数据结构与算法

DATA STRUCTURE

第二十讲 树习题课

胡浩栋

信息管理与工程学院

2017 - 2018 第一学期

课堂内容

- 作业回顾
- 习题

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;
}
```

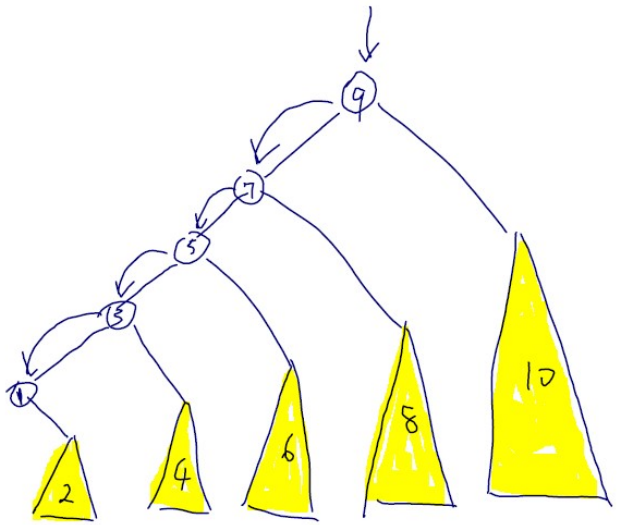
```
Node * BinaryTree::CopyTree(const Node *root)
{
    if (root == nullptr)
    {
        return nullptr;
    }
    Node * copyRoot = new Node(root->key);
    copyRoot->left = CopyTree(root->left);
    copyRoot->right = CopyTree(root->right);

    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;
}
```

从前序/中序恢复

```
BinaryTree::BinaryTree(const vector<int> &preorder, const vector<int> &inorder)
{
    if (preorder.size() != inorder.size())
    {
        return;
    }

    _pRoot = BuildTree(preorder, 0, inorder, 0, inorder.size()-1);
}

Node* BinaryTree::BuildTree(const vector<int> &preorder, int pre,
                            const vector<int> &inorder, int lh, int rh)
{
    if (lh > rh)
    {
        return nullptr;
    }
    int val = preorder.at(pre);
    Node *root = new Node(val);

    int index = lh;
    for (; index <= rh; index++)
    {
        if (inorder.at(index) == val)
        {
            break;
        }
    }
    assert(index <= rh);

    root->left = BuildTree(preorder, pre+1, inorder, lh, index - 1);
    root->right = BuildTree(preorder, pre+index-lh+1, inorder, index + 1, rh);
    return root;
}
```

迭代实现




```
Node * BinaryTree::BuildTree_Iterative(const vector<int> &preorder, const vector<int> &inorder)
{
    assert(preorder.size() == inorder.size());

    // pre index always points to node will be created next
    // in index always points to leftmost child of right child of stack top when push, or stack top itself
    int pre = 0, ind = 0;
    Node *root = new Node(preorder.at(pre++));

    stack<Node *> stk;
    stk.push(root);

    while (true)
    {
        if (inorder[ind] == stk.top()->key)
        {
            Node * top = stk.top();
            stk.pop();
            ind++;

            // all done
            if (ind == inorder.size())
            {
                break;
            }
            // the case that right child is empty
            if (!stk.empty() && inorder[ind] == stk.top()->key)
            {
                continue;
            }
            // set right child and push to stk.
            // Then process right subtree like a new one
            top->right = new Node(preorder.at(pre++));
            stk.push(top->right);
        }
        else
        {
            // Keep pushing left child to stk until leftmost child.
            Node * nd = new Node(preorder.at(pre++));
            stk.top()->left = nd;
            stk.push(nd);
        }
    }

    return root;
}
```

Z/己字形输出

```
void BinaryTree::LevelOrder()
{
    queue<const Node *> treeQueue;

    if (_pRoot != nullptr)
    {
        treeQueue.push(_pRoot);
    }

    int size = 1;
    while (!treeQueue.empty())
    {
        const Node * node = treeQueue.front();
        treeQueue.pop();
        if (node)
        {
            cout << node->key << " ";
            treeQueue.push(node->left);
            treeQueue.push(node->right);
        }
        else
        {
            cout << "$" << " ";
        }

        if (--size == 0)
        {
            cout << endl;
            size = treeQueue.size();
        }
    }
}
```

```
vector<int> preorder = {7,10,4,3,1,2,8,11};
vector<int> inorder = {4,10,3,1,7,11,8,2};
```

```
7
10 2
4 3 8 $
$ $ $ 1 11 $
$ $ $ $

7 2 10 4 3 8 11 1
```

```
void BinaryTree::LevelOrder_Backwards()
{
    queue<const Node *> treeQueue;

    if (_pRoot != nullptr)
    {
        treeQueue.push(_pRoot);
    }

    int depth = 0;
    while (!treeQueue.empty())
    {
        stack<const Node *> stk;
        int size = treeQueue.size();
        while (size-- > 0)
        {
            const Node * node = treeQueue.front();
            treeQueue.pop();
            if (!node)
            {
                continue;
            }

            cout << node->key << " ";

            if (depth % 2 == 1)
            {
                stk.push(node->right);
                stk.push(node->left);
            }
            else
            {
                stk.push(node->left);
                stk.push(node->right);
            }
        }

        while (!stk.empty())
        {
            const Node * node = stk.top();
            stk.pop();
            treeQueue.push(node);
        }
        depth++;
    }
}
```

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
- Kth节点BST，或者找第二大节点
  - 中序遍历，设置count
- BST中的LCA
  - 二分查找
- 二叉树中的LCA
  - 根节点路径
- BST中如何换key
  - Delete+insert

检查树平衡

1. Top down

```
int depth (TreeNode *root)
{
    if (!root) return 0;
    return max (depth(root->left), depth (root->right)) + 1;
}

bool isBalanced (TreeNode *root)
{
    if (!root) return true;

    int left=depth(root->left);
    int right=depth(root->right);

    return abs(left - right) <= 1 &&
           isBalanced(root->left) &&
           isBalanced(root->right);
}
```

2. bottom up

```
int dfsHeight (TreeNode *root)
{
    if (!root) return 0;

    int leftHeight = dfsHeight(root->left);
    if (leftHeight == -1) return -1;

    int rightHeight = dfsHeight(root->right);
    if (rightHeight == -1) return -1;

    if (abs(leftHeight - rightHeight) > 1)  return -1;
    return max (leftHeight, rightHeight) + 1;
}
bool isBalanced(TreeNode *root)
{
    return dfsHeight (root) != -1;
}
```

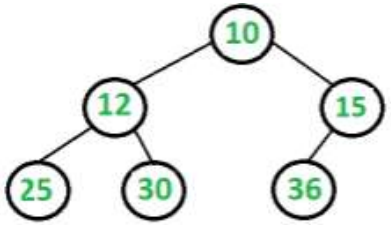
树按层连接

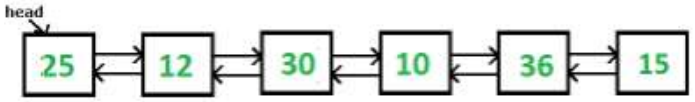
```
Node* getNextRight(Node *nd)
{
    Node *temp = nd->nextRight;

    /* Traverse nodes at nd's level and find and
       the first node's first child */
    while(temp)
    {
        if(temp->left)
        {
            return temp->left;
        }
        if(temp->right)
        {
            return temp->right;
        }
        temp = temp->nextRight;
    }

    // All the nodes at nd's level are leaf nodes
    return nullptr;
}
```

```
void connectLvl(Node* nd)
{
    // Base case
    if (!nd)
    {
        return;
    }

    /* Before setting nextRight of left and right children, set nextRight
    of children of other nodes at same level (because we can access
    children of other nodes using p's nextRight only) */
    if (nd->nextRight)
    {
       connectLvl(nd->nextRight);
    }

    /* Set the nextRight pointer for p's left child */
    if (nd->left)
    {
       if (nd->right)
       {
           nd->left->nextRight = nd->right;
           nd->right->nextRight = getNextRight(nd);
       }
       else
       {
           nd->left->nextRight = getNextRight(p);
       }
       /* Recursively call for next level nodes.  Note that we call only
       for left child. The call for left child will call for right child */
       connectLvl(nd->left);
    }

    /* If left child is NULL then first node of next level will either be
      p->right or getNextRight(p) */
    else if (nd->right)
    {
        nd->right->nextRight = getNextRight(nd);
        connectRecur(nd->right);
    }
    else
    {
       connectRecur(getNextRight(nd));
    }
}
```

树转换成双向链表

```
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);
}
```





树的轮廓

- 输出二叉树的轮廓
  - 左边界（从上到下）
  - 右边界（从下到上）
  - 叶节点（左子树，右子树）
  - 避免重复输出

```
void printLeaves(struct node* root)
{
    if (!root)
    {
        return;
    }

    printLeaves(root->left);

