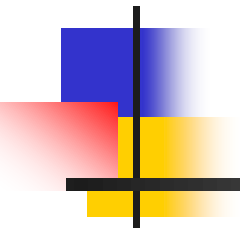
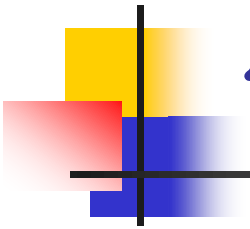


简单数据结构

数据结构（Data Structure）

- 程序 = 数据结构 + 算法
  算法 + 数据结构
- 什么是数据结构
  - 数据元素之间存在的一种或多种特定关系，便是“数据结构”，例如数据元素的顺序和位置关系；
  - 数据结构是计算机存储、组织数据的方式；
  - 精心选择的数据结构可以带来更高的运行效率或者存储效率；
  - 数据结构往往同高效的检索算法和索引技术有关。

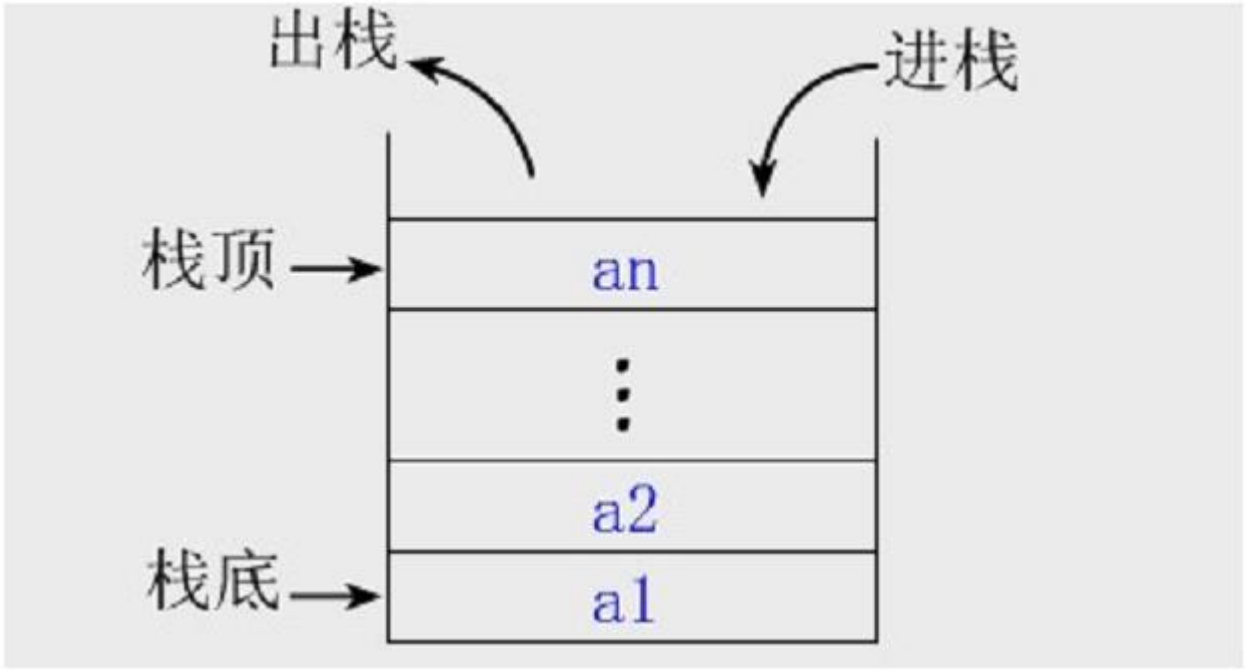


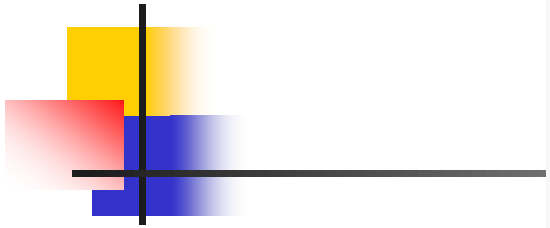
常见的数据结构

- 线性结构：数组、栈、队列、链表
- 树形结构：树、二叉树，以及各种变型
- 图形结构：各种图
- 集合：**set、multiset**

栈（stack）

- 一种具有**FILO**特性的线性表；
- **FILO**：**First In Last Out**（先进后出）；
- 只能在一端进行操作（插入/删除、进栈/出栈）





栈的实现

```
struct Stack {
    int a[maxn];
    int front = 0;

    void push(int num) {
        if(front + 1 < maxn) a[front++] = num;
    }

    void pop() {
        if(front > 0) front--;
    }

    int top() {
        if(front > 0) return a[front - 1];
    }

    int size() {
        return front;
    }
};
```

