ФГБОУ ВО РГУПС

Алгоритмизация и программирование

Динамические структуры данных. Односвязные линейные списки

Лекция 8

© Составление,
О.В. Игнатьева

Ростов-на-Дону
2020

План лекции

- Управление памятью в программировании
- Понятие динамических структур данных
- Типы динамических структур
- Односвязный линейный список

Управление памятью в программировании

Управление памятью

Бурное развитие прогресса и повсеместное внедрение компьютеров и компьютерных технологий в общественную жизнь породило **увеличение объемов информации**, а также ее ценности, поэтому остро возник вопрос о ее обработке и сохранении.

Управление памятью

На сегодняшний день способы обработки и хранения информации значительно упростились, появились программные средства, которые могут обрабатывать **большие объемы информации**. На основе таких программных средств строятся информационные системы.

Управление памятью

В любой информационной системе ключевым элементом является **память**.

Управление памятью - одна из главных задач в программировании.

Одним из способов управления памяти является **динамическое ее распределение**.

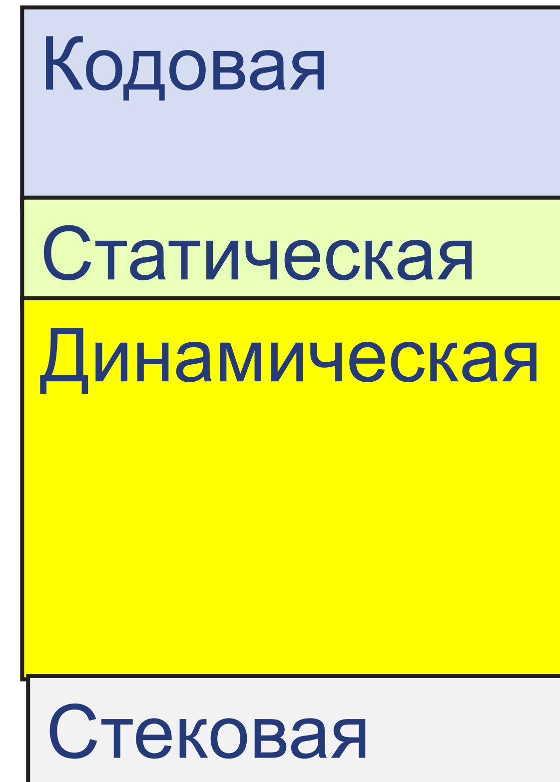
Очень важно уметь создавать программы, которые могут эффективно использовать память.

Модель памяти

Когда начинается выполнение программы, написанной на языке С++, компилятор резервирует память:

- для ее кода – эту память называют **кодовой**
- для глобальных переменных – эту память называют **статической**
- память, которая будет использоваться при вызове функций для хранения их аргументов и локальных переменных называется **стековой или автоматической**
- остальная память компьютера называется **свободной или динамической.**

Схема памяти



Модель памяти

- **Статическая** — выделение памяти до начала исполнения программы. Такая память доступна на протяжении всего времени выполнения программы. Во многих языках для размещения объекта в статической памяти достаточно задекларировать его в глобальной области видимости.

```
int id = 150;   // определение глобальной переменной
int main()
{
      cout << id + 8;    // её использование
}
```

