**ASSIGNMENT NO: 1 DATE:28.11.2023**

**Problem statement:**

Consider a sequence S of n where n ≥ 7 and implement Heap sort on S.

**Theory:**

Heap sort is one of the most efficient in-place, unstable sorting algorithm that requires only O(nlogn) operations regardless of the order of the input to sort the sequence. The main idea behind Heap sort is to maintain a Max heap (Min heap in case of descending sorting order) of size n as an almost complete binary tree of n nodes such that the content of each node is less than or equal to the content of its parent node. Now that the root has the maximum value we have to remove it from the heap. Again we have to build the max heap and remove the root element and repeat the procedure until the whole sequence is sorted in ascending order.

**Pseudo code:**

The pseudo code for heap sort is as follows:

Heapsort(a,n)

{ // a[1:n] is the input sequence containing n elements to be sorted using Heap sort

Heapify(a,n)

for i=n to 2

{ // exchanging the value of the root element with the last element

t=a[i]

a[1]=t

Adjust(a,1,i-1) // Adjusting the tree to maintain the max heap property

}

}

The pseudo code for Heapify function is as follows:

Heapify(a,n)

{ // Readjust the elements in a[1:n] to form a heap

for i=n/2 to 1

{

Adjust(a,i,n)

}

}

The pseudo code for Adjust functions is as follows:

Adjust(a,i,n)

{ // The complete binary trees with roots 2i and 2i+1 are combined with node i

// to form a heap rooted at i

// No node has an address greater than n or less than 1

j=2i

item=a[i]

while(j≤n)

{

if((j<n) and (a[j]<a[j+1]))

{

j=j+1 // Compare left and right child and let j be the bigger child

}

if(item≥a[j])

{

Break from the loop // A position for item is found

}

a[j/2] =a[j]

j=2j

}

a[j/2]= item

}

**Assumptions:**

The following assumption have been made during this assignment

* We are assuming that the given input sequence is of integer type.

**Analysis:**

In this approach inside the Heapsort function we have two more functions Heapify to build the max heap and Adjust to adjust the heap after root deletion. Though the call of Heapify requires only O(n) operations, Adjust possibly requires O(logn) operations for each invocation as the worst-case run time of Adjust is also proportional to the height of the tree. Therefore, if there are n elements in a heap, deleting the maximum can be done in O(logn) time. So for n elements it can be done in O(nlogn) time, hence the time complexity for Heap sort is in O(nlogn) .

**Dry run:**

Let us consider a sequence S of 7 elements as follows: [56 12 88 15 66 23 37]

Now initially we have to build the max heap so that the root node will have the highest value among all the nodes.

After making the max heap we have the sequence as: [88 66 56 37 12 23 15]

Now to have the sequence in sorted order we have to replace the root node with the last element and have to continue this procedure with all the nodes until we have the desired sequence. As of now we have the root node 88 and last node 15 so we will substitute it after substituting we have the sequence as: [15 66 56 37 12 23 88].

Now we have to adjust the remaining sequence i.e. [15 66 56 37 12 23]. After adjusting the sequence will become [15 56 23 37 12 66 88].

In the next iteration the sequence will be [12 37 23 15 56 66 88]. And in next consecutive iterations we sequence will update as follows:

[ 15 23 12 37 56 66 88]

[ 12 15 23 37 56 66 88]

[12 15 23 37 56 66 88]

Now after all iterations we finally have the sorted sequence [12 15 23 37 56 66 88].

**Code:**

#include <stdio.h>

#include <stdlib.h>

void heapsort(int \*, int); // Funtion prototype for Heap sort

void heapify(int \*, int); // Function prototype for heapify

void adjust(int \*, int, int); // Function prototype for adjust which will adjust the heap

int main()

{

int \*a, n;

d o

{

printf("How many elements to be inserted?: ");

scanf("%d", &n);

if (n <= 0)

{

printf("\nEnter a valid number!!!..\n");

}

} while (n <= 0);

// Dynamic memory allocation for input array

a = (int \*)calloc(n, sizeof(int));

printf("Enter the elements:\t");

for (int i = 0; i < n; i++)

{

scanf("%d", &a[i]);

}

heapsort(a, n- 1); // Calling the function for heap sort

printf("\n\narray after heapsort:\n\n");

for (int i = 0; i < n; i++)

{ // printing the sorted array

printf("\t%d", a[i]);

}

printf("\n");

free(a); // Free the input array

return 0;

}

void heapsort(int \*a, int n)

{ // Function definition for heapsort

heapify(a, n); // Calling the function to build the initial max heap

for (int i = n; i >= 1; i--)

{ // placing the root value in it’s appropriate position

int t = a[i];

a[i] = a[0];

a[0] = t;

adjust(a, 0, i- 1); // Calling the function to adjust the heap

}

}

void heapify(int \*a, int n)

{ // Function definition to build the heap

for (int i = n / 2; i >= 0; i--)

{

adjust(a, i, n);

}

}

void adjust(int \*a, int i, int n)

{ // Function definition to adjust the heap

int j = 2 \* i;

int item = a[i];

while (j <= n)

