MIT World Peace University
Advanced Data Structures

Assignment 1

Naman Soni Roll No. 10

Contents

1 Problem Statement

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

2 Objective

1. To study data structure: Circular Linked List.
2. To study differnet operations that can be performed on Circular Linked List.
3. To study applications of Circular Linked List.

3 Theory

3.1 *Circular Linked List*

A circular linked list is a variation of a linked list where the last element in the list points back to the first element, creating a loop. This means that there is no null element at the end of the list, and traversing the list will continue indefinitely. In a circular linked list, there is no concept of "head" and "tail" as the first and last element are connected. This data structure can be useful in certain algorithms or applications where traversing the list in a circular fashion is useful.

3.2 *Diffrence between SLL, CLL and DLL*

SLL (Single Linked List) is a linked list where each element, or node, in the list contains a reference to the next element in the list, but not the previous one. This means that you can traverse the list in one direction, typically from the head (first element) to the tail (last element).

CLL (Circular Linked List) is a variation of a linked list where the last element in the list points back to the first element, creating a loop. This means that there is no null element at the end of the list, and traversing the list will continue indefinitely. In a circular linked list, there is no concept of "head" and "tail" as the first and last element are connected.

DLL (Double Linked List) is a linked list where each element, or node, in the list contains a reference to both the next and previous element in the list. This means that you can traverse the list in both directions, from the head to the tail or from the tail to the head.

In short, SLL is a one-way linked list, CLL is a circular linked list and DLL is a two-way linked list.

3.3 *Various Operations on CLL*

There are several operations that can be performed on a circular linked list:

- Insertion: This operation allows you to add new elements to the list. In a circular linked list, elements can be inserted at the beginning, at the end, or at a specific position in the list.
- Deletion: This operation allows you to remove elements from the list. In a circular linked list, elements can be deleted from the beginning, from the end, or from a specific position in the list.
- Traversal: This operation allows you to iterate through the elements of the list. In a circular linked list, traversal can be done in a circular fashion, starting from the head and going to the tail, and then back to the head again.
- Search: This operation allows you to search for a specific element in the list. In a circular linked list, the search can be done starting from the head and going to the tail, and then back to the head again.
- Reversal: This operation allows you to reverse the order of elements in the list.
- Sorting: This operation allows you to sort the elements of the list in a specific order.

- Length: This operation allows you to find the number of elements in the list.
- Display: This operation allows you to display the elements of the list.
- Concatenation: This operation allows you to join two different circular linked list.
- Splitting: This operation allows you to divide the circular linked list into two different list.

4 Implementation

4.1 *Platform*

Mac OS 64x
Visual Studio Code

4.2 *Input and Output*

TESTCASE NO	**INPUT**	**OUTPUT**
01	Row 1, Cell 2	Row 1, Cell 3
Row 2, Cell 1	Row 2, Cell 2	Row 2, Cell 3

4.3 *Test Conditions*

- Input atleast 5 nodes.
- Addition of two polynomials with atleast 5 terms.
- Evalute polynomial with floating values.

4.4 *Psuedo Code*

```c
/*Problem Statement: Implement polynomial operations using Circular Linked List: Create, Display, Addition and
Evaluation
Name: Naman Soni
Roll No. 10
Batch A1
*/

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

struct node
{
    int coeff;
    int exp;
    struct node *next;
};

void add_data(struct node *head)
{
    int choice = 1;
    struct node *temp = head;

    do
    {

        struct node *curr = (struct node *)malloc(sizeof(struct node));

        printf("\nEnter coefficient:\n");
        scanf("%d", &curr->coeff);
        printf("\nEnter exponent:\n");
        scanf("%d", &curr->exp);
        curr->next = head;
        temp->next = curr;
        temp = temp->next;
        printf("\nDo you want to enter more terms?\nEnter 1 for yes and 0 for no\n");
        scanf("%d", &choice);
    } while (choice != 0);
}

void display(struct node *head)
{
    if (head->next == head)
    {
        printf("\nNo data available");
    }
    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");
}

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;
        }
    }

    // Case when p2 exhausts before p1.
    if (p1 == head1 && p2 != head2)
    {
        result_temp->next = p2;

        // This loop is to make the last node of the result polynomial point to the head of the result polynomial.
        while (result_temp->next != head2)
        {
            result_temp = result_temp->next;
```

```
        }
        result_temp->next = result_head;
    }

    // Case when p1 exhausts before p2.
    else if (p1 != head1 && p2 == head2)
    {
        result_temp->next = p1;
        while (result_temp->next != head1)
        {
            result_temp = result_temp->next;
        }
        result_temp->next = result_head;
    }

    // Case when both p1 and p2 exhaust.
    else if (p1 != head1 && p2 != head2)
    {
        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;
}

int main()
{
    int choice = 0;
    struct node *head = (struct node *)malloc(sizeof(struct node));
    struct node *head1 = (struct node *)malloc(sizeof(struct node));
    struct node *head2 = (struct node *)malloc(sizeof(struct node));
    struct node *added;

    printf("What you want to do:\n1.Insert Polynomial\n2.Addition of Two polynomials:\n");
    scanf("%d", &choice);

    switch (choice)
    {
    case 1:
        printf("Insert polynomial:\n");
        add_data(head);
        display(head);
        break;

    case 2:
        printf("Please enter the first polynomial:\n");
        add_data(head1);
        display(head1);
        printf("\nEnter second polynomial:\n");
        add_data(head2);
        display(head2);
        printf("Addition of Polynomials:");
        added = add_polynomials(head1, head2);
        display(added);
        break;
    default:
        printf("!Invalid!");
```

```
        break;
    }

    return 0;
}
```

Listing 1: Input Code

```
    cd "/Users/cyrus/Desktop/Sem-4/ADS/" && g++ Assignment1.cpp -o Assignment1 && "/Users/
    cyrus/Desktop/Sem-4/ADS/"Assignment1
    cyrus@Namans-MacBook-Air Sem-4 % cd "/Users/cyrus/Desktop/Sem-4/ADS/" && g++ Assignment1
    .cpp -o Assignment1 && "/Users/cyrus/Desktop/Sem-4/ADS/"Assignment1
    What you want to do:
    1.Insert Polynomial
    2.Addition of Two polynomials:
    1
    Insert polynomial:

    Enter coefficient:
    3

    Enter exponent:
    2

    Do you want to enter more terms?
    Enter 1 for yes and 0 for no
    1

    Enter coefficient:
    cyrus@Namans-MacBook-Air ADS % cd "/Users/cyrus/Desktop/Sem-4/ADS/" && g++ Assignment1.
    cpp -o Assignment1 && "/Users/cyrus/Desktop/Sem-4/ADS/"Assignment1
    What you want to do:
    1.Insert Polynomial
    2.Addition of Two polynomials:
    2
    Please enter the first polynomial:

    Enter coefficient:
    3

    Enter exponent:
    2

    Do you want to enter more terms?
    Enter 1 for yes and 0 for no
    1

    Enter coefficient:
    5

    Enter exponent:
    1

    Do you want to enter more terms?
    Enter 1 for yes and 0 for no
    1

    Enter coefficient:

    9

    Enter exponent:
    0

    Do you want to enter more terms?
    Enter 1 for yes and 0 for no
    0
    3x^2+5x^1+9x^0
```

```
    Enter second polynomial:

    Enter coefficient:
    4

    Enter exponent:
    6

    Do you want to enter more terms?
    Enter 1 for yes and 0 for no
    1

    Enter coefficient:
    8

    Enter exponent:
    0

    Do you want to enter more terms?
    Enter 1 for yes and 0 for no
    0
    4x^6+8x^0
    Addition of Polynomials:4x^6+3x^2+5x^1+17x^0
    cyrus@Namans-MacBook-Air ADS %
```

Listing 2: Output

5 Conclusion

Thus, implemented different operations on CLL.

6 FAQ

1. Write an ADT for CLL.
An Abstract Data Type (ADT) for a Circular Linked List (CLL) could include the following operations:

1. Initialize: This operation creates an empty CLL.
2. Insert: This operation adds a new node to the CLL. The new node can be inserted at the beginning, end, or at a specific position in the CLL.
3. Delete: This operation removes a node from the CLL. The node to be deleted can be specified by its position in the CLL.
4. Search: This operation searches for a specific node in the CLL. The search can be based on the data stored in the node or the position of the node in the CLL.
5. Traverse: This operation visits each node in the CLL in a specific order, such as in a clockwise or counter-clockwise direction.
6. Length: This operation returns the number of nodes in the CLL.
7. isEmpty: This operation checks whether the CLL is empty or not.

Here is an example of an ADT for a CLL in C language:

```c
// ADT for a Circular Linked List

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

```
7
8  struct CLL {
9    int length;
10   struct Node* head;
11 }
12
13 // Initialize an empty CLL
14 void init(struct CLL* cll) {
15   cll->length = 0;
16   cll->head = NULL;
17 }
18
19 // Insert a new node at the beginning of the CLL
20 void insert_at_beginning(struct CLL* cll, int data) {
21   struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
22   new_node->data = data;
23
24   if (cll->head == NULL) {
25     new_node->next = new_node;
26     cll->head = new_node;
27   }
28   else {
29     struct Node* temp = cll->head;
30     while (temp->next != cll->head) {
31       temp = temp->next;
32     }
33     temp->next = new_node;
34     new_node->next = cll->head;
35     cll->head = new_node;
36   }
37   cll->length++;
38 }
39
40 // Delete a node from the CLL
41 void delete_node(struct CLL* cll, int position) {
42   if (cll->head == NULL) {
43     printf
```

Listing 3: EXAMPLE

2. How to perform multiplication of two polynomials?
Multiplying two polynomials is a process of combining like terms. The process of multiplying two polynomials is similar to the process of multiplying two numbers, with the difference that each term in a polynomial has an exponent.
Here is the general process for multiplying two polynomials:

1. Write down both polynomials to be multiplied.
2. Distribute the first term of the first polynomial with every term of the second polynomial.
3. Distribute the second term of the first polynomial with every term of the second polynomial.
4. Repeat step 2 and 3 for each term of the first polynomial.
5. Add up all the results obtained in step 2, 3 and 4.

3. Write polynomial addition algorithm if terms are not sorted.
The algorithm for adding two polynomials if the terms are not sorted is as follows:

1. Create an empty polynomial to store the result.
2. Create two pointers, one for each polynomial to be added.
3. Initialize both pointers to the beginning of their respective polynomials.
4. While both pointers are not at the end of their respective polynomials: a. Compare the exponents of the two terms pointed to by the pointers. b. If the exponents are the same, add the coefficients and store the result in the new polynomial with the same exponent. c. If the exponent of one term is greater than the other, add that term to the new polynomial and move the pointer to the next term of that polynomial. d. If the exponent of one term is less than the other, add that term to the new polynomial and move the pointer to the next term of that polynomial.
5. While one of the pointers is not at the end of its respective polynomial, add the remaining terms to the new polynomial.
6. Return the new polynomial which contains the sum of the two polynomials.

Here is an example implementation of the algorithm in C-like language:

