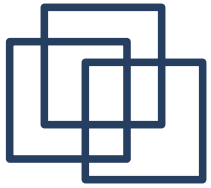
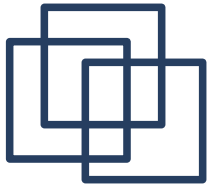


Стек, очередь, дек, деревья

Т.С.Растопшина



Стек

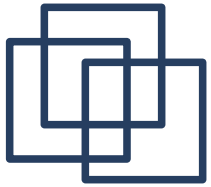
Стеком называется динамическая структура данных, добавление компоненты в которую и исключение компоненты из которой производится из одного конца, называемого вершиной стека.

Стек работает по принципу LIFO (Last-In, First-Out) - поступивший последним, обслуживается первым.

Стек

Использование стека в программировании:

- **Нужно сохранить некоторую работу, которая еще не выполнена до конца, при необходимости переключения на другую задачу. Стек используется для временного сохранения состояния не выполненного до конца задания. После сохранения состояния компьютер переключается на другую задачу. По окончании ее выполнения состояние отложенного задания восстанавливается из стека, и компьютер продолжает прерванную работу.**



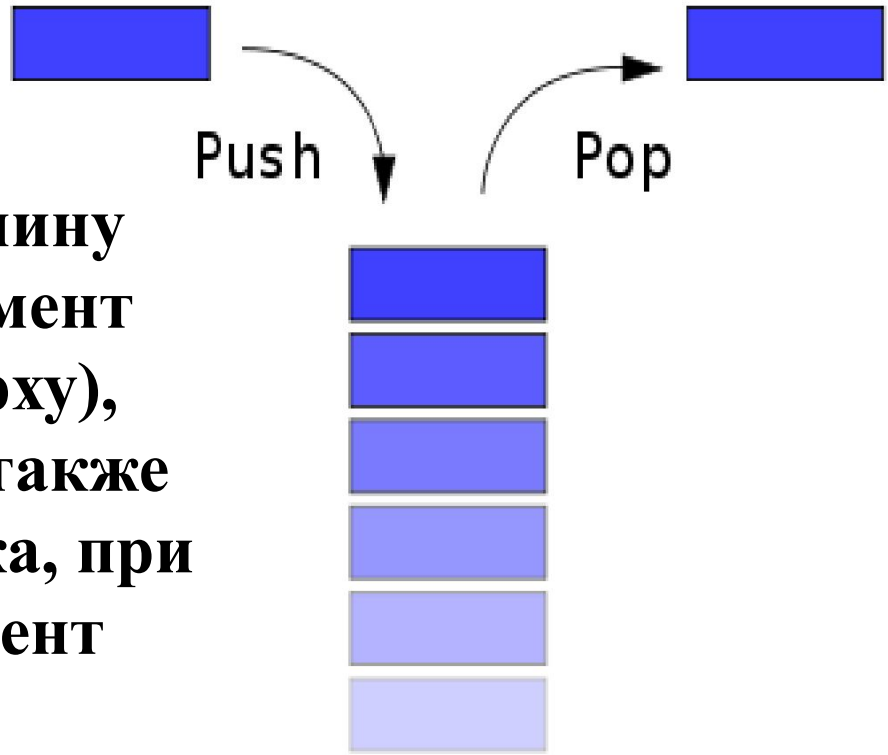
Стек

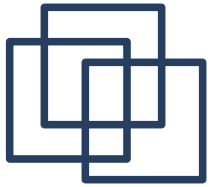
Использование стека в программировании:

- используются при разборе (parsing) грамматик (от простых алгебраических выражений до языков программирования)
- как средство моделирования рекурсии
- как модель исполнения инструкций.

Стек

Добавление элемента, называемое также проталкиванием (push), возможно только в вершину стека (добавленный элемент становится первым сверху), выталкивание (pop) — также только из вершины стека, при этом второй сверху элемент становится верхним





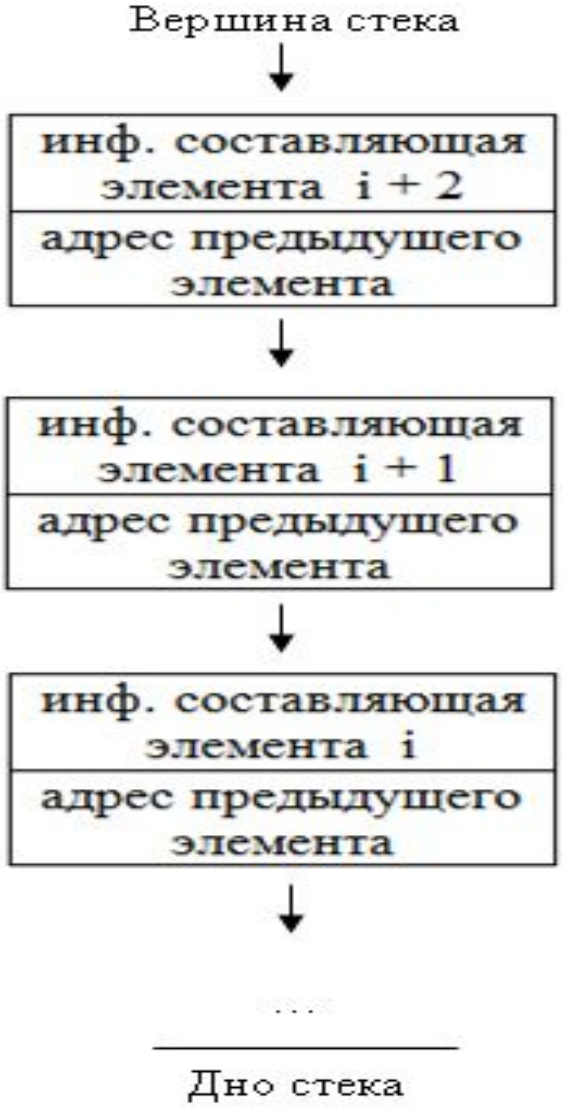
Стек

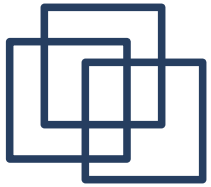
типовые операции над стеком и его элементами:

- **добавление элемента в стек;**
- **удаление элемента из стека;**
- **проверка, пуст ли стек;**
- **просмотр элемента в вершине стека без удаления;**
- **очистка стека.**