Модель памяти

- **Автоматическая** —автоматически выделяет аргументы и локальные переменные функции, а также прочую метаинформацию при вызове функции и освобождает память при выходе из неё.
- Автоматическая память работает на основе принципа стека («последним пришёл — первым ушёл»). При завершении работы функции её данные будут удалены с конца стека, уменьшая его размер.

```
int main()
{
    int a = 3;
    int result = factorial(a);
    cout << result;
}
```

```
int factorial(int n)
{
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Модель памяти

- **Динамическая память** — выделение памяти из ОС по требованию приложения.
- После выделения памяти в распоряжение программы поступает **указатель на начало выделенной памяти**, который, в свою очередь, тоже должен где-то храниться: в статической, автоматической или также в динамической памяти. Для **возвращения памяти** обратно необходим только сам указатель. Попытка использования уже очищенной памяти может привести к завершению программы с ошибкой.
- Языки сверхвысокого уровня используют динамическую память как основную: создают почти все объекты в динамической памяти, а на стеке или в статической памяти держат указатели на эти объекты.
- Максимальный размер динамической памяти зависит от многих факторов: ОС, процессор, аппаратная архитектура в целом, максимальный размер ОЗУ у конкретного устройства.

Динамические структуры данных

Динамические структуры данных

- **Динамические структуры данных** – это любая структура данных, занимаемая объем памяти, который не является фиксированным.
- Иными словами, в подобной структуре может храниться произвольное количество элементов. Размер подобной структуры ограничен только объемом оперативной памяти компьютера.

Динамические структуры данных

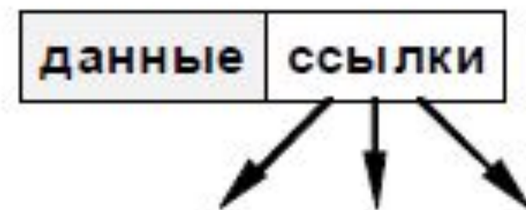
- Структуры одного типа можно объединять не только в массивы. Их можно связывать между собой, создавая так называемые динамические структуры данных.
- Связь между отдельными структурами может быть организована по-разному, и именно поэтому среди динамических данных выделяют **списки, стеки, очереди, деревья, графы** и др.

Динамические структуры данных

Общие определения:

- **Потомок** — элемент структуры, идущий после текущего. В зависимости от вида динамической структуры у элемента может быть более одного потомка.
- **Предок** — элемент структуры, идущий до текущего.
- **Головной элемент** (**Head**) — первый элемент списка.
- **Хвостовой элемент** (**Tail**) — последний элемент списка.

Списки



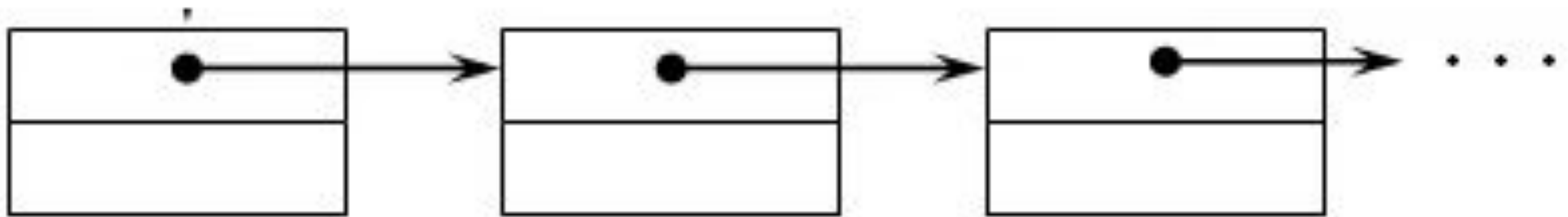
- Динамические структуры представляют собой отдельные элементы, связанные с помощью **ссылок**.
- Каждый элемент (**узел**) состоит из двух областей памяти: **поля данных** и **ссылок**.
- **Ссылки** – это адреса других узлов этого же типа, с которыми данный элемент логически связан. В языке C++ для организации ссылок используются **переменные-указатели.**
- При добавлении нового узла в такую структуру выделяется новый блок памяти и (с помощью ссылок) устанавливаются связи этого элемента с уже существующими.
- Для обозначения конечного элемента в цепи используются **нулевые ссылки** (**NULL**).

Списки

