Nawneeth KS 3D

DOMS | Page No.
Date / /

Index

* we cant use realloc after free();

1. swapping 2 variables using pointers

```
#include<stdlib.h>
void swap(int *a, int *b);
int main(){
  int a,b
  printf("Enter value of a & b:\n");
  scanf("%d %d", &a, &b);
  swap(&a, &b);
  printf("a and b after swapping are
    %d %d \n", a, b);
}
void swap(int *a, int *b){
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

output:

Enter value of of a & b:
1 2
After su
a and b after swapping are
2 1

DOMS | Page No.
Date / /

8. dynamic memory allocation

```
#include<stdlib.h>
void main(){
  int *arr,i,n,m;
  printf("input size \n");
  scanf("%d",&n);
  arr=(int*)malloc(n*sizeof(int));
  for(i=0;i<n;i++){
    scanf("%d",&arr[i]);
  }
  free(arr);
  arr=(int*)calloc(n,sizeof(int));
  for(i=0;i<n;i++){
    scanf("%d",&arr[i]);
  }
  printf("input new size:");
  scanf("%d",&m);
  for(i=0;i<m;i++){
    scanf("%d",&arr[i]);
  }
  free(arr);
}
```

Output:

input size	new size
2	3
Enter elements	enter elements
1	1
2	2
Enter elements	3

3. stack implementation

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 5
int top = -1
int arr[SIZE];

void push(int ele);
int pop();
void display();

int main(){
  int ch, ele;
  do{
    printf("stack \n");
    printf("1.PUSH \n 2.POP \n DISPLAY \n
      Exit \n");
    scanf("%d", &ch);
    switch(ch){
      case 1:
        printf("Enter element to push:");
        scanf("%d", &ele);
        push(ele);
        break;
      case 2:
        ele = pop()
        if (ele != -1) printf("popped %d
                              \n", ele);

        break;
```

```
        case 3:
            display();
            break;
        case 4:
            printf("Exited");
            break;
        default:
            printf("Enter valid option\n");
    } while(choice);
    while(c!=4);
}

void push(int ele){
    if(top==SIZE-1){
        printf("overflow\n");
    else{
        top++;
        stack[top] = ele;
    }
}

int pop(){
    if(top==-1) printf("underflow");
    else{
        int c = stack[top];
        top--;
        return c;
    }
}
```

```
void display(){
  if(top==-1) printf("Empty\n");
  else{
    printf("stack elements");
    for(int i=0; i<=top; i++){
      printf("%d", stack[i]);
    }
    printf("\n")
  }
}
```

output:

```
Stack
1. PUSH 2. POP 3. DISPLAY 4. EXIT
1
Enter element to push 4

Stack
1. PUSH 2. POP 3. DISPLAY 4. EXIT
1
Enter element to push 41

Stack
1. PUSH 2. POP 3. DISPLAY 4. EXIT
2
The popped element is 41

Stack
1. PUSH 2. POP 3. DISPLAY 4. EXIT
3
Stack elments: 4
```

21/12/23

2015 | Page No.
Date / /

Week -1

Outputs

1. Bank

Enter:
1. Create new account
2. to withdraw
3. to deposit
4. check balance
5. exits.

3

Enter amount to deposite: 50000

4

Bantance is 50000

2

Enter amount to withdraw: 10000

1

Enter account name: Nameeth KS
Enter age: 20

5

Thanks.

2. lexiographically

Enter the no of strings : 3
Enter 3 strings
apple
Banana
Orange

Bangna
apple
orange

3. QD

Enter the no of rows and columns 2 2
Enter the elements 1 2 3 4
Enter the key 2
Count 2 is present in array

4 Substring

Enter the main string : Naveeth
Enter the sub-string : vn
substring found at index 2

5 Last occurance
Enter the size of array : 5
Enter 5 elements : 1 2 2 3 1
Enter key : 1
last occurance 1 found at index 4

DOMS | Page No. 5
Date / /

Lab-3

⊙ Conversion of inflex to postflix

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

#define max-size 100;

int isOperator(char ch){
    return (ch=='+' || ch=='-' || ch
    =='/' || ch=='%');
}

int precedence(char op){
    if(op=='+' || op=='-') return 1;
    if(op=='*' || op=='/' || op=='%')
        return 2;
    return 0;
}

void inflixtopostflix(char infix[],
  char postflix[]){
  char stack[max-size];
  int top=-1
  int i,j;

  for(i=0;j=0;infli x[i] != '\0';i++){
    if(inflix[i]>='0' && infli x[i]<='9'){
      postfix[j++] = infli x[i];
    }else if(isOperator(inflix[i])){
```

