

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:

```
1.  public class Counter{  
2.      //init=声明+构造器  
3.      protected int count;//声明  
4.      Counter() {//构造器  
5.          count=0;  
6.      }  
7.  
8.      //3 methods  
9.      public int getCount () {  
10.          return count++;  
11.      }  
12.      //void类型不需要返回值  
13.      public void incrementCount () {count++;}  
14.      public void decrementCount () {count--;}  
15. }
```

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

```
1. int i =5  
2. 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

```
1. public static ...
```

- Local variables:declared in a method

```
1. 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:

```

1.  private static final boolean RED=true;
2.  //这是一个类变量
3.  //既是static又是final
4.  //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:

```

1.  public class SingleLinkedList{
2.      //this is a constructor
3.      public SingleLinkedList() {
4.          head=null;
5.          size=0;
6.      }
7.
8.      //another constructor;

```

```
9.     public SingleLinkedList(int i=1){  
10.         ...  
11.     }
```

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:

```
1.     public class DataArtist { ...  
2.         public void draw(int i) { ... }  
3.         public void draw(double f) { ... }  
4.         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

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

1.4 Objects

```
1.     public class Asshole  
2.     {  
3.         public static void main (String[] args)  
4.         {  
5.             Counter c;  
6.             Counter d = new Counter(); //构造器实例化类，创建新对象，里面可以加参数  
7.             c = new Counter();  
8.             d = c;  
9.         }  
10.    }
```

1.4.1 String Objects

- 字符串对象immutable:创建后就无法再更改值

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

1.4.2 Object References

```
1.     Fucker() c=new Fucker(); //实例化一个类  
2.     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

```

1. int [] anArray=new int[10];
2. double [][] fuck=new double[5][1];
3.
4. for (int i=0; i<anArray.length; i++)
5.     anArray[i]=i;
6.
7. //arraycopy->数组复制
8. public static void arraycopy(Object src,
9.     int srcPos, //src开始复制的位置
10.    Object dest,
11.    int destPos, //dest接受复制的起始位
12.    int length//打算复制的元素
13. )
14.
15. class ArrayCopyDemo {
16.     public static void main(String[] args){
17.         char[] copyA = { 'd', 'e', 'c', 'a', 'f', 'e', 'i', 'n', 'a', 't', 'e',
18.             'd' };
19.         char[] copyB = new char[7];
20.         System.arraycopy(copyA, 2, copyB, 0, 7);
21.         System.out.println(new String(copyB));
22.     }
23. }
```

1.9 Nested Classes嵌套类

```

1. class OuterClass { ...
2.     class NestedClass { ... }
```

```
3. }
```

- 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:

```
1. //调用整个package
2. package java.awt.event;
3.
4. //仅仅调用部分类
5. import java.awt.event.ActionEvent;
6.
7. //调用import的方法
8. Scanner fuck=new Scanner(System.in);
9.
10. //另一种调用类的方法，不需要import
11. java.util.Scanner fuck=new java.util.Scanner(System.in);
```

1.11 I/O Streams: Input->Output

1.11.1 Output Methods

```
1. System.out.print();
2. System.out.printf();
3. System.out.println();
```

1.11.2 Input: Scanner class

- 举个栗子: 迭代读取文件内容

```

1. while(!xxx.hasNext()) {
2.     xxx.nextLine();
3.     ...
4. }
5.
6. //some other functions
7. Next() //next token in the input stream
8. hasNextType()
9. nextType() //the type of next token
10. hasNextLine()
11. nextLine()
12. 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 : 读取文本文件并返回为一行字符串, 再按空格切分为数组

```

1. import java.util.Scanner;
2. import java.io.File;
3. import java.io.FileInputStream;
4. import java.io.IOException;
5. import java.io.InputStreamReader;
6. import java.io.BufferedReader;
7.
8. public MyDlist(String f) throws IOException{
9.     super();
10.    if(f=="stdin"){
11.        Scanner myInput=new Scanner(System.in);
12.        while(true){
13.            String newElement=myInput.nextLine();
14.            if (!newElement.equals("")){
15.                DNode newDNode=new DNode(newElement,null,null);
16.                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
17.            }
18.            else{ break; }
19.        }
20.    }
21.
22.    else{
23.        File myFile=new File(f);
24.        InputStreamReader ISReader=new InputStreamReader(new FileInputStream(myF
ile)); //create a InputStreamReader object
25.        BufferedReader BReader=new BufferedReader(ISReader); //transfer the conte
nt to machine language
26.

```

```

27.         //initialize myLine
28.         String myLine = BReader.readLine();
29.         String temp=BReader.readLine();
30.
31.         while (temp!=null) {
32.             myLine = myLine+" "+temp;
33.             temp=BReader.readLine();
34.         }
35.
36.         String[] splitStrings=myLine.split(" +"); //split the string by one or m
ore spaces
37.         for(int i=0;i<splitStrings.length;i++) {
38.             DNode newDNode=new DNode(splitStrings[i],null,null);
39.             this.addLast(newDNode);
40.         }
41.     }
42. }
```

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

```

1.    //创建父类Bicycle
2.    public class Bicycle{
3.        //create 3 fields
4.        public int cadence;
5.        public int gear;
6.        public int speed;
7.
8.        //constructor
9.        public Bicycle(int startCadence, int startSpeed, int startGear) {
10.            gear = startGear;
11.            cadence = startCadence;
12.            speed = startSpeed; }
```

```

14.         //4 methods
15.     public void setCadence(int newValue) { cadence = newValue; }
16.     public void setGear(int newValue) { gear = newValue; }
17.     public void applyBrake(int decrement) { speed -=decrement; }
18.     public void speedUp(int increment) { speed += increment; }
19. }
20.
21. //创建子类
22. public class MountainBike extends Bicycle {
23.     //add a new field
24.     public int seatHeight;
25.
26.     //constructor
27.     public MountainBike(int startHeight, int startCadence, int startSpeed, int s
tartGear) {
28.         //一般情况下子类无法使用父类构造器,这里使用super
29.         super(startCadence, startSpeed, startGear);
30.         seatHeight = startHeight;
31.     }
32.
33.     //add a new method
34.     public void setHeight(int newValue) { seatHeight = newValue; }
35. }

```

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
 - overriding : 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

```

1. //创建父类并定义两个方法
2. public class Animal{
3.     public static void testClassMethod() {
4.         System.out.println("The class method in Animal.");
5.     }
6.
7.     public void testInstanceMethod() {
8.         System.out.println("The instance method in Animal.");
9.     }
10. }
11.
12. //子类覆盖方法, 每次调用父类都被隐藏
13. public class Cat extends Animal {
14.     public static void testClassMethod() {
15.         System.out.println("The class method in Cat.");
16.     }
17.     public void testInstanceMethod() {
18.         System.out.println("The instance method in Cat."); //overloading
19.     }
20.     public static void main(String[] args) {
21.         Cat myCat = new Cat();
22.         Animal myAnimal = myCat;
23.         Animal.testClassMethod();
24.         myAnimal.testInstanceMethod();
25.     }
26. }
```

- 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

```
1. public class Superclass {  
2.     public void printMethod() {  
3.         System.out.println("Printed in Superclass.");  
4.     }  
5. }  
6. public class Subclass1 extends Superclass {  
7.     public void printMethod(){  
8.         super.printMethod(); //调用父类方法  
9.         System.out.println("Printed in Subclass");  
10.    }  
11.    public static void main(String[] args) {  
12.        Subclass1 s = new Subclass1();  
13.        s.printMethod(); //调用子类方法  
14.    }  
15. }  
16.  
17. //Output:  
18. 'Printed in Superclass.'  
19. '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

```
1.     super() /super(parameter list)
```

- 举个栗子

```
1. public MountainBike(int startHeight, int startCadence, int startSpeed, int start  
2. Gear){  
3.     super(startCadence, startSpeed, startGear);  
4.     seatHeight = startHeight;
```

- 再举个栗子

```
1. public MyDlist() {  
2.     super();  
3. }
```

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

```
1. public class Point{  
2.     public int x=0;//实例变量  
3.     public int y=0;  
4.     public Point(int x,int y){  
5.         x=100;//局部变量  
6.         y=50;  
7.         this.x=x;//优先调用实例变量x=0  
8.     }  
9. }
```

- 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

```
1. class Student {  
2.     public int age;  
3.     public void std(){ //声明Student类的方法std()  
4.         age = 15;  
5.         System.out.println("学生平均年龄为："+age);  
6.     }  
7. }  
8.  
9. class ThisStudent extends Student{  
10.    public int age;  
11.    public void std(){  
12.        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  
13.        age = 18;  
14.        System.out.println("这个学生的年龄为："+age);  
15.        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  
16.        System.out.println(age);  
17.    }  
18. }  
19.  
20. public class TestDif {  
21.     public static void main(String[] args) {  
22.         ThisStudent a = new ThisStudent();  
23.         a.std();  
24.     }  
25. }
```

```
26.  
27.  
28.     >>>  
29.     学生平均年龄为 : 15  
30.     这个学生的年龄为 : 18  
31.     15  
32.     18
```

1.14 Exceptions

```
1.     public method() throws xxxException{  
2.         if(xxx) throw new xxxException("fuck you");  
3.     }
```

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`

```
1.     method(){  
2.         if (exception happened){  
3.             throw new ExceptionName();  
4.         }  
5.     }
```

- 举个栗子: 在声明method时指出异常 `throws`

```
1.     public void Fuck() throws CondomException{  
2.         submethod1(); //如果调用Fuck()时已经抛出异常,就不再检测子方法  
3.         submethod2();  
4.     }
```

1.14.3 Catching Exceptions

```
1.     try  
2.         main block of statements  
3.     catch (exceptionType1 variable1)  
4.         block 1 of statements  
5.     catch (exceptionType2 variable2)  
6.         block 2 of statements
```

```
7.     ...
8.     finally
9.         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

```
1. public interface Sellable{
2.     public String description();
3.     public int listPrice();
4.     public int lowestPrice();
5. }
```

1.15.2 调用接口:创建一个类 `implements` 接口

```
1. public class Photograpph implements Sellable{
2.     private String descript;
3.     private int price;
4.     private boolean color;
5.
6.     public Photograpgh(String desc, int p, boolean c) {
7.         descript = desc; price = p; color = c;
8.     }
9.     //执行接口: 在此处添加接口中methods的详情
10.    public String description() { return descript; }
11.    public int listPrice() { return price; }
12.    public int lowestPrice() { return price/2; }
13.    public boolean isColor() { return color; }
14. }
```

- 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）

```
1. Integer i = new Integer(3);  
2. Number n = i;  
3. //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

```
1. double n=new double(2.0);  
2. Integer i=(Integer) n;//2.0->2  
3. //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：

```
1. Number n;  
2. Integer i;  
3. n=new Integer(3);  
4. if(n instanceof Integer)//true  
5.     i=(Integer) n;  
6. n=new Double(3.1415);  
7. if(n instanceof Integer)//false  
8.     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

```
1. 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

```

1. Algorithm arrayMax(A, n) {
2.     Input: array A of n integers
3.     Output: maximum element of A
4.     currentMax = A[0];
5.     for (i =1; i< n; i++) {
6.         if (A[i]>currentMax) :
7.             currentMax =A[i];
8.     return currentMax;
9. }
```

2.2 Random Access Machine(RAM)模型

- 重要函数

- Constant 1 ($a=b+c$)
- Logarithmic $\log N$ (binary search)
- Linear N (单次循环)
- $N \cdot \log N$ $N \log N$ (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

```

1. Algorithm arrayMax(A,n) {                      //operations
2.     currentMax=A[0];                           //2
3.     for(i=1;i<n;i++)                          //2n
4.         if(A[I]>currentMax)                  //2(n-1)
5.             currentMax=A[i];                  //2(n-1)
6.     //fuck you(注释也算原语操作)      //2(n-1)
7.     return currentMax;                      //1
8. }
```

- 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 演近上界 O

- 定义: 存在正常数 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:

$$f(n) = 2n + 10, g(n) = n, \text{then } 2n + 10 = O(n)$$

$$2n + 10 < cn, \text{then } c = 3, n_0 = 10$$

- 举个反栗:
- $f(n) = n^2, g(n) = n, \text{then } n^2! = O(n)$
- $$n^2 \leq cn, \text{then } n \leq c$$

◦ 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:

- $7n - 2 = O(n);$
- $3n^3 + 20n^2 + 5 = O(n^3);$
- $3\log n + 5 = O(\log n)$

2.6 演进下界 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 演近确界 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 $c_1g(n) \leq f(n) \leq c_2g(n)$
- 当且仅当 $f(n) = O(g(n))$ 且 $f(n) = \Omega(g(n))$ 时, 有 $f(n) = \Theta(g(n))$
 - $f(n) \leq g(n), \text{then } f(n) = O(g(n))$
 - $f(n) \geq g(n), \text{then } f(n) = \Omega(g(n))$
 - $f(n) = g(n), \text{then } f(n) = \Theta(g(n))$
- 举个栗子: $5n^2$
 - $\Omega(n^2)(c = 5, n_0 = 1)$
 - $\Omega(n)(c = 1, n_0 = 1)$
 - $\Theta(n)(c = 5, n_0 = 1)$

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

- Abstract: 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 :

```
1.   ADT name{  
2.       object:  
3.       ...  
4.       relationship:  
5.       ...  
6.       operation set:  
7.           operation1:  
8.           operation2:  
9.           ...  
10.      }
```

3.2 Singly Linked List

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

```
1. //结点类  
2. public class Node{  
3.     private String element;  
4.     private Node next;  
5.     public Node(String s,Node n){ //结点构造器  
6.         element=s;  
7.         next=n;  
8.     }  
9. }
```

```

10.         //返回当前结点的element
11.     public String getElement() {return element;}
12.     //返回下一个结点
13.     public Node getNext() {return next;}
14.     //设置当前结点的element
15.     public void setElement(String s){elemtnt=s;}
16.     //设置下一个结点
17.     public void setNode(Node newN) {next=newN;}
18. }
19.
20. //链表类
21. public class SingleLinkedList{
22.     protected Node head; //注意这里用protected更好,同包的所有类都可以用
23.     protected long size; //numbers of nodes in the list
24.
25.     public SingleLinkedList() { //默认构造器,创建一个空链表
26.         head=null;
27.         size=0;
28.     }
29. }

```

3.2.1 表头插入Node v

- v.next指向原表头Node , head指针指向Node v

```

1. Algorithm addFirst(v){
2.     v.setNext(head); // make v point to the old head node
3.     head = v; // make head point to the new node
4.     size =size+1; // increment the node count
5. }

```

3.2.2 表头删除

- 将表头Node.next指向null,head指针指向第二个Node

```

1. Algorithm removeFirst() {
2.     if ( head == null )
3.         Indicate an error: the list is empty;
4.     t = head ; //原首结点
5.     head = head.getNext(); //head指向第二个Node
6.     t.setNext(null); //原表头Node的next指向Null
7.     size = size - 1;
8. }

```

3.2.3 尾部插入

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

```

1. Algorithm addLast(v) {

```

```

2.         v.setNext(null); //make the new node v point to null object
3.         tail.setNext(v); //make the old tail node point to the new node
4.         tail = v; //make tail point to the new node
5.         size = size +1;
6.     }

```

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

```

1.     Algorithm removeLast () {
2.         if (size=0)
3.             Indicate an error: the list is empty;
4.         else{
5.             if (size=1)
6.                 head = null; //删除唯一一个Node
7.             else{
8.                 t = head;//t指向head
9.                 while (t.getNext () != null) {//这个主要目的是找到倒数第二结点
10.                     s = t;
11.                     t = t.getNext ();
12.                 }
13.                 s.setNext (null); //将倒数第二个结点指向null
14.                 tail=s;
15.             }
16.             size = size - 1;
17.         }
18.     }

```

3.3 Doubly Linked List

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

```

1.     public class DNode{
2.         protected String element;
3.         protected DNode prev,next;
4.         public DNode (String e,,DNode p,DNode n){
5.             element=e;
6.             prev=p;
7.             next=n;
8.         }
9.
10.        public String getElement () {return element;}
11.        public DNode getPrev () {return prev;}

```

```

12.     public DNode getNext() { return next; }
13.     public void setElement(String newE) { elemtn=newE; }
14.     public void setPrevNode(Node newN) { prev=newN; }
15.     public void setNextNode(Node newN) { next=newN; }
16. }
17.
18. public class DoublyLinkedList {
19.     protected Node head; //如果用private->只有当前class可以用这个变量
20.     protected long size;
21.     public DoublyLinkedList() {
22.         head=null;
23.         size=0;
24.     }
25. }
26.
27. //调用
28. DoublyLinkedList fucker=new DoublyLinkedList()
29. fucker.setElement(...);
30. fucker.setNextNode(...);
31. //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

```

1. Algorithm addFirst(v) {
2.     w=header.getNextCode(); //找到首结点w (header只是个特殊链接,非首结点)
3.     v.setNextNode(w);
4.     v.setPrevNode(header);
5.     header.setNextNode(v);
6.     size =size+1;
7. }

```

3.3.2 中部插入

- 在Node v及其后结点Node w之间插入Node z

```

1. Algorithm addAfter(v,z) {
2.     w=v.getNextNode(); //将v后结点设为w
3.     z.setPrevNode(v); //将z的前结点设为v
4.     z.setNextNode(w); //z的后结点设为w,即原来v后面指向的链接
5.     w.setPrevNode(z); //w的前结点指向z
6.     v.setNextNode(z); //v的后结点指向z
7.     size=size+1;
8. }

```

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

```

1. Algorithm removeLast():
2.     if (size = 0)
3.         Indicate an error: this list is empty;
4.     v = trailer.getPrevNode()      //设v为结尾哨兵的前链接
5.     u = v.getPrevNode(); //u为v的前结点
6.     trailer.setPrevNode(u); //u与trailer互相链接,u成为新的尾结点
7.     u.setNextNode(trailer);
8.     v.setPrevNode(null); //v的前后结点指向null
9.     v.setNextNode(null);
10.    size = size -1;
11. }

```

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

```

1. Algorithm remove(v) {
2.     u = v.getPrevNode()      //u为v的前结点
3.     w = v.getNextNode(); //v为u的后结点
4.     w.setPrevNode(u);      //前后结点互相链接
5.     u.setNextNode(w);
6.     v.setPrevNode(null); // 删除v
7.     v.setNextNode(null);
8.     size = size -1;
9. }

```

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 $O(2^k)$

```

1. Algorithm BinaryFib(k) {
2.     Input : Nonnegative integer k
3.     Output : The kth Fibonacci number Fk
4.     if ( k =0 or 1) return 1;
5.     else
6.         return BinaryFib(k - 1) + BinaryFib(k - 2);
7. }

```

- 线性递归 $O(k)$

```

1. Algorithm LinearFibonacci (k) :{
2.     Input : A nonnegative integer k
3.     Output : Pair of Fibonacci numbers (Fk, Fk-1)
4.     if (k = 1) return (k, 0);
5.     else{
6.         (i,j)=LinearFibonacci (k - 1); //i=Fk-1, j=Fk-2
7.         return (i+j,i); //i+j=Fk, i=Fk-1
8.     }
9. }
```

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

```

1. public interface Stack {
2.     public int size();
3.     public boolean isEmpty();
4.     public Object top() throws EmptyStackException;
5.     public void push(Object o);
6.     public Object pop() throws EmptyStackException;
7. }
```

- 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]

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

3.
4. Algorithm pop()
5.
6.     if (isEmpty() ==true)
7.         throw EmptyStackException;
8.     else{
9.         t = t-1;
10.        return S[t + 1]; //即原来的栈顶元素t
11.    }
12. }

13.
14. Algorithm push(o)
15. {
16.     if (t=S.length-1)
17.         throw FullStackException;
18.     else
19.     {    t = t + 1;
20.         S [t] = o;
21.     }
22. }
```

- 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

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

```

6.         S=new Object[capacity];
7.     }
8.
9.     public Object pop() throws EmptyStackException{
10.        if(isEmpty()==true)
11.            throws EmptyStackException("Empty stack!")
12.        Object temp = S[top];
13.        S[top]=null; //删除栈顶元素
14.        top--;
15.        return temp; //返回之前被删除的栈顶元素
16.    }
17. }

```

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:

```

1. //先建立泛型结点class
2. public class Node<Type>{
3.     //声明实例变量element,next
4.     private Type element;
5.     private Node<Type> next;
6.
7.     //构造器 : overloading
8.     //先创建一个空Node:可以在之后调用setElement/setNext
9.     public Node() {this(null,null);}
10.    //创建一个带有给定内容和链接的Node
11.    public Node(Type e,Node<Type> n) {
12.        element=e;
13.        next=n;
14.    }
15.
16.    //methods:
17.    public Type getElement() { return element; }
18.    public Node<Type> getNext() { return next; }
19.    public void setElement(Type newElem) { element = newElem; }
20.    public void setNext(Node<Type> newNext) { next = newNext; }
21.
22. }
23.
24. //建立链表栈
25. public class NodeStack<Type> implements Stack<Type> {
26.     protected Node<Type> top;

```

```

27.     protected int size;
28.
29.     public NodeStack{//构造器:空栈
30.         top=null;
31.         size=0;
32.     }
33.
34.
35.     public int size(){return size;}
36.     public boolean isEmpty(){
37.         if(top==null) return true;
38.         else return false;
39.     }
40.     public void push(Type element){
41.         Node<Type> v=new Node<Type>(element,top); //Node v的next被定义为top(当前栈
顶元素)
42.         top=v;//使用v替换->v变为栈顶元素
43.         size++;
44.     }
45.     public Type pop() throws EmptyStackException{
46.         if(isEmpty()==true) throw new EmptyStackException("Empty stack!");
47.         Type temp=top.getElement();
48.         top=top.getNext(); //删除原首结点
49.         size--;
50.         return temp; //显示被删除原首结点的内容
51.     }
52. }

```

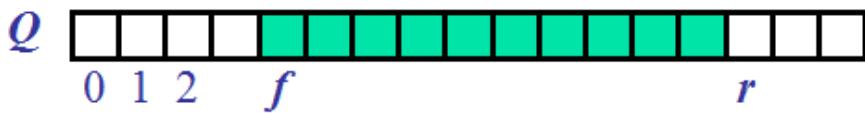
4.2 Queues - FIFO

- 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

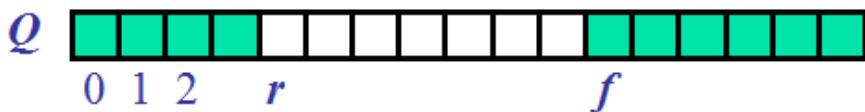
4.2.1 Array-based Queue

- 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

normal configuration



wrapped-around configuration



- Operations:

```

1. Algorithm size() {
2.     return (N - f + r) mod N;
3.
4. Algorithm isEmpty() {
5.     return (f = r);
6.
7. Algorithm enqueue(o) {
8.     if (size() = N-1)
9.         throw FullQueueException;
10.    else {
11.        Q[r] = o;
12.        r = (r+1) mod N ;
13.    }
14. }
15.
16. Algorithm dequeue() {
17.     if (isEmpty()=true)
18.         throw EmptyQueueException
19.     else {
20.         o = Q[f];
21.         f = (f + 1) mod N;
22.         return o;
23.     }
24. }
```

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

```

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

1. public interface Queue {
2.     public int size();
3.     public boolean isEmpty();
4.     public Object front() throws EmptyQueueException;
5.     public void enqueue(Object o);
6.     public Object dequeue() throws EmptyQueueException;
7. }
```

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 $O(1)$

```

1. public void enqueue(Type element) {
2.     Node<Type> fucker = new Node<Type>();
3.     fucker.setElement(element);
4.     fucker.setNext(null);
5.     if (size == 0) head = fucker; //空队列 , fucker为唯一的元素
6.     else tail.setNext(fucker); //原尾结点的链接设置为Node fucker
7.     tail = fucker; // 将Node fucker作为新的尾结点
8.     size++;
9. }
10.
11. public Type dequeue() throws EmptyQueueException {
12.     if (size == 0) throw new EmptyQueueException("Queue is empty.");
13.     Type currentHead = head;
14.     currentHead.setNext(null);
15.     head = head.getNext();
16.     size--;
17.     if (size == 0) tail = null; //the queue is now empty
18.     return currentHead.getElement(); //返回原首结点的值
19. }
```

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 $O(n)$

```

1. Algorithm add(i,e) {
2.     if(size==N) { A.resize(); }
3.     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
4.         A[j+1]=A[j];
5.     } //A[n]=A[n-1],A[n-1]=A[n-2]...A[i+1]=A[i]->空出位置A[i]
6.     A[i]=e;
7.     n=n+1;
8. }
```

- 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 $O(n)$

```

1. Algorithm remove(i) {
2.     e=A[i];
3.     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
4.         A[j]=A[j+1];
5.     } //A[i]=A[i+1],...A[n-2]=A[n-1]
6.     n=n-1;
7.     return e;
8. }
```

- resize() : 当一个数组填满元素时, 将其更换为容量更大的数组

```

1. Algorithm resize() {
2.     create a new array B with size 2N;
3.     for ( j = 0, 1, ... , size-1)
4.     {
5.         B[j]=A[j];
6.     }
7.     A = B;
8.     N=2N;
9. }
```

- Java 实现数组队列

```

1. //建立接口
2. public interface IndexList<E> {
3.     public int size();
4.     public boolean isEmpty();
5.     public void add(int i, E e) throws IndexOutOfBoundsException;
6.     public E get(int i) throws IndexOutOfBoundsException;
7.     public E remove(int i) throws IndexOutOfBoundsException;
8.     public E set(int i, E e) throws IndexOutOfBoundsException;
9. }
10.
11. //建立list class
12. public class ArrayIndexList<E> implements IndexList<E>{
```

```

13.     private E[] A;
14.     private int capacity = 16; //初始化数组A容量
15.     private int size = 0; //初始化元素数量
16.     public ArrayIndexList() {
17.         A = (E[]) new Object[capacity];
18.     }
19.
20.     //method: add e at index r
21.     public void add(int r, E e) throws IndexOutOfBoundsException {
22.         checkIndex(r, size() + 1); //于判断r是否在size() + 1范围内
23.         if (size == capacity) { //resize
24.             capacity *= 2;
25.             E[] B = (E[]) new Object[capacity]; //narrowing : Object的范围>B
26.             for (int i=0; i<size; i++) B[i] = A[i];
27.             A=B;
28.         }
29.         for (int i=size-1; i>=r; i--)
30.             A[i+1]=A[i];
31.         A[r]=e;
32.         size++;
33.     }
34.
35.     //method : remove the element at place r
36.     public E remove(int r) throws IndexOutOfBoundsException {
37.         checkIndex(r, size());
38.         E temp = A[r];
39.         for (int i=r; i<size-1; i++)
40.             A[i] = A[i+1];
41.         size--;
42.         return temp;
43.     }
44. }

```

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: $k = \frac{n}{c}$

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

- $T(n) = O(n + k^2)$, 每次push的平均时间: $O(n)$

- Doubling Strategy Analysis

- 交换次数: $k = \log n$

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

- $T(n) = O(n)$, 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

```
1. public interface Position<E>{  
2.     E element() throws InvalidPositionException;//get element  
3. }
```

- Node List ADT

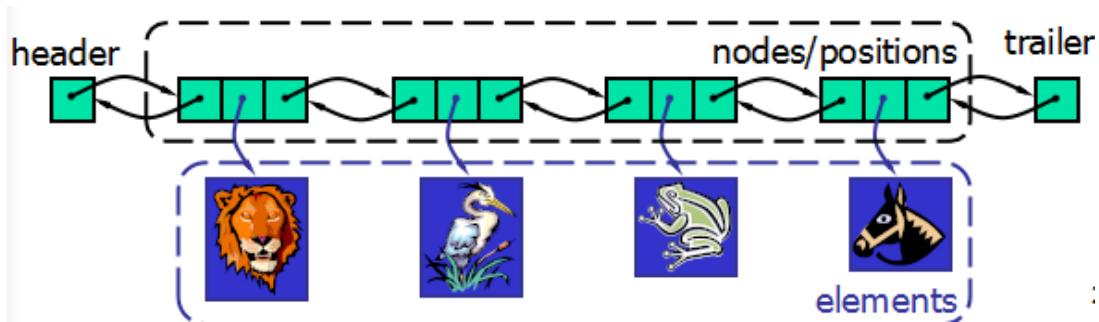
- Generic methods: size(),isEmpty()
- Accessor methods: first(),last(),prev(),next()
- update methods: set(p,e).addBefore(p,e),addAfter(p,e),addFirst(e),addLast(e),remove(p)

- Java interface for the Node List ADT

```
1. public interface PositionList<E>{  
2.     public int size();  
3.     public boolean isEmpty();  
4.  
5.     //返回头尾结点  
6.     public Position<E> first();  
7.     public Position<E> last();  
8.  
9.     //给出指定Node的前一个/后一个Node  
10.    public Position<E> next(Position<E> p) throws InvalidPositionException, BoundaryViolationException;  
11.    public Position<E> prev(Position<E> p) throws InvalidPositionException, BoundaryViolationException;  
12.  
13.    //头尾插入,返回插入位置Position<E> p  
14.    public void addFirst(E e);  
15.    public void addLast(E e);  
16.  
17.    //在给定结点前/后插入新结点  
18.    public void addAfter(Position<E> p,E e) throws InvalidPositionException;  
19.    public void addBefore(Position<E> p,E e) throws InvalidPositionException;  
20.  
21.    //删除给定位置的结点,返回该结点值  
22.    public E remove(Position<E> p) throws InvalidPositionException;  
23.  
24.    //替换某个结点的element并返回旧值  
25.    public E set(Position<E> p, E e) throws InvalidPositionException;  
26. }
```

4.3.4 Doubly Linked List Implementation

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



- Insertion:

```

1. Algorithm insertAfter(p,e) {
2.     Create a new node v;
3.     v.setElement(e);
4.     fucker=p.getNext();
5.     v.setPrev(p);
6.     v.setNext(fucker);
7.     fucker.setPrev(v);
8.     p.setNext(v);
9.     return v;
10. }
```

- Deletion

```

1. Algorithm remove(p) {
2.     pElement=p.element;
3.     pPrev=p.getPrev();
4.     pNext=p.getNext();
5.     pPrev.setNext(pNext);
6.     pNext.setPrev(pPrev);
7.     p.setPrev(null);
8.     p.setNext(null);
9.     return pElement;
10. }
```

4.3.5 实现Position interface ADT

```

1. public class DNode<E> implements Position<E>{
2.     private DNode<E> prev,next;
3.     private E element;
4.
5.     //构造器
6.     public DNode(DNode<E> newPrev,DNode<E> newNext,E elem) {
7.         prev=newPrev;
8.         next=newNext;
9.         element=elem;
```

```

10.     }
11.
12.    public E element() throws InvalidPositionException {
13.        if ((prev == null) && (next == null))
14.            throw new InvalidPositionException("Position is not in a list!");
15.        return element;
16.    }
17.
18.    public DNode<E> getNext() { return next; }
19.    public DNode<E> getPrev() { return prev; }
20.
21.    public void setNext(DNode<E> newNext) { next = newNext; }
22.    public void setPrev(DNode<E> newPrev) { prev = newPrev; }
23.    public void setElement(E newElement) { element = newElement; }
24. }

```

4.3.6 实现NodePositionList ADT

```

1.  public class NodePositionList<E> implements PositionList<E>{
2.      protected int numElements;
3.      protected DNode<E> header,trailer;//special sentinels
4.
5.      //空list构造器
6.      public NodePositionList(){
7.          numElements=0;
8.          header=new DNode<E>(null,null,null);
9.          trailer=new DNode<E>(header,null,null);
10.         header.setNext(trailer);
11.     }
12.
13.     //检查position的合理性:valid->convert to Node
14.     protected DNode<E> checkPosition(Position<E> p) throws
15.     InvalidPositionException{
16.         if (p==null) throw new InvalidPositionException("Null position");
17.         if (p==header) throw new InvalidPositionException("Special sentinel is no
t a valid position");
18.         if (p==trailer) throw new InvalidPositionException("Special sentinel is n
ot a valid position");
19.
20.         try{//注意p->temp为narrowing->显式转换
21.             DNode<E> temp=(DNode<E>) p;
22.             if((temp.getPrev()==null||temp.getNext ()==null))
23.                 throw new InvalidParameterException("not belong to the NodeList"
);
24.             return temp;
25.         } catch(ClassCastException e){
26.             throw new InvalidPositionException("wrong type");
27.         }
28.     }

