MISCELLANEOUS

- scanf("%[^\n]s", s); - take string input up till the newline character
- strcpy(arr[n], x); - copies string 1 to string 2
- __FILE__ - macro to get current filename
- atoi(char *) – character array to int
- itoa(int) – int to character array
- atof
- ftoa
- sprintf(str,"%d",value)
- sscanf(mainfile, "dat%d.csv", &entries);
- %*c – reading a character but ignore it – useful in fscanf for \n
- Int status = fscanf(…) – status = number of args to be read, will be 1 if we get EOF

ENUM

```
typedef enum {
	SUNDAY = 0,
	MONDAY,
	TUESDAY,
	WEDNESDAY,
	THURSDAY,
	FRIDAY,
	SATURDAY
} DayOfWeek;
int main() {
	int today;
	printf("Enter today's day (0 for Sunday, 1 for Monday, ..., 6 for Saturday): ");
	scanf("%d", &today);
	switch (today) {
		case SUNDAY:
		printf("Today is Sunday.\n");
		break;
		case MONDAY:
		printf("Today is Monday.\n");
		break;
		case TUESDAY:
		printf("Today is Tuesday.\n");
		break;
		case WEDNESDAY:
		printf("Today is Wednesday.\n");
		break;
		case THURSDAY:
		printf("Today is Thursday.\n");
		break;
		case FRIDAY:
		printf("Today is Friday.\n");
		break;
		case SATURDAY:
		printf("Today is Saturday.\n");
		break;
		default:
		printf("Invalid input. Please enter a number between 0 and 6.\n");
	}
	return 0;
}
```

MAKEFILE

```
vowel : count_vowels_exe
    ./count_vowels_exe
consonant : count_consonants_exe
    ./count_consonants_exe
count_vowels_exe : count_vowels.o master.o
    gcc -o count_vowels_exe count_vowels.o master.o
count_vowels.o : count_vowels.c
    gcc -c count_vowels.c
count_consonants_exe : count_consonants.o master.o
    gcc -o count_consonants_exe count_consonants.o master.o
count_consonants.o : count_consonants.c
    gcc -c count_consonants.c
master.o : master.c
    gcc -c master.c
clean :
    rm -f *.o
    rm -f *exe
```

POINTER

```
int var = 20;
int *ip = NULL;
printf("Value of null pointer is = %x\n", ip);
// %x can be replaced by %p here as well
ip = &var;
printf("Address of var variable is = %x\n", &var);
printf("Address in ip is = %x\n", ip);
printf("Value of *ip variable is = %d\n", *ip);
```

DMA + MENU

DO NOT DEFINE NEW VARIABLES INSIDE SWITCH CASES

```
#define MAX 100
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int add_end(char** arr, int n){
    char* x = (char*) malloc(MAX*sizeof(char));
    printf("Enter the string to be added : ");
    scanf("%s", x);
    arr = realloc(arr, (n+1)*sizeof(char*));
    arr[n] = (char*) malloc(MAX*sizeof(char));
    strcpy(arr[n], x);
    return n+1;
}
int add_start(char** arr, int n){
    char* y = (char*) malloc(MAX*sizeof(char));
    printf("Enter the name to be added : ");
    scanf("%s", y);
    char** arr_new = malloc((n+1)*sizeof(char*));
    for (int i = 0; i < n+1; i++){
        arr_new[i] = (char*) malloc(MAX*sizeof(char));
    }
    if (arr_new == NULL){
        printf("Unable to allocate memory\n");
        return -1;
    }
    for (int i = 0; i < n+1; i++){
        if (arr_new[i] == NULL){
            printf("Unable to allocate memory\n");
            return -1;
        }
    }
    for (int i = 1; i < n+1; i++){
        arr_new[i] = arr[i-1];
    }
    strcpy(arr_new[0], y);
    free(arr);
    arr = arr_new;
    return n+1;
}
int delete_index(char** arr, int n){
    int index;
    printf("Enter the index to delete : ");
    scanf(" %d", &index);
    if (index > n-1 || index < 0){
        printf("Invalid index !\n");
        return n;
    }
    for(int i = index; i < n; i++){
        arr[i] = arr[i+1];
    }
    return n-1;
}
void display_len(char** arr, int n){
    printf("The length is %d\n", n);
}
void display_all(char** arr, int n){
    for (int i = 0; i < n; i++){
        printf("%s\n", arr[i]);
    }
    printf("\n");
}
int main(){
    int n;
    printf("Enter size of the array: ");
    scanf("%d", &n);
    char** arr = (char **) malloc(n*sizeof(char*));
    for (int i = 0; i < n; i++){
        arr[i] = (char*) malloc(MAX*sizeof(char));
    }
    if (arr == NULL){
        printf("Unable to allocate memory\n");
        return -1;
    }
    for (int i = 0; i < n; i++){
        if (arr[i] == NULL){

            printf("Unable to allocate memory\n");
            return -1;
        }
    }
    printf("Enter the  strings (will terminate at white space): ");
    for (int i = 0; i < n; i++){
        scanf("%s", arr[i]);
    }
    int flag = -1;
    while(flag != 0){
        printf("\nWhat would you like to do ?\n");
        printf("1. Add a string to the end of the array.\n");
        printf("2. Add a string to the beginning of the array.\n");
        printf("3. Delete the element at index \'x\' (taken as input) of the array.\n");
        printf("4. Display the length of the array.\n");
        printf("5. Display all elements of the array in sequence.\n");
        printf("0. Exit.\n");
        printf("Enter your input.\n");
        scanf(" %d", &flag);
        switch(flag){
            case 1:
                n = add_end(arr, n);
                break;
            case 2:
                n = add_start(arr, n);
                break;
            case 3:
                n = delete_index(arr, n);
                break;
            case 4:
                display_len(arr, n);
                break;
            case 5:
                display_all(arr, n);
                break;
            case 0:
                break;
            default:
                printf("Invalid input !\n");
                break;
        }
    }
    free(arr);
    return 0;
}
```

READING ENTIRE FILE

```
FILE *fptr;
    printf("%s", __FILE__);
    fptr = fopen(__FILE__,"r");
    if (fptr == NULL){
        printf("Error opening file");
        exit(1);
    }
    char c;
    while ((c = fgetc(fptr)) != EOF){
        printf("%c", c);
    }
    // char* line = (char *) malloc(100*sizeof(char));
    // while (fgets(line, 100, fptr)){
    //     printf("%s\n", line);
    // }
    return 0;
```

READING WRTING ENTIRE FILE LINE BY LINE, DELETE

```
FILE *fptr;
    fptr = fopen("text1.txt","r");
    if (fptr == NULL){
        printf("Error opening file");
        exit(1);
    }
    FILE *fwriter;
    fwriter = fopen("text2.txt","w");
    if (fwriter == NULL){
        printf("Error opening file");
        exit(1);
    }
    char* line = (char *) malloc(100*sizeof(char));
    while (fgets(line, 100, fptr)){
        fprintf(fwriter, "%s", line);
    }
    fclose(fptr);
    fclose(fwriter);
    FILE *delete;
    delete = fopen("text1.txt","w");
    fclose(delete);
    return 0;
```

TOKENISATION ONE

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
 FILE* fptr = fopen("LOTR.txt","r");
    if (fptr == NULL){
        printf("Unable to read LOTR.txt.\n");
        exit(1);
    }
    char* line = (char *) malloc(100*sizeof(char));
    while (fgets(line, 100, fptr)){
        char* words = strtok(line, " ");
        // this will assign the first word to words (first instance of encountering delimiter)
        // it replaces the delimiter with \0 or NULL
        // we call strtok in loop again to continue where we left off from
        while (words!=NULL){
            // Convert each word to lowercase for case-insensitive comparison
            for (int i = 0; words[i] != '\0'; i++) {
                words[i] = tolower(words[i]);
            }
            // char *strstr(const char *haystack, const char *needle)
            // Finds the first occurrence of the entire string needle (not including the terminating
null character) which appears in the string haystack.
            if (strstr(words,"hobbit")) hobbit++;
            words = strtok(NULL," ");
        }
    }
```

TOKENISATION TWO

```
struct morse_mapping{
    char character;
    char* symbol;
};
const struct morse_mapping map[37] = {{'A', ".-"},{'B', "-..."},{'C', "-.-."},{'D', "-.."},{'E', "."},{'F',
"..-."},{'G', "--."},{'H', "...."},{'I', ".."},{'J', ".---"},{'K', "-.-"},{'L', ".-.."},{'M', "--"},{'N', "-."},{'O', "---
"},{'P', ".--."},{'Q', "--.-"},{'R', ".-."},{'S', "..."},{'T', "-"},{'U', "..-"},{'V', "...-"},{'W', ".--"},{'X', "-..-
"},{'Y', "-.--"},{'Z', "--.."},{'0', "-----"},{'1', ".----"},{'2', "..---"},{'3', "...--"},{'4', "....-"},{'5', "....."},{'6', "-
...."},{'7', "--..."},{'8', "---.."},{'9', "----."},{' ', "/"}
};
int main(){
    FILE* fptr = fopen("msg.txt","r");
    char* line = (char *) malloc(sizeof(char)*200);
    line = fgets(line, 200, fptr);
    char* words = strtok(line, " ");
    while (words != NULL){
        for (int i = 0; i < 37; ++i) {
            if (strcmp(words,map[i].symbol)==0) {
                printf("%c", map[i].character);
                break;
            }
        }
        words = strtok(NULL," ");
    }
    printf("\n");
    fclose(fptr);
    return 0;
}
```

TOKENISATION THREE – READING A CSV WITH TOKENS

```
FILE *fp;
    fp = fopen(argv[1], "r");
    if(fp == NULL)
    {
        printf("Error opening file");
        exit(1);
    }
    char *line = malloc(100);
    Stack *s = newStack();
    int score = 0;
    float cg = 0;
    int i = 0;
    while(fgets(line, 100, fp) != NULL)
    {// READING A FILE WITH LINE BY LINE INT,FLOAT
        char *token;
        token = strtok(line, ",");
        score = atoi(token);
        token = strtok(NULL, ",");
        cg = atof(token);
        // printf("%d: Score: %d, CG: %f\n", i, score, cg);
        // You can uncomment the above line to print the values read from the file
        /*
        Write code to push the score and cg values into the stack while tracking the time and
heap performance
        */
        gettimeofday(&t1, NULL);
        push(s, iftoe(score, cg));
        gettimeofday(&t2, NULL);
        time_taken += (t2.tv_sec - t1.tv_sec) * 1e6;
        time_taken += ((t2.tv_usec - t1.tv_usec)) * 1e-6;

        i++;
    }
    fclose(fp);
```

TOKENISATION FOUR – refer postfix expression

FIVE – refer process scheduling – reading formatted txt : a b c

READING CSV WITH FSCANF – refer insertion sort

TAKING FILE INPUT OF FORM int, [….] – refer BST fileread

READING AND PRINTING

```
struct criminal{
    char *name;
    int age;
    int ID;
    double criminality;
};
typedef struct criminal criminal;
criminal *readCriminals()
{
    FILE *fptr = fopen("criminal_database.txt", "r");
    int n;
    fscanf(fptr, "%d", &n);
    criminal *arr = (criminal*) malloc(sizeof(criminal)*n);
    for (int i = 0; i < n; i++)
    {
        criminal c;
        c.name = (char*) malloc(sizeof(char)*20);
        fscanf(fptr, "%[^,],%d,%d", c.name, &c.age, &c.ID);
        c.criminality = 0.0;
        arr[i] = c;
    }
    fclose(fptr);
    return arr;
}
void mergeAux (criminal *L1, int s1, int e1, criminal *L2, int s2, int e2, criminal *L3, int s3, int
e3)
{
    int i,j,k;
    // Traverse both arrays
    i=s1; j=s2; k=s3;
    while (i <= e1 && j <= e2) {
        if (L1[i].criminality > L2[j].criminality)  L3[k++] = L1[i++];
        else L3[k++] = L2[j++];
    }
    while (i <= e1) L3[k++] = L1[i++];
    while (j <= e2) L3[k++] = L2[j++];
}
void merge(criminal *A, int s, int mid, int e)
{
    criminal *C = (criminal *)malloc(sizeof(criminal) * (e - s + 1));
    mergeAux(A, s, mid, A, mid + 1, e, C, 0, e-s);
    for(int i = 0; i < e - s + 1; i++)
    {
        A[s + i] = C[i];
    }
    free(C);
}
void mergeSort(criminal *A, int st, int en)
{
    if (en - st < 1)
        return;
    int mid = (st + en) / 2;   // mid is the floor of (st+en)/2
    mergeSort(A, st, mid);     // sort the first half
    mergeSort(A, mid + 1, en); // sort the second half
    merge(A, st, mid, en);     // merge the two halves
}
void findCriminality(criminal *criminals)
{
    FILE *fptr = fopen("crimes.txt", "r");
    int n;
    fscanf(fptr, "%d", &n);
    for (int i = 0; i < n; i++)
    {
        int year;
        int ID;
        char *crime;
        crime = (char*) malloc(sizeof(char)*20);
        fscanf(fptr, "%[^,],%d,%d", crime, &year, &ID);
        int index = (int) ID%100;
        if (strstr(crime, "ARSON") != NULL)
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*10;
            }
            else
            {
                criminals[index].criminality += 10;
            }
        }
        else if (strstr(crime, "ASSAULT"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*5;
            }
            else
            {
                criminals[index].criminality += 5;
            }
        }
        else if (strstr(crime, "BURGLARY"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*5;
            }
            else
            {
                criminals[index].criminality += 5;
            }
        }
        else if (strstr(crime, "CRIMINAL MISCHIEF"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*5;
            }
            else
            {
                criminals[index].criminality += 5;
            }
        }
        else if (strstr(crime, "GRAND LARCENY"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*10;
            }
            else
            {
                criminals[index].criminality += 10;
            }
        }
        else if (strstr(crime, "GRAND THEFT AUTO"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*10;
            }
            else
            {
                criminals[index].criminality += 10;
            }
        }
        else if (strstr(crime, "HOMICIDE"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*20;
            }
            else
            {
                criminals[index].criminality += 20;
```

```
            }
        }
        else if (strstr(crime, "BREAKING AND ENTERING"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*5;
            }
            else
            {
                criminals[index].criminality += 5;
            }
        }
        else if (strstr(crime, "ROBBERY"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*10;
            }
            else
            {
                criminals[index].criminality += 10;
            }
        }
    }
    fclose(fptr);
}
int main()
{
    criminal *criminals = readCriminals();
    findCriminality(criminals);
    FILE *fptr = fopen("criminal_database.txt", "r");
    int n;
    fscanf(fptr, "%d", &n);
    fclose(fptr);
    mergeSort(criminals, 0, n-1);
    for (int i = 0; i < n; i++)
    {
        printf("%s %d %d %lf", criminals[i].name, criminals[i].age, criminals[i].ID,
criminals[i].criminality);
    }
    FILE *fwriter = fopen("criminals_sorted.txt", "w");
    fprintf(fwriter, "%d", n);
    for (int i = 0; i < n; i++)
    {
        fprintf(fwriter, "%s %d %d %lf", criminals[i].name, criminals[i].age, criminals[i].ID,
criminals[i].criminality);
    }
    fclose(fwriter);
    free(criminals);

