Лабораторная работа №2
по курсу «Структуры и алгоритмы обработки данных»

ЗАДАЧИ ОБХОДА ДЕРЕВЬЕВ

1. Основные сведения

1.1. Деревья как структуры данных

Деревом мы будем называть структуру данных, которая:

- либо является пустой, т.е. не содержит данных;
- либо состоит из вершины, называемой ***корнем*** дерева, которая содержит некоторые данные (***значение вершины***) и из которой исходят связи, направленные к конечному числу (может быть и нулевому) непересекающихся деревьев.

Приведенное определение является ***рекурсивным***, т.е. содержит ссылку на определяемое понятие. В классической логике подобные определения считались ошибочными, однако в современной дискретной математике они вполне допустимы, если только содержат хотя бы одну нерекурсивную альтернативу (в данном случае – пустое дерево).

Если в дереве имеется связь от вершины A к вершине B, то A называют ***отцом*** вершины B (а B, соответственно, ***сыном*** A). Расширяя эту аналогию, можно говорить о ***потомках*** и ***предках*** вершины. Корень дерева является общим предком всех остальных вершин.

Любая вершина дерева может рассматриваться в качестве корня для ***поддерева***, состоящего из самой этой вершины и всех ее потомков.

Вершина, не имеющая сыновей, называется ***листом***. Все остальные вершины дерева называются ***внутренними***. Путь, ведущий из корня дерева к какому-либо листу, называется ***ветвью*** дерева. Поскольку этот путь единственный, общее количество ветвей равно количеству листьев. Число вершин ветви, включая начальную и конечную, называется ***длиной ветви***. Наибольшая из длин ветвей называется ***высотой дерева***.

Бинарным (двоичным) называется дерево, каждая вершина которого имеет не более двух сыновей. На практике часто удобно считать, что каждая вершина бинарного дерева имеет ровно двух сыновей (условно называемых ***левым*** и ***правым*** сыновьями), но один из сыновей или оба они могут быть пустыми.

Основным назначением деревьев в информатике является представление различных иерархических структур, от структуры каталогов дискового тома до арифметических выражений с вложенными скобками. Кроме того, деревья широко используются в задачах сортировки и поиска данных.

1.2. Представление бинарных деревьев

Известны разнообразные формы представления деревьев с помощью массивов, матриц, сцепленных структур и т.п. Выбор наиболее удобной формы зависит от поставленной задачи и выбранного алгоритма решения. В данной работе не приводится обзор всех форм представления деревьев. Рассматриваются две формы, одна их которых удобна для представления деревьев «на бумаге» и может использоваться при вводе и выводе, а другая позволяет эффективно реализовать операции обхода деревьев и операции, изменяющие структуру дерева.

1.2.1. Представление дерева многоуровневым текстом

Подобная форма представления знакома всем по оглавлениям книг и отчетов научно-технического характера. Названия крупных разделов печатаются, начиная от левого поля листа, а названия входящих в них подразделов – с отступом, причем чем глубже по иерархии расположен подраздел, тем больше отступ. Другой отличительный признак разных уровней дерева – структура номера подраздела. Номер каждого подраздела содержит на одну точку больше, чем номер раздела-«родителя». При этом выделение отступами становится необязательным. В качестве примера иерархического дерева текста можно привести данный текст описания лабораторной работы.

Чтобы применить данную форму представления к бинарным деревьям, нужно договориться о том, как различать «левого» и «правого» сына каждой вершины. Можно, например, установить, что корень дерева имеет номер ‘0’ (строку, а не число!), его левый сын – номер ‘00’, правый сын – номер ‘01’ и т.д., то есть номер сына отличается от номера отца добавлением символа ‘0’ для левого сына или ‘1’ для правого. Исключительно для улучшения зрительного восприятия можно сохранить также отступ на одну-две позиции для каждого следующего уровня дерева. Что же касается данных, задающих значения вершин, то, если эти данные невелики по объему, их проще всего разместить в той же строке, отделив от номера вершины пробелами.

В примере программы, приложенном к описанию работы, используется именно такое входное и выходное представление дерева, как описано выше. Разумеется, эта форма представления не претендует на роль стандарта и при выполнении варианта работы студент может выбрать иное представление дерева.

