Escuela Politécnica del Ejército

LISTAS

Cuaderno de Estructuras de Datos

En el siguiente documento podrán encontrar temas como:

1. Representación en memoria estática y dinámica
2. Operaciones básicas con pilas
3. Notación infija, prefija y postfija
4. Recursividad con ayuda de pilas
5. Aplicaciones

2023

Tercero   
Curso :14675

1. Representación en memoria estática y dinámica
2. Operaciones básicas con pilas
3. Notación infija, prefija y postfija
4. Recursividad con ayuda de pilas
5. Aplicaciones

Las listas enlazadas se utilizan en muchas aplicaciones. Algunas de las aplicaciones más comunes se describen a continuación.

**Aplicaciones de listas enlazadas**

**Editor de texto:** Un editor de texto utiliza una lista enlazada para almacenar las líneas de texto. Cada nodo de la lista contiene una línea de texto y un puntero al siguiente nodo. El puntero del último nodo apunta a NULL.

#include <iostream>

#include <fstream>

using namespace std;

struct Node

{

string line;

Node\* next;

};

class TextEditor

{

private:

Node\* head;

Node\* tail;

public:

TextEditor()

{

head = NULL;

tail = NULL;

}

void addLine(string line)

{

Node\* temp = new Node;

temp->line = line;

temp->next = NULL;

if (head == NULL)

{

head = temp;

tail = temp;

}

else

{

tail->next = temp;

tail = tail->next;

}

}

void display()

{

Node\* temp = head;

while (temp != NULL)

{

cout << temp->line << endl;

temp = temp->next;

}

}

};

int main()

{

TextEditor editor;

editor.addLine("This is the first line.");

editor.addLine("This is the second line.");

editor.addLine("This is the third line.");

editor.display();

return 0;

}

**Simulación de procesos:** En una simulación de procesos, una lista enlazada se utiliza para representar una cola de procesos. Cada nodo de la lista contiene información sobre un proceso y un puntero al siguiente nodo. El puntero del último nodo apunta a NULL.

#include <iostream>

using namespace std;

struct Process

{

int id;

int time;

Process\* next;

};

class ProcessQueue

{

private:

Process\* head;

Process\* tail;

public:

ProcessQueue()

{

head = NULL;

tail = NULL;

}

void addProcess(int id, int time)

{

Process\* temp = new Process;

temp->id = id;

temp->time = time;

temp->next = NULL;

if (head == NULL)

{

head = temp;

tail = temp;

}

else

{

tail->next = temp;

tail = tail->next;

}

}

void runProcesses()

{

Process\* temp = head;

while (temp != NULL)

{

cout << "Running process " << temp->id << " for " << temp->time << " seconds." << endl;

temp = temp->next;

}

}

};

int main()

{

ProcessQueue queue;

queue.addProcess(1, 10);

queue.addProcess(2, 5);

queue.addProcess(3, 15);

queue.runProcesses();

return 0;

}

**Manejo de memoria:** En un sistema operativo, una lista enlazada se utiliza para mantener un registro de las áreas de memoria libre y ocupada. Cada nodo de la lista representa un bloque de memoria y contiene información sobre el tamaño del bloque y un puntero al siguiente nodo. El puntero del último nodo apunta a NULL.

#include <iostream>

using namespace std;

struct MemoryBlock

{

int size;

MemoryBlock\* next;

};

class MemoryManager

{

private:

MemoryBlock\* head;

MemoryBlock\* tail;

public:

MemoryManager()

{

head = NULL;

tail = NULL;

}

void allocateMemory(int size)

{

MemoryBlock\* temp = new MemoryBlock;

temp->size = size;

temp->next = NULL;

if (head == NULL)

{

head = temp;

tail = temp;

}

else

{

tail->next = temp;

tail = tail->next;

}

}

void freeMemory(int size)

{

MemoryBlock\* temp = head;

MemoryBlock\* prev = NULL;

while (temp != NULL)

{

if (temp->size == size)

{

if (prev == NULL)

{

head = temp->next;

}

else

{

prev->next = temp->next;

}

delete temp;

return;

}

prev = temp;

temp = temp->next;

}

}

void display()

{

MemoryBlock\* temp = head;

while (temp != NULL)

{

cout << "Block of size " << temp->size << endl;

temp = temp->next;

}

}

};

int main()

{

MemoryManager manager;

manager.allocateMemory(100);

manager.allocateMemory(200);

manager.allocateMemory(300);

manager.display();

manager.freeMemory(200);

manager.display();

return 0;

}

**Implementación de pilas y colas:** Las pilas y las colas se pueden implementar utilizando listas enlazadas. En una pila, el último elemento insertado es el primero en ser eliminado (LIFO), mientras que en una cola, el primer elemento insertado es el primero en ser eliminado (FIFO).

#include <iostream>

using namespace std;

struct Node

{

int data;

Node\* next;

};

class Stack

{

private:

Node\* top;

public:

Stack()

{

top = NULL;

}

void push(int data)

{

Node\* temp = new Node;

temp->data = data;

temp->next = top;

top = temp;

}

int pop()

{

if (top == NULL)

{

cout << "Stack is empty." << endl;

return -1;

}

else

{

int data = top->data;

Node\* temp = top;

top = top->next;

delete temp;

return data;

}

}

void display()

{

Node\* temp = top;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

};

class Queue

{

private:

Node\* front;

Node\* rear;

public:

Queue()

{

front = NULL;

rear = NULL;

}

void enqueue(int data)

{

Node\* temp = new Node;

temp->data = data;

temp->next = NULL;

if (front == NULL)

{

front = temp;

rear = temp;

}

else

{

rear->next = temp;

rear = rear->next;

}

}

int dequeue()

{

if (front == NULL)

{

cout << "Queue is empty." << endl;

return -1;

}

else

{

int data = front->data;

Node\* temp = front;

if (front == rear)

{

front = NULL;

rear = NULL;

}

else

{

front = front->next;

}

delete temp;

return data;

}

}

void display()

{

Node\* temp = front;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

};

int main()

{

Stack stack;

stack.push(1);

stack.push(2);

stack.push(3);

stack.display();

stack.pop();

stack.display();

Queue queue;

queue.enqueue(1);

queue.enqueue(2);

queue.enqueue(3);

queue.display();

queue.dequeue();

queue.display();

return 0;

}

**Recursividad:** La recursividad se puede implementar utilizando una pila. Cada vez que se llama a una función recursiva, se inserta un registro en la pila que contiene información sobre la llamada. Cuando la función termina, se elimina el registro de la pila y se reanuda la ejecución en el punto donde se llamó a la función.

#include <iostream>

using namespace std;

struct Node

{

int data;

Node\* next;

};

class Stack

{

private:

Node\* top;

public:

Stack()

{

top = NULL;

}

void push(int data)

{

Node\* temp = new Node;

temp->data = data;

temp->next = top;

top = temp;

}

int pop()

{

if (top == NULL)

{

cout << "Stack is empty." << endl;

return -1;

}

else

{

int data = top->data;

Node\* temp = top;

top = top->next;

delete temp;

return data;

}

}

bool isEmpty()

{

return top == NULL;

}

};

void recursiveFunction(int n)

{

Stack stack;

while (n > 0)

{

stack.push(n);

n--;

}

while (!stack.isEmpty())

{

n = stack.pop();

cout << n << " ";

}

cout << endl;

}

int main()

{

recursiveFunction(5);

return 0;

}

1. Referencias

* Malik, D. S. (2013). Estructuras de Datos con C++. 2a. Ed. Cengage Learning.