    return 0;
}
```

LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct node* NODE;
struct node{
    int ele;
    NODE next;
};
typedef struct linked_list* LIST;
struct linked_list{
    int count;
    NODE head;
};
LIST createNewList(){
    LIST myList;
    myList = (LIST) malloc(sizeof(struct linked_list));
    if (myList == NULL){
        printf("Unable to allocate memory.\n");
        exit(1);
    }
    myList->count = 0;
    myList->head = NULL;
    return myList;
}
NODE createNewNode(int value){
    NODE myNode;
    myNode = (NODE) malloc(sizeof(struct node));
    if (myNode == NULL){
        printf("Unable to allocate memory.\n");
        exit(1);
    }
    myNode->ele = value;
    myNode->next = NULL;
    return myNode;
}
void insertAfter(int searchEle, NODE n1, LIST l1){
    if(l1->head == NULL){
        l1->head = n1;
        n1->next = NULL;
        l1->count++;
    }
    else{
        NODE temp = l1->head;
        NODE prev = temp;
        while(temp != NULL){
            if(temp->ele == searchEle) break;
            prev = temp;
            temp = temp->next;
        }
        if (temp == NULL){
            printf("Element not found.\n");
            return;
        }
        else{
            if(temp->next == NULL){
                temp->next = n1;
                n1->next = NULL;
                l1->count++;
            }
            else{
                prev = temp;
                temp = temp->next;
                prev->next = n1;
                n1->next = temp;
                l1->count++;
            }
            return;
        }
    }
    return;
    }
void printList(LIST l1){
    NODE temp = l1->head;
    printf("[HEAD] ->");
    while(temp!=NULL){
        printf(" %d ->", temp->ele);
        temp = temp->next;
    }
    printf("[NULL]\n");
}
void deleteAt(int searchEle,LIST l1){
    if(l1->head == NULL){
        printf("Empty list\n");
        return;
    }
    else{
        NODE temp = l1->head;
        NODE prev = temp;
        if (temp->ele == searchEle){
            l1->head = temp->next;
            free(temp);
            l1->count-;
            return;
        }
        while(temp!=NULL){
            if (temp->ele == searchEle){
                prev->next = temp->next;
                free(temp);
                l1->count--;
                return;
            }
            prev = temp;
            temp = temp->next;
        }
        printf("Element not found.\n");
        return;
    }
}
void insertFirst(NODE value, LIST l1){
    value->next = l1->head;
    l1->head = value;
    l1->count++;
}
void deleteFirst(LIST l1){
    if(l1->head == NULL){
        printf("Empty list\n");
        return;
    }
    else{
        NODE temp = l1->head;
        NODE prev = temp;
        temp = temp->next;
        free(prev);
        l1->head = temp;
        l1->count--;
    }
}
int search(int searchEle, LIST l1){
    if(l1->head == NULL){
        printf("Empty list.\n");
    }
    else{
        int count = 0;
        NODE temp = l1->head;
        NODE prev = temp;
        while(temp != NULL){
            if(temp->ele == searchEle) return count;
            prev = temp;
            temp = temp->next;
            count++;
        }
        if (temp == NULL){
            return -1;
        }
    }
    return -1;
}
//task 8
void rotate(int k, LIST l1){
    NODE temp = l1->head;
    while (temp->next!= NULL){
        temp = temp->next;
    }
    temp->next = l1->head;
    NODE flag = l1->head;
    NODE prev = flag;
    while(k--){
        prev = flag;
        flag = flag->next;
    }
    l1->head = flag;
    prev->next = NULL;
}
bool hasCycle(LIST l1){
    if(l1->head == NULL){
        printf("Empty list.\n");
        return false;
    }
    NODE hare = l1->head;
    NODE tortoise = l1->head;
    while(hare != NULL && tortoise != NULL && hare->next != NULL){
        hare = hare->next->next;
        tortoise = tortoise->next;
        if (hare == tortoise) return true;
    }
    return false;
}
void circularLLCycleCheck(LIST l1){
    NODE temp = l1->head;
    while (temp->next!= NULL){
        temp = temp->next;
    }
    temp->next = l1->head;
    printf("Circular linked list created.\n");
    printf("Detecting cycle - should give true :\n");
    hasCycle(l1) ? printf("Cycle present\n") : printf("Cycle absent");
    printf("Disconnecting circular linked list.\n");
    temp->next = NULL;
    printf("Testing :\n");
    hasCycle(l1) ? printf("Cycle present\n") : printf("Cycle absent");
}
void reverse(LIST l1){
    // home exercise 5
    if (l1->head == NULL || l1->head->next == NULL){
        return;
    }
    NODE curr, prev, nex;
    curr = l1->head;
    prev = NULL;
    nex = NULL;
    while (curr != NULL){
        nex = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nex;
    }
    l1->head = prev;
    return;
}
int main(){
    LIST ll = createNewList();
    int a,b,c,d,f,k;
    int flag = -1;
    while(flag != 0){
        printf("\nWhat would you like to do ?\n");
        printf("1. Add a node and insert after node with a given value.\n");
        printf("2. Delete an element with its value.\n");
        printf("3. Insert at first node.\n");
        printf("4. Delete at first node.\n");
        printf("5. Display all elements.\n");
        printf("6. Search for an element with its value.\n");
        printf("7. Rotate the linked list by k steps.\n");
        printf("8. Check for a cycle.\n");
        printf("9. Demo test on a cyclic linked list.\n");
        printf("10. Reverse the linked list.\n");
        printf("0. Exit.\n");
        printf("Enter your input.\n");
        scanf(" %d", &flag);
        switch(flag){
            case 1:
                printf("Enter value of node and search value.\n");
                scanf("%d %d", &a, &b);
                insertAfter(b, createNewNode(a),ll);
                break;
            case 2:
                printf("Enter value of node to delete.\n");
                scanf("%d", &c);
                deleteAt(c,ll);
                break;
            case 3:
                printf("Enter value of node to add at first position.\n");
                scanf("%d", &d);
                insertFirst(createNewNode(d), ll);
                break;
            case 4:
                deleteFirst(ll);
                break;
            case 5:
                printList(ll);
                break;
            case 6:
                printf("Enter value of element to search for.\n");
                scanf("%d", &f);
                printf("Node is present at location = %d\n", search(f, ll));
                break;
            case 7:
                printf("Enter number of steps to rotate linked list by.\n");
                scanf("%d", &k);
                rotate(k, ll);
                break;
            case 8:
                hasCycle(ll) ? printf("Cycle present\n") : printf("Cycle absent");
                break;
            case 9:
                circularLLCycleCheck(ll);
                break;
            case 10:
                reverse(ll);
                break;
            case 0:
                break;
            default:
                printf("Invalid input !\n");
                break;
        }
```

```
    }
    return 0;
}
```

STACK ARRAY

```
#include "element.h"
#include "stack.h"
#include "heap_usage.h"
#include <stdlib.h>
#define STACK_SIZE 100
struct Stack{
    int top;
    Element data[STACK_SIZE];
};
Stack *newStack(){
    Stack *s = (Stack *)myalloc(sizeof(Stack));
    if(s != NULL)
         s->top = -1;
    return s;
}
bool push(Stack *s, Element e){
    if(s->top == STACK_SIZE - 1)
        return false;
    s->data[++(s->top)] = e;
    return true;
}
Element *top(Stack *s)
{
    if(s->top == -1)
        return NULL;
    return &(s->data[s->top]);
}
Element *pop(Stack *s){
    if(s->top == -1)
        return NULL;
    s->top--;
    return &(s->data[s->top+1]);
}
bool isEmpty(Stack *s){
    if(s->top == -1)
        return true;
    return false;
}
void freeStack(Stack *stack){
    myfree(stack);
}
```

STACK LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"
#include "linked_list.h"
#include "heap_usage.h"
#define STACK_MAX 100
struct Stack{
    LIST l;
};
Stack *newStack(){
    Stack *s = (Stack*) myalloc(sizeof(Stack));
    s->l = createNewList();
    return s;
}
bool push(Stack *stack, Element element){
    if (stack->l->count > STACK_MAX) return false;
    NODE n = createNewNode(element);
    insertNodeIntoList(n,stack->l);
    return true;
}
Element *top(Stack *stack){
    if (stack->l->head == NULL){
        return NULL;
    }
    return stack->l->head;
}
Element *pop(Stack *stack){
    Element *ele = stack->l->head;
    removeFirstNode(stack->l);
    return ele;
}
bool isEmpty(Stack *stack){
    if (stack->l->head == NULL){
        return true;
    }
    return false;
}
void freeStack(Stack *stack){
    destroyList(stack->l);
    myfree(stack);
}
```

POSTFIX EXPRESSION EVALUATION

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "stack.h"
int main() {
    printf("Enter the string to be evaluated in postfix notation:\n");
    Stack *stack = newStack();
    char input[100]; // Assuming a maximum input length of 100 characters
    char *token;
    fgets(input, sizeof(input), stdin);
    token = strtok(input, " "); // Tokenize input string by space
    while (token != NULL) {
        if (isdigit(*token)) {
            push(stack, atoi(token)); // Convert token to integer and push onto stack
        } else {
            int b = *pop(stack);
            int a = *pop(stack);
            int ans;
            switch (*token) {
                case '+':
                    ans = a + b;
                    break;
                case '-':
                    ans = a - b;
                    break;
                case '*':
                    ans = a * b;
                    break;
                case '/':
                    ans = a / b;
                    break;
                default:
                    printf("Invalid operator: %c\n", *token);
                    freeStack(stack);
                    return 1;
            }
            push(stack, ans);
        }
        token = strtok(NULL, " "); // Get next token
    }
    printf("The value of the expression is %d\n", *pop(stack));
    freeStack(stack);
    return 0;
}
```

COMPUTE THE SPAN

```
int main()
{
    int inputs[] = {6, 3, 4, 5, 2};
    int spans[5];
    computeSpans(inputs, spans, 5);
    for (int i = 0; i < 5; i++)
    {
        printf("%d ", spans[i]);
    }
    printf("\n");
    int inputs2[] = {100, 80, 60, 70, 60, 75, 85};
    int spans2[7];
    computeSpans(inputs2, spans2, 7);
    for (int i = 0; i < 7; i++)
    {
        printf("%d ", spans2[i]);
    }
    printf("\n");
    return 0;
}
void computeSpans(int *inputs, int *spans, int n)
{
    Stack *index = newStack();
    for (int i = 0; i < n; i++){
        while (!isEmpty(index) && inputs[*top(index)] <= inputs[i]){
            pop(index);
        }
        if (isEmpty(index)) spans[i] = i+1;
        else spans[i] = i - *top(index);
        push(index, i);
    }
}
```

QUEUE ARRAY

```
#include "element.h"
#include "queue.h"
#include "heap_usage.h"
#include <stdlib.h>
#define QUEUE_SIZE 100
struct Queue{
    int size;
    Element data[QUEUE_SIZE];
};
Queue *createQueue(){
    Queue *q = (Queue *)myalloc(sizeof(Queue));
    if(q != NULL)
         q->size = 0;
    return q;
}
bool enqueue(Queue *q, Element e){
    if(q->size == QUEUE_SIZE)
        return false;
    q->data[q->size++] = e;
    return true;
}
bool dequeue(Queue *q){

    if(q->size == 0)
        return false;
    for (int i = 0; i < q->size-1; i++){
        q->data[i] =  q->data[i+1];
    }
    q->size--;
    return true;
}
int size(Queue *q)
{
   return q->size;
}
Element *front(Queue *q){
    if(q->size == 0)
        return NULL;
    return &(q->data[0]);
}
bool isEmpty(Queue *q){
    if(q->size == 0)
        return true;
    return false;
}
void destroyQueue(Queue *queue){
    myfree(queue);
}
```

QUEUE LINKED LIST

```
#define QUEUE_MAX 100
struct Queue{
    LIST l;
};
Queue *createQueue(){
    Queue *q = (Queue*) myalloc(sizeof(Queue));
    q->l = createNewList();
    return q;
}
bool enqueue(Queue *queue, Element element){
    if (queue->l->count > QUEUE_MAX) return false;
    NODE n = createNewNode(element);
    insertNodeAtEnd(n,queue->l);
    return true;
}
Element *front(Queue *queue){
    if (queue->l->head == NULL){
        return NULL;
    }
    return &queue->l->head->data;
}
bool dequeue(Queue *queue){
    if (queue->l->count == 0) return false;
    removeFirstNode(queue->l);
    return true;
}
bool isEmpty(Queue *queue){
    if (queue->l->head == NULL){
        return true;
    }
    return false;
}
int size(Queue *queue){
    return queue->l->count;
}
void destroyQueue(Queue *queue){
    destroyList(queue->l);
    myfree(queue);
}
```

PROCESS SCHEDULING WITH QUEUE

```
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"
typedef struct process
{
    int pid;
    int arrival_time;
    int burst_time;
} Process;
Process pabtoe(int p, int a, int b);
Element itoe(int i);
int main()
{
    FILE *fp = fopen("fcfs_input.txt", "r");
    if (fp == NULL)
    {
        printf("Error opening file\n");
        exit(1);
    }
    int n;
    fscanf(fp, "%d\n", &n);
    int p, a, b;
    Process *arr = (Process *)malloc(sizeof(Process) * n);
    Queue *q = createQueue();
    for (int i = 0; i < n; ++i)
    {
        fscanf(fp, "%d %d %d\n", &p, &a, &b);
        arr[i] = pabtoe(p, a, b);
    }
    int curr = -1;
    enqueue(q, itoe(0));
    for (int i = 0; !isEmpty(q) || curr != -1; ++i)
    {
        if (curr != -1)
        {
            --(arr[curr].burst_time);
            if (arr[curr].burst_time == 0)
            {
                printf("Process %d finished at time %d\n", arr[curr].pid, i);
                curr = -1;
            }
        }
        if (curr == -1)
        {
            curr = front(q)->j;
            printf("Process %d started at time %d\n", arr[curr].pid, i);
            dequeue(q);
        }
        for (int j = 1 + curr; j < n; ++j)
        {
            if (arr[j].arrival_time == i)
            {
                enqueue(q, itoe(j));
                break;
            }
        }
    }
}
Process pabtoe(int p, int a, int b)
{
    Process e;
    e.pid = p;
    e.arrival_time = a;
    e.burst_time = b;
    return e;
}
Element itoe(int i)
{
    Element e;
    e.j = i;
    return e;
}
```

INSERTION SORT

```
struct person
{
    int id;
    char *name;
    int age;
    int height;
    int weight;
};
typedef struct person person;
void insertInOrder(person v, person *A, int last);
void insertionSort(person *A, int n){
    for (int j = 1; j < n; j++)
    {
        insertInOrder(*(A + j), A, j);
    }
}
void insertInOrder(person v, person *A, int last)
{
    int j = last - 1;
    while (j >= 0 && v.height < (A + j)->height)
    {
        A[j + 1] = A[j];
        j--;
    }
    *(A + j + 1) = v;
}
int main(int argc, char **argv)
{
    char *filename = (char *) malloc(sizeof(char)*strlen(argv[1])); // reading datX.csv
    filename = argv[1];
    char substring[strlen(filename) - 3];
    strncpy(substring, argv[1] + 3, strlen(argv[1]) - 3);
    int n = atoi(substring);
    person *A = (person *)malloc(n * sizeof(person));
    FILE *fp = fopen(filename, "r");
    if (!fp)
    {
        printf("Can't open file\n");
    }
    else
    {
        for (int k = 0; k < n; k++)
        {
            (A + k)->name = (char *)malloc(100 * sizeof(char));
            fscanf(fp, "%d,%[^,],%d,%d,%d", &(A + k)->id, (A + k)->name, &(A + k)->age, &(A + k)->height, &(A + k)->weight);
        }
        fclose(fp);
    }
    FILE *fptr;
    fptr = fopen("insertionSortBenchmarks.txt", "a");
    if (fptr == NULL)
    {
        printf("Error opening the file.");
        exit(1);
    }
    fprintf(fptr, "%d,%f\n", n, time_taken);
    fclose(fptr);
    free(A);
    return 0;
}
```

INSERTION SORT RECURSIVE

```
void insertionSortRecursive(int arr[], int n)
{
    if (n <= 1)
        return;
    insertionSortRecursive(arr, n - 1);
    int last = arr[n - 1];
    int j = n - 2;
    while (j >= 0 && arr[j] > last) {
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = last;
}
```

INSERTION SORT ITERATIVE

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

MERGE SORT

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "intMerge.h"
#include "intMergeAux.h"
void merge(int A[], int s, int mid, int e)
{
    int *C = (int *)myalloc(sizeof(int) * (e - s + 1));
    mergeAux(A, s, mid, A, mid + 1, e, C, 0, e-s);
    for(int i = 0; i < e - s + 1; i++)
    {
        A[s + i] = C[i];
    }
    // myfree(C);
}
void mergeSort(int A[], int st, int en)
{
    if (en - st < 1)
        return;
    int mid = (st + en) / 2;   // mid is the floor of (st+en)/2
    mergeSort(A, st, mid);     // sort the first half
    mergeSort(A, mid + 1, en); // sort the second half
    merge(A, st, mid, en);     // merge the two halves
}
int main(){
    FILE* fptr = fopen("marks.txt","r");
    char *line = (char*) malloc(sizeof(char)*10);
    // int A[1000];
    int *A = (int*) myalloc(sizeof(int)*1000);
    int i = 0;
    while (fgets(line, 10, fptr) != NULL && i <1000){
        A[i++] = atoi(line);
    }
    mergeSort(A, 0, 999);
    myfree(A);
    return 0;
}
void mergeAux (int L1[], int s1, int e1, int L2[], int s2, int e2, int L3[], int s3, int e3) //
ITERATIVE
{
    int i,j,k;
    // Traverse both arrays
    i=s1; j=s2; k=s3;
    while (i <= e1 && j <= e2) {
        // Check if current element of first array is smaller
        // than current element of second array
        // If yes, store first array element and increment first
        // array index. Otherwise do same with second array
        if (L1[i] < L2[j])  L3[k++] = L1[i++];
        else L3[k++] = L2[j++];
    }
    // Store remaining elements of first array
    while (i <= e1) L3[k++] = L1[i++];
    // Store remaining elements of second array
    while (j <= e2) L3[k++] = L2[j++];
}
void mergeAux(int L1[], int s1, int e1, int L2[], int s2, int e2, int L3[], int s3, int e3) //
RECURSIVE
{
    if(s3 > e3) return;
    if (s2 > e2)
    {
        L3[s3] = L1[s1];
        mergeAux(L1, s1 + 1, e1, L2, s2, e2, L3, s3 + 1, e3);
    }
    else if (s1 > e1)
    {
        L3[s3] = L2[s2];
        mergeAux(L1, s1, e1, L2, s2 + 1, e2, L3, s3 + 1, e3);
    }
    else if (L1[s1] >= L2[s2])
    {
        L3[s3] = L2[s2];
        mergeAux(L1, s1, e1, L2, s2 + 1, e2, L3, s3 + 1, e3);
    }
    else if (L1[s1] < L2[s2])
    {
        L3[s3] = L1[s1];
        mergeAux(L1, s1 + 1, e1, L2, s2, e2, L3, s3 + 1, e3);
    }
    return;
}
```

MERGE BY INSERT

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "intMerge.h"
void merge(int A[], int s, int mid, int e)
{
    for (int i = mid + 1; i < e + 1; i++){
        int key = A[i];
        int j = i - 1;
        while (A[j] > key && j > s-1){
            A[j+1] = A[j];
            j--;
        }
        A[j+1] = key;
    }
}
void mergeSort(int A[], int st, int en)
{
    if (en - st < 1)
        return;
    int mid = (st + en) / 2;   // mid is the floor of (st+en)/2
    mergeSort(A, st, mid);      // sort the first half
    mergeSort(A, mid + 1, en); // sort the second half
    merge(A, st, mid, en);      // merge the two halves
}
int main(){
    FILE* fptr = fopen("marks.txt","r");
    char *line = (char*) malloc(sizeof(char)*10);
    // int A[1000];
    int *A = (int*) myalloc(sizeof(int)*1000);
    int i = 0;
    while (fgets(line, 10, fptr) != NULL && i <1000){
        A[i++] = atoi(line);
    }
    return 0;
}
```

