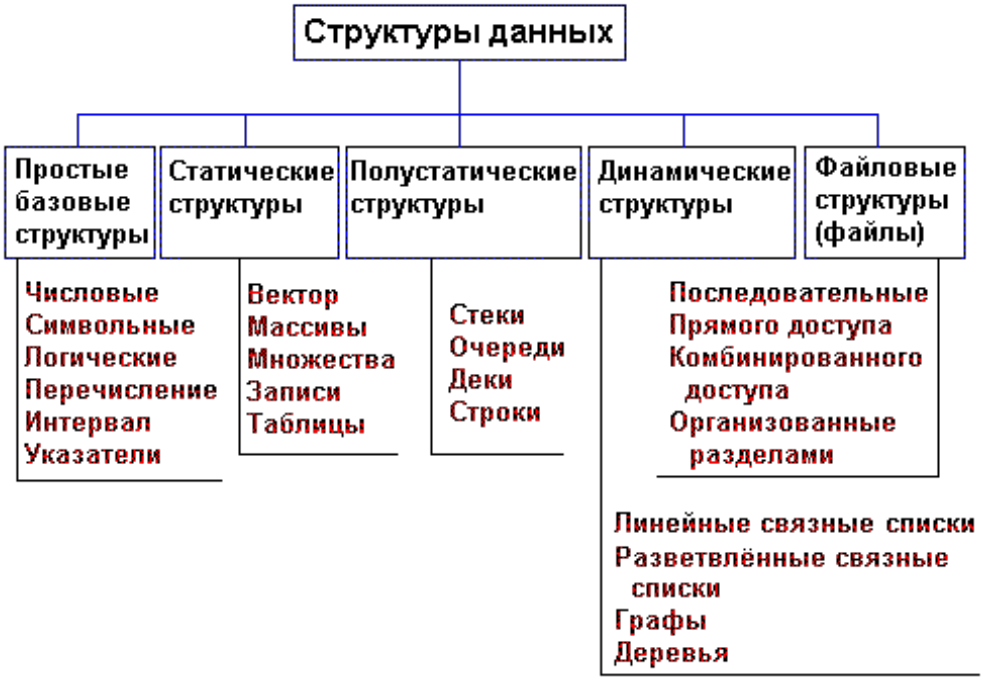
1. Общая классификация структур.



2. Оценка эффективности алгоритмов, временная и емкостная сложность алгоритмов.

Эффективность программы имеет две составляющие: память (или пространство) и время.

Пространственная эффективность измеряется количеством памяти, требуемой для выполнения программы.

Временная эффективность программы определяется временем, необходимым для ее выполнения.

Лучший способ сравнения эффективностей алгоритмов состоит в сопоставлении их порядков сложности. Этот метод применим как к временной, так и пространственной сложности. Порядок сложности алгоритма выражает его эффективность обычно через количество обрабатываемых данных.

3. Выделение памяти под статические и динамические массивы. Расчет адресов памяти элементов массива с использованием дескриптора массива. Вектора Айлиффа.

Память для статических массивов выделяется до начала выполнения программы. Для динамических массивов память выделяется в ходе выполнения программы.

Дескриптор массива содержит: Имя вектора (Vect), Адрес 1-го элемента (Addr(Vect[i1])) Индекс 1-го элемента (i1) , Индекс последнего элемента (iK), Длина элемента (L) , Тип (T).

Расчет адреса памяти этемента J массива vect:

Addr(vect[J]) = Addr(vect[i1]) +L*(J-i1)=Addr(vect[i1])-L*i1+L*J.

Представление массива с помощью векторов Айлиффа:

Для массива любой мерности формируется набор дескрипторов: основного и несколько уровней вспомогательных дескрипторов, называемых векторами Айлиффа. Каждый вектор Айлиффа определенного уровня содержит указатель на нулевые компоненты векторов Айлиффа следующего, более низкого уровня, а векторы Айлиффа самого нижнего уровня содержат указатели групп элементов отображаемого массива. Основной дескриптор массива хранит указатель вектора Айлиффа первого уровня. При такой организации к произвольному элементу B(j1,j2,...,jn) многомерного массива можно обратиться пройдя по цепочке от основного дескриптора через соответствующие компоненты векторов Айлиффа.

4. Стек. Доступ к элементу стека, адресация элемента. Алгоритмы исключения и включения элемента стека. Варианты конкретной реализации стека.

Зачастую стек реализуется в виде однонаправленного списка(каждый элемент списка указывает только на следующий). Но в таком случае невозможно применить операцию обхода элементов. А доступ возможен только к верхнему элементу структуры.

