СОДЕРЖАНИЕ

Размещается на образовательном портале ЦДОТ ТОГУ по решению кафедры «Экономическая кибернетика».

ВВЕДЕНИЕ

Без понимания структур данных и алгоритмов невозможно создать серьезный программный продукт. «Они служат базовыми элементами любой машинной программы. В организации структур данных и процедур их обработки заложена возможность проверки правильности работы программы» [Никлас Вирт]. Поэтому главная задача дисциплины «Структуры данных» научить студентов представлять данные на физическом и логическом уровнях для целенаправленного их использования при разработке прикладных и системных программ; применять базовые алгоритмы программирования при создании прикладного и системного программного обеспечения.

В результате изучения дисциплины студент должен

ЗНАТЬ:

- классификацию структур данных, их особенности, организацию и их представление в памяти ЭВМ;
- типичные операции над структурами данных, их возможности, особенности использования и реализацию;
- средства построения алгоритмов, их свойства и средства описания и изображение ;
- алгоритмы эффективной обработки структур данных при организации вычислительных процессов;
- существующие технологии проектирования программных продуктов и особенности их выполнения;
- современные технологии программирования, их возможности, особенности использования;
- использование на разных этапах компьютерной обработки программ;

УМЕТЬ:

- использовать оптимальные структуры данных при решении инженерных и экономических задач, задач управления и построением баз данных с точки зрения потребности минимальных ресурсов;
- определять операции над структурами данных;
- осуществлять отладку и тестирование разрабатываемых программ.

БЫТЬ ОЗНАКОМЛЕНЫМИ:

- с перспективными структурами данных и средствами их обработки на ЭВМ;

- об использовании структур данных в системном программном обеспечении ЭВМ;

Предметом дисциплины "Структуры данных" есть изучение структур данных и организацию их представление в памяти ЭВМ для разработки алгоритмов выполнения вычислений разных форм обработки информации.

Контрольная работа имет целью:

- привить навыки работы с вычислительной техникой;
- практически реализовывать нестандартное представление данных в памяти ЭВМ, для обеспечения нужных показателей разрабатываемых программ.

Порядок выполнения работы:

1. Ознакомится с методическими рекомендациями;
2. Разработать структуры данных и реализовать их в отдельном заголовочном файле;
3. Реализовать функции обработки блочной структуры данных согласно своему варианту.
4. Разработать тестовую программу для демонстрации работы реализованных функций
5. Оформить отчет по работе.

Основой информационно-методическое обеспечение самостоятельной робот студента есть конспект лекций, рекомендованная из дисциплины **учебная и методическая литература.**

СТРУКТУРЫ ДАННЫХ

Совершенно ясно, что систематический и научный подход к построению программ важен в первую очередь в случае больших программ со сложными данными. В конечном итоге программы представляют собой конкретные формулировки абстрактных алгоритмов, основанные на конкретном языке и структурах данных. Алгоритмы и структуры данных всегда используются совместно: выбор алгоритма существенно зависит от структуры данных и наоборот (хотя интуитивно понятно, что структуры данных - первичны). Примечательно, что существуют близкие аналоги между методами структурирования алгоритмов и типов данных. Например, оператор присваивания – соответствует скалярному типу, составной оператор – записи, оператор цикла for – массиву, оператор цикла while – файлу, рекурсивные алгоритмы – деревьям.

Алгоритм – способ решения вычислительной задачи. Алгоритмы всегда абстрактны – они могут быть реализованы на конкретном языке, и на конкретной машине. Основные характеристики алгоритмов – эффективность, надежность.

Структура данных – способ организации данных, состоящий из структуры памяти для хранения данных, способов ее формирования, модификации и доступа к данным.

Абстрактный тип данных (АТД)– характеризуется только набором операций, которые можно выполнять с данными, не уточняя структуру памяти или реализацию операций. Например, стек – АТД, с основными операциями Pop (добавить элемент в стек) и Push (взять элемент из стека). Конкретно реализован он может либо в непрерывном блоке памяти (массиве) или в виде связного списка. Различным образом можно реализовать и конкретные операции со стеком.

К классическим, наиболее распространенным структурам данных относятся массивы, записи, связанные списки, стеки, очереди, деревья.

1. *Массивы*

Массив – упорядоченная линейная совокупность однородных данных.

Комментарии к определению:

- термин «упорядоченная» означает, что элементы массива пронумерованы;
- термин «линейная» свидетельствует о равноправии всех элементов;

- термин «однородных» означает следующее: в том случае, когда массив формируется из элементарных данных, это могут быть данные лишь одного какого-то типа, например, массив чисел или массив символов; однако, возможна ситуация, когда элементами массива окажутся сложные (структурные) данные, например, массив массивов – в этом случае «однородных» означает, что все элементы имеют одинаковую структуру и размер.

  Количество индексов, определяющих положение элемента в массиве, называется ***мерностью*** массива.

  Так, если индекс *единственный*, массив называется ***одномерным***; часто такой массив называют также ***вектором***, строкой или столбцом. Для записи элементов одномерного массива используется обозначение m_i; в языках программирования приняты обозначения m(i) или m[i].

Массив, элементы которого имеют два индекса, называется ***двумерным*** или ***матрицей***. Пример обозначения: G[3,5]; при этом первый индекс является номером строки, а второй индекс – номером столбца, на пересечении которых находится данный элемент.