Стек

Учитывая специфику стека, указатели должны следовать от последнего элемента (вершина стека) к первому (дно стека)





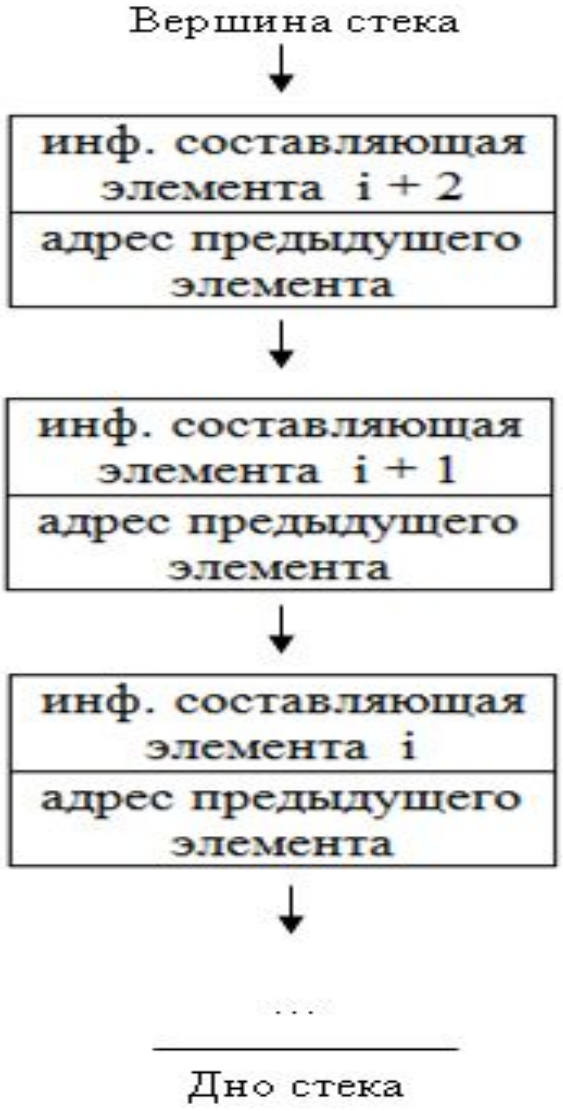
Стек

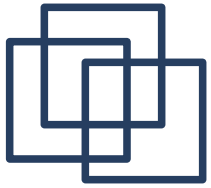
Учитывая специфику стека, указатели должны следовать от последнего элемента (вершина стека) к первому (дно стека)

```
StackItem *Current = Sp;
while(Current != NULL)
{
   cout << Current->Info << " ";
   Current = Current->Next;
}
```

Стек

Для построения логического порядка следования элементов достаточно знать вершинный элемент, все остальное восстанавливается по адресным частям элементов независимо от их реального размещения в памяти.



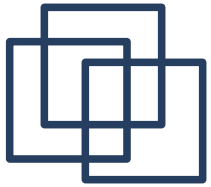


Стек

Для программной реализации элемент стека надо объявить как структуру, содержащую по крайней мере два поля – информационное и связующее.

Например,

```
Struct StackItem {
        int Info;
        StackItem *Next;
};
```

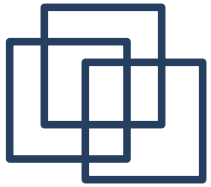


Стек

Для поддержки работы стека необходимо знать адрес элемента, находящегося на вершине стека, т.е. помещенного в стек самым последним:

```
StackItem *Sp=NULL;
```

Конструкция Sp->Info представляет информационную часть, а конструкция Sp->Next - адрес предыдущего элемента, который был помещен в стек непосредственно перед текущим.

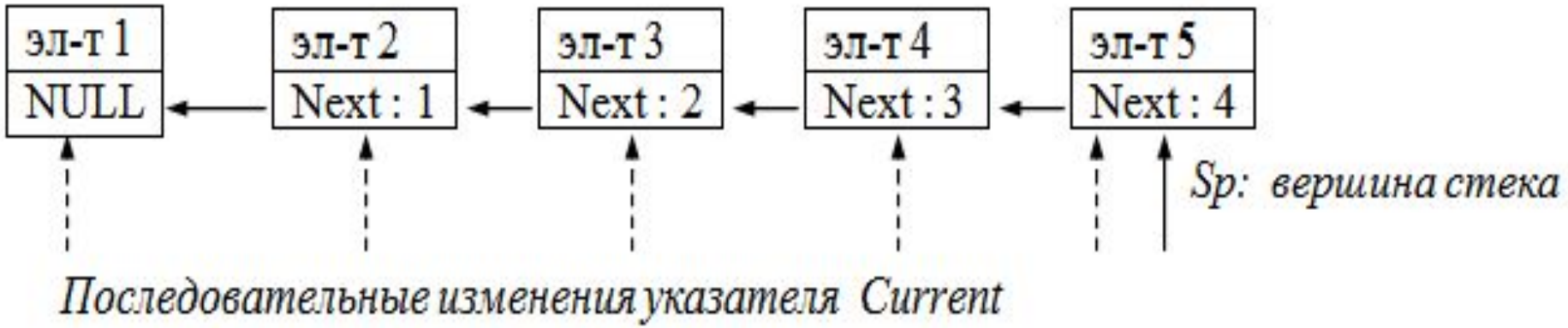


Стек

Для прохода по стеку от вершинного элемента к самому первому элементу необходима вспомогательная ссылочная переменная (например – с именем Current). Она на каждом шаге прохода по стеку должна определять адрес текущего элемента. В самом начале прохода надо установить значение **Current = Top,** а затем циклически менять его на значение **Current->Next** до тех пор, пока не будет достигнут первый элемент стека.

