Напоследок мы рассмотрим структуру данных, которая называется *двоичное дерево* (binary tree). Несмотря на то, что бывает много различных типов деревьев, двоичные деревья играют особую роль, так как в отсортированном состоянии позволяют очень быстро выполнять вставку, удаление и поиск. Каждый элемент двоичного дерева состоит из информационной части и указателей на левый и правый элементы. На рис. 22.8 показано небольшое двоичное дерево.

|  |
| --- |
| корень  ↙  +-------+  |данные |  +---+---+  левое | | | правое  поддерево +---+---+ поддерево  ↘ ↙ ↘ ↙  +-------+ +-------+  |данные | |данные |  +---+---+ +---+---+  | | | | 0 | |  +---+---+ +---+---+  ↙ ↘ ↘  +-------+ +-------+ +-------+  |данные | |данные | |данные |  +---+---+ +---+---+ +---+---+  | 0 | 0 | | 0 | 0 | | 0 | 0 |  +---+---+ +---+---+ +---+---+  ↑ ↑ ↑  '----------------листья-----------------' |
| *Рис. 22.8. Пример двоичного дерева, высота которого равна 3* |

При обсуждении деревьев применяется специальная терминология. Программисты не являются специалистами в области филологии, и поэтому терминология, применяемая в теории графов (а ведь деревья представляют собой частный случай графов!), является классическим примером неправильного употребления слов. Первый элемент дерева называется *корнем* (root). Каждый элемент данных называется *вершиной дерева* (node), а любой фрагмент дерева называется *поддеревом* (subtree). Вершина, к которой не присоединены поддеревья, называется *заключительным узлом* (terminal node) или *листом* (leaf). *Высота* (height) дерева равняется максимальному количеству уровней от корня до листа. При работе с деревьями можно допустить, что в памяти они существуют в том же виде, что и на бумаге. Но помните, что дерево — всего лишь способ логической организации данных в памяти, а память линейна.

В некотором смысле двоичное дерево является особым видом связанного списка. Элементы можно вставлять, удалять и извлекать в любом порядке. Кроме того, операция извлечения не является разрушающей. Несмотря на то, что деревья легко представить в воображении, в теории программирования с ними связан ряд сложных задач. В данном разделе деревья затрагиваются лишь поверхностно.

Большинство функций, работающих с деревьями, рекурсивны, поскольку дерево по своей сути является рекурсивной структурой данных. Другими словами, каждое поддерево, в свою очередь, является деревом. Поэтому разрабатываемые здесь функции будут рекурсивными. Существуют и не рекурсивные версии этих функций, но их код понять намного сложнее.

Способ упорядочивания дерева зависит от того, как к нему впоследствии будет осуществляться доступ. Процесс поочередного доступа к каждой вершине дерева называется *обходом (вершин) дерева* (tree traversal). Рассмотрим следующее дерево:

d

↙ ↘

b f

↙ ↘ ↙ ↘

a c e g

Существует три порядка обхода дерева: *обход симметричным способом*, или *симметричный обход* (inorder), *обход в прямом порядке, прямой обход, упорядоченный обход, обход сверху*, или *обход в ширину* (preorder) и обход в *обратном порядке, обход в глубину, обратный обход, обход снизу* (postorder). При симметричном обходе обрабатывается сначала левое поддерево, затем корень, а затем правое поддерево. При прямом обходе обрабатывается сначала корень, затем левое поддерево, а потом правое. При обходе снизу сначала обрабатывается левое поддерево, затем правое и, наконец корень. Последовательность доступа при каждом методе обхода показана ниже:

Симметричный обход **a b c d e f g**

Прямой обход **d b a c f e g**

Обход снизу **a c b e g f d**

Несмотря на то, что дерево не должно быть обязательно упорядоченным, в большинстве задач используются именно такие деревья. Конечно, структура упорядоченного дерева зависит от способа его обхода. В оставшейся части данной главы предполагается симметричный обход. Поэтому упорядоченным двоичным деревом будет считаться такое дерево, в котором левое поддерево содержит вершины, меньшие или равные корню, а правое содержит вершины, большие корня.

