PILAS

struct nodo{

int dato;

nodo \*siguiente;

};

void agregarpila(nodo \*&pila, int dato);

void sacarpila(nodo \*&pila, int &dato);

int main(int argc, char \*argv[]) {

nodo \*pila = NULL;

int dato;

char decision;

while(decision != 'N'){

cout << "Ingrese un numero: " << endl << ">";

cin >> dato;

agregarpila(pila, dato);

cout << "Va a ingresar otro dato a la pila S/N: " << endl << ">";

cin >> decision;

}

cout << "Sacando elementos de pila: " << endl << ">";

while (pila != NULL){ /// MIENTRAS NO SEA EL FINAL DE LA PILA

sacarpila(pila, dato);

if (pila != NULL){

cout << dato << " , ";

} else {

cout << dato << ".";

}

}

return 0;

}

void agregarpila(nodo \*&pila, int n){

nodo \*nuevo\_nodo = new nodo(); /// CREAR EL ESPACIO EN MEMORIA PARA ALMACENAR UN

nuevo\_nodo -> dato = n; /// CARGAR EL VALOR DENTRO DEL NODO (DATO)

nuevo\_nodo -> siguiente = pila; /// CARGAR EL PUNTERO PILA DENTRO DEL NODO

pila = nuevo\_nodo; /// ASIGNAR EL NUEVO NODO A LA PILA

}

void sacarpila(nodo \*&pila, int &n){

nodo \*aux = pila;

n = aux -> dato;

pila = aux -> siguiente;

delete aux;

}

COLAS

#include <iostream>

#include <iomanip>

#include <stdlib.h>

using namespace std;

struct nodo{

char dato;

nodo \*siguiente;

};

void insertarcola(nodo \*&frente, nodo \*&fin, char n);

bool colavacia(nodo \*frente);

void suprimircola(nodo \*&frente, nodo \*&fin, char &n);

int main(int argc, char \*argv[]) {

nodo \*frente = NULL;

nodo \*fin = NULL;

int opcion;

char dato;

while (opcion != 3){

cout << "------------------ MENU ------------------" << endl;

cout << "1)Insertar caracter a la cola: " << endl;

cout << "2)Mostrar todos los elementos de la cola" << endl;

cout << "3)SALIR" << endl << ">";

cin >> opcion;

switch (opcion){

case 1: cout << "Ingrese el caracter para agregar a la cola: " << endl << ">";

cin >> dato;

insertarcola(frente, fin, dato);

break;

case 2: cout << "Sacando los elementos de la cola: " << endl;

while(frente != NULL){

suprimircola(frente, fin, dato);

if (frente != NULL){

cout << dato << ",";

} else {

cout << dato << "." << endl;

}

}

break;

case 3: break;

}

}

return 0;

}

/// FUNCION INSERTAR ELEMENTOS

void insertarcola(nodo \*&frente, nodo \*&fin, char n){

nodo \*nuevo\_nodo = new nodo();

nuevo\_nodo -> dato = n;

nuevo\_nodo -> siguiente = NULL;

if (colavacia(frente)){

frente = nuevo\_nodo;

} else {

fin -> siguiente = nuevo\_nodo;

}

fin = nuevo\_nodo;

}

/// DETERMINAR SI COLA ESTA VACIA

bool colavacia(nodo \*frente){

return (frente == NULL)? true : false;

}

/// SACAR ELEMENTOS DE LA COLA

void suprimircola(nodo \*&frente, nodo \*&fin, char &n){

n = frente -> dato;

nodo \*aux = frente;

if(frente == fin){

frente = NULL;

fin = NULL;

} else{

frente = frente -> siguiente;

}

delete aux;

}

LISTAS

struct nodo{

int dato;

nodo \*siguiente;