Пример реализации стека на языке Си:

```
struct sNode
{
    char* data;
    sNode* next;
};
```

При проталкивании(*push*) указывается новый элемент, указывающий на элемент бывший до этого головой. Новый элемент теперь становится головным.

При удалении элемента убирается первый, а головным становится тот, на который был указатель у этого объекта(следующий элемент).

```
sNode* head = NULL;

void push(char* tmp)
{
      sNode* node = new sNode;
      node->data = tmp;
      node->next = head;

      head = node;
 }

 char* pop()
 {
      char* tmp;
      tmp = head->data;
      head = head->next;

      return tmp;
 }
```

5. Очереди. Алгоритмы включения, исключения элемента и очистки очереди. Конкретная реализация очереди

Óчередь — структура данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» . Добавление элемента возможно лишь в конец очереди, выборка — только из начала очереди при этом выбранный элемент из очереди удаляется.

```
struct node //описание структуры
{
      char* data;
      node *next;
};

node* First = NULL;
node* Last = NULL;

void push(char* val)//добавить узел в очередь
{
    node *a = new node;
    a->data = val;
    a->next = NULL;
    if(First == NULL)//если это первый элемент то и превый и последний указывают на него
        Last = First = a;
    else
      {
              Last->next = a;
              Last = a;
      }
}

char* pop()//вытолкнуть узел из очереди
{
      if(First == NULL)
              return 0;// спсок пуст
      node *a = First;
      char* val = a->data;
```

```
        First = a->next;
        if(First == NULL)
                Last = NULL;//в списке всего один элемент
        delete a;
        return val;
}
```

1. Двусвязный список. Алгоритм включения, исключения элемента из двусвязного списка. Многосвязные списки.

В двухсвязнном списке - ссылки в каждом узле указывают на предыдущий и на последующий узел в списке. По двусвязному списку можно передвигаться в любом направлении — как к началу, так и к концу. В этом списке проще производить удаление и перестановку элементов, так как всегда известны адреса тех элементов списка, указатели которых направлены на изменяемый элемент.

```
struct node //описание структуры
{
                char* data;
                node *next;
                node *prev;
};

node* First = NULL;
node* Last = NULL;

void Add(char* val)
{
  node *temp=new node; //Выделение памяти под новый элемент структуры
  temp->next=NULL;  //Указываем, что изначально по следующему адресу пусто
  temp->data=val;//Записываем значение в структуру

  if (First!=NULL) //Если список не пуст
  {
    temp->prev=Last; //Указываем адрес на предыдущий элемент в соотв. поле
    Last->next=temp; //Указываем адрес следующего за хвостом элемента
     Last=temp; //Меняем адрес хвоста
  }
  else //Если список пустой
  {
     temp->prev=NULL; //Предыдущий элемент указывает в пустоту
     First=Last=temp; //Голова=Хвост=тот элемент, что сейчас добавили
} };
void Show()
{
      //ВЫВОДИМ СПИСОК С КОНЦА
    node *temp=Last;
     //Временный указатель на адрес последнего элемента
    while (temp!=NULL) //Пока не встретится пустое значение
    {
    printf("%s ", temp->data); //Выводить значение на экран
    temp=temp->prev; //Указываем, что нужен адрес предыдущего элемента
    }
    printf("\n");

 //ВЫВОДИМ СПИСОК С НАЧАЛА
    temp=First; //Временно указываем на адрес первого элемента
     while (temp!=NULL) //Пока не встретим пустое значение
    {
    printf("%s ", temp->data); //Выводим каждое считанное значение на экран
    temp=temp->next; //Смена адреса на адрес следующего элемента
    }
    printf("\n");
```

};

Многосвязные списки: **Многосвязные списки** представляют собой динамические структуры данных, в основу которых положены 1- или 2-связные списки, в которых имеются дополнительные связи между звеньями. Чаще всего, такие связи проводятся между далеко отстоящими звеньями, например, обозначающими категории данных. Пример многосвязного списка показан на следующем рисунке.

2. **Рекурсивные процедуры и функции.**

В программировании рекурсия — вызов функции (процедуры) из неё же самой, непосредственно (*простая рекурсия*) или через другие функции (*сложная* или *косвенная рекурсия*), например, функция A вызывает функцию B, а функция B — функцию A. Количество вложенных вызовов функции или процедуры называется глубиной рекурсии.

