Лекция 5. Динамические структуры данных

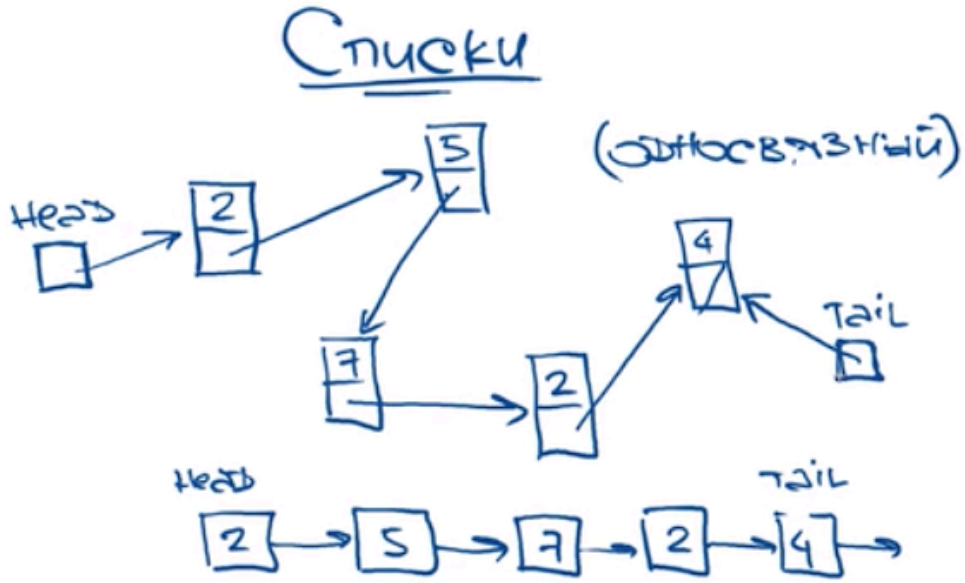
План лекции:

- Списки. Операции со списком.
- Стеки. Очереди. Деки.
- Деревья. Реализация деревьев.

Динамические структуры данных

- Часто в программах надо использовать такие типы данных, размер и структура которых должны меняться в процессе работы
- Динамические массивы здесь не выручают, поскольку заранее нельзя сказать, сколько памяти надо выделить
- Например, надо проанализировать текст и определить, сколько каких слов в нем встретилось, а также расставить их по алфавиту

Списки



Списки

- Ссылки представляют собой 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 устанавливаются связи этого элемента с уже существующими (с помощью ссылок). Для обозначения конечного элемента в цепи используются нулевые ссылки (***NULL*** в языке C++)

Связанный 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.

Связанный список

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 хранить адрес его первого узла – он называется **"головным элементом"** 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.

Связанный список

Правилами языка C++ допускается объявление

```
struct Node {
      char word[40];
      int count;
      Node *next;
};
typedef Node *PNode;
```

Данные

Указатель на следующий элемент

Тип «Указатель на элемент»

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

```
PNode Head;
```

Операции со списком

Создание 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 ***Word***. 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.

Операции со списком

```
Pnode CreateNode ( char Word[] )
{
     PNode NewNode = new Node;
     strcpy(NewNode->word, Word);
     NewNode->count = 1;
     NewNode->next = NULL;
     return NewNode;
}
```

После этого узел надо добавить к списку.

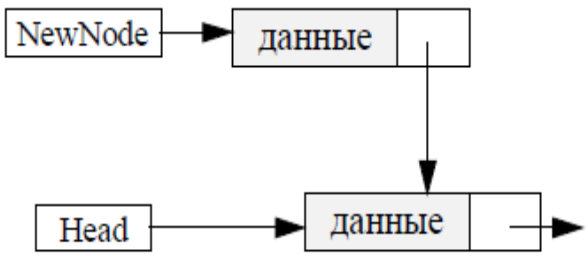
Операции со списком

Добавление узла в начало списка

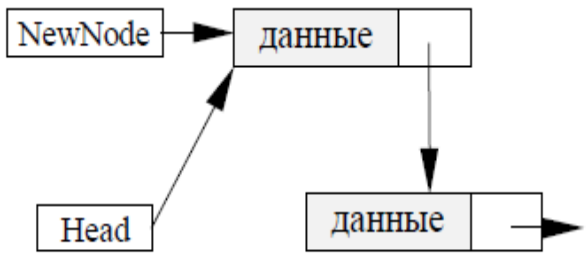
При добавлении нового узла ***NewNode*** в начало списка надо

- установить ссылку узла ***NewNode*** на головной элемент существующего списка
- установить головной элемент списка на новый узел.

1)



2)



Операции со списком

По такой схеме работает функция ***AddFirst***. Предполагается, что адрес начала списка хранится в глобальной переменной ***Head***.

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

Операции со списком

Проход по списку

Другие операции с односвязным списком требуют поиска заданного элемента в списке, то есть, надо просмотреть все элементы пока не найдется нужный. 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 работу, когда найден требуемый элемент или все элементы списка просмотрены.

Операции со списком

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

```
void Find (PNode Head, char Word[])
{
      PNode q = Head;
      while (q && strcmp(q->word, Word))
            q = q->next;
      return q;
}
```

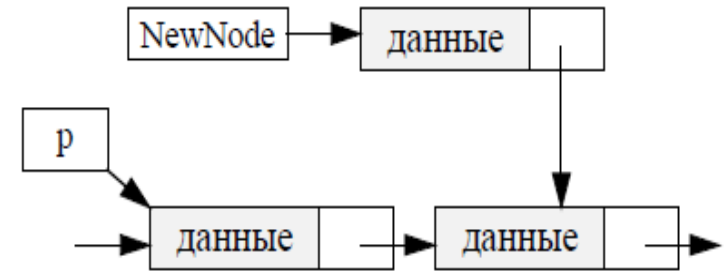
Операции со списком

Добавление узла после заданного

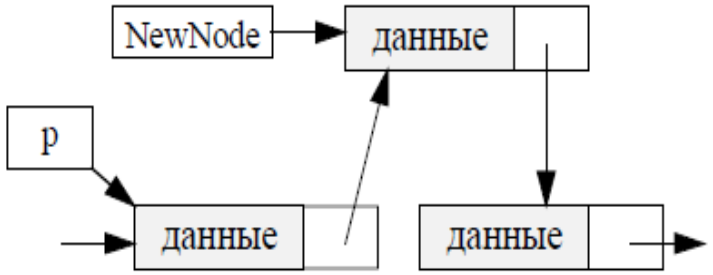
Дан адрес ***NewNode*** нового узла и адрес ***p*** одного из существующих узлов в списке. Требуется вставить в список новый узел после узла с адресом ***p***. Эта операция выполняется в два этапа:

1) установить ссылку нового узла на узел, следующий за данным;
2) установить ссылку данного узла ***p*** на ***NewNode***.