- **Список** — это линейная динамическая структура данных, у каждого элемента может быть только один предок и только один потомок. По сути своей это очень похоже на обыкновенный массив, с той лишь разницей, что размер его не имеет ограничений. Списки также подразделяются на несколько типов.

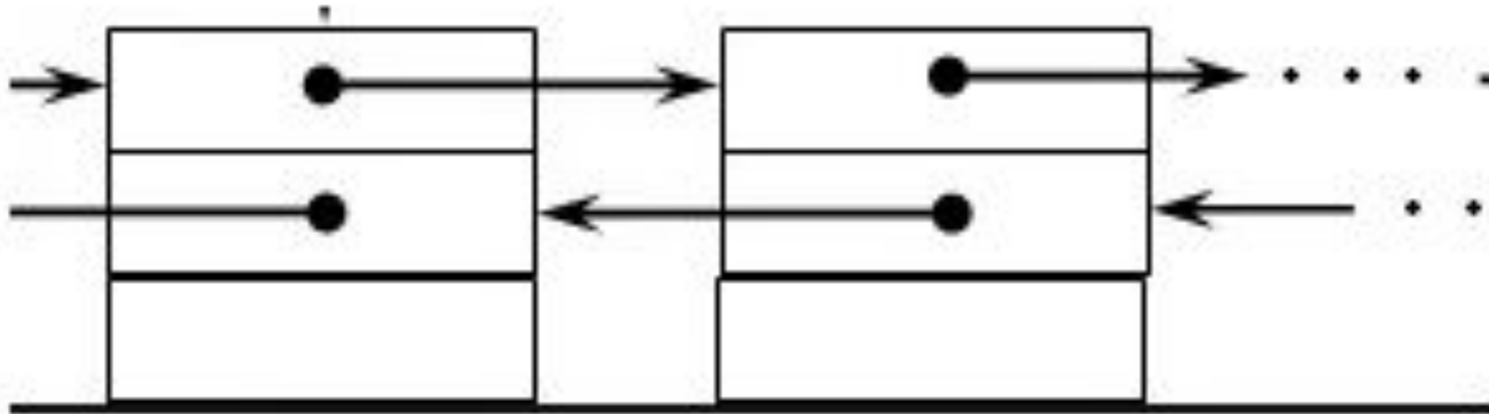
Типы списков

- **Односвязный список** — элемент имеет указатель только на своего потомка.



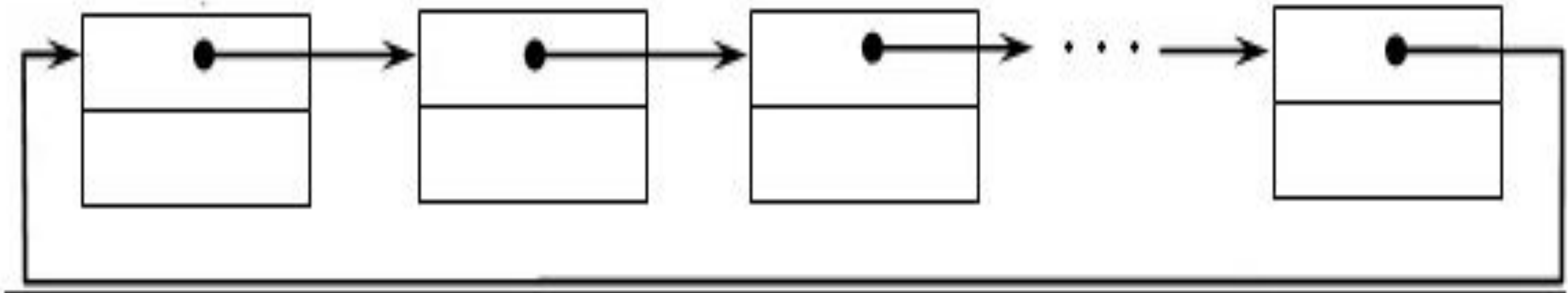
Типы списков

- **Двусвязный список** — элемент имеет указатели и на потомка, и на родителя.



Типы списков

- **Замкнутый (кольцевой, циклический) список** — головной и хвостовой элементы которого указывают друг на друга.



Типы списков

- **Стек** - извлечение и добавление элементов в нем происходит по правилу «Последний пришел, первый вышел» (LIFO — last in, first out). Добавление и извлечение элементов проводится от головы.

Типы списков

- **Очередь** - список, операции чтения и добавления элементов в котором подвержены правилу «Первый пришел, первый вышел» (FIFO — first in, first out) . При этом, при чтении элемента, он удаляется из очереди. Таким образом, для чтения в очереди доступна только голова, в то время как добавление проводится только в хвост.

Достоинства

- эффективное добавление и удаление элементов
- размер ограничен только объёмом памяти компьютера и разрядностью указателей
- динамическое добавление и удаление элементов

Недостатки

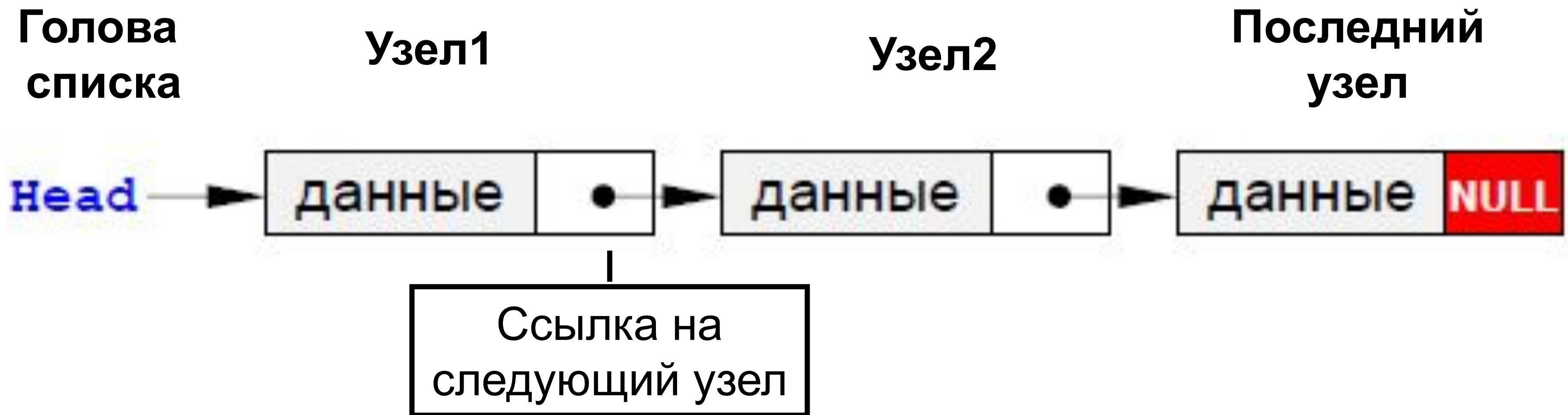
- сложность прямого доступа к элементу, а именно определения физического адреса по его индексу (порядковому номеру) в списке
- на поля-указатели (указатели на следующий и предыдущий элемент) расходуется дополнительная память
- некоторые операции со списками медленнее, чем с массивами, так как к произвольному элементу списка можно обратиться, только пройдя все предшествующие ему элементы
- соседние элементы списка могут быть распределены в памяти нелокально, что снизит эффективность кэширования данных в процессоре
- над связными списками, по сравнению с массивами, гораздо труднее (хоть и возможно) производить параллельные векторные операции, такие, как вычисление суммы: накладные расходы на перебор элементов снижают эффективность распараллеливания

Односвязные линейные списки

Односвязные списки

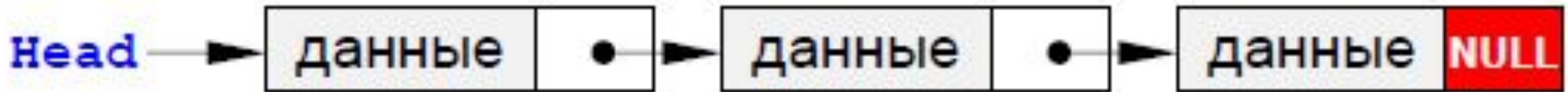
- **Односвязный линейный список – это динамический список, в котором каждый узел содержит всего одну ссылку**.
- Каждый элемент содержит также **ссылку на следующий** за ним элемент.
- **У последнего** в списке элемента поле ссылки содержит **NULL**.
- Чтобы не потерять список, мы должны где-то (в переменной) хранить **адрес его первого узла** – он называется **«головой» списка**.

Односвязные списки



Узел представляет собой структуру, которая содержит поля данные и указатель на такой же узел.

Односвязные списки



В программе надо объявить два новых типа данных – узел списка **Node** и указатель на него **PNode**.

Синтаксис объявления линейного списка в C++:

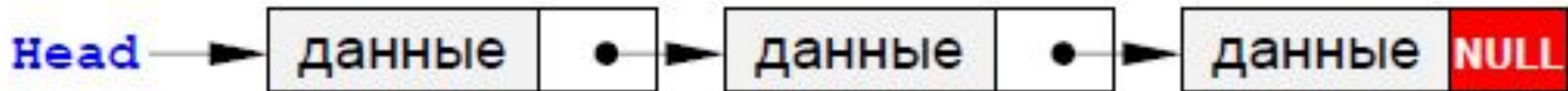
```
struct Node
{
    Тип1  поле1;
    Тип2  поле2;
     .....
     Node *next;
};
Node * PNode;
```

область данных

ссылка на следующий узел

указатель на узел

Односвязные списки



Например, опишем односвязный список для представления словаря русских слов.
Узел представляет собой структуру, которая содержит два поля – строку и указатель на такой же узел.

Пример. Объявление линейного списка словаря:

```
struct Node
{
    string  word;
    Node *next;
};
typedef Node *PNode;
PNode Head=NULL;
PNode pnew;
```

Область данных

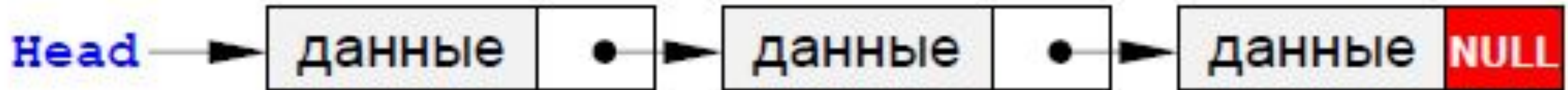
ссылка на следующий узел

тип данных - указатель на узел

Указатель на начало списка. В начале работы в списке нет ни одного элемента, поэтому в указатель Head записываем нулевой адрес NULL

указатель новый узел

Односвязные списки. Операции



Операции над односвязными списками:

- Создание нового узла.
- Добавление узла:
  - В начало списка
  - В конец списка
  - После заданного узла
  - Перед заданным узлом
- Проход по списку
- Поиск узла
- Удаление узла

Создание нового узла

- Для того, чтобы добавить узел к списку, необходимо **создать** его, то есть **выделить память под узел и запомнить адрес выделенного блока.**
- Будем считать, что надо добавить к списку узел, соответствующий новому слову, которое записано в переменной **NewWord.**
- Составим **функцию**, которая создает новый узел в памяти и возвращает его адрес. При записи данных в узел используется обращение к полям структуры через указатель.