};

void insertarlista(nodo \*&lista, int n);

void mostrarlista(nodo \*lista);

void buscarlista(nodo \*lista, int n);

void eliminarnodo(nodo \*&lista, int n);

void eliminarlista(nodo \*&lista, int &n);

int main(int argc, char \*argv[]) {

nodo \*lista = NULL;

int dato;

int opcion;

do {

cout << "--------- MENU ---------" << endl;

cout << "1)Insertar elemento a la lista" << endl;

cout << "2)Mostrar los elementos de la lista" << endl;

cout << "3)Buscar elemento en la lista" << endl;

cout << "4)Eliminar un nodo de la lista" << endl;

cout << "5)Vaciar lista/Eliminar lista" << endl;

cout << "6)SALIR" << endl;

cin >> opcion;

switch(opcion){

case 1: cout << "Digite un numero: " << endl << ">";

cin >> dato;

insertarlista(lista, dato);

break;

case 2: mostrarlista(lista);

cout << endl;

break;

case 3: cout << "Ingrese un numero a buscar: " << endl;

cin >> dato;

buscarlista(lista, dato);

cout << endl;

break;

case 4: cout << "Digite elemento que desea eliminar: " << endl;

cin >> dato;

eliminarnodo(lista, dato);

cout << endl;

break;

case 5: while (lista != NULL){ //Mientras no sea el final de la lista

eliminarlista(lista, dato);

cout << dato << " ->";

}

cout << endl;

break;

case 6:break;

}

} while (opcion != 6);

return 0;

}

void insertarlista(nodo \*&lista, int n){

nodo \*nuevo\_nodo = new nodo();

nuevo\_nodo -> dato = n;

nodo \*aux1 = lista;

nodo \*aux2;

while ((aux1 != NULL) && (aux1 -> dato < n)){

aux2 = aux1;

aux1 = aux1 -> siguiente;

}

if (lista == aux1){

lista = nuevo\_nodo;

} else {

aux2 -> siguiente = nuevo\_nodo;

}

nuevo\_nodo -> siguiente = aux1;

}

void mostrarlista(nodo \*lista){

nodo \*actual = new nodo();

actual = lista;

while (actual != NULL){

cout << actual -> dato << " -> ";

actual = actual -> siguiente;

}

}

void buscarlista(nodo \*lista, int n){

bool band = false;

nodo \*actual = new nodo();

actual = lista;

while ((actual != NULL) && (actual -> dato <= n)){

if (actual -> dato == n){

band = true;

}

actual = actual -> siguiente;

}

if (band == true){

cout << "Elemento " << n << " SI a sido encontrado en la lista" << endl;

} else {

cout << "Elemento " << n << " No a sido encontrado en la lista" << endl;

}

}

void eliminarnodo(nodo \*&lista, int n){

//Preguntar si lista esta vacia

if (lista != NULL){

nodo \*aux\_borrar;

nodo \*anterior = NULL;

aux\_borrar = lista;

//Recorrer la lista

while ((aux\_borrar != NULL) && (aux\_borrar -> dato != n)){

anterior = aux\_borrar;

aux\_borrar = aux\_borrar -> siguiente;

}

//Elemento no encontrado

if (aux\_borrar == NULL){

cout << "ELEMENTO NO ENCONTRADO" << endl;

} else if (anterior == NULL){

lista = lista -> siguiente;

delete aux\_borrar;

} else { // El elemento esta en la lista, pero no es el primer nodo

anterior -> siguiente = aux\_borrar -> siguiente;

delete aux\_borrar;

}

}

}

// Vaciar lista

void eliminarlista(nodo \*&lista, int &n){

nodo \*aux = lista;

n = aux -> dato;

lista = aux -> siguiente;

delete aux;

}

DOBLEMENTE ENLAZADA

struct nodo{

string nombre;

string tutor;

int edad;

float distancia;

string telefono;

string salita;

nodo \*siguiente;

nodo \*anterior;

};

struct nodopila {

nodo \*nino;

nodopila \*siguiente;

};

/\*Esta función agrega un nuevo nodo al inicio de la lista. Crea un nuevo nodo, establece su dato,

siguiente (apuntando al antiguo primer nodo de la lista), y anterior (nullptr ya que será el primer nodo).

