C++ 输入输出与数据结构使用

1.C++ 输入输出

（1）标准输入输出流

```
#include <iostream>
using namespace std;
```

该文件定义了 cin、cout、cerr 和 clog 对象，分别对应于标准输入流、标准输出流、非缓冲标准错误流和缓冲标准错误流。

标准输入流 cin：

```
int T, n;
string s;

cin >> T;
cin >> n >> s; //输入时可以用空格/回车间隔
cout << T << n << s << endl;

//因而不能接受空格输入
//法1：使用cin.getline(str, maxnum)
char s2[30];
cin.getline(s2, 30);
cout << s2 << endl;

//法2：使用string里的getline
string s3;
getline(cin, s3);
cout << s3 << endl;

//输入vector
vector<int> nums;
while(cin >> num){
    nums.push_back(num);
    if(cin.get() == '\n')
        break;
}

//已知size
int n;
cin >> n;
for(int i = 0; i < n; i++){
  cin >> nums[i];
}
```

标准输出流 cout：

```
//基本类型：整型 int i 浮点型 float f 字符型 char c 布尔型 bool b 可以直接输出
int i = 1;
float f = 0.5;
char c = 'a';
bool b = true; //输出时为1
cout << i << f << c << b;

//string 类型字符串直接输出
string s = "abc";
cout << s << endl;

//字符数组也可以直接输出
char str[4] = "abc"; //以\0结束
cout << str << endl;

// 一维，二维数组输出
int arr[2] = {1,2};
for(auto & a:arr)
  cout << a << ' ';

int mat[2][2] = {{1,2}, {3,4}};
for(auto & row:mat){
  for (auto & a:row)
      cout << a << ' ';
  cout << endl;
}

//集合，字典输出（迭代对象：string,vector等等）
set <int>::iterator i;
for(i=s.begin();i!=s.end();i++)
{
  cout << *i << ' ';
}

map<int, int>::iterator i;
for(i = m.begin(); i!=m.end(); i++)
  cout << i->first << ' ' << i->second << endl;
return 0;
```

(2)文件输入输出流

```
#include <fstream>
//创建流对象
//1）ofstream ： 写文件 （2）ifstream ： 读文件 （3）fstream ： 读写文件
ifstream fin；
ofstream fout;

//文件打开
fin.open ("文件路径" ,打开方式)
//打开方式包括(并 ||)：
ios::in //读文件
ios::out //写文件
ios::app //追加写

//验证文件是否打开
if(!fin.is_open())
{
    std::cerr<<"cannot open the file"
}

//读入方式1：按元素读（空格或回车间隔）
char buf[1024]={0}; //就是临时申请一个 1024大的读空间（又叫buffer），并且初始化为0。
while (fin >> buf)
    {
        cout << buf << endl;//每一次的buf是空格或回车键（即白色字符）分开的元素
    }

//读入方式2：使用getline按行读（若还没读够n个字符，如果遇到换行符'\n'则读取终止）
char buf[1024]={0};
while(fin.getline(buf, sizeof(buf)))
  std::cout<<buf<<endl;

//文件关闭
fin.close()

//文件写入
string data = "1234";
fout.open("./sha.txt", ios::app);  //显式指定ios::app才不会清空原文件
fout << data << endl;
fout.close();
```

