16-17-2学期数据结构与算法期末复习

学习资料

> - Choice: basic concepts and medthods 40%
> - Answer questions: course content understanding and analysis 36%
> - Problem solving and algorithm 24%
>
> ---
>
> - judegemnt
> - answer questions
> - run an algorithm
> - application of data structures and algorithms
> - design data structures and algorithms

上册

1.1 Overview: System Life Cycle

p1-4 系统生命周期

- requirements
- analysis
- design
- refinement and coding
- verification

1.3 Data Abstraction and Encapsulation

p7-12 数据抽象与封装

1.5 Algorithm Specification

p25-34 算法规范

1.7 Performance Analysis And Measurement

p37-73 性能评估

- 空间复杂度
- 时间复杂度

2.2 Array as an Abstract Data Type

p84-86 抽象数组

2.3 The Polynomial Abstract Data Type

p86-93 抽象多项式

```
循环链表实现下的多项式加法/乘法
加法
Polynomial operator+(const Polynomial& a,const Polynomial& b)
{
    Term* aCurrent =a.head->link;
    Term* bCurrent =b.head->link;
    Polynomial newPolynomial;
    while(aCurrent!=a.head && bCurrent!=b.head)
    {
        if (aCurrent->exp == bCurrent->exp)
        {
            newPolynomial.addTerm(aCurrent->coef+bCurrent->coef,
                aCurrent->exp);
            aCurrent =aCurrent->link;
            bCurrent =bCurrent->link;
        }
        else if (aCurrent->exp>bCurrent->exp)
        {
            newPolynomial.addTerm(aCurrent->coef,aCurrent->exp);
            aCurrent =aCurrent->link;
            continue;
        }
        else if (aCurrent->exp<bCurrent->exp)
        {
            newPolynomial.addTerm(bCurrent->coef,bCurrent->exp);
            bCurrent =bCurrent->link;
            continue;
        }
    }
```

```
    while(aCurrent!=a.head)
    {
        newPolynomial.addTerm(aCurrent->coef,aCurrent->exp);
        aCurrent =aCurrent->link;
    }
    while(bCurrent!=b.head)
    {
        newPolynomial.addTerm(bCurrent->coef,bCurrent->exp);
        bCurrent =bCurrent->link;
    }
    return newPolynomial;
}
乘法
Polynomial operator*(const Polynomial& a,const Polynomial& b)
{
    Term* aCurrent =a.head->link;
    Polynomial sumPolynomial;
    while(aCurrent!=a.head)
    {
        Polynomial newPolynomial;
        Term* bCurrent =b.head->link;
        while(bCurrent!=b.head)
        {
            newPolynomial.addTerm(aCurrent->coef*bCurrent->coef,
                aCurrent->exp+bCurrent->exp);
            bCurrent=bCurrent->link;
        }
        sumPolynomial =sumPolynomial+newPolynomial;
        aCurrent =aCurrent->link;
    }
    return sumPolynomial;
}
```

4.2 Representation Chains in C++

p173-183 C++链表

- 单链表

```
最简单链表倒链
    void Reverse()
    {
        LinkNode<T>* previousNode =NULL;
        LinkNode<T>* currentNode =head;
        while(currentNode!=NULL)
        {
            LinkNode<T>* nextNode =currentNode->next;
            currentNode->next =previousNode;
            previousNode =currentNode;
            currentNode =nextNode;
```

```
            }
        }
```

- 双链表
- 环链表
- 可用空间表

下册

5.2 Binary Tree

p251-259 二叉树

```
根据所给的前序遍历和中序遍历字符串构造二叉树
    Tree(const string& pre_sequence,const string& in_sequence)
    {
        root =creator(pre_sequence,in_sequence);
    }
    TreeNode* creator(const string& pre_sequence,
        const string& in_sequence)
    {
        if (pre_sequence.length()==0) return NULL;
        int subrootindex =in_sequence.find(pre_sequence[0]);
        return new TreeNode(pre_sequence[0],
                creator(pre_sequence.substr(1,subrootindex),
                    in_sequence.substr(0,subrootindex)),
                creator(pre_sequence.substr(subrootindex+1),
                    in_sequence.substr(subrootindex+1)));
    }
```

5.3.1-6 Binary Tree Traversal

p259-269 二叉树遍历

- 中序遍历[inorder]

```
递归写法[recursive]
    void Inorder(TreeNode<T>* currentNode)
    {//root做参数即遍历
        if(currentNode!=NULL)
        {
            Inorder(currentNode->left);
            //visit currentNode
            Inorder(currentNode->right);
```

```
        }
    }


迭代写法[iterative]
    void Inorder()
    {
        stack<TreeNode<T>*>s;
        TreeNode<T>* currentNode =root;
        while (true)
        {
            while (currentNode!=NULL)
            {
                s.push(currentNode);
                currentNode =currentNode->left;
            }
            if(s.empty()) return;
            currentNode =s.top();
            s.pop();
            //vist currentNode
            currentNode =currentNode->right;
        }
    }
```

- 先序遍历[preorder]

```
迭归写法[recursive]
    void Preorder(TreeNode<T>* currentNode)
    {//root做参数即遍历
        if(currentNode!=NULL)
        {
            //visit currentNode
            Preorder(currentNode->left);
            Preorder(currentNode->right);
        }
    }


迭代写法[iterative]
    void Preorder()
    {
        stack<TreeNode<T>*> s;
        TreeNode<T>* currentNode =root;
        while(true)
        {
            while(currentNode!=NULL)
            {
                //visit currentNode
                s.push(currentNode);
                currentNode =currentNode->leftChild;
            }
            if (s.empty()) return;
```

```
                currentNode =s.top();
                s.pop();
                currentNode =currentNode->rightChild;
            }
        }
```

- 后序遍历[postorder]

```
递归写法[recursive]
    void Postorder(TreeNode<T>* currentNode)
    {//root做参数即遍历
        if(currentNode!=NULL)
        {
            Postorder(currentNode->left);
            Postorder(currentNode->right);
            //visit currentNode
        }
    }


迭代写法[iterative]
    void Postorder()
    {
        stack<TreeNode<T>*> s;
        TreeNode<T>* currentNode =root;
        while(true)
        {
            while(currentNode!=NULL)
            {
                s.push(currentNode);
                currentNode =currentNode->leftChild;
            }
            if(s.empty()) return;
            currentNode =s.top();
            if (currentNode->PostFlag ==true)
            {
                s.pop();
                //vist currentNode
                currentNode->PostFlag =false;
                currentNode =NULL;
                return;
            }
            else
            {
                currentNode->PostFlag =true;
                currentNode =currentNode->rightChild;
            }
        }
    }
```

- 层序遍历[levelorder]

```
迭代写法[iterative]
    void Levelorder()
    {
        queue<TreeNode<T>*> q;
        TreeNode<T>* currenNode =root;
        while (currenNode!=NULL) {
            //visit currentNode
            if(currenNode->left!=NULL)
            {
                q.push(currenNode->left);
            }
            if(currenNode->right!=NULL)
            {
                q.push(currenNode->right);
            }
            if(q.empty()) return;
            currenNode =q.front();
            q.pop();
        }
    }
```