    // Print it if it is a leaf node
    if (!(root->left) && !(root->right))
    {
        cout << root->data << " ";
    }

    printLeaves(root->right);
}


void printBoundary (struct node* root)
{
    if (!root)
    {
        return;
    }

    cout << root->data << " ";

    // Print the left boundary in top-down manner.
    printBoundaryLeft(root->left);

    // Print all leaf nodes
    printLeaves(root->left);
    printLeaves(root->right);

    // Print the right boundary in bottom-up manner
    printBoundaryRight(root->right);
}
```

左右边界

```
void printBoundaryLeft(struct node* root)
{
    if (!root)
    {
        return;
    }

    if (root->left)
    {
        // to ensure top down order, print the node
        // before calling itself for left subtree
        cout << root->data << " ";
        printBoundaryLeft(root->left);
    }
    else if(root->right)
    {
        cout << root->data << " ";
        printBoundaryLeft(root->right);
    }
    // do nothing if it is a leaf node, this way we avoid
    // duplicates in output
}
```

```
void printBoundaryRight(struct node* root)
{
    if (!root)
    {
        return;
    }

    if (root->right)
    {
        // to ensure bottom up order, first call for right
        //  subtree, then print this node
        printBoundaryRight(root->right);
        cout << root->data << " ";
    }
    else if (root->left)
    {
        printBoundaryRight(root->left);
        cout << root->data << " ";
    }
   // do nothing if it is a leaf node, this way we avoid
   // duplicates in output
}
```

递归回顾

- 一维递归
- 二维递归
  - 整体理解（数学归纳法）
  - 局部理解（刨解成迭代）
  - Euler路径就是访问顺序

Heap应用

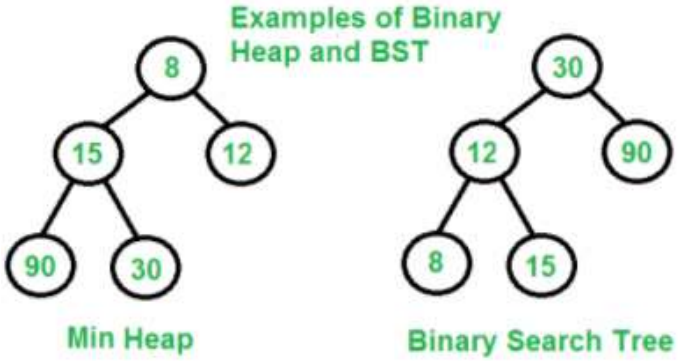
- 数列中求k-th大的数
  - 排序，数数到第k个
  - 用最大堆，连续取k次
  - 前k个数的最小堆，逐个添加后面n-k个数，如果比堆顶最小的元素要大的话。每加一个，同时去掉最小一个。
  - 修改一下BST，每个节点添加子树中节点个数
- Online steam里找最大的k个数
- 合并K个有序数组

PriorityQueue: heap vs BST

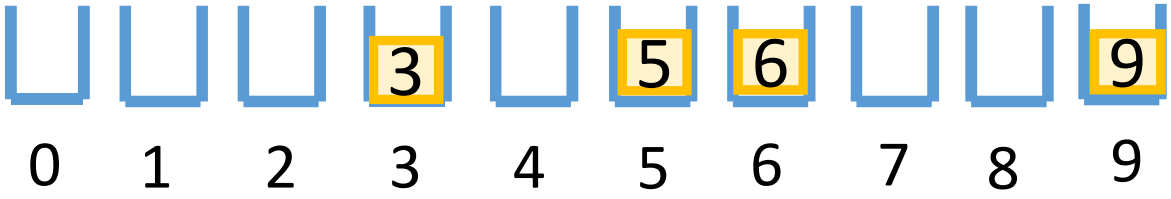
- O(1) 取到优先级最高元素
- O(logn) 插入新元素
- O(logn) 移去优先级最高元素
- O(logn) 改变元素的优先级

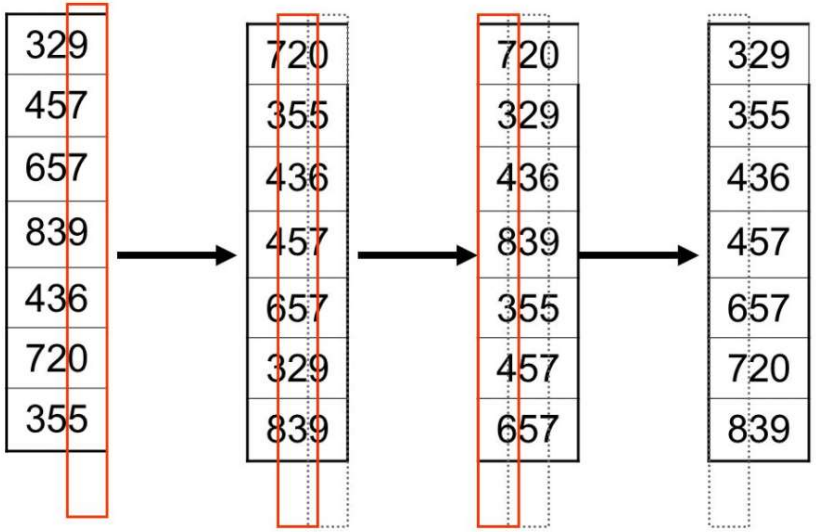
- Heap用数组，cache locality
- 复杂度一样，但实际会快些，complete tree
- 建Heap只要O(n)