Стек

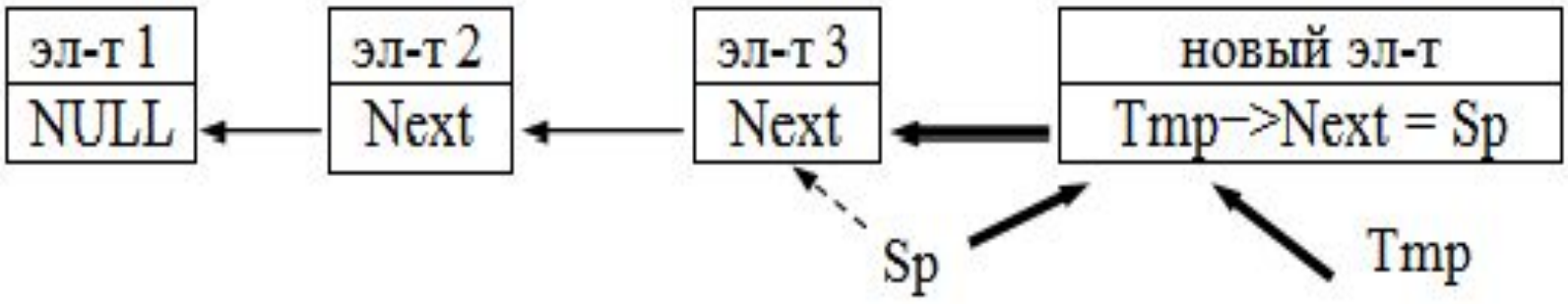
Для прохода надо использовать цикл с неизвестным числом повторений, а признаком его завершения должно быть получение в поле Current->Next пустой ссылки NULL.



Стек

Для добавления нового элемента в вершину стека необходимо выполнить следующие действия:

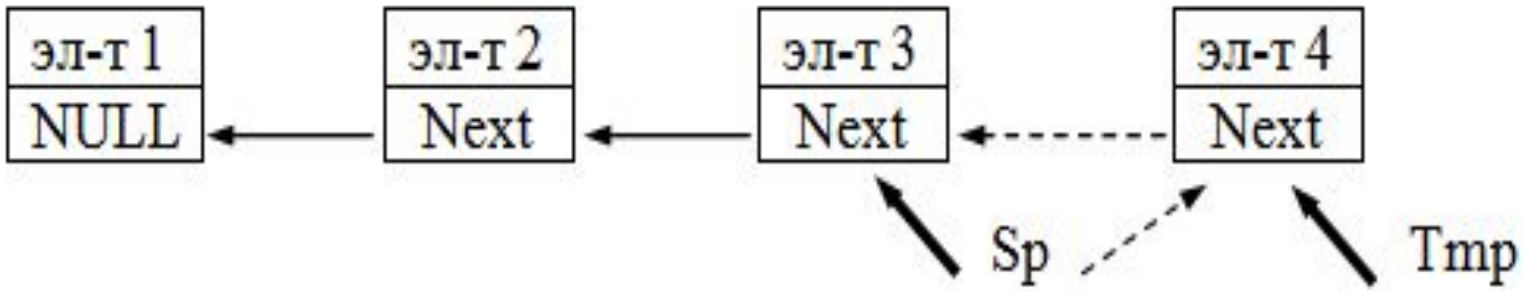
- Выделить память для размещения нового элемента с помощью вспомогательной ссылочной переменной Tmp ;
- Заполнить информационную часть нового элемента;
- Установить адресную часть нового элемента таким образом, чтобы она определяла адрес бывшего вершинного элемента;
- Изменить адрес вершины стека так, чтобы он определял в качестве вершины новый элемент.

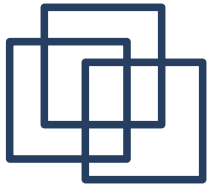


Стек

Для удаления элемента с вершины стека необходимо выполнить следующие действия:

- С помощью вспомогательной переменной Tmp адресовать удаляемый элемент: **Tmp = Sp;**
- Изменить значение переменной Sp на адрес новой вершины стека;
- Обработать удаленный с вершины элемент:
- Освободить, занимаемую им, память.





Стек

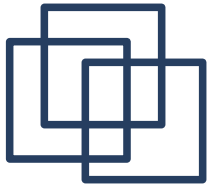
Очистка стека

Для удаления всех элементов стека необходимо выполнить следующие действия:

- С помощью вспомогательной переменной Tmp адресовать удаляемый элемент;
- Удалить элемент;
- Если не «Дно стека», переход к п.1
- Вершине стека присвоить NULL.

Проверка на пустоту стека

Для проверки стека на пустоту достаточно проверить состояние вершины стека. Если Sp=NULL, стек пуст



Очередь

Это динамическая структура данных, добавление элементов в которую выполняется в один конец, называемый хвостом, а выборка — из другого конца , называемый головой. При выборке элемент исключается из очереди.

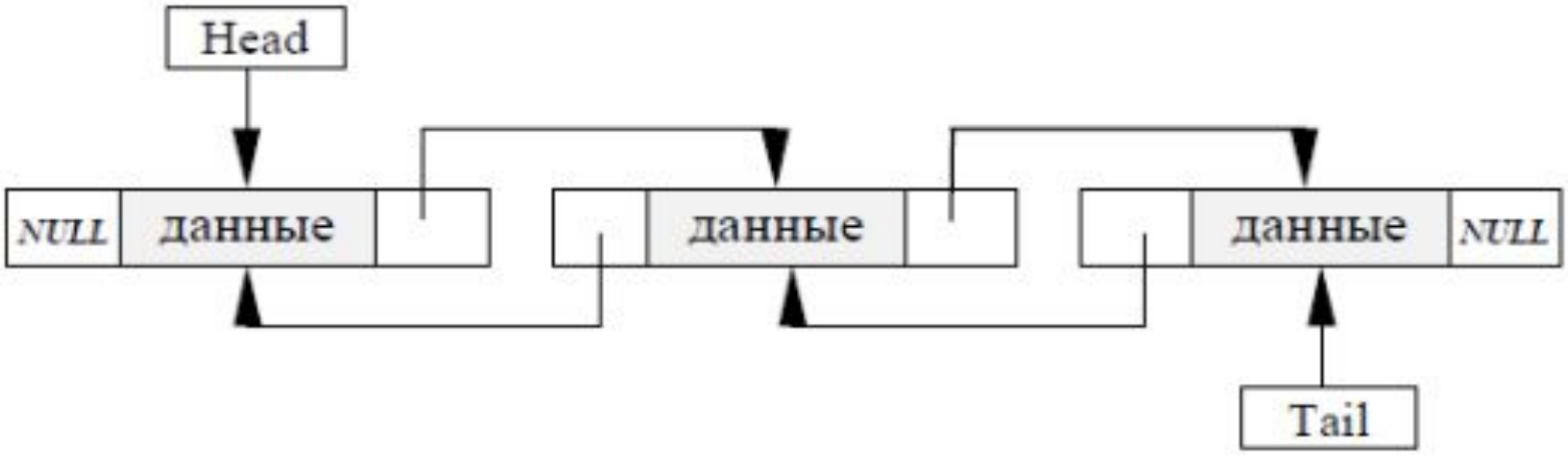
Говорят, что очередь реализует принцип обслуживания FIFO (first in — first out, первым пришел — первым обслужен).