Приведенная ниже функция stree() создает упорядоченное двоичное дерево:

struct tree {

char info;

struct tree \*left;

struct tree \*right;

};

struct tree \*stree(

struct tree \*root,

struct tree \*r,

char info)

{

if(!r) {

r = (struct tree \*) malloc(sizeof(struct tree));

if(!r) {

printf("Не хватает памяти\n");

exit(0);

}

r->left = NULL;

r->right = NULL;

r->info = info;

if(!root) return r; /\* первый вход \*/

if(info < root->info) root->left = r;

else root->right = r;

return r;

}

if(info < r->info)

stree(r,r->left,info);

else

stree(r,r->right,info);

return root;

}

Приведенный выше алгоритм просто следует по ссылкам дерева, переходя к левой или правой ветви очередной вершины на основании содержимого поля info до достижения места вставки нового элемента. Чтобы использовать эту функцию, необходимо иметь глобальную переменную-указатель на корень дерева. Этот указатель изначально должен иметь значение нуль (NULL). При первом вызове функция stree() возвращает указатель на корень дерева, который нужно присвоить глобальной переменной. При последующих вызовах функция продолжает возвращать указатель на корень. Допустим, что глобальная переменная, содержащая корень дерева, называется rt. Тогда функция stree() вызывается следующим образом:

/\* вызов функции street() \*/

rt = street(rt, rt, info);

Функция stree() использует рекурсивный алгоритм, как и большинство процедур работы с деревьями. Точно такая же функция, основанная на итеративных методах, была бы в несколько раз длиннее. Функцию stree() необходимо вызывать со следующими параметрами (слева направо): указатель на корень всего дерева, указатель на корень следующего поддерева, в котором осуществляется поиск, и сохраняемые данные. При первом вызове оба первых параметрах указывают на корень всего дерева. Для простоты в вершинах дерева хранятся одиночные символы. Тем не менее, вместо них можно использовать любой тип данных.

Чтобы обойти созданное функцией stree() дерево в симметричном порядке и распечатать поле info в каждой вершине, можно применить приведенную ниже функцию inorder():

void inorder(struct tree \*root)

{

if(!root) return;

inorder(root->left);

if(root->info) printf("%c ", root->info);

inorder(root->right);

}

Данная рекурсивная функция завершает работу тогда, когда находит заключительный узел (нулевой указатель).

В следующем листинге показаны функции, выполняющие обход дерева в ширину и в глубину.

void preorder(struct tree \*root)

{

if(!root) return;

if(root->info) printf("%c ", root->info);

preorder(root->left);

preorder(root->right);

}

void postorder(struct tree \*root)

{

if(!root) return;

postorder(root->left);

postorder(root->right);

if(root->info) printf("%c ", root->info);

}

Теперь давайте рассмотрим короткую, но интересную программу, которая строит упорядоченное двоичное дерево, а затем, обходя его симметричным образом, отображает его на экране боком. Для отображения дерева требуется лишь слегка модифицировать функцию inorder(). Поскольку на экране дерево распечатывается боком, для корректного отображения правое поддерево необходимо печатать прежде левого. (Технически это противоположность симметричного обхода.) Новая функция называется printtree(), а ее код показан ниже:

void print\_tree(struct tree \*r, int l)

{

int i;

if(r == NULL) return;

print\_tree(r->right, l+1);

for(i=0; iinfo);

print\_tree(r->left, l+1);

}

Далее следует текст всей программы печати дерева. Попробуйте вводить различные деревья, чтобы увидеть, как они строятся.