```

```
29.         //basic methods
30.     public int size(){return numElements;}
31.     public boolean isEmpty(){return (numElements==0);}
32.
33.     //the first position in the list
34.     public Position<E> first() throw EmptyListException{
35.         if(isEmpty()) throw new EmptyListException("empty list");
36.         return header.getNext();
37.     }
38.
39.     //return the position before the given one
40.     public Position<E> prev(Position<E> p) throws InvalidPositionException,Bound
41.     aryViolationException{
42.         DNode<E> v=checkPosition(p);
43.         DNode<E> prev=v.getPrev();
44.         if(prev==header) throw new BoundaryViolationException("Can't advance pas
45.         t the beginning of the list");
46.         return prev;
47.     }
48.
49.     //在指定位置前面插入元素
50.     public void addBefore(Position<E> p,E element) throws
51.     InvalidPositionException{
52.         DNode<E> v=checkPosition(p);
53.         numElements++;
54.         DNode<E> newNode=new DNode<E>(v.getPrev(),v,element);
55.         v.getPrev().setNext(newNode);
56.         v.setPrev(newNode);
57.
58.     //头部插入
59.     public void addFirst(E element){
60.         numElements++;
61.         DNode<E> newNode=new DNode<E>(header,header.getNext(),element);
62.         header.getNext().setPrev(newNode);
63.         header.setNext(newNode);
64.
65.     //指定位置删除，并返回被删除的值
66.     public E remove(Position<E> p) throws InvalidPositionException {
67.         DNode<E> v = checkPosition(p);
68.         numElements--;
69.         DNode<E> vPrev = v.getPrev();
70.         DNode<E> vNext = v.getNext();
71.         vPrev.setNext(vNext);
72.         vNext.setPrev(vPrev);
73.         E vElement = v.element();
74.         v.setNext(null);
75.         v.setPrev(null);
76.         return vElement;
```

```

76.     }
77.
78.     //指定位置替换，并返回原来的值
79.     public E set(Position<E> p, E element) throws InvalidPositionException{
80.         DNode<E> v=checkPosition(p);
81.         E oldElement=v.element();
82.         v.setElement(element);
83.         return oldElement;
84.     }
85. }

```

- 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()

```

1.     if (hadNext ()) {
2.         next ();
3.     }

```

4.4.1 Simple Iterators in Java

```

1.     public interface PositionList <E> extends Iterable <E>{
2.         //all the other methods of the list ADT
3.         public Iterator<E> iterator();
4.     }
5.
6.
7.     public static <E> String toString (PositionList<E> l) {
8.         Iterator<E> it=l.iterator();
9.         String s="[";
10.        while (it.hasNext()){
11.            s+= it.next();

```

```
12.         if(it.hasNext())
13.             s+= ", ";
14.     }
15.     s+="]";
16.     return s;
17. }
```

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)

```

1. Algorithm preOrder (v) {
2.     visit (v); //先访问当前结点v
3.     for each child w of v:
4.         preOrder (w); //使用同样办法遍历v的子树
5. }
```

- 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

```

1. Algorithm postOrder (v) {
2.     for each child w of v:
3.         postOrder (w);
```

```
4.         visit(v);
5.     }
```

- 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

```
1.  /*宽度优先遍历树
2.   * 优先遍历深度相同的结点
3.   * 基于queue:每次先访问这一层的所有结点 (dequeue)
4.   * 访问一个结点同时将其所有子节点加入队尾
5.   * 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
6.   */
7.
8. import java.util.LinkedList;
9. import java.util.Queue;
10. public class BreadthFirstTreeTraversal {
11.     Initialize queue Q storing root();
12.     while (!Q.isEmpty()){
13.         p=Q.dequeue(); //p is the oldest in the queue
14.         visit(p);
15.         for each child in children(p){
16.             Q.enqueue(c);
17.         }
18.     }
19. }
```

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

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

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

- 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

```

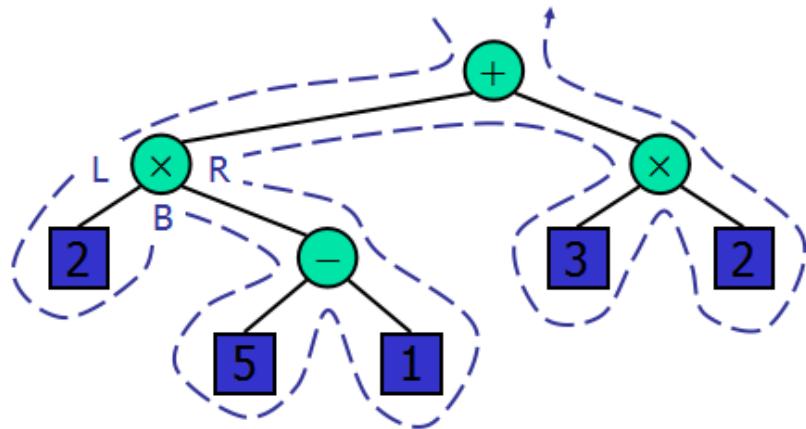
1. Algorithm inOrder(v) {
2.     if (hasLeft(v))
3.         inOrder(left(v));
4.     visit(v);
5.     if (hasRight(v))
6.         inOrder(right(v));
7. }
```

- 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

```

1.  public abstract class EulerTour {
2.      protected BinaryTree tree;
3.      protected void visitExternal(Position p, Result r) {}
4.      protected void visitLeft(Position p, Result r) { }
5.      protected void visitBelow(Position p, Result r) { }
6.      protected void visitRight(Position p, Result r) { }
7.
8.      protected Object eulerTour(Position p) {
9.          Result r = new Result();
10.         if tree.isExternal(p) { visitExternal(p, r); }
11.         else {
12.             visitLeft(p, r);
13.             r.leftResult = eulerTour(tree.left(p));
14.             visitBelow(p, r);
15.             r.rightResult = eulerTour(tree.right(p));
16.             visitRight(p, r);
17.             return r.finalResult;
18.         }
19.         ...
20.     }
21. }
  
```

- Specializations of EulerTour, 基于以下假定

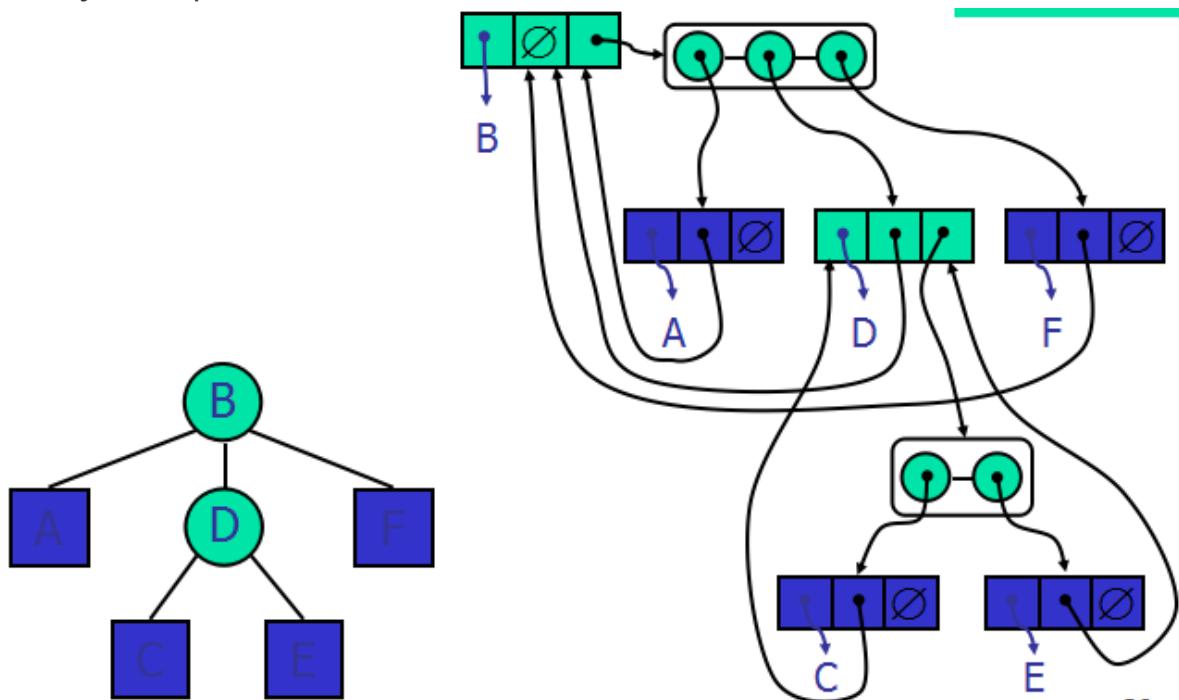
- External nodes store Integer objects
- Internal nodes store Operator objects supporting method operation(Integer, Integer)

```

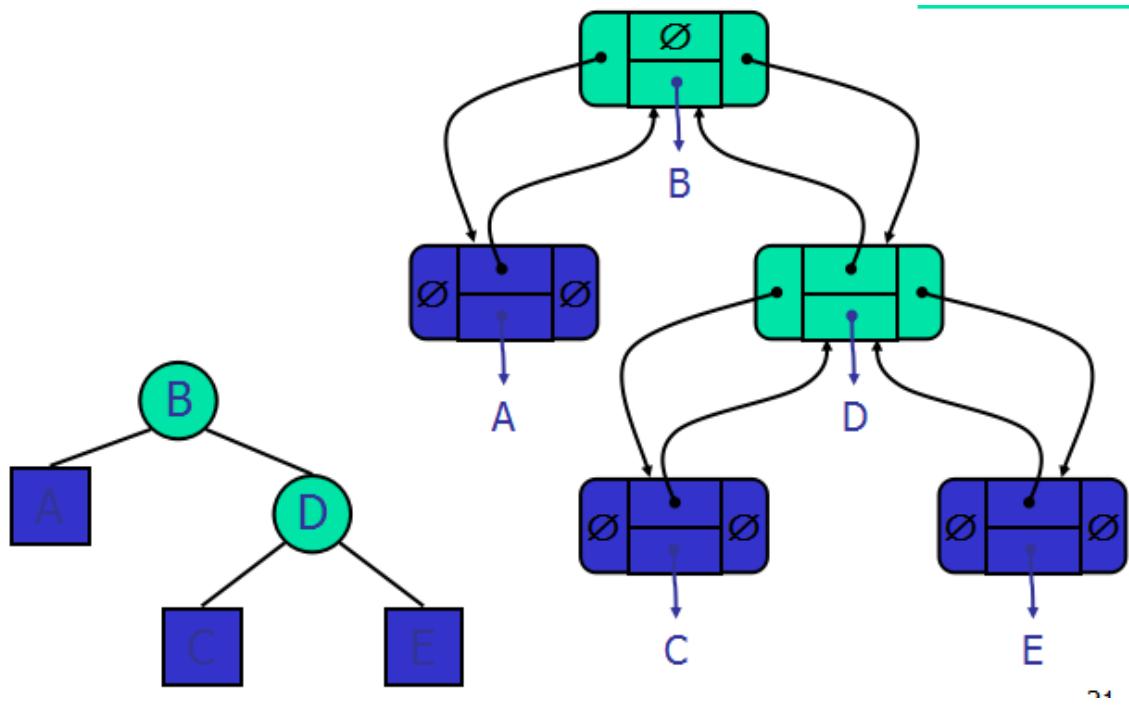
1. public class EvaluateExpression extends EulerTour{
2.     protected void visitExternal(Position p,Result r) {
3.         r.finalResult=(Integer)p.element();
4.     }
5.     protected void visitRight(Position p,Result r) {
6.         Operator op=(Operator)p.element();
7.         r.finalResult=op.operation((Integer)r.leftResult,(Integer)r.rightResult)
8.     }
9. }
```

5.5 Linked Structure for Trees

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

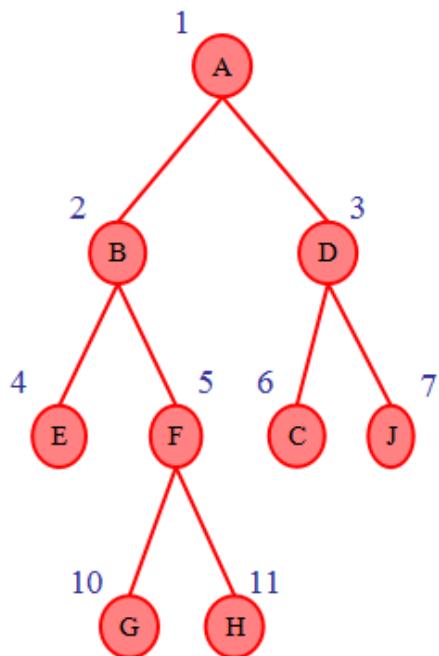


- Linked Structure for Binary Trees
 - 和前面的对比就是子结点被严格限制为左+右



- Array-Based Representation of Binary Trees

- $\text{rank}(\text{root}) = 1$
- $\text{rank}(\text{left node}) = 2 * \text{rank}(\text{parent})$
- $\text{rank}(\text{right node}) = 2 * \text{rank}(\text{parent}) + 1$



- 构建基于链表的二叉树,并实现前中后序遍历

```

1.  /*
2.   * Ref: http://ocaicai.iteye.com/blog/1047397
3.   * 创建基于链表的二叉树,并实现前序中序后序遍历
4.   * 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

```

```

5.     */
6.
7. package tree;
8. import java.util.LinkedList;
9. import java.util.List;
10.
11. public class LinkedBinaryTree2 {
12.     private static class Node {
13.         Node leftChild;
14.         Node rightChild;
15.         int data;
16.
17.         Node(int newData) {
18.             leftChild = null;
19.             rightChild = null;
20.             data = newData;
21.         }
22.     }
23.     private static List<Node> nodeList=null;
24.     protected Node root=null;
25.     public void buildBinaryTree(int[] myData) {
26.         nodeList=new LinkedList<Node>();
27.         //将之前的数据全都存到结点里,再将结点存入到nodeList中
28.         for(int nodeIndex=0;nodeIndex<myData.length;nodeIndex++) {
29.             Node myNode=new Node(myData[nodeIndex]);
30.             nodeList.add(myNode);
31.         }
32.
33.         //使用nodeList构建树,相当于给每个父结点都添加子节点
34.         //首先构建除了最后一个父结点外的internal
35.         //假设total数量为n, internal数量为(n-1)/2
36.         //暂时排除最后一个父结点-->构建n/2-1个父结点
37.         for(int parentIndex=0;2*parentIndex+2<myData.length;parentIndex++) {
38.             int leftIndex=2*parentIndex+1;
39.             int rightIndex=2*parentIndex+2;
40.             nodeList.get(parentIndex).leftChild=nodeList.get(leftIndex);
41.             nodeList.get(parentIndex).rightChild=nodeList.get(rightIndex);
42.         }
43.         //考虑最后一个父结点的两种情况
44.         //若元素数量为奇数则最后一个父结点是满的
45.         //若元素数量为偶数则最后一个父结点只有左子结点
46.         int lastParentIndex=myData.length/2-1;
47.         int leftIndex=2*lastParentIndex+1;
48.         nodeList.get(lastParentIndex).leftChild=nodeList.get(leftIndex);
49.         if (myData.length%2==1) {
50.             int rightIndex=2*lastParentIndex+2;
51.             nodeList.get(lastParentIndex).rightChild=nodeList.get(rightIndex);
52.         }
53.
54.         //最后给root赋值

```

```
55.         root=nodeList.get(0) ;
56.     }
57.
58.     //PreorderTraversal
59.     public static void PreorderTraversal(Node currentNode) {
60.         if(currentNode==null) {
61.             return;
62.         }
63.         System.out.print(currentNode.data+" ");
64.         PreorderTraversal(currentNode.leftChild);
65.         PreorderTraversal(currentNode.rightChild);
66.     }
67.
68.     //InorderTraversal
69.     public static void InorderTraversal(Node currentNode) {
70.         if(currentNode==null) {
71.             return;
72.         }
73.         InorderTraversal(currentNode.leftChild);
74.         System.out.print(currentNode.data+" ");
75.         InorderTraversal(currentNode.rightChild);
76.     }
77.
78.     //PostorderTraversal
79.     public static void PostorderTraversal(Node currentNode) {
80.         if(currentNode==null) {
81.             return;
82.         }
83.         PostorderTraversal(currentNode.leftChild);
84.         PostorderTraversal(currentNode.rightChild);
85.         System.out.print(currentNode.data+" ");
86.     }
87.
88.     //测试用例
89.     public static void main(String[] args) {
90.         LinkedBinaryTree2 myLBT1=new LinkedBinaryTree2();
91.         LinkedBinaryTree2 myLBT2=new LinkedBinaryTree2();
92.         int[] myData1={1,2,3,4,5,6,7,8,9};
93.         myLBT1.buildBinaryTree(myData1);
94.         System.out.println("Preorder of LBT1 is: ");
95.         myLBT1.PreorderTraversal(myLBT1.root);
96.         System.out.println("");
97.         System.out.println("Inorder of LBT1 is: ");
98.         myLBT1.InorderTraversal(myLBT1.root);
99.         System.out.println("");
100.        System.out.println("Postorder of LBT1 is: ");
101.        myLBT1.PostorderTraversal(myLBT1.root);
102.
103.        int[] myData2={2,3,5,7,11,13,17,19,23,29,31,33,37,43,47,53,59};
104.        myLBT2.buildBinaryTree(myData2);
```

```

105.         System.out.println("Preorder of LBT2 is: ");
106.         myLBT2.PreorderTraversal(myLBT2.root);
107.         System.out.println("");
108.         System.out.println("Inorder of LBT2 is: ");
109.         myLBT2.InorderTraversal(myLBT2.root);
110.         System.out.println("");
111.         System.out.println("Postorder of LBT2 is: ");
112.         myLBT2.PostorderTraversal(myLBT2.root);
113.     }
114. }

```

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
- $\text{key}(u) \leq \text{key}(v) \leq \text{key}(w)$
- 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 : $O(\log n)$
- 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;

```

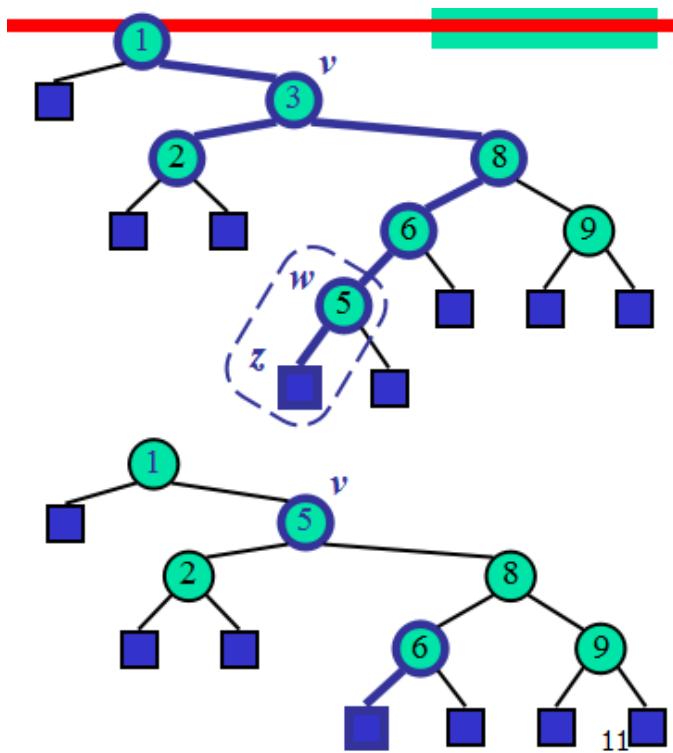
1. Algorithm TreeSearch(k,v) { //Start from node v
2.     if(T.isExternal(v))
3.         return v;
4.     if(k<key(v))
5.         return TreeSearch(k,T.left(v));
6.     else if(k>key(v))
7.         return TreeSearch(k,T.right(v));
8.     else
9.         return v; //Match
10.    return false; //No match
11. }
```

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:
 - successor : 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) : $O(\log n)$

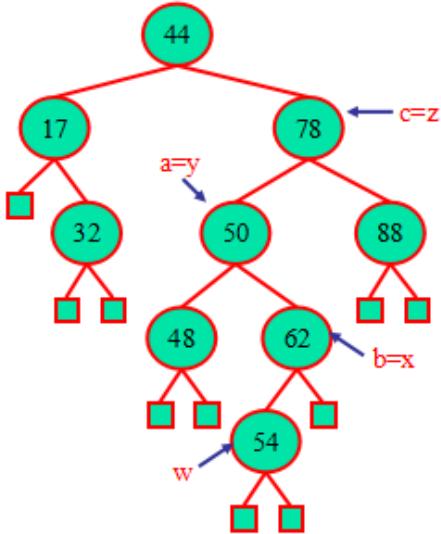
6.2 AVL Trees

- Balanced : 任何结点的两棵子树高度差不超过1
- AVL高度必然为 $O(\log n)$, 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 $n(h) > 2n(h - 2)$
 - $n(h) > 2n(h - 2) > 4n(h - 4) > 8n(h - 6) > \dots > 2^i n(h - 2i)$
 - 令 $i = h/2 - 1$, 得到 $n(h) > 2^{\frac{h}{2}-1}$, 即 $h < 2\log n(h) + 2$

6.2.1 Insertion in an AVL Tree

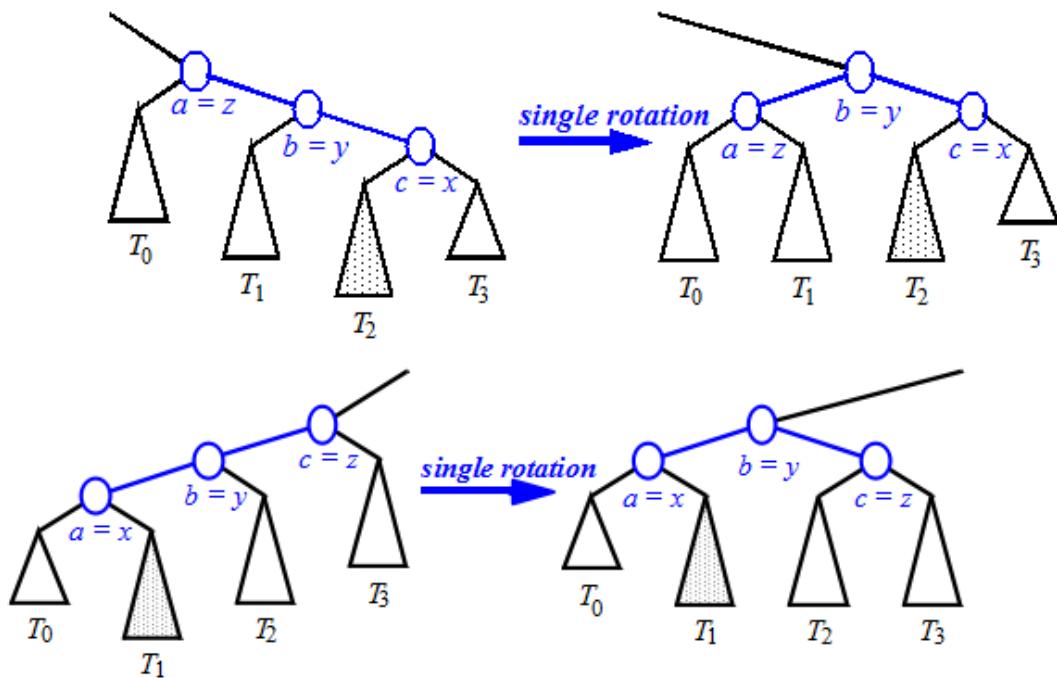
- 插入操作类似BST, 为了防止平衡被破坏, 需要进行Trinode Restructuring
- Trinode Restructuring :

- 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:

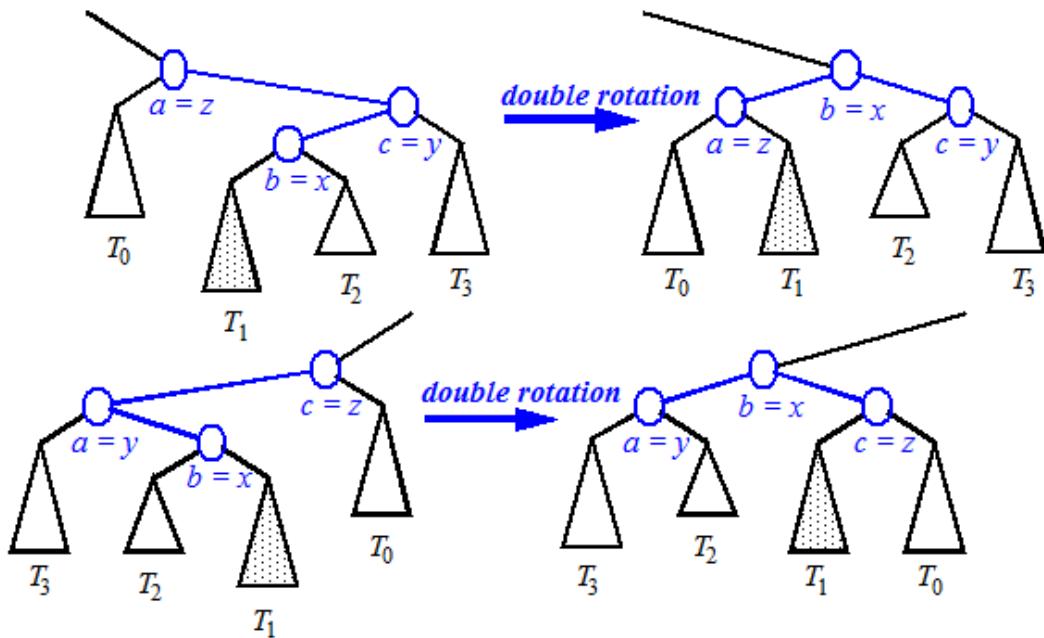
- single rotation :
 - 同侧结点 $x < y < z$ 或者 $x > y > z$;
 - 中间的y结点变为x,z的父结点;



- double rotation :

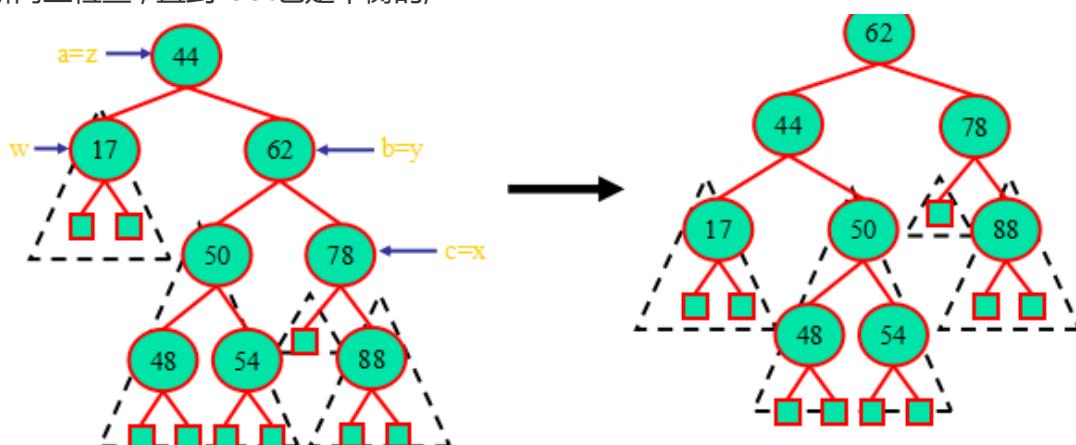
- 一左一右

- 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

```

1. public class AVLTreeMap<K, V> extends TreeMap<K, V>{
2.     //两个构造器
3.     public AVLTreeMap () {
4.         super ();

```

```

5.     }
6.     public AVLTreeMap (Comparator<K> comp) {
7.         super (comp) ;
8.     }
9.
10.    //method 1: 返回位置p的高度
11.    protected int height (Position<Entry<K,V>> p) {
12.        return tree.getAux (p) ;
13.    }
14.
15.    //method 2: 重新计算指定位置的高度
16.    //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
17.    protected void recomputeHeight (Position<Entry<K,V>> p) {
18.        tree.setAux (1+Math.max (height (left (p) ,height (right (p) ))));
19.    }
20.
21.    //method 3: 判断当前位置是否平衡
22.    protected boolean isBalanced (Position<Entry<K,V>> p) {
23.        return Math.abs (height (left (p) )-height (right (p) ))<=1;
24.    }
25.
26.    //method 4: 返回位置p两棵子树高度比较大的那棵
27.    protected Position<Entry<K,V>> tallerChild (Position<Entry<K,V>> p) {
28.        if (height (left (p) )>height (right (p) )) {
29.            return left (p) ;
30.        }
31.        if (height (left (p) )<height (right (p) )) {
32.            return right (p) ;
33.        }
34.        //接下来是p两棵子树高度相同的情况
35.        if (isRoot (p) ){//返回左右都一样
36.            return left (p) ;
37.        }
38.        //下面这个判断还不大清楚,为什么要这么搞呢
39.        if (p==left (parent (p) )){//p是其父结点的左子树
40.            return left (p) ;
41.        }
42.        else{//p是其父结点的右子树
43.            return right (p) ;
44.        }
45.    }
46.
47.    //method 5: 对于不平衡的AVL,重新将其回复到平衡状态
48.    protected void rebalance (Position<Entry<K,V>> p) {
49.        int oldHeight,newHeight;
50.        //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
51.        do{
52.            oldHeight=height (p) ;
53.            //首先先判断当前结点p是否平衡
54.            if (!isBalanced (p) ){//p位置发生不平衡

```

```

55.         //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
56.         p=restructure(tallerChild(tallerChild(p)));
57.         //重新计算p两个子节点的高度
58.         recomputeHeight(left(p));
59.         recomputeHeight(right(p));
60.     }
61.     //若p位置是平衡的，计算p的高度并将p赋值为其父结点
62.     //不断向上检查是否平衡
63.     recomputeHeight(p);
64.     newHeight=height(p);
65.     p=parent(p);
66. }while(oldHeight!=newHeight && p!=null);
67.
68.

69. protected void rebalanceInsert(Position<Entry<K,V>> p) {
70.     rebalance(p); //插入结点，从该结点位置开始判断
71. }
72. protected void rebalanceDelete(Position<Entry<K,V>> p) {
73.     if(!isRoot(p)) {
74.         //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
75.         rebalance(parent(p));
76.     }
77. }
78.
79. }

```

6.2.4 AVL Tree的性能

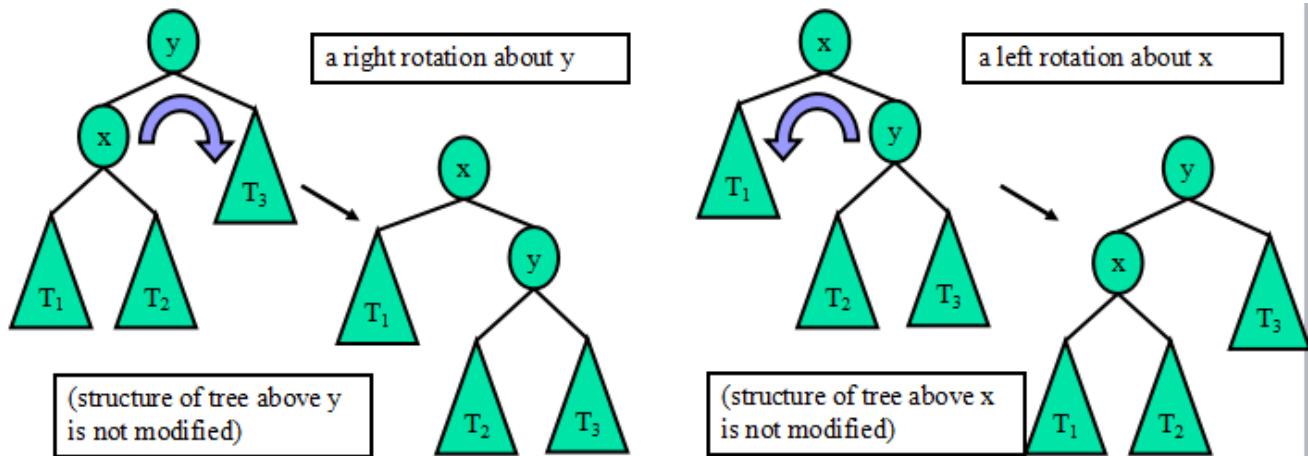
- 一次重构 $O(1)$
- 查找 $O(\log n)$
- 插入(查找+重构) $O(\log n)$
- 删除(查找+重构) $O(\log n)$

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;