1)



2)



Операции со списком

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;
}
```

Операции со списком

Добавление узла перед заданным

Эта схема добавления самая сложная.

Проблема заключается в том, что в односвязном списке мы можем, зная адрес узла, получить адрес предыдущего узла, только пройдя весь список сначала. Задача сведется либо к вставке узла в начало списка (если заданный узел - первый), либо к вставке после заданного узла.

Операции со списком

```
void AddBefore(PNode p, PNode NewNode)
{
    PNode q = Head;
    if (Head == p) {
        AddFirst(Head, NewNode);
        return;
    }
    while (q && q->next!=p) q = q->next;
    if (q) AddAfter(q, NewNode);
}
```

Операции со списком

Такая процедура обеспечивает защиту - если задан узел, не присутствующий в списке, ничего не происходит.

Существует еще один интересный прием: если надо вставить новый узел ***NewNode*** до 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.

Операции со списком

Добавление узла в конец списка

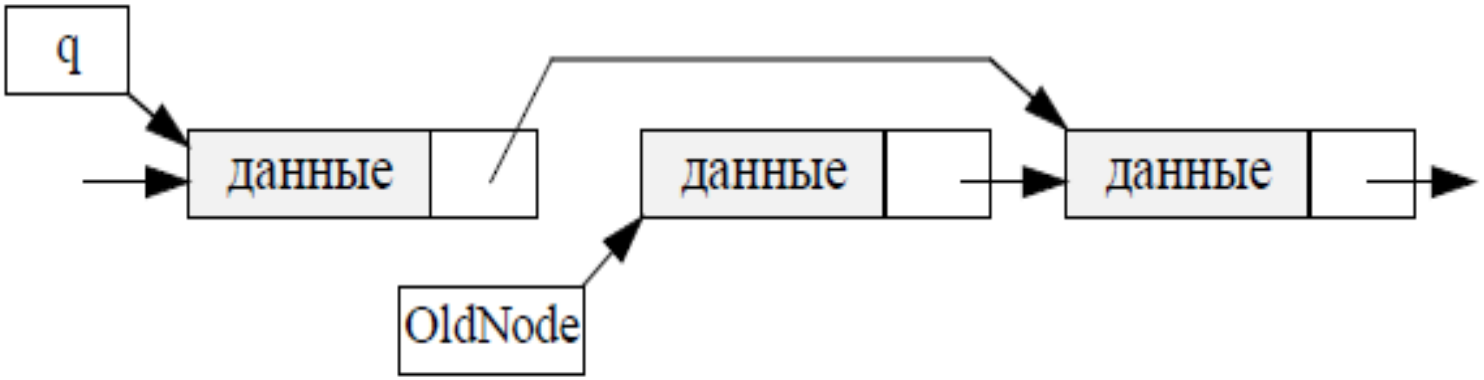
Для решения задачи нужно сначала найти последний узел, у которого ссылка равна *NULL*, а затем воспользоваться процедурой вставки после заданного узла. Отдельно надо обработать случай, когда список пуст.

```
void AddLast(PNode NewNode) {
    PNode q = Head;
    if (Head == NULL)  {
        AddFirst(Head, NewNode);
        return;
    }
    while (q->next) q = q->next;
    AddAfter(q, NewNode);
}
```

Операции со списком

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.



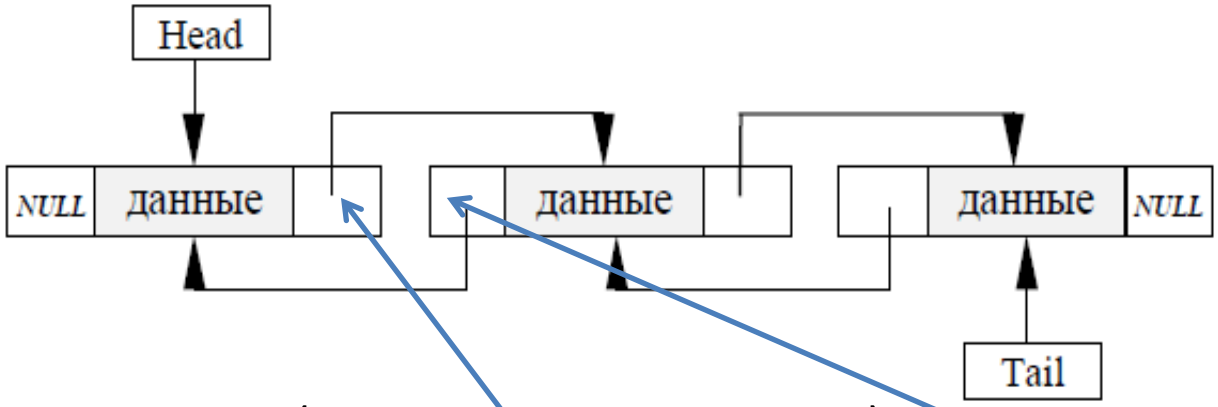
Операции со списком

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

```
void DeleteNode(PNode OldNode) {
    PNode q = Head;
    if (Head == OldNode) Head = OldNode->next;
    else {
        while (q && q->next != OldNode) q = q->next;
        if ( q == NULL ) return;
        q->next = OldNode->next;
    }
    delete OldNode;
}
```

Двусвязный список

Многие проблемы, присущие односвязным спискам, 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***).



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*.

Двусвязный список

Узел объявляется так:

```
struct Node {
    char word[40];
    int count;
    Node *next, *prev;
};
typedef Node *PNode;
```

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