Преимущество рекурсивного определения объекта заключается в том, что такое конечное определение теоретически способно описывать бесконечно большое число объектов. С помощью рекурсивной программы же возможно описать бесконечное вычисление, причём без явных повторений частей программы. Реализация рекурсивных вызовов функций в практически применяемых языках и средах программирования, как правило, опирается на механизм стека вызовов — адрес возврата и локальные переменные функции записываются в стек, благодаря чему каждый следующий рекурсивный вызов этой функции пользуется своим набором локальных переменных и за счёт этого работает корректно. Оборотной стороной этого довольно простого по структуре механизма является то, что на каждый рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине рекурсии может наступить переполнение стека вызовов. Вследствие этого, обычно рекомендуется избегать рекурсивных программ, которые приводят (или в некоторых условиях могут приводить) к слишком большой глубине рекурсии. Критерии выбора для разработки рекурсивных или итеративных алгоритмов: рекурсивные подпрограммы могут приводить к бесконечным вычислениям. необходимо следить, чтобы всегда был нерекурсивный выход. Необходимо убедиться, что максимальная глубина рекрсии не только конечна, но и достаточно мала...

3. Рекурсивные типы данных. Примеры описаний и использования.

Описание типа данных может содержать ссылку на саму себя. Подобные структуры используются при описании списков и графов. Пример описания списка (C++):

```
struct element_of_list
{
  element_of_list *next; /* ссылка на следующий элемент того же типа */
  int data; /* некие данные */
};
```

Рекурсивная структура данных зачастую обуславливает применение рекурсии для обработки этих данных

4. Понятие абстрактных типов данных. Обработка абстрактных типов данных (АТД). Принципы создания программ с использованием АТД.

Абстра́ктный тип да́нных (АТД) — это тип данных, который предоставляет для работы с элементами этого типа определённый набор функций, а также возможность создавать элементы этого типа при помощи специальных функций. Вся внутренняя структура такого типа спрятана от разработчика программного обеспечения — в этом и заключается суть абстракции. Абстрактный тип данных определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями. Конкретные реализации АТД называются структурами данных.

В программировании абстрактные типы данных обычно представляются в виде интерфейсов, которые скрывают соответствующие реализации типов. Программисты работают с абстрактными типами данных исключительно через их интерфейсы, поскольку реализация может в будущем измениться. Такой подход соответствует принципу инкапсуляции в объектно-ориентированном программировании. Сильной стороной этой методики является именно сокрытие реализации. Раз вовне опубликован только интерфейс, то пока структура данных поддерживает этот интерфейс, все программы, работающие с заданной структурой абстрактным типом данных, будут продолжать работать. Разработчики структур данных стараются, не меняя внешнего интерфейса и семантики функций, постепенно дорабатывать реализации, улучшая алгоритмы по скорости, надежности и используемой памяти.

Различие между абстрактными типами данных и структурами данных, которые реализуют абстрактные типы, можно пояснить на следующем примере. Абстрактный тип данных список может быть реализован при помощи массива или линейного списка, с использованием различных методов динамического выделения памяти. Однако каждая реализация определяет один и тот же набор функций, который должен работать одинаково (по результату, а не по скорости) для всех реализаций.

Абстрактные типы данных позволяют достичь модульности программных продуктов и иметь несколько альтернативных взаимозаменяемых реализаций отдельного модуля.

5. Разреженные матрицы, примеры их использования.

Разреженная матрица — это матрица с преимущественно нулевыми элементами. Методы хранения:

Например, линейный связный список, т.е. последовательность ячеек, связанных в определенном порядке. Каждая ячейка списка содержит элемент списка и указатель на положение следующей ячейки. Можно хранить матрицу, используя кольцевой связный список, двунаправленные стеки и очереди. Существует диагональная схема хранения симметричных матриц, а также связные схемы разреженного хранения.

Связная схема хранения матриц, предложенная Кнутом, предлагает хранить в массиве (например, в AN) в произвольном порядке сами элементы, индексы строк и столбцов соответствующих элементов (например, в массивах I и J), номер (из массива AN) следующего ненулевого элемента, расположенного в матрице по строке (NR) и по столбцу (NC), а также номера элементов, с которых начинается строка (указатели для входа в строку – JR) и номера элементов, с которых начинается столбец (указатели для входа в столбец – JC). Данная схема хранения избыточна, но позволяет легко осуществлять любые операции с элементами матрицы.

Наиболее широко используемая схема хранения разреженных матриц – это схема, предложенная Чангом и Густавсоном, называемая: "разреженный строчный формат". Эта схема предъявляет минимальные требования к памяти и очень удобна при выполнении операций сложения, умножения матриц, перестановок строк и столбцов, транспонирования, решения систем линейных уравнений, при хранении коэффициентов в разреженных матрицах и т.п. В этом случае значения ненулевых элементов хранятся в массиве AN, соответствующие им столбцовые индексы – в массиве JA. Кроме того, используется массив указателей, например IA, отмечающих позиции AN и JA, с которых начинаются описание очередной строки. Дополнительная компонента в IA содержит указатель первой свободной позиции в JA и AN.

