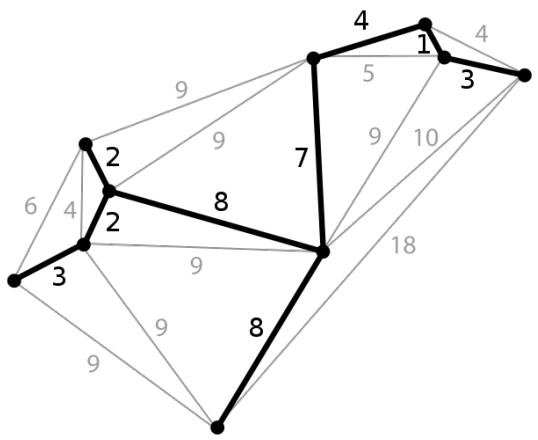


Figure 4: Minimum cost spanning tree for a network

2. *Explain the difference between Prim's and Kruskal's Algorithm for finding minimum cost spanning tree with a small example graph.*

The Main differences between the Two Algorithms are:

(a) **Approach:** Prim's algorithm follows a *greedy* approach where it starts from an arbitrary vertex and builds the tree by adding the next vertex with the lowest cost edge. On the other hand, Kruskal's algorithm is also a greedy approach, but it starts with the minimum edge and adds the next minimum edge which does not create a cycle.

(b) **Data structure:** Prim's algorithm uses a *priority queue* to maintain the vertices with their minimum distance values, while Kruskal's algorithm uses a *disjoint set data structure* to keep track of the connected components.

(c) **Complexity:** Prim's algorithm has a time complexity of $O(E \log V)$ for implementing with a priority queue, while Kruskal's algorithm has a time complexity of $O(E \log E)$ for implementing with a disjoint set data structure.

(d) **Connectivity:** Prim's algorithm generates a single tree rooted at the selected starting vertex, while Kruskal's algorithm generates multiple trees (i.e., forest) and later combines them into a single tree.

(e) **Graph type:** Prim's algorithm is more suitable for dense graphs (i.e., where E is close to V^2), while Kruskal's algorithm works better for sparse graphs (i.e., where E is much smaller than V^2).

(f) **Application:** Prim's algorithm is often used in network routing protocols, where finding the shortest path between two points is essential. On the other hand, Kruskal's algorithm is useful in applications like clustering, where finding the minimum cost connected subgraphs is required.

Prim's Algorithm	**Kruskal's Algorithm**
The tree that we are making or growing always remains connected.	The tree that we are making or growing usually remains disconnected.
Prim's Algorithm grows a solution from a random vertex adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the tree / forest.
Prim's Algorithm is faster for dense graphs.	Kruskal's Algorithm is faster for sparse graphs.

Example:
Consider the following graph:

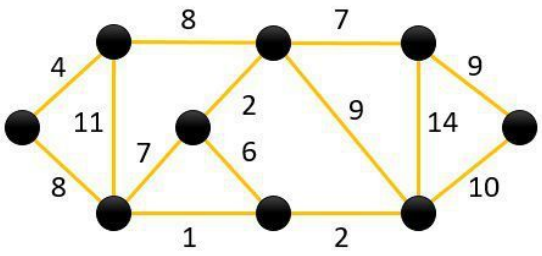


Figure 5: Example graph

Prim's solution:

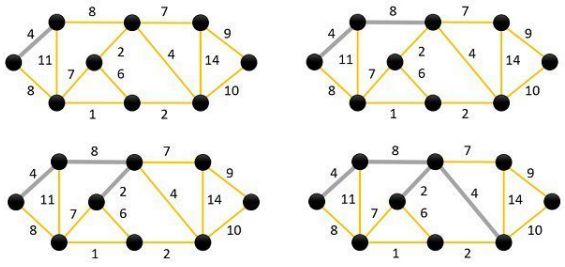


Figure 6: Prim's solution

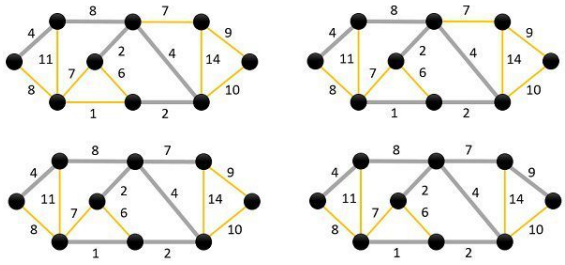


Figure 7: Prim's solution

Kruskal's solution:

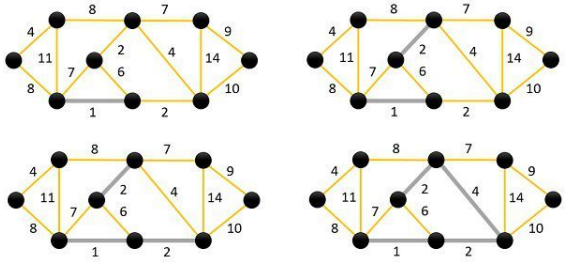


Figure 8: Kruskal's solution

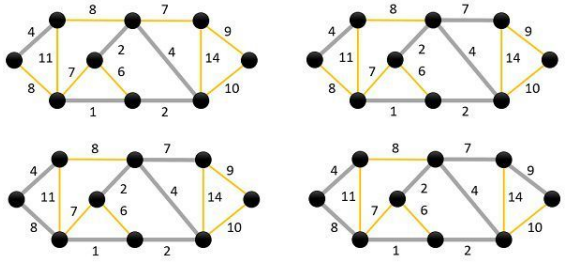


Figure 9: Kruskal's solution

3. *Which algorithmic strategy is used in Prim's algorithm. Explain that algorithmic strategy in brief.*

- Prim's algorithm for finding the minimum cost spanning tree uses a **Greedy approach**, which means that it makes the locally optimal choice at each step with the hope of finding a globally optimal solution.
- The algorithm starts with a single vertex and keeps adding edges to the tree one by one, such that the tree grows gradually and remains connected at all times. At each step, the algorithm selects the edge with the minimum weight that connects a vertex in the tree to a vertex outside the tree. This process continues until all the vertices are included in the tree.
- The key idea behind the algorithm is to start with a small tree and gradually add vertices to it in a way that minimizes the total cost of the tree. The algorithm achieves this by always selecting the edge with the lowest weight that connects a vertex in the tree to a vertex outside the tree. By following this approach, the algorithm ensures that the tree is connected and has the minimum possible cost.
- Overall, Prim's algorithm is an efficient and widely used algorithm for finding the minimum cost spanning tree of a graph, especially when the graph is dense (i.e., has many edges).

MIT World Peace University

Advanced Data Structures
Second Year B. Tech, Semester 4

Implementation of Heap as a Data Structure

Assignment No. 7

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

April 26, 2023

Contents

1 Objectives **1**

2 Problem Statement **1**

3 Theory **1**
- 3.1 Heap . . . 1
- 3.2 Types of Heaps . . . 2
- 3.3 Construction of heaps . . . 2
- 3.4 Data Structures Used for Heap Constructions . . . 3
- 3.5 Time and Space Complexities Associated with Heap . . . 3
- 3.6 Heap Vs Binary Search Trees . . . 3
- 3.7 Applications of Heap . . . 3

4 Platform **3**

5 Test Conditions **4**

6 Input and Output **4**

7 Pseudo Code **4**

8 Time Complexity **5**
- 8.1 Min and Max Heap Creation . . . 5
- 8.2 Min or Max Heap Traversal . . . 5
- 8.3 Heap Sort . . . 5

9 Searching in Heap **5**

10 Code **5**
- 10.1 Program . . . 5

11 Conclusion **7**

12 FAQ **8**

1 Objectives

1. To study the concept of heap
2. To study different types of heap and their algorithms

2 Problem Statement

Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in that subject. Use heap data structure and Heap sort.

3 Theory

3.1 Heap

A heap is a specialized tree-based data structure that satisfies the heap property. The heap property is a condition where each node in the tree is greater than or equal to (in a max heap) or less than or equal to (in a min heap) its children. The root node of the heap is the maximum (or minimum) element in a max (or min) heap.

Heaps are often used to implement priority queues, where elements are extracted in order of priority. For example, in a hospital, patients with the most urgent medical needs are given the highest priority for treatment. A priority queue based on a heap can efficiently manage the order of patients by storing their priority level (e.g., critical, urgent, or routine) in each node and maintaining the heap property.