MERGE BY ITERATION

```
void mergeSort(int arr[], int n)
{
  int curr_size;  // For current size of subarrays to be merged // curr_size varies from 1 to n/2
  int left_start; // For picking starting index of left subarray to be merged
  // Merge subarrays in bottom up manner.  First merge subarrays of size 1 to create sorted
subarrays of size 2, then merge subarrays of size 2 to create sorted subarrays of size 4, and
so on.
  for (curr_size=1; curr_size<=n-1; curr_size = 2*curr_size)
  {
     // Pick starting point of different subarrays of current size
     for (left_start=0; left_start<n-1; left_start += 2*curr_size)
     {
        // Find ending point of left subarray. mid+1 is starting
        // point of right
        int mid = min(left_start + curr_size - 1, n-1);
        int right_end = min(left_start + 2*curr_size - 1, n-1);
        // Merge Subarrays arr[left_start...mid] & arr[mid+1...right_end]
        merge(arr, left_start, mid, right_end);
     }
  }
}
/* Function to merge the two haves arr[l..m] and arr[m+1..r] of array arr[] */
void merge(int arr[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 =  r - m;
   /* create temp arrays */
   int L[n1], R[n2];
   /* Copy data to temp arrays L[] and R[] */
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1+ j];
   /* Merge the temp arrays back into arr[l..r]*/
   i = 0;
   j = 0;
   k = l;
   while (i < n1 && j < n2)
   {
      if (L[i] <= R[j])
      {
         arr[k] = L[i];
         i++;
      }
      else
      {
         arr[k] = R[j];
         j++;
      }
      k++;
   }
   /* Copy the remaining elements of L[], if there are any */
   while (i < n1)
   {
      arr[k] = L[i];
      i++;
```

```
        k++;
    }
    /* Copy the remaining elements of R[], if there are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

FINDING CRIMNALITY

```
struct criminal{
    char *name;
    int age;
    int ID;
    double criminality;
};
typedef struct criminal criminal;
criminal *readCriminals()
{
    FILE *fptr = fopen("criminal_database.txt", "r");
    int n;
    fscanf(fptr, "%d", &n);
    criminal *arr = (criminal*) malloc(sizeof(criminal)*n);
    for (int i = 0; i < n; i++)
    {
        criminal c;
        c.name = (char*) malloc(sizeof(char)*20);
        fscanf(fptr, "%[^,],%d,%d", c.name, &c.age, &c.ID);
        c.criminality = 0.0;
        arr[i] = c;
    }
    fclose(fptr);
    return arr;
}
void mergeAux (criminal *L1, int s1, int e1, criminal *L2, int s2, int e2, criminal *L3, int s3, int
e3)
{
    int i,j,k;
    // Traverse both arrays
    i=s1; j=s2; k=s3;
    while (i <= e1 && j <= e2) {
        if (L1[i].criminality > L2[j].criminality)  L3[k++] = L1[i++];
        else L3[k++] = L2[j++];
    }
    while (i <= e1) L3[k++] = L1[i++];
    while (j <= e2) L3[k++] = L2[j++];
}
void merge(criminal *A, int s, int mid, int e)
{
    criminal *C = (criminal *)malloc(sizeof(criminal) * (e - s + 1));
    mergeAux(A, s, mid, A, mid + 1, e, C, 0, e-s);
    for(int i = 0; i < e - s + 1; i++)
    {
        A[s + i] = C[i];
    }
    free(C);

}
void mergeSort(criminal *A, int st, int en)
{
    if (en - st < 1)
        return;
    int mid = (st + en) / 2;   // mid is the floor of (st+en)/2
    mergeSort(A, st, mid);     // sort the first half
    mergeSort(A, mid + 1, en); // sort the second half
    merge(A, st, mid, en);     // merge the two halves
}
void findCriminality(criminal *criminals)
{
    FILE *fptr = fopen("crimes.txt", "r");
    int n;
    fscanf(fptr, "%d", &n);
    for (int i = 0; i < n; i++)
    {
        int year;
        int ID;
        char *crime;
        crime = (char*) malloc(sizeof(char)*20);
        fscanf(fptr, "%[^,],%d,%d", crime, &year, &ID);
        int index = (int) ID%100;
        if (strstr(crime, "ARSON") != NULL)
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*10;
            }
            else
            {
                criminals[index].criminality += 10;
            }
        }
        else if (strstr(crime, "ASSAULT"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*5;
            }
            else
            {
                criminals[index].criminality += 5;
            }
        }
        else if (strstr(crime, "BURGLARY"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*5;
            }
            else
            {
                criminals[index].criminality += 5;
            }
        }
        else if (strstr(crime, "CRIMINAL MISCHIEF"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*5;
            }
            else
            {
                criminals[index].criminality += 5;
            }
        }
        else if (strstr(crime, "GRAND LARCENY"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*10;
            }
            else
            {
                criminals[index].criminality += 10;
            }
        }
        else if (strstr(crime, "GRAND THEFT AUTO"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*10;
            }
            else
            {
                criminals[index].criminality += 10;
            }
        }
        else if (strstr(crime, "HOMICIDE"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*20;
            }
            else
            {
                criminals[index].criminality += 20;
            }
        }
        else if (strstr(crime, "BREAKING AND ENTERING"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*5;
            }
```

```
            else
            {
                criminals[index].criminality += 5;
            }
        }
        else if (strstr(crime, "ROBBERY"))
        {
            if (year - (2023 - criminals[i].age) <= 18)
            {
                criminals[index].criminality += 0.5*10;
            }
            else
            {
                criminals[index].criminality += 10;
            }
        }
    }
    fclose(fptr);
}
int main()
{
    criminal *criminals = readCriminals();
    findCriminality(criminals);
    FILE *fptr = fopen("criminal_database.txt", "r");
    int n;
    fscanf(fptr, "%d", &n);
    fclose(fptr);
    mergeSort(criminals, 0, n-1);
    for (int i = 0; i < n; i++)
    {
        printf("%s %d %d %lf", criminals[i].name, criminals[i].age, criminals[i].ID,
criminals[i].criminality);
    }
    FILE *fwriter = fopen("criminals_sorted.txt", "w");
    fprintf(fwriter, "%d", n);
    for (int i = 0; i < n; i++)
    {
        fprintf(fwriter, "%s %d %d %lf", criminals[i].name, criminals[i].age, criminals[i].ID,
criminals[i].criminality);
    }
    fclose(fwriter);
    free(criminals);

    return 0;
}
```

INPLACE MERGE SORT

```
// C++ program in-place Merge Sort
#include <stdio.h>
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
// Inplace Implementation
void merge(int arr[], int start, int mid, int end)
{
    int start2 = mid + 1;
    // If the direct merge is already sorted
    if (arr[mid] <= arr[start2]) {
        return;
    }
    // Two pointers to maintain start of both arrays to merge
    while (start <= mid && start2 <= end) {
        // If element 1 is in right place
        if (arr[start] <= arr[start2]) {
            start++;
        }
        else {
            int value = arr[start2];
            int index = start2;

            // Shift all the elements between element 1
            // element 2, right by 1.
            while (index != start) {
                arr[index] = arr[index - 1];
                index--;
            }
            arr[start] = value;

            // Update all the pointers
            start++;
            mid++;
            start2++;
        }
    }
}
/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        // Same as (l + r) / 2, but avoids overflow
        // for large l and r
        int m = l + (r - l) / 2;
        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
/* Driver program to test above functions */
int main()
```

```
{
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        int arr_size = sizeof(arr) / sizeof(arr[0]);
        mergeSort(arr, 0, arr_size - 1);
        printArray(arr, arr_size);
        return 0;
}
```

EXTERNAL MERGE SORT

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define NAMESIZE 30
#define BUFFERSIZE 1000000
typedef struct {
    int id;
    char name[30];
    int age;
    int height;
    int weight;
} Person;
Person* readPerson(FILE *f){
    Person *p = (Person*) malloc(sizeof(Person));
    int status = fscanf(f, "%d,%[^,],%d,%d,%d", &p->id, p->name, &p->age, &p->height, &p-
>weight);
    if (status == 5){
        return p;
    }
    else{
        free(p);
        return NULL;
    }
}
void writePerson(FILE *f, Person *p){
    fprintf(f, "%d,%s,%d,%d,%d\n", p->id, p->name, p->age, p->height, p->weight);
}
int min(int x, int y) { return (x<y)? x :y;}
void merge(Person arr[], int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    Person* L = (Person*) malloc(n1 * sizeof(Person));
    Person* M = (Person*) malloc(n2 * sizeof(Person));
    for (int i = 0; i < n1; i++)
        L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
        M[j] = arr[q + 1 + j];
    int i, j, k;
    i = 0;
    j = 0;
    k = p;
    while (i < n1 && j < n2) {
        if (L[i].height <= M[j].height)
            arr[k++] = L[i++];
        else
            arr[k++] = M[j++];
    }
    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = M[j++];
    free(L);
    free(M);
}
void itermergeSort(Person arr[], int n)
{
  for (int curr_size=1; curr_size<=n-1; curr_size = 2*curr_size)
  {
      for (int l=0; l<n-1; l += 2*curr_size)
      {
          int mid = min(l + curr_size - 1, n-1);
          int r = min(l + 2*curr_size - 1, n-1);
          merge(arr, l, mid, r);
      }
  }
}
char* saveBuffer(Person* buffer, int buffer_size, int file_index){
    char* filename = (char*) malloc(100 * sizeof(char));
    sprintf(filename, "temp%d.csv", file_index);
    FILE* fpt = fopen(filename, "w");
    for (int i = 0; i < buffer_size; i++){
        fprintf(fpt, "%d,%s,%d,%d,%d\n", buffer[i].id, buffer[i].name, buffer[i].age,
buffer[i].height, buffer[i].weight);
    }
    fclose(fpt);
    return filename;
}
char * mergeFiles(char *file1, char *file2, int file_index){
    FILE* fpt1 = fopen(file1, "r");
    FILE* fpt2 = fopen(file2, "r");
    char* filename = (char*) malloc(100 * sizeof(char));
    sprintf(filename, "temp%d.csv", file_index);
    FILE* fpt = fopen(filename, "w");
    Person* p1 = readPerson(fpt1);
    Person* p2 = readPerson(fpt2);
    while (p1 != NULL && p2 != NULL){
        if (p1->height <= p2->height){
            writePerson(fpt, p1);
            free(p1);
            p1 = readPerson(fpt1);
        }
        else{
            writePerson(fpt, p2);
            free(p2);
            p2 = readPerson(fpt2);
        }
    }
    while (p1 != NULL){
        writePerson(fpt, p1);
        free(p1);
        p1 = readPerson(fpt1);
    }

    while (p2 != NULL){
        writePerson(fpt, p2);
        free(p2);
        p2 = readPerson(fpt2);
    }
    fclose(fpt1);
    fclose(fpt2);
    fclose(fpt);
    return filename;
}
int fileIndex = 1;
int main(int argc, char const *argv[])
{
    char mainfile[100];
    strcpy(mainfile, argv[1]);
    FILE* fpt = fopen(mainfile, "r");
    int entries;
    sscanf(mainfile, "dat%d.csv", &entries);
    Person* BUFFER = (Person*) malloc(BUFFERSIZE * sizeof(Person));
    int buffer_size = 0;
    char** tempFiles = (char**) malloc((entries/BUFFERSIZE + 1) * sizeof(char*));
    int tempFiles_size = 0;
    while (1)
    {
        Person *p = readPerson(fpt);
        if (p == NULL){
            break;
        }
        BUFFER[buffer_size++] = *p;
        free(p);
        if (buffer_size == BUFFERSIZE){
            itermergeSort(BUFFER, buffer_size);
            tempFiles[tempFiles_size++] = saveBuffer(BUFFER, buffer_size, fileIndex++);
            buffer_size = 0;
        }
    }
    if (buffer_size > 0){
        itermergeSort(BUFFER, buffer_size);
        tempFiles[tempFiles_size++] = saveBuffer(BUFFER, buffer_size, fileIndex++);
    }
    fclose(fpt);
    free(BUFFER);
    char** tempFilesTEMP = (char**) malloc((entries/BUFFERSIZE + 1) * sizeof(char*));
    int tempFilesTEMP_size = 0;
    while (tempFiles_size != 1)
    {
        for (int i = 0; i + 1 < tempFiles_size; i+=2)
        {
            tempFilesTEMP[tempFilesTEMP_size++] = mergeFiles(tempFiles[i], tempFiles[i+1],
fileIndex++);
            remove(tempFiles[i]);
            remove(tempFiles[i+1]);
            free(tempFiles[i]);
            free(tempFiles[i+1]);
        }
        if (tempFiles_size % 2 == 1){
            tempFilesTEMP[tempFilesTEMP_size++] = tempFiles[tempFiles_size-1];
        }
        tempFiles_size = tempFilesTEMP_size;
        tempFilesTEMP_size = 0;
        char** temp = tempFiles;
        tempFiles = tempFilesTEMP;
        tempFilesTEMP = temp;
    }
    rename(tempFiles[0], "sorted.csv");
    free(tempFiles[0]);
    free(tempFiles);
    free(tempFilesTEMP);
    return 0;
}
```

NOBLE INTEGERS BY SORTING

```
int n;
    printf("Enter the size of the array.\n");
    scanf("%d", &n);
    int *A = (int *)malloc(n * sizeof(int));
    printf("Enter %d elements : \n", n);
    for (int k = 0; k < n; k++)
    {
       scanf("%d", A + k);
    }
    mergeSort(A, 0, n-1);
    int count;
    for (int i = 0; i < n; i++){
        if (A[i] == A[i+1]) continue;
        if (n - i - 1 == A[i]){
            printf("%d\n", A[i]);
            free(A);
            return 0;
       }
    }
    printf("Not found\n");
    free(A);
    return 0;
```

NOBLE INTEGERS BEST METHOD

```
int nobleInteger(int arr[], int n)
{
    // Declare a countArr which keeps count of all elements greater than or equal to arr[i].
Initialize it with zero.
    int countArr[n + 1];
    for (int i = 0; i < n+1; i++) countArr[i] = 0;
    // Iterating through the given array
    for (int i = 0; i < n; i++) {
        // If current element is less than zero, it cannot be a solution so we skip it.
        if (arr[i] < 0) continue;
        // If current element is >= size of input array, if will be greater than all elements which
can be considered as our solution, as it cannot be
        // greater than size of array.
        else if (arr[i] >= n) countArr[n]++;
        // Else we increase the count of elements >= our current array in countArr
        else countArr[arr[i]]++;
    }
    // Initially, countArr[n] is count of elements greater than all possible solutions
    int totalGreater = countArr[n];
    // Iterating through countArr
    for (int i = n - 1; i >= 0; i--) {
        // If totalGreater = current index, means we found arr[i] for which count of elements >=
arr[i] is equal to arr[i]
        if (totalGreater == i && countArr[i] > 0) return i;
        // If at any point count of elements greater than arr[i] becomes more than current index,
then it means we can no longer have a solution
        else if (totalGreater > i) return -1;
        // Adding count of elements >= arr[i] to totalGreater.
        totalGreater += countArr[i];
    }
    return -1;
}
int main()
{
    // int arr[] = { 10, 3, 20, 40, 2 };
    int arr[] = { 1, 3, 3, 4, 5};
    int res = nobleInteger(arr, 5);
    if (res != -1)
        cout << "The noble integer is " << res;
    else
        cout << "No Noble Integer Found";
    return 0;
}
```

