COMP9024 Data Structures and Algorithms

Author : Yunqiu Xu

`UNSW` `Algorithm` `Java`

Week 1 Java Programming Language

- Classes and objects
- Methods
- Primitive data types and operators
- Arrays
- Control flow
- IO streams
- Inheritance and Polymorphism
- Interfaces and Abstract Classes
- Exceptions
- Casting and Generics

1.1 Classes

- class=variables+fields+functions(methods)
- 举个栗子:

```java
public class Counter{
    //init=声明+构造器
    protected int count;//声明
    Counter(){//构造器
        count=0;
    }

    //3 methods
    public int getCount(){
        return count++;
    }
    //void类型不需要返回值
    public void incrementCount(){count++;}
    public void decrementCount(){count--;}
}
```

1.1.1 Class Modifiers

- 访问修饰符:
  - public:可以被所有类实例化或继承
  - public class必须定义在类名同名文件中
  - package:可以被同一个包中的类使用->该关键字可省略
- 修饰符:
  - abstract:declared with the abstract keyword and is empty;
    - 类里有一个抽象方法就必须为抽象类!
    - 但抽象类里不一定有抽象方法!
  - final: no subclasses ; cannot be extended

1.2 Variables

- Instance variables:non-static method
  - 类内方法外，当类被装载时被实体化
  - 只是运行类， 调用类内方法时无法使用实例变量
  - 使用实例变量:
    - Fucker asshole = new Fucker()
    - 创建了一个实例对象后,可以使用类Fucker中的实例变量

```
int i =5
Fucker x=new Fucker();
```

- Class variables:static fields
  - 类内所有方法外
  - 不需加载直接运行类可以调用类变量

```
public static ...
```

- Local variables:declared in a method

```
public int fuck(){int i=1;...}
```

- Parameters...

1.2.1 scope variable modifiers

- public: 任何类访问

- protected: 同包所有类或其所有子类(不一定同包)访问
- private: 只有同类方法可以访问
- 无访问修饰符(friendly)默认可以被同包中的所有class访问
    - 注意该状态和protected不同,子类不在同一个包不能用

1.2.2 usage modifiers

- static:属于某个类而非那个类的实例的变量
- final:常量,需要initial value且不能更改
- 举个栗子:

```java
private static final boolean RED=true;
//这是一个类变量
//既是static又是final
//运行该类而非加载类时,调用类中方法可以使用该变量
```

1.3 Methods

- 访问修饰符 非访问修饰符 返回类型 名称(参数列表){}
- 访问修饰符:public,protected,private,默认在同一个包里
- 非访问修饰符:
    - static
    - final -> can not be overloaded
    - abstract : 类中已声明没有类实现的方法

1.3.1 Constructors

- a special method used to initial the object
- 构造器的名字和类名相同
- 一个类可以有多个构造器
- 构造器方法不允许：
    - 出现非访问修饰符/返回类型
    - throws exception
- 举个栗子:

```java
public class SingleLinkedList{
    //this is a constructor
    public SingleLinkedList(){
        head=null;
        size=0;
    }

    //another constructor;
```

```
    public SingleLinkedList(int i=1){
        ...
    }
```

1.3.2 Overloading Methods

- 同类中的不同方法如果参数不同可以有相同的名字:

```
public class DataArtist { ...
   public void draw(int i) { ... }
   public void draw(double f) { ... }
   public void draw(int i, double f) { ... }   }
```

- 还有个栗子是后面的DFS以及BFS算法

1.3.3 Main method

```
public static void main(String [] args){}
```

1.4 Objects

```
public class Asshole
{
    public static void main (String[] args)
    {
        Counter c;
        Counter d = new Counter();//构造器实例化类, 创建新对象,里面可以加参数
        c = new Counter();
        d = c;
    }
}
```

1.4.1 String Objects

- 字符串对象immutable:创建后就无法再更改值

```
String s ="kilo" + "meters"
```

1.4.2 Object References

```
Fucker() c=new Fucker(); //实例化一个类
c.fucking(); //调用该类内方法
```

1.5 Primitive Types

- boolean/char/byte/short/int/long/float/double

1.6 Operators

1.7 Control Statements

- if-else/switch/while/for/do-while
- 相等与等同:
  - `==`为等同,查看两个对象的存储位置
  - `A.equals(B)`才是判断是否相等的

1.8 Arrays

- 容量固定, 所有元素类型相同

```
int [] anArray=new int[10];
double [][] fuck=new double[5][1];

for (int i=0; i<anArray.length; i++)
    anArray[i]=i;

//arraycopy->数组复制
public static void arraycopy(Object src,
    int srcPos, //src开始复制的位置
    Object dest,
    int destPos, //dext接受复制的起始位
    int length//打算复制的元素
)

class ArrayCopyDemo {
    public static void main(String[] args){
    char[] copyA = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e',
'd' };
    char[] copyB = new char[7];
    System.arraycopy(copyA, 2, copyB, 0, 7);
    System.out.println(new String(copyTo));
    }
}
```

1.9 Nested Classes嵌套类

```
class OuterClass { ...
    class NestedClass { ... }
```

```
}
```

- OuterClass:public/package private
- NestedClass:private/public/protected/package private

- 使用嵌套类的原因:
    - 可以把只使用一次的类分组存在同一个地方;
    - 提升封装功能;
    - 使代码可读性更强;

1.10 Packages

- a package contains following kinds of types:
    - Classes
    - Interfaces
    - Enumerated types
    - Annotations
- 调用方法:

```java
//调用整个package
package java.awt.event;

//仅仅调用部分类
import java.awt.event.ActionEvent;

//调用import的方法
Scanner fuck=new Scanner(System.in);

//另一种调用类的方法，不需要import
java.util.Scanner fuck=new java.util.Scanner(System.in);
```

1.11 I/O Streams: Input->Output

1.11.1 Output Methods

```java
System.out.print();
System.out.printf();
System.out.println();
```

1.11.2 Input:`Scanner class`

- 举个栗子: 迭代读取文件内容

```
while(!xxx.hasNext()){
    xxx.nextLine();
    ...
}

//some other functions
Next()//next token in the input stream
hasNextType()
nextType()//the type of next token
hasNextLine()
nextLine()
findInLine(String s)
```

1.11.3 举个栗子: 处理字符串

- COMP9024 Assignment 1:
    - 建立一个构造器用于构造MyDlist
    - 参数为'stdin': 输入字符串并读取为数组, 以空行结束
    - 参数为file path：读取文本文件病返回为一行字符串,再按空格切分为数组

```
import java.util.Scanner;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.BufferedReader;

public MyDlist(String f) throws IOException{
    super();
    if(f=="stdin"){
        Scanner myInput=new Scanner(System.in);
        while(true){
            String newElement=myInput.nextLine();
            if (!newElement.equals("")){
                DNode newDNode=new DNode(newElement,null,null);
                this.addLast(newDNode); //这里的this表示引用当前对象,如果是super则为引用父类对象
            }
            else{ break; }
        }
    }

    else{
        File myFile=new File(f);
        InputStreamReader ISReader=new InputStreamReader(new FileInputStream(myFile));//create a InputStream object
        BufferedReader BReader=new BufferedReader(ISReader);//transfer the content to machine language
```

```java

        //initialize myLine
        String myLine = BReader.readLine();;
        String temp=BReader.readLine();

        while (temp!=null){
            myLine = myLine+" "+temp;
            temp=BReader.readLine();
        }

        String[] splitStrings=myLine.split(" +"); //split the string by one or more spaces
        for(int i=0;i<splitStrings.length;i++){
        DNode newDNode=new DNode(splitStrings[i],null,null);
                this.addLast(newDNode);
        }
    }
}
```

1.12 OOP Principles

- Abstraction: 化繁为简
- Encapsulation: 隐藏细节
- Modularity: 模块化

1.13 Inheritance

1.13.1 subclass-superclass

- 子类可以继承父类的所有non-private内容(field/method/nested class)
- 子类不会继承父类的constructor
    - 解决措施: super operation
- 举个栗子

```java
//创建父类Bicycle
public class Bicycle{
    //create 3 fields
    public int cadence;
    public int gear;
    public int speed;

    //constructor
    public Bicycle(int startCadence, int startSpeed, int startGear){
        gear = startGear;
        cadence = startCadence;
```

```java
            speed = startSpeed; }

        //4 methods
        public void setCadence(int newValue) { cadence = newValue; }
        public void setGear(int newValue) { gear = newValue; }
        public void applyBrake(int decrement) { speed -=decrement; }
        public void speedUp(int increment) { speed += increment; }
    }

    //创建子类
    public class MountainBike extends Bicycle {
        //add a new field
        public int seatHeight;

        //constructor
        public MountainBike(int startHeight, int startCadence, int startSpeed, int
    startGear){
            //一般情况下子类无法使用父类构造器,这里使用super
            super(startCadence, startSpeed, startGear);
            seatHeight = startHeight;
        }

        //add a new method
        public void setHeight(int newValue) { seatHeight = newValue; }
    }
```

1.13.2 子类的行为:推陈出新

- 直接使用父类的fields
  - 重新声明(hidding, not recommended)
  - 增加新fields
- 直接使用父类的方法, 也可声明新方法
  - overiding : 在子类中声明父类中已有的实例方法
  - hidding : 声明与父类有相同签名的静态方法
- 编写可以开启父类构造器的子类构造器: `implicity; keyword super`

1.13.3 Polymorphism 令对象可以有不同类型

1.13.4 Overriding / Hiding / Overloading

- Overriding : 同名实例方法取子类
  - 参数列表、返回类型[必须相同];
  - 访问权限不能高于父类:父类public子类不能protected
  - final不能重写;
  - static method不能重写但能被声明;
  - 不能被继承的方法也不能被重写;

- Hiding: 子类override后父类的同名method被隐藏
- Overloading:
  - 同一个class里,name相同参数不同;
  - 访问修饰符与返回类型[可以不同],参数[必须不同];
- 完全相同重写, 参数不同重载，重写后父类隐藏
- 举个栗子

```java
//创建父类并定义两个方法
public class Animal{
    public static void testClassMethod() {
        System.out.println("The class method in Animal.");
    }

    public void testInstanceMethod() {
        System.out.println("The instance method in Animal.");
    }
}

//子类覆盖方法，每次调用父类都被隐藏
public class Cat extends Animal {
    public static void testClassMethod(){
        System.out.println("The class method in Cat.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat."); //overloading
    }
    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

- 对于与父类有着相同签名的method

method类型	superclass static method	superclass instance method
subclass static method	hide	compile-time error
subclass instance method	compile-time error	override

1.13.5 Dynamic Dispatch

- 调用某个对象o的method a()时
- 先检查o所在的class T 中是否定义了额method a()

- 是则执行a(), 否则在T的父类S中进行检查
- 不断向上检查直到a()被找到, 否则抛出错误

1.13.6 Accessing Superclass Members

- 使用super关键字可以调用父类方法,同时子类可以重写

```java
public class Superclass {
    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
public class Subclass1 extends Superclass {
    public void printMethod(){
        super.printMethod(); //调用父类方法
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {
        Subclass1 s = new Subclass1();
        s.printMethod(); //调用子类方法
    }
}

//Output:
'Printed in Superclass.'
'Printed in Subclass'
```

1.13.7 Subclass Constructors

- 声明父类构造器的格式
  - 父类构造器只能放在子类构造器的第一行
  - 父类构造器必须被显式调用, 否则报错
  - 父类必须存在无参数构造器, 否则报错

```java
super()/super(parameter list)
```

- 举个栗子

```java
public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear){
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
}
```

- 再举个栗子

```java
public MyDlist(){
```

```java
    super();
}
```

1.13.8 在构造器中使用`this`引用类的实例变量

- 默认优先调用局部变量,加this优先调用实例变量
- static method不属于类实例, 不能使用this
- 举个栗子

```java
public class Point{
    public int x=0;//实例变量
    public int y=0;
    public Point(int x,int y){
        x=100;//局部变量
        y=50;
        this.x=x;//优先调用实例变量x=0
    }
}
```

- 另一种情况: 子类继承父类构造器后,this可用于调用父类未重写的属性或方法`this.method()/this.attribute`
- 如果子类存在重写, 只能使用super来调用父类方法 `super.method()`
- 再举个栗子

```java
class Student {
    public int age;
    public void std(){ //声明Student类的方法std()
        age = 15;
        System.out.println("学生平均年龄为："+age);
    }
}

class ThisStudent extends Student{
    public int age;
    public void std(){
        super.std();  //使用super作为父类对象的引用对象来调用父类对象里面的Std()方法
        age = 18;
        System.out.println("这个学生的年龄为："+age);
        System.out.println(super.age);  //使用super作为父类对象的引用对象来调用父类对象中的age值
        System.out.println(age);
    }
}

public class TestDif {
    public static void main(String[] args) {
        ThisStudent a = new ThisStudent();
```

```
        a.std();
    }
}


>>>
学生平均年龄为：15
这个学生的年龄为：18
15
18
```

1.14 Exceptions

```
public method() throws xxxException{
    if(xxx) throw new xxxException("fuck you");
}
```

1.14.1 Three types of exceptions:

- Checked exception;
- Error;
- Runtime exception;

1.14.2 Throwable Class

- All exception classes are subclasses of Throwable class
- 举个栗子: 抛出异常`throw`

```
method(){
    if (exception happened){
        throw new ExceptionName();
    }
}
```

- 举个栗子:在声明method时指出异常`throws`

```
public void Fuck() throws CondomException{
    submethod1(); //如果调用Fuck()时已经抛出异常,就不再检测子方法
    submothod2();
}
```

1.14.3 Catching Exceptions

```
try
    main block of statements
catch (exceptionType1 variable1)
```

```
    block 1 of statements
catch (exceptionType2 variable2)
    block 2 of statements
…
finally
    block n of statements
```

1.15 Interface

- 仅仅被声明, 不包含数据及body的一组方法集合
- 接口无法被实例化->只能被类执行或是被其他接口继承
- 有些类似python中只是给出名字但是函数段`pass`的methods

1.15.1 创建接口类似创建一个类,但是仅仅声明空方法

```
public interface Sellable{
    public String description();
    public int listPrice();
    public int lowestPrice();
}
```

1.15.2 调用接口:创建一个类`implements`接口

```
public class Photograpph implements Sellable{
    private String descript;
    private int price;
    private boolean color;

    public Photograph(String desc, int p, boolean c){
        descript = desc; price = p; color = c;
    }
    //执行接口: 在此处添加接口中methods的详情
    public String description() { return descript; }
    public int listPrice() { return price; }
    public int lowestPrice() { return price/2; }
    public boolean isColor() { return color; }
}
```

- 接口的一个重要意义是一个类可以继承多个接口, 因为java不允许继承多个类

1.16 抽象类

- 定义:同时声明空与非空方法(仅仅声明空方法就是接口)
- 有抽象方法必然为抽象类,反之不一定

- 类似接口，抽象类也无法被实例化(new)
- 子类必须提供对父类中抽象方法的实现,除非是自身抽象

1.17 类型转换

- widening从小到大：自动转换（隐式）

```
Integer i = new Integer(3);
Number n = i;
//i是integer必然属于number因此不需特地声明
```

- narrowing从大到小：手动转换（显式），不推荐

```
double n=new double(2.0);
Integer i=(Integer) n;//2.0->2
//n是double需要先转换一下才能归类到integer中因此需要声明下
```

1.18 instanceof

- 举个栗子:

```
Number n;
Integer i;
n=new Integer(3);
if(n instanceof Integer)//true
    i=(Integer) n;
n=new Double(3.1415);
if(n instanceof Integer)//false
    i=(Integer) n;
```

1.19 Generics泛型

- 用于表明用例将会采用的数据类型
- 举个栗子:泛型为String, 用例为非String抛出错误

```
Stack<String> myStack=new Stack<String>();
```

Week 2 Analysis of Algorithms

- $O/\Theta/\Omega$

- 渐近分析

2.1 Pseudocode

- 举个栗子

```
Algorithm arrayMax(A, n){
    Input:array A of n integers
    Output: maximum element of A
    currentMax = A[0];
    for (i =1; i< n; i++):
        if  (A[i]>currentMax):
            currentMax =A[i];
    return currentMax;
}
```

2.2 Random Access Machine(RAM)模型

- 重要函数
  - Constant 1 (a=b+c)
  - Logarithmic logN (binary search)
  - Linear N (单次循环)
  - N-Log-N NlogN (merge sort)
  - Quadratic N^2 (双层循环)
  - Cubic N^3 (三层循环)
  - Exponential 2^N (穷举)

2.3 Primitive Operations

- 举个栗子

```
Algorithm arrayMax(A,n){            //operations
    currentMax=A[0];                //2
    for(i=1;i<n;i++)               //2n
        if(A[I]>currentMax)         //2(n-1)
            currentMax=A[i];        //2(n-1)
    //fuck you(注释也算原语操作)     //2(n-1)
    return currentMax;              //1
}
```

- worst case:总共原语操作数8n-2

- 若a为最佳情况,b为最坏情况,则线性复杂度T(n)：

$$a(8n-2) \leq T(n) \leq b(8n-2)$$

2.4 运行时间的增长速率

- 增长速率不受constant factors(环境)及lower-order terms影响
- 改变软硬件环境可以影响绝对速率但是相对速率不会改变

2.5 渐近上界O

- 定义: 存在正常c, n_0使对所有的$n \geq n_0$都有$f(n) \leq cg(n)$则$f(n) = O(g(n))$
- 举个栗子:
  $f(n) = 2n + 10, g(n) = n$,then $2n + 10 = O(n)$
  $2n + 10 < cnn \geq \frac{10}{c-2}$,then $c = 3, n_0 = 10$
- 举个反栗:
  $f(n) = n^2, g(n) = n$,then $n^2! = O(n)$
  $n^2 \leq cn$,then $n \leq c$
  - 由于c是常数,总成立n>c,因此原结论不成立
- 再来几个栗子:
  - $7n - 2 = O(n);$
  - $3n^3 + 20n^2 + 5 = O(n^3);$
  - $3logn + 5 = O(logn)$

2.6 渐进下界Ω

- 定义：存在正常数c, n_0使对所有的$n \geq n_0$，都$f(n) \geq cg(n)$

2.7 渐近确界Θ

- 定义：存在正常数c_1和c_2和n_0，使对所有的$n \geq n_0$，都有$c_1 g(n) \leq f(n) \leq c_2 g(n)$
- 当且仅当$f(n) = O(g(n))$且$f(n) = \Omega(g(n))$时，有$f(n) = \Theta(g(n))$
  - $f(n) <= g(n)$,then $f(n) = O(g(n))$
  - $f(n) >= g(n)$,then $f(n) = \Omega(g(n))$
  - $f(n) = g(n)$,then $f(n) = \Theta(g(n))$
- 举个栗子：$5n^2$

- $\Omega(n^2)(c = 5, n0 = 1)$
- $\Omega(n)(c = 1, n0 = 1)$
- $\Theta(n)(c = 5, n0 = 1)$

Week 3 Linked Lists and Recursion

- Singly Linked Lists
- Doubly Linked Lists
- Recursions

3.1 抽象数据类型ADT:数据集合以及操作集合的统称

- Abstruct: ADT与实现形式无关, 具体实现可以有多种形式
- An ADT specifies:
  - Data stored;
  - Operations;
  - Error conditions;
- 定义ADT ： mathematical / interface
- 实现ADT ： class
- 这里举个栗子：

```
ADT name{
    object:
    ...
    relationship:
    ...
    operation set:
        operation1:
        operation2:
        ...
}
```

3.2 Singly Linked List

- Node=element+link to the next node
- Java 实现单链表

```
//结点类
public class Node{
    private String element;
```

```
    private Node next;
    public Node(String s,Node n){ //结点构造器
        element=s;
        next=n;
    }

    //返回当前结点的element
    public String getElement(){return element;}
    //返回下一个结点
    public Node getNext(){return next;}
    //设置当前结点的element
    public void setElement(String s){elemtnt=s;}
    //设置下一个结点
    public void setNode(Node newN){next=newN;}
}

//链表类
public class SingleLinkedList{
    protected Node head; //注意这里用protected更好,同包的所有类都可以用
    protected long size; //numbers of nodes in the list

    public SingleLinkedList(){ //默认构造器,创建一个空链表
        head=null;
        size=0;
    }
}
```

3.2.1 表头插入Node v

- v.next指向原表头Node，head指针指向Node v

```
Algorithm addFirst(v){
    v.setNext(head); // make v  point  to the old head node
    head = v;  // make head point to the new node
    size =size+1; // increment the node count
}
```

3.2.2 表头删除

- 将表头Node.next指向null,head指针指向第二个Node

```
Algorithm removeFirst(){
    if ( head == null )
        Indicate an error: the list is empty;
    t = head ;  //原首结点
    head = head.getNext(); //head指向第二个Node
    t.setNext(null);   //原表头Node的next指向Null
    size = size - 1;
}
```

3.2.3 尾部插入

- 创建链接指向null的新结点，旧表尾Node的链接指向新Node，tail指针指向新结点

```
Algorithm addLast(v){
    v.setNext(null); //make the new node v point to null object
    tail.setNext(v); //make the old tail node point to the new node
    tail = v ; //make tail point to the new node
    size = size +1;
}
```

3.2.4 尾部删除

- 移除结尾需要先遍历整个链表找到倒数第二结点 $O(n)$
- 可以参考下后面的双链表，直接`getPrevNode`即可

```
Algorithm removeLast(){
    if(size=0)
        Indicate an error: the list is empty;
    else{
        if (size=1)
            head = null; //删除唯一一个Node
        else{
            t = head;//t指向head
            while(t.getNext() != null){//这个主要目的是找到倒数第二结点
                s = t;
                t = t.getNext();
            }
            s.setNext(null);    //将倒数第二个结点指向null
            tail=s;
        }
        size = size - 1;
    }
}
```

3.3 Doubly Linked List

- Node=previous link+element+next link
- 头尾哨兵：header，trailer
- java 实现双链表

```
public class DNode{
    protected String element;
    protected DNode prev,next;
    public DNode(String e,,DNode p,DNode n){
        element=e;
        prev=p;
```

```
            next=n;
        }

        public String getElement(){return element;}
        public DNode getPrev(){return prev;}
        public DNode getNext(){return next;}
        public void setElement(String newE){elemtnt=newE;}
        public void setPrevNode(Node newN){prev=newN;}
        public void setNextNode(Node newN){next=newN;}
    }

    public class DoublyLinkedNode{
        protected Node head; //如果用private->只有当前class可以用这个变量
        protected long size;
        public DoublyLinkedNode(){
            head=null;
            size=0;
        }
    }

    //调用
    DoublyLinkedNode fucker=new DoublyLinkedNode()
    fucker.setElement(...);
    fucker.setNextNode(...);
    //建议使用泛型, 参考后面Stack/queue部分
```

3.3.1 头部插入node v

- 将v插入哨兵header与原先首结点w之间

```
Algorithm addFirst(v){
    w=header.getNextCode(); //找到首结点w(header只是个特殊链接,非首结点)
    v.setNextNode(w);
    v.setPrevNode(header);
    header.setNextNode(v);
    size =size+1;
}
```

3.3.2 中部插入

- 在Node v及其后结点Node w之间插入Node z

```
Algorithm addAfter(v,z){
    w=v.getNextNode(); //将v后结点设为w
    z.setPrevNode(v); //将z的前结点设为v
    z.setNextNode(w); //z的后结点设为w,即原来v后面指向的链接
    w.setPrevNode(z); //w的前结点指向z
    v.setNextNode(z); //v的后结点指向z
    size=size+1;
```

```
}
```

3.3.3 尾部删除

- 原尾结点前链接指向null; 倒数第二个结点指向trailer,成为新的尾结点

```
Algorithm removeLast():{
    if (size = 0)
        Indicate an error: this list is empty;
    v = trailer.getPrevNode()     //设v为结尾哨兵的前链接
    u = v.getPrevNode();//u为v的前结点
    trailer.setPrevNode(u);  //u与trailer互相链接,u成为新的尾结点
    u.setNextNode(trailer);
    v.setPrevNode(null);//v的前后结点指向null
    v.setNextNode(null);
    size = size -1;
}
```

3.3.4 中间删除

- 被删除结点的前后结点互相链接,被删除点的前后链接指向null

```
Algorithm remove(v){
    u = v.getPrevNode()      //u为v的前结点
    w = v.getNextNode();   //v为u的后结点
    w.setPrevNode(u);        //前后结点互相链接
    u.setNextNode(w);
    v.setPrevNode(null);    // 删除v
    v.setNextNode(null);
    size = size -1;
}
```

3.4 Recursion

- Base cases+ Recursive calls
  - Base cases:没有递归,如f(1)=1
  - Recursive calls:最终返回base cases,如f(n)=n*f(n-1)
- 举个栗子:斐波那契数列

$$F_0 = 1, F_1 = 1, F_N = F_{N-1} + F_{N-2}$$

- 普通递归 $O(2^k)$

```
Algorithm BinaryFib(k){
    Input : Nonnegative integer k
    Output : The kth Fibonacci number Fk
```

```
    if ( k =0 or 1) return 1;
    else
        return BinaryFib(k - 1) + BinaryFib(k - 2);
}
```

- 线性递归 $O(k)$

```
Algorithm LinearFibonacci(k):{
    Input : A nonnegative integer k
    Output : Pair of Fibonacci numbers (Fk, Fk-1)
    if  (k = 1) return (k, 0);
    else{
        (i,j)=LinearFibonacci(k - 1);//i=Fk-1,j=Fk-2
        return (i+j,i);//i+j=Fk,i=Fk-1
    }
}
```

Week 4 Stacks , Queues , Lists and Iterators

4.1 Stacks - LIFO

4.1.1 the Stack ADT

- main stack operations:
    - void push(object) //insertion at last
    - object pop() //deletion at last
- Auxiliary stack operations:
    - object top() //仅仅展示栈顶元素而不删除
    - int size()
    - boolean isEmpty()
- Applications:
    - 浏览历史;
    - 文本编辑器撤销操作;
    - 其他算法或数据结构的辅助

4.1.2 Stack Interface in Java

```
public interface Stack {
    public int size();
    public boolean isEmpty();
    public Object top() throws EmptyStackException;
    public void push(Object o);
```

```
    public Object pop() throws EmptyStackException;
}
```

- 对于空栈,pop()与top()是不能操作的 : throws EmptyStackException
- 对于满栈,push()是不能操作的 throws FullStackException
- 但是只有数组栈需要考虑溢出，而链表栈不需要考虑这个！

4.1.3 Array-bases Stack

- 从左至右添加元素,并使用变量t来追踪栈顶元素的位置S[t]

```
Algorithm size()
{   return t + 1;}

Algorithm pop()
{
    if (isEmpty()==true)
        throw EmptyStackException;
    else{
        t = t-1;
        return S[t + 1];//即原来的栈顶元素t
    }
}

Algorithm push(o)
{
    if (t=S.length-1)
        throw FullStackException;
    else
    {   t = t + 1;
        S[t] = o;
    }
}
```

- Preference:
  - size()=n;
  - 空间复杂度O(n);
  - 每次操作时间复杂度O(1);
- Limitation:
  - 预先确定栈容量且不能改变 : final

- 满栈不能push：Exception Overflow

- 使用java实现数组栈

```java
public class ArrayStack implements Stack{
    //作为子类,Stack的元素都被保留
    private Object S[];
    private int top=-1; //初始化栈顶元素
    public ArrayStack(int capacity){ //constructor
        S=new Object[capacity];
    }

    public Object pop() throws EmptyStackException{
        if(isEmpty()==true)
            throws EmptyStackException("Empty stack!")
        Object temp = S[top];
        S[top]=null; //删除栈顶元素
        top--;
        return temp; //返回之前被删除的栈顶元素
    }
}
```

4.1.4 Linked List-based Stack

- 栈顶元素:L.head;
- 使用实例变量top表示当前的size;
- push:头部插入;
- pop:头部删除;
- 和数组栈相比链表栈不会发生栈溢出

- Java 实现链表栈:

```java

//先建立泛型结点class
public class Node<Type>{
    //声明实例变量element,next
    private Type element;
    private Node<Type> next;

    //构造器 : overloading
    //先创建一个空Node:可以在之后调用setElement/setNext
    public Node(){this(null,null);}
    //创建一个带有给定内容和链接的Node
    public Node(Type e,Node<Type> n){
        element=e;
        next=n;
    }

    //methods:
```

```
    public Type getElement() { return element; }
    public Node<Type> getNext() { return next; }
    public void setElement(Type newElem) { element = newElem; }
    public void setNext(Node<Type> newNext) { next = newNext; }
}

//建立链表栈
public class NodeStack<Type> implements Stack<Type> {
    protected Node<Type> top;
    protected int size;

    public NodeStack{//构造器:空栈
        top=null;
        size=0;
    }


    public int size(){return size;}
    public boolean isEmpty(){
        if(top==null) return true;
        else return false;
    }
    public void push(Type element){
        Node<Type> v=new Node<Type>(element,top);//Node v的next被定义为top(当前栈顶元素)
        top=v;//使用v替换->v变为栈顶元素
        size++;
        }
    public Type pop() throws EmptyStackException{
        if(isEmpty()==true) throw new EmptyStackException("Empty stack!");
        Type temp=top.getElement();
        top=top.getNext();//删除原首结点
        size--;
        return temp;//显示被删除原首结点的内容
    }
}
```

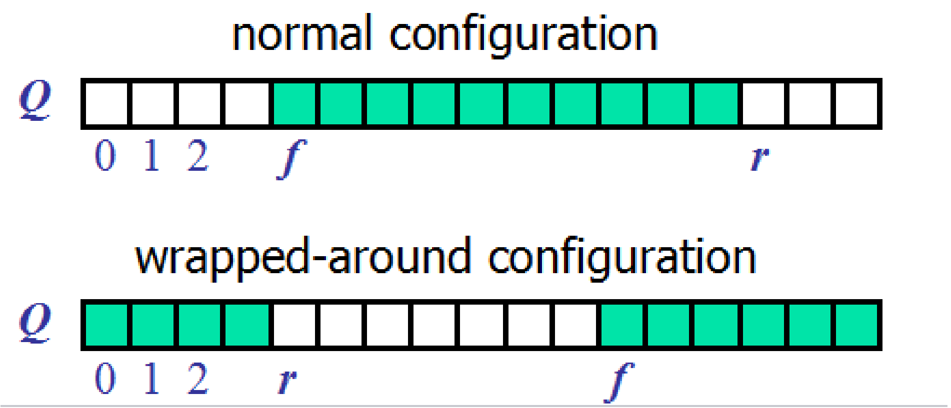
4.2 Queues - FIFO

- Main operations:
  - void enqueue(object)
  - object dequeue() throws EmptyQueueException
  - object front() throws EmptyQueueException
  - int size()
  - boolean isEmpty()

- Empty queue throws an EmptyQueueException
- Applications:
    - Waiting lists,bureaucracy
    - Access to shared resources(printer)
    - Multiprogramming

4.2.1 Array-based Queue

- Variables:
    - array size : N
    - f : 首元素位置
    - r : 尾元素后面的位置，始终为空，从此处加入元素



- Operations:

```
Algorithm size(){
    return (N - f + r) mod N;}

Algorithm isEmpty(){
    return (f = r); }

Algorithm enqueue(o){
    if (size() = N-1)
        throw FullQueueException;
    else{
        Q[r] = o;
        r= (r+1) mod N ;
    }
}

Algorithm dequeue(){
    if (isEmpty()=true)
        throw EmptyQueueException
    else{
        o = Q[f];
        f = (f + 1) mod N;
```

```java
            return o;
        }
    }
```

4.2.2 Queue Interface in Java

- 注意: 和stack不同, java不能直接初始化queue
- 需要自己建立接口或者将Queue初始化为链表

```java
import java.util.LinkedList;
Queue q = new LinkedList();
```

```java
public interface Queue {
    public int size();
    public boolean isEmpty();
    public Object front() throws EmptyQueueException;
    public void enqueue(Object o);
    public Object dequeue() throws EmptyQueueException;
}
```

4.2.3 Linked List-based Implementation of Queue

- enqueue尾部插入 , dequeue头部删除
- 注意同样需要建立泛型Node
- 单链表队列操作时间 $O(1)$

```java
public void enqueue(Type element){
    Node<Type> fucker = new Node<Type>();
    fucker.setElement(element);
    fucker.setNext(null);
    if (size == 0) head = fucker; //空队列 , fucker为唯一的元素
    else tail.setNext(fucker); //原尾结点的链接设置为Node fucker
    tail = fucker; // 将Node fucker作为新的尾结点
    size++;
}

public Type dequeue() throws EmptyQueueException {
    if (size == 0) throw new EmptyQueueException("Queue is empty.");
    Type currentHead = head;
    currentHead.setNext(null);
    head = head.getNext();
    size--;
    if (size == 0) tail = null;  //the queue is now empty
    return currentHead.getElement(); //返回原首结点的值
}
```

4.3 Lists

4.3.1 Array List

- Main operations
  - get(i)/set(i,e)/add(i,e)/remove(i)/resize()
  - 在某位置插入或删除时,该位置后面的元素都会发生位移
  - IndexOutOfBoundsException
- add(i,e) : 将插入点后面的元素移位,最坏情况为头部插入 $O(n)$

```
Algorithm add(i,e){
    if(size==N){ A.resize(); }
    for (j=n-1,n-2,...i){ //倒序:从最后一个元素开始变动一直到i后面第一个元素
        A[j+1]=A[j];
    }//A[n]=A[n-1],A[n-1]=A[n-2]...A[i+1]=A[i]->空出位置A[i]
    A[i]=e;
    n=n+1;
}
```

- remove(i) : 同样需要将删除点后面的元素移位,最坏情况为头部删除 $O(n)$

```
Algorithm remove(i){
    e=A[i];
    for ( j = i, i+1, ... , n-2 ){//从i后第一个元素开始变动一直到整个list最后一个元素
        A[j]=A[j+1];
    } //A[i]=A[i+1],...A[n-2]=A[n-1]
    n=n-1;
    return e;
}
```

- resize() : 当一个数组填满元素时, 将其更换为容量更大的数组

```
Algorithm resize(){
    create a new array B with size 2N;
    for ( j = 0, 1, ... , size-1)
    {    B[j]=A[j];}
    A = B;
    N=2N;
}
```

- Java 实现数组队列

```
//建立接口
public interface IndexList<E> {
    public int size();
```

```java
    public boolean isEmpty();
    public void add(int i, E e) throws IndexOutOfBoundsException;
    public E get(int i) throws IndexOutOfBoundsException;
    public E remove(int i) throws IndexOutOfBoundsException;
    public E set(int i, E e) throws IndexOutOfBoundsException;
}

//建立list class
public class ArrayIndexList<E> implements IndexList<E>{
    private E[] A;
    private int capacity = 16; //初始化数组A容量
    private int size = 0; //初始化元素数量
    public ArrayIndexList() {
        A = (E[]) new Object[capacity];
    }

    //method: add e at index r
    public void add(int r, E e) throws IndexOutOfBoundsException {
        checkIndex(r, size() + 1); //于判断r是否在size()+1范围内
        if (size == capacity){ //resize
            capacity *= 2;
            E[] B =(E[]) new Object[capacity]; //narrowing : Object的范围>B
            for (int i=0; i<size; i++) B[i] = A[i];
            A=B;
        }
        for (int i=size-1; i>=r; i--)
            A[i+1]=A[i];
        A[r]=e;
        size++;
    }

    //method : remove the element at place r
    public E remove(int r) throws IndexOutOfBoundsException {
        checkIndex(r, size());
        E temp = A[r];
        for (int i=r; i<size-1; i++)
            A[i] = A[i+1];
        size--;
        return temp;
    }
}
```

4.3.2 Comparation of the Strategies

- 分析对size为1的空数组进行n次push操作所需的总时间T(n)
- Incremental Strategy Analysis
  - 交换次数: $k = \frac{n}{c}$

$$T(n) \propto n + c + 2c + 3c + 4c + ... + kc = n + \frac{ck(1+k)}{2}$$

  - $T(n) = O(n + k^2)$,每次push的平均时间:$O(n)$
- Doubling Strategy Analysis
  - 交换次数: $k = logn$

$$T(n) \propto n + 1 + 2 + 4 + ... + 2^k = n + 2^{k+1} - 1 = 2n - 1$$

  - $T(n) = O(n)$, push的平均时间$O(1)$

4.3.3 Position->给出数组或链表中指定元素的所在位置

- Interfaces : Position ; PositionList
- Java Interface for the Position ADT

```
public interface Position<E>{
    E element() throws InvalidPositionException;//get element
}
```

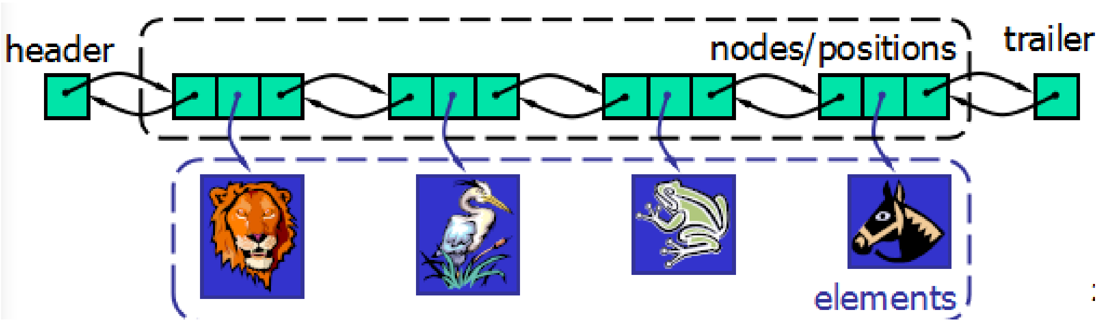
- Node List ADT
  - Generic methods: size(),isEmpty()
  - Accessor methods: first(),last(),prev(),next()
  - update methods: set(p,e).addBefore(p,e),addAfter(p,e),addFirst(e),addLast(e),remove(p)
- Java interface for the Node List ADT

```
public interface PositionList<E>{
    public int size();
    public boolean isEmpty();

    //返回头尾结点
    public Position<E> first();
    public Position<E> last();

    //给出指定Node的前一个/后一个Node
    public Position<E> next(Position<E> p) throws InvalidPositionException, BoundaryViolationException;
    public Position<E> prev(Position<E> p) throws InvalidPositionException, BoundaryViolationException;

    //头尾插入,返回插入位置Position<E> p
    public void addFirst(E e);
    public void addLast(E e);

    //在给定结点前/后插入新结点
    public void addAfter(Position<E> p,E e) throws InvalidPositionException;
```

```
        public void addBefore(Position<E> p,E e) throws InvalidPositionException;

        //删除给定位置的结点,返回该结点值
        public E remove(Position<E> p) throws InvalidPositionException;

        //替换某个结点的element并返回旧值
        public E set(Position<E> p, E e) throws InvalidPositionException;
        }
```

4.3.4 Doubly Linked List Implementation

- element+link to prev+link to next
- Special nodes trailer/header



- Insertion:

```
Algorithm insertAfter(p,e){
    Create a new node v;
    v.setElement(e);
    fucker=p.getNext();
    v.setPrev(p);
    v.setNext(fucker);
    fucker.setPrev(v);
    p.setNext(v);
    return v;
}
```

- Deletion

```
Algorithm remove(p){
    pElement=p.element;
    pPrev=p.getPrev();
    pNext=p.getNext();
    pPrev.setNext(pNext);
    pNext.setPrev(pPrev);
    p.setPrev(null);
    p.setNext(null);
    return pElement;
}
```

4.3.5 实现Position interface ADT

```java
public class DNode<E> implements Position<E>{
    private DNode<E> prev,next;
    private E element;

    //构造器
    public DNode(DNode<E> newPrev,DNode<E> newNext,E elem){
        prev=newPrev;
        next=newNext;
        element=elem;
    }

    public E element() throws InvalidPositionException {
        if ((prev == null) && (next == null))
            throw new InvalidPositionException("Position is not in a list!");
        return element;
    }

    public DNode<E> getNext() { return next; }
    public DNode<E> getPrev() { return prev; }

    public void setNext(DNode<E> newNext) { next = newNext; }
    public void setPrev(DNode<E> newPrev) { prev = newPrev; }
    public void setElement(E newElement) { element = newElement; }
}
```

4.3.6 实现NodePositionList ADT

```java
public class NodePositionList<E> implements PositionList<E>{
    protected int numElements;
    protected DNode<E> header,trailer;//special sentinels

    //空list构造器
    public NodePositionList(){
        numElements=0;
        header=new DNode<E>(null,null,null);
        trailer=new DNode<E>(header,null,null);
        header.setNext(trailer);
    }

    //检查position的合理性:valid->convert to Node
    protected DNode<E> checkPosition(Position<E> p) throws InvalidPositionException{
        if(p==null) throw new InvalidPositionException("Null position");
        if(p==header) throw new InvalidPositionException("Special sentinel is not a valid position");
        if(p==trailer) throw new InvalidPositionException("Special sentinel is not a valid position");
```

```java

        try{//注意p->temp为narrowing->显式转换
            DNode<E> temp=(DNode<E>) p;
            if((temp.getPrev()==null||(temp.getNext()==null))
                throw new InvalidParameterException("not belong to the NodeList");
            return temp;
        } catch(ClassCastException e){
            throw new InvalidPositionException("wrong type");
        }
    }

    //basic methods
    public int size(){return numElements;}
    public boolean isEmpty(){return(numElements==0);}

    //the first position in the list
    public Position<E> first() throw EmptyListException{
        if(isEmpty()) throw new EmptyListException("empty list");
        return header.getNext();
    }

    //return the position before the given one
    public Position<E> prev(Position<E> p) throws InvalidPositionException,BoundaryViolationException{
        DNode<E> v=checkPosition(p);
        DNode<E> prev=v.getPrev();
        if(prev==header) throw new BoundaryViolationException("Can't advance past the beginning of the list");
        return prev;
    }

    //在指定位置前面插入元素
    public void addBefore(Position<E> p,E element) throws InvalidPositionException{
        DNode<E> v=checkPosition(p);
        numElements++;
        DNode<E> newNode=new DNode<E>(v.getPrev(),v,element);
        v.getPrev().setNext(newNode);
        v.setPrev(newNode);
    }

    //头部插入
    public void addFirst(E element){
        numElements++;
        DNode<E> newNode=new DNode<E>(header,header.getNext(),element);
        header.getNext().setPrev(newNode);
        header.setNext(newNode);
    }

```

```java
        //指定位置删除，并返回被删除的值
        public E remove(Position<E> p) throws InvalidPositionException {
            DNode<E> v = checkPosition(p);
            numElements--;
            DNode<E> vPrev = v.getPrev();
            DNode<E> vNext = v.getNext();
            vPrev.setNext(vNext);
            vNext.setPrev(vPrev);
            E vElement = v.element();
            v.setNext(null);
            v.setPrev(null);
            return vElement;
        }

        //指定位置替换，并返回原来的值
        public E set(Position<E> p,E element) throws InvalidPositionException{
            DNode<E> v=checkPosition(p);
            E oldElement=v.element();
            v.setElement(element);
            return oldElement;
        }
    }
```

- Doubly linked list的性能:
  - 空间复杂度: 整体$O(n)$, 每个位置$O(1)$
  - 时间复杂度: List ADT $O(1)$, Position ADT(element()操作
  - 对比数组队列，链表队列不需要牵一发而动全身

4.4 Iterators:逐个地展示元素

- An iterator consists of:
  - a sequence S;
  - a current element in S;
  - methods: go to next element, and set it as current element
- Methods in the Iterator ADT:
  - boolean hasNext()
  - object next()

```java
if(hadNext()){
    next();
}
```

4.4.1 Simple Iterators in Java

```java
public interface PositionList <E> extends Iterable <E>{
    //all the other methods of the list ADT
    public Iterator<E> iterator();
}


public static <E> String toString(PositionList<E> l){
    Iterator<E> it=l.iterator();
    String s="[";
    while (it.hasNext()){
        s+= it.next();
        if(it.hasNext())
            s+= ",";
    }
    s+="]";
    return s;
}
```

4.4.2 Implementing Iterators

- 两个概念:
  - snapshot:freezes the contents of the data structure at a given time;
  - dynamic:follows changes to the data structure;
- 主要区别:
  - 初始创建迭代器实例时都做了什么工作;
  - 每次执行next()更新迭代器做了什么工作;
- Snapshot
  - 复制迭代器对象刚被创建时的元素集合;
  - 原始集合的改变不会影响迭代器;
  - 简单易行,但需要对初始结构进行遍历$O(n)$
- Dynamic
  - 不进行复制,只是在执行next()时对原始结构的一小部分进行遍历;
  - 复杂度低$O(1)$;
  - 缺点：原始结构变化了其行为也会受影响;

Week 5 Trees

- Tree ADT
- Preorder Traversal
- Inorder Traversal

- Postorder Traversal

5.1 Tree Terminology

- Root: Node without parent
- Internal Node: Node with at least one child
- External Node(leaf): Node without child
- Depth of a node : number of ancestors
    - root的深度是0;
    - 子结点深度 = 父结点的depth + 1;
- Height of a tree : maximum depth of any node
    - Height of a leaf : 0;
    - 某结点高=子结点高度 + 1;

5.2 Tree ADT

- Generic methods:
    - integer size();
    - boolean isEmpty();
    - Iterator elements();
    - Iterator positions();
- Accessor methods:
    - position root();
    - position parent(p);
    - positionIterator children(p)
- Query methods:
    - boolean isInternal(p);
    - boolean isExternal(p);
    - boolean isRoot(p);
- Update method:
    - object replace(p,o);

5.2.1 Preorder Traversal

- 注意preorder/postorder/inorder都是DFS遍历
- 先根遍历(先访问后递归) : 始于根结点,依次遍历每棵子树(父结点优先,左子树优先)

```
Algorithm preOrder(v){
    visit(v); //先访问当前结点v
```

```
    for each child w of v:
        preOrder(w);//使用同样办法遍历v的子树
}
```

- Application : 打印文件1-1.1-1.2-2-2.1-2.2-2.3-3-3.1...

5.2.2 Postorder Traversal

- 后根遍历(先递归后访问) : 始于叶结点,只有访问完某个结点的所有子树才会访问该结点

```
Algorithm postOrder(v){
    for each child w of v:
        postOrder(w);
    visit(v);
}
```

- Application : 从子文件夹开始计算每个文件占用的空间

5.2.3 宽度优先遍历