При решении многих прикладных задач приходится иметь дело с разреженными матрицами, то есть с матрицами, имеющими много нулевых элементов. К числу таких задач в первую очередь следует отнести граничные задачи для систем дифференциальных уравнений в частных производных. Возникающие при этом модели - это, как правило, квадратные матрицы высокого порядка с конечным числом ненулевых диагоналей. Известно, что матрица порядка n требует для своего хранения n2 байт оперативной памяти, а время вычислений пропорционально n3. Поэтому, когда количество неизвестных переменных достигает нескольких сотен, вычисления с полными матрицами становятся неэффективными и необходимо принимать во внимание степень разреженности матрицы, то есть отношение количества ненулевых элементов к общему количеству элементов матрицы.

6. Деревья. Виды деревьев и способы их представления.

Деревья – нелинейные структуры данных, использующиеся для представления иерархич. связей, имеющих отношение «один ко многим». Деревья определяются рекурсивно: дерево с базовым типом T – пустое (один корень). Либо узел типа T с конечным числом древовидных структур типа T.. Виды деревьев и способы их представления. 1)В виде вложенных множеств. 2) скобочное представление (A(B(D(I),E(J,K,L)),C(F(O),G(M,N),H(P)))) 3) списки достижимости и смежности 4) графы. Представление в памяти: 1) в виде курсоров на родителей 2) связный список сыновей 3) структуры данных. Виды: двоичное, >2-сильно ветвящееся, полное бинарное (все узлы степени 2 и листья имеют хотя бы одного левого сына), идеально сбалансированное (левое и правое поддеревья отличаются по количеству вершин не больше, чем на 1)

Дерево – это нелинейная структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим».

Виды

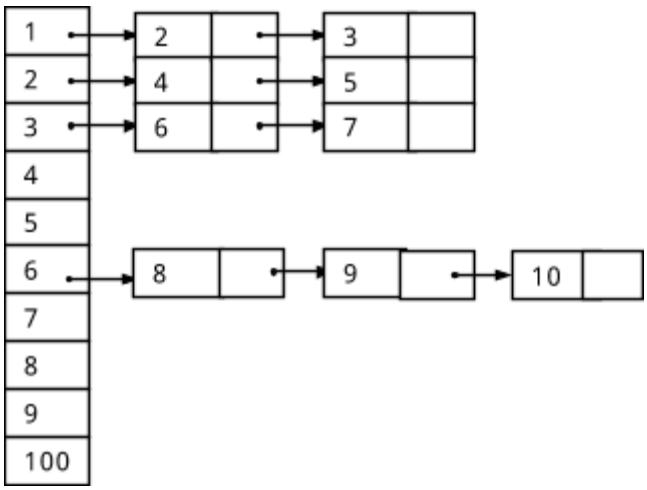
Рис.1

В памяти деревья можно представить в виде связей с предками (еще их называют родителями); связного списка потомков (сыновей) или структуры данных.

Представление указанного дерева (см. рис 1) в виде связей с предками:

№ вершина	1	2	3	4	5	6	7	8	9	10
Родитель	0	1	1	2	2	3	3	6	6	6

Пример представления этого же дерева в виде связного списка сыновей приведен на рис 2:



12. Двоичные деревья. Обход двоичных деревьев. Двоичные деревья поиска (ДДП). Алгоритмы поиска в ДДП.

Дерево двоичного поиска – это такое дерево, в котором все левые потомки моложе предка, а все правые – старше.

Если при построении дерева поочередно располагать узлы слева и справа, то получится дерево, у которого число вершин в левом и правом поддеревьях отличается не более чем на единицу. Такое дерево называется идеально сбалансированным.

N элементов можно организовать в бинарное дерево с высотой не более $\log_2(N)$, поэтому для поиска среди N элементов может потребоваться не больше $\log_2(N)$ сравнений

Поиск элемента (FIND)

Дано: дерево T и ключ K.

Задача: проверить, есть ли узел с ключом K в дереве T, и если да, то вернуть ссылку на этот узел.

Алгоритм:

- Если дерево пусто, сообщить, что узел не найден, и остановиться.
- Иначе сравнить K со значением ключа корневого узла X.
- Если K=X, выдать ссылку на этот узел и остановиться.
- Если K>X, рекурсивно искать ключ K в правом поддереве T.
- Если K<X, рекурсивно искать ключ K в левом поддереве T.

Обход дерева (TRAVERSE)

Есть три операции обхода узлов дерева, отличающиеся порядком обхода узлов.

Первая операция — INFIX_TRAVERSE — позволяет обойти все узлы дерева в порядке возрастания ключей и применить к каждому узлу заданную пользователемфункцию обратного вызова f. Эта функция обычно работает только с парой (K,V), хранящейся в узле. Операция INFIX_TRAVERSE реализуется рекурсивным образом: сначала она запускает себя для левого поддерева, потом запускает данную функцию для корня, потом запускает себя для правого поддерева.

