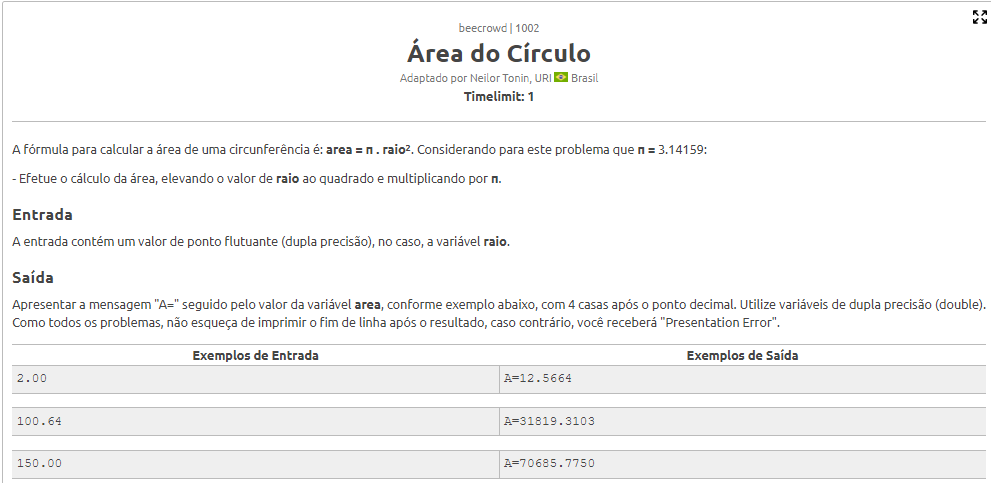
### 

### Areas

**EXEMPLOS**



#include <stdio.h>

int main() {

double raio, area, pi;

pi = 3.14159;

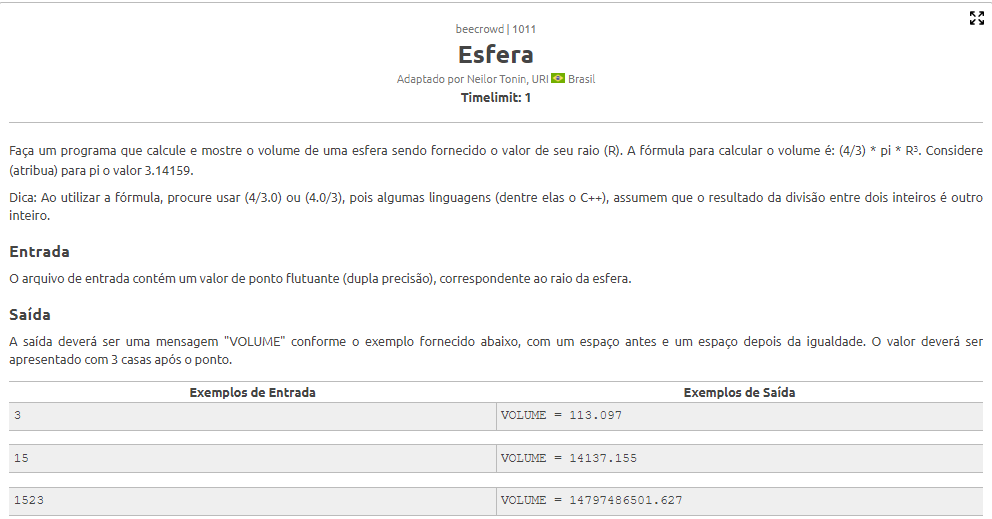
scanf("%lf", &raio);

area = (raio \* raio \* pi);

printf("A=%.4lf\n", area);

return 0;

}



#include <stdio.h>

int main() {

double volume, raio;

scanf("%lf", &raio);

volume = (4.0 / 3.0) \* 3.14159 \* (raio \* raio \* raio);

printf("VOLUME = %.3lf\n", volume);

return 0;

}



#include <stdio.h>

int main() {

int n, notas100, resto100, notas50, resto50, notas20, resto20;

int notas10, resto10, notas5, resto5, notas2, resto2, notas1;

scanf("%d", &n);

notas100 = n / 100;

resto100 = n % 100;

notas50 = resto100 / 50;

resto50 = resto100 % 50;

notas20 = resto50 / 20;

resto20 = resto50 % 20;

notas10 = resto20 / 10;

resto10 = resto20 % 10;

notas5 = resto10 / 5;

resto5 = resto10 % 5;

notas2 = resto5 / 2;

resto2 = resto5 % 2;

notas1 = resto2 / 1;

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

printf("%d nota(s) de R$ 100,00\n", notas100);

printf("%d nota(s) de R$ 50,00\n", notas50);

printf("%d nota(s) de R$ 20,00\n", notas20);

printf("%d nota(s) de R$ 10,00\n", notas10);

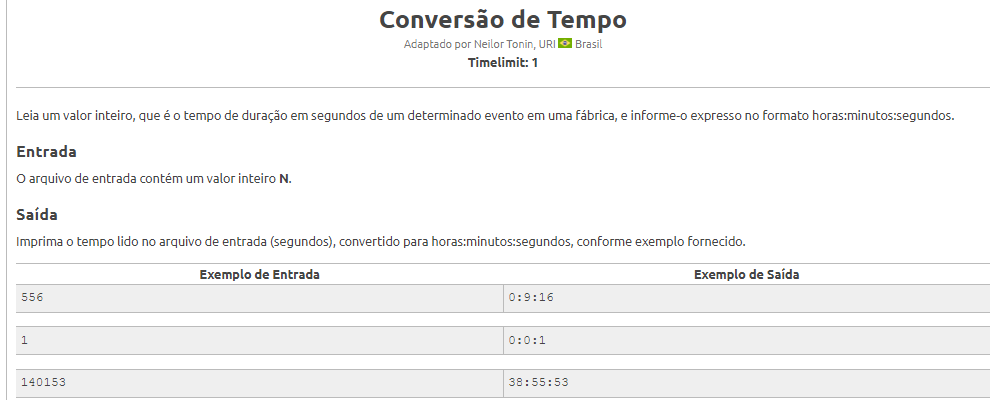
printf("%d nota(s) de R$ 5,00\n", notas5);

printf("%d nota(s) de R$ 2,00\n", notas2);

printf("%d nota(s) de R$ 1,00\n", notas1);

return 0;