Массивы с тремя индексами называются ***трехмерными*** и т.д. Максимальная мерность массива может быть ограничена синтаксисом некоторых языков программирования, либо не иметь таких ограничений.

Максимальное значение индексов определяет ***размер*** массива. Размер массива указывается в блоке описания программы, поскольку при исполнении программы для хранения элементов массива резервируется необходимый объем памяти. Если в процессе исполнения программы размер массива не изменяется (или не может быть изменен), то в этом случае говорят о массивах *фиксированного размера*; если определение размеров массива или их изменение происходит по ходу работы программы, то такие массивы называются *динамическими* (динамически описываемыми).

Массивы – наиболее широко известная структура данных, так как во многих языках программирования это была единственная структура, существовавшая в явном виде. Массив – это регулярная структура: все его компоненты одного базового типа. Массив – это структура со случайным (прямым) доступом: все компоненты могут выбираться произвольно и являются одинаково доступными по индексу. К i-му элементу массива A можно обратиться как A[i]. Описание

регулярного типа задает не только базовый тип (TBase), но и тип индекса (TIndex)

.

```
Type TArray = array [Tindex] of Tbase
```

Простейший пример использования массивов – алгоритм Эратосфена для нахождения простых чисел.

```
Program Eratosphen;
Const  N = 1000;
Type Tarray = array[1..N] of boolean;
Var
        a: Tarray; i,j: integer;
        begin
        a[1]:=false; for i:=2 to N do a[i]:=true;
        for i:=2 to N div 2 do
                if a[i] then for j:=i to N div 2 do a[i*j]:=false;
        for i:=1 to N do
                if a[i] then write(i:4);
        end;
```

Решето Эратосфена – типичный пример алгоритма, использующий структуру данных с прямым доступом ко всем элементам массива. Использование связанного списка вместо массива здесь только бы ухудшило производительность, так как программа имела бы неэффективный доступ к элементам.

2. Запись

Запись – последовательность элементов, которые в общем случае могут быть одного типа. На логическом уровне структура данных (СД) типа запись можно записать следующим образом:

type T = Record

S_1: T_1;

S_2: T_2;

……..

S_n: T_n;

End;

Здесь: T_i изменяется при i = 1, 2, …, n; S_1, …, S_n – идентификаторы полей; T_1, …, T_n – типы данных. Если T_i также является в свою очередь записью, то S_i – иерархическая запись.

Если D_{T1} – множество значений элементов типа T_1, D_{T2} – множество значений элементов типа T_2, … , D_{Tn} – множество значений элементов типа T_n, то D_T – множество значений элементов типа T будет определяться с помощью прямого декартова произведения: $D_T = D_{T1} \times D_{T2} \times \ldots \times D_{Tn}$. Другими словами, множество допустимых значений СД типа запись: $Car(T) = \prod_{i=1}^{n} Car(Ti)$

Допустимые операции для СД типа запись аналогичны операциям для СД типа массив.

Дескриптор СД типа запись включает в себя: условное обозначение, название структуры, количество полей, указатель на первый элемент (в случае прямоугольной СД), характеристики каждого элемента, условные обозначения типа каждого элемента, размер слота, а также смещение, необходимое для вычисления адреса.

Вообще, *смещение* – это адрес компоненты (поля) r_i относительно начального адреса записи r. Смещение вычисляется следующим образом:

$k_i = S_1 + S_2 + \ldots + S_{i-1}$, i=1,2, …,n

где S_i – размер слота каждого элемента записи.

Дескриптор СД типа запись, в отличие от дескриптора СД типа массив, зависит от количества элементов записи.

3. Связанные списки

Другой классический тип данных – ***связанные списки, определенные как стандартные структуры в некоторых языках программирования (например, в Лиспе). Элементы в списках хранятся в виде узлов, хранящих кроме данных еще и указатель (ссылку) на другой узел. Последний узел содержит специальный указатель (нулевой) или ссылку на самого себя.***

Связанные списки имеют два преимущества перед массивами:

– они динамично могут изменять свои размеры,

– позволяют легко реорганизовать порядок элементов, добавлять и удалять элементы, путем редактирования небольшого числа ссылок.

Существует несколько видов связанных списков, в зависимости от ссылок: односвязные, двухсвязные, кольцевые. Конкретная реализация односвязного списка может быть представлена следующим образом:

```
Type Link = ^ node;
    node = record
          elem: Tbase;
          next : Link
       end;
```

Основными операциями со списками являются добавление элемента в список после указателя и удаление элемента из списка после указателя.

```
Procedure AddAfter(p:Link; E:Tbase);
var q: Link;
begin
    New(q);
    q^.elem:=E;
    q^.next:=p^.next;
    p^.next:=q;
end;

Procedure DelAfter(p:Link);
var q: Link;
begin
        q:=p^.next;
        p^.next:=q^.next;
        dispose(q);
end;
```

Более неестественными для односвязного списка являются процедуры добавления элемента перед указателем. Однако простой трюк позволяет решить эту проблему: новый элемент на самом деле вставляется после указателя, а

затем происходит обмен значениями между новым элементов и тем, на который показывал указатель.