2.字符串

标准库类型string表示可变长序列，需要包含string头文件；

（1）定义及初始化

```
string s1; //默认空串
string s2 = s1;
string s2(s1);

string s3 = "abc";  //拷贝初始化
string s4("abc");   //直接初始化

string s6 = string(10, 'c');
string s5(10, 'c'); //直接初始化

//操作
getline(is, s); //从is中读取一行赋给s，返回os
s.empty()  //s为空返回true，否则返回false
s.size() // 字符个数
s[1] //第2个字符
s1 + s2  //字符拼接
s1 == s2 //字符串相同
s1 != s2 //字符串不同

//字符处理
isalnum(c) //c为字母或数字时为true
isalpha(c) //c为字母为true
isdigit(c) //c为数字时为true

issapce(c) //c为空格时为true

tolower(c) //转换为小写
toupper(c) //转换为大写

// 字符串参数
void f(string & str);

// 返回字符串
string f();

//常用字符串函数
//插入
s.push_back('p');
s.insert(s.begin(), 'i');

//遍历
string::iterator it = s.begin();
for(it = s.begin(); it!=s.end();it++)

//查找
s.find('t')
```

```
//取子串
s.substr(start, n); //从start开始取n个

//删除
s.erase(s.begin()+3) // 删除位置3上的元素

//替换
s.replace(pos, len, str) //用str替换从位置pos开始的len个位置上的元素

//string转 int
int num = stoi(str1);

//int 转 string
string str1 = to_string(num);
```

3.数组

数组存储一个固定大小的由相同类型元素构成的顺序集合。在内存中的分配也是连续的，数组中的元素通过数组下标进行访问，数组下标从0开始。最低地址对应于第一个元素，最高地址对应于最后一个元素。

声明数组

```
//静态数组：声明定义时，内存大小就确定
const int size = 5;
int arr[size] = {1,2,3,4,5}; //[]内必须是常量表达式
arr[2] = 6;

//初始化
int arr2[] = {1,2,3};

//动态数组：使用new动态分配内存
int * foo;
foo = new int[5]; //foo指向一个有5个int类型的元素的有效内存块
//可以使用foo[i]或*(foo+i)来访问元素
```

重要区别是：静态数组的大小必须是一个常量表达式，new执行的动态内存分配则可以在运行时使用任何变量值作为大小分配内存。foo是指向动态数据的指针，因此可以删除此数据，并可以将新数据分配给该指针。

```
int * foo = new int[5];
delete[] foo; //释放
foo = new int[10];

int * start = (int *)malloc(n * sizeof(int));
free(start);
int * end = (int *)malloc(n * sizeof(int));
free(end);
```

二维动态数组：

```
int** a = new int*[rowCount];
for (int i=0; i<rowCount; i++)
    a[i] = new int[colCount];
```

静态数组的大小是在编译期间就确定，并且分配的，其内存在使用结束后由计算机自动释放，效率高；动态数组是在程序运行时，由程序员根据实际需要从堆内存中动态申请的，使用结束后由程序员进行释放，效率低。

数组的成员函数：

```
.begin(), .end()
.size(),
.front(), .back() //第一、最后一个元素引用
fill() //用值填充数组
.swap() //交换两个数组的值


//随机数生成
#include <cstdlib>
#include <time.h>

srand((unsigned)time(NULL));  //设置随机数种子

//随机生成一个[0,RAND_MAX]范围内的随机数（RAND_MAX=32767）
double a = rand();

//随机整数
(rand() % (b-a))+a; // 产生[a,b)的随机数
(rand() % (b-a+1))+a; // 产生[a,b]的随机数
(rand() % (b-a))+a+1; // 产生(a,b]的随机数

rand()/double(RAND_MAX); //产生[0,1]的浮点数
```

数组与函数

```
// 传递数组给函数
//方式1：传递一个指向数组的指针
void f(int * arr);

//方式2：形参是一个已定义大小的数组
void f(int arr[10]);

//方式3：形参是一个未定义大小的数组
void f(int arr[]);

//函数返回数组（C++不允许返回一个完整的数组(可以返回数组指针)
//C++ 不支持在函数外返回局部变量的地址（使用static变量，使用new动态分配内存，在函数外定义数组）
int * f(){
  static int arr[10];
  int * arr = new int[10]; //记得delete[]
  return arr;
}

//返回二维数组
//法1：
int ** f(){
  int ** p = new int * [10];  //指针数组
  int n = 0;
  for (int i = 0; i < 10; i++)
        {
                p[i] = new int[10];   //每个指针动态分配空间
                for(int j = 0; j < 10; j++)
                        p[i][j] = n++;
        }

  return p;
}

//法2：
int (*f())[10]   //解引用的结果是一个数组，包含十个元素；
{
        int (*p)[10] = new int[10][10]; //数组指针
        int n = 0;
        for (int i = 0; i < 10; i++)
                for (int j = 0; j < 10; j++)
                  p[i][j] = n++;

        return p;
}
```