```
PNode CreateNode ( string  NewWord )   // функция создания узла
{
      PNode NewNode = new Node;        // указатель на новый узел
      NewNode->word = NewWord;         // записать слово
      NewNode->next = NULL;            // следующего узла нет
      return NewNode;                  // результат функции – адрес узла
}
```

Добавление узла в начало списка

- При добавлении нового узла **NewNode** в начало списка надо:
  - 1) установить ссылку узла **NewNode** на голову существующего списка
  - 2) установить голову списка на новый узел.

Добавление узла в начало списка

- По такой схеме работает функция **AddFirst**. Предполагается, что адрес начала списка хранится в **Head**.
- Важно, что здесь и далее **адрес начала списка передается *по ссылке***, так как при добавлении нового узла он изменяется внутри процедуры.

// функция добавления нового узла в начало списка

```
void AddFirst (PNode &Head, PNode NewNode)
{
    NewNode->next = Head;
    Head = NewNode;
}
```

1) установить ссылку нового узла на голову списка

2) установить голову списка на новый узел

Добавление узла после заданного

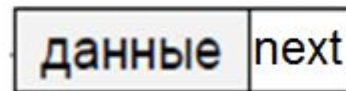
- Дан адрес **NewNode** нового узла и адрес **p** одного из существующих узлов в списке. Требуется вставить в список новый узел после узла с адресом **p**. Эта операция выполняется в два этапа:
  - 1) установить ссылку нового узла на узел, следующий за данным;
  - 2) установить ссылку данного узла **p** на **NewNode**.

2) На что теперь Узел P ссылается?

1) установить ссылку нового узла на узел, следующий за P

На новый узел!

Узел NewNode



NewNode->next = p -> next;

p -> next = NewNode;

Добавление узла после заданного

- По такой схеме работает функция **AddAfter**.
- Передавать будем адрес узла после которого, хотим вставить новый узел и адрес самого нового узла.
- Последовательность операций менять нельзя, потому что если сначала поменять ссылку у узла **p**, будет потерян адрес следующего узла.

// функция добавления нового узла после указанного

```
void AddAfter (PNode p, PNode NewNode)
{
   NewNode->next = p->next;
   p->next = NewNode;
}
```

1) установить ссылку нового узла на узел, следующий за данным

2) установить ссылку данного узла **p** на **NewNode**

Добавление узла перед заданным

- Эта схема добавления самая сложная. Проблема заключается в том, что в простейшем линейном списке для того, чтобы получить адрес предыдущего узла, нужно **пройти весь список сначала**.
- **Задача сведется либо к вставке узла в начало списка (если заданный узел – первый), либо к вставке после заданного узла.**

1)Сначала проверим, вдруг перед узлом P нет никого. Это будет значить, что он самый первый и на него указывает голова

```
if (Head == p)
```

Если это так, то вставим новый узел В начало списка

```
AddFirst(Head, NewNode);
```

2) Иначе, перед узлом p существует узел. Как его найти?

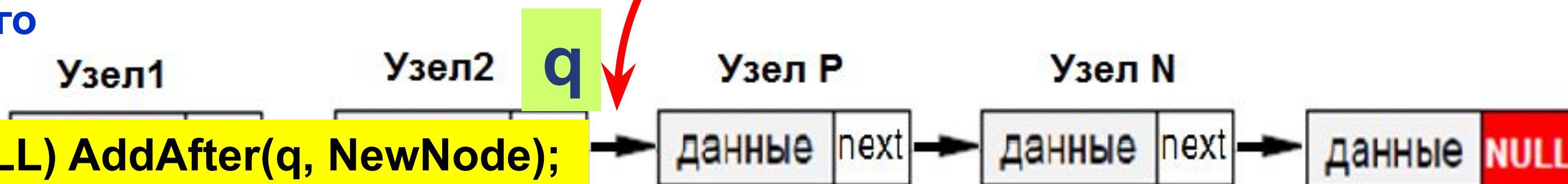
Нужно объявить еще одну переименую, присвоить ей сначала адрес начала списка

```
PNode q = Head;
```

И затем в цикле будем искать адрес узла, у которого ссылка next указывает на p

```
while (q->next!=p && q!=NULL )
q = q->next;
```

Если нашли такой узел, то добавим новый узел после него




```
if (q!=NULL) AddAfter(q, NewNode);
```

Добавление узла перед заданным

// функция добавления нового узла перед заданным

```
void AddBefore(PNode &Head, PNode p, PNode NewNode)
{
    PNode q = Head;
    if (Head == p)              // вставка перед первым узлом
    {   AddFirst(Head, NewNode);
        return;
    }
                  // ищем узел, за которым следует p
    while (q->next!=p  && q!=NULL )
        q = q->next;
                                  если нашли такой узел,
                                   добавить новый после него
    if ( q!=NULL ) AddAfter(q, NewNode);
}
```

Добавление узла в конец списка

- Для решения задачи надо сначала **найти последний узел**, у которого ссылка равна **NULL**, а затем воспользоваться процедурой **вставки после** заданного узла.
- Отдельно надо обработать случай, когда список пуст.

1)Сначала проверим, вдруг список пустой.

