# 数据结构与算法

DATA STRUCTURE

第二十六讲 复习

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

## 课堂内容

- 作业回顾
- 复习

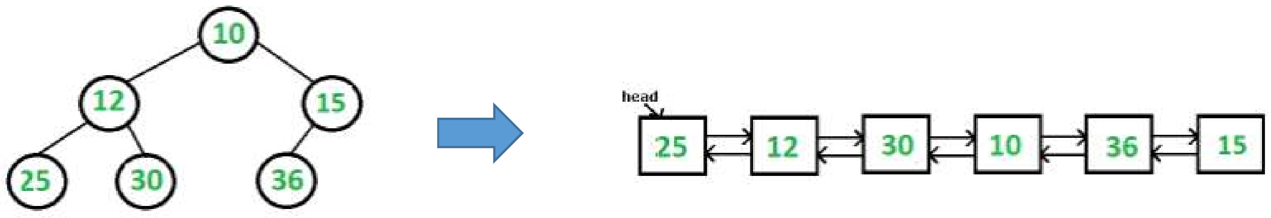
# Tree to Double list

```cpp
  void TreeToDoubleList(DoublyList * outList);

private:
  void TreeToDoubleList(Node * root, DoublyList * outList);
```

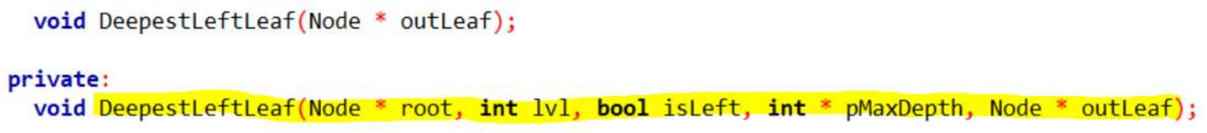
```cpp
void BinaryTree::TreeToDoubleList(DoublyList * outList)
{
    TreeToDoubleList(_pRoot, outList);
    // 这里原树节点都移动到doubly list，更改了连接指针，所以直接断开
    _pRoot = nullptr;
}

void BinaryTree::TreeToDoubleList(Node * root, DoublyList * outList)
{
    if (root == nullptr) return;

    TreeToDoubleList(root->left, outList);

    Node * saveRight = root->right;
    outList->PushBack(root);

    TreeToDoubleList(saveRight, outList);
}
```

```cpp
void DoublyList::PushBack(Node * nd)
{
    if (nd == nullptr)
    {
        return;
    }
    nd->left = nd->right = nullptr;

    if (_tail == nullptr)
    {
        _head = _tail = nd;
    }
    else
    {
        _tail->right = nd;
        nd->left = _tail;
        _tail = nd;
    }
}
```

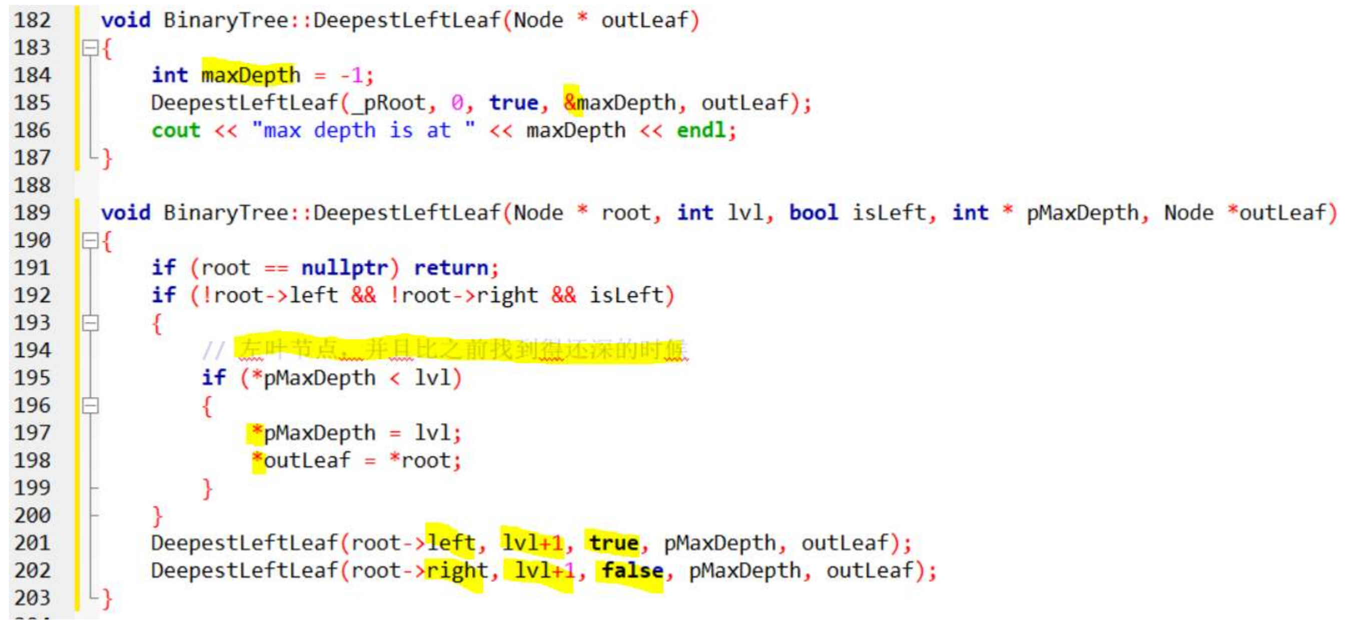
```cpp
void treeToDoublyList(Node *root, Node **head)
{
  if (!root) return;

  static Node *prev = nullptr;

  // Recursively convert left subtree
  treeToDoublyList(p->left, head);

  // Now convert this node
  if (!prev)
  {
    *head = root;
  }
  else
  {
    root->left = prev;
    prev->right = root;
  }

  // updates previous node
  prev = root;

  treeToDoublyList(root->right, head);
}
```



# Deepest left leaf node

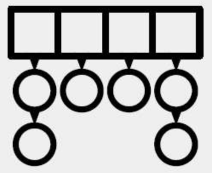


```cpp
  void DeepestLeftLeaf(Node * outLeaf);

private:
  void DeepestLeftLeaf(Node * root, int lvl, bool isLeft, int * pMaxDepth, Node * outLeaf);
```



```cpp
void BinaryTree::DeepestLeftLeaf(Node * outLeaf)
{
    int maxDepth = -1;
    DeepestLeftLeaf(_pRoot, 0, true, &maxDepth, outLeaf);
    cout << "max depth is at " << maxDepth << endl;
}

void BinaryTree::DeepestLeftLeaf(Node * root, int lvl, bool isLeft, int * pMaxDepth, Node *outLeaf)
{
    if (root == nullptr) return;
    if (!root->left && !root->right && isLeft)
    {
        // 左叶节点，并且比之前找到得还深的时候
        if (*pMaxDepth < lvl)
        {
            *pMaxDepth = lvl;
            *outLeaf = *root;
        }
    }
    DeepestLeftLeaf(root->left, lvl+1, true, pMaxDepth, outLeaf);
    DeepestLeftLeaf(root->right, lvl+1, false, pMaxDepth, outLeaf);
}
```

# 图的表示

| For G = (V, E) | $\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$ |  |
|---|---|---|
| **Edge Membership**<br>Is e = {u, v} in E? | O(1) | O(deg(u))<br>**or**<br>O(deg(v)) |
| **Neighbor Query**<br>What are the neighbors of u? | O(\|V\|) | O(deg(v)) |
| **Space requirements** | $O(\|V\|^2)$ | O(\|V\|+\|E\|) |

# 图遍历算法

- 深度优先
  - 深度优先+始末时间
  - 拓扑排序
  - 检测图上的cycle
  - 强连通分支
- 广度优先
  - 按边算的最短路径
  - 检测二分图
  - 无向图上的连通分支

# 最短路径算法和动态规划算法

| | Dijkstra | Bellman-Ford | Floyd-Warshall | DAG上的最短路径 |
|---|---|---|---|---|
| **Problem** | Single source shortest path | Single source shortest path | All pairs shortest path | Single source |
| **Runtime** | $O(\lvert E\rvert+\lvert V\rvert\log(\lvert V\rvert))$ **worst-case** with a fibonacci heap | $O(\lvert V\rvert\,\lvert E\rvert)$ **worst-case** | $O(\lvert V\rvert^3)$ **worst case** | $O(\lvert V\rvert+\lvert E\rvert)$ |

$$d^k(v) = \begin{cases} \infty & k = 0, v \neq s \\ 0 & k = 0, v = s \\ \min\{\underbrace{d^{k-1}(v)}_{\text{Case 1}},\ \underbrace{\min_u\{d^{k-1}(u) + w(u,v)\}}_{\text{Case 2}}\} & \end{cases}$$