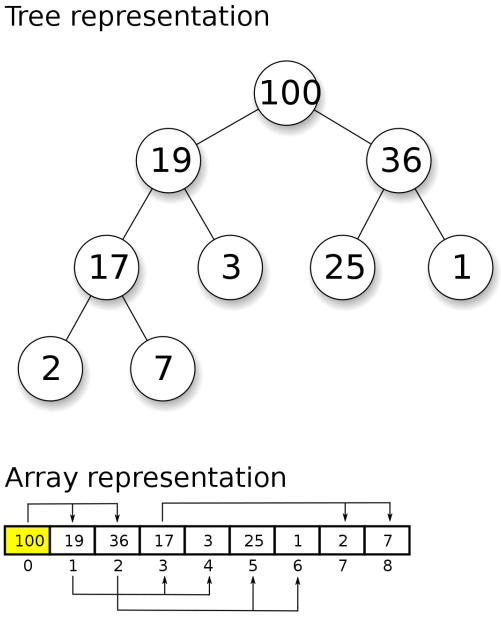


Figure 1:

3.2 Types of Heaps

There are two main types of heaps: max heaps and min heaps. In a max heap, the maximum element is always stored at the root, and every node is greater than or equal to its children. In a min heap, the minimum element is stored at the root, and every node is less than or equal to its children.

Both types of heaps have their own advantages and use cases. Max heaps are often used to implement priority queues, where the highest priority element needs to be extracted first. Min heaps are often used in algorithms such as Dijkstra's shortest path algorithm, where the minimum distance to a vertex needs to be determined.

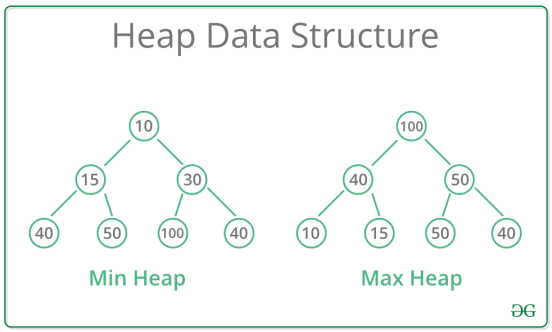


Figure 2: Min Heap and Max Heap

3.3 Construction of heaps

Heaps can be constructed using a variety of algorithms, including the bottom-up construction algorithm and the top-down construction algorithm. The bottom-up construction algorithm starts with a partially ordered set of elements and iteratively adds elements to the heap in a way that maintains the heap property. The top-down construction algorithm starts with an empty heap and iteratively adds elements to the heap in a way that maintains the heap property.

Regardless of the algorithm used, the construction of a heap takes O(n) time, where n is the number of elements in the heap. This is because each element must be inserted into the heap and the heap property must be maintained at each step.

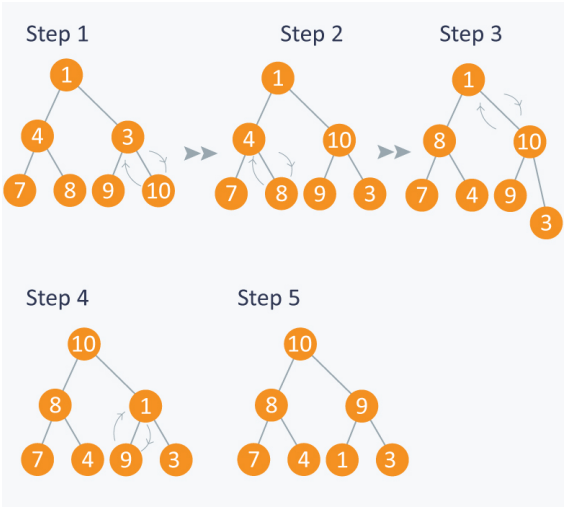


Figure 3:

3.4 Data Structures Used for Heap Constructions

The most common data structure used for heap construction is an array. In an array-based heap, the elements are stored in an array in a way that maintains the heap property. The root element is stored at index 0, and the children of a node at index i are stored at indices 2i+1 and 2i+2.

Linked lists can also be used to implement heaps, but they are less commonly used due to their higher overhead.

3.5 Time and Space Complexities Associated with Heap

The time complexity of heap operations depends on the height of the heap, which is O(log n) for a heap with n elements. The space complexity of a heap is O(n), where n is the number of elements in the heap.

Inserting an element into a heap takes O(log n) time, as the element must be inserted at the bottom of the heap and then sifted up to maintain the heap property. Similarly, extracting the maximum (or minimum) element from a heap takes O(log n) time, as the root element must be removed and then the heap must be reorganized to maintain the heap property.

Heap operations are generally more efficient than operations on other data structures such as arrays or linked lists, particularly for large datasets. However, they can be less efficient than other data structures for small datasets due to the overhead associated with maintaining the heap property.

3.6 Heap Vs Binary Search Trees

Heaps are often compared to binary search trees (BSTs), another tree-based data structure. However, heaps are not as efficient as BSTs for all operations. For example, searching for an element in a heap takes O(n) time, as each element must be searched in the worst case. In contrast, searching for an element in a BST takes O(log n) time, as the search can be narrowed down to a single subtree.

However, heaps are more efficient than BSTs for operations such as finding the maximum (or minimum) element, as the root element is always the maximum (or minimum) element in a heap. In contrast, finding the maximum (or minimum) element in a BST takes O(log n) time, as the maximum (or minimum) element may be located at the bottom of the tree.

3.7 Applications of Heap

Heaps are often used to implement priority queues, where elements are extracted in order of priority. For example, in a hospital, patients with the most urgent medical needs are given the highest priority for treatment. A priority queue based on a heap can efficiently manage the order of patients by storing their priority level (e.g., critical, urgent, or routine) in each node and maintaining the heap property.

Heaps are also used in algorithms such as Dijkstra's shortest path algorithm, where the minimum distance to a vertex needs to be determined. In this case, the heap stores the vertices that have not yet been visited, and the minimum distance to each vertex is stored in each node. The heap property is maintained by updating the minimum distance to each vertex as the algorithm progresses.

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code

Compilers : g++ and gcc on linux for C++

5 Test Conditions

1. Input min 10 elements.
2. Display Max and Min Heap
3. Find Maximum and Minimum marks obtained in a particular subject.

6 Input and Output

1. The minimum cost of the spanning tree.

7 Pseudo Code

Pseudo Code for Creation of Min and Max Heap

```
CreateMaxHeap(array)
    for i from n/2 down to 1 do:
        MaxHeapify(array, i, n)

CreateMinHeap(array)
    for i from n/2 down to 1 do:
        MinHeapify(array, i, n)
```

Pseudo Code for Heapify function

```
heapify(array, size)
Begin
   for i := 1 to size do
      node := i
      par := floor (node / 2)
      while par >= 1 do
         if array[par] < array[node] then
            swap array[par] with array[node]
         node := par
         par := floor (node / 2)
      done
   done
End
```

```
Heapify(array, size)
Begin
   for i := n to 1 decrease by 1 do
      heapify(array, i)
      swap array[1] with array[i]
   done
End
```

