Лекция 24
Тема: Структуры данных. Деревья.

Определение графа

Неориентированный граф *G* — это упорядоченная пара ***G* = (*V*,*E*),** где
V – конечное множество **вершин** или **узлов;**
E - множество неупорядоченных пар различных вершин, называемых **рёбрами**.
Вершины ***u*** и ***v*** называются **концевыми** вершинами ребра ***e* = (*u*,*v*)**.
Два ребра называются **смежными**, если они имеют общую концевую вершину.
Если пары множества **E** упорядочены, то граф является **ориентированным**, а элементы множества **E** – **дугами**.

В неориентированном графе ***(v,w)=(w,v).***

Путь – это последовательность ребер вида ***(v1,v2), (v3,v4),…, (vn-1,vn).***

Говорят, что этот путь идет из вершины v_1 в вершину v_n и имеет длину **n-1**. Часто его обозначают как последовательность вершин $v_1,v_2,v_3,v_4,\ldots, v_{n-1},v_n$.

Тогда -

Путь - конечная последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей в последовательности вершин ребром.

Путь называется **простым**, если все ребра и все узлы на нем , кроме, может быть, первого и последнего, различны.

Определение дерева

Дерево — связный (ориентированный или неориентированный) граф, не содержащий циклов (для любой вершины есть один и только один способ добраться до любой другой вершины)

Корневое дерево

Корневое дерево определяется как конечное множество ***T*** одного или более узлов со следующими свойствами:
существует один корень дерева ***T*** ;
остальные узлы (за исключением корня) распределены среди непересекающихся множеств $T_1,\ldots,T_m$, и каждое из множеств является деревом; деревья $T_1,\ldots,T_m$ называются **поддеревьями** корня ***T***

Определения, связанные с деревьями

- **Степень узла** — количество поддеревьев узла.
- **Концевой узел** (лист) — узел со степенью нуль.
- **Уровень узла** определяется следующим образом:
  - уровень корня дерева *T* равен нулю;
  - уровень корней поддеревьев любой вершины дерева на единицу больше,

чем уровень этой вершины.

- **Высота дерева** – максимальное значение уровня какой-либо вершины дерева
- **Глубина узла v** в дереве – длина пути из корня в **v**.
- **Высота узла v** в дереве – это длина самого длинного пути из **v** в какой-нибудь лист.
- **Высота дерева** – это высота его корня.
- **Уровень узла** равен разности высоты дерева и глубины узла.

Упорядоченным называется дерево, в котором множество сыновей каждого узла упорядочено.

При описании деревьев могут быть использованы термины из ботаники («корень», «лист», «ветвь»), либо из генеалогии («отец», «сын», «предок», «потомок», «брат»).

- **Лес** — множество (обычно упорядочен-ное), не содержащее ни одного непересе-кающегося дерева или содержащее несколько непересекающихся деревьев.
- **Ориентированное дерево** — это ориентированный граф без циклов, в котором в каждую вершину, кроме одной, называемой *корнем ориентированного дерева*, входит одно ребро. В корень ориентированного дерева не входит ни одного ребра. Иногда, термин «ориентированное дерево» сокращают до «дерева».
- **Дерево** – это ориентированный граф без циклов, удовлетворяющий следующим условиям:
  Имеется в точности один узел, называемый корнем, в который не входит ни одно ребро;
  В каждый узел, кроме корня входит ровно одно ребро;
  Из корня к каждому узлу идет путь (единственный).

Обходы деревьев

- Обход дерева – операция, связанная с посещением его вершин (под посещением понимается любая операция над вершиной дерева, не затрагивающая структуру дерева)
- При обходе каждая вершина должна быть посещена ровно один раз

Прямой обход дерева

- посетить корень
- обойти все поддеревья в направлении слева направо

Прямой обход *(просмотр в глубину, просмотр сверху–вниз)* определяется следующим рекурсивным алгоритмом:

- **`посетить корень`**

- **обойти все поддеревья корня, начиная с самого левого**

Обратный обход дерева

- обойти все поддеревья в направлении слева направо
- посетить корень

Обратный обход *(просмотр снизу–вверх)* дерева определяется следующим рекурсивным алгоритмом:

- **обойти все поддеревья корня, начиная с самого левого**
- **посетить корень**

Обход дерева по уровням

- посетить корень
- посетить вершины 1-го, 2-го и т.д. уровней в направлении слева направо

Обход деревьев по уровням :

- **Вначале посещается корень дерева, затем – его сыновья, начиная с самого левого, затем – внуки. Так продолжается до тех пор, пока все вершины не посещены.**

Представление деревьев

Физическая реализация деревьев может быть выполнена различными способами.

1. Поскольку деревья являются частным случаем понятия графа, то для них можно применить любое представление, разработанное для графа общего вида (например, матрицу смежности или матрицу инцидентности).

2. Существуют представления, ориентированные на деревья. Наиболее простым и понятным из них является т.н. *каноническое* представление дерева, когда каждая вершина содержит ссылку на своего отца (корень, естественно, содержит пустую ссылку).