Case 1

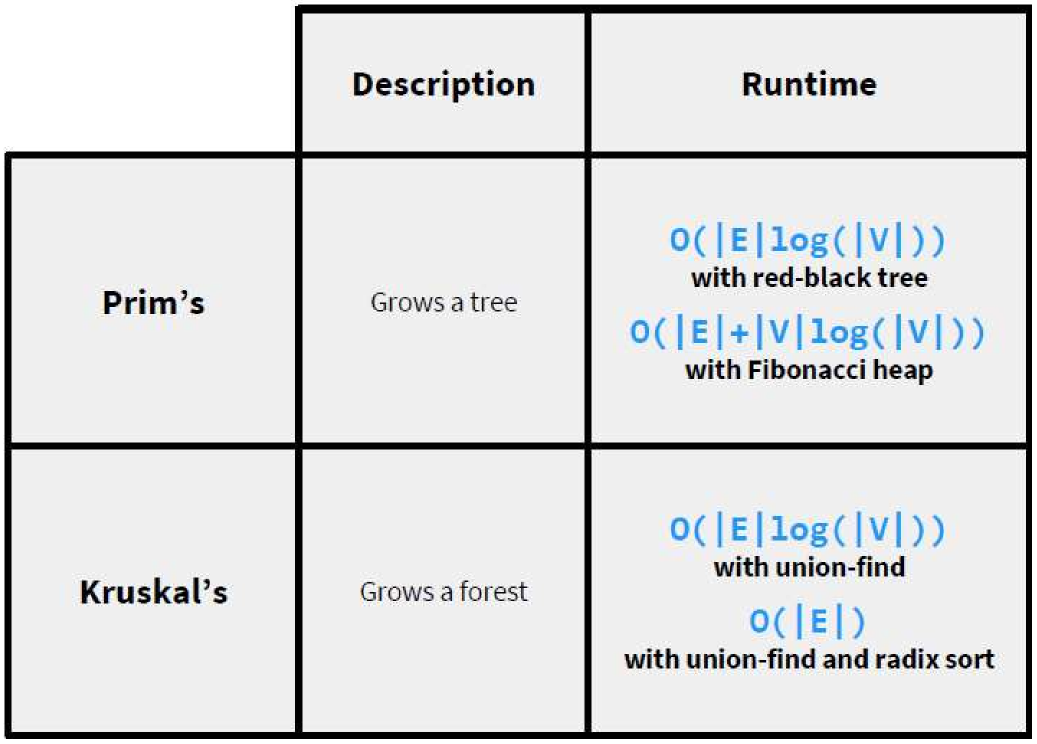
Case 2：对所有的顶点$u$，存在边$\{u,v\}$

$$D^k(u,v) = \begin{cases} w(u,v) & k = 0 \\ \min\{\underbrace{D^{k-1}(u,v)}_{\text{Case 1}},\ \underbrace{D^{k-1}(u,k-1) + D^{k-1}(k-1,v)}_{\text{Case 2}}\} & \end{cases}$$

Case 1

Case 2

# 最小生成树MST和贪婪算法



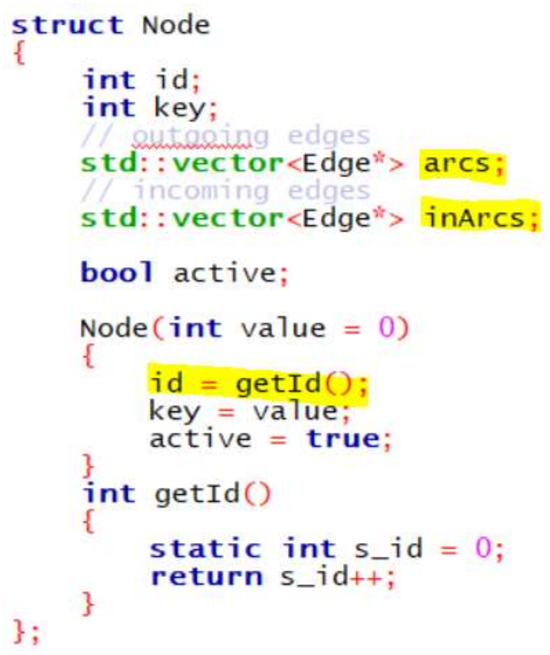
| | Description | Runtime |
|---|---|---|
| **Prim's** | Grows a tree | O(\|E\|log(\|V\|)) **with red-black tree**<br>O(\|E\|+\|V\|log(\|V\|)) **with Fibonacci heap** |
| **Kruskal's** | Grows a forest | O(\|E\|log(\|V\|)) **with union-find**<br>O(\|E\|) **with union-find and radix sort** |

选切割线上的最短边

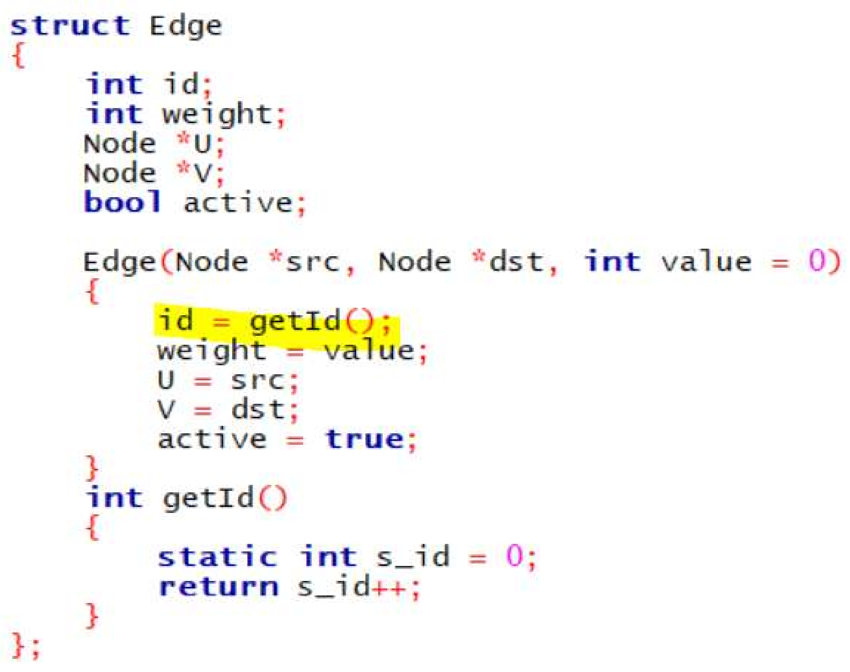
$\boldsymbol{d(v) \leftarrow min\{d(v), w(u,v)\}}$

选剩下边里的最短边

# 点/边的结构体



```cpp
struct Node
{
    int id;
    int key;
    // outgoing edges
    std::vector<Edge*> arcs;
    // incoming edges
    std::vector<Edge*> inArcs;

    bool active;

    Node(int value = 0)
    {
        id = getId();
        key = value;
        active = true;
    }
    int getId()
    {
        static int s_id = 0;
        return s_id++;
    }
};
```

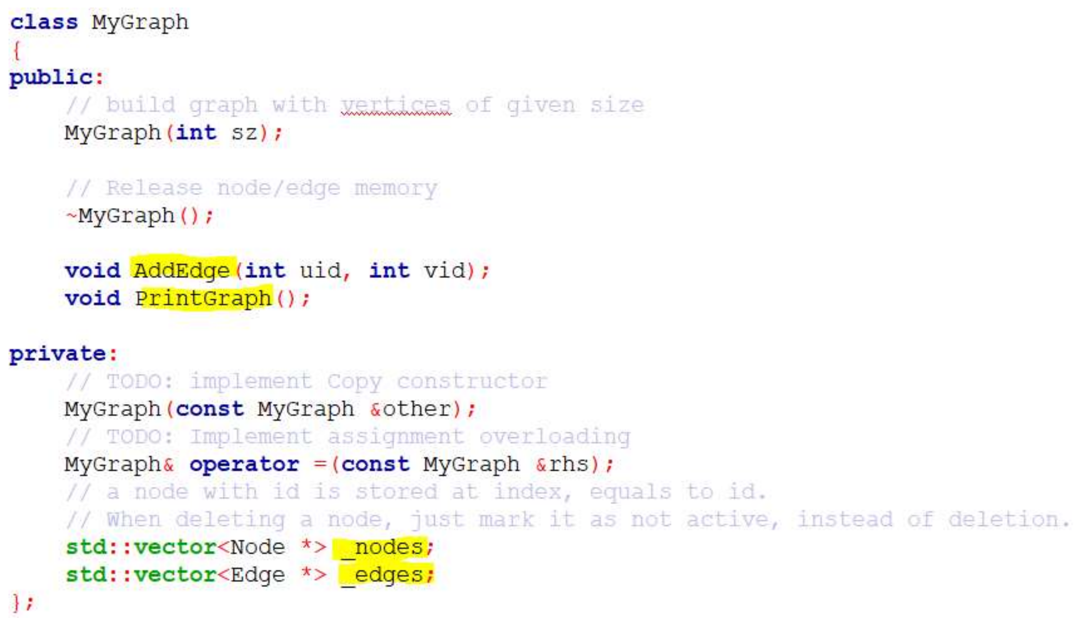


```cpp
struct Edge
{
    int id;
    int weight;
    Node *U;
    Node *V;
    bool active;

    Edge(Node *src, Node *dst, int value = 0)
    {
        id = getId();
        weight = value;
        U = src;
        V = dst;
        active = true;
    }
    int getId()
    {
        static int s_id = 0;
        return s_id++;
    }
};
```

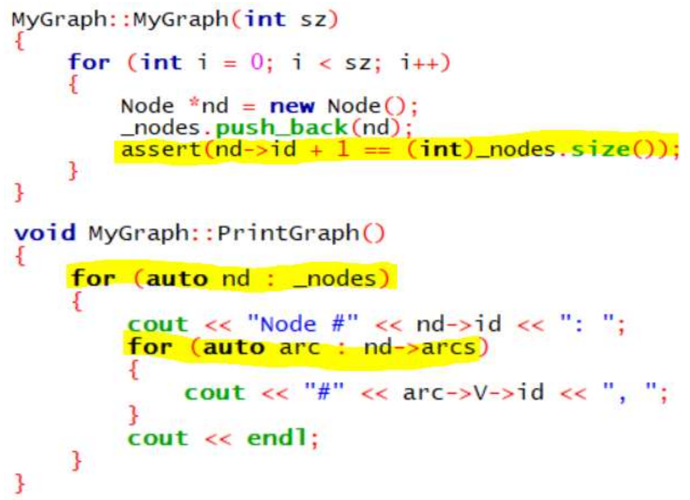
用getId()自动产生连续id，之后可以用id对应点/边

## MyGraph类

- 注意用id对应点/边数组下标，可以很大程度上简化相关算法，不过得保证匹配。
- 比如加点时候，push_back()加到最后一个，然后assert数组大小等于点id+1
- 如果需要删除点，不用直接删除，而是标明不用。这样即避免了数组下标对应的问题，而且可以恢复点。



