Программирование и основы алгоритмизации

Лекция 5

Рекурсивные структуры данных. Деревья

Рекурсивные структуры данных

- По аналогии с рекурсивным вызовом функции существуют структуры данных, допускающие рекурсивное определение: элемент структуры данных содержит один или несколько указателей на элементы такого же типа.
- Формально это соответствует тому факту, что в определении структурированного типа содержится указатель на себя самого

```
struct xxx
  { struct xxx *pp; ...
```

К ним относятся и линейные списки

- Рекурсивные структуры данных естественным образом обрабатываются рекурсивными функциями
- Рекурсивная функция, обрабатывающая структуру данных, получает в качестве параметра указатель на некоторый элемент. Если этот элемент содержит корректные (не равные NULL) указатели на другие элементы и если алгоритм требует их просмотра, то данная функция вызывается рекурсивно с параметром -указателем на новый элемент.

Рекурсивные функции для работы с линейными списками

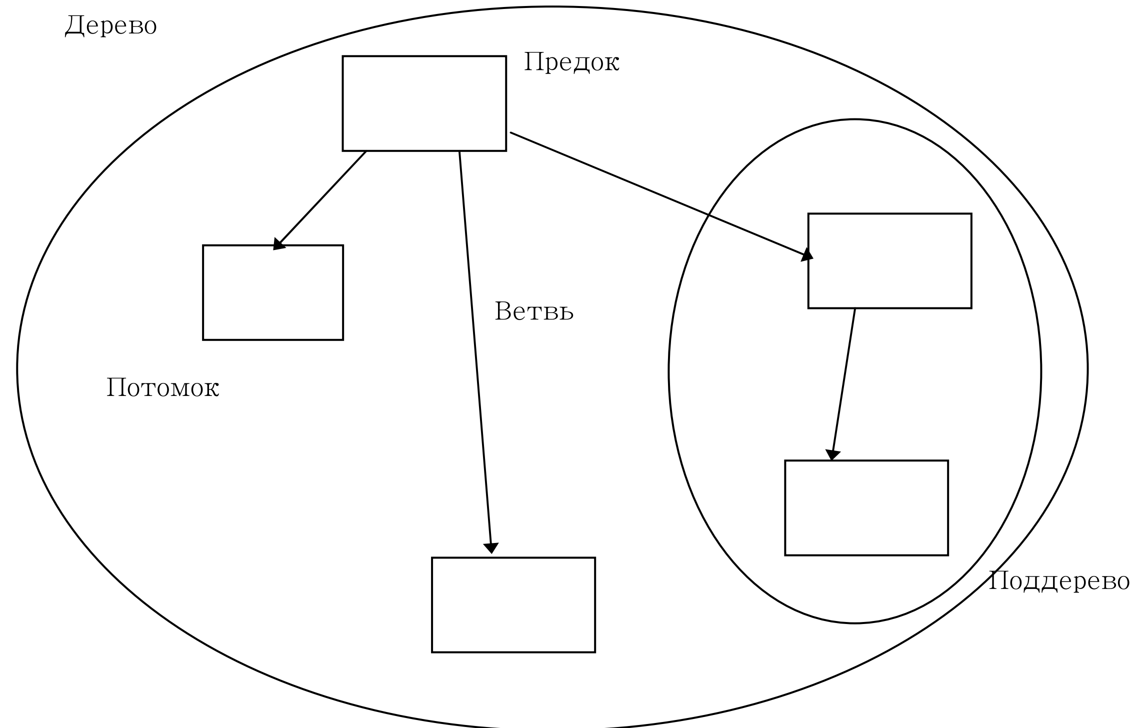
```
struct list
        {
        list *next;
        int         val;
        };
//----- Просмотр списка ----------------------------------
void scan(list *p)
{
if (p == NULL) return;          // указатель NULL - конец списка
cout << p->val << endl;
scan(p->next);                  // рекурсивный вызов для
        указателя
}                               // на следующий элемент
//----- Включение в конец списка -------------------------
// функция получает указатель на ячейку, где должен
// находиться указатель на текущий элемент списка
void insert(list **ph, int v)
{
if (*ph == NULL)                // включить новый элемент по
        адресу
        {                       // указателя со значением NULL
        list *pnew=new list;
        pnew->val=v;
        pnew->next=NULL;
        *ph=pnew;
        }
```

```
else   insert(& (*ph)->next, pnew);
}
//----- включение с сохранением порядка ---------
// функция возвращает возможно измененный
        указатель на
// оставшуюся часть списка
list *insord(list *ph, int v)
{
if (ph==NULL || ph->val > v)
        {
        list *pnew=new list;
        pnew->val=v;
        pnew->next=ph;
        return pnew;
        }
return insord(ph->next);
}
```