}



#include <stdio.h>

int main() {

int n, hora, minuto, segundo;

scanf("%d", &n);

hora = (n / 3600);

n = (n % 3600);

minuto = (n / 60);

segundo = (n % 60);

printf("%d:%d:%d\n", hora, minuto, segundo);

}

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

int main() {

unsigned long long int cartasAt, j, delta, nivel;

int casos, i;

scanf("%d", &casos);

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

scanf("%lld", &cartasAt);

delta = (unsigned long long int)(sqrtl(1 + 12 \* (2 \* cartasAt)));

nivel = (unsigned long long int)floor(

(unsigned long long int)((delta - 1) / 6));

printf("%lld\n", nivel);

for (j = 1; cartasAt >= (j \* 2 + j - 1); j++) {

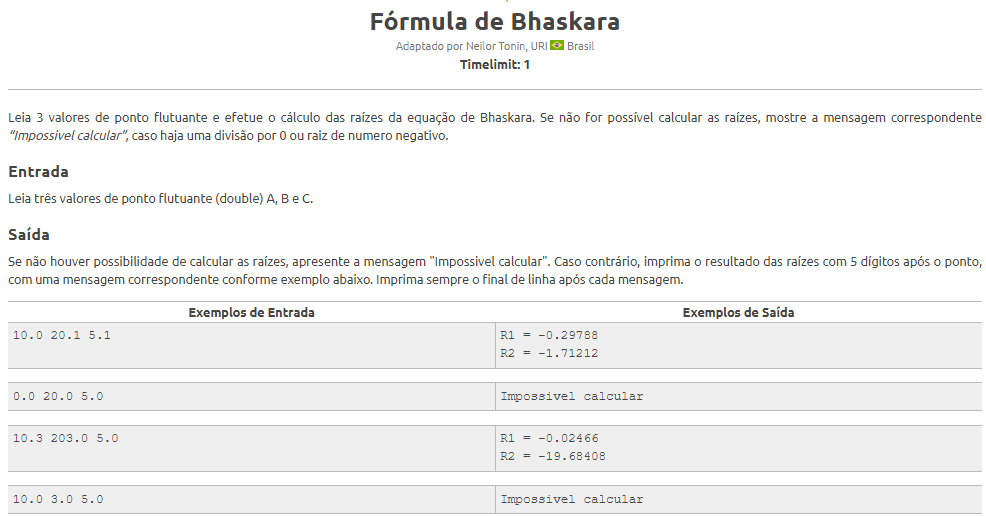
cartasAt -= (j \* 2 + j - 1);

}

printf("%lld\n", j - 1);

}

}



#include <stdio.h>

int main() {

double a, b, c, r1, r2;

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

if ((a == 0) || (b \* b -4 \* a \* c < 0)) {

printf("Impossivel calcular\n");

}

else {

r1 = (-1 \* b + sqrt(b \* b - 4 \* a \* c)) / (2 \* a);

r2 = (-1 \* b - sqrt(b \* b - 4 \* a \* c)) / (2 \* a);

printf("R1 = %.5lf\n", r1);

printf("R2 = %.5lf\n", r2);

}

return 0;

}

### **Exponenciação Binária**

**Recursiva**

long long binpow(long long a, long long b) {

if (b == 0)

return 1;

long long res = binpow(a, b / 2);

if (b % 2)

return res \* res \* a;

else

return res \* res;

}

**Sem recursão**

long long binpow(long long a, long long b) {

long long res = 1;

while (b > 0) {

if (b & 1)

res = res \* a;

a = a \* a;

b >>= 1;

}

return res;

}

**Cálculo de um expoente grande com módulo**

(a \* b ≡(a % m) \* (b % m)(% m))

long long binpow(long long a, long long b, long long m) {

a %= m;

long long res = 1;

while (b > 0) {

if (b & 1)

res = res \* a % m;

a = a \* a % m;

b >>= 1;

}

return res;

}

**Máximo divisor comum**

int gcd (int a, int b) {

return b ? gcd (b, a % b) : a;

}

**Mínimo divisor comum**

int lcm (int a, int b) {

return a / gcd(a, b) \* b;

}

**Recursividade**

#include <stdio.h>

#include <stdlib.h>

int somaRecursiva(int vetor[], int tam) {

int soma = 0;

if (tam == 0) return 0;

soma = somaRecursiva(vetor, tam - 1);

soma = soma + vetor[tam - 1];

return soma;

}

int main() {

int i, vetor[10], ret;

for (i = 0; i < 10; i++) {

vetor[i] = i + 1;

}

ret = somaRecursiva(vetor, 10);

printf("Soma dos valores do vetor: %d\n", ret);

}

**Fibonacci**

pair<int, int> fib (int n) {

if (n == 0)

return {0, 1};

auto p = fib(n >> 1);

int c = p.first \* (2 \* p.second - p.first);

int d = p.first \* p.first + p.second \* p.second;

if (n & 1)

return {d, c + d};

else

return {c, d};