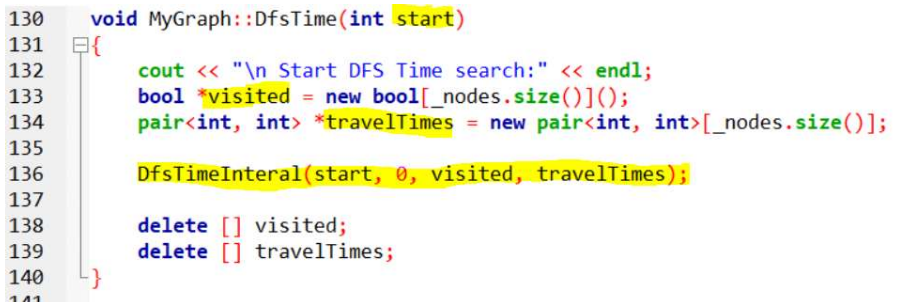
```
class MyGraph
{
public:
    // build graph with vertices of given size
    MyGraph(int sz);

    // Release node/edge memory
    ~MyGraph();

    void AddEdge(int uid, int vid);
    void PrintGraph();

private:
    // TODO: implement Copy constructor
    MyGraph(const MyGraph &other);
    // TODO: Implement assignment overloading
    MyGraph& operator =(const MyGraph &rhs);
    // a node with id is stored at index, equals to id.
    // When deleting a node, just mark it as not active, instead of deletion.
    std::vector<Node *> _nodes;
    std::vector<Edge *> _edges;
};
```



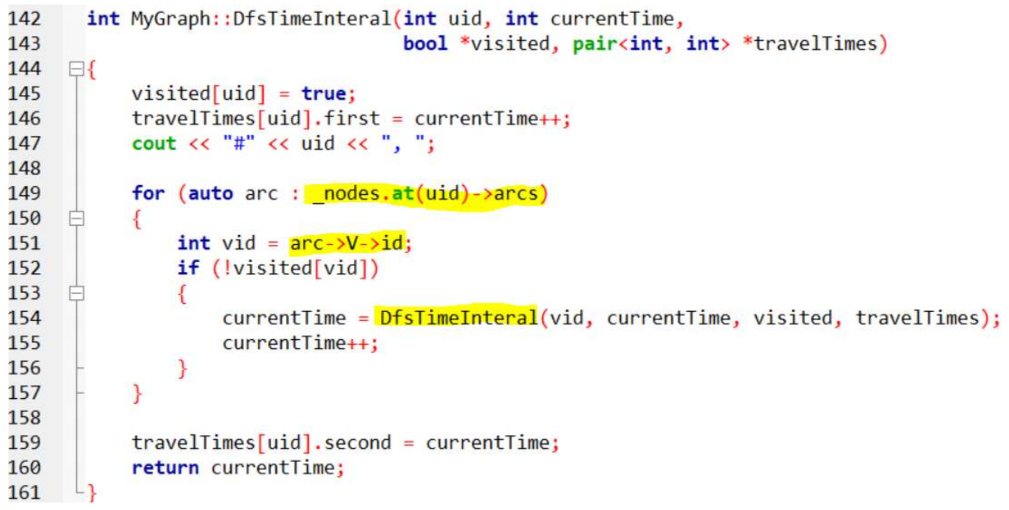
```
MyGraph::MyGraph(int sz)
{
    for (int i = 0; i < sz; i++)
    {
        Node *nd = new Node();
        _nodes.push_back(nd);
        assert(nd->id + 1 == (int)_nodes.size());
    }
}

void MyGraph::PrintGraph()
{
    for (auto nd : _nodes)
    {
        cout << "Node #" << nd->id << ": ";
        for (auto arc : nd->arcs)
        {
            cout << "#" << arc->V->id << ", ";
        }
        cout << endl;
    }
}
```

# Dfs+始末时间

```
DFS(Node u, Time currentTime)
    u.start = currentTime
    currentTime += 1
    u.status  =  🟡
    foreach （v是u的邻接顶点）
        if v.status ==  ○
            currentTime = DFS(v, currentTime)
            currentTime +=1
    u.end = currentTime
    u.status  =  ●
    return currentTime
```



```cpp
void MyGraph::DfsTime(int start)
{
    cout << "\n Start DFS Time search:" << endl;
    bool *visited = new bool[_nodes.size()]();
    pair<int, int> *travelTimes = new pair<int, int>[_nodes.size()];

    DfsTimeInteral(start, 0, visited, travelTimes);

    delete [] visited;
    delete [] travelTimes;
}
```



```cpp
int MyGraph::DfsTimeInteral(int uid, int currentTime,
                            bool *visited, pair<int, int> *travelTimes)
{
    visited[uid] = true;
    travelTimes[uid].first = currentTime++;
    cout << "#" << uid << ", ";

    for (auto arc : _nodes.at(uid)->arcs)
    {
        int vid = arc->V->id;
        if (!visited[vid])
        {
            currentTime = DfsTimeInteral(vid, currentTime, visited, travelTimes);
            currentTime++;
        }
    }

    travelTimes[uid].second = currentTime;
    return currentTime;
}
```

# 找SCC算法

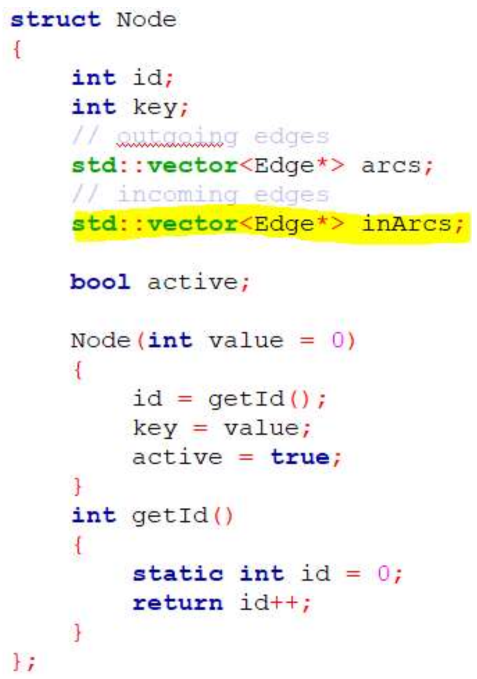
- 在线性时间内找到有向图的SCC

  O(|V|+|E|)

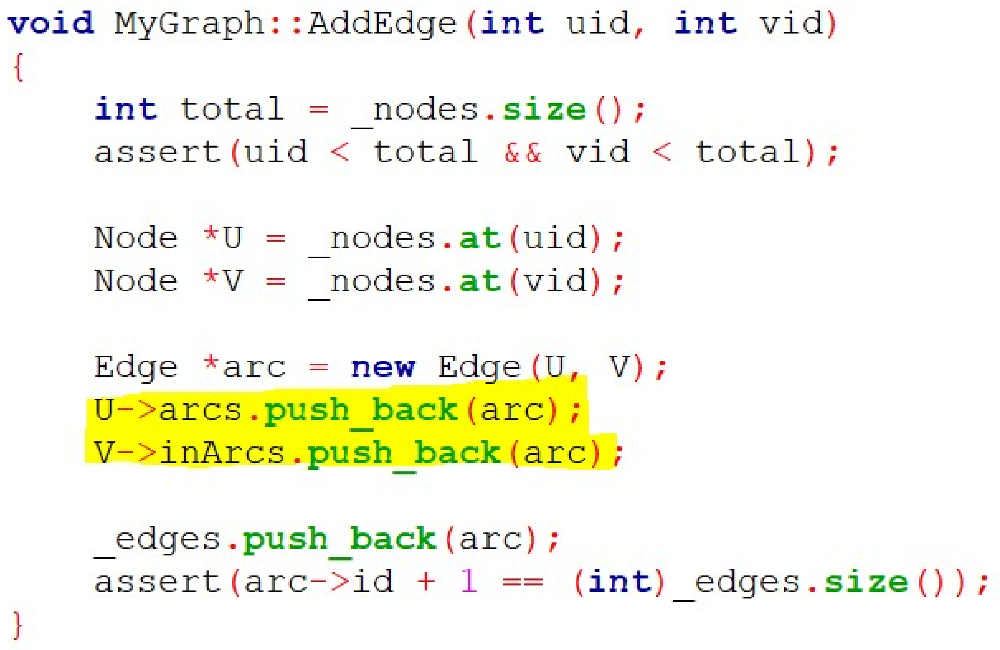
- **Kosaraju**算法
  1) 使用DFS算法+始末时间遍历所有顶点
     - 按起始的位置不同，可能会产生一个DFS森林
  2) 把图里的所有边反向
  3) 再运行一遍DFS算法，不过顺序是从第一次DFS算法中结束时间最晚的顶点作为起始点
     - 同样会产生一个DFS森林
  4) 第二遍DFS算法找到的不同DFS树就是强连通分支