{

if ((j < n) && (a[j] < a[j + 1]))

{ // Compare left and right child and let j be the bigger child

j = j + 1;

}

if (item >= a[j])

{

break; // A position for item is found

}

a[j / 2] = a[j];

j = 2 \* j;

}

a[j / 2] = item;

}

**Output:**

How many elements to be inserted?: 7

Enter the elements: 56 12 88 15 66 23 37

array after heapsort:

12 15 23 37 56 66 88

How many elements to be inserted?: 10

Enter the elements: 25 98 47 19 87 23 66 12 50 33

array after heapsort:

12 19 23 25 33 47 50 66 87 98

**Discussion:**

* Heap sort has a time complexity of O(nlogn) where n is the number of elements to be sorted for all the cases i.e. best, worst, average case.
* Heap sort involves building and maintaining heap which makes this algorithm very costly.
* Heap sort is an unstable algorithm which means it can change the relative order of equal terms.

**ASSIGNMENT NO: 2 DATE:12.12.2023**

**Problem statement:**

Consider a graph G(V,E) use breadth first search (BFS) algorithm to check whether the given graph is connected or not also find the number of components in the graph using the said algorithm.

**Theory:**

A graph G(V,E) is said to be undirected if the graph G is defined abstractly as an ordered pair (V,E) where V is a set and E is a set of multisets of two elements from V. An undirected graph can be represented geometrically as a set of marked points V with a set of lines E between the points.

A graph G(V,E) is said to be unweighted if it’s edges don’t have any weight associated with it.

A graph G(V,E) is said to be connected if for every pair of distinct vertices vi,vj in G there is a path and if not then the graph G is said to be disconnected.

We will solve the given problem using Breadth First Search (BFS) algorithm.

BFS algorithm is one of the simplest algorithm for searching a graph and archetype for many important graph algorithms. Given a graph G(V,E) and a source vertex s breadth first search systematically explores the edges of G to discover every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. In this algorithm we use queue data structure to keep track of next visiting vertex and while we deque a vertex from the queue we increase a count variable by 1, after all the vertices are explored if the value of count is equal to the value of total number of vertices, then we can say that our graph is connected else our graph is disconnected.

Now to find the number of components in the graph we will take the help of the distance array used in BFS which is keeping track of distances from the source vertex to other vertices. This array is initialized with -1 which indicates that all the vertices are unvisited now after the first BFS call we will check whether the array contains -1 or not if it contains -1 then it means that vertex or vertices are unvisited hence the number of component is increased by 1 then we will call the BFS function once again but this time with an unvisited source node after that we will check the distance array once again and if there are no -1 we will conclude that all the components and vertices are discovered.

**Pseudo code:**

The algorithm for BFS(Breadth first search) is as follows:

Algo\_BFS(G(V,E),s)

{ // G(V,E) is the connected graph given as input where V is the set of vertices and E is

// set of edges.

// s is the source vertex

n =|V|

for(i=1 to n)

{ // initializing the distance array d and parent array p to -1

d[i]= -1

parent[i]= -1

}

d[s]= 0 // initializing the source index of distance array to 0

Q 🡨 an empty queue

count 🡨 0

enqueue(Q,s) // inserting the source to queue

While(Not empty (Q))

{

v 🡨dequeue(Q) // dequeue a element from queue and storing it in v

count= count+1

for all u neighbour(v) // finding neighbours of v and storing in u

{

if( d[u] <0 )

{

d[u] = d[v] +1

parent[u]= v

enqueue(Q,u)

}

}

}

if(count == n)

{

print graph is connected

else

print graph is disconnected

}

}

**Assumptions:**

The following assumptions have been made during this assignment

* We are assuming that the given input sequence is of integer type.
* We are assuming that we will only call the BFS function until all the components are detected. Once all the components are detected we will not call BFS function any further.

**Analysis:**

The main idea behind BFS algorithm is to start with a vertex if it is unvisited mark it as visited then put all it’s adjacent vertices into a queue. We have to repeat this step until the queue is empty. Now this algorithm uses queue data structure to store the adjacent vertices. As queue follows first in first out methodology it helps BFS for the breadth wise traversing hence the Breadth First Search. So now the time complexity for operations on queue is in O(1) , and we have to do that for all the vertices so the total time complexity for queue operations is in O(V). We also have to find the adjacent neighbours by traversing through the graph, time complexity for this task is also is in O(V) so the time complexity for the BFS algorithm is in O(V2) given that we are taking the input graph in adjacency matrix format.

However if we use adjacency list format instead of matrix then the time complexity will be in O(V + E) because the time complexity for queue operation will remain the same but the time complexity for finding the neighbours will be in O(E) as now instead of scanning through the matrix now we are scanning the adjacency list. In this assignment we are using adjacency matrix representation.

**Dry run:**

Consider the following disconnected graph with 2 components.

Now we will begin the procedure with source vertex 1. Now initially all the parent array contains only -1. But as we begin our procedure the distance array cell for vertex 1 will be set to 0 then as the procedure advances neighbour vertices 2 and 3 will get into the queue and their respective distance array cell will be updated according to our algorithm. As vertices 4 and 5 are part of another component they will be not marked as visited in the first call as we have chosen vertex 1 as source node which belongs from another component. So, after the first call the distance array will be: [0 1 1 -1 -1].