```
/*宽度优先遍历树
 * 优先遍历深度相同的结点
 * 基于queue:每次先访问这一层的所有结点(dequeue)
 * 访问一个结点同时将其所有子节点加入队尾
 * 根据FIFO规则, 在上一层结点全部出队之前不会访问下一层结点
 */

import java.util.LinkedList;
import java.util.Queue;
public class BreadthFirstTreeTraversal {
    Initialize queue Q storing root();
    while (!Q.isEmpty()){
     p=Q.dequeue(); //p is the oldest in the queue
     visit(p);
     for each child in children(p){
        Q.enqueue(c);
        }
    }
}
```

5.3 Binary Trees

- left child < parent < right child (先左后右)
- applications:
  - arithmetic expressions : internal为操作符,external为操作数
  - decision processes : internal为questions,external为decisions

searching

5.3.1 Properties of proper binary trees

- 完全二叉树：所有子树都止于external
- 一些概念:
  - N : number of nodes;
  - N_e : number of external nodes;
  - N_i : number of internal nodes;
  - H : height
- 各概念之间的关系 (可以互相推导)

1. $h+1 \le n \le 2^{h+1}-1$
2. $1 \le n_E \le 2^h$
3. $h \le n_I \le 2^h - 1$
4. $\log(n+1) - 1 \le h \le n-1$

Also, if T is proper, then T has the following properties:

1. $2h+1 \le n \le 2^{h+1}-1$
2. $h+1 \le n_E \le 2^h$
3. $h \le n_I \le 2^h - 1$
4. $\log(n+1) - 1 \le h \le (n-1)/2$

- 注意完全二叉树与满二叉树的区别:
  - 完全二叉树指除了最后一层其他层都填满元素的二叉树
  - 满二叉树是特殊的完全二叉树,最后一层也被填满
  - 若满二叉树层数为k, 高度为h,则:
    - $k = h$
    - 深度x的结点数量为 2^x
    - 总结点数量为 $2^{x+1} - 1$

5.3.2 BinaryTree ADT

- BinaryTree ADT extends Tree ADT
- Additional methods:
  - position left(p);
  - position right(p);
  - boolean hasLeft(p);
  - boolean hasRight(p);

5.3.3 Inorder Traversal

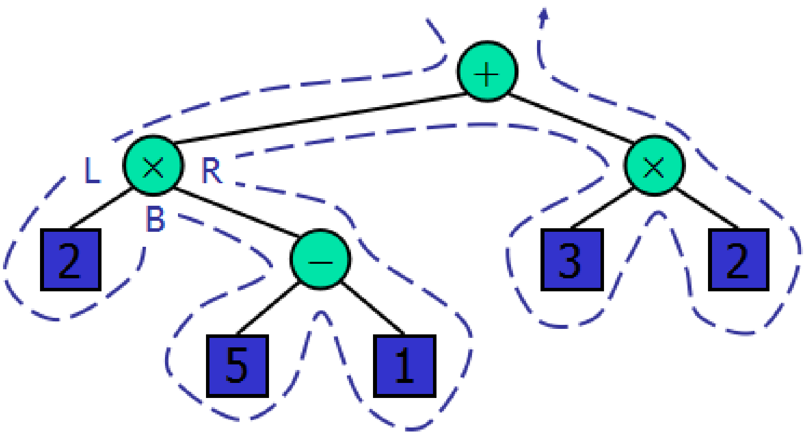
- 二叉树使用中序遍历(左递归，访问，右递归)：某结点的左子树->该结点->该结点的右子树

```
Algorithm inOrder(v){
    if(hasLeft(v))
        inOrder(left(v));
    visit(v);
    if(hasRight(v))
        inOrder(right(v));
}
```

- 之后给出了几个栗子
  - Print arithmetic expressions : inorder traversal
  - Evaluate Arithmetic Expressions ： postorder traversal

5.4 Eular Tour Traversal

- 从左到右一笔画
  - on the left : preorder
  - from below : inorder
  - on the right : postorder



- Template method pattern : 递归调用左右子结点

```
public abstract class EulerTour {
    protected BinaryTree tree;
    protected void visitExternal(Position p, Result r){}
    protected void visitLeft(Position p, Result r){ }
    protected void visitBelow(Position p, Result r) { }
    protected void visitRight(Position p, Result r) { }

    protected Object eulerTour(Position p) {
        Result r = new Result();
        if tree.isExternal(p) { visitExternal(p, r); }
        else {
```

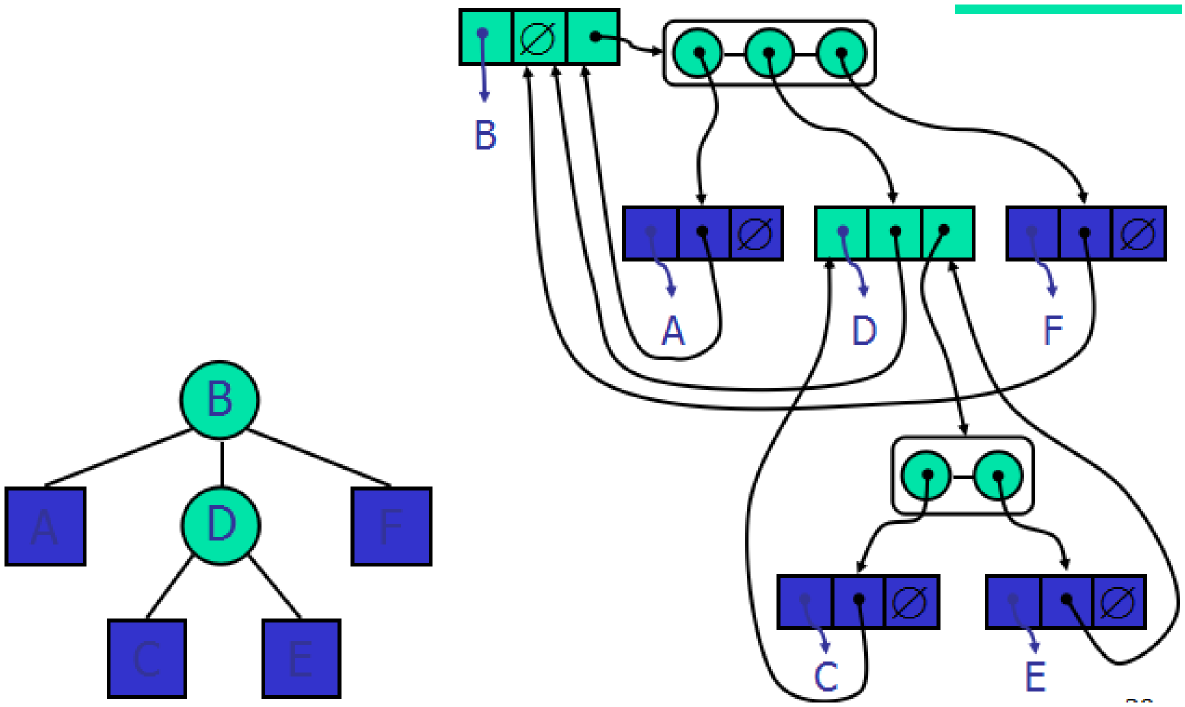
```
                visitLeft(p, r);
                r.leftResult = eulerTour(tree.left(p));
                visitBelow(p, r);
                r.rightResult = eulerTour(tree.right(p));
                visitRight(p, r);
                return r.finalResult;
            }
            ...
        }
    }
```

- Specializations of EulerTour, 基于以下假定
  - External nodes store Integer objects
  - Internal nodes store Operator objects supporting method operation(Integer,Integer)

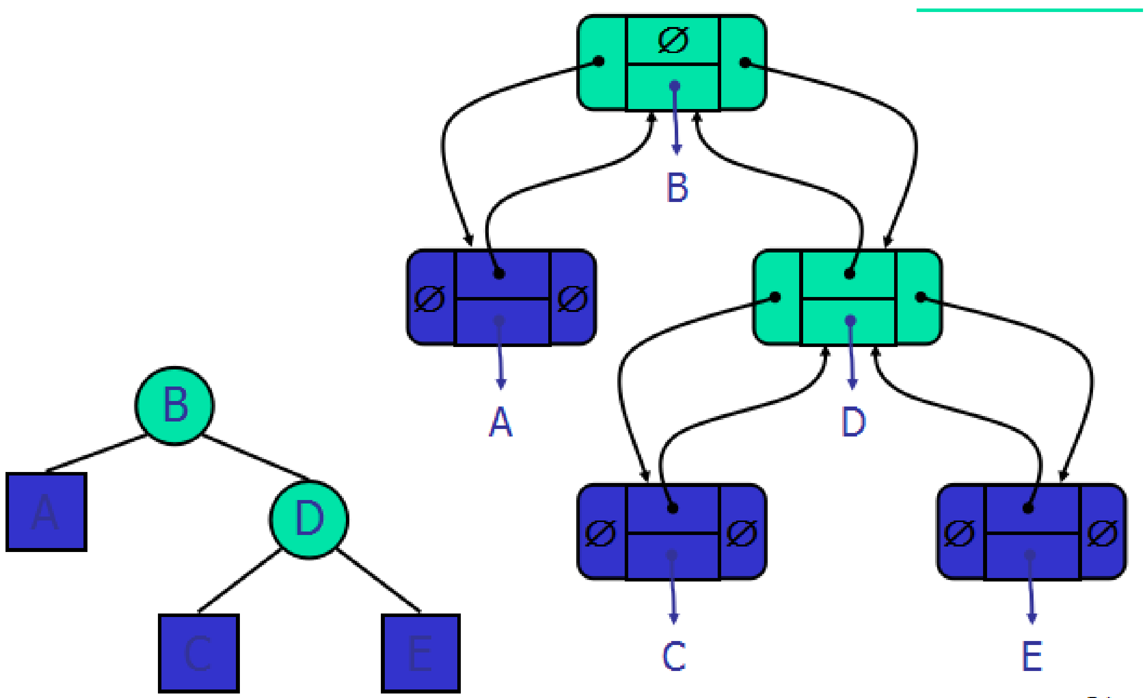
```
public class EvaluateExpression extends EulerTour{
    protected void visitExternal(Position p,Result r){
        r.finalResult=(Integer)p.element();
    }
    protected void visitRight(Position p,Result r){
        Operator op=(Operator)p.element();
        r.finalResult=op.operation((Integer)r.leftResult,(Integer)r.rightResult
);
    }
}
```

5.5 Linked Structure for Trees

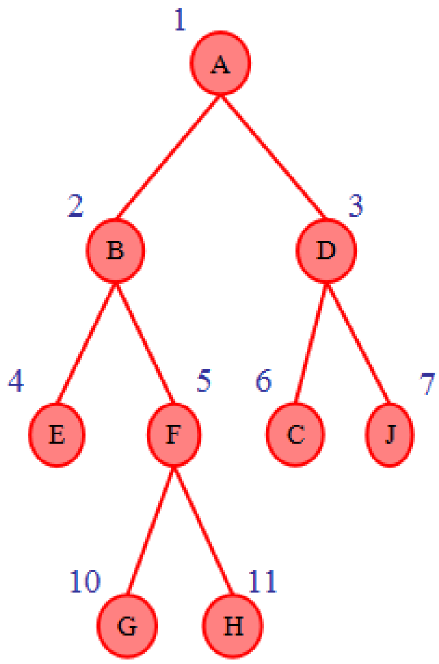
- tree node = element + parent node + sequence of children nodes
- Node objects implement the Position ADT



- Linked Structure for Binary Trees
    - 和前面的对比就是子结点被严格限制为左+右



- Array-Based Representation of Binary Trees
    - rank(root) = 1
    - rank(left node) = 2 * rank(parent)
    - rank(right node) = 2 * rank(parent) + 1



- 构建基于链表的二叉树,并实现前中后序遍历

```
/*
 * Ref: http://ocaicai.iteye.com/blog/1047397
 * 创建基于链表的二叉树,并实现前序中序后序遍历
 * 基本思路:对于index为k的结点, 其左子节点为2k+1,右子节点为2k+2
 */

package tree;
import java.util.LinkedList;
import java.util.List;

public class LinkedBinaryTree2 {
    private static class Node {
        Node leftChild;
        Node rightChild;
        int data;

        Node(int newData) {
            leftChild = null;
            rightChild = null;
            data = newData;
        }
    }
    private static List<Node> nodeList=null;
    protected Node root=null;
    public void buildBinaryTree(int[] myData){
        nodeList=new LinkedList<Node>();
        //将之前的数据全都存到结点里,再将结点存入到nodeList中
        for(int nodeIndex=0;nodeIndex<myData.length;nodeIndex++){
            Node myNode=new Node(myData[nodeIndex]);
```

```
                nodeList.add(myNode);
            }

            //使用nodeList构建树,相当于给每个父结点都添加子节点
            //首先构建除了最后一个父结点外的internal
            //假设total数量为n,internal数量为(n-1)/2
            //暂时排除最后一个父结点-->构建n/2-1个父结点
            for(int parentIndex=0;2*parentIndex+2<myData.length;parentIndex++){
                int leftIndex=2*parentIndex+1;
                int rightIndex=2*parentIndex+2;
                nodeList.get(parentIndex).leftChild=nodeList.get(leftIndex);
                nodeList.get(parentIndex).rightChild=nodeList.get(rightIndex);
            }
            //考虑最后一个父结点的两种情况
            //若元素数量为奇数则最后一个父结点是满的
            //若元素数量为偶数则最后一个父结点只有左子结点
            int lastParentIndex=myData.length/2-1;
            int leftIndex=2*lastParentIndex+1;
            nodeList.get(lastParentIndex).leftChild=nodeList.get(leftIndex);
            if (myData.length%2==1){
                int rightIndex=2*lastParentIndex+2;
                nodeList.get(lastParentIndex).rightChild=nodeList.get(rightIndex);
            }

            //最后给root赋值
            root=nodeList.get(0);
        }

        //PreorderTraversal
        public static void PreorderTraversal(Node currentNode){
            if(currentNode==null){
                return;
            }
            System.out.print(currentNode.data+" ");
            PreorderTraversal(currentNode.leftChild);
            PreorderTraversal(currentNode.rightChild);
        }

        //InorderTraversal
        public static void InorderTraversal(Node currentNode){
            if(currentNode==null){
                return;
            }
            InorderTraversal(currentNode.leftChild);
            System.out.print(currentNode.data+" ");
            InorderTraversal(currentNode.rightChild);
        }

        //PostorderTraversal
        public static void PostorderTraversal(Node currentNode){
```

```java
            if(currentNode==null){
                return;
            }
            PostorderTraversal(currentNode.leftChild);
            PostorderTraversal(currentNode.rightChild);
            System.out.print(currentNode.data+" ");
        }

        //测试用例
        public static void main(String[] args){
            LinkedBinaryTree2 myLBT1=new LinkedBinaryTree2();
            LinkedBinaryTree2 myLBT2=new LinkedBinaryTree2();
            int[] myData1={1,2,3,4,5,6,7,8,9};
            myLBT1.buildBinaryTree(myData1);
            System.out.println("Preorder of LBT1 is: ");
            myLBT1.PreorderTraversal(myLBT1.root);
            System.out.println("");
            System.out.println("Inorder of LBT1 is: ");
            myLBT1.InorderTraversal(myLBT1.root);
            System.out.println("");
            System.out.println("Postorder of LBT1 is: ");
            myLBT1.PostorderTraversal(myLBT1.root);

            int[] myData2={2,3,5,7,11,13,17,19,23,29,31,33,37,43,47,53,59};
            myLBT2.buildBinaryTree(myData2);
            System.out.println("Preorder of LBT2 is: ");
            myLBT2.PreorderTraversal(myLBT2.root);
            System.out.println("");
            System.out.println("Inorder of LBT2 is: ");
            myLBT2.InorderTraversal(myLBT2.root);
            System.out.println("");
            System.out.println("Postorder of LBT2 is: ");
            myLBT2.PostorderTraversal(myLBT2.root);
        }
    }
```

Week 6 Search Trees

- BST;
- AVL Trees;
- Splay Trees;
- (2,4) Trees;
- Red-Black Trees;

6.1 BST

- Nodes u,v,w : u 为 v 的左子结点 , w 为 v 的右子结点
- $key(u) <= key(v) <= key(w)$
- inorder traversal;

6.1.1 Ordered Dictionaries

- first();
- last();
- successors(k) : 升序排列大于k的所有元素
- predecessors(k): 降序排列小于k的所有元素
- successor(k):大于k的最小结点 (k右子结点的左子树末端)
- predecessor(k):小于k的最大结点 (k左结点的右子树末端)

6.1.2 Binary Search

- Search : $O(logn)$
- Insert & Remove : $O(n)
  - worst case: shift n/2 items after operation

6.1.3 Search

- 算法分析:
  - 始于root,从上到下搜索key;
  - 是否访问下一个node取决于k与当前结点的key的比较结果；
  - 到达leaf还没找到 , return null;

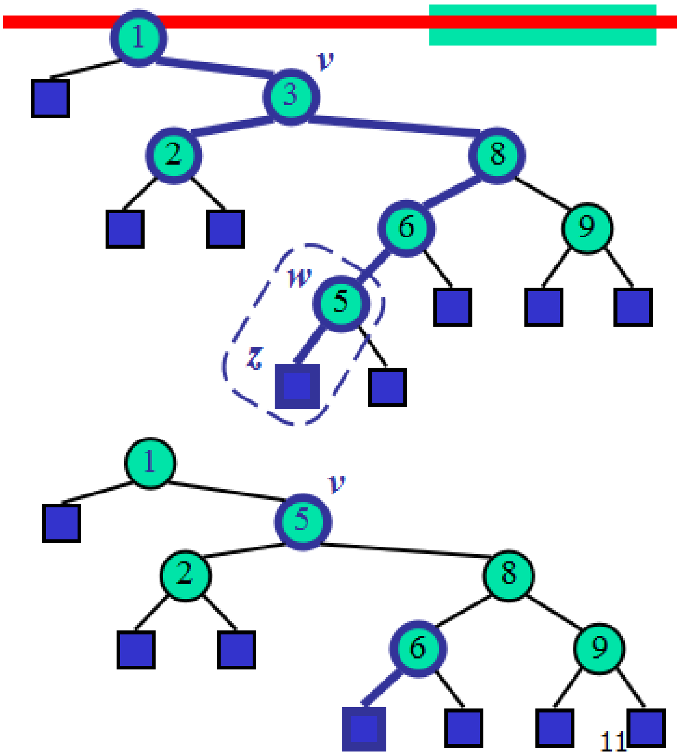
```
Algorithm TreeSearch(k,v){//Start from node v
    if(T.isExternal(v))
        return v;
    if(k<key(v))
        return TreeSearch(k,T.left(v));
    else if(k>key(v))
        return TreeSearch(k,T.right(v));
    else
        return v;//Match
    return false; //No match
}
```

6.1.4 Insertion Node v at Key k

- 算法分析:
  - 先对key k进行搜索(TreeSearch);
  - 若tree中不存在key k,插入到合适的位置;
  - 若已经存在key k,更改储存内容为Node v;

6.1.5 Deletion

- 算法分析:
  - 先进行搜索, 确保树中存在key k并确定其位置Node v;
  - 如果存在k, 对相应Node v 按以下几种情况讨论:
    - 删除v时将v连接的叶结点w一并删除;
    - 如果v只有一棵子树w，则用这一棵子树替换v;
    - v有两棵子树：寻找 v 的predecessor 或者 successor 替换 v;
- 查询predecessor以及successor:
  - succcessor：右子结点的左子树末端;
  - predecessor：左子结点的右子树末端



6.1.6 BST性能分析

- space complexity : $O(n)$;
- size/isEmpty : $O(1)$;
- find/inset/remove : $O(h), h \leq \frac{n-1}{2}$;
- worse case (inbalance) : $O(n)$
- best case (AVL) : $O(logn)$

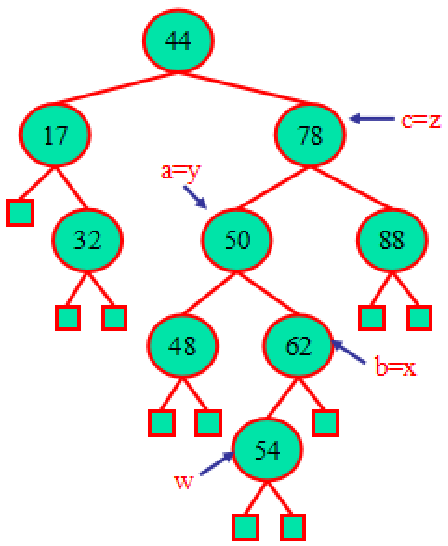
6.2 AVL Trees

- Balanced：任何结点的两棵子树高度差不超过1
- AVL高度必然为$O(logn)$, 证明如下:
  - 假定n(h)为高度为h的AVL树的internal nodes 数量;

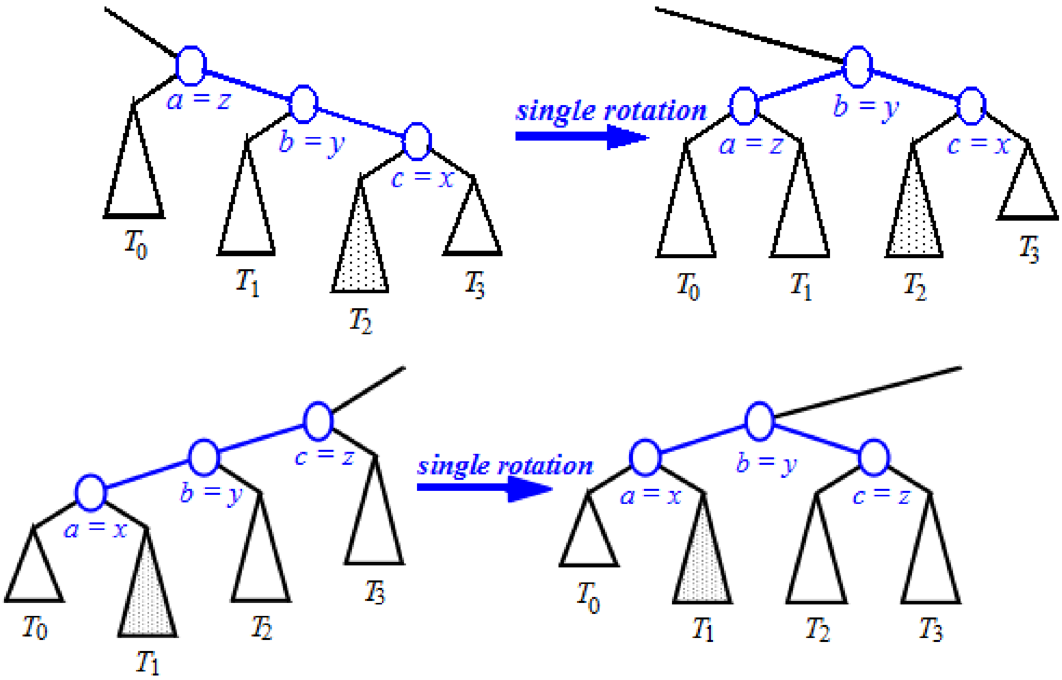
- $n(1) = 1, n(2) = 2$
- 对于$n > 2$, 有$n(h) = 1 + n(h-1) + n(h-2)$
- $n(h-1) > n(h-2)$, 得到 $n(h) > 2n(h-2)$
- $n(h) > 2n(h-2) > 4n(h-4) > 8n(h-6) > \ldots > 2^i n(h-2i)$
- 令$i = h/2 - 1$, 得到 $n(h) > 2^{\frac{h}{2}-1}$, 即$h < 2logn(h) + 2$

6.2.1 Insertion in an AVL Tree

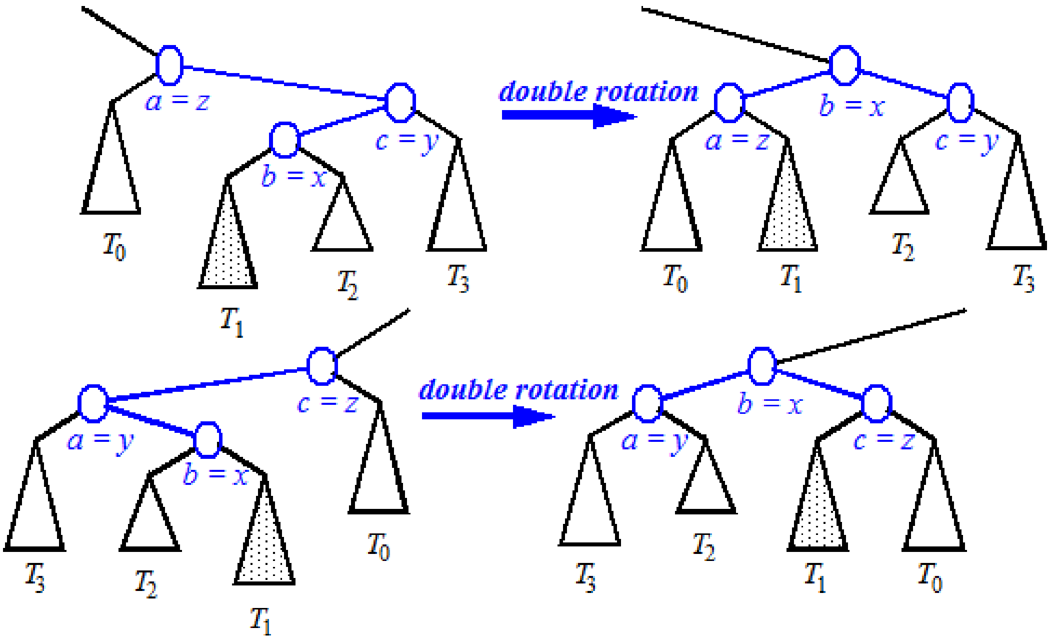
- 插入操作类似BST，为了防止平衡被破坏，需要进行Trinode Restructuring
- Trinode Restructuring：
  - 首先检查新结点所有的长辈结点看是否破坏平衡性;
  - 假定长辈结点z破坏了平衡性(两棵子树高度差>1), 需要进行重构;
  - 将z子结点中高度比较大的定为y (就是比较长的那棵子树);
  - 将y子结点中高度比较大的定为x;
  - 根据x,y,z的相对大小，通过rotation将x,y,z重构为inOrder结构;



- Rotation:
  - single rotation :
    - 同侧结点 $x < y < z$ 或者 $x > y > z$;
    - 中间的y结点变为x,z的父结点;



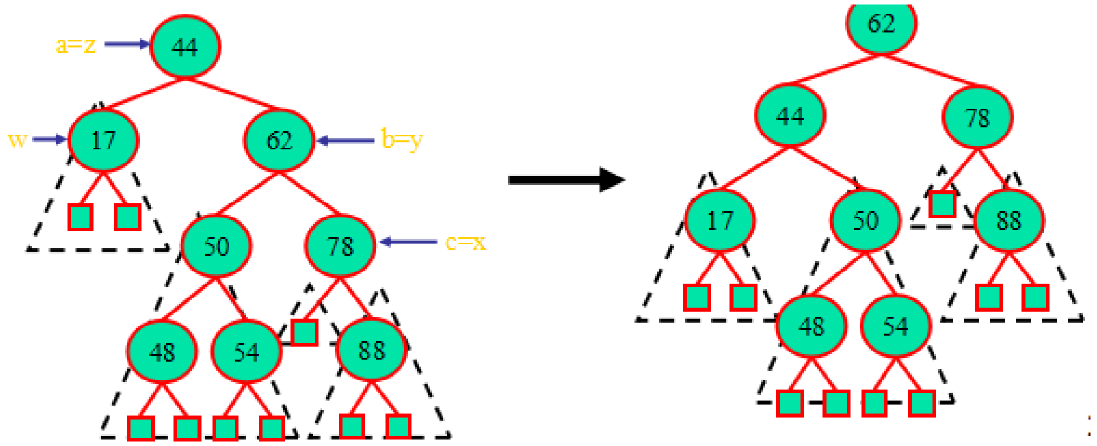
- double rotation :
  - 一左一右
  - x结点变为y,z的父结点;
  - x为z的predecessor/successor



6.2.2 Removal in an AVL Tree

- 前面操作同BST，为防止不平衡，需要进行Rebalancing
- Rebalancing after removal
  - 检查被删除结点的所有长辈结点是否平衡;
  - 假设z为不平衡结点，其子结点w为被删除结点的长辈结点;
  - 类似重构, 令z子结点中比较高的为y(即另一个子结点)，y子结点中比较高的为x;

  - 对x,y,z进行旋转重构;
  - 不断向上检查，直到root也是平衡的;



6.2.3 实现AVL Tree

```
public class AVLTreeMap<K,V> extends TreeMap<K,V>{
    //两个构造器
    public AVLTreeMap(){
        super();
    }
    public AVLTreeMap(Comparator<K> comp){
        super(comp);
    }

    //method 1: 返回位置p的高度
    protected int height(Position<Entry<K,V>> p){
        return tree.getAux(p);
    }

    //method 2: 重新计算指定位置的高度
    //这个和之前普通树的计算方法一样:两棵子树高度比较大的导读加一
    protected void recomputeHeight(Position<Entry<K,V>> p){
        tree.setAux(1+Math.max(height(left(p),height(right(p)))));
    }

    //method 3: 判断当前位置是否平衡
    protected boolean isBalanced(Position<Entry<K,V>> p){
        return Math.abs(height(left(p))-height(right(p)))<=1;
    }

    //method 4: 返回位置p两棵子树高度比较大的那棵
    protected Position<Entry<K,V>> tallerChild(Position<Entry<K,V>> p){
        if(height(left(p))>height(right(p))){
            return left(p);
        }
        if(height(left(p))<height(right(p))){
            return right(p);
```

```
            }
            //接下来是p两棵子树高度相同的情况
            if(isRoot(p)){//返回左右都一样
                return left(p);
            }
            //下面这个判断还不大清楚,为什么要这么搞呢
            if(p==left(parent(p))){//p是其父结点的左子树
                return left(p);
            }
            else{//p是其父结点的右子树
                return right(p);
            }
        }

        //method 5: 对于不平衡的AVL,重新将其回复到平衡状态
        protected void rebalance(Position<Entry<K,V>> p){
            int oldHeight,newHeight;
            //当p之前的高度和重新计算后的高度不等且p非叶结点时执行循环
            do{
                oldHeight=height(p);
                //首先先判断当前结点p是否平衡
                if(!isBalanced(p)){//p位置发生不平衡
                    //将p高度比较高的子结点的高度比较高的子结点进行重构,并赋值给p
                       p=restructure(tallerChild(tallerChild(p)));
                    //重新计算p两个子节点的高度
                    recomputeHeight(left(p));
                    recomputeHeight(right(p));
                }
                //若p位置是平衡的,计算p的高度并将p赋值为其父结点
                //不断向上检查是否平衡
                recomputeHeight(p);
                newHeight=height(p);
                p=parent(p);
            }while(oldHeight!=newHeight && p!=null);
        }

        protected void rebalanceInsert(Position<Entry<K,V>> p){
            rebalance(p);//插入结点,从该结点位置开始判断
        }
        protected void rebalanceDelete(Position<Entry<K,V>> p){
            if(!isRoot(p)){
                //删除结点从该结点父结点开始判断,如果本身就是根节点则什么都没有了
                rebalance(parent(p));
            }
        }

    }
```

6.2.4 AVL Tree的性能

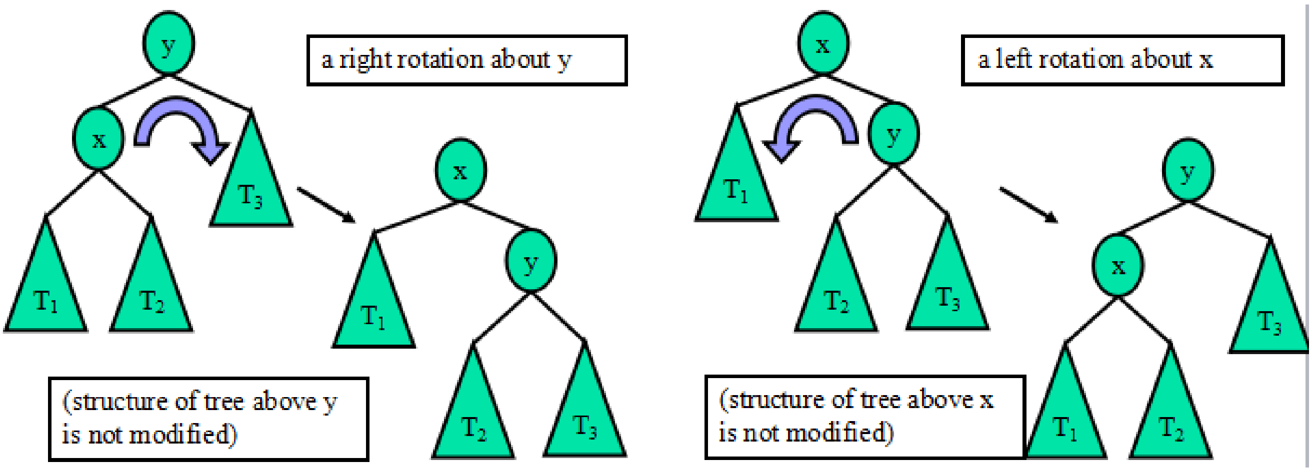
- 一次重构$O(1)$
- 查找$O(logn)$
- 插入(查找+重构) $O(logn)$
- 删除(查找+重构) $O(logn)$

6.3 Splay Trees

- 伸展树是一种特殊的BST,适合多次连续操作
- 类似AVL树,可以通过旋转来改变结点分布,进而减小深度;
- 差别在于, 伸展树不一定是平衡的,左右子树可以相差任意深度;
- Goal：指定某个结点x，通过不断旋转将x传送到到root位置;

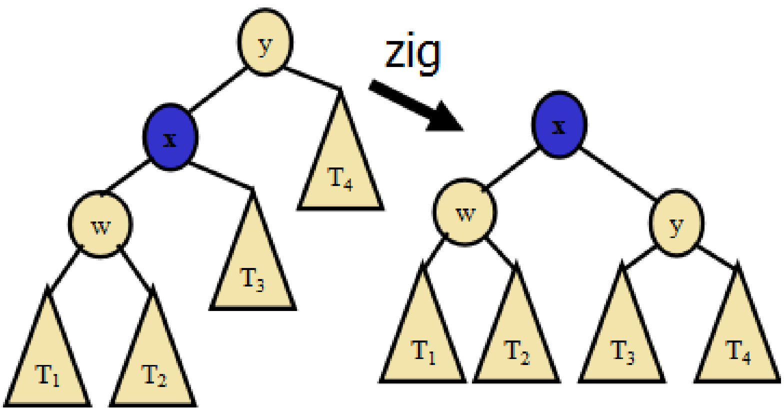
6.3.1 splay操作：通过不断旋转将指定结点传送到root位置

- 在每次操作后(包括搜索), 伸展树都会发生旋转
- 右旋转与左旋转

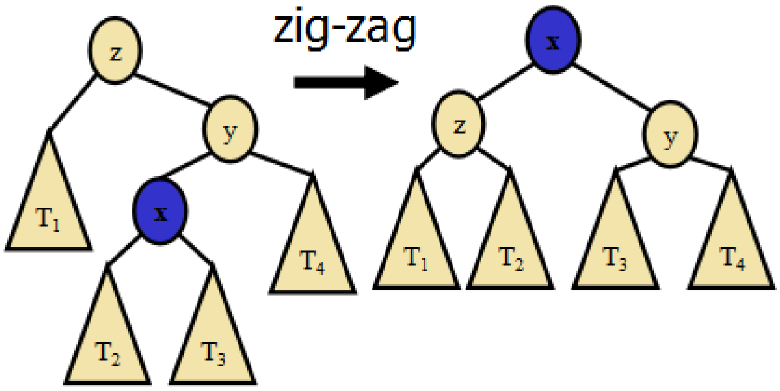


6.3.2 zig-zag过程

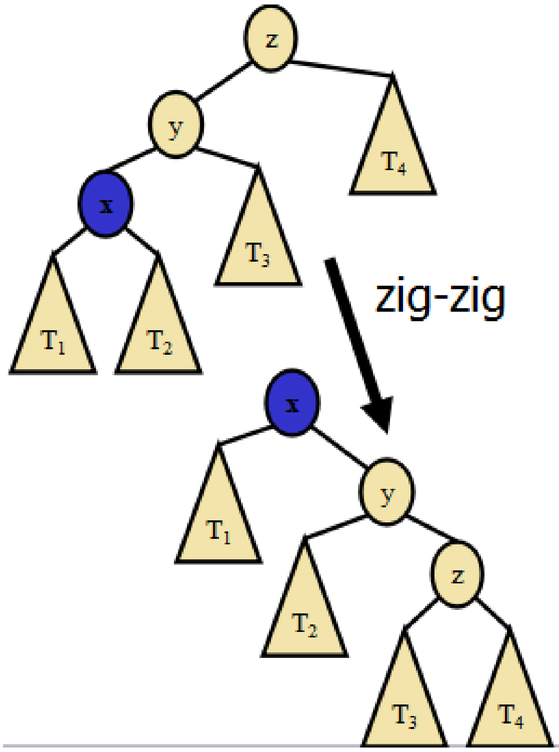
- zig：目标结点是根结点子结点时, 进行一次单旋转，使目标结点变为根结点



- zig-zag:目标/父结点/祖父结点呈zig-zag构型，进行异侧双旋转, 使目标结点到达原祖父结点位置



- zig-zig:同样将目标结点转换为祖父结点，同侧双旋转

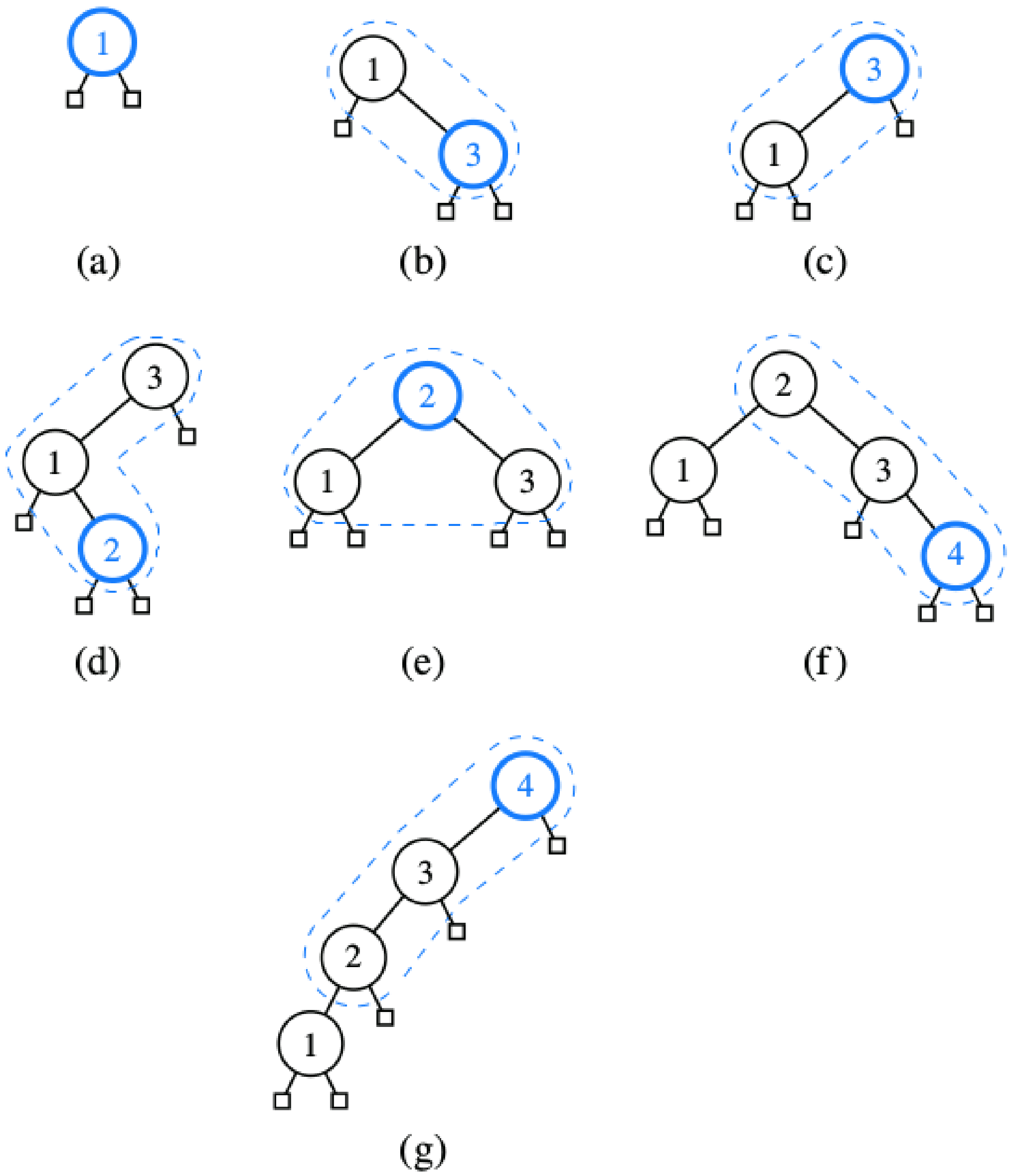



```
private void splay(Position<Entry<K,V>> p){
    while(!isRoot(p)){
        //首先得到p的父亲和祖父进行比较
        Position<Entry<K,V>> parent=parent(p);
        Position<Entry<K,V>> grandParent=parent(parent);

        //case 1: parent是根结点--> zig
        if(grandParent==null){
            rotate(p);
        }
        //case 2: 左左结构--> zig-zig
        else if(parent==left(grandParent) && p==left(parent)){
            rotate(parent);
            rotate(p);
        }
```

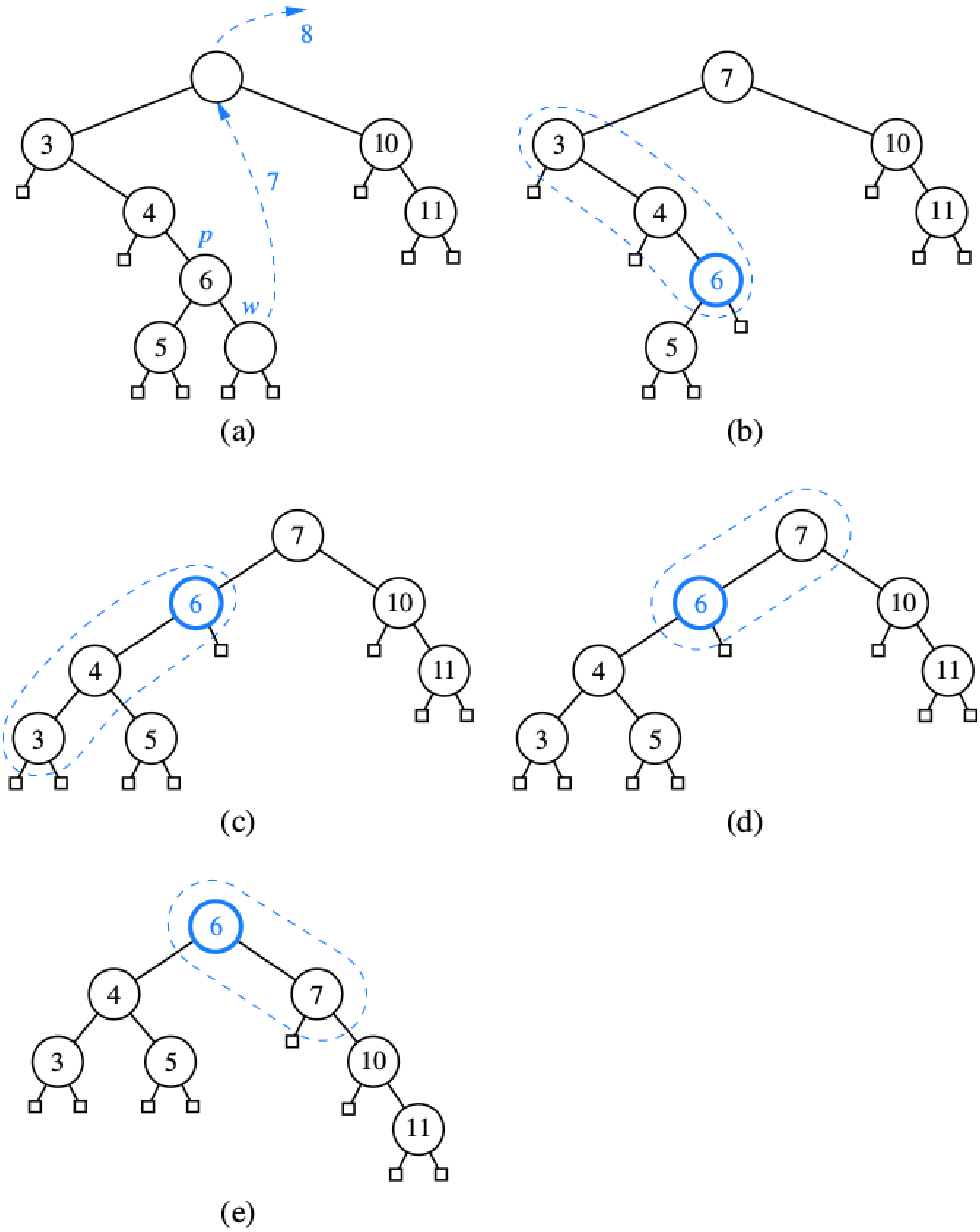
```
            //case 3: 右右结构--> zig-zig
        else if(parent==right(grandParent) && p==right(parent)){
            rotate(parent);
            rotate(p);
        }
        //case 4: zig-zag
        else{
            rotate(p);
            rotate(p);
        }
    }
```

6.3.3 选择splay node

- find(k):
  - 找到该结点则从该结点开始splay;
  - 没找到该结点,则从搜索终止位置(叶结点)的父结点开始splay;
- insert(k,v):
  - 使用被插入的新结点进行splay;



- remove(k):
  - 使用被删除结点的predecessor w替换此结点
  - 从w原来的父结点p开始进行splay过程




```java
protected void rebalanceAccess(Position<Entry<K,V>> p){
    if(isExternal(p)){
        p=parent(p);
    }
    if(p!=null){
        splay(p);
    }
}
```

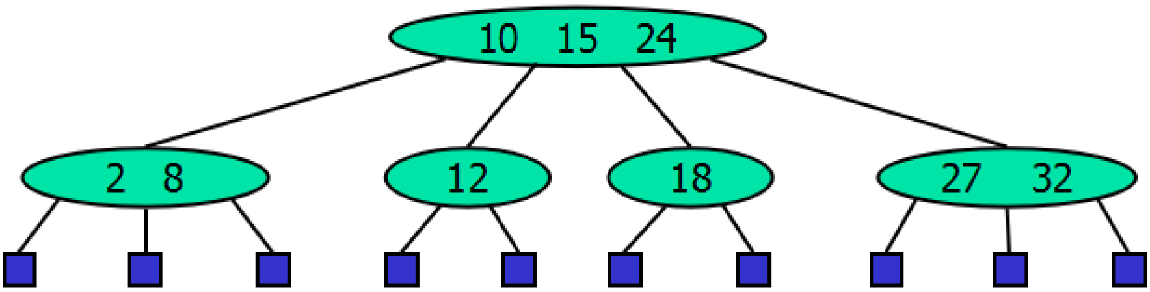
```java
protected void rebalanceInsert(Position<Entry<K,V>> p){
    splay(p);
}