TRIPLET SUMS EQUAL TO 0

```
int main()
{
    int n;
    printf("Enter the size of the array.\n");
    scanf("%d", &n);

    int *A = (int *)malloc(n * sizeof(int));
    printf("Enter %d elements : \n", n);
    for (int k = 0; k < n; k++)
    {
      scanf("%d", A + k);
    }
    mergeSort(A, 0, n-1);
    for (int i = 0; i < n - 2; i++) {
        // Skip duplicates
        if (i > 0 && A[i] == A[i - 1])
            continue;
        int left = i + 1;
        int right = n - 1;
        // Two-pointer technique
        while (left < right) {
            int total = A[i] + A[left] + A[right];
            if (total == 0) {
                // Print the triplet
                printf("(%d, %d, %d)\n", A[i], A[left], A[right]);
                // Skip duplicates
                do {
                    left++;
                } while (left < right && A[left] == A[left - 1]);
                // Skip duplicates
                do {
                    right--;
                } while (left < right && A[right] == A[right + 1]);
            } else if (total < 0) {
                left++;
            } else {
                right--;
            }
        }
    }
    free(A);
    return 0;
}
```

BINARY SEARCH TREE

```
void traverse_pre_order(Node *node)
{
    if (node == NULL)
    {
        return;
    }
    printf("%d ", node->value);
    traverse_pre_order(node->left);
    traverse_pre_order(node->right);
}

void traverse_post_order(Node *node)
{
    if (node == NULL)
    {
        return;
    }
    traverse_post_order(node->left);
    traverse_post_order(node->right);
    printf("%d ", node->value);
}

void traverse_in_order_alternate(Node *node)
{
    if (node == NULL)
    {
        printf("NULL ");
        return;
    }
    traverse_in_order(node->left);
    printf("%d ", node->value);
    traverse_in_order(node->right);
}
```

```
BST *constructBST(int *arr, int n)
{
    BST *bst = new_bst();
    for (int i = 0; i < n; i++)
    {
        insert(bst, arr[i]);
    }
    return bst;
}
int maxValue(struct node* node)
{
    if (node == NULL)
    {
        return 0;
    }
    int leftMax = maxValue(node->left);
    int rightMax = maxValue(node->right);
    int value = 0;
    if (leftMax > rightMax)
    {
        value = leftMax;
    }
    else
    {
        value = rightMax;
    }
    if (value < node->value)
    {
        value = node->value;
    }
    return value;
}
int minValue(struct node* node)
{
    if (node == NULL)
    {
        return 1000000000;
    }
    int leftMax = minValue(node->left);
    int rightMax = minValue(node->right);
    int value = 0;
    if (leftMax < rightMax)
    {
        value = leftMax;
    }
    else
    {
        value = rightMax;
    }
    if (value > node->value)
    {
        value = node->value;
    }
    return value;
}
```

```
int BSTCheck(Node *node)
{
    if (node == NULL) return 1;
    if (node->left != NULL && maxValue(node->left) > node->value) return 0;
    if (node->right != NULL && minValue(node->right) < node->value) return 0;
    if (!BSTCheck(node->left) || !BSTCheck(node->right)) return 0;
    return 1;
}
int BSTCheckIterative(Node *node)
{   // Morris Traversal
    Node *current = node;
    Node *prev = NULL;
    while (current != NULL)
    {
        if (current->left == NULL)
        {
            // case 1 : No left child, process current node
            if (prev != NULL && prev->value > current->value)
            {
                return 0;
            }
            prev = current;
            current = current->right;
        }
        else
        {// case 2 : left child exists, find the predecessor
            Node *pred =  current->left;
            while (pred->right != NULL && pred->right != current)
                pred = pred->right;
            if (pred->right == NULL)
            {
                pred->right = current;
                current = current->left;
            }
            else
            {
                // remove threaded link
                // if the threaded link has been established it mean we
                // have visited the left subtree and need to process the current node
                pred->right = NULL;
                // process the current node
                if (prev != NULL && prev->value > current->value)
                    return 0;
                prev = current;
                current = current->right;
            }
        }
    }
    return 1;
}
int height(Node *node)
{
    if (node == NULL)
        return -1;
    else
    {
        int lh = height(node->left);
        int rh = height(node->right);
        if (lh > rh) return lh + 1;
        else return rh + 1;
    }
}
Node *removeHalfNode(Node *node)
{
    if (node == NULL) return NULL;
    node->left = removeHalfNode(node->left);
    node->right = removeHalfNode(node->right);
    if (node->left==NULL && node->right==NULL)
        return node;
    if (node->left == NULL)
    {
        Node *new_node = node->right;
        free(node); // To avoid memory leak
        return new_node;
    }
    if (node->right == NULL)
    {
        Node *new_node = node->left;
        free(node); // To avoid memory leak
        return new_node;
    }
    return node;
}
void traverse_level_order(Node *node)
{
    Queue *q = createQueue();
    Node *current = node;
    while (current != NULL)
    {
        printf(" %d ", current->value);
        if (current->left != NULL)
            enqueue(q, current->left);
        if (current->right != NULL)
            enqueue(q, current->right);
        current = front(q);
        dequeue(q);
    }
}
void traverse_level_order_reverse(Node *node)
{
    Queue *q = createQueue();
    Stack *s = newStack();
    Node *current = node;
    push(s, node->value);
    while (current != NULL)
    {
        if (current->right != NULL)
        {
            enqueue(q, current->right);
            push(s, current->right->value);
        }
        if (current->left != NULL)
        {
            enqueue(q, current->left);
            push(s, current->left->value);
        }
        current = front(q);
        dequeue(q);
    }
    while (!isEmptyStack(s))
    {
        printf(" %d ", *top(s));
        pop(s);
    }
}
void flattenHelper(Node *node, LIST ll)
{
    if (node == NULL)
    {
        return;
    }
    insertFirst(createNewNode(node->value), ll);
    flattenHelper(node->left, ll);
    flattenHelper(node->right, ll);
}
LIST flatten(BST *bst)
{
    LIST flat = createNewList();
    flattenHelper(bst->root, flat);
    reverse(flat);
    return flat;
}
void flattenInPlace(Node* root)
{
    // using Morris traversal
    // traverse till root is not NULL
    while (root) {
        // if root->left is not NULL
        if (root->left != NULL) {
            // set curr node as root->left;
            Node* curr = root->left;
            // traverse to the extreme right of curr
            while (curr->right) {
                curr = curr->right;
            }
            // join curr->right to root->right
            curr->right = root->right;
            // put root->left to root->right
            root->right = root->left;
            // make root->left as NULL
            root->left = NULL;
        }
        // now go to the right of the root
        root = root->right;
    }
```

```
}
void kthSmallestHelper(Node *node, Queue *q)
{
    if (node == NULL)
    {
        return;
    }
    kthSmallestHelper(node->left, q);
    enqueue(q, node);
    kthSmallestHelper(node->right, q);
}

Node *kthSmallest(BST *bst, int k)
{
    Queue *q = createQueue();
    kthSmallestHelper(bst->root, q);
    Node *answer = malloc(sizeof(Node));
    printf(" The %dth smallest element is = ", k);
    while (k--)
    {
        answer = front(q);
        dequeue(q);
    }
    printf("%d\n", answer->value);
    return answer;
}
void traverse_for_ID(Node *node, int ID, Node *answer)
{
    if (node == NULL)
    {
        return;
    }
    traverse_for_ID(node->left, ID, answer);
    if (node && node->value.id == ID)
    {
        answer->value = node->value;
        answer->left = node->left;
        answer->right = node->right;
    }
    traverse_for_ID(node->right, ID, answer);
}
// Queue *search_queue(BST *bst, Person key)
// {
//     Node *current = bst->root;
//     Queue *q = createQueue();
//     while (current != NULL)
//     {
//         if (key.height == current->value.height && key.id == current->value.id)
//         {
//             return q;
//         }
//         else if (key.height < current->value.height)
//         {
//             enqueue(q, current);
//             current = current->left;
//         }
//         else
//         {
//             enqueue(q, current);
//             current = current->right;
//         }
//     }
//     return NULL;
// }
// Node *LCA(BST *bst, int ID1, int ID2)
// {
//     Node *one = malloc(sizeof(Node));
//     traverse_for_ID(bst->root, ID1, one);
//     Node *two = malloc(sizeof(Node));
//     traverse_for_ID(bst->root, ID2, two);
//     if (ID1 == ID2) return one;
//     if (one->value.id != ID1 || two->value.id != ID2)
//     {
//         printf("ID not found !\n");
//         return NULL;
//     }
//     else
//     {
//         Queue *q1 = search_queue(bst, one->value);
//         Queue *q2 = search_queue(bst, two->value);
//         Node *ancestor = NULL;
//         while (front(q1) && front(q2) && front(q1)->value.height == front(q2)->value.height)
//         {
//             ancestor = front(q1);
//             dequeue(q1);
//             dequeue(q2);
//         }
//         return ancestor;
//     }
//     return NULL;
// }
Node *LCABetter(BST *bst, int ID1, int ID2)
{
    Node *one = malloc(sizeof(Node));
    traverse_for_ID(bst->root, ID1, one);
    Node *two = malloc(sizeof(Node));
    traverse_for_ID(bst->root, ID2, two);
    if (one->value.id != ID1 || two->value.id != ID2)
    {
        printf("ID not found !\n");
        free(one);
        free(two);
        return NULL;
    }
    if (ID1 == ID2) return one;
    else
    {
        Node *current = bst->root;
        Node *prev = current;
        while (current != NULL)
        {
            prev = current;
            if (prev->value.id == one->value.id)
            {
                return one;
            }
            if (prev->value.id == two->value.id)
            {
                return two;
            }
            if (one->value.height >= current->value.height && two->value.height < current-
>value.height)
            {
                return current;
            }
            else if(one->value.height < current->value.height && two->value.height >= current-
>value.height)
            {
                return current;
            }
            else if (one->value.height >= current->value.height && two->value.height >= current-
>value.height)
            {
                current = current->right;
            }
            else if (one->value.height < current->value.height && two->value.height < current-
>value.height)
            {
                current = current->left;
            }
        }
        return prev;
    }
}
```

TAKING FILE INPUT OF FORM int, [….]

```
BST **fileReader(char *name, int length)
{
    FILE *fptr = fopen(name, "r");
    if (fptr == NULL){
        printf("Error opening file");
        exit(1);
    }

    BST ** bst_arr = (BST**) malloc(length*sizeof(BST*));
    for (int j = 0; j < length; j++)
    {
        int n;
        char *num = (char *) malloc(sizeof(int));
        fscanf(fptr, "%[^,],[", num);
        char *line = (char*) malloc(sizeof(int));
        n = atoi(num);
        int *arr = (int*) calloc(n, sizeof(int));
        for (int i = 0; i < n-1; i++){
            fscanf(fptr, "%s", line);
            arr[i] = atoi(line);
        }
        fscanf(fptr, "%s]\n", line);
        arr[n-1] = atoi(line);
        bst_arr[j] = (BST*) malloc(sizeof(BST));
        bst_arr[j] = constructBST(arr,n);
    }
    fclose(fptr);
    return bst_arr;
}
```

TAKING FILE INPUT OF FORM int, [….] WITH STRTOK

```
int main(){
    FILE *fp;
    fp =fopen("n_integers.txt","r");
    char line = (char )malloc(200 *sizeof(char));

    int ROWS=0;
    int *arr = (int *)malloc (sizeof(int*)* 1000);
    for(int i=0;i<1000;i++){
        arr[i] = (int *) calloc (1000,sizeof(int));
    }
    while( fgets(line, 200 , fp)!= NULL){
        char *str = strtok(line, "[");
        int size = atoi(str);
        int i=0;
        str = strtok(NULL, " ");
        while(i<size){
            arr[ROWS][i]= atoi(str);
            i++;
            if(i== size) break;
            str = strtok(NULL, " ");
            if(str == NULL){
                fgets(line, 200 , fp);
                str = strtok(line," ");
            }
        }
        fgets(line, 200 , fp);
        ROWS++;
    }
    for(int i=0;i< ROWS;i++){
        printf("ROW NO: %d\n",i);
        for(int j=0;j<1000;j++){
            if(arr[i][j] == 0){
                break;
            }
            printf("%d ", arr[i][j]);
        }
        printf("\n\n");
    }
}
```

OPTIMAL BST SOLUTION 1

```
// Dynamic Programming code for Optimal Binary Search
// Tree Problem
#include <stdio.h>
```

```
#include <limits.h>
// A utility function to get sum of array elements
// freq[i] to freq[j]
int sum(int freq[], int i, int j);
/* A Dynamic Programming based function that calculates
minimum cost of a Binary Search Tree. */
int optimalSearchTree(int keys[], int freq[], int n)
{
        /* Create an auxiliary 2D matrix to store results
        of subproblems */
        int cost[n][n];
        /* cost[i][j] = Optimal cost of binary search tree
        that can be formed from keys[i] to keys[j].
        cost[0][n-1] will store the resultant cost */
        // For a single key, cost is equal to frequency of the key
        for (int i = 0; i < n; i++)
                cost[i][i] = freq[i];
        // Now we need to consider chains of length 2, 3, ... .
        // L is chain length.
        for (int L=2; L<=n; L++)
        {
                // i is row number in cost[][]
                for (int i=0; i<=n-L+1; i++)
                {
                        // Get column number j from row number i and
                        // chain length L
                        int j = i+L-1;
                        int off_set_sum = sum(freq, i, j);
                        cost[i][j] = INT_MAX;

                        // Try making all keys in interval keys[i..j] as root
                        for (int r=i; r<=j; r++)
                        {
                        // c = cost when keys[r] becomes root of this subtree
                        int c = ((r > i)? cost[i][r-1]:0) +
                                        ((r < j)? cost[r+1][j]:0) +
                                        off_set_sum;
                        if (c < cost[i][j])
                                cost[i][j] = c;
                        }
                }
        }
        return cost[0][n-1];
}
// A utility function to get sum of array elements
// freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
        int s = 0;
        for (int k = i; k <=j; k++)
        s += freq[k];
        return s;
}
// Driver program to test above functions
int main()
```

```
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ",
                optimalSearchTree(keys, freq, n));
    return 0;
}
```

OPTIMAL BST SOLUTION 2

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 1000
// Declare global cost matrix
int cost[MAX][MAX];
// Helper function to calculate the sum of frequencies from index i to j
int Sum(int freq[], int i, int j) {
    int s = 0;
    for (int k = i; k <= j; k++)
        s += freq[k];
    return s;
}
// Recursive function to find the optimal cost of a BST using memoization
int optCost_memoized(int freq[], int i, int j) {
    // Reuse cost already calculated for the subproblems.
    // Since we initialize cost matrix with 0 and frequency for a tree of one node,
    // it can be used as a stop condition
    if (cost[i][j])
        return cost[i][j];
    // Get sum of freq[i], freq[i+1], ... freq[j]
    int fsum = Sum(freq, i, j);
    // Initialize minimum value
    int Min = INT_MAX;
    // One by one consider all elements as
    // root and recursively find cost of
    // the BST, compare the cost with min
    // and update min if needed
    for (int r = i; r <= j; r++) {
        int c = optCost_memoized(freq, i, r - 1) + optCost_memoized(freq, r + 1, j) +
fsum;
        if (c < Min) {
            Min = c;
            // replace cost with new optimal calc
            cost[i][j] = c;
        }
    }
    // Return minimum value
    return cost[i][j];
}
// Main function to calculate the minimum cost of a BST
int optimalSearchTree(int keys[], int freq[], int n) {
    // Here array keys[] is assumed to be
    // sorted in increasing order. If keys[]
    // is not sorted, then add code to sort
    // keys, and rearrange freq[] accordingly.
```