- INFIX_TRAVERSE (f) — обойти всё дерево, следуя порядку (левое поддерево, вершина, правое поддерево).

- PREFIX_TRAVERSE (f) — обойти всё дерево, следуя порядку (вершина, левое поддерево, правое поддерево).
- POSTFIX_TRAVERSE (f) — обойти всё дерево, следуя порядку (левое поддерево, правое поддерево, вершина).

INFIX_TRAVERSE:

Дано: дерево T и функция f

Задача: применить f ко всем узлам дерева T в порядке возрастания ключей

Алгоритм:

- Если дерево пусто, остановиться.
- Иначе
- Рекурсивно обойти левое поддерево T.
- Применить функцию f к корневому узлу.
- Рекурсивно обойти правое поддерево T.

В простейшем случае, функция f может выводить значение пары (K,V). При использовании операции INFIX_TRAVERSE будут выведены все пары в порядке возрастания ключей. Если же использовать PREFIX_TRAVERSE, то пары будут выведены в порядке, соответствующим описанию дерева, приведённого в начале статьи.

13. Сбалансированные деревья.

При добавлении узлов в дерево мы будем их равномерно располагать слева и справа, то получится дерево, у которого число вершин в левом и правом поддеревьях отличается не более, чем на единицу. Такое дерево называется **идеально сбалансированным**.

Для построения идеально сбалансированного дерева используется рекурсия. Для дерева из **n** узлов, где **nl** - количество узлов в левом поддереве, **nr** - количество узлов в правом поддереве, алгоритм построения идеально сбалансированного дерева описывается следующим образом:

1. выбрать одну вершину в качестве корня;
2. рекурсивно построить левое поддерево с **nl = n div 2** узлами;
3. рекурсивно построить правое поддерево с **nr = n – nl 1** узлами.

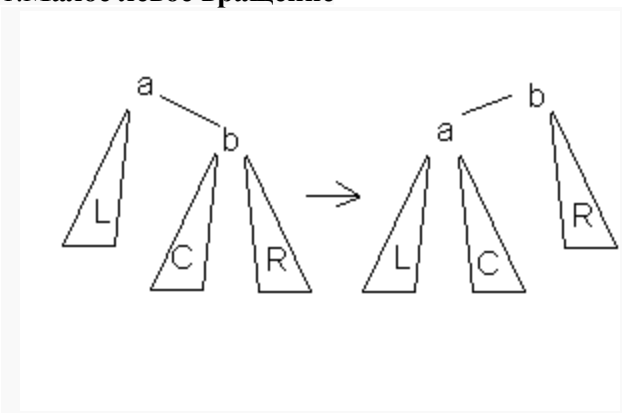
Адельсон-Вельский и Ландис сформулировали менее жесткий критерий сбалансированности таким образом: двоичное дерево называется сбалансированным, если у каждого узла дерева высота двух поддеревьев отличается не более чем на единицу. Такое дерево называется *АВЛ-деревом.*

Балансировка

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины.

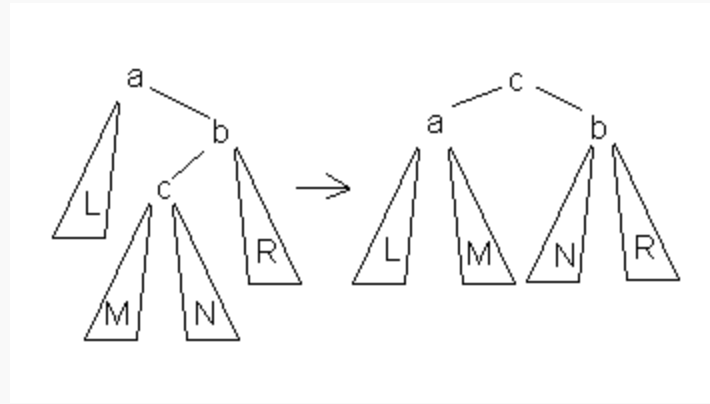
Используются 4 типа вращений:

1.**Малое левое вращение**



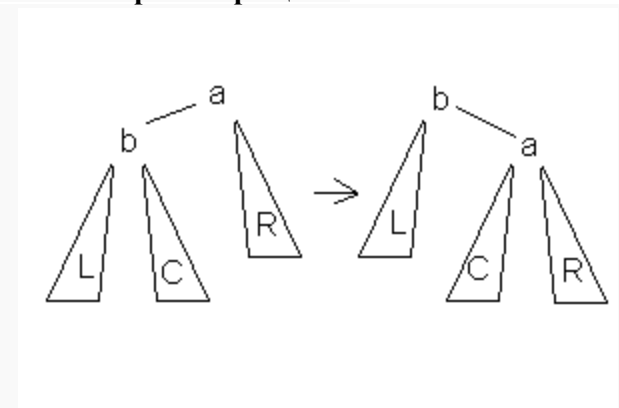
Данное вращение используется тогда, когда (высота b-поддерева - высота L) = 2 и высота C <= высота R.