8 Time Complexity

8.1 Min and Max Heap Creation

- **Time Complexity:**

$$O(n \log(n))$$

- **Space Complexity:**

$$O(n)$$

8.2 Min or Max Heap Traversal

- **Time Complexity:**

$$O(n \log(n))$$

- **Space Complexity:**

$$O(n)$$

8.3 Heap Sort

- **Time Complexity:**

$$O(n \log(n))$$

- **Space Complexity:**

$$O(n)$$

9 Searching in Heap

- **Time Complexity:**

$$O(n)$$

10 Code

10.1 Program

```
// program for heap implementation using array
// Author: Krishnaraj Thadesar
// ADS Assignment 7
// Heap Sort and Heapify


#include <iostream>
using namespace std;

class Heap
{
private:
    int *arr;
    int size;

    void heapify(int i)
    {
```

```
18          int largest = i;
19          int left = 2 * i + 1;
20          int right = 2 * i + 2;
21
22          if (left < size && arr[left] > arr[largest])
23          {
24              largest = left;
25          }
26
27          if (right < size && arr[right] > arr[largest])
28          {
29              largest = right;
30          }
31
32          if (largest != i)
33          {
34              swap(arr[i], arr[largest]);
35              heapify(largest);
36          }
37      }
38
39  public:
40      Heap(int *arr, int n)
41      {
42          this->arr = arr;
43          this->size = n;
44
45          // build max heap
46          for (int i = n / 2 - 1; i >= 0; i--)
47          {
48              heapify(i);
49          }
50      }
51
52      void heapSort()
53      {
54          int size = this->size;
55          // sort the array using heap sort
56          for (int i = size - 1; i >= 0; i--)
57          {
58              swap(arr[0], arr[i]);
59              size--;
60              heapify(0);
61          }
62      }
63      void print()
64      {
65          for (int i = 0; i < size; i++)
66          {
67              cout << arr[i] << " ";
68          }
69      }
70
71      int getmax()
72      {
73          return arr[0];
74      }
75  };
76
```

```
int main()
{
    int no_of_marks;
    cout << "Welcome to ADS Assignment 7" << endl;
    cout << "Enter the number of marks: ";
    cin >> no_of_marks;

    int marks[no_of_marks];
    cout << "Enter the marks: ";
    for (int i = 0; i < no_of_marks; i++)
    {
        cin >> marks[i];
    }

    Heap MarksHeap(marks, no_of_marks);

    cout << "The maximum marks are: " << MarksHeap.getmax() << endl;

    cout << "The sorted marks are: " << endl;
    MarksHeap.heapSort();

    MarksHeap.print();

    return 0;
}
```

```
Welcome to ADS Assignment 7
Enter the number of marks: 12
Enter the marks: 3
2
5
8
14
9
10
12
7
7
8
3
The maximum marks are: 14

The sorted marks are:

14 12 8 10 5 7 8 7 9 3 3 2
```

11 Conclusion

Thus, we have understood the importance and use of Heaps as a Data structure, and how they are better and more efficient than Binary Search Trees. We have also understood the working of Heap Sort and how it is implemented.

12 FAQ

1. **Discuss with suitable example for heap sort?**
   Heap sort is a comparison-based sorting algorithm that works by first organizing the data to be sorted into a binary heap. The heap is then repeatedly reduced to a sorted array by extracting the largest element from the heap and inserting it into the output array. The heap is reconstructed after each extraction.

   An example of heap sort can be shown using the following array of numbers:

   [12, 11, 13, 5, 6, 7]

   First, we build a max heap from the given array. The max heap is a binary tree where the parent node is greater than or equal to its children. After building the max heap, the array becomes:

   [13, 11, 12, 5, 6, 7]

   The first element of the array is the largest number in the heap, so we move it to the end of the array and reduce the heap size by one. The array now becomes:

   [7, 11, 12, 5, 6, 13]

   We then rebuild the max heap from the remaining elements and repeat the process until the heap is empty. The sorted array is obtained by repeatedly extracting the maximum element from the heap and appending it to the output array. The final sorted array is:

   [5, 6, 7, 11, 12, 13]

2. **Compute the time complexity of heap sort?**
   The time complexity of heap sort is O(n log n) in the worst case, where n is the number of elements to be sorted. This is because the max heap can be built in O(n) time and each extraction from the heap takes O(log n) time. Therefore, the total time complexity of heap sort is O(n log n). Heap sort is efficient for large datasets and is a good choice when a stable sort is not required.

MIT World Peace University

Advanced Data Structures
Second Year B. Tech, Semester 4

Linear Probing with and without Replacement for Hashing

Assignment No. 8

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

April 15, 2023

Contents

1 Objectives **1**

2 Problem Statement **1**

3 Theory **1**
3.1 What is Hashing? 1
3.1.1 Hashing in Comparison with other Searching Techniques 1
3.2 Hash Functions 2
3.2.1 Hash Function 2
3.3 Collision Resolution Techniques 2
3.3.1 Without Replacement 3
3.3.2 With Replacement 3

4 Platform **4**

5 Test Conditions **4**

6 Input and Output **4**

7 Pseudo Code **4**
7.1 Linear Probing with Replacement 4
7.2 Linear Probing without Replacement 4

8 Time Complexity **5**
8.1 Linear Probing 5

9 Code **5**
9.1 Program 5

10 Conclusion **15**

11 FAQ **16**

1 Objectives

1. To study hashing techniques
2. To implement different hashing techniques
3. To study and implement linear probing with and without replacement
4. To study how hashing can be used to model real world problems

2 Problem Statement

Implement Direct access file using hashing (linear probing with and without replacement) perform following operations on it:

1. Create Database
2. Display Database
3. Add a record
4. Search a record
5. Modify a record

3 Theory

3.1 What is Hashing?

Hashing is a technique to convert a range of key values into a range of indexes of an array. The idea is to use the key itself as an index into an array that is why it is also called as direct access table.

3.1.1 Hashing in Comparison with other Searching Techniques

- **Sequential Search**:
  - The worst case time complexity of the sequential search is O(n)
  - The worst case time complexity of the hashing is O(1)
  - The sequential search is not suitable for large data sets
  - The hashing is suitable for large data sets
- **Binary Search**:
  - The worst case time complexity of the binary search is O(log n)
  - The worst case time complexity of the hashing is O(1)
  - The binary search is only suitable for sorted data sets
  - The hashing is suitable for unsorted data sets
- **Binary Tree**:

- The worst case time complexity of the binary tree search is O(log n)
- The worst case time complexity of the hashing is O(1)
- The binary tree search is only suitable for sorted data sets
- The hashing is suitable for unsorted data sets

3.2 Hash Functions

3.2.1 Hash Function

A hash function is a function that maps a given key to a location in the hash table. The hash function is used to calculate the index of the array where the data is to be stored or retrieved from.

Different types of Hash functions are:

1. **Division Method**

   The division method is one of the simplest hashing methods. It works by computing the remainder of the key when divided by the table size, using a hash function of the form $h(key) = key \mod table_size$. The result is the index of the slot in the hash table where the key-value pair should be stored.

2. **Multiplication Method**

   The multiplication method is another common hashing method. It works by multiplying the key by a constant A in the range $(0, 1)$ and then extracting the fractional part of the product. The result is then multiplied by the table size and rounded down to obtain the index of the slot in the hash table where the key-value pair should be stored. The hash function has the form $h(key) = \lfloor table_size * (key * A \mod 1) \rfloor$.

3. **Universal Hashing**

   Universal hashing is a family of hashing methods that use a randomly chosen hash function from a set of functions to minimize collisions. The set of functions is chosen to be large enough that the probability of two different keys having the same hash value is small, and the function is chosen randomly each time a new hash table is created. The hash function has the form $h(key) = ((a * key + b) \mod p) \mod table_size$, where a and b are randomly chosen integers and p is a large prime number.

4. **Perfect Hashing**

   Perfect hashing is a technique that is used when the keys are known in advance and fixed. It works by creating a hash function that maps each key to a unique index in the hash table, without any collisions. This is achieved by using two levels of hashing: the first level maps the keys to a set of buckets, and the second level uses a different hash function to map each key within a bucket to a unique index in the hash table. This approach guarantees that there are no collisions, but requires more memory and computation than other hashing methods.

3.3 Collision Resolution Techniques

The main types of Collision Resolution techniques are:

- **Open Addressing**:
  Open addressing is a family of hashing methods that use the hash table itself to resolve collisions, by storing each key-value pair in the next available slot in the table. There are several methods of open addressing, including:

  - **Linear Probing**
    Linear probing is an open addressing method where, when a collision occurs, the algorithm searches for the next available slot in the table, by linearly checking each slot in the table until an empty slot is found. The hash function has the form $h(key, i) = (h'(key) + i) \mod table\_size$, where $h'(key)$ is the primary hash function and i is the number of the probe.
  - **Quadratic Probing**
    Quadratic probing is similar to linear probing, but uses a quadratic function to search for the next available slot. The hash function has the form $h(key, i) = (h'(key) + c_1 \cdot i + c_2 \cdot i^2) \mod table\_size$, where c_1 and c_2 are constants that depend on the hash table size.
  - **Double Hashing**
    Double hashing is another open addressing method that uses two hash functions to determine the next available slot in the table. The hash function has the form $h(key, i) = (h_1(key) + i \cdot h_2(key)) \mod table\_size$, where h_1 and h_2 are two different hash functions.

- **Separate Chaining**

  Separate chaining is a hashing method that uses linked lists to store the key-value pairs that hash to the same slot in the table. When a collision occurs, the key-value pair is added to the linked list at the appropriate slot. The hash function has the form $h(key) = key \mod table\_size$.

- **Double Hashing**

  Double hashing is both an open addressing method and a separate chaining method. It uses two hash functions to determine the slot in the table, and if there is a collision, it uses a linked list to store the key-value pairs that hash to the same slot. The hash function has the form $h(key, i) = (h_1(key) + i \cdot h_2(key)) \mod table\_size$, where h_1 and h_2 are two different hash functions.