Другой способ решения этой задачи использует две ссылки, одна из которых отстает от второй на один шаг. После того как передняя ссылка достигнет указателя, операция добавления сведется к добавлению после отстающей ссылки. Для реализации этого алгоритма необходимо иметь в начале списка фиктивный элемент – голову Head.

```
    Procedure AddBefore(p:Link; E:Integer);
var q1,q2: Link;
begin
   q1:=Head;
   q2:=q1^.next;
   while q2<>p do
   begin
   q1:=q2;
   q2:=q2^.next;
   end;
   AddAfter(q1,E);
end;
```

В качестве примера использования циклического списка рассмотрим проблему Джозефа. N человек по кругу совершают самоубийство, убивая каждого M человека. Надо найти порядок смерти всех людей. Например, если N=9, M=5, то порядок будет следующий: 5 1 7 4 3 6 9 2 8.

```
    Type Link = ^ node;
        node = record
                elem: integer;
                next : Link
           end;
    var i,
       N,
       M : integer;
       L, P : Link;
```

```
begin
    read(N,M);
    new(P);
    P^.elem:=1;
    L:=P;
    for i := 2 to N do
    begin
        new(p^.next);
        p:=p^.next;
        p^.elem:=i
    end;
    p^.next:=L;
    while p <> p^.next do
    begin
        for i:=1 to M-1 do p:=p^.next;
        write(p^.next^.elem:3);
        L:=p^.Next;
        p^.next:=L^.next;
        dispose(L)
    end;
    writeln(p^.elem:3);
    dispose(p)
end.
```

Использование массива в данном алгоритме ухудшит программу, так как ее эффективность зависит от того насколько быстро удаляется элемент.

В заключение заметим, что связанные списки могут быть конкретно реализованы на массивах, используя вместо указателей параллельный массив индексов. Все данные будут содержаться в массиве Elem, а вся структура списка – в массиве Next. Например, список с заглавным и конечным звеном изображен на рисунке. Здесь next[0]=3; поэтому список начинается с элемента elem[3]=H и читается как HELLO.

Elem Next

	3
E	4
O	6
H	1
L	5
L	2
	6

Реализация процедуры удаления из списка после указателя будет выглядеть следующим образом:

```
Procedure DelAfter(p:integer);
begin
        next[p]:=next[next[p]]
end;
```

4. Стеки

Стек (магазин) является упорядоченной, линейной, неоднородной структурой. Эти структура реализуются в виде специальным образом организованных областей ОЗУ компьютера либо в качестве самостоятельных блоков памяти. В стеке ячейки памяти (или регистры стековой памяти) соединяются друг с другом таким образом, что при занесении данных в первую ячейку содержимое всех остальных сдвигается в соседние вниз, при считывании – содержимое сдвигается вверх по ячейкам, как показано на Рис.3.1. Другими словами, вход в стек возможен только через первую ячейку (***вершину стека***), поэтому извлекаться первой будет та информация, которая была занесена последней, подобно пассажиру переполненного автобуса. (Часто стек называют памятью типа LIFO (**L**ast–**I**n **F**irst–**O**ut : последним вошел – первым вышел).). Отличие очереди от стека только в том, что извлечение информации производится в порядке «первым вошел – первым вышел», т.е. со дна стека.

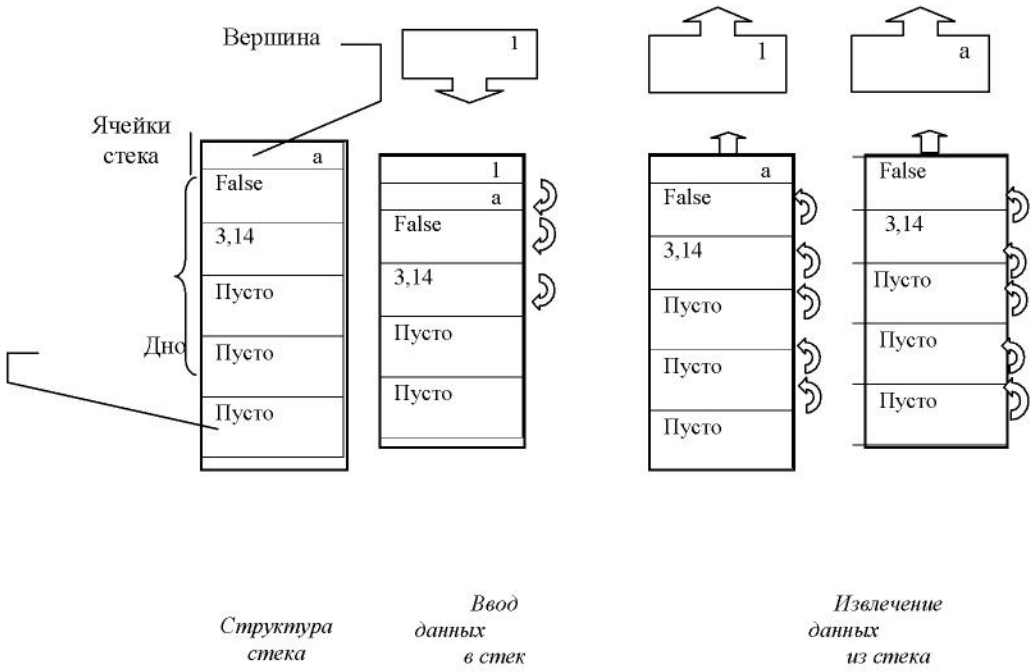


Рис. 3.1. Структура стека

Таким образом, данные имеют порядок расположения и они равноправны – поэтому структура упорядоченная и линейная. Однако в общем случае в ячейках стека могут содержаться данные разных типов – по этому признаку структура оказывается неоднородной.