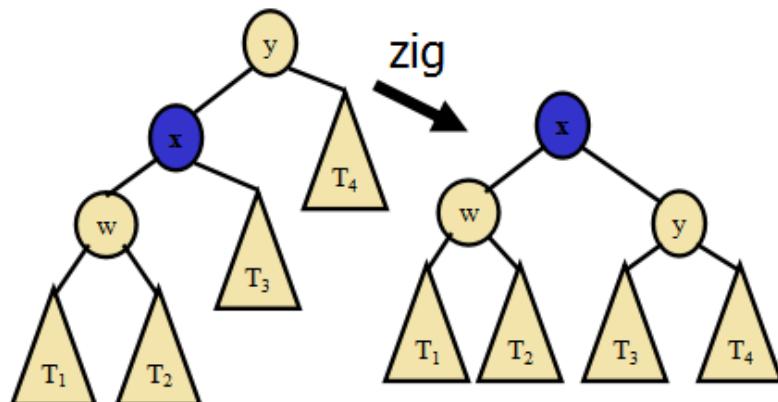
6.3.1 splay操作 : 通过不断旋转将指定结点传送到root位置

- 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

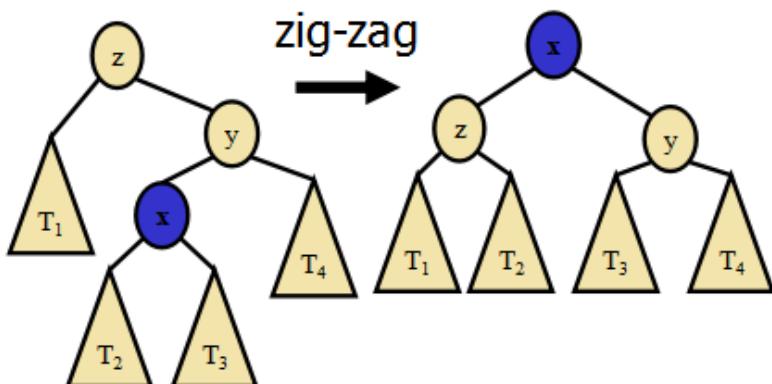


6.3.2 zig-zag过程

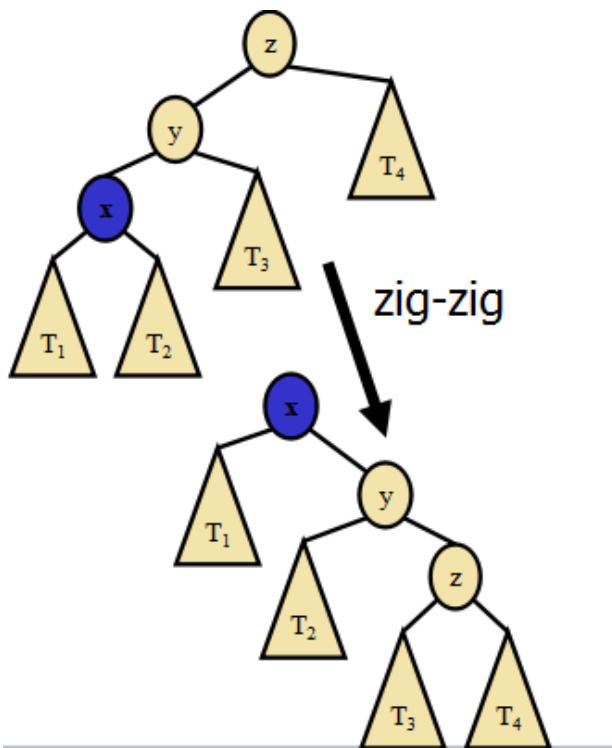
- zig : 目标结点是根结点子结点时, 进行一次单旋转, 使目标结点变为根结点



- 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



```

1. private void splay(Position<Entry<K, V>> p) {
2.     while (!isRoot(p)) {
3.         //首先得到p的父亲和祖父进行比较
4.         Position<Entry<K, V>> parent=parent(p);
5.         Position<Entry<K, V>> grandParent=parent(parent);
6.
7.         //case 1: parent是根结点--> zig
8.         if (grandParent==null) {
9.             rotate(p);
10.        }
11.        //case 2: 左左结构--> zig-zig
12.        else if (parent==left(grandParent) && p==left(parent)) {
13.            rotate(parent);
14.            rotate(p);
15.        }
16.        //case 3: 右右结构--> zig-zig
17.        else if (parent==right(grandParent) && p==right(parent)) {
18.            rotate(parent);
19.            rotate(p);
20.        }
21.        //case 4: zig-zag
22.        else{
23.            rotate(p);
24.            rotate(p);
25.        }
26.    }

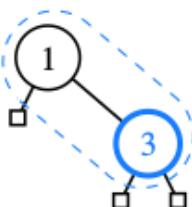
```

6.3.3 选择splay node

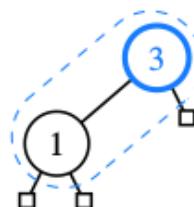
- `find(k):`
 - 找到该结点则从该结点开始splay;
 - 没找到该结点,则从搜索终止位置(叶结点)的父结点开始splay;
- `insert(k,v):`
 - 使用被插入的新结点进行splay;



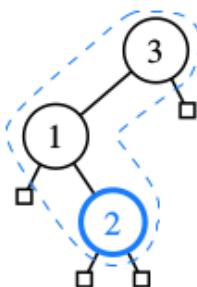
(a)



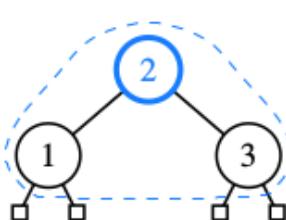
(b)



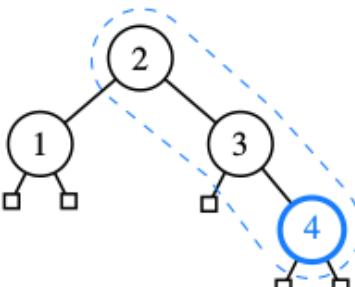
(c)



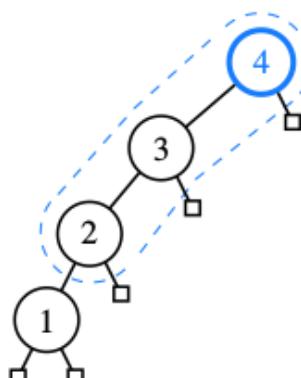
(d)



(e)

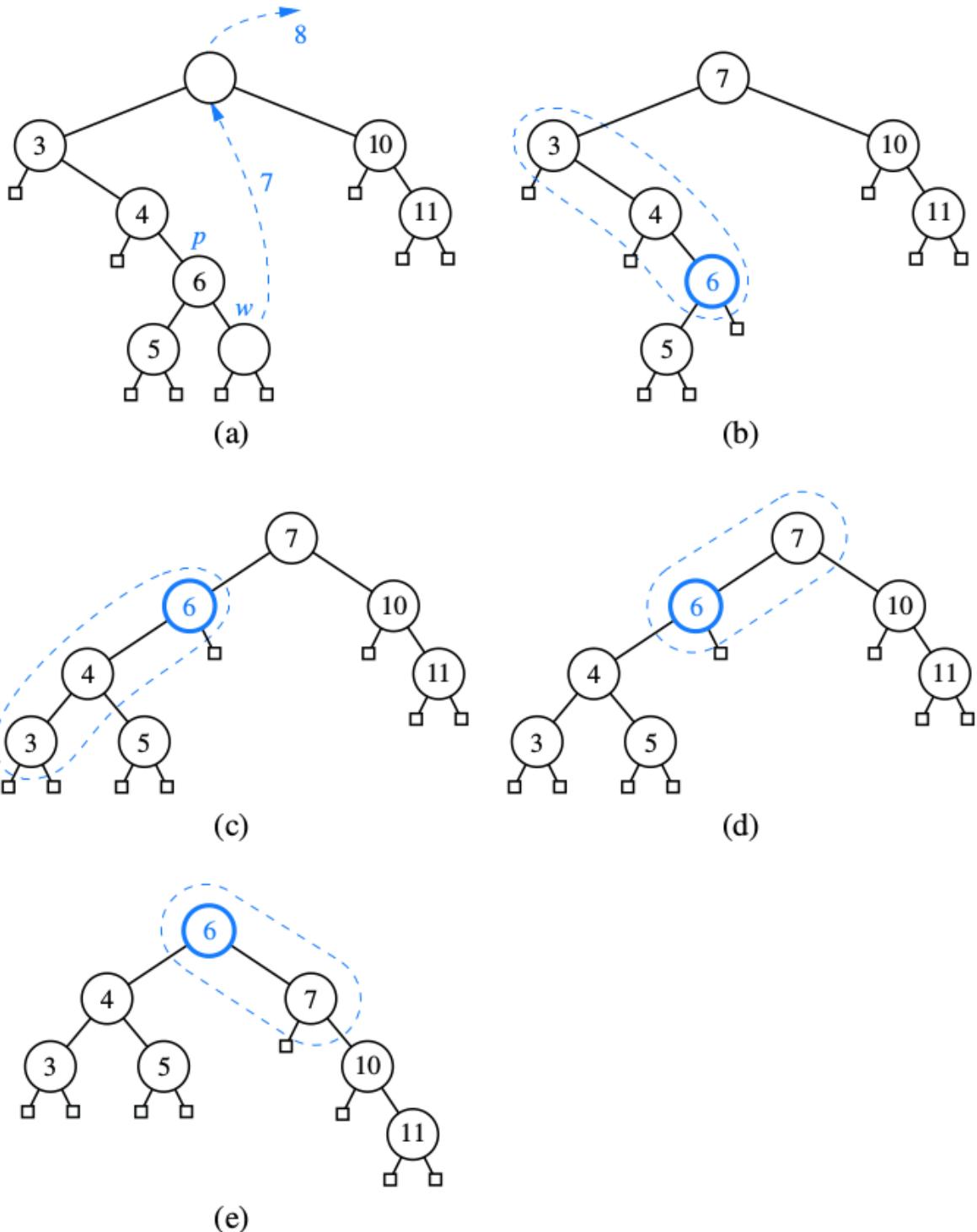


(f)



(g)

- `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



```

1. protected void rebalanceAccess(Position<Entry<K,V>> p) {
2.     if(isExternal(p)) {
3.         p=parent(p);
4.     }
5.     if(p!=null) {
6.         splay(p);
7.     }
8. }
```

```

10.     protected void rebalanceInsert(Position<Entry<K,V>> p) {
11.         splay(p);
12.     }
13.
14.     //注意这里splay 结点为p (w的父结点)
15.     protected void rebalanceDelete(Position<Entry<K,V>> p) {
16.         if (!isRoot(p)) {
17.             splay(parent(p));
18.         }
19.     }

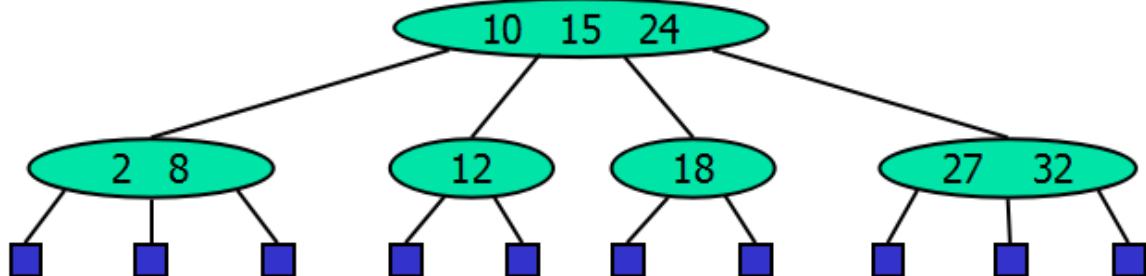
```

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 $O(\log n)$
- 因此伸展树可以保证m次连续搜索仅需要 $m \log n$ 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



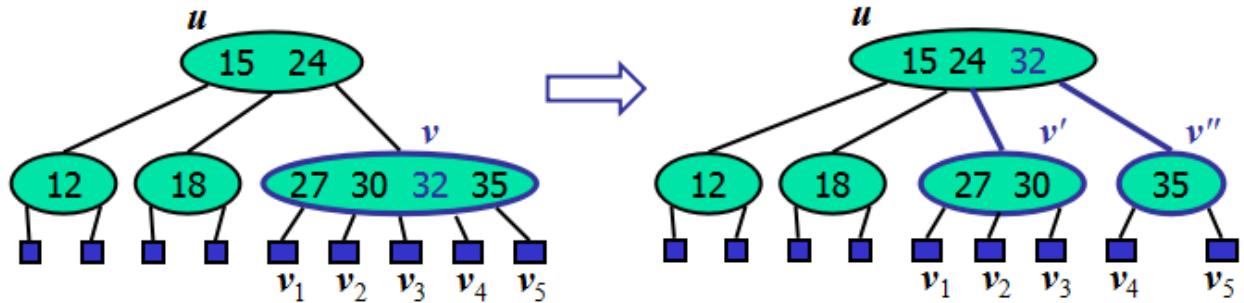
6.4.1 Height of a (2,4) Tree $O(\log n)$

- 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

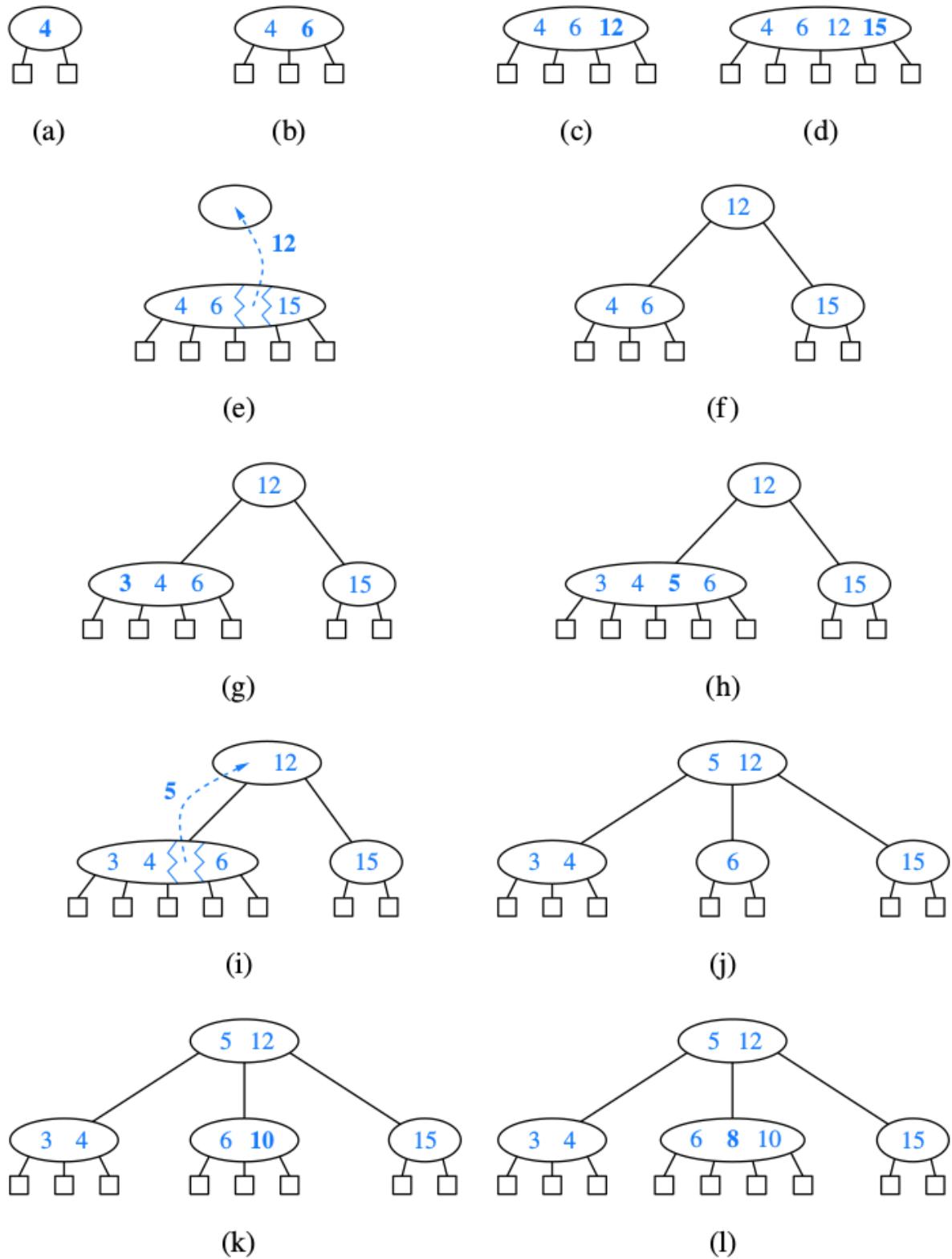


- Insertion + split $O(\log n)$

```

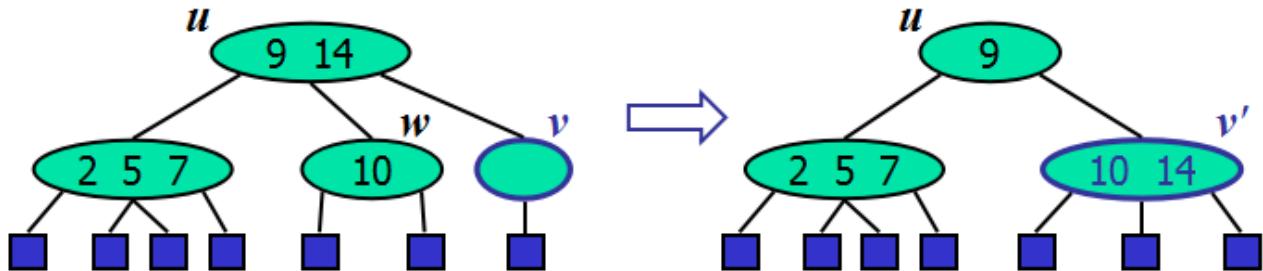
1. Algorithm insert(k, o) {
2.     search for key k to locate the insertion node v;
3.     add the new entry (k, o) at node v;
4.     while ( overflow(v) ) {
5.         if (isRoot(v))
6.             create a new empty root above v;
7.         v = split(v);
8.     }
9. }
```

- 更复杂的栗子:

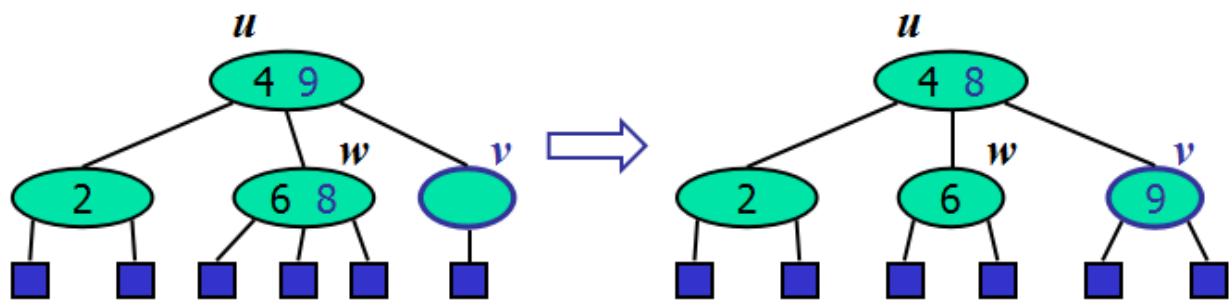


6.4.3 Deletion

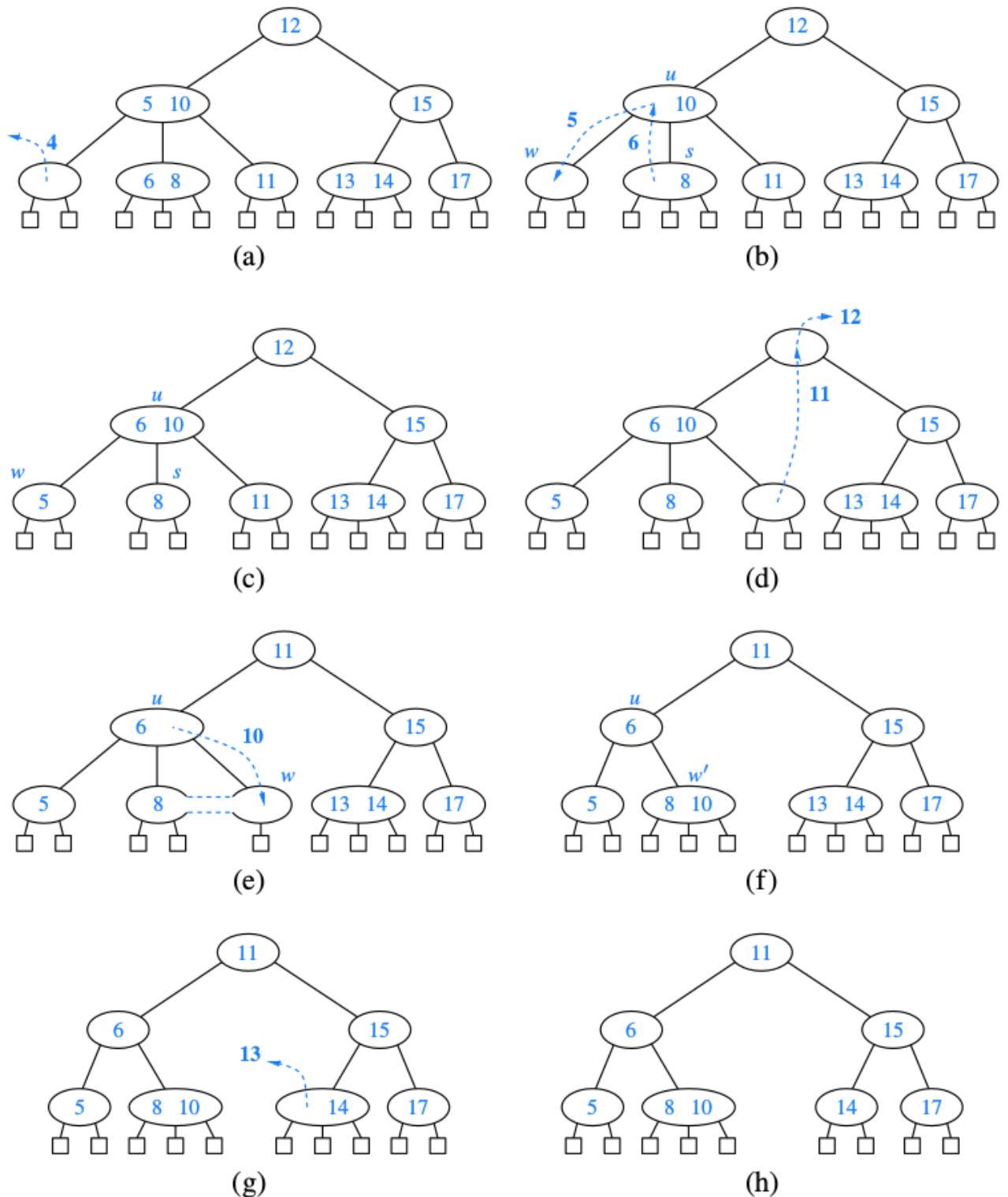
- **删除操作**: 用被删除点的successor(predecessor)替换其位置
- **删除后可能破坏规则(underflow)**, 需要进行fusion/transfer
- **Fusion**:



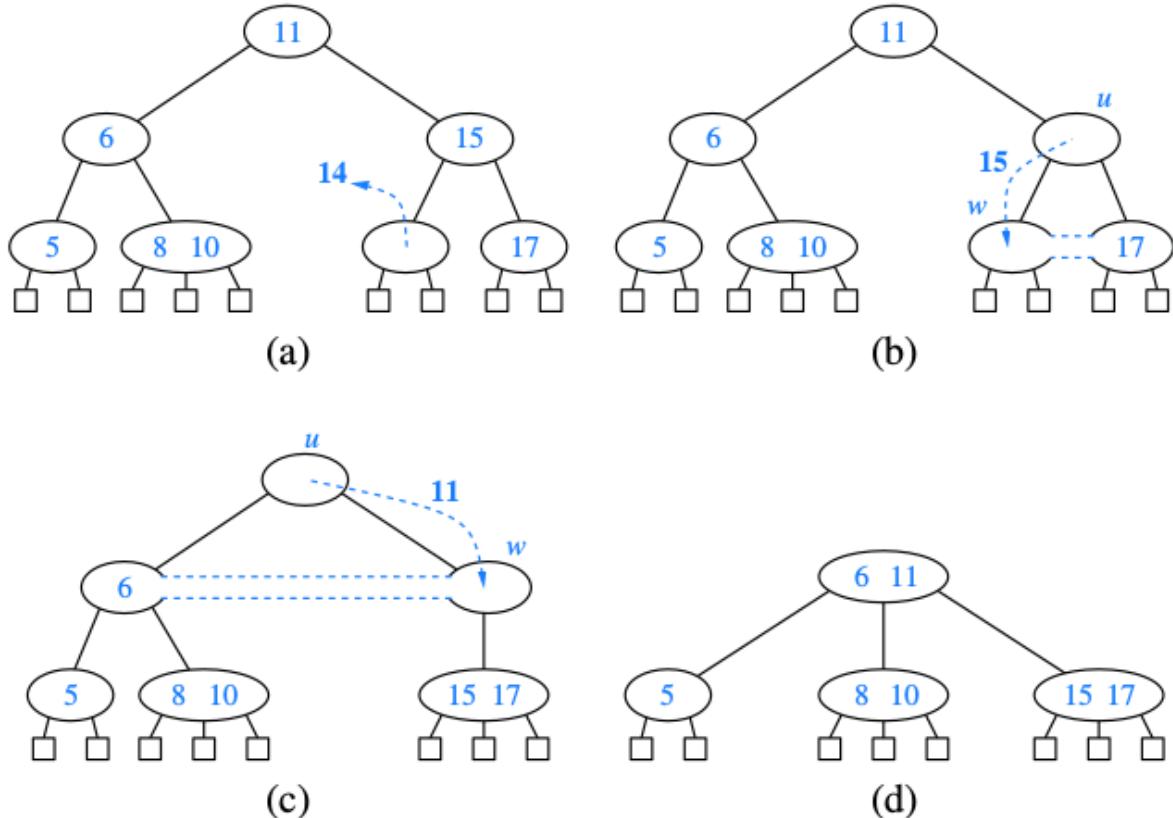
- Transfer:



- 更复杂的栗子:



- 最后一个栗子：多次变换最后高度降低

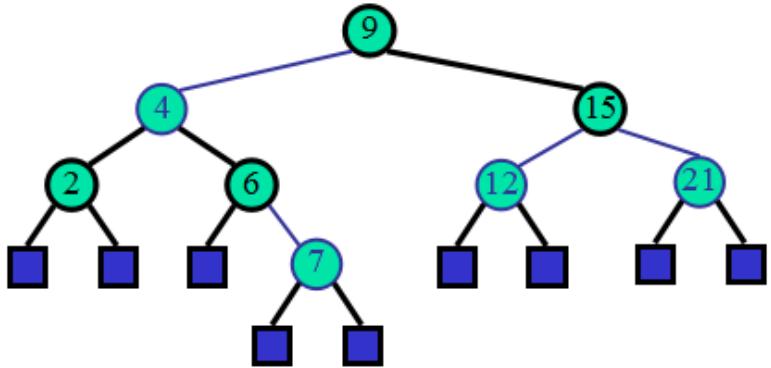


6.4.4 (2,4) Tree 的性能

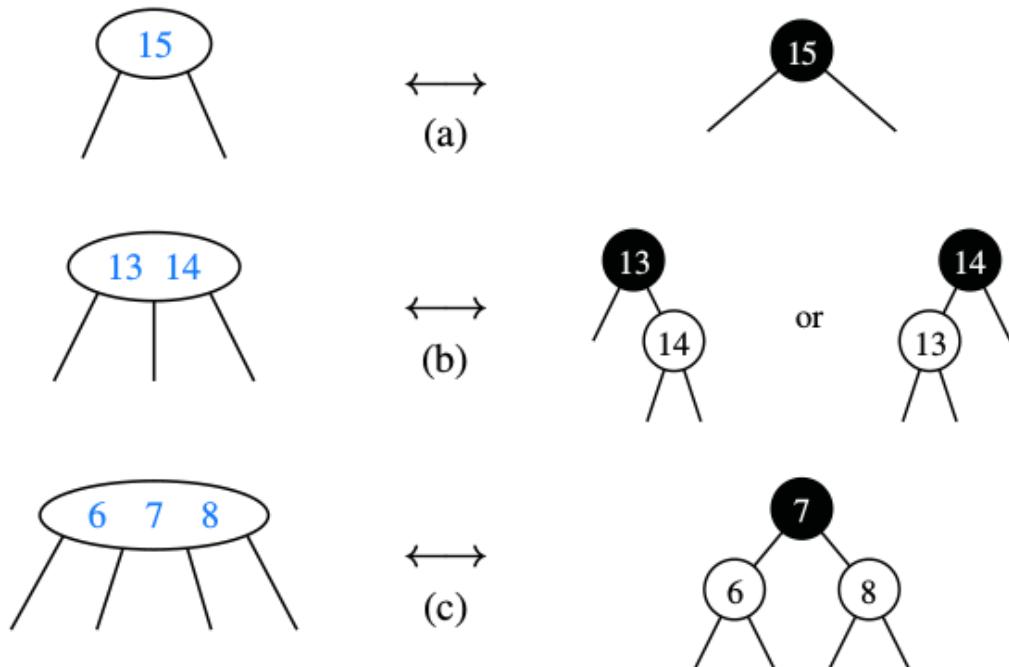
- search/insert/remove $O(\log n)$
- split/fusion/transfer $O(1)$

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



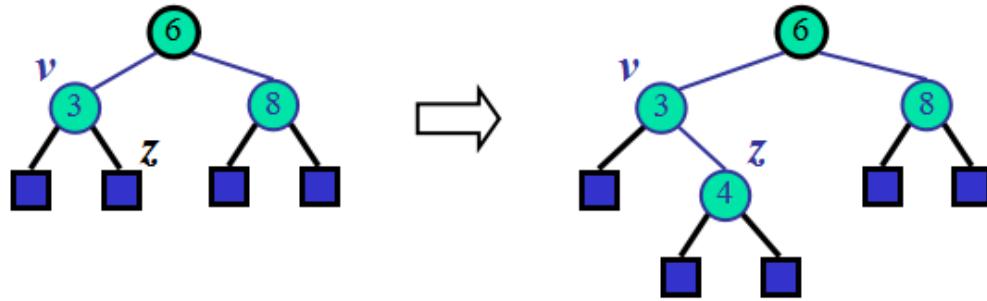
- 红黑树与(2,4)树的比较



6.5.1 Height of a RB Tree: $O(\log n)$

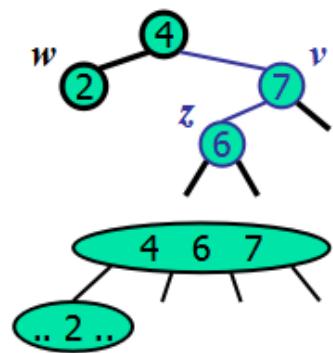
6.5.2 Insertion:

- 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

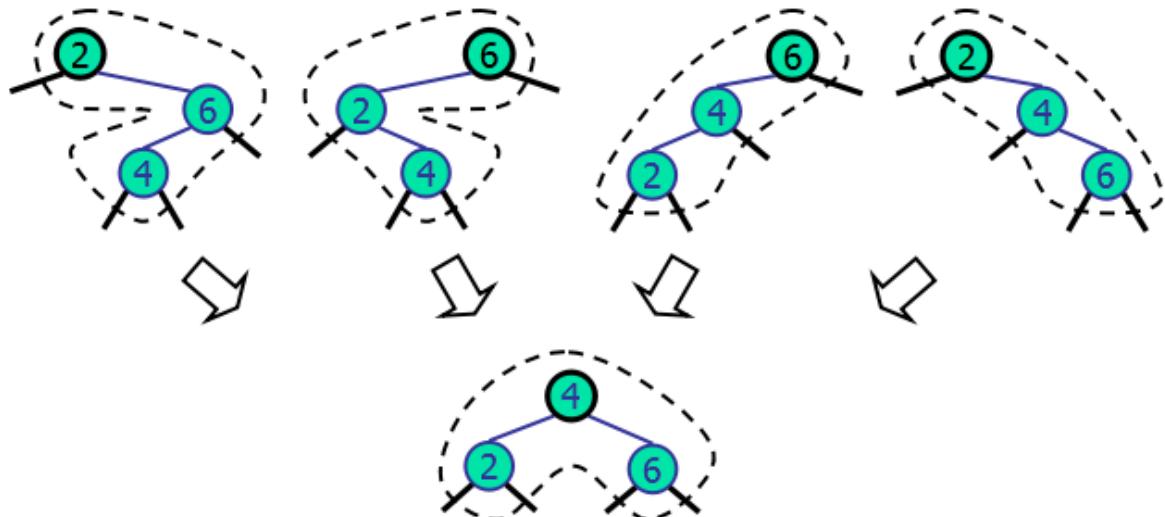


- Case 1: 叔结点w(v同级)是黑的，进行restructuring

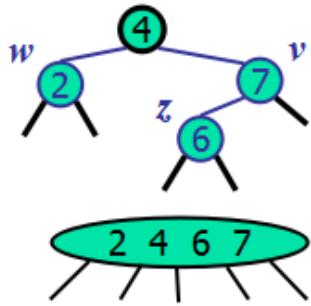
相当于更改4-结点的布局;