```
        return optCost_memoized(freq, 0, n - 1);
}
int main() {
        int keys[] = {10, 12, 20};
        int freq[] = {34, 8, 50};
        int n = sizeof(keys) / sizeof(keys[0]);
        // cost[i][j] = Optimal cost of binary search
        // tree that can be formed from keys[i] to keys[j].
        // cost[0][n-1] will store the resultant cost
        memset(cost, 0, sizeof(cost));
        // For a single key, cost is equal to
        // frequency of the key
        for (int i = 0; i < n; i++)
                cost[i][i] = freq[i];
        cout << "Cost of Optimal BST is " << optimalSearchTree(keys, freq, n) << endl;
        return 0;
}
```

LAB TEST 2023

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_WORD_LEN 128
#define NUMBER_OF_ALPHABETS 26
/* structure to node the header of the linked list of words
      word: character array for the word string to be stored
      next: pointer to the next node in the linked list
*/
typedef struct wordsLLNode{
   char word[MAX_WORD_LEN];
   struct wordsLLNode * next;
} wordsLLNode;
/* structure to store the header of the linked list of words
      node: pointer to the head node of the linked list
*/
typedef struct wordsLLHeader{
   wordsLLNode * node;
} wordsLLHeader;
/* structure to store the tail of the linked list of words
      node: pointer to the tail node of the linked list
*/
typedef struct wordsLLTail{
   wordsLLNode * node;
} wordsLLTail;
/* structure to store the linked list of words
      header: header of the linked list
      tail: tail of the linked list
      length: length of the linked list
*/
typedef struct record{
   wordsLLHeader header;
   wordsLLTail tail;
   int length;
} record;
// function to find max of two numbers
```

```
int max(int a, int b){
   if(a>b) return a;
   else return b;
}
// function to create a new node with a given word stored
wordsLLNode * createNewNode(char * word){
   wordsLLNode * newNode = (wordsLLNode *) malloc(sizeof(wordsLLNode));
   newNode->next = NULL;
   strncpy(newNode->word,word,strlen(word));
   newNode->word[strlen(word)] = '\0';
   return newNode;
}
// function to add a node to a record with given word string
void addNodeToRecord(record * r, char * word){
   wordsLLNode * newNode = createNewNode(word);
   if(r->length==0){
      r->header.node = newNode;
      r->tail.node = newNode;
   } else{
      r->tail.node->next = newNode;
      r->tail.node = newNode;
   }
   r->length++;
}
// function takes the name of the file as an input parameter, creates a wordBuckets array,
reads the words from the input file line by line and inserts them into their appropriate buckets
of the wordBuckets array, and returns the wordBuckets array
record * readData(char * fileName){
   FILE * fp = fopen(fileName, "r");
   record * wordBuckets = (record *) malloc(sizeof(record)*NUMBER_OF_ALPHABETS);
   char word[MAX_WORD_LEN];
   while(fgets(word,MAX_WORD_LEN,fp)){
      //removes newline character from tail if read
      if(word[strlen(word)-1]=='\n')
         word[strlen(word)-1] = '\0';
      int firstAlphabetNumber;
      if(word[0]<='z' && word[0]>='a')
         firstAlphabetNumber = word[0]-'a';
      else
         firstAlphabetNumber = word[0]-'A';
      addNodeToRecord(wordBuckets + firstAlphabetNumber,word);
   }
   fclose(fp);
   return wordBuckets;
}
// function takes the wordBuckets array as an input parameter, and finds the maximum gap
between any two adjacent words stored in the wordBuckets array, when they are
lexicographically ordered
int findmaxGap(record * wordBuckets){
   int maxGap = 0;
   int currAlpha = -1;
   for(int i=0;i<NUMBER_OF_ALPHABETS;i++){
      if(wordBuckets[i].length!=0){
         if(currAlpha!=-1)
            maxGap = max(maxGap,i-currAlpha);
            currAlpha = i;
        }
    }

    return maxGap;
}
// function takes the head of a linked list as an input partitions that linked list into two
partitions
wordsLLNode * partitionLinkedList(wordsLLHeader head, int numberOfNodes){
    wordsLLNode * part1head=NULL, * part1tail=NULL;
    wordsLLNode * part2head=NULL, * part2tail=NULL;
    wordsLLNode * currNode = head.node;
    for(int i=0;i<numberOfNodes;i++){
        if(currNode->word[2]>'m'){
            if(part2head==NULL){
                part2head = currNode;
                part2tail = currNode;
            }
            else{
                part2tail->next = currNode;
                part2tail = currNode;
            }
        } else{
            if(part1head==NULL){
                part1head = currNode;
                part1tail = currNode;
            }
            else{
                part1tail->next = currNode;
                part1tail = currNode;
            }
        }
        currNode = currNode->next;
    }
    if(part1head==NULL){
        part2tail->next=NULL;
        return part2head;
    }
    else{
        part1tail->next = part2head;
        if(part2tail) part2tail->next=NULL;
        return part1head;
    }
}
// function takes the wordBuckets array as an input parameter, and partitions each linked list
into two partitions
void partitionLists(record * wordBuckets){
    for(int i=0; i<NUMBER_OF_ALPHABETS; i++){
        if(wordBuckets[i].length==0) continue;
        wordBuckets[i].header.node =
partitionLinkedList(wordBuckets[i].header,wordBuckets[i].length);
    }
}
// function takes the wordBuckets array as an input parameter, and prints the words stored in
it.
```

```
void printData(record * wordBuckets){
    for(int i=0; i<NUMBER_OF_ALPHABETS; i++){
        wordsLLNode * itr =  wordBuckets[i].header.node;
        while(itr!=NULL){
            printf("%s\n",itr->word);
            itr = itr->next;
        }
    }
}
// function takes 2 input strings returns 0 if 1st input is lexicographically bigger else returns 1
int stringCompare(char * a, char * b){
    int lena = strlen(a), lenb = strlen(b);
    int minLen = lena<lenb ? lena : lenb;
    for(int i=0; i<minLen; i++){
        if(a[i]<b[i]) return 1;
        else if(a[i]>b[i]) return 0;
        else continue;
    }
    if(lena>lenb) return 0;
    else return 1;
}
// Merges 2 paritioned sorted arrays
wordsLLNode * mergeIn(wordsLLNode * head, int st1, int st2, int en2){
    wordsLLNode * head1=head, * head2, * newListHead=NULL, * newListTail=NULL;
    wordsLLNode * insertStart = head, * insertEnd=NULL;
    for(int i=0;i<st1;i++){
        head1 = head1->next;
        if(i!=st1-1) insertStart = insertStart->next;
    }
    head2 = head1;
    for(int i=0;i<st2-st1;i++)
        head2 = head2->next;
    int len1 = st2-st1, len2 = en2-st2+1;
    int coveredLen1 = 0, coveredLen2 = 0;
    for(int i=0;i<len1+len2; i++){
        if(coveredLen1==len1){
            if(newListHead==NULL){
                newListHead = head2;
                newListTail = head2;
            } else{
                newListTail->next = head2;
                newListTail = head2;
            }
            head2 = head2->next;
            coveredLen2++;
        } else if(coveredLen2==len2){
            if(newListHead==NULL){
                newListHead = head1;
                newListTail = head1;
            } else{
                newListTail->next = head1;
                newListTail = head1;
            }
            head1 = head1->next;
            coveredLen1++;
        } else if(stringCompare(head1->word,head2->word)){
            if(newListHead==NULL){
                newListHead = head1;
                newListTail = head1;
            } else{
                newListTail->next = head1;
                newListTail = head1;
            }
            head1 = head1->next;
            coveredLen1++;
        } else{
            if(newListHead==NULL){
                newListHead = head2;
                newListTail = head2;
            } else{
                newListTail->next = head2;
                newListTail = head2;
            }
            head2 = head2->next;
            coveredLen2++;
        }
    }
    if(len2==0)
        newListTail->next = head1;
    else newListTail->next = head2;
    if(st1==0) return newListHead;
    else{
        insertStart->next = newListHead;
        return head;
    }
    return head;
}
// function takes the wordBuckets array as an input parameter, and lexicographically orders
each linked list stored in it using iterative merge sort
void mergeSortBuckets(record * wordBuckets){
    for(int i=0; i<NUMBER_OF_ALPHABETS; i++){
        if(wordBuckets[i].length<=1) continue;
        int maxSlSz, slSz, st1, last=wordBuckets[i].length-1;
        for(maxSlSz=1; wordBuckets[i].length>maxSlSz; maxSlSz*=2) ;
        /* Postcondition: maxSlSz/2 < n <= maxSlSz */
        maxSlSz /= 2;
        /* Postcondition: maxSlSz < n <= 2*maxSlSz */
        for (slSz=1; slSz<=maxSlSz; slSz*=2) {
            for (st1=0; st1<=last; st1=st1+2*slSz) {
                int st2=st1+slSz;
                int en2=st2+slSz-1;
                if (st2 > last) continue;
                if (en2 > last){
                    en2 = last;
                }
                wordBuckets[i].header.node = mergeIn(wordBuckets[i].header.node, st1, st2, en2);
            }
        }
    }
}
```

```
int main(int noOfArgs, char* args[]){
   if(noOfArgs<2){
      printf("Enter file name\n");
      exit(0);
   }
   record * wordBuckets = readData(args[1]);
   int maxGap = findmaxGap(wordBuckets);
   printf("Maximum gap is: %d\n\n",maxGap);
   printf("Printing wordBuckets array after partitioning:\n");
   partitionLists(wordBuckets);
   printData(wordBuckets);
   printf("\nPrinting wordBuckets array after lexicographical ordering:\n");
   mergeSortBuckets(wordBuckets);
   printData(wordBuckets);
   return 0;
}
```

MOCK LAB TEST – PROCESS.C

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include "process.h"
#define INITIAL_SIZE_ARRAY_DEQUE 2
// Use the next_first and next_last pointers to track the circular buffer additions and deletions
// Feel free to modify the struct if you want to use a different technique to track the two ends
of the array deque.
typedef struct process_array_deque {
   process **processes;
   size_t next_first;
   size_t next_last;
   size_t size;
   size_t capacity;
} array_deque;
// creates an empty process array deque with INITIAL_SIZE_ARRAY_DEQUE size of the
internal processes array
// and returns a pointer to it
array_deque *create_empty_process_array_deque();
/**
 * adds to the front of the array deque in constant time "on average"
 * @return true if the addition was successful, false otherwise
 * Time Complextiy: O(1) on average
 */
bool add_first_array_deque(array_deque *ad, process p);
/**
 * adds to the back of the array deque in constant time "on average"
 * @return true if the addition was successful, false otherwise
 * Time Complextiy: O(1) on average
 */
bool add_last_array_deque(array_deque *ad, process p);
/**
 * removes the front of the array deque in constant time "on average"
 * @return true if the addition was successful, false otherwise
 * Time Complextiy: O(1) on average
 */
bool remove_first_array_deque(array_deque *ad, process *p);
```

```
/**
 * removes the back of the array deque in constant time "on average"
 * @return true if the addition was successful, false otherwise
 * Time Complextiy: O(1) on average
 */
bool remove_last_array_deque(array_deque *ad, process *p);
/**
 * @return the size of the array deque
 * Time Complextiy: Theta(1)
 */
size_t get_size_array_deque(array_deque *ad);
void print_array_deque(array_deque* ad);
static unsigned int get_first_index(array_deque *ad);
static unsigned int get_last_index(array_deque *ad);
static unsigned int decrement_index(array_deque *d, size_t index);
static unsigned int increment_index(array_deque *d, size_t index);
static void resize_if_needed(array_deque *d);
static process *create_process(process p);
array_deque *create_empty_process_array_deque() {
    // COMPLETE
    array_deque* ad = (array_deque*) malloc(sizeof(array_deque));
    ad->processes = malloc(INITIAL_SIZE_ARRAY_DEQUE*sizeof(process*));
    ad->capacity=INITIAL_SIZE_ARRAY_DEQUE;
    ad->size=0;
    ad->next_first=0;
    ad->next_last=0;
    return ad;
}
static unsigned int decrement_index(array_deque *ad, size_t index) {
    index = index - 1;
    if (index == -1) {
        index = ad->capacity - 1;
    }
    return index;
}
static unsigned int increment_index(array_deque *d, size_t index) {
    return (index + 1) % d->capacity;
}
static unsigned int get_first_index(array_deque *ad) {
    return increment_index(ad, ad->next_first);
}
static unsigned int get_last_index(array_deque *ad) {
    return decrement_index(ad, ad->next_last);
}

bool add_first_array_deque(array_deque *ad, process p) {
    // COMPLETE
    process * to_add = create_process(p);
    if(!to_add){
        return false;
    }
    resize_if_needed(ad);
    ad->processes[ad->next_first]=to_add;
    ad->next_first = decrement_index(ad, ad->next_first);
    ad->size++;
    return true;

}
static process *create_process(process p) {
    process *pro = malloc(sizeof(process));
    if (!pro) return NULL;
    *pro = p;
    return pro;
}
bool add_last_array_deque(array_deque *ad, process p) {
    // COMPLETE
    process * to_add = create_process(p);
    if(!to_add){
        return false;
    }
    resize_if_needed(ad);
    ad->processes[ad->next_last]=to_add;
    ad->next_last = increment_index(ad, ad->next_last);
    ad->size++;
    return true;
}
bool remove_first_array_deque(array_deque *ad, process *p) {
    // COMPLETE
    if(!p){
        return false;
    }
    *p = *(ad->processes[ad->next_first]);
    ad->next_first = increment_index(ad, ad->next_first);
    ad->size--;
    resize_if_needed(ad);
    if(!ad){
        return false;
    }
    return true;
}
bool remove_last_array_deque(array_deque *ad, process *p) {
    // COMPLETE
    if(!p){
        return false;
    }
    *p = *(ad->processes[ad->next_last]);
    ad->next_last = decrement_index(ad, ad->next_last);
    ad->size--;
    resize_if_needed(ad);
    if(!ad){
        return false;
    }
    return true;
}
size_t get_size_array_deque(array_deque *ad) {
    return ad->size;
}
static void resize_if_needed(array_deque *ad) {
    // COMPLETE
    process **new_processes = calloc(2 * ad->capacity, sizeof(process *));
```

```
if (new_processes == NULL) {
    // Handle allocation failure
}
// Copy elements from the old array to the new one
for (size_t i = 0; i < ad->size; i++) {
    new_processes[i] = ad->processes[(ad->next_first + i) % ad->capacity];
}
free(ad->processes); // Free memory of the old array
ad->processes = new_processes; // Update pointer to the new array
}
void print_array_deque(array_deque *ad) {
    if (!ad || !ad->processes) {
        printf("Array deque is NULL\n");
        return;
    }
    for(int i = 0; i < ad->size; i++) {
        if (ad->processes[i]) {
            process p = *(ad->processes[i]);
            printf("%-10s%-15d%-15d%-15d%-15d\n",
                   p.name,
                   p.arrival,
                   p.cpu_burst,
                   p.wait,
                   p.turnaround);
        } else {
            printf("NULL\n");
        }
    }
}


int main(void) {
    array_deque *ad = create_empty_process_array_deque();
    process p1 = {"p1", 1, 0, 8, 0, 0, 8};
    process p2 = {"p2", 2, 1, 4, 0, 0, 4};
    process p3 = {"p3", 3, 4, 9, 0, 0, 9};
    process p4 = {"p4", 4, 2, 5, 0, 0, 5};
    process p5 = {"p5", 5, 3, 2, 0, 0, 2};
    add_first_array_deque(ad, p1);
    add_first_array_deque(ad, p2);
    add_first_array_deque(ad, p3);
    add_first_array_deque(ad, p4);
    add_first_array_deque(ad, p5);
    add_last_array_deque(ad, p1);
    print_array_deque(ad);
    process curr;
    remove_first_array_deque(ad, &curr);
    remove_first_array_deque(ad, &curr);
    print_array_deque(ad);
}
```

MOCK LAB TEST – SCHEDULER.C

```
#include "scheduler.h"
#include "array_deque.h"
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include "linked_deque.h"
// The TIME QUANTUM used by this Round Robin Simulator
#define TIME_QUANTUM 3
static void print_stats(process p);
static process **read_processes_from_file(char *filename, int *num_processes_ptr);
/**
 * DO NOT MODIFY THIS FUNCTION
 * Reads the processes from the file in the format Process_Name Process_id Arrival_time
CPU_burst
 * subsequently stores the number of processes in the location pointed by
num_processes_ptr
 * @return an array of process pointers read from the file
 */
static process **read_processes_from_file(char *filename, int *num_processes_ptr) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        fprintf(stderr, "Error: Could not open file '%s'\n", filename);
        exit(EXIT_FAILURE);
    }
    // Read the number of processes from the first line
    int num_processes;
    fscanf(file, "%d", &num_processes);
    *num_processes_ptr = num_processes;
    // Allocate memory for the process pointers
    process **processes = malloc(num_processes * sizeof(process *));
    if (processes == NULL) {
        fprintf(stderr, "Error: Failed to allocate memory for processes\n");
        exit(EXIT_FAILURE);
    }
    // Read each process from the file
    for (int i = 0; i < num_processes; i++) {
        process *p = malloc(sizeof(process));
        if (p == NULL) {
            fprintf(stderr, "Error: Failed to allocate memory for process\n");
            exit(EXIT_FAILURE);
        }
        // Read the process data from the file
        char name[32];
        unsigned int pid, arrival, cpu_burst;
        fscanf(file, "%s %u %u %u", name, &pid, &arrival, &cpu_burst);
        p->name = strdup(name);
        p->pid = pid;
        p->arrival = arrival;
        p->cpu_burst = cpu_burst;