Описанный способ организации данных оказывается удобным при работе с подпрограммами, обслуживании прерываний, решении многих задач.

Основные процедуры для стека – Push (элемент заносится «заталкивается» в стек) и Pop (элемент забирается “выталкивается” из стека).

Конкретная реализация стека может быть выполнена как на списке с указателями, так и на массиве. Например, реализация основных операций на списочной структуре может выглядеть следующим образом:

```
Type Link = ^node;
    node = record
              elem: integer;
              next: link;
```

```
        end;
    Stack = record head,z : Link end;

procedure StackInit(var S : Stack);
begin
 with S do
 begin
  new(head); new(z);
  head^.next:=z; z^.next:=z;
 end;
end;

function StackEmpty(S: Stack): boolean;
begin
    StackEmpty:=S.head^.next = S.z
end;

procedure Push(var S: stack;v: integer);
var P : Link;
begin
    new(P);
    P^.elem:=v;
    P^.next:=S.head^.next;
    S.head^.next:=P
end;

function Pop(var S: stack): integer;
var P: link;
begin
    P:= S.head^.next;
    Pop:=P^.elem;
    S.head^.next:=P^.next;
    dispose(P);
end;
```

5. Очереди

Еще одна классическая структура с ограниченным доступом к данным это очередь. Способ доступа к данным ограничен 2 концами - началом и концом очереди по принципу «первый пришел, первый ушел» (FIFO - first in, first out)

Рассмотрим пример реализации основных операций - добавить в начало очереди Insert, удалить из конца очереди Remove.

```
Type Link = ^node;
    node = record
             elem: integer;
                    next: link;
                end;
           Queue = record head,tail : Link end;

        procedure QueueInit(var Q : Queue);
        begin
         with Q do
         begin
           new(head); new(tail);
           head^.next:=tail; tail^.next:=tail;
         end;
        end;

        function QueueEmpty(Q: Queue): boolean;
        begin
            QueueEmpty:=Q.head^.next = Q.tail
        end;

        procedure Insert(var Q: Queue; v: integer);
        var P : Link;
        begin
            new(P);
            Q.tail^.elem:=v;
```

```
    Q.tail^.next:=P;
    P^.next:=P;
    Q.tail:=P;
end;

function Remove(var Q: Queue): integer;
var P: link;
begin
    P:= Q.head^.next;
    Remove:=P^.elem;
    Q.head^.next:=P^.next;
    dispose(P);
end;
```

6. Деревья поиска

Дерево или ***иерархия*** является примером *нелинейной* структуры. В ней элемент каждого уровня (за исключением самого верхнего) входит в один и только один элемент следующего (более высокого) уровня. Элемент самого высокого уровня называется ***корнем***, а самого нижнего уровня – ***листьями***. Отдельные элементы могут быть однородными или нет. Примером подобной организации служат файловые структуры на внешних запоминающих устройствах компьютера.

Деревья одна из наиболее распространенных динамических структур в вычислительных алгоритмах. Деревья впервые появились для манипуляции с формулами при развитии компиляторов в 1951, 1952-54гг. Первый обзор был сделан в 1961 г. в Исследовательском отчете корпорации IBM.

Определим дерево, как конечное множество T, которое либо пусто, либо в нем имеется один специально обозначенный узел, называемый корнем, а все остальные узлы содержатся в непересекающихся множествах T_1, T_2,..., T_m, каждое из которых является деревом. Деревья T_1, T_2,..., T_m называют поддеревьями данного корня.

Заметим, что это определение рекурсивно, так как определяет дерево в терминах самого дерева. Основной характеристикой этих динамических структур является рекурсивность. Поэтому рекурсивные алгоритмы обработки деревьев являются наиболее уместными.

ПРИМЕР:

m=0 имеем 1 корень

m=1 имеем 1 корень и 1 поддерево

a

b

добавим еще 1 узел

a a

b c b

c

Имеем m=1 m=2

добавим еще 1 узел

a a a a

b b c b c b d

c d c d

d

Имеем m=1 m=1 m=2 m=3

...

Другие способы изображения деревьев

1. вложенные множества
2. вложенные скобки
3. как списки со сдвигом

Число поддеревьев m узла называется степенью узла. Степень дерева - это максимальная из степеней всех узлов дерева. Если относительный порядок поддеревьев T1, T2,..., Tm важен, то говорят, что дерево упорядочено. Упорядоченное дерево степени два называется бинарным деревом. Таким образом, в бинарном дереве каждый узел имеет не более чем два поддерева, которые называются левым и правым поддеревом, соответственно.

Основные определения

1. **Степень узла** - число поддеревьев, выходящих из этого узла.
2. **Степень дерева** - максимальная из всех степеней узлов.

3. **Лист (**терминальный узел, внешний узел) - узел степени 0.
4. **Уровень узла** - количество узлов на пути от корня до этого элемента.
5. **Высота дерева** - максимальный из уровней узлов
6. **Упорядоченное дерево** - дерево, у которого важен порядок следования поддеревьев.

a a

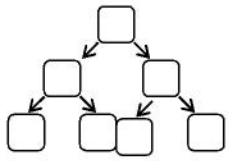
c b b c

7. **Бинарное дерево** - упорядоченное дерево степени 2.