Most of these techniques can be implemented in 2 ways

3.3.1 Without Replacement

[Without Replacement] In this technique, when a collision occurs, the new element is simply discarded.

3.3.2 With Replacement

[With Replacement] In this technique, when a collision occurs, the old element is replaced by the new element.

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers : g++ and gcc on linux for C++

5 Test Conditions

1. Input at least 10 nodes.
2. Display collision with replacement and without replacement.

6 Input and Output

1. The minimum cost of the spanning tree.

7 Pseudo Code

7.1 Linear Probing with Replacement

```
void HashTable::insert_without_replacement(int key)
{
    int index = hash(key);
    if (table[index] == -1)
        table[index] = key;
    else
    {
        int i = 1;
        while (table[(index + i) % SIZE] != -1)
            i++;
        table[(index + i) % SIZE] = key;
    }
}
```

7.2 Linear Probing without Replacement

```
void HashTable::insert_with_replacement(int key)
    {
        int index = hash(key);

        // there is no value there.
        if (table[index] == -1)
            table[index] = key;
        // the value that is already there, belongs there, then check, and then
    find another empty slot and insert there.
        else if (hash(table[index]) == index)
        {
            int i = 1;
            while (table[(index + i) % SIZE] != -1)
                i++;
            table[(index + i) % SIZE] = key;
```

```
        }
        // the value that is already there, does not belong there, then replace it with the new value, and push the existing value down.
        else
        {
            // find empty slot
            int i = 1;
            while (table[(index + i) % SIZE] != -1)
                i++;

            int temp = table[index]; // current value that doesnt belong there.
            table[index] = key;
            table[(index + i) % SIZE] = temp;
        }
    }
```

8 Time Complexity

8.1 Linear Probing

- **Time Complexity:**

$$O(n)$$

It would be 1 if the it was the best case, and n for worst case, where all the slots would be filled, and we would have to keep probing over all the elements.

- **Space Complexity:**

$$O(n)$$

For storing the Table, as there is no additional array or table required to store and hash.

9 Code

9.1 Program

```
// Program for linear probing with and without replacement.

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

const int SIZE = 10;

class Employee
{

public:
    string name;
    int id;
    int age;
    Employee()
    {
        name = "";
        id = 0;
        age = 0;
```

```
21      }
22      Employee(string name, int id, int age)
23      {
24          this->name = name;
25          this->id = id;
26          this->age = age;
27      }
28      void accept_data()
29      {
30          cout << "Enter name: " << endl;
31          cin >> name;
32          cout << "Enter id: " << endl;
33          cin >> id;
34          cout << "Enter age: " << endl;
35          cin >> age;
36      }
37
38      void print_data()
39      {
40          cout << "Name: " << name << endl;
41          cout << "Id: " << id << endl;
42          cout << "Age: " << age << endl;
43      }
44  };
45
46  class HashTable
47  {
48  public:
49      HashTable();
50      int insert_without_replacement(int key);
51      int insert_with_replacement(int key);
52      void print();
53      int table[SIZE];
54      void read_data_from_file()
55      {
56          // read the records from the file
57          cout << "Reading data from the file. " << endl;
58          Employee temp;
59          fstream file("data.txt", ios::binary | ios::in);
60          for (int i = 0; i < SIZE; i++)
61          {
62              file.read((char *)&temp, sizeof(Employee));
63              cout << "Employee " << i + 1 << ": " << endl;
64              temp.print_data();
65              cout << endl;
66          }
67          // file.close();
68      }
69
70      void write_data_to_file(Employee hashed_employees[])
71      {
72          // Create a file in binary write mode.
73          fstream file("data.txt", ios::binary | ios::out);
74
75          // Write all the employee objects into the file in the order of the hash
    employee array.
76          for (int i = 0; i < SIZE; i++)
77          {
78              cout << "Writing employee " << i + 1 << endl;
```

```
            cout << "Employee " << i + 1 << ": " << endl;
            hashed_employees[i].print_data();
            file.write((char *)&hashed_employees[i], sizeof(Employee));
        }
        // Close the file.
        file.close();
    }

private:
    int hash(int key);
};

HashTable::HashTable()
{
    for (int i = 0; i < SIZE; i++)
        table[i] = -1;
}

int HashTable::hash(int key)
{
    return key % SIZE;
}

int HashTable::insert_without_replacement(int key)
{
    int index = hash(key);
    if (table[index] == -1)
        table[index] = key;
    else
    {
        int i = 1;
        while (table[(index + i) % SIZE] != -1)
        {
            i++;
            if (i == SIZE)
            {
                cout << "Table is full. " << endl;
                return 1;
            }
        }
        table[(index + i) % SIZE] = key;
    }
    return 0;
}

int HashTable::insert_with_replacement(int key)
{
    int index = hash(key);

    // there is no value there.
    if (table[index] == -1)
        table[index] = key;
    // the value that is already there, belongs there, then check, and then find another empty slot and insert there.
    else if (hash(table[index]) == index)
    {
        int i = 1;
        while (table[(index + i) % SIZE] != -1)
        {
```

```
137             i++;
138             if (i == SIZE)
139             {
140                 cout << "Table is full. " << endl;
141                 return 1;
142             }
143         }
144         table[(index + i) % SIZE] = key;
145     }
146     // the value that is already there, does not belong there, then replace it
    with the new value, and push the existing value down.
147     else
148     {
149         // find empty slot
150         int i = 1;
151         while (table[(index + i) % SIZE] != -1)
152         {
153             i++;
154             if (i == SIZE)
155             {
156                 cout << "Table is full. " << endl;
157                 return 1;
158             }
159         }
160 
161         int temp = table[index]; // current value that doesnt belong there.
162         table[index] = key;
163         table[(index + i) % SIZE] = temp;
164     }
165     return 0;
166 }
167 
168 void HashTable::print()
169 {
170     for (int i = 0; i < SIZE; i++)
171         cout << i << ":" << table[i] << " " << endl;
172     cout << endl;
173 }
174 
175 int main()
176 {
177     int size;
178     cout << "Enter size of table: " << endl;
179     cin >> size;
180     Employee employees[size];
181     Employee hashed_employees[SIZE];
182     HashTable hash_table;
183     // Accept data for all employees
184     for (int i = 0; i < size; i++)
185     {
186         cout << "Enter data for employee " << i + 1 << endl;
187         employees[i].accept_data();
188         cout << endl
189              << endl;
190         cout << "Employee " << i + 1 << ": " << endl;
191         employees[i].print_data();
192         cout << endl;
193     }
194 
```

```
    // Provide choice to users if they wanna insert with or without replacement
    int choice;
    cout << "Enter 1 to insert with replacement, 2 to insert without replacement:
    " << endl;
    cin >> choice;
    if (choice == 1)
    {
        // Insert data into hash table with replacement
        for (int i = 0; i < size; i++)
        {
            hash_table.insert_with_replacement(employees[i].id);
        }
    }
    else
    {
        // Insert data into hash table without replacement
        for (int i = 0; i < size; i++)
        {
            hash_table.insert_without_replacement(employees[i].id);
        }
    }

    cout << "Hash table now looks like this. " << endl;
    hash_table.print();

    cout << "Inserting data into the file. " << endl;

    // depending on the order of id in hash table, write the hashed_employee array
    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (employees[j].id == hash_table.table[i])
            {
                hashed_employees[i].age = employees[j].age;
                hashed_employees[i].id = employees[j].id;
                hashed_employees[i].name = employees[j].name;
            }
        }
    }
    hash_table.write_data_to_file(hashed_employees);
    hash_table.read_data_from_file();

    return 0;
}
```

```
Enter size of table:
4
Enter data for employee 1
Enter name:
Krish
Enter id:
12
Enter age:
21


Employee 1:
Name: Krish
```

```
Id: 12
Age: 21

Enter data for employee 2
Enter name:
Part
Enter id:
42
Enter age:
22


Employee 2:
Name: Part
Id: 42
Age: 22

Enter data for employee 3
Enter name:
Ram
Enter id:
23
Enter age:
32


Employee 3:
Name: Ram
Id: 23
Age: 32

Enter data for employee 4
Enter name:
Ramesh
Enter id:
24
Enter age:
21


Employee 4:
Name: Ramesh
Id: 24
Age: 21

Enter 1 to insert with replacement, 2 to insert without replacement:
1
Hash table now looks like this.
0:-1
1:-1
2:12
3:23
4:24
5:42
6:-1
7:-1
8:-1
9:-1