Такое представление удобно, когда надо представить не одно дерево, а *лес* из нескольких деревьев. Но операций, которые удобно выполнять с помощью такого представления, не слишком много. Так, можно легко определить предков каждой вершины, но не ее потомков. Кроме того, модификация дерева затруднена при этом представлении.

3. Следующее представление дерева заключается в том, что каждая его вершина содержит список ссылок на сыновей этой вершины. Теперь можно легко выполнить поиск всех потомков конкретной вершины (но не ее предков!).

Часто два последних представления комбинируют, получая нечто вроде двунаправленного списка. Хотя полученная структура и является несколько избыточной, она позволяет легко проходить дерево в обоих направлениях: от листьев к корням и обратно.

Рассмотрим конкретную реализацию этого представления на примере бинарных деревьев.

Двоичные деревья

Двоичное (бинарное) дерево (ориентированное) — это ориентированное дерево, в котором исходящие степени вершин (число исходящих рёбер) не превосходят 2.

На двоичном дереве основаны следующие структуры данных:

- двоичное дерево поиска
- двоичная куча
- красно-чёрное дерево
- АВЛ-дерево
- Фибоначчиева куча

Двоичное дерево поиска

Двоичное дерево поиска (бинарное поисковое дерево, БПД) – структура данных, предназначенная для выполнения следующих операций:

- поиска элемента по значению
- добавления нового элемента
- удаления элемента из дерева

Структура БПД

- Сыновья каждой вершины различаются: выделяются левый и правый сын
- Значения хранятся в каждой вершине дерева
- Для любого узла значения в левом поддереве меньше, а значения в правом поддереве – больше значения, хранящегося в корне

Бинарное поисковое дерево

Концевой обход бинарного дерева *(симметричный или внутренний)*

- Обойти левое поддерево
- Посетить корень
- Обойти правое поддерево

Механизм поиска в БПД

Если дерево пусто, то поиск завершился неудачно. В противном случае сравниваем искомое значение со значением корня. Если эти значения равны, то искомое значение найдено; иначе рекурсивно выполняем эту же процедуру для левого или правого поддерева, в зависимости оттого, что больше – значение корня или искомое значение.

Поиск в БПД требует в лучшем случае O(log **n**) операций сравнения, где **n** – число элементов.

Однако, если дерево не сбалансировано (т.е. длина одной из ветвей существенно больше длины другой), то эффективность поиска снижается и в худшем случае может достичь O(**n**) операций, когда дерево вырождается в единственную ветвь.

Поиск значения в БПД

Пусть K – искомое значение, T – значение в корне дерева

```
if (K==T) поиск завершен успешно
else if (K<T)
  if (есть левый сын)
    выполнить поиск в левом поддереве
  else
    поиск завершен неудачно
else
  if (есть правый сын)
    выполнить поиск в правом поддереве
  else
    поиск завершен неудачно
```

Примеры поиска в БПД

Примеры поиска в БПД

Вставка новой вершины в БПД

- если БПД пусто, создаем единственную вершину и делаем ее корнем;
- иначе выполняем поиск вершины с добавляемым значением;
- если поиск успешен, вставка невозможна;
- в противном случае добавляем новую вершину и делаем ее сыном (левым или правым) той вершины, на которой остановились в процессе поиска

Пример создания БПД последовательностью вставок

Добавляемые элементы: 7, 19, 14, 3, 5, 8, 22, 9, 1, 20, 2

Проблемы балансировки

Лекция 25
Тема: Структуры данных. Деревья. (продолжение)

Упражнение: *Что делает следующая программа?*
(см. конспект)

Физическая реализация БПД

Наиболее удобной реализацией БПД представляется задание каждой его вершины в динамической памяти. В статическую область помещается лишь указатель на корень дерева:

```
struct TreeItem{
   int Info;
   TreeItem* Father;
   TreeItem* LSon;
   TreeItem* RSon;
   TreeItem(){LSon=RSon=NULL;}
   };

TreeItem* Root = NULL;
// пустое дерево

bool find(TreeItem* R,int a, TreeItem* &t)
// поиск элемента
// при удачном поиске функция возвращает true,
// а параметр t содержит указатель на
// найденную вершину;
// при неудачном поиске функция возвращает
// false,  а параметр t содержит указатель на
// вершину, на которой  завершился поиск;
// при попытке поиска в пустом дереве
// параметр t содержит NULL

{  if (R == NULL)
   {
     t = NULL;
     return false;
   }
   t = R;
```

```
   for (;;)
   {
     if (t->Info == a)
         return true;
     if (t->Info > a)
     {
         if (t->LSon == NULL)
             return false;
         t = t->LSon;
     }
     else
     {
         if (t->RSon == NULL)
             return false;
         t = t->RSon;
     }
   }
}

bool Insert(TreeItem* &R,int info)
//вставка элемента по значению
//в бинарное поисковое дерево;
//функция возвращает false
//при обнаружении дубликата
{  TreeItem *s,*q;
   if (find(R,info,s))
     return false;
   q = new TreeItem;
   q->Info = info;
   if (s == NULL)
   { R = q;
     q->Father = NULL;
   }
   else
   { q->Father = s;
     if (s->Info < info)
         s->RSon = q;
     else
         s->LSon = q;
   }
   return true;
}
```

```
void print(TreeItem* &s)
{  cout << s->Info << " ";}
//**********************************
void post_order_walk(TreeItem* s,
                     void vizit(TreeItem* &s))
   // обратный обход дерева
   // выполняя действия над всеми элементами
  // дерева в функции vizit
{
   if (s != NULL)
   {
     post_order_walk(s->LSon,vizit);
     post_order_walk(s->RSon,vizit);
     vizit(s);
   }
}

void EraseItem(TreeItem * &s)
// удаление листа дерева
{
   if (s->Father != NULL)
   { if ((s->Father)->LSon == s)
          (s->Father)->LSon = NULL;
     else
          (s->Father)->RSon = NULL;
   }
 else Root=NULL; //если лист-корень
 delete s;
 s = NULL;
}

void post_order(TreeItem* R, void vizit(TreeItem* &s))
// вспомогательная функция
{
   if (R == NULL) // проверка на пустоту
   {
     cout << "дерево пустое" << endl;
     return;
   }
   post_order_walk(R, vizit);
}
```