/\* Эта программа выводит на экран двоичное дерево. \*/

#include <stdlib.h>

#include <stdio.h>

struct tree {

char info;

struct tree \*left;

struct tree \*right;

};

struct tree \*root; /\* начальная вершина дерева \*/

struct tree \*stree(struct tree \*root,

struct tree \*r, char info);

void print\_tree(struct tree \*root, int l);

int main(void)

{

char s[80];

root = NULL; /\* инициализация корня дерева \*/

do {

printf("Введите букву: ");

gets(s);

root = stree(root, root, \*s);

} while(\*s);

print\_tree(root, 0);

return 0;

}

struct tree \*stree(

struct tree \*root,

struct tree \*r,

char info)

{

if(!r) {

r = (struct tree \*) malloc(sizeof(struct tree));

if(!r) {

printf("Не хватает памяти\n");

exit(0);

}

r->left = NULL;

r->right = NULL;

r->info = info;

if(!root) return r; /\* первый вход \*/

if(info < root->info) root->left = r;

else root->right = r;

return r;

}

if(info < r->info)

stree(r, r->left, info);

else

stree(r, r->right, info);

return root;

}

void print\_tree(struct tree \*r, int l)

{

int i;

if(!r) return;

print\_tree(r->right, l+1);

for(i=0; iinfo);

print\_tree(r->left, l+1);

}

По существу, данная программа сортирует вводимую информацию. Метод сортировки является одной из разновидностей сортировки методом вставок, которая была рассмотрена в предыдущей главе. В среднем случае производительность может быть вполне хорошей.

Если вы запускали программу печати дерева, вы, вероятно, заметили, что некоторые деревья являются *сбалансированными* (balanced), т.е. каждое поддерево имеет примерно такую же высоту, как и остальные, а некоторые деревья очень далеки от этого состояния. Например, дерево **a**⇒**b**⇒**c**⇒**d** выглядит следующим образом:

a

↘

b

↘

c

↘

d

В этом дереве нет левых поддеревьев. Такое дерево называется *вырожденным*, поскольку фактически оно выродилось в линейный список. В общем случае, если при построении дерева вводимые данные являются случайными, то получаемое дерево оказывается близким к сбалансированному. Если же информация предварительно отсортирована, создается вырожденное дерево. (Поэтому иногда при каждой вставке дерево корректируют так, чтобы оно было сбалансированным, но этот процесс довольно сложен и выходит за рамки данной главы.)

В двоичных деревьях легко реализовываются функции поиска. Приведенная ниже функция возвращает указатель на вершину дерева, в которой информация совпадает с ключом поиска, либо нуль (NULL), если такой вершины нет.

struct tree \*search\_tree(struct tree \*root, char key)

{

if(!root) return root; /\* пустое дерево \*/

while(root->info != key) {

if(keyinfo) root = root->left;

else root = root->right;

if(root == NULL) break;

}

return root;

}

К сожалению, удалить вершину дерева не так просто, как отыскать. Удаляемая вершина может быть либо корнем, либо левой, либо правой вершиной. Помимо того, к вершине могут быть присоединены поддеревья (количество присоединенных поддеревьев может равняться 0, 1 или 2). Процесс переустановки указателей подсказывает рекурсивный алгоритм, приведенный ниже:

struct tree \*dtree(struct tree \*root, char key)

{

struct tree \*p,\*p2;

if(!root) return root; /\* вершина не найдена \*/

if(root->info == key) { /\* удаление корня \*/

/\* это означает пустое дерево \*/

if(root->left == root->right){

free(root);

return NULL;

}

/\* или если одно из поддеревьев пустое \*/

else if(root->left == NULL) {

p = root->right;

free(root);

return p;

}

else if(root->right == NULL) {

p = root->left;

free(root);

return p;

}

/\* или есть оба поддерева \*/

else {

p2 = root->right;

p = root->right;

while(p->left) p = p->left;

p->left = root->left;

free(root);

return p2;

}

}

if(root->info < key) root->right = dtree(root->right, key);

else root->left = dtree(root->left, key);

return root;

}

Необходимо также следить за правильным обновлением указателя на корень дерева, описанного вне данной функции, поскольку удаляемая вершина может быть корнем. Лучше всего с этой целью указателю на корень присваивать значение, возвращаемое функцией dtree():

root = dtree(root, key);

Двоичные деревья — исключительно мощное, гибкое и эффективное средство. Поскольку при поиске в сбалансированном дереве выполняется в худшем случае log2n сравнений, оно намного лучше, чем связанный список, в котором возможен лишь последовательный поиск.