哈尔滨工业大学（深圳）2021 年春《数据结构》

第二次作业 树型结构

学号	姓名	成绩
200110619	梁鑫嵘	

表1

（一）概念题

1. 在二叉树的顺序存储结构中，实际上隐含着双亲的信息，因此可和三叉链表（含有父链指针）对应。假设每个指针域占4个字节，每个信息域占k个字节。试问：对于一棵有 n 个结点的二叉树，在**顺序存储结构**中最后一个节点的下标为m，在什么条件下顺序存储结构比三叉链表更节省空间？

在顺序储存结构中最后一个节点下标为m，则占用空间mk个字节；

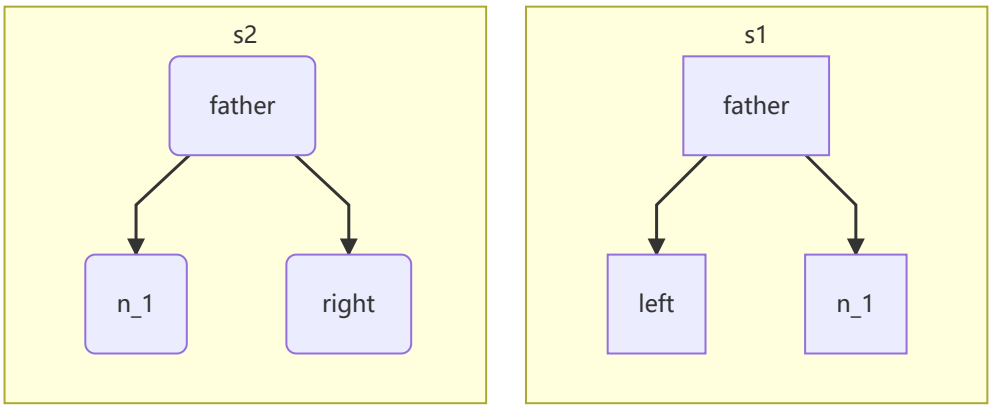
在三叉二叉树中有n个节点，每个节点占空间$(12+k)$个字节，总占用$n(12+k)$字节。

所以当$n(12+k)>mk$的时候，顺序储存比三叉链表更加省空间。

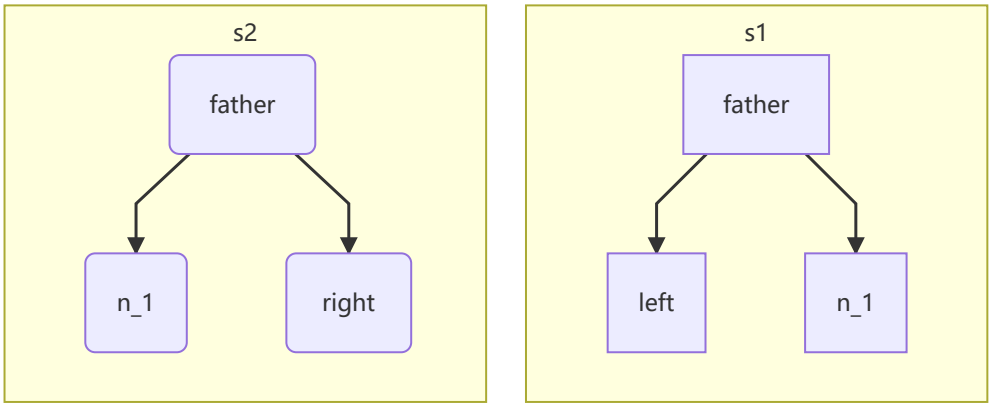
2. 对于二叉树T的两个结点n_1和n_2，我们应该选择二叉树T结点的前序、中序和后序中哪两个序列来判断结点n_1必定是结点n_2的祖先？ 试给出判断的方法。 （不需证明判断方法的正确性）

选择前序和后序遍历。方法如下：

1. 检查前序遍历序列，如n_2在n_1之前出现，如果是情况s_2则n_2是n_1祖先，如果是情况s_1说明n_2是n_1的祖先或者在n_1的左兄弟子树中；如果n_2在n_1之后出现，如果是情况$s2$则n_1必然是n_2的祖先，如果是情况s_1则n_2在n_1的左兄弟子树。



2. 检查后序遍历序列，如n_2在n_1之前出现，如果是情况s_1则n_1是n_2祖先，如果是情况s_2则在n_1的左兄弟子树中；如果n_2在n_1之后出现，如果是情况$s2$则n_1是n_2的祖先或者n_2是n_1的祖先，如果是情况s_1则n_2在n_1的左兄弟子树。