```
    struct Term {
        int coefficient;
        int exponent;
        struct Term* next;
      };

      struct Polynomial {
        struct Term* head;
      };

      struct Polynomial* add(struct Polynomial* poly1, struct Polynomial* poly2)
        struct Poly
```

Listing 4: example

MIT World Peace University
Advanced Data Structures

Assignment 2

Naman Soni Roll No. 10

Contents

1 Problem Statement

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

2 Objective

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

3 Theory

3.1 *Trees*

Trees are a type of data structure used to represent hierarchical relationships between elements. In computer science, trees are often used to represent hierarchical relationships between elements such as files and directories in a file system, or in the representation of a decision tree for machine learning algorithms.

A tree consists of nodes, which are connected by edges. Each node in a tree has zero or more child nodes, and a single parent node, except for the root node, which has no parent. The root node is the topmost node in the tree, and all other nodes are descended from it. Nodes that have no children are called leaf nodes.

Trees have several important properties, such as being able to represent hierarchical relationships efficiently, supporting searching, insertion, and deletion operations in logarithmic time, and being able to be easily traversed and manipulated.

There are several different types of trees, such as binary trees, AVL trees, B-trees, and heap trees, each with its own unique properties and uses.

3.2 *Different definitions related to binary tree*

In computer science, 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. Some common definitions related to binary trees are:

- Root: The topmost node in a binary tree is called the root. It has no parent node.
- Leaf: A node that has no children is called a leaf node.
- Parent: A node that has one or more child nodes is called a parent node.
- Child: A node that is a direct descendant of a parent node is called a child node.
- Siblings: Nodes that have the same parent are called siblings.
- Depth: The depth of a node is the number of edges from the root node to that node.
- Height: The height of a binary tree is the number of edges from the root node to the deepest leaf node.
- Subtree: The set of nodes and edges that are descendants of a given node is called a subtree.
- Full binary tree: A binary tree is considered full if every node has either 0 or 2 children.
- Perfect binary tree: A binary tree is considered perfect if all its leaf nodes are at the same depth and every parent node has exactly two children.

3.3 *Different Traversals (Inorder, Preorder and Postorder)*

1. Inorder traversal: In an inorder traversal, the left subtree of a node is visited first, then the node itself, and finally the right subtree. This results in visiting the nodes in ascending order if the tree is used to represent an ordered set.

2. Preorder traversal: In a preorder traversal, the node itself is visited first, then the left subtree, and finally the right subtree. This is useful for creating a copy of the tree or for creating an output that represents the structure of the tree.

3. Postorder traversal: In a postorder traversal, the left subtree is visited first, then the right subtree, and finally the node itself. This is useful for freeing up memory in a tree-like data structure, as the children can be deleted before the parent.

4 Implementation

4.1 *Platform*

Operating System: Mac OS 64-bit
IDE Used: Visual Studio Code

4.2 *Test Conditions*

1. Input at least 10 nodes.

2. Display all traversals of binary tree with 10 nodes. (recursive and non-recursive)

4.3 *Input-Output Code*

```
#include <iostream>
using namespace std;
class treenode {
  public :
  string data;
  treenode *left;
  treenode *right;
  friend class tree;
};
class stack{
  int top;
  treenode *data [30];
  public:
  stack()
  {
    top=-1;
  }
  void push (treenode *temp)
  {
    top++;
    data[top] = temp;
  }
  treenode*pop()
  {
    return data[top--];
  }
  int isempty(){
    if (top == -1)
    {
      return 1;
    }
    else
```

```
    {
      return 0;
    }
  }
  friend class tree;
};

class tree {
  public:
  treenode *root;
  tree(){
    root = NULL;
  }
  void create_r()
  {
    root = new treenode;
    cout<<"Enter ROOT Data:"<<endl;
    cin>>root->data;
    root->left = NULL;
    root->right = NULL;
    create_r(root);
  }

  void create_r(treenode * temp){
    char choice1, choice2;
    cout << "Enter whether you want to add the data to the left of \""<< temp->data <<"\"
  or not (y/n): ";
    cin >> choice1;
    if (choice1 == 'y'){
      treenode *curr1 = new treenode();
      cout << "Enter the data for the left of \""<< temp->data << "\" node: ";
      cin >> curr1 -> data;
      temp -> left = curr1;
      create_r(curr1);
    }
    cout << "Enter whether you want to add the data to the right of \""<<temp->data<<"\"
  or not (y/n): ";
    cin >> choice2;
    if (choice2 == 'y'){
      treenode *curr2 = new treenode();
      cout << "Enter the data for the Right of \""<< temp->data <<"\" node:";
      cin >> curr2 -> data;
      temp -> right = curr2;
      create_r(curr2);

    }
  }
  //inorder recursive
  void inorder_traversal_r (treenode * node){
    if (node == NULL){
      return;
    }
    inorder_traversal_r(node->left);
    cout << node->data << "\n";
    inorder_traversal_r(node->right);
  }
  //preorder recursive
  void preorder_traversal_r(treenode* temp)
  {
    if (temp!= NULL)
    {
      cout<< temp->data << endl;
      preorder_traversal_r(temp->left);
      preorder_traversal_r(temp->right);
    }
  }
  // postorder recursive
  void postorder_traversal_r(treenode*temp)
```

```
    {
      if(temp!=NULL)
      {
        postorder_traversal_r(temp->left);
        postorder_traversal_r(temp->right);
        cout<<temp->data<<endl;
      }
    }
    //Inorder Non recursive
    void inorder_traversal_non_recursive(treenode*temp)
    {
      temp =root;
      stack st;
      while (1)
      {
        while(temp!=NULL)
        {
          st.push(temp);
          temp = temp->left;
        }
        if(st.isempty()==1)
        {
          break;
        }
        temp=st.pop();
        cout<< temp->data<<endl;
        temp = temp->right;
      }
    }

    //Postorder non recursive
    void postorder_traversal_non_recursive(treenode*temp)
    {
      temp = root;
      stack st;
      while(1)
      {
        while(temp!=NULL)
        {
          st.push(temp);
          temp = temp->left;
        }
        if (st.data[st.top]->right == NULL)
        {
          temp=st.pop();
          cout<<temp->data<<endl;
        }
        while(st.isempty()==0 && st.data[st.top]->right==temp)
        {
          temp=st.pop();
          cout<<temp->data;
        }
        if (st.isempty()==1)
        {
          break;
        }
        temp = st.data[st.top]->right;
      }
    }

    //Preorder Non recursive
    void preorder_traversal_non_recursive(treenode*temp)
    {
      temp = root;
      stack st;
      while(1)
      {
        while(temp!=NULL)
```

```
        {
          cout<<temp->data<<endl;
          st.push(temp);
          temp =temp->left;
        }
        if(st.isempty()==1)
        {
          break;
        }
        temp=st.pop();
        temp = temp->right;
      }
    }
};
/*Test cases
1. left skewed
2. Right skewed
3. Complete Tree
4. Full tree
5. Normal Tree
6. Binary search Tree
*/
int main() {
  tree bt;
  // treenode * root = new treenode();
  // cout << "Enter the data for the root node: ";
  // cin >> root -> data;
  int ch;
  char c;
  do {

    cout<<"\nWhat you want to do:\n1.Enter Tree\n2.Inorder Recursive\n3.Preorder Recursive
  \n4.Postorder Recursive\n5.Inorder Non Recursive\n6.Preorder Non Recursive\n7.Postorder
  Non Recursive\nchoose:"<<endl;
    cin>>ch;
    switch (ch)
    {
      case 1:
      bt.create_r();
      break;
      case 2:
      cout << "The inorder display of the tree is: " << endl;
      bt.inorder_traversal_r(bt.root);
      break;
      case 3:
      cout<<"The preorder display of the tree is:" << endl;
      bt.preorder_traversal_r(bt.root);
      break;
      case 4:
      cout<<"The postorder display of the tree is:"<<endl;
      bt.postorder_traversal_r(bt.root);
      break;
      case 5:
      cout<<"The Inorder Non Recursive Display of the tree is:"<<endl;
      bt.inorder_traversal_non_recursive(bt.root);
      break;
      case 6:
      cout<<"The Preorder Non Recursive Display of the tree is:"<<endl;
      bt.preorder_traversal_non_recursive(bt.root);
      break;
      case 7:
      cout<<"The Postorder Non Recursive Display of the tree is:"<<endl;
      bt.postorder_traversal_non_recursive(bt.root);
      break;
      default:
      cout<<"Invalid choice";
      break;
    }
```

```
233
234         cout << "Do you want to continue(y/n):";
235         cin>> c;
236       }while (c == 'y');
237
238
239       return 0;
240     }
```

Listing 1: Input

```
1     What you want to do:
2   1. Enter Tree
3   2.Inorder Recursive
4   3.Preorder Recursive
5   4.Postorder Recursive
6   5.Inorder Non Recursive
7   6.Preorder Non Recursive
8   7.Postorder Non Recursive
9   choose:
10  1
11
12  Enter ROOT Data:
13  1
14  Enter whether you want to add the data to the left of "1" or not (y/n):  y
15  Enter the data for the left of "1" node: 2
16  Enter whether you want to add the data to the left of "2" or not (y/n):  y
17  Enter the data for the left of "2" node: 3
18  Enter whether you want to add the data to the left of "3" or not (y/n):  y
19  Enter the data for the left of "3" node: 4
20  Enter whether you want to add the data to the left of "4" or not (y/n): y
21  Enter the data for the left of "4" node: 5
22  Enter whether you want to add the data to the left of "5" or not (y/n):  n
23  Enter whether you want to add the data to the right of "5" or not (y/n): n
24  Enter whether you want to add the data to the right of "4" or not (y/n): n
25  Enter whether you want to add the data to the right of "3" or not (y/n): n
26  Enter whether you want to add the data to the right of "2" or not (y/n): n
27  Enter whether you want to add the data to the right of "1" or not (y/n): n
28
29  Do you want to continue(y/n): y
30
31  What you want to do:
32  1. Enter Tree
33  2.Inorder Recursive
34  3.Preorder Recursive
35  4.Postorder Recursive
36  5.Inorder Non Recursive
37  6.Preorder Non Recursive
38  7.Postorder Non Recursivechoose:
39  2
40  The inorder display of the tree is:
41  5 4 3 2 1
42  Do you want to continue(y/n): y
43
44  What you want to do:
45  1. Enter Tree
46  2.Inorder Recursive
47  3.Preorder Recursive
48  4.Postorder Recursive
49  5.Inorder Non Recursive
50  6.Preorder Non Recursive
51  7.Postorder Non Recursive
52  choose:
53  3
54  The preorder display of the tree is:
55  1 2 3 4 5
56  Do you want to continue(y/n): y
57
```

```
What you want to do:
1. Enter Tree
2.Inorder Recursive
3.Preorder Recursive
4.Postorder Recursive
5.Inorder Non Recursive
6.Preorder Non Recursive
7.Postorder Non Recursive
choose:
4
The postorder display of the tree is:
5 4 3 2 1
Do you want to continue(y/n): y

What you want to do:
1. Enter Tree
2.Inorder Recursive
3.Preorder Recursive
4.Postorder Recursive
5.Inorder Non Recursive
6.Preorder Non Recursive
7.Postorder Non Recursive
choose:
5
The Inorder Non Recursive Display of the tree is:
5 4 3 2 1
Do you want to continue(y/n): y