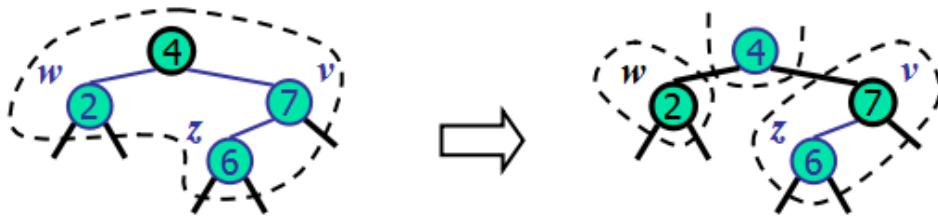
- v>z>x(v为x右,z为v左):z(predecessor)变为x/v的父结点;
- x>z>v(v为x左,z为v右):z(successor),同上;
- x>v>z(两个都是左):v变为x/z的父结点;(单左旋)
- z>v>x(两个都是右):同上;(单右旋)
- 移动完成后进行变色: 新的父结点变为黑色,子结点红色
- 总之就是找这三个点的中间点变为另两个点的父结点



- 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



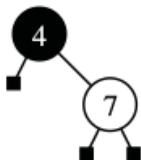
```

1. Algorithm insert(k, o){
2.     search(k) ->确定插入点z;
3.     将(k,o)插入z处,并将该点设定为红色(只要是插入就是红);
4.
5.     while doubleRed(z){ //z和其父结点都是红的
6.         if ( isBlack(sibling(parent(z)))) { //先确定叔结点w的颜色
7.             z = restructure(z); //w黑色->四种重构
8.             return;
9.         }
10.        else // w红色->recoloring->wv黑,xz红
11.            z = recolor(z);
12.    }
13. }
```

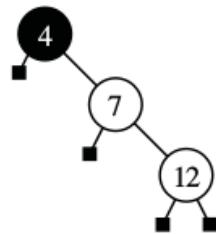
- 栗子: 构建一棵红黑树



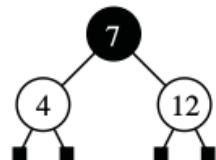
(a)



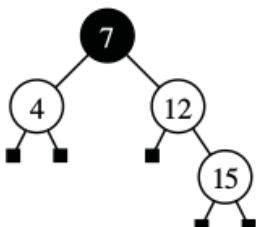
(b)



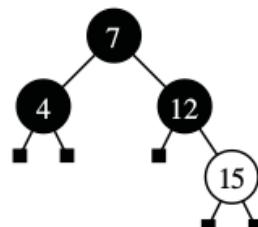
(c)



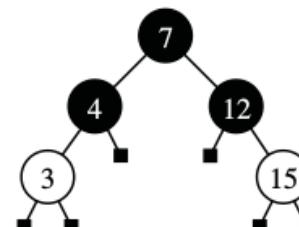
(d)



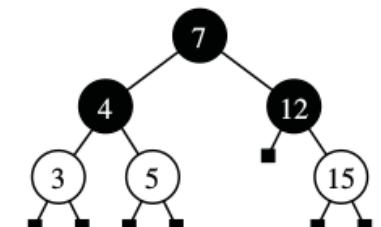
(e)



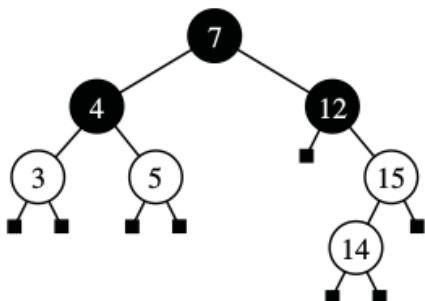
(f)



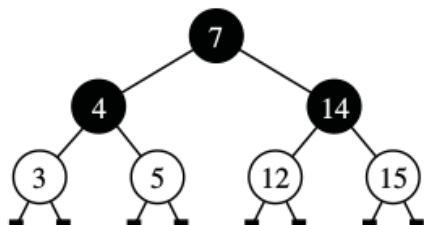
(g)



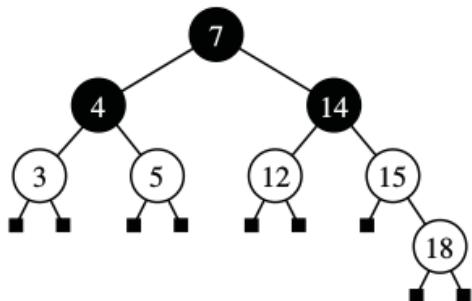
(h)



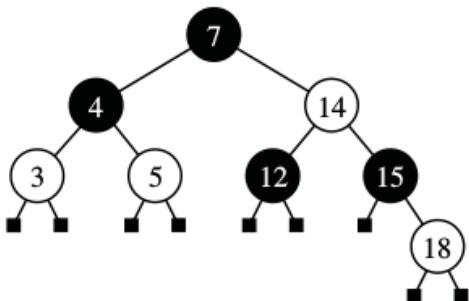
(i)



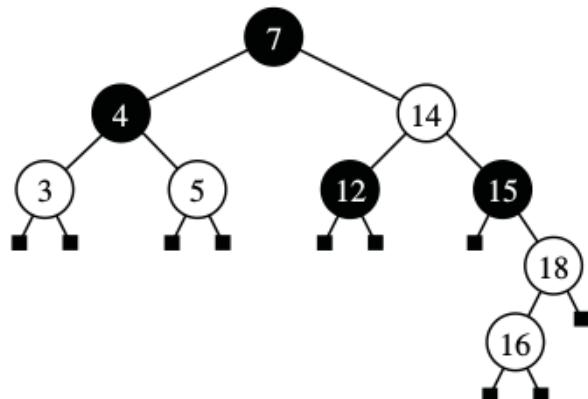
(j)



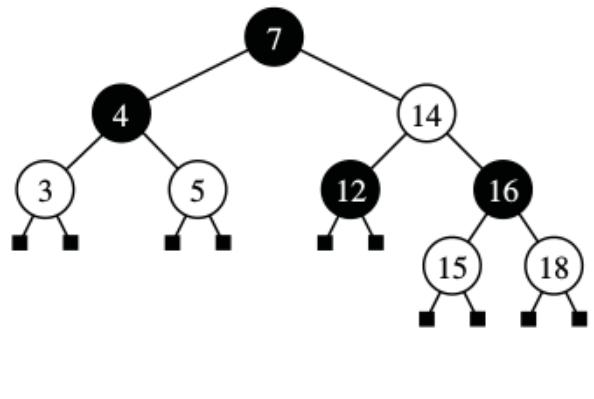
(k)



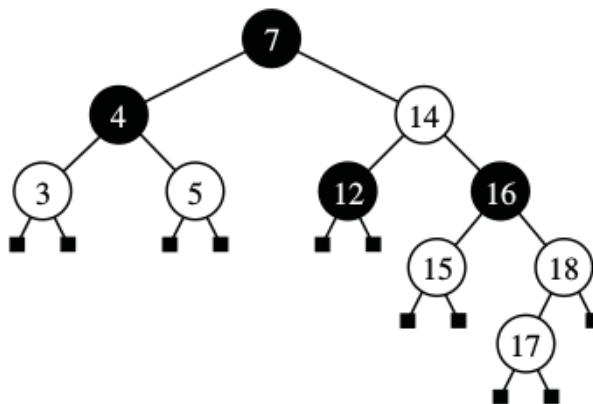
(l)



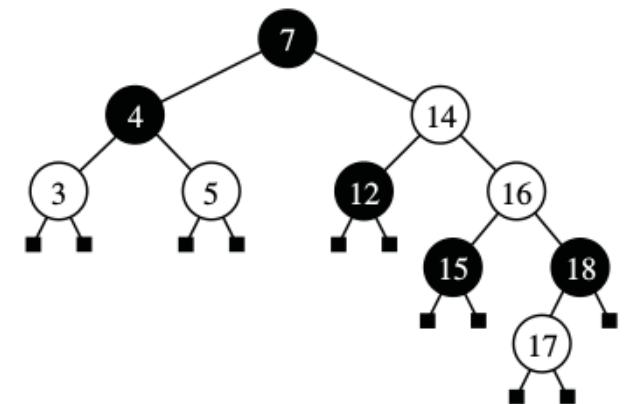
(m)



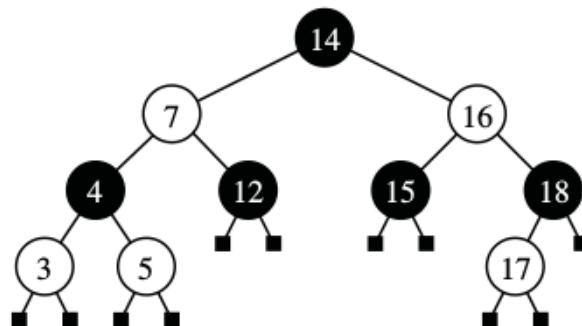
(n)



(o)



(p)

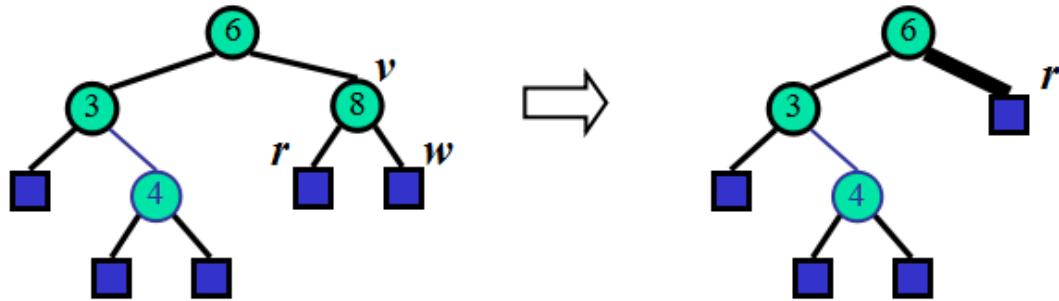


(q)

6.5.3 Deletion

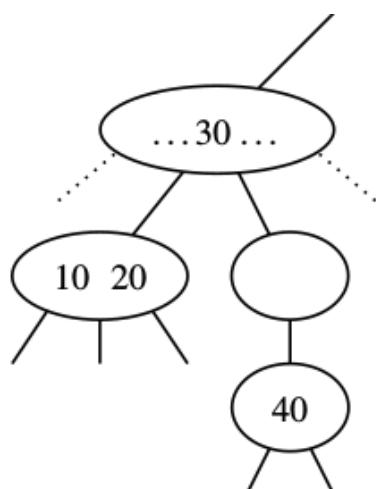
- 先命名几个目标结点:
 - v : 指定被删除的internal node;
 - w : 指定被删除的external node,v的子结点;
 - r : w的兄弟结点,v的另一个子结点;
- 如果v/r至少一个是红的 , r变成黑色替换到v的位置

- 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

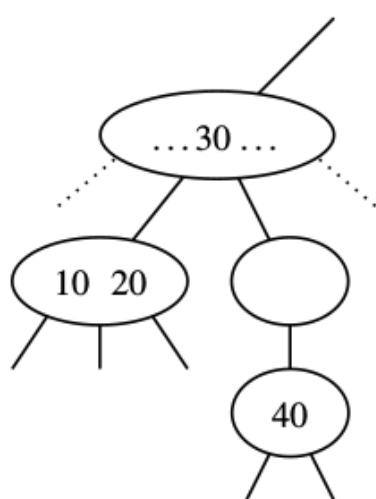
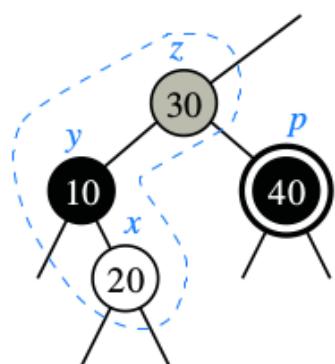


- reorganization 的几种情况

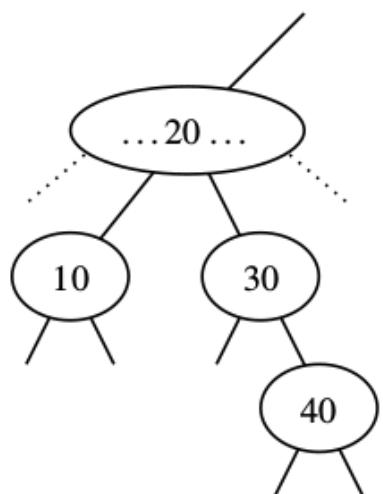
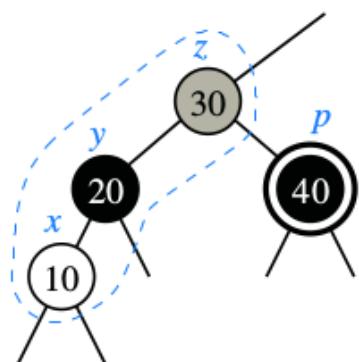
- Case 1 : r黑 , r有一个红子结点
 - 对r以及其子结点进行restructuring(等价于transfer);



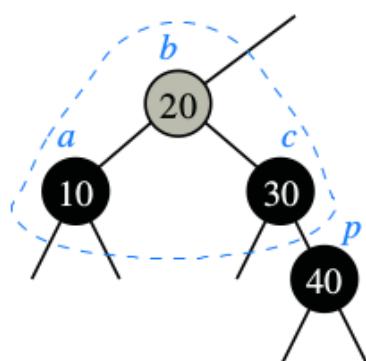
(a)



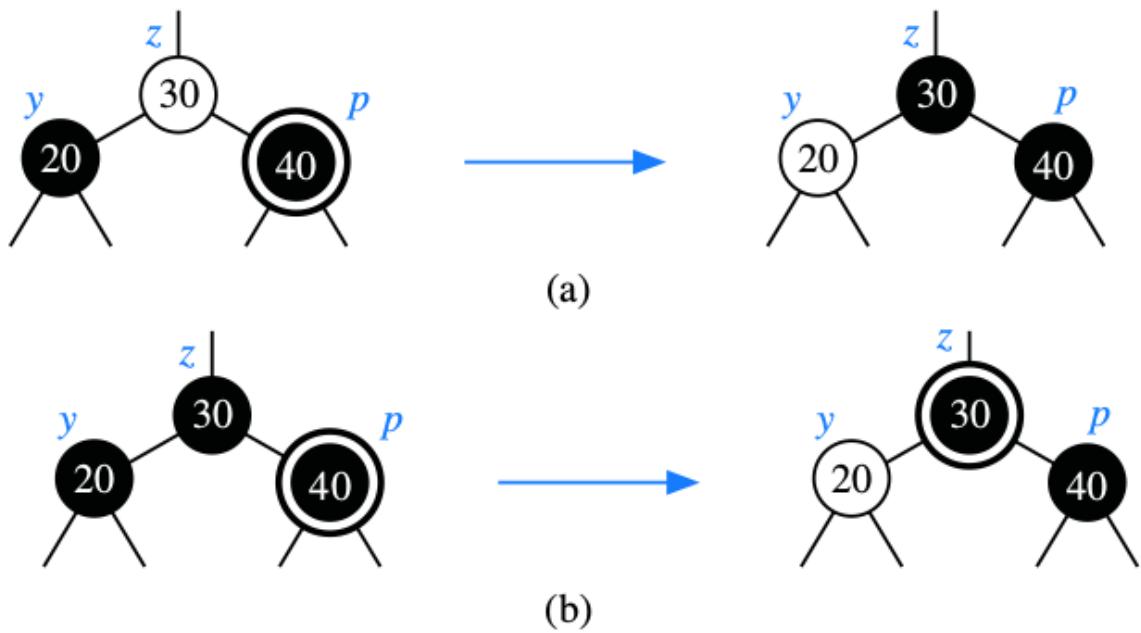
(b)



(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

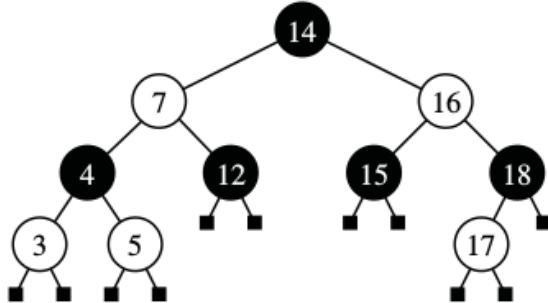


- Case 3 : r红

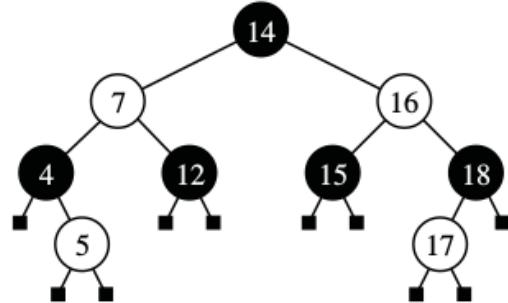
- r的父结点(即原先v的父结点)、兄弟结点w以及r的一个孩子进行restructuring , r变为父结点



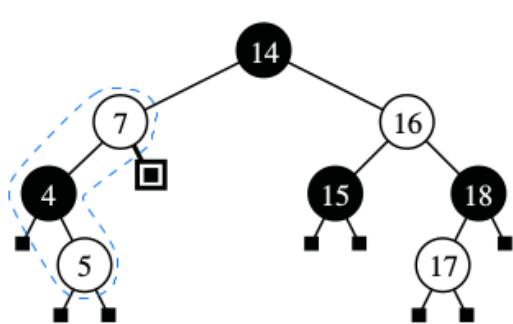
- 栗子: 删除一棵红黑树



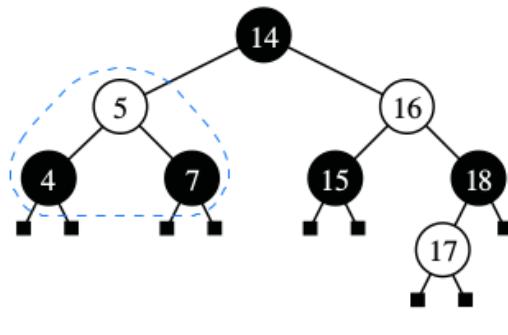
(a)



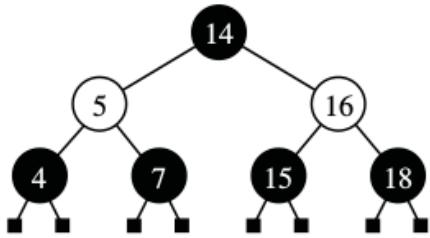
(b)



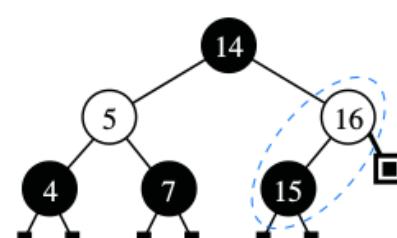
(c)



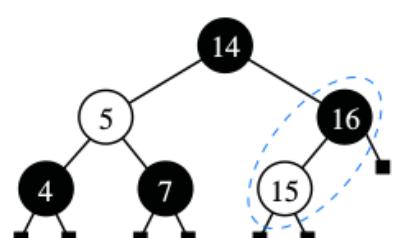
(d)



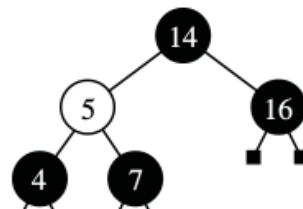
(e)



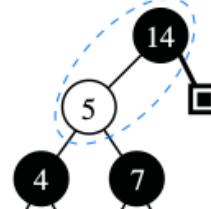
(f)



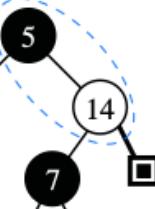
(g)



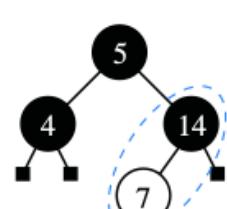
(h)



(i)



(j)



(k)

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);

```
1. public interface Entry{  
2.     public Object key();  
3.     public Object value();  
4. }
```

7.1.3 Comparator ADT

- compare(a,b);//返回i:ab,i>0
- 举个栗子:

```
1. public class NumComparator implements Comparator<Integer>{  
2.     public int compare(int a,int b){  
3.         if(a>b) return 1;  
4.         else if(a==b) return 0;  
5.         else return -1;  
6.     }  
7. }
```

7.1.4 Priority Queue Sorting

- 可用于对comparable elements 排序

```
1. Algorithm PQ-Sort (S,C)  
2. INPUT: sequence S, comparator C for elements in S;  
3. OUTPUT: 使用C进行升序排列过的S;  
4. {
```

```

5.         P=priority queue with comparator C;
6.
7.         //先一个个移除S中的元素,插入P
8.         while (S.isEmpty ()>0) {
9.             e=S.removeFirst ();
10.            P.insert (e,0);
11.        }
12.
13.        //一个个移除P中最小的元素,
14.        while (P.isEmpty ()>0) {
15.            e=P.removeMin ().key ();
16.            S.insertLast (e);
17.        }
18.    }

```

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

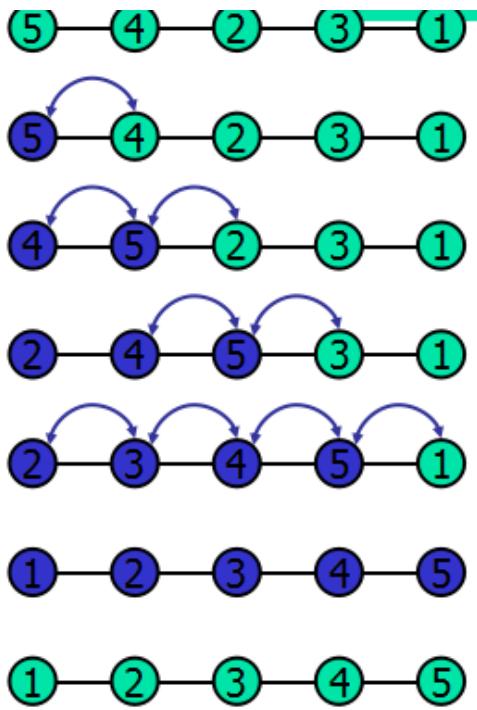
- 基于未排序list的优先队列 $O(n^2)$
- 先随便插入P，每次移除P中最小的元素插入S
 - 插入n个元素 $O(n)$
 - 移除最小元素 $1 + 2 + \dots + n = O(n^2)$
- 选择排序：每次从剩余元素中选取最小的交换到相应位置

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 $1 + 2 + \dots + n = O(n^2)$
 - 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

- 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

- 每轮都对全体相邻元素两两比较交换
- 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 $O(n)$

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, $\text{key}(v) \geq \text{key}(\text{parent}(v))$;
- 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 $h = O(\log n)$
- 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

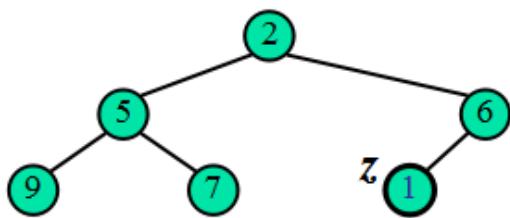
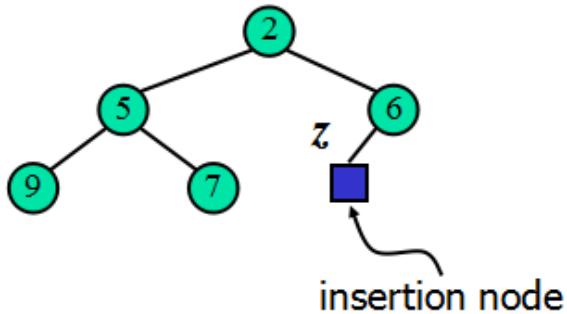
- 每次将新entry插入当前结尾结点的下一个位置, 并使用upheap()来更新du

```

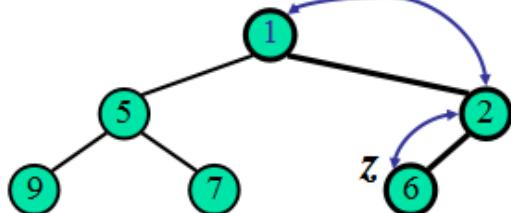
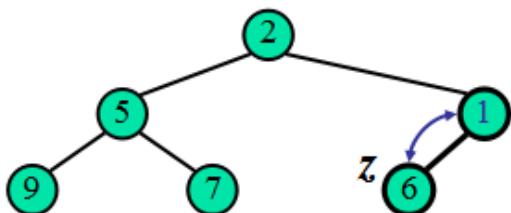
1.     Algorithm Insert{
2.         找到插入点Node z (当前结尾结点的下一个位置);
3.         Store k at z;

```

```
4.         upheap ()  
5.     }
```



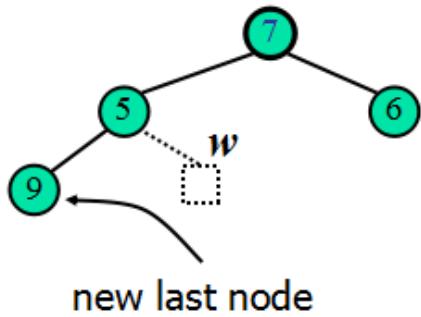
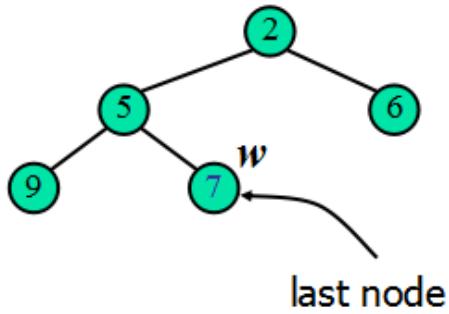
- + upheap : 插入新entry后heap-order可能被破坏
- + 新插入结点与其父结点进行比较 , 直到满足heaporder或者新结点到达root;
- + upheap过程 $O(\log n)$



7.2.2 Deletion

- 返回root位置entry, 并将结尾entry替换到root ,之后使用downheap恢复堆排序

```
1.     Algorithm remove {  
2.         min = root.getEntry ();  
3.         root.setEntry (lastNode);  
4.         //删除原尾结点所在位置;  
5.         downheap ();  
6.     }
```

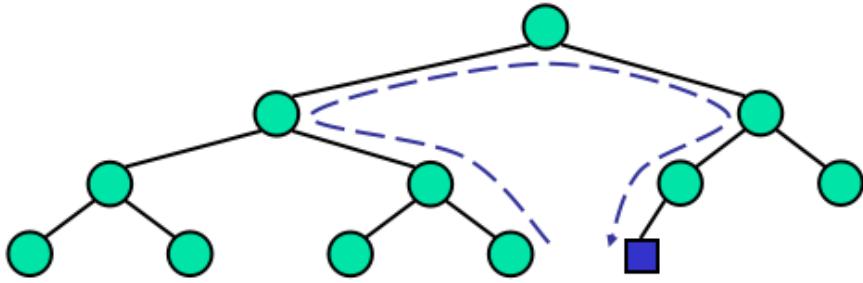


- + Downheap : 结尾entry替换到root后 , heaporder会被破坏
- + 从root开始 , 不断与子结点比较交换直到到达底层或是满足heap-order;
- + 优先交换子结点比较小的那个;
- + downheap过程 $O(\log n)$;



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 $O(\log n)$

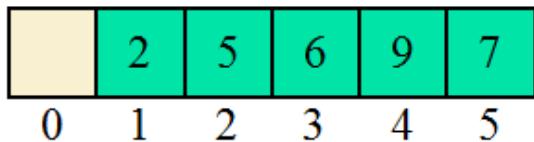
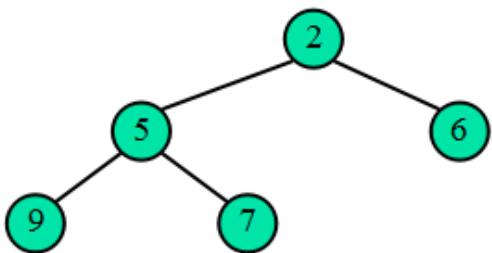


7.2.4 Heap-Sort

- 使用heap实现优先队列:
 - 空间复杂度 $O(n)$;
 - insert()/removeMin() : $O(\log n)$
 - size()/isEmpty()/min(): $O(1)$
 - locating: $O(1)$ for array-based heap , $O(\log n)$ for linked-tree heap;
- 基于heap的优先队列排序 : $O(n \log n)$
- 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;
- 使用java实现heap : *Data Structures and Algorithms by Goodrich and Tamassia* P378

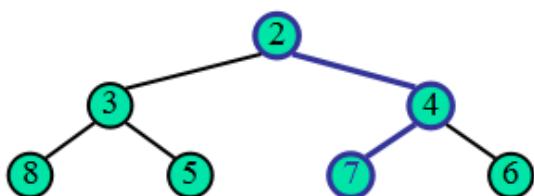
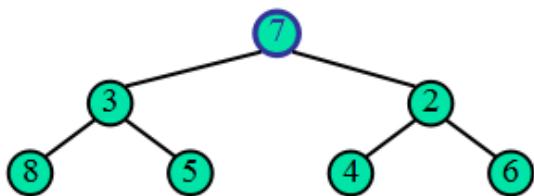
7.2.5 Array list-based heap implementation

- 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;
- 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;



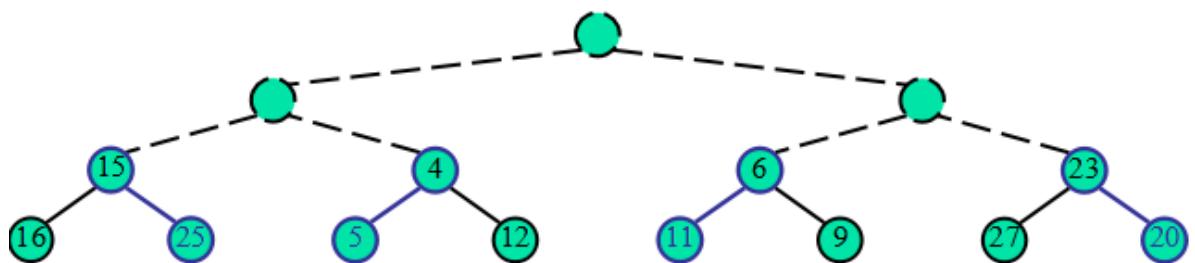
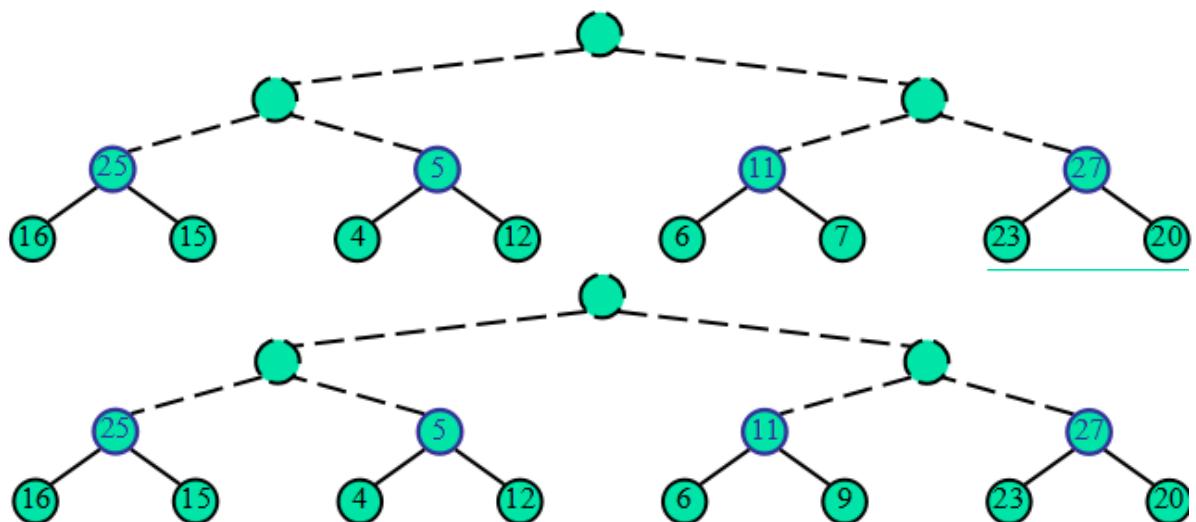
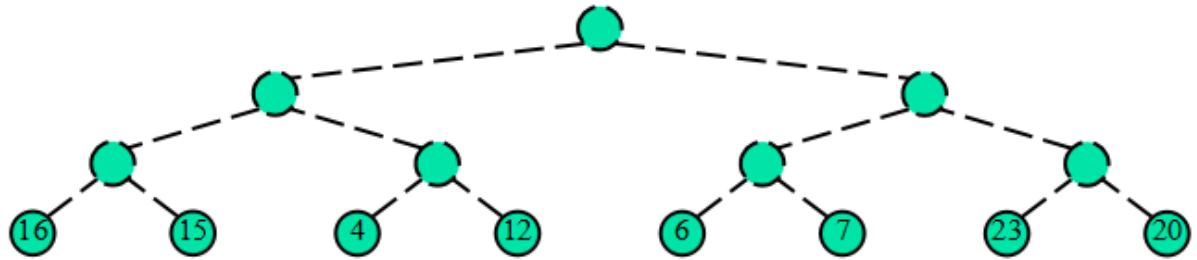
7.2.6 Merging two heaps