3. 一棵深度为H的满k叉树有如下性质：第H层上的结点都是叶子结点，其余各层上每个结点都有k棵非空子树。如果按层次顺序从1开始对全部结点编号，问：

   （1）各层的结点数目是多少？

   （2）编号为 p 的结点的父结点（若存在）的编号是多少？

   （3）编号为 p 的结点的第i个儿子结点（若存在）的编号是多少？

   （4）编号为 p 的结点有右兄弟的条件是什么？其右兄弟的编号是多少？

   1. 第h层节点数目为k^{h-1}
   2. $\lfloor \frac{p}{k} \rfloor$
   3. $kp+i$
   4. $p \bmod k \neq k-1$，编号为：$p+1\ if\ p \bmod k \neq k-1$

4. 已一棵度为k的树中有n_1个度为1的结点，n_2个度为2的结点，…，n_k个度为k的结点，问该树中有多少个叶子结点(n_0)？

   节点数 = 分支数目 + 1，$\Sigma_{i=0}^{\Sigma_{i=0}^{k} i} d_i = \Sigma_{i=0}^{k} i + x - 1$，所以
   左式 $= \Sigma_{i=0}^{\frac{k(k+1)}{2}} d_i = \Sigma_{i=0}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$ = 右式 $= \frac{k(k+1)}{2} + x - 1$，求得$x = \frac{2k^3-2k+6}{6}$

（一）算法设计

针对本部分的每一道题，要求：

（1） 采用C或C++语言设计数据结构；

（2） 给出算法的基本设计思想；

（3） 根据设计思想，采用C或C++语言描述算法，关键之处给出注释；

（4） 说明你所设计算法的时间复杂度和空间复杂度。

1. 已知一棵二叉树按顺序方式存储在数组`int A[1..n]`中。设计算法，求出下标分 别为i和j（i<=n，j<=n）的两个结点的最近的公共祖先结点的位置和值。

   基本设计思想：

   1. 把i和j提升到同一层
   2. 再每次向上一层寻找，直到找到相同的祖先

```cpp
int findFather(int i, int j) {
  if (i <= 1 || j <= 1) return 0;
  if (i == j) return i / 2;
  int t = 1;
  // 使得 i 小于等于 j
  if (i > j) {
    int tmp = i;
    i = j;
    j = tmp;
  }
  if (2 <= i && i <= 3) {
    // 特殊判断边界
    t = 2;
  } else {
    // 得到i的高度
    while (t * 2 < i) {
      t *= 2;
    }
  }
  // 降低到同一层
  while (j > t * 2) j /= 2;
  if (i == j) return i / 2;
  while (i != j) {
    i /= 2;
    j /= 2;
  }
  return i;
}
// a[findFather(i, j)]即为所求的值
```

2. 假设二叉树引采用二叉链表存储，在二叉树T中查找值为x的结点，试编写算法打印值为x的结点的所有祖先，假设值为x的结点不多于一个。试分析该算法的时间复杂度。

```cpp
#include <stdio.h>

using T = int;
const size_t TREE_MAX_NODE_NUM = 10;
class Tree {
 public:
  T data{};
  Tree *left = nullptr, *right = nullptr;
  Tree(T d) : data(d) {}
};

// 对树做前序遍历，每个节点都执行一遍某动作
template <typename F>
bool treeAplly(Tree *t, Tree **stack, Tree **&top, F const &f) {
  if (!t) return true;
  if (stack != nullptr) *(top++) = t;
  if (!f(t)) return false;
  if (!(treeAplly(t->left, stack, top, f) &&
        treeAplly(t->right, stack, top, f)))
    return false;
  if (stack != nullptr) top--;
  return true;
}

void findAllFathers(Tree *t, T val) {
  // 寻找到的目标节点
  Tree *node = nullptr;
  // 访问栈
  Tree **stack = new Tree *[TREE_MAX_NODE_NUM];
  Tree **top = stack;
  treeAplly(t, stack, top, [&val, &node](Tree *tr) -> bool {
    // 访问到目标节点就停止，栈保留到从根节点到目标节点的路径
```

```
    // printf("visit: %d\n", tr->data);
    if (tr->data == val) {
      node = tr;
      return false;
    }
    return true;
  });
  // 找不到
  if (node == nullptr) return;
  Tree **p = stack;
  // 一路向下打印父亲节点
  while (p != top - 1) {
    printf("%d%s", (*p)->data, (p == top - 2 ? "\n" : " -> "));
    p++;
  }
}

int main() {
  // 建立一棵树
  Tree *root = new Tree(0);
  root->left = new Tree(1);
  root->right = new Tree(2);
  Tree *left = root->left, *right = root->right;
  left->left = new Tree(4);
  left->right = new Tree(6);
  left = left->left;
  left->left = new Tree(7);
  left->right = new Tree(9);
  findAllFathers(root, 9);
  return 0;
}
```