Для уничтожения дерева применяется обратный обход, концевой обход применить нельзя, т.к. посещение вершины приведет к ее удалению, и понятие «правое поддерево» станет неопределенным.

```
void Destroy(TreeItem * &R)
// уничтожение дерева
{
     post_order(R, EraseItem);
     R=NULL;
}

int main()
{
   setlocale(LC_ALL,".1251");
   // распечатаем пустое дерево
   post_order(Root,print);
   int x;
   TreeItem* s;
 // добавим 8 элементов
   for (int i=1; i<9; i++)
   {
     cout<<"введите вершину "<< i << " : ";
     cin >> x;
     Insert(Root,x);
   }

 post_order(Root,print);
 // распечатаем дерево
   cout << endl;
   // удалить лист
   cout <<"какой удалить лист? ";
   cin >> x;
   if (find(Root,x,s))
     EraseItem(s); // удаление листа дерева
   else
     cout<<"такого листа нет"<<endl;

 post_order(Root,print);
 // распечатаем дерево
   cout << endl;
 // удалить лист
```

```
      cout << "какой удалить лист? ";
      cin >> x;
      if (find(Root,x,s))
        EraseItem(s); // удаление листа дерева
      else
        cout<<"такого листа нет"<<endl;
      post_order(Root,print);
     // распечатаем дерево
      cout << endl;
      Destroy(Root);// удаление дерева
      post_order(Root,print);
     // распечатаем пустое дерево
      return 0;
   }

void EraseNode(TreeItem* &s)
// удаление вершины, имеющей не более одного сына
{  TreeItem *q;
   if (s->LSon != NULL)
       q=s->LSon;
   else
       q=s->RSon;
   if (q!=NULL)
       q->Father=s->Father;
   if (s->Father == NULL)
       Root=q;
   else
       if ((s->Father)->LSon == s)
           (s->Father)->LSon = q;
       else
           (s->Father)->RSon = q;
   delete s;    s=NULL;
}

    bool Erase(TreeItem* &R,int info)
   // удаление вершины по значению
   {
      TreeItem *s,*q;
      if (!find(R,info,s))
        return false;
      else
```

```
   {
     if ((s->LSon != NULL)&&(s->RSon != NULL))
     {
          q=s->RSon;
          while (q->LSon!=NULL)
               q=q->LSon;
          s->Info=q->Info;
          EraseNode(q);
     }
     else
          EraseNode(s);
     return true;
   }
}

int main()
{
   setlocale(LC_ALL,".1251");
   // распечатаем пустое дерево обратным обходом
   post_order(Root,print);
   int n,x;
   TreeItem* s;
   cout << "сколько вершин?" << endl;
   cin >> n;
   // добавим n элементов
   for (int i=1; i<=n; i++)
   {
     cout << "введите вершину " << i << " : ";
     cin >> x;
     Insert(Root,x);
   }

  post_order(Root,print);// распечатаем дерево
   cout << endl;
   cout << "какую удалить вершину? ";
   cin >> x;
   if (Erase(Root,x)); // удаление вершины дерева
   else
     cout << "такой вершины нет"  << endl;
   post_order(Root,print);// распечатаем дерево
   cout << endl;
   cout << "какую удалить вершину? ";
```

```
   cin >> x;
   if (Erase(Root,x)); // удаление вершины дерева
   else
     cout << "такой вершины нет"  << endl;
   post_order(Root,print);//распечатаем дерево
   cout << endl;
   Destroy(Root);// удаление дерева
   post_order(Root,print);// распечатаем дерево
   return 0;
}
```

Выполнить программу для следующего теста:

- 8 вершин: 8 5 10 6 3 12 9 11
- Нарисовать построенное дерево.
- Сделать обратный обход.
- Удалить: 8
- Нарисовать полученное дерево.
- Сделать обратный обход.
- Удалить: 5
- Нарисовать полученное дерево.
- Сделать обратный обход.

Конец лекции

STOP