```

```
Inserting data into the file.
Writing employee 1
Employee 1:
Name:
Id: 0
Age: 0
Writing employee 2
Employee 2:
Name:
Id: 0
Age: 0
Writing employee 3
Employee 3:
Name: Krish
Id: 12
Age: 21
Writing employee 4
Employee 4:
Name: Ram
Id: 23
Age: 32
Writing employee 5
Employee 5:
Name: Ramesh
Id: 24
Age: 21
Writing employee 6
Employee 6:
Name: Part
Id: 42
Age: 22
Writing employee 7
Employee 7:
Name:
Id: 0
Age: 0
Writing employee 8
Employee 8:
Name:
Id: 0
Age: 0
Writing employee 9
Employee 9:
Name:
Id: 0
Age: 0
Writing employee 10
Employee 10:
Name:
Id: 0
Age: 0
Reading data from the file.
Employee 1:
Name:
Id: 0
Age: 0

Employee 2:
Name:
```

```
Id: 0
Age: 0

Employee 3:
Name: Krish
Id: 12
Age: 21

Employee 4:
Name: Ram
Id: 23
Age: 32

Employee 5:
Name: Ramesh
Id: 24
Age: 21

Employee 6:
Name: Part
Id: 42
Age: 22

Employee 7:
Name:
Id: 0
Age: 0

Employee 8:
Name:
Id: 0
Age: 0

Employee 9:
Name:
Id: 0
Age: 0

Employee 10:
Name:
Id: 0
Age: 0

Enter size of table:
2
Enter data for employee 1
Enter name:
krish
Enter id:
124
Enter age:
21


Employee 1:
Name: krish
Id: 124
Age: 21

```

```
Enter data for employee 2
Enter name:
Tony
Enter id:
4
Enter age:
23


Employee 2:
Name: Tony
Id: 4
Age: 23

Enter 1 to insert with replacement, 2 to insert without replacement:
2
Hash table now looks like this.
0:-1
1:-1
2:-1
3:-1
4:124
5:4
6:-1
7:-1
8:-1
9:-1

Inserting data into the file.
Writing employee 1
Employee 1:
Name:
Id: 0
Age: 0
Writing employee 2
Employee 2:
Name:
Id: 0
Age: 0
Writing employee 3
Employee 3:
Name:
Id: 0
Age: 0
Writing employee 4
Employee 4:
Name:
Id: 0
Age: 0
Writing employee 5
Employee 5:
Name: krish
Id: 124
Age: 21
Writing employee 6
Employee 6:
Name: Tony
Id: 4
Age: 23
```

```
Writing employee 7
Employee 7:
Name:
Id: 0
Age: 0
Writing employee 8
Employee 8:
Name:
Id: 0
Age: 0
Writing employee 9
Employee 9:
Name:
Id: 0
Age: 0
Writing employee 10
Employee 10:
Name:
Id: 0
Age: 0
Reading data from the file.
Employee 1:
Name:
Id: 0
Age: 0

Employee 2:
Name:
Id: 0
Age: 0

Employee 3:
Name:
Id: 0
Age: 0

Employee 4:
Name:
Id: 0
Age: 0

Employee 5:
Name: krish
Id: 124
Age: 21

Employee 6:
Name: Tony
Id: 4
Age: 23

Employee 7:
Name:
Id: 0
Age: 0

Employee 8:
Name:
Id: 0
```

```
Age: 0

Employee 9:
Name:
Id: 0
Age: 0

Employee 10:
Name:
Id: 0
Age: 0
```

10 Conclusion

Thus, we have implemented linear probing with and without replacement.

11 FAQ

1. **Write different types of hash functions.**
   There are several types of Hashing Functions. Here are a few:

   - **Division Method:** This method is the simplest of all hash functions. It simply divides the key by the table size and uses the remainder as the hash value. The hash function is:

     $$h(k) = k \mod m$$

   - **Multiplication Method:** In this method, the hash value is obtained by multiplying the key with a constant A and then taking the fractional part.
   - **Universal Hashing:** In this method, the hash function is obtained by using a universal hash function.
   - **Mid Square Method:** In this method, the key is first squared and the middle digits are then taken as the hash value.
   - **Random Number Method:** In this method, a random number is generated and then multiplied with the key to obtain the hash value.
   - **Folding Method:** In this method, the key is divided into equal parts and then added to obtain the hash value.
   - **Exponential Method:** In this method, the key is multiplied by a constant A and then the fractional part is taken as the hash value.
   - **Truncation Method:** In this method, the key is divided into equal parts and then the first part is taken as the hash value.

2. **Explain chaining with and without replacement with example.**

   Chaining is a collision resolution technique used in hash tables to resolve collisions by storing multiple keys in the same slot in the table, with each slot containing a linked list of key-value pairs.

   (a) With Replacement: Chaining with replacement involves replacing the old value with the new one when a collision occurs, while chaining without replacement involves inserting the new value into the linked list without replacing the old one.

   Here is an example of chaining with replacement:
   Suppose we have a hash table of size 5, and the hash function maps keys to slots as follows:

   - *key "apple" → slot 3*
   - *key "banana" → slot 1*
   - *key "cherry" → slot 3*
   - *key "date" → slot 0*

When we try to insert the key "cherry", we find that slot 3 is already occupied by the key "apple". In chaining with replacement, we replace the old value ("apple") with the new one ("cherry"), resulting in the linked list at slot 3 containing only the key-value pair ("cherry", value). Similarly, when we try to insert the key "orange", we find that slot 1 is already occupied by the key "banana". We replace the old value ("banana") with the new one ("orange"), resulting in the linked list at slot 1 containing only the key-value pair ("orange", value). The resulting hash table looks like this:

- *slot 0: ("date", value)*
- *slot 1: ("orange", value)*
- *slot 2: empty*
- *slot 3: ("cherry", value)*

(b) Chaining without replacement:

Suppose we have the same hash table and keys as in the previous example.

When we try to insert the key "cherry", we find that slot 3 is already occupied by the key "apple". In chaining without replacement, we add the key-value pair ("cherry", value) to the linked list at slot 3, resulting in the linked list containing both key-value pairs ("apple", value) and ("cherry", value).

Similarly, when we try to insert the key "orange", we find that slot 1 is already occupied by the key "banana". We add the key-value pair ("orange", value) to the linked list at slot 1, resulting in the linked list containing both key-value pairs ("banana", value) and ("orange", value).

The resulting hash table looks like this:

- *slot 0: ("date", value)*
- *slot 1: ("banana", value) → ("orange", value)*
- *slot 2: empty*
- *slot 3: ("apple", value) → ("cherry", value)*

In chaining without replacement, multiple key-value pairs can be stored in the same slot, without replacing any existing values.

3. **Explain quadratic probing with example**

Quadratic Probing *Quadratic probing is a technique used to resolve collisions in hash tables. When a collision occurs, meaning that two or more keys are mapped to the same slot, quadratic probing searches for the next available slot by adding a quadratic sequence of values to the original hash value until an empty slot is found.*

To illustrate how quadratic probing works, consider the following example. We have a hash table with 10 slots, and the following keys are inserted using a hash function:

- *Slot 0: ("date", value)*
- *Slot 1: empty*
- *Slot 2: empty*
- *Slot 3: ("apple", value)*
- *Slot 4: empty*
- *Slot 5: empty*
- *Slot 6: empty*
- *Slot 7: ("banana", value)*
- *Slot 8: empty*
- *Slot 9: ("cherry", value)*

Suppose we want to insert the key "fig" into the hash table. The hash function maps "fig" to slot 9, but we find that slot 9 is already occupied by the key "cherry".
To resolve this collision using quadratic probing, we start at slot 9 and search for the next available slot by adding a quadratic sequence of values to the original hash value.

Here's how we can find the next available slot using quadratic probing:

(a) Starting from slot 9, we add 1 to get slot 0. But slot 0 is already occupied by "date".

(b) We add 4 to the original hash value to get slot 13. We need to wrap around to the beginning of the hash table since the hash table only has 10 slots. So slot 13 becomes slot 3. But slot 3 is already occupied by "apple".

(c) We add 9 to the original hash value to get slot 18. We need to wrap around again to the beginning of the hash table. So slot 18 becomes slot 8. But slot 8 is empty, so we can insert "fig" into slot 8.