- 迭代器[iterator]

```
中序迭代器
    class InorderIterator
    {
    public:
        InorderIterator(const BinaryTree<T>& t){currentNode =t.root;}
        T* Next()
        {
            T* temp =NULL;
            while(currentNode!=NULL)
            {
                s.push(currentNode);
                currentNode =currentNode->leftChild;
            }
            if (s.empty())return temp;
            currentNode =s.top();
            s.pop();
            temp =&(currentNode->data);
            currentNode =currentNode->rightChild;
            return temp;
        }
    private:
        stack<TreeNode<T>*>s;
        TreeNode<T>* currentNode;
    };


先序迭代器
    class PreorderIterator
    {
```

```
    public:
        PreorderIterator(const BinaryTree<T>& t){currentNode =t.root;}
        T* Next()
        {
            T* temp =NULL;
            if (currentNode!=NULL)
            {
                temp= &(currentNode->data);
                if (currentNode->leftChild!=NULL)
                {
                    s.push(currentNode);
                    currentNode =currentNode->leftChild;
                    return temp;
                }
                if(currentNode->rightChild!=NULL)
                {
                    s.push(currentNode);
                    currentNode =currentNode->rightChild;
                    return temp;
                }
                if(!s.empty())
                {
                    currentNode=s.top()->rightChild;
                    s.pop();
                    return temp;
                }
                return NULL;
            }
            return temp;
        }
    private:
        stack<TreeNode<T>*>s;
        TreeNode<T>* currentNode;
    };


后序迭代器
    class PostorderIterator
    {
    public:
        PostorderIterator(const BinaryTree<T>& t){currentNode =t.root;}
        T* Next()
        {
            T* temp =NULL;
            while(true)
            {
                while(currentNode!=NULL)
                {
                    s.push(currentNode);
                    currentNode =currentNode->leftChild;
                }
                if(s.empty()) return temp;
                currentNode =s.top();
```

```
                if (currentNode->PostFlag ==true)
                {
                    s.pop();
                    temp =&(currentNode->data);
                    currentNode->PostFlag =false;
                    currentNode =NULL;
                    return temp;
                }
                else
                {
                    currentNode->PostFlag =true;
                    currentNode =currentNode->rightChild;
                }
            }
        }
    private:
        stack<TreeNode<T>*>s;
        TreeNode<T>* currentNode;
    };


层序迭代器
    class LevelorderIterator
    {
    public:
        LevelorderIterator(const BinaryTree<T>& t){currentNode =t.root;}
        T* Next()
        {
            T* temp =NULL;
            if (currentNode!=NULL)
            {
                temp= &(currentNode->data);
                if (currentNode->leftChild!=NULL)
                {
                    q.push(currentNode->leftChild);
                }
                if (currentNode->rightChild!=NULL)
                {
                    q.push(currentNode->rightChild);
                }
                if (!q.empty())
                {
                    currentNode =q.front();
                    q.pop();
                }
                else
                {
                    currentNode =NULL;
                }
            }
            return temp;
        }
    private:
```

```
        queue<TreeNode<T>*>q;
        TreeNode<T>* currentNode;
    };
```

- 应用

```
拷贝(先序遍历)
    TreeNode<T>* Copy(TreeNode<T>* currentNode)
    {//另一棵树root做参数时即为拷贝构造
        if (currentNode!=NULL)
        {
            TreeNode<T>* newNode =new TreeNode<T>(currentNode->data,
                Copy(currentNode->left),Copy(currentNode->right));
            return newNode;
        }
        else
        {
            return NULL;
        }
    }

删除(后序遍历)
    void Clear(TreeNode<T>* currentNode)
    {//root做参数时即为析构
        if (currentNode!=NULL)
        {
            Clear(currentNode->left);
            Clear(currentNode->right;
            delete currentNode;
        }
    }

计算叶节点数
    int countLeaves(TreeNode<T>* currentNode)
    {//root做参数时即计算全部叶节点数
        int count=0;
        countLeaves(currentNode,count);
        return count;
    }
    void countLeaves(TreeNode<T>* currentNode,int &count)
    {
        if (currentNode!=NULL)
        {
            if (currentNode->leftChild ==NULL
                && currentNode->rightChild ==NULL)
            {
                count++;
                return;
            }
            countLeaves(currentNode->left,count);
            countLeaves(currentNode->right,count);
        }
```

```
    }

计算树高
    int countLevels(TreeNode<T>* currentNode)
    {//root做参数时即计算整棵树高
        if(currentNode==NULL) return 0;
        return (1+max(countLevels(currentNode->left),
            countLevels(currentNode->right)));
    }
```

5.6 Heap

p279-287 最大堆：以一维数组模拟树结构来实现，0号位置为空，1号位置为最大元素，左子节点序号为父节点序号2倍，右字节点序号为父节点序号2倍+1

```
template<class T>
class MaxHeap
{
    MaxHeap (int _capacity =10)
    {
        capacity =max(10,_capacity);
        heap =new T[capacity+1];
        heapSize =0;
    }
    bool IsEmpty()const{return heapSize==0;}
    T& Top(){return heap[1];}

    void Push(const T& e)
    {
        if (heapSize ==capacity)
        {
            T* temp =new T[capacity*2+1];
            copy(heap+1,heap+heapSize+1,temp+1);
            delete [] heap;
            heap =temp;
            capacity *=2;
        }
        int currentNode =++heapSize;
        while(currentNode !=1 && heap[currentNode/2]<e)
        {
            heap[currentNode] =heap[currentNode/2];
            currentNode/=2;
        }
        heap[currentNode] =e;
    }

    void Pop()
    {//删掉1号元素，再把末尾元素暂存（假象地）存在1号位置然后再确定其真实位置
        heap[1].~T();
```

```
        T lastE =heap[heapSize];
        heapSize--;
        int currentNode =1;
        int child =2;
        while(child<=heapSize)
        {
            if (child<heapSize && heap[child]<heap[child+1]) child++;
            if (lastE>=heap[child]) break;
            heap[currentNode] =heap[child];
            currentNode =child;
            child *=2;
        }
        heap[currentNode] =lastE;
    }
private:
    T* heap;
    int heapSize;
    int capacity;
};
```

5.7.1-4 Binary Search Tree

p287-293 字典的实现方式一：二叉查找树

```
查找一个对
    pair<K,E>* Get(TreeNode<pair<K,E>>* currentNode,const K& key)
    {//root做参数即查找整棵树
        if(currentNode==NULL) return NULL;
        if(key <currentNode->data.first)
            return Get(currentNode->left,key);
        if(key >currentNode->data.first)
            return Get(currentNode->right,key);
        return &(currentNode->data);
    }

插入一个对
    void Insert(const pair<K,E>& thePair)
    {
        TreeNode<pair<K,E>>* previousNode =NULL;
        TreeNode<pair<K,E>>* currentNode =root;
        while (currentNode!=NULL)
        {
            previousNode =currentNode;
            if(thePair.first <currentNode->data.first)
                currentNode =currentNode->left;
            else if(thePair.first >currentNode->data.first)
                currentNode =currentNode->right;
            else
            {
                currentNode->data.second =thePair.second;
```