}

**Números Primos**

int n;

vector<bool> is\_prime(n+1, true);

is\_prime[0] = is\_prime[1] = false;

for (int i = 2; i \* i <= n; i++) {

if (is\_prime[i]) {

for (int j = i \* i; j <= n; j += i)

is\_prime[j] = false;

}

}

**Fatorial primo mínimo**

const int N = 10000000;

vector<int> lp(N+1);

vector<int> pr;

for (int i=2; i <= N; ++i) {

if (lp[i] == 0) {

lp[i] = i;

pr.push\_back(i);

}

for (int j=0; j < (int)pr.size() && pr[j] <= lp[i] && i\*pr[j] <= N; ++j) {

lp[i \* pr[j]] = pr[j];

}

}

**Verifica se o número é primo**

using u64 = uint64\_t;

using u128 = \_\_uint128\_t;

u64 binpower(u64 base, u64 e, u64 mod) {

u64 result = 1;

base %= mod;

while (e) {

if (e & 1)

result = (u128)result \* base % mod;

base = (u128)base \* base % mod;

e >>= 1;

}

return result;

}

bool check\_composite(u64 n, u64 a, u64 d, int s) {

u64 x = binpower(a, d, n);

if (x == 1 || x == n - 1)

return false;

for (int r = 1; r < s; r++) {

x = (u128)x \* x % n;

if (x == n - 1)

return false;

}

return true;

};

bool MillerRabin(u64 n, int iter=5) { // returns true if n is probably prime, else returns false.

if (n < 4)

return n == 2 || n == 3;

int s = 0;

u64 d = n - 1;

while ((d & 1) == 0) {

d >>= 1;

s++;

}

for (int i = 0; i < iter; i++) {

int a = 2 + rand() % (n - 3);

if (check\_composite(n, a, d, s))

return false;

}

return true;

}

**Lista / Linked List**

#include <stdio.h>

#include <stdlib.h>

struct Nodo {

int info;

struct Nodo \*prox;

};

typedef struct Nodo nodo;

int tam;

void inicia(nodo \*LISTA);

int vazia(nodo \*LISTA);

nodo \*aloca();

void insereInicio(nodo \*LISTA);

void insereFim(nodo \*LISTA);

void insere(nodo \*LISTA);

void exibe(nodo \*LISTA);

void mostra(nodo \*LISTA);

nodo \*retira(nodo \*LISTA);

nodo \*retiraInicio(nodo \*LISTA);

nodo \*retiraFim(nodo \*LISTA);

void libera(nodo \*LISTA);

int main() {

nodo \*LISTA = (nodo \*)malloc(sizeof(nodo));

if (!LISTA) {

printf("Sem memoria disponivel!\n");

exit(1);

} else {

inicia(LISTA);

}

int op;

do {

printf("Escolha a opcao\n");

printf("1. Zerar lista\n");

printf("2. Exibir lista\n");

printf("3. Adicionar nodo no inicio\n");

printf("4. Adicionar nodo no final\n");

printf("5. Escolher onde inserir\n");

printf("6. Retirar do inicio\n");

printf("7. Retirar do fim\n");

printf("8. Escolher de onde tirar\n");

printf("9. Mostrar uma determinada posicao\n");

printf("0. Sair\n");

printf("Opcao: ");

scanf("%d", &op);

nodo \*tmp;

switch (op) {

case 0:

libera(LISTA);

break;

case 1:

libera(LISTA);

inicia(LISTA);

break;

case 2:

exibe(LISTA);

break;

case 3:

insereInicio(LISTA);

break;

case 4:

insereFim(LISTA);

break;

case 5:

insere(LISTA);

break;

case 6:

tmp = retiraInicio(LISTA);

printf("Retirado: %3d\n\n", tmp->info);

break;

case 7:

tmp = retiraFim(LISTA);

printf("Retirado: %3d\n\n", tmp->info);

break;

case 8:

tmp = retira(LISTA);

printf("Retirado: %3d\n\n", tmp->info);

break;

case 9:

printf("Mostra uma posicao.\n");

mostra(LISTA);

break;

default:

printf("Opcao invalida\n\n");

}

} while (op != 0);

return 0;

}

// funÃ§Ã£o para iniciar a lista

void inicia(nodo \*LISTA) {

LISTA->prox = NULL;

tam = 0;

}

// funÃ§Ã£o para verificar se a lista estÃ¡ vazia

int vazia(nodo \*LISTA) {

if (LISTA->prox == NULL)

return 1;

else

return 0;

}

// funÃ§Ã£o para alocar nodos

nodo \*aloca() {

nodo \*novo = (nodo \*)malloc(sizeof(nodo));

if (!novo) {

printf("Sem memoria disponivel!\n");

exit(1);

} else {

printf("Novo elemento: ");

scanf("%d", &novo->info);

return novo;

}

}

// funÃ§Ã£o para inserir um nodo no inÃ­cio

void insereInicio(nodo \*LISTA) {

nodo \*novo = aloca();

novo->prox = LISTA->prox;

LISTA->prox = novo;

tam++;

}

// funÃ§Ã£o para inserir no final da lista

void insereFim(nodo \*LISTA) {

nodo \*novo = aloca();

novo->prox = NULL;

if (vazia(LISTA))

LISTA->prox = novo;

else {

nodo \*tmp = LISTA->prox;

while (tmp->prox != NULL) tmp = tmp->prox;

tmp->prox = novo;

}

tam++;

}

// funÃ§Ã£o para inserir em qualquer posiÃ§Ã£o da lista existente

void insere(nodo \*LISTA) {

int pos, cont;

printf("Em que posicao, [de 1 ate %d] voce deseja inserir: ", tam);

scanf("%d", &pos);

if ((pos > 0) && (pos <= tam)) {

if (pos == 1) {

insereInicio(LISTA);