Now we will choose vertex number 4 as the new source vertex which will make the vertex 5 visited. So now after the second call the distance array will be: [0 1 1 0 1]. Now we don not have any -1 in the distance array hence we can conclude that all vertices have been visited and as it requires 2 BFS calls for visiting all the vertices we can say that the number of components in the given input graph is 2.

**Code:**

#include <stdlib.h>

#include <stdio.h>

int bfs(int \*\*, int, int, int \*); // Function prototype for bfs

int main()

{

    int \*\*a, n, s, d, i, j, bfs\_counter = 0;

    do

    { // Taking input for the graph in adjacency matrix format

  printf("enter the no of vertices in the graph: ");

        scanf("%d", &n);

    } while (n <= 0);

    a = (int \*\*)calloc(n, sizeof(int \*)); // Dynamically allocating memory

    int \*dist = (int \*)calloc(n, sizeof(int));

    for (i = 0; i < n; i++)

    {

        a[i] = (int \*)calloc(n, sizeof(int));

        dist[i] = -1;

    }

    printf("\nenter the edges:\n");

    do

    {

        printf("\n\tenter the vertex pair for which there is an edge: ");

        scanf("%d%d", &s, &d);

        a[s - 1][d - 1] = 1;

        a[d - 1][s - 1] = 1;

        printf("\n do  you want to enter more no edges:[0/1] ");

        scanf("%d", &s);

    } while (s != 0);

    printf("\nthe input graph is:\n");

    for (i = 0; i < n; i++)

    {

        printf("\n");

        for (j = 0; j < n; j++)

        {

            printf("%d", a[i][j]);

            printf("\t");

        }

    }

    printf("\nplease enter the source node: ");

    scanf("%d", &s);

    s = s - 1;

    int count = bfs(a, n, s, dist); // calling the bfs function

    bfs\_counter++;

    if (count == n)

    {

        printf("\n graph is connected, there is only 1 component in the given graph\n");

        printf("\nthe distance array:\t");

        for (i = 0; i < n; i++)

        {

            printf("\t%d\t", dist[i]);

        }

        printf("\n");

    }

    else

    {

        printf("\n graph is not connected, there is more than 1 component in the given graph\n");

        printf("\nthe distance array:\t");

        for (i = 0; i < n; i++)

        {

            printf("\t%d\t", dist[i]);

        }

        printf("\nDo you want to apply bfs again?[0/1]:\t");

        scanf("%d", &d);

        while (d != 0)

        {

            printf("\nEnter the source vertex:\t");

            scanf("%d", &s);

            bfs(a, n, s - 1, dist);

            bfs\_counter++;

            printf("\nthe distance array:\t");

            for (i = 0; i < n; i++)

            {

                printf("\t%d\t", dist[i]);

            }

            printf("\nDo you want to appy bfs again?[0/1]:\t");

            scanf("%d", &d);

        }

    }

    printf("\nNumber of components detected: %d\n", bfs\_counter++);

    free(a);

    free(dist);

    return 0;

}

// Function definition for bfs

int bfs(int \*\*a, int n, int s, int \*d)

{

    int \*parent = (int \*)calloc(n, sizeof(int));

    int \*q = (int \*)calloc(n, sizeof(int));

    int i, count = 0, k, u, v, front = -1, rear = -1;

    for (i = 0; i < n; i++)

    {

        parent[i] = -1;

    }

    d[s] = 0; // set the value to 0 for source vertex in distance array

    q[++rear] = s; // enqueue the source vertex

    while (front != rear)

    {

        v = q[++front]; // dequeue a vertex from the queue

        count++;

        for (u = 0; u < n; u++) // scanning through the vertices

        {

            if (a[v][u] == 1) // finding the adjacent vertex

            {

                if (d[u] < 0) // if vertex is unvisited mark it visited and // update the distance array

                {

                    d[u] = d[v] + 1;

                    parent[u] = v;

                    q[++rear] = u;

                }

            }

        }

    }

    free(q);

    free(parent);

    return count;

}

**Output:**

enter the no of vertices in the graph: 5

enter the edges:

enter the vertex pair for which there is an edge: 1 2

do you want to enter more no edges:[0/1] 1

enter the vertex pair for which there is an edge: 2 3

do you want to enter more no edges:[0/1] 1

enter the vertex pair for which there is an edge: 1 3

do you want to enter more no edges:[0/1] 1

enter the vertex pair for which there is an edge: 4 5

do you want to enter more no edges:[0/1] 0

the input graph is:

0 1 1 0 0

1 0 1 0 0

1 1 0 0 0

0 0 0 0 1

0 0 0 1 0

please enter the source node: 1

graph is not connected, there is more than 1 component in the given graph

the distance array: 0 1 1 -1 -1

Do you want to apply bfs again?[0/1]: 1

Enter the source vertex: 4

the distance array: 0 1 1 0 1

Do you want to appy bfs again?[0/1]: 0

Number of components detected: 2

**Discussion:**

* Time complexity for BFS depends on the graph input format for adjacency matrix it is in O(V2) and for adjacency list it is O(V+E)
* As the name suggests BFS traverses the graph level wise hence it visited the neighbours first which are in the same level.