What you want to do:
1. Enter Tree
2.Inorder Recursive
3.Preorder Recursive
4.Postorder Recursive
5.Inorder Non Recursive
6.Preorder Non Recursive
7.Postorder Non Recursive
choose:
 6
The Preorder Non Recursive Display of the tree is:
1 2 3 4 5
Do you want to continue(y/n): y

What you want to do:
1. Enter Tree
2.Inorder Recursive
3.Preorder Recursive
4.Postorder Recursive
5.Inorder Non Recursive
6.Preorder Non Recursive
7.Postorder Non Recursive
choose:
7
The Postorder Non Recursive Display of the tree is:
5 4 3 2 1
Do you want to continue(y/n): n
```

Listing 2: output

5 Conclusion

Thus, implemented different operations on CLL.

6 FAQ's

1. Explain any one application of binary tree with suitable example.

   Ans. One common application of binary trees is in the implementation of search algorithms, such as binary search.

   In a binary search algorithm, a sorted list of elements is represented as a binary tree, with the value of the root node being the middle element of the list. If the value being searched for is less than the root node, the search continues in the left subtree. If the value being searched for is greater than the root node, the search continues in the right subtree. This process continues recursively until the value is found or it is determined that the value is not in the list.

   For example, consider the following sorted list of integers: [2, 5, 7, 9, 12, 15, 20]. To perform a binary search for the value 12, we can represent the list as a binary tree as follows:

```
            12
           /  \
          7    15
          / \   /
        5   9 20
        /
      2

```

Listing 3: Example

   Starting at the root node, 12, we see that the value 12 is equal to the root node, so the search is successful and we return the index of the node. If we were searching for the value 7, we would follow the left subtree and continue the search, until we find the node with value 7 and return its index.

2. Explain sequential representation of binary tree with example.

   Ans. A binary tree can be represented in a sequential manner, also known as an array representation, where the tree nodes are stored in an array in a specific order. This representation is commonly used when implementing binary trees in programming languages.

   In the sequential representation, the root node is stored at the first position in the array (index 0), and its children are stored at the next two positions (index 1 and 2). The children of the node at position i are stored at positions 2i + 1 and 2i + 2, where i is the index of the node.

   Here is an example of a binary tree and its sequential representation:

```
  18
  /  \
            15     30
           / \           / \
      40 50 100 40

```

Listing 4: Example

   Array representation: [18, 15, 30, 40, 50, 100, 40]

   In this example, the root node 18 is stored at index 0, its left child 15 is stored at index 1, and its right child 30 is stored at index 2. The left child of 15 is stored at index 3, the right child of 15 is stored at index 4, the left child of 30 is stored at index 5, and the right child of 30 is stored at index 6.

This sequential representation allows us to access the nodes of the binary tree in an efficient manner, as we can calculate the position of a node's children and parent based on its index in the array.

3. Write inorder, preorder and postorder for following tree.

Ans.

- Inorder of the tree is: 8 40 7 15 9 50 18 100 30 40
- Postorder of the tree is: 8 7 40 9 50 15 100 40 30 18
- Preorder of the tree is: 18 15 40 8 7 50 9 30 100 40

MIT World Peace University
Advanced Data Structures

Assignment 3

Naman Soni Roll No. 10

Contents

1 Problem Statement

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

- Insert a keyword
- Delete a keyword
- Create mirror image and display level wise
- Copy

2 Objectives

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

3 Theory

3.1 *Binary Search Tree*

A binary search tree is a type of data structure used to store a collection of elements in a sorted order. It is called a "binary" tree because each node in the tree has at most two children: a left child and a right child.

The structure of the tree is such that the value of any node in the left subtree is less than the value of its parent node, and the value of any node in the right subtree is greater than the value of its parent node. This property makes it possible to search for an element in the tree efficiently, by repeatedly comparing the target value with the value of the current node and then traversing either the left or right subtree accordingly.

Binary search trees can be used for a variety of operations, including insertion, deletion, and searching for elements. They are commonly used in computer science for implementing algorithms like binary search and quicksort. However, if the tree is not balanced (i.e. if one subtree is much deeper than the other), the performance of these operations can degrade and lead to inefficiencies.

3.2 *Breadth First Traversal*

Breadth First Traversal, also known as level order traversal, is a tree traversal algorithm that visits all the nodes of a tree in a breadth-first manner, i.e. it visits all the nodes at each level before moving on to the next level.

Starting from the root node, the algorithm visits all the nodes at level 1 (i.e. the immediate children of the root node) from left to right, then visits all the nodes at level 2 (i.e. the children of the nodes at level 1) from left to right, and so on until all the nodes in the tree have been visited.

The algorithm uses a queue data structure to keep track of the nodes to be visited, and a visited set to keep track of the nodes that have already been visited. Initially, the root node is added to the queue and marked as visited. Then, while the queue is not empty, the algorithm dequeues the next node in the queue, visits it, and enqueues all its unvisited children (if any). The algorithm continues until the queue is empty, at which point all nodes in the tree will have been visited in breadth-first order.

Breadth First Traversal has a time complexity of O(n), where n is the number of nodes in the tree, as it visits each node exactly once. It is commonly used in graph algorithms, such as finding the shortest path between two nodes in an unweighted graph.

3.3 *Different operations on binary search tree.(copy ,mirror image and delete)*

Here are brief descriptions of different operations on binary search trees:

1. Copy: To copy a binary search tree, we need to create a new tree and copy all the nodes of the original tree into the new tree. We can use either recursive or iterative approach for this operation. In the recursive approach, we start from the root node of the original tree and copy it into the new tree. Then, we recursively copy the left and right subtrees of the root node. In the iterative approach, we use a stack to traverse the original tree and create a new node for each visited node.

2. Mirror Image: To create a mirror image of a binary search tree, we need to swap the left and right subtrees of each node in the tree. This can be done recursively by traversing the tree and swapping the subtrees of each node.

3. Delete: To delete a node from a binary search tree, we first need to find the node to be deleted. We then need to handle three cases:

   - If the node has no children, we simply remove it from the tree.
   - If the node has one child, we replace it with its child.
   - If the node has two children, we find the node with the smallest value in its right subtree (or the node with the largest value in its left subtree), copy its value to the node to be deleted, and then delete the node with the smallest value (or largest value).

   We can use a recursive or iterative approach for this operation. In the recursive approach, we start from the root node and recursively search for the node to be deleted. In the iterative approach, we use a loop to traverse the tree and find the node to be deleted.

4 Implementation

4.1 *Platform*

Visual Studio Code
Mac OS 64 bit

5 Test Conditiions

- Input at least 10 nodes.
- Display binary search tree levelwise traversals of binary search tree with 10 nodes
- Display mirror image and copy operations on BST

6 Conclusion

Thus, implemented Dictionary using Binary search tree.

7 FAQ's

(a) Explain application of BST?

Ans. Binary Search Trees (BSTs) have a wide range of applications in computer science some of them are:

- Searching: BSTs are commonly used to perform searching operations efficiently. Because of the tree structure of BSTs and their ordering property, it is possible to search for a specific element in O(log n) time, where n is the number of elements in the tree.
- Sorting: BSTs can be used to sort a set of elements efficiently. By inserting elements into a BST in sorted order and then performing an in-order traversal of the tree, we can retrieve the elements in sorted order.
- Symbol table: A symbol table is a data structure that is used to store key-value pairs, and it is commonly used in compilers, interpreters, and other software systems. BSTs can be used to implement symbol tables efficiently, with operations such as insert, delete, and search running in O(log n) time.
- File system: BSTs can be used to implement file systems efficiently, with directories and files represented as nodes in the tree. By maintaining the ordering property of the BST, we can quickly search for files and navigate through the directory structure.
- Network routing: BSTs can be used to implement network routing tables efficiently, with IP addresses and routing information stored as key-value pairs in the tree. By searching the tree for a given IP address, we can quickly determine the appropriate route for a packet to take.

(b) Explain with example deletion of a node having two child.?

Ans. Deleting a node with two children from a binary search tree can be a bit more complex than deleting a node with no children or one child. Here is an example of how to delete a node with two children:

- Suppose we have the following binary search tree:

We want to delete the node with key value 10. This node has two children, so we need to find the node with the smallest key value in its right subtree to replace it. Here are the steps we can follow:

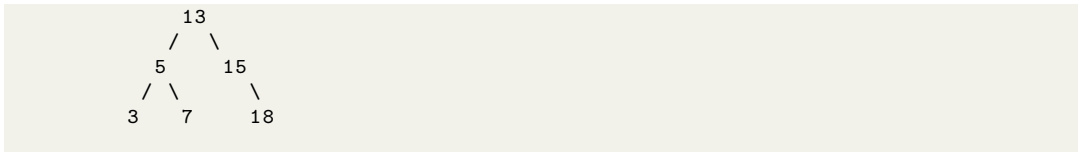
Find the node with the smallest key value in the right subtree of the node to be deleted. In this case, the right subtree of 10 is:

The smallest key value in this subtree is 13.

Copy the key value of the node we found in step 1 (13) to the node we want to delete (10)

Delete the node we found in step 1 (13). Because it has no children, we can simply remove it from the tree.

The resulting tree looks like this:



- Define skewed binary tree.?

  Ans. A skewed binary tree is a type of binary tree in which all of the nodes are either left or right children of their parent. In other words, the tree is "lopsided" in one direction, with all of the nodes either slanting to the left or slanting to the right.

  There are two types of skewed binary trees:

  – Left-skewed binary tree: In a left-skewed binary tree, all of the nodes have a left child, except for the leaf nodes, which have no children. An example of a left-skewed binary tree is:

```
1
 \
  2
   \
    3
     \
      4
       \
        5
```

  – Right-skewed binary tree: In a right-skewed binary tree, all of the nodes have a right child, except for the leaf nodes, which have no children. An example of a right-skewed binary tree is:

```
        1
       /
      2
     /
    3
   /
  4
 /
5
```

  Skewed binary trees can be unbalanced and can lead to inefficient operations such as searching and inserting, as the time complexity of these operations may be O(n) in the worst case, where n is the number of nodes in the tree. Therefore, balancing techniques such as rotation and rebalancing may be applied to improve the performance of skewed binary trees.

MIT World Peace University
Advanced Data Structures

Assignment 4

Naman Soni Roll No. 10

Contents

1 Problem Statement

Implement threaded binary tree and perform inorder traversal.

2 Objective

- To study the data Structure : Threaded Binary Tree
- To study the advantages of Threaded Binary Tree over Binary Tree

3 Theory

3.1 *The data structure : Threaded Binary Tree*

A Threaded Binary Tree is a modified version of a binary tree where each node has a reference or pointer to its in-order predecessor or successor, called threaded links. These threaded links make traversal of the tree more efficient, as they eliminate the need for recursive function calls and stack space for storing nodes.

In a threaded binary tree, the null pointers of leaf nodes are replaced by pointers to their in-order predecessor and successor. For a node that has a left child, its predecessor link points to the maximum node in its left subtree. For a node that has a right child, its successor link points to the minimum node in its right subtree.

There are two types of threaded binary trees: singly threaded and doubly threaded. In a singly threaded tree, each node has either a predecessor or a successor thread, while in a doubly threaded tree, each node has both predecessor and successor threads.

Threaded binary trees are useful in applications where traversal of the tree is a common operation, such as in database indexing and searching, and they can save both memory and computational resources. However, they are more complicated to implement than standard binary trees, and the threaded links must be maintained when nodes are inserted or removed from the tree.

3.2 *Space Utilization in Threaded Binary Tree*

Space utilization in a threaded binary tree is a measure of how efficiently the memory is used to store the tree structure. Compared to a regular binary tree, a threaded binary tree can potentially save space by using the null pointers of leaf nodes to store threaded links to their in-order predecessors or successors.

In a singly threaded binary tree, each leaf node has one threaded link, which replaces one null pointer. In a doubly threaded binary tree, each leaf node has two threaded links, which replace two null pointers. This means that a threaded binary tree can potentially use up to 50 % less memory than a regular binary tree.

However, the actual space savings depend on the size and shape of the tree, as well as the implementation of the threaded links. In some cases, the overhead of maintaining the threaded links may outweigh the space savings.

Overall, space utilization is an important consideration when designing and implementing a threaded binary tree, and it is important to balance the benefits of space savings with the cost of additional complexity in the implementation.

4 Implementation

4.1 *Platform*

- 64-bit Mac OS
- Open Source C++ Programming tool like Visual Studio Code

4.2 *Test Conditions*

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

5 Conclusion

Thus, implemented threaded binary tree with inorder traversal.

6 FAQ's

6.1 *Why TBT can be traversed without stack?*

A Threaded Binary Tree (TBT) can be traversed without using a stack because of the presence of the threaded links between nodes. These links provide a way to navigate the tree without having to use a recursive function call or a stack to keep track of the nodes.

For example, in-order traversal of a TBT involves visiting the left subtree, the current node, and the right subtree in that order. To traverse a TBT in-order without a stack, we start at the root node and follow the left pointers until we reach a node with a null left pointer, which means we have reached the leftmost node. At this point, we visit the node, and then follow its successor link to the next node in the in-order sequence. We repeat this process until we reach the rightmost node, which has a null successor link.

By using the threaded links instead of a stack, we can traverse the tree without incurring the overhead of a recursive function call or the additional memory required to store nodes on the stack. This can make the traversal more efficient, especially for large or deep trees, where the stack space required for recursive traversal can be significant.

6.2 *What are the advantages and disadvantages of TBT?*

Advantages of Threaded Binary Trees (TBT's):

1. Improved traversal efficiency: TBTs allow for in-order traversal without using a stack, which can improve traversal efficiency by reducing memory usage and eliminating the need for recursive function calls.
2. Space optimization: TBTs can potentially use less memory than regular binary trees by using null pointers in leaf nodes to store threaded links to their in-order predecessors or successors.
3. Fast searching: TBTs can be used for searching and sorting, and their threaded links can make these operations faster by reducing the number of comparisons needed to find a specific node.

Disadvantages of Threaded Binary Trees (TBT's)

1. More complex implementation: Implementing TBTs requires additional logic to maintain the threaded links when nodes are inserted or removed from the tree. This can make the implementation more complex and error-prone than regular binary trees.

2. Increased overhead: Maintaining the threaded links can add overhead to the insertion and removal operations, which may reduce the overall performance of the tree.

3. Limited flexibility: TBTs are specifically designed for in-order traversal and may not be well-suited for other types of tree operations or data structures.

6.3 *Write application of TBT*

Threaded Binary Trees (TBTs) can be used in a variety of applications, including:

1. In-order traversal: TBTs were originally designed for efficient in-order traversal of binary trees. TBTs can be used to traverse the tree without using a stack, which can be useful for large or deep trees where stack space is a concern.

2. Searching and sorting: TBTs can be used for searching and sorting operations, where their threaded links can improve performance by reducing the number of comparisons needed to find a specific node.

3. Memory optimization: TBTs can potentially use less memory than regular binary trees by using null pointers in leaf nodes to store threaded links to their in-order predecessors or successors. This can be useful in memory-constrained environments, such as embedded systems or mobile devices.

4. Data compression: TBTs can be used in data compression algorithms, such as Huffman coding, where they can be used to efficiently represent the frequency of characters in a text string.

5. Expression evaluation: TBTs can be used in expression evaluation, where they can be used to represent and evaluate arithmetic expressions, such as postfix notation or expression trees.

MIT World Peace University
Advanced Data Structures

Assignment 5

Naman Soni Roll No. 10

Contents

1 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.

2 Objective

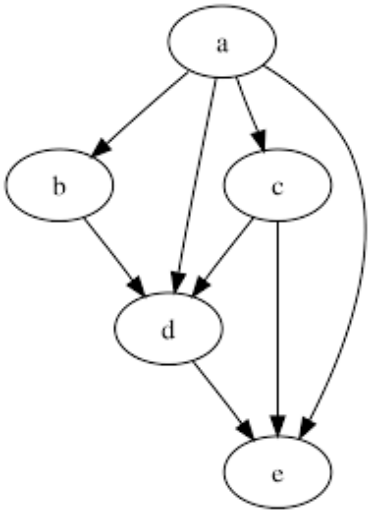
1. To study data structure Graph and its representation using adjacency list
2. To study and implement recursive Depth First Traversal and use of stack data structure for recursive Depth First Traversal
3. To study and implement Breadth First Traversal
4. To study how graph can be used to model real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

3 Theory

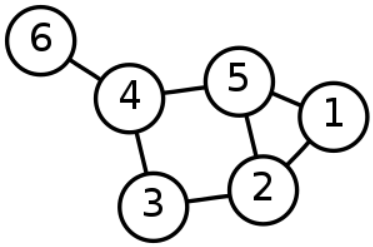
3.1 *Graph and its types*

Graphs are used to represent complex relationships between data elements. Graphs can be directed or undirected, and can have weighted edges or unweighted edges.
Some common operations on graphs include traversal (visiting all the vertices in the graph), shortest path finding (finding the shortest path between two vertices), and minimum spanning tree finding (finding a subset of edges that form a tree that connects all vertices in the graph with minimum total weight).
Graph algorithms are used in many areas of computer science, such as network optimization, social network analysis, computational biology, and artificial intelligence. Some common graph algorithms include Breadth-First Search, Depth-First Search, Dijkstra's Algorithm, and Kruskal's Algorithm.
There are 5 types of graphs:

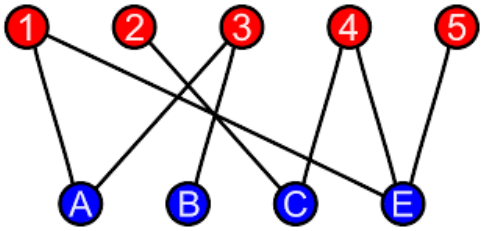
- **Directed Acyclic Graph (DAG):** A DAG is a directed graph that contains no directed cycles. That is, it is impossible to start at any vertex and follow a sequence of edges that eventually loops back to the starting vertex. DAGs are used to model dependencies between tasks or events, and are commonly used in scheduling and resource allocation problems.



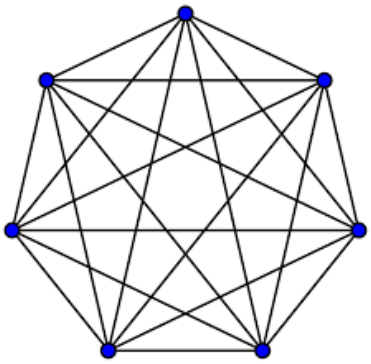
- **Weighted Graph:** A weighted graph is a graph in which each edge is assigned a numerical weight or cost. Weighted graphs are used to represent many types of real-world networks, such as transportation networks, social networks, and communication networks. They are also used in optimization problems, such as the shortest path problem and the minimum spanning tree problem.



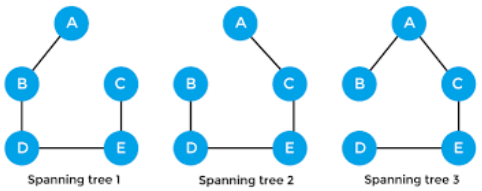
- **Bipartite Graph:** A bipartite graph is a graph whose vertices can be divided into two disjoint sets, such that every edge connects a vertex from one set to a vertex from the other set. Bipartite graphs are used to represent many types of relationships, such as buyers and sellers in a market, students and classes in a university, and books and authors in a library.



- **Complete Graph:** A complete graph is a graph in which every pair of distinct vertices is connected by an edge. Complete graphs are used to model many types of networks, such as social networks, communication networks, and transportation networks. They are also used in optimization problems, such as the traveling salesman problem.



- **Spanning Tree:** A spanning tree of a connected graph is a tree that includes all of the graph's vertices and a subset of its edges, such that the tree is connected and acyclic. Spanning trees are used to represent many types of relationships, such as hierarchies in organizations and networks of roads or pipelines.



3.2 *Representation of graph using adjacency list with one example and diagram.*

One common way to represent a graph in computer science is using an adjacency list. In this representation, each vertex in the graph is associated with a list of its neighboring vertices. Here's an example of a graph and its corresponding adjacency list representation:

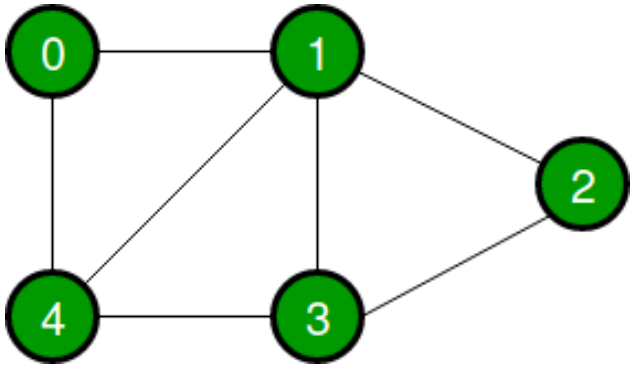


Figure 1: Undirected Graph

An array of linked lists is used. The size of the array is equal to the number of vertices. Let the array be an array[ ]. An entry array[i] represents the linked list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

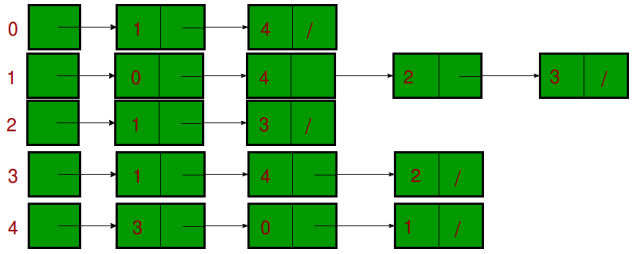


Figure 2: Adjacency List

3.3 *Graph Traversals DFT and BFT with example and diagrams*

Graph traversal refers to the process of visiting all vertices in a graph. There are two common methods for traversing a graph: Depth-First Traversal (DFT) and Breadth-First Traversal (BFT).

Depth-First Traversal: In DFT, we start at a vertex and visit as far as possible along each branch before backtracking. This process continues until all vertices have been visited. Here's an example of DFT on a graph:

```
    2
   / \
  0   3
  | \  \
  1  4  5
```

Starting at vertex 2, we follow the first branch until we reach vertex 0. From there, we continue exploring along the first branch until we reach vertex 1. Since vertex 1 has no more unexplored neighbors, we backtrack to vertex 0 and continue along the second branch, visiting vertex 4. Finally, we backtrack to vertex 2 and continue along the second branch, visiting vertex 3 and then vertex 5.

The order in which we visited the vertices in this example is: '2, 0, 1, 4, 3, 5'.

Here's a diagram illustrating the DFT process:

```
    Step 1:
        2
       / \
      0   3
      | \  \
      1  4  5

Visited: 2

Step 2:
        2
       / \
      0   3
      | \  \
      1  4  5

Visited: 2, 0

Step 3:
        2
       / \
      0   3
      | \  \
      1  4  5

Visited: 2, 0, 1

Step 4:
        2
       / \
      0   3
      | \  \
      1  4  5
Visited: 2, 0, 1, 4
Step 5:
        2
       / \
      0   3
      | \  \
      1  4  5

Visited: 2, 0, 1, 4, 3

Step 6:
        2
       / \
      0   3
      | \  \
      1  4  5

Visited: 2, 0, 1, 4, 3, 5
```

Breadth-First Traversal: In BFT, we start at a vertex and visit all of its neighbors before moving on to any of their neighbors. This process continues until all vertices have been visited. Here's an example of BFT on a graph:

Starting at vertex 2, we visit all its neighbors first: vertices 0 and 3. Then, we visit all the neighbors of vertices 0 and 3: vertices 1, 4, and 5. The order in which we visited the vertices in this example is: '2, 0, 3, 1, 4, 5'.
Here's a diagram illustrating the BFT process:

```
    Step 1:
        2
       / \
      0   3
      |    \
      1     5
           /
          4

Visited: 2

Step 2:
        2
       / \
      0   3
      |    \
      1     5
           /
          4

Visited: 2, 0, 3

Step 3:
        2
       / \
      0   3
      |    \
      1     5
           /
          4

Visited: 2, 0, 3, 1, 4

Step 4:
        2
       / \
      0   3
      |    \
      1     5
           /
          4

Visited: 2, 0, 3, 1, 4, 5
```

4 Implementation

4.1 *Platform*

- 64-bit Mac OS
- Open Source C++ Programming tool like Visual Studio Code

4.2 *Test Conditions*

1. Input at least 5 nodes.

2. Display DFT (recursive and non recursive)and BFT

5 Conclusion

Thus, we have represented graph using adjacency list and performed DFT and BFT on it. We have also discussed the time complexity of the algorithms.

6 FAQ's

6.1 *Explain two applications of graph.*

Ans. Graphs are a powerful data structure that can be used to model and solve a wide variety of problems. Here are two common applications of graphs:

1. **Social Networks:** Graphs are widely used to model social networks such as Facebook, LinkedIn, and Twitter. In a social network, each user is represented as a node in the graph, and the relationships between users (such as friendship, following, or connection) are represented as edges. This graph can be used to analyze the structure of the social network, identify influential users or communities, and suggest new connections or friends for a user.

2. **Routing and Navigation:** Graphs can also be used to model road networks, transportation systems, and other types of networks where routes need to be optimized. In this case, each intersection or location is represented as a node in the graph, and the roads or connections between them are represented as edges. Using graph algorithms such as Dijkstra's algorithm or the A* algorithm, it is possible to find the shortest path or fastest route between two locations on the network. This can be used for navigation systems, logistics optimization, and other applications where efficient routing is important.

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

Ans. Adjacency matrix and adjacency list are two common ways to represent graphs in computer science. Here are some advantages of using an adjacency list over an adjacency matrix:

1. Space efficiency: Adjacency lists are more space-efficient than adjacency matrices, especially for sparse graphs. In an adjacency matrix, we have to allocate space for every possible edge, even if the edge does not exist in the graph. For a sparse graph, this can lead to a lot of wasted space. In contrast, an adjacency list only stores the edges that actually exist in the graph, making it more space-efficient.

2. Efficient iteration over neighbors: In an adjacency list, it is easy to iterate over the neighbors of a given vertex. We just need to iterate over the list of edges connected to that vertex. In contrast, in an adjacency matrix, we have to iterate over an entire row or column to find the neighbors of a vertex, which can be slower for large graphs.

3. Easy addition and removal of edges: In an adjacency list, it is easy to add or remove edges from the graph. We just need to add or remove an edge from the list of edges for each vertex. In contrast, in an adjacency matrix, we have to update the entire row and column for each vertex whenever we add or remove an edge, which can be slower for large graphs.

4. Memory allocation: Adjacency list requires memory allocation only for edges that exist, thus it is more flexible for dynamic allocation of memory.

6.3 *Why transversal in graph is different than traversal in tree*

Ans. Traversal in a graph is different from traversal in a tree for several reasons:

1. Multiple paths: In a tree, there is only one unique path from the root to any leaf node. However, in a graph, there can be multiple paths between two vertices. This means that a traversal algorithm for a graph must be able to handle the possibility of visiting a vertex multiple times along different paths.

2. Loops and cycles: Graphs can contain loops or cycles, which means that a traversal algorithm may encounter the same vertex or edge more than once during the traversal. In contrast, trees do not contain loops, so a traversal algorithm for a tree does not need to check for cycles.

3. Connected components: A graph can be disconnected, meaning that there are two or more separate subgraphs that are not connected by any edges. In this case, a traversal algorithm must be able to visit each connected component separately. In contrast, a tree is always a single connected component.

4. Undefined root: A tree always has a well-defined root node, but a graph does not. This means that a traversal algorithm for a graph must be able to start from any vertex in the graph.

MIT World Peace University
Advanced Data Structures

Assignment 6

Naman Soni Roll No. 10

Contents

1 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.

2 Objective

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

3 Theory

3.1 *What is Spanning tree. Explain with example.*

A spanning tree is a subset of edges in an undirected graph that connects all vertices in the graph without forming any cycles. In other words, it is a tree that spans all vertices in the graph.

A spanning tree can be obtained from any connected undirected graph. The spanning tree will have the same number of vertices as the original graph, but a smaller number of edges. The minimum number of edges required to form a spanning tree is n-1, where n is the number of vertices in the graph.

Here is an example:

Consider the following undirected graph:

```
   A
  / \
   B - C
  / \ / \
 D - E - F
```

To obtain a spanning tree from this graph, we can start with any vertex (let's say vertex A) and perform a depth-first search or breadth-first search. As we visit each vertex, we add the edges that connect that vertex to its unvisited neighbors to the spanning tree. We continue this process until we have visited all vertices in the graph.

One possible spanning tree for this graph is:

```
  A
  \
   B - C
  /   /
   D - E
    \
     F
```

3.2 *Example of weighted graph and its cost adjacency matrix (with diagrams)*

A weighted graph is a graph in which each edge has a numerical weight assigned to it. The weight can represent any kind of cost, distance, or value associated with the edge. Here is an example of a weighted graph and its cost adjacency matrix:

Consider the following weighted graph:

```
  2
  A --- B
  |     | \
   3|     |1 \4
  |     |   \
  C --- D --- E
     5     2
```

In this graph, each edge is labeled with its weight. For example, the edge between A and B has weight 2, and the edge between B and D has weight 4.

To represent the weights in the form of a cost adjacency matrix, we can create a matrix where each row and column corresponds to a vertex in the graph. The entry in row i and column j represents the cost of the edge connecting vertex i to vertex j, if such an edge exists. If there is no edge between i and j, the entry is typically set to infinity.

For the above graph, the cost adjacency matrix would be:

```
  A    B    C    D    E
  +-----------------------
  A |   0    2    3   Inf  Inf
  B |   2    0  Inf    4    1
  C |   3  Inf   0    5  Inf
  D | Inf   4    5    0    2
  E | Inf   1  Inf    2    0
```

The entries corresponding to edges with weights are filled in with the weight values, while the entries corresponding to non-existent edges are set to infinity. For example, the entry in row A and column E is Inf, since there is no direct edge between A and E.

This cost adjacency matrix can be used to perform various graph algorithms that require knowledge of edge weights, such as Dijkstra's shortest path algorithm or Prim's minimum spanning tree algorithm.

3.3 *Explanation of Prim's algorithm using the above example graph*

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. The algorithm starts with a single vertex and then adds the edge of minimum weight that connects the tree to a vertex that is not yet in the tree. It repeats this process until all vertices are in the tree.

Here's how Prim's algorithm would work on the example graph given in the previous question:

```
  2
  A --- B
  |     | \
   3|     |1 \4
  |     |   \
  C --- D --- E
     5     2
```

1. Choose a starting vertex, say A, and add it to the minimum spanning tree (MST). The MST now contains only one vertex: A.

```
  2
  A --- B
  |     | \
   3|     |1 \4
  |     |   \
  C --- D --- E
     5     2

  {A}
```

2. Find the minimum weight edge that connects any vertex in the MST to a vertex that is not yet in the MST. In this case, the minimum weight edge is the edge between A and B, with weight 2. Add this edge to the MST.

```
  2
A --- B
      \
     4 \
        \
C --- D --- E
   5     2

{A, B}
```

3. Find the minimum weight edge that connects any vertex in the MST to a vertex that is not yet in the MST. In this case, there are two options: the edge between B and D with weight 4, and the edge between B and E with weight 1. Choose the edge between B and E since it has the smaller weight. Add this edge to the MST.

```
  2
A --- B
      \
     1 \
        \
C --- D --- E
   5     2

{A, B, E}
```

4. Find the minimum weight edge that connects any vertex in the MST to a vertex that is not yet in the MST. In this case, there are two options: the edge between B and D with weight 4, and the edge between D and C with weight 5. Choose the edge between D and C since it has the smaller weight. Add this edge to the MST.