4. vector使用

标准库类型vector表示类型相同的对象的集合，大小可扩展。使用需包含：

```
#include <vector>
using namespace std;
```

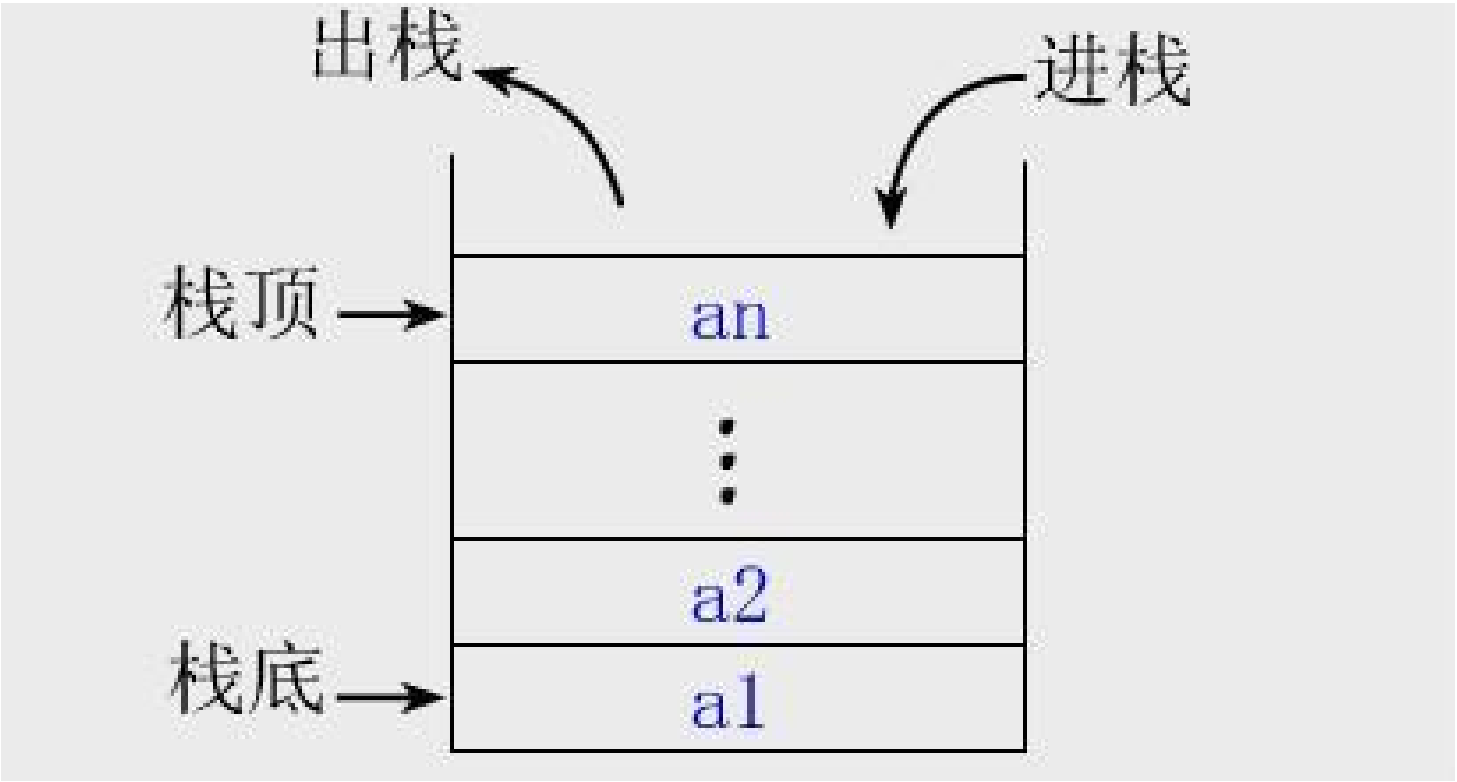
定义及初始化：

```
typedef int T;
vector<T> v1;  //空vector
vector<T> v2(v1); //v1的副本
vector<T> v2 = v1;
vector<T> v3(n ,val); //n个重复的val
vector<T> v4(n); //n个重复初始值

vector<T> v5{a, b, c,...};
vector<T> v6 = {a, b, c,...}; //列表初始化

//操作
v.empty(); //空为真
v.size(); //元素个数
v.push_back(t); //向v中添加一个值为t的元素
v.pop_back(); //删除最后一个元素
v[n];
v.resize(n); //调整容器的长度大小，使其能容纳n个元素
v.resize(n,t);

vec.insert(vec.begin()+i,a); //在第i个元素后面插入a;

vec.erase(vec.begin()+2); //删除第3个元素

vec.erase(vec.begin()+i,vec.end()+j); //删除区间[i,j-1];区间从0开始
vec.clear(); //清空

reverse(vec.begin(),vec.end()); // 将元素翻转
sort(vec.begin(),vec.end());  //默认是按升序
//若要降序
bool Comp(const int &a,const int &b)
{
    return a>b;
}
sort(vec.begin(),vec.end(),Comp);


//迭代器
vector<T>::iterator it; //迭代器类型
for(it=vec.begin();it!=vec.end();it++)
    cout<<*it<<endl;

*it; // 取值
v.begin();
v.end();
v++;

//函数
```

```
void f(vector<T> & vec);
vector<T> f();

//二维vector
vector<vector<T>> vec2; //定义
vector<vector<T>> vec2(row, vector<T>(col));  //默认初始化
```