Si la lista no está vacía, actualiza el puntero anterior del antiguo primer nodo para que apunte al nuevo nodo.

Finalmente, actualiza el puntero de la lista para que apunte al nuevo nodo. \*/

void agregarinicio(nodo \*&lista, string nombre, string tutor, int edad, float distancia, string telefono);

void agregarpila(nodopila \*&pila, nodo \*nino);

void distribuirninos(nodo \*&lista, nodopila \*&salitaverde, nodopila \*&salitanegra);

void imprimirlista(nodo \*lista);

void imprimirpila(nodopila \*pila);

int main(int argc, char \*argv[]) {

nodo \*lista = NULL;

nodopila \*salitaverde = NULL;

nodopila \*salitanegra = NULL;

agregarinicio(lista, "Juan Perez", "Maria Lopez", 3, 3.5, "123456789");

agregarinicio(lista, "Ana Garcia", "Carlos Garcia", 5, 2.5, "987654321");

agregarinicio(lista, "Luis Martinez", "Juana Martinez", 2, 1.0, "111222333");

agregarinicio(lista, "Sofia Hernandez", "Laura Hernandez", 4, 3.0, "444555666");

// Imprimir la lista

cout << "Lista en orden: ";

imprimirlista(lista);

// Distribuir los ninos en las pilas

distribuirninos(lista, salitaverde, salitanegra);

// Imprimir las pilas

cout << "Salita Verde:" << endl;

imprimirpila(salitaverde);

cout << "Salita Negra:" << endl;

imprimirpila(salitanegra);

// Imprimir la lista después de la distribución

cout << "Lista después de la distribucion:" << endl;

imprimirlista(lista);

return 0;

}

/\*Esta función agrega un nuevo nodo al inicio de la lista. Crea un nuevo nodo, establece su dato,

siguiente (apuntando al antiguo primer nodo de la lista), y anterior (nullptr ya que será el primer nodo).

Si la lista no está vacía, actualiza el puntero anterior del antiguo primer nodo para que apunte al nuevo nodo.

Finalmente, actualiza el puntero de la lista para que apunte al nuevo nodo. \*/

void agregarinicio(nodo \*&lista, string nombre, string tutor, int edad, float distancia, string telefono){

nodo \*nuevo\_nodo = new nodo();

nuevo\_nodo -> nombre = nombre;

nuevo\_nodo -> tutor = tutor;

nuevo\_nodo -> edad = edad;

nuevo\_nodo -> distancia = distancia;

nuevo\_nodo -> telefono = telefono;

nuevo\_nodo -> salita = "";

nuevo\_nodo -> siguiente = lista;

nuevo\_nodo -> anterior = NULL;

if (lista != NULL){

lista -> anterior = nuevo\_nodo;

}

lista = nuevo\_nodo;

}

void agregarpila(nodopila \*&pila, nodo \*nino) {

nodopila \*nuevonodo = new nodopila();

nuevonodo -> nino = nino;

nuevonodo -> siguiente = pila;

pila = nuevonodo;

}

void distribuirninos(nodo \*&lista, nodopila \*&salitaverde, nodopila \*&salitanegra) {

nodo \*actual = lista;

while (actual != NULL) {

nodo \*aux = actual;

actual = actual -> siguiente;

bool eliminar = false;

if ((aux -> edad < 4 && aux -> edad > 2 && aux -> distancia > 2)) {

aux -> salita = "Verde";

agregarpila(salitaverde, aux);

eliminar = true;

} else if (aux -> edad >= 4 && aux -> edad < 6 && aux -> distancia > 2) {

aux -> salita = "Negra";

agregarpila(salitanegra, aux);

eliminar = true;