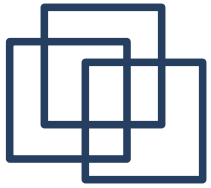
В программировании очереди применяются очень широко — например, при моделировании, буферизованном вводе-выводе или диспетчеризации задач в операционной системе.

Очередь

Для реализации очереди можно выбрать двунаправленный список.

Для доступа к списку используется не одна переменная-указатель, а две – ссылка на «голову» списка (*Head*) и на «хвост» - последний элемент (*Tail*)

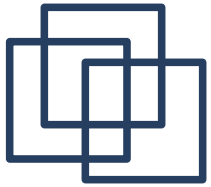




Очередь

Типовые операции над очередью и ее элементами:

- **добавление элемента в хвост очереди;**
- **удаление элемента из головы очереди;**
- **проверка, пуста ли очередь;**
- **очистка очереди.**



Очередь

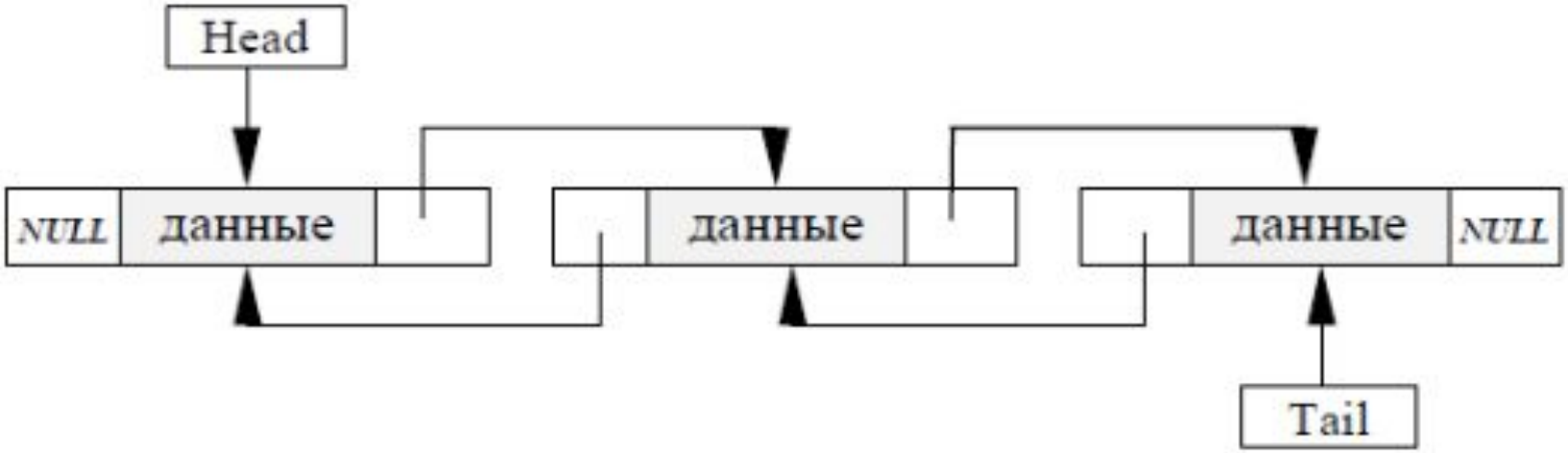
Для программной реализации элемент очереди надо объявить элемент как структуру, содержащую три поля – информационное и два связующих.

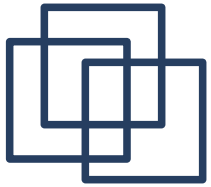
Например,

```
struct QueueElem {
        double Elem;
        QueueElem *nextElement;
        QueueElem *prevElement;
};
```

Очередь

Для поддержки работы очереди необходимо знать адрес элемента, находящегося в голове очереди QueueElem *head, и адрес элемента, находящегося в хвосте очереди QueueElem *tail.

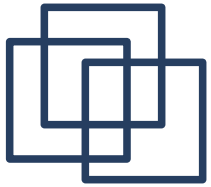




Очередь

Для того чтобы создать очередь надо создать голову списка.

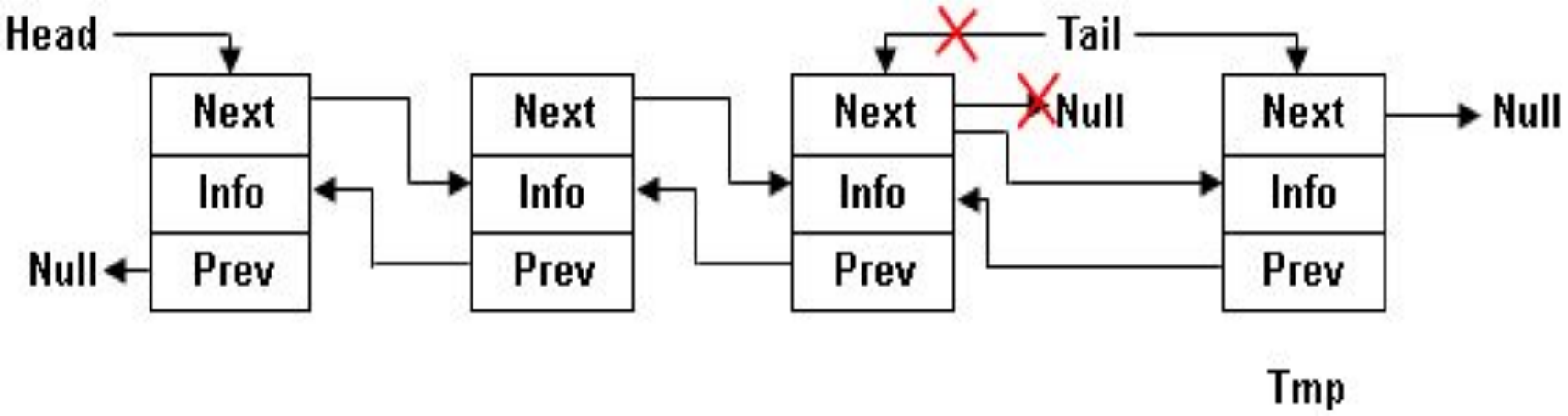
Для создания нового элемента надо выделить под него память (например оператором new) и связать с основным списком - создать указатели на следующий и предыдущий элементы равные 0. для первого элемента указатель на предыдущий элемент всегда равен 0