- 栈的应用
  - 括号匹配
    判断表达式中的括号是否匹配

    (())、 ()() 、()((、 ())(

  - 表达式求值

    1 + (3 * (7 +5) / (4 + 7 * (3 + 5))) * 2

- 表达式求值

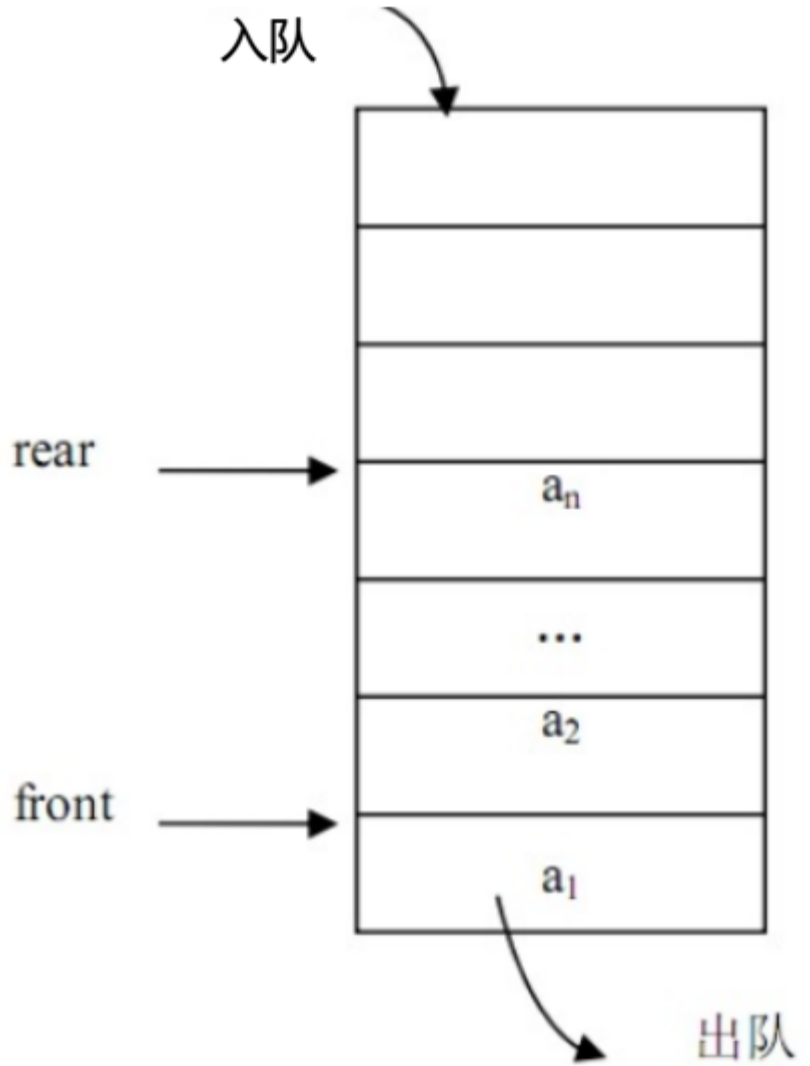
#1 + (3 * (7 +5) / (4 + 7 * (3 + 5))) * 2#

θ_1 \ θ_2	+	–	*	/	(	)	#
+	>	>	<	<	<	>	>
–	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	E
)	>	>	>	>	E	>	>
#	<	<	<	<	<	E	=

- 建立两个栈，一个存放运算符的O栈；一个存放操作数或运算结果的D栈；
- 从左向右扫描表达式：
  - 遇到数字，压入D栈；
  - 遇到运算符θj，则与O栈的栈顶运算符θi比较：
    - 若 O 栈为空或者 θi < θj， 则将 θj 压入 O 栈；
    - 若 θi = θj，则弹O栈（消括号或结束符）；
    - 若 θi > θj， 则取出 θi（弹O栈），并从D中取出两个数字（弹2次D栈），进行计算，将计算结果压入D栈；
- 最后O为空，D中只有一个结果数字；
  否则表达式有误。

队列（Queue）

- 具有**FIFO**特性的线性表
- **FIFO：First In First Out**
  （先进后出）
- 在一端插入（入队），
  在另一端删除（出队）

