DAY 1

Structure and union :

1.union Apple

{

int a[20];

double b[10];

char c[20];

}a1,a2;

void main()

{

printf("Size of apple =%d",sizeof(a2));

}

2. #include<stdio.h>

struct Apple

{

int a;

double b;

}a1,a2;

void main()

{

printf("Size of apple =%d",sizeof(a2));

}

TRESS:

Height of tree,level,balance or unbalanced,traversal

Inorder🡪 LVR (left,root,right)

Preorder🡪VLR(root left right)

Postorder🡪(left right root)

#include <stdio.h>

#include <stdlib.h>

// Node structure for binary tree

typedef struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

} TreeNode;

// Function to create a new node

TreeNode \*newNode(int data) {

TreeNode \*node = (TreeNode \*)malloc(sizeof(TreeNode));

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

// Function to get the height of a binary tree

int height(TreeNode \*root) {

if (root == NULL)

return 0;

int leftHeight = height(root->left);

int rightHeight = height(root->right);

return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

}

// Function to check if a binary tree is balanced

int isBalanced(TreeNode \*root) {

if (root == NULL)

return 1;

int leftHeight = height(root->left);

int rightHeight = height(root->right);

if (abs(leftHeight - rightHeight) <= 1 && isBalanced(root->left) && isBalanced(root->right))

return 1;

return 0;

}

// Main function

int main() {

// Creating a sample binary tree

TreeNode \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

root->left->left->left = newNode(8);

// Check if the binary tree is balanced

if (isBalanced(root))

printf("The binary tree is balanced.\n");

else

printf("The binary tree is not balanced.\n");

return 0;

}

DAY 2

MEMORY ALLOCATION

Structure multiple dtype

Union specifird dtype

MALLOC,CALLOC,REALLOC

#include <stdio.h>

#include <stdlib.h>

// Node structure for binary tree

typedef struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

} TreeNode;

// Function to create a new node

TreeNode \*newNode(int data) {

TreeNode \*node = (TreeNode \*)malloc(sizeof(TreeNode));

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

// Function to get the height of a binary tree

int height(TreeNode \*root) {

if (root == NULL)

return 0;

int leftHeight = height(root->left);

int rightHeight = height(root->right);

return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

}

// Function to check if a binary tree is balanced

int isBalanced(TreeNode \*root) {

if (root == NULL)

return 1;

int leftHeight = height(root->left);

int rightHeight = height(root->right);

if (abs(leftHeight - rightHeight) <= 1 && isBalanced(root->left) && isBalanced(root->right))

return 1;

return 0;

}

// Main function

int main() {

// Creating a sample binary tree

TreeNode \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

root->left->left->left = newNode(8);

// Check if the binary tree is balanced

if (isBalanced(root))

printf("The binary tree is balanced.\n");

else

printf("The binary tree is not balanced.\n");

return 0;

}

Avl tree

Sel balancing tree

Balancing factor = height of left sub tree-height of right sub tree

QUEUE

FIFO

// Queue implementation in C

#include <stdio.h>

#define SIZE 5

void enQueue(int);

void deQueue();

void display();

int items[SIZE], front = -1, rear = -1;

int main() {

//deQueue is not possible on empty queue

deQueue();

//enQueue 5 elements

enQueue(1);

enQueue(2);

enQueue(3);

enQueue(4);

enQueue(5);

// 6th element can't be added to because the queue is full

enQueue(6);

display();

//deQueue removes element entered first i.e. 1

deQueue();

//Now we have just 4 elements

display();

return 0;

}

void enQueue(int value) {

if (rear == SIZE - 1)

printf("\nQueue is Full!!");

else {

if (front == -1)

front = 0;

rear++;

items[rear] = value;

printf("\nInserted -> %d", value);

}

}

void deQueue() {

if (front == -1)

printf("\nQueue is Empty!!");

else {

printf("\nDeleted : %d", items[front]);

front++;

if (front > rear)

front = rear = -1;

}

}

// Function to print the queue

void display() {

if (rear == -1)

printf("\nQueue is Empty!!!");

else {

int i;

printf("\nQueue elements are:\n");

for (i = front; i <= rear; i++)

printf("%d ", items[i]);