We insert the key-value pair ("fig", value) into slot 8, and the resulting hash table looks like this:

- *Slot 0: ("date", value)*
- *Slot 1: empty*
- *Slot 2: empty*
- *Slot 3: ("apple", value)*
- *Slot 4: empty*
- *Slot 5: empty*
- *Slot 6: empty*
- *Slot 7: ("banana", value)*
- *Slot 8: ("fig", value)*
- *Slot 9: ("cherry", value)*

Here, quadratic probing allowed us to find the next available slot by searching through a quadratic sequence of values until an empty slot was found.

MIT World Peace University

Advanced Data Structures
Second Year B. Tech, Semester 4

Implementation of AVL as a Data Structure

Assignment No. 9

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

April 26, 2023

Contents

1 Objectives **1**

2 Problem Statement **1**

3 Theory **1**
3.1 Different Cases of Rotations in AVL Tree 1
3.2 Construction of AVL Tree as a Data Structure for Creation of Dictionary 2
3.3 Searching and Deleting in AVL Tree 3
3.4 Advantages of AVL Tree over Other Data Structures 3
3.5 Disadvantages of AVL Tree over Other Data Structures 3

4 Platform **4**

5 Test Conditions **4**

6 Input and Output **4**

7 Pseudo Code **4**

8 Time Complexity **6**
8.1 Creation, Searching, Insertion and Deletion in AVL Trees 6

9 Code **6**
9.1 Program 6

10 Conclusion **13**

11 FAQ **14**

1 Objectives

1. 1. To study the concept of AVL trees
2. 2. To study different rotations applied on AVL tree

2 Problem Statement

A Dictionary stores keywords and its meaning. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword.

3 Theory

An AVL (Adelson-Velskii and Landis) tree is a self-balancing binary search tree in which the heights of the left and right subtrees of any node differ by at most one. The balancing property of AVL trees ensures that the worst-case time complexity of search, insert and delete operations is O(log n), where n is the number of nodes in the tree.

The height of a node is the number of edges in the longest path from the node to a leaf. An AVL tree is balanced if and only if the heights of its left and right subtrees differ by at most one. If the height difference is more than one, the tree is rebalanced by performing one or more rotations.

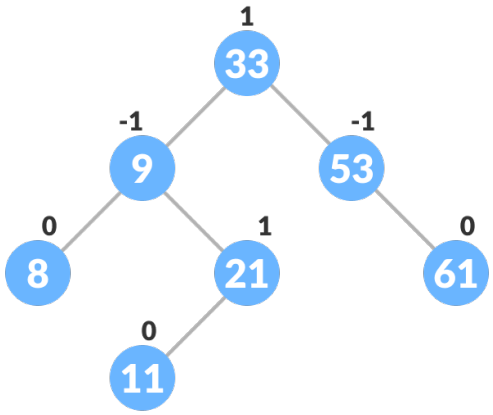


Figure 1: Example of an AVL Tree

3.1 Different Cases of Rotations in AVL Tree

There are four possible cases of rotation in AVL trees:

1. *Left-Left (LL)*: Case: This case occurs when the height of the left subtree of a node is greater than the height of the right subtree by more than one, and the height of the left subtree's left

child is greater than or equal to the height of its right child. To fix this, we perform a right rotation at the unbalanced node.

2. *Right-Right (RR)*: Case: This case occurs when the height of the right subtree of a node is greater than the height of the left subtree by more than one, and the height of the right subtree's right child is greater than or equal to the height of its left child. To fix this, we perform a left rotation at the unbalanced node.

3. *Left-Right (LR)*: Case: This case occurs when the height of the left subtree of a node is greater than the height of the right subtree by more than one, and the height of the left subtree's right child is greater than the height of its left child. To fix this, we perform a left rotation at the left child, followed by a right rotation at the unbalanced node.

4. *Right-Left (RL)*: Case: This case occurs when the height of the right subtree of a node is greater than the height of the left subtree by more than one, and the height of the right subtree's left child is greater than the height of its right child. To fix this, we perform a right rotation at the right child, followed by a left rotation at the unbalanced node.

3.2 Construction of AVL Tree as a Data Structure for Creation of Dictionary

AVL trees are commonly used as data structures for the creation of dictionaries. A dictionary is a collection of key-value pairs, where each key is associated with a value. In an AVL tree-based dictionary, the keys are stored in the tree, and the associated values are stored in the nodes.

To construct an AVL tree-based dictionary, we start with an empty AVL tree. For each key-value pair to be inserted, we perform a binary search in the tree to find the correct position for insertion. If the key is already present in the tree, we update its value. If the key is not present, we create a new node with the key-value pair and insert it into the tree. After insertion, we check if the tree is still balanced. If it is not, we perform one or more rotations to balance it.

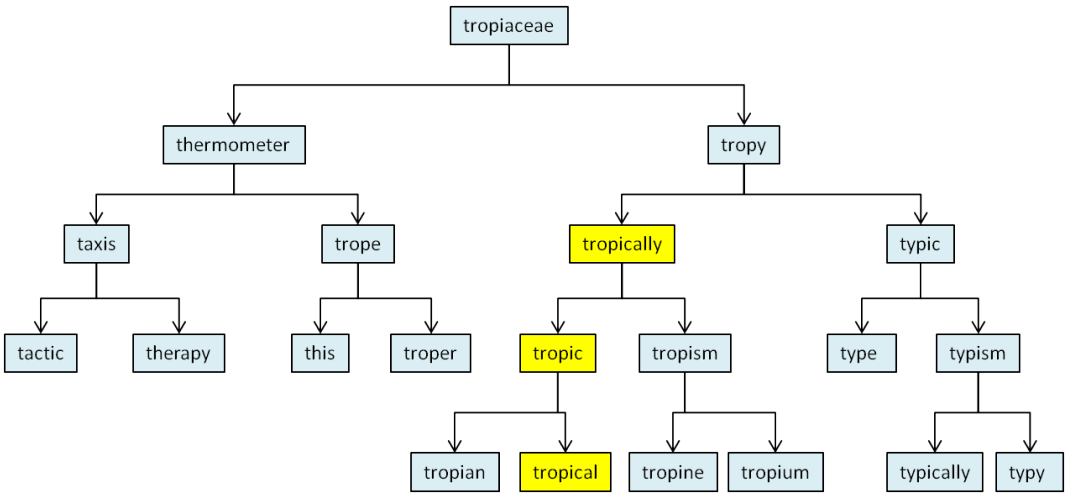


Figure 2: Example of an AVL Tree being used as a Dictionary

3.3 Searching and Deleting in AVL Tree

To search for a key in the AVL tree-based dictionary, we perform a binary search in the tree. If the key is found, we return its associated value. Otherwise, we return a null value to indicate that the key is not present in the dictionary.

To delete a key from the AVL tree-based dictionary, we perform a binary search to find the node containing the key. If the key is found, we delete the node and perform one or more rotations to balance the tree if necessary. If the node to be deleted has two children, we replace it with the successor or predecessor node, which is the node with the smallest or largest key in its right or left subtree, respectively.

3.4 Advantages of AVL Tree over Other Data Structures

The AVL tree-based dictionary has several advantages over other data structures, such as hash tables or binary search trees.

1. AVL trees are self-balancing, which means that the height of the left and right subtrees of any node differ by at most one. This ensures that the worst-case time complexity of search, insert, and delete operations is O(log n), where n is the number of nodes in the tree. Hash tables have an average-case time complexity of O(1), but their worst-case time complexity can be O(n) if all the keys hash to the same slot. Binary search trees have a worst-case time complexity of O(n) if the tree is unbalanced.

2. AVL trees are more space-efficient than hash tables. The space complexity of an AVL tree is O(n), where n is the number of nodes in the tree. The space complexity of a hash table is O(n), where n is the number of key-value pairs in the table.

3. AVL trees have a guaranteed worst-case time complexity of O(log n) for search, insert, and delete operations, regardless of the distribution of the keys. Hash tables have an average-case time complexity of O(1), but their worst-case time complexity can be O(n) if all the keys hash to the same slot. Binary search trees have a worst-case time complexity of O(n) if the tree is unbalanced.

3.5 Disadvantages of AVL Tree over Other Data Structures

1. Overhead: Maintaining the balance of AVL trees requires additional operations and memory compared to regular binary search trees, which can add overhead to the implementation.

2. Rotations: Whenever an insertion or deletion violates the balance condition of an AVL tree, rotations are required to restore the balance. These rotations can be complex and time-consuming, especially in larger trees.

3. Limited use: AVL trees are not always the best choice for certain types of applications. For example, in situations where insertions and deletions are infrequent and searches are more common, a simple binary search tree may be a better choice.