**ASSIGNMENT NO: 3 DATE: 02.01.2024**

**Problem statement:**

write a menu driven c program to create two polynomial and show their addition, subtraction and multiplication.

**Theory:**

Polynomial multiplication is a fundamental operation in algebra and computer science. It involves multiplying two polynomials to obtain a new polynomial that represents their product. Polynomials are mathematical expressions composed of variables raised to non-negative integer powers, each multiplied by a coefficient. For example, the polynomial has coefficients 3, 2, and 1 for the powers 2, 1, and 0 of the variable x, respectively.

In polynomial multiplication, the goal is to find the product of two polynomials and denoted as . The resulting polynomial will have degrees corresponding to the sum of the degrees of the multiplied polynomials. The process involves multiplying each term of one polynomial by each term of the other and then summing up like terms.

One efficient way to represent polynomials and perform multiplication is by using linked lists. A linked list is a data structure that consists of nodes, each containing a data element and a reference (link) to the next node. In the context of polynomials, each node can represent a term, with the coefficient and exponent as data elements.

**Pseudo Code:**

Pseudo code for addition is as follows:

Addition(poly1,poly2)

{

Allocate memory for poly3 which is the resultant polynomial

While(poly1 and poly2 are not null)

{

Traverse the two polynomial and compare their exponent

if(poly1.exponent = poly2. exponent)

{

poly3.coefficient=poly1.coefficient+poly2.coefficient

poly3.exponent=poly1.exponent

}

}

}

Pseudo code for subtraction is as follows:

Subtraction(poly1,poly2)

{

Allocate memory for poly3 which is the resultant polynomial

While(poly1 and poly2 are not null)

{

Traverse the two polynomial and compare their exponent

if(poly1.exponent = poly2. exponent)

{

poly3.coefficient=poly1.coefficient-poly2.coefficient

poly3.exponent=poly1.exponent

}

}

}

Pseudo code for multiplication is as follows:

Multiplication(poly1,poly2)

{

Allocate memory for poly3 which is the resultant polynomial

While(poly1 and poly2 are not null)

{

poly3.coefficient=poly1.coefficient\*poly2.coefficient

poly3.exponent=poly1.exponent+poly2.exponent

}

}

**Assumption:**

The following assumptions have been made during this assignment

* We are assuming that the given input sequence is of non-negative integer type.
* We are assuming that we will enter our polynomials and then we will perform manipulation over the polynomials. And during subtraction user will be asked how they want to subtract those.

**Analysis:**

in this menu driven polynomial operations the main idea is to take two different polynomials and then do the operations based on their coefficient and exponent.In case of addition when the exponentials of the variables are same then they remained same in the resultant polynomial & when the exponent of the one variable is larger than the corresponding variable of the other polynomials then the large value will remained in the resultant polynomial. In case of subtraction we have to check the type of subtraction we needed ,i.e if we need to subtract the second polynomial from the first one or vice versa,then we have to check the exponential of the co efficient same as the addition mentioned above and have to placed it same as the addition process. In case of multiplication the coefficient of the corresponding variables will be multiplied by each other and the exponentials corresponding the variables will get added t each other and will placed in the resultant polynomial. after all the process has been done we have to show the resultant polynomial. So it is evident that the time complexity for polynomial operations is in O(n2) because we have to scan through one polynomial to find its corresponding exponent in another polynomial.

**Code:**

#include <stdio.h>

#include <stdlib.h>

typedef struct poly

{

int coef, expo;

struct poly \*next;

} POLY;

POLY \*createpolynode(int co, int ex)

{

POLY \*p;

p = (POLY \*)malloc(sizeof(POLY));

p->coef = co;

p->expo = ex;

p->next = NULL;

return p;

}

POLY \*insert\_polynomial(POLY \*start, int co, int ex) // creating the polynomial exp in a linked list

{

POLY \*p, \*q;

p = createpolynode(co, ex);

if (start == NULL || ex > start->expo) // when the list is empty or the exponent taken is greater than the exponent of start

{

p->next = start;

start = p; // taking the largest exponent node at start

}

else

{

q = start;

while (q->next != NULL && (q->next)->expo >= ex) // traversing the list until NULL and checking the exponent of next term is greater than the taken exponent or not

{

q = q->next; // if condition satisfies then procced to next node

}

p->next = q->next;

q->next = p; // taking the exponent in a dsending order i.e in proper place

}

return start; // returning the new value of start

}

POLY \*insertadd\_polynomial(POLY \*start,POLY \*node) // creating the polynomial exp in a linked list

{

POLY \*p, \*q;

if (start == NULL) // when the list is empty or the exponent taken is greater than the exponent of start

{

start = node; // taking the largest exponent node at start

}

else

{

q = start;

while (q->next != NULL && (q->next)->expo >node->expo) // traversing the list until NULL and checking the exponent of next term is greater than the taken exponent or not