ПРИМЕРЫ ДЕРЕВЬЕВ

1. Генеалогическое дерево

а) родословная (родители лица)- бинарное дерево



б) родовая схема (потомки лица) - сильно ветвящееся дерево

Ной

Сим (5)

Елам Ассур Арфаксад Луд Арам

Хам(4)

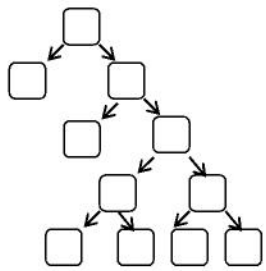
Хуш Мицраим Фут Ханаан

Иафет (7)

Гомер Магог Мадай Иаван.Фувал Мешек Фирас

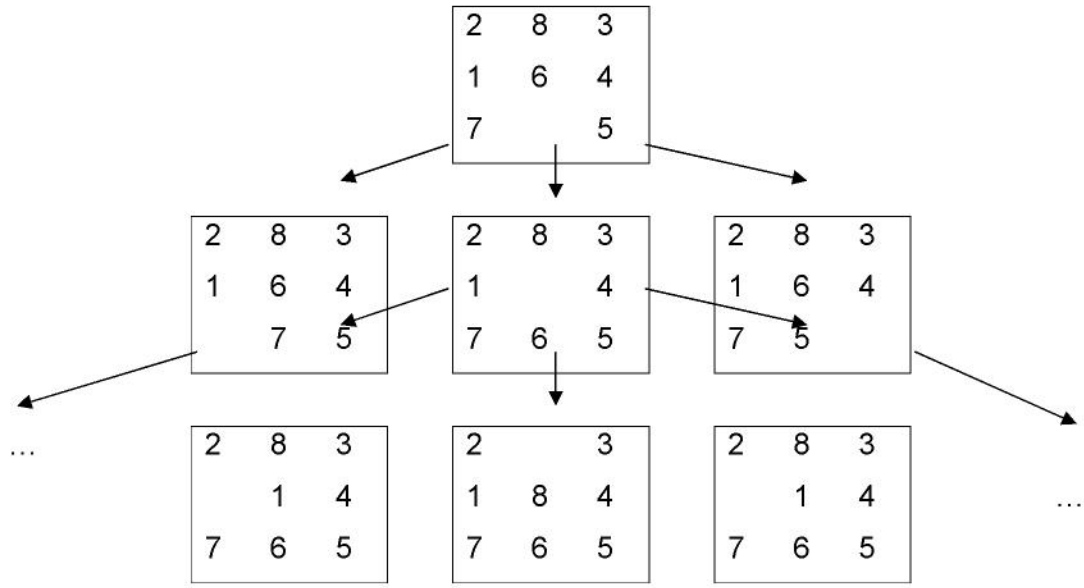
2. Представление алгебраических выражений

a-b/(c/d+e/f)



3. Стратегии игр

Игра в пятнашки (в восьмерки)



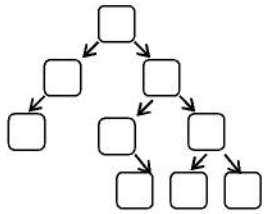
Опишем дерево как указатель на узел следующим образом:

```
type Link = ^Node;
     Node = record
            elem: Tbase;
            left, right: Link
            end;
```

Узел состоит из трех полей: в поле elem хранится элемент, а поля left и right хранят ссылки на левое и правое поддеревья, соответственно.

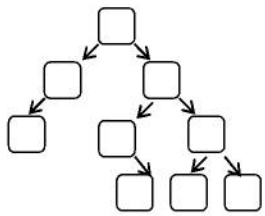
```
Procedure PrintTree(T: Link; h: integer);
var i: integer;
begin
if T<> nil then
  begin
     PrintTree(T^.right, h+1);
     for i:= 1 to h do  write('   '); writeln(T^.elem);
     PrintTree(T^.left,h+1);
  end;
end;
```

Обходы деревьев



Обход дерева в префиксном порядке: abdcegfhj

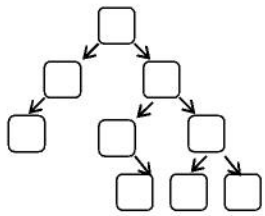
```
Procedure PreOrderR(T: Link);
begin
if t<> nil then
  begin
     write(T^.elem:3);
     PreOrderR(T^.left);
     PreOrderR(T^.right);
  end;
end;
```



Обход дерева в постфиксном порядке: dbaegchfj

```
Procedure PostOrderR(T: Link);
begin
if t<> nil then
  begin
     PostOrderR(T^.left);
     write(T^.elem:3);
```

```
      PostOrderR(T^.right);
  end;
end;
```



Обход дерева в ендфиксном порядке: dbgehjfca

```
Procedure EndOrderR(T: Link);
begin
if t<> nil then
  begin
      EndOrderR(T^.left);
      EndOrderR(T^.right);
      write(T^.elem:3);
  end;
end;
```

Особый вид бинарного дерева - дерево поиска, организованное таким образом, что для каждого узла T все элементы в левом поддереве меньше элемента узла T, а все элементы в правом поддереве больше элемента узла T.

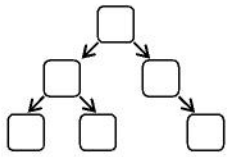


Рис. 6.1. Дерево поиска.