//注意这里splay 结点为p(w的父结点)
protected void rebalanceDelete(Position<Entry<K,V>> p){
    if(!isRoot(p)){
        splay(parent(p));
    }
}
```

6.3.4 Splay Tree的性能

- P495有性能分析的证明
- 每次操作的运行时间正比于splaying time;
- 在深度d进行splay$O(d)$ 对某点进行splay的时间等同于找到该点的时间
- 单次搜索时和BST类似可能需要$O(n)$
- splay过程平均耗费$O(logn)$
- 因此伸展树可以保证m次连续搜索仅需要$mlogn$而非mn次操作, 适合连续多次操作

6.4 (2,4) Tree

- 在*Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne*中为(2,3) Tree, 不允许 4- 结点存在
- (2,4) tree 规则: 每个结点最多包含3个结点(4个子结点),所有的external深度相同



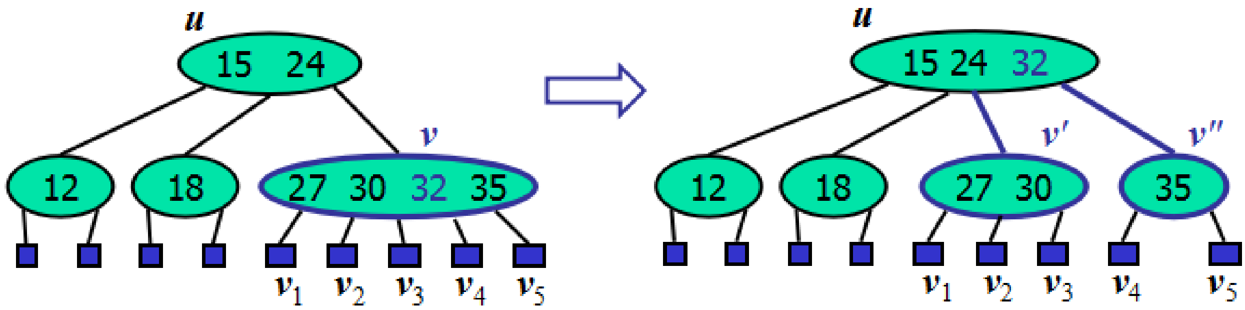
6.4.1 Height of a (2,4) Tree $O(logn)$

- 对于此的证明可以从以下不等式入手
  - 最高情况: 所有内部结点都是单结点
  - 最矮情况: 所有内部结点都是三结点

$$\frac{1}{2}log(n+1) \leq h \leq log(n+1)$$

6.4.2 Insertion

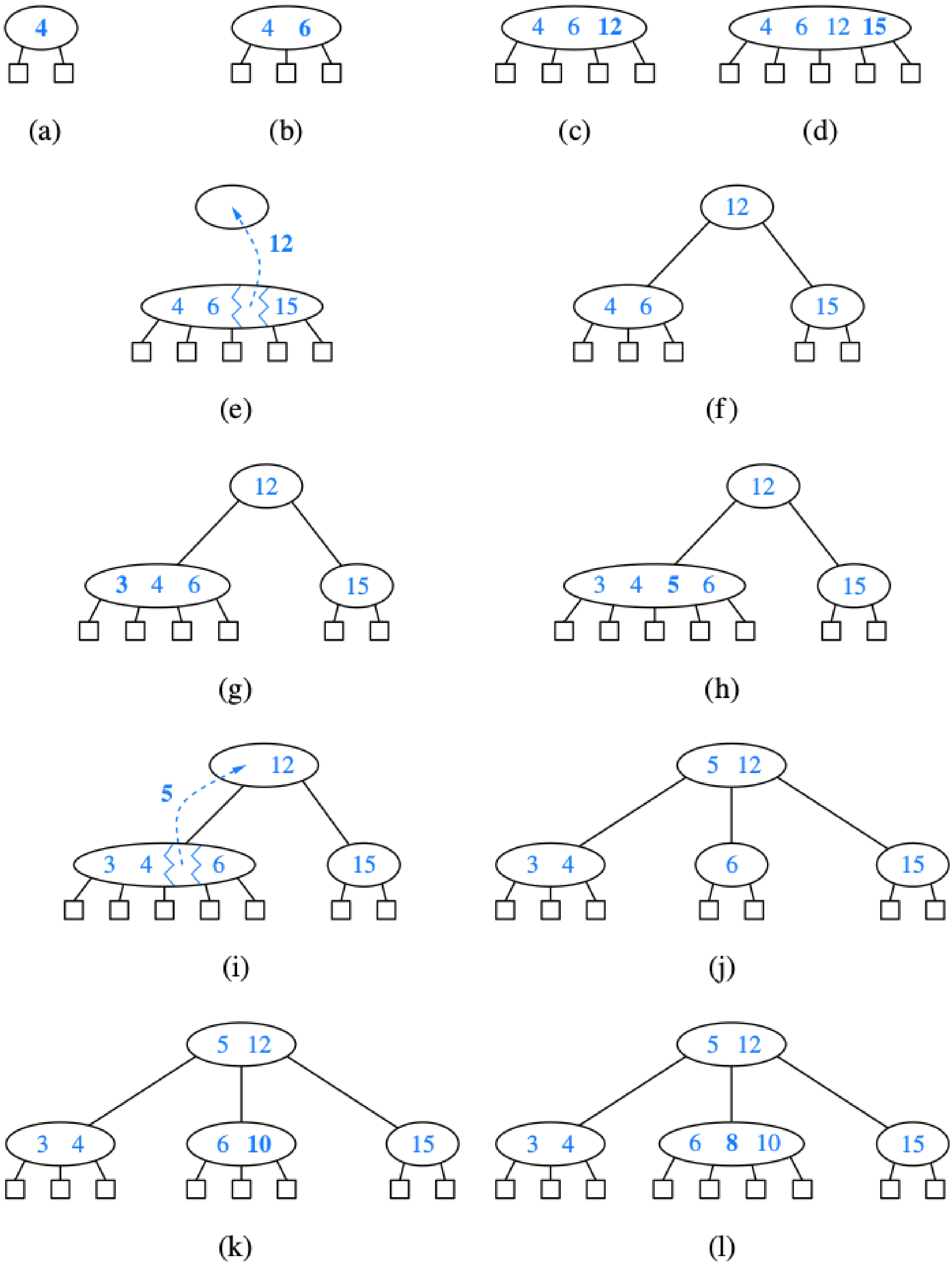
- 插入后可能破坏规则(overflow), 需要进行split



- Insertion + split $O(logn)$

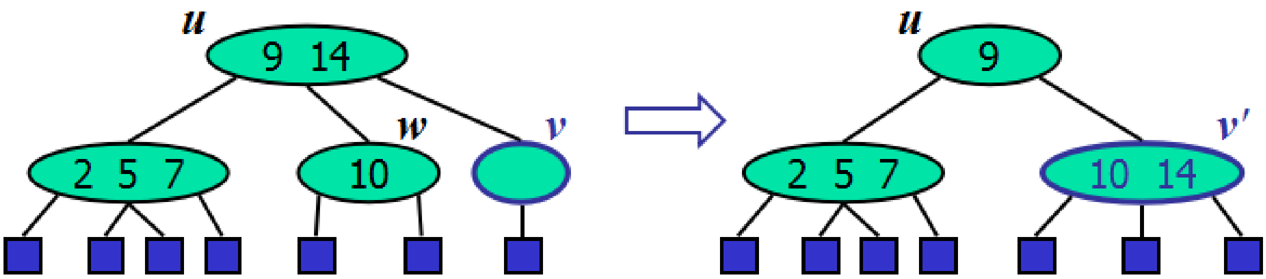
```
Algorithm insert(k, o){
    search for key k to locate the insertion node v;
    add the new entry (k, o) at node v;
    while ( overflow(v) ){
        if  (isRoot(v))
            create a new empty root above v;
        v = split(v);
    }
}
```

- 更复杂的栗子:

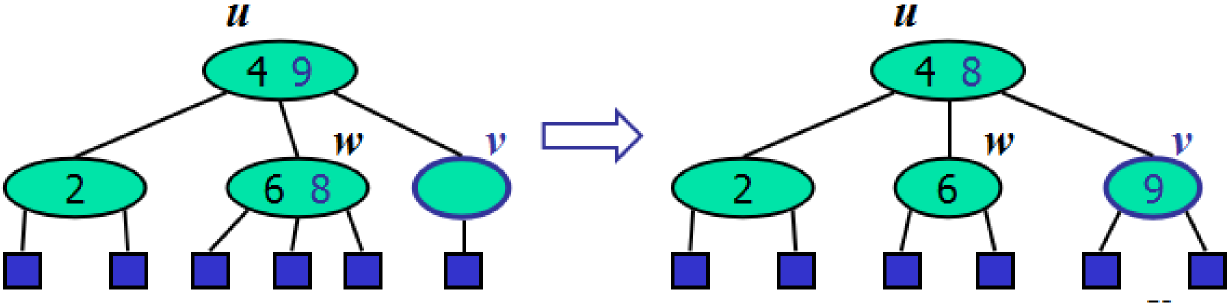


6.4.3 Deletion

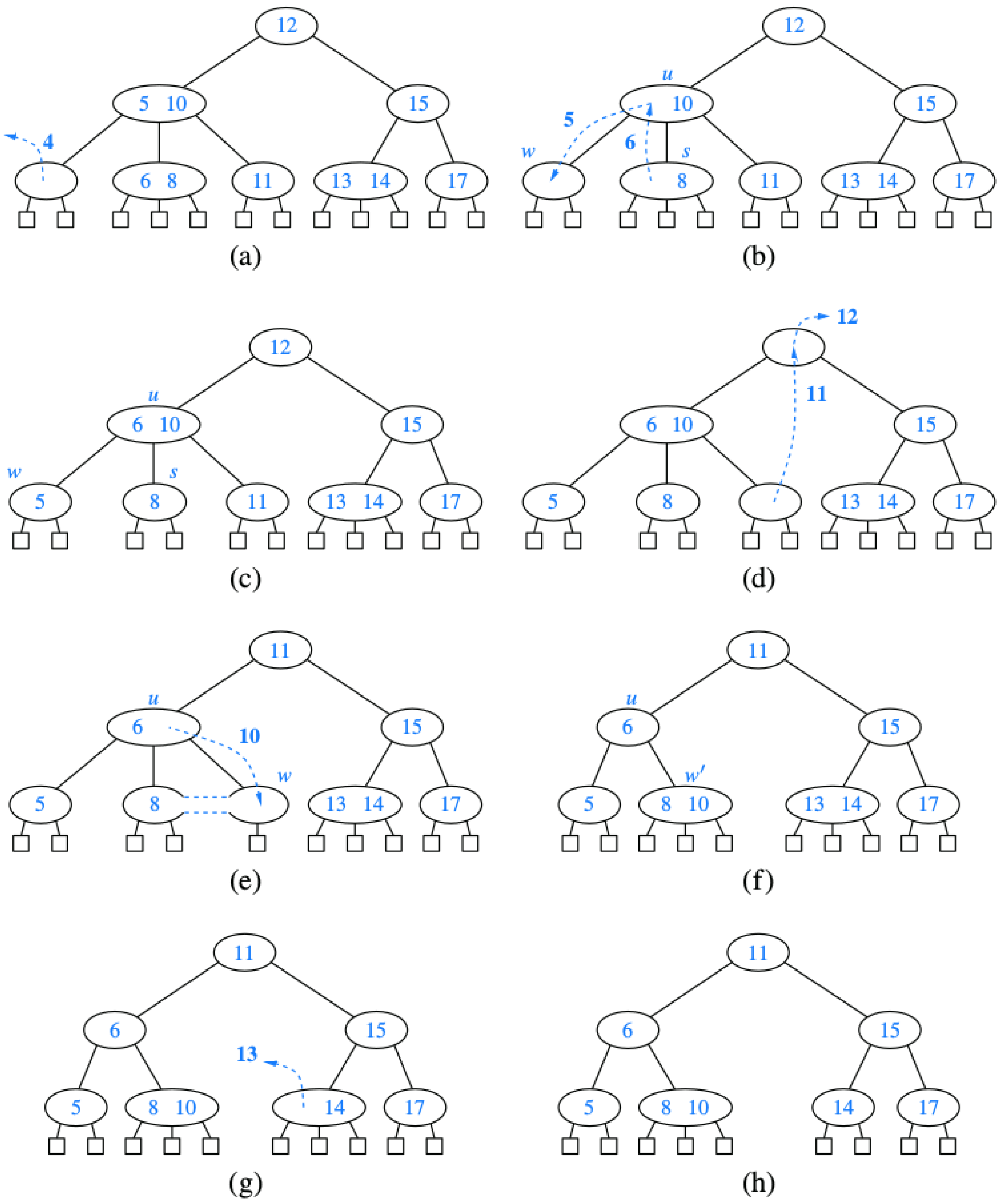
- 删除操作: 用被删除点的successor(predecessor)替换其位置
- 删除后可能破坏规则(underflow)，需要进行fusion/transfer
- Fusion:



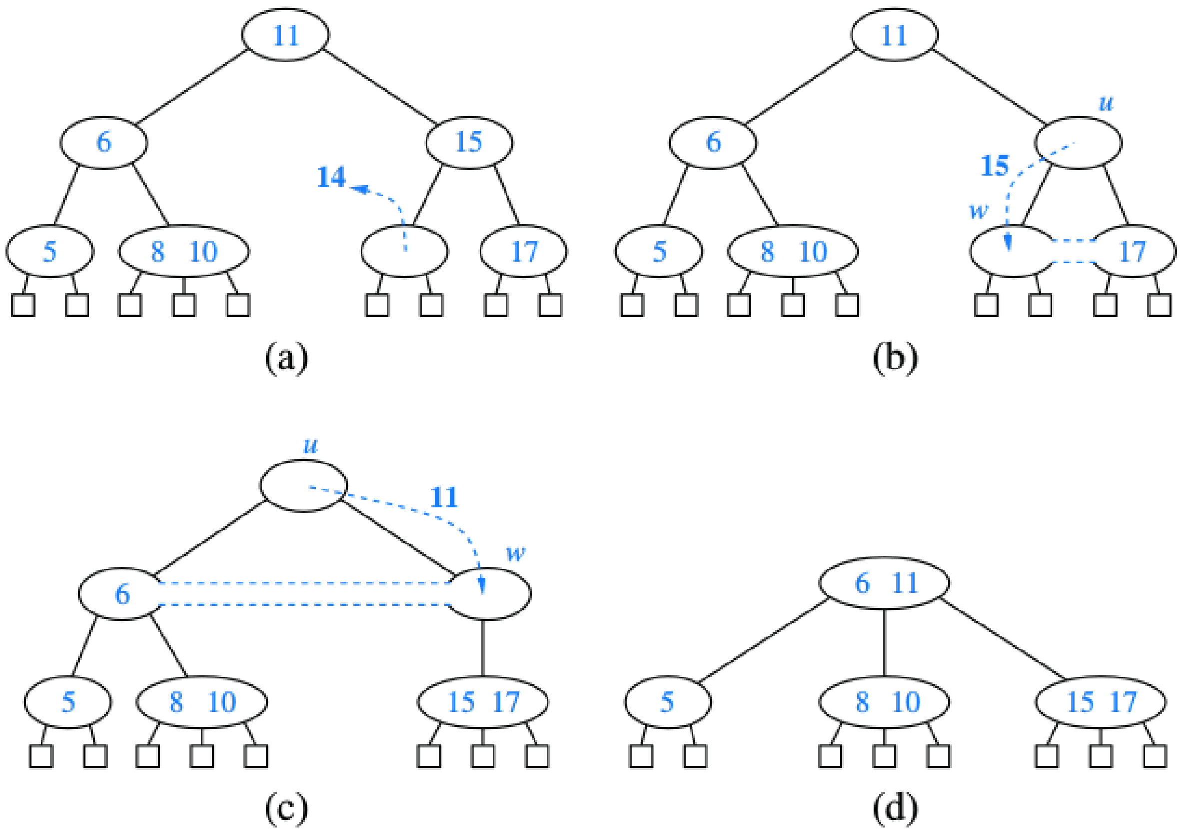
- Transfer:



- 更复杂的栗子:



- 最后一个栗子: 多次变换最后高度降低

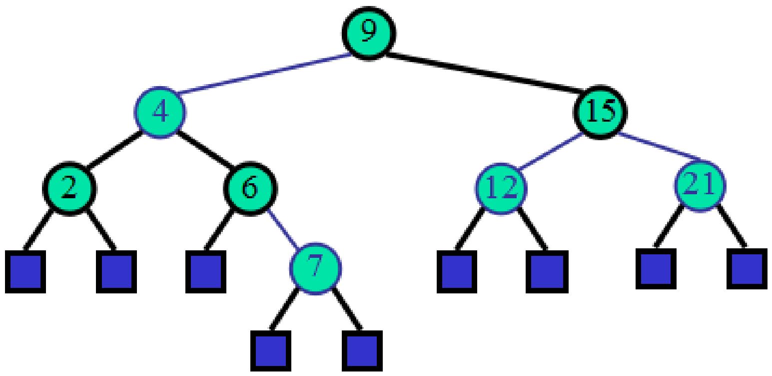


6.4.4 (2,4) Tree 的性能

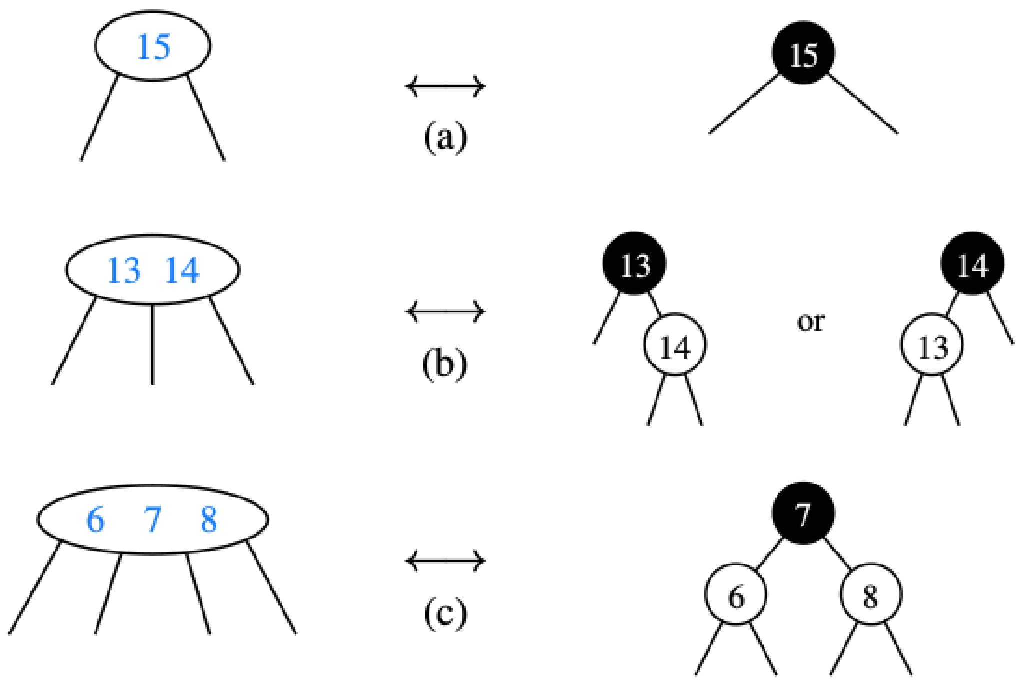
- search/insert/remove $O(logn)$
- split/fusion/transfer $O(1)$

6.5 Red-Black Trees

- COMP9024不会考察红黑树
- *Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne*和其他教材略有不同, 红黑链接形式
- 红黑树是一种特殊的二叉树，满足以下规则:
    - root与external为黑色;
    - 红结点的子结点都是黑的;
    - 完美黑色平衡:每个叶结点的black depth(到根节点需要经过的黑色结点数量)相同



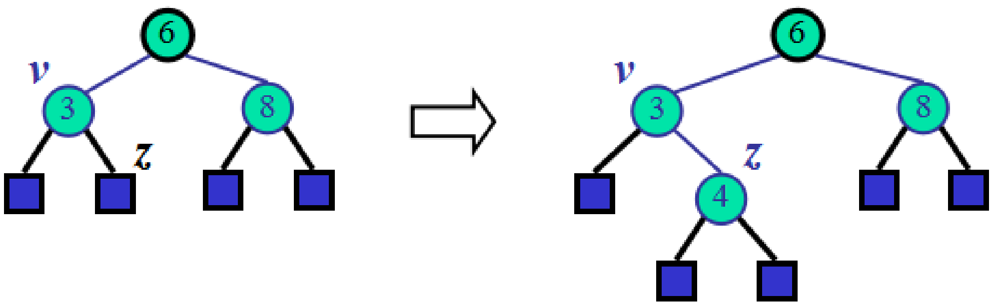
- 红黑树与(2,4)树的比较



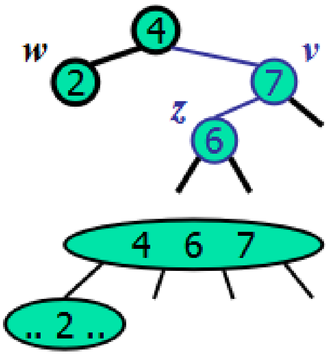
6.5.1 Height of a RB Tree: $O(logn)$

6.5.2 Insertion:

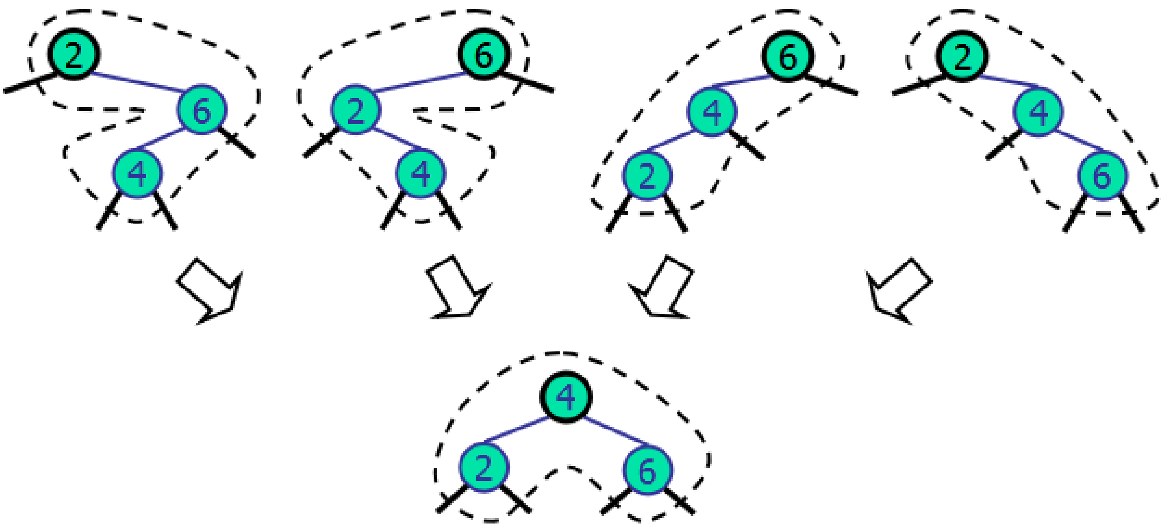
- 插入操作后红黑树规则可能被破坏, 需要Restructuring或者Recoloring
  - 被插入结点的颜色统一初始化为红色;
  - 由于根结点和叶结点始终保持是黑的;
  - 可能出现父子都是红结点的情况，需要转换;
- Double red: 被插入结点z和父结点v都是红的,注意原祖父结点x是黑的



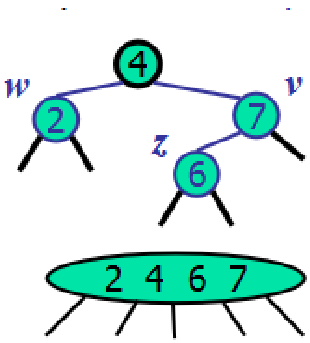
- Case 1:叔结点w(v同级)是黑的，进行restructuring
  相当于更改4-结点的布局;



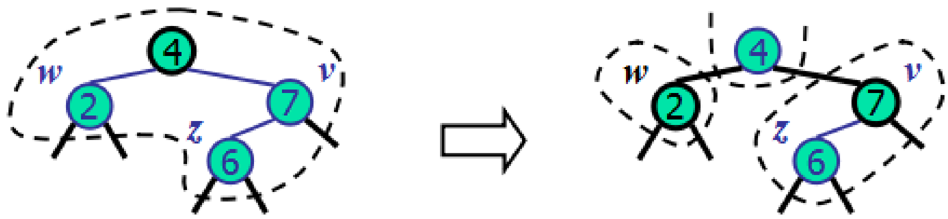
  - v>z>x(v为x右,z为v左):z(predecessor)变为x/v的父结点;
  - x>z>v(v为x左,z为v右):z(successor),同上;
  - x>v>z(两个都是左):v变为x/z的父结点;(单左旋)
  - z>v>x(两个都是右):同上;(单右旋)
  - 移动完成后进行变色: 新的父结点变为黑色,子结点红色
  - 总之就是找这三个点的中间点变为另两个点的父结点



- Case 2: w是红的,相当于x的子树一个单红一个双红 ,进行recoloring (该操作类似split)

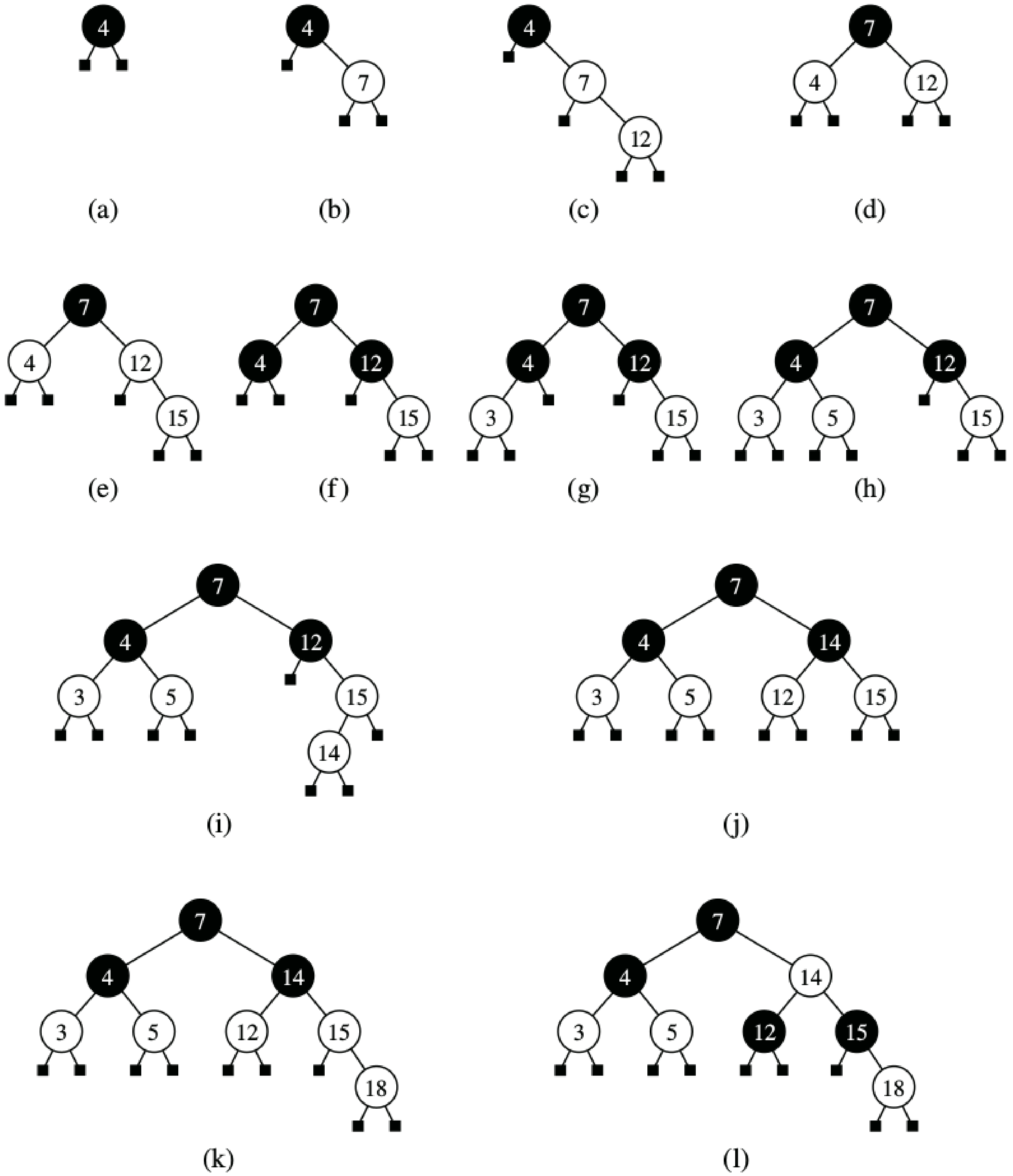


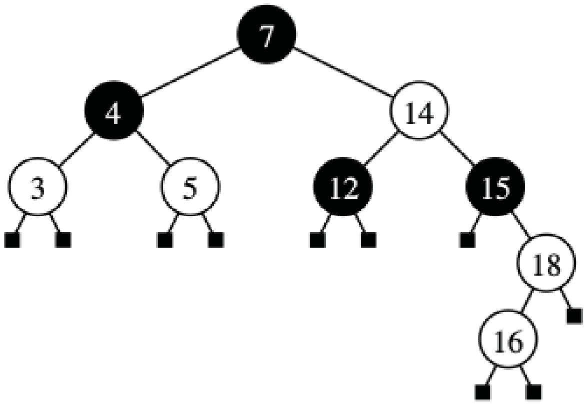
- x变为红的,wv变为黑的;
- 如果x是根结点,x不变色;
- 最后xz红wv黑



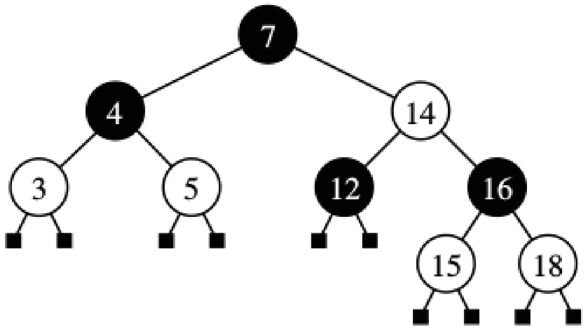

```
Algorithm insert(k, o){
    search(k)->确定插入点z;
    将(k,o)插入z处,并将该点设定为红色(只要是插入就是红);

    while doubleRed(z){ //z和其父结点都是红的
        if ( isBlack(sibling(parent(z)))) { //先确定叔结点w的颜色
            z = restructure(z);//w黑色->四种重构
            return;
        }
        else // w红色->recoloring->wv黑,xz红
            z = recolor(z);
    }
}
```

- 栗子: 构建一棵红黑树

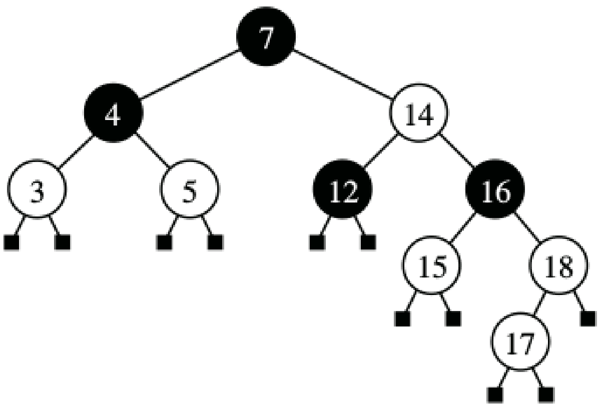

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)



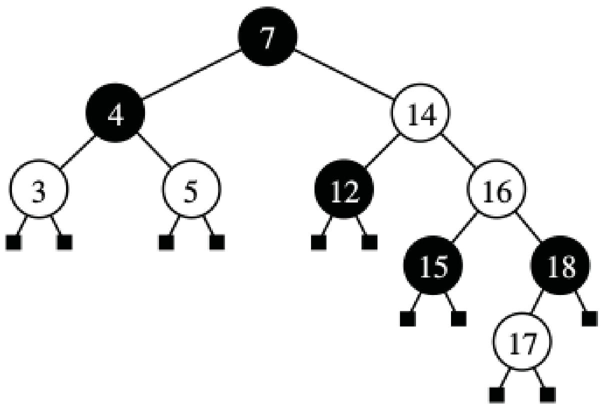
(m)



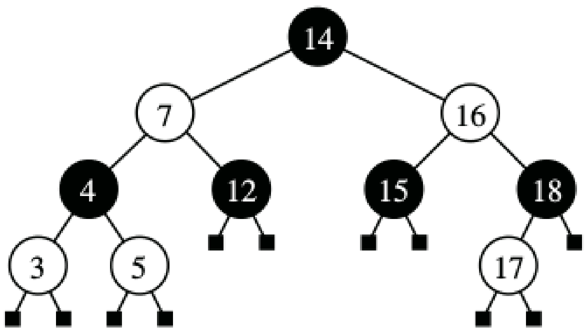
(n)



(o)



(p)

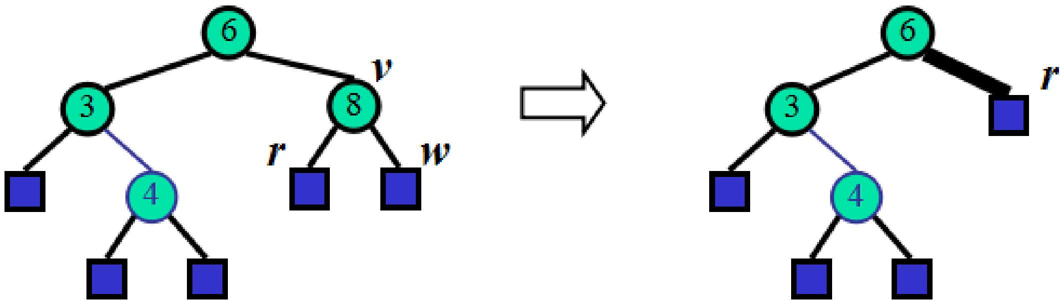


(q)

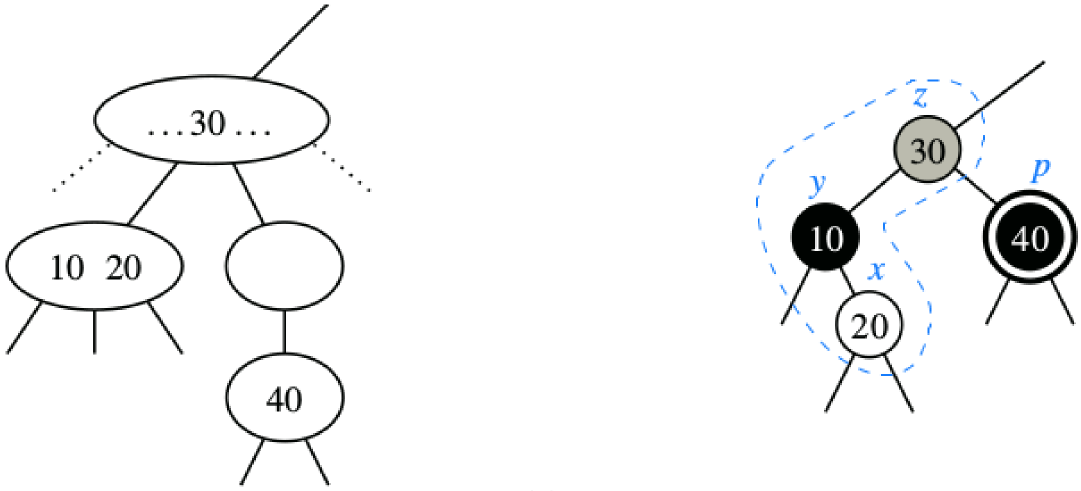
6.5.3 Deletion

- 先命名几个目标结点:
    - v : 指定被删除的internal node;
    - w : 指定被删除的external node,v的子结点;
    - r : w的兄弟结点,v的另一个子结点;
- 如果v/r至少一个是红的，r变成黑色替换到v的位置

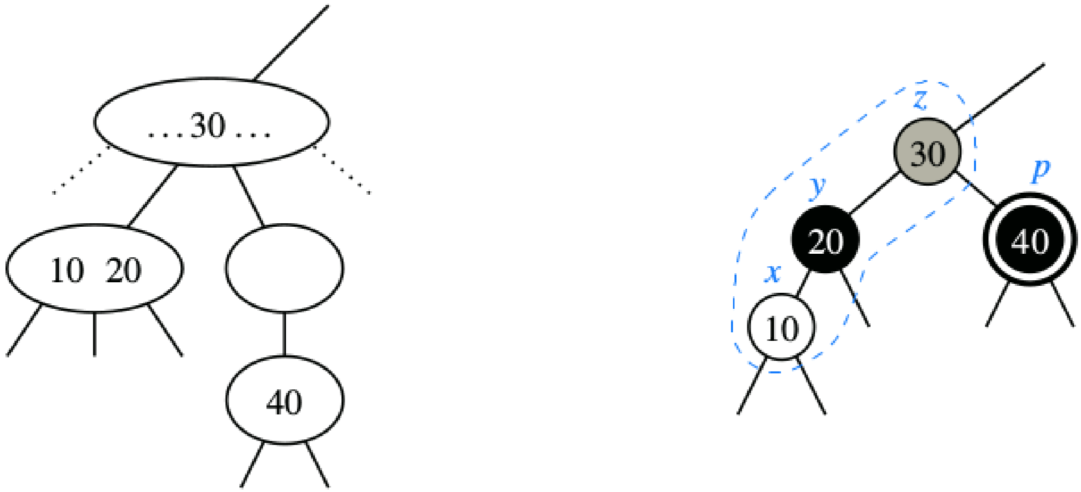
- 如果v/r都是黑的, 发生double black(相当于给涂了两次黑色) , 黑色平衡被破坏需要进行reorganization



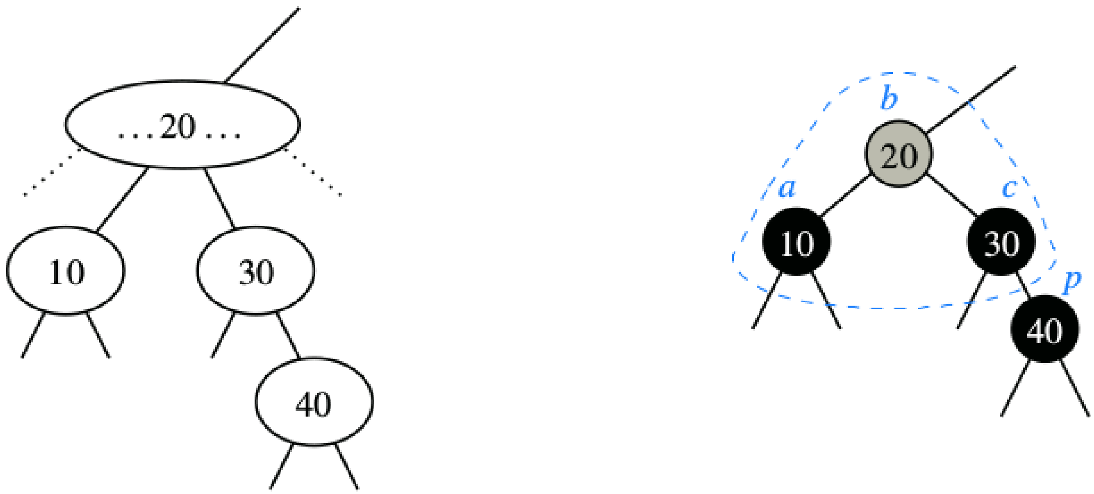
- reorganization 的几种情况
  - Case 1 : r黑 , r有一个红子结点
    - 对r以及其子结点进行restructuring(等价于transfer);



(a)

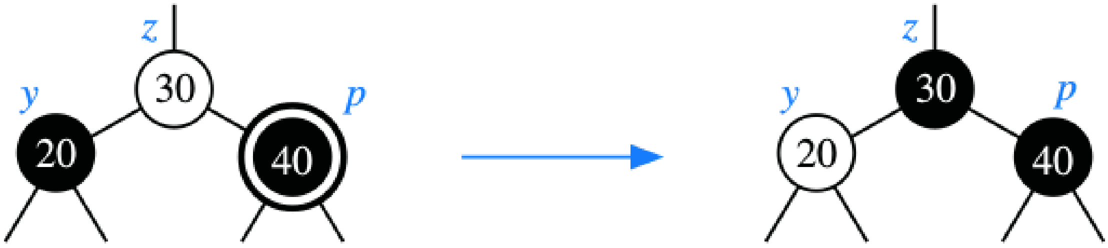


(b)

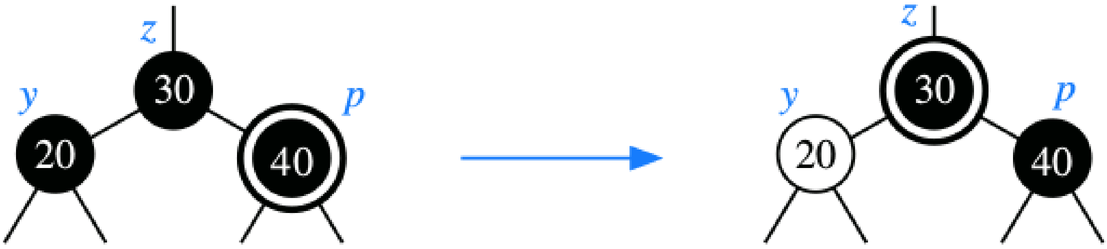


(c)

- Case 2：r黑，且其子结点都是黑的
  - recolor(r)，r变为红色(等价于fusion);
  - 如果变成了double red，循环重构

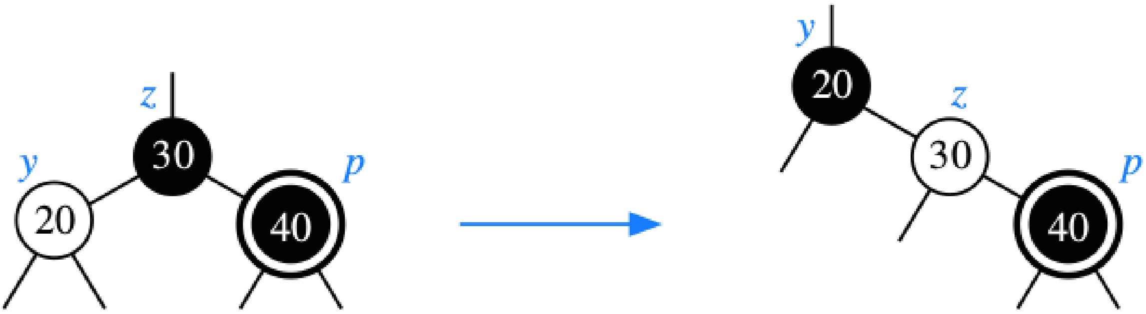


(a)

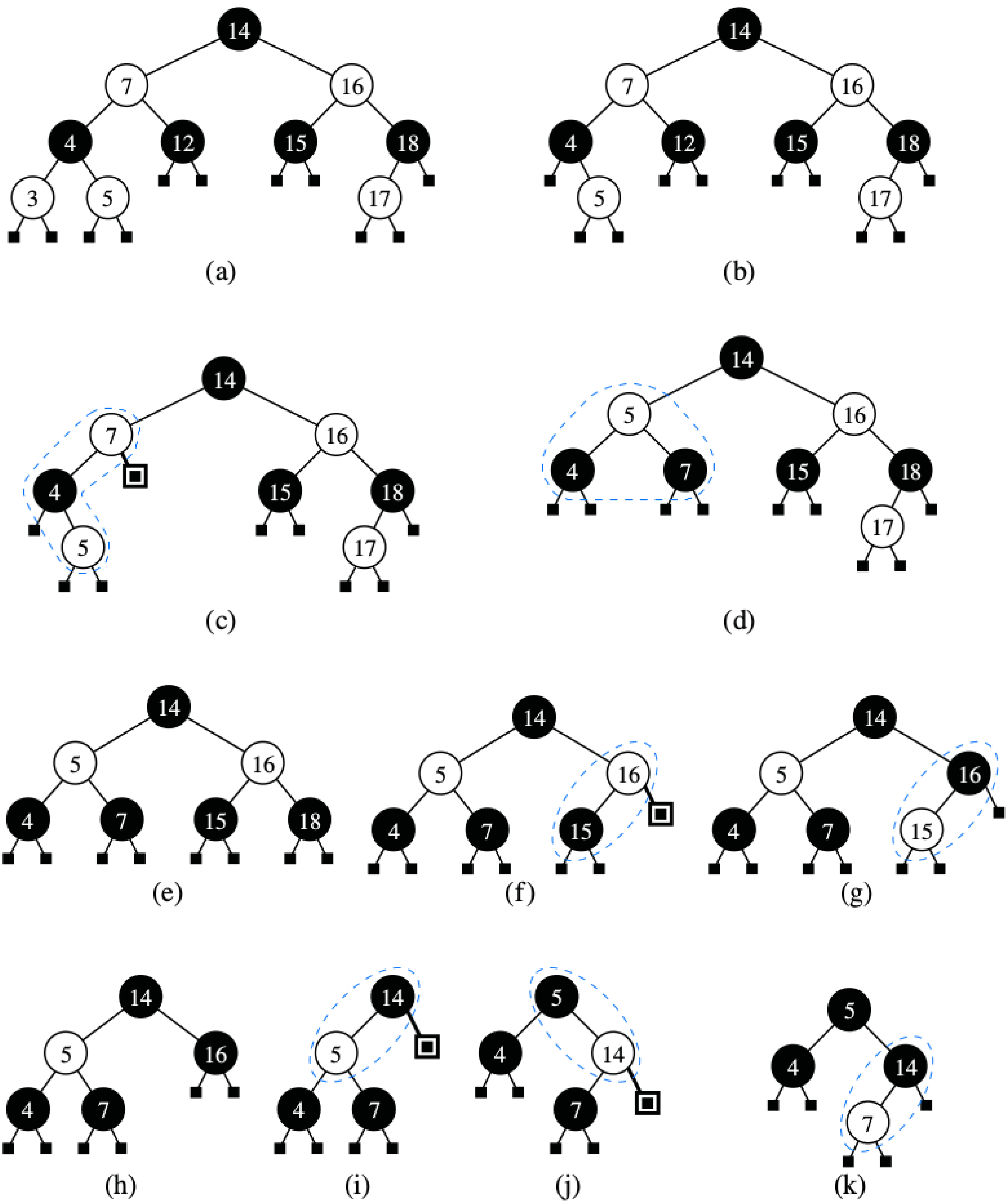


(b)

- Case 3 : r红
  - r的父结点(即原先v的父结点)、兄弟结点w以及r的一个孩子进行restructuring，r变为父结点



- 栗子: 删除一棵红黑树



Week 7 Priority Queues

- Priority Queues
- Heaps
- Adaptable Priority Queues

7.1 Priority Queues

7.1.1 Priority Queue ADT

- priority queue stores a collection of entries (key-value pairs)
- main operations:
  - insert(k,x);
  - removeMin();
  - min();
  - size()/isEmpty();
- applications:
  - stock market

7.1.2 Entry ADT

- entry = key-value pair
  - entry允许相同key存在;
  - map不允许相同key (Week 9);
  - dict允许相同key (Week 9);