}

printf("\n");

}

main()

{

float i;

for(i=1;i<=10;i++)

{

printf("%.1f \n",i/10);

}

}

PREINCREMENT POSTINCREMENT

main()

{

int a=50;

int b= --a - a-- + a++ - ++a - a-- + --a;

printf("a=%d b=%d",a,b);

}

/\*

pre inc/dec

1st pri - memory

2nd pri - display

post inc/dec

1st pri - display

2nd pri - memory

\*/

main()

{

int a=10;

int \*b=&a;

int \*\*c=&b;

int \*\*\*d=&c;

int \*\*\*\*e=&d;

printf("a =%d \n",a);

printf("own b address =%x \n",&b);

printf("b contains a address =%x \n",b);

printf("a address =%x \n",&a);

printf("----------------- \n");

printf("own address of b = %x \n",&b);

printf("own address of c = %x \n",&c);

printf("own address of d = %x \n",&d);

printf("own address of e = %x \n",&e);

printf("----------------- \n");

printf("address b contains = %x \n",b);

printf("address c contains = %x \n",c);

printf("address d contains = %x \n",d);

printf("address e contains = %x \n",e);

printf("----------------- \n");

}

INFIX PREFIX POSTFIX

A+b,ab+,+ab

A,b operands,, =,/,\* operatrs

All conversions includes several ruels follow it.

Top ()[]{}

1st ^

2nd \* /

3rd + -

infix: a+b

postfix: ab+

prefix: +ab

Infix Input: a+b\*c-(d/e)+f

Step Data Stack Output

1 a a

2 + + a

3 b + ab

4 \* +\* ab

5 c +\* abc

6 - - abc\*+

7 ( -( abc\*+

8 d -( abc\*+d

9 / -(/ abc\*+d

10 e -(/ abc\*+de

11 ) - abc\*+de/

12 + + abc\*+de/-

13 f + abc\*+de/-f

14 Postfix Final Ans: abc\*+de/-f+

Input: a+b\*c-(d/e)+f

Rev Ip: f+(e/d)-c\*b+a

Step Data Stack Output

1 f f

2 + + f

3 ( +( f

4 e +( fe

5 / +(/ fe

6 d +(/ fed

7 ) + fed/

8 - - fed/+

9 c - fed/+c

10 \* -\* fed/+c

11 b -\* fed/+cb

12 + + fed/+cb\*-

13 a + fed/+cb\*-a

14 Final Ans: fed/+cb\*-a+

15 Rev Final Ans : +a-\*bc+/def

Postfix Ans: abc\*+de/-f+

Step Data Stack Output

1 a a

2 b ab

3 c abc

4 \*

RECURSION

#include <stdio.h>

int sum(int n);

int main() {

int number, result;

printf("Enter a positive integer: ");

scanf("%d", &number);

//result = sum(number);

sum(number);

//printf("sum = %d", result);

return 0;

}

int sum(int n) {

printf("n= %d\n",n);

if (n != 0)

// sum() function calls itself

return n + sum(n-1);

printf("%d",n + sum(n-1));

//printf("%d",sum(n-1));

else

//printf("else n=%d",n);

return n;

}

/\*

void recurse()

{

... .. ...

recurse();

... .. ...

}

int main()

{

... .. ...

recurse();

... .. ...

}

\*/

#include<stdio.h>

int main(){

int c =0;

int n;

scanf("%d",&n);

printf("Reminding number: ");

while(n<21){

n++;

c++;

}

printf("%d\n",c);

}

LINKED LIST

// Linked list implementation in C

#include <stdio.h>

#include <stdlib.h>q

// Creating a node

struct node {

int value;

struct node \*next;

};

// print the linked list value

void printLinkedlist(struct node \*p) {

while (p != NULL) {

printf("%d ", p->value);

p = p->next;

}

}

int main() {

// Initialize nodes

struct node \*head;

struct node \*one = NULL;

struct node \*two = NULL;

struct node \*three = NULL;

struct node \*four = NULL;

struct node \*five = NULL;

struct node \*six = NULL;