## FindScc

- 每个节点保存了出边集合和入边集合
- 每添加一条边，需要在两个顶点结构体里加边

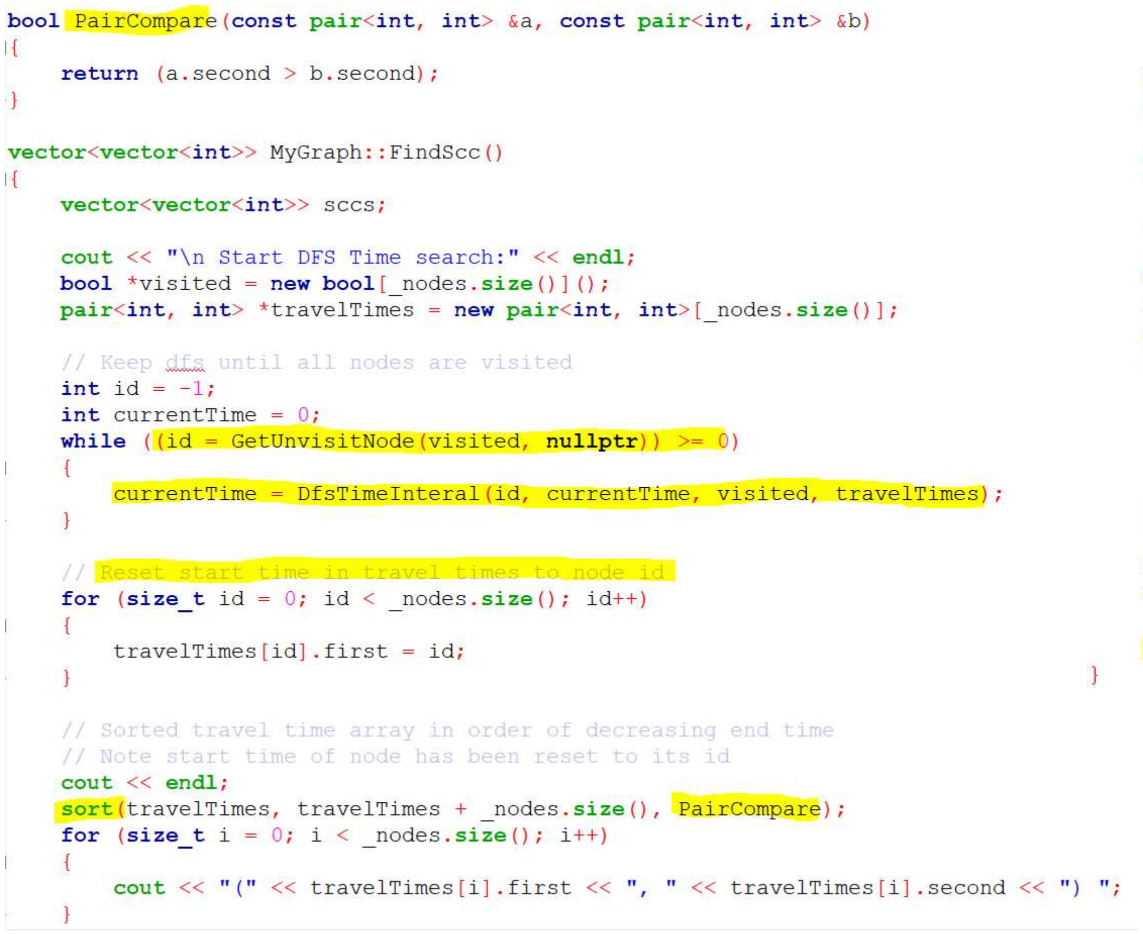


```cpp
struct Node
{
    int id;
    int key;
    // outgoing edges
    std::vector<Edge*> arcs;
    // incoming edges
    std::vector<Edge*> inArcs;

    bool active;

    Node(int value = 0)
    {
        id = getId();
        key = value;
        active = true;
    }
    int getId()
    {
        static int id = 0;
        return id++;
    }
};
```

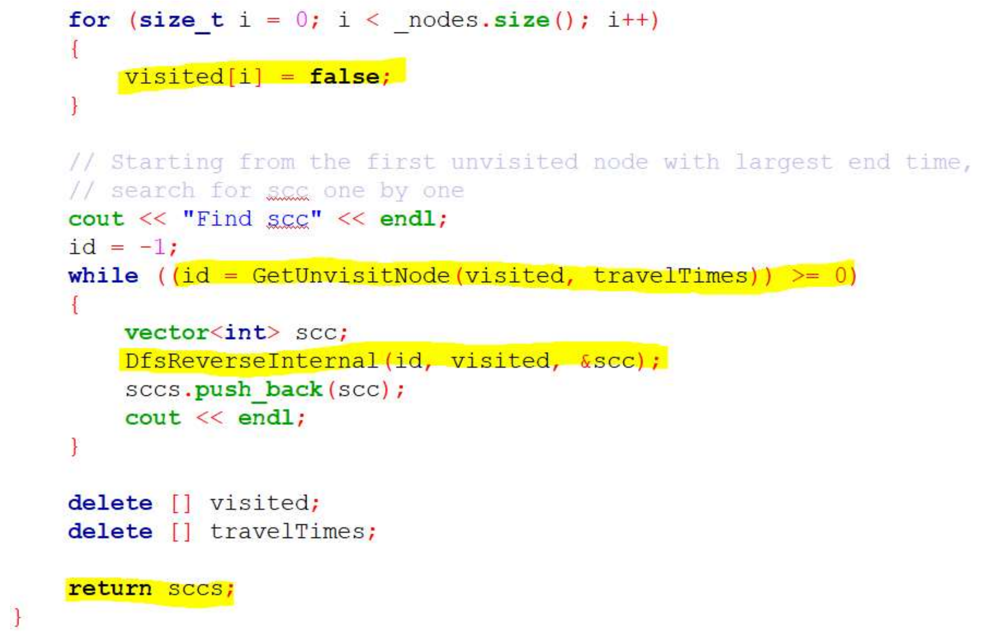


```cpp
void MyGraph::AddEdge(int uid, int vid)
{
    int total = _nodes.size();
    assert(uid < total && vid < total);

    Node *U = _nodes.at(uid);
    Node *V = _nodes.at(vid);

    Edge *arc = new Edge(U, V);
    U->arcs.push_back(arc);
    V->inArcs.push_back(arc);

    _edges.push_back(arc);
    assert(arc->id + 1 == (int)_edges.size());
}
```

# FindScc



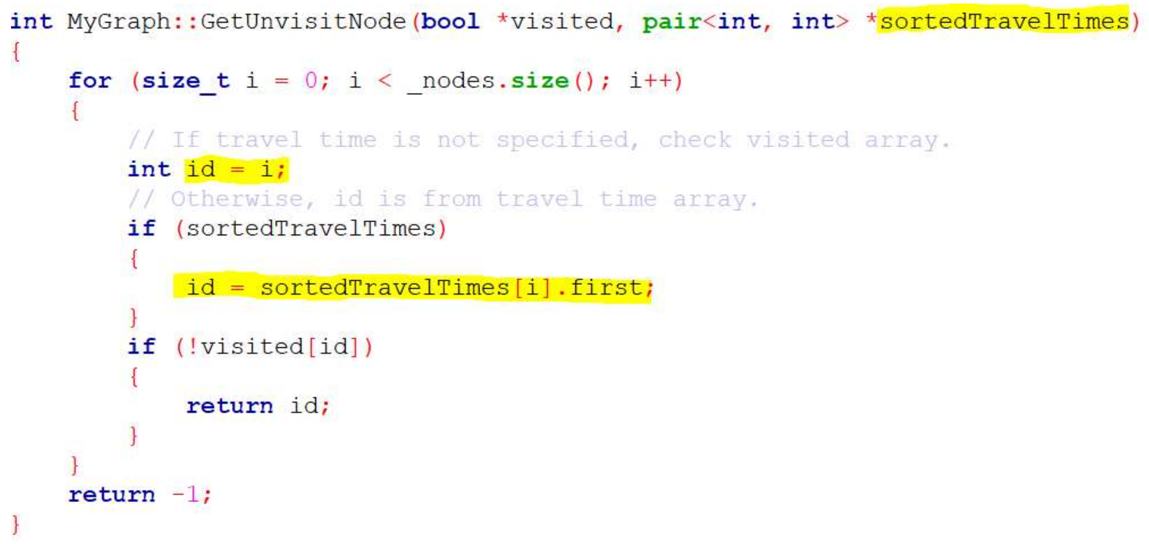
```
bool PairCompare(const pair<int, int> &a, const pair<int, int> &b)
{
    return (a.second > b.second);
}

vector<vector<int>> MyGraph::FindScc()
{
    vector<vector<int>> sccs;

    cout << "\n Start DFS Time search:" << endl;
    bool *visited = new bool[_nodes.size()]();
    pair<int, int> *travelTimes = new pair<int, int>[_nodes.size()];

    // Keep dfs until all nodes are visited
    int id = -1;
    int currentTime = 0;
    while ((id = GetUnvisitNode(visited, nullptr)) >= 0)
    {
        currentTime = DfsTimeInteral(id, currentTime, visited, travelTimes);
    }

    // Reset start time in travel times to node id
    for (size_t id = 0; id < _nodes.size(); id++)
    {
        travelTimes[id].first = id;
    }

    // Sorted travel time array in order of decreasing end time
    // Note start time of node has been reset to its id
    cout << endl;
    sort(travelTimes, travelTimes + _nodes.size(), PairCompare);
    for (size_t i = 0; i < _nodes.size(); i++)
    {
        cout << "(" << travelTimes[i].first << ", " << travelTimes[i].second << ") ";
    }
```



```
    for (size_t i = 0; i < _nodes.size(); i++)
    {
        visited[i] = false;
    }

    // Starting from the first unvisited node with largest end time,
    // search for scc one by one
    cout << "Find scc" << endl;
    id = -1;
    while ((id = GetUnvisitNode(visited, travelTimes)) >= 0)
    {
        vector<int> scc;
        DfsReverseInternal(id, visited, &scc);
        sccs.push_back(scc);
        cout << endl;
    }

    delete [] visited;
    delete [] travelTimes;

    return sccs;
}
```