1.2.2. Представление бинарного дерева сцепленной структурой

В качестве внутренней формы представления бинарного дерева в программе удобно выбрать структуру, состоящую из записей, соответствующих вершинам дерева и содержащих указатели на сыновние вершины. Соответствующие типы данных могут быть описаны таким образом:

```
type
  Element = ...;
  Tree = ^Node;
  Node = record
    Data:  Element; {Значение вершины}
    Left:  Tree; {Левый сын}
    Right: Tree; {Правый сын}
  end;
```

Здесь тип **Element** может представлять собой любой тип данных, удобный для конкретной задачи: целое или вещественное число, строку и т.п. Тип дерева есть просто указатель на корневую вершину. Пустому дереву соответствует значение **nil**. Каждая вершина состоит из поля данных, задающих значение вершины, и указателей на левого и правого сыновей, которые сами, по определению, являются деревьями.

Память для хранения вершин дерева может выделяться динамически, по мере создания дерева, с помощью процедуры **New**. В случае удаления вершины ее память должна освобождаться с помощью процедуры **Dispose.**

Как альтернативу, можно размещать вершины дерева в обычном массиве записей. При этом тип данных **Tree** лучше сделать не указателем на корневую запись, а просто целочисленным значением индекса элемента массива, содержащего эту запись. Соответственно, пустому дереву должно соответствовать значение, выходящее за пределы индексов массива (например, 0, если индексы массива начинаются с 1). Если в задаче

требуется удалять некоторые вершины, то можно ввести дополнительное поле записи, содержащее флаг удаления.

1.3. Задачи обхода дерева

Обход дерева – это общее название для разнообразных процедур, которые предусматривают просмотр всех вершин дерева и выполнение заданных операций в каждой вершине.

В качестве примеров простейших задач обхода дерева можно привести подсчет числа всех вершин дерева или подсчет только его листьев, определение высоты дерева, определение суммы или максимума данных, связанных с вершинами.

Алгоритмы решения задач обхода дерева, как правило, удобнее всего формулировать в рекурсивном виде, путем сведения задачи, поставленной для всего дерева, к аналогичным задачам для поддеревьев, порожденных сыновьями корня. Так например, число вершин непустого дерева, очевидно, равно суммарному числу вершин его поддеревьев плюс единица. В связи с этим наиболее естественной формой для программирования алгоритмов обхода являются рекурсивные процедуры или функции.

Большое количество алгоритмов обхода для случая бинарных деревьев может быть представлено следующей схемой:

```
procedure Traversal(T: Tree);
begin
  if T = nil then begin
    Обработка для случая пустого дерева;
  end
  else begin
    Первое посещение корня;
    Traversal(T^.Left); {Рекурсивный вызов для левого поддерева}
    Второе посещение корня;
    Traversal(T^.Right); {Рекурсивный вызов для правого поддерева}
    Третье посещение корня;
  end;
end;
```

Обработка для случая пустого дерева часто (но не всегда) сводится к «ничего не делать».

Под «посещением корня» здесь имеются в виду какие-то действия, которые следует выполнить, находясь в корневой вершине дерева.

В значительной части задач достаточно выполнить одно посещение корня. Например, для задачи подсчета вершин дерева достаточно один раз увеличить счетчик вершин, чтобы учесть корневую вершину. В зависимости от того, выполняется ли посещение вершины до обхода поддеревьев, между обходами или после обходов, говорят о направлении обхода «***сверху вниз***», «***слева направо***» или «***снизу вверх***».

Часто встречаются задачи обхода, которые в исходной постановке не позволяют дать рекурсивную формулировку алгоритма. В таких случаях обычно ищут более общую постановку задачи, допускающую рекурсивное решение. В итоге решение исходной задачи сводится к разработке двух процедур:

- рекурсивной процедуры для решения общей задачи;
- нерекурсивной процедуры-оболочки для исходной задачи, роль которой сводится к инициализации некоторых переменных (если это необходимо) и вызову рекурсивной процедуры с подходящими значениями параметров.

В качестве примера можно привести процедуру печати дерева из программы, прилагаемой к данному описанию. Рекурсивная процедура **`RecursePT`** выполняет печать любого поддерева в формате, описанном в подразделе 1.2.1. Ее входным параметром

является номер вершины-корня, к которому будут добавляться ‘0’ и ‘1’ для поддеревьев. Нерекурсивная оболочка **PrintTree** нужна только для того, чтобы задать исходное значение номера. В такой же паре процедур для ввода дерева – **ReadTree** и **RecurseRT** – нерекурсивная оболочка выполняет несколько больше работы, связанной с инициализацией переменных и проверкой на пустоту файла.