// Allocate memory

one = malloc(sizeof(struct node));

two = malloc(sizeof(struct node));

three = malloc(sizeof(struct node));

four = malloc(sizeof(struct node));

five = malloc(sizeof(struct node));

six = malloc(sizeof(struct node));

// Assign value values

one->value = 1;

two->value = 2;

three->value = 3;

four->value = 4;

five->value = 5;

six->value = 6;

// Connect nodes

one->next = two;

two->next = three;

three->next = NULL;

four->next = five;

five->next = six;

six->next = NULL;

// printing node-value

head = one;

printLinkedlist(head);

}

#include<stdlib.h>

#include<stdio.h>

struct Node{

int data;

struct Node \*next;

};

void deleteStart(struct Node\*\* head){

struct Node\* temp = \*head;

// If head is NULL it means Singly Linked List is empty

if(\*head == NULL){

printf("Impossible to delete from empty Singly Linked List");

return;

}

// move head to next node

\*head = (\*head)->next;

printf("Deleted: %d\n", temp->data);

free(temp);

}

void insertStart(struct Node\*\* head, int data){

// dynamically create memory for this newNode

struct Node\* newNode = (struct Node\*) malloc(sizeof(struct Node));

// assign data value

newNode->data = data;

// change the next node of this newNode

// to current head of Linked List

newNode->next = \*head;

//re-assign head to this newNode

\*head = newNode;

printf("Inserted %d\n",newNode->data);

}

void display(struct Node\* node){

printf("\nLinked List: ");

// as linked list will end when Node is Null

while(node!=NULL){

printf("%d ",node->data);

node = node->next;

}

printf("\n");

}

int main()

{

struct Node\* head = NULL;

insertStart(&head,100);

insertStart(&head,80);

insertStart(&head,60);

insertStart(&head,40);

insertStart(&head,20);

display(head);

deleteStart(60);

deleteStart(&head);

display(head);

return 0;

}

DOUBLY LINKED LSIT

#include <stdio.h>

#include <stdlib.h>

// node creation

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

// insert node at the front

void insertFront(struct Node\*\* head, int data) {

// allocate memory for newNode

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

// assign data to newNode

newNode->data = data;

// make newNode as a head

newNode->next = (\*head);

// assign null to prev

newNode->prev = NULL;

// previous of head (now head is the second node) is newNode

if ((\*head) != NULL)

(\*head)->prev = newNode;

// head points to newNode

(\*head) = newNode;

}

// insert a node after a specific node

void insertAfter(struct Node\* prev\_node, int data) {

// check if previous node is null

if (prev\_node == NULL) {

printf("previous node cannot be null");

return;

}

// allocate memory for newNode

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

// assign data to newNode

newNode->data = data;

// set next of newNode to next of prev node

newNode->next = prev\_node->next;

// set next of prev node to newNode

prev\_node->next = newNode;

// set prev of newNode to the previous node

newNode->prev = prev\_node;

// set prev of newNode's next to newNode

if (newNode->next != NULL)

newNode->next->prev = newNode;

}

// insert a newNode at the end of the list

void insertEnd(struct Node\*\* head, int data) {

// allocate memory for node

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

// assign data to newNode

newNode->data = data;

// assign null to next of newNode

newNode->next = NULL;

// store the head node temporarily (for later use)

struct Node\* temp = \*head;

// if the linked list is empty, make the newNode as head node

if (\*head == NULL) {

newNode->prev = NULL;

\*head = newNode;

return;

}

// if the linked list is not empty, traverse to the end of the linked list

while (temp->next != NULL)

temp = temp->next;

// now, the last node of the linked list is temp

// assign next of the last node (temp) to newNode

temp->next = newNode;

// assign prev of newNode to temp

newNode->prev = temp;

}

// delete a node from the doubly linked list

void deleteNode(struct Node\*\* head, struct Node\* del\_node) {

// if head or del is null, deletion is not possible

if (\*head == NULL || del\_node == NULL)

return;

// if del\_node is the head node, point the head pointer to the next of del\_node

if (\*head == del\_node)

\*head = del\_node->next;

// if del\_node is not at the last node, point the prev of node next to del\_node to the previous of del\_node

if (del\_node->next != NULL)

del\_node->next->prev = del\_node->prev;