```
  2
A --- B
      \
     1 \
        \
C --- D    E
   5     2

{A, B, E, D, C}
```

5. All vertices are now in the MST, so the algorithm terminates. The minimum spanning tree for the given graph is:

```
  2
A --- B
      \
     1 \
        \
C --- D    E
```

The minimum spanning tree has total weight 5+2+1=8, which is the sum of the weights of the three edges in the tree.

4 Implementation

4.1 *Platform*

- 64-bit Mac OS
- Open Source C++ Programming tool like Visual Studio Code

subsection*Test Conditions*

1. Input at least 5 nodes.
2. Display Minimum cost spanning tree and Minimum cost for the graph.

5 Conclusion

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

6 FAQ's

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

Minimum cost spanning trees (MSTs) have a wide range of applications in various fields. Here are two common applications:

1. Network Design: In network design, MSTs are used to minimize the total cost of constructing a communication network, while ensuring that all nodes are connected. This can be useful in designing electrical power grids, telecommunications networks, computer networks, and transportation systems. By finding an MST for a given set of nodes and edges, we can determine the minimum cost of constructing a network that connects all nodes.

2. Clustering: MSTs can also be used for clustering analysis in data mining and machine learning. In this application, MSTs are used to identify clusters or groups of similar data points based on their pairwise distances. The MST is constructed by treating the data points as nodes in the graph, and the pairwise distances between them as the weights of the edges. The MST can then be used to identify clusters of data points that are closely related to each other, based on the distance between them. This can be useful in image segmentation, customer segmentation, and anomaly detection.

Overall, MSTs are a powerful tool in graph theory that have numerous practical applications in many fields.

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

Prim's algorithm and Kruskal's algorithm are two popular algorithms used to find a minimum cost spanning tree for a weighted undirected graph. Here is a small example graph to explain the difference between the two algorithms:

Prim's Algorithm: Prim's algorithm starts with a single vertex and grows the minimum spanning tree by adding the edge with the minimum weight that connects the tree to a vertex that is not yet in the tree.

- Choose an arbitrary vertex, say A, and add it to the minimum spanning tree (MST). The MST now contains only one vertex: A.

- Find the minimum weight edge that connects any vertex in the MST to a vertex that is not yet in the MST. In this case, the minimum weight edge is the edge between A and B, with weight 2. Add this edge to the MST.

- Find the minimum weight edge that connects any vertex in the MST to a vertex that is not yet in the MST. In this case, the minimum weight edge is the edge between B and D, with weight 1. Add this edge to the MST.

- Find the minimum weight edge that connects any vertex in the MST to a vertex that is not yet in the MST. In this case, the minimum weight edge is the edge between D and E, with weight 4. Add this edge to the MST.

- Find the minimum weight edge that connects any vertex in the MST to a vertex that is not yet in the MST. In this case, the minimum weight edge is the edge between B and C, with weight 3. Add this edge to the MST.

The final MST obtained by Prim's algorithm is:

```
   2      3
A ------ B ------ C
      |
      |5
      |
      D ------ E
          4
```

Kruskal's Algorithm: Kruskal's algorithm starts by sorting all the edges in the graph by weight and then iteratively adding the edges with the minimum weight that do not create a cycle in the MST.

- Sort all the edges in the graph by weight: (B,D), (A,B), (B,C), (D,E), (C,E), (A,D).
- Pick the edge with the minimum weight, (B,D), and add it to the MST.
- Pick the next edge with the minimum weight, (A,B), and add it to the MST.
- Pick the next edge with the minimum weight, (B,C), and add it to the MST.
- Pick the next edge with the minimum weight, (D,E), and add it to the MST.
- Pick the next edge with the minimum weight, (C,E), and add it to the MST.

The final MST obtained by Kruskal's algorithm is:

```
      3
A ------ B ------ C
      |
      |5
      |
      D ------ E
```

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

Prim's algorithm uses a greedy algorithmic strategy to find the minimum cost spanning tree in a weighted undirected graph. A greedy algorithm always makes the locally optimal choice at each step with the hope of finding a global optimum.

The algorithm starts with a single vertex and grows the minimum spanning tree by adding the edge with the minimum weight that connects the tree to a vertex that is not yet in the tree. The steps for Prim's algorithm are as follows:

- Choose an arbitrary vertex, say A, and add it to the minimum spanning tree (MST). The MST now contains only one vertex: A.
- Find the minimum weight edge that connects any vertex in the MST to a vertex that is not yet in the MST. Add this edge to the MST and the vertex to the set of vertices in the MST.
- Repeat step 2 until all vertices are in the MST.

The key idea behind the algorithm is that at each step, we add the edge with the minimum weight that connects the MST to a vertex that is not yet in the MST. This ensures that the MST is always growing by the minimum amount possible at each step.

Prim's algorithm is a computationally efficient way to find the minimum cost spanning tree for a weighted undirected graph, especially when the graph is dense (i.e., has many edges). The time complexity of Prim's algorithm is O(ElogV) using a binary heap or Fibonacci heap to maintain the set of vertices that are not yet in the MST, where E is the number of edges and V is the number of vertices in the graph.

MIT World Peace University
Advanced Data Structures

Assignment 7

Naman Soni Roll No. 10

Contents

1 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.

2 Objective

- To study the concept of heap
- To study different types of heap and their algorithms

3 Theory

3.1 *What is a heap?*

A heap is a specialized tree-based data structure that is used to efficiently manage and prioritize dynamically changing data. A heap is typically used to implement a priority queue, which is a collection of elements where each element has an associated priority. The heap allows for efficient insertion and removal of elements while maintaining a specific ordering based on their priorities.

There are two types of heaps: max heaps and min heaps. In a max heap, the element with the highest priority is always at the root of the tree, while in a min heap, the element with the lowest priority is at the root. Heaps can be implemented as arrays, where the elements are stored in a specific order, or as trees, where the elements are organized in a hierarchy.

The most common operations performed on a heap include insertion of a new element, removal of an element with the highest or lowest priority, and querying the element with the highest or lowest priority. These operations can be performed in logarithmic time, which makes heaps an efficient data structure for managing large amounts of data in real-time applications such as job scheduling, network routing, and resource allocation.

3.2 *Write different types of Heaps*

There are two main types of heaps:

- Max Heap - In a max heap, the maximum element is always at the root of the tree.
- Min Heap - In a min heap, the minimum element is always at the root of the tree.
- In addition to these, there are also two variations of heaps:
- Binary Heap - A binary heap is a type of heap data structure where each node has at most two child nodes. Binary heaps are often implemented using arrays and are commonly used to implement priority queues.
- Fibonacci Heap - A Fibonacci heap is a type of heap data structure that provides faster amortized time complexity than binary heaps for some operations. It consists of a collection of trees, and the heap's structure is more relaxed than that of a binary heap. Fibonacci heaps are commonly used in algorithms such as Dijkstra's algorithm and Prim's algorithm for graph traversal and minimum spanning tree problems.

3.3 *Construction of heap and Data Structure used for creation.*

A heap can be constructed using an array-based implementation or a tree-based implementation.

In an array-based implementation, the heap is stored in an array where the first element of the array represents the root of the tree. The children of the root node are located at indices 2i+1 and 2i+2, where i is the index of the parent node. This allows for efficient traversal and manipulation of the heap, as the tree structure is implicitly defined by the indices of the array.

In a tree-based implementation, the heap is represented using a binary tree where each node has at most two child nodes. The heap property is maintained by ensuring that the parent node always has a higher priority than its children. This allows for more efficient insertion and deletion operations, as the tree structure can be easily manipulated by swapping nodes and adjusting their priorities.

The construction of a heap involves building the heap from an unordered set of elements. This is typically done using a bottom-up approach called heapify, where the elements are recursively swapped with their children until the heap property is satisfied. The heapify operation has a time complexity of O(n) and is commonly used to build a heap from an array of elements.

To summarize, the data structure used for the creation of a heap depends on the implementation approach. In an array-based implementation, the heap is stored in an array, while in a tree-based implementation, the heap is represented using a binary tree. The construction of a heap involves using the heapify operation to build the heap from an unordered set of elements.

4 Implementation

4.1 *Platform*

- 64-bit Mac OS
- Open Source C++ Programming tool like Visual Studio Code

subsection*Test Conditions*

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

5 Conclusion

Thus, we have implemented Min and Max Heap.

6 FAQ's

6.1 *Discuss with suitable example for heap sort?*

Heap sort is a sorting algorithm that uses a heap data structure to sort an array of elements. It works by building a max heap from the array and repeatedly extracting the maximum element and placing it at the end of the array until all elements are sorted. The algorithm has a time complexity of O(n log n) and is an in-place sorting algorithm, meaning it sorts the array in situ without needing extra memory.

Let's take an example to illustrate the heap sort algorithm. Consider an unsorted array of integers:

```
[10, 8, 3, 7, 1, 9, 4, 2, 6, 5]
```

Step 1: Building a Max Heap We start by building a max heap from the array. This involves rearranging the elements of the array to satisfy the heap property, where the parent node has a higher priority than its children. We can start from the middle of the array and work our way up to the root, performing the heapify operation on each node. After building the max heap, the array would look like:

```
[10, 8, 9, 7, 6, 3, 4, 2, 1, 5]
```

Step 2: Sorting the Array We repeatedly extract the maximum element from the heap and place it at the end of the array until all elements are sorted. This involves swapping the root node (which is the maximum element) with the last element of the heap, reducing the heap size by one, and performing the heapify operation on the root node to maintain the heap property. After each iteration, the sorted portion of the array grows from right to left. The array after sorting would look like:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Thus, the heap sort algorithm has successfully sorted the unsorted array in ascending order.

6.2 *Compute the time complexity of heap sort?*

The time complexity of heap sort is O(n log n), where n is the number of elements in the input array. This is because the algorithm involves two main operations - building a heap and repeatedly extracting the maximum element from the heap.

The building of the heap takes O(n) time because each node of the heap needs to be heapified. Since there are n nodes in a heap, the total time taken for building the heap is O(n).

The repeated extraction of the maximum element takes O(log n) time because it involves swapping the root element with the last element in the heap and then performing the heapify operation on the new root to restore the heap property. Since we perform this operation n times, the total time taken for sorting is O(n log n).

Therefore, the time complexity of heap sort is O(n log n) in the worst case. Heap sort has a good worst-case time complexity and is often used in situations where we need to sort large datasets in-place.

MIT World Peace University
Advanced Data Structures

Assignment 8

Naman Soni Roll No. 10

Contents

1 Problem Statement

Implement Direct access file using hashing (linear probing with and without replacement) perform following operations on it a. Create Database b. Display Database c. Add a record d. Search a record e. Modify a record

2 Objectives

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

3 Theory

3.1 *What is Hashing? Compare hashing with other searching techniques.*

Hashing is a technique of generating a unique code, called a hash, from a given input data, such as a string or a file. The hash function takes the input and returns a fixed-size hash value, which can be used to identify the data uniquely. The process of generating a hash is fast, and the resulting hash is typically much smaller than the original data.

Hashing is commonly used in computer science for searching and indexing. A hash table is a data structure that stores key-value pairs, where the keys are hashed to map to an index in an array. The value associated with the key is stored at that index. When searching for a key, the hash function is used to map the key to an index in the array, and then the value is retrieved from that index.

Compared to other searching techniques, hashing has several advantages:

- **Efficiency:** Hashing has a constant time complexity of O(1) for both insertion and retrieval. This means that the time required to search for a value is independent of the size of the data set.
- **Memory usage:** Hashing uses less memory than other search techniques such as binary search trees, since it only stores the hash values instead of the original data.
- **Collision resolution:**Hashing allows for collision resolution, which means that when two keys hash to the same index, the hash table can still store both keys and their associated values.

However, hashing also has some limitations:

- **Collisions:** If the hash function is not well-designed, it can lead to collisions, where two different keys produce the same hash value. This can result in slower performance and require more memory.
- **Unordered:** Hashing does not maintain any order between the keys. This means that if you need to retrieve the keys in a specific order, hashing may not be the best option.
- **Limited search:** Hashing is only useful for exact match searching. If you need to search for values that are close to a given value, hashing may not be the best technique.

3.2 *Write different hash functions*

Here are some examples of hash functions:

1. **Simple Hash Function:** This is a very basic hash function that adds up the ASCII values of each character in the string and returns the sum modulo some value, such as the size of the hash table.

```
    def simple_hash(s, table_size):
    sum = 0
    for i in range(len(s)):
        sum += ord(s[i])
    return sum % table_size


```

2. **Polynomial Rolling Hash Function:** This hash function is commonly used in string search algorithms. It generates a hash value by treating the string as a polynomial with a base of a prime number, such as 31, and calculates the hash value using rolling technique.