Деревья поиска играют особую роль в алгоритмах обработки данных. Они используются в лексикографических задачах, построениях частотных словарей, задачах сортировки данных. Основное достоинство этой структуры данных – то, что она идеально подходит для решения задачи поиска. Место каждого элемента можно найти, двигаясь от корня влево или вправо в зависимости от значения элемента.

Реализация нерекурсивного алгоритма поиска может выглядеть следующим образом:

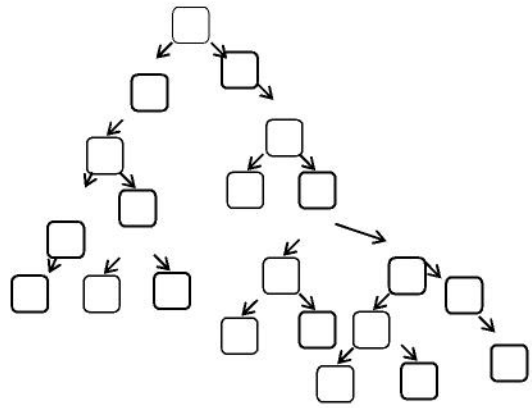
```
function Locate(x: integer; T: Link):Link;
var found: boolean;
begin
    found:= false;
    while (T <> nil) and not found do
        if T^.elem = x then found := true
        else if T^.elem > x then T:=T^. left
            else T:=T^.right;
    Locate := T
end;
```

Для упрощения процедуры поиска можно использовать дерево с узлом-барьером (последний пустой узел z, в который засылается элемент для поиска x)

```
function Locate(x: integer; T: Link):Link;
begin
    z^.elem:=x;
    while T^.elem <> x do
        if T^.elem > x then T:=T^. left else T:=T^.right;
    Locate := T
end;
```

Основные операции при работе с деревьями поиска - поиск по дереву с включением нового элемента и поиск по дереву с исключением элемента. Эти задачи широко используются в том случае, когда дерево растет или сокращается в ходе самой программы.

Предположим, что нам нужно последовательно добавить в дерево поиска 21 узел с элементами 8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18. В результате будет построено следующее дерево:



Включение нового элемента в дерево поиска возможно единственным способом и осуществляется очень простой рекурсивной процедурой. Добавление нового элемента x сводится к рекурсивному поиску нужного места в левом или правом поддереве, в зависимости от соотношения x и элемента, хранящегося в корне T^.elem:

```
Procedure Search(x: integer; var T: Link);
begin
    if T = nil then
    begin new(T); T^.elem:=x; T^.left:=nil;T^.right:=nil; end
    else
    if x<T^.elem then search(x, T^.left)
    else  if x>T^.elem then search(x, T^.right);
end;
```

Время работы этого алгоритма очень сильно зависит от формы дерева. В лучшем случае, для поиска места для добавления узла в дерево, содержащее N элементов, может понадобиться не более logN сравнений, если дерево идеально сбалансировано. В худшем случае, в бинарном дереве, построенном из N произвольно взятых ключей, может потребоваться около N сравнений.

Задача удаления узла из дерева реализуется несколько сложнее. Различаются три случая удаления элемента x:

Элемента x нет в дереве.

Узел, содержащий элемент x, имеет степень не более 1.

Узел, содержащий элемент x, имеет степень 2.

Случаи 2 и 3 показаны на рис.6. 2 и рис. 6.3 соответственно.

Рис. 6.2. Удаление узла степени не более 1 (элемент 6)

Случай 2 не представляет сложности. Предыдущий узел соединяется либо с единственным поддеревом удаляемого узла (если степень удаляемого узла равна 1), либо не будет иметь поддерева совсем (если степень узла равна 0)

Намного сложнее, если удаляемый узел имеет два поддерева. В этом случае будем заменять удаляемый элемент самым правым элементом из его левого поддерева (можно использовать симметричный метод: самым левым элементом из правого поддерева).

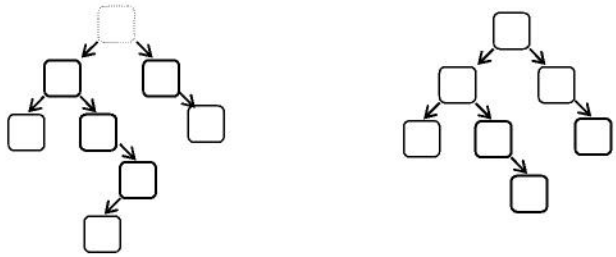
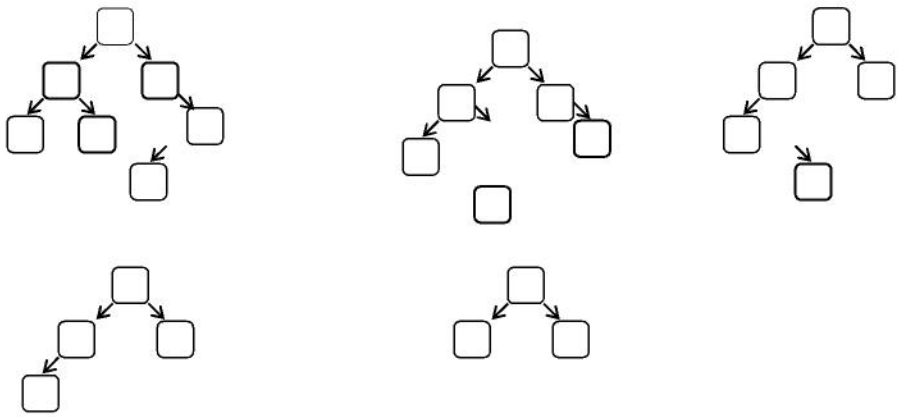