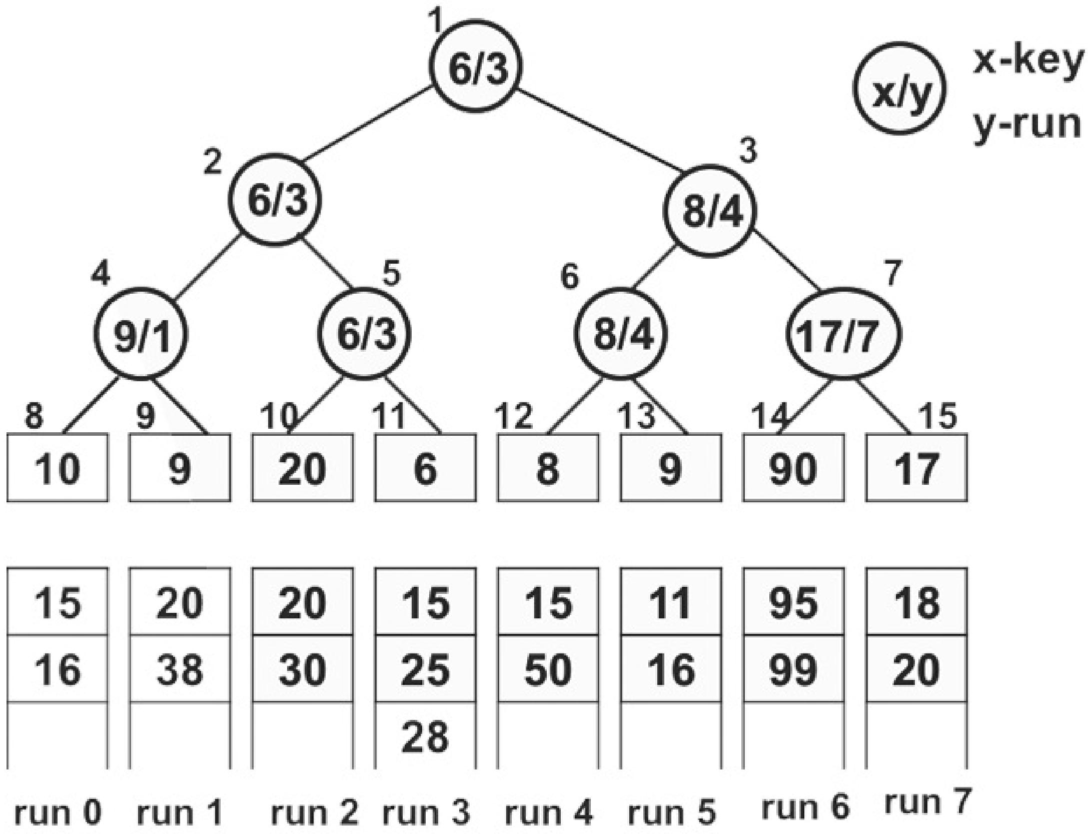
```
                return;
            }
        }
        currentNode =new TreeNode<pair<K,E>>(thePair);
        if(root!=NULL)
        {
            if(thePair.first <previousNode->data.fist)
                previousNode->left =currentNode;
            else previousNode->right =currentNode;
        }
        else
        {
            root =currentNode;
        }
    }

删除一个对
    void Delete(TreeNode<pair<K,E>>* previousNode,
        TreeNode<pair<K,E>>* currentNode,const K& key)
    {//驱动函数中NULL、root做参数
        if (currentNode ==NULL) return;
        if (key <currentNode->data.first)
            Delete(currentNode,currentNode->left,key);
        else if (key >currentNode->data.first)
            Delete(currentNode,currentNode->right,key);
        else
        {
            if (previousNode!=NULL)
            {
                if (key <previousNode->data.first)
                {
                    if(currentNode->left!=NULL)
                    {
                        if (currentNode->left->right!=NULL)
                        {
                            pair<K,E> temp =currentNode->left->data;
                            Delete(currentNode,currentNode->left,
                                currentNode->left->data.first);
                            currentNode->data =temp;
                            return;
                        }
                        else
                        {
                            previousNode->left =currentNode->left;
                            currentNode->left->right
                                =currentNode->right;
                            currentNode->~TreeNode();
                            return;
                        }
                    }
                    else
                    {
                        previousNode->left =currentNode->right;
```

```
                        currentNode->~TreeNode();
                        return;
                    }
                }
                else
                {
                    if(currentNode->right!=NULL)
                    {
                        if (currentNode->right->left!=NULL)
                        {
                            pair<K,E> temp =currentNode->right->data;
                            Delete(currentNode,currentNode->right,
                                currentNode->right->data.first);
                            currentNode->data =temp;
                            return;
                        }
                        else
                        {
                            previousNode->right =currentNode->right;
                            currentNode->right->left =currentNode->left;
                            currentNode->~TreeNode();
                            return;
                        }
                    }
                    else
                    {
                        previousNode->right =currentNode->left;
                        currentNode->~TreeNode();
                        return;
                    }
                }
            }
        }
    }
```

5.8 *Selecion Tree

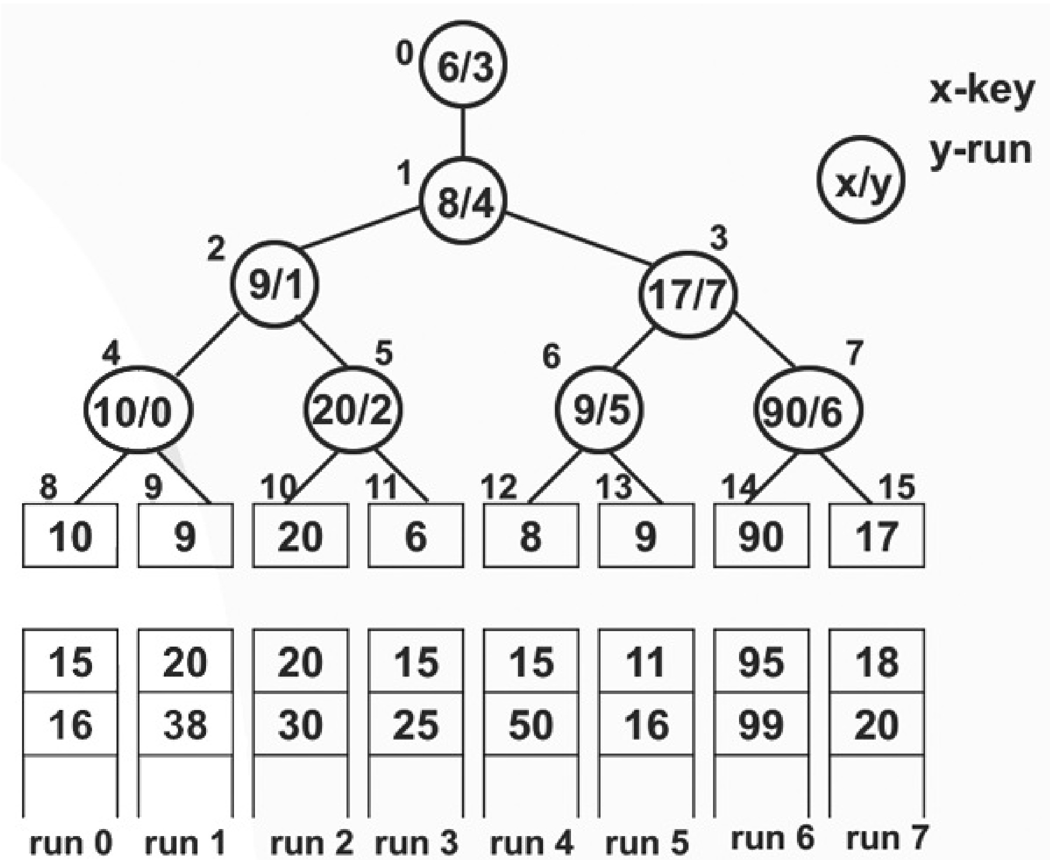
p297-299 胜者树： 一位数组模拟树结构实现，0号节点不存放，1号节点（树根）存放最小元素；父节点存放子节点中较小值