}

// Eliminar el nodo de la lista

if (eliminar) {

if (aux -> anterior != NULL) {

aux -> anterior -> siguiente = aux -> siguiente;

} else {

lista = aux -> siguiente;

}

if (aux->siguiente != NULL) {

aux -> siguiente -> anterior = aux -> anterior;

}

}

}

}

void imprimirlista(nodo \*lista){

nodo \*aux = lista;

while (aux != NULL) {

cout << "Nombre: " << aux -> nombre << ", Tutor: " << aux -> tutor << ", Edad: " << aux -> edad << ", Distancia: " << aux -> distancia

<< " km, Telefono: " << aux -> telefono << ", Salita: " << aux -> salita << endl;

aux = aux -> siguiente;

}

cout << endl;

}

void imprimirpila(nodopila \*pila) {

nodopila \*aux = pila;

while (aux != NULL){

nodo \*nino = aux -> nino;

cout << nino -> nombre << " (" << nino -> edad<< " años, " << nino -> distancia << " km)" << endl;

aux = aux -> siguiente;

}

cout << endl;

}

ARBOLES

struct nodo{

int dato;

nodo \*der;

nodo \*izq;

nodo \*padre;

};

nodo \*crearnodo(int n, nodo \*padre);

void insertarnodo(nodo \*&arbol, int n, nodo \*padre);

void mostrararbol(nodo \*arbol, int contador);

bool busqueda(nodo \*arbol, int n);

void preorden(nodo \*arbol);

void inorden(nodo \*arbol);

void postorden(nodo \*arbol);

void eliminar(nodo \*arbol, int n);

void eliminarnodo(nodo \*nodoeliminar);

nodo \*minimo(nodo \*arbol);

void reemplazar(nodo \*arbol, nodo \*nuevonodo);

void destruirnodo(nodo \*nodo);

int main(int argc, char \*argv[]) {

nodo \*arbol = NULL;

int dato, opcion, contador = 0;

do {

cout << "MENU" << endl;

cout << "1)INSERTAR UN NUEVO NODO" << endl;

cout << "2)MOSTRAR EL ARBOL COMPLETO" << endl;

cout << "3)BUSCAR UN ELEMENTO EN EL ARBOL" << endl;

cout << "4)RECORRER ARBOL EN PREORDEN" << endl;

cout << "5)RECORRER EL ARBOL EN INORDEN" << endl;

cout << "6)RECORRER EL ARBOL EN POSTORDEN" << endl;

cout << "7)ELIMINAR UN NODO DEL ARBOL" << endl;

cout << "8)SALIR" << endl << ">";

cin >> opcion;

switch (opcion){

case 1: cout << "Digite un numero: " << endl << ">";

cin >> dato;

insertarnodo(arbol, dato, NULL);

cout << endl;

break;

case 2: cout << "Mostrando el arbol completo: " << endl;

mostrararbol(arbol, contador);

cout << endl;

break;

case 3: cout << "Ingrese el elemento a buscar: " << endl << ">";

cin >> dato;

if (busqueda(arbol, dato) == true){

cout << "ELEMENTO " << dato << " A SIDO ENCONTRADO EN EL ARBOL" << endl;

} else {

cout << "ELEMENTO NO ENCONTRADO" << endl;

}

case 4: cout << "RECORRIDO EN PREORDEN: " << endl;

preorden(arbol);

cout << endl;

break;

case 5: cout << "RECORRIDO EN INORDEN: " << endl;

inorden(arbol);

cout << endl;

break;

case 6: cout << "RECORRIDO EN POSTORDEN: " << endl;

postorden(arbol);

cout << endl;

break;

case 7: cout << "Digite el nodo que quiere eliminar: " << endl << ">";

cin >> dato;

eliminar(arbol, dato);

cout << endl;

break;

case 8: break;

}

} while (opcion != 8);

return 0;