При разбиении программы на рекурсивную и нерекурсивную части следует внимательно отнестись к выбору параметров рекурсивной процедуры. В приведенной схеме единственным параметром является обрабатываемое дерево. В общем случае процедура может потребовать дополнительных параметров, однако для рекурсивной процедуры заслуживают быть параметрами только те величины, которые действительно должны передаваться в виде отдельной копии при каждом вызове. Те величины, которые могут храниться в единственном экземпляре и совместно использоваться всеми рекурсивными вызовами, лучше описывать как глобальные (по отношению к рекурсивной процедуре) переменные. Это, как минимум, позволит уменьшить размер используемого стека, а иногда позволяет также избежать ошибок в программе.

1.4. Устранение рекурсии при программировании обхода

Рекурсивный подход является наиболее естественным при разработке процедур, работающих с деревьями, ибо он соответствует рекурсивной природе самих деревьев. В то же время известно, что эффективность программы практически всегда может быть улучшена, если программист сумеет «заменить рекурсию итерацией», т.е. вместо последовательности рекурсивных вызовов будет выполнять циклическую обработку данных.

Фактически выполнение рекурсивного вызова означает неявное сохранение в стеке значений формальных параметров, локальных переменных процедуры, а также адреса возврата из процедуры и другой информации, которую компилятор сочтет нужным сохранить (например, значения регистров процессора). Отказываясь от рекурсии, программист обычно вынужден явно работать со стеком (возможно, описав для этого переменную-стек), но при этом он может более точно определить, какие данные следует сохранять в стеке, а какие нет. Кроме того, отпадает необходимость заносить в стек адреса возврата, а также исключается многократное выполнение таких операций, как вызов процедуры и возврат из нее. В результате удается уменьшить как время работы программы, так и размер памяти, используемой для стека.

Замена рекурсии итерацией – не очень простая операция, требующая некоторых интеллектуальных усилий со стороны программиста. Кроме того, получающаяся в результате итеративная программа, как правило, больше по размеру и менее понятна, чем исходная рекурсивная. Поэтому, если главным критерием является минимизация трудозатрат на разработку программы, то лучше остановиться на рекурсивной реализации. Если же требуется получить максимально эффективную программу, то желательно устранить рекурсию. В любом случае, квалифицированный программист должен владеть обоими подходами к программированию процедур обработки рекурсивных структур, чтобы в конкретных случаях выбрать из них наиболее подходящий.

Для процедур обхода дерева преобразование рекурсии в итерацию выполняется достаточно просто. Рассмотрим это на примере обхода дерева «сверху вниз». Исходный рекурсивный вариант выглядит следующим образом:

```
procedure UpDownTraversRec(T: Tree);
begin
  if T = nil then begin
    Обработка для случая пустого дерева;
  end
  else begin
```

```
    Посещение корня;
    UpDownTraversRec(T^.Left); {Для левого поддерева}
    UpDownTraversRec(T^.Right); {Для правого поддерева}
  end;
end;
```

Для обеспечения нерекурсивного варианта будет описана переменная **Stack**, хранящая значения типа **Tree**. В данном простом случае вряд ли оправдано определять для стека процедуры занесения (**Push**), извлечения (**Pop**) и проверки пустоты (**Empty**), соответствующие действия будут записаны в тексте основной процедуры.

```
const
  N = ...; {Глубина стека; должна быть не меньше числа вершин дерева}
procedure UpDownTraversNonRec(T: Tree);
var
  Stack: array[1..N] of Tree;
  Top: Integer; {Указатель вершины стека}
begin
  Top := 1;
  Stack[Top] := T; {В стеке корень дерева}
  while Top > 0 do begin {Пока стек не пуст}
    T := Stack[Top]; {Взять поддерево из вершины стека}
    Top := Top - 1;
    if T = nil then begin
      Обработка для случая пустого дерева;
    end
    else begin
      Посещение корня;
      Stack[Top+1] := T^.Left; {Занести два поддерева в стек}
      Stack[Top+2] := T^.Right;
      Top := Top + 2;
    end;
  end;
end;
```

Суть преобразования заключается в том, что все поддеревья, для которых раньше выполнялся рекурсивный вызов, теперь помещаются в стек, откуда они будут выбраны для просмотра на следующих итерациях цикла. При этом важно обеспечить правильный порядок просмотра вершин, соответствующий требуемому направлению обхода, в данном случае – «сверху вниз».