- 示例图



p299-301 败者树：一位数组模拟树结构实现，先构造胜者树（上树过程），0号节点存放根节点元素后，父节点元素自上而下依次刷新为子节点中较大值（败者，下树过程）

- 示例图

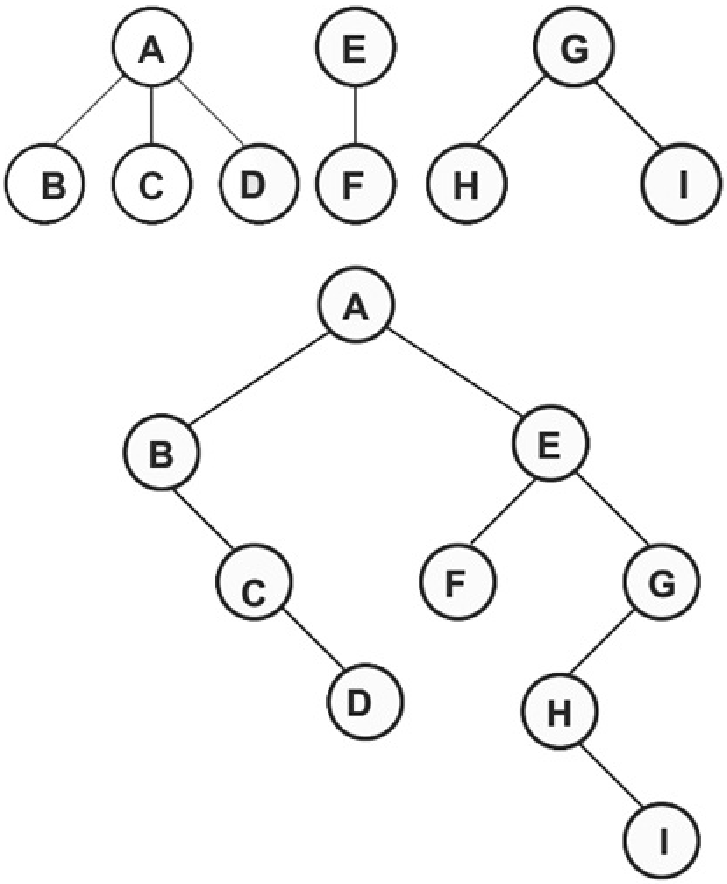



```
template<class T>
int* LoserTree(queue<T>* records,int k)
{//sort records[1:k] to a losertree
    int* index =new int[2*k];
    //up the tree
    for(int i=1;i<=k;i++) index[k-1+i] =i;
    for(int i=2*k-2;i>=2;i-=2)
    {
        index[i/2] =(records[index[i]].front()
                <records[index[i+1]].front()?
                    index[i]:index[i+1]);
    }
    //down the tree
    index[0]=index[1];
    for (int i=1;i<k;i++)
    {
        index[i] =(records[index[i*2]].front()
                >records[index[i*2+1]].front()?
                    index[i*2]:index[i*2+1]);
    }
    return index;
}
```

5.9 Transforming a Forest into a Binary Tree

p301-304 将森林用二叉树表示：每棵树按照层遍历，对应生成的二叉树中父节点为层先序，右子节点为同层后序节点，左子节点为下层第一个元素

- 示例图



5.10 Representation of Disjoint Sets

p304-315 并查集

```
class Sets
{
public:
     Sets(int _n)
     {
         n=_n;
         parent =new int[n];
         fill(parent,parent+n,-1);
     }

     void Union(int i,int j)
     {//按权合并
         i =Find(i);
         j =Find(j);
```

```
        if(i==j) return;
        int temp= parent[i]+parent[j];
        if(parent[i]>parent[j])
        {
            parent[i]=j;
            parent[j]=temp;
        }
        else
        {
            parent[j]=i;
            parent[i]=temp;
        }
    }

    int Find(int i)
    {//坍缩查找
        int root =i;
        for(;parent[root]>0;root=parent[root]);
        while (i!=root)
        {//i与root间还有元素
            int current =parent[i];
            parent[i]=root;
            i =current;
        }
    }
private:
    int* parent;
    int n;
};
```

6.1.2,3 Graph

p325-340 图的表示

- 邻接矩阵[adjacency matrix]
  二维数组实现

> 无向简单无权图[graph]
>
> $$a[i][j]=\begin{cases}1, & \text{if E<i,j>} \in \text{E(G)} \\ 0, & \text{if E<i,j>} \notin \text{E(G)}\end{cases}$$
>
> $$d_i=\sum_{j=0}^{n-1}a[i][j]$$
>
> ---

> 无向简单带权图[Wgraph]
>
> $$a[i][j] = \begin{cases} E<i,j>.weight, & \text{if E<i,j>} \in \text{E(G)} \\ 0, & \text{if E<i,j>} \notin \text{E(G)} \end{cases}$$
>
> ---
>
> 有向简单无权图[digraph]
>
> $$a[i][j] = \begin{cases} 1, & \text{if E<i,j>} \in \text{E(G)} \\ 0, & \text{if E<i,j>} \notin \text{E(G)} \end{cases}$$
>
> $$d_i out = \sum_{j=0}^{n-1} a[i][j]$$
>
> $$d_i in = \sum_{j=0}^{n-1} a[j][i]$$

- 邻接链表[adjacency list]
  一位数组+链表实现

```
class LinkedGraph
{
public:
    LinkedGraph (const int vertices)
    {
        e =0;
        n =vertices;
        adjLists = new Chain<int>[n];
        Chain<int> empty;
        fill(adjLists, adjLists+n,empty);
    }
private:
    Chain<int>* adjLists;
    int n;
    int e;
};
```

- 邻接多表[adjacency multilist]

```
class Graph;
class Edge
{
    friend class Graph;
private:
    bool m;//访问标记
    int vertex1;//边顶点一
```

```
    int vertex2;//边顶点二
    Edge* path1;//指向依附于顶点一的下一条边
    Edge* path2;//指向依附于顶点二的下一条边
    int weight;//边的权值
};
class Graph
{
public:
    void InsertEdge(int u,int v,int weight)
    {
        Edge* p = new Edge();
        p->m =false;
        p->vertex1 =u;
        p->vertex2 =v;
        p->weight =weight;
        //指向上次插入的边，初始为空
        p->path1 =adjMultiLists[u];
        p->path2 =adjMultiLists[v];
        //指向最后一条插入的边
        adjLists[u] =adjLists[v] =p;
        e++;
    }
private:
    Edge* adjLists;//指向最后一条插入的依附于顶点[0:n-1]的边
    int n;
    int e;
};
```

6.2.1,2 Elementary Graph Operations

p341-344

- 深度优先搜索[Depth First Search, DFS]
  时间复杂度用邻接表表示时 $O(n+e)$ ；用邻接矩阵表示时$O(n^2)$

```
void DFS(const int start)
{
    bool* visited =new bool[n];
    fill(visited,visted+n,false);
    DFS(start,visited);
    delete [] visited;
}
void DFS(const int v,bool* visited)
{
    visited[v] =true;
    foreach (int w, w adjacent to v)
    {//depend on the representation
        if (visited[w]==false)
        {
            //visit
```

```
                DFS(w);
            }
        }
    }
```