}

// FUNCION PARA CREAR NUEVO NODO

nodo \*crearnodo(int n, nodo \*padre){

nodo \*nuevo\_nodo = new nodo();

nuevo\_nodo -> dato = n;

nuevo\_nodo -> der = NULL;

nuevo\_nodo -> izq = NULL;

nuevo\_nodo -> padre = padre;

return nuevo\_nodo;

}

// FUNCION PARA INSERTAS ELEMENTOS EN EL ARBOL

void insertarnodo(nodo \*&arbol, int n, nodo \*padre){

if (arbol == NULL){ // SI EL ARBOL ESTA VACIO

nodo \*nuevo\_nodo = crearnodo(n, padre);

arbol = nuevo\_nodo;

} else { // SI EL ARBOL TIENE UN NODO O MAS

int valoraiz = arbol -> dato; //OBTENEMOS VALOR DE LA RAIZ

if (n < valoraiz){ // SI EL ELEMENTO ES MENOR A LA RAIZ A LA IZQUIERDA PA

insertarnodo(arbol -> izq, n, arbol);

} else {

insertarnodo(arbol -> der, n, arbol);

}

}

}

//MOSTRANDO ARBOL COMPLETO

void mostrararbol(nodo \*arbol, int contador){

if (arbol == NULL){ // ARBOL VACIO?

return;

} else {

mostrararbol(arbol -> der, contador+1);

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

cout << " ";

}

cout << arbol -> dato << endl;

mostrararbol(arbol -> izq, contador+1);

}

}

//FUNCION PARA BUSCAR ELEMENTO EN EL ARBOL

bool busqueda(nodo \*arbol, int n){

if(arbol == NULL){ //SI EL ARBOL ESTA VACIO

return false;

} else if(arbol -> dato == n){ // SI EL NODO ES IGUAL AL ELEMENTO

return true;

} else if(n < arbol -> dato){

return busqueda(arbol -> izq, n);

} else {

return busqueda(arbol -> der, n);

}

}

void preorden(nodo \*arbol){

if (arbol == NULL){ // ARBOL VACIO?

return;

} else{

cout << arbol -> dato << " - "; //RAIZ

preorden(arbol -> izq);

preorden(arbol -> der);

}

}

void inorden(nodo \*arbol){

if (arbol == NULL){

return;

} else {

inorden(arbol -> izq);

cout << arbol -> dato << " - ";

inorden(arbol -> der);

}

}

void postorden(nodo \*arbol){

if (arbol == NULL){

return;

} else {

postorden(arbol -> izq);

postorden(arbol -> der);

cout << arbol -> dato << " - ";

}

}

void eliminar(nodo \*arbol, int n){

if (arbol == NULL){ //ARBOL VACIO

return;

} else if (n < arbol -> dato){ // SI EL VALOR ES MENOR BUSCAR POR LA IZQUIERDA

eliminar(arbol -> izq, n);

} else if (n > arbol -> dato){ // SI EL VALOR ES MAYOR BUSCA POR LA DERECHA

eliminar(arbol -> der, n);

} else { // SI YA LO ENCONTRASTE

eliminarnodo(arbol);

}

}

// FUNCION ELIMINAR NODO

void eliminarnodo(nodo \*nodoeliminar){

if (nodoeliminar -> izq && nodoeliminar -> der){ //SI EL NODO TIENE HIJO IZQ Y DER

nodo \*menor = minimo(nodoeliminar -> der);

nodoeliminar -> dato = menor -> dato;

eliminarnodo(menor);

}else if (nodoeliminar -> izq){ // SI TIENE HIJO IZQUIERDO

reemplazar(nodoeliminar, nodoeliminar -> izq);

destruirnodo(nodoeliminar);

} else if(nodoeliminar -> der){ // SI TIENE UN HIJO DER

reemplazar(nodoeliminar, nodoeliminar -> der);

destruirnodo(nodoeliminar);

} else { //NO TIENE HIJOS

reemplazar(nodoeliminar, NULL);

destruirnodo(nodoeliminar);