```
public interface Entry{
    public Object key();
    public Object value();
}
```

7.1.3 Comparator ADT

- compare(a,b);
- 举个栗子:

```
public class NumComparator implements Comparator<Integer>{
    public int compare(int a,int b){
        if(a>b) return 1;
        else if(a==b) return 0;
        else return -1;
    }
}
```

- 直接使用自带的Comparator类:

```
import java.util.Comparator;
public class XXX{
    public static void main(String[] args){
        Comparator<K> Comp=new DefaultComparator<K>();
        int comp=Comp.compare(keyA, keyB);
```

```
        }
    }
```

7.1.4 Priority Queue Sorting

- 可用于对comparable elements 排序

```
Algorithm PQ-Sort(S,C)
INPUT: sequence S,comparator C for elements in S;
OUTPUT:使用C进行升序排列过的S;
{
    P=priority queue with comparator C;

    //先一个个移除S中的元素,插入P
    while(S.isEmpty()>0){
        e=S.removeFirst();
        P.insert(e,0);
    }

    //一个个移除P中最小的元素,
    while(P.isEmpty()>0){
        e=P.removeMin().key();
        S.insertLast(e);
    }
}
```

7.1.5 List-based PQ

- unsorted list : 4-5-2-3-1
    - insert : $O(1)$ (直接头尾插入)
    - removeMin()/min() : $O(n)$ (需要先找到最小的)
- sorted list : 1-2-3-4-5
    - insert : $O(n)$ (按顺序插入)
    - removeMin()/min() : $O(1)$ (有序队列)
    - 排序后的队列不适合插入，但很适合查找

7.1.6 Selection-Sort

- 基于未排序list的优先队列 $O(n^2)$
- 先随便插入P，每次移除P中最小的元素插入S
    - 插入n个元素 $O(n)$
    - 移除最小元素 $1+2+...+n=O(n^2)$
- 选择排序 : 每次从剩余元素中选取最小的交换到相应位置

```
for (i = 0; i < len - 1; i++) {
    //先把最小元素初始化为当前位置元素,位置定为min
```

```
        min = i;
        //在后面所有元素中寻找更小的元素,更新min
        for (j = i + 1; j < len; j++){
            if (arr[min] > arr[j]){
                min = j;
            }
        }
        //当前位置i和最小位置min进行元素交换
        temp = arr[min];
        arr[min] = arr[i];
        arr[i] = temp;
    }
```

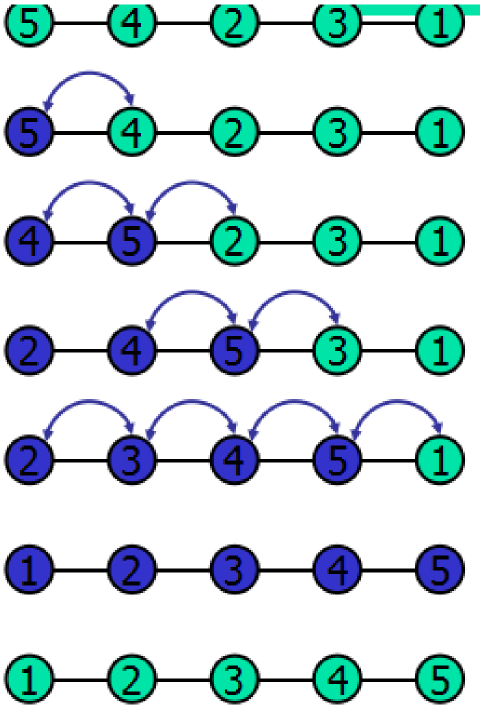
7.1.7 Insertion-Sort

- 基于有序数组的优先队列 $O(n^2)$
- 插入过程中排序(P时刻保持有序),移除时直接按顺序移除
    - 插入n个元素 $1+2+..+n=O(n^2)$
    - 移除最小元素 : $O(n)$
- 每次插入新元素，都把当前队列进行排序 ,保证插入下一个元素前队列是有序的

7.1.8 In-place Insertion-Sort

```
//每次都将current_element移动到合适的位置
for(int k=1;k<size;k++){
    int current_element=data[k];
    //先把新元素放到队尾,此时前面data[0],data[1],...,data[k-1]是有序的
    //同前面有序元素进行比较， 最后将新元素插入合适位置
    for(int j=k-1;j>=0;j--){
        if(data[j]>current_element){
            data[j+1]=data[j];
            data[j]=current_element;
        }
    }
}
```

- 不需要外部的新数据结构(前两个都是list+queue) $O(n^2)$
- 保证list从开始到当前位置这部分是有序的
- 使用交换(swap)来代替更改list



7.1.9 Bubble sort

- 冒泡排序类似选择排序, 前者每次把最大的元素放在最后,后者每次把最小的元素放在最前
- 区别在于冒泡排序每轮都对全体相邻元素两两比较交换, 而选择排序只进行比较最后再交换
  - 第一轮比较n-1次将最大元素放到队尾
  - 第二轮比较n-2次将第二大元素放到倒数第二位置
  - ...
  - 最后一轮比较一次确定前两个元素的位置
- 冒泡排序也是$O(n^2)$算法, 但很适合本身就有序的队列，冒泡一次就行$O(n)$

```
for(int i=0;i<myL.length-1;i++){
    //length-1-i后面的元素是已经被交换到合适位置的大元素
    for(int j=0;j<myL.length-1-i;j++){
        //从前面把比较大的元素向后推进
        if(myL[j+1]<myL[j]){
            int temp=myL[j+1];
            myL[j+1]=myL[j];
            myL[j]=temp;
        }
    }
}
```

7.2 Heaps

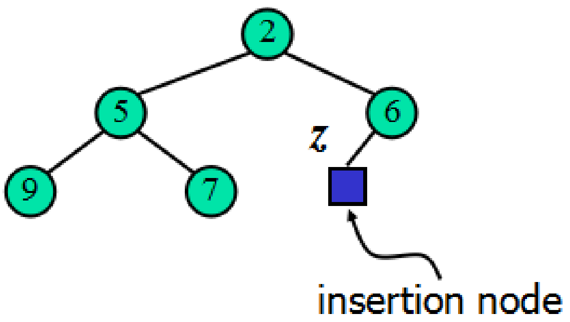
- Heap的形状类似二叉树，但规则不同
  - 每个子结点都保证不小于父结点，root最小 (最小堆);

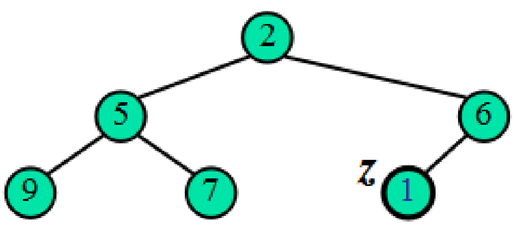
- Heap-order : 除了root的internal node v , key(v) >= key(parent(v));
- Complete Binary Tree:
  - 深度为h的这一层结点数为 2^h
  - 上一层没被填满绝对不在下一层添加子树
  - 尾结点是深度h(最底层)的最右结点;
- 包含n个结点的heap高度 $h = O(logn)$
- 使用heap来实现priority queue
  - 每个internal node中包含储存一个entry
  - 需要跟踪结尾结点的位置

7.2.1 Insertion

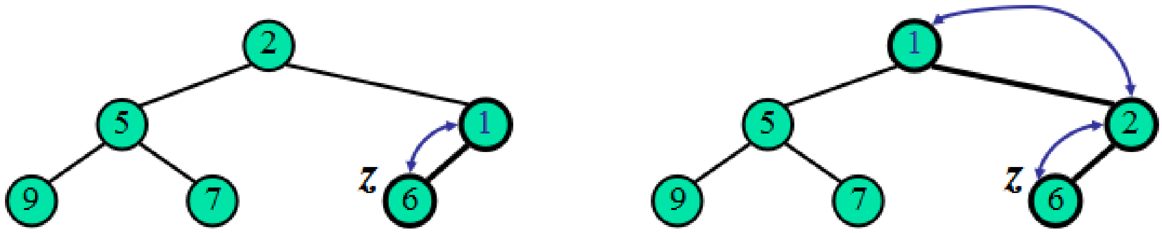
- 每次将新entry插入当前结尾结点的下一个位置, 并使用upheap()来更新du

```
Algorithm Insert{
    find insert position Node z; //当前尾结点的下一个位置
    Store k at z;
    upheap();
}
```





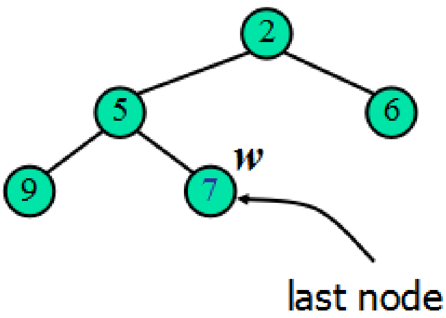
- upheap : 插入新entry后heap-order可能被破坏
  - 新插入结点与其父结点进行比较 , 直到满足heaporder或者新结点到达root;
  - upheap过程 $O(logn)$

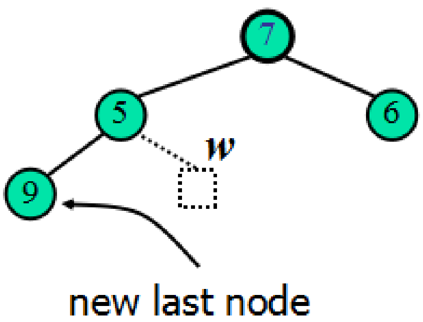


7.2.2 Deletion

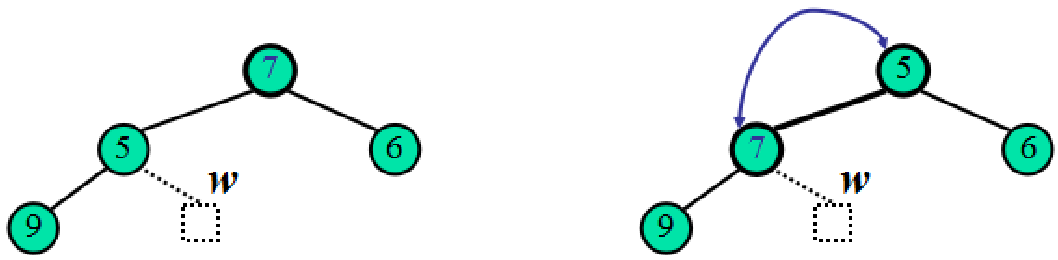
- 每次删除的都是根结点: removeMin()
- 返回root位置元素, 并将结尾元素替换到root ,之后使用downheap恢复堆排序
- downheap比较耗时间, 总体复杂度: $O(logn)$

```
Algorithm remove{
    min = root.getEntry();
    root.setEntry(lastNode);//将尾结点元素替换到root
    delete original position;
    downheap();
    return min;
}
```



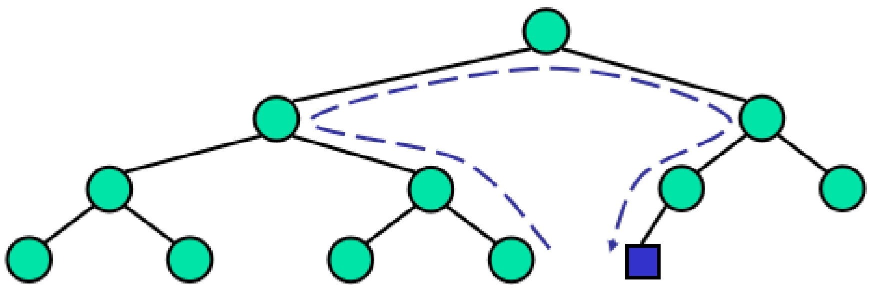


- Downheap : 结尾entry替换到root后 , heaporder会被破坏
  - 从root开始 , 不断与子结点比较交换直到到达底层或是满足heap-order;
  - 优先交换子结点比较小的那个;
  - downheap过程 $O(logn)$;



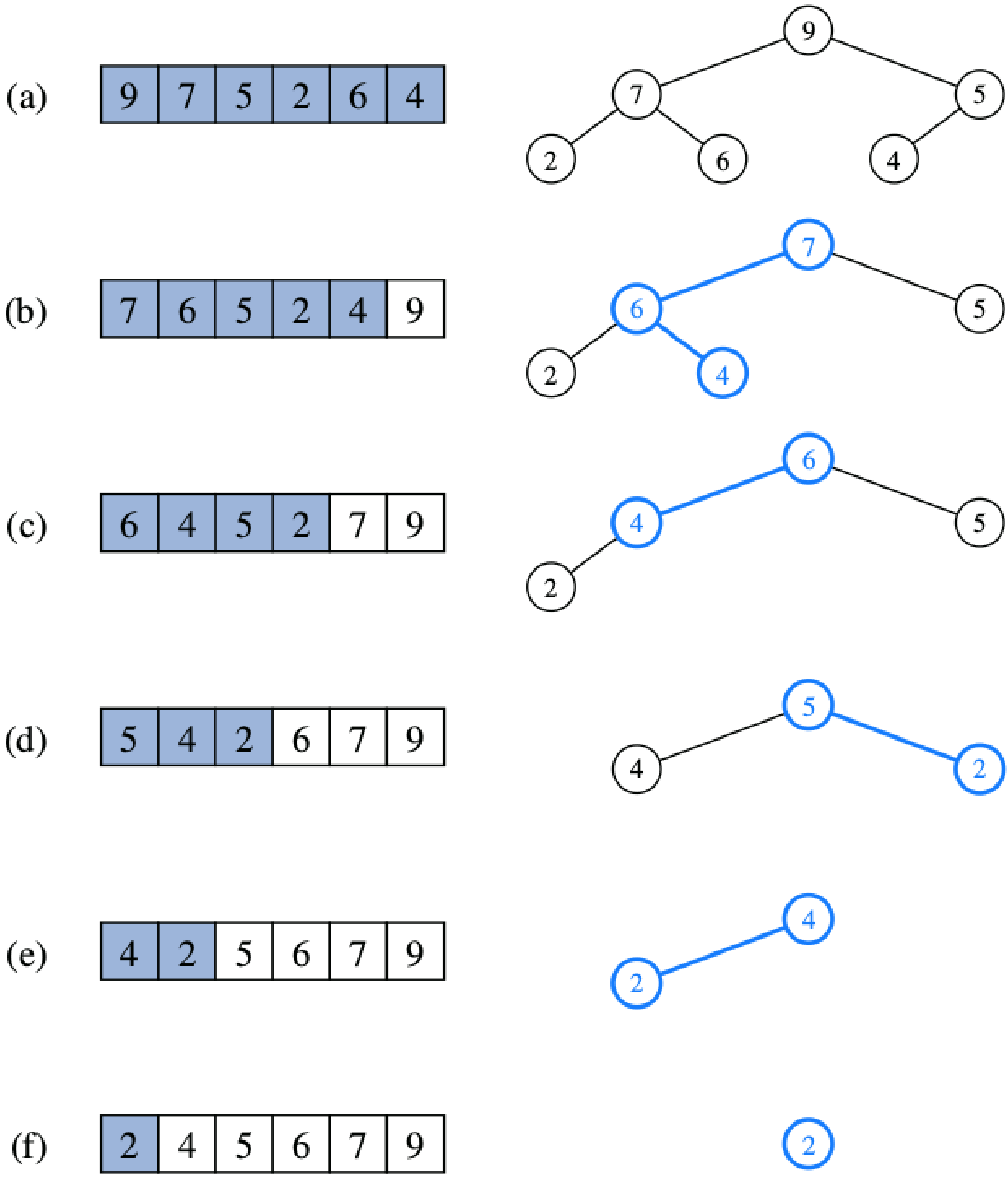
7.2.3 Updating the last node

- 每次插入或删除操作时都需要更新尾结点位置;
- 更新步骤:
    - 先从指定结点不断向上直到到达根结点或是根结点的左子结点;
    - 如果到达的是左子结点，切换到该结点的兄弟结点(根结点的右子结点);
    - 从根结点的右子结点开始，不断向左下移直到到达叶结点;
    - 该叶结点即为当前尾结点;
- 更新操作耗时 $O(logn)$



7.2.4 Heap-Sort

- 使用heap实现优先队列:
    - 空间复杂度$O(n)$;
    - insert()/removeMin() : $O(logn)$
    - size()/isEmpty()/min(): $O(1)$
    - locating:$O(1)$ for array-based heap , $O(logn)$for linked-tree heap;
- 基于heap的优先队列排序 :$O(nlogn)$
- 实现堆排序:
    - 给出一个堆和一个空队列;
    - 每次对堆进行removeMin()操作，提取堆顶元素;
    - 将提取出来的元素一次插入空队列，实现排序;
- 书里的栗子是最大堆, 每次提取堆顶最大元素放在未排序队列尾部



- 使用java实现heap：*Data Structures and Algorithms by Goodrich and Tamassia* P378

```
package net.datastructures;
import java.util.Comparator;

//基于堆的优先队列

public class HeapPriorityQueue<K,V> implements PriorityQueue<K,V> {
    protected CompleteBinaryTree<Entry<K,V>> heap;// underlying heap
    protected Comparator<K> comp;    // comparator

```

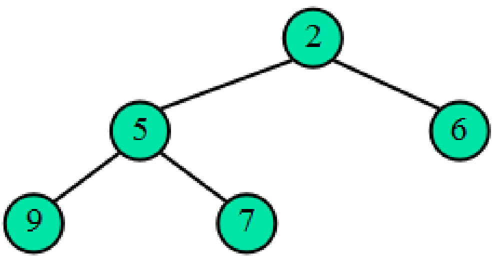
```
    /** 嵌套类: Entry */
    protected static class  MyEntry<K,V> implements Entry<K,V> {
        protected K key;
        protected V value;
        public MyEntry(K k, V v) { key = k; value = v; }
        public K getKey() { return key; }
        public V getValue() { return value; }
        public String toString() { return "(" + key  + "," + value + ")"; } //重写toString(), 不然打印出来的是哈希码
  }
  /** Creates an empty heap with the default comparator */
    public HeapPriorityQueue() {
        heap = new ArrayListCompleteBinaryTree<Entry<K,V>>();     // use an array list
        comp = new DefaultComparator<K>();     // use the default comparator
  }
    /** Creates an empty heap with the given comparator */
     public HeapPriorityQueue(Comparator<K> c) {
        heap = new ArrayListCompleteBinaryTree<Entry<K,V>>();
        comp = c;
  }


    public void setComparator(Comparator<K> c) throws IllegalStateException {
        if(!isEmpty())  // this is only allowed if the priority queue is empty
        throw new IllegalStateException("Priority queue is not empty");
        comp = c;
    }


    public int size() { return heap.size(); }
    public boolean isEmpty() { return heap.size() == 0; }
  public Entry<K,V> min() throws EmptyPriorityQueueException {
        if (isEmpty())
        throw new EmptyPriorityQueueException("Priority queue is empty");
        return heap.root().element();
  }

    public Entry<K,V> insert(K k, V x) throws InvalidKeyException {
        checkKey(k);
        Entry<K,V> entry = new MyEntry<K,V>(k,x);
        upHeap(heap.add(entry));
        return entry;
    }
    public Entry<K,V> removeMin() throws EmptyPriorityQueueException {
        if (isEmpty())
            throw new EmptyPriorityQueueException("Priority queue is empty");
        Entry<K,V> min = heap.root().element();
        if (size() == 1)
            heap.remove();
```

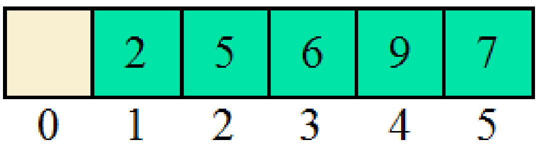
```
        else {
            heap.replace(heap.root(), heap.remove());
            downHeap(heap.root());
        }
        return min;
    }
    /** Determines whether a given key is valid */
    protected void checkKey(K key) throws InvalidKeyException {
        try {
            comp.compare(key,key);
        }
        catch(Exception e) {
            throw new InvalidKeyException("Invalid key");
        }
    }

    //begin#fragment auxiliary
    /** Performs up-heap bubbling */
    protected void upHeap(Position<Entry<K,V>> v) {
        Position<Entry<K,V>> u;
        while (!heap.isRoot(v)) {
            u = heap.parent(v);
            if (comp.compare(u.element().getKey(), v.element().getKey()) <= 0) break;
            swap(u, v);
        v = u;
        }
    }
    /** Performs down-heap bubbling */
    protected void downHeap(Position<Entry<K,V>> r) {
        while (heap.isInternal(r)) {
            Position<Entry<K,V>> s;     // the position of the smaller child
            if (!heap.hasRight(r))
                s = heap.left(r);
            else if (comp.compare(heap.left(r).element().getKey(),
                                heap.right(r).element().getKey()) <=0)
                s = heap.left(r);
            else
            s = heap.right(r);
        if (comp.compare(s.element().getKey(), r.element().getKey()) < 0) {
            swap(r, s);
            r = s;
        }
        else
            break;
        }
    }
  /** Swaps the entries of the two given positions */
    protected void swap(Position<Entry<K,V>> x, Position<Entry<K,V>> y) {
        Entry<K,V> temp = x.element();
```

```
            heap.replace(x, y.element());
            heap.replace(y, temp);
      }
      /** Text visualization for debugging purposes */
        public String toString() {
            return heap.toString();
        }
        //end#fragment auxiliary
    }
```

7.2.5 Array list-based heap implementation

- 一开始就是个普通数组, 之后按堆排序放置元素
- 从root到leaf进行构造
- 使用n+1长度的array list来表示带有n个key的heap
- 对于rank为i的node:
    - 左子结点排名2i;
    - 右子结点排名2i+1;
    - 书上是从零开始左边2i+1右边2i+2;
- 结点之间的链接并非是显式保存的
- rank 0 为null , 从rank 1(root) 开始实现堆
- insert():在rank n+1 的位置进行插入;
- removeMin():在rank 1 的位置进行删除;

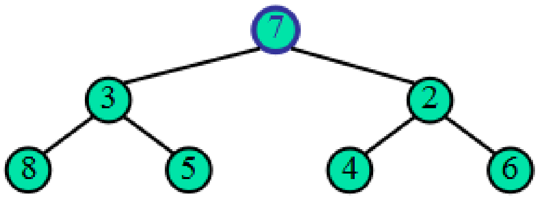


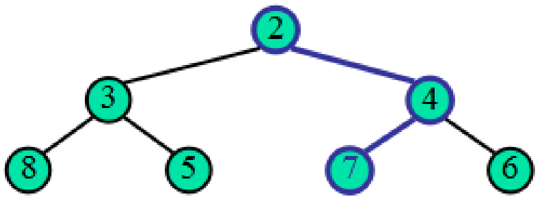


7.2.6 Merging two heaps

- Input 2 heaps and a key k;
- 创建一个新heap,根结点key为k,两棵子树为打算merge的heaps;
- 检验各结点是否满足heap-order;

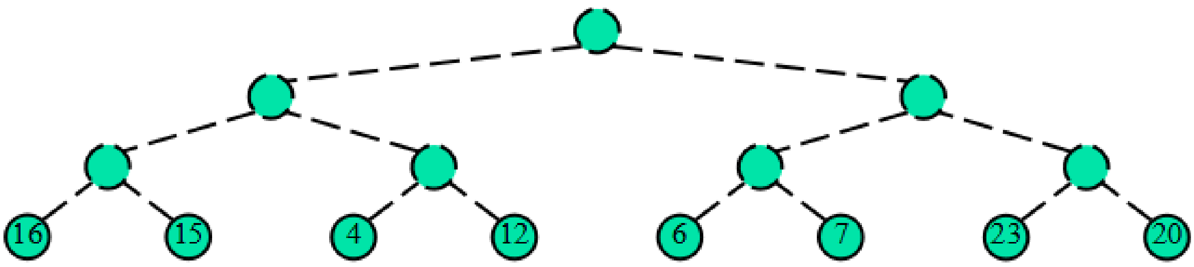
- 对破坏规则的使用downheap来重构;

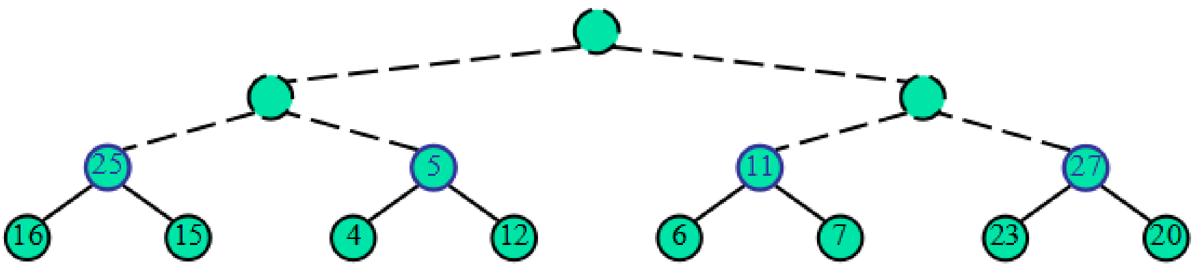


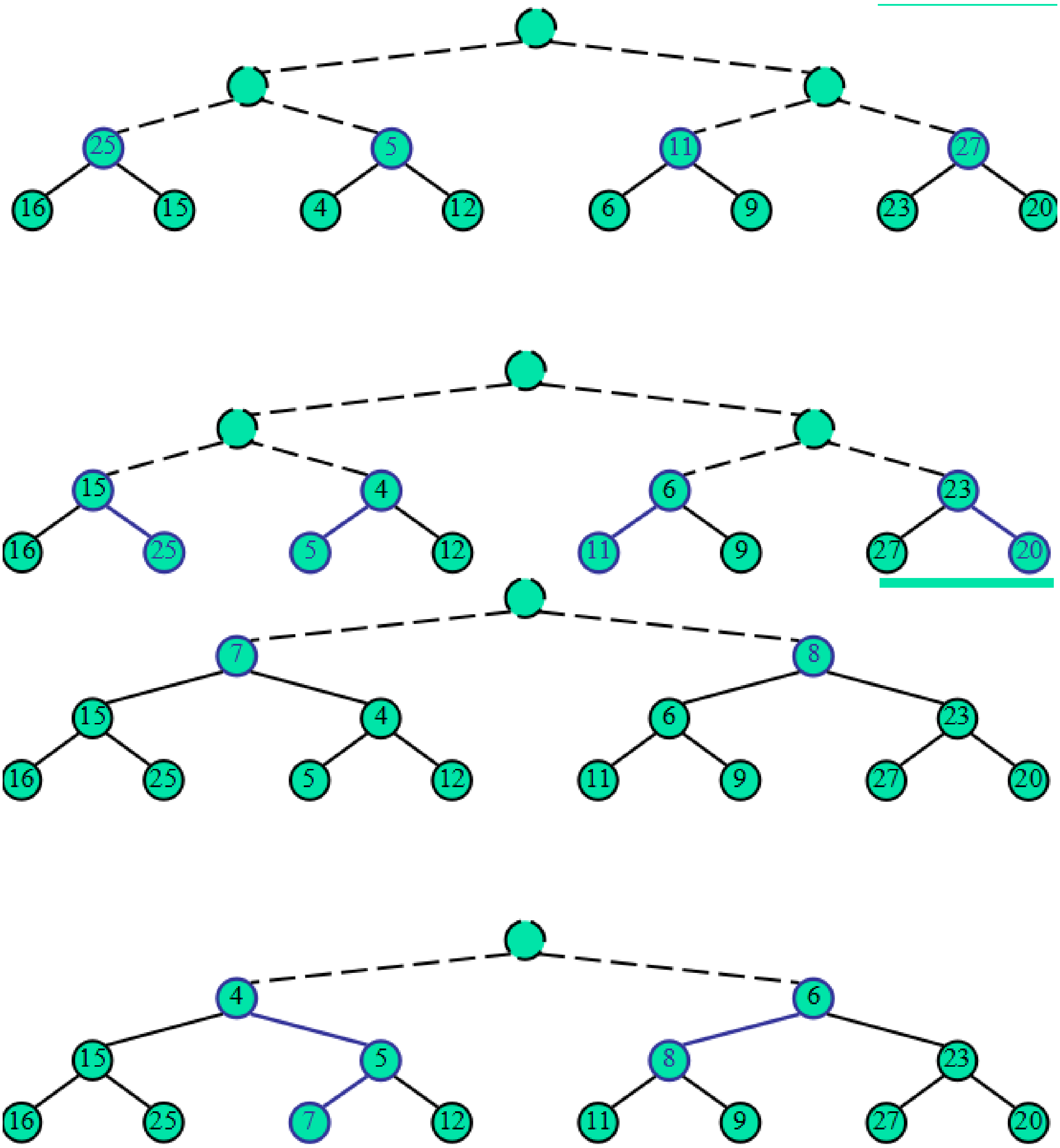


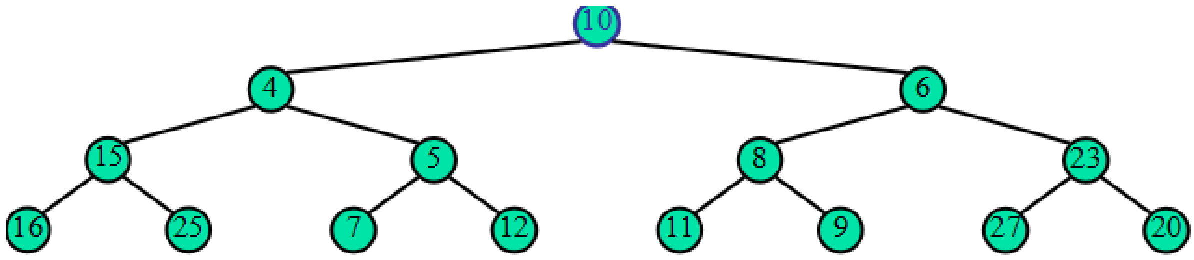
7.2.7 Bottom-up Heap Construction

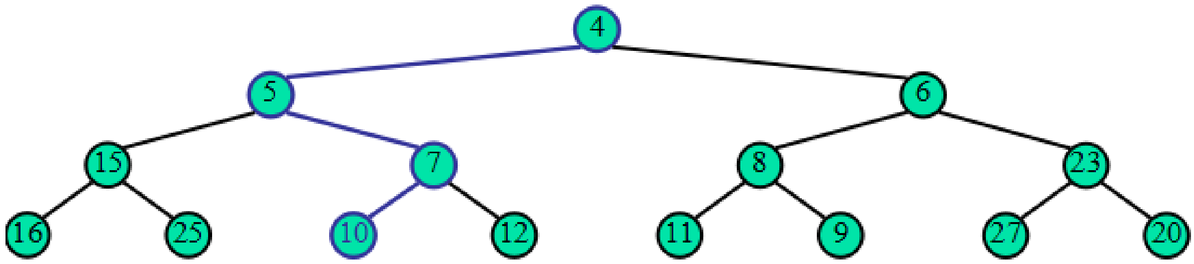
- $\log n$ phases,构造时间$O(n)$, 比从上到下省时间
- 其实就是不断归并子堆的过程
- 对于phase i : 将一对拥有$2^{(i+1)}$ 元素的堆被归并成一个$2^{(i+1)} - 1$ 元素的堆






25 5 11 27
16 15 4 12 6 9 23 20
15 4 6 23
16 25 5 12 11 9 27 20
7 8
15 4 6 23
16 25 5 12 11 9 27 20
4 6
15 5 8 23
16 25 7 12 11 9 27 20





7.3 Adaptable PQ

7.3.1 Methods of the APQ ADT

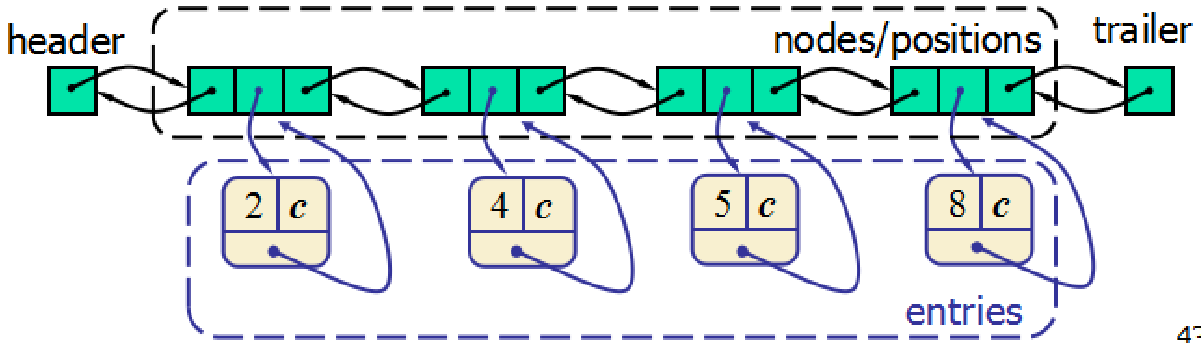
- remove(e) : 删除P中的某个键值对并返回entry e;
- replaceKey(e,k) : 使用k替换并返回原键值对e中的key;
- replaceValue(e,x) : 使用x替换并返回原键值对e中的value;

7.3.2 Locating Entries

- 实现上述功能需要先用一种高效的查找方法来对entry e进行定位;
- Locator-Aware Entries的效率高于遍历整个数据集;

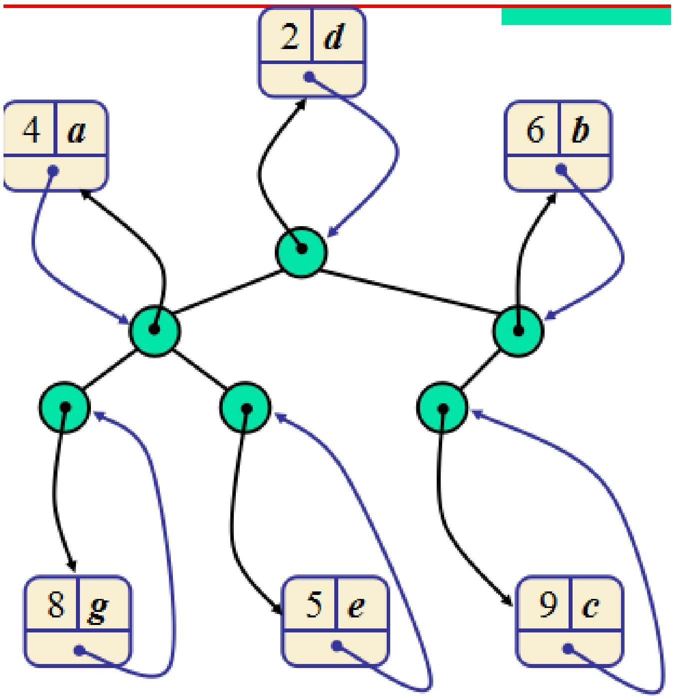
7.3.3 List Implementation

- location-aware list entry是一种object,包含以下项目:
  - key;
  - value;
  - position or rank of the item in the list;
- list中的每个position依次储存entry
- 在swap过程中back pointers(ranks)也被更新



7.3.4 Heap Implementation

- location-aware heap entry是一种object,包含以下项目:
  - key;
  - value;
  - position of the entry in the underlying heap;
- heap中的每个position依次储存entry
- 交换entries 时更新back pointers



7.3.5 性能对比

- location-aware可以优化需要预先定位的methods
- []为经过location-aware优化的methods

Methods	Unsorted list	Sorted list	Heap
size/isEmpty	1	1	1

Methods	Unsorted list	Sorted list	Heap
insert	1	n	log n
min	n	1	1
removeMin	n	1	log n
remove	[1]	[1]	[log n]
replaceKey	[1]	n	[log n]
replaceValue	[1]	[1]	[1]

Week 8 Sorting and Sets

- Merge Sort;
- Quick Sort;
- Bucket Sort;
- Radix Sort;
- Sorting Lower Bound;
- Union-Find Partition Structures;

8.1 Merge Sort

8.1.1 Devide and Conquer

- Merge-sort基于分治策略;
- 与heap对比
  - 都使用了comparator,时间$O(nlogn)$;
  - merge-sort不使用PQ，可以连续输入数据;

8.1.2 归并算法

```
Algorithm Merge(A,B)
Input:各包含n/2 元素的sequences A,B;
Output:已被排好序的序列AUB
{
    S=[empty sequence];
    while(-A.isEmpty() && -B.isEmpty()){//A与B非空
        if(A.first().element()<B.first().element())
            S.insertLast(A.remove(A.first()));
        else
            S.insertLast(B.remove(B.first()));
    } //A中当前元素与B中当前元素比较,比较小的插入S
```

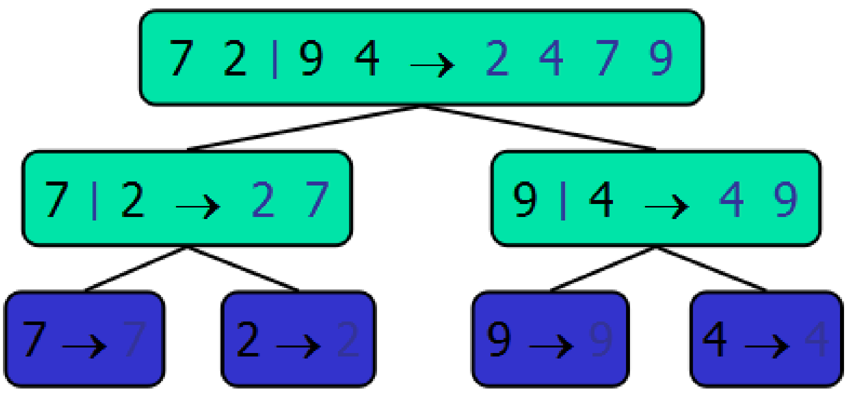
```
    while(A.isEmpty()==true){
        S.insertLast(B.remove(B.first()));
    } //A已空 , 将B剩余元素填入S
    while(B.isEmpty()==true){
        S.insertLast(A.remove(A.first()));
    } //B已空 , 将A剩余元素填入S

    return S;
}
```

- python 实现

```
def merge(listA,listB):
    resultList=[]
    while len(listA)!=0 and len(listB)!=0:
        if listA[0]<listB[0]:
            resultList.append(listA[0])
            listA=listA[1:]
        else:
            resultList.append(listB[0])
            listB=listB[1:]
    while len(listA)!=0:
        resultList.extend(listA)
    while len(listB)!=0:
        resultList.extend(listB)
    return resultList
```

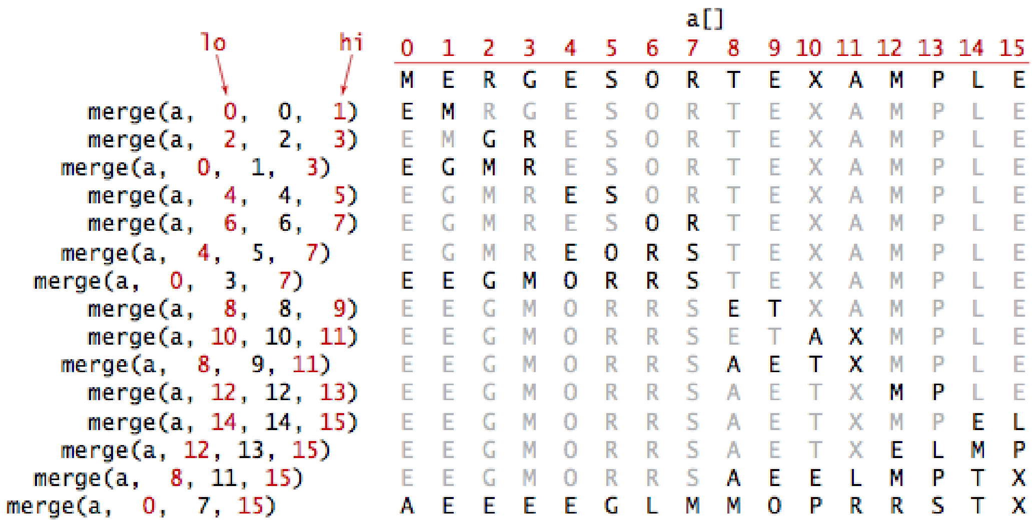
8.1.3 Merge-Sort Tree

- 从上到下devide , 从下到上conquer;
- root : initial call;
- leaves : calls on subsequences of size 0 or 1;



8.1.4 归并排序的Java实现

```java
public class Merge{
    private static Comparable[] temp; //建立辅助数组

    //method merge:对已经排好序的数组a[first,middle],a[middle+1,last]进行归并
    private static void merge(Comparable[] a,int first,int middle,int last){
        int i= first, j=middle + 1;  //[first,middle]+[middle+1,last]
        int m = middle,n = last;
        int k = 0;

        while (i<=m && j<=n){
            if (temp[i] <= temp[j])  a[k++] = temp[i++];
            else  a[k++] = temp[j++];
        }
        while (i<=m) a[k++] = temp[i++];
        while (j<=n) a[k++] = temp[j++];
    }

    //method sort:递归使用merge
    private static void sort(Comparable[] a,int first,int last){//a[first,last]排序,每个private sort包含两个sort 递归
        if(last<=first) return;
        int middle=first+(last-first)/2;//中间为分界点
        sort(a,first,middle);//前一半排序
        sort(a,middle+1,last);//后一般排序
        merge(a,first,middle,last); //调用merge
    }

    //main method
    public static void sort(Comparable[] a){
        temp=new Comparable[a.length]; //分配空间
        sort(a,0,a.length-1);//调用private method sort
    }
}
```

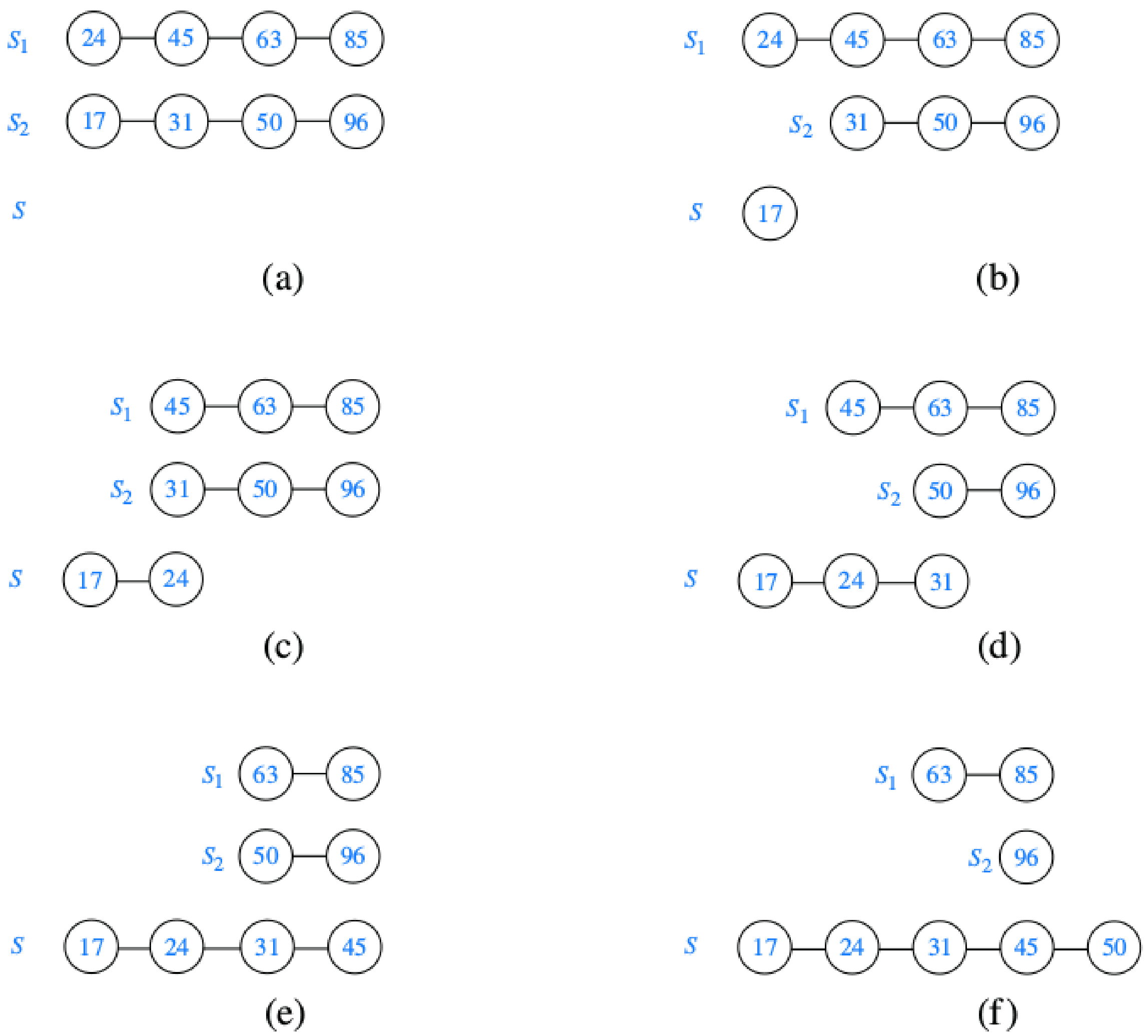


Trace of merge results for top-down mergesort

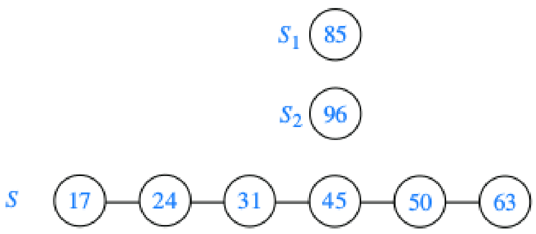
- 另一种实现方法: 基于链表队列

```
public static void merge(Queue Q1,Queue Q2,Queue Q, Comparator Comp){
    while(!Q1.isEmpty() && !Q2.isEmpty()){
        if(Comp.compare(Q1.first(),Q2.first())<0){//q1<q
            Q.enqueue(Q1.dequeue());
        }
        else{
            Q.enqueue(Q2.dequeue());
        }
    }
    while(!Q1.isEmpty()){
        Q.enqueue(Q1.dequeue());
    }
    while(!Q2.isEmpty()){
        Q.enqueue(Q2.dequeue());
    }
}

public static void mergeSort(Queue Q, Comparator Comp){
    int n=Q.size();
    if(n<2) return; //不需要排序了
    //devide: 前半部分放在Q1里, 后半部分放在Q2里
    Queue Q1=new LinkedQueue();
    Queue Q2=new LinkedQueue();
    while(Q1.size()<n/2){
        Q1.enqueue(Q.dequeue());
    }
    while(QQ.isEmpty()){
        Q2.enqueue(Q.dequeue());
    }