} else {

nodo \*atual = LISTA->prox;

nodo \*anterior = LISTA;

nodo \*novo = aloca();

for (cont = 1; cont < pos; cont++) {

anterior = atual;

atual = atual->prox;

}

anterior->prox = novo;

novo->prox = atual;

tam++;

}

} else {

printf("PosiÃ§Ã£o invalida\n\n");

}

}

// funÃ§Ã£o para mostrar a lista

void exibe(nodo \*LISTA) {

system("CLS");

if (vazia(LISTA)) {

printf("Lista vazia!\n\n");

} else {

nodo \*tmp;

tmp = LISTA->prox;

printf("Lista:");

while (tmp != NULL) {

printf("%5d", tmp->info);

tmp = tmp->prox;

}

printf("\n ");

int cont;

for (cont = 0; cont < tam; cont++) printf(" ^ ");

printf("\nOrdem:");

for (cont = 0; cont < tam; cont++) printf("%5d", cont + 1);

printf("\n\n");

}

}

// funÃ§Ã£o para remover nodo do inÃ­cio da lista

nodo \*retiraInicio(nodo \*LISTA) {

if (LISTA->prox == NULL) {

printf("Lista ja esta vazia\n");

return NULL;

} else {

nodo \*tmp = LISTA->prox;

LISTA->prox = tmp->prox;

tam--;

return tmp;

}

}

// funÃ§Ã£o para retirar nodos do final da lista

nodo \*retiraFim(nodo \*LISTA) {

if (LISTA->prox == NULL) {

printf("Lista ja vazia\n\n");

return NULL;

} else {

nodo \*ultimo = LISTA->prox, \*penultimo = LISTA;

while (ultimo->prox != NULL) {

penultimo = ultimo;

ultimo = ultimo->prox;

}

penultimo->prox = NULL;

tam--;

return ultimo;

}

}

// funÃ§Ã£o para retirar nodos de qualquer posiÃ§Ã£o existente

nodo \*retira(nodo \*LISTA) {

int op, cont;

printf("Que posicao, [de 1 ate %d] voce deseja retirar: ", tam);

scanf("%d", &op);

if ((op > 0) && (op <= tam)) {

if (op == 1) {

return retiraInicio(LISTA);

} else {

nodo \*atual = LISTA->prox, \*anterior = LISTA;

for (cont = 1; cont < op; cont++) {

anterior = atual;

atual = atual->prox;

}

anterior->prox = atual->prox;

tam--;

return atual;

}

} else {

printf("Elemento invalido\n\n");

return NULL;

}

}

// funÃ§Ã£o para mostrar o conteÃºdo de uma posiÃ§Ã£o da lista

void mostra(nodo \*LISTA) {

int posi, cont;

printf("O conteudo de que posicao, [de 1 ate %d] voce deseja ver: ", tam);

scanf("%d", &posi);

if ((posi > 0) && (posi <= tam)) {

nodo \*atual = LISTA->prox;

for (cont = 1; cont < posi; cont++) {

atual = atual->prox;

}

printf("O valor da posicao %d = %d.\n", posi, atual->info);

} else {

printf("Posicao invalida informada.\n");

}

}

// funÃ§Ã£o para liberar os nodos da lista

void libera(nodo \*LISTA) {

if (!vazia(LISTA)) {

nodo \*proxNodo, \*atual;

atual = LISTA->prox;

while (atual != NULL) {

proxNodo = atual->prox;

free(atual);

atual = proxNodo;

}

}

}

**Pilha / Stack**

#include <stdio.h>

#include <stdlib.h>

struct Nodo {

int dado;

struct Nodo \*anterior;

};

typedef struct Nodo nodo;

int tam;

void inicia(nodo \*TOPO);

void exibe(nodo \*TOPO);

void libera(nodo \*TOPO);

void push(nodo \*TOPO);

nodo \*pop(nodo \*TOPO);

void peek(nodo \*TOPO);

void mostraTamanho();

int main(void) {

nodo \*TOPO = (nodo \*)malloc(sizeof(nodo));

nodo \*tmp;

if (!TOPO) {

printf("Sem memoria disponivel!\n");

exit(1);

} else {

inicia(TOPO);

int op;

do {

printf("Escolha a opcao\n");

printf("1. Esvaziar PILHA\n");

printf("2. Exibir PILHA\n");

printf("3. PUSH\n");

printf("4. POP\n");

printf("5. PEEK\n");

printf("0. Sair\n");

printf("Opcao: ");

scanf("%d", &op);

switch (op) {

case 0:

libera(TOPO);

break;

case 1:

libera(TOPO);

inicia(TOPO);

break;

case 2:

exibe(TOPO);

break;

case 3:

push(TOPO);

break;

case 4:

tmp = pop(TOPO);

if (tmp != NULL) printf("Retirado: %3d\n\n", tmp->dado);

break;

case 5:

peek(TOPO);

break;

default:

printf("Comando invalido\n\n");

}

} while (op);

free(TOPO);

return 0;

}

}

void inicia(nodo \*TOPO) { // inicia a pilha

TOPO->anterior = NULL;

tam = 0;

}

int vazia(nodo \*TOPO) { // verifica se a pilha estÃ¡ vazia

if (TOPO->anterior == NULL)

return 1;

else

return 0;

}

nodo \*aloca() { // aloca um novo nodo

nodo \*novo = (nodo \*)malloc(sizeof(nodo));

if (!novo) {

printf("Sem memoria disponivel!\n");

exit(1);

} else {

printf("Novo elemento: ");

scanf("%d", &novo->dado);

return novo;