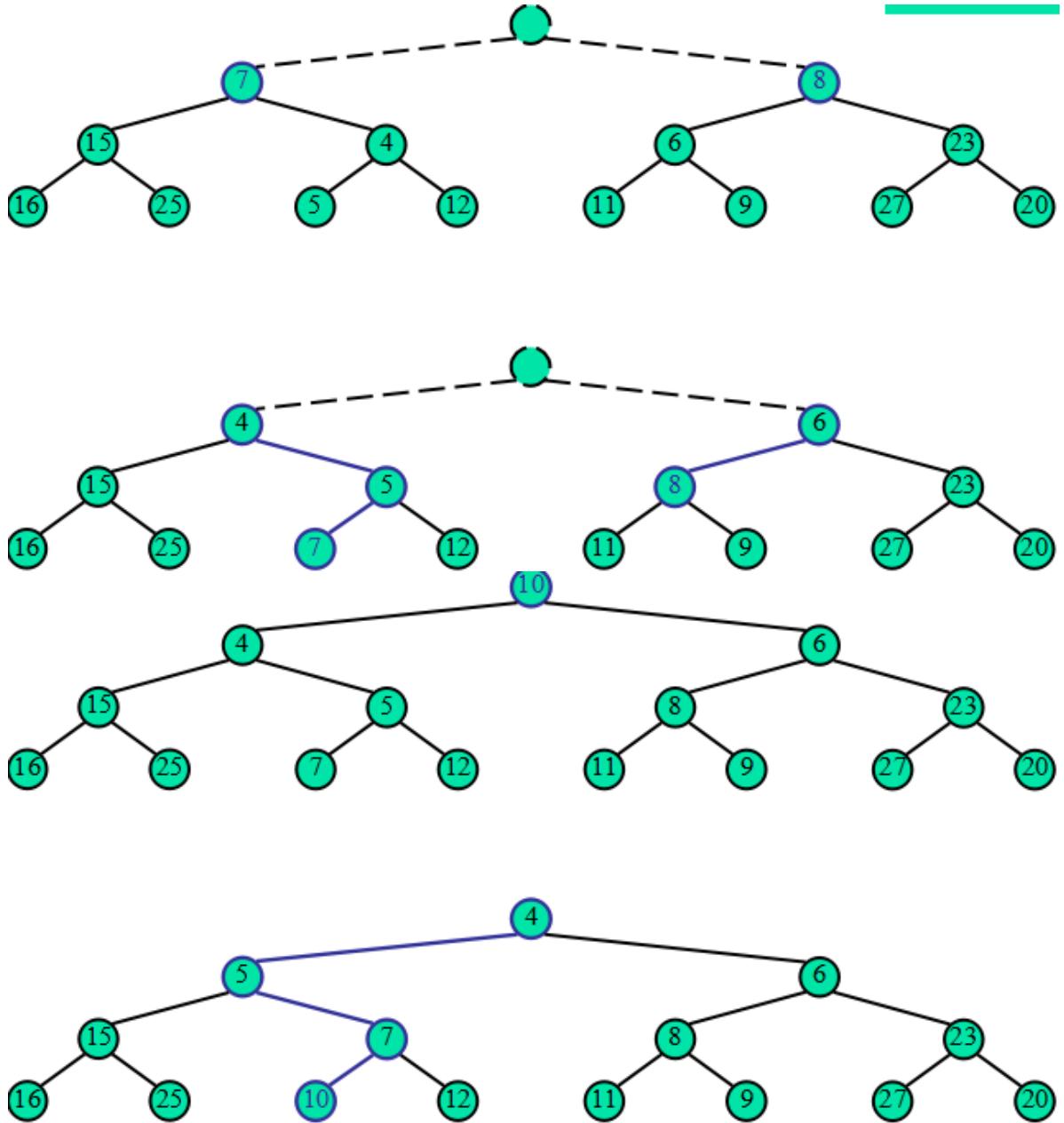
- Input 2 heaps and a key k;
- 创建一个新heap,根结点key为k,两棵子树为打算merge的heaps;
- 检验各结点是否满足heap-order;
- 对破坏规则的使用downheap来重构;



7.2.7 Bottom-up Heap Construction

- $\log n$ phases, 构造时间 $O(n)$, 比从上到下省时间
- 对于phase i : 将一对拥有 2^{i-1} keys的heaps被merge成一个 $2^{i+1}-1$ keys的heap





7.3 Adaptable PQ

7.3.1 Methods of the APQ ADT

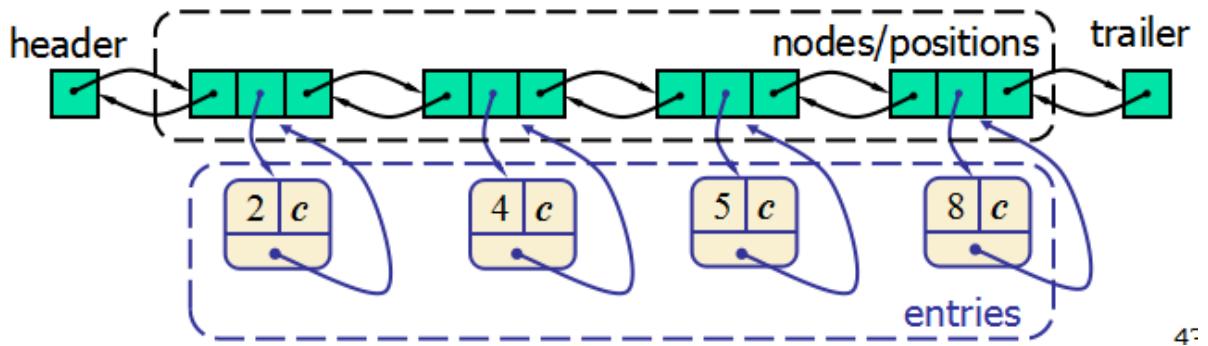
- `remove(e)` : 删除P中的某个键值对并返回entry e;
- `replaceKey(e,k)` : 使用k替换并返回原键值对e中的key;
- `replaceValue(e,x)` : 使用x替换并返回原键值对e中的value;

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

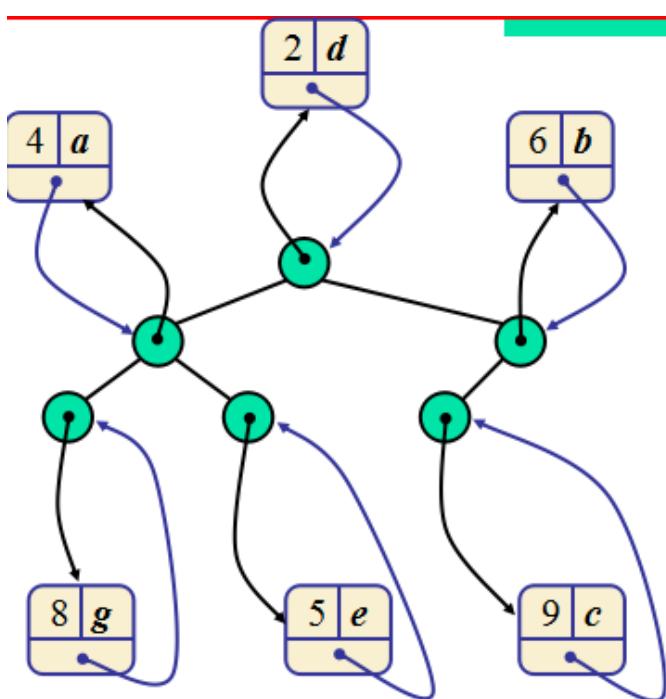
- location-aware list entry是一种object,包含以下项目:
 - key;
 - value;
 - position or rank of the item in the list;
- list中的每个position依次储存entry
- 在swap过程中back pointers(ranks)也被更新



47

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
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

1. Algorithm Merge (A, B)
2. Input: 各包含 $n/2$ 元素的sequences A, B;
3. Output: 已被排好序的序列 $A \cup B$
4. {

```

5.     S=[empty sequence];
6.     while(~A.isEmpty() && ~B.isEmpty()){//A与B非空
7.         if(A.first().element()<B.first().element())
8.             S.insertLast(A.remove(A.first()));
9.         else
10.            S.insertLast(B.remove(B.first()));
11.    } //A中当前元素与B中当前元素比较,比较小的插入S
12.
13.    while(A.isEmpty()==true){
14.        S.insertLast(B.remove(B.first()));
15.    } //A已空 , 将B剩余元素填入S
16.    while(B.isEmpty()==true){
17.        S.insertLast(A.remove(A.first()));
18.    } //B已空 , 将A剩余元素填入S
19.
20.    return S;
21. }

```

- python 实现

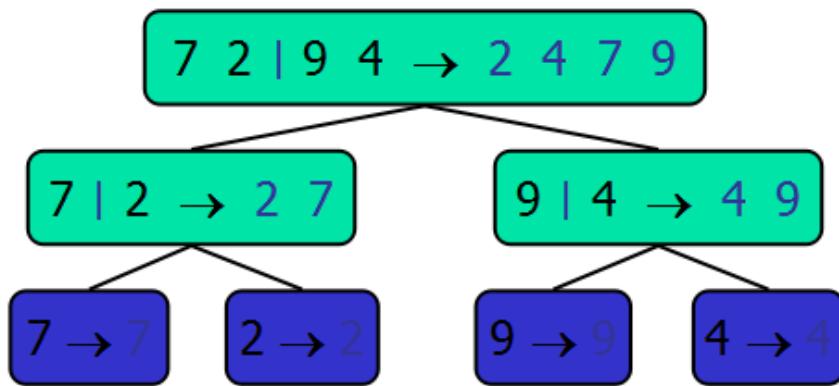
```

1. def merge(listA,listB):
2.     resultList=[]
3.     while len(listA)!=0 and len(listB)!=0:
4.         if listA[0]<listB[0]:
5.             resultList.append(listA[0])
6.             listA=listA[1:]
7.         else:
8.             resultList.append(listB[0])
9.             listB=listB[1:]
10.    while len(listA)!=0:
11.        resultList.extend(listA)
12.    while len(listB)!=0:
13.        resultList.extend(listB)
14.    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

```

1.  public class Merge{
2.      private static Comparable[] temp; //建立辅助数组
3.
4.      //method merge:对已经排好序的数组a[first,middle],a[middle+1,last]进行归并
5.      private static void merge(Comparable[] a,int first,int middle,int last){
6.          int i= first, j=middle + 1;  // [first,middle]+[middle+1,last]
7.          int m = middle,n = last;
8.          int k = 0;
9.
10.         while (i<=m && j<=n) {
11.             if (temp[i] <= temp[j]) a[k++] = temp[i++];
12.             else a[k++] = temp[j++];
13.         }
14.         while (i<=m) a[k++] = temp[i++];
15.         while (j<=n) a[k++] = temp[j++];
16.     }
17.
18.     //method sort:递归使用merge
19.     private static void sort(Comparable[] a,int first,int last){ //a[first,las
t]排序,每个private sort包含两个sort 递归
20.         if(last<=first) return;
21.         int middle=first+(last-first)/2;//中间为分界点
22.         sort(a,first,middle); //前一半排序
23.         sort(a,middle+1,last); //后一半排序
24.         merge(a,first,middle,last); //调用merge
25.     }
26.
27.     //main method
28.     public static void sort(Comparable[] a){
29.         temp=new Comparable[a.length]; //分配空间
30.         sort(a,0,a.length-1); //调用private method sort
31.     }
32. }
```

8.1.5 对归并排序的分析

- height : $O(\log n)$
- 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 $O(n \log n)$,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

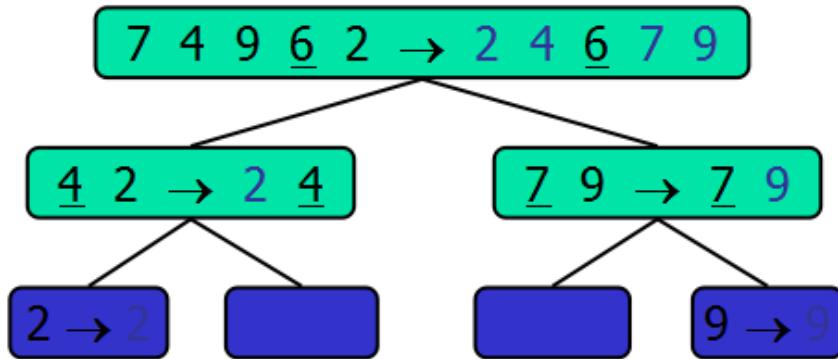
8.2.1 Partition

```

1. Algorithm Partition(S,p)
2. Input: Sequence S, Position p of the pivot;
3. Output: 将S按小于等于大于pivot,分为三个子序列L/E/G;
4.
5.     L, E, G = [empty];
6.     x = S.remove(p); // 将x作为pivot从S中移除
7.
8.     while (S.isEmpty() < 0) { // S中还有元素
9.         y = S.remove(S.first());
10.        if (y < x) L.insertLast(y);
11.        else if (y == x) E.insertLast(y);
12.        else G.insertLast(y);
13.    }
14.    return L, E, G;
15. }
```

- 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



- Java 实现

```

1. public class Quick
2. {
3.     private static void sort(Comparable[] a, int first, int last)
4.     {
5.         if (last <= first) return;
6.         int j = partition(a, first, last); // 该函数用于选定分界点 a[j]
7.         sort(a, first, j - 1);
8.         sort(a, j + 1, last);
9.     }
10.
11.    public static void sort(Comparable[] a)
12.    {
13.        StdRandom.shuffle(a); // 输入数据后打乱，得到随机数组
14.        sort(a, 0, a.length - 1);
15.    }
16. }
```

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 : $O(n \log n)$
- 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

```

1. Algorithm inPlaceQS(S, l, r)
2. Input: Seq S, rank l, rank r;
3. Output: 使用l/r划分并排序S中元素
```

```

4. {
5.     if ( l >= r ) return;
6.     在l,r之间随机选一个整数定为 i ;
7.     x=S.elemAtRank (i);
8.     p=inPlacePartition (x);
9.     inPlaceQS (S,l,p-1);
10.    inPlaceQS (S,p+1,r);
11. }

```

- 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;

```

1. Algorithm InPlacePartition(S,0,n-1):
2. Input: Sequence S with n elements;
3. Output: the partition point
4. {
5.     pivot=S[0];
6.     point left=1;
7.     point right=n-1;
8.     while(true){
9.         while (S[left]<=pivot) { //从左到右运行left指针
10.             left++;
11.             if (left>n-1) break;
12.         }
13.         while (S[right]>pivot) { //从右到左运行right指针
14.             right--;
15.             if (right<0) break;
16.         }
17.         if (left>=right) break; //指针相遇 , 切分结束
18.         exchange(S[left],S[right]); //指针未相遇 , 交换元素
19.     }
20.     exchange(pivot,S[right]);
21.     return right;
22. }

```

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	$O(n \log n)$	原位, 不稳定, 空间复杂度 $O(1)$
Merge-sort	$O(n \log n)$	非原位, 稳定, 空间复杂度 $O(n)$
Quick-sort	$O(n \log n)$	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)

```

1.  public static void quickSort(Object[] S, Comparator c) {
2.      if (S.length<2) return; //已经排好序了
3.      quickSortStep(S,c,0,S.length-1);
4.  }
5.
6.  private static void quickSortStep(Object[] S, Comparator c,
7.      int leftBound, int rightBound ) {
8.      if (leftBound >= rightBound) return; //左边界>=右边界->null
9.      Object temp; // temp object used for swapping
10.     Object pivot = S[rightBound]; //初始化最右元素为pivot
11.     int leftIndex = leftBound;
12.     int rightIndex = rightBound-1;
13.
14.     while (leftIndex <= rightIndex) {
15.         //向右扫描寻找大于pivot的
16.         while ((leftIndex<=rightIndex) && (c.compare(S[leftIndex],pivot)<=0) )
17.             leftIndex++;
18.         //向左扫描寻找小于pivot的
19.         while ((rightIndex>=leftIndex) && (c.compare(S[rightIndex],pivot)>=0) )
20.             rightIndex--;
21.         //交换找到的两个元素
22.         if (leftIndex < rightIndex) {
23.             temp = S[rightIndex];
24.             S[rightIndex] = S[leftIndex];
25.             S[leftIndex] = temp;
26.         }
27.     } //不断循环直到指针相遇
28.
29.     //将pivot从最右交换到指针相遇的地方S[leftIndex]
30.     temp = S[rightBound];
31.     S[rightBound] = S[leftIndex];
32.     S[leftIndex] = temp;
33.
34.     quickSortStep(S,c,leftBound,leftIndex-1);
35.     quickSortStep(S,c,leftIndex+1,rightBound);

```

36. }

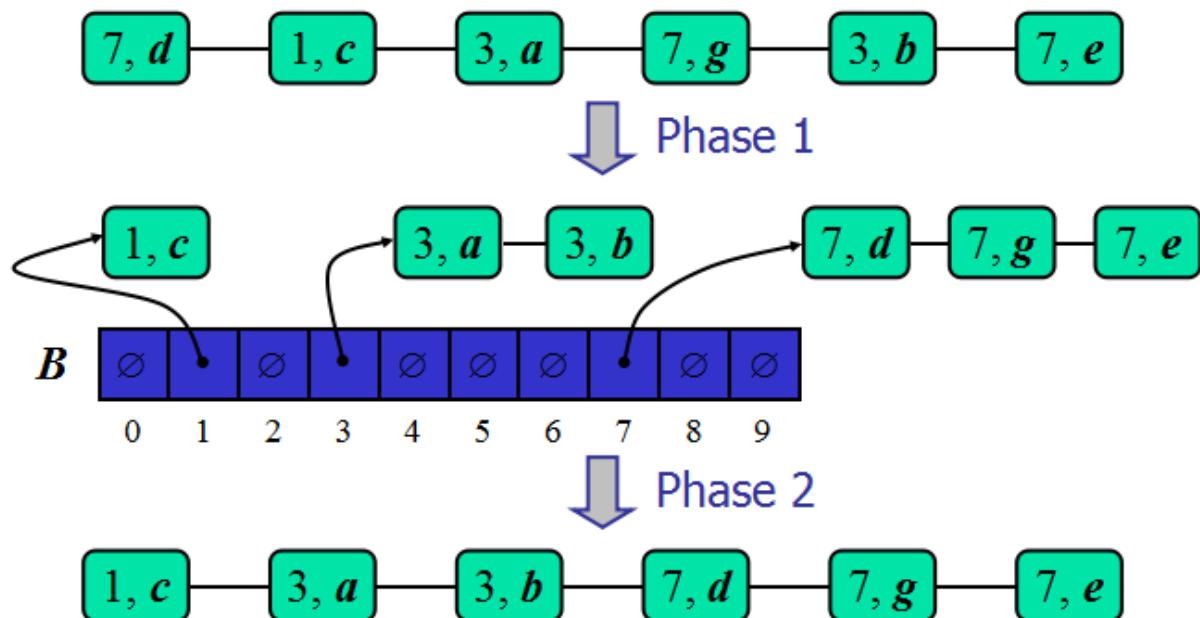
8.3 Bucket-Sort and Radix-Sort

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

```
1. Algorithm BucketSort (S,N)
2. Input : n个entries, N个keys (n >= N);
3. Output : sorted S ;
4.
5.     B=N个空序列组成的数组;
6.     while (S.isEmpty () <0) {
7.         (k,o)=S.remove (S.first ());
8.         B [k].insertLast ((k,o));
9.     } //将key相同的entries收到同一个bucket里
10.
11.    for (i=0;i++;i<=N-1) {
12.        while (B [i].isEmpty () <0) {
13.            f=B [i].first ();
14.            (k,o)=B [i].remove (f);
15.            S.insertLast ((k,o));
16.        }
17.    } //对每个bucket分别排序,然后按key大小再塞回S
18. }
```



- 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

- Lexicographic Order
 - d-tuple : 一个带有d个keys(k_1, k_2, \dots, 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...

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

\Leftrightarrow

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

- 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;

```

1. Algorithm lexicographicSort (S)
2. Input: sequence S of N-tuples;
3. Output: sequence S sorted in lexicographic order;
4.
5.   for ( i = N; i>=1; i--; )
6.     stableSort (S, Ci);
7. }
```

- 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
- 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 $O(d(n + N))$

```

1. Algorithm radixSort (S, N)
2. Input: 包含N-tuples的序列S. 对于每个tuple  $(x_1, \dots, x_N)$  in S,  $x_i \in [0, N-1]$  ;
3. Output: 字典顺序的S;
4. {
5.     for (i=N; i>=1; i--)
6.         bucketSort (S, N);
7. }

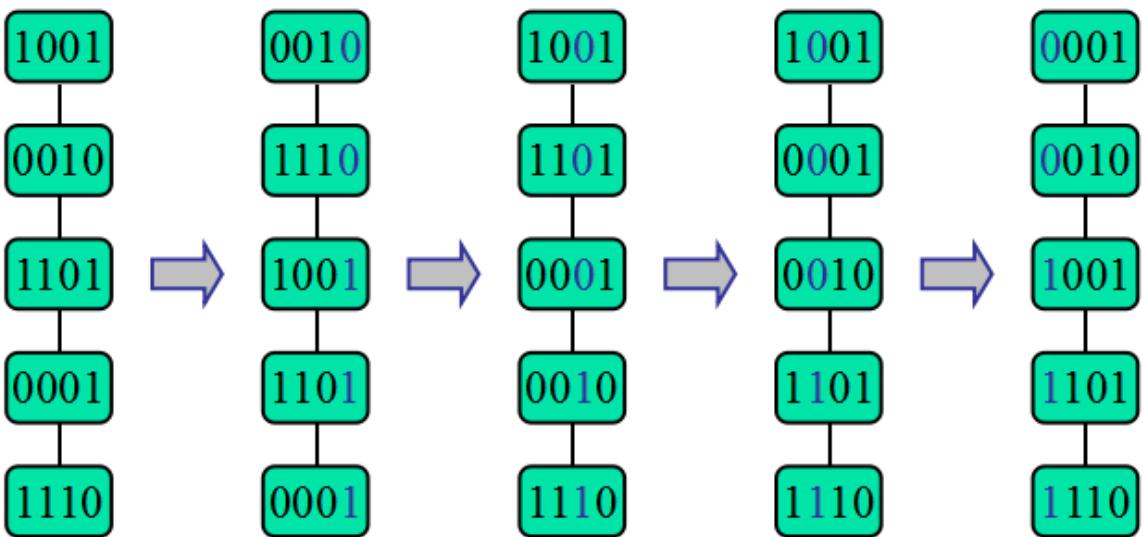
```

- 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
 $x = x_{b-1} \dots x_1 x_0$, 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

```

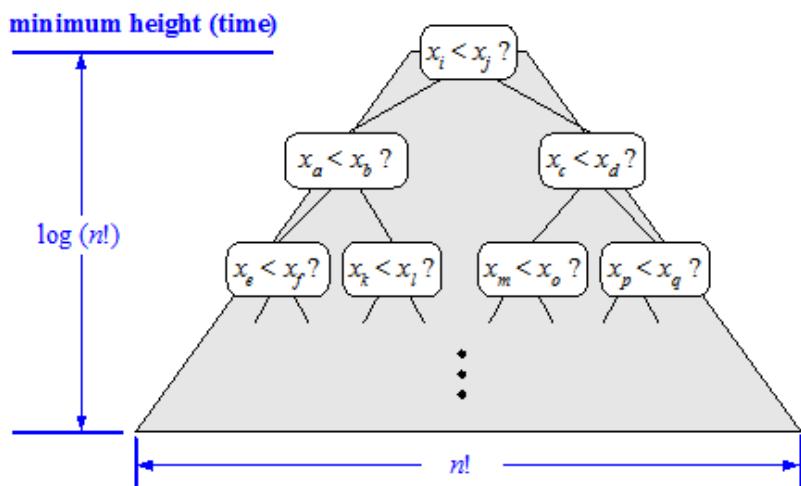
1. Algorithm binaryRadixSort (S)
2. Input: S of n b-bit integers;
3. Output: 对S进行排序, S中每个元素x被替换为item (0, x) ;
4. {
5.     for (i=0; i<=b-1; i++) {
6.         使用x中的bit位  $x_i$  替换 (k, x) 中的key k;
7.         bucketSort (S, 2);
8.     }
9. }

```



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 $\Omega(n \log n)$

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;
- prune : 随机选取pivot x , 将S切分为L/E/G;
- search : 取决于被查找数k , 选择在E/L/G三个子集中的一个进行递归;

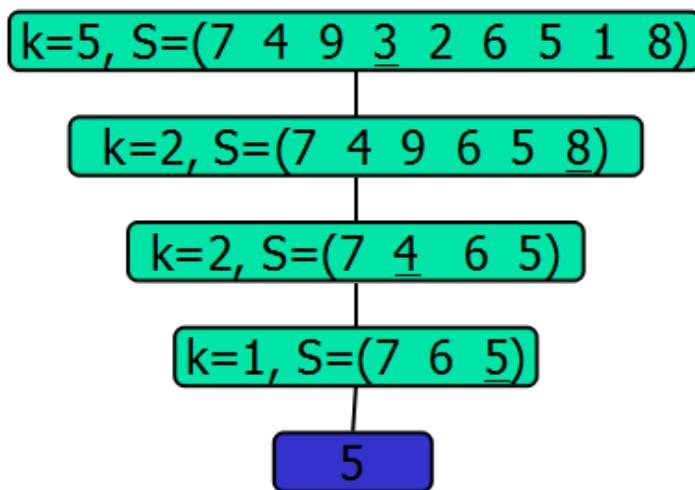
8.5.2 Partition

```

1.   Algorithm partition(S,p)
2.     Input:sequence S, position p
3.     Output: subsequences L/E/G
4.     {
5.       L,E,G = [empty sequences];
6.       x = S.remove(p);
7.       while (S.isEmpty()<0)
8.       {
9.         y = S.remove(S.first());
10.        if (y<x) L.insertLast(y);
11.        else if (y==x) E.insertLast(y);
12.        else G.insertLast(y);
13.      }
14.      return L,E,G;
15.    }

```

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

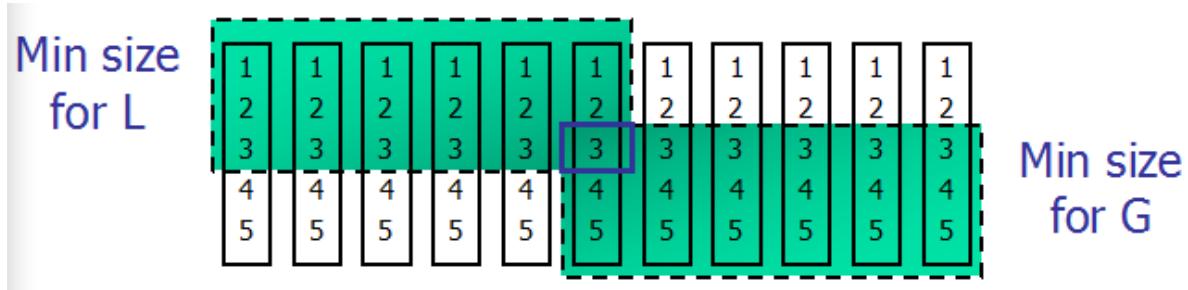


8.5.3 快速选择的性能

- 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

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

- Set intersection：“与”，S中的元素为AB都有的

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

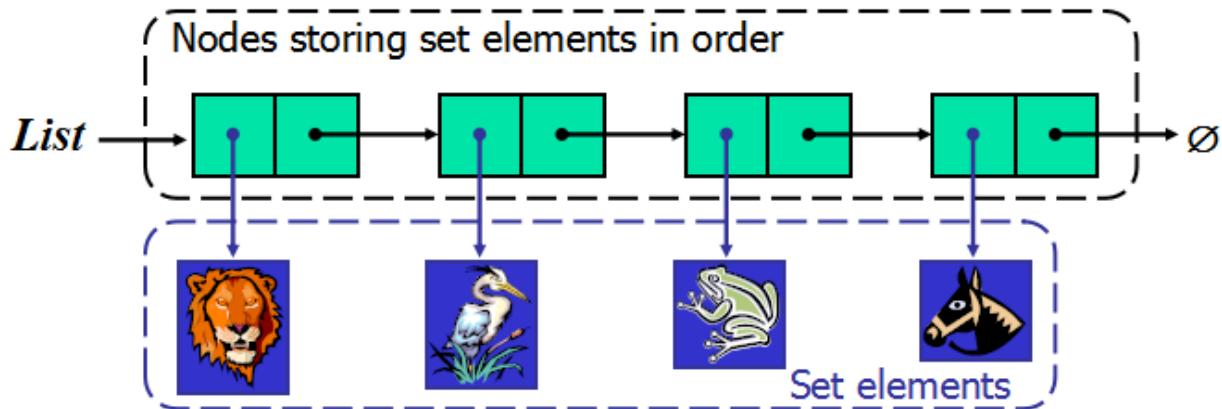
- Set subtraction：“非”，A中有而B中没有的元素

```
1. 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 , $O(1)$

```
1. Algorithm genericMerge (A, B) {
2.     S=[empty sequence];
3.     while ((-A.isEmpty()) && (-B.isEmpty())) { //AB均非空
4.         a=A.first().element();
5.         b=B.first().element();
6.         if (a<b) {
7.             aIsLess(a,S);
8.             A.remove(A.first());
9.         }
10.        else if (a>b) {
11.            bIsLess(B,s);
12.            B.remove(B.first());
13.        }
14.        else{
15.            bothAreEqual(a,b,S);
16.            A.remove(A.first());
17.            B.remove(B.first());
18.        }
19.    }
20. }
```

- 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

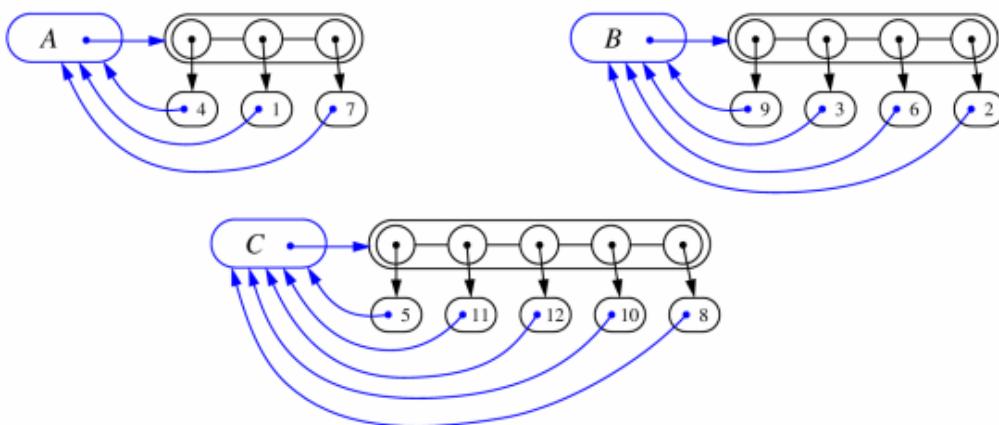
- 用于处理一些不相交集合 (Disjoint Sets) 的合并及查询问题
- 时间复杂度 $O(n \log n)$
- 举个栗子: 给出一组结点, 判断是否连通

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;

