**Двусвязный список**

Отличие от односвязанного списка состоит в том, что в двусвязном (или двунаправленном списке) узел состоит не из двух, а из трех частей. В третьем компоненте хранится указатель на предыдущий элемент.

//узел (звено) списка

struct node

{

//Информационный элемент звена списка

int value;

// Указатель на предыдущее звено списка

node \*prev;

// Указатель на следующее звено списка

node \*next;

};

**Формирование двусвязного списка**

1. Отведем место для указателей в статической памяти и зарезервируем место для динамического объекта.
2. Присвоим значение переменной ptail, и поместим в информационное поле значение элемента.
3. Присвоим указателю на предыдущий элемент значение NULL (т. к. элемент первый - предыдущего нет).
4. Поместим в поле звена адрес еще одного - нового динамического объекта.
5. В новый добавленный объект записываем значение, в указатель на следующее звено записываем NULL, т. к. объект добавляется в конец.
6. В указатель на предыдущий элемент записываем адрес предыдущего объекта.
7. Переменная ptail должна содержать адрес последнего добавленного элемента, т. к. он добавлен в конец.
8. Двусвязный список из двух элементов готов.
9. Вставка элемента в двусвязный список.
10. Выделить память под новый узел.
11. Записать в новый узел значение.
12. В указатель на предыдущий узел записать адрес узла, который должен располагаться перед новым узлом.
13. Записать в указатель на следующий узел адрес узла, который должен быть расположен после нового узла.
14. В предыдущем узле заменяем значение указателя на следующий узел на адрес нового узла.
15. В следующем узле заменяем значение указателя на предыдущий узел на адрес нового узла.

На рисунке изображен двусвязный циклический список:

**Удаление элемента из двусвязного списка**

1. Записать адрес узла, следующего за удаляемым узлом, в указатель на следующий узел узла, являющегося предыдущим для удаляемого узла.
2. Записать адрес узла, являющегося предыдущим для удаляемого, в указатель на предыдущий узел узла, следующего за удаляемым узлом.
3. Удалить узел, предназначенный для удаления.

Такое построение позволяет производить движение по списку, как в прямом, так и в обратном направлении.

**Реализация двусвязного списка**

#include <iostream>

using namespace std;

struct Elem

{

int data; // данные

Elem \* next, \* prev;

};

class List

{

// Голова, хвост

Elem \* Head, \* Tail;

// Количество элементов

int Count;

public:

// Конструктор

List();

// Конструктор копирования

List(const List&);

// Деструктор

~List();

// Получить количество

int GetCount();

// Получить элемент списка

Elem\* GetElem(int);

// Удалить весь список

void DelAll();

// Удаление элемента, если параметр не указывается,

// то функция его запрашивает

void Del(int pos = 0);

// Вставка элемента, если параметр не указывается,

// то функция его запрашивает

void Insert(int pos = 0);

// Добавление в конец списка

void AddTail(int n);

// Добавление в начало списка

void AddHead(int n);

// Печать списка

void Print();

// Печать определенного элемента

void Print(int pos);

List& operator = (const List&);

// сложение двух списков (дописывание)

List operator + (const List&);

// сравнение по элементам

bool operator == (const List&);

bool operator != (const List&);

bool operator <= (const List&);

bool operator >= (const List&);

bool operator < (const List&);

bool operator > (const List&);

// переворачивание списка

List operator - ();

};

List::List()

{

// Изначально список пуст

Head = Tail = NULL;

Count = 0;

}

List::List(const List & L)

{

Head = Tail = NULL;

Count = 0;

// Голова списка, из которого копируем

Elem \* temp = L.Head;

// Пока не конец списка

while(temp != 0)

{

// Передираем данные

AddTail(temp->data);

temp = temp->next;

}

}

List::~List()

{

// Удаляем все элементы

DelAll();

}

void List::AddHead(int n)

{

// новый элемент

Elem \* temp = new Elem;

// Предыдущего нет

temp->prev = 0;

// Заполняем данные

temp->data = n;

// Следующий - бывшая голова

temp->next = Head;