}

}

void peek(nodo \*TOPO) { // mostra o conteÃºdo do nodo no topo da pilha

if (vazia(TOPO)) {

printf("PILHA vazia!\n\n");

} else {

nodo \*temp;

temp = TOPO->anterior;

printf("TOPO: %5d\n", temp->dado);

}

}

void exibe(nodo \*TOPO) { // mostra toda a pilha

if (vazia(TOPO)) {

printf("PILHA vazia!\n\n");

} else {

nodo \*tmp;

tmp = TOPO->anterior;

int cont = 1;

printf("ORDEM\tPILHA:\n");

while (tmp != NULL) {

printf("%d >\t%5d\n", cont, tmp->dado);

tmp = tmp->anterior;

cont++;

}

}

printf("\n\n");

}

void libera(nodo \*TOPO) { // esvazia a pilha

if (!vazia(TOPO)) {

nodo \*temp;

while (TOPO->anterior != NULL) {

temp = TOPO->anterior;

TOPO->anterior = temp->anterior;

free(temp);

}

}

}

void push(nodo \*TOPO) { // empilha um nodo

nodo \*novo = aloca();

novo->anterior = TOPO->anterior;

TOPO->anterior = novo;

tam++;

}

nodo \*pop(nodo \*TOPO) { // desempilha um nodo

if (TOPO->anterior == NULL) {

printf("PILHA ja vazia\n\n");

return NULL;

} else {

nodo \*temp = TOPO->anterior;

TOPO->anterior = temp->anterior;

tam--;

return temp;

}

}

void mostraTamanho() {

if (tam == 0) {

printf("\nPilha vazia\n\n");

} else {

printf("\nA pilha tem %d nodos.\n", tam);

}

}

**Fila / Queue**

#include <stdio.h>

#include <stdlib.h>

struct Nodo {

int dado;

struct Nodo \*proximo;

};

typedef struct Nodo nodo;

int tam;

void mostraTamanho();

void inicia(nodo \*PF, nodo \*UF);

int vazia(nodo \*PF);

nodo \*aloca();

void exibe(nodo \*PF);

void libera(nodo \*PF, nodo \*UF);

void peek(nodo \*PF);

void queue(nodo \*PF, nodo \*UF);

nodo \*dequeue(nodo \*PF, nodo \*UF);

int main() {

nodo \*PF = (nodo \*)malloc(sizeof(nodo));

nodo \*UF = (nodo \*)malloc(sizeof(nodo));

inicia(PF, UF);

nodo \*temp;

if ((!PF) || (!UF)) {

printf("Sem memoria disponivel!\n");

exit(1);

} else {

int op;

do {

printf("Escolha a opcao\n");

printf("1. Zerar FILA\n");

printf("2. Exibir FILA\n");

printf("3. QUEUE\n");

printf("4. DEQUEUE\n");

printf("5. PEEK\n");

printf("6. Exibir o tamanho\n");

printf("0. Sair\n");

printf("Opcao: ");

scanf("%d", &op);

switch (op) {

case 0:

libera(PF, UF);

break;

case 1:

libera(PF, UF);

inicia(PF, UF);

break;

case 2:

exibe(PF);

break;

case 3:

queue(PF, UF);

break;

case 4:

temp = dequeue(PF, UF);

if (temp != NULL) {

printf("Nodo retirado = %d", temp->dado);

}

break;

case 5:

peek(PF);

break;

case 6:

mostraTamanho();

break;

default:

printf("Opcao invalida.\n");

}

} while (op);

free(PF);

free(UF);

}

return 0;

}

void inicia(nodo \*PF, nodo \*UF) {

PF->proximo = NULL;

UF->proximo = NULL;

tam = 0;

}

void mostraTamanho() {

if (tam == 0) {

printf("Fila vazia\n");

} else {

printf("A fila tem %d nodos.\n", tam);

}

}

int vazia(nodo \*PF) {

if (PF->proximo == NULL)

return 1;

else

return 0;

}

nodo \*aloca() {

nodo \*novo = (nodo \*)malloc(sizeof(nodo));

if (!novo) {

printf("Sem memoria disponivel!\n");

exit(1);

} else {

printf("Novo elemento: ");

scanf("%d", &novo->dado);

return novo;

}

}

void exibe(nodo \*PF) {

if (vazia(PF)) {

printf("FILA vazia!\n");

} else {

nodo \*tmp;

tmp = PF->proximo;

int cont = 1;

printf("ORDEM\tFILA:\n");

while (tmp != NULL) {

printf("%d >\t%5d\n", cont, tmp->dado);

tmp = tmp->proximo;

cont++;

}

}

printf("\n");

}

void libera(nodo \*PF, nodo \*UF) {

if (!vazia(PF)) {

nodo \*atual;

while (PF->proximo != NULL) {

atual = PF->proximo;

PF->proximo = atual->proximo;

free(atual);

}

UF->proximo = NULL;

}

tam = 0;

}

void peek(nodo \*PF) {

if (vazia(PF)) {

printf("FILA vazia!\n\n");

} else {

nodo \*temp;

temp = PF->proximo;

printf("Primeiro nodo da fila: %5d\n", temp->dado);

}

}

void queue(nodo \*PF, nodo \*UF) {

nodo \*novo = aloca();

novo->proximo = NULL;

nodo \*temp;

if (vazia(PF)) {

PF->proximo = novo;

} else {

temp = UF->proximo;

temp->proximo = novo;

}

UF->proximo = novo;

tam++;

}

nodo \*dequeue(nodo \*PF, nodo \*UF) {

if (vazia(PF)) {

printf("A FILA ja esta vazia.\n\n");

return NULL;

} else {

nodo \*temp = PF->proximo;

PF->proximo = temp->proximo;

if (vazia(PF)) {

printf("\nA FILA esvaziou.\n\n");

UF->proximo = NULL;

}

tam--;

return temp;

}

}

**Progressão Aritmética**

negrito r negrito espaço negrito igual a negrito espaço negrito a com negrito 2 negrito espaço subscrito fim do subscrito negrito – negrito espaço negrito a com negrito 1 subscrito

Onde,

r é a razão da PA;

a2 é o segundo termo;

a1 é o primeiro termo.