```
PNode Head, Tail;
```

Для пустого списка оба указателя равны *NULL*.

Операции с двусвязным списком

Основными операциями являются добавление элемента в список и исключение из списка.

При этих операциях важно правильно работать со ссылками на предыдущие элементы узлов.

Операции с двусвязным списком

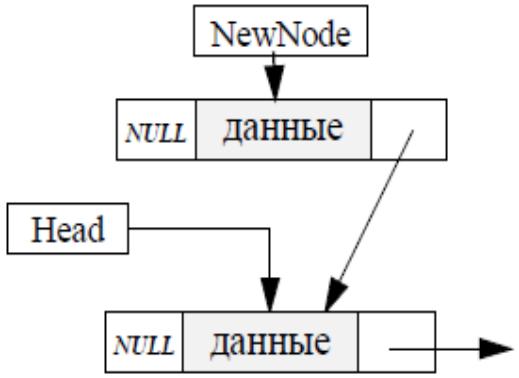
Добавление узла в начало списка

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

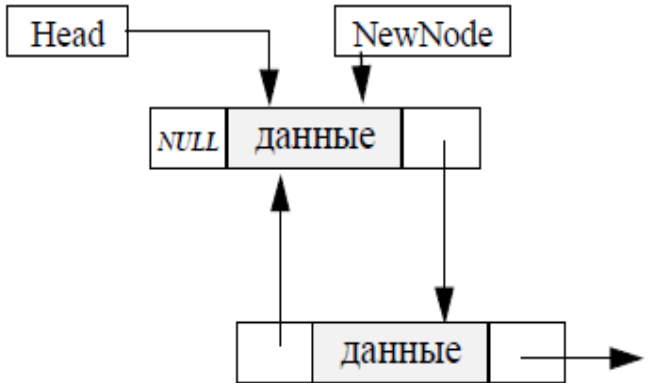
1) установить ссылку ***next*** узла *NewNode* на голову существующего списка и его ссылку ***prev*** в *NULL*;
2) установить ссылку ***prev*** бывшего первого узла (если он существовал) на *NewNode*;
3) установить голову списка на новый узел;
4) если в списке не было ни одного элемента, хвост списка также устанавливается на новый узел.

Операции с двусвязным списком

1-2)



3-4)



По такой схеме работает следующая функция:

```
void AddFirst(PNode NewNode) {
    NewNode->next = Head;
    NewNode->prev = NULL;
    if ( Head ) Head->prev = NewNode;
    Head = NewNode;
    if ( ! Tail ) Tail = Head;
}
```

Операции с двусвязным списком

Добавление узла в конец списка

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

Проход по списку

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

Операции с двусвязным списком

Добавление узла после заданного

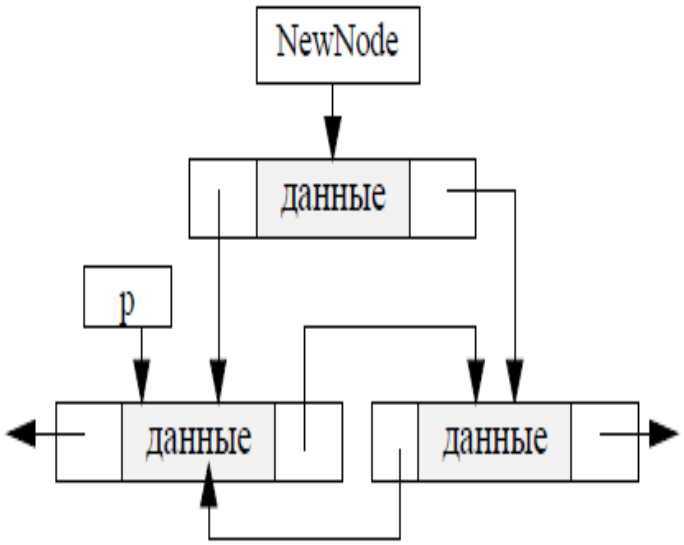
Дан адрес ***NewNode*** нового узла и адрес ***p*** одного из существующих узлов в списке.

Требуется вставить в список новый узел после заданного узла. Если данный узел является последним, то операция сводится к добавлению в конец списка. Если узел ***p*** - не последний, то операция вставки выполняется в два этапа:

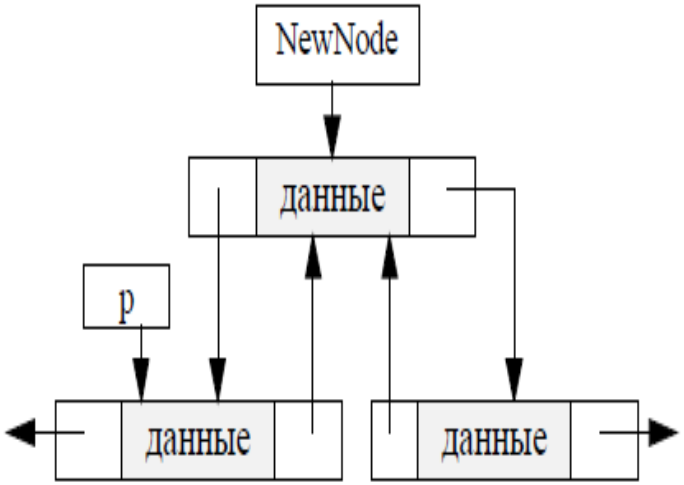
- установить ссылки нового узла на следующий за данным (***next***) и предшествующий ему (***prev***);
- установить ссылки соседних узлов так, чтобы включить ***NewNode*** в список.

Операции с двусвязным списком

1)



2)



Операции с двусвязным списком

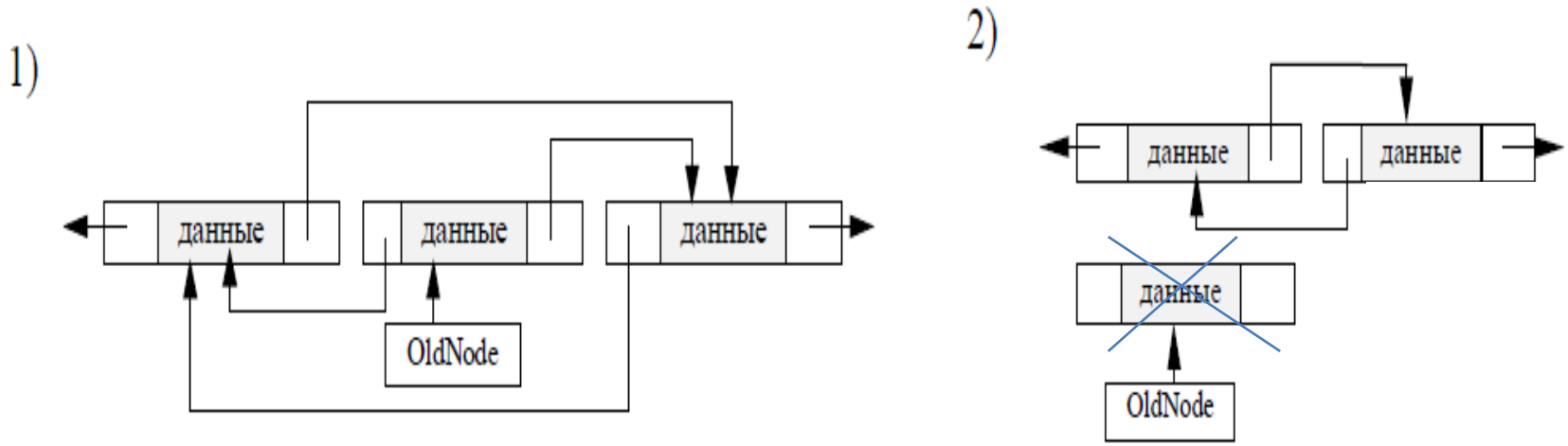
Такой метод реализует приведенная ниже функция (она учитывает также возможность вставки элемента в конец списка - именно для этого в параметрах передаются ссылки на голову и хвост списка)