```
while(top>=0 && precedence(stack[top])>=
  precedence(inflix[i])){
  postfix[j++] = stack[top--];
}
stack[++top] = inflix[i];
}else if(inflix[i] == '('){
  stack[++top] = inflix[i];
}else if(inflix[i] == ')'){
  while(top>=0 && stack[top] != '('){
    postflix[j++] = stack[top--];
  }
  if(top>=0 && stack[top]=='('){
    top--;
  }
}
}
while(top>=0){
  postflix[j++] = stack[top--];
}
postflix[j] = '\0';
}

int main(){
  char infix[max_size], postfix[
  max_size];

  printf("Enter the inflix: ");
  scanf("%s", inflix);
```

```
infixtoPostfix (infix, postfix);
printf("postfix: %s\n", postfix);
return 0;
}
```

Output:

Enter infix: 3+((4*5)/6)%9
Postfix: 345*6/9%+

Evaluation of postfix

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

#define max_size 100

int stack[max_size];
int top = -1;

void push(int item){
    if(top == max_size - 1){
        printf("Stack overflow\n");
        exit(1);
    }
    stack[++top] = item;
}

int pop(){
    if(top == -1){
        printf("underflow\n");
        exit(1);
    }
    return stack[top--];
}

int isop(char ch){
    return ch=='+'||ch=='-'||
           ch=='*'||ch=='/'||ch=='
}
```

inf= 3+4

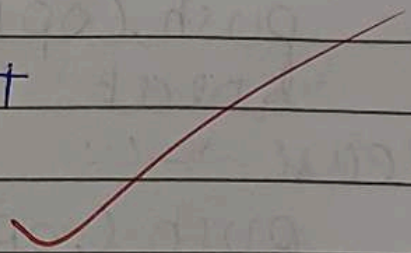
```
int eval(char postfix[])
    int i=0;
    while (postfix[i] != '\0'){
      char current = postfix[i];
      if(isdigit(current)){
        push(current - '0');
      }else if(isop(current)){
        int op2 = pop();
        int op1 = pop();
        switch(current){
          case '+':
            push(op1+op2);
            break;
          case '-':
            push(op1-op2);
            break;
          case '*':
            push(op1*op2);
            break;
          case '/':
            push(op1/op2);
            break;
          case '%':
            push(op1%op2);
            break;
        }
      }
      i++
    }
    return pop();
}
```

```
int main(){
    char post[100];
    printf("Enter");
    scanf("%s",post);

    int re = eval(post);
    printf("Result: %d\n",re);
    return 0;
}
```

Output:

Enter: 345*6/9%+
Result: 6



implementation of
Linear queue

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

#define max 3

int front = -1, rear = -1;
int queue[max];

void insert(int item){
  if(rear == max-1){
    printf("Overflow \n");
    exit(exit fail());
  }else{
    if(rear == -1 && front == -1){
      front = rear = 0;
      queue[rear] = item;
    }else{
      if(rear == max-1)
      rear = rear + 1;
      queue[rear] = item;
    }
  }
}

int delete(){
  int n, clc, d;
  do{
```

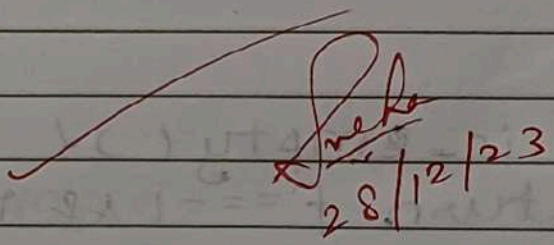
```
int n, ele, d;
do
printf("1. Insert\n2. Delete\n3. Exit\n");
scanf("%d", &n);
switch(n){
 case 1:
   printf("Enter the element:");
   scanf("%d", &ele);
   insert(ele);
   break;

 case 2:
   d = delete();
   if (d == -1){
     printf("underflow\n");
     exit(1);
   }
   printf("The element is %d\n", d);
   break;

 case 3:
   printf("Exit\n");
   break;
 default:
   printf("Right choice.\n");
}while(n != 3);
return 0;
}
```

output

1. Insert
2. Delete
3. exit
1
Enter the element: 34
1. Insert
2. Delete
3. exit
The element deleted is 34
1. Insert
2. Delete
3. exit
2
~~The item~~
underflow



11/1/24

Circular Queue

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

#define MAX 6

int cq[MAX];
int f = -1, r = -1

bool is_full(){
    return (r+1)%MAX == f;
}

bool is_empty(){
    return f==-1 && r==-1;
}

void insert(int item){
    if(is_full()){
        printf("Overflow: Circular queue
            is full\n");
        return;
}}

    if(is_empty()){
        front = rear = 0;
    }else{
```

```
    r = (r+1) % MAX;
}
cq[r] = item;
printf("Enqueued: %d\n", item);
}

int dequeue(){
  if(is_empty()){
    printf("Underflow\n");
    return -1;
  }

  int d = cq[f];

  if(f == r){
    f = r = -1;
  }else{
    f = (f+1) % MAX;
  }
  printf("Dequeue: %d\n", d);
  return d;
}

int main(){
  int n, ch;
  do{
    printf("\n1. Insert\n2. Delete\n3
      Exit\n");
    scanf("%d", &ch);
    rose
```

```
switch(n);
case 1:
    printf("Enter the element to
     be inserted: ");
    scanf("%d", &ele);
    insert(ele);
    break
case 2:
    int d = dequeue();
    if (d != -1)d
        printf("%d", d);
    }
    break;

case 3:
    printf("marks\n");
    break;

default:
    printf("Please enter the correct
                        option\n");
}while(n != 3);

return 0;

}
```

Output:

```
1. Insert
2. Delete
3. Exit
1
Enter the element to be inserted: 23
Enqueue: 23

1. Insert
2. Delete
3. Exit
Dequeued: 23

1. Insert
2. Delete
3. Exit
Underflow: Circular queue is empty
```

11/1/24

Singly Linked List

Q To insert an element

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

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

struct Node* Atbegin(struct Node *first,
                     int data){
    struct Node* ptr=(struct Node*)
                malloc(sizeof(struct Nod
                ));

    ptr->data = data;
    ptr->next = first;
    fi
    return ptr;
};