// Если элементы есть?

if(Head != 0)

Head->prev = temp;

// Если элемент первый, то он одновременно и голова и хвост

if(Count == 0)

Head = Tail = temp;

else

// иначе новый элемент - головной

Head = temp;

Count++;

}

void List::AddTail(int n)

{

// Создаем новый элемент

Elem \* temp = new Elem;

// Следующего нет

temp->next = 0;

// Заполняем данные

temp->data = n;

// Предыдущий - бывший хвост

temp->prev = Tail;

// Если элементы есть?

if(Tail != 0)

Tail->next = temp;

// Если элемент первый, то он одновременно и голова и хвост

if(Count == 0)

Head = Tail = temp;

else

// иначе новый элемент - хвостовой

Tail = temp;

Count++;

}

void List::Insert(int pos)

{

// если параметр отсутствует или равен 0, то запрашиваем его

if(pos == 0)

{

cout << "Input position: ";

cin >> pos;

}

// Позиция от 1 до Count?

if(pos < 1 || pos > Count + 1)

{

// Неверная позиция

cout << "Incorrect position !!!\n";

return;

}

// Если вставка в конец списка

if(pos == Count + 1)

{

// Вставляемые данные

int data;

cout << "Input new number: ";

cin >> data;

// Добавление в конец списка

AddTail(data);

return;

}

else if(pos == 1)

{

// Вставляемые данные

int data;

cout << "Input new number: ";

cin >> data;

// Добавление в начало списка

AddHead(data);

return;

}

// Счетчик

int i = 1;

// Отсчитываем от головы n - 1 элементов

Elem \* Ins = Head;

while(i < pos)

{

// Доходим до элемента,

// перед которым вставляемся

Ins = Ins->next;

i++;

}

// Доходим до элемента,

// который предшествует

Elem \* PrevIns = Ins->prev;

// Создаем новый элемент

Elem \* temp = new Elem;

// Вводим данные

cout << "Input new number: ";

cin >> temp->data;

// настройка связей

if(PrevIns != 0 && Count != 1)

PrevIns->next = temp;

temp->next = Ins;

temp->prev = PrevIns;

Ins->prev = temp;

Count++;

}

void List::Del(int pos)

{

// если параметр отсутствует или равен 0, то запрашиваем его

if(pos == 0)

{

cout << "Input position: ";

cin >> pos;

}

// Позиция от 1 до Count?

if(pos < 1 || pos > Count)

{

// Неверная позиция

cout << "Incorrect position !!!\n";

return;

}

// Счетчик

int i = 1;

Elem \* Del = Head;

while(i < pos)

{

// Доходим до элемента,

// который удаляется

Del = Del->next;

i++;

}

// Доходим до элемента,

// который предшествует удаляемому

Elem \* PrevDel = Del->prev;

// Доходим до элемента, который следует за удаляемым

Elem \* AfterDel = Del->next;

// Если удаляем не голову

if(PrevDel != 0 && Count != 1)

PrevDel->next = AfterDel;

// Если удаляем не хвост

if(AfterDel != 0 && Count != 1)

AfterDel->prev = PrevDel;

// Удаляются крайние?

if(pos == 1)

Head = AfterDel;

if(pos == Count)

Tail = PrevDel;

// Удаление элемента

delete Del;

Count--;

}

void List::Print(int pos)

{

// Позиция от 1 до Count?

if(pos < 1 || pos > Count)

{

// Неверная позиция

cout << "Incorrect position !!!\n";

return;

}

Elem \* temp;

// Определяем с какой стороны

// быстрее двигаться

if(pos <= Count / 2)

{

// Отсчет с головы

temp = Head;

int i = 1;

while(i < pos)

{

// Двигаемся до нужного элемента

temp = temp->next;

i++;

}

}

else

{

// Отсчет с хвоста

temp = Tail;

int i = 1;

while(i <= Count - pos)

{

// Двигаемся до нужного элемента

temp = temp->prev;

i++;

}

}

// Вывод элемента

cout << pos << " element: ";

cout << temp->data << endl;