    //recursion(conquer)
```

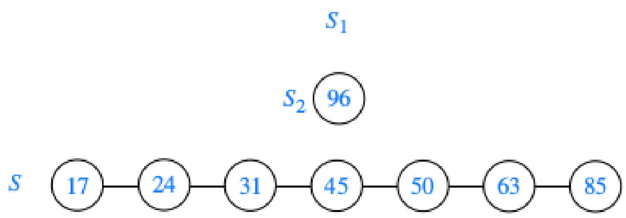
```
        mergeSort(Q1,Comp);
        mergeSort(Q2,Comp);
        merge(Q1,Q2,Q,Comp);
    }
```



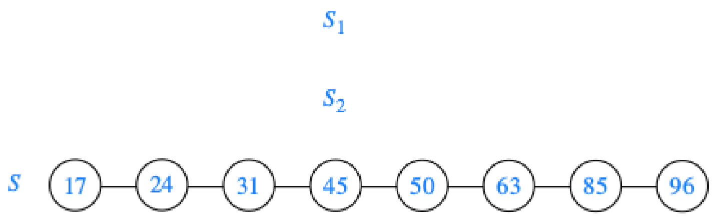
(a) (b) (c) (d) (e) (f)



(g)



(h)



(i)

- BU Merge: 自下而上

```
// stably merge a[lo .. mid] with a[mid+1 ..hi] using aux[lo .. hi]
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    // precondition: a[lo .. mid] and a[mid+1 .. hi] are sorted subarrays
    assert isSorted(a, lo, mid);
        assert isSorted(a, mid+1, hi);

    // copy to aux[]
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    // merge back to a[]
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if(i > mid) a[k] = aux[j++];
        else if (j > hi) a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else a[k] = aux[i++];
    }

    // postcondition: a[lo .. hi] is sorted
    assert isSorted(a, lo, hi);
}

// mergesort a[lo..hi] using auxiliary array aux[lo..hi]
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
```

```
        sort(a, aux, lo, mid);
        sort(a, aux, mid + 1, hi);
        merge(a, aux, lo, mid, hi);
    }
```

	a[i] 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 2																
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 4																
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 8																
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 16																
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for bottom-up mergesort

8.1.5 对归并排序的分析

- height : $O(logn)$
- 对于深度i的结点,工作量为$O(n)$
  - 分离归并2^i个sequences,每个size $\frac{n}{2^i}$;
  - 递归命令$2^(i+1)$;
- 归并排序的总运行时间$O(nlogn)$, 算是比较快的排序算法
- 但对于比较小的数组，可以使用插入等比较简单的算法;

8.2 快速排序

- 应用十分广泛,与归并排序是互补的;
- 归并排序先递归后排序,快速排序先排序后递归;
- 快排流程:先随机选择一个pivot X,分别对其左右部分进行排序,然后直接合并

8.2.1 Partition

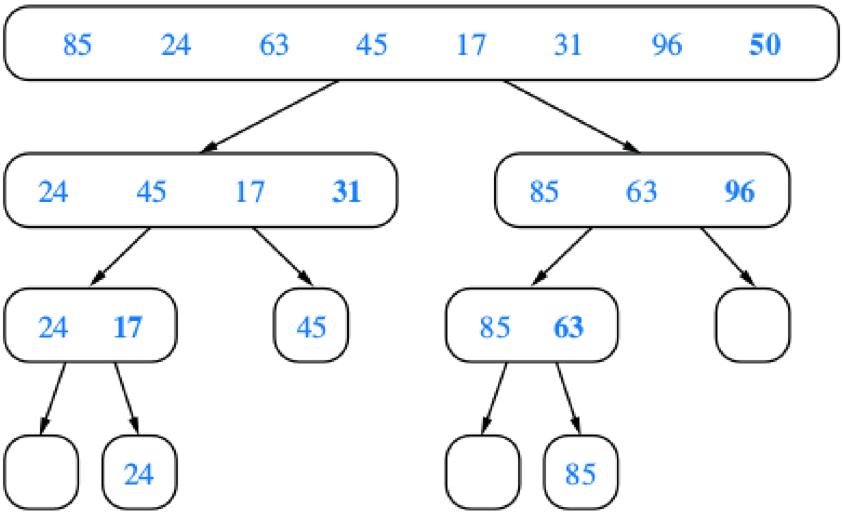
```
Algorithm Partition(S,p)
Input:Sequence S, Position p of the pivot;
Output:将S按小于等于大于pivot,分为三个子序列L/E/G;
{
    L,E,G=[empty];
```

```
    x=S.remove(p);//将x作为pivot从S中移除

    while(S.isEmpty()<0){//S中还有元素
        y=S.remove(S.first());
        if(y<x) L.insertLast(y);
        else if(y==x) E.insertLast(y);
        else G.insertLast(y);
    }
    return L,E,G;
}
```

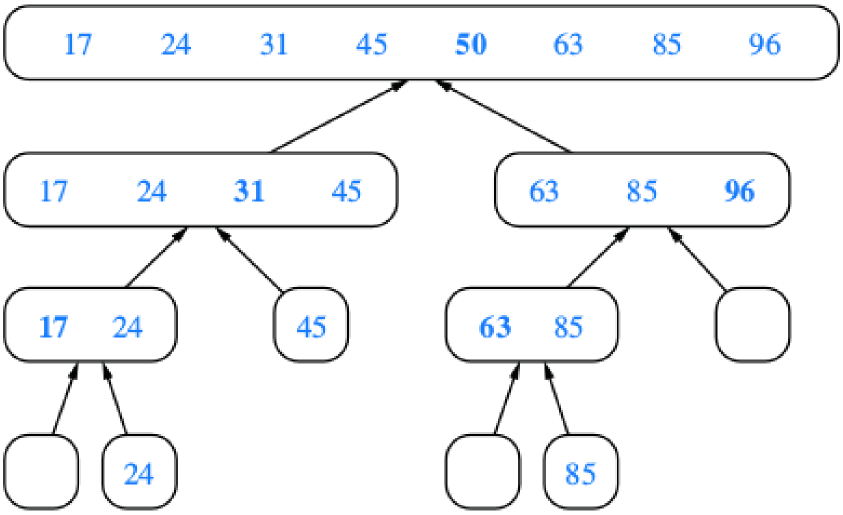
- 算法分析:
  - 该算法为非原位切分,需要用到额外数据结构(L/E/G);
  - 原序列头部删除,插入新序列尾部,每次操作$O(1)$;
  - 切分步骤总时间复杂度$O(n)$;

8.2.2 Quick-Sort Tree

- 下面的图每次都把当前最右边的作为切分比较点
  - $50 \rightarrow 31, 96 \rightarrow \ldots$



(a)



- Java 实现

```
public class Quick
{
    private static void sort(Comparable[] a,int first,int last)
    {
        if(last<=first) return;
        int j=partition(a,first,last); //该函数用于选定分界点a[j]
        sort(a,first,j-1);
        sort(a,j+1,last);
    }

    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);//输入数据后打乱,得到随机数组
        sort(a,0,a.length-1);
    }
```

```
}
```

8.2.3 Quick-sort 性能分析

- worst case $O(n^2)$: pivot是序列最小/最大值，相当于没切分
- average case : $O(nlogn)$
- good call : L/G的size都小于S的四分之三，获得good call的概率为50%;
- bad call : L/G之一大于S的四分之三;
- 快速排序的局限性:
  - 仅仅是比较高效的排序算法，有时不需要排序
  - 不大适合数据量小或bad call时 (e.g. 对有序数组, 建议使用bubble sort)

8.2.4 In-Place Quick-Sort

```
Algorithm inPlaceQS(S,l,r)
Input:Seq S,rank l,rank r;
Output:使用l/r划分并排序S中元素
{
    if ( l >= r ) return;
    在l,r之间随机选一个整数定为 i ;
    x=S.elemAtRank(i);
    p=inPlacePartition(x);
    inPlaceQS(S,l,p-1);
    inPlaceQS(S,p+1,r);
}
```

- In-place Partition
  - 使用两个指针j、k将S分为L/E/G三个子集;
  - 使用j指针从左至右扫描直到找到大于等于x的元素j';
  - 使用k指针从右至左扫描直到找到小于x的元素k';
  - j'/k' 交换位置;
  - 继续上述过程直到两个指针相遇;

```
Algorithm InPlacePartition(S,0,n-1):
Input:Sequence S with n elements;
Output: the partition point
{
    pivot=S[0];
    point left=1;
    point right=n-1;
    while(true){
        while (S[left]<=pivot){ //从左到右运行left指针
            left++;
```

```
                if (left>n-1) break;
            }
            while (S[right]>pivot){ //从右到左运行right指针
                right--;
                if (right<0) break;
            }
            if (left>=right) break; //指针相遇 , 切分结束
            exchange(S[left],S[right]); //指针未相遇 , 交换元素
        }
        exchange(pivot,S[right]);
        return right;
    }
```

8.2.5 对排序算法的总结

- 排序算法稳定性：相等的两个元素排序前后相对次序不变
    - 稳定排序：插入/归并/冒泡
    - 不稳定排序：选择/快速/heap

Algorithm	Time	Notes
Selection-sort	$O(n^2)$	原位，不稳定，适合小型序列
Insertion-sort	$O(n^2)$	原位，稳定，适合小型序列
Heap-sort	$O(nlogn)$	原位，不稳定，空间复杂度 $O(1)$
Merge-sort	$O(nlogn)$	非原位，稳定，空间复杂度 $O(n)$
Quick-sort	$O(nlogn)$	原位，不稳定，随机序列最快算法, 但不适合有序序列

8.2.8 Java实现(8.2.2的扩展版)

```
public static void quickSort(Object[] S, Comparator c){
    if (S.length<2) return; //已经排好序了
    quickSortStep(S,c,0,S.length-1);
}

private static void quickSortStep(Object[] S, Comparator c,
int leftBound, int rightBound ){
    if (leftBound >= rightBound) return; //左边界>=右边界->null
    Object temp;  // temp object used for swapping
    Object pivot = S[rightBound];//初始化最右元素为pivot
    int leftIndex = leftBound;
    int rightIndex = rightBound-1;

    while (leftIndex <= rightIndex) {
        //向右扫描寻找大于pivot的
```

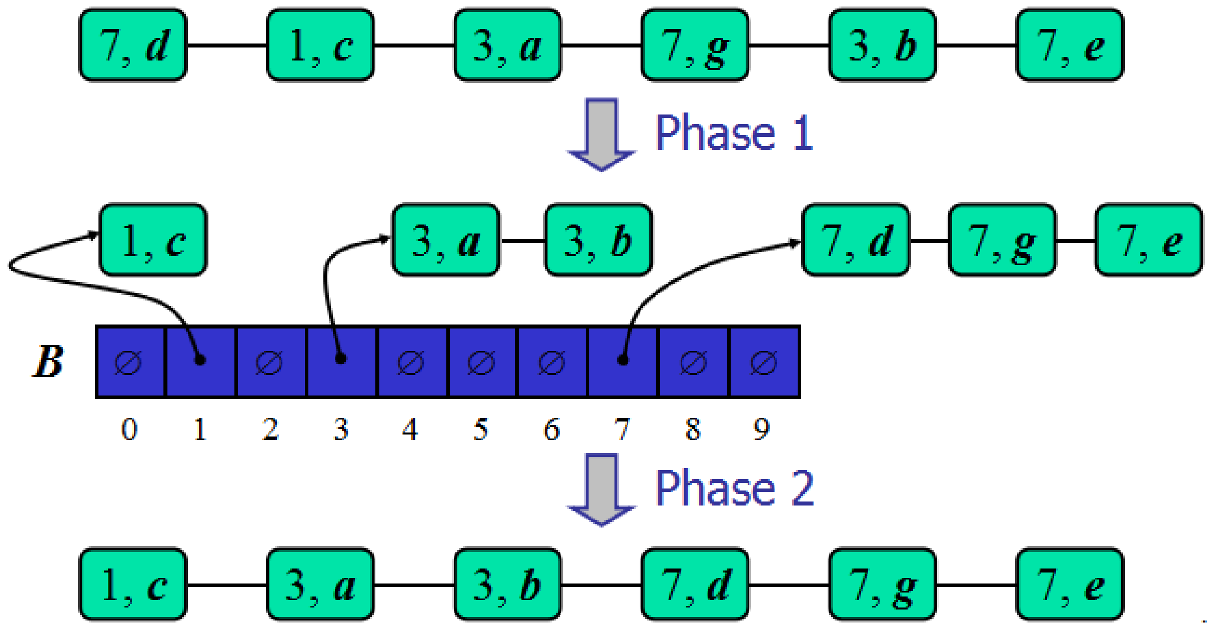
```
            while (c.compare(S[leftIndex],pivot)<=0)
                leftIndex++;
            //向左扫描寻找小于pivot的
            while (c.compare(S[rightIndex],pivot)>=0))
                rightIndex--;
            //交换找到的两个元素
            temp = S[rightIndex];
            S[rightIndex] = S[leftIndex];
            S[leftIndex] = temp;
        } //不断循环直到指针相遇
        //将pivot从最右交换到指针相遇的地方S[leftIndex]
        temp = S[rightBound];
        S[rightBound] = S[leftIndex];
        S[leftIndex] = temp;

        quickSortStep(S,c,leftBound,leftIndex-1);
        quickSortStep(S,c,leftIndex+1,rightBound);
    }
```

8.3 Bucket-Sort and Radix-Sort

8.3.1 Bucket-Sort

- 基本思想:
    - 将key相同的entries归类到一个个bucket中;
    - 先对每个bucket中的元素按value排序;
    - 对每个bucket按key排序

```
Algorithm BucketSort(S,N)
Input : n个entries,N个keys (n >= N);
Output : sorted S ;
{
    B=N个空序列组成的数组;
    while(S.isEmpty()<0){
        (k,o)=S.remove(S.first());
        B[k].insertLast((k,o));
    }//将key相同的entries收到同一个bucket里

    for(i=0;i++;i<=N-1){
        while(B[i].isEmpty()<0){
            //这里不需要考虑排序,直接扔出来就好
            (k,o)=B[i].remove(B[i].first());
            S.insertLast((k,o));
        }
    }//对每个bucket分别排序,然后按key大小再塞回S
}
```



- 算法分析:
  - Phase 1 $O(n)$: 将entry(k,o)按key移入bucket B[k],逐渐清空S;
  - Phase 2 $O(n+N)$: for i=0,...,N-1,将B[i]的entry移到S尾部;
  - Bucket-Sort总时间$O(n+N)$
  - bucket是一种稳定排序：任意两个key相同的items的相对次序被保存

8.3.2 Lexicographic Sort

Example 12.5: *Consider the following sequence S (we show only the keys):*

$$S = ((3,3),(1,5),(2,5),(1,2),(2,3),(1,7),(3,2),(2,2)).$$

If we sort S stably on the first component, then we get the sequence

$$S_1 = ((1,5),(1,2),(1,7),(2,5),(2,3),(2,2),(3,3),(3,2)).$$

If we then stably sort this sequence S_1 using the second component, we get the sequence

$$S_{1,2} = ((1,2),(2,2),(3,2),(2,3),(3,3),(1,5),(2,5),(1,7)),$$

which is unfortunately not a sorted sequence. On the other hand, if we first stably sort S using the second component, then we get the sequence

$$S_2 = ((1,2),(3,2),(2,2),(3,3),(2,3),(1,5),(2,5),(1,7)).$$

If we then stably sort sequence S_2 using the first component, we get the sequence

$$S_{2,1} = ((1,2),(1,5),(1,7),(2,2),(2,3),(2,5),(3,2),(3,3)),$$

which is indeed sequence S lexicographically ordered.

- Lexicographic Order
  - d-tuple : 一个带有d个keys$(k_1, k_2, \ldots, k_d)$的序列;
  - k_i : tuple的第i个维度;
  - 两个d-tuple的字典顺序定义如下: 先比较第一维，然后第二维，第三维...

$$(x_1, x_2, \ldots x_d) < (y_1, y_2, \ldots y_d)$$

$$\Leftrightarrow$$

$$(x_1 < y_1) \cup (x_1 y_1) \cap [(x_2, \ldots x_d) < (y_2, \ldots y_d)]$$

- 字典排序算法
  - C_i : 用于比较i维的comparator;
  - stableSort(S,C_i) : 使用C_i进行比较的稳定排序算法;
  - 通过执行N次stableSort(每次一个维度),将N-tuples的序列排序为字典顺序;
  - 运行时间$O(NT(n))$，其中$T(n)$是每次stableSort的运行时间;

```
Algorithm lexicographicSort(S)
Input:sequence S of N-tuples;
Output:sequence S sorted in lexicographic order;
{
    for ( i = N; i>=1; i--; )
    //每次循环先决定尾部数字的相对顺序
        stableSort(S,Ci);
}
```

- 举个栗子

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)

(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)

(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

8.3.3 Radix-Sort

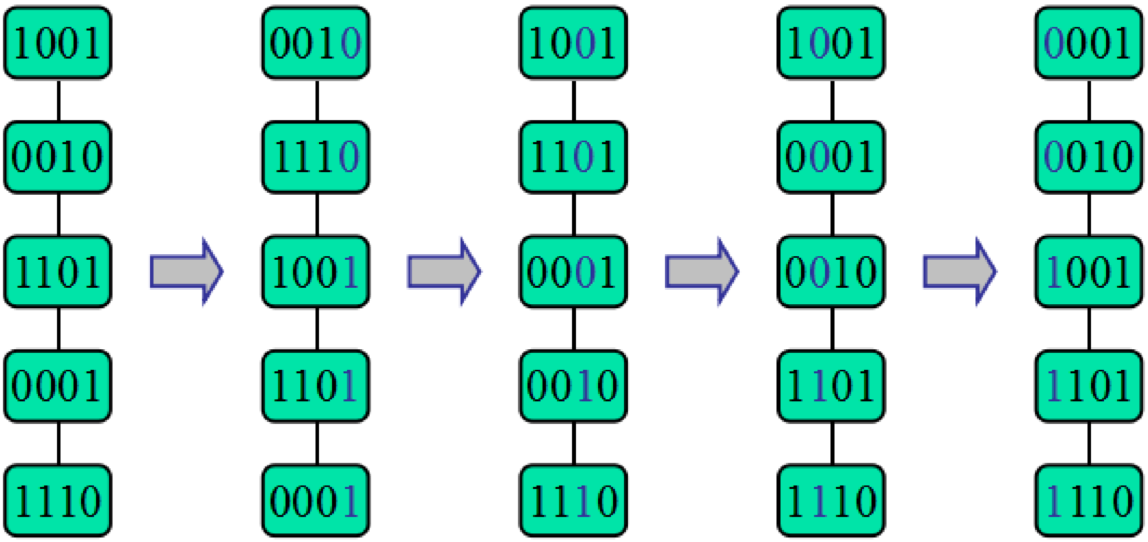
- Radix-Sort是一种特殊的字典排序，使用bucket排序作为stableSort
  - 每次选取的bucket其实就是字典排序从后向前的位数
  - 将选出来的bucket key进行字典排序, 然后将entries归位
  - 更换位数(bucket)开始下一次排序
- 适用条件: tuples,每个维度i的keys是[0,N-1]范围内的整数;
- 运行时间$O(d(n+N))$

```
Algorithm radixSort(S,N)
Input: 包含N-tuples的序列S.对于每个tuple(x1,...xN)in S,xi∈[0,N-1];
Output: 字典顺序的S;
{
    for(i=N;i>=1;i--)
        bucketSort(S,N);
}
```

- 举个栗子: 使用Radix-Sort对二进制数进行排序
  - n个b-bit整数组成的序列$x=x_{b-1}...x_1x_0 , e.g. "10110110";
  - 对每个元素进行radix排序(N=2)
  - 运行时间$O(bn)$, 取决于序列元素数n及位数b
  - 位数固定的话，为线性复杂度

```
Algorithm binaryRadixSort(S)
```

```
Input: S of n b-bit integers;
Output:对S进行排序,S中每个元素x被替换为item(0,x);
{
    for(i=0;i<=b-1;i++){
        使用x中的bit位xi替换(k,x)中的key k;
        bucketSort(S,2);
    }
}
```

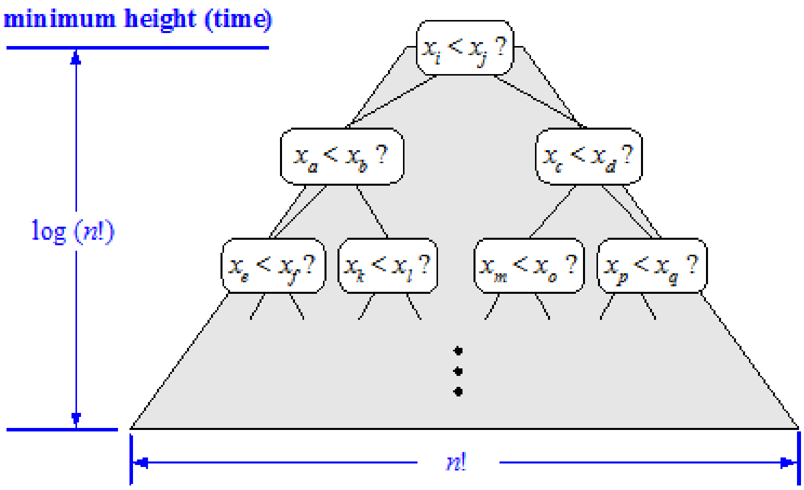


- 再举个栗子: via VisualAlgo
  - 原数据:
    $$[3221, 1, 10, 9680, 577, 9420, 7, 5622, 4793, 2030, 3138, 82, 2599, 743, 4127]$$
  - 最后一位构建bucket并对bucket的key字典排序
    $[10, 9680, 9420, 2030], [3221, 1], [5622, 82], [4793, 743], [577, 7, 4127], [3138], [2599]$
  - 归位:
    $$[10, 9680, 9420, 2030, 3221, 1, 5622, 82, 4793, 743, 577, 7, 4127, 3138, 2599]$$
  - 倒数第二位构建bucket并对bucket的key字典排序
    $[1, 7], [10], [9420, 3221, 5622, 4127], [2030, 3138], [743], [577], [9680, 82], [4793, 2599]$
  - 归位:
    $$[1, 7, 10, 9420, 3221, 5622, 4127, 2030, 3138, 743, 577, 9680, 82, 4793, 2599]$$
  - 倒数第三位构建bucket并对bucket的key字典排序
    $[1, 7, 10, 2030, 82], [4127, 3138], [3221], [9420], [577, 2599], [5622, 9680], [743, 4793]$
  - 归位:
    $$[1, 7, 10, 2030, 82, 4127, 3138, 3221, 9420, 577, 2599, 5622, 9680, 743, 4793]$$
  - 倒数第四位构建bucket并对bucket的key字典排序
    $[1, 7, 10, 82, 577, 743], [2030, 2599], [3138, 3221], [4127, 4793], [5622], [9420, 9680]$
  - 归位, 排序完毕:

$$[1, 7, 10, 82, 577, 743, 2030, 2599, 3138, 3221, 4127, 4793, 5622, 9420, 9680]$$

8.4 Sorting Lower Bound

- 之前见到的几种排序都是基于比较的(comparison-based)
- 决策树：最小高度为$log(n!)$



- Lower bound：注意任何基于比较的排序算法时间复杂度至少为$log(n!)$

$$log(n!) \geq (n/2)log(n/2)$$

- 排序算法的时间复杂度下限 $\Omega(nlogn)$

8.5 Selection

- 选择是一个先排序后查找的过程

8.5.1 Quick-Select

- 基于prune-and-search的随机化选择;
- 选择排名为k的元素(第k小)
- prune：随机选取pivot x，将S切分为L/E/G;
- search：选择在E/L/G三个子集中的一个进行递归;

```
Algorithm QuickSelect(S,k)
Input: Sequence S, rank k
Output: the kth smallest element in S
{
    if(length(S)==1){
        return S.first();
    }
    randomly pick a position p;
```

```
        L,E,G=Partition(S,p);
        if(length(L)>=k){
            return QuickSelect(L,k);
        }
        else if(length(L)+length(E)>=k){
            return E.first();
        }
        else{
            return QuickSelect(G,k);
        }
    }
```

8.5.2 Partition

```
Algorithm Partition(S,p)
Input:sequence S, position p
Output:subsequences L/E/G
{
    L,E,G = [empty sequences];
    x = S.remove(p);
    while (S.isEmpty()<0)
    {
        y = S.remove(S.first());
        if (y<x) L.insertLast(y);
        else if (y==x) E.insertLast(y);
        else G.insertLast(y);
    }
    return L,E,G;
}
```

- 算法分析:
  - 逐步移除-比较-插入合适的子序列
  - partition操作总时间$O(n)$

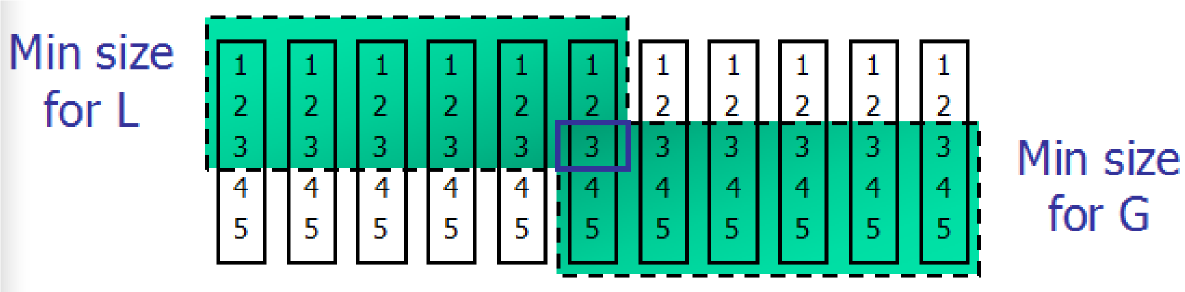
8.5.3 快速选择的性能

- 类似之前LEG的划分:
  - good call : LG都小于S的四分之三;
  - bad call:LG有一个过大了;
- 平均运行时间$O(n)$

8.5.4 使用Deterministic Selection 优化选择pivot的过程

- 最坏情况下依然$O(n)$
- 主要思想 : 通过递归算法本身,寻找合适的pivot进行快速选择
  - 将S每5个元素分成一组, 获得n/5个子集;
  - 选取每个子集的中位数, 形成一个新集合;

- 给新集合分组取中位数, 不断递归;
- 最后余下的元素为合适的pivot;



8.6 Sets

8.6.1 Set Operations

- Basic operations :
  - union;
  - intersection;
  - subtraction;
- Set union : "或" , AB中所有的元素都会在S中出现

```
if (aIsLess(a,S)) {S.insertFirst(a);}
if (bIsLess(b,S)) {S.insertLast(b);}
if (bothAreEqual(a,b,S)) {S.insertLast(a);}
```

- Set intersection : "与" , S中的元素为AB都有的

```
if(aIsLess(a,S)) {do nothing;}
if(bIsLess(b,S)) {do nothing;}
if(bothAreEqual(a,b,S)) {S.insertLast(a);}
```

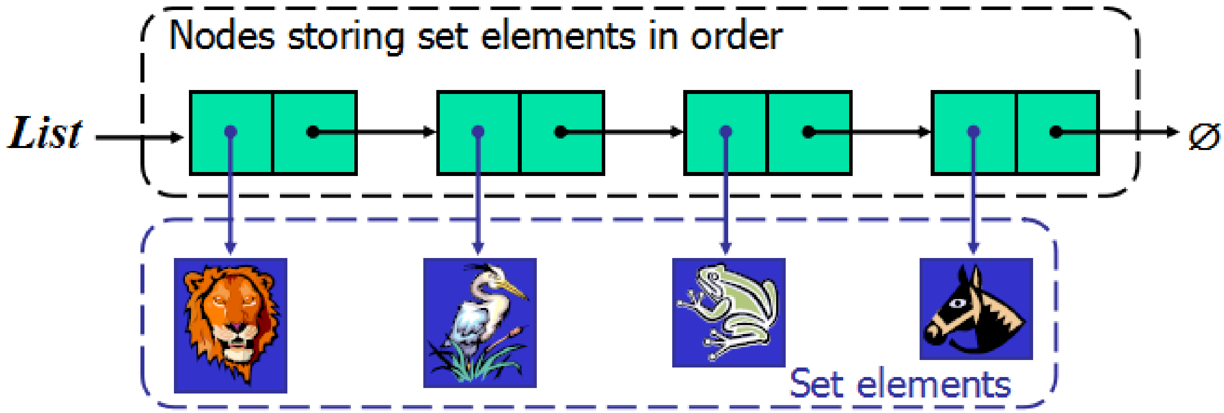
- Set subtraction : "非" , A中有而B中没有的元素

```
if ((aIsless(a,S)) && (!bIsLess(b,S))) {S.insertLast(a);}
```

- 基本的set操作运行时间$O(N_a+N_b)$, 取决于set A/B的大小

8.6.2 在List中储存set

- 空间复杂度 $O(n)$;



8.6.3 Generic Merging

- 合并两个有序lists A,B
- Methods:
  - genericMerge , $O(n_A + n_B)$
  - aIsLess , $O(1)$
  - bIsLess , $O(1)$
  - bothAreEqual , $O(1)$

```
Algorithm genericMerge(A,B){
    S=[empty sequence];
    while((-A.isEmpty()) && (-B.isEmpty())){ //AB均非空
        a=A.first().element();
        b=B.first().element();
        if(a<b){
            aIsLess(a,S);
            A.remove(A.first());
        }
        else if(a>b){
            bIsLess(B,s);
            B.remove(B.first());
        }
        else{
            bothAreEqual(a,b,S);
            A.remove(A.first());
            B.remove(B.first());
        }
    }
}
```

- 算法分析
  - 类似归并排序：A,B第一个元素比较,比较小的那个从set中剪切到S...
  - 在后三个methods运行时间为$O(1)$的前提下，genericMerge复杂度 $O(n_A + n_B)$

8.6.4 使用generic merge进行set操作

- 任何set operations可以使用泛型归并实现
- 几个栗子:
  - intersection : 两个list共有的元素
  - union : 两个list所有的元素，但重复元素只添加一次
- 所有的methods都是线性复杂度

8.7 Union-Find Partition Structures

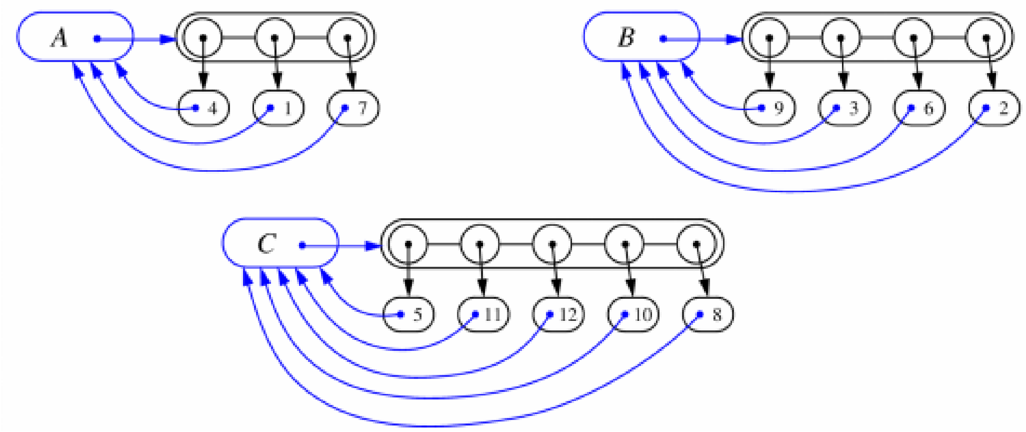
- 用于处理一些不相交集合（Disjoint Sets）的合并及查询问题
- 时间复杂度 $O(nlogn)$
- 举个栗子: 给出一组结点, 判断是否连通

8.7.1 Partitions with Union-Find Operations

- makeSet(x) : 创建一个包含x的singleton set(单元素集)并返回该set中存储x的位置;
- union(A,B): 返回set($A \cup B$) 并删除两个旧数据集;
- find(p): 确定元素p属于哪些子集，返回包含元素p的sets;

8.7.2 基于list的实现

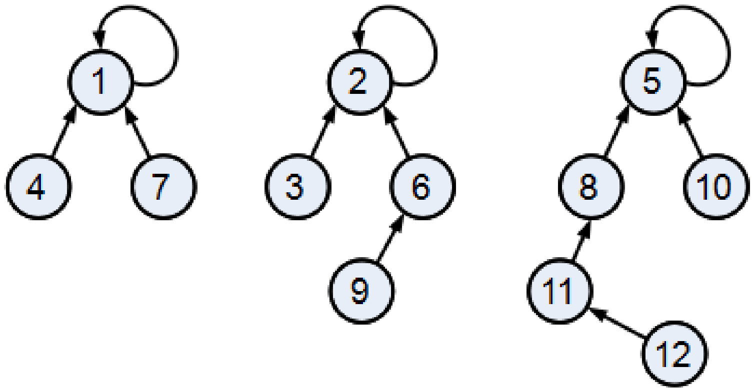
- 每个set储存在链表序列内;
- 每个node储存一个object(element,reference)，reference都指向链表的名字;



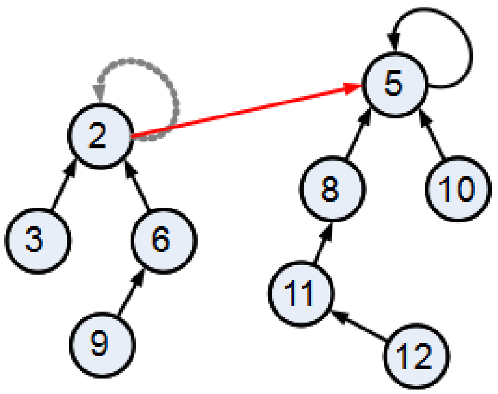
- 分析:
  - 每次union操作：将元素从小set移动到大set;
  - 这个大set的size至少是小set的两倍;
  - 因此一个元素最多被移动$O(logn)$时间;
  - n次union-find耗时$O(nlogn)$;

8.7.3 基于tree的实现

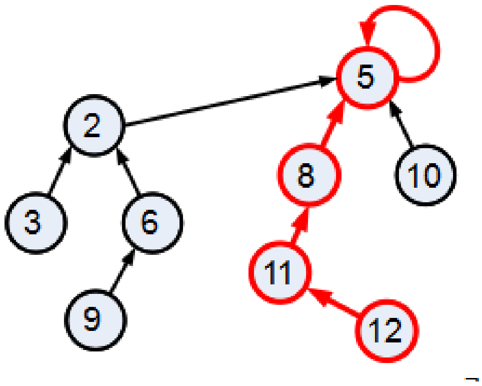
- 每个元素储存在node里,且包含一个指向set名的指针;
- 对于根结点v,set指针指向自身,那么v也是一个set名;
- 子结点的链接都是指向父结点的;
- 每个set都是一棵树:根为指针指向自身的node;



- Union-Find Operations
  - union: 将小树的root指向大树的root ,小树被并入大树;



- find : 返回这个node所在的root(set name)
  - 从指定node开始向上查找,直到到达指向自身的那个node(root);
  - 找到了根结点就可以得知set的名字了;



8.7.4 使用python实现union-find

```
def find(c): #find the root of c
    '''
    5->0->1, 1 is root
    id[1]=1, find(1)=1
    id[0]=id[id[1]]=1,find(0)=1
    id[5]=id[id[0]]=id[id[id[1]]],find(5)=1
    '''
    while(c!=id[c]):
        c=id[c]
    return c

def connected(a,b):
    if find(a)==find(b):
        return True
    else:
        return False

def union(A,B): #B的id设置为A的id
    '''
    union(3,5)
    id[3]
    >>>3
    id[5]
    >>>3
    '''
    AID=find(A)
    BID=find(B)
    if AID==BID:
        return
    else:
        id[BID]=AID


if __name__=="__main__":
    data_list= [(3, 5),(4, 2),(5, 2),(2, 1),(3, 1),(4, 1)]
```

```python

    #初始化所有元素id为自身(全都是root)
    id=[0]*10
    for item in data_list:
        id[item[0]]=item[0]
        id[item[1]]=item[1]

    nonredundant=[]
    for item in data_list:
        a=item[0]
        b=item[1]
        if connected(a,b)==True:
            print "{} and {} is connected".format(a,b)
            continue
        else:
            nonredundant.append(item)
            union(a,b)
    print "The nonredundant facts are:"
    for item in nonredundant:
        print "R({},{})".format(item[0],item[1])
```

Week 9 Maps and Dictionaries

- Maps;
- Hash Tables;
- Dictionaries;
- Skip Lists;

9.1 Maps

- map : 可搜索的键值对集合
- map不允许相同的key出现
- main operations : searching/inserting/deleting
- applications :
  - 无序Map: 计算词频
  - 有序Map: 航班数据库
- 注意哈希表只能用于处理无序Map,因为处理冲突时会打乱顺序
- 跳表可用于处理有序Map,始终是有序的

9.1.1 Map ADT

- get(k) : 返回key/null
- put(k,v) : k存在则赋值并返回旧value , k不存在则添加键值对

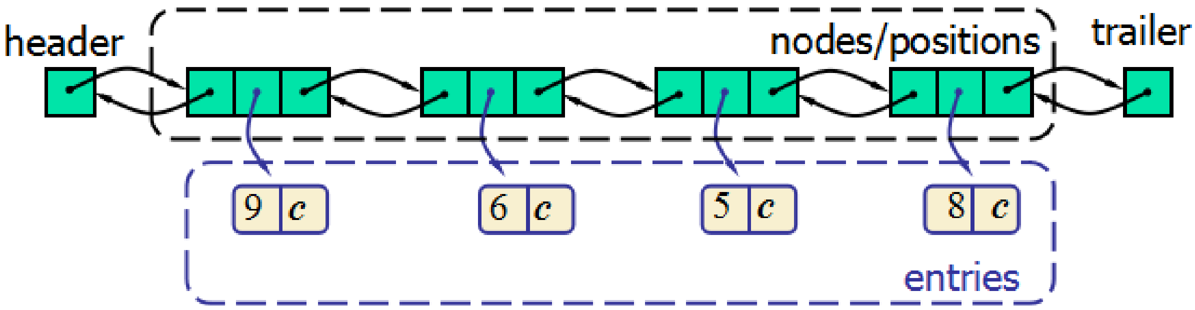
- remove(k) : 删除key-value,返回value,不存在返回null
- size()/isEmpty();
- keys()/values()/entries();

- java interface

```
public interface Map<Key,Value>{
    public Value get(Key k);
    public Value put(Key k,Value v);
    public Value remove(Key k);
    public int size();
    public boolean isEmpty();
    public Iterable<Key> keys();
    public Iterable<Value> values();
    public Iterable<Key,Value> entries();
}
```

9.1.2 list-based map

- based on unsorted list



- 基本操作算法

```
Algorithm get(k){
    B=S.positions(); //B:迭代器,遍历S中的positions
    while(B.hasNext()){
        p=B.next();
        if(p.element().key()==k)
            return p.element().value();
    }
    return null;//No match
}

Algorithm put(k,v){
    B=S.positions();
    while(B.hasNext()){
        p=B.next();
        if(p.element().key()==k){
```

```
                t=p.element().value();
                B.replace(p,(k,v));
                return t;
            }
        }
        S.insertLast((k,v));
        n=n+1;
        return null;
    }

    Algorithm remove(k){
        B=S.positions();
        while(B.hasNext()){
            p=B.next();
            if(p.element().key()==k){
                t=p.element().value();
                S.remove(p);
                n=n-1;
                return t;
            }
        }
        return null;
    }
```

- java实现参考 *Data Structures and Algorithms by Goodrich and Tamassia* P408
- 性能分析
  - put() $O(1)$: 对于unsorted list可以直接在头尾插入
  - get()/remove() $O(n)$: 最差情况是遍历所有后找不到
  - unsorted-list 适合小map或者主要操作为put()的map

9.2 Hash Tables

- 对map中的keys使用哈希函数，改变keys的格式

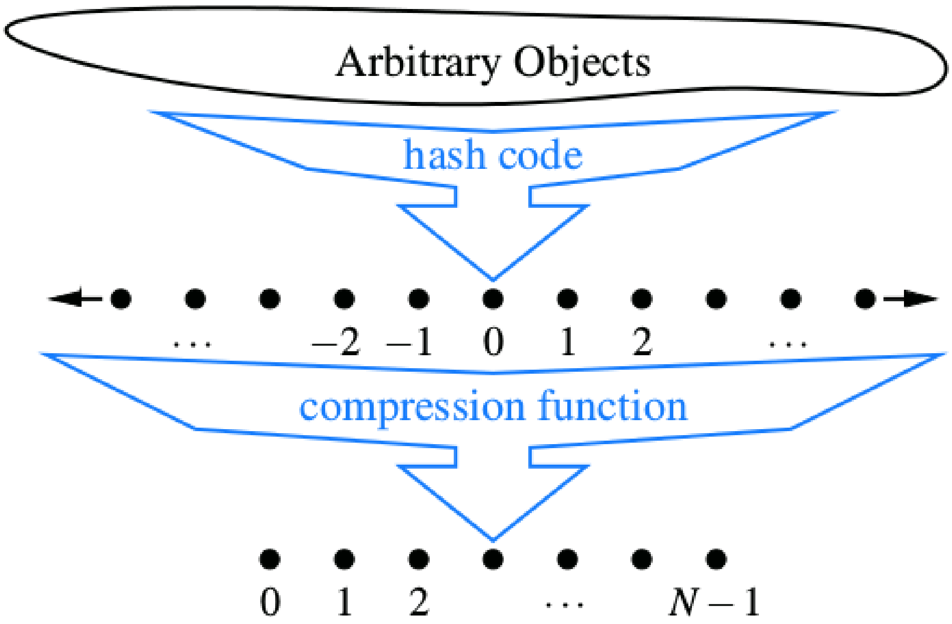
9.2.1 Hash Functions and Hash Tables

- 哈希函数h : 将key x转换为hash value ([0,N-1]范围内的整数);
  - 举个栗子 h(x)=x mod N
  - h is hash function , h(x) is hash value
- 哈希表 : 包含哈希函数h以及一个size为N的array
  - hash table的目标在于将键值对(k,o)存储在index i=h(k)的位置
- 使用哈希表实现map:
  - 将map中每个键key转换为哈希值h(key);

- 将哈希值存储在哈希表A中: A[h(key)]=key;
- 一般情况下A[i]=i仅包含一个元素，否则造成冲突(两个keys的哈希值相同);
- 处理冲突:
  - Separate Chaining;
  - Open Addressing;
  - Linear Probing;

9.2.2 Hash functions

- 哈希函数由以下两部分构成:
  - Hash code h1 : keys -> integers(hash code)
  - Compression function h2 : integers -> [0,N-1]



- $h(x) = h_2(h_1(x))$ 先使用h1将原值变为哈希码,再使用h2将哈希码变为哈希值
- 分成两部分的优点:
  - 仅仅是h2受哈希表size影响;
  - h1可被用于任何size的哈希表;

9.2.3 Hash codes(h1 function)

- Memory address
  - 将key object的内存地址转化为integer;
  - 不适合numeric/strings;
- Integer cast
  - 将键的bits转化为integer作为哈希码;
  - 适合byte/short/int/float这些比较短的类型;

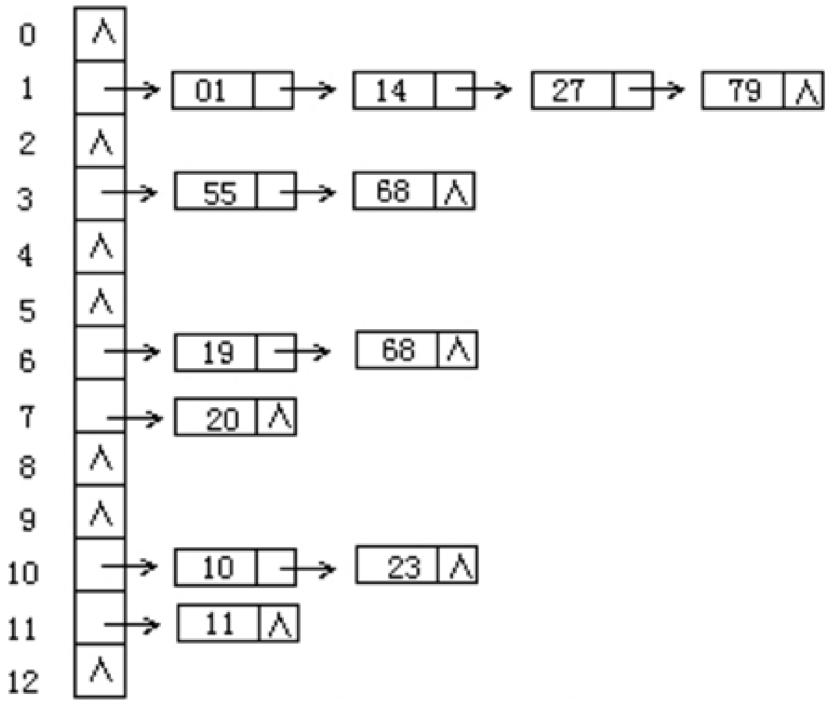
    - 举个栗子 : Float.floatToIntBits(x)
- Component sum
    - 将key的bits切分为固定长度(16/32 bits)的部分并累加;
    - 适合long/double这些比较长的integer类型;
- Polynomial accumulation
    - 切分(8/16/32)但并非累加而是组成多项式$a_0 a_1 \dots a_{n-1}$;
    - 给定一个常数z计算 $p(z) = a_0 + a_1 z + a_2 z^2 + \dots a_{n-1} z^{n-1}$;
    - 适合strings;
    - p(z)可以使用Horner规则计算, $O(n)$
        - $p_0(z) = a_{n-1}$
        - $p_i(z) = a_{n-i-1+zpi-1}(z), i = [1, \dots n-1]$
        - $p(z) = p_{n-1}(z)$
    - 变种: Cyclic shift

9.2.4 Compression functions(h2 function)

- Division : 求哈希码与哈希表size的余数
    - $h_2(y) = y \bmod N$;
    - 选取质数N作为哈希表容量;
- Multiply,Add and Divide(MAD):
    - $h_2(y) = [(ay + b) \bmod p] \bmod N$;
    - p为大于N的质数;
    - $a, b \in [0, p-1]$;
    - $a \bmod N! = 0$;

9.2.5 处理冲突

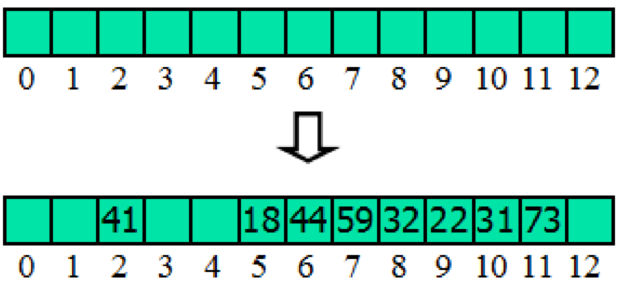
- Separate chaining
    - 对于存在冲突的bucket(不止一个元素),将这些元素加入一个新链表
    - 优点:简单易行
    - 缺点:需要额外空间(每个冲突项都需要改造成一个链表)



- Map Methods with Separate Chaining used for Collisions