Рис.6. 3. Удаление узла степени 2 (элемент 6)

Пусть задано дерево поиска из 7 элементов: 10 15 18 5 13 8 3. Удалив из этого дерева последовательно элементы 13 15 5 10, имеем:



Рекурсивный алгоритм поиска узла с исключением выглядит следующим образом:

```
procedure Delete(x: integer;var T:Link);
var P:Link;
procedure Del(var R: Link);
begin if  R^.right <> nil then del(R^.right)
      else begin P^.elem:=R^.elem; P:=R; R:=R^.left end;
end;
begin if T=nil then writeln('такого элемента в дереве нет!')
      else if x < T^.elem then delete(x,T^.left)
      else if x > T^.elem then  delete(x,T^.right)
      else begin P:=T;
            if P^.right = nil then T:=P^.left
            else if P^.left = nil then T:=P^.right
            else del(P^.left);
            dispose(P)
      end;
end;
```

Рассмотрим те же задачи без использования рекурсии. Нерекурсивный метод поиска по дереву с включением узла не представляет сложностей и предлагается для самостоятельной работы.

Для реализации нерекурсивного метода поиска по дереву с исключением узла надо помнить пройденный путь хотя бы на один шаг назад. Это означает, что надо иметь указатель на предыдущий узел и знать по какому поддереву (левому или правому) мы перешли от него к удаляемому узлу.

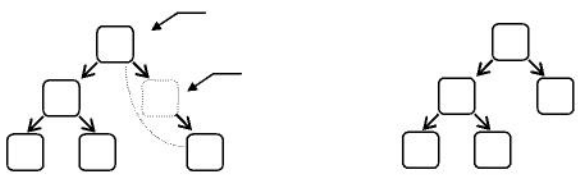


Рис.6. 4. Нерекурсивное удаление узла степени меньше 2 (элемент 6)

Например (рис.6.4), если указатель P_1 установлен на удаляемый элемент (6), имеющий одно правое поддерево, а указатель P_2 установлен на предыдущий узел и направление пути от P_2 к P_1 - правое, то удаление узла сводится к изменению связи P_2^.right:=P_1^.right. Если направление пути было левое, тогда связь меняется следующим образом P_2^.left:=P_1^.right. Если удаляемый элемент по адресу P_1, имеет одно левое поддерево, то при правом повороте будем устанавливать связь P_2^.right:=P_1^.left, а при левом повороте P_2^.left:=P_1^.left. Если удаляемый элемент по адресу P_1, не имеет ни одного поддерева, тогда выполнено условие P_1^.left = P_1^.right = nil и предыдущие связи также устанавливают правильный вид дерева.

При удалении узла степени два, необходимо установить два дополнительных указателя в левом поддереве удаляемого узла: указатель R_1 на самый правый элемент этого поддерева, а указатель R_2 на его предыдущий узел. Удаление узла сводится теперь к следующим изменениям: элемент из узла по адресу R_1 пересылается в узел по адресу P_1 и узел R_1 исключается с помощью связи R_2^.right:=R_1^.left (рис.6. 5).

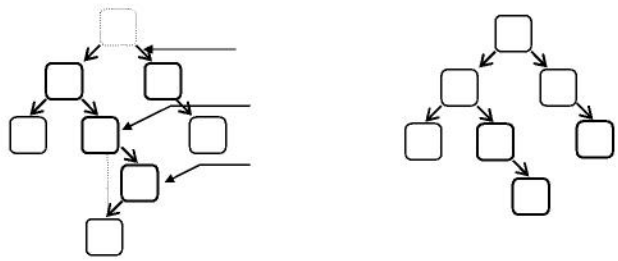


Рис.6. 5. Нерекурсивное удаление узла степени 2 (элемент 6)

На первый взгляд кажется, что направление движения по левому поддереву запоминать не надо, так как мы двигаемся к его самому правому элементу, то есть поворачиваем все время вправо. Но это правило нарушается в единственном случае, когда у левого поддерева нет правой ветви (рис.6. 6). Удаление узла сводится теперь к следующим изменениям: элемент из узла по адресу R_1 пересылается в узел по адресу P_1 и узел R_1 исключается с помощью связи

R_2^.left:=R_1^.left. Таким образом, все-таки придется запоминать направление поворотов в левом поддереве.

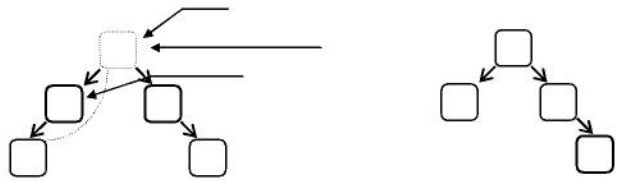


Рис.6. 6. Нерекурсивное удаление узла степени 2 без правой ветви левого поддерева (элемент 6)

Заметим также, что в случае, когда удаляемый элемент находится в корне, для предыдущего указателя P_2 нет места в дереве. Эта проблема является стандартной проблемой обработки граничных данных. Для ее решения можно использовать стандартный прием: рассмотрим дерево с пустым узлом перед корнем. Тогда указатель P_2 устанавливается на пустой узел и алгоритм становится корректным (рис.6. 7).