Дерево

- Определение дерева имеет исключительно рекурсивную природу
- Элемент этой структуры данных называется вершиной
- Дерево представляет собой либо отдельную вершину, либо вершину, имеющую ограниченное число указателей на другие деревья (ветвей)
- Нижележащие деревья для текущей вершины называются поддеревьями, а их вершины - потомками. По отношению к потомкам текущая вершина называется предком.

Дерево (2)



Дерево -- либо пустая вершина, либо вершина с множеством ветвей к аналогичным деревьям.

Дерево (3)

```
struct      tree
{
int    val;                    // Значение элемента
tree  *child[4];  // Указатели на потомков
};
```

```
//------Рекурсивный обход дерева
void   ScanTree(tree *p)
{
int     i;
if (p == NULL)     return; // следующей вершины нет
for (i=0; i<4; i++)           // рекурсивный обход
   ScanTree(p->child[i]);// потомков с передачей
}                                          // указателей на них
```

Нахождение максимальной глубины дерева

```
//------Максимальная длина ветвей дерева
int MaxDepth(tree *p)
{ int      i, max, nn;
if (p == NULL)   return 0;                          // следующей вершины нет
for (max = MaxDepth(p->child[0]), i=1; i<4; i++)
    {                                               // обход потомков
    nn = MaxDepth(p->child[i]);
    if (nn > max) max = nn;
    }                                               // возвращается глубина с
return max + 1;                                     // учетом текущей вершины
```

Включение вершины

```
//------Включение вершины в дерево на
       заданную глубину
int Insert(tree *ph, int v, int d)
// результат логический - вершина включена
// ph - указателя на текущую вершину
// d  - текущая глубина включения
{
if (d == 0)
{
for (int i=0; i<4; i++)
       if (ph->child[i] == NULL)
                  {

                  tree *pn=new tree;
                  ph->child[i]=pn;
                  pn->val=v;
                  for (i=0; i<4; i++) pn->child[i]=NULL;
                  return 1;
                  };
  return 0;  }
else
    if (Insert(ph->child[i], v , d-1)) return 1;
return 0;
}
```

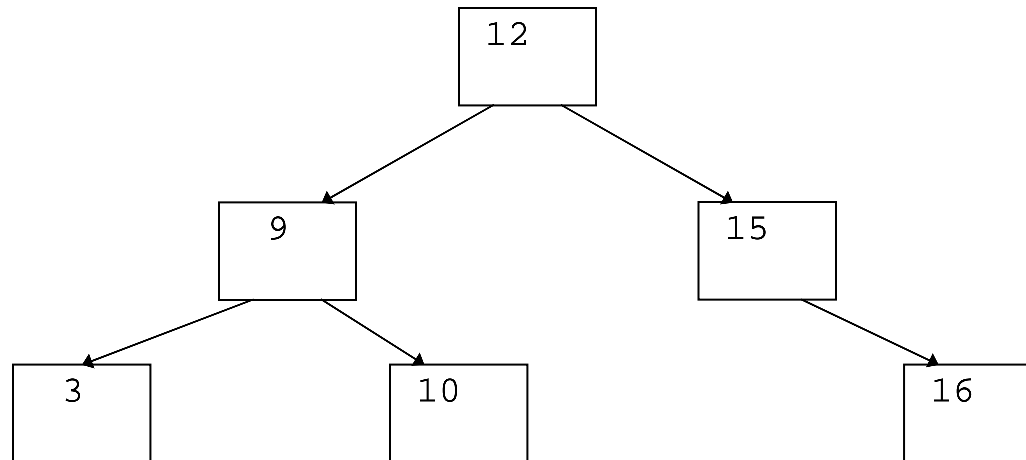
```
void main()
{
// пример вызова функции
tree PH={1,{NULL,NULL,NULL,NULL}};
Insert(&PH, 5, MaxDepth(&PH));
}
```