// if del\_node is not the first node, point the next of the previous node to the next node of del\_node

if (del\_node->prev != NULL)

del\_node->prev->next = del\_node->next;

// free the memory of del\_node

free(del\_node);

}

// print the doubly linked list

void displayList(struct Node\* node) {

struct Node\* last;

while (node != NULL) {

printf("%d->", node->data);

last = node;

node = node->next;

}

if (node == NULL)

printf("NULL\n");

}

int main() {

// initialize an empty node

struct Node\* head = NULL;

insertEnd(&head, 5);

insertFront(&head, 1);

insertFront(&head, 6);

insertEnd(&head, 9);

// insert 11 after head

insertAfter(head, 11);

// insert 15 after the seond node

insertAfter(head->next, 15);

displayList(head);

// delete the last node

deleteNode(&head, head->next->next->next->next->next);

displayList(head);

}

int main()

{

/\*printf("mango city",printf("salem",printf("bye bye")));\*/

/\*int a,b,c,d;

scanf("%d",&a,scanf("%d",&b),scanf("%d",&c),scanf("%d",&d));

printf("a=%d b=%d c=%d d=%d",a,b,c,d);\*/

/\*return 5;\*/

/\*if(0.1)

{

printf("chandru vs Mrs.chandru");

}\*/

/\*if(printf("hello"))

{

printf("how is ur x chandru?");

}\*/

/\*if(printf("%d",0))

{

printf("how is ur x chandru?");

}\*/

/\*

if(printf("0"))

{

printf("how is ur x chandru?");

}

\*/

/\*if(printf(""))

{

printf("how is ur x chandru?");

}\*/

/\*

if(0)

{

a:

printf("hello");

}

else

{

printf("hai");

goto a;

}

\*/

int a=10;

if(a-- == 10)

{

printf("tata");

}

//a=10;

if(9==a--)

{

printf("bye");

}

}

GRAPHS

BFS AND DFS

// BFS algorithm in C

#include <stdio.h>

#include <stdlib.h>

#define SIZE 40

struct queue {

int items[SIZE];

int front;

int rear;

};

struct queue\* createQueue();

void enqueue(struct queue\* q, int);

int dequeue(struct queue\* q);

void display(struct queue\* q);

int isEmpty(struct queue\* q);

void printQueue(struct queue\* q);

struct node {

int vertex;

struct node\* next;

};

struct node\* createNode(int);

struct Graph {

int numVertices;

struct node\*\* adjLists;

int\* visited;

};

// BFS algorithm

void bfs(struct Graph\* graph, int startVertex) {

struct queue\* q = createQueue();

graph->visited[startVertex] = 1;

enqueue(q, startVertex);

while (!isEmpty(q)) {

printQueue(q);

int currentVertex = dequeue(q);

printf("Visited %d\n", currentVertex);

struct node\* temp = graph->adjLists[currentVertex];

while (temp) {

int adjVertex = temp->vertex;

if (graph->visited[adjVertex] == 0) {

graph->visited[adjVertex] = 1;

enqueue(q, adjVertex);

}

temp = temp->next;

}

}

}

// Creating a node

struct node\* createNode(int v) {

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

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Creating a graph

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*));

graph->visited = malloc(vertices \* sizeof(int));

int i;

for (i = 0; i < vertices; i++) {

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

// Add edge

void addEdge(struct Graph\* graph, int src, int dest) {

// Add edge from src to dest

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

// Create a queue

struct queue\* createQueue() {

struct queue\* q = malloc(sizeof(struct queue));

q->front = -1;

q->rear = -1;

return q;

}

// Check if the queue is empty

int isEmpty(struct queue\* q) {

if (q->rear == -1)

return 1;

else

return 0;

}

// Adding elements into queue

void enqueue(struct queue\* q, int value) {

if (q->rear == SIZE - 1)

printf("\nQueue is Full!!");

else {

if (q->front == -1)

q->front = 0;

q->rear++;

q->items[q->rear] = value;

}

}

// Removing elements from queue

int dequeue(struct queue\* q) {

int item;

if (isEmpty(q)) {

printf("Queue is empty");

item = -1;