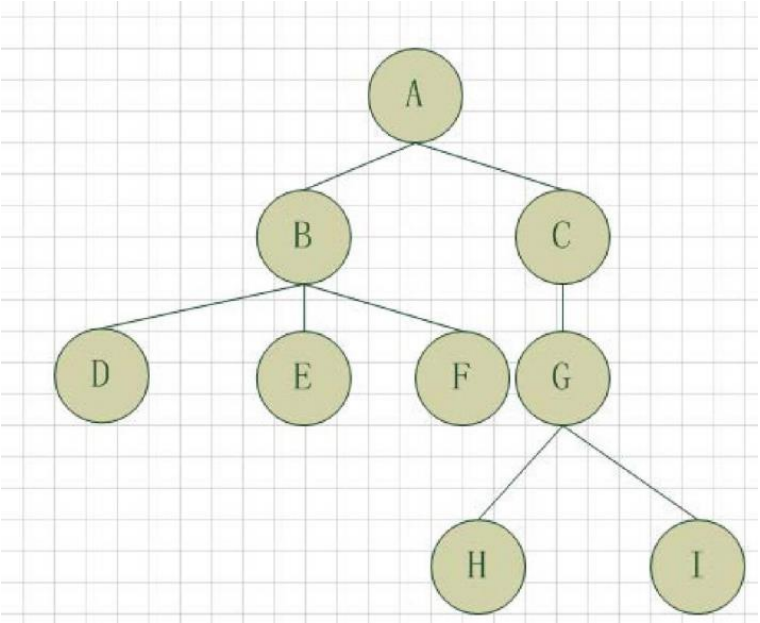


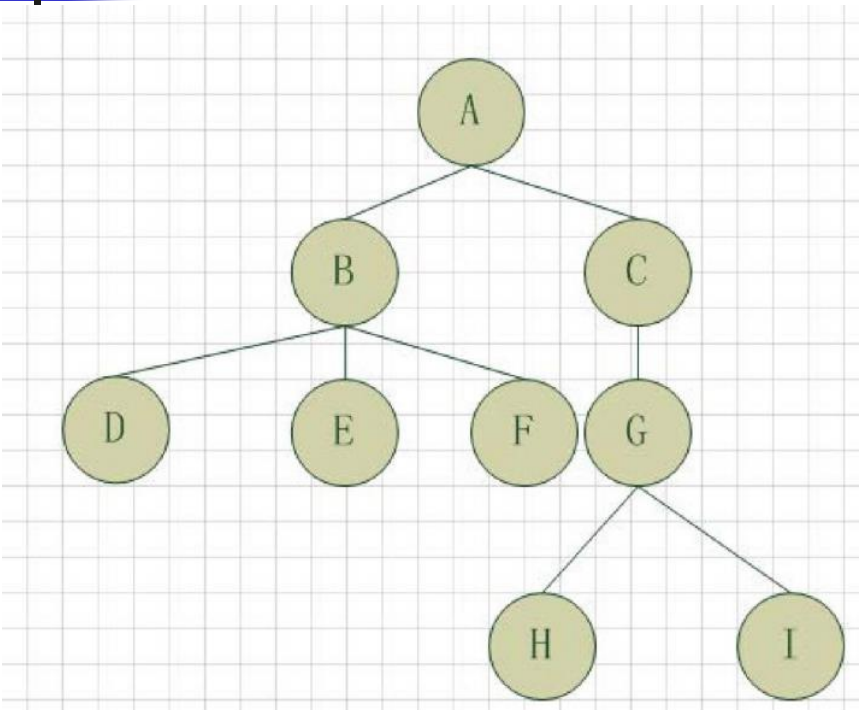
队列的实现

```
struct Queue
{
    int a[maxn];
    int front = 0, rear = 0;

    void push(int num)
    {
        if (rear + 1 < maxn)
            a[rear++] = num;
    }

    void pop()
    {
        if (front < rear)
            front++;
    }
    int top()
    {
        if (front < rear)
            return a[front];
    }

    int size()
    {
        return rear - front;
    }
};
```

树（Tree）

- 包含n（n>0）个结点的有穷集，其中：
  - 每个元素称为结点（node）；
  - 有一个特定的结点被称为根结点或树根（root）；
  - 除根结点外，其余数据元素被分为m（m≥0）个互不相交的集合T1， T2， ……Tm-1，其中每一个集合Ti（1<=i<=m）本身也是一棵树，被称作原树的子树（subtree）；
- 空集也是一棵树；





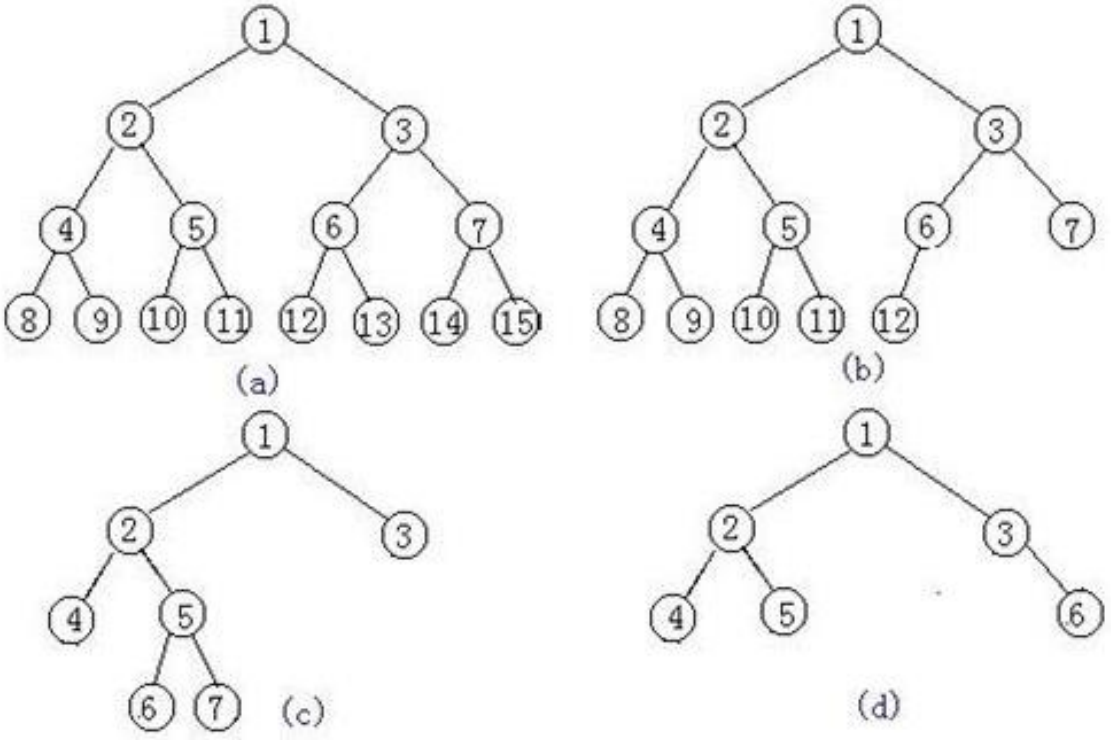
- 基本概念
  - 结点
  - 根结点
  - 父结点
  - 子结点
  - 深度
  - 宽度