```
void AddAfter (PNode p, PNode NewNode) {
    if ( ! p->next )
        AddLast (Head, Tail, NewNode);
    else {
        NewNode->next = p->next;
        NewNode->prev = p;
        p->next->prev = NewNode;
        p->next = NewNode;
    }
}
```

Операции с двусвязным списком

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. Затем узел удаляется и память, которую он занимает, освобождается. Эти этапы показаны на рисунке. Отдельно проверяется, не является ли удаляемый узел первым или последним узлом списка.



Операции с двусвязным списком

```
void Delete(PNode &Head, PNode &Tail, PNode OldNode)
{
    if (Head == OldNode) {
        Head = OldNode->next;
        if ( Head )
            Head->prev = NULL;
        else Tail = NULL;
    }
    else {
        OldNode->prev->next = OldNode->next;
        if ( OldNode->next )
            OldNode->next->prev = OldNode->prev;
        else Tail = NULL;
    }
    delete OldNode;
}
```

Операции с двусвязным списком

Циклические списки

Иногда список (односвязный или двусвязный) замыкают в кольцо, то есть указатель ***next*** последнего элемента указывает на первый элемент, а (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.

Стеки, очереди, деки

Данные структуры представляют собой списки, для которых разрешены только операции вставки и удаления первого и/или последнего элемента.

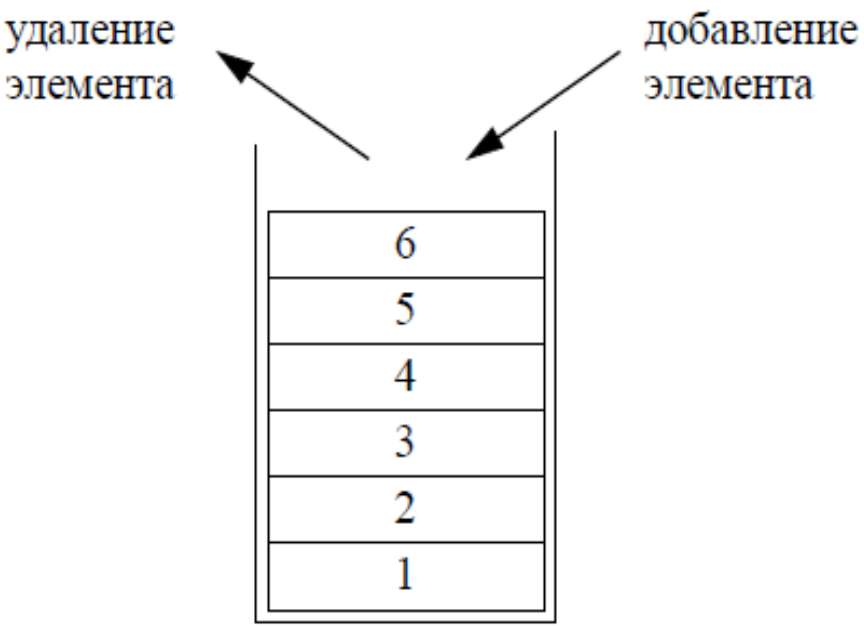
Стек

Стек - это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо только с одного конца, который называется вершиной стека.

Стек называют структурой типа ***LIFO*** (Last In - First Out) - последним пришел, первым ушел. Моделью стека может служить стопка бумаги - чтобы достать какой-то лист надо снять все листы, которые лежат на нем, а положить новый лист можно только сверху всей стопки.

Стек

На рисунке показан стек, содержащий 6 элементов.



Стек

В современных компьютерах стек используется для

- размещения локальных переменных
- размещения параметров процедуры или функции
- сохранения адреса возврата (по какому адресу надо вернуться из процедуры)
- временного хранения данных, особенно при программировании на Ассемблере

Стек

На стек выделяется ограниченная область памяти

При каждом вызове процедуры в стек добавляются новые элементы (параметры, локальные переменные, адрес возврата). Поэтому если вложенных вызовов будет много, стек переполнится.

Очень опасной в отношении переполнения стека является рекурсия, поскольку она как раз и предполагает вложенные вызовы одной и той же процедуры или функции. При ошибке в программе рекурсия может стать бесконечной, кроме того, стек может переполниться, если вложенных вызовов будет слишком много.

Стек

Рассмотрим пример стека, в котором хранятся целые числа. Для удобства будем реализовывать стек на основе двусвязного списка. Объявление узла списка и указателя на узел будет выглядеть так:

```
struct Node {
    int data;
    Node *next, *prev;
};
typedef Node *PNode;
```

Стек

Для удобства чтобы не работать с отдельными указателями на хвост и голову списка, объявим структуру, в которой будет храниться вся информация о стеке:

```
struct Stack {
      PNode Head, Tail;
};
```

В самом начале надо записать в обе ссылки стека *NULL*.

Для стека определены две основные операции:

Добавление элемента на вершину стека

Получение верхнего элемента с вершины стека

Стек

Добавление элемента на вершину стека

Фактически это добавление нового элемента в начало двусвязного списка. Эта процедура уже была написана ранее, теперь ее придется немного переделать, чтобы работать не с отдельными указателями, а со структурой типа *Stack*. В параметрах процедуры указывается не новый узел, а только данные для этого узла, то есть целое число. Память под новый узел выделяется в процедуре, то есть, скрыта от нас и снижает вероятность ошибки.

Стек

Добавление элемента на вершину стека

```
void Push ( Stack &S, int i )
{
        PNode NewNode;
        NewNode = new Node;
        NewNode->data = i;
        NewNode->next = S.Head;
        NewNode->prev = NULL;
        if ( S.Head ) S.Head->prev = NewNode;
        S.Head = NewNode;
        if ( ! S.Tail ) S.Tail = S.Head;
}
```

Стек

Получение верхнего элемента с вершины стека

Получение верхнего элемента и удаление его с вершины стека. Для этого надо написать функцию, которая удаляет верхний элемент и возвращает его данные

```
int Pop ( Stack &S )
{
    PNode TopNode = S.Head;
    int i;
    if ( ! TopNode ) return 0;
    i = TopNode->data;
    S.Head = TopNode->next;
    if ( S.Head ) S.Head->prev = NULL;
    else S.Tail = NULL;
    delete TopNode;
    return i;
}
```

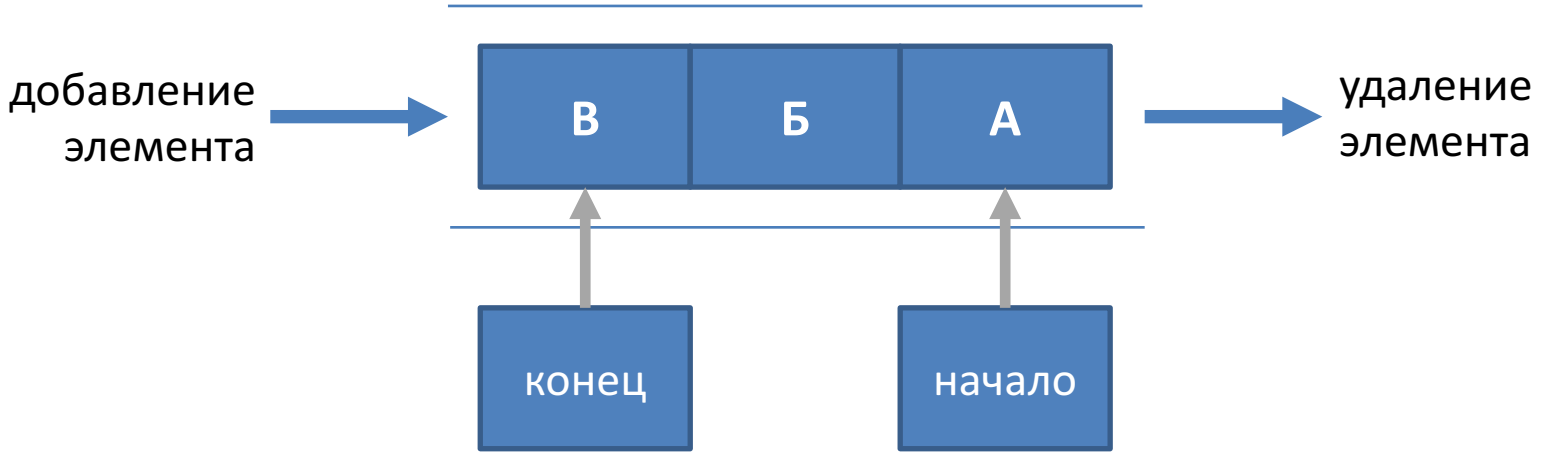
Очередь

Очередь - это упорядоченный набор элементов, в котором добавление новых элементов допустимо с одного конца (он называется начало очереди), а удаление существующих элементов - только с другого конца, который называется концом очереди.

Хорошо знакомой моделью является очередь в магазине. Очередь называют структурой типа ***FIFO*** (First In - First Out) - первым пришел, первым вышел.

Очередь

- На рисунке изображена очередь из 3-х элементов.



- Наиболее известные примеры применения очередей в программировании - очереди сообщений операционных систем *Windows, UNIX*. Очереди используются также для моделирования в задачах массового обслуживания (например, обслуживания клиентов в банке) и т.д.

Дек

Дек (*deque*) - это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо с любого конца.

Дек может быть реализован на основе стека. Остается добавить только две функции - добавление нового элемента в конец (*PushTail*) и получение последнего элемента с его удалением из дека (функция *PopTail*). Они легко получаются из уже написанных *Push* и *Pop* путем замены *Head* на *Tail* и *prev* на *next*, и наоборот.

Дек

```
void PushTail ( Stack &S, int i )
{
    PNode NewNode = new Node;
    NewNode->data = i;
    NewNode->prev = S.Tail;
    NewNode->next = NULL;
    if ( S.Tail ) S.Tail->next = NewNode;
    S.Tail = NewNode;
    if ( ! S.Head ) S.Head = S.Tail;
}
```

Дек