## FindScc

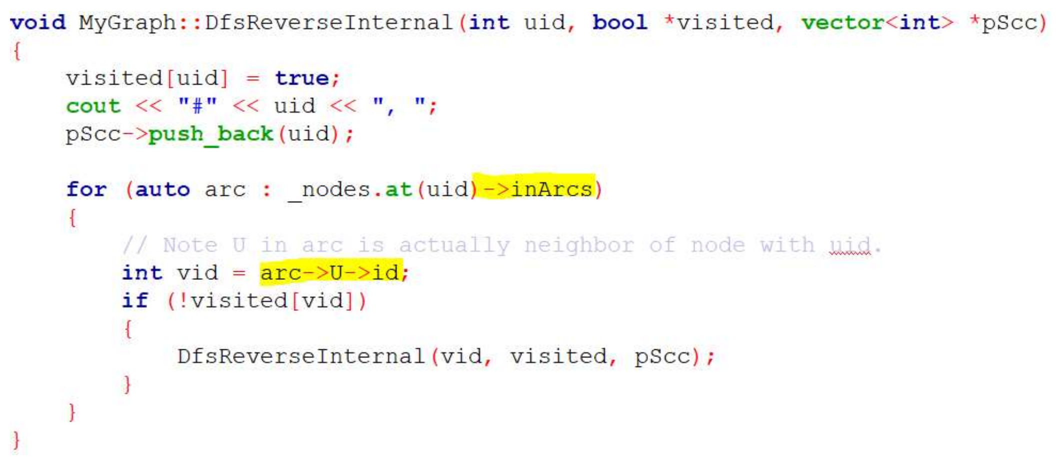
- 辅助函数，找下一个没访问过的节点
- 如果有按结束时间排序的traveltime，需要找结束时间最晚的第一个没访问过的节点



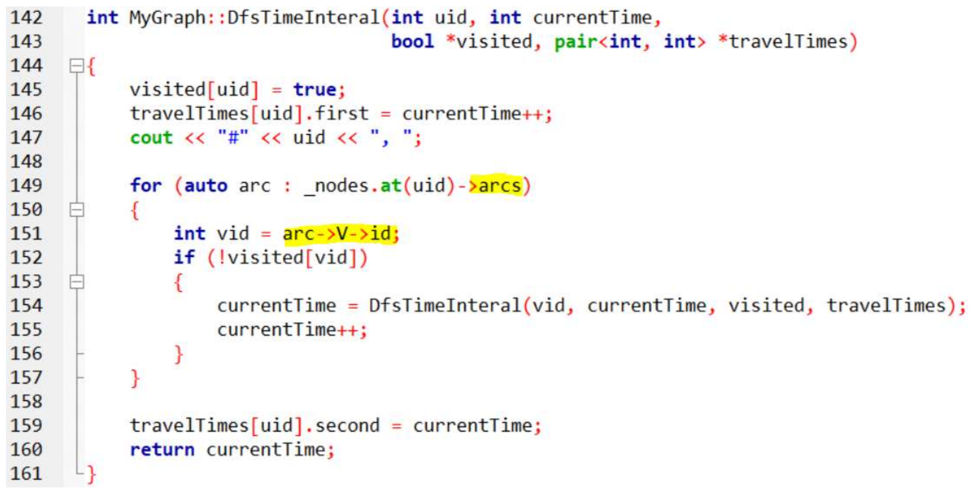
```
int MyGraph::GetUnvisitNode(bool *visited, pair<int, int> *sortedTravelTimes)
{
    for (size_t i = 0; i < _nodes.size(); i++)
    {
        // If travel time is not specified, check visited array.
        int id = i;
        // Otherwise, id is from travel time array.
        if (sortedTravelTimes)
        {
            id = sortedTravelTimes[i].first;
        }
        if (!visited[id])
        {
            return id;
        }
    }
    return -1;
}
```

# FindScc

- 按反向边的DFS递归函数
- 因为我们在节点里加了入边，所以直接用入边进行DFS
- 但注意邻接顶点是arc里的U



```
void MyGraph::DfsReverseInternal(int uid, bool *visited, vector<int> *pScc)
{
    visited[uid] = true;
    cout << "#" << uid << ", ";
    pScc->push_back(uid);

    for (auto arc : _nodes.at(uid)->inArcs)
    {
        // Note U in arc is actually neighbor of node with uid.
        int vid = arc->U->id;
        if (!visited[vid])
        {
            DfsReverseInternal(vid, visited, pScc);
        }
    }
}
```



```
int MyGraph::DfsTimeInteral(int uid, int currentTime,
                             bool *visited, pair<int, int> *travelTimes)
{
    visited[uid] = true;
    travelTimes[uid].first = currentTime++;
    cout << "#" << uid << ", ";

    for (auto arc : _nodes.at(uid)->arcs)
    {
        int vid = arc->V->id;
        if (!visited[vid])
        {
            currentTime = DfsTimeInteral(vid, currentTime, visited, travelTimes);
            currentTime++;
        }
    }

    travelTimes[uid].second = currentTime;
    return currentTime;
}
```

## Dijkstra算法

- 初始化
  - 未完成 🟡
  - 完成 🔵
- 最短距离迭代计算过程

```
For each vertex $u$
    $d(u) = \infty$;  $u.status =$ 🟡
$d(s) = 0$
```

```
For $i = 1, \ldots, n$:
    Find $u$ with status 🟡,  such that $d(u)$ is min
    For each neighbor $v$ with status 🟡:
        $d(v) \leftarrow min\{d(v), d(u) + w(u,v)\}$
    $u.status =$ 🔵
```

1) 取一个“未完成”的顶点$u$，其预估值$d(u)$是最小的
2) 对于顶点$u$的所有“未完成”邻接顶点$v$，进行单步收敛操作
   $d(v) \leftarrow min\{d(v), d(u) + w(u,v)\}$
3) 然后把顶点$u$的状态设置成“完成”

# Dijkstra

```
struct HeapNode
{
    int id;
    int priority;
    HeapNode(int key, int pri)
    {
        id = key;
        priority = pri;
    }
};
```

```
class MinHeap
{
public:
    // Add a dummy node to heap so that 1st node starts at index 1.
    MinHeap() { m_data.push_back(HeapNode{-1, -1}); }

    // A recursive method to heapify a subtree with root at given index
    // This method assumes that the subtrees are already heapified
    void MinHeapify(int i);

    // to extract the root which is the minimum element
    int ExtractMin();

    // Returns the minimum key (key at root) from min heap
    HeapNode GetMin() { return m_data[1]; }

    // Inserts a new key with priority 'pri'
    void InsertKey(int key, int pri);

private:
    std::vector<HeapNode> m_data;
};
```

```
bool MyGraph::Dijkstra(int s, int t)
{
    MinHeap pq; // min heap as priority queue
    int sz = (int)_nodes.size();

    // flag indicating whether a node has been visited.
    bool *visited = new bool[sz]();
    // If a shorter distance is found, parent link should be updated
    int *parent = new int[sz]();

    // Estimate distance from source node 's'.
    int *dist = new int[sz]();
    for (int id = 0; id < sz; id++)
    {
        // initialize to max, except for source node itself
        dist[id] = id == s ? 0 : INT_MAX;
        parent[id] = -1;
        // Maintain a priority queue for unvisited nodes
        pq.insertKey(id, dist[id]);
    }

    bool success = false;
    for (int i = 0; i < sz && !success; i++)
    {
        // It is difficult to implement decreaseKey, as we should
        // find out which node in heap corresponding to adjacent node.
        // Trick here: Always insert node with updated dist[].
        // Consequently, we should mark a node visited and remove min node if visited.
        while (visited[pq.getMin().id])
        {
            pq.extractMin();
        }

        // Now the min node is unvisited node.
        HeapNode hn = pq.getMin();
        pq.extractMin();

        // This indicated other unvisited nodes are not reachable.
        if (hn.priority == INT_MAX)
        {
            break;
        }
```

# Dijkstra

```
// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int min = i;
    if (l < (int)m_data.size() && m_data[l].priority < m_data[min].priority)
    {
        min = l;
    }
    if (r < (int)m_data.size() && m_data[r].priority < m_data[min].priority)
    {
        min = r;
    }

    if (min != i)
    {
        swap(m_data[i], m_data[min]);
        MinHeapify(min);
    }
}
// Inserts a new key with priority 'pri'
void MinHeap::insertKey(int key, int pri)
{
    // First insert the new key at the end
    m_data.push_back(HeapNode{key, pri});
    int i = m_data.size() - 1;


    // Fix the min heap property if it is violated
    while (i > 1 && m_data[parent(i)].priority > m_data[i].priority)
    {
        swap(m_data[i], m_data[parent(i)]);
        i = parent(i);
    }
}
```

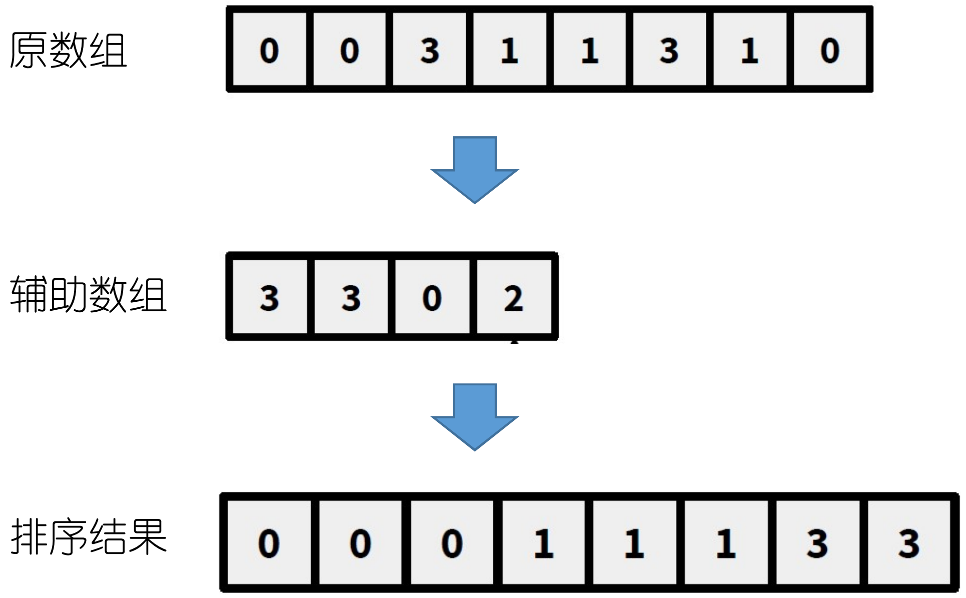
```
        visited[hn.id] = true;
        for (auto arc : _nodes.at(hn.id)->arcs)
        {
            int vid = arc->V->id;
            if (visited[vid])
            {
                continue;
            }

            // For each unvisited neighbor, update estimate dist[].
            if (dist[vid] > dist[hn.id] + arc->weight)
            {
                // update estimate distance, parent link
                // Trick: Insert updated value into heap, instead of decreasing key.
                dist[vid] = dist[hn.id] + arc->weight;
                parent[vid] = hn.id;
                pq.insertKey(vid, dist[vid]);
            }
            if (vid == t)
            {
                success = true;
                break;
            }
        }
    }


    if (success)
    {
        cout << "Find path from #" << s << " to #" << t << endl;
        int curr = t;
        while (parent[curr] != -1)
        {
            cout << "#" << curr << "["<< dist[curr] << "]" << " <- ";
            curr = parent[curr];
        }
        cout << "#" << curr << "["<< dist[curr] << "]" << endl;
    }
    else
    {
        cout << "No path from #" << s << " to #" << t << endl;
    }

    return success;
}
```