- 广度优先搜索[Breadth First Search, BFS]
  时间复杂度用邻接表表示时 $O(n+e)$；用邻接矩阵表示时$O(n^2)$

```
void BFS (const int start)
{
    bool* visited =new bool[n];
    fill(visited,visited+n,false);
    //visit start
    visited[start] =true;
    queue<int> q;
    q.push(start);
    while (!q.empty())
    {
        int v =q.front();
        q.pop();
        foreach (int w, w adjacent to v)
        {//depend on the representation
            if (visited[w]==false)
            {
                //visit
                q.push(w);
                visited[w] =true;
            }
        }
    }
    delete [] visited;
}
```

6.2.5 Biconnected Components

p347-352 双连通分支

- 深度优先号[depth-first number, dfn] 深度优先搜索序号，父节点序号小于子节点序号
- 回边[back edge] 原图中除去深度优先搜索生成树树边以外连接深度优先搜索树父子节点的边
- 横边[cross edge] 原图中除去树边和回边的边
- 回溯序号[low] 某一节点通过自己的或者子节点的树边和回边（有向的）能回到的最小深度优先号

$$low(w) = min\{dfn(w), min\{low(x)|x is a child of w\}, min\{dfn(x)|E<w,x> is a back edge\}\}$$

- 割点/接合点[articulation point] 回溯序号和深度优先号相等的点
- 双连通图[biconnected graph] 没有割点的图

```
基本原理
void DfnLow (const int x )
{
    int count =1;
    int* dfn =new int[n];
    int* low =new int[n];
    fill(dfn,dfn+n,0);
    fill(low,low+n,0);
    DfnLow(x,-1,count);
    delete [] dfn;
    delete [] low;
}
void DfnLow (int u,int v,int& count)
{
    dfn[u] =low[u] =count++;
    foreach (int w, w adjacent from u)
    {//depend on the representation
        if(dfn[w] ==0)
        {//tree edge
            DfnLow (w,u,count);
            low[u] =min(low[u],low[w]);
        }
        else if(w!=v)
        {//back edge
            low[u] =min(low[u],dfn[w]);
        }
    }
}
应用：计算桥
void Bridge ()
{//cutPoint is a queue<int> data member of the graph
    cutPoint.clear();
    int count =1;
    int* dfn =new int[n];
    int* low =new int[n];
    fill(dfn, dfn+n,0);
    fill(low, low+n,0);
    Bridge (0,-1,count);
    delete [] dfn;
    delete [] low;
}
void Bridge (int u,int v,int& count)
{
    dfn[u] = low[u] = num++;
    for (int w=0;w<n;w++)
    {
        if (g[u][w]<UNCONNECTED)
        {
            if (dfn[w] == 0) {
                Bridge (w,u,count);
                low[u] = min(low[u], low[w]);
                if (low[w] >= dfn[u]) {
```

```
                    cutPoint.push(u);
                    if (cutPoint.size()>=2)
                    {
                        int x=cutPoint.front();
                        cutPoint.pop();
                        int y=cutPoint.front();
                        cutPoint.pop();
                        cout<<"The bridge:<"<<x<<","<<y<<">"<<endl;
                    }
                }
            }
            else if (w != v) low[u] =min(low[u],dfn[w]);
        }
    }
}
```

6.5 Activity Networks

p375-389 活动网络

- AOV活动网络[Activity-on-Vertex Networks]
  点表示活动，有向边表示活动先后次序
  拓扑排序[Topological order]
- AOE活动网络[Activity-on-Edge Networks]
  点表示事件，有向边表示活动
  事件最早发生时间
  活动最早开始时间
  事件最晚发生时间
  活动最晚开始时间
  关键活动：最早开始与最晚开始时间相等的活动

7.2 Insertion Sort

p399-401 插入排序，小规模时因控制简单，效率最高；稳定排序；时间复杂度 $O(n^2)$

```
template<class T>
void InsertionSort(T* array,const int n)
{//sort array[0:n-1]
    for(int i=1;i<n;i++)
    {
        T current = array[i];
        int j=i-1;
        for(;j>=0;j--)
        {
```

```
                if(current <array[j])
                {
                    array[j+1] =array[j];
                }
                else
                {
                    break;
                }
            }
            array[j+1] =current;
        }
    }
```

7.3 Quick Sort

p402-405 快速排序，大规模时综合性能最好；不稳定排序；时间复杂度 $O(nlogn)$

```
template<class T>
void QuickSort(T* array,const int start,const int end)
{//sort array[0:n-1]
    if(start<end)
    {
        T& pivot =array[start];
        int i=start+1;
        int j=end;
        do
        {
            for(;array[i]<pivot;i++);
            for(;array[j]>pivot;j--);
            if(i<j) swap(array[i],array[j]);
        }while (i<j);
        swap(pivot,array[j]);
        QuickSort(array,start,j-1);
        QuickSort(array,i,end);
    }
}
```