2.**Большое левое вращение**



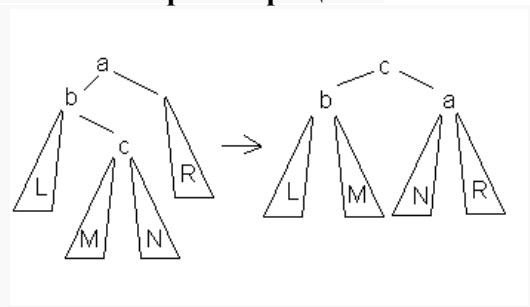
Данное вращение используется тогда, когда (высота b-поддерева - высота L) = 2 и высота c-поддерева > высота R.

3.**Малое правое вращение**



Данное вращение используется тогда, когда (высота b-поддерева — высота R) = 2 и высота C <= высота L.

4.**Большое правое вращение**



Данное вращение используется тогда, когда (высота b-поддерева - высота R) = 2 и высота c-поддерева > высота L.

14 Операции включения и исключения в сбалансированных деревьях.

Процесс включения узла фактически состоит из трех последовательно выполняемых подзадач:

1) проход по пути поиска (пока не убедимся, что элемента с таким ключом в дереве нет);
2) включение нового узла и определение показателя сбалансированности (Bal);
3) «отступление» (возврат) по пути поиска, с проверкой показателей сбалансированности для каждой вершины, и если необходимо, то проведение балансировки.

Алгоритм удаления узла из сбалансированного дерева основан на алгоритме удаления из дерева (на замене удаляемого узла на самого левого потомка из правого поддерева или на самого правого потомка из левого поддерева) с учетом операции балансировки, то есть тех же поворотов узлов.[1]

15 Матричное и списковое представление графов.

Графы в памяти могут представляться различным способом. Один из видов представления графов – это матрица смежности B(n*n); В этой матрице элемент b[i,j]=1, если ребро, связывающее вершины Vi и Vj существует и b[i,j]=0, если ребра нет. У неориентированных графов матрица смежности всегда симметрична.

Во многих случаях удобнее представлять граф в виде так называемого списка смежностей. Список смежностей содержит для каждой вершины из множества вершин V список тех вершин, которые непосредственно связаны с этой вершиной. Каждый элемент (ZAP[u]) списка смежностей является записью, содержащей данную вершину и указатель на следующую запись в списке (для последней записи в списке этот указатель – пустой). Входы в списки смежностей для каждой вершины графа хранятся в таблице (массиве) (BEG [u])

Пример неориентированного графа приведен на рис. 1, матрица смежности для этого графа – на рис. 2, а список смежности – на рис. 3.

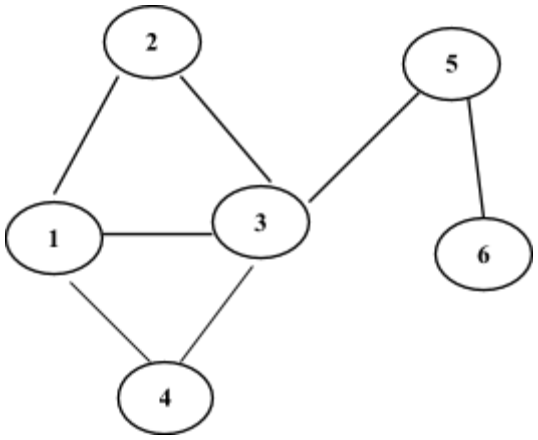
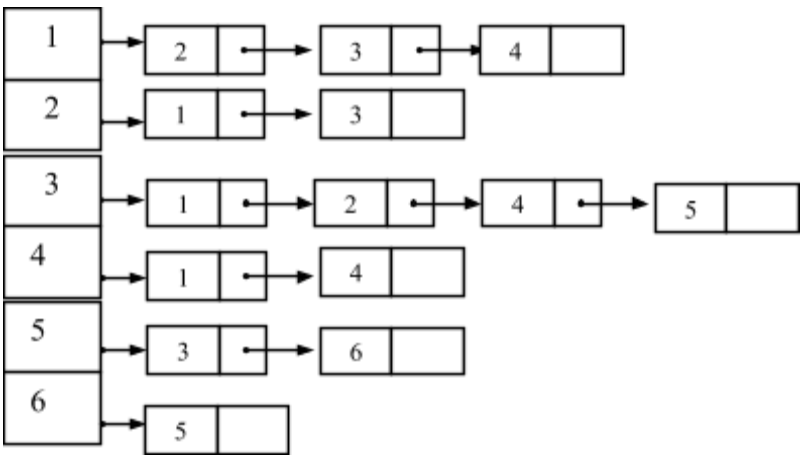