Sendo assim, os termos de uma progressão aritmética podem ser escritos da seguinte forma:

negrito PA negrito espaço negrito igual a negrito espaço negrito a com negrito 1 subscrito negrito vírgula negrito espaço negrito parêntese esquerdo negrito a com negrito 1 subscrito negrito mais negrito r negrito parêntese direito negrito vírgula negrito espaço negrito parêntese esquerdo negrito a com negrito 1 subscrito negrito mais negrito 2 negrito r negrito parêntese direito negrito vírgula negrito espaço negrito parêntese esquerdo negrito a com negrito 1 subscrito negrito mais negrito 3 negrito r negrito parêntese direito negrito vírgula negrito espaço negrito. negrito. negrito. negrito vírgula negrito espaço negrito parêntese recto esquerdo negrito a com negrito 1 subscrito negrito mais negrito parêntese esquerdo negrito n negrito menos negrito 1 negrito parêntese direito negrito r negrito parêntese recto direito

Note que em uma PA de n termos a fórmula do termo geral (an) da sequência é:

an = a1 + (n – 1) r

### **Soma dos termos de uma PA**

A soma dos termos de uma progressão aritmética é calculada pela fórmula:

negrito S com negrito n subscrito negrito igual a numerador negrito parêntese esquerdo negrito a com negrito 1 subscrito negrito mais negrito a com negrito n subscrito negrito parêntese direito negrito. negrito n sobre denominador negrito 2 fim da fração

Onde, n é o número de termos da sequência, a1 é o primeiro termo e an é o enésimo termo. A fórmula é útil para resolver questões em que é dado o primeiro e o último termo.

Quando um problema apresentar o primeiro termo e a razão da PA, você pode utilizar a fórmula:

negrito S com negrito n subscrito negrito igual a numerador negrito n negrito. negrito parêntese recto esquerdo negrito 2 negrito a com negrito 1 subscrito negrito mais negrito parêntese esquerdo negrito n negrito menos negrito 1 negrito parêntese direito negrito r negrito parêntese recto direito sobre denominador negrito 2 fim da fração

**Progressão Geométrica**

negrito q negrito espaço negrito igual a negrito espaço numerador negrito a com negrito 2 subscrito sobre denominador negrito a com negrito 1 subscrito negrito espaço fim da fração

Onde,

q é a razão da PG;

a2 é o segundo termo;

a1 é o primeiro termo.

Uma progressão geométrica de n termos pode ser representada da seguinte forma:

negrito a com negrito 1 subscrito negrito vírgula negrito espaço negrito a com negrito 1 subscrito negrito q negrito vírgula negrito espaço negrito a com negrito 1 subscrito negrito q à potência de negrito 2 negrito vírgula negrito espaço negrito a com negrito 1 subscrito negrito q à potência de negrito 3 negrito vírgula negrito espaço negrito a com negrito 1 subscrito negrito q à potência de negrito 4 negrito vírgula negrito espaço negrito. negrito. negrito. negrito vírgula negrito espaço negrito a com negrito 1 subscrito negrito. negrito q à potência de negrito parêntese esquerdo negrito n negrito menos negrito 1 negrito parêntese direito fim do exponencial

### **Soma dos termos de uma PG**

A soma dos termos de uma progressão geométrica é calculada pela fórmula:

negrito S com negrito n subscrito negrito igual a numerador negrito a com negrito 1 subscrito negrito parêntese esquerdo negrito q à potência de negrito n negrito menos negrito 1 negrito parêntese direito sobre denominador negrito q negrito menos negrito 1 fim da fração

Sendo a1 o primeiro termo, q a razão comum e n o número de termos.

Se a razão da PG for menor que 1, então utilizaremos a fórmula a seguir para determinar a soma dos termos.

negrito S com negrito n subscrito negrito igual a numerador negrito a com negrito 1 subscrito negrito parêntese esquerdo negrito 1 negrito espaço negrito menos negrito espaço negrito q à potência de negrito n negrito parêntese direito sobre denominador negrito 1 negrito espaço negrito menos negrito espaço negrito q fim da fração

Essas fórmulas são utilizadas para uma PG finita. Caso a soma pedida seja de uma PG infinita com 0 < q < 1 , a fórmula utilizada é:

negrito S com negrito infinito subscrito negrito igual a numerador negrito a com negrito 1 subscrito sobre denominador negrito 1 negrito espaço negrito menos negrito espaço negrito q fim da fração

**Análise Combinatória**

**Arranjos**

Nos **arranjos**, os agrupamentos dos elementos dependem da ordem e da natureza dos mesmos.

Para obter o arranjo simples de *n* elementos tomados, p a p (p ≤ n), utiliza-se a seguinte expressão:



**Permutações**

As **permutações** são agrupamentos ordenados, onde o número de elementos (n) do agrupamento é igual ao número de elementos disponíveis.

Note que a permutação é um caso especial de arranjo, quando o número de elementos é igual ao número de agrupamentos. Desta maneira, o denominador na fórmula do arranjo é igual a 1 na permutação.