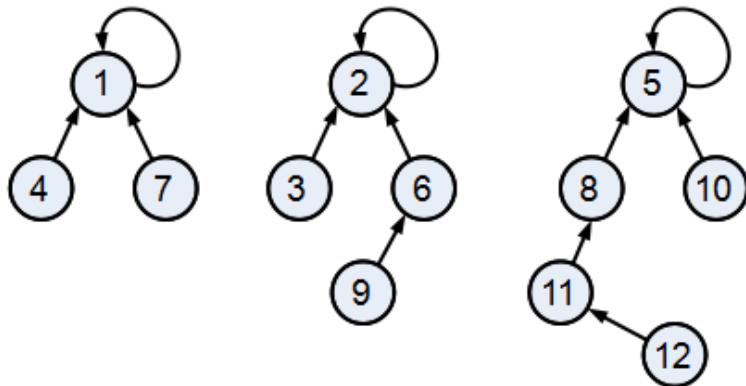


- 分析:
 - 每次union操作 : 将元素从小set移动到大set;
 - 这个大set的size至少是小set的两倍;
 - 因此一个元素最多被移动 $O(\log n)$ 时间;

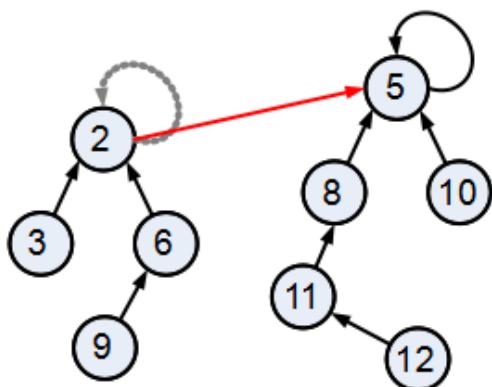
- n次union-find耗时 $O(n \log n)$;

8.7.3 基于tree的实现

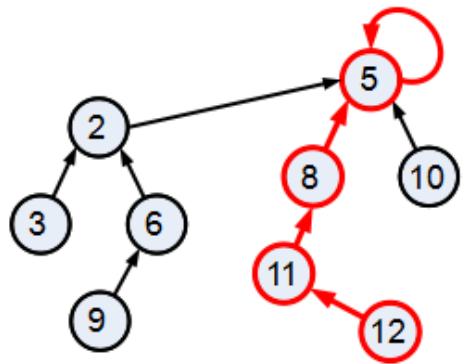
- 每个元素储存在node里,且包含一个指向set名的指针;
- 对于根结点v, set指针指向自身,那么v也是一个set名;
- 子结点的链接都是指向父结点的;
- 每个set都是一棵树:根为指针指向自身的node;



- 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;
```



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 : address book/student-record database

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

```

1.  public interface Map<Key,Value>{
2.      public Value get (Key k);
3.      public Value put (Key k,Value v);
4.      public Value remove (Key k);
5.      public int size ();
6.      public boolean isEmpty ();
7.      public Iterable<Key> keys ();

```

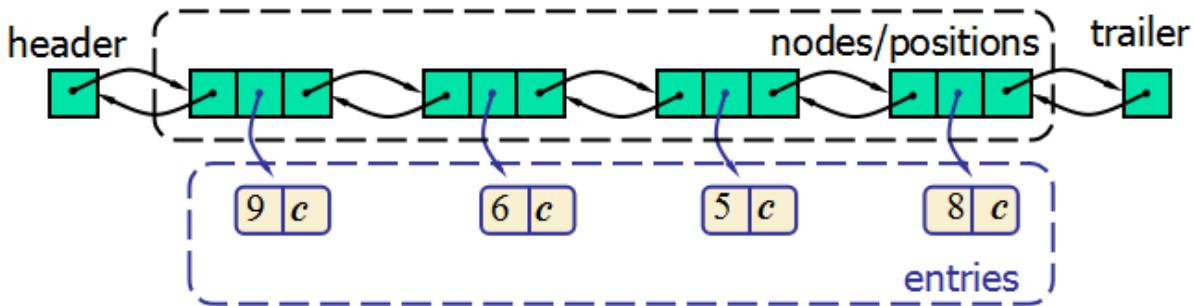
```

8.     public Iterable<Value> values();
9.     public Iterable<Key, Value> entries();
10.    }

```

9.1.2 list-based map

- based on unsorted list



- 基本操作算法

```

1. Algorithm get (k) {
2.     B=S.positions () ; //B:迭代器,遍历S中的positions
3.     while (B.hasNext ()) {
4.         p=B.next ();
5.         if (p.element () .key () ==k)
6.             return p.element () .value ();
7.     }
8.     return null; //No match
9. }
10.
11. Algorithm put (k, v) {
12.     B=S.positions ();
13.     while (B.hasNext ()) {
14.         p=B.next ();
15.         if (p.element () .key () ==k) {
16.             t=p.element () .value ();
17.             B.replace (p, (k, v));
18.             return t;
19.         }
20.     }
21.     S.insertLast ((k, v));
22.     n=n+1;
23.     return null;
24. }
25.
26. Algorithm remove (k) {
27.     B=S.positions ();
28.     while (B.hasNext ()) {
29.         p=B.next ();

```

```

30.         if (p.element () .key () ==k) {
31.             t=p.element () .value ();
32.             S.remove (p);
33.             n=n-1;
34.             return t;
35.         }
36.     }
37.     return null;
38. }

```

- 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 $h(x)=x \bmod N$
 - 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 h_1 : keys -> integers(hash code)
 - Compression function h_2 : integers -> [0,N-1]

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

9.2.3 Hash codes(h_1 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

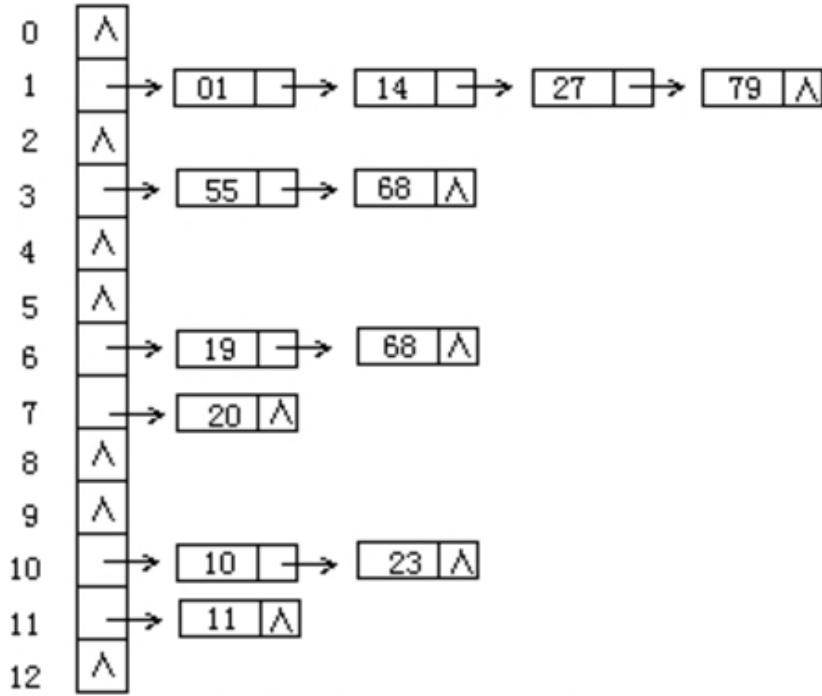
$$p(z) = a_0 + a_1 z + a_2 z^2 + \dots a_{n-1} z^{n-1};$$
 +适合strings; + $p(z)$ 可以使用Horner规则计算,
 $O(n) + p_0(z) = a_{\{n-1\}} + p_i(z) = a_{\{n-i-1\}} + z^{p_i-1}(z), i=[1, \dots, n-1] + p(z) = p_{\{n-1\}}(z)$

9.2.4 Compression functions(h_2 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

```

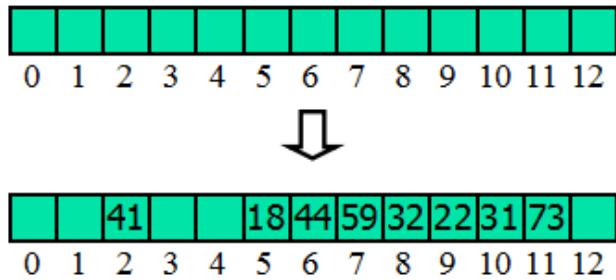
1. Algorithm get(k)
2. Output: 在map中找到key并返回值, 其中A为哈希表
3. {
4.     return A[h(k)].get(k); // delegate the get to the list-based map at
A[h(k)]
5. }
6.
7. Algorithm put(k,v)
8. Output: insertion or set
9. {
10.    t = A[h(k)].put(k,v); // delegate the put to the list-based map at A[h(k)]
11.    if (t==null)
12.        n = n + 1;
13.    return t ;
14. }
15.
16. Algorithm remove(k)
17. Output:删除key并返回值, 找不到返回null
18. {
19.    t = A[h(k)].remove(k);
20.    if (t!=null) // k was found
21.        n = n-1;
22.    return t ;
23. }
```

- 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;

```

1. Algorithm get (k) {
2.     hashValue=h (k) ;
3.     p=0;
4.     while (p!=N) {
5.         c=A[hashValue];
6.         if (c==null) return null;
7.         else if (c.key ()==k) return c.element ();
8.         else{
9.             hashValue=(hashValue+1) mod N;
10.            p=p+1;
11.        }
12.    }
13. }
```

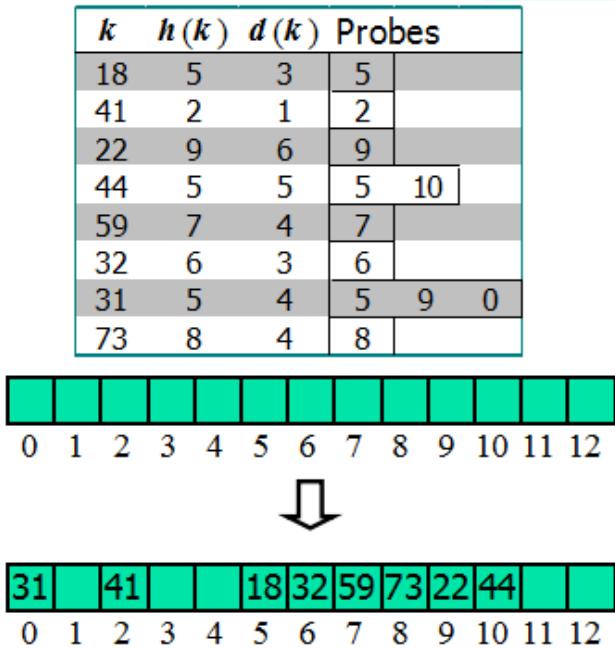
```
12.         }
13.     return null;
14. }
```

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

```
1. Algorithm put (k, o)
2. {
3.     if the table is full->throws an exception;
4.     起始位置:cell h(k)
5.     连续probe cell直到:
6.         找到i=h(k)->替换内容;
7.         没找到->在cell i新增键值对 (k, o);
8.     }
9.
10. Algorithm remove (k)
11. {
12.     找寻key k的键值对;
13.     找到了->替换该键值对为特殊项目 AVAILABLE, 并返回o;
14.     没找到->返回null;
15. }
```

9.2.6 Double Hashing

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



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 $\alpha = n/N$ 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

9.2.8 Java实现hash map : slides P30-38 此处暂略

```
public class HashTable implements Map{}
```

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)

- `entries()` : 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

```

1. Algorithm findAll (k)
2. Input: key k
3. Output: iterator of entries with key equal to k
4.
5.     L=[empty list];
6.     B=D.entries ();
7.     while(B.hasNext ()) {
8.         e=B.next ();
9.         if (e.key ()==k)
10.             L.insertLast (e);
11.     }
12.     return L.elements ();
13. }
14.
15. Algorithm insert (k,v)
16. Input: key k,value v
17. Output: add the entry(k,v) to D
18.
19.     Create a new entry e=(k,v);
20.     S.insertLast (e);
21.     return e;
22. } //由于允许相同key,不需要检查是否已经存在
23.
24. Algorithm remove (e)
25. Input:entry e;
26. Output:找到e就删除找不到返回null
27.
28.     B=S.positions ();
29.     while(B.hasNext ()) {
30.         p=B.next ();
31.         if (p.element ()==e) {
32.             S.remove (p);
33.             return e;
34.         }
35.     }
36.     return null;
37. }
```

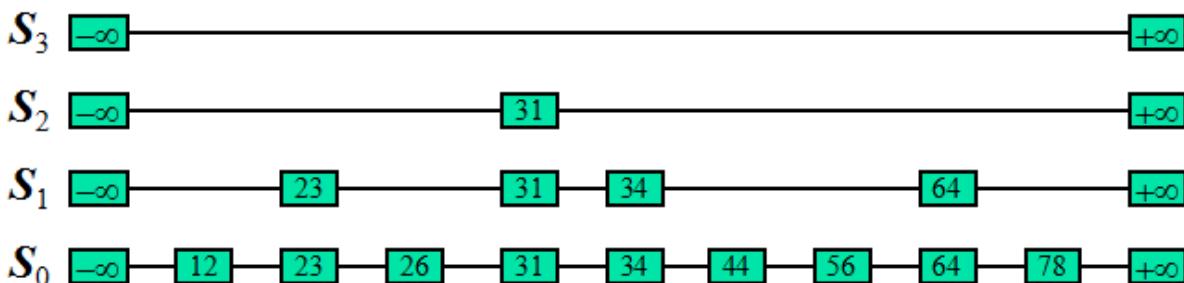
9.3.3 Hash-based dictionary

9.3.4 Search Table

- based on sorted sequence
 - sorted by key;
 - 使用external comparator来比较keys;
- 性能:
 - find , $O(\log n)$: 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lists S_0, S_1, \dots, S_h
 - 每个 S_i 包含special keys(sentinels) $+\infty, -\infty$
 - S_0 包含 S 中的所有 keys, 升序;
 - 每个list是上一个list的子序列 : $S_h \in S_{h-1} \dots \in S_1 \in S_0$
 - 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

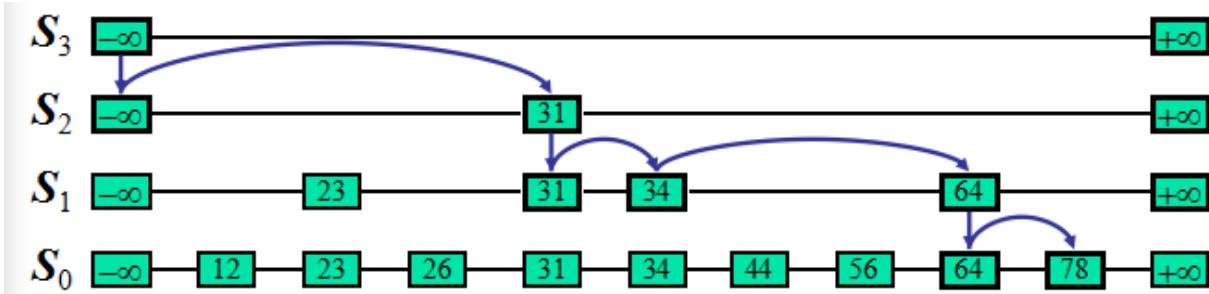
```
1. Algorithm SkipSearch(k) :
2.   Input: given key k;
3.   Output: the position of entry(k,p), 找不到返回小于k的最大key所在位置;
4.   {
5.     p=MINUS_INF; //起始点, 最上层最左侧-∞
6.     while (below(p) != null) {
```

```

7.         currentKeyNext=next (p) .key ();
8.         if (k==currentKeyNext) {
9.             return next (p);
10.        }
11.        else if (k>currentKeyNext) {
12.            p=next (p); //同level前进
13.        }
14.        else p=below (p); //原位下降;
15.    }
16.    return p; //没找到->小于k的最大key所在位置
17. }

```

- j举个栗子：搜索78



9.4.2 Randomized Algorithms

- 包含以下语句

```

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

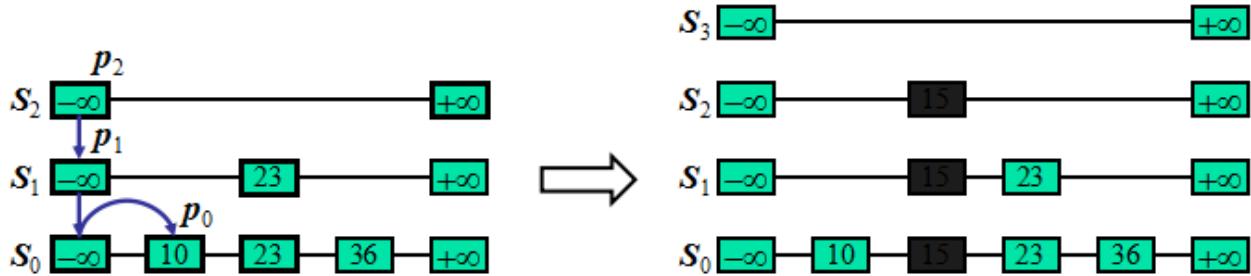
```

- 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

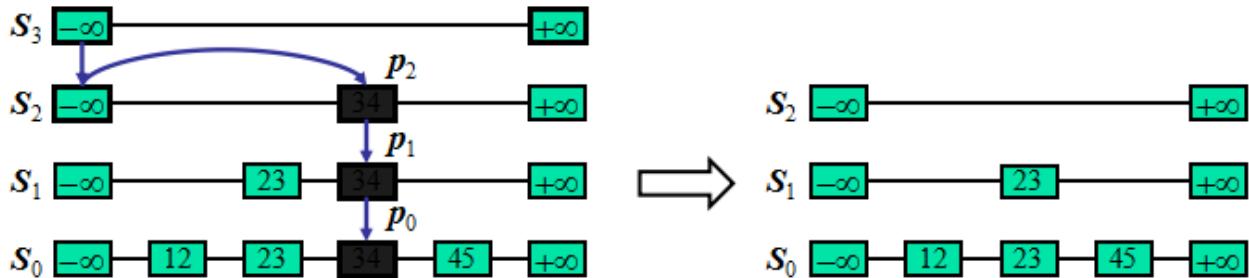
- 插入操作需要用到随机化算法
 - 类似抛硬币,重复抛直到得到背面,在这期间得到正面的次数为i
 - \$S_i\$ 为新元素被插入的最上层level
 - 确定 \$S_i\$:
 - if(i>h) 增加一些只含 \$+\infty, -\infty\$ 的空list \$S_{h+1}, \dots, S_i\$;
 - else 排除多余的list \$S_{i+1}, \dots, S_h\$, 最后剩下 \$S_0, S_1, \dots, S_i\$;
 - 使用SkipSearch(k)查询是否已经存在entry:
 - 如果已经存在,return p并将value替换为v;
 - 如果不存在,返回的是小于k的最大key所在位置 \$p_0, p_1, \dots, p_i\$;

- 将新entry(k,v)插入到p₀,p₁,...p_i后面
- 举个栗子 : i=2,插入15



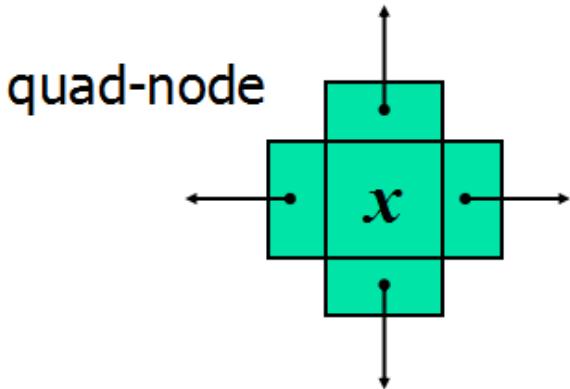
9.4.4 Deletion

- 先确定S_i;
- SkipSearch(k),找到删除点p₀,p₁,...p_i;
- 从每个list S_j中删除相应位置p_j;
- 对于只包含两个特殊keys的list,留下level最小的那个,剩下全删除;
- 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 $O(\log n)$
- 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 $3\log n$ 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 $O(\log n)$
- 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(): $O(\log n)$;
- entries()/keys()/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

```

1. Algorithm BruteForceMatch(T, P)
2. Input: Text T(n), Pattern P(m)
3. Output: 如果找到了match P的部分,标出在T中的位置,否则返回-1
4. {
5.     for(i=0; i<n-m+1; i++) {
6.         j=0;
7.         while(j<m & T[i+j]==P[j])
8.             j=j+1;
9.         if(j==m) //match->给出output,结束循环
10.            return i;
11.    }
12.    return -1;
13. }
```

- 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=aaaaaaah,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

HERE IS A SIMPLE EXAMPLE
EXAMPLE

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

HERE IS A SIMPLE EXAMPLE
EXAMPLE

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

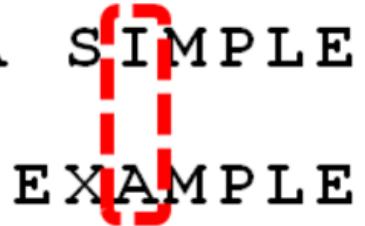
HERE IS A SIMPLE EXAMPLE
EXAMPLE

- $P[6] == T[15]$ 匹配 : 不断向前匹配直到P₂发生不匹配

HERE IS A SIMPLE EXAMPLE



HERE IS A SIMPLE EXAMPLE



- 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

HERE IS A SIMPLE EXAMPLE



- Boyer-Moore算法基于Last-Occurrence 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 $i+1=0 \rightarrow$ 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) \rightarrow 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;

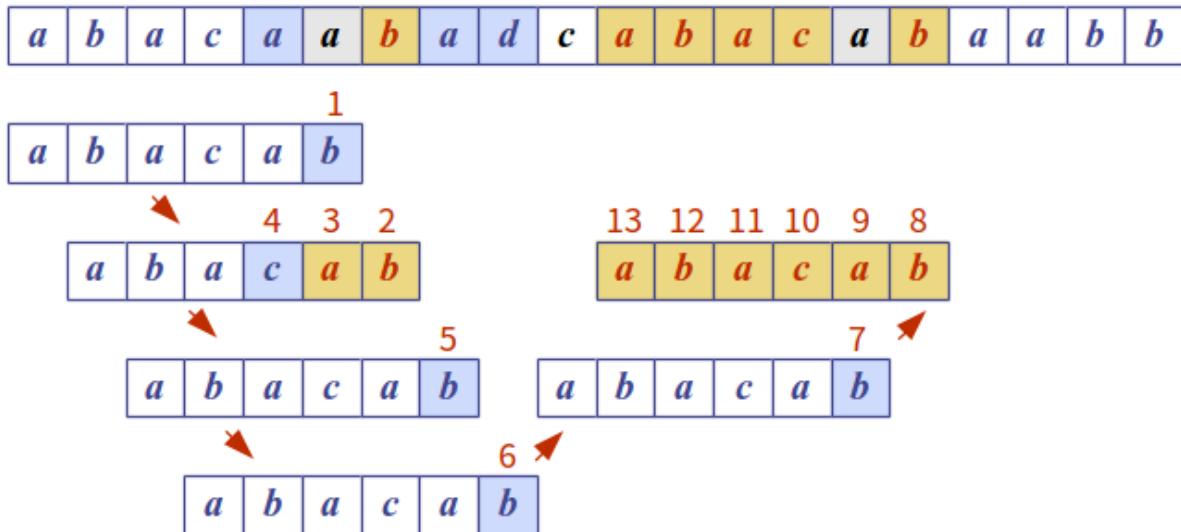
```
1. Algorithm BoyerMooreMatch(T, P, S)
2.   Input: Text T (n), Pattern P (m), 字符集S
3.   Output: T中与P相匹配的位置
4.   {
```

```

5.         L=lastOccurenceFunction(P,S);
6.
7.         //初始比较位置 : P的最后一位P[m-1]
8.         i=m-1;
9.         j=m-1;
10.
11.        while(i<=n-1){
12.            if(T[i]==P[j]) { //发生匹配
13.                if(j == 0) { //完全匹配
14.                    return i;
15.                }
16.                else{ //尚未完全匹配 , 看前一位是否匹配
17.                    i = i - 1;
18.                    j = j - 1;
19.                }
20.            }
21.
22.            else{ //发生不匹配 , 进行移动
23.                l = L[T[i]]; //在P中查询最后出现不匹配字符T[i]的位置
24.                i = i+m-min(j,1 + l); //算法核心 : 移位公式
25.                j = m - 1; //同时j回归初始位置重新开始比较
26.            }
27.        }
28.        return -1; // no match
29.    }

```

- 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 Algorithm

- 算法流程:

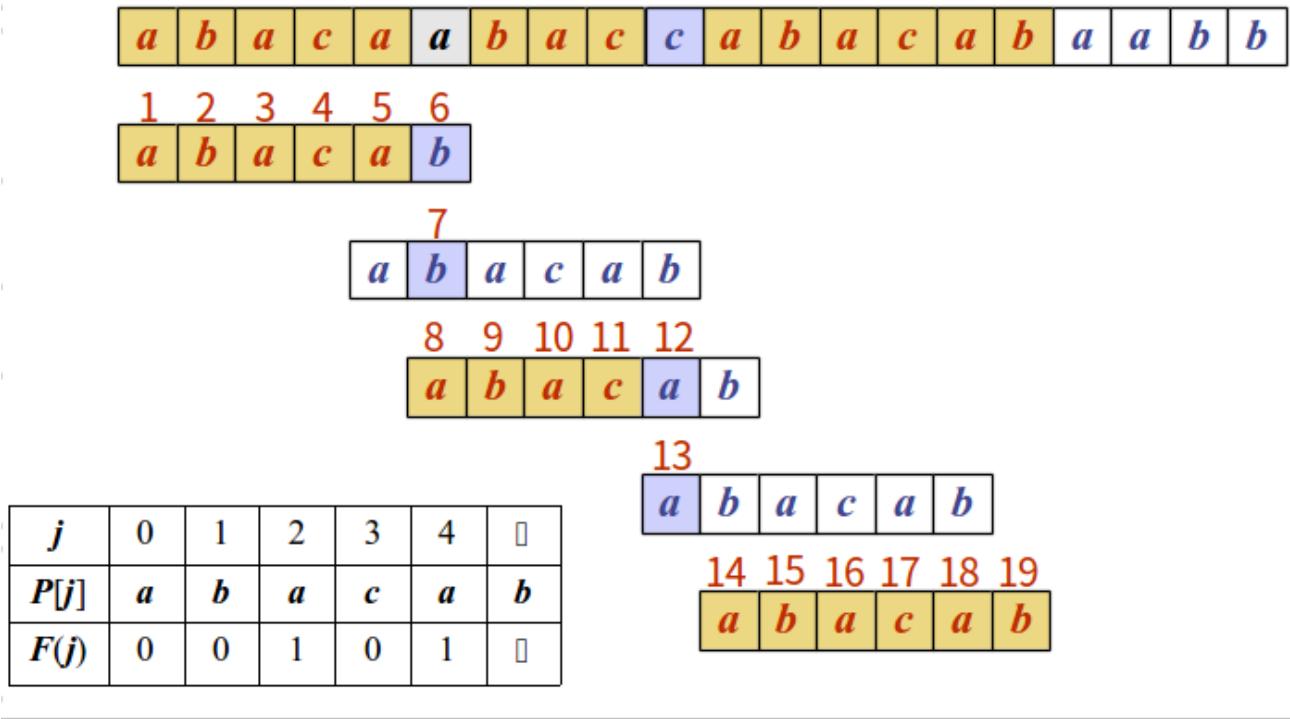
- 开始和暴力查找类似,从左至右匹配:

- 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：

```

1.   j : 0 1 2 3 4 5
2.   P[j]: a b a a b a
3.   F(j): 0 0 1 1 2 3
4.
5.   a:前缀[],后缀[]-->F[0]=0
6.   ab:前缀[a],后缀[b]-->F[1]=0
7.   aba:前缀[a,ab],后缀[ab,ba]-->F[2]=1
8.   abaa:前缀[a,ab,aba],后缀[aa,baa]-->F[3]=1
9.   abaab:前缀[a,ab,aba,abaa],后缀[b,ab,aab,baab]-->F[4]=2
10.  abaaba:前缀[a,ab,aba,abaa,abaab],后缀[ba,aba,aaba,baaba]-->F[5]=3
11.
12. T: abaab[x]
13. P: abaab[a]
14. 在j=5 处不匹配-->F[4]=2
15. 已经匹配的字符数5
16. 因此:前移 5-2=3位
17. abaabx
     abaaba

```



- KMP Algorithm

```

1.   Algorithm KMPMatch(T, P)
2. {
3.     F=failureFunction(P); //得到部分匹配表
4.     //初始化比较位
5.     i=0;
6.     j=0;
7.     while(i<n){
8.       if(T[i]==P[j]){
9.         if(j==m-1)
10.           return i-j;//完全匹配
11.         else{//第一位匹配,比较第二位,第三位,...
12.           i++;
13.           j++;
14.         }
15.       }
16.       else{
17.         if(j>0) j=F(j-1); //部分不匹配,调用F(j-1)
18.         //j=j-(j-F(j-1))=F(j-1)相当于后移j-F(j-1)位
19.         else i++; //j==0-->开头不匹配,直接后移位
20.       }
21.     }
22.   }
23. }
```

- 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