        // Initialize the other process fields to 0
        p->turnaround = 0;
        p->wait = 0;
        p->remaining_time = cpu_burst;

        processes[i] = p;
    }
    fclose(file);
    return processes;
}
void visualize_round_robin(char *path) {
    int num_processes;
    process **processes = read_processes_from_file(path, &num_processes);
    printf("Number of processes: %d\n", num_processes);
    printf("%-10s%-15s%-15s%-15s%-15s\n", "Process", "Arrival Time", "Burst Time",
"Waiting Time", "Turnaround Time");
    linked_deque *ld = create_linked_process_deque();
    // COMPLETE using the ld for storing processes as described
    int runningtime=0;
    int i;
    // for(i=0;i<num_processes;i++){
    //     print_stats(*processes[i]);
    // }
    add_last_linked_deque(ld, *processes[0]);
    int start = 1;
    while(ld->list->size!=0){
        process a;
        remove_first_linked_deque(ld, &a);
        if(a.remaining_time==0){
            if(a.cpu_burst>TIME_QUANTUM){
                a.remaining_time = a.cpu_burst-3;
                for(i=start;i<num_processes;i++){
                    if(processes[i]->arrival<=runningtime+TIME_QUANTUM){
                        add_last_linked_deque(ld, *processes[i]);
                        start++;
                    }
                }
                add_last_linked_deque(ld, a);
                runningtime+=TIME_QUANTUM;
            }
            else{
                for(i=start;i<num_processes;i++){
                    if(processes[i]->arrival<=runningtime+a.cpu_burst){
                        add_last_linked_deque(ld, *processes[i]);
                        start++;
                    }
                }
                runningtime+=a.cpu_burst;
                a.remaining_time=0;
                a.turnaround=runningtime-a.arrival;
                a.wait=a.turnaround-a.cpu_burst;
                print_stats(a);
            }
        }
        else{
            if(a.remaining_time>3){
                a.remaining_time-=TIME_QUANTUM;
                for(i=start;i<num_processes;i++){
                    if(processes[i]->arrival<=runningtime+TIME_QUANTUM){
                        add_last_linked_deque(ld, *processes[i]);
                        start++;
                    }
                }
                runningtime+=TIME_QUANTUM;
                add_last_linked_deque(ld,a);
            }
            else{
                for(i=start;i<num_processes;i++){
                    if(processes[i]->arrival<=runningtime+a.remaining_time){
                        add_last_linked_deque(ld, *processes[i]);
                        start++;
                    }
                }
                runningtime+=a.remaining_time;
                a.remaining_time=0;
                a.turnaround=runningtime-a.arrival;
                a.wait=a.turnaround-a.cpu_burst;
                print_stats(a);
            }
        }
    }
    // Free the allocated memory
    for (int i = 0; i < num_processes; i++) {
        free(processes[i]->name);
        free(processes[i]);
    }
    free(processes);
}
// Prints the stats for the process p to stdout
static void print_stats(process p) {
    printf("%-10s%-15d%-15d%-15d%-15d\n",
           p.name,
           p.arrival,
           p.cpu_burst,
           p.wait,
           p.turnaround);
}
```

CTYPE functions:

1. int **isalnum**(int c)

This function checks whether the passed character is alphanumeric.

2. int isalpha(int c)

This function checks whether the passed character is alphabetic.

3. int iscntrl(int c)

This function checks whether the passed character is control character.

4. int **isdigit**(int c)

This function checks whether the passed character is decimal digit.

5. int isgraph(int c)

This function checks whether the passed character has graphical representation using locale.

6. int islower(int c)

This function checks whether the passed character is lowercase letter.

7. int isprint(int c)

This function checks whether the passed character is printable.

8. int ispunct(int c)

This function checks whether the passed character is a punctuation character.

9. int **isspace**(int c)

This function checks whether the passed character is white-space.

10. int **isupper**(int c)

This function checks whether the passed character is an uppercase letter.

11. int isxdigit(int c)

This function checks whether the passed character is a hexadecimal digit.

STRING functions:

1. void *memchr(const void *str, int c, size_t n)

Searches for the first occurrence of the character c (an unsigned char) in the first n bytes of the string pointed to, by the argument str.

2. int memcmp(const void *str1, const void *str2, size_t n)

Compares the first n bytes of str1 and str2.

3. void *memcpy(void *dest, const void *src, size_t n)

Copies n characters from src to dest.

4. void *memmove(void *dest, const void *src, size_t n)

Another function to copy n characters from str2 to str1.

5. void *memset(void *str, int c, size_t n)

Copies the character c (an unsigned char) to the first n characters of the string pointed to, by the argument str.

6. char *strcat(char *dest, const char *src)

Appends the string pointed to, by src to the end of the string pointed to by dest.

7. char *strncat(char *dest, const char *src, size_t n)

Appends the string pointed to, by src to the end of the string pointed to, by dest up to n characters long.

8. char *strchr(const char *str, int c)

Searches for the first occurrence of the character c (an unsigned char) in the string pointed to, by the argument str.

9. int strcmp(const char *str1, const char *str2)

Compares the string pointed to, by str1 to the string pointed to by str2.

•if Return value < 0 then it indicates str1 is less than str2.

•if Return value > 0 then it indicates str2 is less than str1.

•if Return value = 0 then it indicates str1 is equal to str2.

10. int strncmp(const char *str1, const char *str2, size_t n)

Compares at most the first n bytes of str1 and str2.

11. int strcoll(const char *str1, const char *str2)

Compares string str1 to str2. The result is dependent on the LC_COLLATE setting of the location.

12. char ***strcpy**(char *dest, const char *src)

Copies the string pointed to, by src to dest.

13. char ***strncpy**(char *dest, const char *src, size_t n)

Copies up to n characters from the string pointed to, by src to dest.

14. size_t strcspn(const char *str1, const char *str2)

Calculates the length of the initial segment of str1 which consists entirely of characters not in str2.

