# 1. Списки

## Динамические структуры данных

Часто в серьезных программах надо использовать данные, размер и структура которых должны **меняться** в процессе работы. Динамические массивы здесь не выручают, поскольку заранее нельзя сказать, сколько памяти надо выделить – это выясняется только в процессе работы. Например, надо проанализировать текст и определить, какие слова и в каком количество в нем встречаются, причем эти слова нужно расставить по алфавиту.

В таких случаях применяют данные особой структуры, которые представляют собой отдельные элементы, связанные с помощью **ссылок**. **данные ссылки**

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 Си 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**).

### Линейный список

В простейшем случае каждый узел содержит всего одну ссылку. Для определенности будем считать, что решается задача частотного анализа текста – определения всех слов, встречающихся в тексте и их количества. В этом случае область данных элемента включает строку (длиной не более 40 символов) и целое число.

**Head** данные данные данные **NULL**

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. В программе надо объявить два новых типа данных – узел списка **Node** и указатель на него **PNode**. Узел представляет собой структуру, которая содержит три поля - строку, целое число и указатель на такой же узел. Правилами языка Си допускается объявление

**struct Node {**

**char word[40]; // область данных**

**int count;**

**Node \*next; // ссылка на следующий узел**

**}; typedef Node \*PNode; // тип данных: указатель на узел**

В дальнейшем мы будем считать, что указатель **Head**указывает на начало списка, то есть, объявлен в виде

**PNode Head = NULL;**

Первая буква «P» в названии типа **PNode**происходит от слова *pointer* – указатель (англ.) В начале работы в списке нет ни одного элемента, поэтому в указатель **Head**записывается нулевой адрес **NULL**.

## Создание элемента списка

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. Обратите внимание, что при записи данных в узел используется обращение к полям структуры через указатель.

**PNode CreateNode ( char NewWord[] )**

**{**

**PNode NewNode = new Node; // указатель на новый узел strcpy(NewNode->word, NewWord); // записать слово NewNode->count = 1; // счетчик слов = 1 NewNode->next = NULL; // следующего узла нет return NewNode; // результат функции – адрес узла }**

После этого узел надо добавить к списку (в начало, в конец или в середину).

## Добавление узла

#### Добавление узла в начало списка

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

1) 2)

**NewNode NewNode**

данные

данные

данные

данные

**Head**  **Head**

По такой схеме работает процедура **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.

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

#### 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**.

1) 2)

данные

**NewNode**

**p**

данные

данные

данные

**NewNode**

**p**

данные

данные

Последовательность операций менять нельзя, потому что если сначала поменять ссылку у узла **p**, будет потерян адрес следующего узла.

**void AddAfter (PNode p, PNode NewNode)**

**{**

**NewNode->next = p->next;**

**p->next = NewNode;**

**}**

#### Добавление узла перед заданным

Эта схема добавления самая сложная. Проблема заключается в том, что в простейшем линейном списке (он называется *односвязным*, потому что связи направлены только в одну сторону) 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.

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

Такая процедура обеспечивает «защиту от дурака»: если задан узел, не присутствующий в списке, то в конце цикла указатель **q** равен **NULL** и ничего не происходит.

Существует еще один интересный прием: если надо вставить новый узел **NewNodeдо** заданного узла **p**, вставляют узел **после** этого узла, а потом выполняется обмен данными между узлами **NewNode**и **p**. Таким образом, по адресу**p**в самом деле будет расположен узел с новыми данными, а по адресу **NewNode *–*** с теми данными, которые были в узле **p**, то есть мы решили задачу*.*Этот прием не сработает, если адрес нового узла **NewNode**запоминается где-то в программе и потом используется, поскольку по этому адресу будут находиться другие данные.

#### 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.

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

## 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.

**PNode p = Head; // начали 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;
2. пока текущий элемент существует (указатель – не **NULL**), проверить нужное условие и перейти к следующему элементу;
3. закончить, когда найден требуемый элемент или все элементы списка просмотрены.

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.

|  |
| --- |
| **PNode Find (PNode Head, char NewWord[])**  **{**  **PNode q = Head;** |
| **while (q && strcmp(q->word, NewWord)) q = q->next;** |
| **return q; }** |

Вернемся к задаче построения алфавитно-частотного 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 функция **strcmp** возвращает «разность» первого и второго слова.

|  |
| --- |
| **PNode FindPlace (PNode Head, char NewWord[])**  **{**  **PNode q = Head;** |
| **while (q && (strcmp(q->word, NewWord) > 0))** |
| **q = q->next; return q; }** |

Эта функция вернет адрес узла, перед которым надо вставить новое слово (когда функция **strcmp** вернет положительное значение), или **NULL**, если слово надо добавить в конец списка.

## 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 нашли узел, за которым идет удаляемый узел, надо просто переставить ссылку.

данные

**q**

данные

данные

**OldNode**

Отдельно обрабатывается случай, когда удаляется первый элемент списка. При удалении узла освобождается память, которую он занимал.

Отдельно рассматриваем 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.

|  |
| --- |
| **void DeleteNode(PNode &Head, PNode OldNode)**  **{**  **PNode q = Head; if (Head == OldNode)**  **Head = OldNode->next; // удаляем первый элемент else {** |
| **while (q && q->next != OldNode) // ищем элемент** |
| **q = q->next;** |
| **if ( q == NULL ) return; // если не нашли, выход q->next = OldNode->next;**  **}**  **delete OldNode; // освобождаем память }** |

## Барьеры

Вы заметили, что для рассмотренного варианта списка требуется отдельно обрабатывать граничные случаи: добавление в начало, добавление в конец, удаление одного из крайних элементов. Можно значительно упростить приведенные выше процедуры, если установить два барьера – фиктивные первый и последний элементы. Таким образом, в списке всегда есть хотя бы два элемента-барьера, а все рабочие узлы находятся между ними.