基本性质：

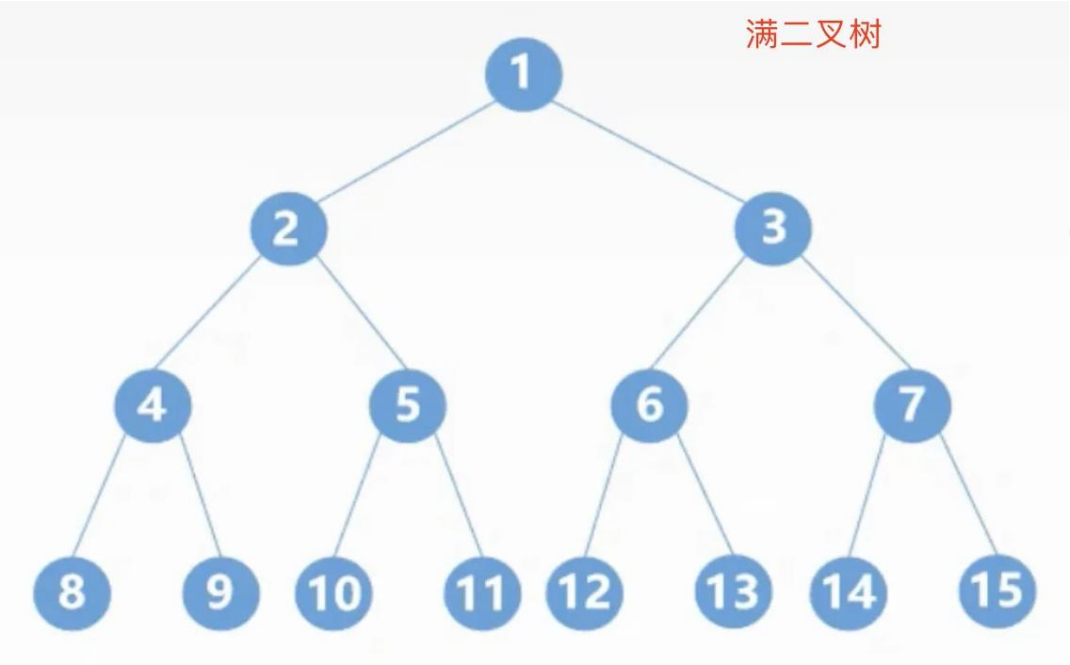
- 边数 = 顶点数 − 1；
- 任意两点存在唯一路径；
- 树是连通的并且任意一条边均为桥（去掉后不连通）；
- 在树中任意不同两点上加一条边会得到一个圈。

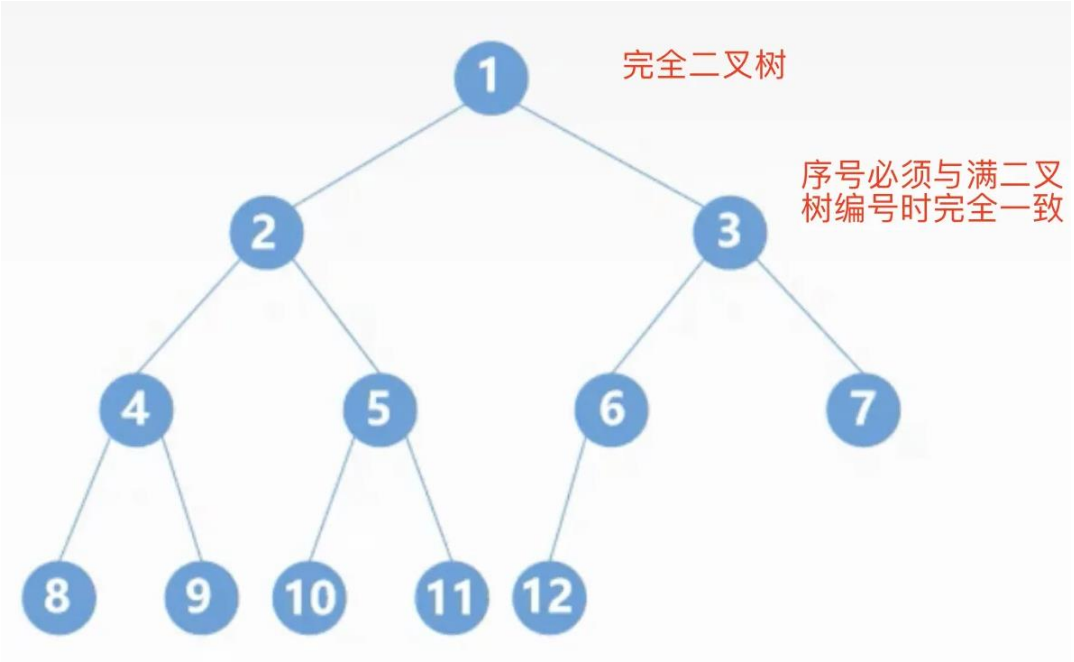
二叉树（Binary Tree）

- 每个节点最多有两个子树的树结构。
  - 左子树
  - 右子树

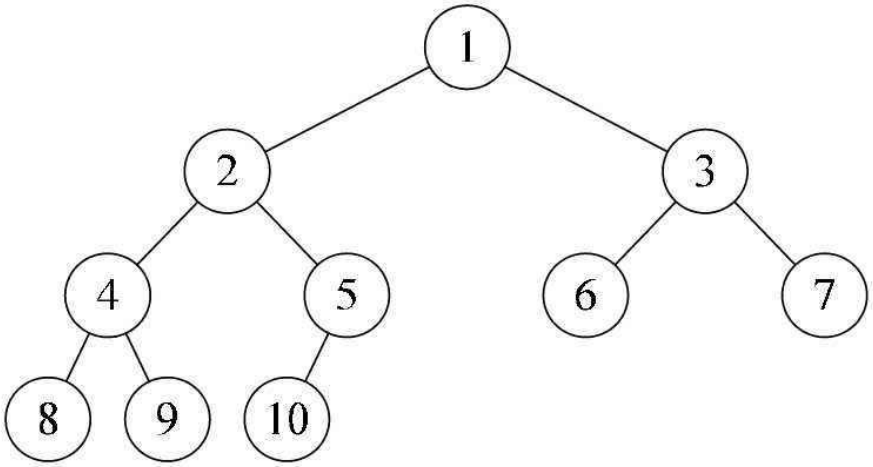


- 满二叉树
  一棵深度为**k**，且有**2^(k-1)**个节点的二叉树，称之为满二叉树；
- 完全二叉树
  深度为**k**，有**n**个节点的二叉树，当且仅当其每一个节点都与深度为**k**的满二叉树中序号为**1** 至**n**的节点对应时，称之为完全二叉树。