Очередь. Добавление элемента

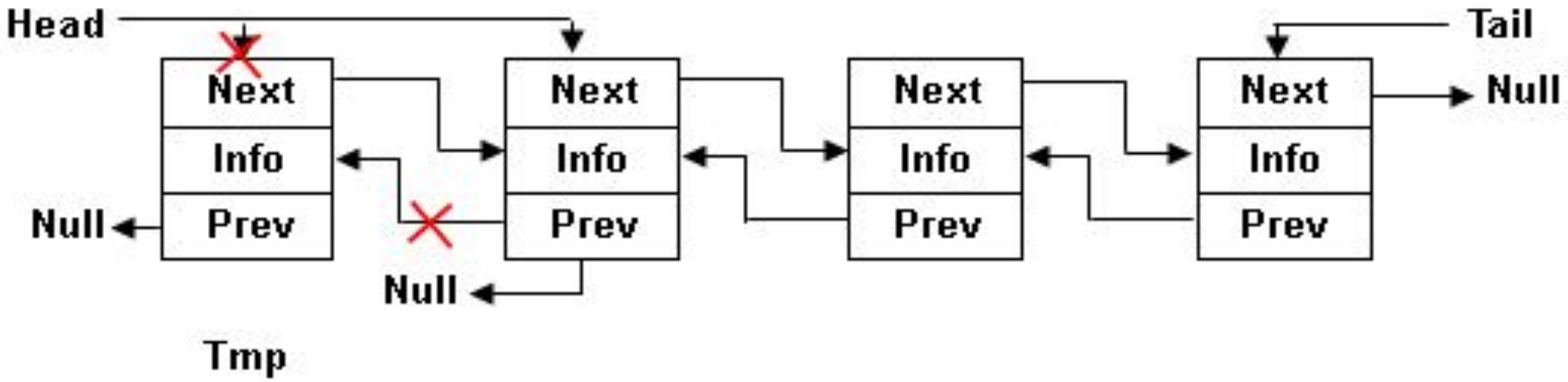
- Выделить память для размещения нового элемента с помощью вспомогательной ссылочной переменной Tmp ;
- Заполнить информационную часть нового элемента;
- Установить адресную часть нового элемента таким образом, чтобы она определяла адрес бывшего последнего элемента очереди;
- Установить адресную часть бывшего последнего элемента очереди таким образом, чтобы она определяла адрес нового элемента;
- Создать новому элементу указатель на следующий элемент равный Null;
- Изменить адрес хвоста очереди так, чтобы он определял в качестве конца очереди новый элемент.

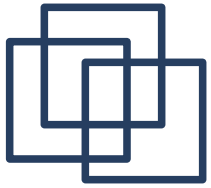
Очередь. Добавление элемента



Очередь. Извлечение элемента

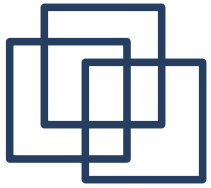
- Извлечь информационную часть первого элемента очереди
- С помощью вспомогательной переменной Tmp адресовать удаляемый элемент: Tmp = Head;
- Изменить указатель второго элемента очереди на предыдущий элемент равный Null;
- Изменить значение переменной Head на адрес второго элемента очереди;
- Освободить память, занимаемую Tmp.





Очередь. Очистка очереди

1. С помощью вспомогательной переменной Tmp адресовать удаляемый элемент: Tmp = Head;
2. Изменить указатель элемента, следующего за удаляемым, на предыдущий элемент равный Null;
3. Изменить значение переменной Head на адрес следующего за удаляемым;
4. Освободить память, занимаемую Tmp;
5. Если в очереди есть еще элементы, переход к п.1;
6. Голове и хвосту очереди присвоить NULL.

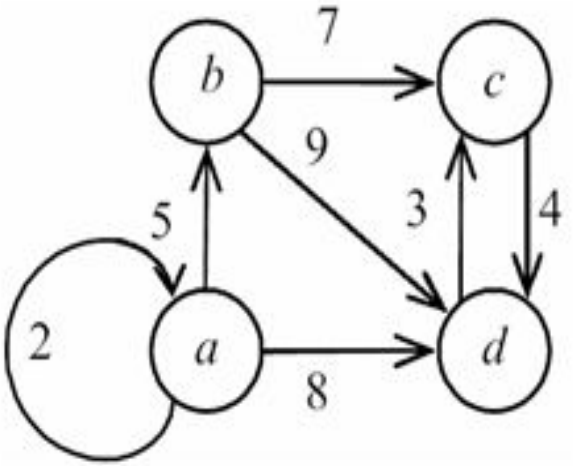


Дек

Дек является симбиозом стека и очереди - это та же структура, но на этот раз с ней можно работать с обеих концов. Таким образом, если мы будем работать с деком только с левого края, то фактически получается стек. Аналогично мы получим стек, если будем работать только с правого края (конца дека).