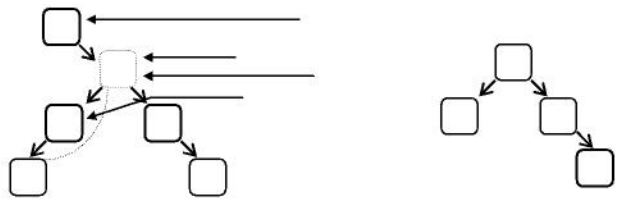


Рис.6. 7. Удаление корня дерева.

Чтобы реализовать дерево с пустым узлом, необходимо изменить процедуру инициализации пустого дерева: теперь пустое дерево это не указатель на nil, а указатель на пустой узел. При этом оговаривается для конкретности, что само дерево начинается с правого поддерева пустого узла, то есть первый поворот всегда будет правый.

```
procedure TreeInitialize(var T: Link);
begin
        new(T);
        T^.left:=nil;
        T^.right:=nil
end;
procedure Delete(x: integer; T:Link);
var     P1,P2, R1, R2:Link;
```

```
        found, left: boolean;
begin
        P2:=T; {Установка указателей P2 и P1, поворот направо}
        P1:=T^.right;
        left :=false;
        found:= false;
      while (P1 <> nil) and not found do
            if P1^.elem = x then found := true
            else if P1^.elem > x then
                {поворот налево}
                begin P2:=P1; P1:=P1^. left; left := true
                end
                else {поворот направо}
                begin P2:=P1; P1:=P1^.right; left :=false
                end;
      if P1=nil then writeln('такого элемента в дереве нет!')
      else begin
        {удаление узла степени не более 1}
        if P1^.right = nil then if  left then P2^.left:=P1^.left
                                else P2^.right :=P1^.left
        else if P1^.left = nil then if  left then P2^.left:=P1^.right
                                    else P2^.right :=P1^.right
            else begin
            {удаление узла степени 2}
               R2:=P1; R1:=P1^.left; left:=true;
               while R1^.right <> nil do
                  begin R2:=R1; R1:=R1^.right; left:=false end;
               P1^.elem:=R1^.elem;
               if left then R2^.left:=R1^.left
               else R2^.right:=R1^.left;
               P1:=R1
            end;
        dispose(P1)
      end
end;
```

ЗАДАНИЕ

Задание: Разработать информационную системы с применением динамических структур данных. Для решения поставленной задачи рекомендуется использовать динамические структуры (списки, деревья, очереди, стеки и т.п.) в том случае, если для решения поставленной задачи их использование окажется более целесообразным. Обеспечить возможность выполнения следующих операций над выбранными структурами данных:

- инициализацию;
- добавление новых элементов;
- удаление элементов;
- перемещение по структуре данных;
- поиск элементов структуры данных, отвечающих заданным критериям;
- вывод всех элементов структуры данных на экран.

Содержание пояснительной записки:

- задание на контрольную работу
- назначение и область применения разрабатываемой программы
- основные возможности и характеристики программы
- постановка задачи
- структурная схема фрагмента информационной системы
- таблица имен
- иерархия объектов
- инструкция по работе с программой
- заключение и выводы
- литература.

Номер варианта для выполнения контрольной работы выбирается по последней цифре зачетной книжки.

Варианты:

0. билетная касса автовокзала
1. склад
2. поликлиника
3. отдел кадров

4. программа телепередач
5. планирование работ
6. спортивные соревнования
7. телефонный справочник
8. учет аудиторного фонда
9. каталогизатор дискет и компакт-дисков.

ЛИТЕРАТУРА

1. Альфред Ахо, Джон Э. Хопкрофт, Д. Ульман. Структуры данных и алгоритмы.: Пер. с англ.: Уч. пос. - М.: Издательский дом "Вильямс", 2001. – 384 с.
2. Арт Фридман, Ларс Кландер, Марк Михаэлис, Херб Шильдт. C/C++ Алгоритмы и приемы программирования.-М.: Издательство БИНОМ,2003.-560с.
3. Браун Кен. Основные концепции структур данных и реализация в C++. -М.: Мир., 2002.– 320 с.
4. Голубь. Н.Г.. Искусство программирования на ассемблере. Лекции и упражнения.СПб.: ООО «ДиаСофтЮП», 2002. – 656 с.
5. Д. Кнут. Искусство программирования для ЭВМ. Т.1. Основные алгоритмы. - М.:"Мир", 1976. - 734 с.
6. Д. Кнут. Искусство программирования для ЭВМ. Т.3. Сортировка и поиск. М., "Мир", 1978 г., переиздание - М.,Изд-во "Вильямс", 2000 г.
7. Майкл Мейн, Уолтер Савитч. Структуры данных и другие объекты в C++: Пер. с англ.-М.: Издательский дом “Вильямс”, 2002. – 832 с.
8. Майкл Т. Гудрич, Роберто Тамассия. Структуры дынных и алгоритмы в Java. -Минск: ООО «Новое знание», 2003.-670с.
9. Н. Вирт. Алгоритмы и структуры данных.- М.:Мир – 2001.– 352 с.
10. Т. Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. -М.: МЦНМО, 2001. – 960 с.
11. Юров В.. ASSEMBLER - Санкт-Петербург, ПИТЕР, 2001. – 780 с.
12. Э. Майника. Алгоритмы оптимизации на сетях и графах.–М.: Мир., 1981.– 324 с.