```

1. Algorithm failureFunction(P)
2. Input: the pattern P with m elements
3. Output: F=failureFunction(P)
4.
5.     F[0]=0; //前缀[],后缀[]
6.     //多于一个元素的情况
7.     i=1; //后缀
8.     j=0; //前缀
9.     while(i<m) {
10.         if(P[i]==P[j]) { //前头==后尾
11.             F[i]=j+1;
12.             i++;
13.             j++;
14.         }
15.         else if(j>0) {
16.             j=F[j-1]
17.         }
18.         else{ //j==0且ij元素不等
19.             F[i]=0;
20.             i++;
21.         }
22.     }
23. }
```

- 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

- $i=4, j=0 \rightarrow$ 出现匹配 $\rightarrow i++, j++, F_4=1$
- $i=5, j=1 \rightarrow$ 出现匹配 $\rightarrow i++, j++, F_5=2$
- $i=6, j=2 \rightarrow$ 不匹配, 且 $j>0 \rightarrow j=F_1=0 \rightarrow$ 头尾不匹配 $\rightarrow F_6=0, i++$

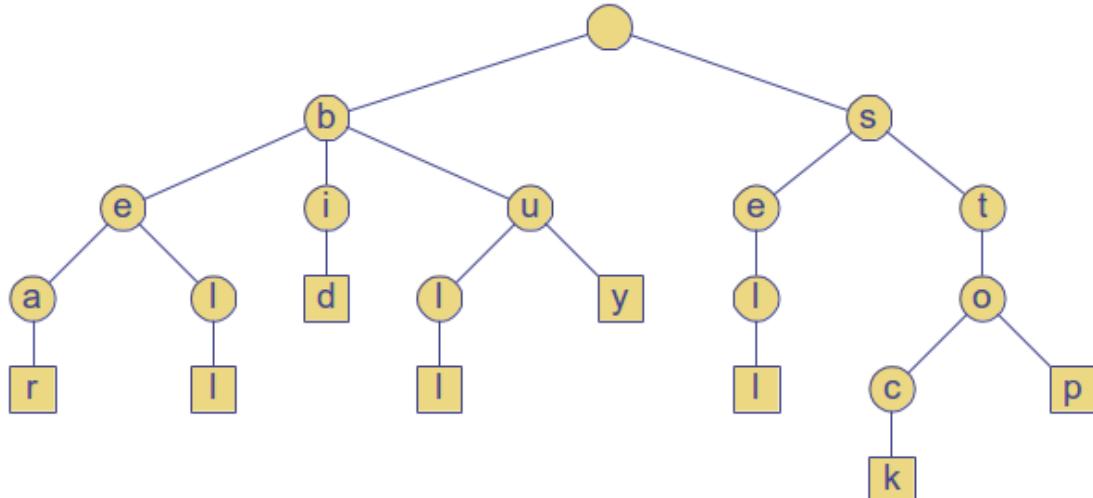
10.2 Tries

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 $O(m)$

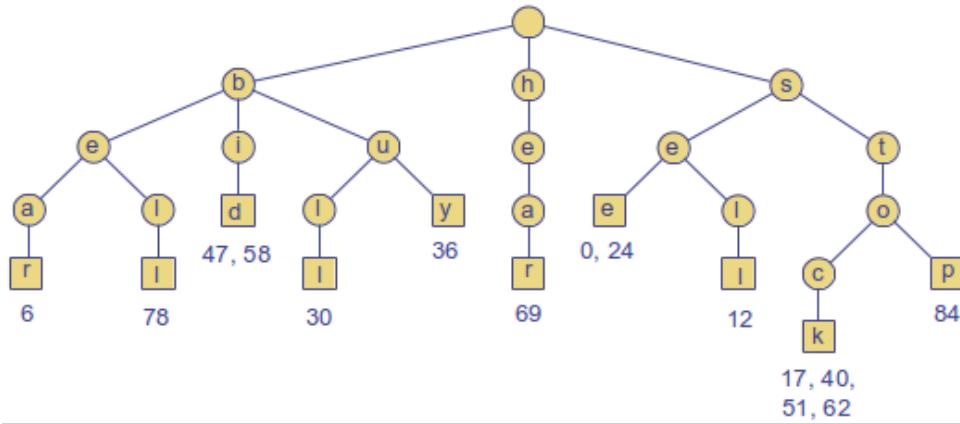
10.2.2 Standard Tries

- 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)!



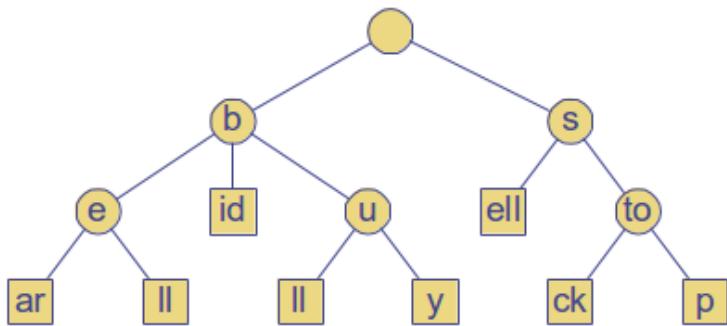
- 分析字典树:
 - n : S中字符串的数量;
 - m : 每次操作动用字符串参数的数量;
 - d : 字母表的size;
 - 空间复杂度 : $O(n)$
 - 搜索/插入/删除操作 : $O(dm)$
- word matching with a trie
 - 将文本插入trie;
 - 每次操作始于搜索:先查询首字母,然后一步步查询单词;
 - 到达叶结点还能给出这个单词的位置;

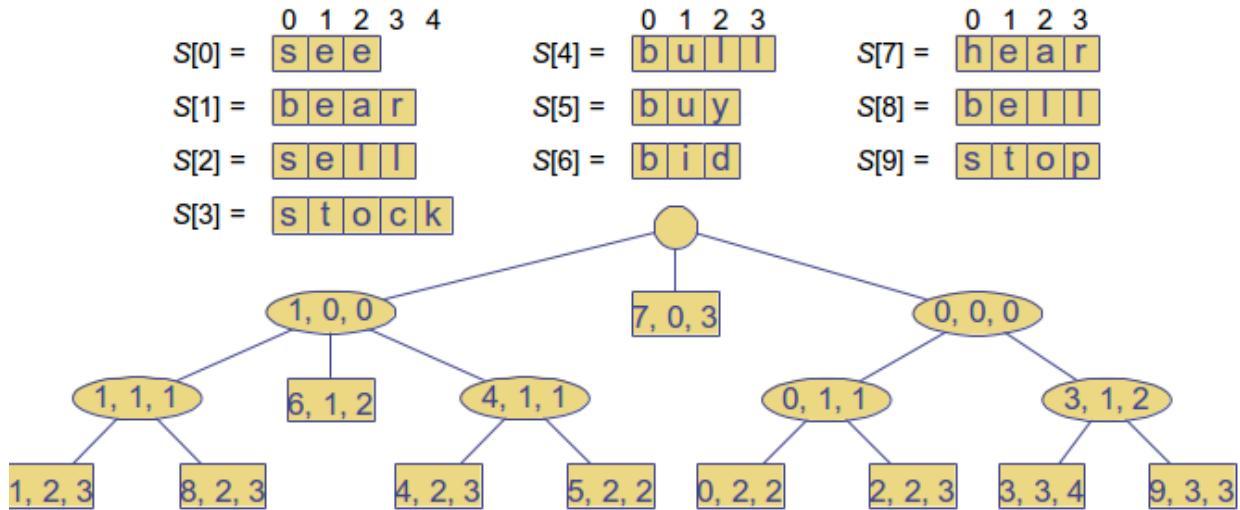
s	e	e	a	b	e	a	r	?	s	e	l	l	s	t	o	c	k	!					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s	e	e	a	b	u	l	l	?	b	u	y		s	t	o	c	k	!					
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	
b	i	d		s	t	o	c	k	!	b	i	d		s	t	o	c	k	!				
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68		
h	e	a	r		t	h	e	b	e	l	l	?	s	t	o	p	!						
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88				



10.2.3 Compressed Tries

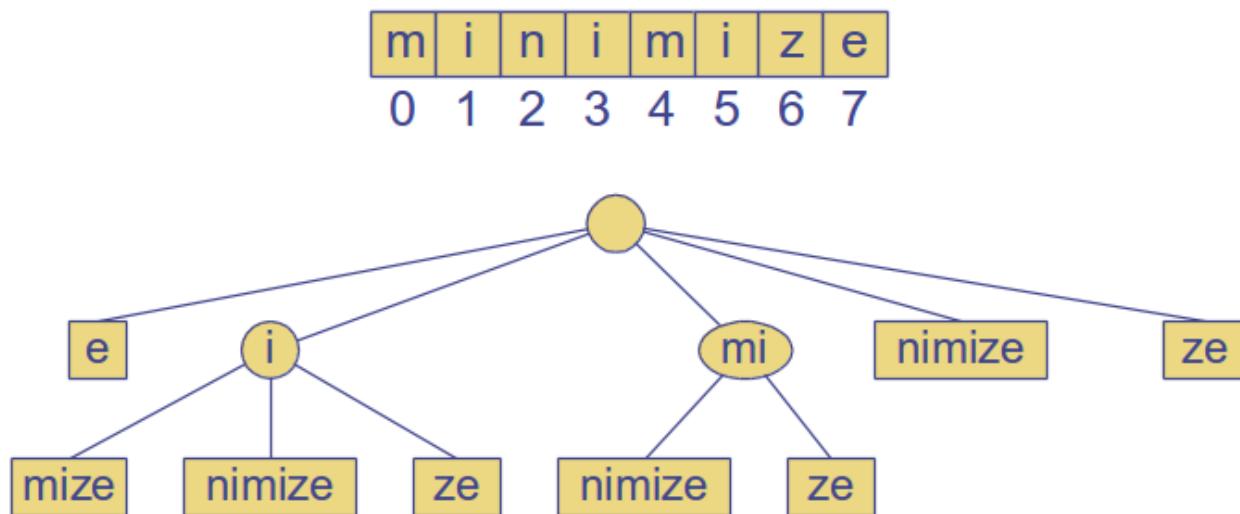
- 压缩多余结点;
 - 保证每个internal都有至少两个子结点;
 - 若某个internal只有一个子结点, 进行压缩;
 - 空间复杂度 $O(s)$, 其中S为array中的字符串数量



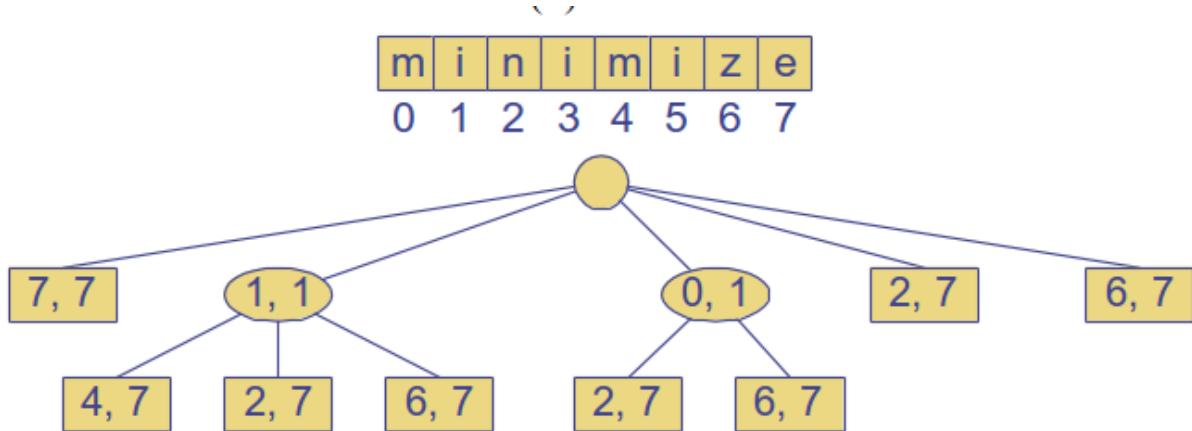


10.2.4 Suffix Trie后缀树

- 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:

```

1. Algorithm suffixTrieMatch
2. Input: Compact suffix trie T for a text X and pattern P;
3. Output: x中完全匹配的起始index,或者"P不是子字符串";
4.
5.     p=P.length;
6.     j=0;
7.     v=T.root();
8.     while(f==true || T.isExternal(v)) {
9.         f=true;
10.        for (each child w of v) {
11.            //已经匹配了j+1个字符
12.            i=start(w); //w的起始index(首字母)
13.            if(P[j]==T[i]){//对子树w进行处理
14.                x=end(w)-i+1;//end(w)->end index of w
15.                if(p<=x){//后缀<=Node label
16.                    if(P[j:j+p-1]==X[i:i+p-1]) return i-j;
17.                    else return "P不是x的子字符串";
18.                }
19.                else{
20.                    if(P[j:j+x-1]==X[i:i+x-1]){
21.                        p=p-x;//更新后缀长度
22.                        j=j+x;//更新后缀起始index
23.                        v=w;
24.                        f=false;
25.                        break for循环;
26.                    }
27.                }
28.            }
29.        }
30.    }
31.    return "P不是x的子字符串";
32. }
```

- 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;

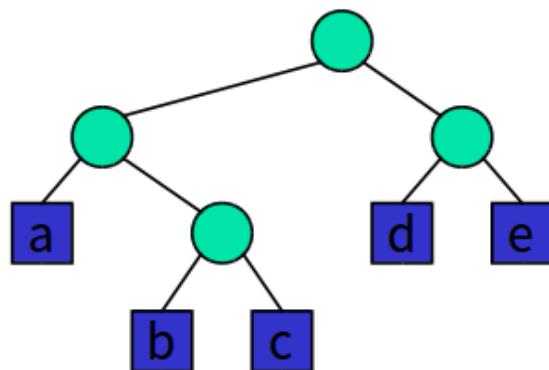
10.3.2 Text compression

- 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

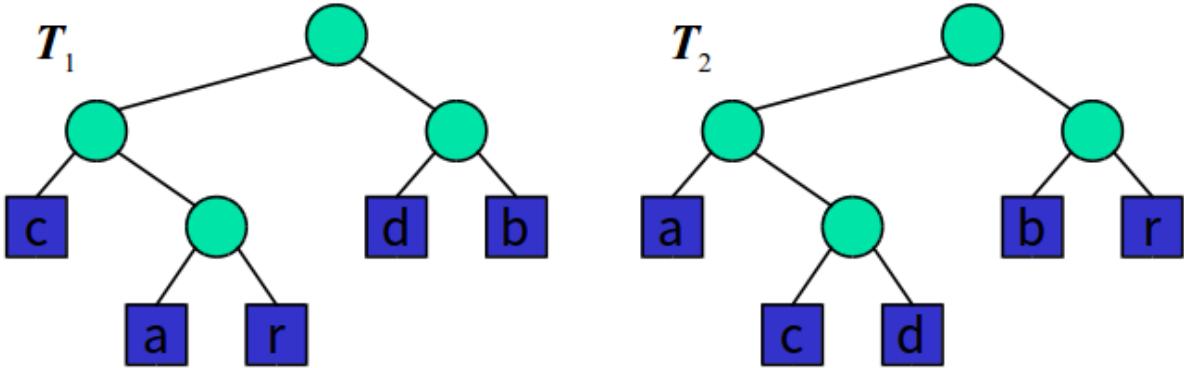
- code:将字母表内每个字母都转化为二进制编码;
- prefix code: code的一种;没有code-word是另一个code-word的前缀;
- encoding tree: 用于展示prefix code
 - 每个external储存一个字母;
 - 每个字母的code word取决于从root到叶结点的路径;
 - 0为左子树,1为右子树
 - 例如从 r o o t d某个字母的路径是左-右-左,则该字母的codeword为"010"

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;

- $X = \text{abracadabra}$
- 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

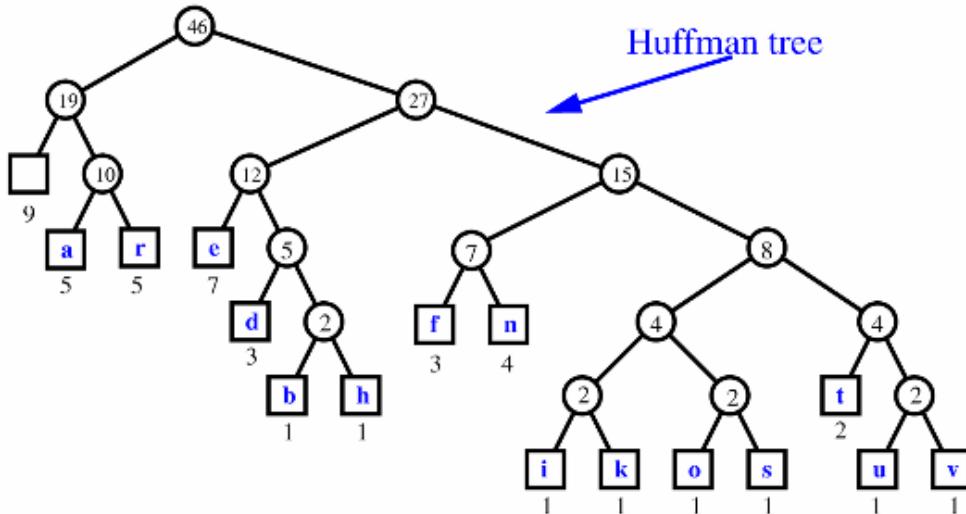
```

1. Algorithm HuffmanEncoding(X)
2. Input: String X of size n;
3. Output: optimal encoding trie for X; //最小化编码后的size
4.
5. C=distinctCharacters(X); //构建字母表
6. computeFrequencies(C,X); //计算字母表中每个元素的频率
7. Q=new empty heap;
8. for all c in C{
9.     T=new single-node tree storing c;
10.    Q.insert(getFrequency(c).T);
11. } //将每个元素及其频率以键值对形式插入Heap
12. while (Q.size()>1){ //构建heap
13.     f1=Q.minKey();
14.     T1=Q.removeMin();
15.     f2=Q.minKey();
16.     T2=Q.removeMin();
17.     //每次移出并merge两个最小的entry, 其key和为其父结点的key
18.     T=join(T1,T2);
19.     Q.insert(f1+f2,T); //合并后再插入Q中
20. }
21. return Q.removeMin();
22. }
```

- 算法分析:
 - 运行时间 $O(n + d \log d)$, 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 $\leq W$ 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: $\sum_{i \in S} x_i \leq W$

```

1. Algorithm fractionalKnapsack(S,W)
2. Input: set S(benefit bi, weight wi), max weight W;
3. Output: 返回得到最优化的xi;
4.
5.     for i in S{//初始化xi,value
6.         xi=0;
7.         vi=bi/wi;//value
8.     }
9.     w=0;//total weight
10.    while(w<W){
11.        remove item i with the largest vi;
12.        xi=min(wi,W-w);
13.        w=w+xi;
14.    }

```

15. }

- 算法分析: 运行时间 $O(n \log n)$

10.3.6 Task Scheduling

- 本课程不考察
- Given: set T with n tasks, 每个task包含
 - start time : s_i
 - 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

```
1. Algorithm taskSchedule(T)
2. Input: Set T of tasks (start time si, end time fi);
3. Output: 在时间表不冲突的情况下, 动用最少机器完成所有任务;
4. {
5.     m=0; // 初始化需要动用的机器数
6.     while (!T.isEmpty()) {
7.         remove si最小的task i; // 早开始的优先度高
8.         if (已有的某台机器j可以执行i) {
9.             将schedule i添加给machine j;
10.        }
11.        else {
12.            m=m+1;
13.            将schedule i添加给新机器 m;
14.        }
15.    }
16. }
```

- 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 $O(n \log n)$

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 $A_i * A_{i+1} * \dots * A_j$;
 - 使用 $N_{i,j}$ 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: $(A_0 * \dots * A_i) * (A_{i+1} * \dots * A_{n-1})$;
 - 最优解 $N_{0,n-1} = N_{0,i} + N_{i+1,n-1} + \text{time}$
 - 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:

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

- 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

```

1.   Algorithm matrixChain(S)
2.     Input: Sequence S of n matrices to be multiplied;
3.     Output: number of operations
4.   {
5.     for(i=1;i<n;i++) { // 初始化所有的Ni=0
6.       Ni=0;
7.
8.       for(b=1;b<n;b++) {
9.         for(i=0;i<n-b;i++) {
10.           j=i+b;
11.           Ni,j=+infinity;
12.           for(k=i;k<j;k++) {
13.             Ni,j=min{Ni,j,Ni,k+N(k+1,j)+di,d(j+1)};
}

```

```

14.         }
15.     }
16. }
17. }

```

- 算法分析:

- 先从最小的 $N_{0,0}$ 开始, 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;

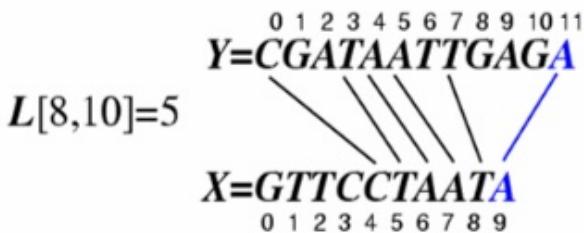
10.4.7 Subsequences:

- 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

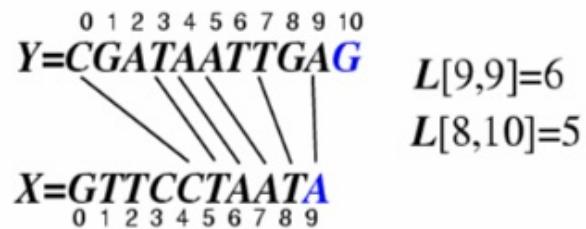
10.4.8 Longest Common Subsequence(LCS) Problem

- 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 $x_i == y_j$ (match), then $L[i,j] = L[i-1,j-1] + 1$;
 - else (not match), $L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$;

Case 1:



Case 2:

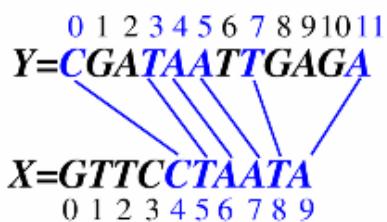


- LCS Algorithm $O(nm)$

```

1.   Algorithm LCS (X, Y)
2.   Input: Strings X (n) , Y (m) ;
3.   Output: LCS L[i,j] of X,Y;
4.   {
5.       //先初始化L=0
6.       for(i=0;i<n;i++) L[i,-1]=0;
7.       for(j=0;j<m;j++) L[-1,j]=0;
8.
9.       //开始匹配
10.      for(i=0;i<n;i++) {
11.          for(j=0;j<m;j++) {
12.              if (x==y) L[i,j]=1; //开头就匹配->L[0,0]=1
13.              else if (xi==yj) L[i,j]=L[i-1,j-1]+1;
14.              else L[i,j]=max{L[i-1,j],L[i,j-1]};
15.          }
16.      }
17.      return array L;
18.  }

```



Week 11 Graphs I

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

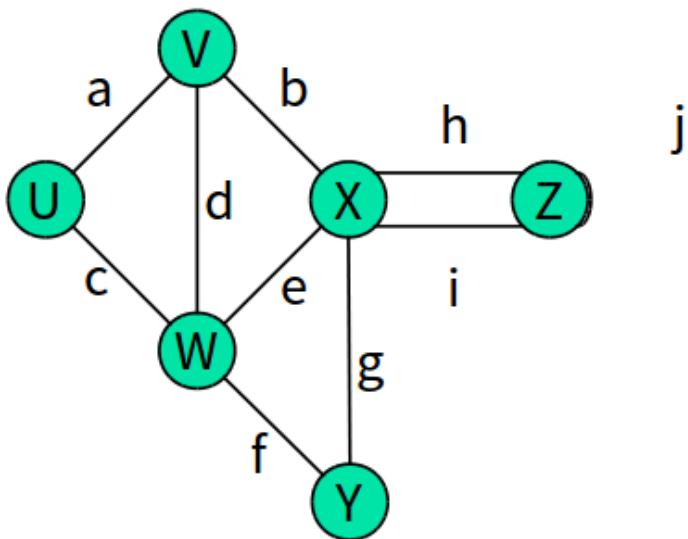
11.1 Graphs

- 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

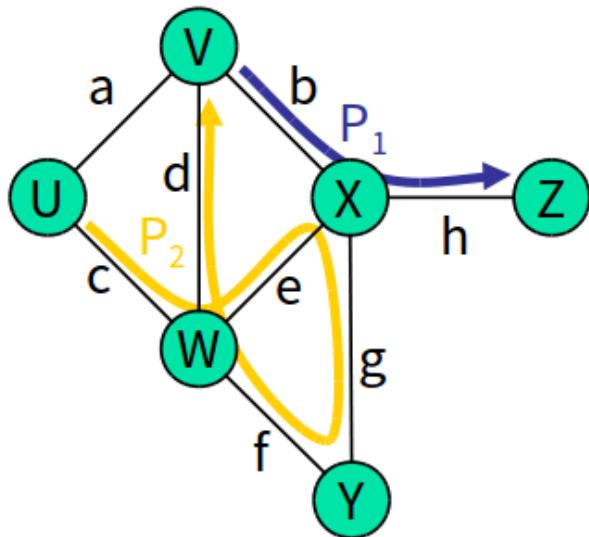
- Directed edge:
 - 有序顶点对(u,v);
 - 始于u,止于v,逆过程不成立;
 - 栗子 : flight;
- Undirected edge:
 - 无序顶点对(u,v),uv可互相访问;
 - 栗子 : flight route;
- Directed graph:
 - 所有的边都是directed edges;
 - 栗子 : route network;
- Undirected graph:
 - 所有的边都是undirected edges;
 - 栗子 : flight network;

11.1.2 Terminology

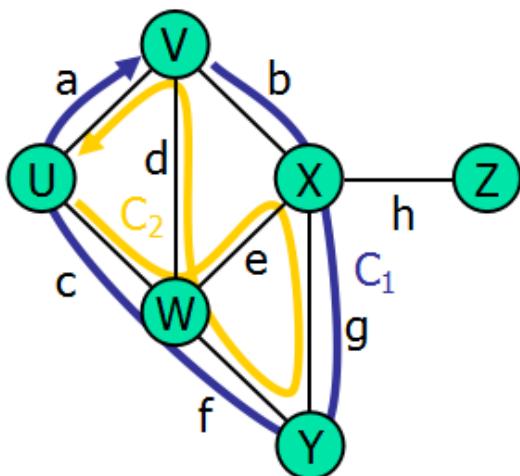


- End vertices(end points) of an edge : 某条边的两个端点;
 - endVertices(a) = (U,V)
- Adjacent vertices : 与某个端点间隔一条边的端点;

- $\text{opposite}(U, a) = V$
- Edge incident on a vertex : 与某个端点连接的所有边;
 - $\text{incidentEdges}(V) = (a, b, d)$
- Degrees of a vertex : 某个端点连接边的数量;
 - $\text{degree}(V) = 3$
- 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



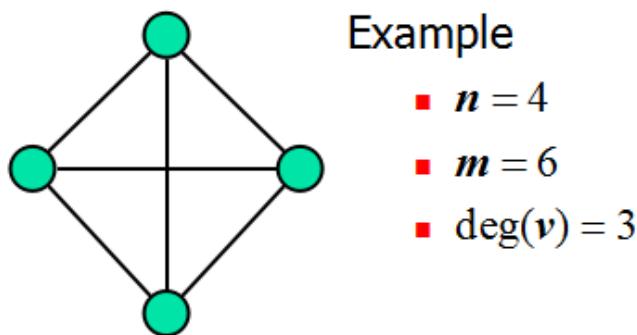
- Path : 从某个端点到另一个端点的路径;
 - U - W - Y - X - W - V
- Simple path : 经过的端点和边不重复;
 - V - X - Z



- 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:
 - $m \leq n(n - 1)/2$;
 - $\deg(v) \leq n - 1$;



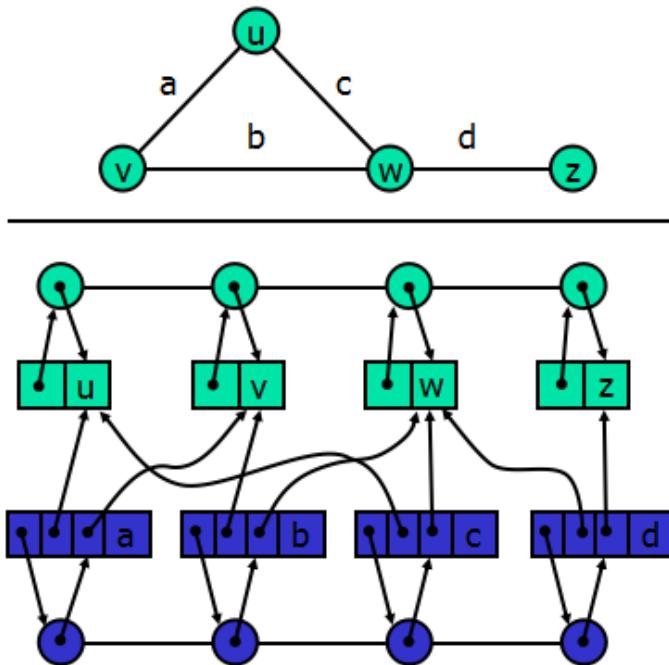
11.1.4 Graph ADT

- Accessor methods:
 - Vertex endVertices(Edge e) : 给定edge e, 返回两个vertices;
 - Vertex opposite(Vertex 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(Vertex 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;

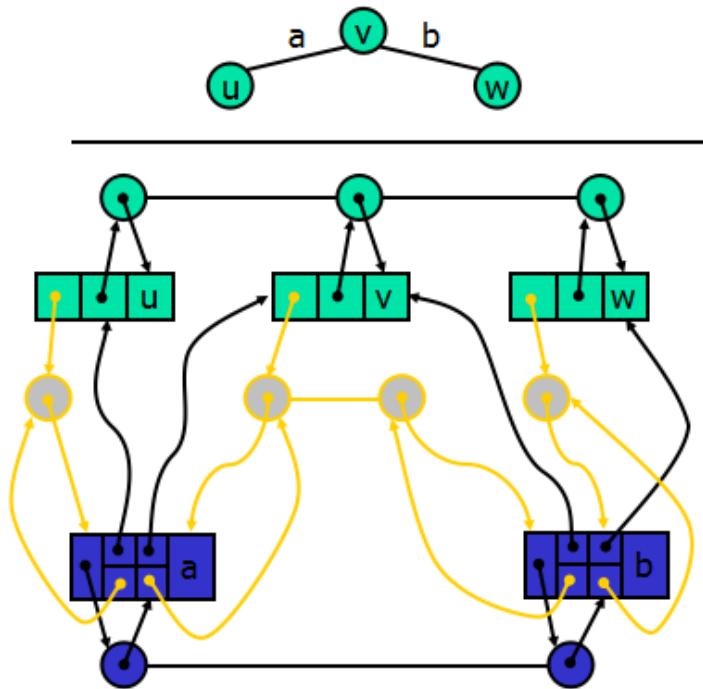
11.1.5 Edge List Structure

- 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;



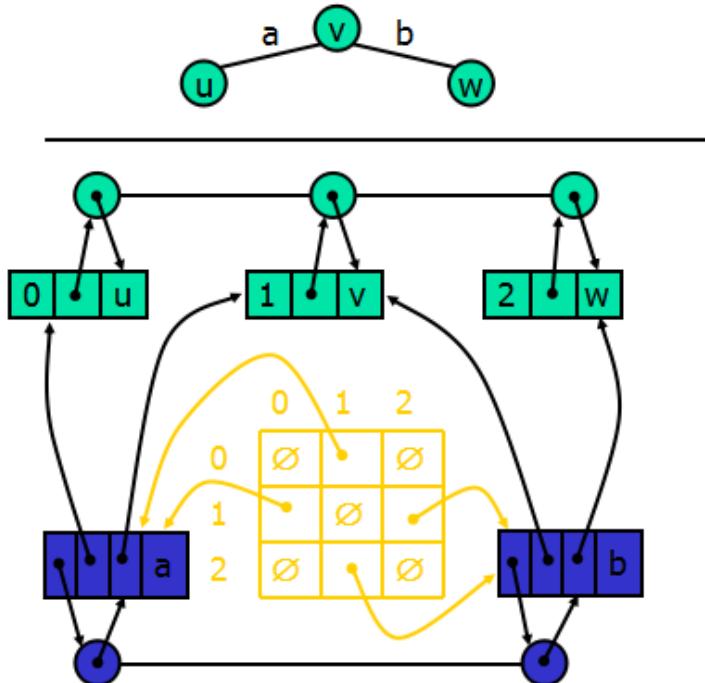
11.1.6 Adjacency List Structure

- 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 adjacency array:
 - reference to edge object for adjacent vertices;
 - Null for nonadjacent vertices;



11.1.8 三种结构的性能 Performance//

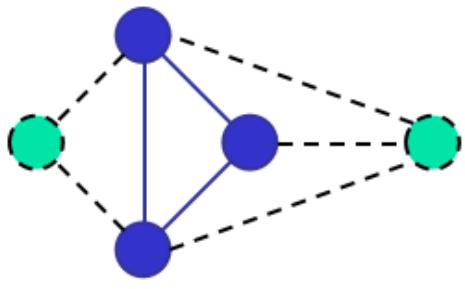
n vertices ; m edges ; 无平行边及自循环	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n + m)$	$O(n + m)$	$O(n^2)$
incidentEdges(v)	$O(m)$	$O(\deg(v))$	$O(n)$
areAdjacent(v,w)	$O(m)$	$O(\min(\deg(v), \deg(w)))$	$O(1)$
insertVertex(o)	$O(1)$	$O(1)$	$O(n^2)$
insertEdge(v,w,o)	$O(1)$	$O(1)$	$O(1)$
removeVertex(v)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

11.2 Depth-First Search

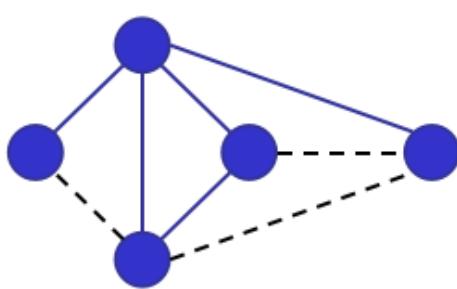
- Applications:
 - path finding;
 - cycle finding;

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



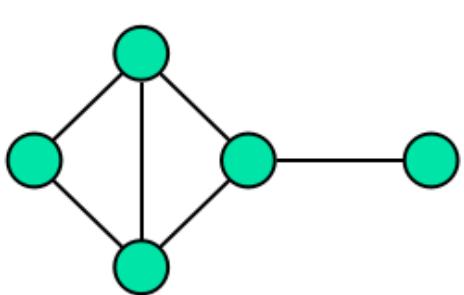
Subgraph



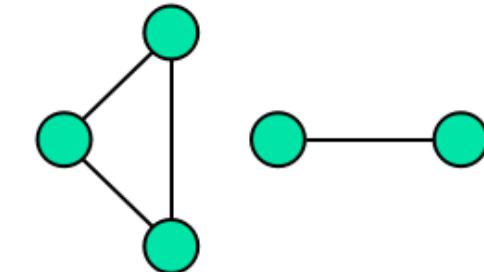
Spanning subgraph

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;



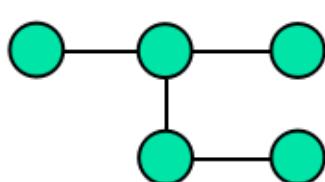
Connected graph



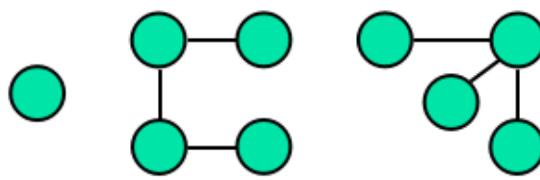
Non connected graph with two connected components

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;



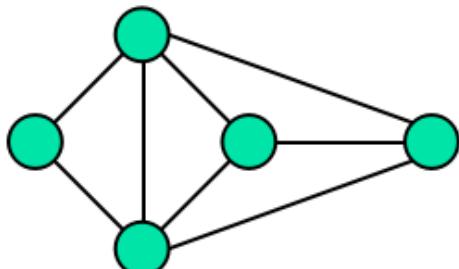
Tree



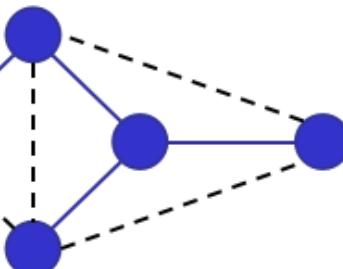
Forest

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)

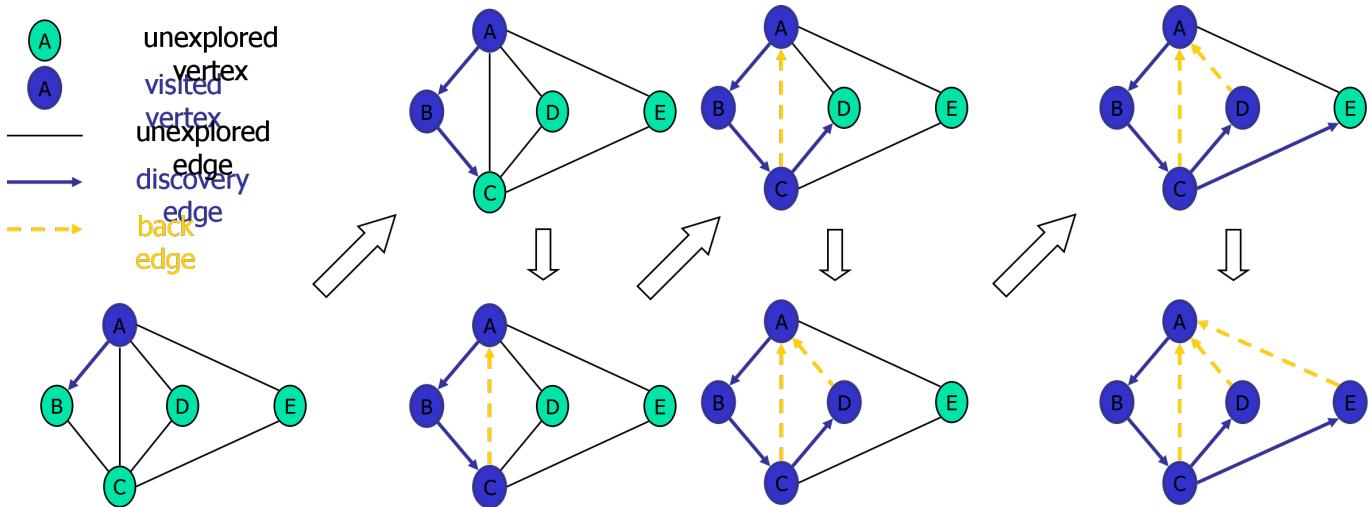
- 遍历图的过程:
 - 访问G所有的vetices、edges;
 - 确定G是否connected;
 - 给出G的connecgted components;

- 给出G的spanning forest;
- 时间复杂度 $O(n + m)$

11.2.6 DFS Algorithm