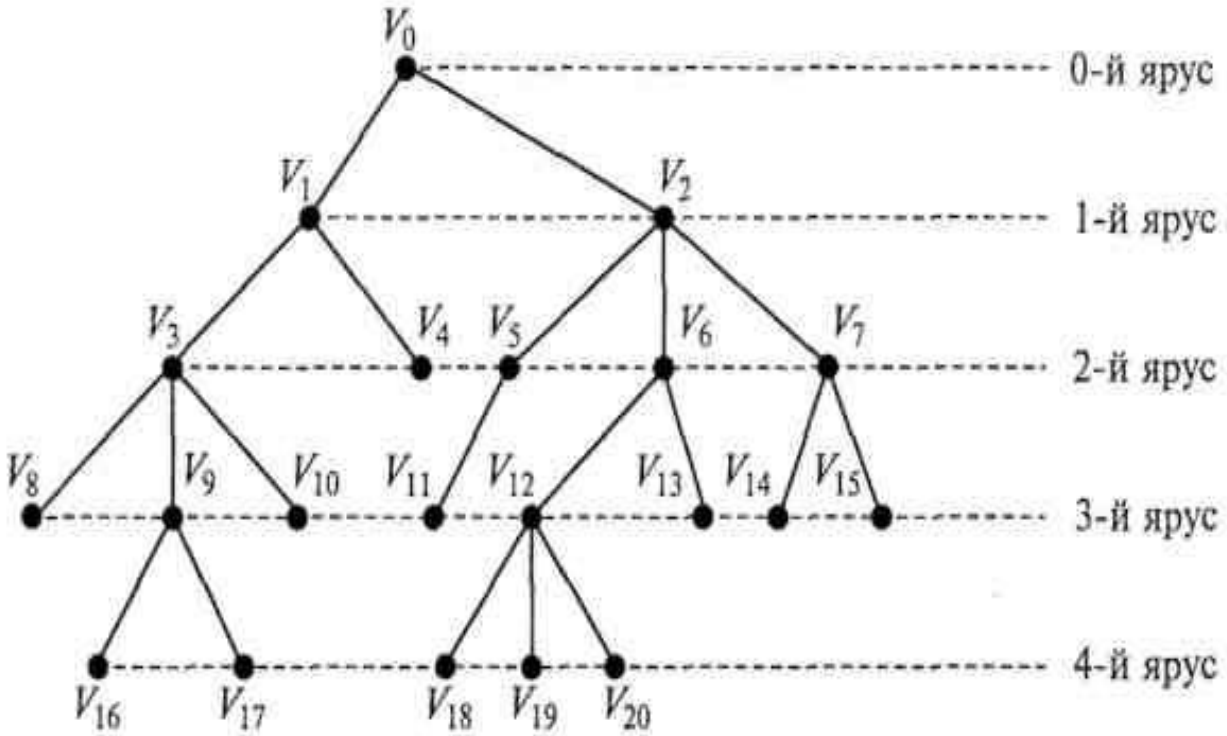
Граф

Граф – это совокупность двух конечных множеств: множества точек и множества линий, попарно соединяющих некоторые из этих точек. Множество точек называется вершинами (узлами) графа. Множество линий, соединяющих вершины графа, называются ребрами (дугами) графа.



Деревья

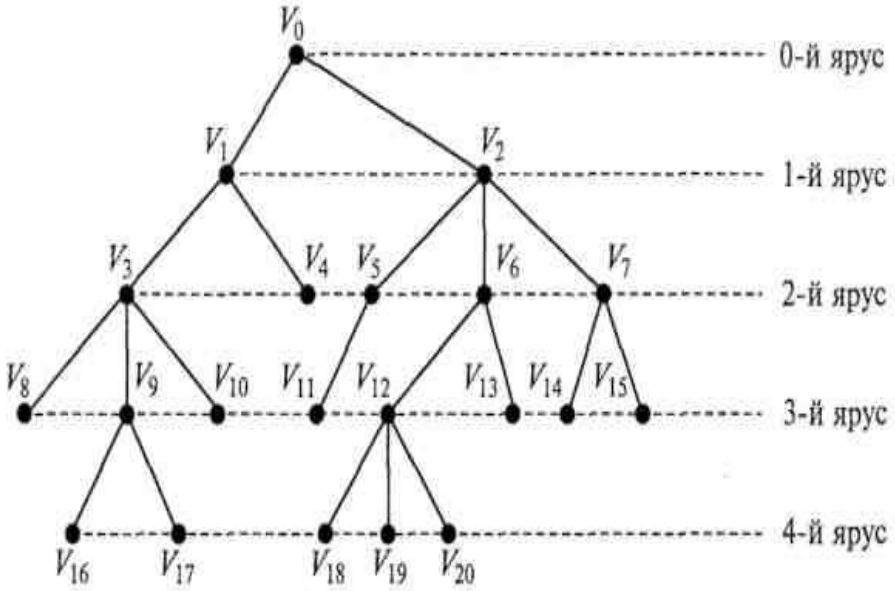
Деревом называют конечный связный граф с выделенной вершиной (**корнем**), не имеющий циклов.



Деревья

Для каждой пары вершин дерева – **узлов** – существует единственный маршрут, поэтому вершины удобно классифицировать по степени удалённости от корневой вершины

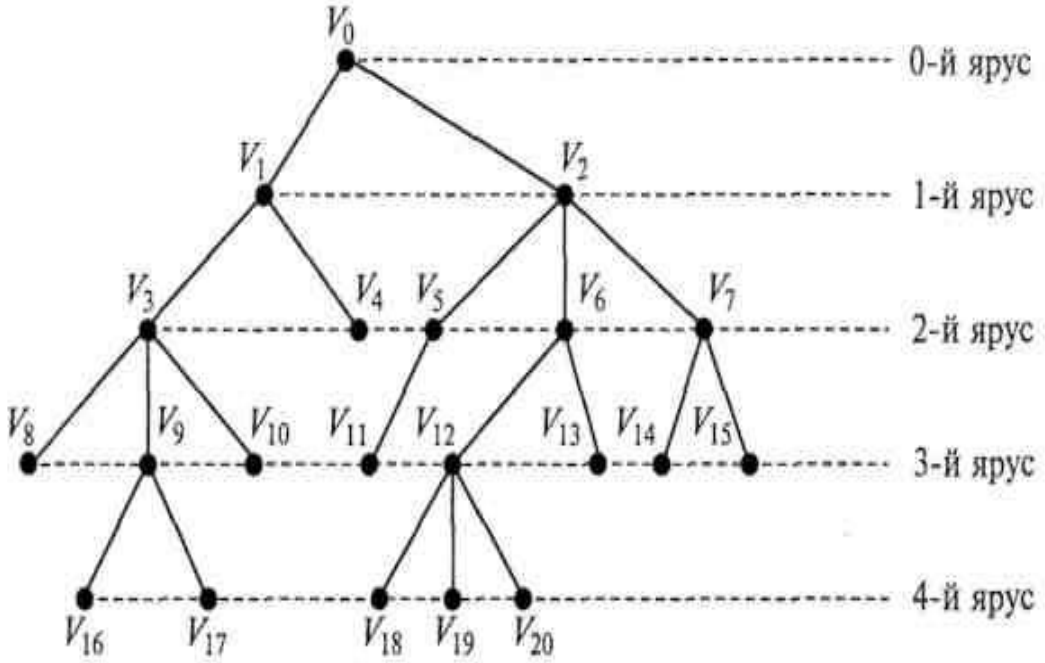
.