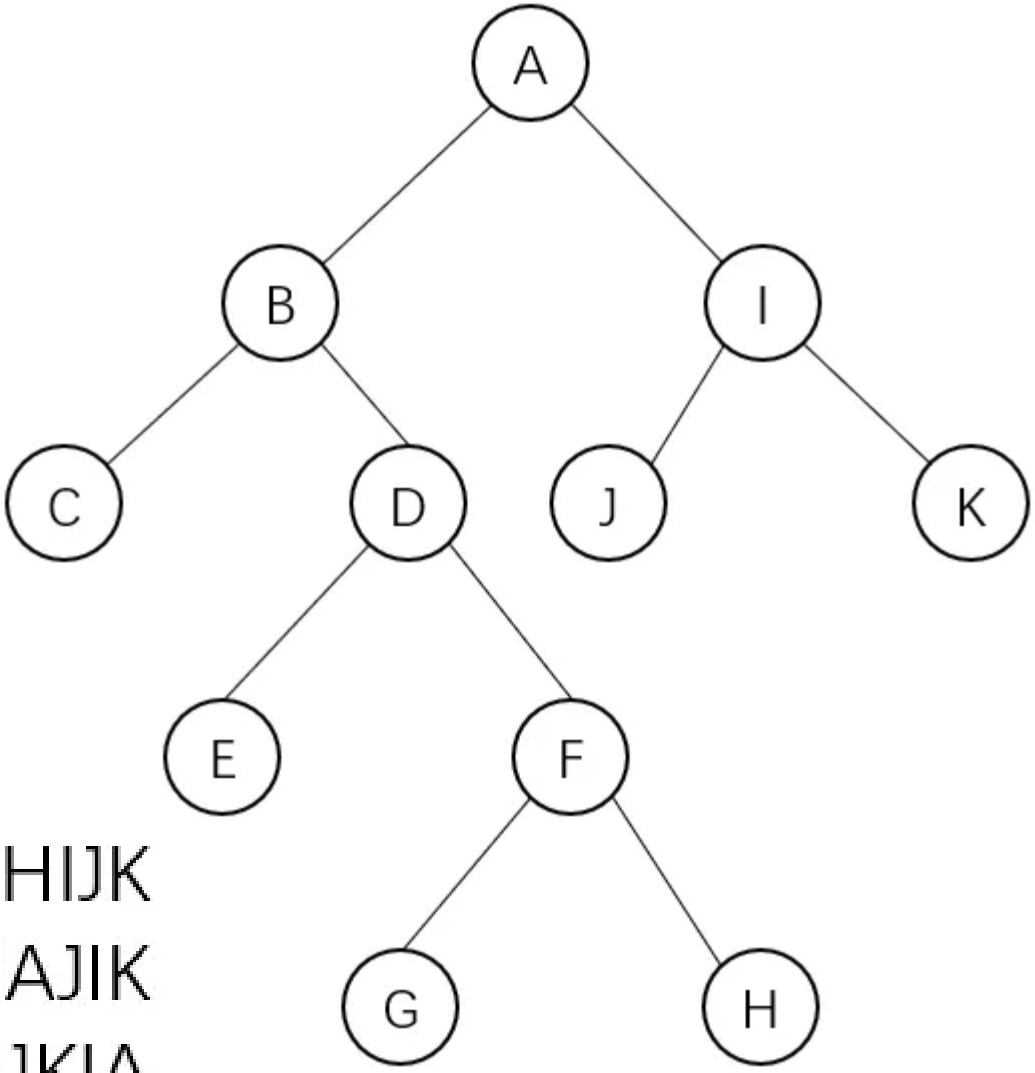

满二叉树
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


完全二叉树
序号必须与满二叉树编号时完全一致
1
2
3
4
5
6
7
8
9
10
11
12

- 完全二叉树的特点
  - 除了最后一层外，结点没有缺失；
  - 可以用数组表示（存储）
    - 对于下标为 **i** 的结点，
      左子树结点的下标为 **2*i**，
      右子树结点的下标为 **2*i+1** ，
      父结点的下标为 **i/2**（向下取整）