```
    def rolling_hash(s, table_size):
    p = 31  # prime number
    hash_value = 0
    power_p = 1
    for i in range(len(s)):
        hash_value = (hash_value + (ord(s[i]) - ord('a') + 1) * power_p) % table_size
        power_p = (power_p * p) % table_size
    return hash_value


```

3. **SHA-256 Hash Function:** This is a cryptographic hash function that produces a fixed-size output of 256 bits. It is widely used in security applications, such as digital signatures and message authentication.

```
    import hashlib

def sha256_hash(s):
    return hashlib.sha256(s.encode()).hexdigest()


```

4. **MD5 Hash Function:** This is another cryptographic hash function that produces a 128-bit output. It is widely used in checksums and data integrity verification.

```
    import hashlib

def md5_hash(s):
    return hashlib.md5(s.encode()).hexdigest()


```

3.3 *Explain hash collision resolution techniques.*

Hash collision occurs when two or more keys map to the same hash value or index in a hash table. To deal with hash collisions, there are various techniques that can be used for resolving collisions. Here are some of the most commonly used hash collision resolution techniques:

1. **Chaining:** In this technique, each slot in the hash table contains a linked list of elements that have the same hash value. When a collision occurs, the new element is simply added to the linked list at the appropriate slot. When searching for an element, the hash function is used to find the appropriate slot, and then a linear search is performed on the linked list at that slot. This technique is simple and works well when the hash table is large and the number of collisions is small.

2. **Open addressing:** In this technique, when a collision occurs, the algorithm looks for the next available slot in the hash table and places the element there. There are different methods of finding the next slot, such as linear probing (checking the next slot), quadratic probing (checking the next slot plus a quadratic function of the attempt number), and double hashing (using a second hash function to find the next slot). When searching for an element, the hash function is used to find the initial slot, and then the probing technique is used to find the actual slot where the element is stored. Open addressing works well when the hash table is small and the number of collisions is relatively high.

3. **Robin Hood hashing:** This is a variation of open addressing where elements are placed in a sorted order based on their distance from their original slot. When a new element collides with an existing element, the algorithm compares the distances of the elements from their original slots. If the new element is closer to its original slot, it replaces the existing element, and the existing element is moved to the new slot. If the existing element is closer, then the algorithm continues to probe for the next available slot. This technique can reduce the average probe length and improve cache performance.

4. **Cuckoo hashing:** In this technique, each element is assigned to two hash functions and two hash tables. When a collision occurs in the first hash table, the element is moved to the second hash table using its second hash function. If a collision occurs in the second hash table, the element is moved back to the first hash table using its first hash function. This process continues until either an empty slot is found or a maximum number of retries is reached. Cuckoo hashing has a constant time complexity for insertion, deletion, and lookup, but it requires more memory than other techniques.

4 Implementation

4.1 *Platform*

- 64-bit Mac OS
- Open Source C++ Programming tool like Visual Studio Code

subsection*Test Conditions*

1. Input min 10 elements.
2. Display collision with replacement and without replacement.

5 Conclusion

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

6 FAQ's

6.1 *Write different types of hash functions.*

Ans. There are several types of hash functions that can be used depending on the specific use case. Here are some of the most commonly used types:

- **Cryptographic hash functions:** These are hash functions that are designed to be very difficult to reverse or compute collisions. They are commonly used for secure communication, digital signatures, and password storage. Examples of cryptographic hash functions include SHA-256, SHA-3, and MD5.
- **Non-cryptographic hash functions:** These are hash functions that are designed for speed and efficiency rather than security. They are commonly used in data structures such as hash tables, bloom filters, and hash-based message authentication codes (HMAC). Examples of non-cryptographic hash functions include MurmurHash, Jenkins Hash, and FNV Hash.

- **Perfect hash functions:** These are hash functions that guarantee no collisions for a given set of keys. They are used when the keys are known in advance and fixed, such as in a compiler symbol table or a command-line interface. Examples of perfect hash functions include Pearson Hash, Minimal Perfect Hashing, and Cuckoo Hashing.
- **Universal hash functions:** These are hash functions that are designed to uniformly distribute keys across the hash table, reducing the likelihood of collisions. They are commonly used in open addressing and chaining collision resolution techniques. Examples of universal hash functions include Tabulation Hashing, Multiplicative Hashing, and Carter and Wegman's Hashing.
- **Pseudo-random permutation functions:** These are hash functions that are designed to behave like a random permutation of the input keys. They are commonly used in block ciphers and stream ciphers. Examples of pseudo-random permutation functions include AES, Blowfish, and ChaCha

6.2 *Explain chaining with amp; without replacement with example.*

Ans. Chaining is a collision resolution technique used in hash tables and hash maps. When two or more keys hash to the same index, chaining stores them in a linked list or another data structure at that index. There are two types of chaining: with replacement and without replacement.

Chaining with replacement: In chaining with replacement, when a collision occurs, the new key-value pair is added to the linked list at the same index as the existing key-value pairs. The linked list is updated to include the new key-value pair, and the hash table or hash map size remains constant. Here is an example of chaining with replacement:

Suppose we have a hash table with 5 slots, and we want to store the following key-value pairs:

- key: 10, value: "apple"
- key: 36, value: "orange"
- key: 49, value: "grape"
- key: 62, value: "pineapple"
- key: 75, value: "mango"

We use a hash function that maps keys to indices in the range [0,4]. Here is the result of the hashing:

- key: 10, index: 0
- key: 23, index: 3
- key: 36, index: 1
- key: 49, index: 4
- key: 62, index: 2
- key: 75, index: 0 (collision with key 10)

The resulting hash table with chaining with replacement would look like this:

Index	Linked List
0	(10, “apple”) -¿ (75, “mango”)
1	(36, “orange”)
2	(62, “pineapple”)
3	(23, “banana”)
4	(49, “grape”)

In this example, key 75 collides with key 10, which is already stored in the linked list at index 0. The new key-value pair (75, ”mango”) is added to the linked list, and the hash table size remains constant.

Chaining without replacement: In chaining without replacement, when a collision occurs, the new key-value pair is added to the next available slot in the hash table or hash map. If there are no available slots, the hash table or hash map is resized. Here is an example of chaining without replacement:

Suppose we have a hash table with 5 slots, and we want to store the same key-value pairs as in the previous example. Again, we use a hash function that maps keys to indices in the range [0,4]. Here is the result of the hashing:

- key: 10, index: 0
- key: 23, index: 3
- key: 36, index: 1
- key: 49, index: 4
- key: 62, index: 2
- key: 75, index: 0 (collision with key 10)

Since we are using chaining without replacement, the hash table needs to be resized to accommodate the new key-value pair. We double the size of the hash table to 10 slots, and the new hash values are:

- key: 10, index: 0
- key: 23, index: 3
- key: 36, index: 1
- key: 49, index: 4
- key: 62, index: 2
- key: 75, index: 5

The resulting hash table with chaining without replacement would look like this: In this example, the new

Index	Key-Value Pair
0	(10, “apple”)
1	(36, “orange”)
2	(62, “pineapple”)
3	(23, “banana”)
4	(49, “grape”)
5	(75, “mango”)
6	-
7	-
8	-
9	-

key-value pair (75, “mango”) was added to the next available slot in the resized hash table. The hash table now has twice as many slots as before, and the keys are more evenly distributed across the slots.

6.3 *Explain quadratic probing with example*

Ans. Quadratic probing is a collision resolution technique used in hash tables and hash maps. When two or more keys hash to the same index, quadratic probing stores the new key-value pair in the next available slot. The next available slot is determined by adding a quadratic function of the number of collisions to the original hash value. Here is an example of quadratic probing:

MIT World Peace University
Advanced Data Structures

Assignment 9

Naman Soni Roll No. 10

Contents

1 Problem Statement

A Dictionary stores keywords amp; 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.

2 Objective

- To study the concept of AVL trees
- To study different rotations applied on AVL tree

3 Theory

3.1 What is AVL tree?

AVL tree is a self-balancing binary search tree that was named after its inventors, Adelson-Velskii and Landis. It is a height-balanced binary search tree, which means that the height of the left and right subtrees of any node differs by at most one. The balance factor of a node is defined as the difference between the height of its left and right subtrees. If the balance factor of a node is greater than 1 or less than -1, the tree is considered unbalanced and needs to be rebalanced.

AVL tree supports efficient insertion, deletion, and lookup operations, with a worst-case time complexity of O(log n), where n is the number of nodes in the tree. This makes AVL tree a suitable data structure for applications that require frequent data manipulation and searching.

The self-balancing feature of AVL tree is achieved through rotations, which are operations that modify the tree structure while maintaining the search tree property and balance factor. There are four types of rotations that can be performed on an AVL tree: left-rotation, right-rotation, left-right-rotation, and right-left-rotation. These rotations are used to balance the tree and maintain the height balance property.

AVL tree has a few advantages over other self-balancing binary search trees, such as red-black tree. AVL tree provides faster search and insertion operations, but has a slightly slower deletion operation. Red-black tree, on the other hand, provides faster deletion operation, but has a slightly slower search and insertion operations.

Overall, AVL tree is a powerful data structure that provides efficient searching and manipulation operations, and is suitable for applications that require frequent data modification and searching.

3.2 Explain Different cases of AVL trees.

In an AVL tree, the balance factor of a node is the difference between the height of its left and right subtrees. If the balance factor of a node is greater than 1 or less than -1, the tree is considered unbalanced and needs to be rebalanced. There are four cases of AVL trees that arise when a node is inserted or deleted, and the balance factor of one or more nodes becomes greater than 1 or less than -1. These cases are as follows:

- Left-Left Case: In this case, the balance factor of a node A is greater than 1 and the balance factor of its left child B is also greater than 1. This means that the left subtree of A is taller than the right subtree, and the left subtree of B is taller than its right subtree. To rebalance the tree, we perform a right rotation on A to make B the new root of the subtree, and then perform a left rotation on B to restore the balance of the tree.
- Right-Right Case: In this case, the balance factor of a node A is less than -1 and the balance factor of its right child B is also less than -1. This means that the right subtree of A is taller than the left

subtree, and the right subtree of B is taller than its left subtree. To rebalance the tree, we perform a left rotation on A to make B the new root of the subtree, and then perform a right rotation on B to restore the balance of the tree.

- Left-Right Case: In this case, the balance factor of a node A is greater than 1 and the balance factor of its left child B is less than -1. This means that the left subtree of A is taller than the right subtree, and the right subtree of B is taller than its left subtree. To rebalance the tree, we perform a left rotation on B to make its right child C the new root of the subtree, and then perform a right rotation on A to make C the new root of the subtree and restore the balance of the tree.
- Right-Left Case: In this case, the balance factor of a node A is less than -1 and the balance factor of its right child B is greater than 1. This means that the right subtree of A is taller than the left subtree, and the left subtree of B is taller than its right subtree. To rebalance the tree, we perform a right rotation on B to make its left child C the new root of the subtree, and then perform a left rotation on A to make C the new root of the subtree and restore the balance of the tree.

By performing the appropriate rotations in each of these cases, we can maintain the balance of the AVL tree and ensure that the height of the tree is always logarithmic in the number of nodes.

3.3 Construction of AVL trees and Data Structure used for creation.

AVL trees are constructed using a recursive algorithm that inserts new nodes into the tree and ensures that the balance factor of each node is between -1 and 1. The algorithm works as follows:

- Start with an empty AVL tree.
- For each new node to be inserted, perform a standard binary search tree insertion.
- After the new node is inserted, check the balance factor of its parent node. If the balance factor is greater than 1 or less than -1, the tree is unbalanced and a rotation needs to be performed.
- Perform the appropriate rotation to rebalance the tree. There are four types of rotations: left rotation, right rotation, left-right rotation, and right-left rotation. The choice of rotation depends on the balance factor of the node and its children.
- Recursively check the balance factor of the parent node of the current node and perform rotations if necessary.
- Repeat steps 2 to 5 for all nodes to be inserted.

The data structure used for creating AVL trees is a linked list of nodes. Each node in the tree contains a key and a pointer to its left and right children. In addition, each node also contains the height of its left and right subtrees and the balance factor, which is the difference between the heights of the left and right subtrees.

To implement the rotations, additional information such as the parent of each node is also stored in the node structure. This information is used to update the pointers of the rotated nodes and their ancestors.

AVL trees are usually implemented using a self-balancing binary search tree library or module provided by the programming language. Many programming languages provide such libraries or modules, including C++, Java, Python, and many others. These libraries or modules provide efficient and optimized implementations of the AVL tree algorithms, making it easy to use AVL trees in various applications.

4 Implementation

4.1 *Platform*

- 64-bit Mac OS

- Open Source C++ Programming tool like Visual Studio Code

subsection*Test Conditions*

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

5 Conclusion

Thus, we have implemented various rotation on AVL trees

6 FAQ's

6.1 *Discuss AVL trees with suitable example?*

Sure, let's consider an example to understand how AVL trees work. Suppose we want to insert the following values into an empty AVL tree: 10, 20, 30, 40, 50, 60, and 70.

Insert 10 into the empty tree. The tree now looks like this:

```
10
```

Insert 20 into the tree. The balance factor of the root node is now -1, which is within the range of -1 to 1. So, the tree remains balanced.

```
  10
   \
   20
```

Insert 30 into the tree. The balance factor of the root node is now -2, which is less than -1. So, the tree is unbalanced and we need to perform a rotation. In this case, we need to perform a left rotation on the root node to make 20 the new root of the subtree.

```
  10
   \
   20
     \
     30
```

Insert 40 into the tree. The balance factor of the root node is now -2, which is less than -1. So, the tree is unbalanced and we need to perform a left rotation on the root node to make 30 the new root of the subtree.

```
  10
   \
   30
  /  \
 20   40
```

Insert 50 into the tree. The balance factor of the root node is now -2, which is less than -1. So, the tree is unbalanced and we need to perform a left-right rotation on the root node. First, we perform a right rotation on the left child of the root (30) to make 40 the new root of the left subtree. Then, we perform a left rotation on the root (30) to make 40 the new root of the entire subtree.

```
  10
   \
   40
  /  \
 30   50
/
20
```

Insert 60 into the tree. The balance factor of the root node is now 2, which is greater than 1. So, the tree is unbalanced and we need to perform a right rotation on the root node to make 40 the new root of the subtree.

```
    30
   /  \
  20   40
         \
          50
           \
            60
```

Insert 70 into the tree. The balance factor of the root node is now 2, which is greater than 1. So, the tree is unbalanced and we need to perform a right-left rotation on the root node. First, we perform a left rotation on the right child of the root (40) to make 50 the new root of the right subtree. Then, we perform a right rotation on the root (30) to make 50 the new root of the entire subtree.

```
    40
   /  \
  30   50
 /    /  \
20   45   60
           \
            70
```

Now, we have successfully inserted all the values into the AVL tree and ensured that the balance factor of each node is between -1 and 1. The height of the AVL tree is now logarithmic in the number of nodes, which ensures efficient searching, insertion, and deletion operations.

6.2 *Compute the time complexity of AVL tree creation?*

The time complexity of creating an AVL tree depends on the number of nodes being inserted.

When inserting a new node, we need to perform a search to find the correct position for insertion, which takes O(log n) time in a balanced tree like an AVL tree. Once we find the correct position, we need to insert the node and adjust the balance factors of the nodes along the path from the root to the newly inserted node. This adjustment involves rotations, which also take O(log n) time.

Therefore, the time complexity of creating an AVL tree with n nodes is O(n log n), since we need to perform O(log n) operations for each of the n nodes being inserted. This time complexity ensures that the AVL tree remains balanced and efficient for searching, insertion, and deletion operations.

MIT World Peace University
Advanced Data Structures

Theory Assignment

Naman Soni Roll No. 10

Contents

1 Explain AA Trees with an example.

Ans. AA trees are a self-balancing binary search tree that were invented by Arne Andersson in 1993. They are similar to red-black trees and provide efficient average-case performance for a variety of operations, including search, insertion, and deletion.

AA trees are balanced by enforcing two properties:

- **Level property:** Every leaf node has the same level, which is the distance from the root.
- **Skew property:** Every right child of a node has a level less than or equal to its parent, and every left child of a node has a level strictly less than its parent.

To maintain these properties, AA trees use two operations:

- **Skew:** If a node has a right child whose level is the same as its own, rotate the node to the left so that its right child becomes its parent.
- **Split:** If a node has two left children whose levels are the same, rotate the node to the right so that its left child becomes its parent, and then increment the level of the new right child.

Here's an example of an AA tree:

```
      7
    /   \
   3     18
  / \   /  \
 1   5 12  20
       /  /  \
      9  19  25
```

In this tree, every leaf node has the same level (4), and the skew property is satisfied because every right child has a level less than or equal to its parent, and every left child has a level strictly less than its parent.

To perform an insertion, let's insert the value 15 into the tree. First, we traverse the tree as we would in a normal binary search tree until we find the appropriate location to insert the new node:

```
      7
    /   \
   3     18
  / \   /  \
 1   5 12  20
       /  /  \
      9  19  25

Insert 15:
      7
    /   \
   3     18
  / \   /  \
 1   5 12  20
       /  /  \
      9  19  25
           \
           15
```

Now, we need to rebalance the tree to maintain the level and skew properties. We start by performing a skew operation on the new node's parent:

```
      7
    /   \
   3     18
  / \   /  \
 1   5 12  20
       /  /  \
      9  19  25
         /   \
        15   19
```

Next, we perform a split operation on the new node's grandparent:

```
      12
    /    \
   7      18
  / \    /  \
 3   9 15  20
/ \         / \
1   5       19  25
```

Now, the tree is balanced according to the level and skew properties.

2 Explain B+ Tree with an example.

Ans. A B+ tree is a type of self-balancing tree that is commonly used for indexing and organizing large amounts of data in databases and file systems. It has a similar structure to a B-tree but with some key differences, such as all data being stored in the leaf nodes and internal nodes only containing keys.

Here's an example of a B+ tree:

```
             +----+
             | 30 |
             +----+
          /          \
     +----+           +----+
     | 50 |           | 70 |
     +----+           +----+
    /      \         /      \
+----+   +----+  +----+   +----+
| 10 |   | 20 |  | 40 |   | 60 |
+----+   +----+  +----+   +----+
```

In this example, each node has a maximum of three keys and four pointers to child nodes. The tree is balanced because all paths from the root to the leaf nodes have the same length.

To perform a search in the B+ tree, we start at the root and compare the search key to the keys in the node. If the search key is less than the first key, we follow the leftmost pointer to the next node. If the search key is greater than or equal to the last key, we follow the rightmost pointer to the next node. Otherwise, we follow the pointer to the node whose key range contains the search key.

For example, if we wanted to search for the value 20 in the above tree, we would start at the root node and find that the key range of the leftmost child node is less than 20, but the key range of the second child node contains 20, so we would follow the pointer to that node:

```
             +----+
             | 30 |
             +----+
           /        \
    +----+            +----+
    | 50 |            | 70 |
    +----+            +----+
   /      \          /      \
+----+   +----+  +----+   +----+
| 10 |   | 20 |  | 40 |   | 60 |
+----+   +----+  +----+   +----+
                    ^
                    |
                search
```

Then, we would search the leaf node for the value 20 and find that it is present in the node:

```
    +----+
    | 20 |
    +----+
     /  \
```

```
  prev  next
```

To perform an insertion in the B+ tree, we start at the root and traverse the tree to find the leaf node where the new key should be inserted. If the node has room for the new key, we insert it and update the keys in the internal nodes on the path from the root to the leaf. If the node is full, we split it into two nodes and promote the median key to the parent node. This may propagate up the tree, causing additional splits and promotions until the root is reached and possibly requiring the tree to grow taller.

For example, if we wanted to insert the value 35 into the above tree, we would start at the root node and find that the key range of the rightmost child node is greater than 35, so we would follow the pointer to that node:

```
            +----+
            | 30 |
            +----+
          /        \
    +----+          +----+
    | 50 |          | 70 |
    +----+          +----+
    /      \        /      \
+----+   +----+  +----+   +----+
| 10 |   | 20 |  | 40 |   | 60 |
+----+   +----+  +----+   +----+
                   ^
                   |
                 insert
```

Since the node is full, we split it into two nodes and promote the median key (40) to the parent node:

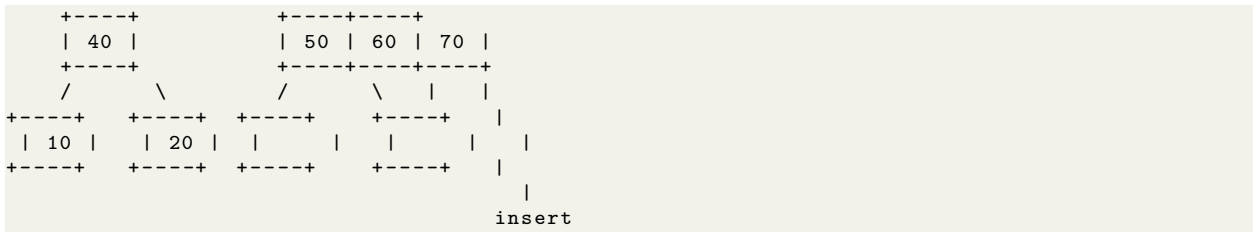
```
        +----+
        | 30 |
        +----+
      /        \
+----+          +----+----+
| 40 |          | 50 | 70 |
+----+          +----+----+
/      \           /      \
+----+   +----+  +----+   +----+
| 10 |   | 20 |  |    |   | 60 |
+----+   +----+  +----+   +----+
                    ^
                    |
                  insert
```

Now, we want to insert the value 45 into the tree. We start at the root node and find that the key range of the rightmost child node is greater than 45, so we follow the pointer to that node:

```
          +----+
          | 30 |
          +----+
        /        \
   +----+          +----+----+
   | 40 |          | 50 | 70 |
   +----+          +----+----+
  /      \        /      \   |
+----+   +----+  +----+   +----+
| 10 |   | 20 |  |    |   | 60 |
+----+   +----+  +----+   +----+
                   ^
                   |
                 insert
```

Since the node is full, we split it into two nodes and promote the median key (50) to the parent node:

```
            +----+
            | 30 |
            +----+
          /        \
```



Now, the B+ tree has been updated with the new value 45. The tree remains balanced and all paths from the root to the leaf nodes still have the same length.‘’