Struct Node * middle(struct Node *start
        int data, int pos){
  Struct Node* ptr =(struct Node *)
  ptr->data = data;

  struct Node * p = start;
```

```
                                    && p!=NULL
int i=0
while (i<pos-1){
    p=p->next;
    i++;
}
if (p==NULL){
    printf("out of bounds\n");
    free(ptr);
    return start;
}
ptr->next = p->next
p->next = ptr
return start;
}

struct Node * end(struct Node *finst, int d){
  struct Node* ptr = ~~Node~~
                     (struct Node *)
ptr->data = data;
ptr->next = NULL

if(start == NULL) return ptr;

struct Node *p = start;
while(p->next != NULL){
    p = p->next;
}
p->next=ptr
return start;
}
```

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

int main(){
  struct Node * first = NULL;
  int choice, data, pos;

  do{
  printf(" choose an option\n
          1. Insert at beg\n
          2. Insert at middle\n
          3. Insert at end\n
          4. Display\n
          5. Exit\n");
          scanf("%d", &choice);
switch(choice){
  case 1:
  printf("Enter a element : ");
  scanf("%d", &data);
  first = insertstart(first, data);
  break;

case 2:
  scanf("%d", &data);
        scanf("%d", &pos);
        first = middle(first, data,
                       pos);
        break;
```

```
case 3:
    printf("Enter the new element in ");
    scanf("%d", &data);
    first = end(first, data);
    break;

case 4:
    exit break
}
}while (choice != 5);
return 0;
}
```

output:

```
Choose an option:
1. Insert at beg    4. Display
2. Insert at mid    5. exit
3. Insrt at end
1
Enter element 1
1 inserted at bcgining
2
Enter element 2
Enter position 0
2 inserted at middle
3
Enter element 3
3 inserted at end
4
Element 1
Element 2
Element 3
```

5
Quitting

Sneha
11/1/24

lab-5

perform operations on dulte for singly linked list

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

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

void atAtIndex(struct Node**, int i,
int d){
  struct Node *new = (struct Node *)
                  mallo c(sizeof(struct
new->data = data;                 Node));

if(i==0){
  new->next = *head;
  *head = new
}else{
  struct Node* temp = *head
  for(int i=0; i<i-1 && temp!=NULL
    temp = temp->next;          i+1
  }

if(temp==NULL){
  printf("Invalid index!\n");
  free(new);
  return;
}
```

CO13

DOMS | Page No.
Date / /

```
    new->next = temp->next
    temp->next = new;
}

void deletestart (struct Node** head){
    if (*head == NULL){
        printf("List is empty");
        return;
    }

    struct Node* temp = head;
    head = temp->next;
    free(temp);

    printf("
}

void deleteIndex (struct Node* start, int
                                    i){

    if (*head == NULL){
        printf("List is empty");
        return;
    }

    struct Node * temp = head
    if (index == 0){
        head = temp->next;
        free(temp);
        ptr;
    } else {
```

```
for(int i=0; i<index-1 &&
    temp!=NULL; i++){
    temp = temp->next;
}
if(temp==NULL || temp->next==NULL){
  printf("Invalid index!\n");
  return 0;
}

struct Node* todelete = temp->next;
temp->next = todelete->next;
free(todelete);
}
}

int main(){
  struct Node* head = NULL;
  int choice, index, data;

while(1){
  printf("\n~~dd~~ ~~element at beg~~\n
        2. Delete at first
        3. Delete at index
        4. Delete at end
        5. Display
        6. Exit");
```

```
switch (choice){
  case 1:
    printf("Enter index and data to
        add: ");
    scanf("%d %d", &index, &data);
    break;

  case 2:
    deleteAtStart(head);
    break;

  case 3:
    printf("Enter index to delete: ");
    scanf("%d", &index);
    deleteAtIndex(head, index);
    break

  case 4:
    deleteAtEnd(head)

  case 5:
    ~~deleteAt~~ display(head);

  case 6:
    exit(0);

  default:
    printf("Invalid choice!\n");
  }
}
return 0;
```

output:

1. Add element at a given index
2. Delete at start
3. Delete at index
4. Delete at end
5. Display
6. Exit

1
Enter index and data: 0 1
Element 1 added at 0

1
Enter index and data: 1 2
Element 2 added at 1

4
Element delete at the end

5
Linked List: 1

2
Element deleted at start

6
Exiting....

Sneha
18/11/24

25-01-24

lab-5

```
#include<stdlib.h>
struct Node{
  int data;
  struct Node* next;
};

void sort(struct Node* first){

  struct Node * ptr, *p;
  for(ptr=first; ptr!=NULL; ptr=ptr->next){
    bool s=false;
    for(p=first; p!=NULL; p=p->next){
      if(p->data>(p->next)->data){
        int t=p->data;
        p->data=(p->next)->data
        (p->data)->next=t;
      }
    if(!s) break;
  }
}

Struct Node* reverse(struct Node* first){
  struct Node * prev=NULL
  struct Node *current=first
  struct Node* next=NULL;

  while(current!=NULL){
    next=current->next
    current->next=prev;
    prev=current
  } current=next;
```

```
int main(){

    struct Node *f, *s, *t;
    f = (struct Node*) malloc (sizeof(Node));
    s = (struct Node*) malloc (sizeof(Node));
    t = (struct Node*) malloc (sizeof(Node));