- 二叉树的遍历



先序遍历为： ABCDEFGHIJK

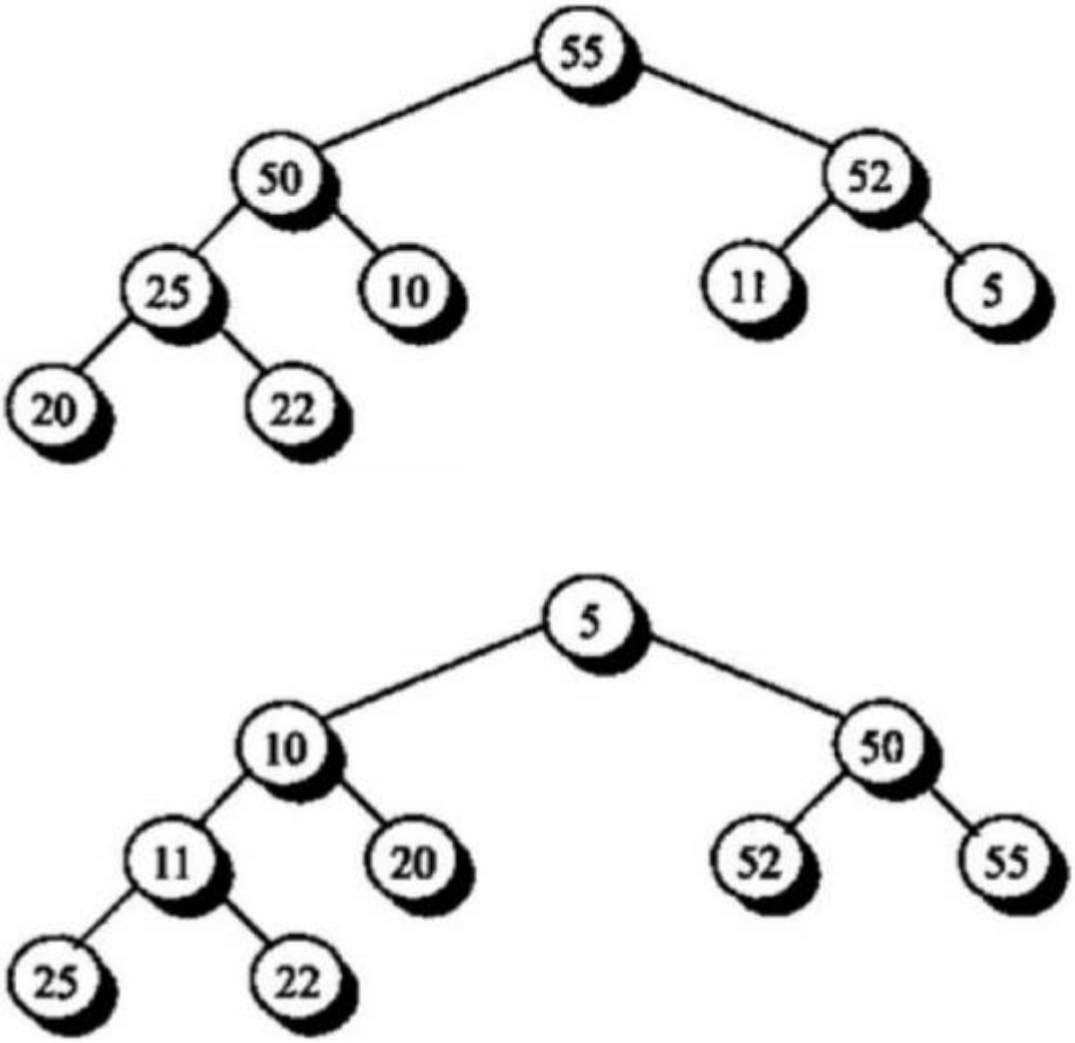
中序遍历为： CBEDGFHAJIK

后序遍历为： CEGHFDBJKIA

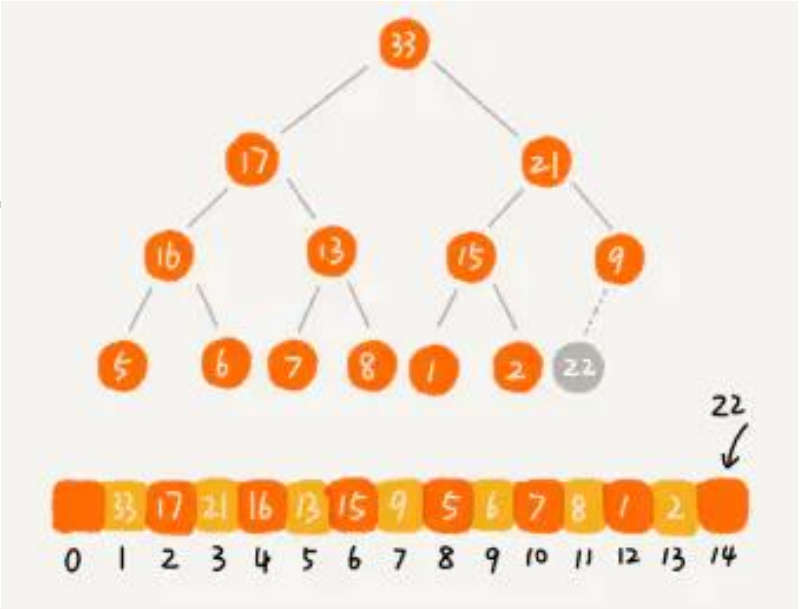
```
void DataPrint(struct Tree *root)
{
    if (root!=NULL)
    {
        printf("%d",root->data);
        DataPrint(root->LeftChild);
        DataPrint(root->RightChild);
    }
}
```