{

q = q->next; // if condition satisfies then procced to next node

}

if(q->next!=NULL && (q->next)->expo==node->expo)

{

(q->next)->coef+=node->coef;

}

else

{

node->next = q->next;

q->next = node;

} // taking the exponent in a dsending order i.e in proper place

}

return start; // returning the new value of start

}

POLY \*take\_input\_polynomial(POLY \*start) // for taking inputs of the polynomials

{

int n, i;

int co, ex;

do

{

printf("\nEnter the number of terms of the polynomial(any non-negative integer other than 0):");

scanf("%d", &n);

} while (n <= 0); // to avoid negative or 0 value as no. of terms can't be that

for (i = 1; i <= n; i++) // loop for taking given no. of inputs

{

printf("\nEnter the coefficient for term %d: ", i);

scanf("%d", &co);

printf("\nEnter the exponent for term %d: ", i);

scanf("%d", &ex);

start = insert\_polynomial(start, co, ex); // calling the function to create the polynomial expression

}

return start; // returning new value of start

}

void print\_polynomial(POLY \*start) // for printing resultant polynomials

{

POLY \*p;

if (start == NULL) // when the list is empty

{

printf("\nNo such polynomial is present...!");

}

else

{

p = start;

while (p != NULL) // traversing until p is NULL

{

printf("%dx^%d", p->coef, p->expo); // printing each term of polynomial

p = p->next;

if (p != NULL && p->coef >= 0) // avoiding printing of "+" in case of negative term or after the last term

{

printf("+"); // printing "+" if condition satisfies

}

}

printf("\n");

}

}

void add\_polynomial(POLY \*start1, POLY \*start2)

{

POLY \*p1 = start1;

POLY \*p2 = start2;

POLY \*start3 = NULL;

while (p1 != NULL && p2 != NULL) // traversing both the polynomials until NULL

{

if (p1->expo == p2->expo) // when the exponents of the polynomials are equal

{

start3 = insert\_polynomial(start3, p1->coef + p2->coef, p1->expo); // adding coefficients and calling the insert function to make a resultant polynomial

p1 = p1->next;

p2 = p2->next;

}

else if (p1->expo > p2->expo) // when 1st polynomials exponent is greater

{

start3 = insert\_polynomial(start3, p1->coef, p1->expo); // returning 1st polynomials coefficient and exponent

p1 = p1->next;

}

else if (p1->expo < p2->expo) // when 2nd polynomials exponent is greater

{

start3 = insert\_polynomial(start3, p2->coef, p2->expo); // returning 2nd polynomials coefficient and exponent

p2 = p2->next;

}

}

while (p1 != NULL) // when 2nd one reaches NULL but the 1st doesn't

{

start3 = insert\_polynomial(start3, p1->coef, p1->expo); // just returning 1st one

p1 = p1->next;

}

while (p2 != NULL) // when 1st one reaches NULL but the 2nd doesn't

{

start3 = insert\_polynomial(start3, p2->coef, p2->expo); // just returning 2nd one

p2 = p2->next;

}

printf("\nResultant polynomial after addition is:\n");

print\_polynomial(start3); // calling print function to print resultant polynomial

}

void subtract\_polynomial1(POLY \*start1, POLY \*start2) // function for poly1-poly2

{

POLY \*p1 = start1;

POLY \*p2 = start2;

POLY \*start3 = NULL;

while (p1 != NULL && p2 != NULL) // traversing both the polynomials until NULL

{

if (p1->expo == p2->expo) // when the exponents of the polynomials are equal

{

start3 = insert\_polynomial(start3, p1->coef - p2->coef, p1->expo); // subtracting coefficients and calling the insert function to make a resultant polynomial

p1 = p1->next;

p2 = p2->next;

}

else if (p1->expo > p2->expo) // when 1st polynomials exponent is greater

{

start3 = insert\_polynomial(start3, p1->coef, p1->expo); // returning 1st polynomials coefficient and exponent

p1 = p1->next;

}

else if (p1->expo < p2->expo) // when 2nd polynomials exponent is greater

{

start3 = insert\_polynomial(start3, -p2->coef, p2->expo); // returning 2nd polynomials -ve coefficient and exponent

p2 = p2->next;

}

}

while (p1 != NULL)

{

start3 = insert\_polynomial(start3, p1->coef, p1->expo); // returning 1st polynomials coefficient and exponent

p1 = p1->next;

}

while (p2 != NULL)

{

start3 = insert\_polynomial(start3, -p2->coef, p2->expo); // returning 2nd polynomials -ve coefficient and exponent

p2 = p2->next;

}

printf("\nResultant polynomial after subtraction is:\n");

print\_polynomial(start3); // calling print function to print resultant polynomial

}

void subtract\_polynomial2(POLY \*start1, POLY \*start2) // function for poly2-poly1

{

POLY \*p1 = start1;

POLY \*p2 = start2;

POLY \*start3 = NULL;

while (p1 != NULL && p2 != NULL)

{

if (p1->expo == p2->expo)