```
if (Head == NULL)
```

Если это так, то вставим новый узел в начало списка

```
AddFirst(Head, NewNode);
```

2) Иначе, нужно найти последний узел. Как его найти?

Нужно объявить еще одну переменую, присвоить ей сначала адрес начала списка

```
PNode q = Head;
```

И затем в цикле будем искать адрес узла, у которого ссылка next указывает на NULL

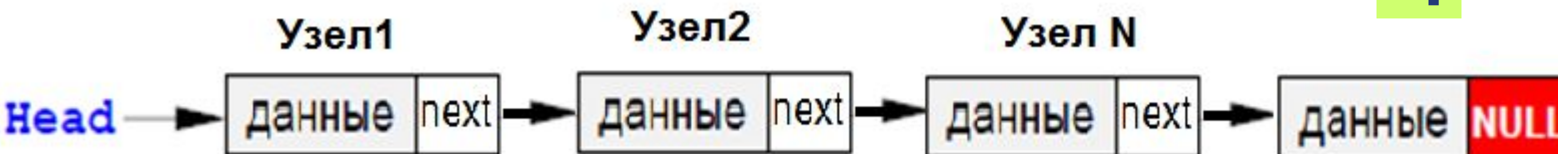
```
while (q->next != NULL )
    q = q->next;
```

Если нашли такой узел, то добавим новый узел после него

```
AddAfter(q, NewNode);
```

Узел NewNode

q



Добавление узла в конец списка

// функция добавления нового узла в конец списка

```
void AddLast(PNode &Head, PNode NewNode)
{
    PNode q = Head;          // если список пуст,
    if (Head == NULL)        то вставляем  первый элемент
    {
        AddFirst(Head, NewNode);
        return;
    }
    while (q->next) q = q->next;     // ищем последний узел

    AddAfter(q, NewNode);    если нашли такой узел,
                             добавить новый после него
}
```

Проход по списку

Для того, чтобы пройти весь список и сделать что-либо с каждым его элементом, надо начать с головы и, используя указатель **next**, продвигаться к следующему узлу.

// Обход списка

```
PNode p = Head;          // начинаем с головы списка

while ( p != NULL )      // пока не дошли до конца
{                        // делаем что-нибудь с узлом p
    p = p->next;         // переходим к следующему узлу
}
```

Поиск узла в списке

Часто требуется найти в списке нужный элемент (его адрес или данные). Надо учесть, что требуемого элемента может и не быть, тогда просмотр заканчивается при достижении конца списка.
Такой подход приводит к следующему **алгоритму**:

1) начать с головы списка;

```
PNode q = Head;
```

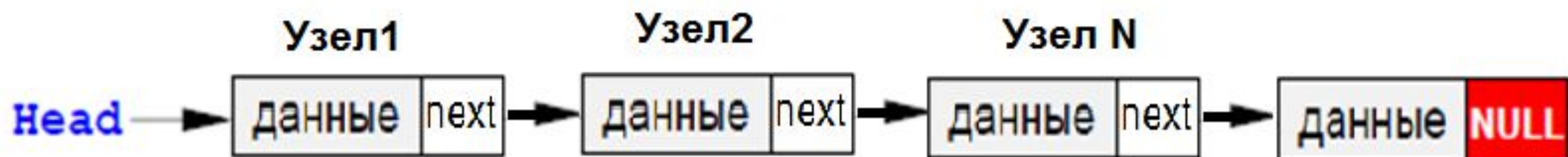
2) пока текущий элемент существует (указатель – не NULL), проверить нужное условие и перейти к следующему элементу;

```
while (q->word != NewWord  && q!=NULL)
      q = q->next;
```

3) закончить, когда найден требуемый элемент или все элементы списка просмотрены.

```
return q;
```

Поиск по данным



Поиск узла в списке

Например, следующая функция ищет в списке элемент, соответствующий заданному слову (для которого поле **word** совпадает с заданной строкой **NewWord**), и возвращает его адрес или **NULL**, если такого узла нет.

// функция поиска узла в списке