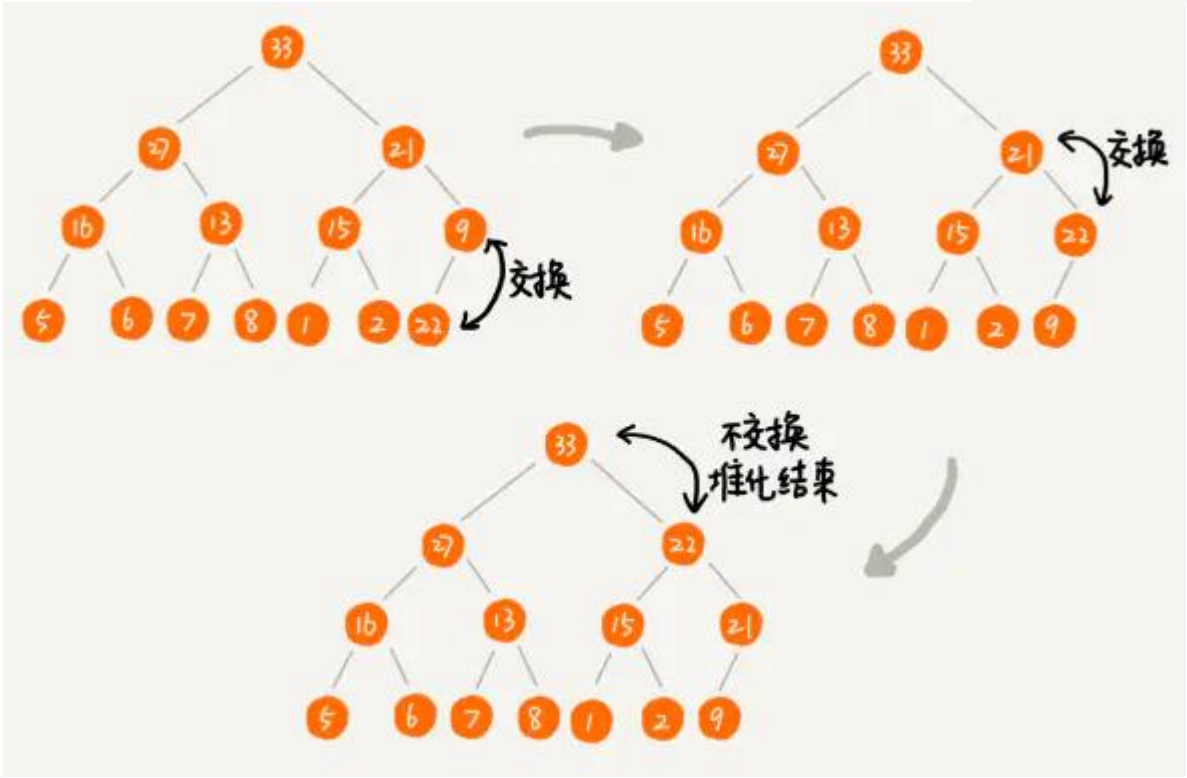
堆（Heap）

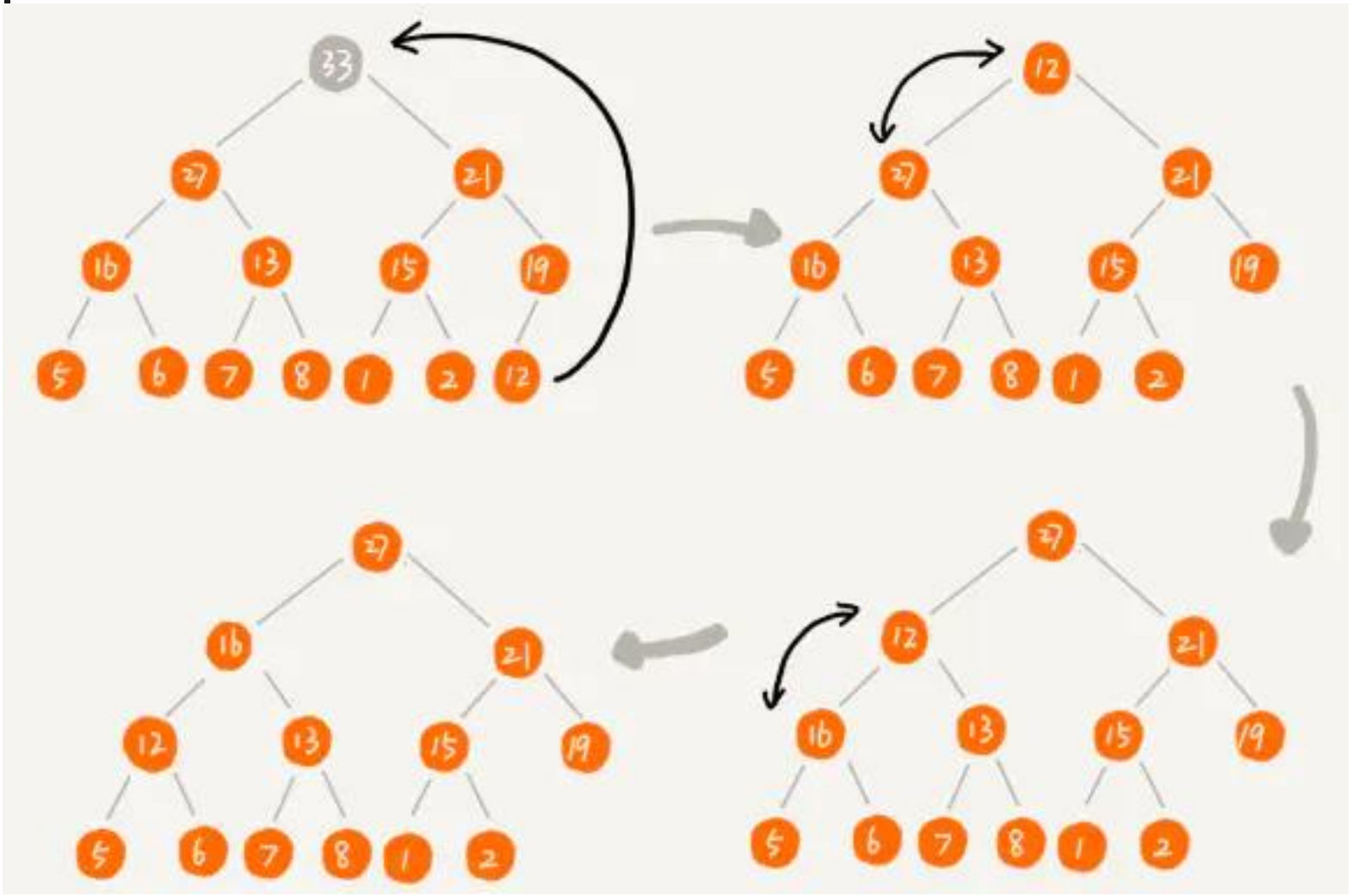
- 堆是一个完全二叉树；
- 堆中的每个结点的值都必须大于等于（或小于等于）其子树中每个结点的值，这种情况称为大根堆（小根堆）；