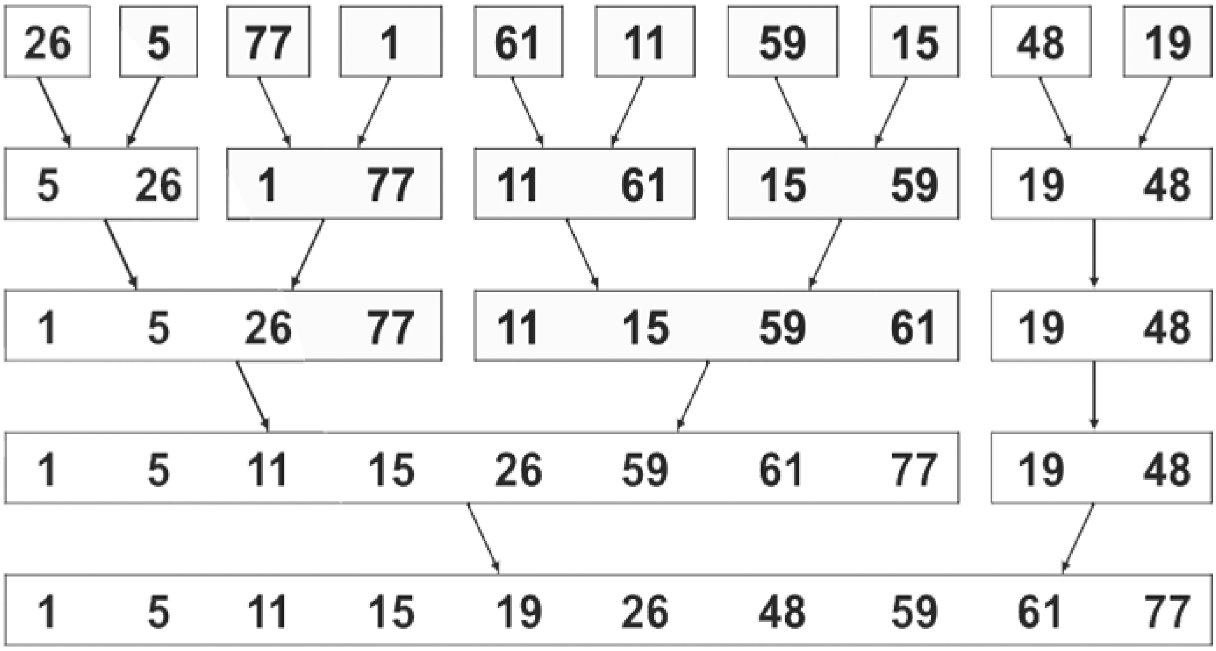
7.5 Iterative Merge Sort

p409-410 二路迭代归并排序，稳定排序；时间复杂度 $O(nlogn)$

```
template <class T>
void Merge (T* initList,T* mergedList,
            const int start1,const int start2,
            const int n)
{
    int i1=start1,iResult=start1,i2=start2;
```

```
    for (;i1<start2 && i2<n;iResult++)
    {
        if (initList[i1] <= initList[i2])
        {//stable
            mergedList[iResult]=initList[i1++];
        }
        else
        {
            mergedList[iResult]=initList[i2++];
        }
    }
    copy(initList+i1,initList+start2,
         mergedList+iResult);
    copy(initList+i2,initList+n,
         mergedList+iResult);
}
template <class T>
void MergePass (T* initList,T* resultList,
                const int n, const int block_size)
{
    int i=0;
    for (;i<n-2*block_size+1;i+=2*block_size)
        Merge(initList,resultList,
              i,i+block_size,i+2*block_size);
    //merge remaining list of length<2*block_size
    if (n-i>block_size)
    {
        Merge(initList,resultList,
              i,i+block_size,n);
    }
    else
    {
        copy(initList+i,initList+n,resultList+i);
    }
}
template <class T>
void MergeSort (T* a,const int n)
{//Sort a[0:n-1]
    T *tempList =new T[n];
    for (int block_size=1;block_size<=n;block_size*=2)
    {
        MergePass(a,tempList,n,block_size);
        block_size*=2;
        MergePass(tempList,a,n,block_size);
    }
    delete [] tempList;
}
```

- 示例图



7.6 Heap Sort

p414-416 堆排序，利用最大堆原理实现，不稳定排序；时间复杂度 $O(nlogn)$

```
template<class T>
void Heapify(T* array,const int root,const int n)
{
    T current=array[root];
    int j =2*root+1;
    for(;j<n;j =j*2+1)
    {
        if(j<n-1 && array[j]<array[j+1]) j++;
        if(current>= array[j]) break;
        array[(j+1)/2-1]=array[j];
    }
    array[(j+1)/2-1]=current;
}
template<class T>
void HeapSort(T* array,const int n)
{//sort array[0:n-1]
   for(int i=n/2-1;i>=0;i--)
   {
       Heapify(array,i,n);
   }

   for(int i=n-2;i>=0;i--)
   {
       swap(array[0],array[i+1]);
       Heapify(array,0,i+1);
   }
```

```
}
```

7.7 *RadixSort

p417-422 桶排序，稳定排序；时间复杂度 $O(kn)$

```
int digit(int e,const int i,const int r)
{

    return (int)(e/pow((double)r,i))%r;
}
void RadixSort(int* a,const int n,const int r)
{//sort int array[0:n-1]
    if (a==NULL || r<=0 || n<=0) return;

    int maxelement =a[0];
    for(int i=1;i<n;i++) if (maxelement<a[i]) maxelement =a[i];
    int d=(int)(log((double)maxelement)/log((double)r));
    if(pow((double)r,d)<=maxelement) d++;

    int* front =new int[r];
    int* end =new int[r];
    int* link =new int[n];
    int root=0;

    for (int i=0;i<n-1;i++) link[i]=i+1;
    link[n-1]=-1;

    for (int i=0;i<d;i++)
    {
        fill(front,front+r,-1);

        for (int current =root;current!=-1;current =link[current])
        {
            int index =digit(a[current],i,r);

            if (front[index] ==-1)
            {
                front[index] =current;
            }
            else
            {
                link[end[index]]=current;
            }
            end[index] =current;
        }

        int start_bin =0;
        for(;start_bin<r && front[start_bin]==-1;start_bin++);
        root =front[start_bin];
```

```
        int end_bin =start_bin;
        for(int j=end_bin;j<r;j++)
        {
            if (front[j]!=-1)
            {
                link[end[end_bin]] =front[j];
                end_bin =j;
            }
        }

        link[end[end_bin]]=-1;

    }

    delete []end;
    delete []front;

    int* temp =new int[n];

    for (int i=root,j=0;i!=-1;i =link[i],j++)
    {
        temp[j] =a[i];
    }

    copy(temp,temp+n,a);
    delete []temp;
    delete []link;
}
```

7.10.2 k-Way Merging

p442-443 外排序：k路归并排序，以k叶2叉败者树原理实现，对应m条外存记录，理论时间复杂度：$O(nlog_2m)$，与k无关，适当提高k的值可以减少外存访问次数，但不是越多越好

```
/*
r[i] --- the k records in the tree of loser
l[i] --- loser of the tournament played at node i
l[0]/q --- winner of the tournament
rn[i] --- the run number to which r[i] belongs
rc --- current run number
rq --- run number of r[q]
rmax --- number of runs that will be generated
lastRec --- last record output
*/
template <class T>
void Runs (T *r)
{
    r = new T[k];
    int* rn =new int[k];
```

```
    int* l=new int[k];
    for (int i =0;i < k;i++)
    {//input records
        ReadRecord(r[i]);
        rn[i] = 1;
    }
    InitializeLoserTree();
    int q =l[0];
    int rq=1,rc=1,rmax=1;
    T LastRec;
    while (true)
    {// output runs
        if (rq != rc)
        {// end of run
            //output end of run marker;
            if (rq > rmax) break;
            else rc = rq;
        }
        WriteRecord(r[q]); LastRec=r[q];
        //input new record into tree
        if (end of input)
        {
            rn[q]=rmax+1;
        }
        else
        {
            ReadRecord(r[q]);
            if (r[q]<LastRec) //new record belongs to next run
                rn[q] = rmax = rq+1;
            else rn[q] = rc;
        }
        rq=rn[q];
        // reconstruct the tree of losers
        for (int t=(k+q)/2;t;t/=2)
        {// t is initialized to be parent of q
            if (rn[l[t]]<rq || rn[l[t]]==rq && r[l[t]]<r[q])
            {// t is the winner
                swap(q, l[t]);
                rq = rn[q];
            }
        }
    }
    delete [] r; delete [] rn; delete [] l;
}
```

8.2.1,2,4 Static Hashing

p459-461 字典的实现方式二：静态哈希散列

- b 哈希表的大小 hashtable[0:b-1]

- s 每格哈希表的容量/槽数
- n 总记录数
- T 关键字取值范围/个数
- 关键字密度[key density] n/T
- 装载密度[loading density] $\alpha = n/(s*b)$

p461-464 哈希函数

1. 除法取余数 $h(key) = key\%D(D \neq 2^r)$
2. 平方取中间r位（转化为二进制进行运算） $b = 2^r$
3. 数值分析（事先知道关键字所有可能的取值，找出不同的段作为关键字的哈希值）