Assim a permutação é expressa pela fórmula:



**Combinações**

As **combinações** são subconjuntos em que a ordem dos elementos não é importante, entretanto, são caracterizadas pela natureza dos mesmos.

Assim, para calcular uma combinação simples de *n* elementos tomados p a p (p ≤ n), utiliza-se a seguinte expressão:

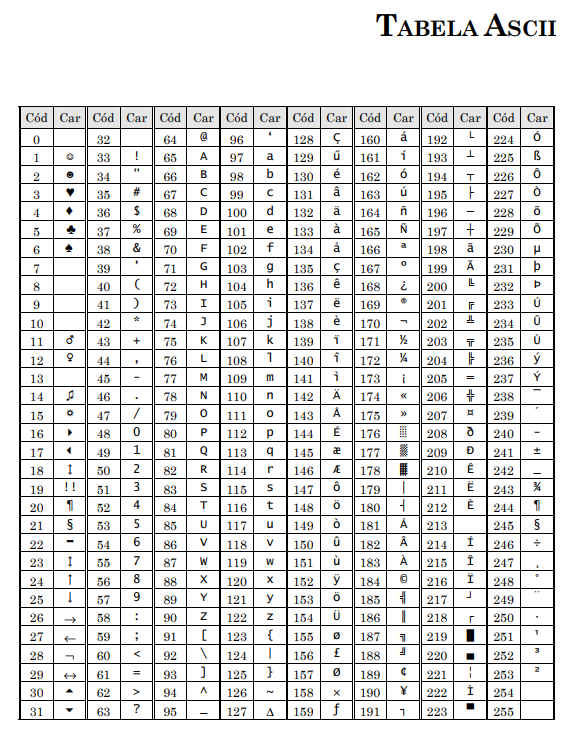


* Combinação com repetição são todos os agrupamentos não ordenados, escolhendo *k* elementos de um conjunto com *n* elementos, podendo ter repetição.
* Para calcular todas as combinações com repetição, utilizamos a fórmula:

Fórmula da combinação com repetição.

* Podemos relacionar a combinação com repetição com uma combinação simples pela fórmula:

CRn,k= Cn+k-1,k



## **Bibliotecas**

**<CTYPE.H>**

**[int isalnum (int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "isalnum)**

RETORNA:

Um valor diferente de zero caso o caractere seja alfanumérico ou 0 caso contrário.

RECEBE:

um argumento inteiro, cujo valor seja um caractere (ou seja, possa ser representado como um unsigned char) ou o valor EOF.

### **[int isalpha(int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "isalpha)**

RETORNA:

1 caso seja uma letra do alfabeto maiúscula, 2 caso seja minúscula, 0 caso não esteja no alfabeto.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que isalpha() receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

### **[int iscntrl(int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "iscntrl)**

RETORNA:

um valor diferente de zero caso seja um caractere de controle (códigos de 0 a 31 na tabela ASCII) ou 0 caso contrário.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que iscntrl() receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

### **[int isdigit(int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "isdigit)**

RETORNA:

um valor diferente de zero quando o caractere recebido é um dígito de 0 a 9 ou 0 caso contrário.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que isdigit() receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

### **[int isgraph(int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "isdigit)**

RETORNA:

um valor diferente de zero caso seja um caractere que possa ser imprimido (exceto o espaço) ou 0 caso contrário.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que isgraph() receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

### **[int islower(int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "islower)**

RETORNA:

um valor diferente de zero caso o caractere seja uma letra minúscula ou zero caso contrário.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que islower () receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

### **[int isprint(int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "isprint)**

RETORNA:

um valor diferente de zero caso seja um caractere imprimível (incluindo o espaço) ou zero caso contrário.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que isprint() receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

### **[int ispunct(int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "ispunct)**

RETORNA:

um valor diferente de zero caso seja um sinal de pontuação (qualquer caractere imprimível que não seja espaço ou alfanumérico) ou 0 caso contrário.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que ispunct() receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

### **[int isspace(int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "isspace)**

RETORNA:

um valor diferente de zero caso seja um espaço ou um caractere de mudança de linha (‘\n’), de retorno de carro (‘\r’), de mudança de página (‘\f’), ou de tabulações horizontal (‘\t’) ou vertical (‘\v’) ou 0 caso contrário.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que isspace() receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

### **[int isupper(int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "isupper)**

RETORNA:

um valor diferente de um caso seja uma letra maiúscula (A a Z) ou 0 caso contrário.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que isupper() receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

### **[int isxdigit(int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "isxdigit)**

RETORNA:

um valor diferente de zero caso o caractere seja um dígito de um número hexadecimal ou 0 caso contrário.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que isxdigit() receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

### **[int tolower(int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "tolower)**

RETORNA:

caso o argumento da função seja uma letra maiúscula, retorna sua versão em letra minúscula; caso contrário o próprio argumento é retornado sem modificações.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que tolower() receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

### **[int toupper (int c)](https://petbcc.ufscar.br/ctypefuncoes/" \l "toupper)**

RETORNA:

caso o argumento da função seja uma letra minúscula, retorna sua versão em maiúscula; caso contrário o próprio argumento é retornado sem modificações.

RECEBE:

um único argumento (caractere) que será convertido para inteiro. Mesmo que toupper() receba um inteiro como argumento, o caractere é passado para a função. Internamente, o caractere é convertido em seu valor ASCII para a verificação.

# **<MATH.H>**

### **[double acos(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "acos)**

RETORNA:

o cosseno de um ângulo; isto é, um número entre -1 e 1, tal que -1<=cosseno<=1

RECEBE:

o ângulo, em radianos, do cosseno que foi informado como parâmetro ou NaN se o argumento estiver fora do domínio de entrada.