} else {

item = q->items[q->front];

q->front++;

if (q->front > q->rear) {

printf("Resetting queue ");

q->front = q->rear = -1;

}

}

return item;

}

// Print the queue

void printQueue(struct queue\* q) {

int i = q->front;

if (isEmpty(q)) {

printf("Queue is empty");

} else {

printf("\nQueue contains \n");

for (i = q->front; i < q->rear + 1; i++) {

printf("%d ", q->items[i]);

}

}

}

int main() {

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 4);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

addEdge(graph, 3, 4);

bfs(graph, 0);

return 0;

}

// DFS algorithm in C

#include <stdio.h>

#include <stdlib.h>

struct node {

int vertex;

struct node\* next;

};

struct node\* createNode(int v);

struct Graph {

int numVertices;

int\* visited;

// We need int\*\* to store a two dimensional array.

// Similary, we need struct node\*\* to store an array of Linked lists

struct node\*\* adjLists;

};

// DFS algo

void DFS(struct Graph\* graph, int vertex) {

struct node\* adjList = graph->adjLists[vertex];

struct node\* temp = adjList;

graph->visited[vertex] = 1;

printf("Visited %d \n", vertex);

while (temp != NULL) {

int connectedVertex = temp->vertex;

if (graph->visited[connectedVertex] == 0) {

DFS(graph, connectedVertex);

}

temp = temp->next;

}

}

// Create a node

struct node\* createNode(int v) {

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

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Create graph

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*));

graph->visited = malloc(vertices \* sizeof(int));

int i;

for (i = 0; i < vertices; i++) {

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

// Add edge

void addEdge(struct Graph\* graph, int src, int dest) {

// Add edge from src to dest

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

// Print the graph

void printGraph(struct Graph\* graph) {

int v;

for (v = 0; v < graph->numVertices; v++) {

struct node\* temp = graph->adjLists[v];

printf("\n Adjacency list of vertex %d\n ", v);

while (temp) {

printf("%d -> ", temp->vertex);

// temp = temp->next;

}

printf("\n");

}

}

int main() {

struct Graph\* graph = createGraph(4);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 2, 3);

printGraph(graph);

DFS(graph, 2);

return 0;

}

ZOHO

main()

{

int a=10;

if(a>1 | a++<5 | ++a==11)

{

printf("hello %d",a);

}

else

{

printf("hai %d",a);

}

}

#include <stdio.h>

#include<stdlib.h>

#include<math.h>

int main() {

int date,month,year;

printf("Enter the Year:");

scanf("%d",&year);

int count=log10(year)+1;

if(count>4){

printf("Invalid");

return 0;

}

printf("Enter the month:");

scanf("%d",&month);

if(month<1 || month>12){

printf("Invalid");

return 0;

}

printf("Enter the date:");

scanf("%d",&date);

if(month==1 || month==3 || month==5 || month==7 || month==8 ||month==10||month==12){

if(date<1 || date>31){

printf("Invalid");

return 0;

}

}

else if(month==4||month==6||month==9||month==11){

if(date<1 || date>30){

printf("Invalid");

return 0;

}

}

else if(month==2){

if((year%100==0 &&year%400==0)||(year%100!=0 &&year%4==0)){

if(date<1 || date>29){

printf("Invalid");

return 0;

}

}

else{

if(date<1 || date>28){

printf("Invalid");

return 0;

}

}

}

int m;

if(month>=3){

m=month-2;

}

else if(month<=2){

m=month+10;

}

int d=year%100;

int c=year/100;

int f=date+((13\*m-1)/5)+d+(d/4)+(c/4)-(2\*c);

int n;

if(f<0){

n=abs(f%7);

n=7-n;

}

else{

n=f%7;

}

switch(n){

case 0:

printf("The day is Sunday");

break;

case 1:

printf("The day is Monday");

break;

case 2:

printf("The day is Tuesday");

break;

case 3:

printf("The day is Wednesday");

break;

case 4:

printf("The day is Thursday");

break;

case 5:

printf("The day is Friday");

break;

case 6:

printf("The day is Saturday");

break;

}

}