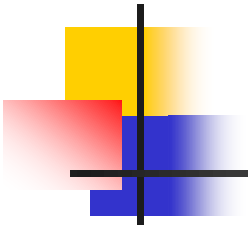


- 堆的操作
  - 插入、删除
  - 堆化

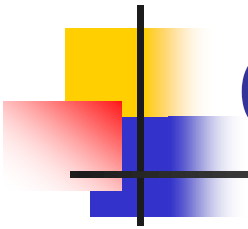

33
17
21
16
13
15
9
5
6
7
8
1
2
22
22
33 17 21 16 13 15 9 5 6 7 8 1 2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14


交换
交换
交换
堆化结束


33
27
21
16
13
15
19
5
6
7
8
1
2
12
12
27
21
16
13
15
19
5
6
7
8
1
2
27
27
12
21
16
13
15
19
5
6
7
8
1
2
27
16
21
12
13
15
19
5
6
7
8
1
2



- 堆的应用——堆排序
  - 时间复杂度：**O(nlogn)**
  - 空间复杂度：**O(1)**
  - 不稳定的排序算法，性能没有快排好；
- 堆的应用——**Top-k**问题
  - 从**n**个元素中找到**Top-k**的元素，并且针对不断添加进来的数据，都要获取最新的**Top-k**元素，怎么处理？



C++ STL的容器

- 堆栈容器（**stack**）
- 队列容器（**queue**）
- 优先队列容器（堆）
- 数组容器（**vector**）
- ……

堆栈容器（stack）

- **#include <stack>**
  using namespace std;

```
stack<int> sta;
sta.push(1);
int topElement = sta.top();
sta.pop();
sta.empty();
sta.size();
```

队列容器（queue）

- **#include <queue>**
  using namespace std;

```
queue<int> que;
que.push(1);
int frontElement = que.front();
que.pop();
que.empty();
que.size();
```

优先队列容器（堆）

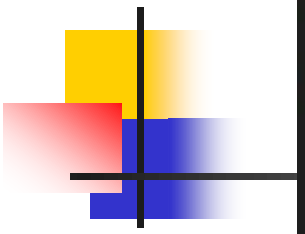
- **#include <priority_queue>**
  using namespace std;

```
priority_queue<int> que2;
que2.push(1);
int minElement = que2.top();
que2.pop();
que2.empty();
que.size();
```

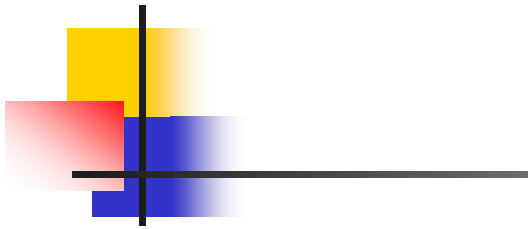
```
17  //大根堆
18  priority_queue<int> G;
19  //小根堆
20  priority_queue<int,vector<int>,greater<int>> L;
21  int x;
22  while (cin>>x){
23      G.push(x);
24      L.push(x);
25  }
26  while (!G.empty()){
27      cout<<G.top()<<endl;
28      G.pop();
29  }
30  while (!L.empty()){
31      cout<<L.top()<<endl;
32      L.pop();
33  }
```

数组容器（vector）

- **#include <vector>**
  using namespace std;
- **vector**可以被看成一个“超级数组”
- 既可以和**C**语言的数组一样用下标访问，也可以像链表一样动态改变长度。

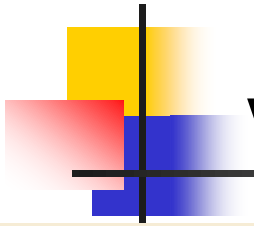


```cpp
vector<int> arr(100);
vector<int> list;

for (int i = 0; i < 100; i++)
{
    scanf("%d", &arr[i]);
    cout << arr[i] << endl;
}
for (int i = 0; i < 100; i++)
{
    int a;
    cin >> a;
    list.push_back(a);
    printf("%d\n", list[i]);
}
```



- **Vector常见操作**

```
list.size(); // 数组元素个数 O(1)
list.clear(); // 一键清空数组 O(n)
list.empty(); // 数组是否为空 O(1)
list.begin(); // 数组的首元素迭代器 O(1)
list.end(); // 数组最后一个元素的下一个元素的迭代器 O(1)
            // 该元素实际在数组中不存在
list.erase(p1); // 删除数组某个迭代器所在位置的数字 O(n)
list.push_back(1); // 往数组后面添加元素 O(1);
list.pop_back(); // 删除数组最后一个元素 O(1);
```



Vector操作示例

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    vector<int> v;
    v.push_back(1); //依次输入1、 2、 3
    v.push_back(2);
    v.push_back(3);
    for(vector<int>::iterator it = v.begin(); it != v.end(); it++){
        cout << *it << " ";
    }

    return 0;
}
```

参考资料

- **https://www.jianshu.com/p/e8e267879b61**
- **https://blog.csdn.net/m0_56494923/article/details/123263285**
- **……**