## 排序算法

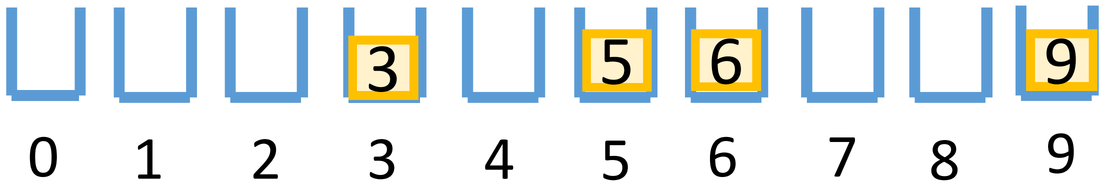
- 比较排序$O(nlogn)$
  - MergeSort
  - QuickSort

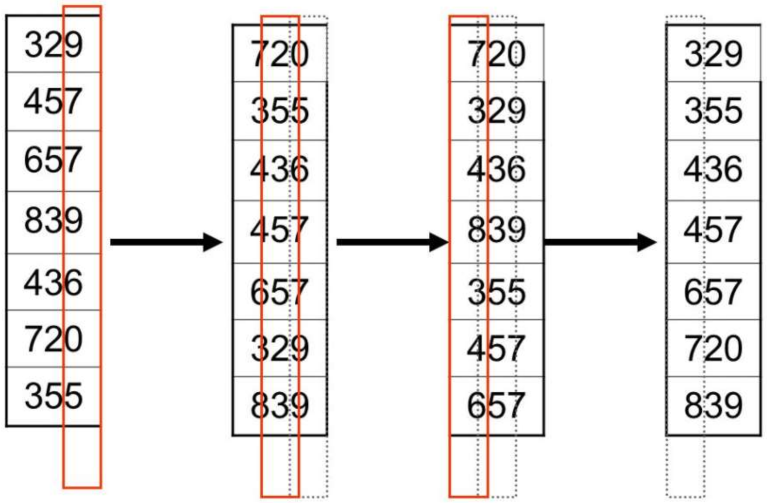
- 特殊排序$O(n)$
  - CoutingSort
  - RadixSort
  - BucketSort

## Couting sort



# Bucket sort和radix sort





## 基本数据结构

- 数组和链表
- Stack
- Queue
- Recursion
- Heap
- HeapSort

## Heap应用

- 数列中求k-th小的数
    - 排序，数数到第k个
    - 用最小堆，连续取k次
    - 前k个数的最大堆，逐个添加后面n-k个数。每加一个，同时去掉最大元素
    - 建立BST，递归查询FindKth(Node * root, int k)
- Online steam里找最大的k个数
- 合并K个有序数组

## 串匹配算法

- 暴力算法
- KR算法（hashing值）
- **KMP（最长前后缀匹配长度，next数组）**
- BM （好后缀和坏字符规则）
- 改进的Boyer–Moore–Horspool算法（只有坏字符规则）

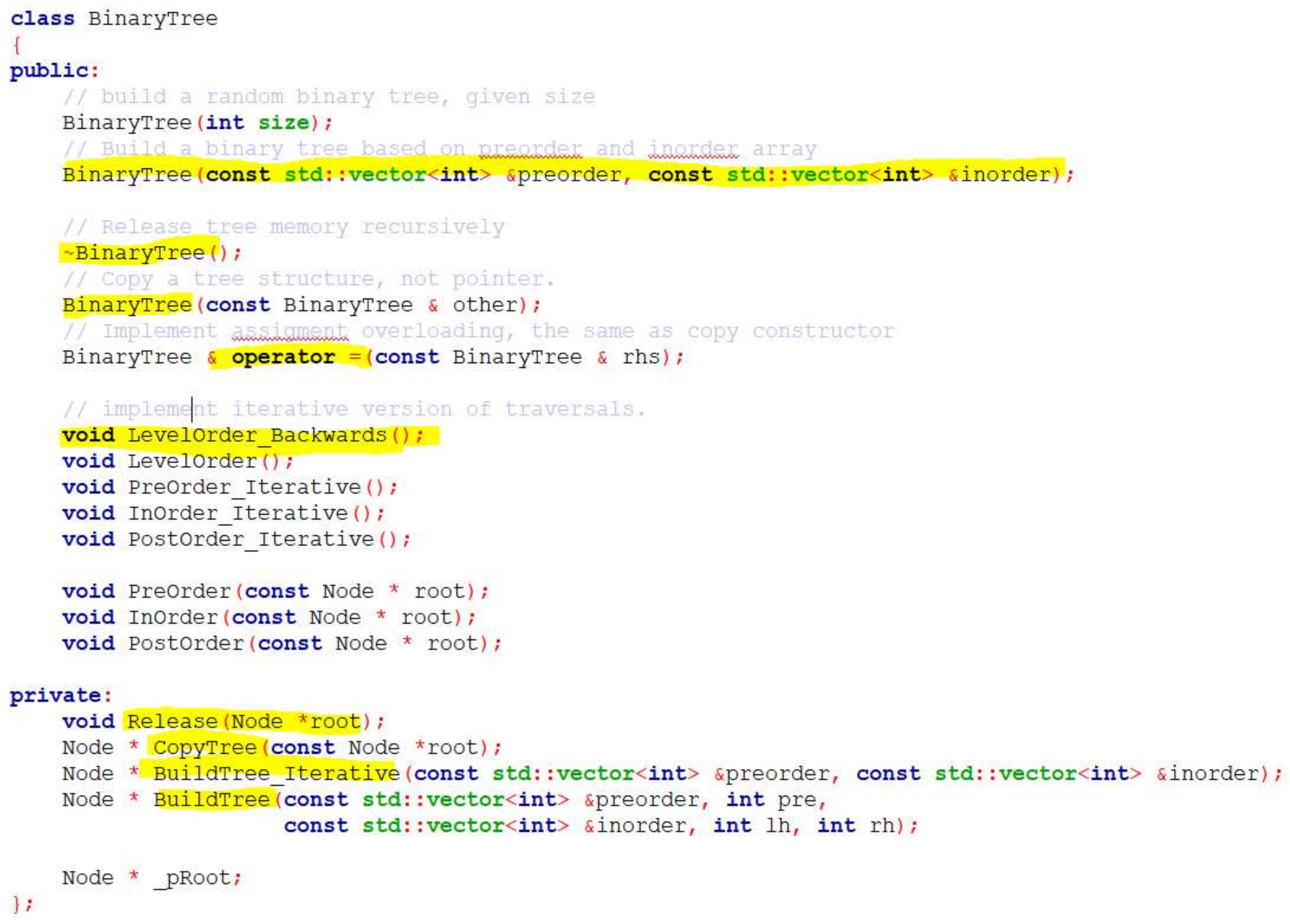
## 树

- 数的遍历
  - 中序，前序，后序
  - 遍历迭代算法
- 二叉树（线索化）
- 二叉查找树
  - AVL
  - 红黑树
- B树和B+树
- Trie
- Hash

## 比较

| | Sorted linked lists | Sorted arrays | Balanced BSTs | Hash tables |
|---|---|---|---|---|
| **Search** | O(n) **expected & worst-case** | O(log n) **expected & worst-case** | O(log n) **expected & worst-case** O(n) worst-case for generic BSTs | O(1) **expected** O(n) **worst-case** |
| **Insert/ Delete** | O(n) **expected & worst-case** without a pointer to the element | O(n) **expected & worst-case** | O(log n) **expected & worst case** | O(1) **expected** O(n) **worst-case** without a pointer to the element |

# BinaryTree类



```
class BinaryTree
{
public:
    // build a random binary tree, given size
    BinaryTree(int size);
    // Build a binary tree based on preorder and inorder array
    BinaryTree(const std::vector<int> &preorder, const std::vector<int> &inorder);

    // Release tree memory recursively
    ~BinaryTree();
    // Copy a tree structure, not pointer.
    BinaryTree(const BinaryTree & other);
    // Implement assigment overloading, the same as copy constructor
    BinaryTree & operator =(const BinaryTree & rhs);

    // implement iterative version of traversals.
    void LevelOrder_Backwards();
    void LevelOrder();
    void PreOrder_Iterative();
    void InOrder_Iterative();
    void PostOrder_Iterative();

    void PreOrder(const Node * root);
    void InOrder(const Node * root);
    void PostOrder(const Node * root);

private:
    void Release(Node *root);
    Node * CopyTree(const Node *root);
    Node * BuildTree_Iterative(const std::vector<int> &preorder, const std::vector<int> &inorder);
    Node * BuildTree(const std::vector<int> &preorder, int pre,
                     const std::vector<int> &inorder, int lh, int rh);

    Node * _pRoot;
};
```