4.栈

只允许在一端进行插入或删除操作的线性表，LIFO 后进先出。
栈顶（Top）：线性表允许进行插入和删除的一端。
栈底（Bottom）：固定的，不允许进行插入和删除的另一端。



使用STL中的栈：

```
//头文件
#include <stack>

//定义
stack<int> s;

//栈的操作
s.empty() //判断栈空

s.size()  //栈中元素个数

s.top()  //返回栈顶元素

s.pop()  //删除栈顶元素

s.push(x) //在栈顶压入新元素x
```

3.队列

队列（queue）是一种先进先出的，必须从队尾插入新元素，队列中的元素只能从队首出。

```
//定义
queue<int> q;

//基本操作
q.push(x)  //将x元素从队列末端入队

q.pop() //将队头第一个元素删除

q.front() //队首元素

q.back() //队尾元素

q.size()  //队中元素个数
```

4.链表

STL中的List，双向链表，可高效地进行插入删除元素。

```
#include <list>

list<int> c;

c.begin();  c.end();
c.erase(it1, it2);

c.front(); //第一个元素的引用
c.back(); // 最后一个元素的引用

c.push_front() //增加一个新的元素在 list 的前端。

c.pop_front() //删除 list 的第一个元素。

c.push_back() //增加一个新的元素在 list 的尾端。

c.pop_back() //删除 list 的最末个元素。
```

5.树

以分支关系定义的层次结构。
树的节点结构：节点元素、节点第一个孩子、节点的兄弟节点。

```
struct Treenode{
  Object element;
  TreeNode * firstChild;
  TreeNode * nextSibling;
};
```

二叉树结点结构：结点元素、左孩子、右孩子。（二叉树每个节点至多只有两颗子树，且有左右之分，顺序不颠倒。）

```
struct TreeNode{
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode():val(0), left(nullptr), right(nullptr){}
};
```

树的遍历：

- 前序遍历：先根节点、再左孩子，最后右孩子；
- 中序遍历：先左孩子、再根节点、最后右孩子；
- 后续遍历：先左孩子、再右孩子、最后根节点；

6.堆

heap用数组实现的一颗完全二叉树。在最大堆中，父结点的值比每一个子结点的值都要大。在最小堆中，父结点的值比每一个子结点的值都要小。区别于二叉搜索树，二叉树左孩子必须比父结点小且右孩子必须必比父结点大（中序遍历得到有序序列）。