    &
    f->data = -1;
    s->data = 10;
    t->data = 3;

    f->next = s;
    s->next = t;
    t->next = NULL;

      reverse
    f = sort(f)          // reversing
    display(f)

    sort(f)              // sorting

}
```

Output:

```
// input &
Element: -1
Element: 10
Element: 3
```

// reversing

Element: 3
Element: 10
Element: -1

// sorting

Element: -1
Element: 3
Element: 10

Implementation of stack & Queues

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

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

struct Node *push(struct Node *top, int data){
  struct Node *newNode = (struct Node*)
  if(newNode == NULL){
    printf("memory allocation\n");
  }
```

```
    newNode->data = data;
    newNode->next = top;

    return newNode;
}

struct Node* pop(struct Node* top){
    if(top==NULL) return NULL;
    struct Node* temp = top;
    top = top->next;
    free(temp);
    return top;
}

void display(struct Node* top){
    if(top==NULL) return;
    printf("Stack Elements: ");
    while(current!=NULL){
        printf("%d", current->data);
        current = current->next;
    }
    printf("\n");
}

int main(){
    struct Node* top = NULL;
    int c, e;
    do{
        printf("1. Push 2. Pop 3. Display 4. Exit");
        scanf("%d", &choice);
```

```
switch(c)
  case 1:
    printf("Enter the element");
    scanf("%d", &el);
    top = push(top, el);
    break;

  case 2:
    top = pop(top);
    break;

  case 3:
    display(top);
    break;

  case 4:
    printf("Exiting\n");

}while(c!=4);

return 0;

}
```

Output:

1. PUSH
2. POP
3. Display
4. EXIT

1.
Enter element: 12

1.
Enter element -10

3.
-10 12

2.
Removed -10

2.
Removed 12

2.
Overflow

4.
Exiting.

Queue

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

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

struct Node* en(struct Node* rear, int d){
  struct Node* newNode =(struct Node*)
                    malloc(sizeof(Node));
  newNode->data = data;
  newNode->next = NULL

  if(rear==NULL) return newNode;

  rear->next = newNode
  return newNode;
}

struct Node* deque(struct Node* front){
  if(front==NULL){return NULL;
  struct Node* tmp = front;
  front = front->next
  free(tmp);
  return front;
}
```

```
void display(struct Node, * front){
  printf("Queue element \n") return;
  while (current != NULL){
    printf("%d", current -> data);
    current = current -> next;
  }
  printf("\n");
}

int main(){
  struct Node * f = NULL
  struct Node * r = NULL
  ~~rear = create(rear)~~
  // adding element
  rear = enque(rear, 1)
  rear = enque(rear, 2)
  ~~rear~~
  // removing elements
  front = deque(front)
  front = deque(front)
  // displaying element
  display(front)

return 0;
}
```

output:

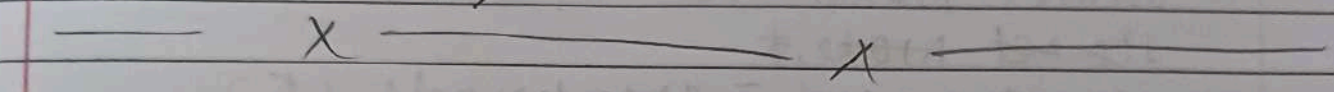
input: 1 2

display: 1 2

dequeue: 1

dequeue: 2

dequeue: overflow



Concatenating

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

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

Struct Node * contact(struct Node* l1,
 Struct Node *l2){
  struct Node* current = l1;
  while(current->next != NULL){
    current = current->next;
  }
  current->next = l2;
  return l2;
}
```

```
int main(){
  struct Node* l1 = createNode(1);
  l1->next = createNode(2);
  l2->next->next = concatNode(3);

  struct Node* l2 = createNode(4);
  struct Node*
  l1->next = createNode(5)
  struct->next->next->next = concatNode(6);

  printf("original list 1: ");
  printlist(l1);
  printf("original list 2: ");
  printf(l2);

  = contact(l1, l2)
  printlist(l1);

  return 0;

}
```

output:

l1 : 1 → 2 → 3 → NULL
l2 : 4 → 10 → 6 → NULL

Concatenated list : 1 → 2 → 3 → 4 → 10 → 6

1-2-24

Q. Implement all the primitive operations of doubly linked list.

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

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

struct Node* Insert( first, int data, int pos
  struct Node* ptr = new Node.(data);
  if( pos==0 ){
    ptr->next = first;
    if( first != NULL) first->prev=NULL;
    else first=NULL;
    free(ptr);
    return first;

Struct Node* p1=first , p2=first;
int i=0
while( i!=pos && p2 != NULL){
  p1=p2;
  p2=p2->Next;
  i++;
}
```

```
<
if (i != pos) return;
p1 -> next = ptr
ptr -> ncxt = p2
p2 -> prw = ptr
ptr -> prw = p1;
Return first;
}

struct Node* Delete(first, int pos){
  if (pos == 0){
    struct Node * ptr = first;
    if (first -> ncxt != NULL){
      first = first -> ncxt;
      First -> prw = NULL;
    else first = NULL
    free(ptr);
    return first;
  }

  struct Node* p1 = first, *p2 = first;
  int i = 0
  while (i != pos && p2 != NULL){
    p1 = p2
    p2 = p2 -> ncxt;
    i++;
  }
  if (i != pos) return 0;
  p1 -> ncxt = p2 -> ncxt;
  if(p2 -> ncxt != NULL) p2 -> ncxt -> prw = p1;
  free(p2)
rtn first first
}
```