```
int PopTail ( Stack &S )
{
    PNode LastNode = S.Tail;
    int i;
    if ( ! LastNode ) return 0;
    i = LastNode->data;
    S.Tail = LastNode->prev;
    if ( S.Tail ) S.Tail->next = NULL;
    else S.Head = NULL;
    delete LastNode;
    return i;
}
```

Дек

Дек позволяет моделировать очередь или стек, для каждого типа списков допустимы свои функции. Так, для стека разрешены *Push* и *Pop*, а для очереди - *Push* и *PopTail* или *PushTail* и *Pop* (по выбору). Для полного дека доступны все четыре функции.

Деревья

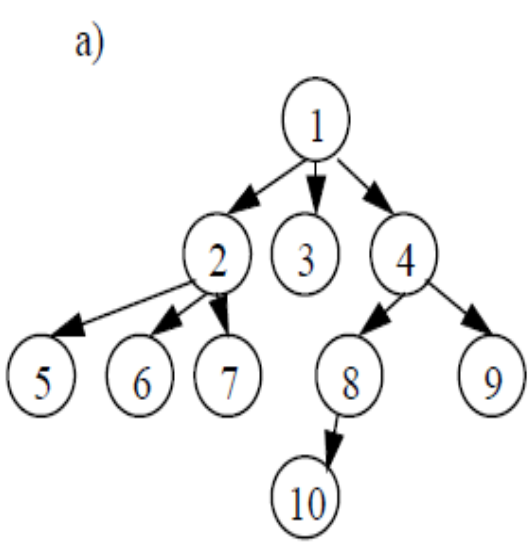
Дерево - это совокупность узлов (вершин) и соединяющих их направленных ребер (дуг), причем в каждый узел (за исключением одного - корня) ведет ровно одна дуга.

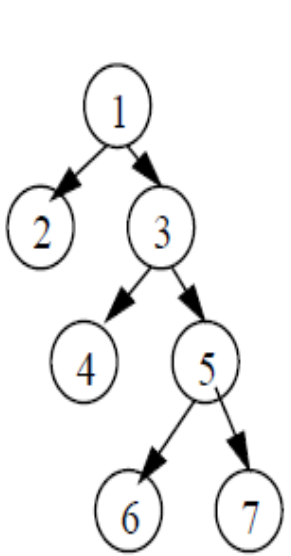
Корень - это начальный узел дерева, в который не ведет ни одной дуги.

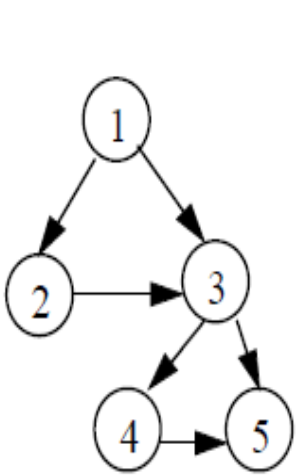
Примером может служить генеалогическое дерево - в корне дерева находитесь вы сами, от вас идет две дуги к родителям, от каждого из родителей - две дуги к их родителям и т.д.

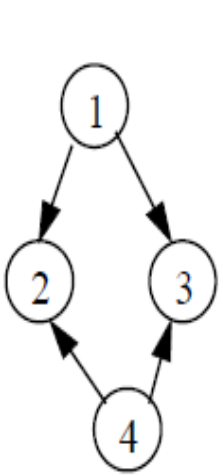
Деревья

Например, на рисунке структуры **а)** и **б)** являются деревьями, а **в)** и **г)** - нет









Деревья

Предком узла ***x*** называется узел дерева, из которого существует путь в узел ***x***.

Потомком узла ***x*** называется узел дерева, в который существует путь из узла ***x***.

Родителем узла ***x*** называется узел дерева, из которого существует непосредственная дуга в узел ***x***.

Сыном (дочерним элементом) узла ***x*** называется узел дерева, в который существует непосредственная дуга из узла ***x***.

Деревья

Уровнем узла *x* называется длина пути (количество дуг) от корня к данному узлу. Считается, что корень находится на уровне 0.

Листом дерева называется узел, не имеющий потомков.

Внутренней вершиной называется узел, имеющий потомков.

Высотой дерева называется максимальный уровень листа дерева.

Упорядоченным деревом называется дерево, все вершины которого упорядочены (то есть имеет значение последовательность перечисления потомков каждого узла).

Деревья

Рекурсивное определение

Дерево представляет собой типичную рекурсивную структуру (определяемую через саму себя). Как и любое рекурсивное определение, определение дерева состоит из двух частей - первая определяет условие окончания рекурсии, а второе - механизм ее использования.

- пустая структура является деревом
- дерево - это корень и несколько связанных с ним деревьев (поддеревьев)

Таким образом, **размер памяти**, необходимый для хранения дерева, **заранее неизвестен**, потому что **неизвестно, сколько узлов будет в него входить**.

Двоичные деревья

На практике используются главным образом деревья особого вида, называемые двоичными **(бинарными)**.

Двоичным деревом называется дерево, каждый узел которого имеет не более двух дочерних узлов

Можно определить двоичное дерево и рекурсивно:

1) пустая структура является двоичным деревом
2) дерево - это корень и два связанных с ним двоичных дерева, которые называют левым и правым поддеревом

Двоичные деревья

Двоичные деревья упорядочены, то есть различают левое и правое поддеревья. Типичным примером двоичного дерева является генеалогическое дерево. В других случаях двоичные деревья используются тогда, когда на каждом этапе некоторого процесса надо принять одно решение из двух возможных.

Строго двоичным деревом называется дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Это означает, что в строго двоичном дереве нет вершин, у которых есть только одно поддерево.

Двоичные деревья

Полным двоичным деревом называется дерево, у которого все листья находятся на одном уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Реализация деревьев

Описание вершины

Вершина дерева, как и узел любой динамической структуры, имеет две группы данных: полезную информацию и ссылки на узлы, связанные с ним.

Для двоичного дерева таких ссылок будет две – ссылка на левого сына и ссылка на правого сына. В результате получаем структуру, описывающую вершину (предполагая, что полезными данными для каждой вершины является одно целое число):