堆（优先队列）定义：

```
//priority_queue
#include <queue>

//定义：priority_queue<Type, Container, Functional>

priority_queue <int,vector<int>,greater<int> > q; //最小堆
priority_queue <int,vector<int>,less<int> >q; //最大堆

//对于基础类型 默认是大顶堆
priority_queue<int> a; //相当于priority_queue<int, vector<int>, less<int>> a;
```

```
a.push(i);  // push 插入元素到队尾 (并排序)
a.pop();   // pop 弹出队头元素
a.top();  // 返回堆顶元素

vector<int> nums = {2, 1, 5, 3, 4};
priority_queue<int, vector<int>, greater<int>> a; //最小堆
vector<int> res;
for(auto &n:nums){
    a.push(n);
}

for(int i = 0; i < nums.size(); i++){
    res.push_back(a.top());
    a.pop();
}
```

STL heap还包括：

```
#include <algorithm>

vector<int> a = { 3, 6, 2, 1, 7, 4, 9, 5 };

//make_heap建堆：
make_heap(a.begin(), a.end());  //默认大顶堆 less<int>()
make_heap(a.begin(), a.end(), greater<int>());

// 添加元素，新元素插入在底层的vector的end()处
a.push_back(20);
push_heap(a.begin(), a.end()); //push_heap做出调整

// 删除堆顶元素
pop_heap(a.begin(), a.end());
a.pop_back();
```

插入堆并自动调整堆：

```
//先插入元素
v1.push_back(200);
v2.push_back(200);

//调用push_heap
push_heap(v1.begin(), v1.end());
push_heap(v2.begin(), v2.end(), greater<int>());

//堆排序
sort_heap(a.begin(), a.end());
```

7.集合

集合(Set)是一种包含已排序对象的关联容器，不允许有重复元素。（支持插入、删除，但不能修改）

```
set<int> num;

num.empty();       //判断num是否为空

num.begin();       //返回set容器的第一个元素的地址

num.end();        //返回set容器的最后一个元素地址


//添加元素a
num.insert(a);

//遍历（不提供下表操作符）
set<int>::iterator it;
for(it = num.begin(); it!=num.end(); it++)
  cout <<*it<<endl;

//查询：
iter = num.find(1);
if (iter != num.end())
  cout << *iter << endl

//统计出现次数
num.count(1)  //返回1存在，0不存在

//删除元素
iter = num.find(2);
num.erase(iter);

num.clear();     //删除num中的所有的元素
```

8.哈希映射

map是STL的一个关联容器，它提供一对一的hash。

第一个称为关键字(key)，每个关键字只能在map中出现一次；
第二个称为该关键字的值(value)；

STL的map底层是用红黑树实现的，查找时间复杂度是log(n)；
STL的hash_map底层是用hash表存储的，查询时间复杂度是O(1)；

hash表是以空间换时间的，因而hash_map的内存消耗肯定要大，一般情况下，如果记录非常大，考虑hash_map，查找效率会高很多，如果要考虑内存消耗，则要谨慎使用hash_map。

map提供一个很常用的功能，那就是提供key-value的存储和查询功能。如需要记录一个人名和相应的存储，而且随时增加，要快速查找和修改。

hash_map基于hash table（哈希表）。哈希表最大的优点是：把数据存储和查询消耗的时间大大降低，几乎可以看成是常数时间；而代价仅仅是消耗比较多的内存。然后在当前可利用内存越来越多的情况下，用空间换时间的做法是值得的。

```
//构造
map<int, string> maps;

//插入元素
maps.insert(pair<int, string>(000, "s_0"));

maps.insert(map<int, string>::value_type(001, "s_1"));

mapStudent[123] = "s_123";

//遍历
map<int, string>::iterator it;
for(it = maps.begin(); it!=maps.end(); it++){
  cout<< it->first <<':'<<it->second << endl;
}

//查找元素：find 返回迭代器指向当前查找元素的位置否则返回map::end()位置
iter = maps.find(123);
if(iter != maps.end())
  cout<<iter->second<<endl;

//统计元素出现个数
maps.count(123)  //若有为1，若无为0

//删除元素
iter = maps.find(123);
maps.erase(iter); //迭代器删除

int n = maps.erase(123); //返回成功删除元素的个数

//清空
maps.erase(maps.begin(), maps.end());
maps.clear();
```