- Heap不用存子节点指针，容易实现
- Fibonacci heap可以在O(1)时间内插入/decrease key

- BST查找快
- BST可以中序遍历来排序
- BST可以在O(logn)时间内找到kth元素



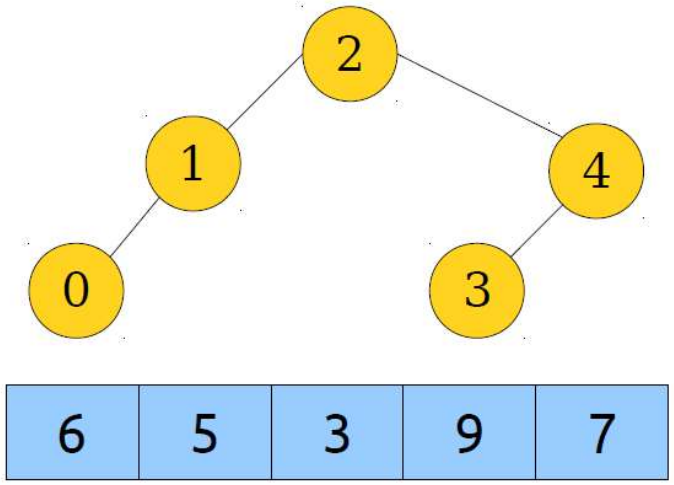
Bucket sort和radix sort





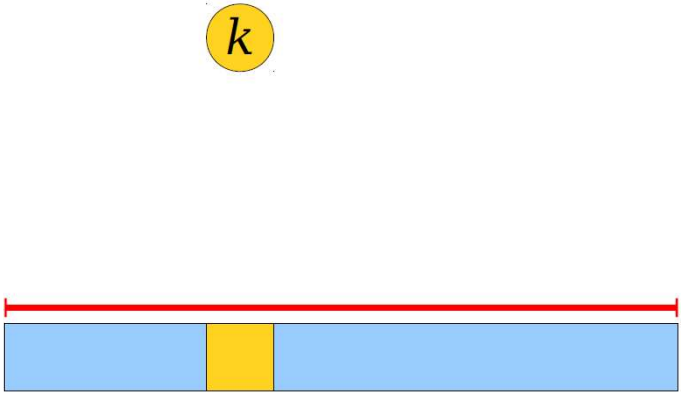
Cartesian tree

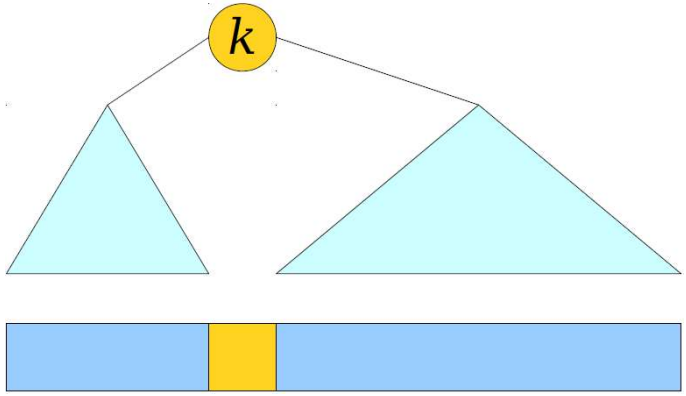
- 一个从数组引申出来的二叉树，满足heap性质
- 数组的RMQ对应数上的LCA



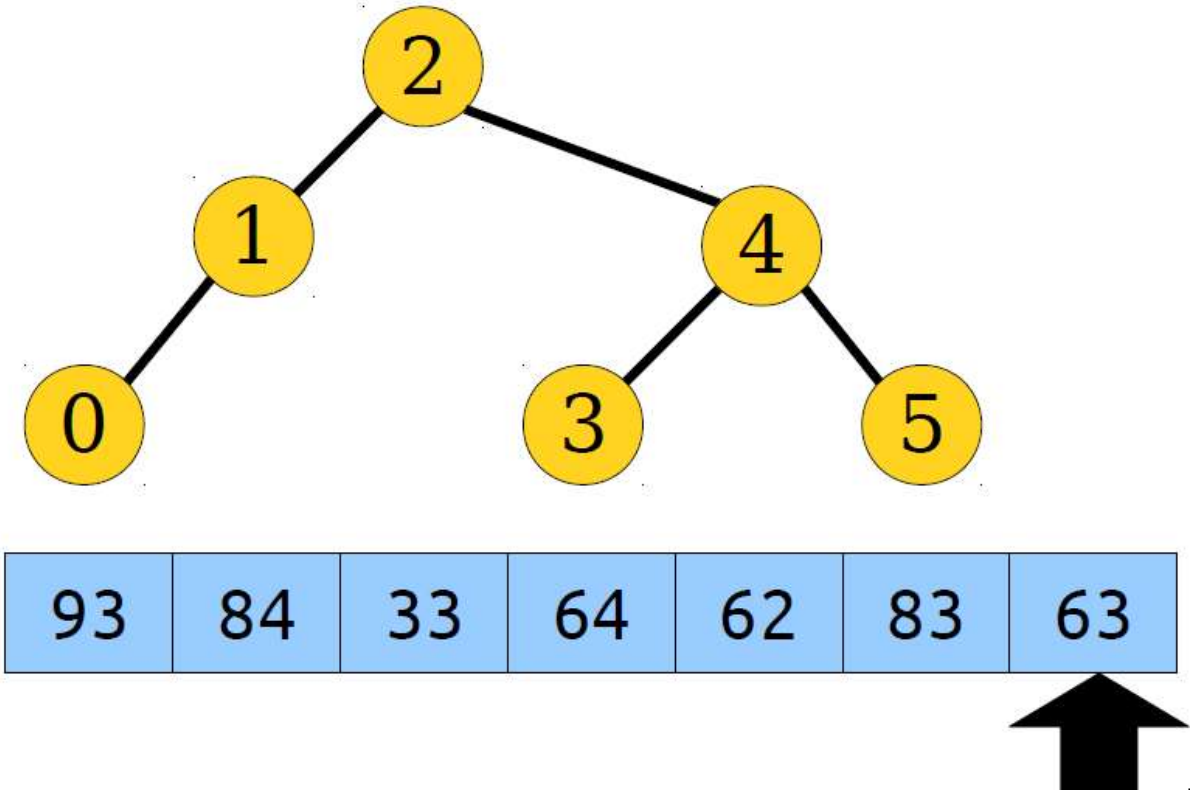
Cartesian tree

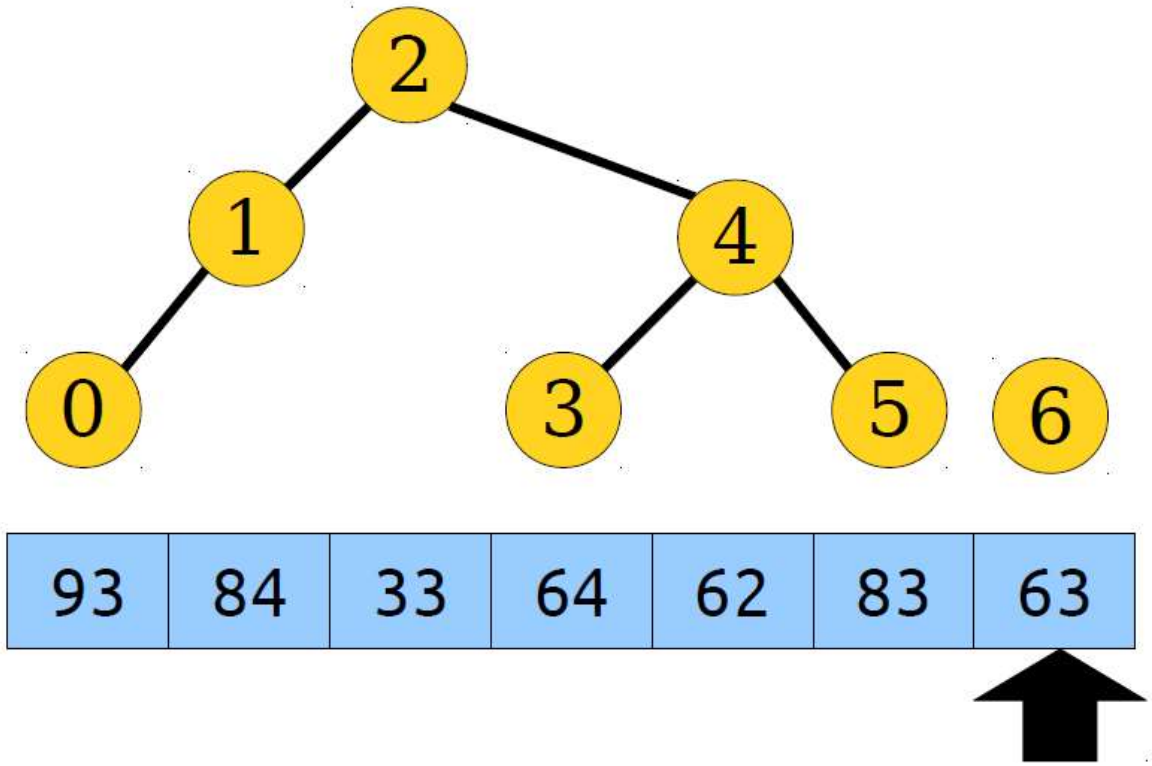
- 每一层建立需要扫描一遍数组
- 类似quicksort，最差可能O(n2)

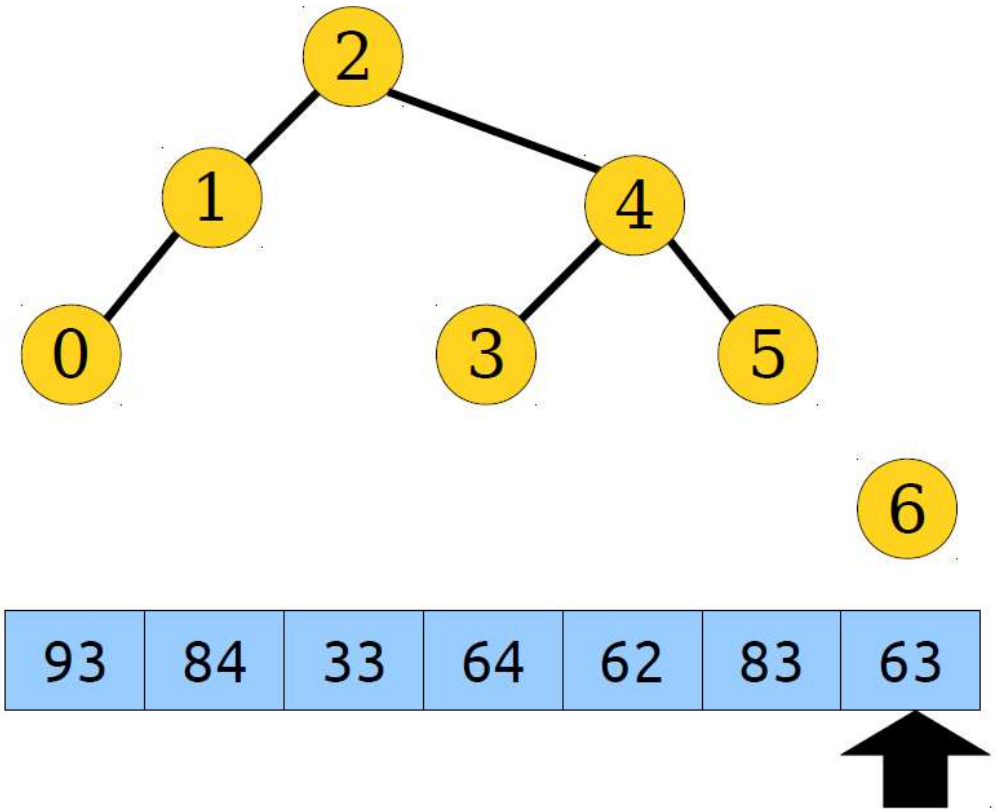


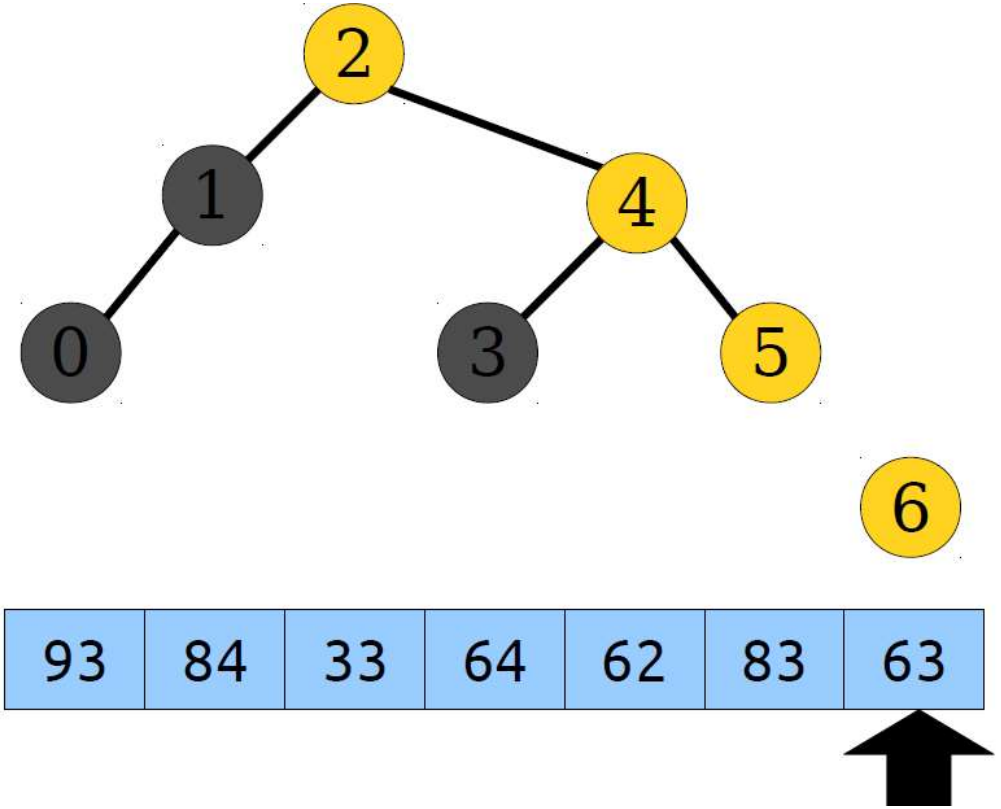


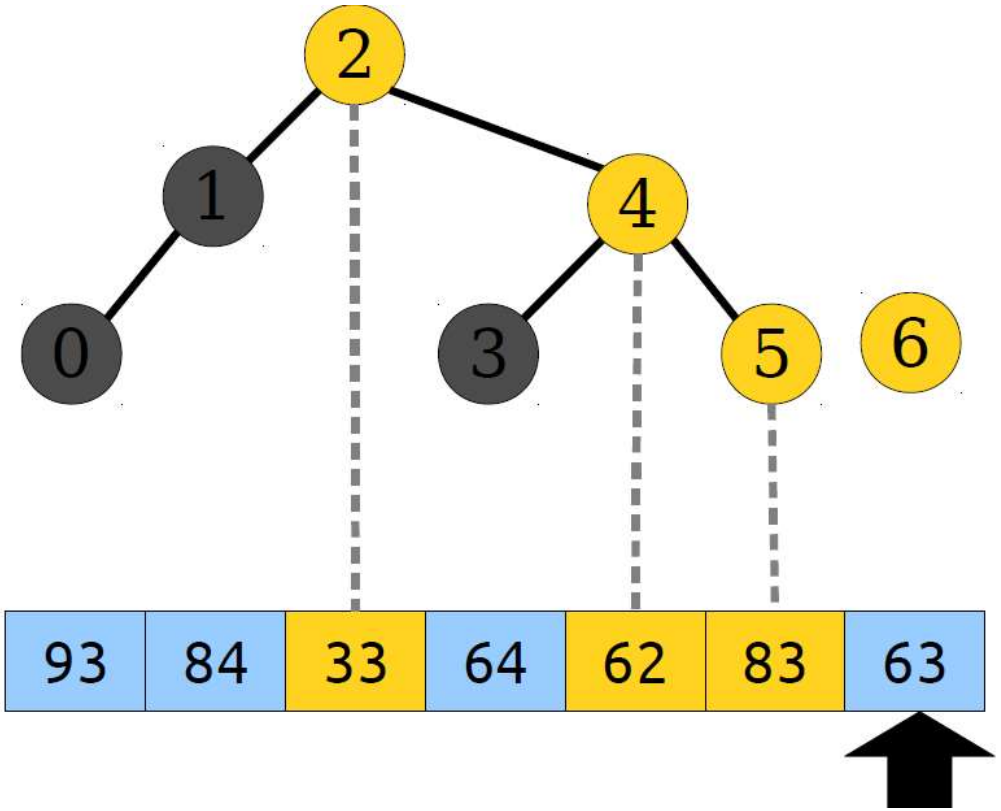
观察1



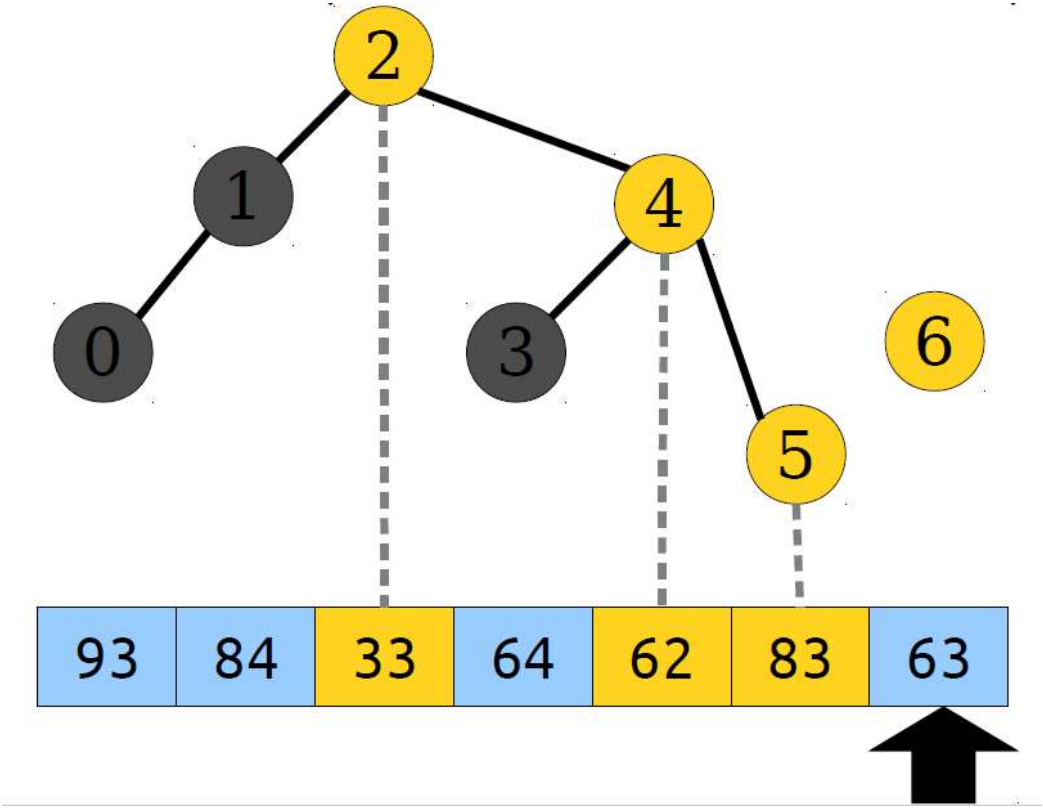

2
1
4
0
3
5
6
93
84
33
64
62
83
63

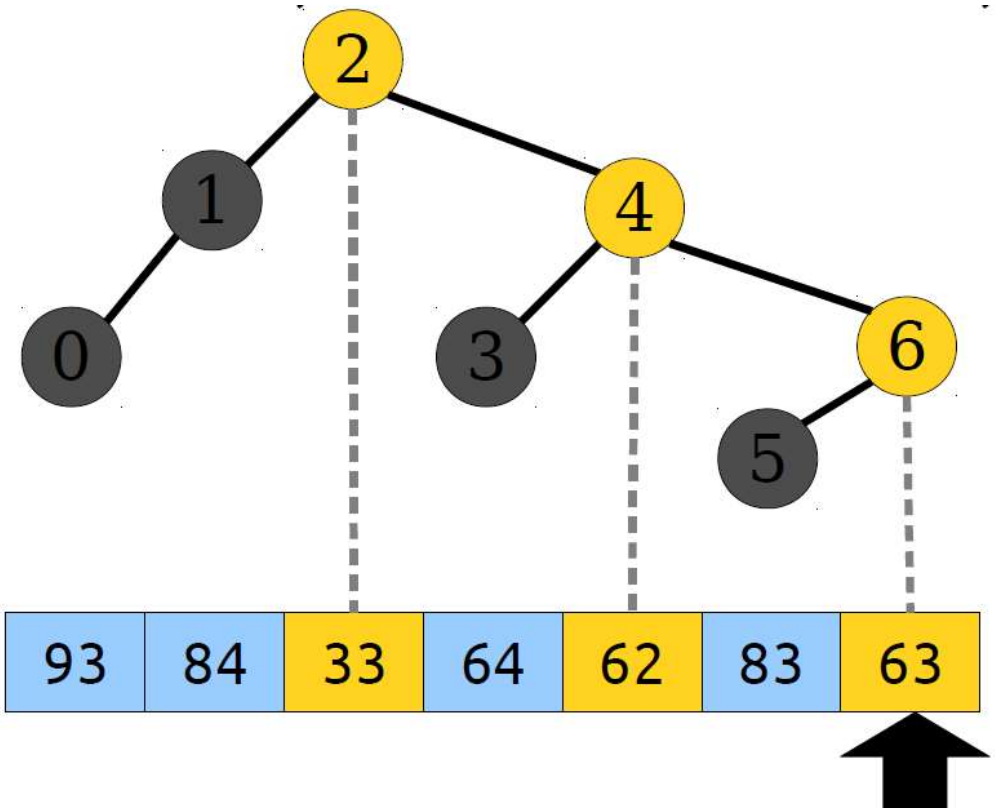

2
1
4
0
3
5
6
93
84
33
64
62
83
63

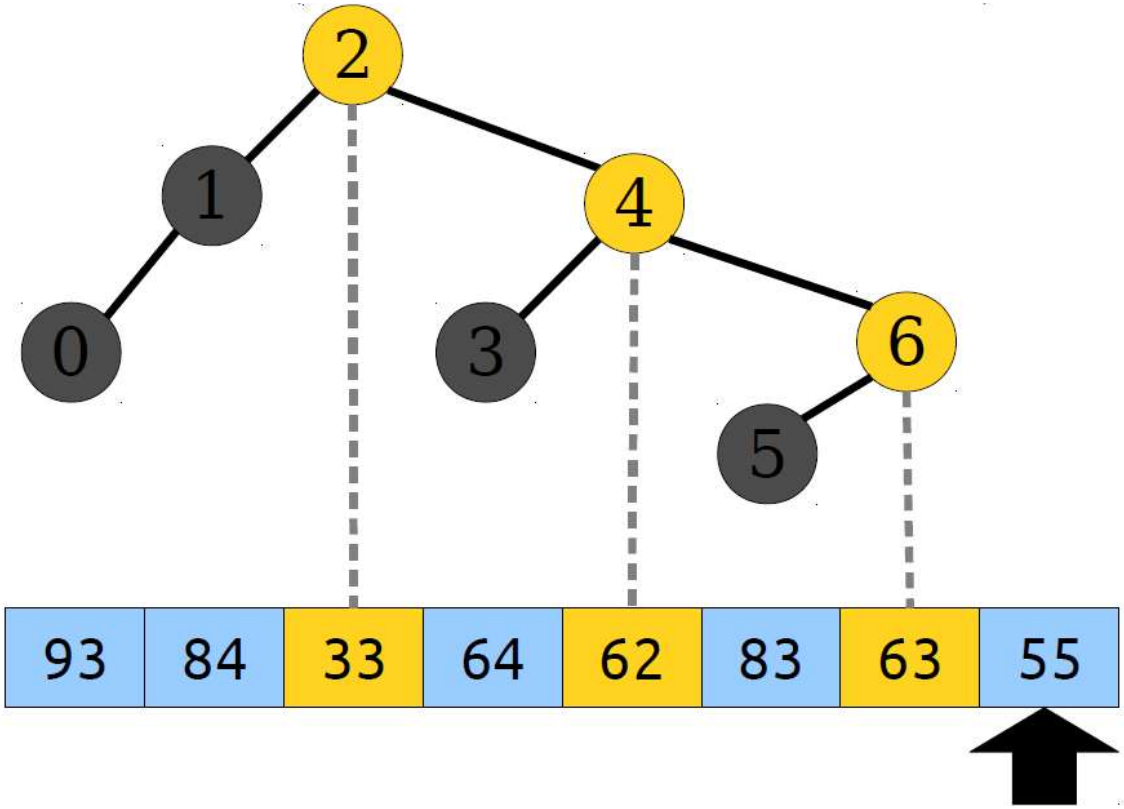

2
1
4
0
3
5
6
93
84
33
64
62
83
63

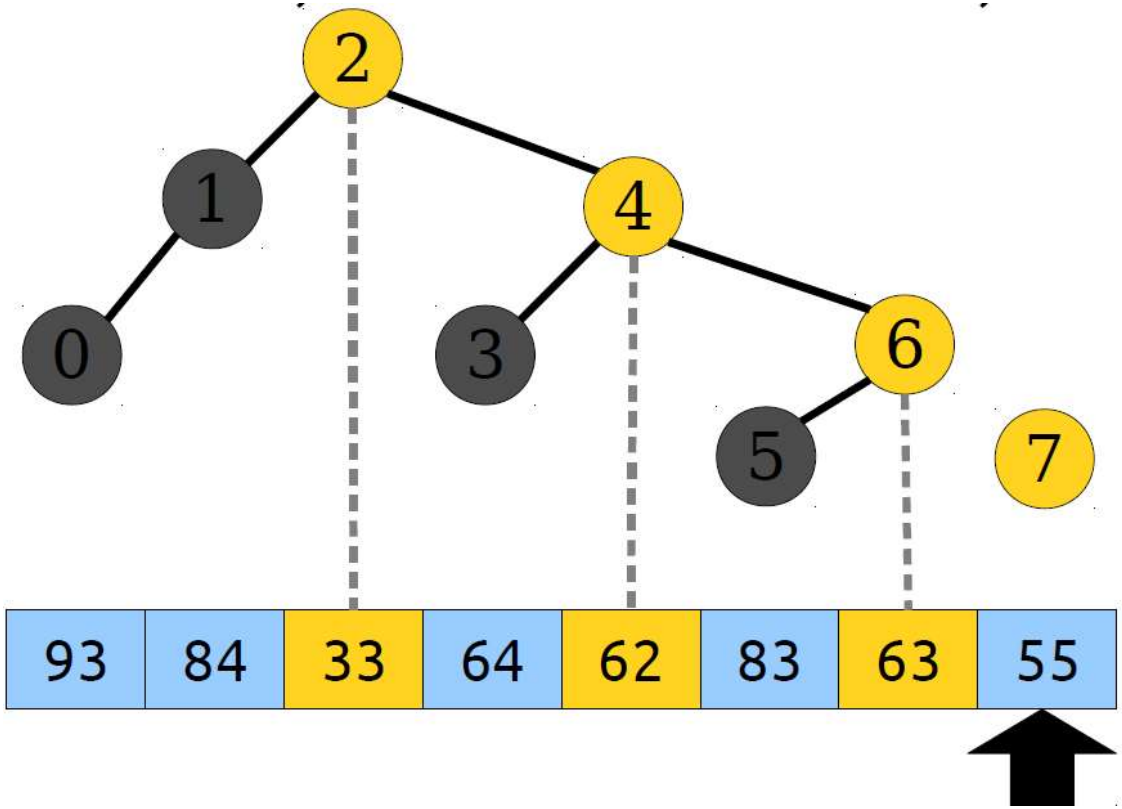

2
1
4
0
3
5
6
93
84
33
64
62
83
63

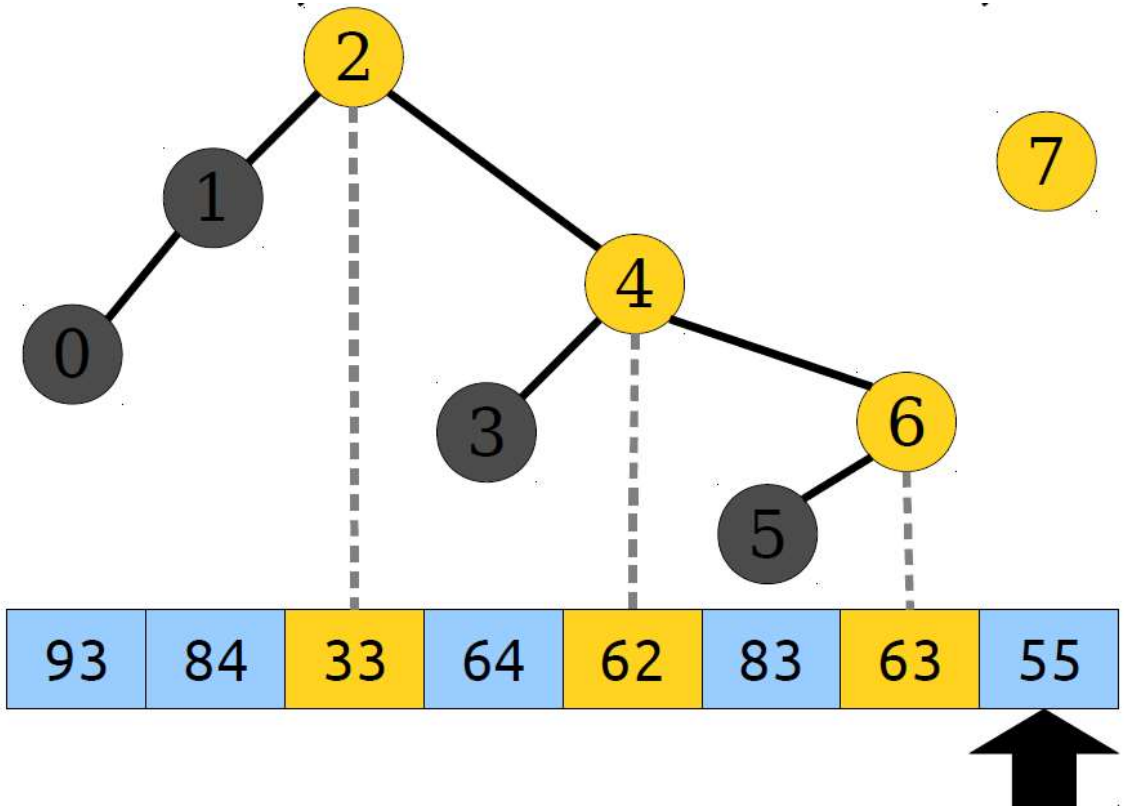
Cartesian tree

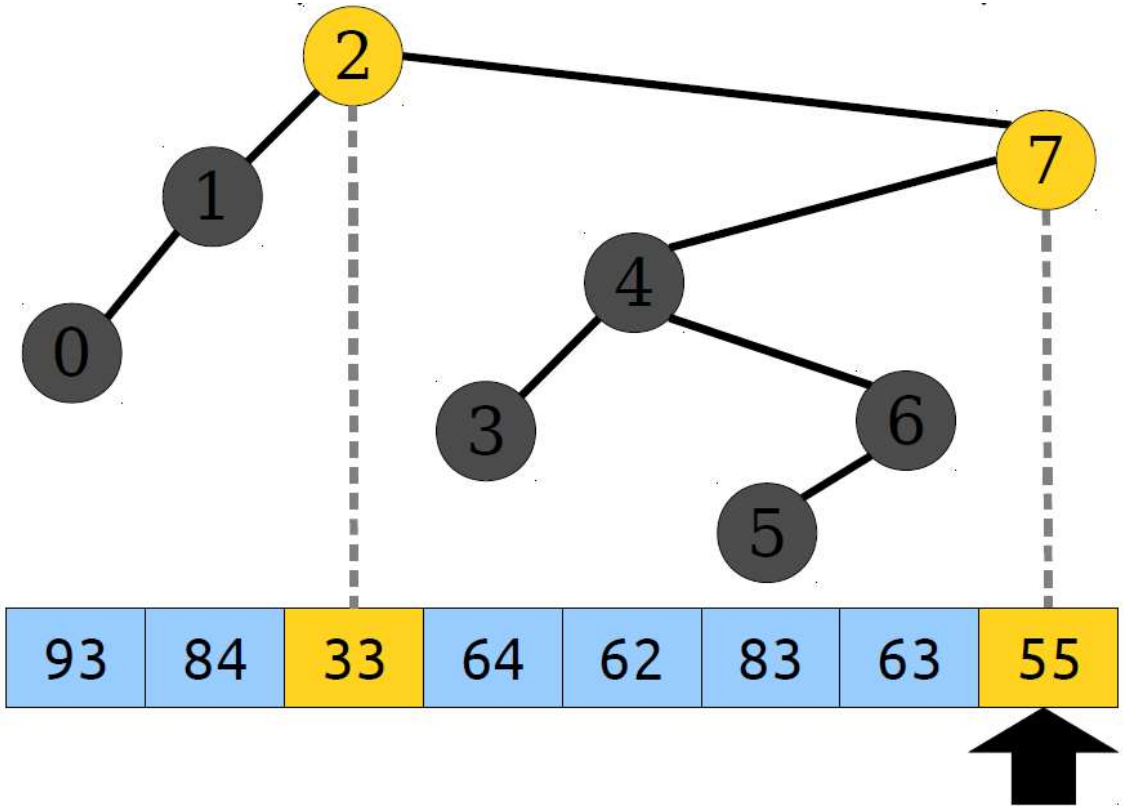



2
1
4
0
3
6
5
93
84
33
64