Рис. 1

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	0	0	0
3	1	1	0	1	1	0
4	1	0	1	0	0	0
5	0	0	1	0	0	1
6	0	0	0	0	1	0

Рис. 2

BEG[u] ZAP[u]



16 .Поиск путей в графах. Алгоритм поиска в глубину. Алгоритм поиска в ширину.

Алгоритм поиска в глубину

Пусть задан граф $G = (V, E)$, где V — множество вершин графа, E — множество ребер графа. Предположим, что в начальный момент времени все вершины графа окрашены в *белый* цвет. Выполним следующие действия:

1. Из множества всех *белых* вершин выберем любую вершину, обозначим её v_1.
2. Выполняем для неё процедуру DFS(v_1).
3. Повторяем шаги 1-3 до тех пор, пока множество *белых* вершин не пусто.

Процедура DFS (параметр — вершина $u \in V$)

1. Перекрашиваем вершину u в *черный* цвет.
2. Для всякой вершины w, смежной с вершиной u и окрашенной в белый цвет, выполняем процедуру DFS(w).

Поиск в ширину

Поиск в ширину выполняется в следующем порядке: началу обхода s приписывается метка 0, смежным с ней вершинам — метка 1. Затем поочередно рассматривается окружение всех вершин с метками 1, и каждой из входящих в эти окружения вершин приписываем метку 2 и т. д.

Если исходный граф связный, то поиск в ширину пометит все его вершины. Дуги вида (i, i+1) порождают остовный бесконтурный орграф, содержащий в качестве своей части остовное ордерево, называемое поисковым деревом.

Легко увидеть, что с помощью поиска в ширину можно также занумеровать вершины, нумеруя вначале вершины с меткой 1, затем с меткой 2 и т. д.

17 Алгоритм построения каркасов в графе

Вначале текущее множество рёбер устанавливается пустым. Затем, пока это возможно, проводится следующая операция: из всех рёбер, добавление которых к уже имеющемуся множеству не вызовет появление в нём цикла, выбирается ребро минимального веса и добавляется к уже имеющемуся множеству. Когда таких рёбер больше нет, алгоритм завершён. Подграф данного графа, содержащий все его вершины и найденное множество рёбер, является его остовным деревом минимального веса.

Алгоритм состоит из следующей последовательности действий:

1. Создается список ребер E, содержащий длину ребра, номер исходной вершины ребра (i), номер конечной вершины ребра (j), признак включения данного ребра в дерево.
2. Данный список упорядочивается в порядке возрастания длин ребер.
3. Просматривается список E и выбирается из него ребро с максимальной длиной, еще не включенное в результирующее дерево и не образующее цикла с уже построенными ребрами.
4. Если все вершины включены в дерево и количество ребер на единицу меньше количества вершин, то алгоритм свою работу закончил. В противном случае осуществляется возврат к пункту 3.

1.2 Псевдокод алгоритма

```
Отсортировать ребра в порядке возрастания весов
инициализировать структуру разбиений
edgeCount=l
while edgeCount<=E and includedCount<=M-l do
parentl=FindRoot (edge [edgeCount]. start)
parent2=FindRoot (edge [edgeCount]. end)
if parentl/=parent2 then
добавить edge [edgeCount] в остовное дерево
includedCount=includedCount=l
Union (parent1,parent2)
end if
edgeCount=edgeCount+l
end while.
```

E - число ребер в графе;

V - число вершин в графе

edgeCount - переменная;

includedCount - переменная;

Parent - массив, в котором под каждый элемент множества, с которым мы собираемся работать, отведена одна ячейка;

FindRoot (x) - (x элемент, для которого мы хотим найти корень части, его содержащей) процедура, начинающая с ячейки Parent [x], в которой хранится номер непосредственного предка элемента x;

Union - функция, выполняющая объединение двух частей в одну.

18. Эйлеров и гамильтонов пути в графе.