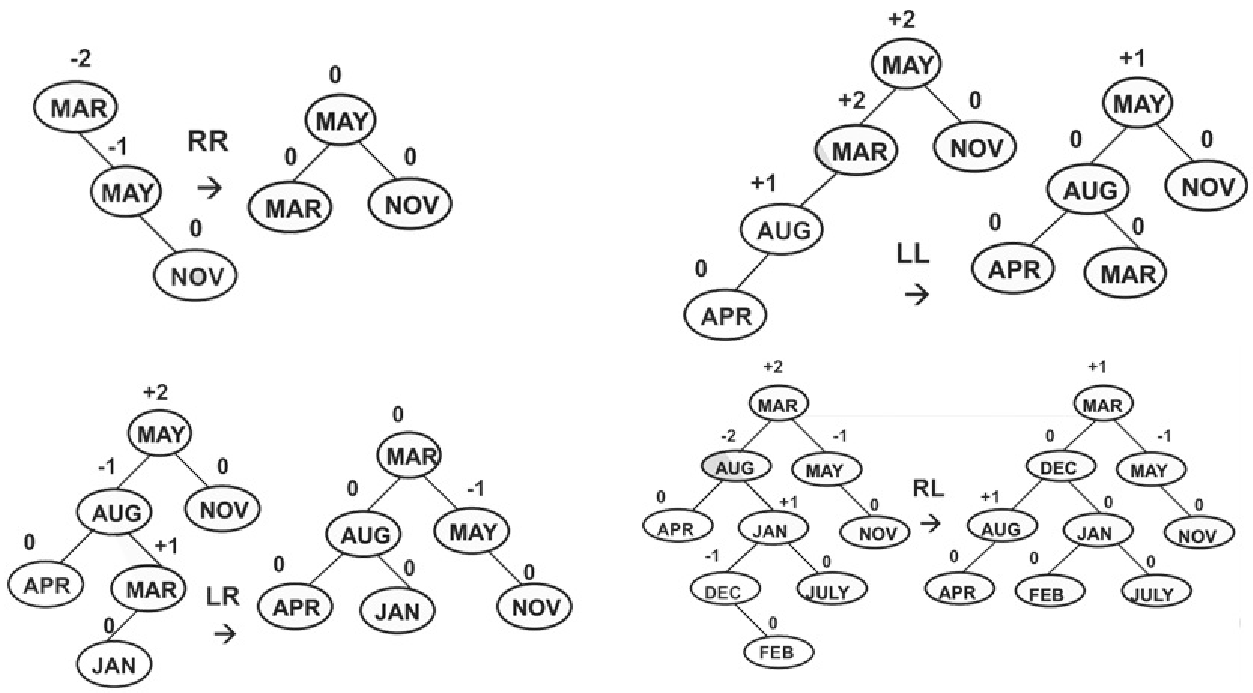
p467-473 溢出处理

1. 开放地址
   当发生溢出时，循环查找最近的一个空位插入；平均查找时间 $(2-\alpha)/(2-2\alpha)$
2. 链表
   当发生溢出时，插入到该位置的链表头/尾；平均查找时间 $1+\alpha/2$