15. char *strerror(int errnum)

Searches an internal array for the error number errnum and returns a pointer to an error message string.

16. size_t **strlen**(const char *str)

Computes the length of the string str up to but not including the terminating null character.

17. char *strpbrk(const char *str1, const char *str2)

Finds the first character in the string str1 that matches any character specified in str2.

18. char *strrchr(const char *str, int c)

Searches for the last occurrence of the character c (an unsigned char) in the string pointed to by the argument str.

19. size_t strspn(const char *str1, const char *str2)

Calculates the length of the initial segment of str1 which consists entirely of characters in str2.

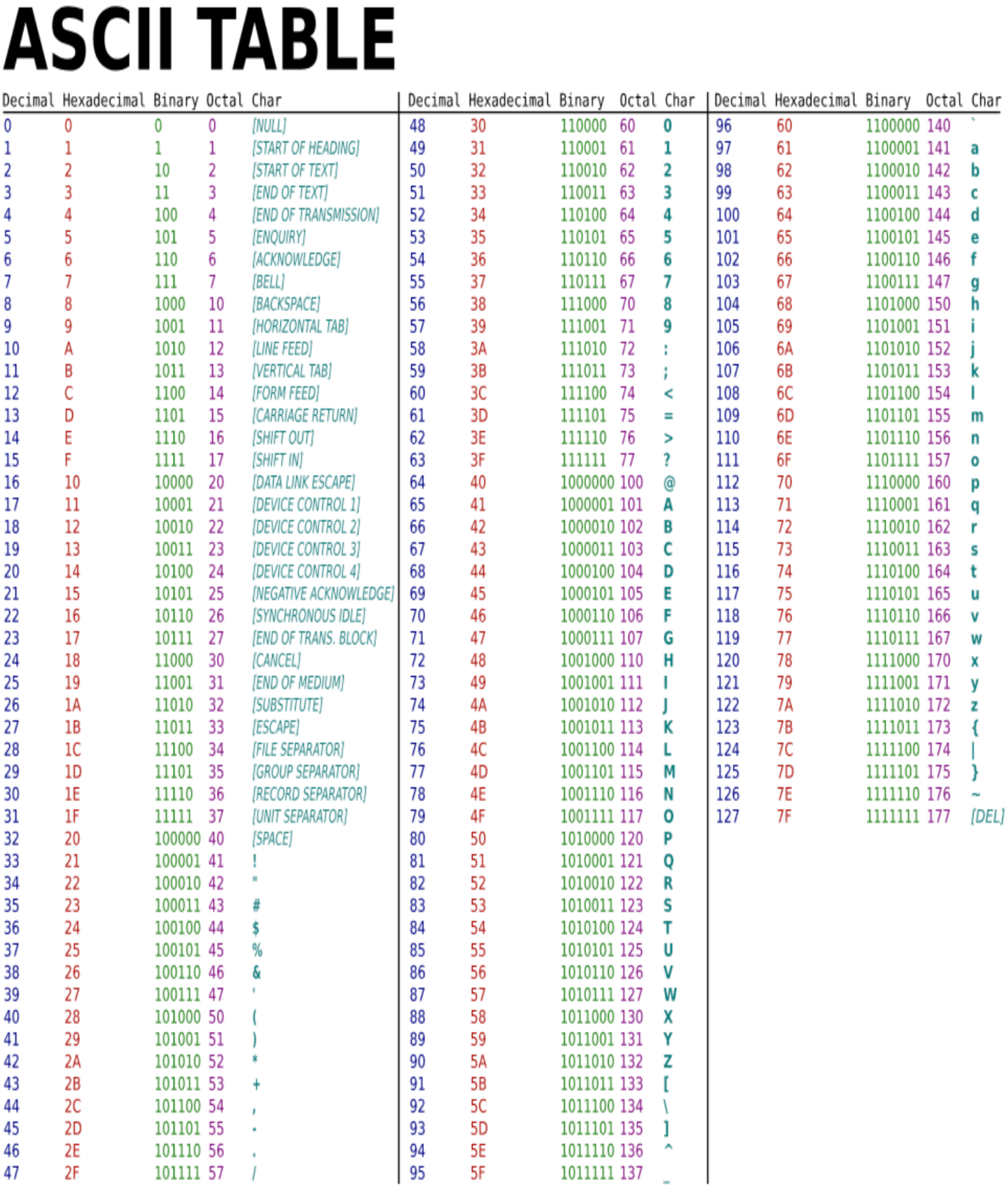
20. char ***strstr**(const char *haystack, const char *needle)3

Finds the first occurrence of the entire string needle (not including the terminating null character) which appears in the string haystack.

21. char ***strtok**(char *str, const char *delim)

Breaks string str into a series of tokens separated by delim.

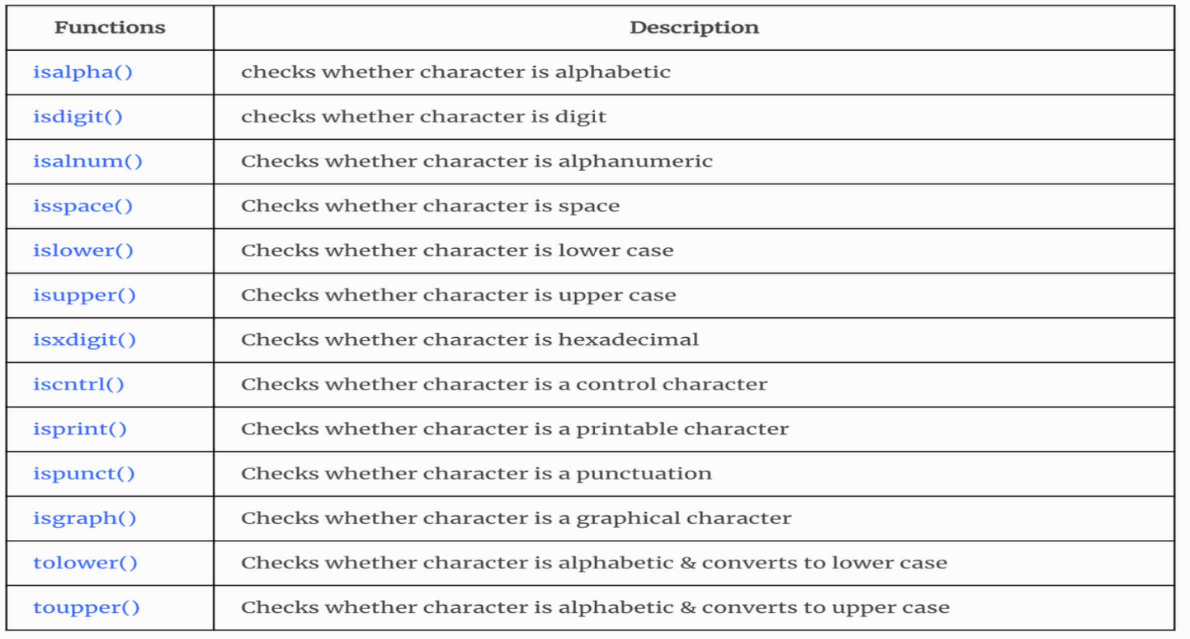
22. size_t strxfrm(char *dest, const char *src, size_t n) Transforms the first n characters of the string src into current locale and places them in the string dest.



ASCII TABLE

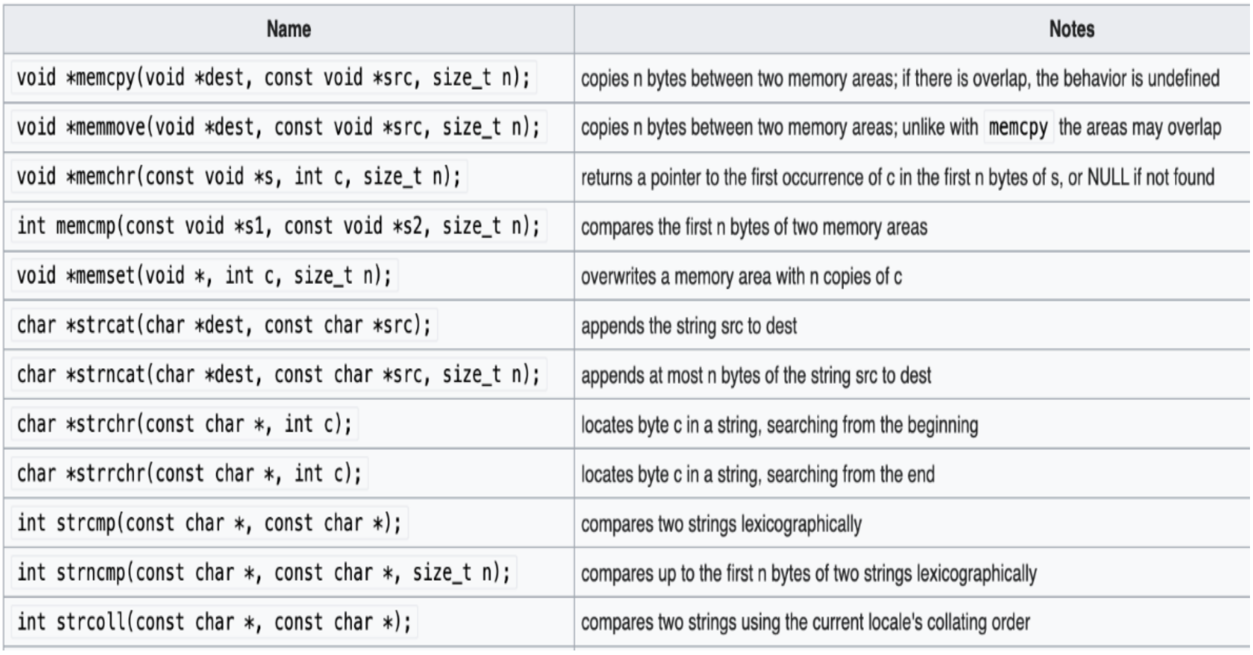
Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[END OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

ctype.h

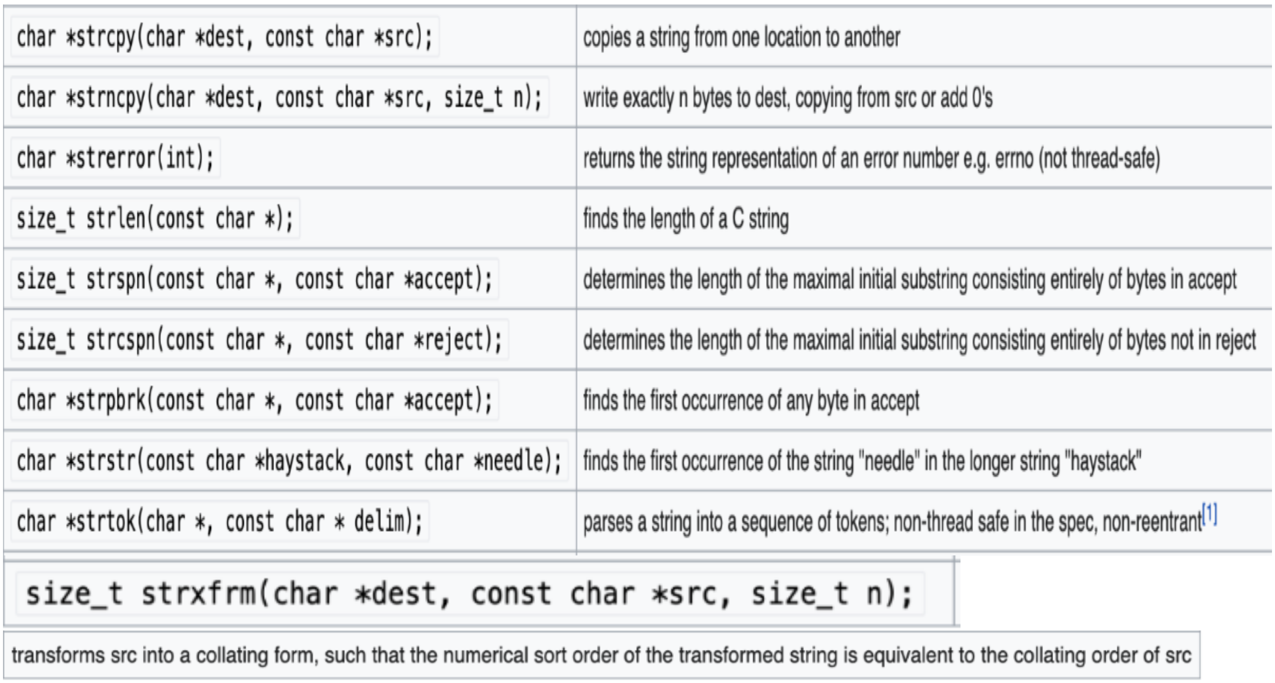


Functions	Description
isalpha()	checks whether character is alphabetic
isdigit()	checks whether character is digit
isalnum()	Checks whether character is alphanumeric
isspace()	Checks whether character is space
islower()	Checks whether character is lower case
isupper()	Checks whether character is upper case
isxdigit()	Checks whether character is hexadecimal
iscntrl()	Checks whether character is a control character
isprint()	Checks whether character is a printable character
ispunct()	Checks whether character is a punctuation
isgraph()	Checks whether character is a graphical character
tolower()	Checks whether character is alphabetic & converts to lower case
toupper()	Checks whether character is alphabetic & converts to upper case

string.h

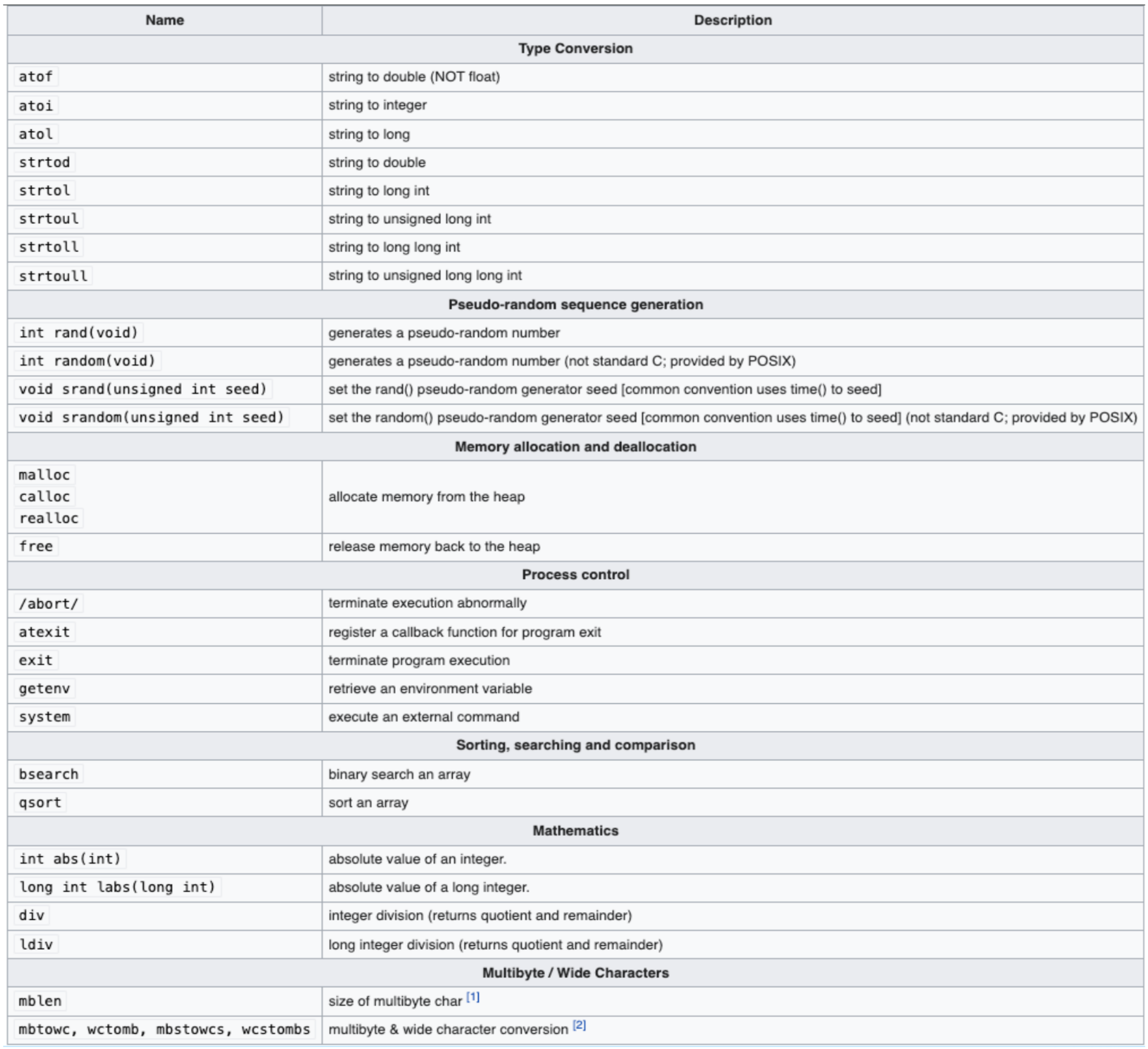


Name	Notes
`void *memcpy(void *dest, const void *src, size_t n);`	copies n bytes between two memory areas; if there is overlap, the behavior is undefined
`void *memmove(void *dest, const void *src, size_t n);`	copies n bytes between two memory areas; unlike with `memcpy` the areas may overlap
`void *memchr(const void *s, int c, size_t n);`	returns a pointer to the first occurrence of c in the first n bytes of s, or NULL if not found
`int memcmp(const void *s1, const void *s2, size_t n);`	compares the first n bytes of two memory areas
`void *memset(void *, int c, size_t n);`	overwrites a memory area with n copies of c
`char *strcat(char *dest, const char *src);`	appends the string src to dest
`char *strncat(char *dest, const char *src, size_t n);`	appends at most n bytes of the string src to dest
`char *strchr(const char *, int c);`	locates byte c in a string, searching from the beginning
`char *strrchr(const char *, int c);`	locates byte c in a string, searching from the end
`int strcmp(const char *, const char *);`	compares two strings lexicographically
`int strncmp(const char *, const char *, size_t n);`	compares up to the first n bytes of two strings lexicographically
`int strcoll(const char *, const char *);`	compares two strings using the current locale's collating order



`char *strcpy(char *dest, const char *src);`	copies a string from one location to another
`char *strncpy(char *dest, const char *src, size_t n);`	write exactly n bytes to dest, copying from src or add 0's
`char *strerror(int);`	returns the string representation of an error number e.g. errno (not thread-safe)
`size_t strlen(const char *);`	finds the length of a C string
`size_t strspn(const char *, const char *accept);`	determines the length of the maximal initial substring consisting entirely of bytes in accept
`size_t strcspn(const char *, const char *reject);`	determines the length of the maximal initial substring consisting entirely of bytes not in reject
`char *strpbrk(const char *, const char *accept);`	finds the first occurrence of any byte in accept
`char *strstr(const char *haystack, const char *needle);`	finds the first occurrence of the string "needle" in the longer string "haystack"
`char *strtok(char *, const char * delim);`	parses a string into a sequence of tokens; non-thread safe in the spec, non-reentrant[1]
`size_t strxfrm(char *dest, const char *src, size_t n);`	transforms src into a collating form, such that the numerical sort order of the transformed string is equivalent to the collating order of src

stdlib.h

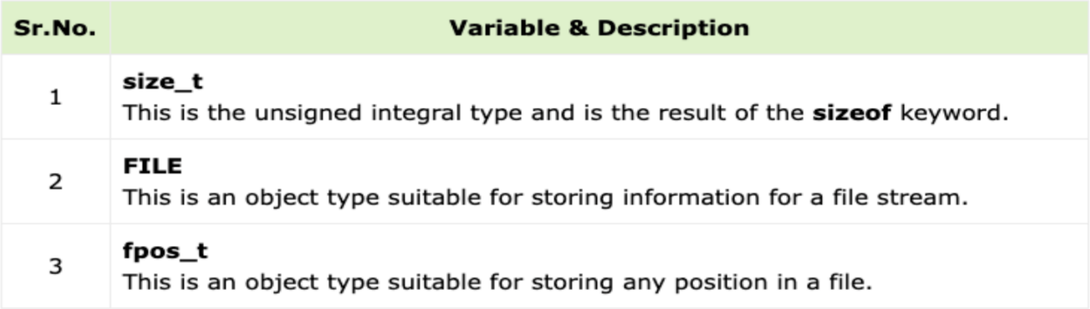


Name	Description
Type Conversion	
`atof`	string to double (NOT float)
`atoi`	string to integer
`atol`	string to long
`strtod`	string to double
`strtol`	string to long int
`strtoul`	string to unsigned long int
`strtoll`	string to long long int
`strtoull`	string to unsigned long long int
Pseudo-random sequence generation	
`int rand(void)`	generates a pseudo-random number
`int random(void)`	generates a pseudo-random number (not standard C; provided by POSIX)
`void srand(unsigned int seed)`	set the rand() pseudo-random generator seed [common convention uses time() to seed]
`void srandom(unsigned int seed)`	set the random() pseudo-random generator seed [common convention uses time() to seed] (not standard C; provided by POSIX)
Memory allocation and deallocation	

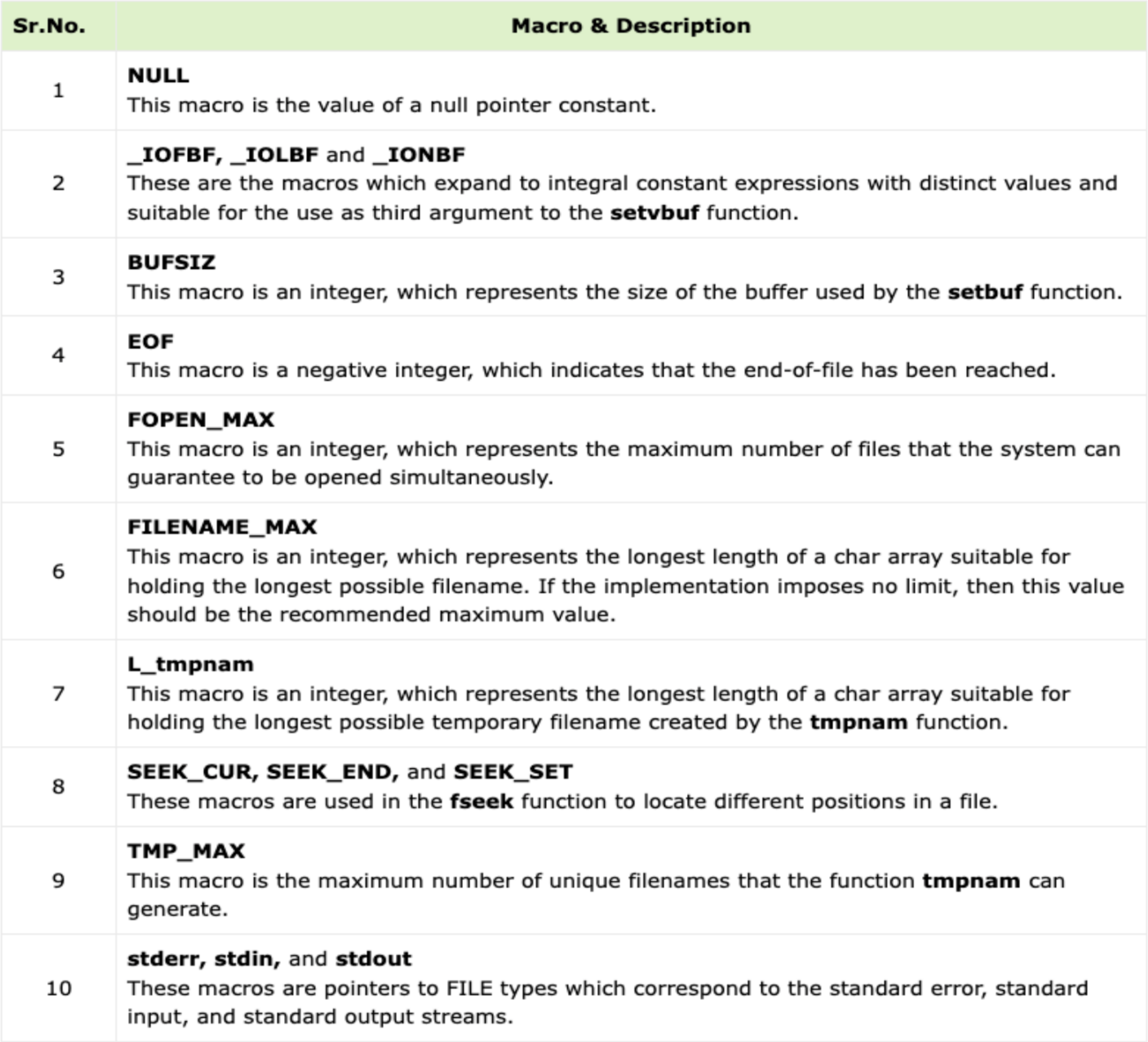
`malloc`
`calloc`
`realloc`
	allocate memory from the heap

`free`	release memory back to the heap
Process control	
`/abort/`	terminate execution abnormally
`atexit`	register a callback function for program exit
`exit`	terminate program execution
`getenv`	retrieve an environment variable
`system`	execute an external command
Sorting, searching and comparison	
`bsearch`	binary search an array
`qsort`	sort an array
Mathematics	
`int abs(int)`	absolute value of an integer.
`long int labs(long int)`	absolute value of a long integer.
`div`	integer division (returns quotient and remainder)
`ldiv`	long integer division (returns quotient and remainder)
Multibyte / Wide Characters	
`mblen`	size of multibyte char [1]
`mbtowc, wctomb, mbstowcs, wcstombs`	multibyte & wide character conversion [2]