4. Implementation complexity: Implementing an AVL tree can be more complex than implementing a regular binary search tree, which can lead to more errors and bugs in the code.

5. Memory overhead: In some cases, AVL trees may consume more memory than other types of binary search trees, due to their need to maintain balance. This can be a concern in memory-constrained environments or when dealing with very large data sets.

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers : g++ and gcc on linux for C++

5 Test Conditions

1. Input min 10 elements.

6 Input and Output

1. The Elements in Ascending Order, or Descending order

7 Pseudo Code

Pseudo Code for Creation of AVL Trees

```
    struct Node
        string word
        string definition
        struct Node *l
        struct Node *r
        friend class AVL_Tree

    class AVL_Tree
    public:
        Node *head
        AVL_Tree()
            head = new Node
            head->l = NULL
            head->r = NULL
            head->definition = "AVL Dictionary"
            head->word = "Head"
```

Pseudo Code for Rotations of AVL Trees

```
    Node *RR_rotation(Node *parent)
    Node *t
    t = parent->r
    parent->r = t->l
    t->l = parent
    cout << "Right-Right Rotation Performed" << endl
    return t
  Node *LL_rotation(Node *parent)
    Node *t
```

```
    t = parent->l
    parent->l = t->r
    t->r = parent
    cout << "Left-Left Rotation Performed" << endl
    return t
  Node *LR_rotation(Node *parent)
    Node *t
    t = parent->l
    parent->l = RR_rotation(t)
    cout << "Left-Right Rotation Performed" << endl
    return LL_rotation(parent)
  Node *RL_rotation(Node *parent)
    Node *t
    t = parent->r
    parent->r = LL_rotation(t)
    cout << "Right-Left Rotation Performed" << endl
    return RR_rotation(parent)
```

Pseudo Code for Balancing of AVL Trees

```
    Node *balance_AVL_Tree(Node *t)
    int bal_factor = find_difference(t)
    if (bal_factor > 1)
      if (find_difference(t->l) > 0)
        t = LL_rotation(t)
      else
        t = LR_rotation(t)
    else if (bal_factor < -1)
      if (find_difference(t->r) > 0)
        t = RL_rotation(t)
      else
        t = RR_rotation(t)
    return t
```

Pseudo Code for Insertion of AVL Trees

```
  Node *insert_words(Node *r, string word, string definition)
    if (r == NULL)
      r = new Node
      r->word = word
      r->definition = definition
      r->l = NULL
      r->r = NULL
      return r
    else if (strcmp(word.c_str(), r->word.c_str()) < 0)
      r->l = insert_words(r->l, word, definition)
      r = balance_AVL_Tree(r)
    else if (strcmp(word.c_str(), r->word.c_str()) >= 0)
      r->r = insert_words(r->r, word, definition)
      r = balance_AVL_Tree(r)
    return r
```

8 Time Complexity

8.1 Creation, Searching, Insertion and Deletion in AVL Trees

- **Time Complexity:**

$$O(n\log(n))$$

- **Space Complexity:**

$$O(n)$$

9 Code

9.1 Program

```
#include <iostream>
#include <cstdio>
#include <sstream>
#include <algorithm>
#include <string.h>
#define pow2(n) (1 << (n))
using namespace std;
struct Node
{
  string word;
  string definition;
  struct Node *l;
  struct Node *r;
  friend class AVL_Tree;
};

class AVL_Tree
{
public:
  Node *head;
  AVL_Tree()
  {
    head = new Node;
    head->l = NULL;
    head->r = NULL;
    head->definition = "AVL Dictionary";
    head->word = "Head";
  }

  int find_height(Node *t)
  {
    int h = 0;
    if (t != NULL)
    {
      int l_height = find_height(t->l);
      int r_height = find_height(t->r);
      int max_height = max(l_height, r_height);
      h = max_height + 1;
    }
    return h;
  }
  int find_difference(Node *t)
```

```
{
  int l_height = find_height(t->l);
  int r_height = find_height(t->r);
  int b_factor = l_height - r_height;
  return b_factor;
}
Node *RR_rotation(Node *parent)
{
  Node *t;
  t = parent->r;
  parent->r = t->l;
  t->l = parent;
  cout << "Right-Right Rotation Performed" << endl;
  return t;
}
Node *LL_rotation(Node *parent)
{
  Node *t;
  t = parent->l;
  parent->l = t->r;
  t->r = parent;
  cout << "Left-Left Rotation Performed" << endl;
  return t;
}
Node *LR_rotation(Node *parent)
{
  Node *t;
  t = parent->l;
  parent->l = RR_rotation(t);
  cout << "Left-Right Rotation Performed" << endl;
  return LL_rotation(parent);
}
Node *RL_rotation(Node *parent)
{
  Node *t;
  t = parent->r;
  parent->r = LL_rotation(t);
  cout << "Right-Left Rotation Performed" << endl;
  return RR_rotation(parent);
}
Node *balance_AVL_Tree(Node *t)
{
  int bal_factor = find_difference(t);
  if (bal_factor > 1)
  {
    if (find_difference(t->l) > 0)
      t = LL_rotation(t);
    else
      t = LR_rotation(t);
  }
  else if (bal_factor < -1)
  {
    if (find_difference(t->r) > 0)
      t = RL_rotation(t);
    else
      t = RR_rotation(t);
  }
  return t;
}
```

```
102  Node *insert_words(Node *r, string word, string definition)
103  {
104    if (r == NULL)
105    {
106      r = new Node;
107      r->word = word;
108      r->definition = definition;
109      r->l = NULL;
110      r->r = NULL;
111      return r;
112    }
113    else if (strcmp(word.c_str(), r->word.c_str()) < 0)
114    {
115      r->l = insert_words(r->l, word, definition);
116      r = balance_AVL_Tree(r);
117    }
118    else if (strcmp(word.c_str(), r->word.c_str()) >= 0)
119    {
120      r->r = insert_words(r->r, word, definition);
121      r = balance_AVL_Tree(r);
122    }
123    return r;
124  }
125  void display_AVL_tree(Node *p, int l)
126  {
127    int i;
128    if (p != NULL)
129    {
130      display_AVL_tree(p->r, l + 1);
131      cout << endl;
132      if (p == head)
133        cout << "Root -> ";
134      for (i = 0; i < l && p != head; i++)
135        cout << endl;
136      cout << p->word << ": " << p->definition << endl;
137      display_AVL_tree(p->l, l + 1);
138    }
139  }
140  void inorder_traversal(Node *t)
141  {
142    if (t == NULL)
143      return;
144    inorder_traversal(t->l);
145    cout << t->word << " ";
146    inorder_traversal(t->r);
147  }
148  void preorder_traversal(Node *t)
149  {
150    if (t == NULL)
151      return;
152    cout << t->word << " ";
153    preorder_traversal(t->l);
154    preorder_traversal(t->r);
155  }
156  void postorder_traversal(Node *t)
157  {
158    if (t == NULL)
159      return;
160    postorder_traversal(t->l);
```

```
    postorder_traversal(t->r);
    cout << t->word << " ";
  }
};

int main()
{
  int c, i;
  string word, definition;
  AVL_Tree avl;
  while (1)
  {
    cout << "1.Insert Element into the tree" << endl;
    cout << "2.show Balanced AVL Tree" << endl;
    cout << "3.InOrder traversal" << endl;
    cout << "4.PreOrder traversal" << endl;
    cout << "5.PostOrder traversal" << endl;
    cout << "6.Exit" << endl;
    cout << "Enter your Choice: ";
    cout << endl
       << endl;
    cin >> c;
    switch (c)
    {
    case 1:
      cout << "Enter the word to be inserted: ";
      cin >> word;
      cout << "Enter the definition of the word: ";
      cin >> definition;
      avl.head = avl.insert_words(avl.head, word, definition);
      break;
    case 2:
      if (avl.head == NULL)
      {
        cout << "Tree is Empty" << endl;
        continue;
      }
      cout << "Balanced AVL Tree:" << endl;
      avl.display_AVL_tree(avl.head, 1);
      cout << endl;
      break;
    case 3:
      cout << "Inorder Traversal:" << endl;
      avl.inorder_traversal(avl.head);
      cout << endl;
      break;
    case 4:
      cout << "Preorder Traversal:" << endl;
      avl.preorder_traversal(avl.head);
      cout << endl;
      break;
    case 5:
      cout << "Postorder Traversal:" << endl;
      avl.postorder_traversal(avl.head);
      cout << endl;
      break;
    case 6:
      exit(1);
      break;
```

```
    default:
      cout << "Wrong Choice" << endl;
    }
  }
  return 0;
}
```

```
-> krishnaraj@Krishnaraj-Arch -> /run/media/krishnaraj/Classes/University/Second Year/Second Semester/Advanced Data Structures/Programs -> main -> cd "/run/media/krishnaraj/Classes/University/Second Year/Second Semester/Advanced Data Structures/Programs/" && g++ Assignment_9.cpp -o Assignment_9 && "/run/media/krishnaraj/Classes/University/Second Year/Second Semester/Advanced Data Structures/Programs/"Assignment_9
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Pineapple
Enter the definition of the word: fruit
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Apple
Enter the definition of the word: Fruit
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Keys
Enter the definition of the word: object
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Laptop
Enter the definition of the word: computer
Right-Right Rotation Performed
Left-Right Rotation Performed
```

```
Left-Left Rotation Performed
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Guava
Enter the definition of the word: fruit
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Arc_Reactor
Enter the definition of the word: heart
Left-Left Rotation Performed
Right-Left Rotation Performed
Right-Right Rotation Performed
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Suit
Enter the definition of the word: if_ur_nothing_without_it_you_shouldnt_have_it
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Genius
Enter the definition of the word: Tony
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Billionnaire
Enter the definition of the word: Tony
```

```
Left-Left Rotation Performed
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Philanthropist
Enter the definition of the word: Tony
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

2
Balanced AVL Tree:

Suit: if_ur_nothing_without_it_you_shouldnt_have_it
Pineapple: fruit
Philanthropist: Tony
Laptop: computer
Keys: object
Root -> Head: AVL Dictionary
Guava: fruit
Genius: Tony
Billionnaire: Tony
Arc_Reactor: heart
Apple: Fruit

1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

3
Inorder Traversal:
Apple Arc_Reactor Billionnaire Genius Guava Head Keys Laptop Philanthropist
    Pineapple Suit
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

4
Preorder Traversal:
Head Arc_Reactor Apple Genius Billionnaire Guava Laptop Keys Pineapple
    Philanthropist Suit
```

```
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

5
Postorder Traversal:
Apple Billionnaire Guava Genius Arc_Reactor Keys Philanthropist Suit Pineapple
    Laptop Head
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 6
```