```
struct Node {
    int Key;
    Node *Left, *Right;
};
typedef Node *PNode;
```

Реализация деревьев

Идеально сбалансированные деревья

Для большинства практических задач наиболее интересны такие деревья, которые имеют минимально возможную высоту при заданном количестве вершин *n*. Очевидно, что минимальная высота достигается тогда, когда на каждом уровне (кроме, возможно, последнего) будет максимально возможное число вершин.

Дерево называется ***идеально сбалансированным***, если число вершин в его левом и правом поддеревьях отличается не более чем на 1.

Реализация деревьев

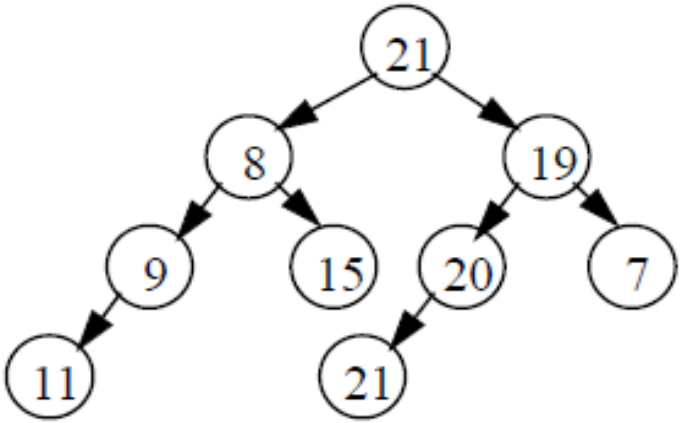
Построение идеально сбалансированных деревьев

Предположим, что задано *n* чисел. Требуется построить из них идеально сбалансированное дерево. Алгоритм решения этой задачи предельно прост:

- Взять одну вершину в качестве корня и записать в нее первое нерассмотренное число.
- Построить этим же способом идеально сбалансированное левое поддерево из ***n1=n/2*** вершин (целая часть от деления).
- Построить этим же способом идеально сбалансированное правое поддерево из ***n2=n-n1-1*** вершин.

Реализация деревьев

- Заметим, что по построению левое поддерево всегда будет содержать столько же вершин, сколько правое поддерево, или на 1 больше.
- Для массива данных **21, 8, 9, 11, 15, 19, 20, 21, 7** по этому алгоритму строится идеально сбалансированное дерево, показанное на рисунке.



Реализация деревьев

В основной программе нам надо объявить указатель на корень нового дерева, задать массив данных (можно читать данные из файла) и вызвать функцию, возвращающую указатель на построенное сбалансированное дерево.

```
int data[] = { 21, 8, 9, 11, 15, 19, 20, 21, 7 };
PNode Tree;
n = sizeof(data) / sizeof(int) - 1;
Tree = MakeTree (data, 0, n);
```

Функция ***MakeTree*** принимает три параметра: массив данных, номер первого неиспользованного элемента и количество элементов в новом дереве.

Возвращается указатель на новое дерево (типа ***PNode***).

Реализация деревьев

```
PNode MakeTree (int data[], int &from, int n)
{
    PNode Tree;
    int n1, n2;
    if ( n == 0 ) return NULL;
    Tree = new Node;
    Tree->Key = data[from++];
    n1 = n / 2;
    n2 = n - n1 - 1;
    Tree->Left = MakeTree(data, from, n1);
    Tree->Right = MakeTree(data, from, n2);
    return Tree;
}
```

Реализация деревьев

- Номер первого невыбранного элемента (параметр ***from***) надо передавать по ссылке (объявлять со знаком ***&***), потому что он изменяется при каждом новом рекурсивном вызове.
- Другой вариант - сделать массив ***data*** и переменную ***from*** глобальными и исключить их из списка параметров. Однако при этом теряется гибкость процедуры - очень сложно будет в одной программе строить несколько разных деревьев с разными данными, поскольку процедура будет жестко привязана к глобальным переменным.

Реализация деревьев

Обход дерева

Одной из необходимых операций при работе с деревьями является обход дерева, во время которого надо посетить каждый узел по одному разу и (возможно) вывести информацию, содержащуюся в вершинах.

Пусть в результате обхода надо напечатать значения поля данных всех вершин в определенном порядке. Существуют три варианта обхода рассматриваемых далее.

Реализация деревьев

Первый вариант обхода

- Просмотр в ширину (сверху вниз), при котором сначала посещается корень (выводится информация о нем), затем левое поддерево, а затем - правое. Такой обход называют обходом типа **КЛП** (корень - левое - правое), другое название - ***префиксный (прямой) обход***.

Реализация деревьев

Второй вариант обхода

- Просмотр в симметричном порядке (слева направо), при котором сначала посещается левое поддерево, затем корень, а затем - правое. Такой обход называют обходом типа ЛКП (левое - корень - правое), другое название - ***инфиксный (симметричный) обход***.

Реализация деревьев

Третий вариант обхода

- Просмотр снизу вверх, при котором сначала посещается левое поддерево, затем правое, а затем - корень. Такой обход называют обходом типа ЛПК (левое - правое - корень), другое название - ***постфиксный*** ***(обратный) обход***.

Реализация деревьев

- Пример рекурсивной процедуры просмотра дерева слева направо. Обратите внимание, поскольку дерево является рекурсивной структурой данных, при работе с ним естественно широко применять рекурсию.

```
void PrintLKP(PNode Tree)
{
    if ( ! Tree ) return;
    PrintLKP(Tree->Left);
    printf("%d ", Tree->Key);
    PrintLKP(Tree->Right); // рекурсия
}
```

Остальные варианты обхода программируются аналогично

Реализация деревьев

Сортировка и поиск с помощью дерева

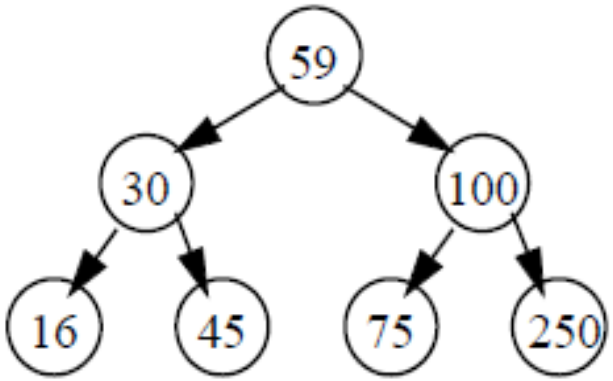
- Деревья очень удобны для поиска в них информации. Предположим, что существует массив данных и с каждым элементом связан ключ - число, по которому выполняется поиск. Пусть ключи для элементов таковы: **59,100,75,30,16,45,250**.
- Для этих данных нам надо много раз проверять, присутствует ли среди ключей заданный ключ ***x***, и если присутствует - вывести всю связанную с этим элементом информацию.

Реализация деревьев

- Если данные организованы в виде неотсортированного массива, то для поиска в худшем случае надо сделать ***n*** сравнений элементов “алгоритм имеет сложность порядка O(n)” (сравнивая последовательно с каждым элементом, пока не найдется нужный или пока не закончится массив).

Реализация деревьев

Теперь предположим, что данные организованы в виде дерева



Такое дерево обладает следующим важным свойством:

- Значения ключей всех вершин левого поддерева вершины *x* меньше ключа *x*, а значения ключей всех вершин правого поддерева *x* больше или равно ключу вершины *x*.

Реализация деревьев

Для поиска нужного элемента в таком дереве требуется не более 3 сравнений вместо 7 при поиске в списке или массиве, то есть поиск проходит значительно быстрее.

Как же, имея массив данных, построить такое дерево?

1. Сравнить ключ очередного элемента массива с ключом корня.
2. Если ключ нового элемента меньше, включить его в левое поддерево, если больше или равен, то в правое.
3. Если текущее дерево пустое, создать новую вершину и включить в дерево.

Реализация деревьев

Приведенная программа реализует этот алгоритм