Рассмотрим теперь, как можно выполнить нерекурсивный обход «слева направо». В этом случае тело процедуры может выглядеть так:

```
begin
  Top := 0;
  repeat
    if T <> nil do begin
      Top := Top + 1;
      Stack[Top] := T;
      T := T^.Left;
    end
    else begin
      Обработка для случая пустого дерева;
      if Top > 0 do begin
        {Когда вершина извлекается из стека,
         ее левое поддерево уже обработано}
```

```
        T := Stack[Top];
        Top := Top - 1;
        Посещение корня;
        T := T^.Right;
      end;
    end;
  until Top = 0;
end;
```

Нерекурсивное выполнение обхода «снизу вверх» несколько сложнее. Его реализация оставляется в качестве самостоятельного упражнения, а в случае затруднений можно обратиться к учебному пособию по данному курсу.

2. Порядок выполнения работы

Выполнение лабораторной работы заключается в разработке и отладке программы работы с бинарными деревьями согласно полученному варианту задания.

Для выполнения работы можно использовать любой язык программирования, содержащий необходимые средства. Продвинутый интерфейс программы допускается, но совершенно не обязателен.

Отчет о лабораторной работе оформляется на бумаге в печатном или рукописном виде. На титульном листе указывается название работы и состав бригады. В отчете приводится формулировка задания, а также комментированный текст разработанной программы.

Отчет и программа представляются также в электронном виде.

3. Пример программы

Рассмотрим следующую задачу. Даны два бинарных дерева. Будем считать деревья равными, если они имеют одинаковую структуру и соответствующие вершины содержат равные значения. Требуется проверить, равны ли заданные деревья.

Данная задача в рекурсивном варианте требует обхода «снизу вверх», поскольку в общем случае нельзя решить вопрос о равенстве деревьев, не ответив предварительно на тот же вопрос для каждого из поддеревьев. При этом нужно параллельно двигаться по обоим сравниваемым деревьям. С другой стороны, надо отметить, что в данной задаче обход может быть прерван досрочно, как только будет найдено хотя бы одно различие.

Нерекурсивный вариант сравнения запрограммирован как просмотр всех пар поддеревьев в поисках различий, связанных с их корнями. Таких различий может быть только два: одно из поддеревьев пусто, а парное ему – нет, либо оба поддерева непусты, но их корни имеют разные значения. Деревья будут признаны равными, если ни одного подобного различия не будет найдено.

В прилагаемом проекте **Laba2** содержится пример программы на Delphi, выполняющей поставленную задачу. Можно использовать текст примера как основу, особенно в части, касающейся ввода и вывода дерева.

4. Варианты заданий

В каждом варианте требуется написать программу, которая вводит дерево, заданное в текстовом представлении, отображает введенное дерево, строит сцепленное представление дерева, выполняет заданные действия с помощью как рекурсивной, так и нерекурсивной версии алгоритма и выводит полученные результаты.

1) Удалить все листья бинарного дерева, содержащие отрицательные значения (в том числе и те вершины, которые станут листьями после удаления их сыновей).

2) Бинарное дерево называется сортированным, если для любой вершины все ее левое поддерево содержит значения, меньшие или равные значению этой вершины, а правое поддерево – большие или равные. Проверить, является ли заданное дерево сортированным.
3) Дано бинарное дерево, содержащее числовые значения. Требуется заменить значение каждой внутренней вершины суммой значений всех ее потомков.
4) Бинарное дерево представляет арифметическое выражение: листья содержат числа или имена переменных, а внутренние вершины – знаки операций +, –, *, /. Требуется выдать это выражение в обычной форме записи, причем скобки следует расставлять только там, где это действительно необходимо (т.е. «(a + b) * c», но не «(a * b) + c»).
5) Требуется найти в бинарном дереве вершину, содержащую заданное значение данных. Если вершина найдена, следует выдать значения вершины, лежащей перед найденной вершиной в порядке обхода слева направо, и вершины, лежащей после найденной.
6) В заданном бинарном дереве требуется найти вершины с максимальным и минимальным значением и поменять эти вершины местами. Требуется поменять сами вершины, а не их значения.
7) Будем называть весом ветви сумму значений всех вершин этой ветви. Требуется подсчитать средний вес ветвей заданного бинарного дерева.
8) Значением каждой вершины дерева является буква. Требуется выдать все слова, образующиеся при «чтении» ветвей от корня к листу.