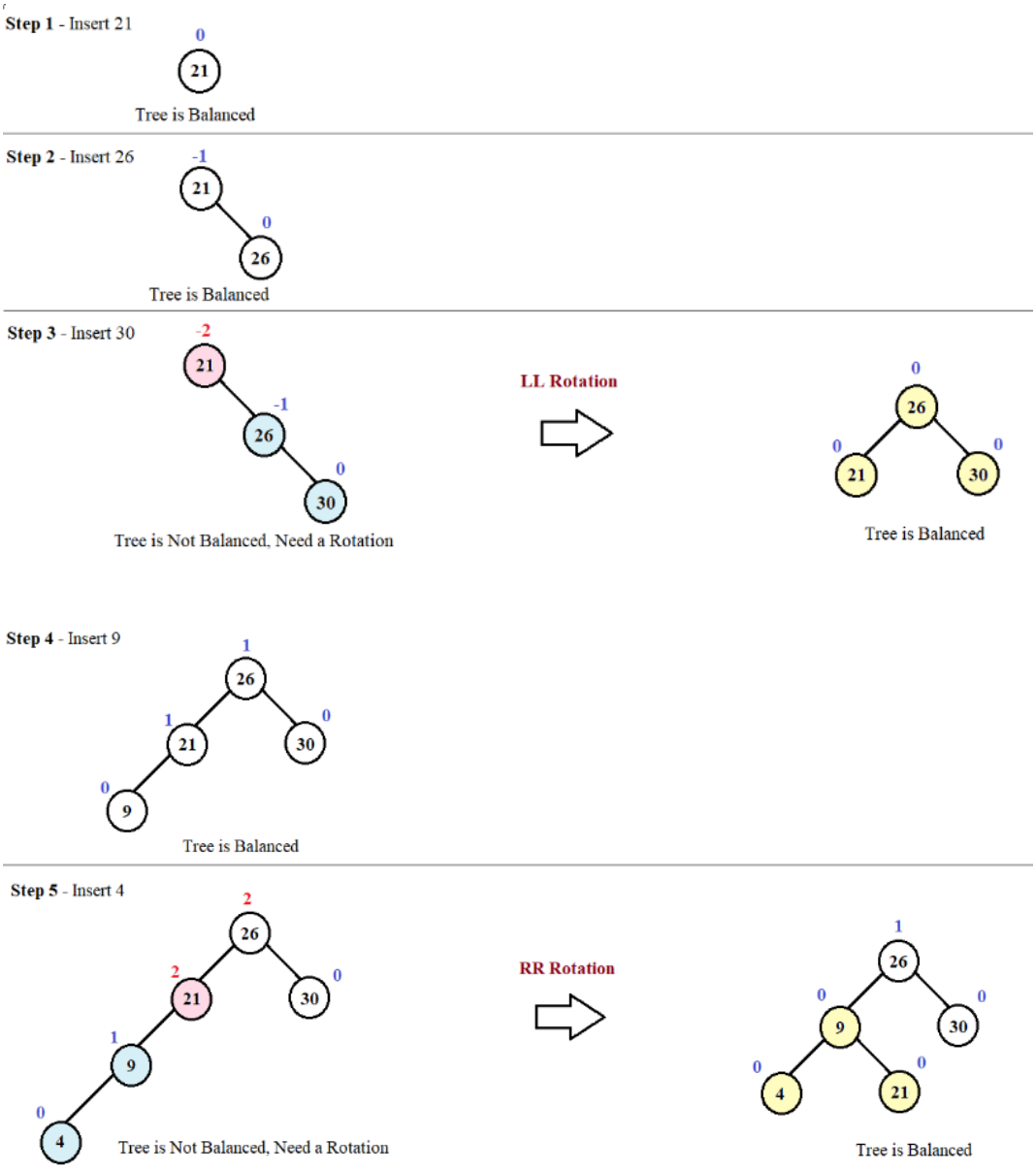
10 Conclusion

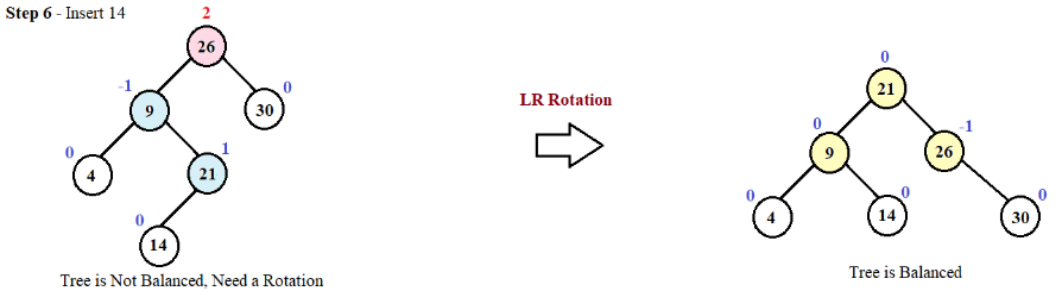
Thus, we have understood the importance and use of AVL trees as a Data structure, and how they are better and more efficient than Binary Search Trees.

11 FAQ

1. **Discuss AVL trees with suitable example?**
   AVL trees are a type of self-balancing binary search tree that maintain a balance between left and right subtrees to ensure fast search, insertion, and deletion operations with a worst-case time complexity of O(log n), where n is the number of nodes in the tree.

   Let us look at an example of an AVL tree. Consider the following AVL tree:
   AVL Tree for the given Sequence 21, 26, 30, 9, 4, 14





2. **Compute the time complexity of AVL tree creation?**

   The time complexity of creating an AVL tree depends on the number of elements n that need to be inserted. In the worst case, when the elements are inserted in sorted order, the time complexity of AVL tree creation is O(n log n), since each insertion may require a series of rotations to maintain balance. In the best case, when the elements are inserted in a balanced manner, the time complexity of AVL tree creation is O(n). This is because each node is inserted and balanced in constant time, so the total time is proportional to the number of nodes inserted.

   In practice, AVL trees are very efficient for creating and maintaining a dictionary because they provide fast search, insertion, and deletion operations. However, they do require additional memory to store the balance factor of each node, and the overhead of maintaining balance can add to the running time of operations. Overall, AVL trees are a good choice for applications that require a balanced binary search tree and where the number of insertions and deletions is not too frequent.