```
int main(){
  struct Node* head = NULL;
  int ch, data, pos;

while(1){
  printf("2. Insert\n2. Delete\n3. Display
    \n4. Exit");
  printf("Enter the choice: ");
  scanf("%d", &ch);

switch(ch){
  case 1:
  printf("Enter data to insert: ");
  scanf("%d", &data);
  printf("Enter pos: ");
  scanf("%d", &pos);
  head = insert(head, data, pos);
  break;

  case 2:
  printf("Enter pos to delete: ");
  scanf("%d", &pos);
  head = Delete(head, pos);
  break;
 case 3:
  display(head)
  break;
 case 4:
  exit(0)
 }
return 0;
}
```

output:

Options
1. Insert
2. Delete
3. Display
4. Exit

1.
Enter data: 1
Enter position: 0

1
Enter data: -1
Enter position: 0

1
Enter data: 2
Enter position: 1

2
Enter position: 2

2
Enter position: 5
Invalid

3
Element: -1
Element: 1

4.
Exiting...

11/2/24

Date / / /

15-2-24

2. TO implement Binary search Tree

```
#include<stdio.h>
struct TreeNode{
   int key;
   struct TreeNode* left;
   struct TreeNode* right;
};

struct TreeNode* createNode(int key){
   struct TreeNode* newNode;
   newNode->key = key
   newNode->left = newNode->right = NULL;
   return newNode;
}

struct TreeNode* insert(root, key){
   if(root==NULL)return rootNode(key);
   if(key<root->key)
      root->left=insert(root->left, key);
   else if(key>root->key)
      root->right=insert(root->right, key);
   return root;
}
```

```
void inorderTraversal(struct TreeNode* root){
    if(root != NULL)
        inorderTraversal(root->left);
        printf("%d", root->key);
        inorderTraversal(root->right);
    }
}

void preorderTraversal(root){
    if(root != NULL){
        postorderTraversal(root->left)
        postTraver(root->right)
        printf("%d", root->key)
    }
}

int main(){
    struct TreeNode* root = NULL;
    int key;

    printf("Enter node : ");
    while(1){
        scanf("%d", &key);
        if(key == -1)
            break;
        root = insert(root, key)
    }

    inorder(root);
    preorder(root);
    postorder(root);
```

```
printf("In-order Traversal: \n")
inorderTraversal(root)

preorderTraversal(root);

postorderTraversal(root);