C++STL中常用的容器和类型，支持下标[]运算的有：
vector、deque、map、unorder_map、string。（支持随机访问）

不支持下标[]运算的有：list, set, unordered_set, stack

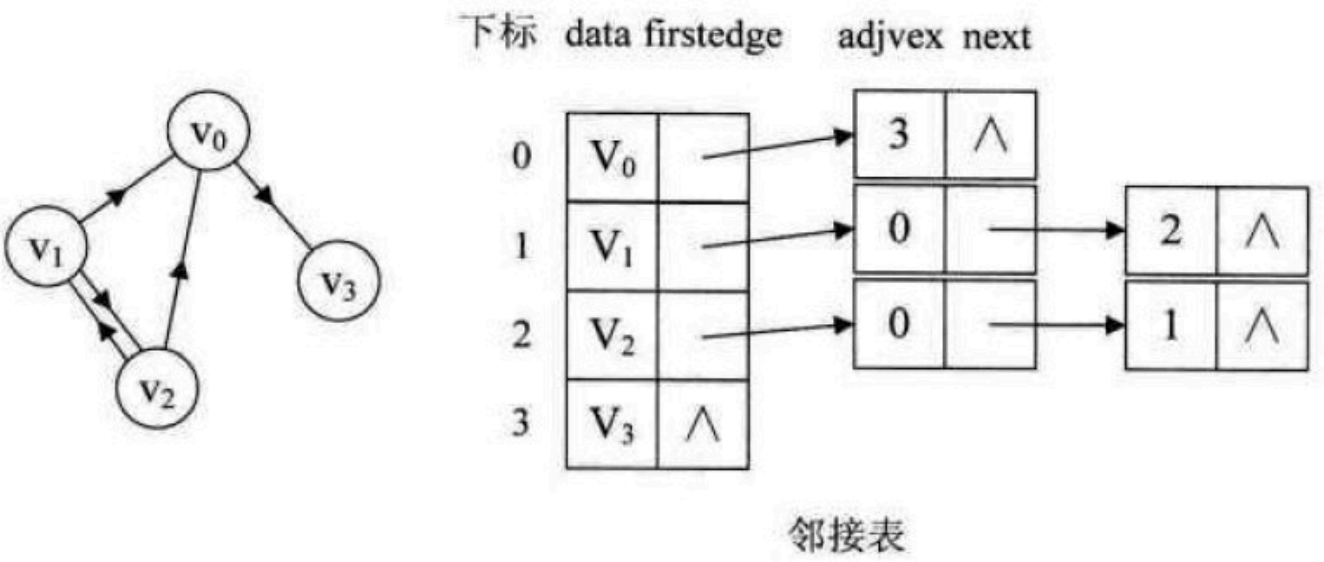
8.图

图（Graph）由有限个顶点和顶点之间的边的集合组成，表示为$G(V,E)$。
若任意两个顶点之间的边都是无向边，则为无向图（Undirected graphs）。
若任意两个顶点之间的边都是有向边，则是有向图（Directed graphs）。
有些图的边或弧具有与它相关的数值为权（Weight）。

图的存储：
要存储各个顶点本身的信息，顶点与顶点之间的关系。存储结构有邻接矩阵、邻接表、十字链表、邻接多重表等。

1. 邻接矩阵：用一个一维数组vertex存储图中顶点的信息，二维数组arc存储各顶点之间的邻接关系。
   顶点i的度：第i行非零元素个数。
   i与j之间存在边：$arc[i][j]=1$
   i的所有邻接点：arc第i行所有为1的位置
   有向图：求出度行求和，入度列求和
2. 邻接表（适用边数相对顶点较少的图）：
   顶点表节点：vertex + firstedge
   边表节点：adjvex + next