# 构造/析构函数

```
BinaryTree::BinaryTree(const BinaryTree &other)
    :
    _pRoot(nullptr)
{
    _pRoot = CopyTree(other._pRoot);
}
```

```
BinaryTree::~BinaryTree()
{
    DeleteTree();
}
```

```
BinaryTree& BinaryTree::operator =(const BinaryTree & rhs)
{
    if (this == &rhs)
    {
        return *this;
    }

    DeleteTree();
    _pRoot = CopyTree(rhs._pRoot);

    return *this;
}
```

```
Node * BinaryTree::CopyTree(const Node *root)
{
    if (root == nullptr)
    {
        return nullptr;
    }
    Node * copyRoot = new Node(root->key);
    copyRoot->left = CopyTree(root->left);
    copyRoot->right = CopyTree(root->right);

    return copyRoot;
}
```

```
void BinaryTree::DeleteTree()
{
    Release(_pRoot);
    _pRoot = nullptr;
}
```

```
void BinaryTree::Release(Node * root)
{
    if (root == nullptr)
    {
        return;
    }
    Release(root->left);
    Release(root->right);
    root->left = nullptr;
    root->right = nullptr;
    delete root;
}
```

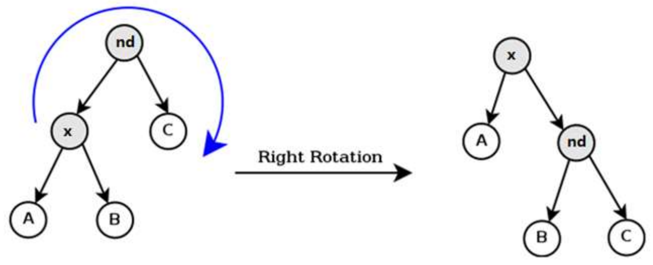
## 辅助函数

```cpp
struct Node
{
    int key;
    int height;
    Node * left;
    Node * right;

    Node(int val)
    {
        key = val;
        height = 0;
        left = right = nullptr;
    }
};
```

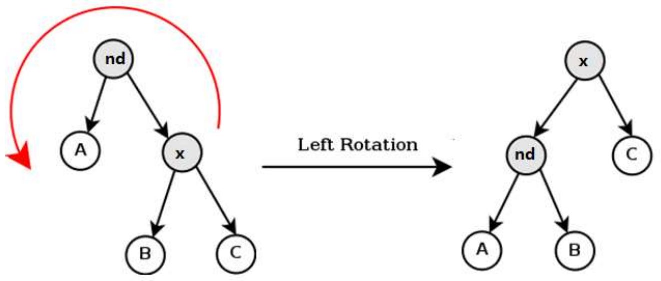
```cpp
int AvlTree::getHeight(Node *nd)
{
    if (nd == nullptr)
    {
        return -1;
    }
    return nd->height;
}

int AvlTree::getBalance(Node *nd)
{
    if (nd == nullptr)
    {
        return 0;
    }
    return getHeight(nd->left) - getHeight(nd->right);
}
```

## 旋转

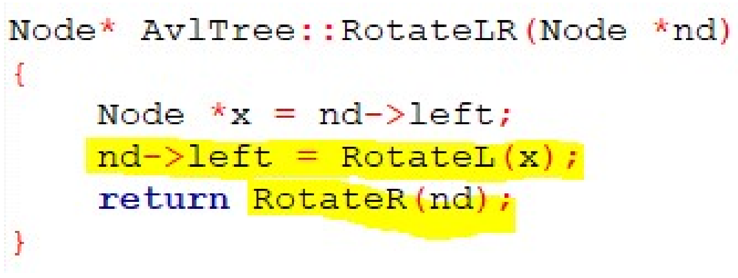
```
Node* AvlTree::RotateR(Node *nd)
{
    Node *x = nd->left;
    nd->left = x->right;
    x->right = nd;

    nd->height = max(getHeight(nd->left), getHeight(nd->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

    return x;
}
```



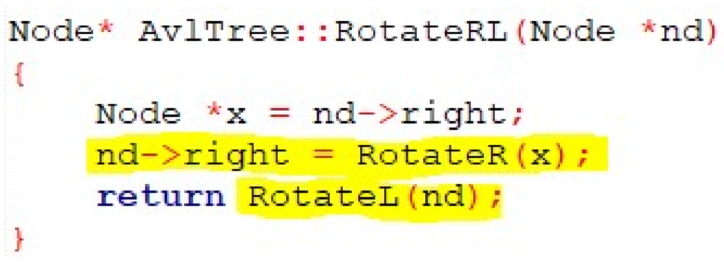
```
Node* AvlTree::RotateL(Node *nd)
{
    Node *x = nd->right;
    nd->right = x->left;
    x->left = nd;

    nd->height = max(getHeight(nd->left), getHeight(nd->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

    return x;
}
```



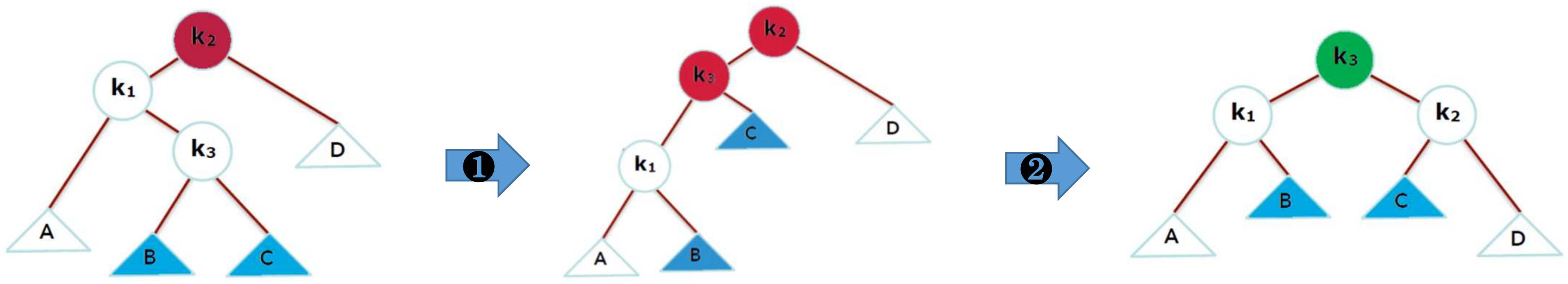
# 左旋右旋LR 和 右旋左旋RL



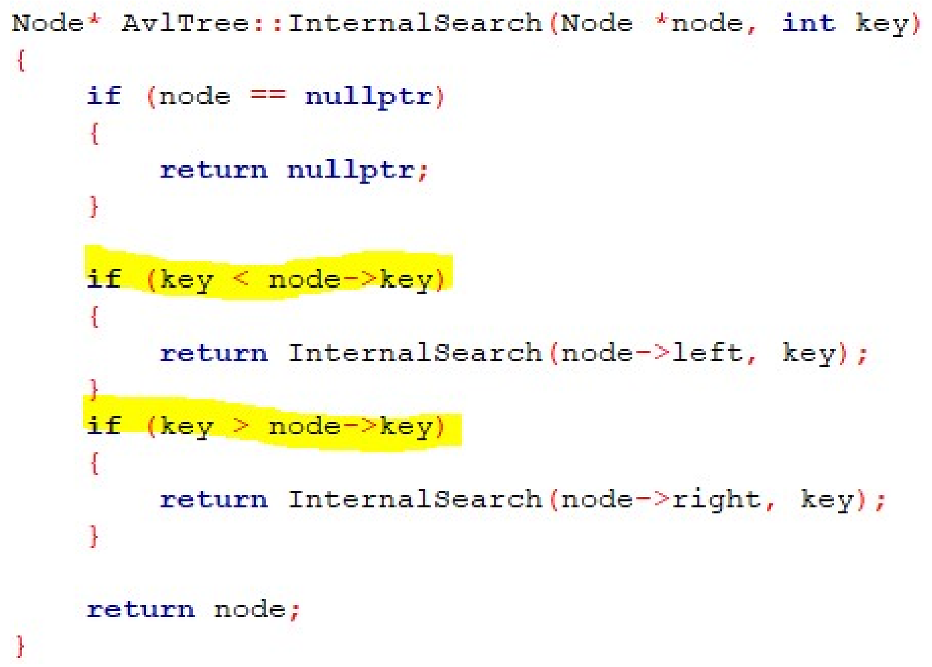
```
Node* AvlTree::RotateLR(Node *nd)
{
    Node *x = nd->left;
    nd->left = RotateL(x);
    return RotateR(nd);
}
```



```
Node* AvlTree::RotateRL(Node *nd)
{
    Node *x = nd->right;
    nd->right = RotateR(x);
    return RotateL(nd);
}
```

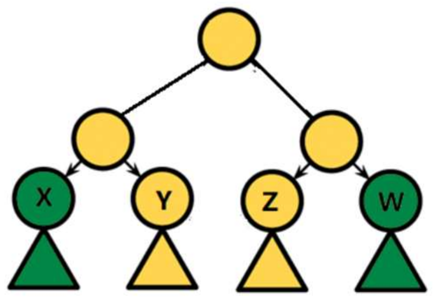


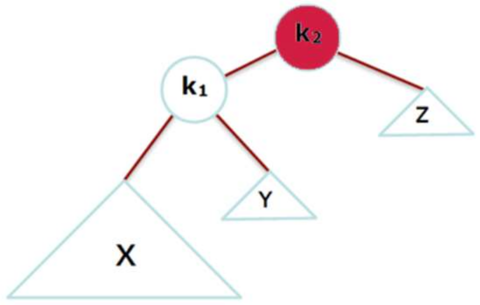
## 查找

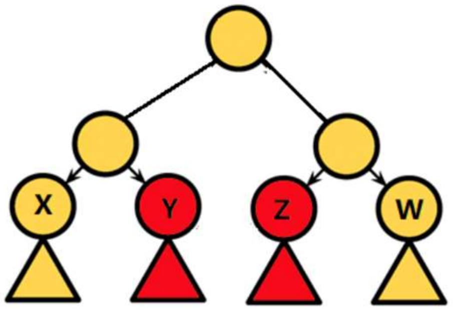


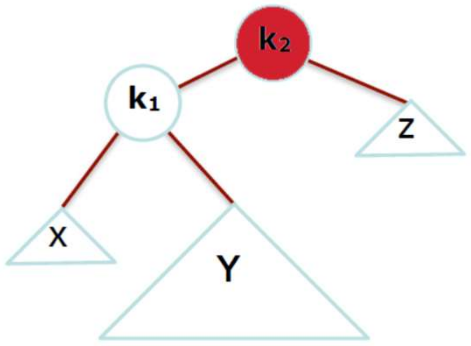
```
Node* AvlTree::InternalSearch(Node *node, int key)
{
    if (node == nullptr)
    {
        return nullptr;
    }

    if (key < node->key)
    {
        return InternalSearch(node->left, key);
    }
    if (key > node->key)
    {
        return InternalSearch(node->right, key);
    }

    return node;
}
```