Деревья

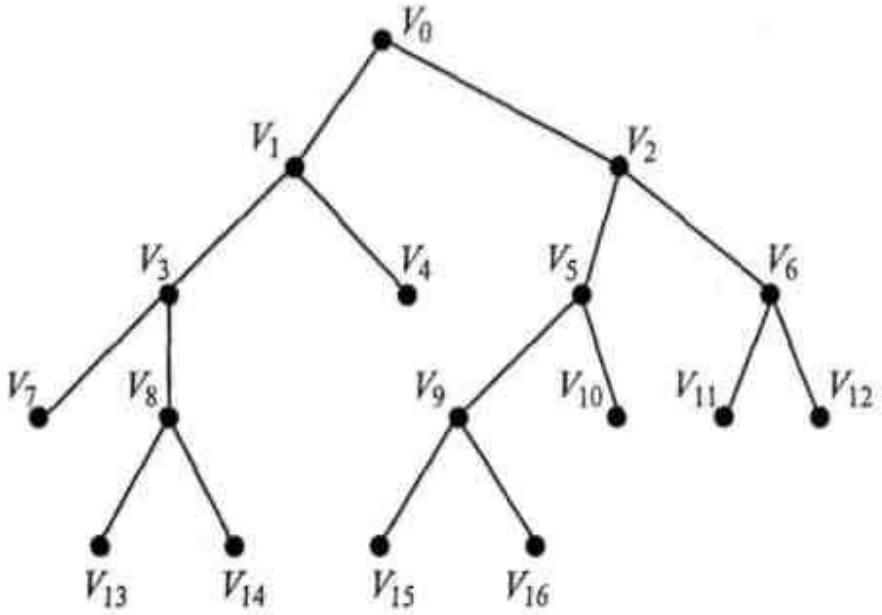
Висячие вершины, за исключением корневой, называются **листьями**.

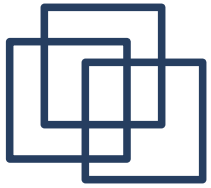
Число путей в каждом дереве соответствует числу висячих вершин (листьев).



Бинарные деревья

Деревья, в которых каждый узел либо является листом, либо образует два поддерева: левое и правое, называются **бинарными деревьями** и используются при делении множества на два взаимоисключающих подмножества по какому-то признаку (дихотомическое деление)





Бинарные деревья

Ключевые термины:

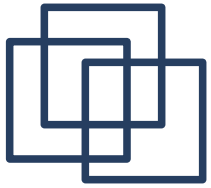
Бинарное (двоичное) дерево – это дерево, в котором каждая вершина имеет не более двух потомков.

Вершина (узел) дерева – это каждый элемент дерева.

Ветви дерева – это направленные дуги, которыми соединены вершины дерева.

Высота (глубина) дерева – это количество уровней, на которых располагаются его вершины.

Корень дерева – это начальный узел дерева, ему соответствует нулевой уровень.



Бинарные деревья

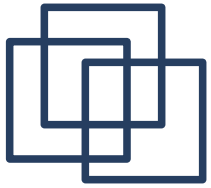
Ключевые термины:

Листья дерева – это вершины, в которые входит одна ветвь и не выходит ни одной ветви.

Неполное бинарное дерево – это дерево, уровни которого заполнены не полностью.

Нестрогое бинарное дерево – это дерево, у которого вершины имеют степень ноль (у листьев), один или два (у узлов).

Обход дерева – это упорядоченная последовательность вершин дерева, в которой каждая вершина встречается только один раз.



Бинарные деревья

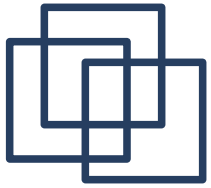
Ключевые термины:

Полное бинарное дерево – это дерево, которое содержит только полностью заполненные уровни.

Потомки – это все вершины, в которые входят ветви, исходящие из одной общей вершины.

Почти сбалансированное дерево – это дерево, у которого длины всевозможных путей от корня к внешним вершинам отличаются не более, чем на единицу.

Предок – это вершина, из которой исходят ветви к вершинам следующего уровня.



Бинарные деревья

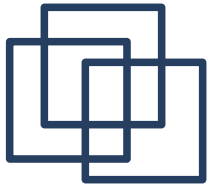
Ключевые термины:

Степень вершины – это количество дуг, которое выходит из этой вершины.

Степень дерева – это максимальная степень вершин, входящих в дерево.

Строгое бинарное дерево – это дерево, у которого вершины имеют степень ноль (у листьев) или два (у узлов).

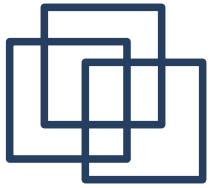
Упорядоченное дерево – это дерево, у которого ветви, исходящие из каждой



Бинарные деревья

В программировании при решении большого класса задач используются бинарные деревья.

Бинарные деревья могут применяться для поиска данных в специально построенных деревьях (базы данных), сортировки данных, вычислений арифметических выражений, кодирования.