}

void List::Print()

{

// Если в списке присутствуют элементы, то пробегаем по нему

// и печатаем элементы, начиная с головного

if(Count != 0)

{

Elem \* temp = Head;

cout << "( ";

while(temp->next != 0)

{

cout << temp->data << ", ";

temp = temp->next;

}

cout << temp->data << " )\n";

}

}

void List::DelAll()

{

// Пока остаются элементы, удаляем по одному с головы

while(Count != 0)

Del(1);

}

int List::GetCount()

{

return Count;

}

Elem \* List::GetElem(int pos)

{

Elem \*temp = Head;

// Позиция от 1 до Count?

if(pos < 1 || pos > Count)

{

// Неверная позиция

cout << "Incorrect position !!!\n";

return 0;

}

int i = 1;

// Ищем нужный нам элемент

while(i < pos && temp != 0)

{

temp = temp->next;

i++;

}

if(temp == 0)

return 0;

else

return temp;

}

List & List::operator = (const List & L)

{

// Проверка присваивания элемента "самому себе"

if(this == &L)

return \*this;

// удаление старого списка

this->~List(); // DelAll();

Elem \* temp = L.Head;

// Копируем элементы

while(temp != 0)

{

AddTail(temp->data);

temp = temp->next;

}

return \*this;

}

// сложение двух списков

List List::operator + (const List& L)

{

// Заносим во временный список элементы первого списка

List Result(\*this);

// List Result = \*this;

Elem \* temp = L.Head;

// Добавляем во временный список элементы второго списка

while(temp != 0)

{

Result.AddTail(temp->data);

temp = temp->next;

}

return Result;

}

bool List::operator == (const List& L)

{

// Сравнение по количеству

if(Count != L.Count)

return false;

Elem \*t1, \*t2;

t1 = Head;

t2 = L.Head;

// Сравнение по содержанию

while(t1 != 0)

{

// Сверяем данные, которые

// находятся на одинаковых позициях

if(t1->data != t2->data)

return false;

t1 = t1->next;

t2 = t2->next;

}

return true;

}

bool List::operator != (const List& L)

{

// Используем предыдущую функцию сравнения

return !(\*this == L);

}

bool List::operator >= (const List& L)

{

// Сравнение по количеству

if(Count > L.Count)

return true;

// Сравнение по содержанию

if(\*this == L)

return true;

return false;

}

bool List::operator <= (const List& L)

{

// Сравнение по количеству

if(Count < L.Count)

return true;

// Сравнение по содержанию

if(\*this == L)

return true;

return false;

}

bool List::operator > (const List& L)

{

if(Count > L.Count)

return true;

return false;

}

bool List::operator < (const List& L)

{

if(Count < L.Count)

return true;

return false;

}

// переворот

List List::operator - ()

{

List Result;

Elem \* temp = Head;

// Копируем элементы списка, начиная с головного,

// в свой путем добавления элементов в голову,

// таким образом временный список Result будет содержать

// элементы в обратном порядке

while(temp != 0)

{

Result.AddHead(temp->data);

temp = temp->next;

}

return Result;

}

// Тестовый пример

void main()

{

List L;

const int n = 10;

int a[n] = {0,1,2,3,4,5,6,7,8,9};

// Добавляем элементы, стоящие на четных индексах, в голову,

// на нечетных - в хвост

for(int i = 0; i < n; i++)

if(i % 2 == 0)

L.AddHead(a[i]);

else

L.AddTail(a[i]);

// Распечатка списка

cout << "List L:\n";

L.Print();

cout << endl;

// Вставка элемента в список

L.Insert();

// Распечатка списка

cout << "List L:\n";

L.Print();

// Распечатка 2-го и 8-го элементов списка

L.Print(2);

L.Print(8);

List T;

// Копируем список

T = L;

// Распечатка копии

cout << "List T:\n";

T.Print();

// Складываем два списка (первый в перевернутом состоянии)

cout << "List Sum:\n";

List Sum = -L + T;

// Распечатка списка

Sum.Print();

}