62
83
63


2
1
4
0
3
6
5
93
84
33
64
62
83
63
55

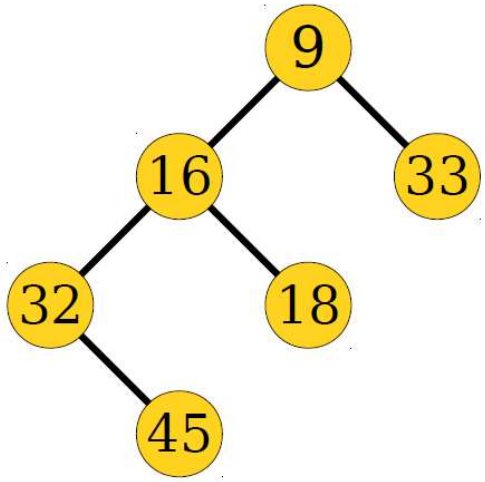

2
1
4
0
3
6
5
7
93
84
33
64
62
83
63
55


2
7
1
4
0
3
6
5
93
84
33
64
62
83
63
55


2
7
1
4
0
3
6
5
93
84
33
64
62
83
63
55

算法

- 维护stack，保存右边界节点
- 插入新节点时，
  - 如果栈顶元素比较大，就一直pop
  - 所以要么栈空了，要么栈顶元素比新节点要小
  - 最后一个pop的元素作为新节点的左子节点（如果没有pop，左子树为空）
  - 新节点的父节点是栈顶的元素（作为栈顶的右子节点），可以是空
  - 新节点压入stack



数据结构库

建立自己的库

- 库library是一些代码的集合，方便在不同的程序里调用。一般包括两部分
  - 头文件，用来给别的程序include
  - 已经编译好的二进制代码，实现了相应函数功能

- 库文件是预先编译好的，因为
  - 库文件比较少变动，没必要每次重新编译
  - 二进制代码防止别人看到，修改源文件
- 比如C++ Standard Library,
  - vector, list, stack, queue

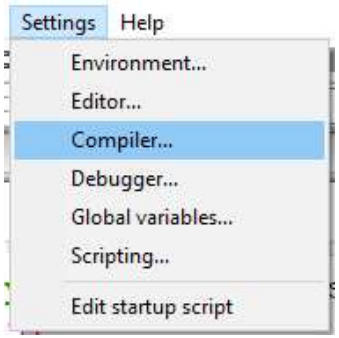
库的种类

- Static library
  - 直接链接到你的程序代码，和直接编译源文件没什么区别
  - Windows里后缀名为.lib，在linux里后缀名叫.a （archive）
  - 缺点是它变成你程序的一部分，费空间，而且版本升级不方便
- Dynamic library
  - 动态库是在程序运行的时候加载的，也叫shared lib
  - Windows里后缀名为.dll，在linux里后缀名叫.so （shared object）
  - 好处是很多程序都可以共用一个库，而且版本升级方便
  - 坏处是程序要使用动态库，需要定义接口，然后动态加载

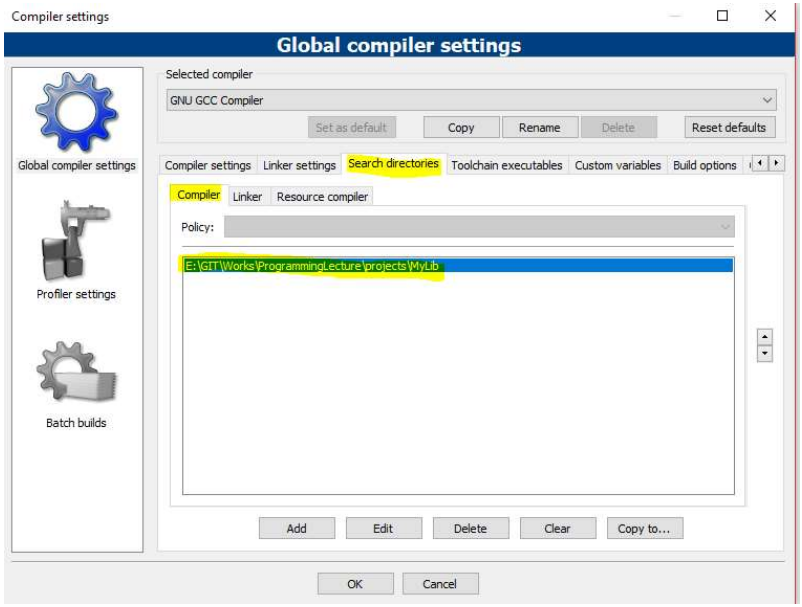
使用静态库的步骤

- 建立/下载库
- 编译器要知道库头文件在哪里（需要设置好路径）
- 设置好库文件（二进制代码）的路径