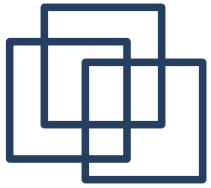
Бинарные деревья

Описание бинарного дерева выглядит следующим образом:

```
struct имя_типа {
        информационное поле;
        адрес левого поддерева;
        адрес правого поддерева;
       };
```

где информационное поле – это поле любого ранее объявленного или стандартного типа;

адрес левого (правого) поддерева – это указатель на



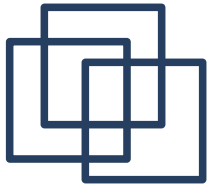
Бинарные деревья

Например,

```
struct BinaryTree{
        int Data; //поле данных
        BinaryTree* Left; //указатель на левый потомок
        BinaryTree* Right; //указатель на правый
//потомок
};
```

Корень дерева:

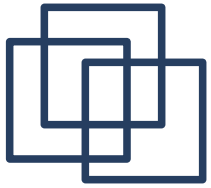
```
BinaryTree* BTree = NULL;
```



Бинарные деревья

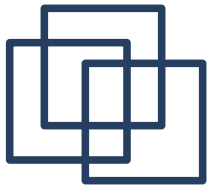
Основными операциями, осуществляемыми с бинарными деревьями, являются:

- создание бинарного дерева;
- печать бинарного дерева;
- обход бинарного дерева;
- вставка элемента в бинарное дерево;
- удаление элемента из бинарного дерева;
- проверка пустоты бинарного дерева;
- удаление бинарного дерева



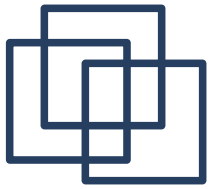
Бинарные деревья

```
//создание бинарного дерева
void Make_Binary_Tree(BinaryTree** Node, int n){
  BinaryTree** ptr;//вспомогательный указатель
  srand(time(NULL)*1000);
  while (n > 0) {
    ptr = Node;
    while (*ptr != NULL) {
      if ((double) rand()/RAND_MAX < 0.5)
        ptr = &((*ptr)->Left);
      else ptr = &((*ptr)->Right);
    }
    (*ptr) = new BinaryTree();
    cout << "Введите значение ";
    cin >> (*ptr)->Data;
    n--;
  }
}
```



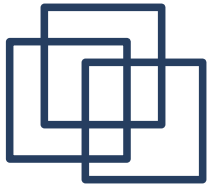
Бинарные деревья

```
//печать бинарного дерева
void Print_BinaryTree(BinaryTree* Node, int l){
  int i;
  if (Node != NULL) {
    Print_BinaryTree(Node->Right, l+1);
    for (i=0; i< l; i++) cout << "    ";
    printf ("%4ld", Node->Data);
    Print_BinaryTree(Node->Left, l+1);
  }
  else cout << endl;
}
```



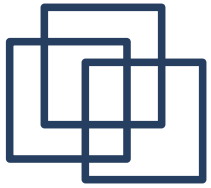
Бинарные деревья

```
//прямой обход бинарного дерева
void PreOrder_BinaryTree(BinaryTree* Node){
   if (Node != NULL) {
      printf ("%3ld",Node->Data);
      PreOrder_BinaryTree(Node->Left);
      PreOrder_BinaryTree(Node->Right);
   }
}
//обратный обход бинарного дерева
void PostOrder_BinaryTree(BinaryTree* Node){
   if (Node != NULL) {
      PostOrder_BinaryTree(Node->Left);
      PostOrder_BinaryTree(Node->Right);
      printf ("%3ld",Node->Data);
   }
}
```



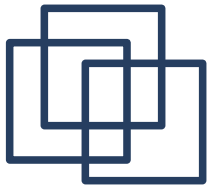
Бинарные деревья

```
//симметричный обход бинарного дерева
void SymmetricOrder_BinaryTree(BinaryTree* Node){
  if (Node != NULL) {
    PostOrder_BinaryTree(Node->Left);
    printf ("%3ld",Node->Data);
    PostOrder_BinaryTree(Node->Right);
  }
}
```



Бинарные деревья

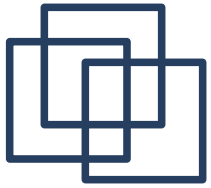
```
//вставка вершины в бинарное дерево
void Insert_Node_BinaryTree(BinaryTree** Node,int Data) {
  BinaryTree* New_Node = new BinaryTree;
  New_Node->Data = Data;
  New_Node->Left = NULL;
  New_Node->Right = NULL;
  BinaryTree** ptr = Node;//вспомогательный указатель
  srand(time(NULL)*1000);
```



Бинарные деревья

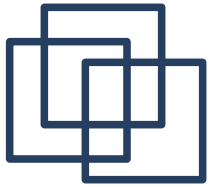
```
//вставка вершины в бинарное дерево

   while (*ptr != NULL) {
      double q = (double) rand()/RAND_MAX;
      if ( q < 1/3.0) ptr = &((*ptr)->Left);
      else if ( q > 2/3.0) ptr = &((*ptr)->Right);
      else break;
   }
   if (*ptr != NULL) {
      if ( (double) rand()/RAND_MAX < 0.5 )
        New_Node->Left = *ptr;
      else New_Node->Right = *ptr;
      *ptr = New_Node;
   }
   else{
      *ptr = New_Node;
   }
}
```



Бинарные деревья

```
//удаление вершины из бинарного дерева
void Delete_Node_BinaryTree(BinaryTree** Node,int Data){
  if ( (*Node) != NULL ){
    if ((*Node)->Data == Data){
      BinaryTree* ptr = (*Node);
      if ( (*Node)->Left == NULL && (*Node)->Right == NULL ) (*Node) = NULL;
      else if ((*Node)->Left == NULL) (*Node) = ptr->Right;
      else if ((*Node)->Right == NULL) (*Node) = ptr->Left;
      else {
        (*Node) = ptr->Right;
        BinaryTree ** ptr1;
        ptr1 = Node;
        while (*ptr1 != NULL)
          ptr1 = &((*ptr1)->Left);
        (*ptr1) = ptr->Left;
      }
      delete(ptr);
```

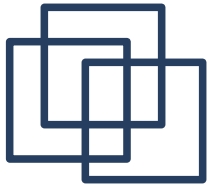


Бинарные деревья

```
//удаление вершины из бинарного дерева
void Delete_Node_BinaryTree(BinaryTree** Node,int Data){
  if ( (*Node) != NULL ){
    if ((*Node)->Data == Data){

      ...

       delete(ptr);
       Delete_Node_BinaryTree(Node,Data);
    }
    else {
      Delete_Node_BinaryTree(&((*Node)->Left),Data);
      Delete_Node_BinaryTree(&((*Node)->Right),Data);
    }
  }
}
```



Бинарные деревья

```
//проверка пустоты бинарного дерева
bool Empty_BinaryTree(BinaryTree* Node){
  return ( Node == NULL ? true : false );
}

//освобождение памяти, выделенной под бинарное дерево
void Delete_BinaryTree(BinaryTree* Node){
  if (Node != NULL) {
    Delete_BinaryTree(Node->Left);
    Delete_BinaryTree(Node->Right);
    delete(Node);
  }
}
```