Произвольный путь в графе, проходящий через каждое ребро графа точно один раз, называется эйлеровым путем. При этом, если по некоторым вершинам путь проходит неоднократно, то он является непростым. Если путь замкнут, то имеем эйлеров цикл. Для существования эйлерова пути в связном графе необходимо и достаточно, чтобы граф содержал не более двух вершин нечетной степени [2,3]

Путь в графе, проходящий в точности один раз через каждую вершину графа (а не каждое ребро) и соответствующий цикл называются гамильтоновыми и существуют не для каждого графа, как и эйлеров путь. В отличие от эйлеровых путей неизвестно ни одного простого необходимого и достаточного условия для существования гамильтоновых путей. Неизвестен даже алгоритм полиномиальной сложности, проверяющий существование гамильтонова пути в произвольном графе.

Поиск эйлерова цикла в графе

```
procedure find_all_cycles (v)
var массив cycles
1. пока есть цикл, проходящий через v, находим его
   добавляем все вершины найденного цикла в массив cycles (сохраняя порядок обхода)
```

удаляем цикл из графа
2. идем по элементам массива cycles
каждый элемент cycles[i] добавляем к ответу
из каждого элемента рекурсивно вызываем себя: find_all_cycles (cycles[i])

19 Пути в графах. Алгоритм поиска кратчайших путей в графе.

Обозначения

V — множество вершин графа

- E — множество ребер графа
- $w[ij]$ — вес (длина) ребра ij
- a — вершина, расстояния от которой ищутся
- U — множество посещенных вершин
- $d[u]$ — по окончании работы алгоритма равно длине кратчайшего пути из a до вершины u
- $p[u]$ — по окончании работы алгоритма содержит кратчайший путь из a в u

Псевдокод

Присвоим $d[a] \leftarrow 0,\ p[a] \leftarrow a$
Для всех $u \in V$ отличных от a
присвоим $d[u] \leftarrow \infty$
Пока $\exists v \notin U$
Пусть $v \notin U$ — вершина с минимальным $d[v]$
занесём v в U
Для всех $u \notin U$ таких, что $vu \in E$
если $d[u] > d[v] + w[vu]$ то
изменим $d[u] \leftarrow d[v] + w[vu]$
изменим $p[u] \leftarrow p[v], u$

Описание

В простейшей реализации для хранения чисел *d*[*i*] можно использовать массив чисел, а для хранения принадлежности элемента множеству U — массив булевых переменных.

В начале алгоритма расстояние для начальной вершины полагается равным нулю, а все остальные расстояния заполняются большим положительным числом (бóльшим максимального возможного пути в графе). Массив флагов заполняется нулями. Затем запускается основной цикл.

На каждом шаге цикла мы ищем вершину v с минимальным расстоянием и флагом равным нулю. Затем мы устанавливаем в ней флаг в 1 и проверяем все соседние с ней вершины u. Если в них (в u) расстояние больше, чем сумма расстояния до текущей вершины и длины ребра, то уменьшаем его. Цикл завершается, когда флаги всех вершин становятся равны 1, либо когда у всех вершин с флагом 0 $d[i] = \infty$. Последний случай возможен тогда и только тогда, когда граф *G* не связан.

20. Пути в графах. Алгоритм поиска кратчайших путей между всеми вершинами графа.

Пусть вершины графа пронумерованы от 1 до и введено обозначение для длины кратчайшего пути от до , который кроме самих вершин проходит только через вершины . Очевидно, что — длина (вес) ребра , если таковое существует (в противном случае его длина может быть обозначена как).

Существует два варианта значения :

1. Кратчайший путь между не проходит через вершину , тогда
2. Существует более короткий путь между , проходящий через , тогда он сначала идёт от до , а потом от до . В этом случае, очевидно,

Таким образом, для нахождения значения функции достаточно выбрать минимум из двух обозначенных значений.

Тогда рекуррентная формула для имеет вид:

— длина ребра

Алгоритм Флойда-Уоршелла последовательно вычисляет все значения для от 1 до . Полученные значения являются длинами кратчайших путей между вершинами

Псевдокод[править]

На каждом шаге алгоритм генерирует матрицу Матрица содержит длины кратчайших путей между всеми вершинами графа. Перед работой алгоритма матрица заполняется длинами рёбер графа.

```
for k = 1 to n
  for i = 1 to n
    for j = 1 to n
      W[i][j] = min(W[i][j], W[i][k] + W[k][j])
```

21. Использование структуры Б-деревьев для хранения файлов.

Основные достоинства

- Во всех случаях полезное использование пространства вторичной памяти составляет свыше 50 %. С ростом степени полезного использования памяти не происходит снижения качества обслуживания.
- Произвольный доступ к записи реализуется посредством малого количества подопераций (обращения к физическим блокам).
- В среднем достаточно эффективно реализуются операции включения и удаления записей; при этом сохраняется естественный порядок ключей с целью последовательной обработки, а также соответствующий баланс дерева для обеспечения быстрой произвольной выборки.
- Неизменная упорядоченность по ключу обеспечивает возможность эффективной пакетной обработки.