- 在程序源文件里#include<头文件>
- 在源文件里可以直接使用库函数
- 最后编译并运行

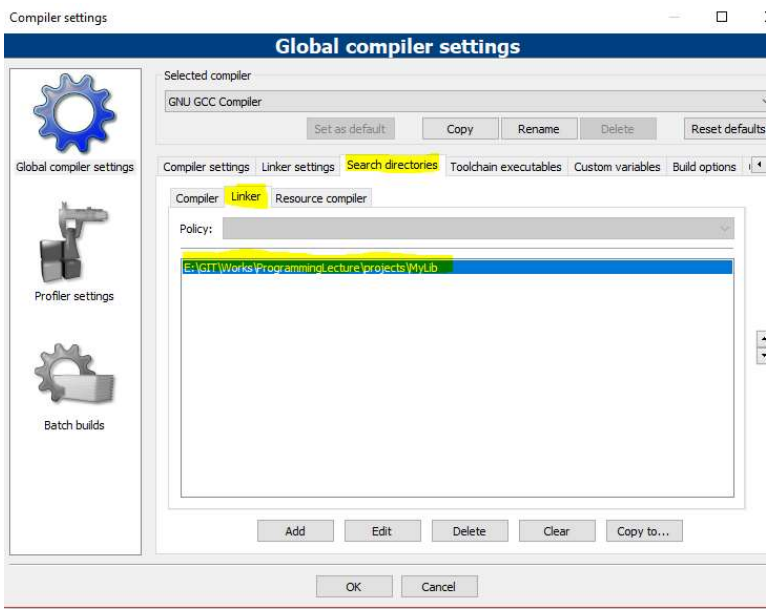
1. 设置库文件目录



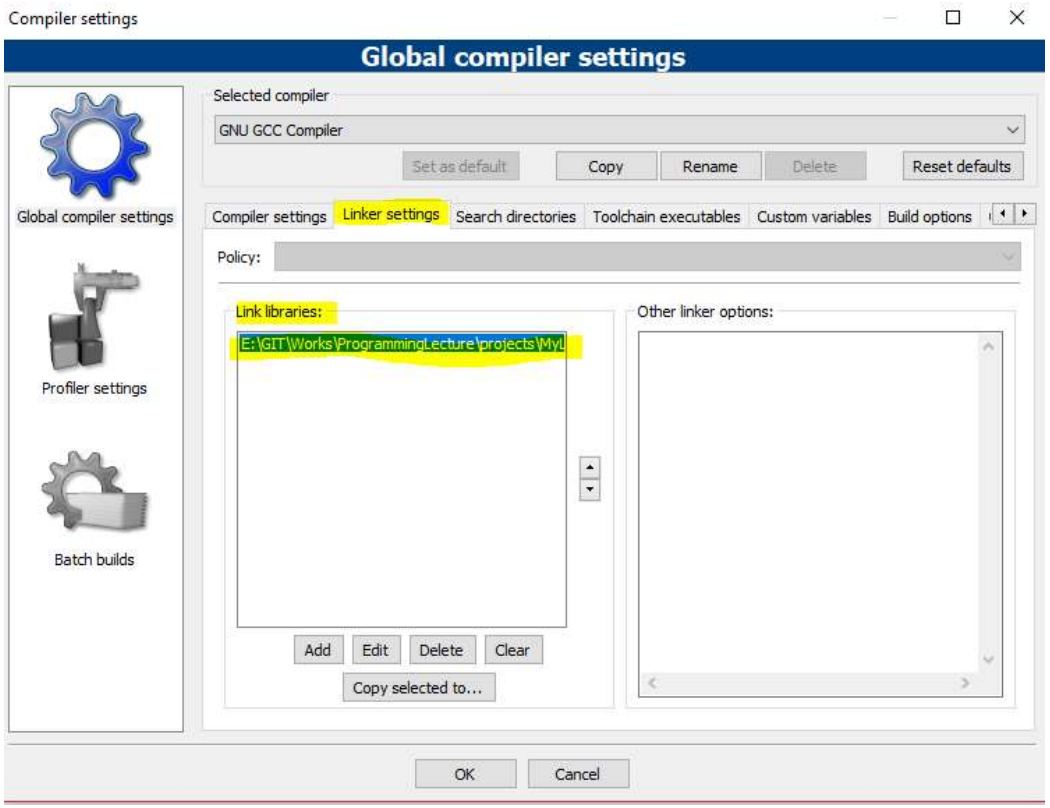
1. 设置头文件.h路径



2. 设置库文件.lib路径



2. 库文件linker设置

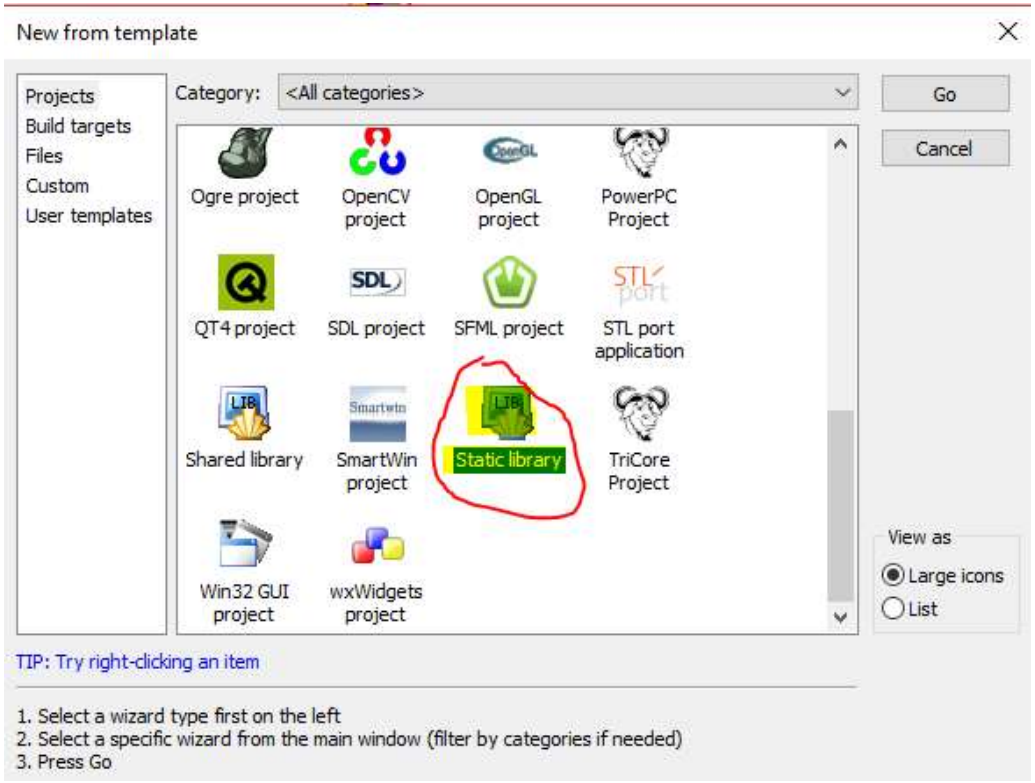


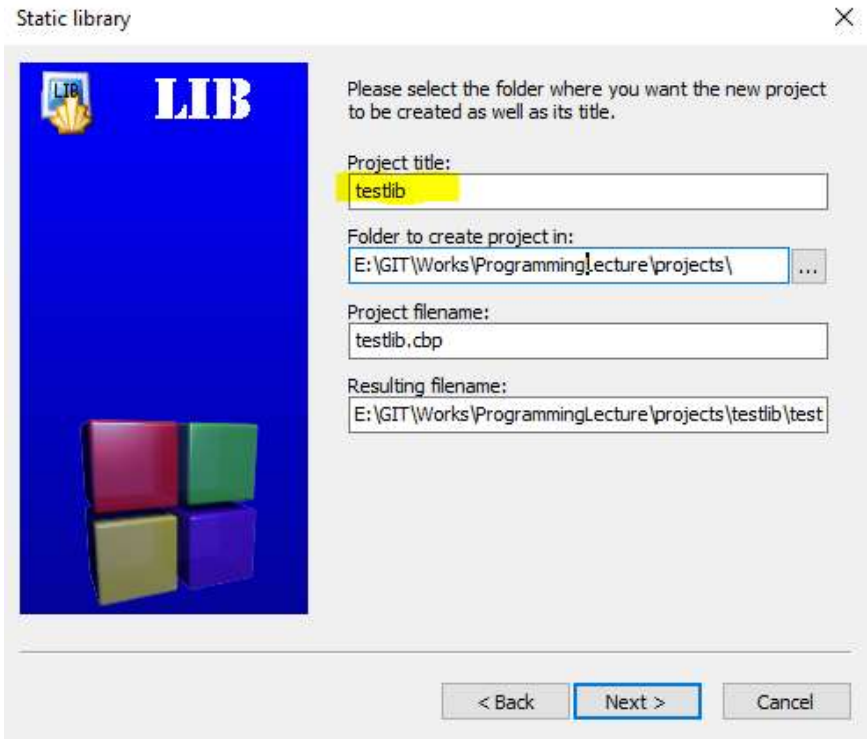
3. #include “myheader.h”

- 调用库类，库函数
- 编译并运行
- Done

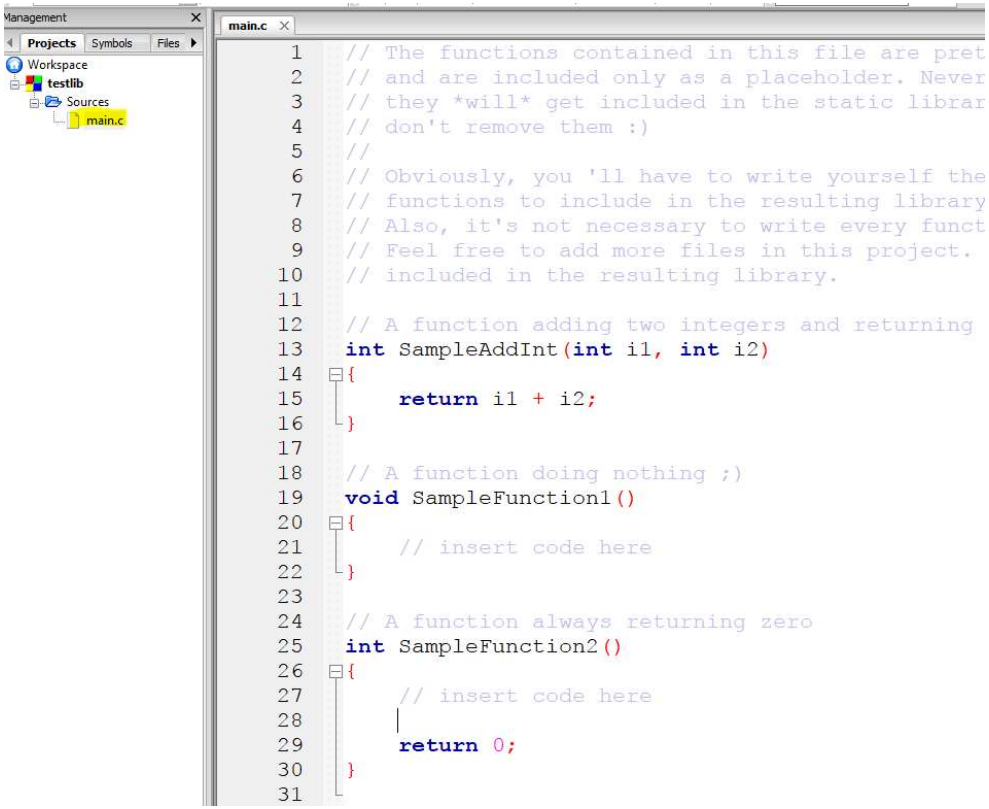
如何建立自己的静态库

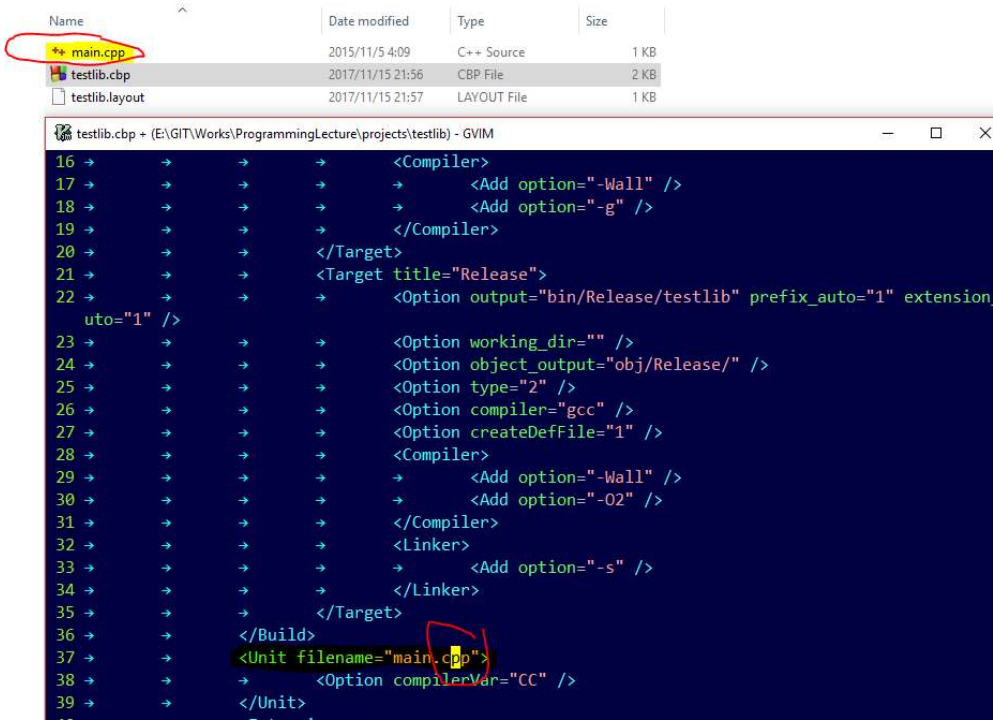
- 建立新的project， 选static library





如何建立自己的静态库


Management
Projects
Symbols
Files
Workspace
testlib
Sources
main.c
main.c
1 // The functions contained in this file are pret
2 // and are included only as a placeholder. Never
3 // they *will* get included in the static librar
4 // don't remove them :)
5 //
6 // Obviously, you 'll have to write yourself the
7 // functions to include in the resulting library
8 // Also, it's not necessary to write every funct
9 // Feel free to add more files in this project.
10 // included in the resulting library.
11
12 // A function adding two integers and returning