```
void AddToTree (PNode &Tree, int data)
{
    if ( ! Tree )
    {
       Tree = new Node;
       Tree->Key = data;
       Tree->Left = NULL;
       Tree->Right = NULL;
       return;
    }
    if ( data < Tree->Key )
       AddToTree ( Tree->Left, data );
    else
       AddToTree ( Tree->Right, data );
}
```

Реализация деревьев

Важно, что указатель на корень дерева надо передавать по ссылке, так как он может измениться при создании новой вершины. Чтобы получить все ключи по возрастанию, надо пройти дерево слева направо, распечатывая ключи вершин.

Надо заметить, что в результате работы этого алгоритма не всегда получается дерево минимальной высоты – все зависит от порядка выбора элементов.

Для оптимизации поиска используют так называемые сбалансированные или АВЛ-деревья (но не идеально сбалансированные) деревья, у которых для любой вершины, высоты левого и правого поддеревьев отличаются не более чем на 1. Добавление в них нового элемента иногда сопровождается некоторой перестройкой дерева.

Реализация деревьев

Приведенный алгоритм можно модифицировать так, чтобы быстро искать одинаковые элементы в массиве чисел.

Один из способов решения этой задачи - перебрать все элементы массива и сравнить каждый со всеми остальными. Однако для этого требуется очень большое число сравнений.

С помощью двоичного дерева можно значительно ускорить поиск.

Для этого надо в структуру вершины включить еще одно поле - счетчик найденных дубликатов *count*.

```
struct Node {
    int Key;
    int Count;
    Node *Left, *Right;
};
```

Реализация деревьев

При создании узла в счетчик записывается единица (найден один элемент). Поиск дубликатов происходит по следующему алгоритму:

1) Сравнить ключ очередного элемента массива с ключом корня.
2) Если ключ нового элемента равен ключу корня, то увеличить счетчик корня и стоп.
3) Если ключ нового элемента меньше, включить его в левое поддерево, если больше или равен - в правое.
4) Если текущее дерево пустое, создать новую вершину (со значением счетчика 1) и включить в дерево.

Реализация деревьев

Теперь, когда дерево сортировки построено, очень легко искать элемент с заданным ключом.

- Сначала проверяем ключ корня, если он равен искомому, то нашли.
- Если он меньше искомого, ищем в левом поддереве корня, если больше - то в правом.

Реализация деревьев

Приведенная функция возвращает адрес нужной вершины, если поиск успешный, и *NULL*, если требуемый элемент не найден.

```
PNode Search (PNode Tree, int what)
{
    if ( ! Tree ) return NULL;
    if ( what == Tree->Key ) return Tree;
    if ( what < Tree->Key )
        return Search ( Tree->Left, what );
    else
        return Search ( Tree->Right, what );
}
```

Контейнеры в C++

Для управления наборами объектов в стандартной библиотеке C++ определены контейнеры. Контейнер представляет коллекцию объектов определенного типа. Последовательный контейнер (sequential container) позволяет контролировать порядок, в котором элементы располагаются в коллекции, и управлять доступом к этим элементам.

- Таким образом, стандартная библиотека C++ по умолчанию содержит ряд контейнеров, которые представляют определенные структуры данных. За исключением класса array все они поддерживают добавление и удаление элементов. Основное различие между ними состоит в том, как они обеспечивают добавление и удаление элементов, а также доступ к элементам в контейнере. И в зависимости от ситуации и потребностей можно использовать тот или иной тип контейнеров
- Для использования определенного контейнера в программу необходимо добавить соответствующий заголовочный файл, который, как правило, называется по имени класса контейнера.

Типы последовательных контейнеров

vector: массив переменного размера. Поддерживает произвольный доступ к любому элементу в контейнере. Обеспечивает добавление и удаление элементов из любого места контейнера.

deque: двусторонняя очередь. Поддерживает произвольный доступ к любому элементу в контейнере. Обеспечивает удаление и добавление элементов в начале и в конце контейнера.

list: двухсвязный список. Поддерживает только последовательный двунаправленный доступ к элементам. Обеспечивает удаление и добавление элементов в начале и в конце контейнера.

forward_list: односвязный список. Поддерживает только однонаправленный последовательный доступ к элементам. Обеспечивает удаление и добавление элементов в начале и в конце контейнера.

array: массив фиксированного размера. Поддерживает произвольный доступ к любому элементу в контейнере. Добавлять или удалять элементы из контейнера нельзя.

string: представляет контейнер, аналогичный вектору, который состоит из символов, то есть строку.

Важное замечание!

Подробно изучать работу с контейнерами данных вы будете в следующем курсе «Объектно-ориентированное программирование».
В нашем курсе работу с динамическими структурами данных (списками, деревьями) нужно **выполнить с использованием типа данных структура (struct) и указателей.**

Реализация заданий с использованием контейнеров данных возможна только как дополнительное решение, за которое могут быть начислены баллы только за творческий рейтинг.