```
PNode Find (PNode Head, string NewWord)
{     //начать с головы списка
    PNode q = Head;

    while (q->word != NewWord && q != NULL)
        q = q->next;

    return q;
}
```

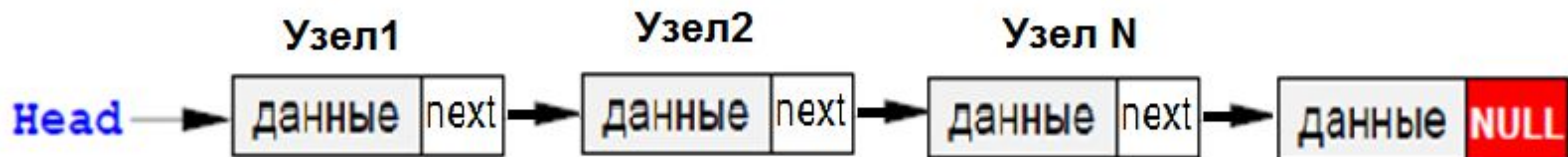
//пока текущий элемент существует (указатель – не NULL), проверить нужное условие и перейти к следующему элементу

закончить, когда найден требуемый элемент или все элементы списка просмотрены

Поиск узла по порядку в списке

- Вернемся к задаче построения алфавитного словаря. Для того, чтобы добавить новое слово в нужное место (**в алфавитном порядке**), требуется **найти адрес узла, *перед* которым надо вставить новое слово**.
- Это будет первый от начала списка узел, для которого «его»
- слово окажется «больше», чем новое слово.
- Поэтому достаточно просто изменить условие в цикле **while** в функции **Find**, учитывая, что сравнение **q->word < NewWord** возвращает значение «больше» или «меньше» по естественному лексикографичексому порядку.

Поиск по данным в определённом порядке - алфавитном



Поиск узла по порядку в списке

1) начать с головы списка;

```
PNode q = Head;
```

2) пока текущий элемент существует (указатель – не NULL), проверить условие и перейти к следующему элементу;

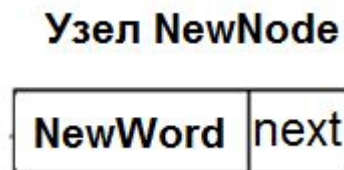
```
while (q->word  <  NewWord  && q!=NULL)
      q = q->next;
```

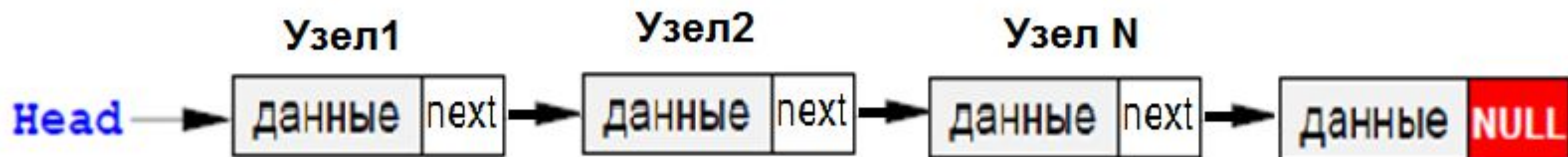
3) закончить, когда найден требуемый элемент или все элементы списка просмотрены.

```
return q;
```

Куда его вставить?

Будем искать место – адрес узла, перед которым нужно вставить новый узел.
Чтобы его данные NewWord были меньше, чем данные у узла перед ним.





Поиск узла по порядку в списке

Эта функция вернет адрес узла, перед которым надо вставить новое слово , когда сравнение вернет true, или **NULL**, если слово надо добавить в конец списка.

// функция поиска узла по порядку в списке

```
PNode FindPlace (PNode Head, string NewWord)
{    //начать с головы списка
    PNode q = Head;

    while (q->word < NewWord && q != NULL)
        q = q->next;

    return q;
}
```

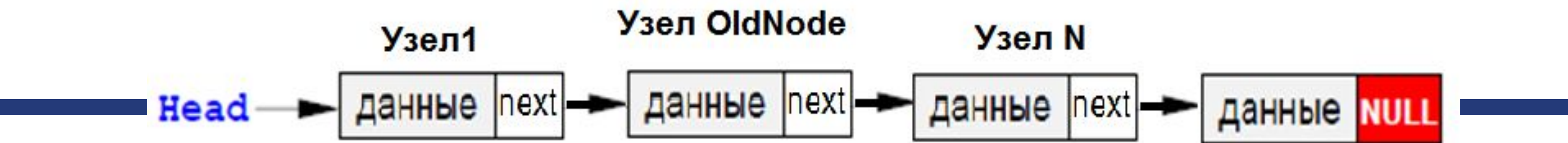
//пока текущий элемент существует (указатель – не NULL), проверить нужное условие и перейти к следующему элементу

закончить, когда найден требуемый элемент или все элементы списка просмотрены

Удаление узла

- Эта процедура также связана с **поиском** заданного узла по всему списку, так как нам надо поменять ссылку у **предыдущего узла**, а перейти к нему непосредственно невозможно.
- Если мы и нашли узел, за которым идет удаляемый узел, надо **просто переставить ссылку**.
- Отдельно обрабатывается **случай**, когда удаляется **первый элемент списка**. В этом случае адрес удаляемого узла совпадает с адресом головы списка **Head** и надо просто записать в **Head** адрес следующего элемента.
- При удалении узла **освобождается память**, которую он занимал.

Удаление узла



1) Найдем узел, который ссылается на удаляемый узел