{

start3 = insert\_polynomial(start3, p2->coef - p1->coef, p2->expo); // just doing the reverse of previous subtraction

p1 = p1->next;

p2 = p2->next;

}

else if (p1->expo > p2->expo)

{

start3 = insert\_polynomial(start3, -p1->coef, p1->expo); // just doing the reverse of previous subtraction

p1 = p1->next;

}

else if (p1->expo < p2->expo)

{

start3 = insert\_polynomial(start3, p2->coef, p2->expo); // just doing the reverse of previous subtraction

p2 = p2->next;

}

}

while (p1 != NULL)

{

start3 = insert\_polynomial(start3, -p1->coef, p1->expo); // just doing the reverse of previous subtraction

p1 = p1->next;

}

while (p2 != NULL)

{

start3 = insert\_polynomial(start3, p2->coef, p2->expo); // just doing the reverse of previous subtraction

p2 = p2->next;

}

printf("\nResultant polynomial after subtraction is:\n");

print\_polynomial(start3); // calling print function to print resultant polynomial

}

void mult\_polynomial(POLY \*start1, POLY \*start2)

{

POLY \*p1 = start1;

POLY \*p2 = start2;

POLY \*start3 = NULL;

while (p1 != NULL)

{

p2=start2;

while(p2!=NULL)

{

POLY \*p = createpolynode(p1->coef\*p2->coef,p1->expo+p2->expo);

start3=insertadd\_polynomial(start3,p);

p2=p2->next;

}

p1=p1->next;

}

printf("\nResultant polynomial after multiplication is:\n");

print\_polynomial(start3);

}

void main()

{

POLY \*start1 = NULL;

POLY \*start2 = NULL;

int ch;

char sch;

while (1)

{

printf("\n===============MAIN MENU===============");

printf("\n1.Insert the 1st Polynomial\n2.Insert the 2nd Polynomial\n3.Addition of the two Polynomials\n4.Subtraction of the two Polynomials\n5.Multiplication of two polynomials\n6.Exit from the System");

printf("\nEnter your choice:");

scanf("%d", &ch);

switch (ch)

{

case 1:

start1 = take\_input\_polynomial(start1);

printf("\nYour 1st polynomial is:\n");

print\_polynomial(start1);

break;

case 2:

start2 = take\_input\_polynomial(start2);

printf("\nYour 2nd polynomial is:\n");

print\_polynomial(start2);

break;

case 3:

add\_polynomial(start1, start2);

break;

case 4:

do

{

printf("\nWhich type of Subtraction you want to perform?");

printf("\na.Subtract:(polynomial 1)-(polynomial 2)\nb.Subtract:(polynomial 2)-(polynomial 1)\nc.Quit Subtraction and back to Main menu");

printf("\nEnter your choice:");

sch = getche();

sch = tolower(sch);

switch (sch)

{

case 'a':

subtract\_polynomial1(start1, start2); // for poly1-poly2

break;

case 'b':

subtract\_polynomial2(start1, start2); // for poly2-poly1

break;

case 'c':

break;

default:

printf("\nWrong choice given...!");

}

} while (sch != 'c');

break;

case 5:

mult\_polynomial(start1,start2);

break;

case 6:

exit(0);

break;

default:

printf("\nWrong choice given...!");

}

}

free(start1);

free(start2);

}

**Output:**

===============MAIN MENU===============

1.Insert the 1st Polynomial

2.Insert the 2nd Polynomial

3.Addition of the two Polynomials

4.Subtraction of the two Polynomials

5.Multiplication of two polynomials

6.Exit from the System

Enter your choice:1

Enter the number of terms of the polynomial(any non-negative integer other than 0):2

Enter the coefficient for term 1: 11

Enter the exponent for term 1: 2

Enter the coefficient for term 2: 3

Enter the exponent for term 2: 1

Your 1st polynomial is:

11x^2+3x^1

===============MAIN MENU===============

1.Insert the 1st Polynomial

2.Insert the 2nd Polynomial

3.Addition of the two Polynomials

4.Subtraction of the two Polynomials

5.Multiplication of two polynomials

6.Exit from the System

Enter your choice:2

Enter the number of terms of the polynomial(any non-negative integer other than 0):2

Enter the coefficient for term 1: 31

Enter the exponent for term 1: 3

Enter the coefficient for term 2: 6

Enter the exponent for term 2: 2

Your 2nd polynomial is:

31x^3+6x^2

===============MAIN MENU===============

1.Insert the 1st Polynomial

2.Insert the 2nd Polynomial

3.Addition of the two Polynomials

4.Subtraction of the two Polynomials

5.Multiplication of two polynomials

6.Exit from the System

Enter your choice:3

Resultant polynomial after addition is:

31x^3+17x^2+3x^1

===============MAIN MENU===============

1.Insert the 1st Polynomial

2.Insert the 2nd Polynomial

3.Addition of the two Polynomials

4.Subtraction of the two Polynomials

5.Multiplication of two polynomials

6.Exit from the System

Enter your choice:4

Which type of Subtraction you want to perform?