return 0;
}
```

output:

5 3 1 2 4 7 6 8

Enter nodes: ~~1 2 3 4 5 6 7 8~~

in-order: 1 2 3 4 5 6 7 8

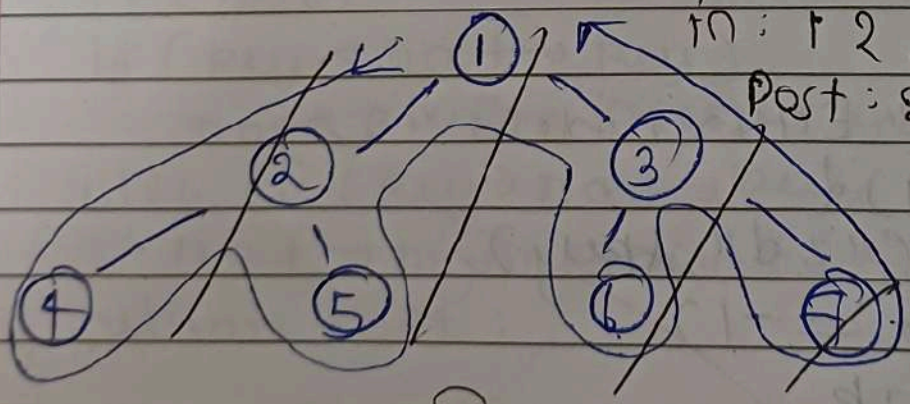
Pre-order: 4 2 5 1 6 3 7

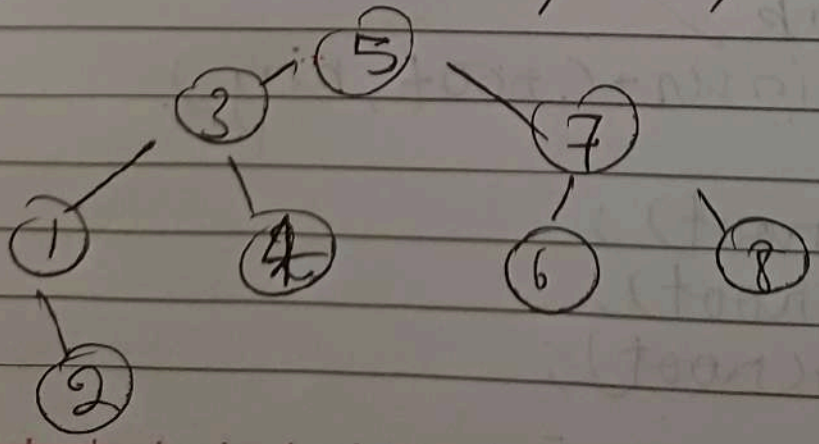
Post-order: 4 5 2 6 7 3 1

Pre: 2 1 3 4 5 7 6 8

in: 1 2 3 4 5 6 7 8

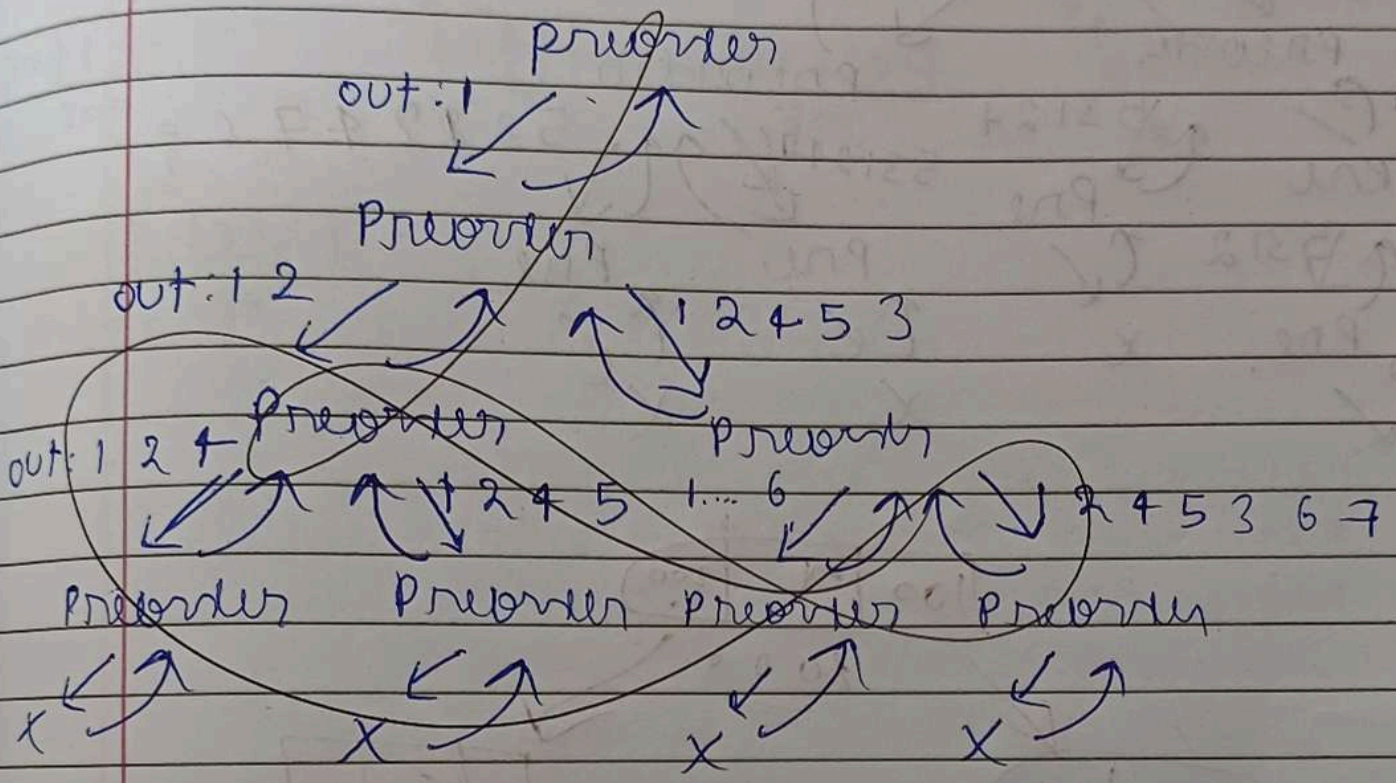
Post: 2 1 4 3 6 8 7 5

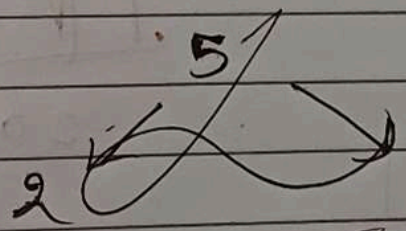


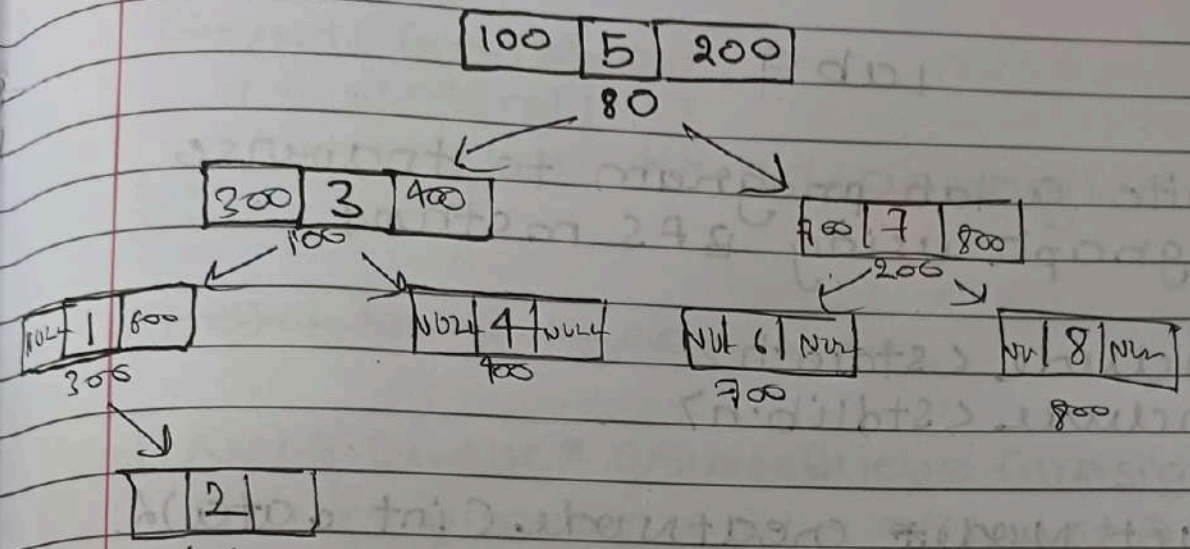


Tracing

root != NULL ⟶ output : 1






100 5 200
80
300 3 400
100
700 7 800
200
NULL 1 600
300
NULL 4 NULL
400
NULL 6 NULL
700
NULL 8 NULL
800
2
600

22/2/24

lab 9

1. Write a lab program to traverse a graph using BFS method.

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

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

struct Queue* create(unsigned capacity

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

struct Queue{
    int rear, front;
    unsigned capacity;
    int* arr;
};
```

```
struct Graph{
  int vertices;
  struct Node ** adjacencyList;
};

struct Node * createNode

struct Queue* createQueue (unsigned cap){
  struct Queue* queue =(struct Queue *)
  malloc(sizeof(struct Queue));
  struct
  Queue->capacity = capacity
  Queue->front = queue->rear = -1
  Queue->array = (int *)malloc
   (queue->capacity * sizeof(int));
  return queue;
}

struct Graph* createGraph(int vertices){
  struct Graph* graph =(struct Graph*)
  graph->vertices = vertices
  return graph;
}

void addEdge(struct Graph* graph,
int src, int dest){
  struct Node* newNode *;
  newNode = createNode(src);
}
```

```
void BFS(struct Graph* graph, int
        startVertex){
  struct Queue* queue = createQueue(100);
  int visited = (int*) malloc(graph->
  vertices * sizeof(int));
  enqueue(queue, startVertex);
  while(!isEmpty(queue)){
    int currentVertex = dequeue(queue)
    while(temp != NULL)
      int adjacentVertex = temp->data;
      if(!visited){
        visited[adjacentVertex] = 1;
        enqueue(queue, adjacentVertex)
      }
      temp = temp->next;
    }
  }
  free(queue->array);
  free(queue);
  free(visited);
}

int main(){
  int vertices = 4;
  struct Graph* graph = createGraph(
  addEdge(Graph, 0, 1));
  addEdge(Graph, 0, 2);
  addEdge(Graph, 1, 2);
  addEdge(Graph, 2, 0);
  addEdge(Graph, 2, 3);
  addEdge(Graph, 3, 3);
```

```
int startVertex = 1;
BFS(graph, startVertex);
return 0;
}
```