邻接表容易求每个顶点的出度，逆邻接表容易求入度。



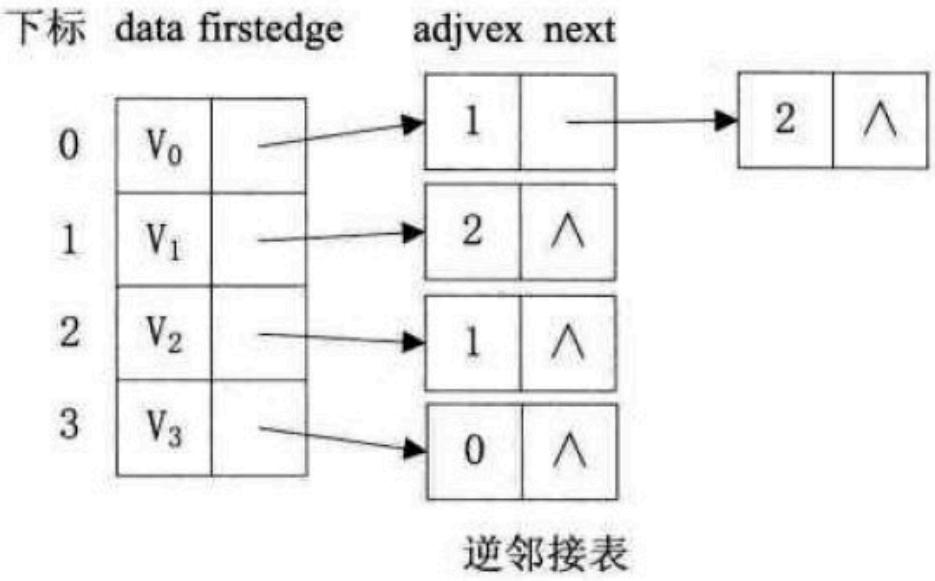


图 7-4-7

图的遍历：从指定一个顶点出发，开始访问其他顶点（每个顶点访问一次），不能重复访问，直至所有点被访问。

1. 深度优先搜索（DFS）
   从指定的第一个点开始，沿着向下的顶点不重复的遍历下去，直到尽头，退回至上一个顶点，接着不重复遍历。
2. 广度优先搜索（BFS）
   从指定的第一个点开始，先寻找跟它连接的所有顶点，再继续这些顶点继续深入。（逐层搜索）

BFS 通过队列实现，访问前先入队，访问后出队，邻接节点入队。

DFS 通过栈实现，递归。

最小生成树：

1. 普利姆（Prim）算法
2. 克鲁斯卡尔（Kruskal）算法

递归、回溯、DFS：

递归：函数调用本身解决问题，一种算法结构。

回溯：通过不同的尝试生成问题的解，类似穷举，但会剪枝。

DFS：回溯是DFS的一种，回溯再求解过程中不保留完整的树结构，而DFS记录完整的树。用标志 is_trav 记录访问过的状态；

递归的一般结构：

```
void f()
{
     if(符合边界条件)
     {
        ///////
        return;
     }

     //某种形式的调用
     f();
}
```

牛客网 & 赛码网 输入输出格式

1）单行输入：

```
cin >> m >> n;  //单行以空格隔开
```

2）多测试用例输入：

```
//不定长输入
int num;
while(cin >> num){
  cout << num <<endl;
  if(cin.get() == '\n')
    break;
}
```

3）使用getline读一行

```
//法2：使用string里的getline
string s3;
getline(cin, s3);
cout << s3 << endl;
```

python

```python
#单个输入
input = int(input())

#单行多个输入
inputs = list(map(int, input().split(' ')))  # 以空格间隔
```