```
while (q->next != OldNode && q!=NULL )
    q = q->next;
```

2) Изменяем ссылку у предыдущего узла на ссылку, который указывает OldNode

```
q->next = OldNode->next;
```

4) Освобождаем память, которую занимал узел.

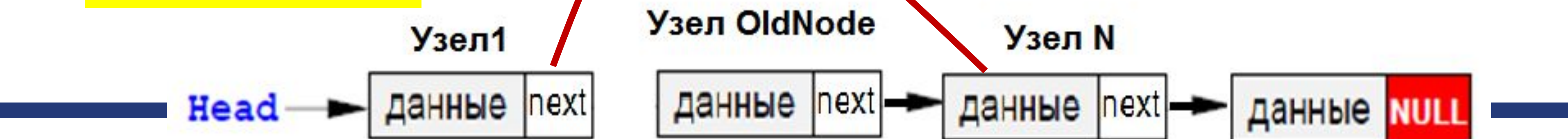
```
delete OldNode;
```

3) Отдельно обрабатывается случай, когда удаляется первый элемент списка.

```
if (Head == OldNode)
```

В этом случае адрес удаляемого узла совпадает с адресом головы списка Head и надо просто записать в Head адрес следующего элемента.

```
Head = OldNode->next;
```



Удаление узла

// функция удаления узла из списка

```
void DeleteNode(PNode &Head, PNode OldNode)
{
        PNode q = Head;
        if (Head == OldNode)
            Head = OldNode->next;      // удаляем первый элемент
        else
        {                      // ищем элемент, который ссылается на удаляемый
             while (q->next != OldNode  && q!=NULL)
                  q = q->next;

             if ( q == NULL ) return;     // если не нашли, то выход
            q->next = OldNode->next;     // изменяем ссылки - узел перед
                                         // удаляемым теперь будет ссылаться
                                         // на узел через ссылку удаляемого
        }
        delete OldNode;      // освобождаем память
}
```

Пример программы на односвязный линейный список

Пример. Односвязный список словарь

```
#include <iostream>
#include <string>
using namespace std;
// описание динамической структуры
struct Node
{
     string word;
     int count;
     Node *next;
};
typedef Node *PNode;
// Создание элемента списка
PNode CreateNode ( string NewWord )
{
   PNode NewNode = new Node;
   NewNode->word= NewWord;
   NewNode->count = 1;
   NewNode->next = NULL;
   return NewNode; /
}
 // и так далее описание всех функций
```

Пример (продолжение)

```
int main()
{
    PNode Head = NULL;
    PNode pnew, pfind;
    int t; string newslovo;
    do
    {
      cout<<"введите от 1 до 5 или 0 - выход"<<endl;
      cout<<" 0 - выход "<<endl;
      cout<<" 1 - добавить новый элемент в конец списка "<<endl;
      cout<<" 2 - вывод списка "<<endl;
      cout<<" 3 - добавить новый элемент после выбранного "<<endl;
      cout<<" 4 - добавить новый элемент по алфавиту "<<endl;
      cout<<" 5 - добавить новый элемент перед выбранным "<<endl;
      cout<<" 6 - удалить элемент  "<<endl;
      cout<<endl;
            cin>>t;
            switch (t)
      {

}
```

Пример (продолжение)

```
case 1 :
                    cout<<"введите новое слово = ";
          cin>>newslovo;
          pnew=CreateNode(newslovo); //создаем новый узел
          if (Head==NULL)
              AddFirst (Head, pnew) ;    //вставляем на первое место
          else
              AddLast (Head, pnew) ;    //вставляем в конец списка
          break;
     case 2 :
          pnew=Head;
          while (pnew!=NULL)
           {
                cout<<pnew->word<<"\t"<<pnew->count<<endl;
                pnew=pnew->next;
           }
          break;
         case 3:
            /////
          };
     }
  while (t!=0);
  return 0;
```

Спасибо за внимание!

Литературные источники

- 1) К. Поляков. Программирование на языке С++.
- 2). Харви Дейтел, Пол Дейтел. Как программировать на С++. - М: Вильямс, - 1011 с.
- 3). Струструп Б. Программирование: принципы и практика использования С++. – М. : Вильямс, 2011. – 1248 с.
- 4). Струструп Б. Язык программирования С++. – М.: Бином. - 1054 с.
- 5). Лафоре Р. Объектно-ориентированное программирование в С++. Питер, 2004. – 922 с.
- 6). Шилдт Г. С++: руководство для начинающих, 2-е издание. – М: Вильямс, 2005. -672 с.