- Количество просматриваемых вершин от уровня к уровню растет в геометрической прогрессии
- Чтобы эффективно использовать деревья для поиска данных надо включать в вершины дерева данные таким образом, что наиболее часто используемые будут располагаться ближе к корню, и при этом анализ текущий вершин позволит сделать вывод о том, стоит ли “опускаться” в поддеревья
- Пример. Дерево, каждая вершина которого содержит строку, организовано так, что самая короткая строка в поддереве находится в корневой вершине. Тогда при поиске слова в дереве будет соблюдаться принцип - чем оно короче, тем меньше вершин будет просмотрено

Поиск слова в дереве

```
struct tree1
    {
    char *key;                  // Ключевое слово
    char *data;                 // Искомая информация
    tree1 *child[4];// Потомки
    };
char    *find(tree1 *ph, char *keystr)
{ char *s;
if (ph==NULL) return NULL;
if (strcmp(ph->key,keystr)==0) return ph->data;
                                    // Вершина найдена
if (strlen(keystr)<strlen(ph->key)) return NULL;
                                    // Короткие строки - ближе к корню
for (int i=0; i<4; i++)
    if ((s=find(ph->child[i],keystr))!=NULL) return s;
return NULL;
}
```

Двоичное дерево



Значения вершин левого поддерева всегда меньше, а значения вершин правого поддерева -больше значения в самой вершине.

```
struct  btree
   {
   int   val;
   btree *left,*right;
   };
```

Двоичное дерево (2)

- Указанное свойство позволяет применить в двоичном дереве алгоритм двоичного поиска. Действительно, каждое сравнение искомого значения и значения в вершине двоичного дерева позволяют выбрать для следующего шага правое или левое поддерево.
- Алгоритмы включения и исключения вершин дерева не должны нарушать указанное свойство: при включении вершины дерева поиск места ее размещения производится путем аналогичных сравнений

Поиск в двоичном дереве

```
//----- Рекурсивный поиск в двоичном дереве---------------
//   Возвращается указатель на найденную вершину
btree     *Search(btree *p, int v)
{
if (p==NULL) return(NULL);                    // Ветка пустая
if (p->val == v) return(p);           // Вершина найдена
if (p->val > v)                               // Сравнение с текущим
   return(Search(p->left,v));         // Левое поддерево
else
   return(Search(p->right,v));        // Правое поддерево
}
//----- Включение значения в двоичное дерево--------------
// функция возвращает указатель на созданную вершину,
// либо на существующее поддерево
```

Включение вершины в двоичное дерево

```
// функция возвращает указатель на созданную
     вершину,
// либо на существующее поддерево
btree *Insert(btree *pp, int v)
{
if (pp == NULL)        // Найдена свободная ветка
   {                   // Создать вершину дерева
   btree *q = new btree;           // и вернуть
     указатель
   q->val = v;
   q->left = q->right = NULL;
   return q;
   }
if (pp->val == v) return pp;
if (pp->val > v)                   // Перейти в левое
     или
     pp->left=Insert(pp->left,v); // правое поддерево
else
     pp->right=Insert(pp->right,v);
return pp;
}
```

```
void main()
{
// пример вызова
btree *ss=Search(ph,5);


ph=Insert(ph,6);
}
```

Сбалансированные деревья

- Поиск в двоичном дереве требует количества сравнений, не превышающего максимальной длины ветви дерева, или максимальной длины цепочки его вершин
- Следовательно, условием эффективности поиска в дереве является равенство длин его ветвей (сбалансированность)
- В хорошо сбалансированном дереве длины ветвей дерева отличаются не более, чем на **1**
- Линейный список – вырожденный случай дерева (с одной ветвью)

```
//  Рекурсивный обход двоичного дерева с
// выводом
//  значений вершин в порядке возрастания
void   Scan(btree *p)
{
if (p==NULL) return;
Scan(p->left);
cout << p->val << endl;
Scan(p->right);
}
```

Нумерация вершин двоичного дерева

```
//  Рекурсивный обход двоичного дерева с нумерацией вершин
//  снизу-вверх слева-направо, n - текущий номер вершины
int Scan(btree *p, int n)
{
if (p==NULL) return n;
Scan(p->left,n);
n++;
cout << n <<  p->val << endl;
n=Scan(p->right,n);
return n;
}
```