output:

Breadth First Traversal
(starting vertex from 1): 1 2 0 3

Leet code-1

Min stack

```
typedef struct {
  int * array;
  int topindex;
  int *minarray;
} minstack;

minstack* minStackCreate() {
  minstack * m = malloc(sizeof(minstack));
  m->array = (int*)calloc(300001, sizeof(int));
  m->minarray = (int*)calloc(300001);
  m->topindex = 0
  return m;
}

void minstackpush(minstack * m, int val){
  m->array[m->topIndex] = val;
  m->topindex++;
}

void minstackpop(minstack *m){
  m->topindex--;
}

int minstackTop(minstack* m){
  return m->array[m->topindex-1]
}
```

```
int minstackmin(minstack *m){
    return m->minarray[m->topindex-1]
}

void minstackFree(minstack *){
    free(m->array);
    free(m->minarray)
    free(ms);
}
```

last:

Leetcode-2

Reversing a given linked list.

```
struct ListNode* reverse(){
  struct ListNode *p = head, *k = head;
  int i = 1, kk = 0;
  int a[100000];
  while(p != NULL){
    if(i >= left && i <= right){
      arr[kk++] = p->val;
    }
    p = p->next;
    i++;
  }
  i = 1
  while(k != NULL){
    if(i >= left && i <= right){
      k->val = a[--kk]
    }
    k = k->next;
    i++;
  }
  return head;
}
```

DOMS | Page No.
Date / /

Leetcode-3

Split input Linked List

```
struct ListNode** splitarray(){
  struct ListNode** ans = struct**
  struct ListNode *prw;
  int base, len=0, pant =0;
  for(struct ListNode *tmp=he.ad){
    len++;
  }
  base = len/k;
  for(int i = len%.k ; i>0 ; i--){
    ans[pant] = head;
    pant++;
    for(int i=0 ; i<(base+1) ; i++){
      prw = head;
      head = head.->next;
    }
    prw ->next = NULL;
  }
  if(base){
    for(int i=pant; i<k; i++){
      ans[pant] = he.ad;
      pant ++;
      prw ->next = NULL;
    }
  }
  *returnsize = k
  return ans
}
```