1. 使用前序遍历，复杂度$O(N)$
2. 每次访问节点都会修改一次路径栈，所以修改的时间复杂度是$O(N)$
3. 所以总体的时间复杂度是$O(N)$

3. 一棵二叉树T的繁茂度定义为**各层结点数的最大值**（也称二叉树的宽度）和二叉树的**高度**的乘积。试设计算法，求给定二叉树T的繁茂度。

```
#include <stdio.h>

using T = int;
const size_t TREE_MAX_NODE_NUM = 40;
class Tree {
 public:
  T data{};
  Tree *left = nullptr, *right = nullptr;
  Tree(T d) : data(d) {}
};

class Queue {
 public:
  Tree *data = nullptr;
  Queue *next = nullptr;
  int depth = -1;
  Queue(Tree *p) : data(p) {}
  Queue(int d) : depth(d) {}
  Queue() {}
  void link(Queue *q) { this->next = q; }
};

int BFS(Tree *t) {
  int depthMax = 0;
  int widthMax = 0;
```

```
  Queue *front = new Queue(t), *temp = nullptr;
  front->depth = 0;
  Queue *top = new Queue(1);
  front->link(top);
  Tree *f = nullptr;
  while (front != top) {
    // pop
    f = front->data;
    int depthNow = front->depth;
    if (depthNow > depthMax) depthMax = depthNow;
    // printf("visit: %d | %d\n", depthNow, f->data);
    temp = front;
    front = front->next;
    delete temp;
    if (f->left != nullptr) {
      top->data = f->left;
      top->depth = depthNow + 1;
      top->next = new Queue(nullptr);
      top = top->next;
    }
    if (f->right != nullptr) {
      top->data = f->right;
      top->depth = depthNow + 1;
      top->next = new Queue(nullptr);
      top = top->next;
    }
    // 即将轮到下一层了
    if (front->next != nullptr) {
      if (front->next->depth > depthNow) {
        Queue *t = front;
        // 实际上是下一层的宽度
        int widthNow = 0;
        while (t != top) {
          t = t->next;
          widthNow++;
        }
        // printf("widthNow = %d\n", widthNow);
        if (widthNow > widthMax) widthMax = widthNow;
      }
    }
  }
  return widthMax * depthMax;
}

int main() {
  // 建立一棵树
  Tree *root = new Tree(0);
  root->left = new Tree(1);
  root->right = new Tree(2);
  Tree *left = root->left, *right = root->right;
  left->left = new Tree(4);
  left->right = new Tree(6);
  right->left = new Tree(10);
  left = left->left;
  left->left = new Tree(7);
  left->right = new Tree(9);
  printf("BFS(root) = %d\n", BFS(root));
  return 0;
}
```

4. 设计算法，对于二叉树T中每一个元素值为x的结点，删去以它为根的子树，并释放相应的空间。

```
#include <stdio.h>

```

```cpp
using T = int;
const size_t TREE_MAX_NODE_NUM = 40;
class Tree {
 public:
  T data{};
  Tree *left = nullptr, *right = nullptr;
  Tree(T d) : data(d) {}
  // 利用析构函数自动释放内存
  ~Tree() {
    printf("\t~deleting %d\n", this->data);
    if (this->left) delete this->left;
    if (this->right) delete this->right;
  }
};

class Queue {
 public:
  Tree *data = nullptr;
  Queue *next = nullptr;
  Queue(Tree *p) : data(p) {}
  Queue() {}
  void link(Queue *q) { this->next = q; }
};

void BFS(Tree *t, T val) {
  if (t->data == val) {
    delete t;
    return;
  }
  Queue *front = new Queue(t), *temp = nullptr;
  Queue *top = new Queue(nullptr);
  front->link(top);
  Tree *f = nullptr;
  while (front != top) {
    // pop
    f = front->data;
    printf("visit: %d\n", f->data);
    temp = front;
    front = front->next;
    delete temp;
    if (f->left != nullptr) {
      if (f->left->data == val) {
        delete f->left;
      } else {
        top->data = f->left;
        top->next = new Queue(nullptr);
        top = top->next;
      }
    }
    if (f->right != nullptr) {
      if (f->right->data == val) {
        delete f->right;
      } else {
        top->data = f->right;
        top->next = new Queue(nullptr);
        top = top->next;
      }
    }
  }
}

int main() {
  // 建立一棵树
  Tree *root = new Tree(0);
  root->left = new Tree(1);
```

```
  root->right = new Tree(2);
  Tree *left = root->left, *right = root->right;
  left->left = new Tree(4);
  left->right = new Tree(6);
  right->left = new Tree(4);
  left = left->left;
  left->left = new Tree(7);
  left->right = new Tree(9);
  BFS(root, 4);
  return 0;
}
```