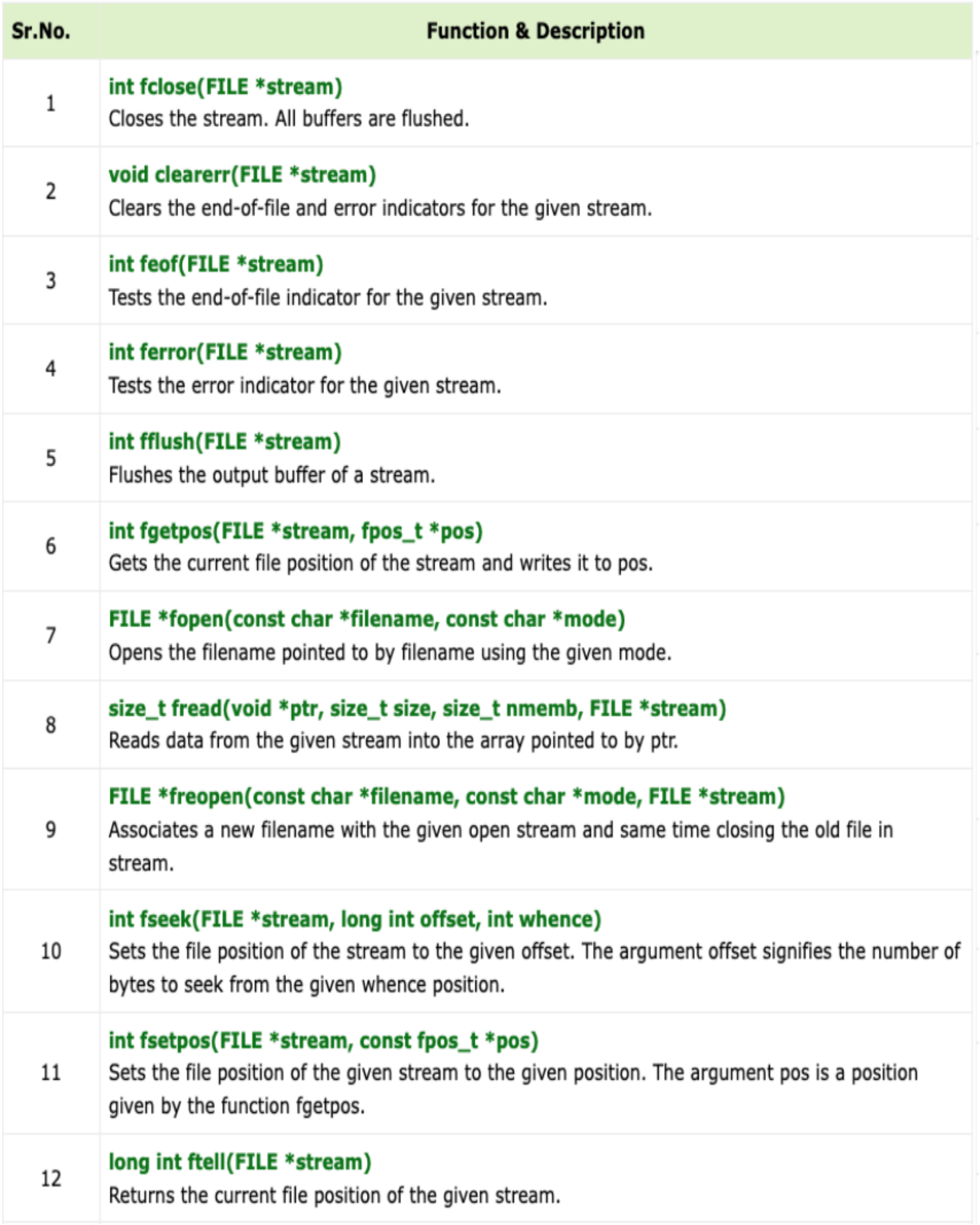
stdio.h

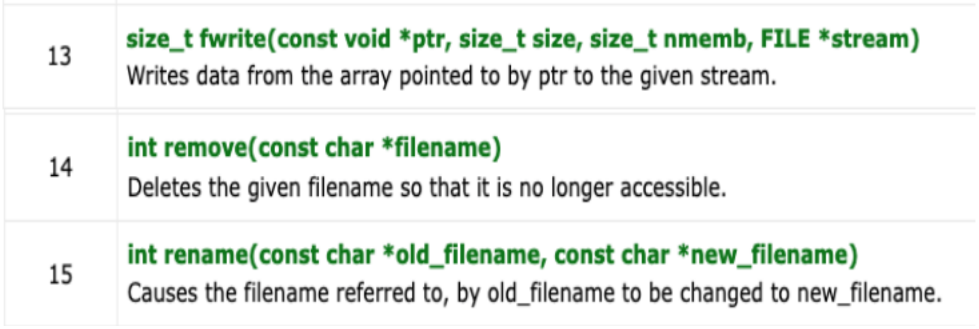


Sr.No.	Variable & Description
1	**size_t** This is the unsigned integral type and is the result of the **sizeof** keyword.
2	**FILE** This is an object type suitable for storing information for a file stream.
3	**fpos_t** This is an object type suitable for storing any position in a file.

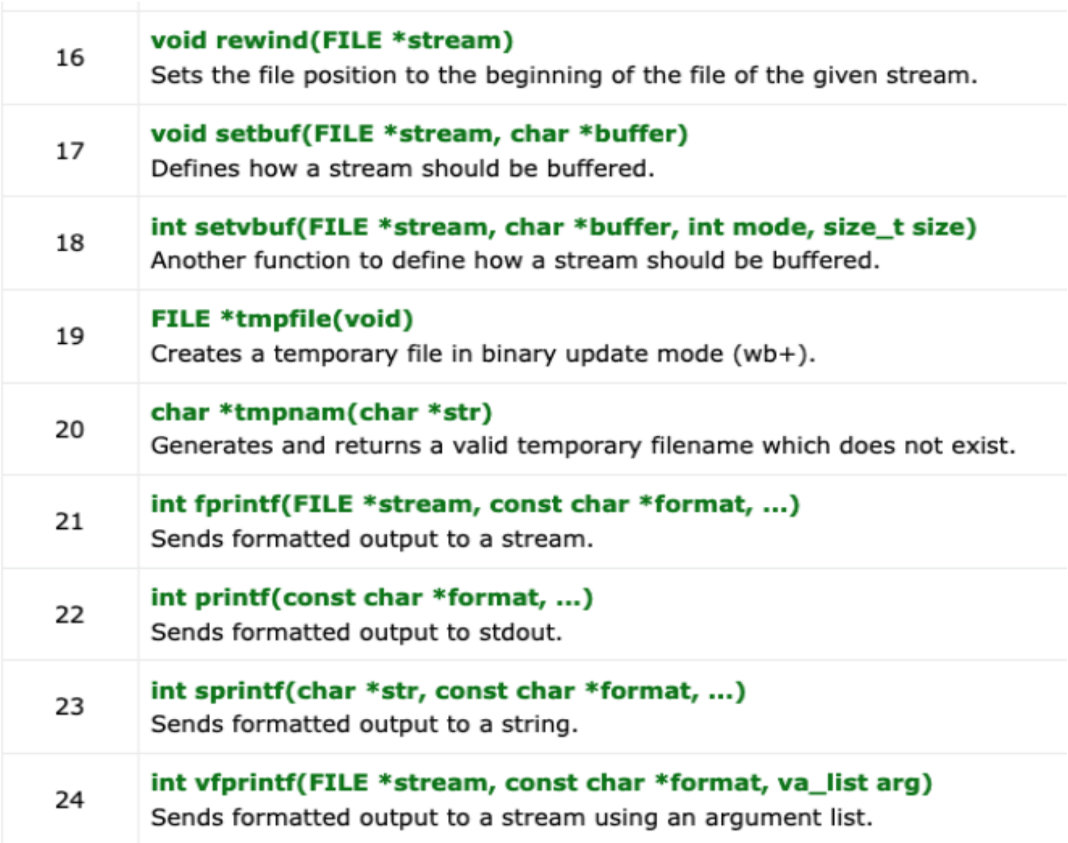


Sr.No.	Macro & Description
1	**NULL** This macro is the value of a null pointer constant.
2	**_IOFBF, _IOLBF** and **_IONBF** These are the macros which expand to integral constant expressions with distinct values and suitable for the use as third argument to the **setvbuf** function.
3	**BUFSIZ** This macro is an integer, which represents the size of the buffer used by the **setbuf** function.
4	**EOF** This macro is a negative integer, which indicates that the end-of-file has been reached.
5	**FOPEN_MAX** This macro is an integer, which represents the maximum number of files that the system can guarantee to be opened simultaneously.
6	**FILENAME_MAX** This macro is an integer, which represents the longest length of a char array suitable for holding the longest possible filename. If the implementation imposes no limit, then this value should be the recommended maximum value.
7	**L_tmpnam** This macro is an integer, which represents the longest length of a char array suitable for holding the longest possible temporary filename created by the **tmpnam** function.
8	**SEEK_CUR, SEEK_END,** and **SEEK_SET** These macros are used in the **fseek** function to locate different positions in a file.
9	**TMP_MAX** This macro is the maximum number of unique filenames that the function **tmpnam** can generate.
10	**stderr, stdin,** and **stdout** These macros are pointers to FILE types which correspond to the standard error, standard input, and standard output streams.

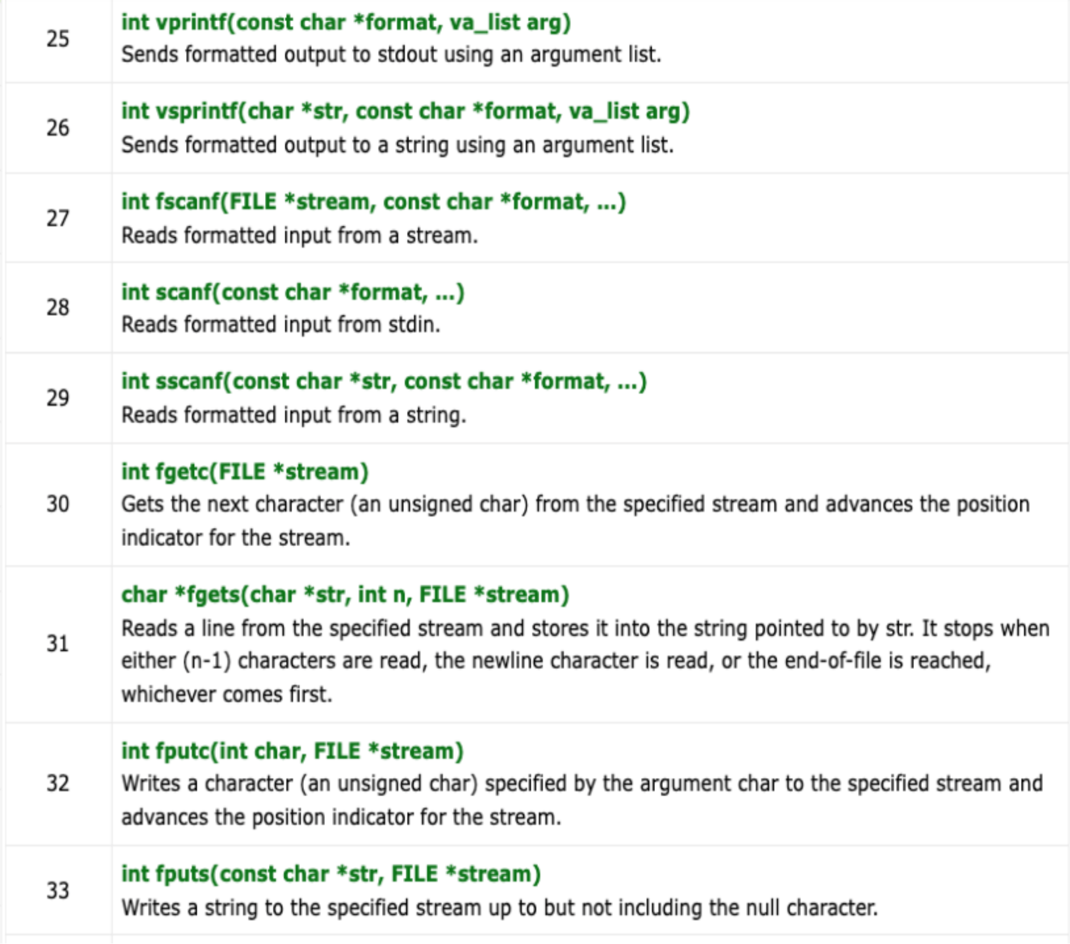




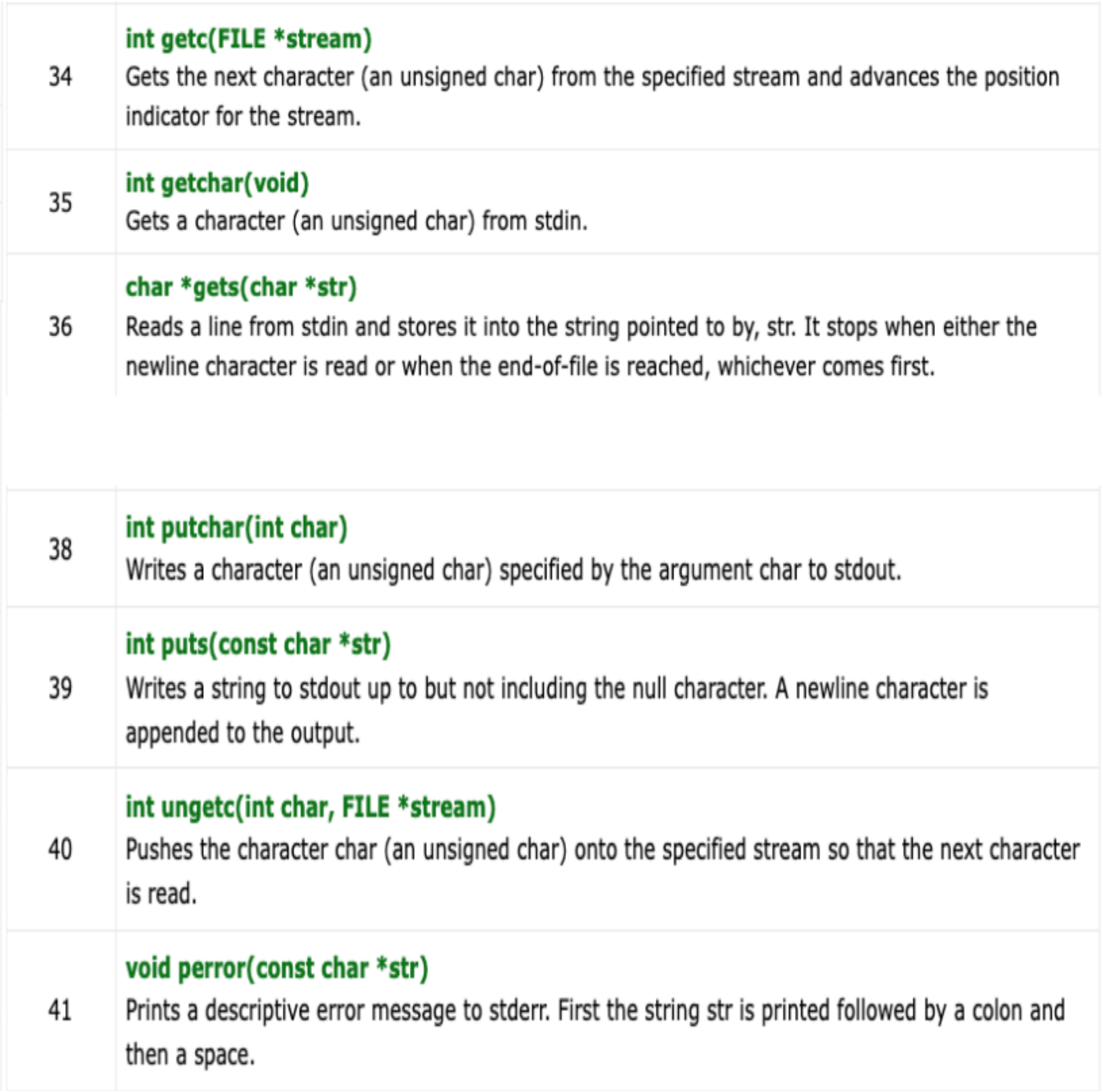
Sr.No.	Function & Description
1	**int fclose(FILE *stream)** Closes the stream. All buffers are flushed.
2	**void clearerr(FILE *stream)** Clears the end-of-file and error indicators for the given stream.
3	**int feof(FILE *stream)** Tests the end-of-file indicator for the given stream.
4	**int ferror(FILE *stream)** Tests the error indicator for the given stream.
5	**int fflush(FILE *stream)** Flushes the output buffer of a stream.
6	**int fgetpos(FILE *stream, fpos_t *pos)** Gets the current file position of the stream and writes it to pos.
7	**FILE *fopen(const char *filename, const char *mode)** Opens the filename pointed to by filename using the given mode.
8	**size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)** Reads data from the given stream into the array pointed to by ptr.
9	**FILE *freopen(const char *filename, const char *mode, FILE *stream)** Associates a new filename with the given open stream and same time closing the old file in stream.
10	**int fseek(FILE *stream, long int offset, int whence)** Sets the file position of the stream to the given offset. The argument offset signifies the number of bytes to seek from the given whence position.
11	**int fsetpos(FILE *stream, const fpos_t *pos)** Sets the file position of the given stream to the given position. The argument pos is a position given by the function fgetpos.
12	**long int ftell(FILE *stream)** Returns the current file position of the given stream.
13	**size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)** Writes data from the array pointed to by ptr to the given stream.
14	**int remove(const char *filename)** Deletes the given filename so that it is no longer accessible.
15	**int rename(const char *old_filename, const char *new_filename)** Causes the filename referred to, by old_filename to be changed to new_filename.



16	**void rewind(FILE *stream)** Sets the file position to the beginning of the file of the given stream.
17	**void setbuf(FILE *stream, char *buffer)** Defines how a stream should be buffered.
18	**int setvbuf(FILE *stream, char *buffer, int mode, size_t size)** Another function to define how a stream should be buffered.
19	**FILE *tmpfile(void)** Creates a temporary file in binary update mode (wb+).
20	**char *tmpnam(char *str)** Generates and returns a valid temporary filename which does not exist.
21	**int fprintf(FILE *stream, const char *format, ...)** Sends formatted output to a stream.
22	**int printf(const char *format, ...)** Sends formatted output to stdout.
23	**int sprintf(char *str, const char *format, ...)** Sends formatted output to a string.
24	**int vfprintf(FILE *stream, const char *format, va_list arg)** Sends formatted output to a stream using an argument list.



25	**int vprintf(const char *format, va_list arg)** Sends formatted output to stdout using an argument list.
26	**int vsprintf(char *str, const char *format, va_list arg)** Sends formatted output to a string using an argument list.
27	**int fscanf(FILE *stream, const char *format, ...)** Reads formatted input from a stream.
28	**int scanf(const char *format, ...)** Reads formatted input from stdin.
29	**int sscanf(const char *str, const char *format, ...)** Reads formatted input from a string.
30	**int fgetc(FILE *stream)** Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.
31	**char *fgets(char *str, int n, FILE *stream)** Reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.
32	**int fputc(int char, FILE *stream)** Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream.
33	**int fputs(const char *str, FILE *stream)** Writes a string to the specified stream up to but not including the null character.



34	**int getc(FILE *stream)** Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.
35	**int getchar(void)** Gets a character (an unsigned char) from stdin.
36	**char *gets(char *str)** Reads a line from stdin and stores it into the string pointed to by, str. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first.
38	**int putchar(int char)** Writes a character (an unsigned char) specified by the argument char to stdout.
39	**int puts(const char *str)** Writes a string to stdout up to but not including the null character. A newline character is appended to the output.
40	**int ungetc(int char, FILE *stream)** Pushes the character char (an unsigned char) onto the specified stream so that the next character is read.
41	**void perror(const char *str)** Prints a descriptive error message to stderr. First the string str is printed followed by a colon and then a space.