## Двусвязный список

Многие проблемы при работе с односвязным списком вызваны тем, что в них невозможно перейти к предыдущему элементу. Возникает естественная идея – хранить в памяти ссылку не только на следующий, но и на предыдущий элемент списка. Для доступа к списку используется не одна переменная-указатель, а две – ссылка на «голову» списка (**Head**) и на «хвост» - последний элемент (**Tail**).

**Head**

данные

данные

данные

**NULL**

**NULL**

**Tail**

Каждый узел содержит (кроме полезных данных) также ссылку на **следующий** за ним узел (поле **next**) и предыдущий (поле **prev**). Поле **next** у последнего элемента и поле **prev** у первого содержат **NULL**. Узел объявляется так:

|  |
| --- |
| **struct Node {**  **char word[40]; // область данных int count;**  **Node \*next, \*prev; // ссылки на соседние узлы**  **};**  **typedef Node \*PNode; // тип данных «указатель на узел»** |

В дальнейшем мы будем считать, что указатель **Head**указывает на начало списка, а указатель **Tail**– на конец списка:

**PNode Head = NULL, Tail = NULL;**

Для пустого списка оба указателя равны **NULL**.

## Операции с двусвязным списком

#### Добавление узла в начало списка

При добавлении нового узла **NewNode**в начало списка надо

1. установить ссылку **next**узла **NewNode**на голову существующего списка и его ссылку **prev** в **NULL**;
2. установить ссылку **prev**бывшего первого узла (если он существовал) на **NewNode**; 3) установить голову списка на новый узел;

4) если в списке не было ни одного элемента, хвост списка также устанавливается на новый узел.

1) **NewNode** 2) **NewNode** 3) **NewNode**

**Head**  **Head**

данные

**NULL**

данные

**NULL**

данные

**NULL**

данные

**NULL**

**Head**

данные

**NULL**

данные

**NULL**

По такой схеме работает следующая процедура:

|  |
| --- |
| **void AddFirst(PNode &Head, PNode &Tail, PNode NewNode)**  **{**  **NewNode->next = Head; NewNode->prev = NULL;**  **if ( Head ) Head->prev = NewNode; Head = NewNode;**  **if ( ! Tail ) Tail = Head; // этот элемент – первый }** |

#### Добавление узла в конец списка

Благодаря симметрии добавление нового узла **NewNode**в конец списка проходит совершенно аналогично, в процедуре надо везде заменить **Head**на **Tail** и наоборот, а также поменять **prev** и **next**.

#### 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 **p**. Если узел **p** является последним, то операция сводится к добавлению в конец списка (см. выше). Если узел **p**– не последний, то операция вставки выполняется в два этапа:

1. установить ссылки нового узла на следующий за данным (***next***) и предшествующий ему (***prev***);
2. установить ссылки соседних узлов так, чтобы включить **NewNode**в список.

1) **NewNode** 2) **NewNode**

данные

**NULL**

данные

**NULL**

данные

**NULL**

**p**

данные

**NULL**

данные

**NULL**

данные

**NULL**

**p**

Такой метод реализует приведенная ниже процедура (она учитывает также возможность вставки элемента в конец списка, именно для этого в параметрах передаются ссылки на голову и хвост списка):

**void AddAfter (PNode &Head, PNode &Tail,**

**PNode p, PNode NewNode)**

**{**

**if ( ! p->next )**

**AddLast (Head, Tail, NewNode); // вставка в конец списка else {**

**NewNode->next = p->next; // меняем ссылки нового узла NewNode->prev = p;**

**p->next->prev = NewNode; // меняем ссылки соседних узлов**

**p->next = NewNode;**

**}**

**}**

Добавление узла перед заданным выполняется аналогично.

#### Поиск узла в списке

Проход по двусвязному списку может выполняться в двух направлениях – от головы к хвосту (как для односвязного) или от хвоста к голове.

#### Удаление узла

Эта процедура также требует ссылки на голову и хвост списка, поскольку они могут измениться при удалении крайнего элемента списка. На первом этапе устанавливаются ссылки соседних узлов (если они есть) так, как если бы удаляемого узла не было бы. Затем узел удаляется и память, которую он занимает, освобождается. Эти этапы показаны на рисунке внизу. Отдельно проверяется, не является ли удаляемый узел первым или последним узлом списка.

**OldNode**

данные

**NULL**

данные

**NULL**

данные

**NULL**

|  |
| --- |
| **void Delete(PNode &Head, PNode &Tail, PNode OldNode)**  **{**  **if (Head == OldNode) {**  **Head = OldNode->next; // удаляем первый элемент if ( Head )**  **Head->prev = NULL;**  **else Tail = NULL; // удалили единственный элемент**  **} else {**  **OldNode->prev->next = OldNode->next; if ( OldNode->next )**  **OldNode->next->prev = OldNode->prev;**  **else Tail = NULL; // удалили последний элемент**  **}**  **delete OldNode;**  **}** |

## Циклические списки

Иногда список (односвязный или двусвязный) замыкают в кольцо, то есть указатель **next**последнего элемента указывает на первый элемент, и (для двусвязных списков) указатель **prev**первого элемента указывает на последний. В таких списках понятие «хвоста» списка не имеет смысла, для работы с ним надо использовать указатель на «голову», причем «головой» можно считать любой элемент.