a.Subtract:(polynomial 1)-(polynomial 2)

b.Subtract:(polynomial 2)-(polynomial 1)

c.Quit Subtraction and back to Main menu

Enter your choice:a

Resultant polynomial after subtraction is:

-31x^3+5x^2+3x^1

Which type of Subtraction you want to perform?

a.Subtract:(polynomial 1)-(polynomial 2)

b.Subtract:(polynomial 2)-(polynomial 1)

c.Quit Subtraction and back to Main menu

Enter your choice:b

Resultant polynomial after subtraction is:

31x^3-5x^2-3x^1

Which type of Subtraction you want to perform?

a.Subtract:(polynomial 1)-(polynomial 2)

b.Subtract:(polynomial 2)-(polynomial 1)

c.Quit Subtraction and back to Main menu

Enter your choice:c

===============MAIN MENU===============

1.Insert the 1st Polynomial

2.Insert the 2nd Polynomial

3.Addition of the two Polynomials

4.Subtraction of the two Polynomials

5.Multiplication of two polynomials

6.Exit from the System

Enter your choice:5

Resultant polynomial after multiplication is:

341x^5+159x^4+18x^3

**Discussion:**

* In this assignment we have chosen linked list representation for polynomial which makes it space efficient.
* We have only considered single variable polynomial.
* If we have a large polynomial then computation can be a bit hectic.

**ASSIGNMENT NO: 4 DATE: 02.01.2024**

**Problem statement:**

write a c program to create an AVL tree.

**Theory:**

An AVL tree is a height balanced binary search tree where every node has a balance factor of -1,0,1, where the balance factor for a given node is defined as the difference between the height of the left and right subtree of that node. Now if the node has a disbalance factor then we need to perform some rotations to make it balanced. The type of rotation required to make the tree balanced is based on the following conditions:

* Disbalance due to right-right insertion: For this case we need to perform a single left rotation to the disbalanced node.
* Disbalance due to left-left insertion: For this case we need to perform a single right rotation to the disbalanced node.
* Disbalance due to left-right insertion: For this case we need to perform double rotation to the disbalanced node first a left rotation to the node inserted right and then a left rotation to the disbalanced node.
* Disbalance due to right-left insertion: For this case we need to perform double rotation to the disbalanced node first a right rotation to the node inserted left and then a right rotation to the disbalanced node.

**Algorithm:**

**Steps**

1. ptr = ROOT
2. If (ptr = NULL) // insertion into a null tree, creates a single node tree
3. ptr = Nptr
4. ptr 🡪 HEIGHT = 1 // set height as 1
5. return
6. else
7. If (Nptr 🡪 DATA < ptr 🡪 DATA) then
8. Insert( ptr🡪 LCHILD, Nptr)
9. Lptr = ptr🡪LCHILD
10. Rptr = ptr🡪RCHILD // right sub tree of ptr
11. If (Rptr = NULL) then // if right sub tree is empty
12. hR = 0
13. Else
14. hR= Rptr🡪HEIGHT //height of the right subtree
15. hL= Lptr🡪HEIGHT
16. bf = hL – hR
17. If (bf = 2) then
18. If (Nptr🡪DATA < Lptr🡪DATA) then
19. LeftToRightRotation(ptr)
20. Else
21. LeftToRightRotation(ptr)
22. EndIf
23. Ptr🡪HEIGHT = ComputeHeight(ptr) // calculate the height
24. EndIf
25. EndIf
26. Else
27. If (Nptr🡪DATA > ptr🡪DATA) then
28. InsertHBT (ptr🡪RCHILD, Nptr)
29. Rptr = ptr🡪RCHILD
30. Lptr = ptr🡪LCHILD
31. If (Lptr = NULL) then
32. hL = 0
33. Else
34. hL = Lptr🡪HEIGHT
35. hR = Rptr🡪HEIGHT
36. Bf = hL – hR
37. If(bf = -2) then
38. If (Nptr🡪DATA > Rptr🡪DATA) then
39. RightToLeftRotation(ptr)
40. Else
41. RightToLeftRotation(ptr)
42. EndIf
43. Ptr🡪HEIGHT = ComputeHeight(ptr)
44. EndIf
45. EndIf
46. Else
47. Print Nptr🡪DATA “is already exist in the tree”
48. EndIf
49. EndIf
50. EndIf
51. Stop

Algorithm for left-to-left rotation:

**Steps**

1. Aptr = Pptr 🡪 LCHILD
2. Pptr 🡪 LCHILD =Aptr 🡪RCHILD
3. Aptr🡪RCHILD=Pptr
4. Pptr 🡪HEIGHT = computeheight(Pptr)
5. Aptr 🡪HEIGHT= computeheight(Aptr)
6. Pptr=Aptr
7. STOP

Algorithm for right-to-right rotation:

**Steps**

1. Bptr = Pptr 🡪 RCHILD
2. Pptr 🡪 RCHILD =Bptr 🡪LCHILD
3. Bptr🡪LCHILD=Pptr
4. Pptr 🡪HEIGHT = computeheight(Pptr)
5. Bptr 🡪HEIGHT= computeheight(Bptr)
6. Pptr=Bptr
7. STOP

Algorithm for right-to-left rotation:

**Steps**

1. Aptr = Pptr 🡪 RCHILD
2. LeftToLeftRotation(Aptr)
3. RightToRightRotation(Pptr)
4. STOP

Algorithm for left-to-right rotation:

**Steps**

1. Aptr = Pptr 🡪 LCHILD
2. RightToRightRotation(Aptr)
3. LeftToLeftRotation(Pptr)
4. STOP

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

// function prototypes

struct node \*getnode(int);

struct node \*insert(struct node \*, struct node \*, int \*);

struct node \*rotateleft(struct node \*);

struct node \*rotateright(struct node \*);

struct node \*rightbalance(struct node \*, int \*);

struct node \*leftbalance(struct node \*, int \*);

void inorder(struct node \*);

typedef struct node // defining the node structure

{

int data;

struct node \*left;

struct node \*right;

int bf;

} node;

int main()

{

node \*root = NULL;

int tall = 0;

int n, item;

printf("Enter the number of elements to be inserted: ");

scanf("%d", &n);

printf("Enter the elements: ");

for (int i = 0; i < n; i++)

{

scanf("%d", &item);

node \*newnode = getnode(item);

root = insert(root, newnode, &tall);

}

printf("\nInorder traversal of the AVL tree: \n"); // inorder traversal of the tree

inorder(root);

printf("\n");

}

struct node \*getnode(int item) // initializing a node

{

struct node \*p;

p = (struct node \*)malloc(sizeof(struct node));

p->data = item;

p->left = NULL;

p->right = NULL;

p->bf = 999;

return p;

}

struct node \*insert(node \*root, node \*new, int \*tall)

{

if (root == NULL)

{

root = new;

root->bf = 0;

root->left = root->right = NULL;

\*tall = 1;

}

else if (new->data == root->data)

{

printf("\n Duplicate element in AVL tree\n");

}

else if (new->data < root->data)

{

root->left = insert(root->left, new, tall);

if (\*tall == 1)

{

switch (root->bf)

{

case 1:

root = leftbalance(root, tall);

break;

case 0:

root->bf = 1;

break;

case -1:

root->bf = 0;

\*tall = 0;

break;

}

}

}

else

{

root->right = insert(root->right, new, tall);

if (\*tall == 1)

{

switch (root->bf)

{

case 1:

root->bf = 0;

\*tall = 0;

break;

case 0:

root->bf = -1;

break;

case -1:

root = rightbalance(root, tall);

break;

}

}

}

return root;

}

struct node \*rotateleft(node \*p) // function definition for left rotation

{

node \*temp;

if (p == NULL)

{

printf("\n cannot rotate empty tree\n");

}

else if (p->right == NULL)

{

printf("\n can not rotate left");

}

else

{

temp = p->right;

p->right = temp->left;

temp->left = p;

}

return temp;

}

struct node \*rotateright(node \*p) // function definition for right rotation

{

node \*temp;

if (p == NULL)

{

printf("\n cannot rotate empty tree\n");

return p;

}

else if (p->left == NULL)

{

printf("\n can not rotate right");

return p;

}

else

{

temp = p->left;

p->left = temp->right;

temp->right = p;

}

return temp;

}

struct node \*rightbalance(node \*root, int \*tall)

{

node \*rs = root->right;

node \*ls;

switch (rs->bf)

{

case -1:

root->bf = rs->bf = 0;

root = rotateleft(root);

\*tall = 0;

break;

case 0:

printf("\ntree already balanced\n");

break;

case 1:

ls = rs->left;

switch (ls->bf)

{

case -1:

root->bf = 1;

rs->bf = 0;

break;

case 0:

root->bf = rs->bf = 0;

break;

case 1:

root->bf = 0;

rs->bf = -1;

break;

}

ls->bf = 0;

root->right = rotateright(rs);

root = rotateleft(root);

\*tall = 0;

}

return root;

}

struct node \*leftbalance(node \*root, int \*tall)

{

node \*ls = root->left;

node \*rs;

switch (ls->bf)

{

case 1:

root->bf = ls->bf = 0;

root = rotateright(root);

\*tall = 0;

break;

case 0:

printf("\ntree already balanced\n");

break;

case -1:

rs = ls->right;

switch (rs->bf)

{

case 1:

root->bf = -1;

ls->bf = 0;

break;

case 0:

root->bf = ls->bf = 0;

break;

case -1:

root->bf = 0;

ls->bf = 1;

break;

}

rs->bf = 0;

root->left = rotateleft(ls);

root = rotateright(root);

\*tall = 0;

}

return root;

}

void inorder(node \*start)

{

if (start != NULL)

{

inorder(start->left);

printf("%d \t", start->data);

inorder(start->right);

}

}

**Output:**

Enter the number of elements to be inserted: 7

Enter the elements: 12 56 3 9 14 60 21

Inorder traversal of the AVL tree:

3 9 12 14 21 56 60

**Discussion:**

* As AVL tree is a height balanced tree so we will always have the tree in perfectly balanced form so we will not get any skew formed tree under any circumstances.
* As it is a height balanced binary search tree searching can be done in O(log n) time.