```
Algorithm get(k)
Output: 在map中找到key并返回值,其中A为哈希表
{
    return A[h(k)].get(k);   // delegate the get to the list-based map at A[h(k)]
}

Algorithm put(k,v)
Output:insertion or set
{
    t = A[h(k)].put(k,v);    // delegate the put to the list-based map at A[h(k)]
    if (t==null)
        n = n + 1;
    return t ;
}

Algorithm remove(k)
Output:删除key并返回值,找不到返回null
{
    t = A[h(k)].remove(k);
    if (t!=null) // k was found
        n = n-1;
    return t ;
}
```

- Open Addressing:
  - 将冲突项目放进表中的另一个cell;
  - 优点:不需要额外的数据结构;
  - 缺点:有点复杂
  - 根据存储方式不同有多个变种,如Linear Probing;
- Linear probing:
  - Open Addressing的一种;
  - 将冲突项目放入表中下一个空cell;
  - 每一个cell都可以被用作probe;
  - 直接将冲突项存储于哈希表本身;
    - $h(x) = x \bmod 13$
    - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



- Search with Linear Probing
  - 起始位置:cell h(k)
  - 连续probe坐标直到:
    - 找到key k;
    - 找到空cell;
    - 遍历N cells后没找到;

```
Algorithm get(k){
    hashValue=h(k);
    p=0;
    while(p!=N){
        c=A[hashValue];
        if(c==null) return null;
        else if(c.key()==k) return c.element();
        else{
            hashValue=(hashValue+1) mod N;
```

```
            p=p+1;
        }
    }
    return null;
}
```

- Updates with Linear Probing
  - 特殊对象AVAILABLE：用于替换被删除的元素

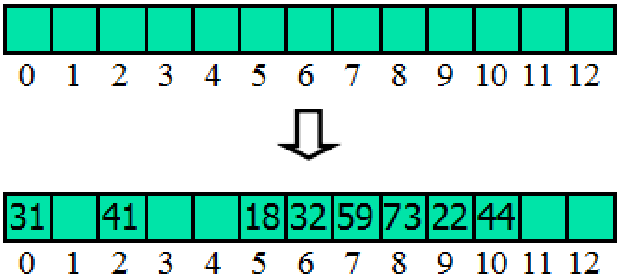
```
Algorithm put(k,o)
{
    if the table is full->throws an exception;
    起始位置:cell h(k)
    连续probe cell直到:
        找到i=h(k)->替换内容;
        没找到->在cell i新增键值对(k,o);
}

Algorithm remove(k)
{
    找寻key k的键值对;
    找到了->替换该键值对为特殊项目AVAILABLE,并返回o;
    没找到->返回null;
}
```

- $\text{Load factor} = \frac{n}{N}$
  - 对于separate chaining, 载荷不能过于接近1防止过载
  - 对于open addressing, 则可以尽可能填满空位

9.2.6 Double Hashing

- 使用二级哈希函数d(k)
  - $(i + jd(k)) mod N$;
- 注意该函数不能有零点;
- 发生冲突时,将冲突项目置于$(i + jd(k)) mod N, j = [0, \ldots N-1]$中第一个空置的cell;
- N必须为质数
- 常用compression functions:
  - $d_2(k) = q - k mod q$, q为小于N的质数;
  - $d_2(k)$的可能取值为1,2,...q;

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



9.2.7 Hashing 的性能

- worst case $O(n)$:插入搜索删除过程中所有的keys都冲突;
- 哈希表中每个bucket包含的key平均数量为: n/N;
- load factor α= n/N会影响哈希表性能;
- 如果n=N(无冲突), 操作复杂度$O(1)$
- 各种操作平均运行时间为$O(1)$, 强于unsorted list 的$O(n)$
- 哈希表适用于：小型数据库/编译器/浏览器搜索

Method	Unsorted List	Hash Table	
		expected	worst case
get	$O(n)$	$O(1)$	$O(n)$
put	$O(n)$	$O(1)$	$O(n)$
remove	$O(n)$	$O(1)$	$O(n)$
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
entrySet, keySet, values	$O(n)$	$O(n)$	$O(n)$

9.2.8 Java实现hash map ：

```
public class HashTableMap<K,V> implements Map<K,V> {

    /** Nested class: HashEntry */
    public static class HashEntry<K,V> implements Entry<K,V> {
        protected K key;
        protected V value;
        public HashEntry(K k, V v) { key = k; value = v; }
```

```java
        public V getValue() { return value; }
        public K getKey() { return key; }
        public V setValue(V val) {
            V oldValue = value;
            value = val;
            return oldValue;
        }
        public boolean equals(Object o) {
            HashEntry<K,V> ent;
            try { ent = (HashEntry<K,V>) o; }
            catch (ClassCastException ex) { return false; }
            return (ent.getKey() == key) && (ent.getValue() == value);
        }

        /** virsualization, override toString() */
        public String toString() {
              return "(" + key + "," + value + ")";
        }
    }

    protected Entry<K,V> AVAILABLE = new HashEntry<K,V>(null, null);
    protected int n = 0;        // number of entries in the dictionary
    protected int prime, capacity; // prime factor and capacity of bucket array
    protected Entry<K,V>[] bucket;// bucket array
    protected long scale, shift;   // the shift and scaling factors

    /** Constructor1:  prime factor 109345121 and capacity 1000. */
    public HashTableMap() { this(109345121,1000); }
    /** Constructor2:  prime factor 109345121 and given capacity. */
    public HashTableMap(int cap) { this(109345121, cap); }
    /** Constructor3:  given prime factor and capacity. */
    public HashTableMap(int p, int cap) {
        prime = p;
        capacity = cap;
        bucket = (Entry<K,V>[]) new Entry[capacity]; // safe cast
        java.util.Random rand = new java.util.Random();
        scale = rand.nextInt(prime-1) + 1;
        shift = rand.nextInt(prime);
    }

    /** Determines whether a key is valid. */
    protected void checkKey(K k) {
        if (k == null) throw new InvalidKeyException("Invalid key: null.");
    }

    /** Hash function: applying MAD method to default hash code. */
    public int hashValue(K key) {
        return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
    }
```

```java


    /** Method: size()*/
    public int size() { return n; }
    /** Method: isEmpty() */
    public boolean isEmpty() { return (n == 0); }
    /** Returns an iterable object containing all of the keys. */
    public Iterable<K> keys() {
        PositionList<K> keys = new NodePositionList<K>();
        for (int i=0; i<capacity; i++)
            if ((bucket[i] != null) && (bucket[i] != AVAILABLE))
                keys.addLast(bucket[i].getKey());
        return keys;
    }

    /** findKey() - returns index of found key or -(a + 1),
    * a为第一个空位 */
    protected int findEntry(K key) throws InvalidKeyException {
        int avail = -1;
        checkKey(key);
        int i = hashValue(key);
        int j = i;
        do {
            Entry<K,V> e = bucket[i];
            if ( e == null) {
                if (avail < 0)
                    avail = i;  // key is not in table
                break;
            }
            if (key.equals(e.getKey())) // we have found our key
                return i;   // key found
            if (e == AVAILABLE) {   // bucket is deactivated
                if (avail < 0)
                    avail = i;  // remember that this slot is available
            }
            i = (i + 1) % capacity; // keep looking
        } while (i != j);
        return -(avail + 1);  // first empty or available slot
    }

    /** Returns the value associated with a key. */
    public V get (K key) throws InvalidKeyException {
        int i = findEntry(key);  // helper method for finding a key
        if (i < 0) return null;  // there is no value for this key, so reutrn null
        return bucket[i].getValue();     // return the found value in this case
    }


    /** method: Put  */
```

```
    public V put (K key, V value) throws InvalidKeyException {
        int i = findEntry(key); //find the appropriate spot for this entry
        if (i >= 0) //  this key has a previous value
            return ((HashEntry<K,V>) bucket[i]).setValue(value); // set new value
        if (n >= capacity/2) {
            rehash(); // rehash to keep the load factor <= 0.5
            i = findEntry(key); //find again the appropriate spot for this entry
        }
        bucket[-i-1] = new HashEntry<K,V>(key, value); // convert to proper index
        n++;
        return null;    // there was no previous value
    }

    /**method: rehash(), doubles the size of the hash table and rehashes all the entries. */
    protected void rehash() {
        capacity = 2*capacity;
        Entry<K,V>[] old = bucket;
        bucket = (Entry<K,V>[]) new Entry[capacity]; // new bucket is twice as big
        java.util.Random rand = new java.util.Random();
        scale = rand.nextInt(prime-1) + 1;      // new hash scaling factor
        shift = rand.nextInt(prime);        // new hash shifting factor
        for (int i=0; i<old.length; i++) {
            Entry<K,V> e = old[i];
            if ((e != null) && (e != AVAILABLE)) { // a valid entry
            int j = - 1 - findEntry(e.getKey());
            bucket[j] = e;
            }
        }
    }

    /** Method: remove() */
    public V remove (K key) throws InvalidKeyException {
        int i = findEntry(key);     // find this key first
        if (i < 0) return null;     // nothing to remove
        V toReturn = bucket[i].getValue();
        bucket[i] = AVAILABLE;      // mark this slot as deactivated
        n--;
        return toReturn;
    }

    /** Returns an iterable object containing all of the entries. */
    public Iterable<Entry<K,V>> entries() {
        PositionList<Entry<K,V>> entries = new NodePositionList<Entry<K,V>>();
        for (int i=0; i<capacity; i++)
            if ((bucket[i] != null) && (bucket[i] != AVAILABLE))
```

```
                        entries.addLast(bucket[i]);
                        return entries;
        }
        /** Returns an iterable object containing all of the values. */
        public Iterable<V> values() {
            PositionList<V> values = new NodePositionList<V>();
            for (int i=0; i<capacity; i++)
                if ((bucket[i] != null) && (bucket[i] != AVAILABLE))
                    values.addLast(bucket[i].getValue());
                    return values;
        }
}
```

9.3 Dictionaries

9.3.1 Dictionary ADT

- Dictionary : 和前面的map类似,也是可搜索键值对的集合
- 不同之处在于dictionary允许存在相同key的items
- Main operations : searching/inserting/deleting
- Applications : word-definition pairs/信用卡认证/DNS客户名
- Main methods:
    - find(k);
    - findAll(k);
    - insert(k,o);
    - remove(e) : 删键值对,这个和map的remove(k)不同(存在相同key)
    - entris() : 迭代器逐个返回键值对
    - size()/isEmpty();

9.3.2 List-Based Dictionary

- log file/audit trail : 基于无序序列的字典
- 性能:
    - insert , $O(1)$: 首尾插入;
    - find()/remove() , $O(n)$: 遍历无序序列;
- 基于无序序列的字典的只适合小级别或是插入为主的情况

```
Algorithm findAll(k)
Input: key k
Output: iterator of entries with key equal to k
{
    L=[empty list];
    B=D.entries();
    while(B.hasNext()){
```

```
        e=B.next();
        if(e.key()==k)
            L.insertLast(e);
    }
    return L.elements();
}

Algorithm insert(k,v)
Input: key k,value v
Output: add the entry(k,v) to D
{
    Create a new entry e=(k,v);
    S.insertLast(e);
    return e;
}//由于允许相同key,不需要检查是否已经存在

Algorithm remove(e)
Input:entry e;
Output:找到e就删除找不到返回null
{
    B=S.positons();
    while(B.hasNext()){
        p=B.next();
        if(p.element()==e){
            S.remove(p);
            return e;
        }
    }
    return null;
}
```

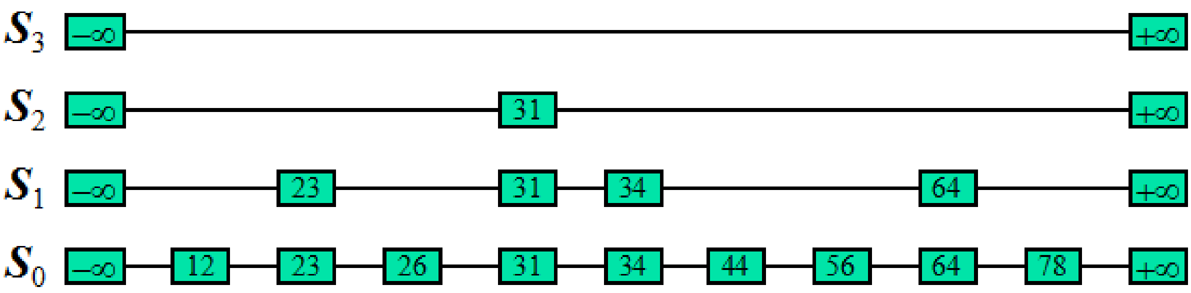
9.3.3 Hash-based dictionary

9.3.4 Search Table

- based on sorted sequence
  - sorted by key;
  - 使用external comparator来比较keys;
- 性能:
  - find , $O(logn)$: 二分查找;
  - insert , $O(n)$: 需要挪位置;
  - remove , $O(n)$: 需要挪位置;
- 这个同样适合小字典或是以插入为主的情况

9.4 Skip lists

- Skip list：含有不同数量键值对的层 $S_0, S_1, \ldots S_h$
  - 第一层S_0包含所有元素
  - 每层都是有序链表
  - 每个S_i包含special keys(sentinels) $+\infty, -\infty$
  - 每个list是上一个list的子序列：$S_h \in S_{h-1} \ldots \in S_1 \in S_0$，若某个元素出现再第i层,则所有比i小的层都有该元素
  - top list(最后一个序列) S_h只含两个特殊keys $+\infty, -\infty$;

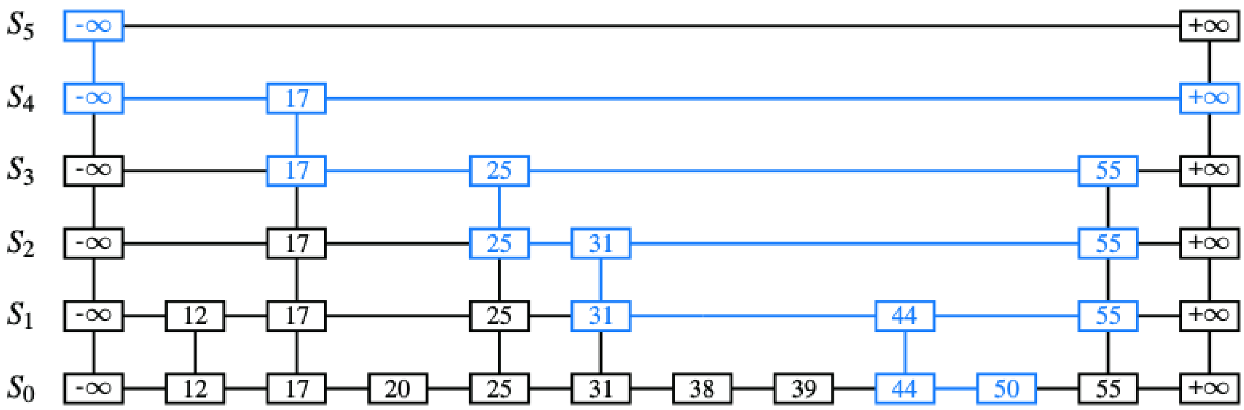


- 基本移动操作 $O(1)$：假定p为S_1中的31
  - next(p):与p相同level的下一个位置(S_1中的34)
  - prev(p):与p相同level的上一个位置(S_1中的23)
  - above(p):上一个level的相同位置(S_2中的31)
  - below(p):下一个level的相同位置(S_0中的31)

9.4.1 Search

```
Algorithm SkipSearch(k):
Input: given key k;
Output:the position of entry(k,p),找不到返回小于k的最大key所在位置;
{
    p=MINUS_INF;//起始点,最上层最左侧-∞
    while(below(p)!= null){
        currentKeyNext=next(p).key();
        if(k==currentKeyNext){
            return next(p);
        }
        else if(k>currentKeyNext){//在同行,还有继续前进的空间
            p=next(p);
        }
        else p=below(p);//说明currentKeyNext过大了,需要下降
    }
    return p;//没找到->小于k的最大key所在位置
}
```

- 举个栗子：搜索50



9.4.2 Randomized Algorithims

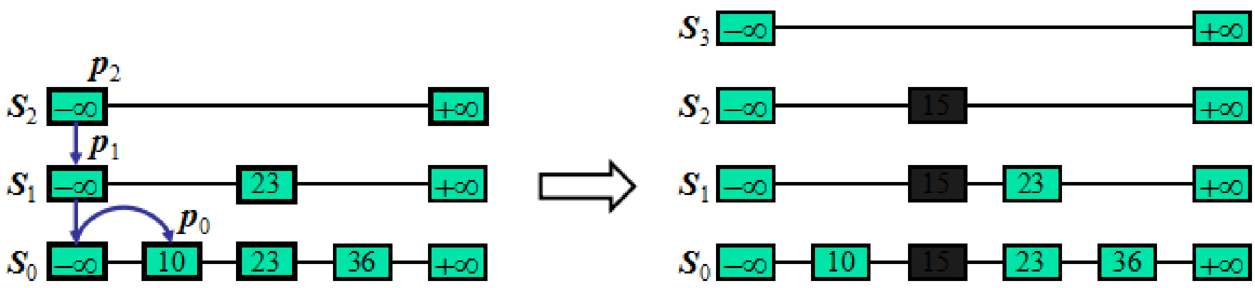
- 在算法中使用了随机函数，且随机函数的返回值直接或者间接的影响了算法的执行流程或执行结果
- 包含以下语句

```
b=random();//[0,1]
if(b==0) do A;
else do B;
```

- 平均运行时间基于以下假定:
  - 每次随机的几率相同且相互独立
- 最坏情况运行时间非常大,但发生几率很小

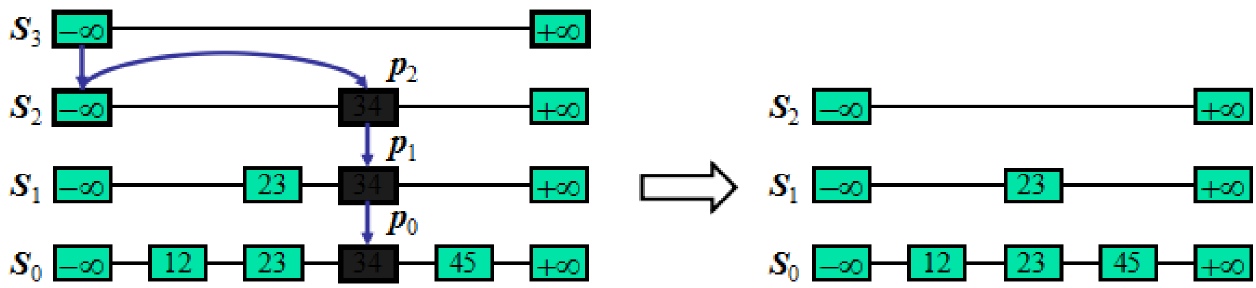
9.4.3 Insertion

- 插入操作需要用到随机化算法决定插入最后插入的层
  - 类似抛硬币 ,重复抛直到得到背面 ,在这期间得到正面的次数为i
  - S_i为新元素被插入的最上层level
  - 确定S_i:
    - if(i>h) 增加一些只含$+\infty, -\infty$的空序列 $S_{h+1}, ... S_i$;
    - else 排除多余的list $S_{i+1}, ... S_h$ ，最后剩下$S_0, S_1, ... S_i$;
  - 使用SkipSearch(k)查询是否已经存在entry:
    - 如果已经存在,return p并将value替换为v;
    - 如果不存在,返回的是小于k的最大key所在位置 p0,p1,...pi;
  - 将新entry(k,v)插入到p0,p1,...pi后面
- 举个栗子 : i=2,插入15



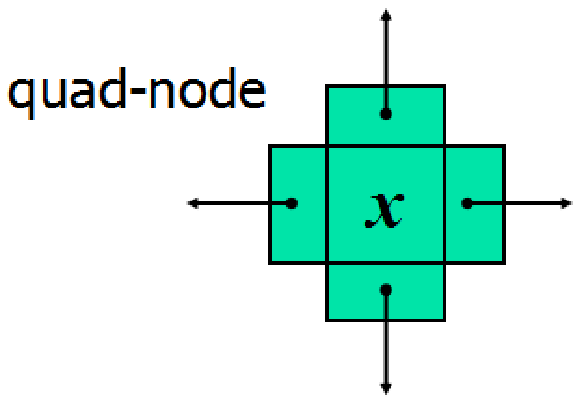
9.4.4 Deletion

- SkipSearch(k),从最高层开始寻找删除点p0,p1,...pi;
- 从每个list Sj中删除相应位置pj;
- 若删除点位于最大层, 需要对层数进行更新(多个只有最大最小边界的层最终只保留一个)
- 举个栗子: 删除34



9.4.5 使用quad-node实现skip list

- quad-node:带四个链接的结点!
    - entry;
    - link to prev node;
    - link to next node;
    - link to node below;
    - link to node above;



- 另外定义两个special keys: PLUS_INF/MINUS_INF

9.4.6 Space Usage

- Skip list的空间复杂度取决于每次调用插入时的random bits
  - Fact 1 : 连续i次都是1的概率为 $1/2^i$;
  - Fact 2 : n个键值对中每个出现在set中的概率为p , 该set的平均size为 np;
- 对于n个键值对的skip list:
  - 对于Fact 1 : 向S_i插入一个键值对的概率为$1/2^i$;
  - 对于Fact 2: list S_i的平均size为 $n/2^i$;
- Skip list使用的node数量平均为:

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

- 结论:平均空间复杂度 $O(n)$

9.4.7 Height

- Height决定了插入操作的时间复杂度
- 跳表的高度为 $O(logn)$
- Fact 3 : 每个event发生概率p, 则至少一个event发生的概率最多为np;
- 对于n个键值对的skip list:
  - 对于Fact 1:向S_i插入一个键值对的概率为$1/2^i$;
  - 对于Fact 3:list S_i至少一个item的概率不大于$n/2^i$;
- 高度至多为$3logn$的概率至少为$1 - 1/n^2$;

9.4.8 Search and Update times

- search时间正比于:上移下移的次数
  - drop-down 次数;
  - scan-forward 次数;
- drop-down与高度有关 $O(logn)$
- Fact 4: 平均需要投两次硬币才能得到背面;
  - 根据Fact 4 , scan-forward的平均次数也是2次;
- 因此Scan-forward的期望次数-->O(log n)
- 结论:搜索操作的时间复杂度O(log n),插入和删除操作差不多

9.4.9 Summary

- Skip list是一种基于随机插入算法的数据结构,可用于实现字典;

- 平均空间复杂度$O(n)$;
- size()/isEmpty():$O(1)$;
- search()/insert()/remove():$O(logn)$;
- entries()/keys()/values():$O(n)$;

Method	Running Time
size, isEmpty	$O(1)$
get	$O(\log n)$ expected
put	$O(\log n)$ expected
remove	$O(\log n)$ expected
firstEntry, lastEntry	$O(1)$
ceilingEntry, floorEntry lowerEntry, higherEntry	$O(\log n)$ expected
subMap	$O(s+\log n)$ expected, with s entries reported
entrySet, keySet, values	$O(n)$

Week 10 Text Processing

- Pattern Matching
- Tries
- Greedy Method and Text Compression
- Dynamic Programming

10.1 Pattern Matching

- Brute-Force : 从P头比较,找不到严格后移一位 , $O(nm)$
- Boyer-Moore : 从P尾比较+Last-occurrence function , $O(nm+s)$
- Knuth-Morris-Pratt : 从P头比较,出现冲突时可以跳过一些已经比较过的位置;

10.1.1 Brute-Force Pattern Matching

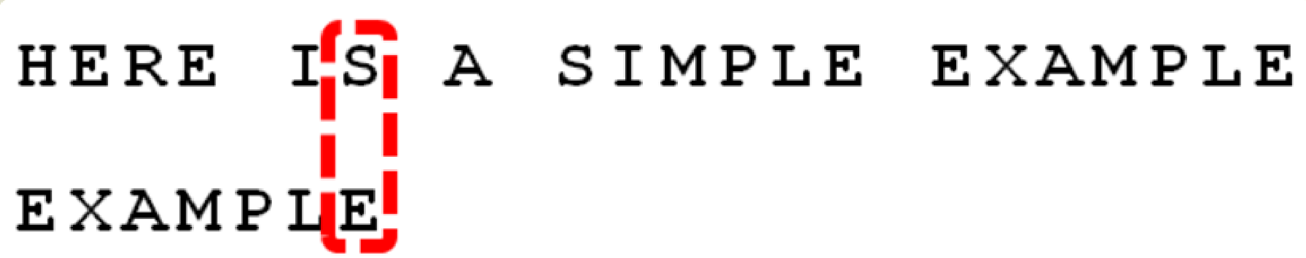
```
Algorithm BruteForceMatch(T,P)
Input: Text T(n),Pattern P(m)
Output: 如果找到了match P的部分,标出在T中的位置,否则返回-1
{
    for(i=0;i<n-m+1;i++){
        j=0;
        while(j<m ∩ T[i+j]==P[j])
```

```
                j=j+1;
            if(j==m) //match->给出output,结束循环
                return i;
        }
        return -1;
    }
```

- 算法分析:
    - 从头开始比较, 若T的某个位置T[i]与P[0]匹配
        - 比较T[i+1]与P1是否匹配;
        - 比较T[i+j]与P[j]是否匹配;
        - 若T[i+m]与P[m]匹配, 完全匹配, 返回位置i;
        - 中间如果有一点没匹配都要跳出j循环,P向前移动一格,开始i循环
    - 时间复杂度 $O(nm)$
    - wosrt cast：结尾才匹配(e.g. T=aaaaaah,P=aah)或完全不匹配
    - 没完全match前一格一格往前推

10.1.2 Boyer-Moore Heuristics

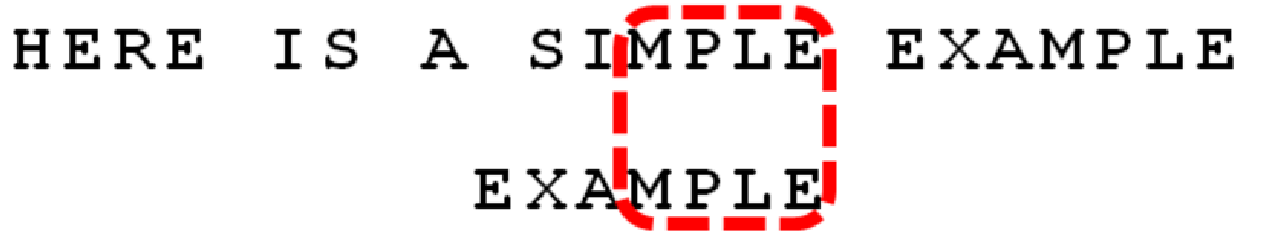
- 网上的一个栗子，和本课程有一定差异，目前以本课程为主



- Last-Occurance：E6,X1,A2,M3,P4,L5
- P[6]!=T[6],i=i+m=13：后移7位

- P[6]!=T[13],L[P]=4,i=i+m-min(j,l+1)=15：后移2位

- P[6]==T[15]匹配：不断向前匹配直到P2发生不匹配



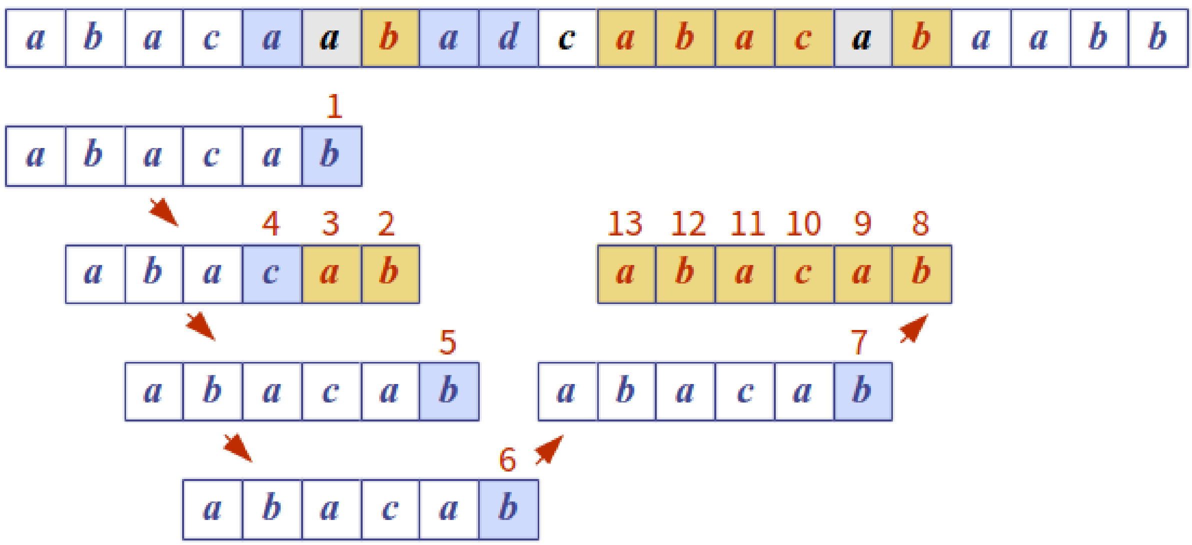
- P[2]与T[11]不匹配,L[T[11]]=-1,min(j,l+1)=0：移动7位
- P[6]!=T[-2],L[T[-2]]=5,min(j,l+1)=6：移动1位

- Boyer-Moore算法基于Last-Occurance Function
  - 该函数返回字符集中每个字符在P中最后出现的位置
  - 栗子:
    - 字符集S={a,b,c,d}
    - P=abacab
    - L(a)=4,L(b)=5,L(c)=3,L(d)=-1
  - 该函数操作时间取决于size(P)=m,以及size(S)=s：$O(m+s)$
- Boyer-Moore Algorithm
  - 核心移位函数为i=i+m-min(j,l+1)
  - 每次循环从P[m-1]开始比较,如果最后一位匹配就倒数第二,倒数第三...
  - 一旦不匹配发生,看不匹配字符T[i]在P中最后出现的位置;L(T[i])
  - 结尾就不匹配l+1=0-->直接后移m位
  - 如果j后移(m-j)
  - j>l+1(不匹配字符未曾出现或是出现在j前面)->后移(m-l-1)
  - 只要需要移位,都要把比较位还原成m-1 重新比较;

```
Algorithm BoyerMooreMatch(T,P,S)
Input:Text T(n),Pattern P(m),字符集S
```

```
Output:T中与P相匹配的位置
{
    L=lastOccurenceFunction(P,S);

    //初始比较位置 : P的最后一位P[m-1]
    i=m-1;
    j=m-1;

    while(i<=n-1){
        if(T[i]==P[j]){ //发生匹配
            if(j == 0){ //完全匹配
                return i;
            }
            else{ //尚未完全匹配 , 看前一位是否匹配
                i = i - 1;
                j = j - 1;
                }
            }

        else{  //发生不匹配 , 进行移动
            l = L[T[i]]; //在P中查询最后出现不匹配字符T[i]的位置
            i = i+m-min(j,1 + l); //算法核心 : 移位公式
            j = m - 1; //同时j回归初始位置重新开始比较
        }
    }
    return -1;  //  no match
}
```

- 举个栗子



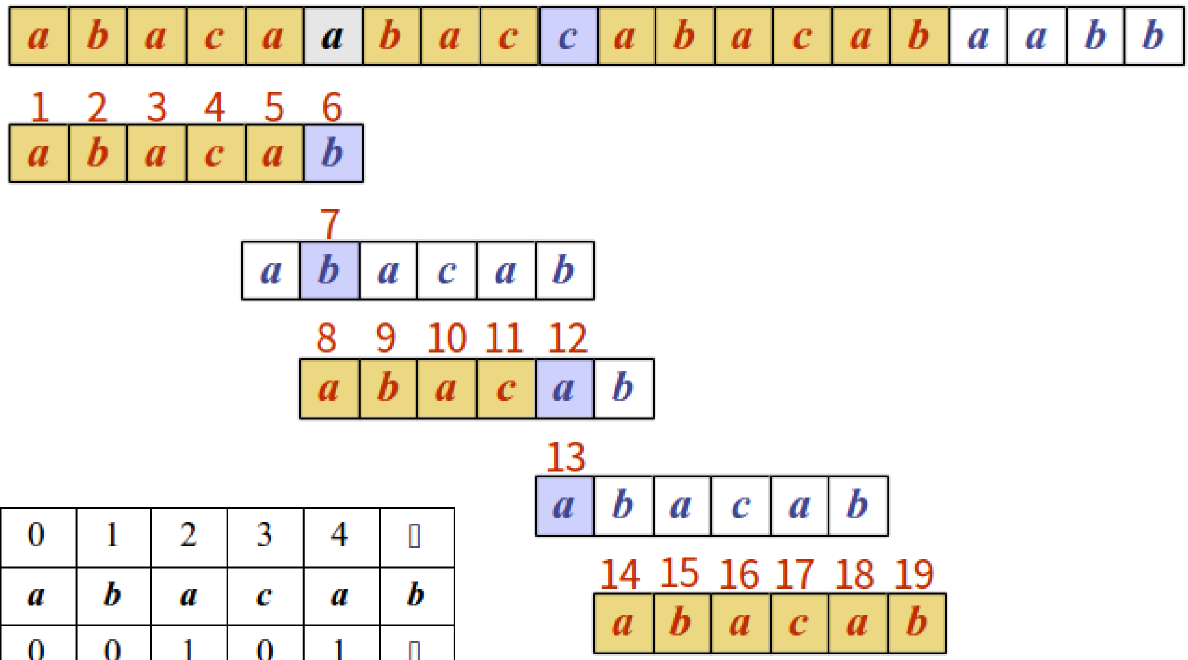
- 该算法时间复杂度$O(nm+s)$, 在英语文本处理上要远强于暴力查找;

10.1.3 Knuth-Morris-Pratt Algorithim

- 算法流程:

    - 开始和暴力查找类似,从左至右匹配:
        - 第一个不匹配：整个P都向后移动一位继续进行匹配
        - 第一个匹配：看下一个是否匹配...
    - 发生不匹配:
        - 跳过前面已经比较过的部分避免一些多余的比较
- 基本思路:
    - 先求出Pattern的部分匹配值F();
    - 发生冲突时：对应匹配值F(j-1)
    - 移动位数 = 已匹配的字符数(当前位置j)-对应的部分匹配值F(j-1)
- 使用KMP Failure Function求取部分匹配值
    - 前缀：除了最后一个字符外的全部头部组合
    - 后缀：除了第一个字符外的全部尾部组合
    - F(j)：前缀和后缀共有的最长组合的长度
- 举个栗子说明failure function：

```
j :   0  1  2  3  4  5
P[j]: a  b  a  a  b  a
F(j): 0  0  1  1  2  3

a:前缀[],后缀[]-->F[0]=0
ab:前缀[a],后缀[b]-->F[1]=0
aba:前缀[a,ab],后缀[a,ba]-->F[2]=1
abaa:前缀[a,ab,aba],后缀[a,aa,baa]-->F[3]=1
abaab:前缀[a,ab,aba,abaa],后缀[b,ab,aab,baab]-->F[4]=2
abaaba:前缀[a,ab,aba,abaa,abaab],后缀[a,ba,aba,aaba,baaba]-->F[5]=3

T: abaab[x]
P: abaab[a]
在j=5 处不匹配-->F[4]=2
已经匹配的字符数5
因此:前移 5-2=3位
abaabx
   abaaba
```



j	0	1	2	3	4	▯
***P*[*j*]**	***a***	***b***	***a***	***c***	***a***	***b***
***F*(*j*)**	0	0	1	0	1	▯

- KMP Algorithm

```
Algorithm KMPMatch(T,P)
{
    F=failureFunction(P);//得到部分匹配表
    //初始化比较位
    i=0;
    j=0;
    while(i<n){
        if(T[i]==P[j]){
            if(j==m-1)
                return i-j;//完全匹配
            else{//第一位匹配,比较第二位,第三位,...
                i++;
                j++;
            }
        }
        else{
            if(j>0) j=F(j-1);//部分不匹配,调用F(j-1)
            //j=j-(j-F(j-1))=F(j-1)相当于后移j-F(j-1)位
            else i++;//j==0-->开头不匹配,直接后移位
        }
        return -1;
    }
}
```

- KMP算法分析:

- Failure function : $O(m)$
- while-loop : 不大于2n次迭代
- KMP的运行时间$O(m+n)$, 优于暴力查找 $O(mn)$

- Failure Function

```
Algorithm failureFunction(P)
Input:the pattern P with m elements
Output: F=failureFunction(P)
{
    F[0]=0;//前缀[],后缀[]
    //多于一个元素的情况
    i=1;//后缀
    j=0;//前缀
    while(i<m){
        if(P[i]==P[j]){//前头==后尾
            F[i]=j+1;
            i++;
            j++;
        }
        else if(j>0){
            j=F[j-1]
        }
        else{//j==0且ij元素不等
            F[i]=0;
            i++;
        }
    }
}
```

- Failure function算法分析:
  - j=0 且头尾不等的话,那么该位置F一定为0;
  - 若头尾出现了匹配,j++;
- 举个栗子 : abacabb
  - i=4,j=0-->出现匹配-->i++,j++,F4=1
  - i=5,j=1-->出现匹配-->i++,j++,F5=2
  - i=6,j=2-->不匹配,且j>0-->j=F1=0-->头尾不匹配-->F6=0,i++

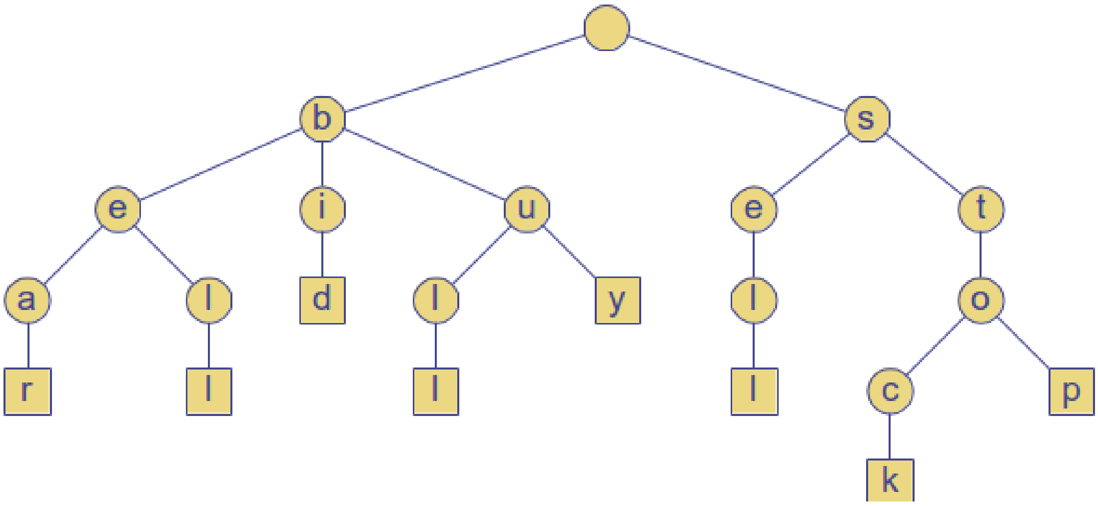
10.2 Tries

10.2.1 Preprocessing Strings

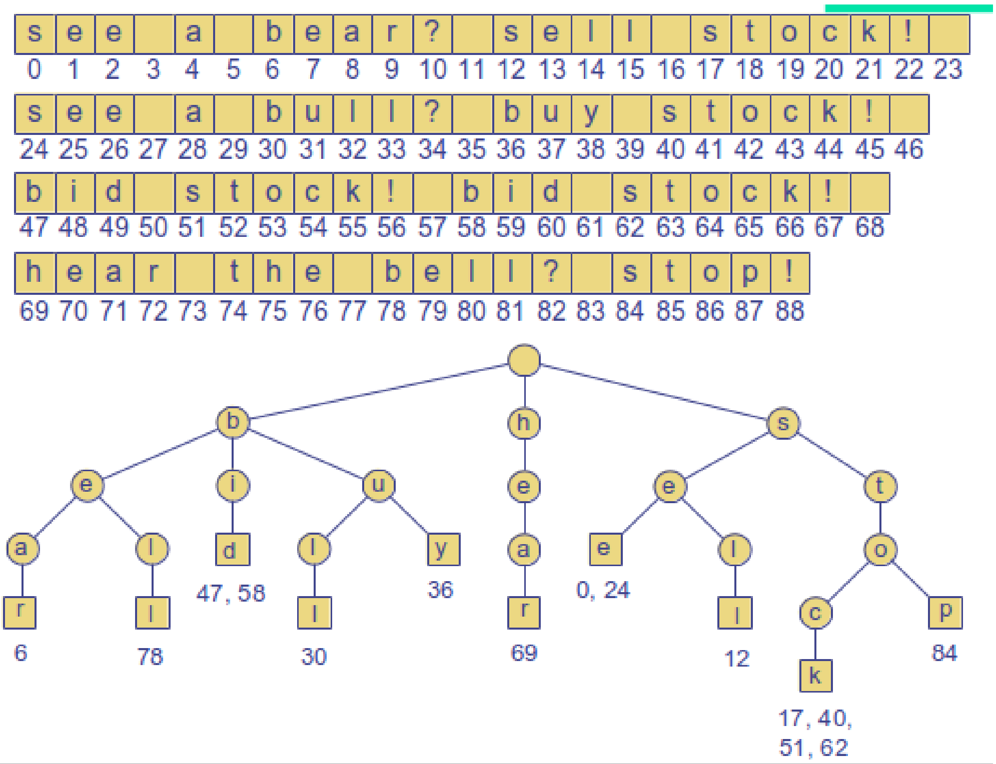
- 字典树是一种文本前处理数据结构 , 前处理可提升pattern matching的效率 $O(m)$

10.2.2 Standard Tries

- 基本构造:
  - 除了root外每个node都配有一个字母标签, root 代表空格;
  - 结点的几棵子树按字母表顺序排序，root的zi子结点为单词首字母;
  - 从external到root构成S中一个字符串
  - 每个leaf还储存了这条路径代表单词出现的位置(首字母)!

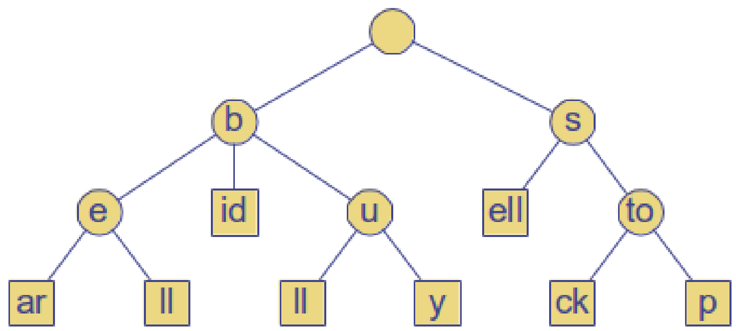


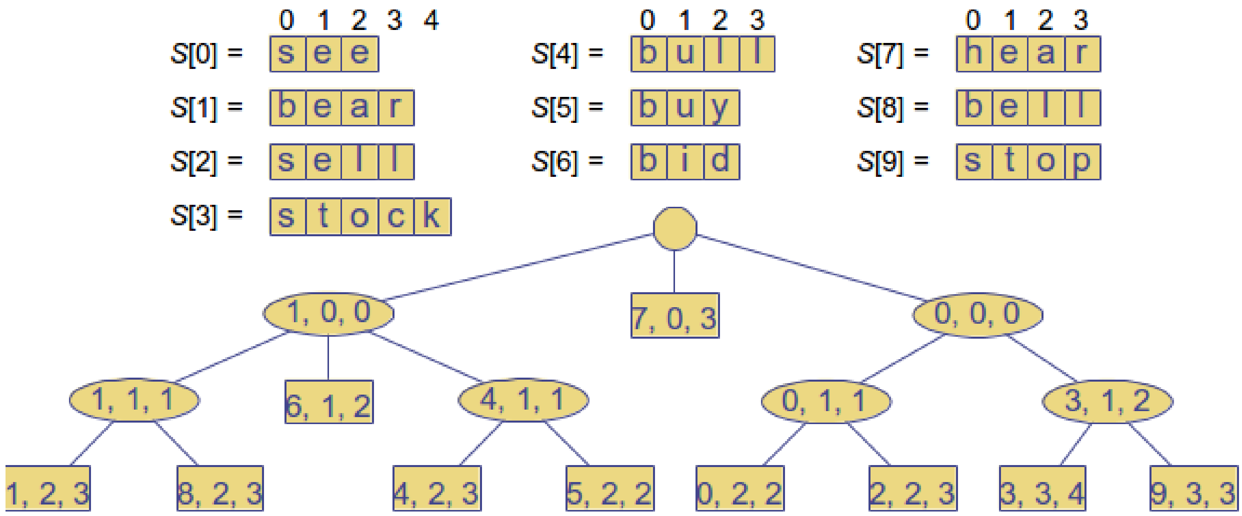
- 分析字典树:
  - n : S中字符串的数量;
  - m : 每次操作动用字符串参数的数量;
  - d : 字母表的size;
  - 空间复杂度 : $O(n)$
  - 搜索/插入/删除操作 : $O(dm)$
- word matching with a trie
  - 将文本插入trie;
  - 每次操作始于搜索:先查询首字母,然后一步步查询单词;
  - 到达叶结点还能给出这个单词的位置;



10.2.3 Compressed Tries

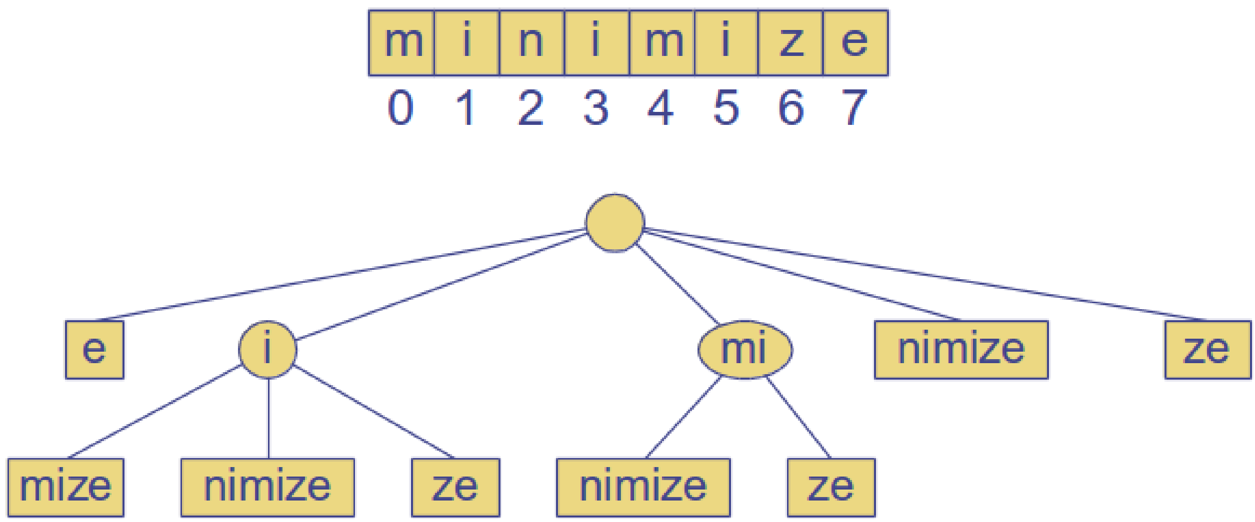
- 压缩多余结点;
- 保证每个internal都有至少两个子结点;
- 若某个internal只有一个子结点, 进行压缩;
- 空间复杂度 $O(s)$,其中S为array中的字符串数量



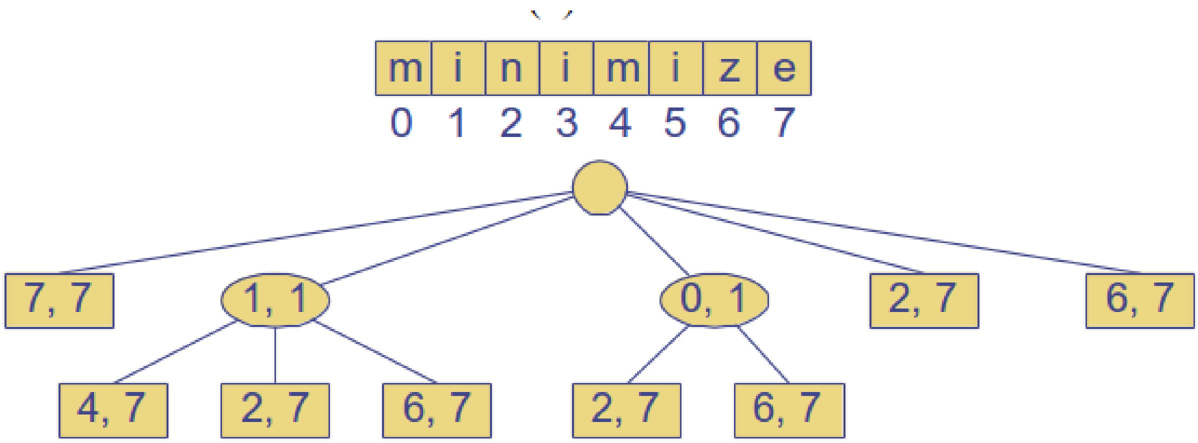


10.2.4 Suffix Trie后缀树

- 使用compressed trie表示某个字符串X的全部后缀



- 坐标：从头到尾,例如"nimize"=[2,7]



- 使用后缀字典树进行Pattern Matching:

```
Algorithm suffixTrieMatch
Input: Compact suffix trie T for a text X and pattern P;
Output: X中完全匹配的起始index,或者"P不是子字符串";
{
    p=P.length;
    j=0;
    v=T.root();
    while(f==true || T.isExternal(v)){
        f=true;
        for (each child w of v){
            //已经匹配了j+1个字符
            i=start(w); //w的起始index(首字母)
            if(P[j]==T[i]){//对子树w进行处理
                x=end(w)-i+1;//end(w)->end index of w
                if(p<=x){//后缀<=Node label
                if(P[j:j+p-1]==X[i:i+p-1]) return i-j;
                    else return "P不是X的子字符串";
                }
                else{
                    if(P[j:j+x-1]==X[i:i+x-1]){
                        p=p-x;//更新后缀长度
                        j=j+x;//更新后缀起始index
                        v=w;
                        f=false;
                        break for循环;
                    }
                }
            }
        }
    }
    return "P不是X的子字符串";
}
```

- 该算法基于以下假定:
    - 如果node v存在标签(i,j),且Y为从root到v路径上长度为y的字符串,则X[j-y+1,...j]=Y;
- 对Suffix Tries的分析:
    - n:字符串X的size;
    - d:alphabet的size;
    - m:Pattern的size;
    - 空间复杂度 $O(n)$;
    - 字符匹配操作 $O(dm)$;
    - 构建字典树 $O(n)$;

10.3 Greedy Method and Text Compression

10.3.1 the greedy method technique

- configurations: different choices/collections/values to find;
- objective function : 分配给前面的结构,取决于非贪婪还是贪婪;
- 举个栗子 : 总金额一致,如何尽可能让买到的商品总价值更高;

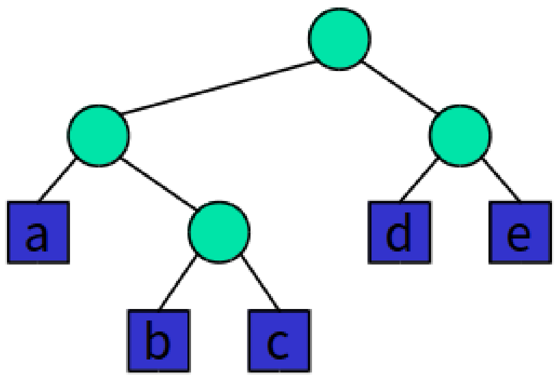
10.3.2 Text compression

- 将给定字符串X编码为更小的字符串Y , 节省memory, bandwidth
- Huffman encoding
  - 计算每个字母c的出现频率f(c);
  - 高频字母 , short code word;
  - 每个code word都不能是另一个code的前缀;
  - 使用optimal encoding tree来确定code words;

10.3.3 Encoding tree

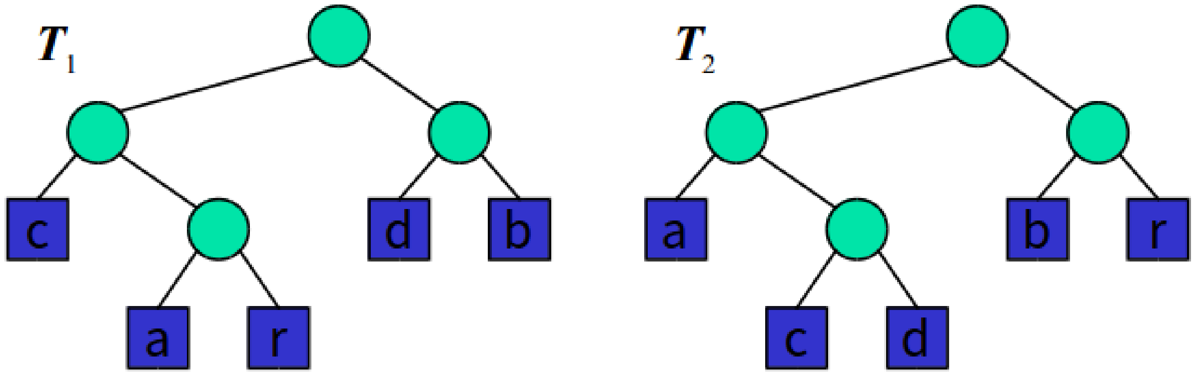
- code:将字母表内每个字母都转化为二进制编码;
- prefix code: code的一种,没有code-word是另一个code-word的前缀;
- encoding tree: 用于展示prefix code
  - 每个external储存一个字母;
  - 每个字母的code word取决于从root到叶结点的路径;
  - 0为左子树,1为右子树
  - 例如从ｒｏｏｔd某个字母的路径是左-右-左,则该字母的codeword为"010"

00	010	011	10	11
a	b	c	d	e



- optimal encoding tree:让字符串X的code尽可能短
  - 高频字母的code-words很短;
  - 低频字母code-words较长;
  - 栗子 : T2优于T1;

- X = abracadabra
- T_1 encodes X into 29 bits
- T_2 encodes X into 24 bits



10.3.4 Huffman Algorithm

- 构建optimal encoding trie : 高频浅低频深

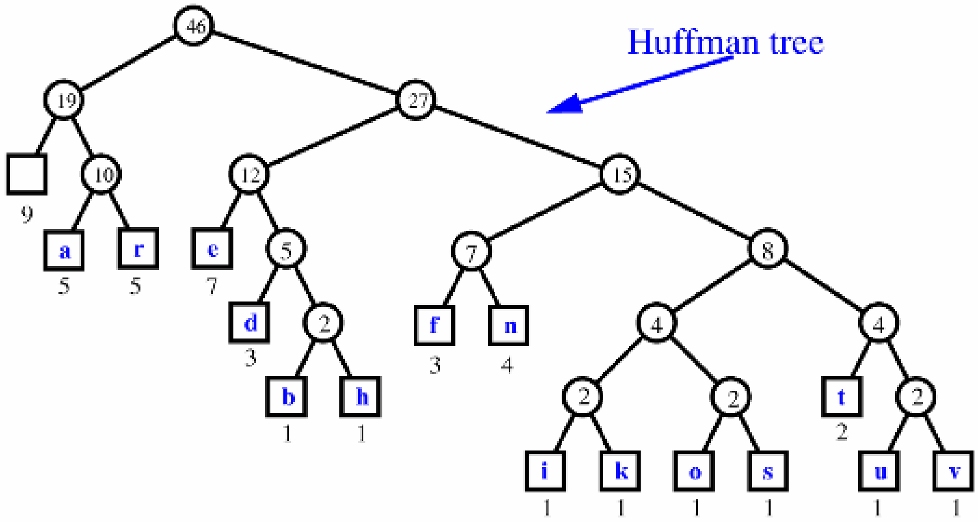
```
Algorithim HuffmanEncoding(X)
Input: String X of size n;
Output: optimal encoding trie for X;//最小化编码后的size
{
    C=distinctCharacters(X);//构建字母表
    computeFrequencies(C,X);//计算字母表中每个元素的频率
    Q=new empty heap;
    for all c in C{
        T=new single-node tree storing c;
        Q.insert(getFrequency(c).T);
    }//将每个元素及其频率以键值对形式插入Heap
    while (Q.size()>1){//构建heap
        f1=Q.minKey();
        T1=Q.removeMin();
        f2=Q.minKey();
        T2=Q.removeMin();
        //每次移出并merge两个最小的entry,其key和为其父结点的key
        T=join(T1,T2);
        Q.insert(f1+f2,T);//合并后再插入Q中
    }
    return Q.removeMin();
}
```

- 算法分析:
  - 运行时间 $O(n + dlogd)$,d为字母表size;
  - 基于heap PQ;
  - 该算法类似从底层组建最大heap的过程:
    - 首先两个低频作为子树联结到一起,父结点的key就是子结点的key之和

- 最后得到的root就是X的size;

String: **a fast runner need never be afraid of the dark**

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



10.3.5 Fractional Knapsack Problem

- 本课程不考察!
- Given: Set S with n items,每个item i包含
  - b_i : positive benefit
  - w_i : positive weight
- Goal: 在weight <= W的前提下尽可能最大化benefit
- fractional knapsack problem:
  - x_i : 取走item i的比例，比如某个item只取一半
  - 目标：maximize $\sum_{i\in S} b_i(x_i/w_i)$
  - Constraint: $\sum_{i\in S} x_i \leq W$

```
Algorithm fractionalKnapsack(S,W)
Input:set S(benefit bi,weight wi), max weight W;
Output:返回得到最优化的xi;
{
    for i in S{//初始化xi,value
        xi=0;
        vi=bi/wi;//value
    }
    w=0;//total weight
    while(w<W){
        remove item i with the largest vi;
        xi=min(wi,W-w);
```

```
            w=w+xi;
        }
    }
```

- 算法分析: 运行时间$O(nlogn)$

10.3.6 Task Scheduling

- 本课程不考察
- Given: set T with n tasks,每个task包含
  - start time : si
  - finish time : $f_i(s_i < f_i)$
- Goal:使用最少的人力完成所有任务 (尽可能排满每个人的时间表)
- Greedy choice:根据起始时间分配任务,尽可能少人力

```
Algorithm taskSchedule(T)
Input:Set T of tasks(start time si,end time fi);
Output:在时间表不冲突的情况下,动用最少机器完成所有任务;
{
    m=0;//初始化需要动用的机器数
    while(!T.isEmpty()){
        remove si最小的task i;//早开始的优先度高
        if (已有的某台机器j可以执行i){
            将schadule i添加给machine j;
        }
        else {
            m=m+1;
            将schedule i添加给新机器 m;
        }
    }
}
```

- 算法分析:
  - 每次取出最先开始的未分配任务;
  - 检查这项任务能不能分配给已有任务的员工;
  - 如果没有员工能接这项任务,调一个新员工;
  - 复杂度$O(nlogn)$

10.4 Dynamic Programming

10.4.1 Matrix Chain-Products

- 本课程不考察
- 矩阵相乘 : $C_{ab} = A_{ac} * B_{cb}$,时间复杂度$O(abc)$

10.4.2 Greedy Approach

- Idea 1: 每次尽可能选择操作最多的
    - $A = 10 * 5; B = 5 * 10; C = 10 * 5; D = 5 * 10$
    - 操作数$(A * B) * (C * D) = 2000$ 不如 $A * ((B * C) * D) = 1000$
- Idea 2: 每次尽可能选择操作最少的
    - $A = 101 * 11; B = 11 * 9; C = 9 * 100; D = 100 * 99$
    - $A * ((B * C) * D) = 228789$ 不如 $(A * B) * (C * D)$
- Greedy approach并不能优化计算量

10.4.3 Recursive Approach

- 将问题拆分为subproblems:
    - 找到最佳的组合$A_i * Ai + 1 ... * A_j$;
    - 使用N_{ij}表示当前subproblem的操作数;
    - 整个问题的总操作数为N_{0,n-1};
- Subproblem optimality:
    - final multiply is at index i: $(A_0 * ... * A_i) * (A_{i+1} * ... * A_{n-1})$;
    - 最优解$N_{0,n-1} = N_{0,i} + N_{i+1,n-1} + time$
    - time 为上一次递归的计算时间

10.4.4 Characterizing Equation

- $A_i = d_i * d_{i+1}$ matrix;
- $N_{i,j}$的normal equation:

$$N_{i,j} = \min_{i \le k < j} (N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1})$$

- 子问题并非独立的,存在覆盖

10.4.5 Dynamic Programming Algorithm

- 子问题存在覆盖, 不能使用递归 : 从下向上构建优化的子问题

```
Algorithm matrixChain(S)
Input:Sequence S of n matrices to be multiplied;
Output:number of operations
{
    for(i=1;i<n;i++){ //初始化所有的Nii=0
        Nii=0;

    for(b=1;b<n;b++){
        for(i=0;i<n-b;i++){
            j=i+b;
            Nij=+infinity;
```

```
                for(k=i;k<j;k++){
                    Nij=min{Nij,Nik+N(k+1,j)+did(k+1)d(j+1)};
                }
            }
        }
    }
```

- 算法分析:
  - 先从最小的$N_{0,0}$开始,然后然后逐步计算到大矩阵；
  - $N_{i,j}$的值源于之前i行j列的$entries;$ +核心代码为$normalequation;$ +时间复杂度：O(n^3)$，至少强于指数级；

10.4.6 General dynamic programming technique

- 适用条件:
  - Simple subproblems;
  - Subproblem optimality: 全局优化值可以被划分为优化的子问题;
  - Subproblem overlap: 子问题不互相独立, 从底向上构建;

10.4.7 Subsequences:

- 格式为$x_{i_1}x_{i_2}\dots x_{i_k},(i_j < i_{j+1})$的字符串;
- 和substring的主要区别在于子序列是有序的
- 举个栗子说明区别:
  - 'DFGHK' is subsequence
  - 'DAGH' is not subsequence

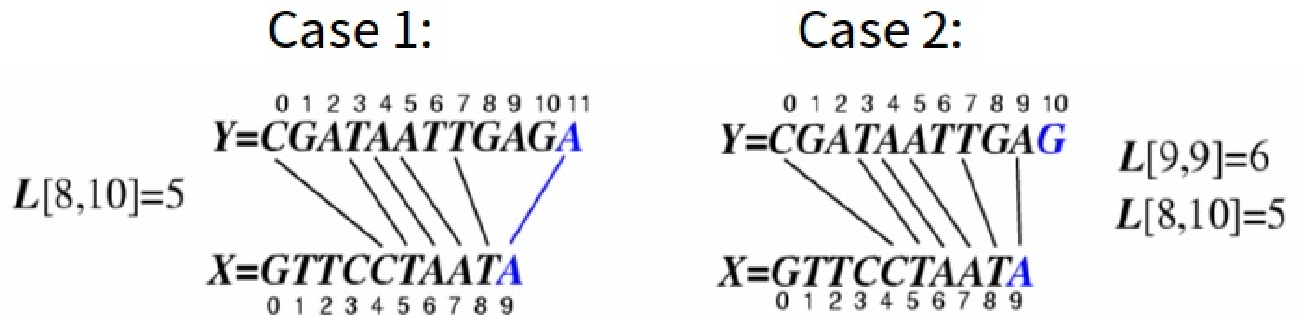
10.4.8 Longest Commom Subsequence(LCS) Problem

- 最长共有子序列

  - Given : strings X,strings Y;
  - Goal : 找到XY共有的最长的子序列;
  - 栗子: X=ABCDEFG , Y=XZACKDFWGH , LCS=ACDFG
- 暴力查找LCS

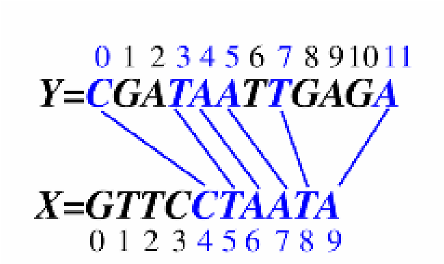
  - 查找X所有的子序列，then查找其中也是Y子序列的，finally pick the longest one;
  - 复杂度 $O(2^n)$
- 动态规划解决LCS问题

  - L[i,j]为X[0..i]，Y[0..j]的LCS;
  - L[-1,k] = L[k,-1]=0 (空集不存在LCS);
  - if xi==yj (match)，then L[i,j]=L[i-1,j-1]+1;
  - else (not match)，L[i,j]=max{L[i-1,j],L[i,j-1]};



- LCS Algorithm $O(nm)$

```
Algorithm LCS(X,Y)
Input: Strings X(n),Y(m);
Output: LCS L[i,j] of X,Y;
{
    //先初始化L=0
    for(i=0;i<n;i++) L[i,-1]=0;
    for(j=0;j<m;j++) L[-1,j]=0;

    //开始匹配
    for(i=0;i<n;i++){
        for(j=0;j<m;j++){
            if (x0==y0) L[i,j]=1;//开头就匹配->L[0,0]=1
            else if (xi==yj) L[i,j]=L[i-1,j-1]+1;
            else L[i,j]=max{L[i-1,j],L[i,j-1]};
        }
    }
    return array L;
}
```



Week 11 Graphs I

- Undirected Graphs and Directed Graphs;
- Depth-First Search;
- Breadth-First Search;
- Transitive Closure;
- Topological Sorting;

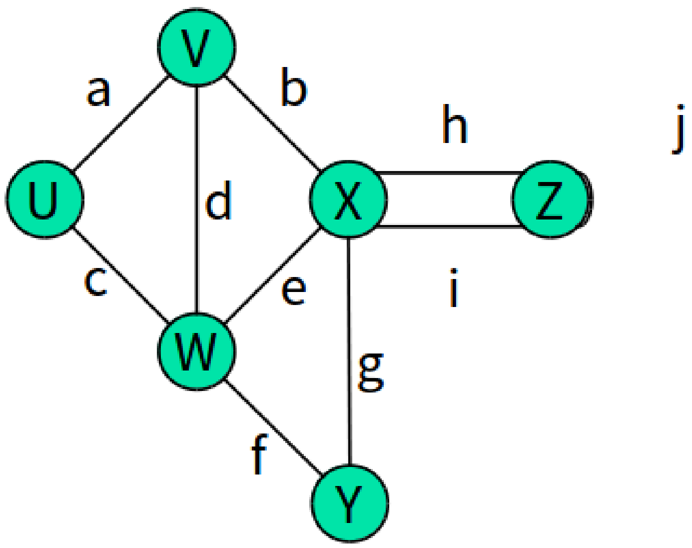
11.1 Graphs

- 定义:graph=pair(V,E)
  - V(vertices) : a set of nodes;
  - E(edges) : a collection of pairs of vertices;
- 端点和边都是store element的positions

11.1.1 Edge types

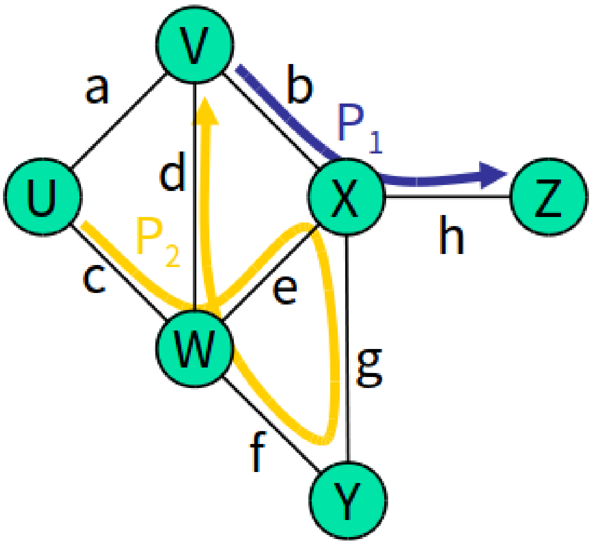
- Directed edge:
  - 有序顶点对(u,v);
  - 始于u,止于v,逆过程不成立;
  - 栗子 : flight;
- Undirected edge:
  - 无序顶点对(u,v),uv可互相访问;
  - 栗子 : flight route;
- Directed graph:
  - 所有的边都是directed edges;
  - 栗子 : route network;
- Undirected graph:
  - 所有的边都是undirected edges;
  - 栗子 : flight network;

11.1.2 Terminology

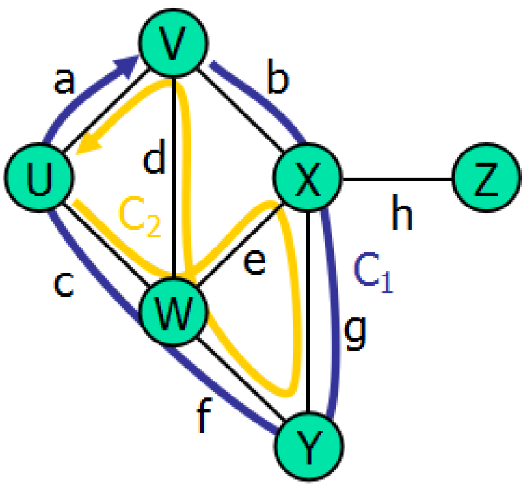


- End vertices(end points) of an edge : 某条边的两个端点;
  - endVertices(a) = (U,V)
- Adjacent vertices : 与某个端点间隔一条边的端点;

- opposite(U,a) = V
- Edge incident on a vertex : 与某个端点连接的所有边;
    - incidentEdges(V) = (a,b,d)
- Degrees of a vertex : 某个端点连接边的数量;
    - degree(V) = 3
- Parallel edges : 起止端点相同的两条边;
    - h , i
- Self-loop : 起止端点是同一个点的某条边;
    - j



- Path : 从某个端点到另一个端点的路径;
    - U - W - Y - X - W - V
- Simple path : 经过的端点和边不重复;
    - V - X - Z

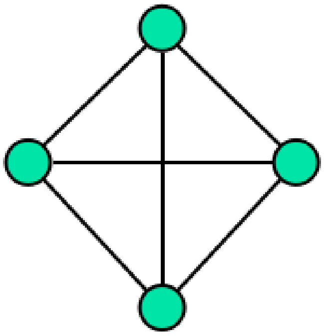


- Cycle : 从一个端点出发绕个圈返回这个端点;
    - 同上面path的栗子
- Simple cycle : 经过的端点和边不重复;
    - V - X - Y - W - U - V

11.1.3 Properties

- Given:

    - n : number of vertices;
    - m : number of edges;
    - deg(v) : degree of vertex v;
- $\sum_v deg(v) = 2m$ (每条边两端点，计算两次)
- 对于不带自循环以及多边的无向图:
    - $m \leq n(n-1)/2$;
    - $dev(v)i) \leq n-1$;



Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

11.1.4 Graph ADT

- Accessor methods:

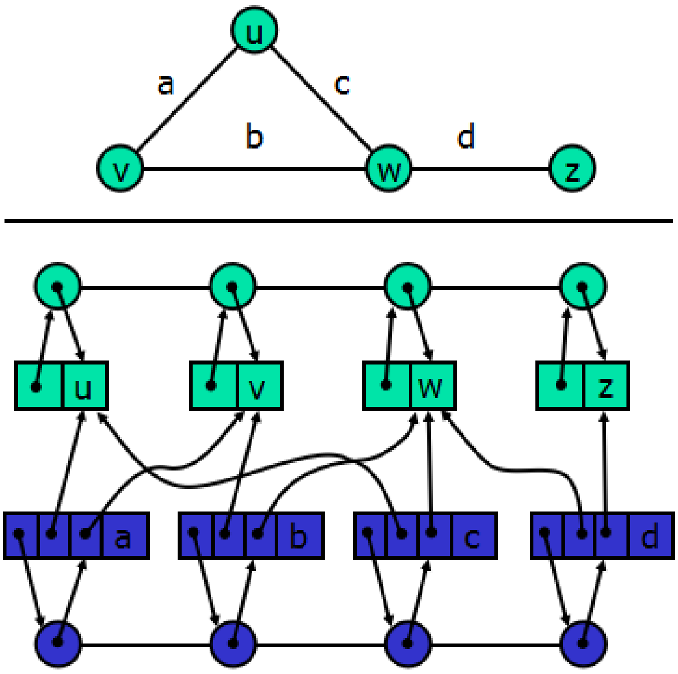
    - Vertex endVertices(Edge e) : 给定edge e,返回两个vertices;
    - Vertex opposite(Vervex v, Edge e) : 给定edge e和其中一个端点,返回另一个 vertex;
    - boolean areAdjacent(Vertex v, Vertex w) : 判断是否是一条边的两端点;
    - void replace(Vervex v, Element x) : 将vertex v储存的元素替换为x;
    - void replace(Edge e, Element x) : 将edge e储存的元素替换为x;
- Update methods:

    - insertVertex(Element o) : 插入一个端点，储存元素为o;
    - insertEdge(v,w,o) : 创建新边(v,w)，储存元素为o;
    - removeVertex(Vertex v) : 删除端点v及与该端点连接的所有边;
    - remove edge(Edge e) : 删除某条边，注意两个端点是保留的;
- Iterator methods:

- incidentEdges(v) : 给出端点v连接的所有边;
- vertices() : 给出图中所有端点;
- edges() : 给出图中所有的边;

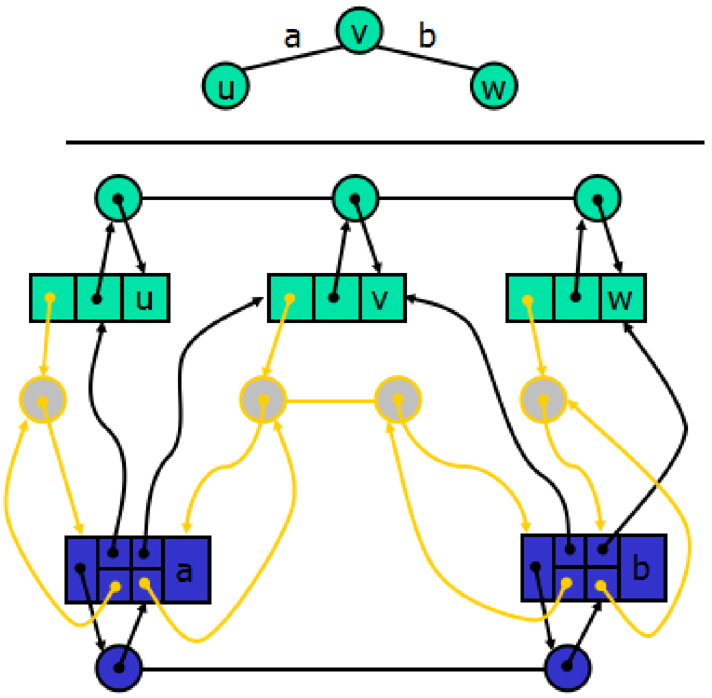
11.1.5 Edge List Structure

- Object Vertex:
  - element;
  - reference to position in vertex sequence;
- Object Edge:
  - element;
  - start vertex;
  - end vertex;
  - reference to position in edge sequence;
- Sequence Vertex:存储vertex的序列;
- Sequence Edge:存储edge对象的序列;



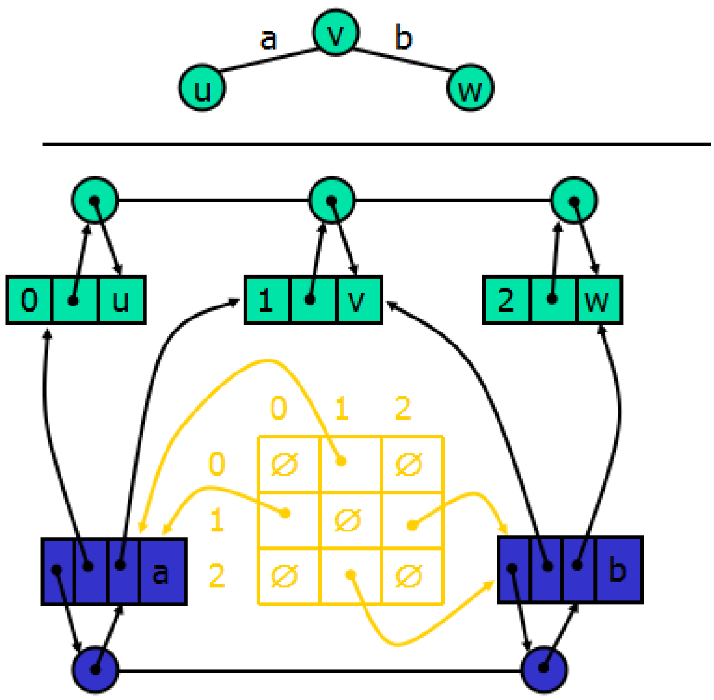
11.1.6 Adjacency List Structure

- Edge list structure;
- Incidence sequence:
  - 包含元素为某端点v相连边的链接;
- Augmented edge objects:
  - 指向Incidence sequence中元素的位置;



11.1.7 Adjacency Matrix Structure

- Edge list structure;
- Augmented vertex objects:index associated with vertex;
- 2D-Array adjacentry array:
  - reference to edge object for adjacent vertices;
  - Null for nonadjacent vertices;



11.1.8 三种结构的性能 Performance//

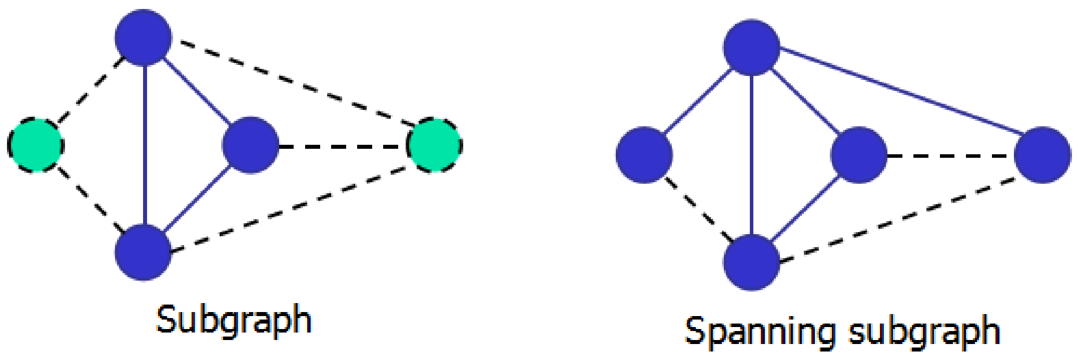
n vertices ; m edges ; 无平行边及自循环	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n+m)$	$O(n+m)$	$O(n^2)$
incidentEdges(v)	$O(m)$	$O(deg(v))$	$O(n)$
areAdjacent(v,w)	$O(m)$	$O(min(deg(v),deg(w)))$	$O(1)$
insertVertex(o)	$O(1)$	$O(1)$	$O(n^2)$
insertEdge(v,w,o)	$O(1)$	$O(1)$	$O(1)$
removeVertex(v)	$O(m)$	$O(deg(v))$	$O(n^2)$
removeEdge(e)	$O(1)$	$O(1)$	$O(1)$

11.2 Depth-First Search

- Applications:
  - path finding;
  - cycle finding;

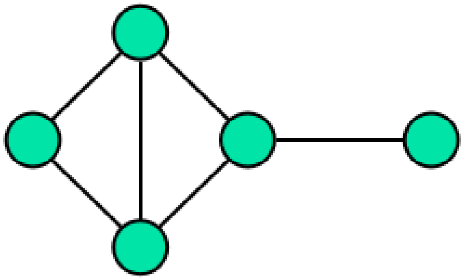
11.2.1 Subgraphs

- subgraph S与graph G的关系:
  - S的端点是G端点的子集;
  - S的边是G的边的子集;
- Spanning subgraph:包含G所有的端点,但边要少
  - 子图的端点不多于母图,边少于母图
  - Spanning Tree的边数为n-1

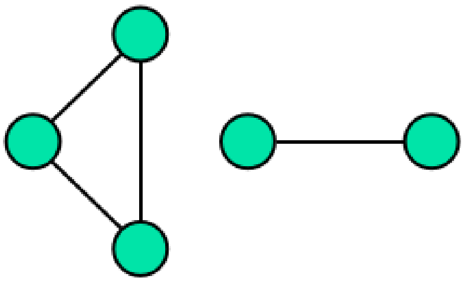


11.2.2 Connectivity

- 定义:任意两个端点都能有路径相连，则connected;
- Connected component : G中connected subgraph的数量;



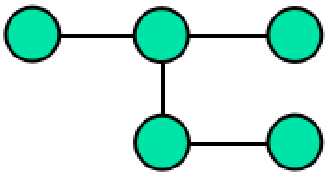
Connected graph



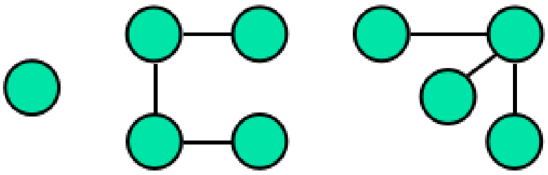
Non connected graph with two connected components

11.2.3 Trees and Forests

- Tree : 完全连通的无向图 , no cycles;
- Forest : 不完全连通的无向图,no cycles , 其中每个连通的部分都是tree;
- 树和森林的区别在于是否connected;



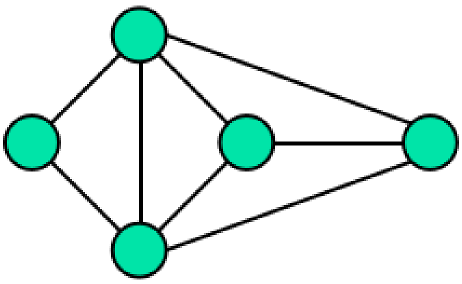
Tree



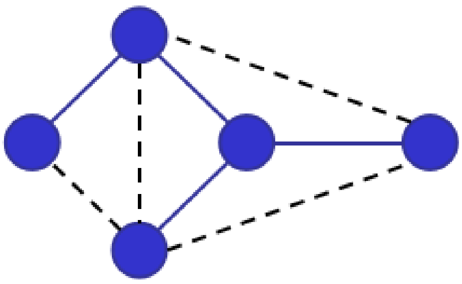
Forest

11.2.4 Spanning Trees and Forests

- spanning tree : spanning subgraph that is a tree;
- 除非原图就是棵树,否则spanning tree不是唯一的 (边可变)
- spanning forest : spanning subgraph that is a forest;



Graph



Spanning tree

11.2.5 Depth-First Search(DFS)

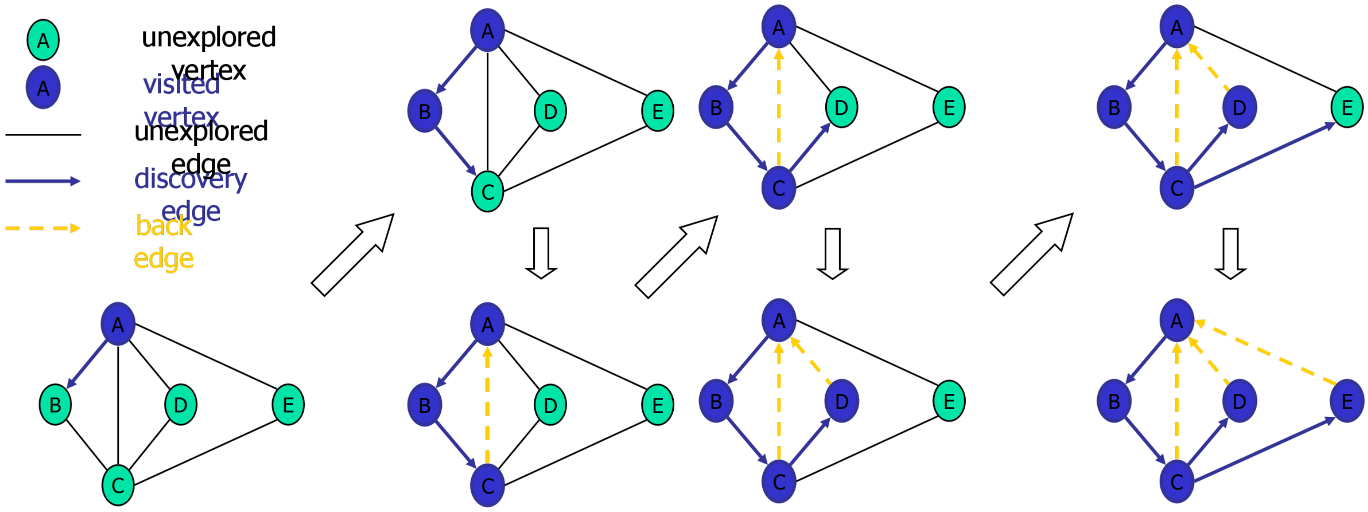
- 遍历图的过程:
  - 访问G所有的vetices、edges;
  - 确定G是否connected;

- 给出G的connecgted components;
- 给出G的spanning forest;

- 时间复杂度 $O(n+m)$

11.2.6 DFS Algorithm

```
Algorithm DFS(G,v)
Input: graph G and a start vertex v;
Output: 给v的connected component的几条边做标记,前进边discovery edges,后退边back edges;
{
    setLabel(v,VISITED);//将起始端点设为VISITED
    for e in G.incidentEdge(v){//e:与v相连的边
        if(getLabel(e)==UNEXPLORED){
            w=opposite(v,e);//另一侧端点
            if(getLabel(w)==UNEXPLORED){
                //边和另一侧端点都没有被探索->更改边标签
                setLabel(e,DISCOVERY);//注意此时点标签还是UNEXPOLRED
                DFS(G,w);//递归
            }
            else{//w已经被探索了
                setLabel(e,BACK);
            }
        }
    }
}

Algorithm DFS(G)
Input:graph G;
Output:给G所有的边都加上标签
{
    //初始化端点和边的标签为"UNEXPLORED"
    for u in G.vertices(){
        setLabel(u,UNEXPLORED);
    }
    for e in G.edges(){
        setLabel(e,UNEXPLORED);
    }
    for v in G.vertices(){//对新端点调用前面的算法
        if(getLabel(v)==UNEXPLORED) DFS(G,v);
    }
}
```



11.2.7 DFS的性质与分析

- Properties:
  - DFS(G,v)访问v的connected component的所有边和端点,并对边做标记;
  - DFS(G,v)的DISCOVERY边构成了v的spanning tree;
- Analysis:
  - set/get a label : $O(1)$;
  - 每个端点被标记两次 : UNEXPLORED -> VISITED;
  - 每条边被标记两次 : UNEXPLORED->DISCOVERY/BACK;
  - 每次调用DFS(G,v) , 执行一次incidentEdge();
  - 运行时间 $O(n+m)$;

11.2.8 Application : Path Finding

- 给定端点v,z,得到path
  - v : 起始端点;
  - Stack S : 记录轨迹(v到当前端点的路径);
  - 到达终点z , 以stack contents的形式返回路径;

```
Algorithm pathDFS(G,v,z)
Input: Graph G, start vertex v, end vertex z;
Output: the path from v to z( elements in Stack S);
{
    setLabel(v,VISITED);
    S.push(v);//v入栈作为起点
    if(v==z) return S.elements(); //递归到最后就会得到这个结果
    for e in G.incidentEdges(v){
        if(getLabel(e)==UNEXPLORED){
            w=opposite(v,e);
            if(getLabel(w)==UNEXPLORED){
                setLabel(e,DISCOVERY);
```

```
                S.push(e);//把e加入栈
                pathDFS(G,w,z);//递归
                S.pop(e);//返回e,并将e移出栈
            }
            else setLabel(e,BACK);
        }
    }
    S.pop(v);//返回栈顶元素v,并将v移出栈
}
```

11.2.9 Application : Cycle Finding

- 查找simple cycle
  - 同样使用Stack S跟踪路径;
  - 只要出现了BACK edge(v,w),以stack中一部分的形式返回cycle;

```
Algorithm cycleDFS(G,v,z){
    setLabel(v,VISITED);
    S.push(v);//v入栈作为起点
    if(v==z) return S.elements();
    for e in G.incidentEdges(v){
        if(getLabel(e)==UNEXPLORED){
            w=opposite(v,e);
            if(getLabel(w)==UNEXPLORED){
                setLabel(e,DISCOVERY);
                S.push(e);//把e加入栈
                pathDFS(G,w,z);//递归
                S.pop(e);//返回e,并将e移出栈
            }
            else{
                T=new empty stack;
                repeat{
                    o=S.pop();
                    T.push(o);
                }
                until(o==w);
                return T.elements;
            }
        }
    S.pop(v);
}
```

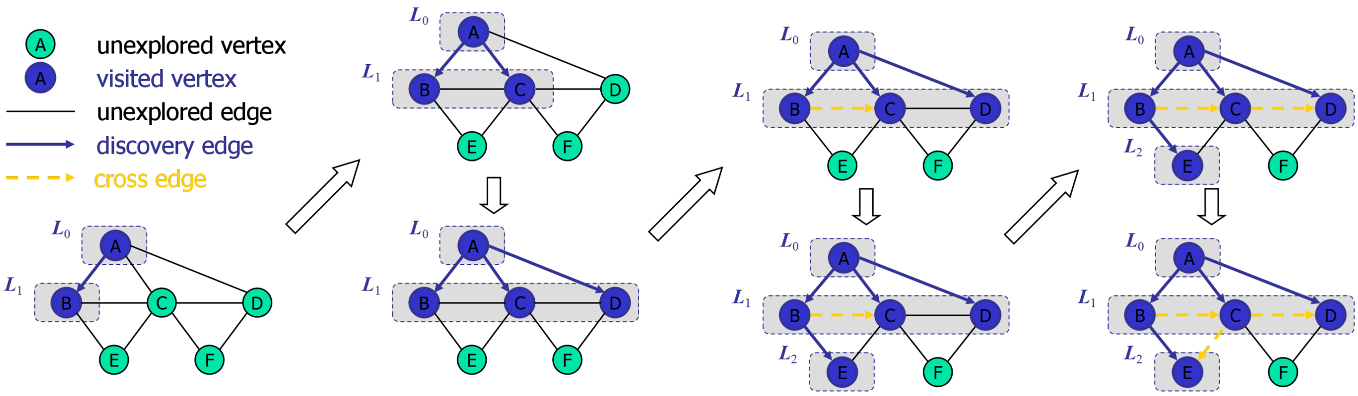
11.3 Breadth-First Search

- Applications
  - 给定端点，返回最短路径;

- 给出simple cycle;

11.3.1 BFS Algorithm

```
Algorithm BFS(G,s){//和DFS(G,v)相比不用递归
    L0=new empty sequence;
    L0.insertLast(s);
    setLabel(s,VISITED)//起始端点
    i=0;
    while(-Li.isEmpty()){
        L(i+1)=new empty sequence;
        for v in Li.elements(){
            for e in G.incidentEdges(v){
                if(getLabel(e)==UNEXPLORED){
                    w=opposite(v,e);
                    if(getLabel(w)==UNEXPLORED){
                        setLabel(e,DISCOVERY);
                        setLabel(w,VISITED);
                        L(i+1).insertLast(w);
                    }
                    else setLabel(e,CROSS);
                }
            }
        }
        i++;
    }
}