Lectcode-4

Rotate List

```
int getlength(struct ListNode* head)
{
  if(head==NULL)
     return 0;
  return 1+ getlength(head->next);
}

struct ListNode* Rotate(struct ListNode*
  if(head==NULL||k==0)
    return head;
  int len = getlength(head);

  if(len==1) return head;
  for(int i=0;i<k%.length;i++){
    struct ListNode *p = head;
    while(p->next->next!=NULL){
      p=p->next;
    }
    struct ListNode *a=(struct ListNod
    malloc(sizeof(struct ListNode));
    a->val = p->next->val;
    a->next =head
    head = a;
  }
  return head;
}
```

DOMS | Page No.
Date / /

Leetcode - 5
Swap Node

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

void swapnode(Node* root, int depth)
{
  if(root==NULL) return;
  if(current%depth==0)
  {
    Node *temp = root->left;
    root->left = root->right
    root->right = temp;
  }
  swapNodedepth(root->left, depth, current+1);
  swapNodedepth(root->right, depth, current+1);

int ** swapNode(int indexes_rows, int
indexes_columns, int** indexes){
  Node * root = (Node*) malloc(sizeof(Node));
  root->data=1;
  root->left=NULL;
  root->right=NULL;

  Node * queue[indexes_rows];
  int front=-1=rear;

  return result;
}
```

Write a program to check whether a given graph is connected or not using DFS method

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

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

struct Graph{
  int vertices;
  struct Node** adjacencyList;
};

struct Node* createNode(int data){
  struct Node* newNode = (struct Node*)
  newNode->data = data;
  newNode->next = NULL;
  return newNode;
}

struct Graph* createGraph(int vert){
  struct Graph* graph = (struct Graph*)
  graph->vertices = vertices;
  graph->adjacency = (struct Node**)
  for(int i=0; i<vertices; i++)
    graph->adjacency[i]=NULL;
}
return graph;
```

```
void addEdge(struct Graph* graph, int
    src, int dest){
struct Node* newNode = createNode(dest);
newNode->next = graph->adjacencyList;
graph->adjacencyList[src] = newNode;

newNode = createNode(src);
newNode->next = graph->adjacency
graph-> newNode;
}

void DFSUtil(struct Graph* graph, int v){
visited[vertex] = 1;
struct Node* temp = graph->vertex;
while(temp!=NULL){
 int adjacencyvertex = temp->data;
 if(!visited[adjacencyvertex])
 temp = temp->next;
 }
}


int isConnected(struct Graph* graph){
 int* visited = (int*) malloc(graph->vertices
 for(int i=0; i<graph->vertices; i++)
   visited[i]=0;


 DFSUtil(graph, 0, visited
 free(visited);
 return 1;
}
```

```
int main() {
    int vertices = 4;
    struct Graph* graph = createGraph
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);

    if(isConnected(graph))
        printf("The graph is connected.\n");
    else
        printf("The graph isn't connected.\n");
    return 0;
}
```

29/02/2024

DOMS | Page No.
Date / /

Lab-10
Hashing

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

#define MAX_EMPLOYEES 100

typedef struct {
    int key;
} Employee;

typedef struct {
    Employee * table[MAX_EMPLOYEES];
    int size;
} HashTable;

int hashFunction (int key, int m){
    return key % m;
}

void insert (Employee * employee,
HashTable * hashTable, int m){
    int key = employee -> key;
    int index = hashFunction(key, m);

    while (hashTable -> table[index] != NULL){
        index = (index + 1) % m;
    }
    hashTable -> table[index] = employee;
    hashTable -> size ++;
}
```

```
Employee * search(int key, HashTable*
hashTable, int m){

int index = hashFunction(key, m);

while(hashTable->table[index]
    = NULL){
if(hashTable->table[index]->key
    == key){
    return hashTable->table[index];
}
index = (index + 1) % m;
return NULL;
}

void display(HashTable* hashTable){
printf("\n Hash Table: \n");
for(int i=0; i<MAX_EMPLOYEES; i++){
if(hashTable->table[i] != NULL){
    printf("index %d: key %d \n",
i, hashTable->table[i]->key);
}
}
}
```

```
int main()

    HashTable hashTable;
    hashTable.size = 0;

    int m;
    printf("Enter the number of
      memory location: ");
    scanf("%d", &m);

    Employee * employees[MAX_EMPLOYEES]
    int n;
    printf("Enter the number of
      employee records: ");
    scanf("%d", &n);

    for(int i=0; i<n; i++){
      employee[i] =(Employee*)
      malloc(sizeof(Employee));
      printf("Enter key for Employee
        %d: ", i+1);
      scanf("%d", &employees[i]->key);

    for (int i=0; i< MAX_EMPLOYEES; i++){
      hashTable.table[i] = NULL;
    }
```

```
Employee* foundEmployee =
search(searchKey, &HashTable,
if(foundEmployee != NULL){
printf("Employee with key %d
 found:\n", searchKey);
}
else{
printf("Employee with key %d
 not found.\n", searchKey);

return 0;
}
```

output:

```
Enter the number of memory locations:
Enter the number of employee records:
Enter the employee records:
Enter Key for Employee 1: 213
Enter key for Employee 2: 543
Enter key for Employee 3: 567
Enter key for Employee 4: 876
Enter key for Employee 5: 345

Hash Table:
Index 0 : key 345
Index 1 : key 876
Index 2 : key 567
Index 3 : key 213
Index 4 : key 543
```

DOMS | Page No.
Date / /

Enter the key to search : 333

Element not found

X X