### **[double asin(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "asin)**

RETORNA:

o seno de um ângulo, tal que, -1 <= seno <= 1.

RECEBE:

o ângulo, em radianos, cujo seno foi informado como parâmetro ou NaN se o argumento estiver fora do domínio de entrada.

### **[double atan(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "atan)**

RETORNA:

a tangente de um ângulo.

RECEBE:

o ângulo, em radianos, cuja tangente foi informado como parâmetro.

### **[double atan2(double x, double y)](https://petbcc.ufscar.br/mathfuncoes/" \l "atan2)**

RETORNA:

o arco tangente de y / x no intervalo de π a π radianos, dos valores informados como parâmetros.

RECEBE:

dois argumentos do tipo double: x, y. Elas são representações de um ponto no plano cartesiano.

### **[double cos(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "cos)**

RETORNA:

o valor do cosseno do ângulo que foi informado como parâmetro.

RECEBE:

um ângulo em radianos.

### **[double sin(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "sin)**

RETORNA:

o valor do seno do ângulo que foi informado como parâmetro.

RECEBE:

um ângulo em radianos.

### **[double tan(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "tan)**

RETORNA:

o valor da tangente do ângulo que foi informado como parâmetro.

RECEBE:

um ângulo em radianos.

### **[double cosh(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "cosh)**

RETORNA:

o cosseno hiperbólico do valor informado como parâmetro.

RECEBE:

o valor de um ângulo em radianos.

### **[double sinh(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "sinh)**

RETORNA:

o seno hiperbólico do valor informado como parâmetro.

RECEBE:

o valor de um ângulo em radianos.

### **[double tanh(double x, double y)](https://petbcc.ufscar.br/mathfuncoes/" \l "tanh)**

RETORNA:

a tangente hiperbólica do valor informado como parâmetro.

RECEBE:

o valor de um ângulo em radianos.

### **[double exp(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "exp)**

RETORNA:

o número de Euler (~2.71) elevado a x: ex = exp(x), informado como parâmetro.

RECEBE:

um número x do tipo double.

### **[double log(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "log)**

RETORNA:

o valor de ln x, ou seja, o logaritmo de x na base e.

RECEBE:

um valor x do tipo double.

### **[double log10(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "log10)**

RETORNA:

o valor de ln x, ou seja, o logaritmo de x na base e.

RECEBE:

um valor x do tipo double.

### **[double modf(double x, double \*p)](https://petbcc.ufscar.br/mathfuncoes/" \l "modf)**

RETORNA:

a parte decimal de x no valor da função e a parte inteira de x no parâmetro p.

RECEBE:

o valor double x e o endereço de um double p.

### **[double pow(double x, double y)](https://petbcc.ufscar.br/mathfuncoes/" \l "pow)**

RETORNA:

x elevado a y.

RECEBE:

dois valores do tipo double, x e y, em que x será a base e y será o expoente.

### **[double sqrt(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "sqrt)**

RETORNA:

a raiz quadrada de x ou NaN se x for negativo.

RECEBE:

x, um valor do tipo double.

### **[double ceil(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "ceil)**

RETORNA:

o menor inteiro maior ou igual a x.

RECEBE:

x, um valor do tipo double.

### **[double fabs(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "fabs)**

RETORNA:

o módulo de x.

RECEBE:

x, um valor do tipo double.

### **[double floor(double x)](https://petbcc.ufscar.br/mathfuncoes/" \l "floor)**

RETORNA:

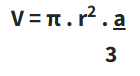
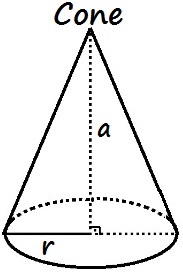
o maior valor inteiro menor ou igual a x.

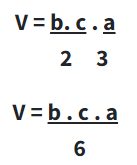
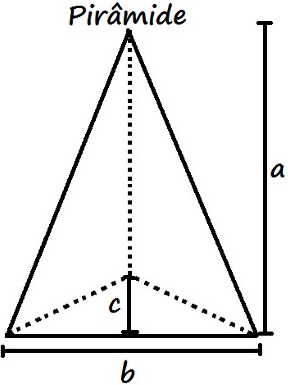
RECEBE:

x, um valor do tipo double.

**Volume**

Prisma, cilindro: *área da base* x *altura*

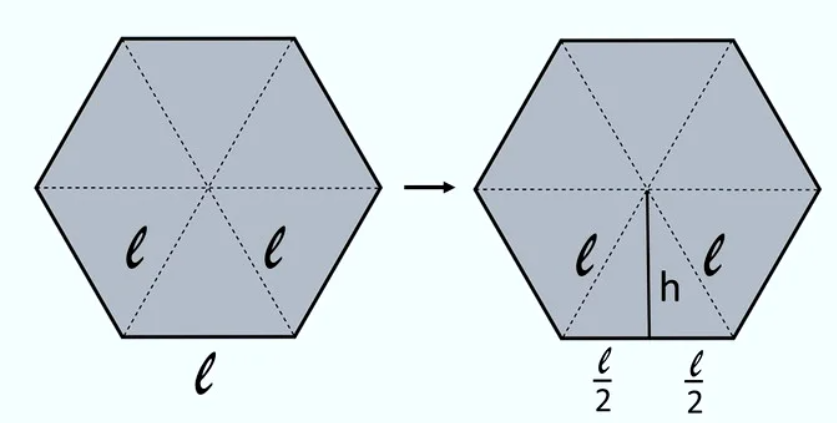
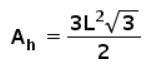




(sempre área da base x altura dividido por 3)

Esfera: V = 4/3 πr³

Área do hexágono



Área do pentágono