Algorithm BFS(G)
Input:graph G;
Output:给G所有的边都加上标签
{
    for u in G.vertices(){
        setLabel(u,UNEXPLORED);
    }
    for e in G.edges(){
        setLabel(e,UNEXPLORED);
    }
    for v in G.vertices(){
        if(getLabel(v)==UNEXPLORED) BFS(G,v);
    }
}
```



- 可视化解读
  - 每一层就是一个L_i;
  - 访问A端点所有的边, then B端点所有的边，...
  - 不把每个端点连接的边都访问了就不跳到下一个端点
  - 既有同层又有下层，优先访问同层(CROSS)
  - 访问某个未访问的点：DISCOVERY
  - 访问某个已被访问的点：CROSS

11.3.2 性质与分析

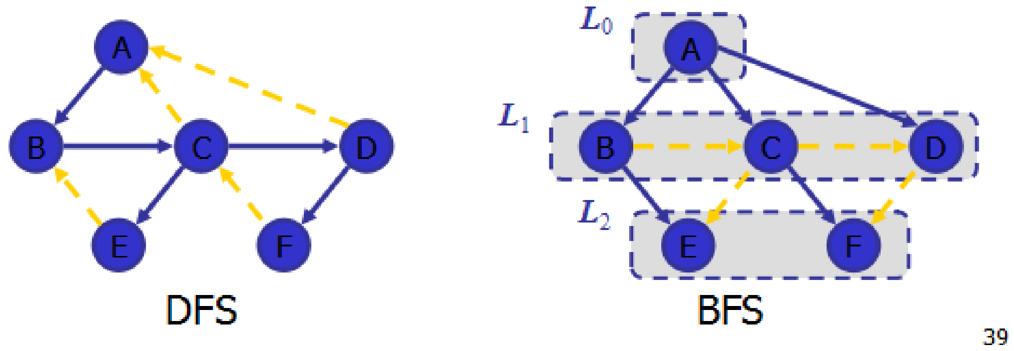
- G_s : connected component of s;
- Properties:
  - BFS(G,s)访问所有的端点以及G_s的边;
  - DISCOVERY边构成G_s 的spanning tree T_s;
  - 对于L_i的每个端点v:
    - T_s中从s到v的路径有i条边;
    - G_s中从s到v的路径至少i条边;
- Analysis:
  - set/get label : $O(1)$
  - vertex被标记两次：UNEXPLORED->VISITED;
  - edge被标记两次：UNEXPLORED->DISCOVERY/CROSS;
  - 每个端点只会被插入某个L_i一次;
  - 对于每个端点,调用一次incidentEdges();
  - 运行时间$O(n+m)$;

11.3.3 Application

- Compute the connected components of G;
- Compute a spanning forest of G;
- Find a simple cycle in G, or report that G is a forest;

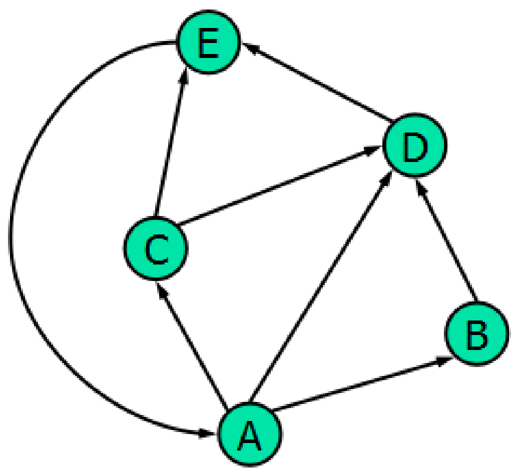
- 给定两个端点找最小路径,或返回"不相连";

11.3.4 DFS VS BFS



Applications	DFS	BFS
Spanning forest , connected components , paths , cycles	Y	Y
Biconnected components	Y	N
Shortest paths	N	Y
Label	Back edge(v,w) : w是v的祖先	Cross edge(v,w) : w是v的同级或下级, 且w已被访问

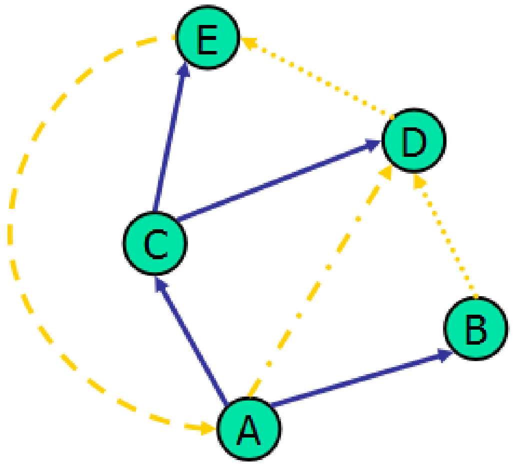
11.4 Directed Graphs:G=(V,E)



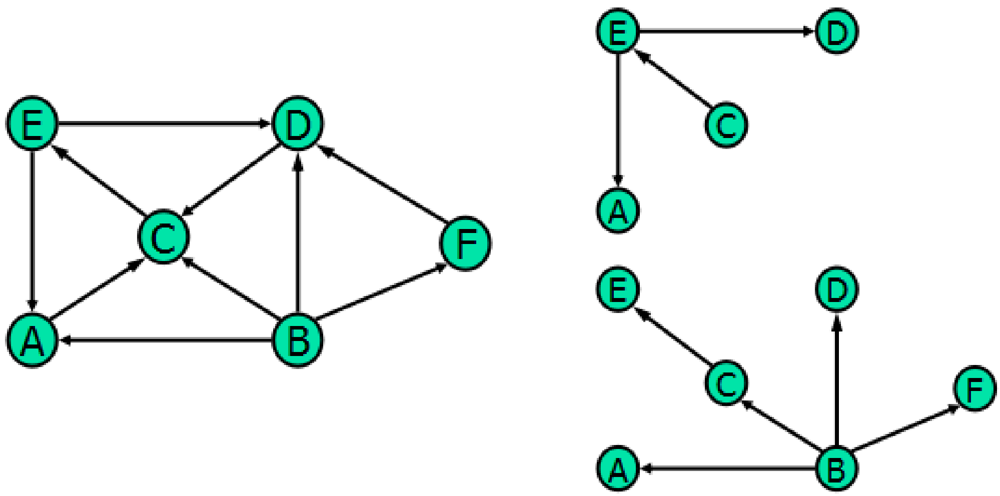
- Digraph : 所有的边都是有向的
- Applications:
    - 单行线;
    - flights;

    - 任务日程表：edge(a,b)在完成任务a前不可开始任务b;
- if G(m,n) is simple , $m \le n * (n-1)$$

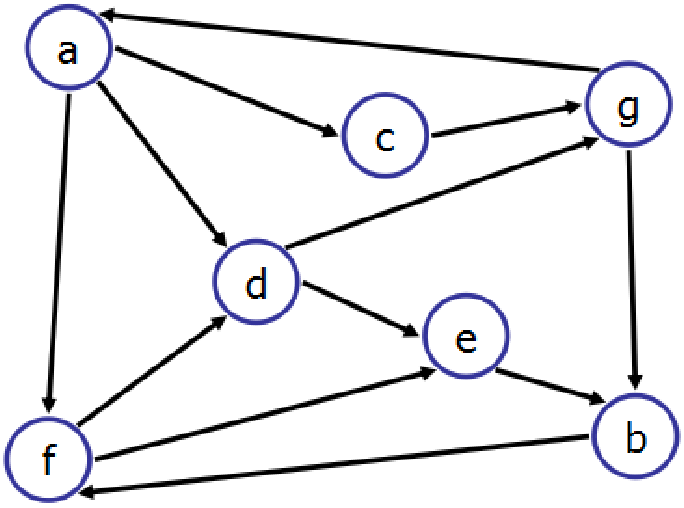
11.4.1 Directed DFS



- 沿着方向遍历edges
- 四种边：DISCOVERY/BACK/FORWARD/CROSS
- Reachability：给定端点v,经过有向路径可到达的端点数量



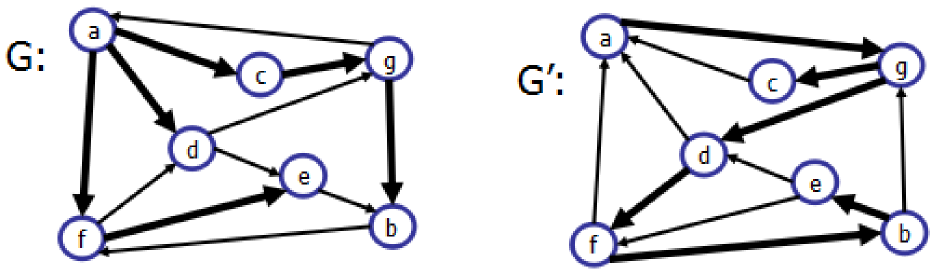
11.4.2 Strong Connectivity



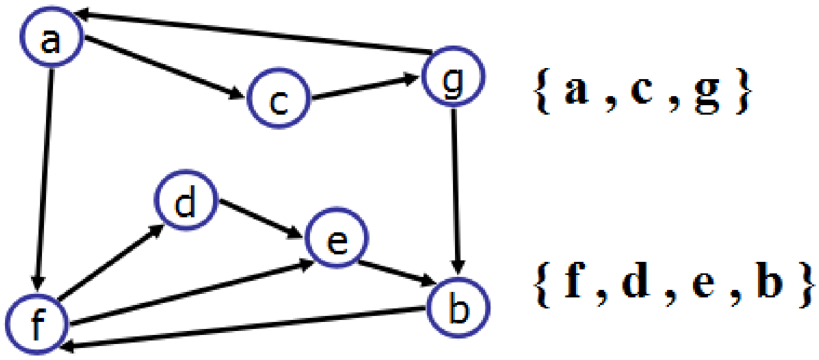
- 定义:每个端点都可以到达其余所有端点
- $reachable(v_i) = n - 1$

```
Algorithm IsStrongConnectivity
{
    Random pick a vertex in G;
    DFS(G,v);
    if (存在未被访问的端点w): return false
    G'=端点与G相同,但所有边的方向相反
    DFS(G',v);
    if (存在未被访问的端点w):return false;
    else: return true;
}
```

- 分析:
  - 正遍历一次,反遍历一次;
  - $O(n+m)$

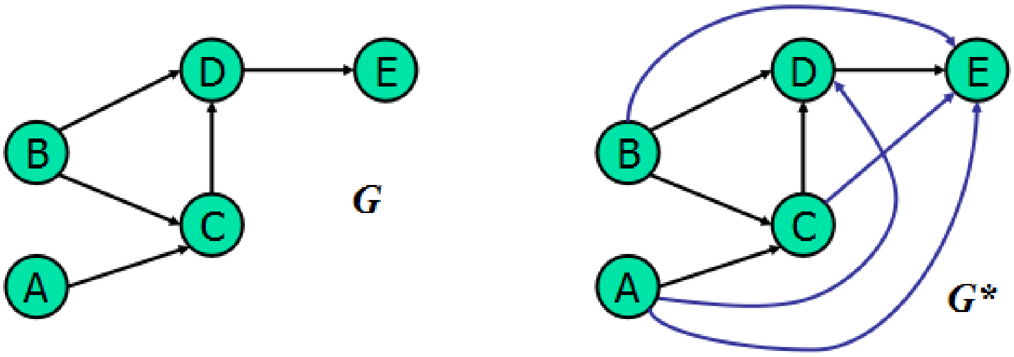


- Strong Connected Components:
  - 保证Strong connectivity的前提下得到的最大subgraph
  - 同样可以通过DFS得到,但更复杂!



11.4.3 Transitive Closure

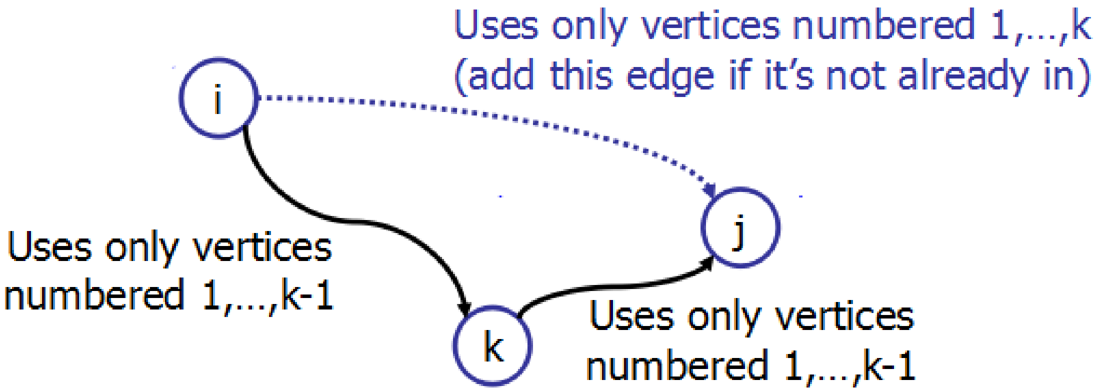
- 传递闭包可以为有向图提供可到达性信息
- 定义：给定有向图G，G的传递闭包G*为满足以下条件的有向图
  - G*与G的端点数相同(spanning);
  - if (u -> v) 为G中一个有向路径,则在G*中(u -> v)为一条有向边;
  - 其实就是给可到达的两个端点多连上一条线令其能直接到达



- 实现传递闭包：
  - DFS : $O(n(n+m))$
  - Floyd-Warshall Algorithm

11.4.4 Floyd-Warshall Algorithm

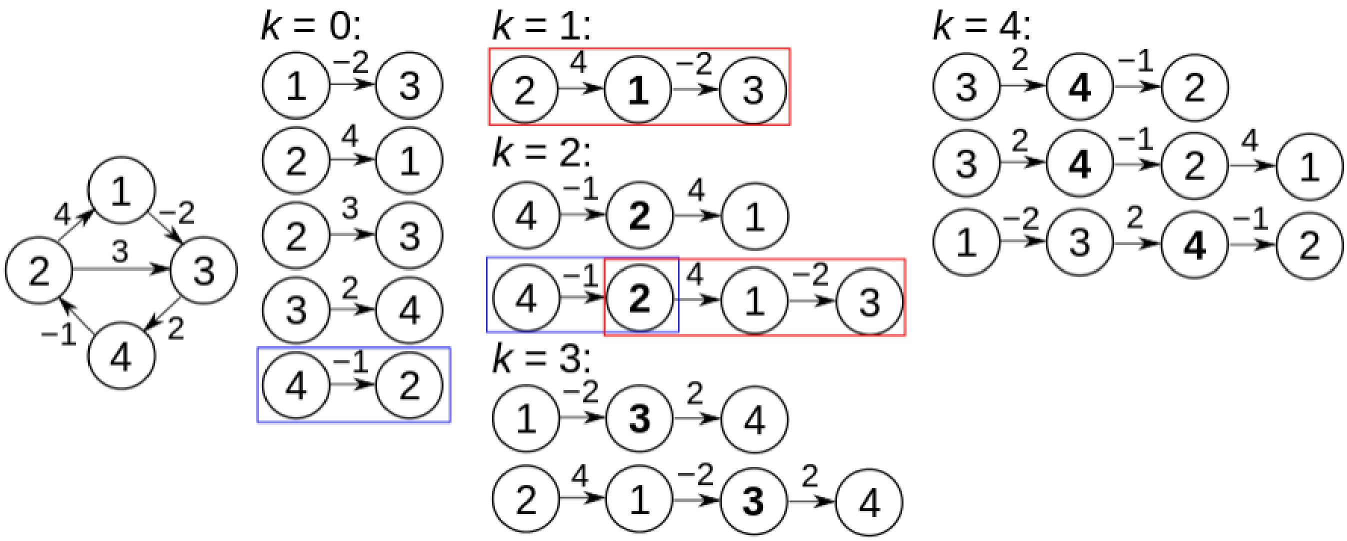
- 任意两点间的最短路径(实现传递闭包)
- 思路:
  - 给所有端点编号:1,2,...,n;
  - 取一个端点k作为中转端点



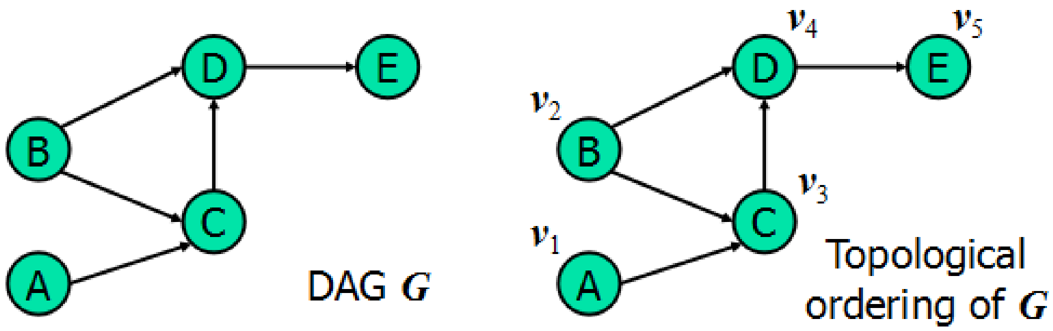

```
Algorithm FloydWarshall(G)
Input:digraph G;
Output:transitive closure G* of G;
{
    i = 1;
    for v in G.vertices(){//将每个端点编号
        denote v as vi ;
        i = i + 1;
    }
    G0 = G;
    for (k in [1,n]){
        Gk = Gk -1;
        for (i in [1,n]) and (i!=k):
            for (j in [i,n]) and (j!= i,k):
                if (G(k-1).areAdjacent(vi,vk)) and (G(k-1).areAdjacent(vk,vj)):
                    if Gk.areAdjacent(vi,vj):
                        Gk.insertDirectedEdge(vi,vj,k);
    }
  return Gn
}
```

- 比较复杂，之后考虑补图
- 运行时间$O(n^3)$,仅与端点数有关
- 维基上的理解
  - 设$D(i,j,k)$为从i到j的只以(1...k)集合中的结点为中间结点的最短路径长度;
  - 若最短路径经过k：$D(i,j,k) = D(i,k,k-1) + D(k,j,k-1)$;
  - 最短路径不经过k：$D(i,j,k) = D(i,j,k-1)$

$$D(i,j,k) = min(D(i,k,k-1) + D(k,j,k-1), D(i,j,k-1))$$



11.4.5 DAGs and Topological Ordering



- DAG(directed acyclic graph)：有向无环图,没有cycle的有向图;
- 在DAG中，任何一个端点都没法回到自身
- 拓扑有序：给端点编号,对于每条边$(v_i, v_j), i < j$;
- 有且仅有DAG才能实现拓扑有序

11.4.6 拓扑排序算法

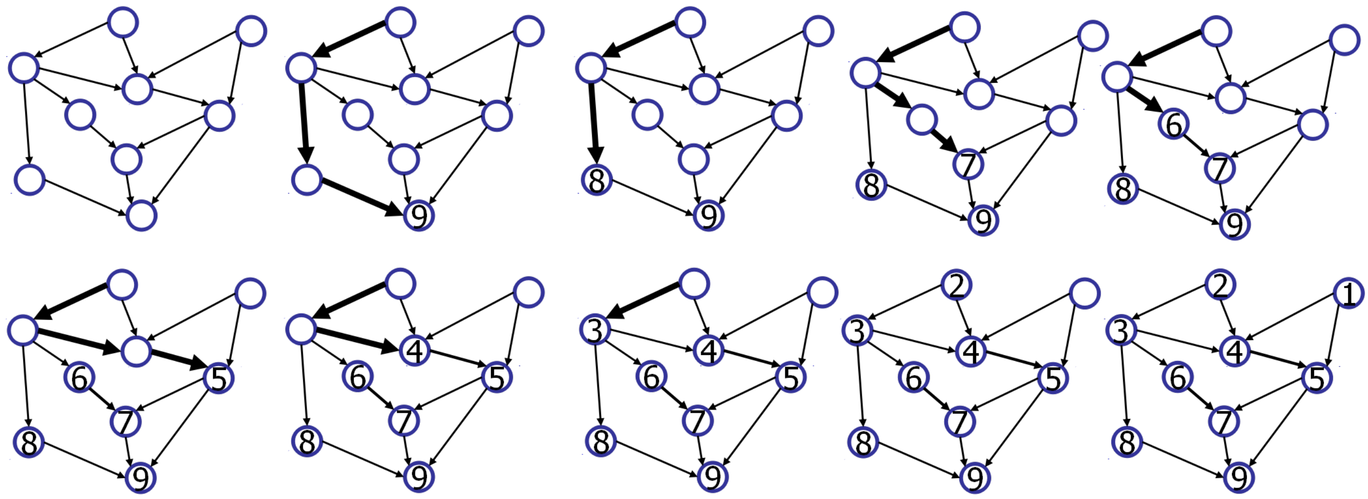
- $O(n+m)$

```
Method TopologicalSort(G)
{
    H=G;//先创建一个副本
    n=G.numVertices(){
    while (!H.isEmpty()){}
        令v作为一个没有outgoing edges 的端点;
        Label v=n;
        n--;
        H.remove(v);
    }
}
```

11.4.7 DFS实现拓扑排序 $O(m+n)$

- 在一个结点的所有子结点未被编号前,不给这个结点编号

```
Algorithm topologicalDFS(G,v)
Input:graph G and a start vertex v in G;
Output: 给v所有的connected component加上标签;
{
    setLabel(v,VISITED);
    for e in G.incidentEdges(v){
        if(getLabel(e)==UNEXPLORED){
            w=opposite(v,e);
            if(getLabel(w)==UNEXPLORED){
                setLabel(e,DISCOVERY);
                topologicalDFS(G,w);
            }
        }
    }
    Label v with topological index n;//替换原DFS的BACK Label
    n--;
    return;
}

Algorithm topologicalDFS(G)
Input: digraph G;
Output: topological ordering of G;
{
    n=G.numBertices();
    for u in G.vertices(){
        setLabel(u,UNEXPLORED);
    }
    for e in G.edges(){
        setLabel(e,UNEXPLORED);
    }
    for v in G.vertices(){
        if(getLabel(v)==UNEXPLORED){
            topologicalDFS(G,v);
        }
    }
}
```



Week 12 Graphs II

- Shortest Paths
- Minimum Spanning Trees

12.1 Shortest Paths

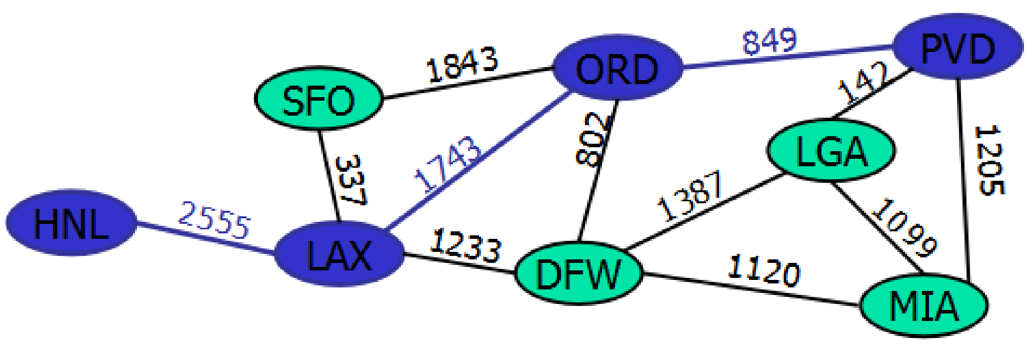
- 之前的路径算法包括BFS/DFS/Floyd-Warshall

12.1.1 Weighted Graphs

- 给每条边都加上权重：价格,距离等

12.1.2 最短路径

- 给定权重图中两个端点u / v ,得到边权重总和最小的路径;
- 最短路径的子路径也是最短路径;
- 从起始端点到其他任意端点的最短路径组成一棵树;



12.1.3 Dijkstra Algorithm

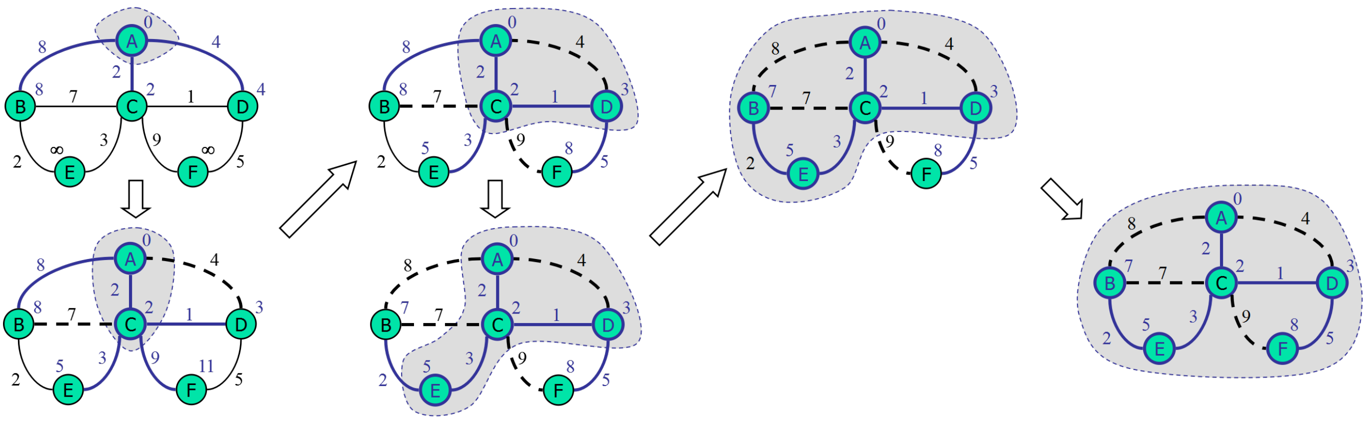
- distance(s,v):两端点之间的最短路径;
- 该算法的目标:给定起始点s,计算s到每个端点vi的距离label D(vi);
- 预设:
    - connected graph;
    - undirected edges;
    - nonnegative edge weight;
- At each step:
    - 计算s到端点v_i的距离d,加标签$D(v_i)=d$,并将该端点加入输出集合;
    - 计算与v_i相邻且还未加入输出集合的点;
    - 基于edge relaxation

12.1.4 Edge relaxation

- edge e=(u,z) : u是最近被加入输出集合的端点,z还未被加入;
- relaxation:以如下形式对D(z)进行更新

$$D(z)=min(D(z),D(u)+weight(e))$$

- 每个点都有初始距离参数，如果这个距离参数大于(前一个点+相邻边)，进行更新;
- 可能有许多路径，找其中距离最短的,更新初始距离参数



12.1.5 实现 Dijkstra 算法

```
Algorithm DijkstraDistance(G,s)
Input:
    simple undirected weighted G;
    weight>=0;
    distinguished vertex s;
Output:label D[u]->s到u的距离(u为G中任意一点)
{
    for v in G.vertices(){//初始化每个端点的距离label
        if(v==s){
            D[v]==0;
        }
```

```
        else D[v]=infinity;
    }

    Create a priority queue Q://需要里面的removeMin()
        element=vertex v in G;
        key=D[v];

    while(!Q.isEmpty()){
        u=Q.removeMin();
        for z in Q that is adjacent to u{//relaxation
                D[z]=min{D[z],D[u]+w((u,z))}
                在Q中更新z的key为D[z];
        }
    }
    return label D[u] for all u;
}
```

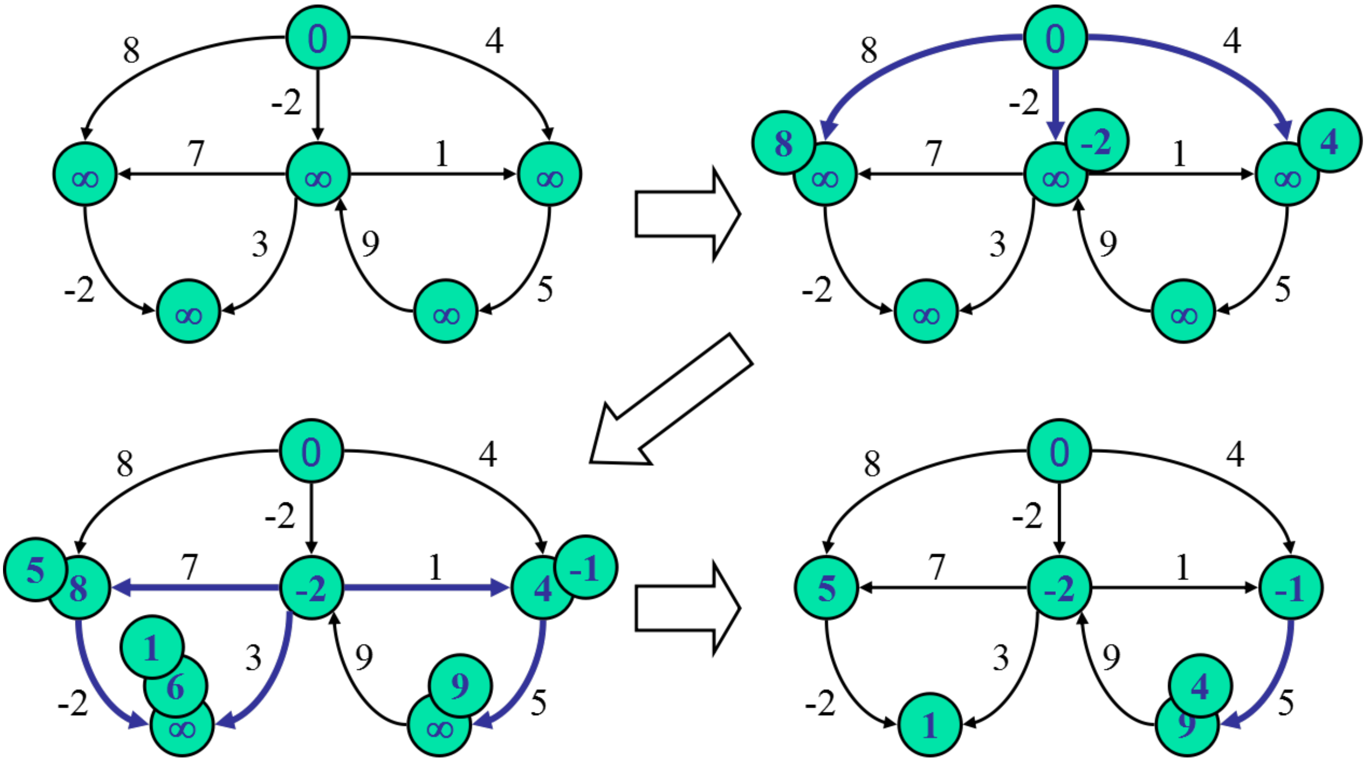
- 算法分析:
    - 创建优先队列Q:
        - adaptable PQ: $O(nlogn)$;
        - bottom-up heap constructure: $O(n)$;
    - while循环的每一步:
        - Q.removeMin() : $O(logn)$
        - relaxation : $O(deg(u)logn)$
        - while循环总共需要 : $O((n+m)logn)$
    - 整个算法: $O(mlogn)$
    - 该算法基于贪婪 : 距离升序增加端点
    - 不能有负的边权重

12.1.6 Bellman-Ford Algorithm

- 本课程不考察
- 即使边的权重是负的也能运行
- 要求G是有向图 , 运行时间$O(mn)$长于Dijkstra , 但可计算负权重

```
Algorithm BellmanFordDistance(G,s)
{
    for v in G.vertices(){
        if(v==s) D[v]=0;
        else D[v]=infinity;
    }
    for(i=1;i<n;i++){
        for e in G.edges(){
            u=G.origin(e);//有向边的起始端点
            z=G.opposite(u,e);//该边的终止端点
```

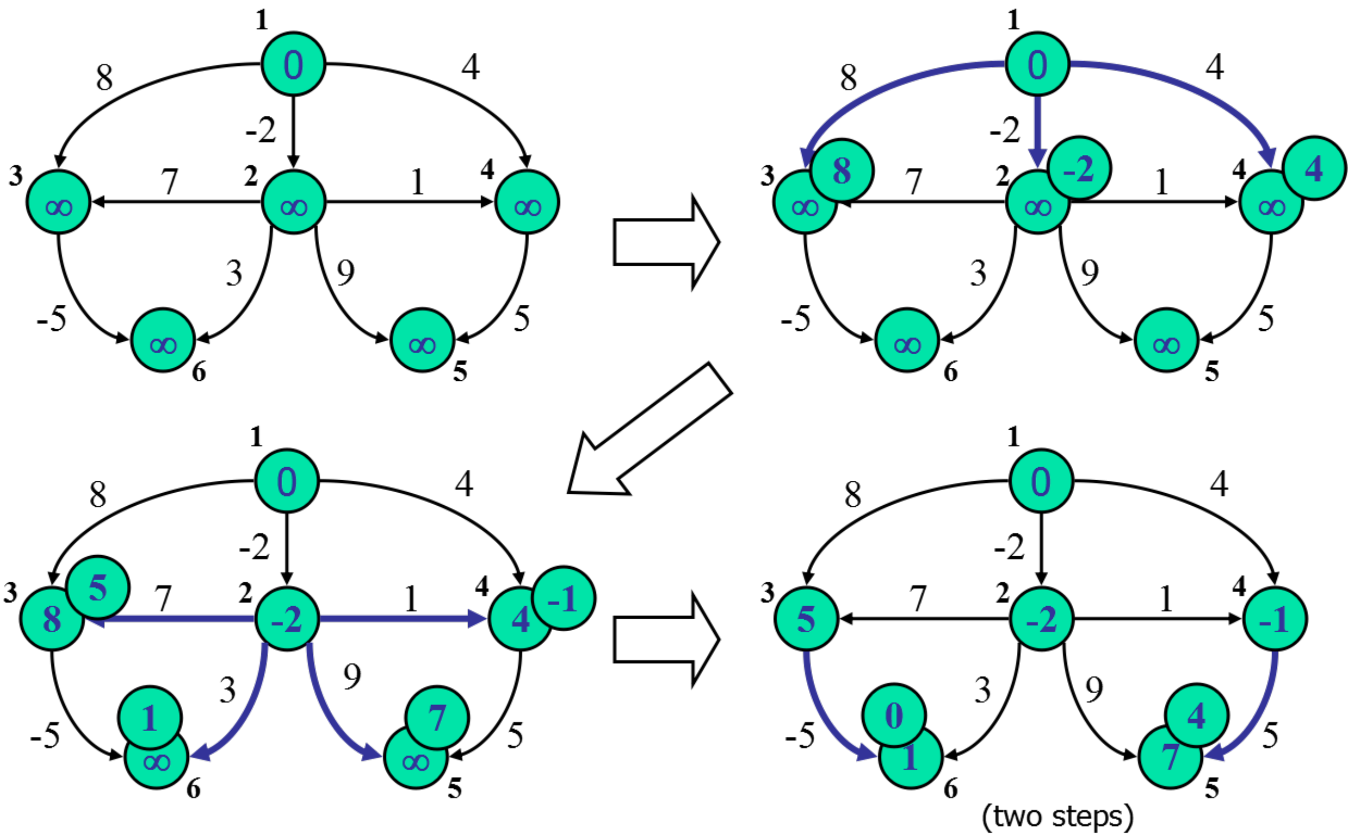
```
                r=D[u]+weight(e);
                D[z]=min{D[z],r};
            }
        }
    }
```



12.1.7 DAG-based Algorithm

- 本课程不考察
- 基于拓扑有序：对任意边,起始点权重不大于终点
- 同样可以对负权重边进行操作，但比Dijkstra算法还快：$O(m+n)$
- 要求：有向无环图

```
Algorithim DagDistance(G,s)
{
    for v in G.vertices(){
        if(v==s) D[v]=0;
        else D[v]=infinity;
    }
    Perform a topological sort of the vertices;
    for(u=1;u<=n;u++){
        z=G.opposite(u,e);
        r=D[u]+weight(e);
        D[z]=min{D[z],r};
    }
}
```

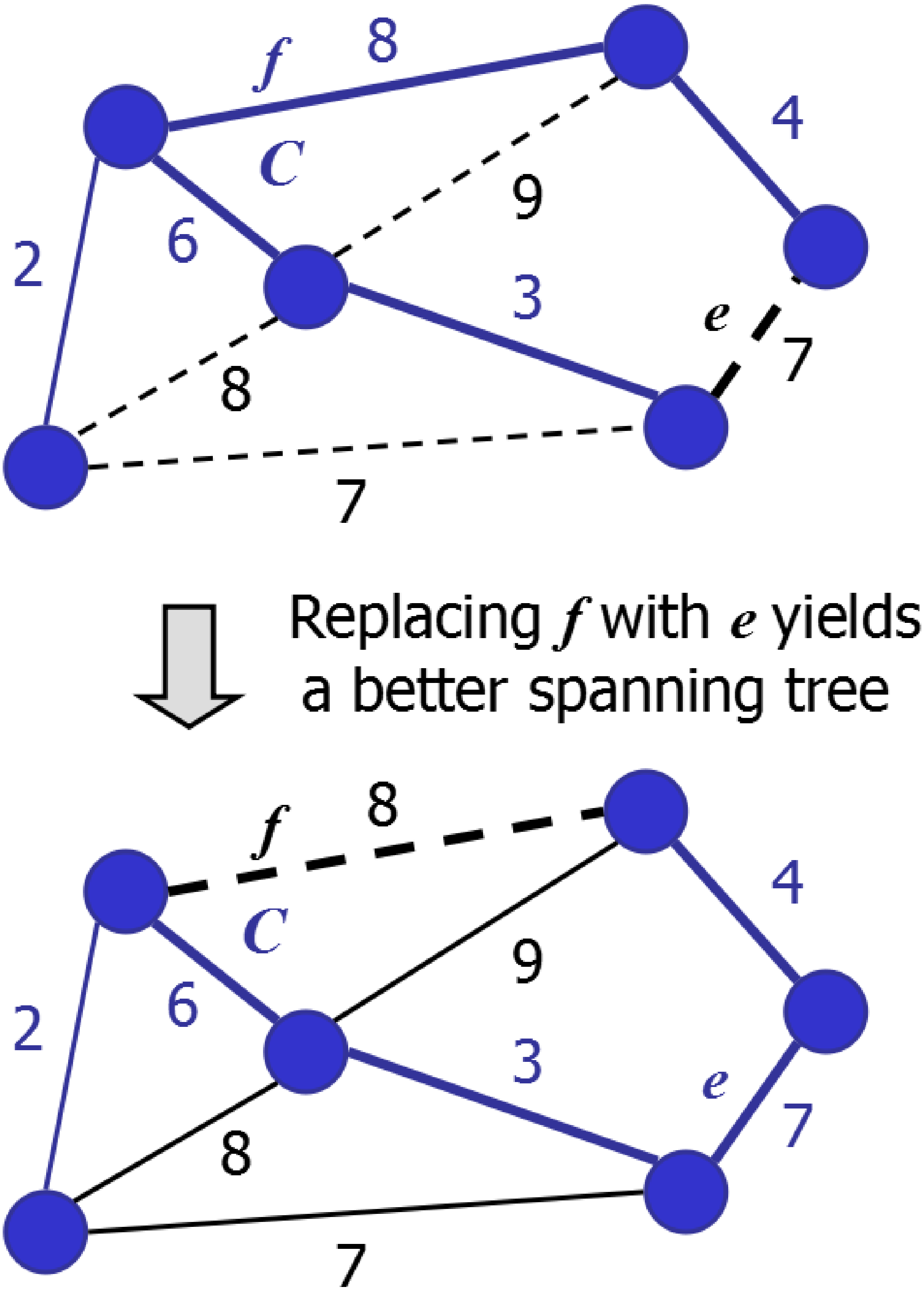


12.2 Minimum Spanning Trees(MST)

- Spanning tree: 包括了图中所有的点, 边组合可变
- MST:找出边权重加和最小的spanning tree;
- Applications : 通讯网和运输网

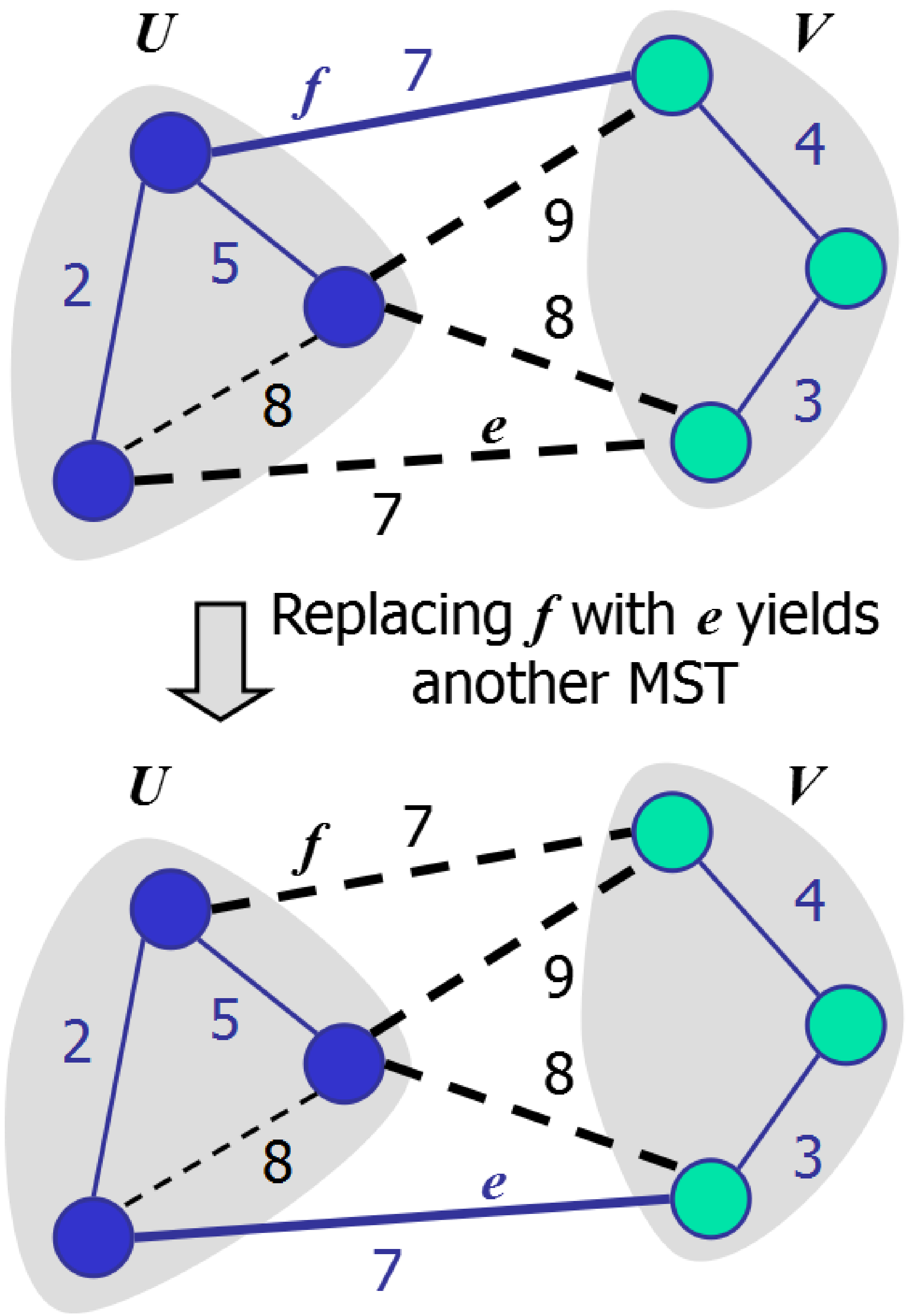
12.2.1 Cycle property

- T: 权重图G的MST;
- e: 一条不在T内的边, e与T的一部分组成了一个cycle C;
- 对C的任意一条边f,有 $weight(f) \leq weight(e)$
  - 因为如果不是这样，e一定会存在于MST中
  - 新加边e也是cycle C中权重最大的边



12.2.2 Partition Property

- 将G的端点切分成U/V两部分子集;
- e: 连接U/V的权重最小的边;
- 则G的MST必然包含e;



12.2.3 Kruskal Algorithm

- 和前面Dijkstra算法类似,使用优先队列Q储存未被加入输出集的内容:
  - Key:weight;
  - Element:edge;
- 该算法返回MST T;

```
Algorithm KruskalMST(G){
    for v in G.vertices(){
        define a Cloud(v) of {v};
    }

    Create priority queue Q://储存未被加入输出集的所有边
        element:edge e in G;
        key:weight;

    T=null;//初始化输出集MST为0

    while (T.edges().length()<n-1){//注意MST边数为n-1
        edge e =T.removeMin();
        u,v=endpoints(v);
        if(Cloud(v)!=Cloud(u){
            T.insertLast(e);
            Merge Cloud(v) and Cloud(u);
        }
    }
    return T;
}
```

- 算法分析:
    - 该算法构建了一森林的树cloud(v);
    - 如果某条边e连接了不同的树，将其加入T;
    - 构建数据结构实现partition
        - find(u) : 返回储存u的集合
        - union(u,v) : 找到储存u/v的集合并用这两个集合的并集替换

12.2.4 Representation of a Partition

- 每个set储存一个sequence;
- 每个element包含一个reference用于回退到set:
    - find(u) : return set where u is a member , $O(1)$
    - union(u,v) : 将比较小的set中的元素移动到大set并更新链接;
        - 操作时间$O(min(n_u, n_v))$
    - 对每个元素，将其加入两倍原集合大小的新集合，操作时间$O(logn)$
- 基于Partition的Kruskal算法实现: $O((m+n)logn)$
- 这里先略过栗子

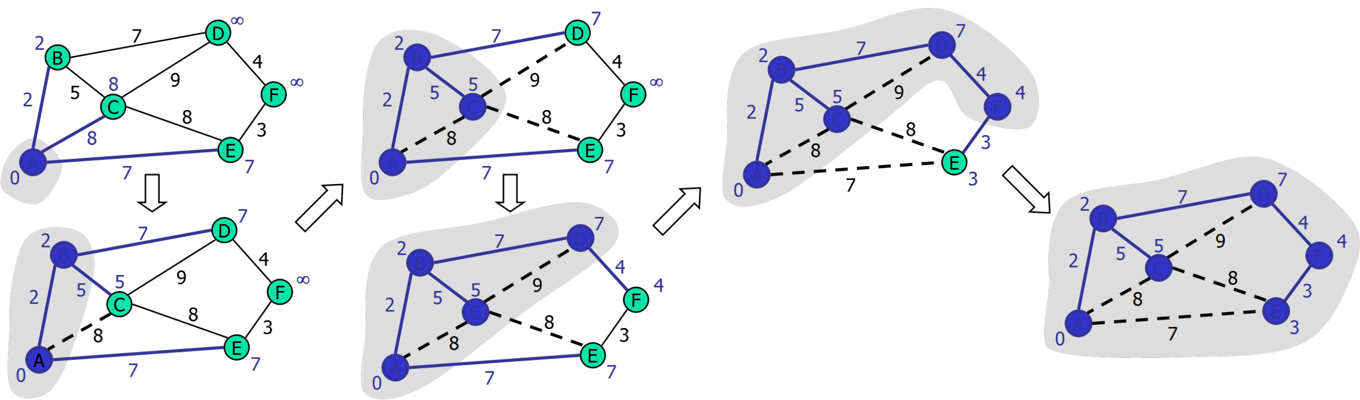
```
Algorithm KruskalMST(G)
Input: weighted graph G;
Output: MST T for G;
{
```

```
    P:partition of vertices in G,每个端点都是一个set;
    Q:PQ储存edges in G,sorted by key(weight);
    T:initially empty tree;
    while(!Q.isEmpty()){
        (u,v)=Q.removeMin().element();
        if(P.find(u)!=P.find(v)){
            Add e(u,v) to T;
            P.union(u,v);
        }
    }
    return T;
}
```

12.2.5 Prim-Jarnik Algorithm

- 与Dijkstra算法类似，都是对连通图操作
- 选定一个端点v,以其为起点创造MST (a cloud of vertices)
- 对每个端点u加标签D(u)：使u同cloud中端点相连的最小边的权重
- Each step:
    - 将MST外端点u加上标签D(u)后,将u加入cloud;
    - 更新u相邻点的标签;

```
Algorithm PrimJarnikMST(G)
{
    //初始化起始点v,剩余点u,T
    randomly pick any v of G;
    D[v]=0;
    for ((u in G.vertices()) && (u!=v)) D[u]= infinity;
    T=null;
    Create priority queue Q storing each vertex u:
        entry((u,null),D[u]);
        element=(u,null);
        key=D[u];

    while(!Q.isEmpty()){
        (u,e)=Q.removeMin();
        T.insertLast(vertex u,edge e);//每次拿出最小元素加入T
        for (z in Q) && (areAdjacent(z,u){
            if(w((u,z))<D[z]){
                D[z]=w((u,z));
                z.setElement(z,(u,z));//更新端点z在Q中的element
                z.setKey(D[z]);//更新端点z在Q中的key
            }
        }
    }
    return T;
}
```



- 算法分析:
  - Graph operation : 对每个端点调用一次 incidentEdges;
  - Label operation:
    - set/get z的距离/parent/标签: $O(deg(z))$;
    - set/get a label: $O(1)$;
  - PQ operation:
    - 每个端点都被插入一次 + 删除一次: $O(logn)$;
    - 端点w的key每次变化耗时$O(logn)$, 变化次数最多$O(deg(w))$;
  - 总运行时间:
    - 如果G的结构是adjacency list structure : $O((n+m)logn)$;
    - 如果G是连通图 : $O(mlogn)$;