13 int SampleAddInt(int i1, int i2)
14 {
15 return i1 + i2;
16 }
17
18 // A function doing nothing ;)
19 void SampleFunction1()
20 {
21 // insert code here
22 }
23
24 // A function always returning zero
25 int SampleFunction2()
26 {
27 // insert code here
28
29 return 0;
30 }
31


Name
Date modified
Type
Size
main.cpp
2015/11/5 4:09
C++ Source
1 KB
testlib.cbp
2017/11/15 21:56
CBP File
2 KB
testlib.layout
2017/11/15 21:57
LAYOUT File
1 KB
testlib.cbp + (E:\GIT\Works\ProgrammingLecture\projects\testlib) - GVIM
16 <Compiler>
17 <Add option="-Wall" />
18 <Add option="-g" />
19 </Compiler>
20 </Target>
21 <Target title="Release">
22 <Option output="bin/Release/testlib" prefix_auto="1" extension_
uto="1" />
23 <Option working_dir="" />
24 <Option object_output="obj/Release/" />
25 <Option type="2" />
26 <Option compiler="gcc" />
27 <Option createDefFile="1" />
28 <Compiler>
29 <Add option="-Wall" />
30 <Add option="-O2" />
31 </Compiler>
32 <Linker>
33 <Add option="-s" />
34 </Linker>
35 </Target>
36 </Build>
37 <Unit filename="main.cpp">
38 <Option compilerVar="CC" />
39 </Unit>

如何建立自己的静态库

- 编译成功后会输出库文件*.a
- 把头文件和库文件.a放到共享目录
- 别的程序就可以编译并链接库文件成可执行文件

Git onboard

- **Git Basic**
  - Please first take some time to read for GIT basics: https://git-scm.com/doc, Chapter 1&2 is enough to begin
  - Short tutorial covering the basics: http://rogerdudler.github.io/git-guide/
- **Git global setup**
  - git config --global user.name “yourname"
  - git config --global user.email “youremail@mail.com"
- **Create a new repository**
  - git clone http://121.40.159.129/UserName/Tools.git
  - cd Tools
  - touch README.md
  - git add README.md
  - git commit -m "add README"
  - git push -u origin master

Git onboard

- **Existing folder**
  - cd existing_folder
  - git init
  - git remote add origin http://121.40.159.129/UserName/Tools.git
  - git add .
  - git commit
  - git push -u origin master
- **Existing Git repository**
  - cd existing_repo
  - git remote add origin http://121.40.159.129/UserName/Tools.git
  - git push -u origin --all
  - git push -u origin --tags

Q&A

Thanks!