10.2 AVL Tree

p564-578 AVL树，动态调整的二叉查找树

- 插入调整示例1



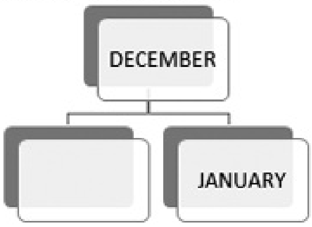
- 插入调整示例2

P578_05

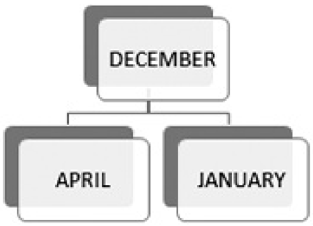
01 empty

02 insert DECEMBER

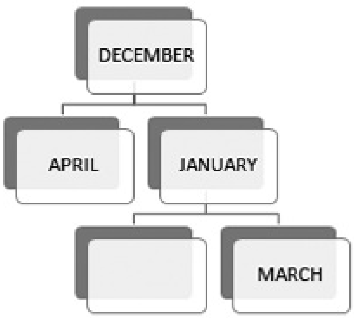
03 insert JANUARY



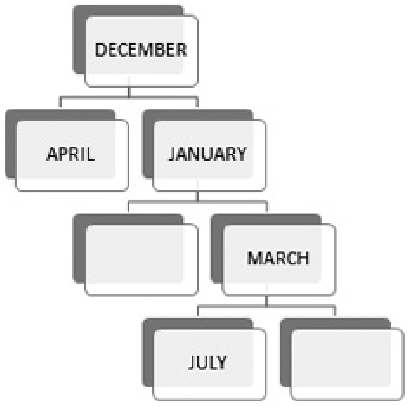
04 insert APRIL



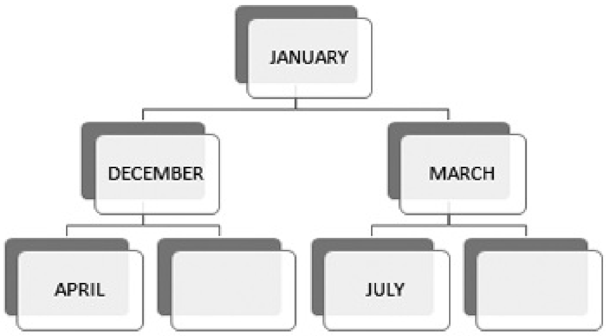
05 insert MARCH



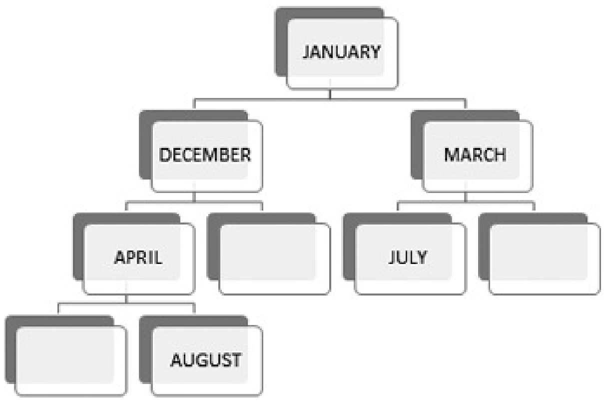
06 insert JULY



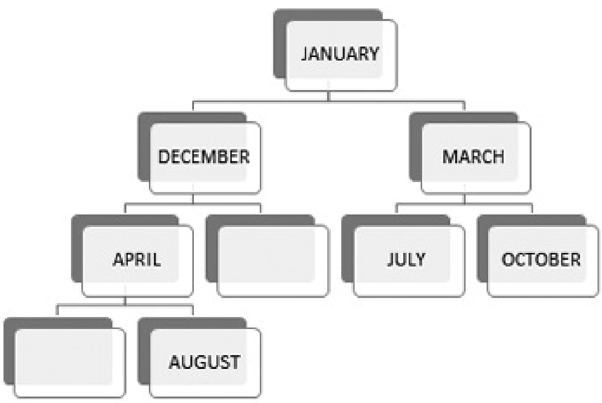
RL



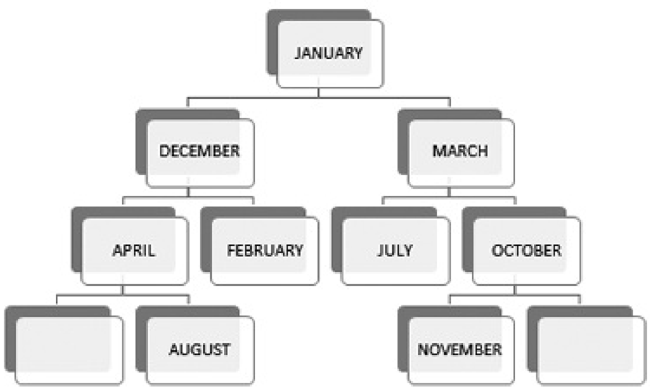
07 insert AUGUST



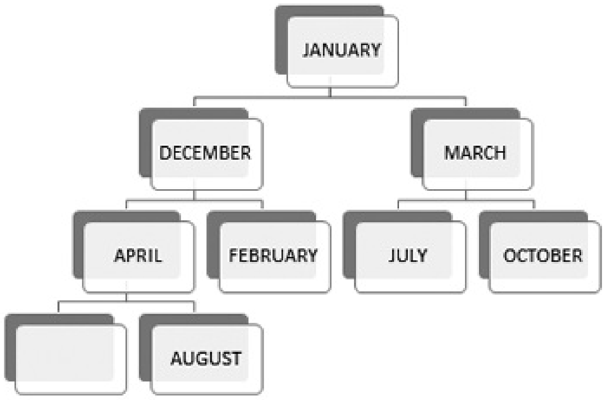
08 insert OCTOBER



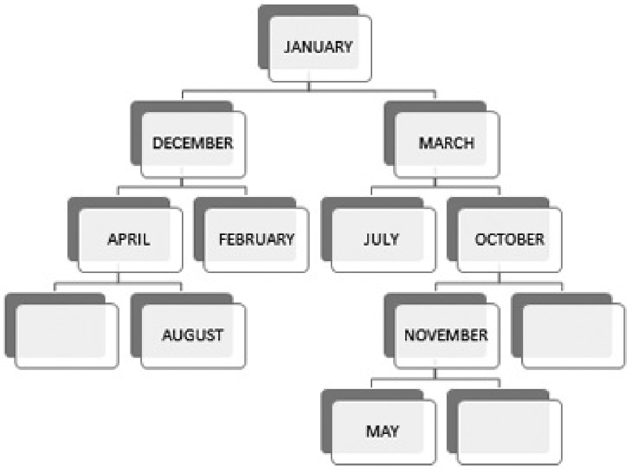
10 insert NOVEMBER



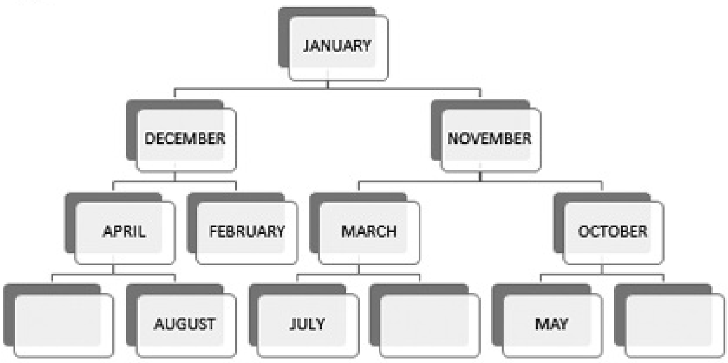
09 insert FEBRUARY



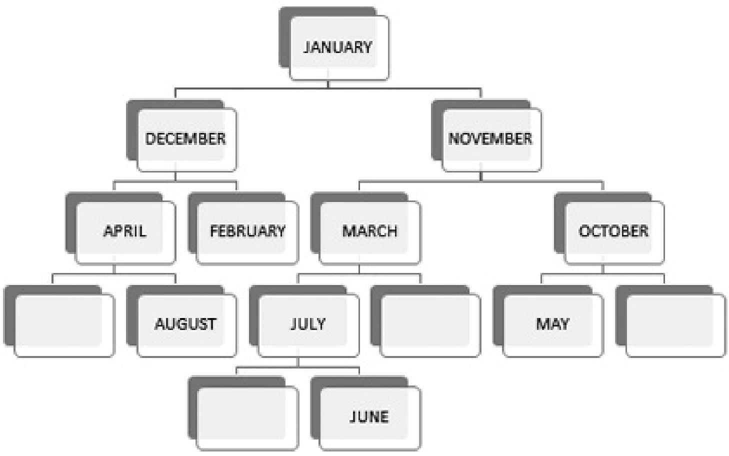
11 insert MAY



RL



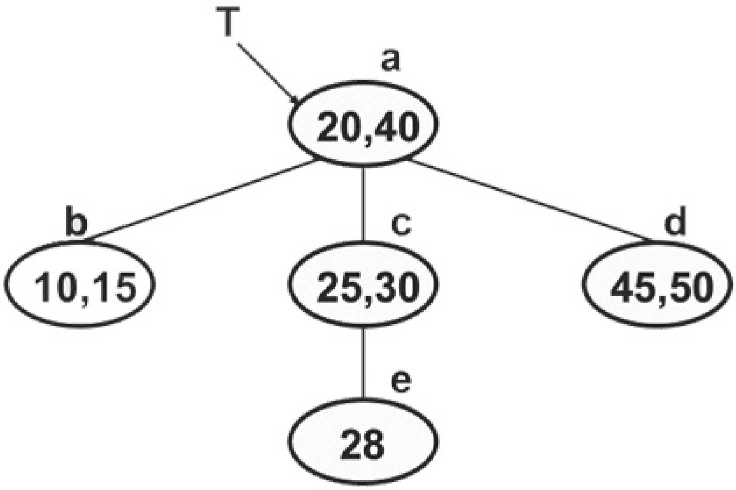
12 insert JUNE



11.1 m-Way Search Tree

p606-609 外存查找：m路查找树

- 示例图

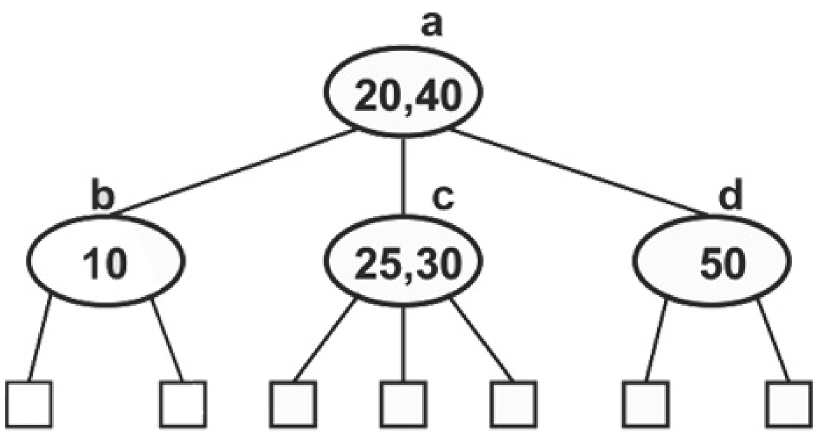


node	schematic format
节点	节点元素数，第0子节点,(元素En，子节点An)//An的E1大于本节点En
a	2,b,(20,c),(40,d)
b	2,0,(10,0),(15,0)
c	2,0,(25,e),(30,0)
d	2,0,(45,0),(50,0)
e	1,0,(28,0)

11.2.1 B-Tree

p609-611 B树：特别限定的m路查找树
定义限制：

1. The root has at least two children.
   根节点至少有2个子节点
2. All nodes other than the root node and external nodes have at least $\lceil m/2 \rceil$ children.
   除根节点以外的所有节点（包括外节点）都至少有m/2向上取整个子节点
3. All external nodes are at the same level.
   所有外节点都在同一层
   - 示例图：



A B-tree of order 3