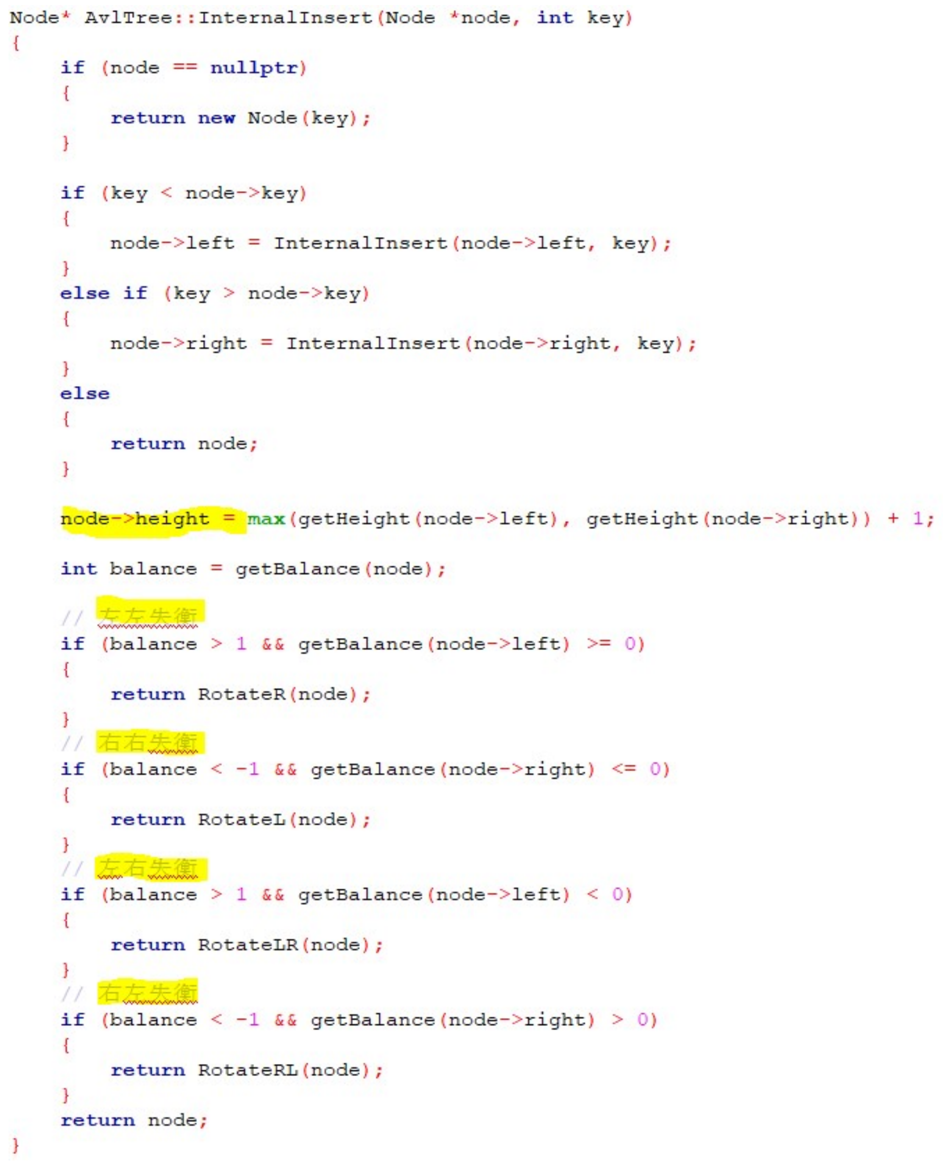
# 插入算法







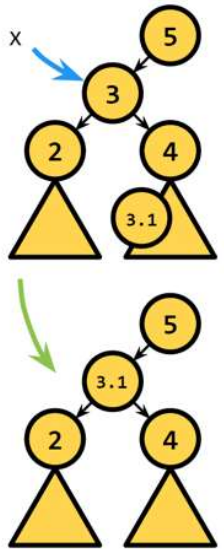




```
Node* AvlTree::InternalInsert(Node *node, int key)
{
    if (node == nullptr)
    {
        return new Node(key);
    }

    if (key < node->key)
    {
        node->left = InternalInsert(node->left, key);
    }
    else if (key > node->key)
    {
        node->right = InternalInsert(node->right, key);
    }
    else
    {
        return node;
    }

    node->height = max(getHeight(node->left), getHeight(node->right)) + 1;

    int balance = getBalance(node);

    // 左左失衡
    if (balance > 1 && getBalance(node->left) >= 0)
    {
        return RotateR(node);
    }
    // 右右失衡
    if (balance < -1 && getBalance(node->right) <= 0)
    {
        return RotateL(node);
    }
    // 左右失衡
    if (balance > 1 && getBalance(node->left) < 0)
    {
        return RotateLR(node);
    }
    // 右左失衡
    if (balance < -1 && getBalance(node->right) > 0)
    {
        return RotateRL(node);
    }
    return node;
}
```

## 删除算法

```
    node->height = max(getHeight(node->left), getHeight(node->right)) + 1;

    int balance = getBalance(node);

    // 左左失衡
    if (balance > 1 && getBalance(node->left) >= 0)
    {
        return RotateR(node);
    }
    // 右右失衡
    if (balance < -1 && getBalance(node->right) <= 0)
    {
        return RotateL(node);
    }
    // 左右失衡
    if (balance > 1 && getBalance(node->left) < 0)
    {
        return RotateLR(node);
    }
    // 右左失衡
    if (balance < -1 && getBalance(node->right) > 0)
    {
        return RotateRL(node);
    }

    return node;
}
```



```
Node* AvlTree::InternalDelete(Node *node, int key)
{
    if (node == nullptr)
    {
        return node;
    }

    if (key < node->key)
    {
        node->left = InternalDelete(node->left, key);
    }
    else if (key > node->key)
    {
        node->right = InternalDelete(node->right, key);
    }
    else
    {
        if (node->left && node->right)
        {
            // 找到后继结点
            Node * x = node->right;
            while (x->left)
            {
                x = x->left;
            }

            // 后继直接复制
            node->key = x->key;

            // 转化为删除后继
            node->right = InternalDelete(node->right, x->key);
        }
        else
        {
            Node * t = node;
            node = node->left ? node->left : node->right;
            delete t;
            if (node == nullptr)
            {
                return nullptr;
            }
        }
    }
```

## 树的结构/形状

- 把二叉树左右翻转（左子树和右子树互换）
- 判断两棵树是否相等
- 判断是不是子树
- 判断二叉树是不是镜像结构
- 验证是不是二叉查找树
- 输出树的轮廓

## 树的高度/深度/节点个数/距离

- 树中节点的个数
- 树的最大宽度
- 节点的高度/深度
- 判断树是不是按高度平衡的
- 树中最深的叶节点
- 树中最深的左叶节点
- 树中两个节点间的最短路径
- 输出离根节点距离位k的节点

## 树的构造

- 把有序数组转化为二叉查找树
- 按前序/中序恢复二叉树
- 按中序/后序恢复二叉树
- 把二叉树转化为链表（inplace）
- 把二叉树转化为双向链表
- 给定[1,n]，可以建立多少种不同的BST
- 合并两个二叉查找树是

## 树的查找

- 前驱/后继，包括线索化树
- 给定BST的一个节点，求其中序的前驱和后继
- 前序/中序/后序遍历
- 把同一层的节点连接起来（不用额外空间）
- BST中第K小的数
- 输出BST中前K个数，或者和
- 打印根节点到叶节点最长的路径
- 二叉树中两个节点的LCA
- BST中查找给定范围内的节点[x, y]
- 找BST中给定值的Floor and Ceil
- 一个二叉查找树中有两个元素错误的互换了，怎么纠正

## 提示

- 二叉树最深的叶节点
    - void DeepLeaf(TreeNode * root, int lvl, int &depth)
- 二叉树最深的左叶节点
    - void DeepLeftLeaf(TreeNode *root, bool isleft, int lvl, int &depth)
- 二叉树所有根节点到叶节点路径
    - Void AllPaths(TreeNode *root, string pathTillNow, vector<string> &results)
- 二叉树任意两节点路径和的最大值

```
int maxPathDown(TreeNode* node, int &maxValue) {
if (node == nullptr) return 0;
    int left = max(0, maxPathDown(node->left, maxValue));
    int right = max(0, maxPathDown(node->right, maxValue));
    maxValue = max(maxValue, left + right + node->val);
    return max(left, right) + node->val;
}
```

## 提示

- 验证BST
  - Bool isValidBST(TreeNode* root, TreeNode* minNode, TreeNode* maxNode)
  - bool isValidBST(TreeNode* root, long min, long max)
- 恢复BST，如果其中两个节点被错误的交换了
  - 中序遍历，设置first， second，firstsuccessor
- BST找第二大节点
  - 中序遍历，设置count
- 二叉树中的LCA
  - 根节点路径
- BST中如何换key
  - Delete+insert

## Q&A

Thanks!