}

}

// FUNCION PARA DETERMINAR EL NODO MAS IZQ POSIBLE

nodo \*minimo(nodo \*arbol){

if (arbol == NULL){ // si el arbol esta vacio

return NULL; //retorna nulo

}

if (arbol -> izq){ // SI TIENE HIJO IZQ

return minimo(arbol -> izq); // BUSCAMOS LA PARTE MAS IZQ POSIBLE

} else { // SI NO TIENE HIJO IZQUIERDO

return arbol; // RETORNAMOS EL MISMO NODO

}

}

// FUNCION PARA REEMPLAZAR DOS NODOS

void reemplazar(nodo \*arbol, nodo \*nuevonodo){

if (arbol -> padre){

// arbol -> padre asignarle su nuevo hijo

if (arbol -> dato == arbol -> padre -> izq -> dato){

arbol -> padre -> izq = nuevonodo;

} else if (arbol -> dato == arbol -> padre -> der -> dato){

arbol -> padre -> der = nuevonodo;

}

}

if (nuevonodo){

// asignarle su nuevo padre

nuevonodo -> padre = arbol -> padre;

}

}

// FUNCION PARA DESTRUIR UN NODO

void destruirnodo(nodo \*nodo){

nodo -> izq = NULL;

nodo -> der = NULL;

delete nodo;

}

// ## CONTAR NODOS HOJA ##

int contarNodosHoja(nodeArbol\* arbol){

int hojas = 0;

if (arbol != nullptr){

nodoPila\* pila = nullptr;

push(pila,arbol);

while(!isEmpty(pila)){

arbol = pop(pila);

if (arbol->izq == nullptr && arbol->der == nullptr)

hojas++;

if (arbol->izq != nullptr)

push(pila,arbol->izq);

if (arbol->der != nullptr)

push(pila,arbol->der);

}

}

return hojas;

}

void mostrarNodosHoja(nodeArbol\* arbol){

if (arbol != nullptr){

nodoPila\* pila = nullptr;

push(pila,arbol);

while(!isEmpty(pila)){

arbol = pop(pila);

if (arbol->izq == nullptr && arbol->der == nullptr)

cout << arbol->dato << " ";

if (arbol->der != nullptr)

push(pila,arbol->der);

if (arbol->izq != nullptr)

push(pila,arbol->izq);

}

}

}

int contarHojasRecursivo(nodeArbol\* arbol){

if (arbol == nullptr)

return 0;

if (arbol->izq == nullptr && arbol->der == nullptr)

return 1;

return contarNodosHoja(arbol->izq) + contarNodosHoja(arbol->der);

}

void mostrarNodosHojaRecursivo(nodeArbol\* arbol){

if (arbol != nullptr){

if (arbol->izq == nullptr && arbol->der == nullptr)

cout << arbol->dato << " ";

mostrarNodosHojaRecursivo(arbol->izq);

mostrarNodosHojaRecursivo(arbol->der);

}

}

void BPNv2(nodeArbol\* arbol){ // Barrido por niveles, pero indicando el nivel.

if (arbol == nullptr)

cout << "Arbol vacio";

else{

int nivel = 0, nivelActual;

nodoCola\* frente = nullptr; nodoCola\* fondo = nullptr;

nodeArbol\* aux;

insert(frente,fondo,arbol,nivel);

cout << "Nodo rey: ";

while (!ColaVacia(frente)){

aux = get(frente,fondo, nivelActual);

// Procesamiento

if (nivel != nivelActual)

cout << "\nNivel " << nivel+1 << ": ";

cout << aux->dato << " ";

// Obtener altura actual

if (nivel < nivelActual)

nivel = nivelActual;

if (aux->izq != nullptr)

insert(frente,fondo,aux->izq,nivel+1);

if (aux->der != nullptr)

insert(frente,fondo,aux->der,nivel+1);

}

}

}