```

1. Algorithm DFS (G, v)
2. Input: graph G and a start vertex v;
3. Output: 给v的connected component的几条边做标记, 前进边discovery edges, 后退边back edge
4. s;
5. {
6.     setLabel (v, VISITED); //将起始端点设为VISITED
7.     for e in G.incidentEdge (v) { //e:与v相连的边
8.         if (getLabel (e) == UNEXPLORED) {
9.             w=opposite (v, e); //另一侧端点
10.            if (getLabel (w) == UNEXPLORED) {
11.                //边和另一侧端点都没有被探索->更改边标签
12.                setLabel (e, DISCOVERY); //注意此时点标签还是UNEXPLORED
13.                DFS (G, w); //递归
14.            }
15.            else{ //w已经被探索了
16.                setLabel (e, BACK);
17.            }
18.        }
19.    }
20.
21. Algorithm DFS (G)
22. Input:graph G;
23. Output:给G所有的边都加上标签
24. {
25.     //初始化端点和边的标签为"UNEXPLORED"
26.     for u in G.vertices () {
27.         setLabel (u, UNEXPLORED);
28.     }
29.     for e in G.edges () {
30.         setLabel (e, UNEXPLORED);
31.     }
32.     for v in G.vertices () { //对新端点调用前面的算法
33.         if (getLabel (v) == UNEXPLORED) DFS (G, v);
34.     }
35. }
```



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;

```

1. Algorithm pathDFS (G, v, z)
2. Input: Graph G, start vertex v, end vertex z;
3. Output: the path from v to z( elements in Stack S );
4. {
5.     setLabel (v,VISITED);
6.     S.push (v); //v入栈作为起点
7.     if(v==z) return S.elements(); //递归到最后就会得到这个结果
8.     for e in G.incidentEdges (v) {
9.         if (getLabel (e)==UNEXPLORED) {
10.             w=opposite (v,e);
11.             if (getLabel (w) ==UNEXPLORED) {
12.                 setLabel (e,DISCOVERY);

```

```

13.         S.push (e); //把e加入栈
14.         pathDFS (G, w, z); //递归
15.         S.pop (e); //返回e，并将e移出栈
16.     }
17.     else setLabel (e, BACK);
18. }
19. }
20. S.pop (v); //返回栈顶元素v，并将v移出栈
21. }

```

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;

```

1. Algorithm cycleDFS (G, v, z) {
2.     setLabel (v, VISITED);
3.     S.push (v); //v入栈作为起点
4.     if (v==z) return S.elements();
5.     for e in G.incidentEdges (v) {
6.         if (getLabel (e)==UNEXPLORED) {
7.             w=opposite (v, e);
8.             if (getLabel (w)==UNEXPLORED) {
9.                 setLabel (e, DISCOVERY);
10.                S.push (e); //把e加入栈
11.                pathDFS (G, w, z); //递归
12.                S.pop (e); //返回e，并将e移出栈
13.            }
14.        else{
15.            T=new empty stack;
16.            repeat{
17.                o=S.pop ();
18.                T.push (o);
19.            }
20.            until (o==w);
21.            return T.elements;
22.        }
23.    }
24.    S.pop (v);
25. }

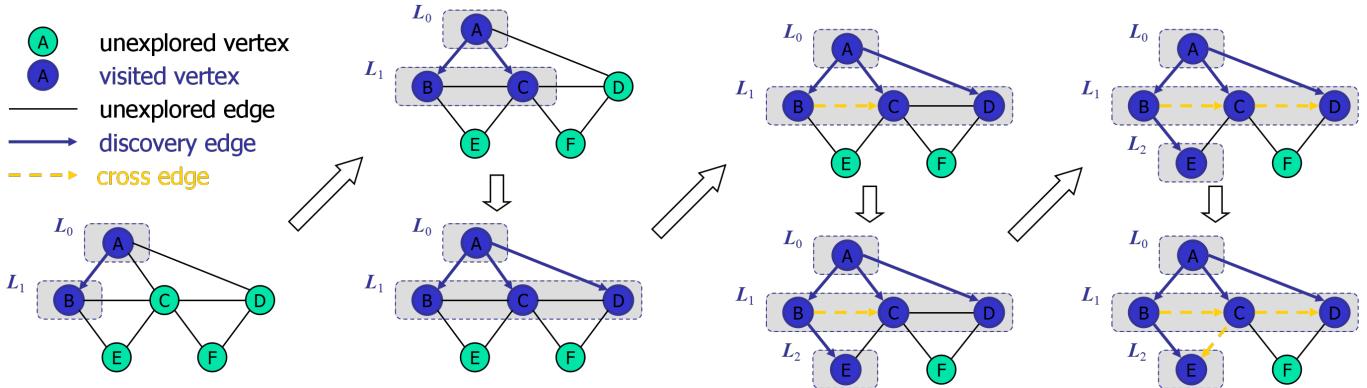
```

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

```
1.  Algorithm BFS (G, s) { //和DFS (G, v) 相比不用递归
2.      L0=new empty sequence;
3.      L0.insertLast (s);
4.      setLabel (s, VISITED) //起始端点
5.      i=0;
6.      while (-Li.isEmpty ()) {
7.          L(i+1)=new empty sequence;
8.          for v in Li.elements () {
9.              for e in G.incidentEdges (v) {
10.                  if (getLabel (e) ==UNEXPLORED) {
11.                      w=opposite (v, e);
12.                      if (getLabel (w) ==UNEXPLORED) {
13.                          setLabel (e, DISCOVERY);
14.                          setLabel (w, VISITED);
15.                          L(i+1).insertLast (w);
16.                      }
17.                      else setLabel (e, CROSS);
18.                  }
19.              }
20.          }
21.          i++;
22.      }
23.  }
24.
25. Algorithm BFS (G)
26. Input:graph G;
27. Output:给G所有的边都加上标签
28. {
29.     for u in G.vertices () {
30.         setLabel (u, UNEXPLORED);
31.     }
32.     for e in G.edges () {
33.         setLabel (e, UNEXPLORED);
34.     }
35.     for v in G.vertices () {
36.         if (getLabel (v) ==UNEXPLORED) BFS (G, v);
37.     }
38. }
```



- 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

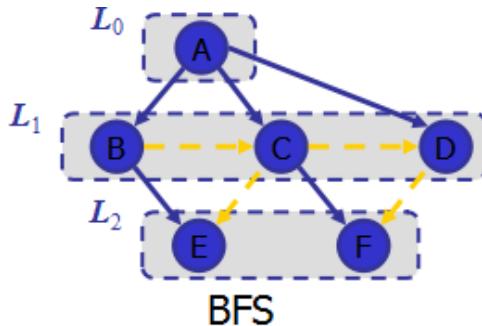
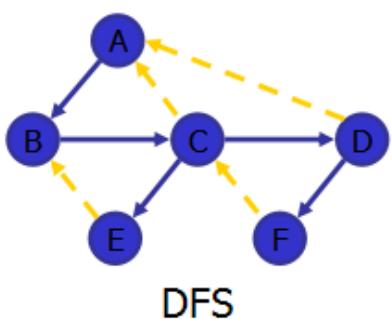
- G_s : connected component of s ;
- Properties:
 - BFS(G, s)访问所有的端点以及 G_s 的边;
 - DISCOVERY边构成 G_s 的spanning tree T_s ;
 - 对于 L_i 的每个端点 v :
 - T_s 中从 s 到 v 的路径有 i 条边;
 - G_s 中从 s 到 v 的路径至少 i 条边;
- Analysis:
 - set/get label : $O(1)$
 - vertex 被标记两次 : UNEXPLORED->VISITED;
 - edge 被标记两次 : UNEXPLORED->DISCOVERY/CROSS;
 - 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";

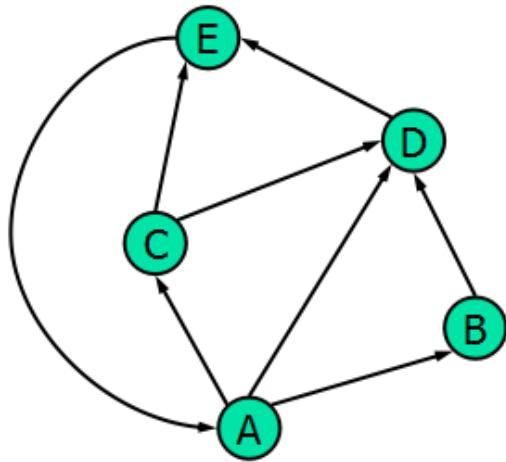
11.3.4 DFS VS BFS



39

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

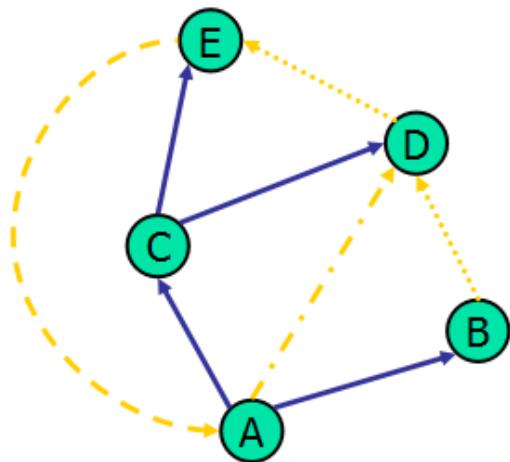
11.4 Directed Graphs: $G = (V, E)$



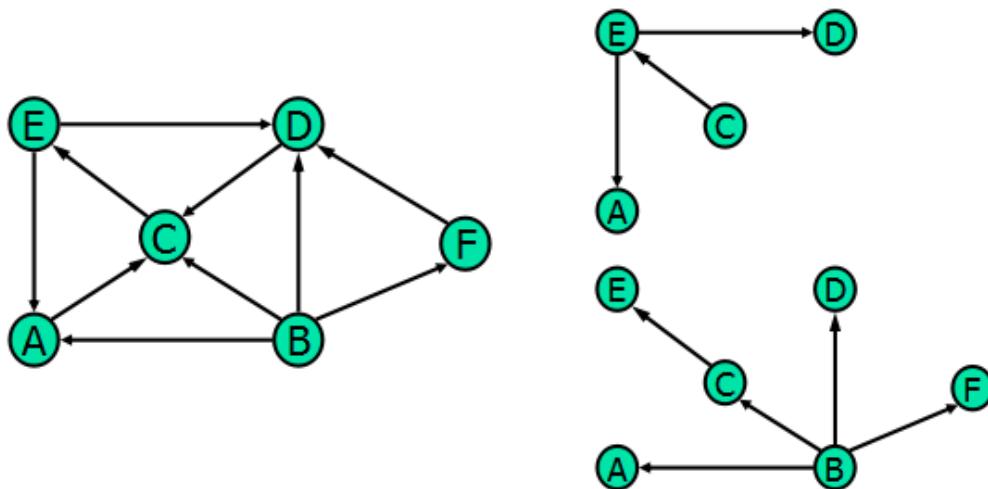
- Digraph : 所有的边都是有向的
- Applications:
 - 单行线;
 - flights;

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

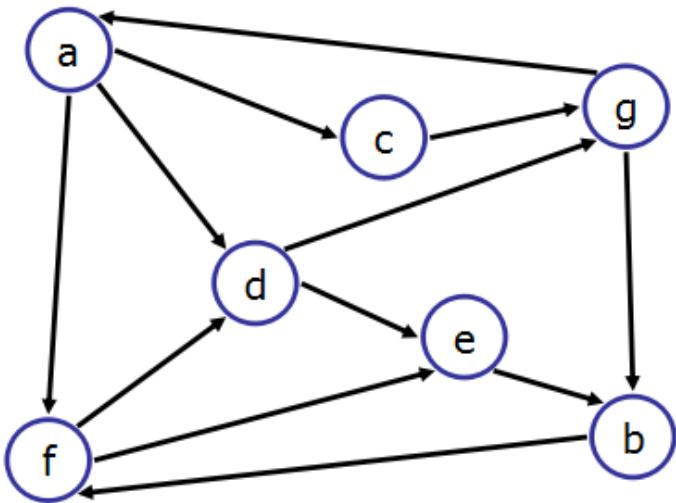
11.4.1 Directed DFS



- 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



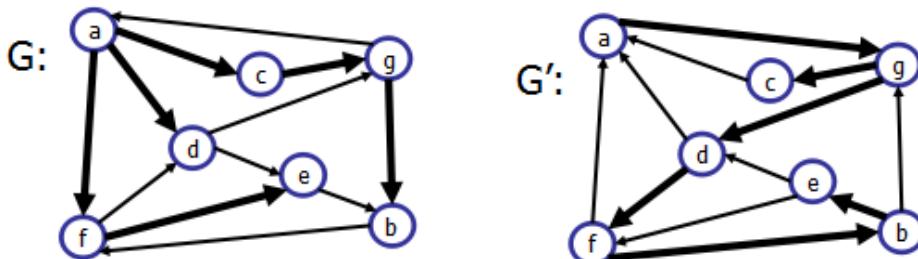
- 定义:每个端点都可以到达其余所有端点
- $\text{reachable}(v_i) = n - 1$

```

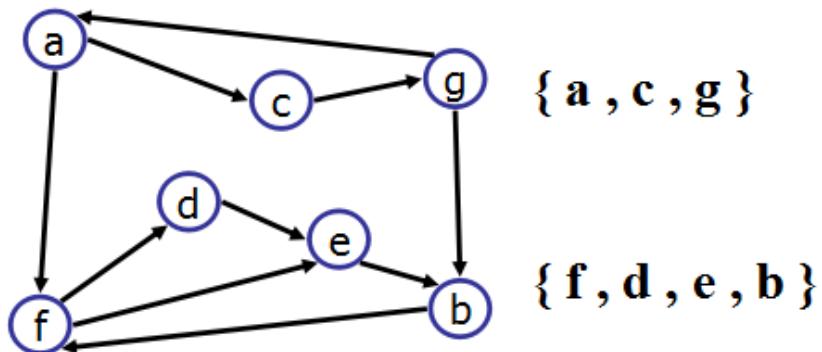
1. Algorithm IsStrongConnectivity
2. {
3.     Random pick a vertex in G;
4.     DFS(G, v);
5.     if (存在未被访问的端点w): return false
6.     G' = 端点与G相同,但所有边的方向相反
7.     DFS(G', v);
8.     if (存在未被访问的端点w): return false;
9.     else: return true;
10. }

```

- 分析:
 - 正遍历一次,反遍历一次;
 - $O(n + m)$

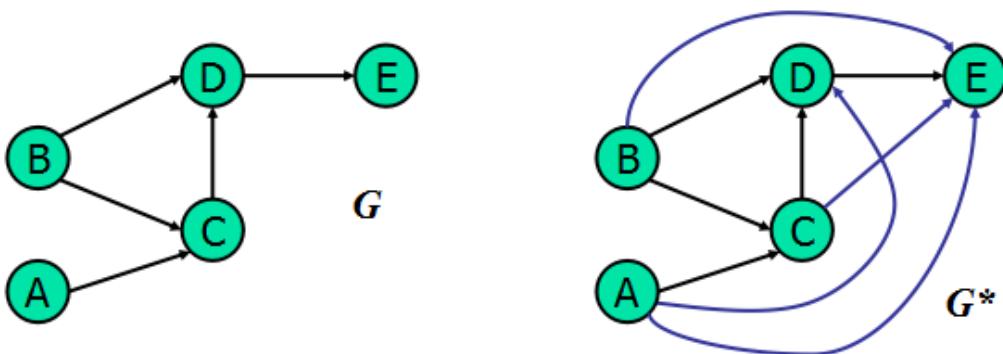


- Strong Connected Components:
 - 保证Strong connectivity的前提下得到的最大subgraph
 - 同样可以通过DFS得到,但更复杂!



11.4.3 Transitive Closure

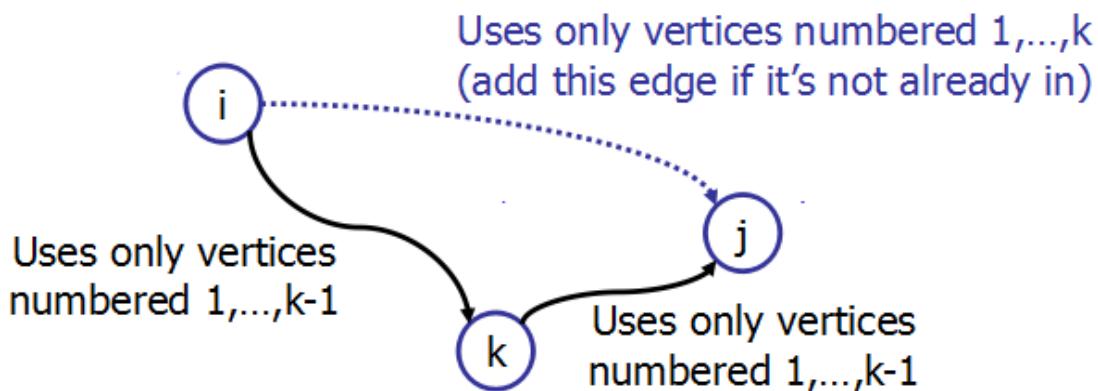
- 传递闭包可以为有向图提供可到达性信息
- 定义：给定有向图G，G的传递闭包 G^* 为满足以下条件的有向图
 - G^* 与G的端点数相同(spanning);
 - if $(u \rightarrow v)$ 为G中一个有向路径，则在 G^* 中 $(u \rightarrow v)$ 为一条有向边;
 - 其实就是给可到达的两个端点多连上一条线令其能直接到达



- 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

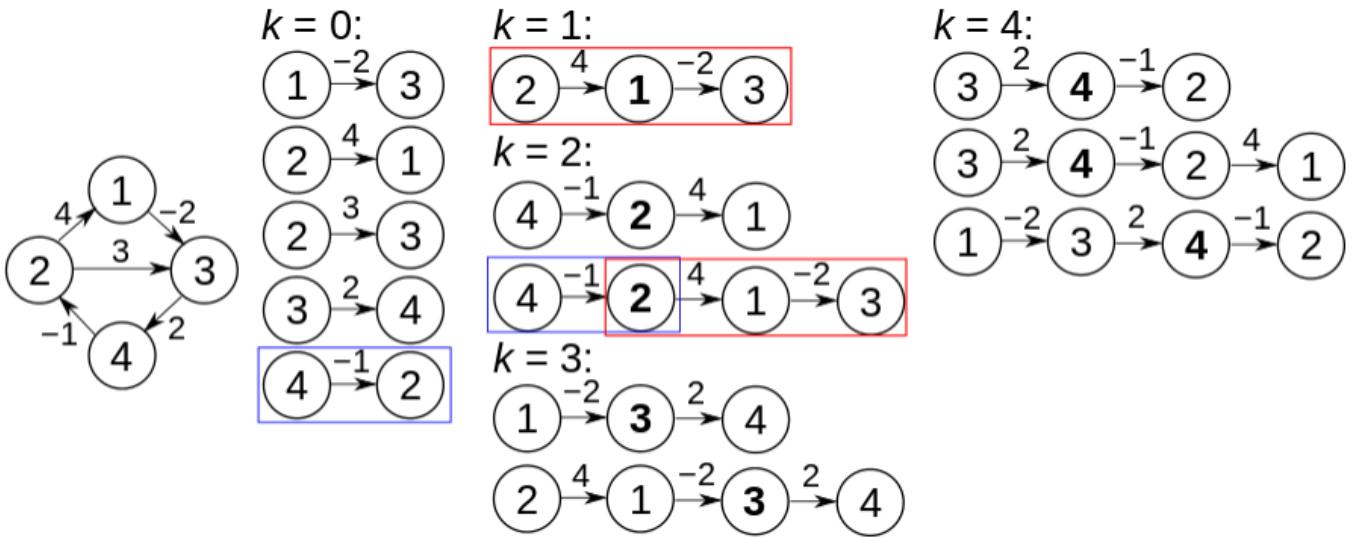


```

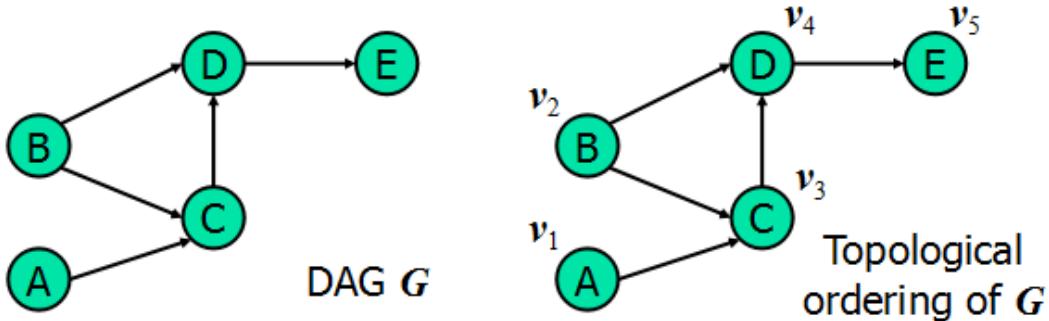
1. Algorithm FloydWarshall(G)
2. Input: digraph G;
3. Output: transitive closure  $G^*$  of G;
4.
5.     i = 1;
6.     for v in G.vertices() { // 将每个端点编号
7.         denote v as  $v_i$  ;
8.         i = i + 1;
9.     }
10.     $G_0 = G$ ;
11.    for ( $k$  in [1, n]) {
12.         $G_k = G_{k-1}$ ;
13.        for ( $i$  in [1, n]) and ( $i \neq k$ ) :
14.            for ( $j$  in [1, n]) and ( $j \neq i, k$ ) :
15.                if ( $G_{k-1}$ .areAdjacent( $v_i, v_k$ ) and ( $G_{k-1}$ .areAdjacent( $v_k, v_j$ )) :
16.                    if  $G_k$ .areAdjacent( $v_i, v_j$ ) :
17.                         $G_k$ .insertDirectedEdge( $v_i, v_j, k$ );
18.                }
19.            return  $G_n$ 
20.    }

```

- 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 $(1 \dots k)$ 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

- $O(n + m)$

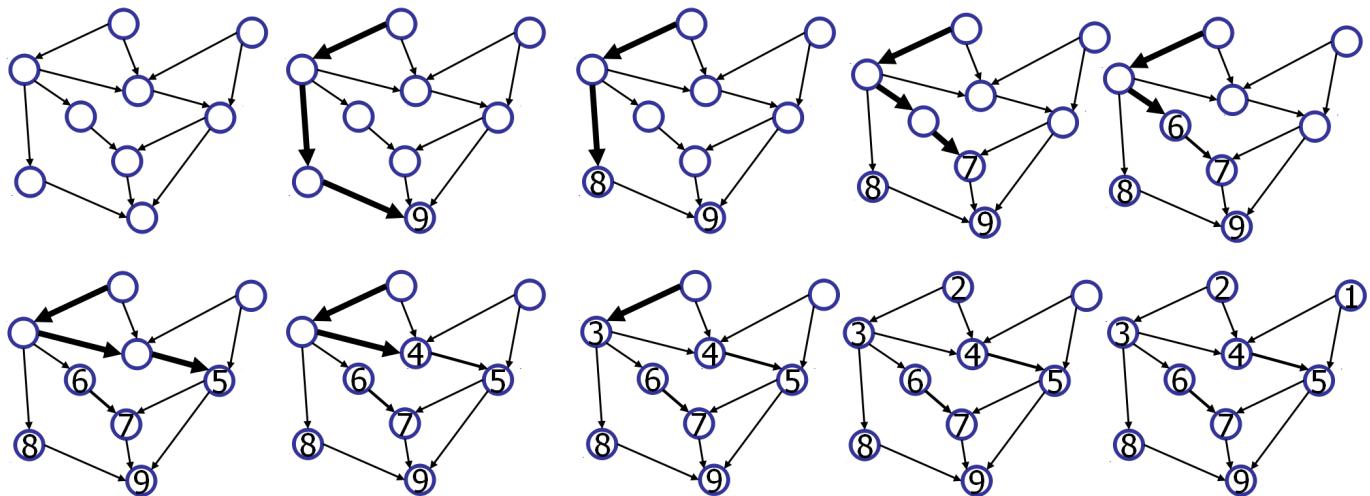
```

1. Method TopologicalSort (G)
2. {
3.     H=G; //先创建一个副本
4.     n=G.numVertices ();
5.     while (!H.isEmpty ()) {}
6.         令v作为一个没有outgoing edges 的端点;
7.         Label v=n;
8.         n--;
9.         H.remove (v);
10.    }
11. }
```

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

```
1. Algorithm topologicalDFS (G,v)
2. Input: graph G and a start vertex v in G;
3. Output: 给v所有的connected component加上标签;
4.
5.     setLabel (v,VISITED);
6.     for e in G.incidentEdges (v) {
7.         if (getLabel (e)==UNEXPLORED) {
8.             w=opposite (v,e);
9.             if (getLabel (w)==UNEXPLORED) {
10.                 setLabel (e,DISCOVERY);
11.                 topologicalDFS (G,w);
12.             }
13.         }
14.     }
15.     Label v with topological index n;//替换原DFS的BACK Label
16.     n--;
17.     return;
18. }
19.
20. Algorithm topologicalDFS (G)
21. Input: digraph G;
22. Output: topological ordering of G;
23.
24.     n=G.numBertices ();
25.     for u in G.vertices () {
26.         setLabel (u,UNEXPLORED);
27.     }
28.     for e in G.edges () {
29.         setLabel (e,UNEXPLORED);
30.     }
31.     for v in G.vertices () {
32.         if (getLabel (v)==UNEXPLORED) {
33.             topologicalDFS (G,v);
34.         }
35.     }
36. }
```



Week 12 Graphs II

- Shortest Paths
- Minimum Spanning Trees

12.1 Shortest Paths

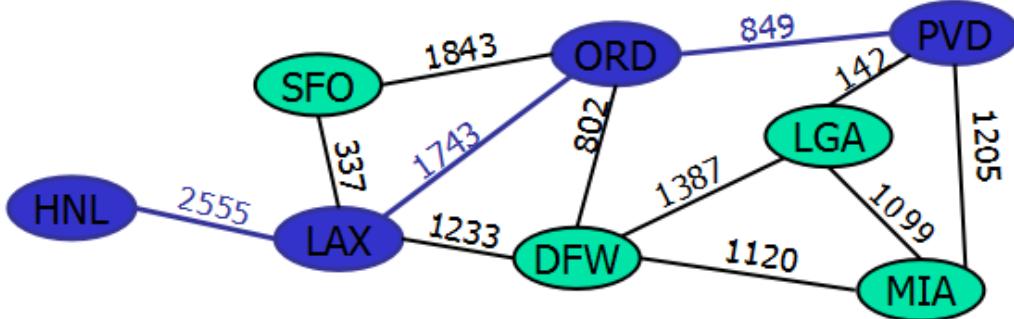
- 之前的路径算法包括BFS/DFS/Floyd-Warshall

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

- $\text{distance}(s,v)$: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 $D(v_i)$;

- 预设:

- connected graph;
- undirected edges;
- nonnegative edge weight;

- At each step:

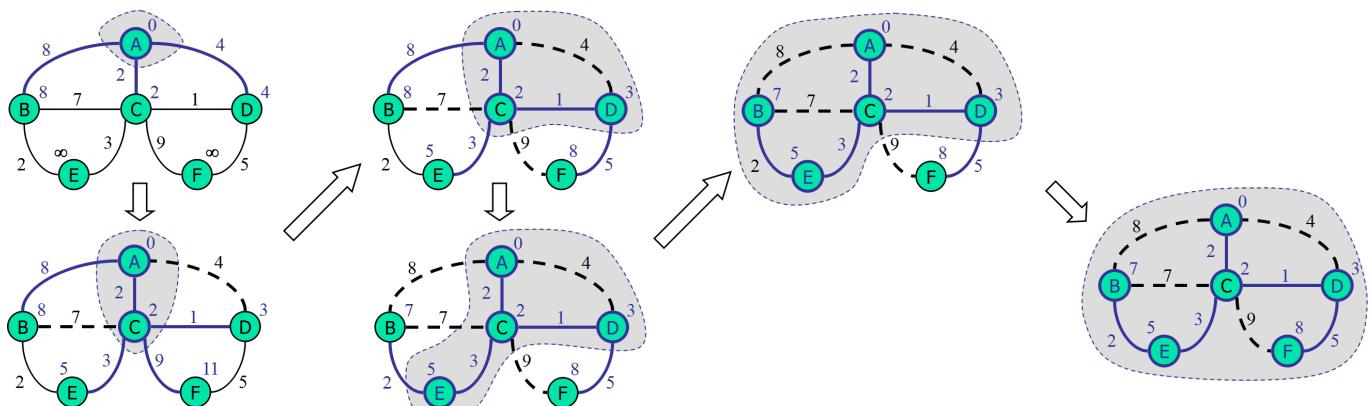
- 计算s到端点 v_i 的距离d,加标签 $D(v_i) = d$,并将该端点加入输出集合;
- 计算与 v_i 相邻且还未加入输出集合的点;
- 基于edge relaxation

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

$$D(z) = \min(D(z), D(u) + \text{weight}(e))$$

- 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

```

1. Algorithm DijkstraDistance(G, s)
2. Input:
3.     simple undirected weighted G;
4.     weight >= 0;
5.     distinguished vertex s;
6. Output: label D[u] -> s到u的距离 (u为G中任意一点)
7. {
8.     for v in G.vertices() { //初始化每个端点的距离label
9.         if (v==s) {
10.             D[v]==0;
11.         }
12.         else D[v]=infinity;

```

```

13.     }
14.
15.     Create a priority queue Q://需要里面的removeMin()
16.     element=vertex v in G;
17.     key=D[v];
18.
19.     while (!Q.isEmpty()) {
20.         u=Q.removeMin();
21.         for z in Q that is adjacent to u{//relaxation
22.             D[z]=min{D[z], D[u]+w((u,z))};
23.             在Q中更新z的key为D[z];
24.         }
25.     }
26.     return label D[u] for all u;
27. }
```

- 算法分析:

- 创建优先队列Q:
 - adaptable PQ: $O(n \log n)$;
 - bottom-up heap construction: $O(n)$;
- while循环的每一步:
 - $Q.removeMin()$: $O(\log n)$
 - relaxation : $O(\deg(u) \log n)$
 - while循环总共需要 : $O((n + m) \log n)$
- 整个算法: $O(m \log n)$
- 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

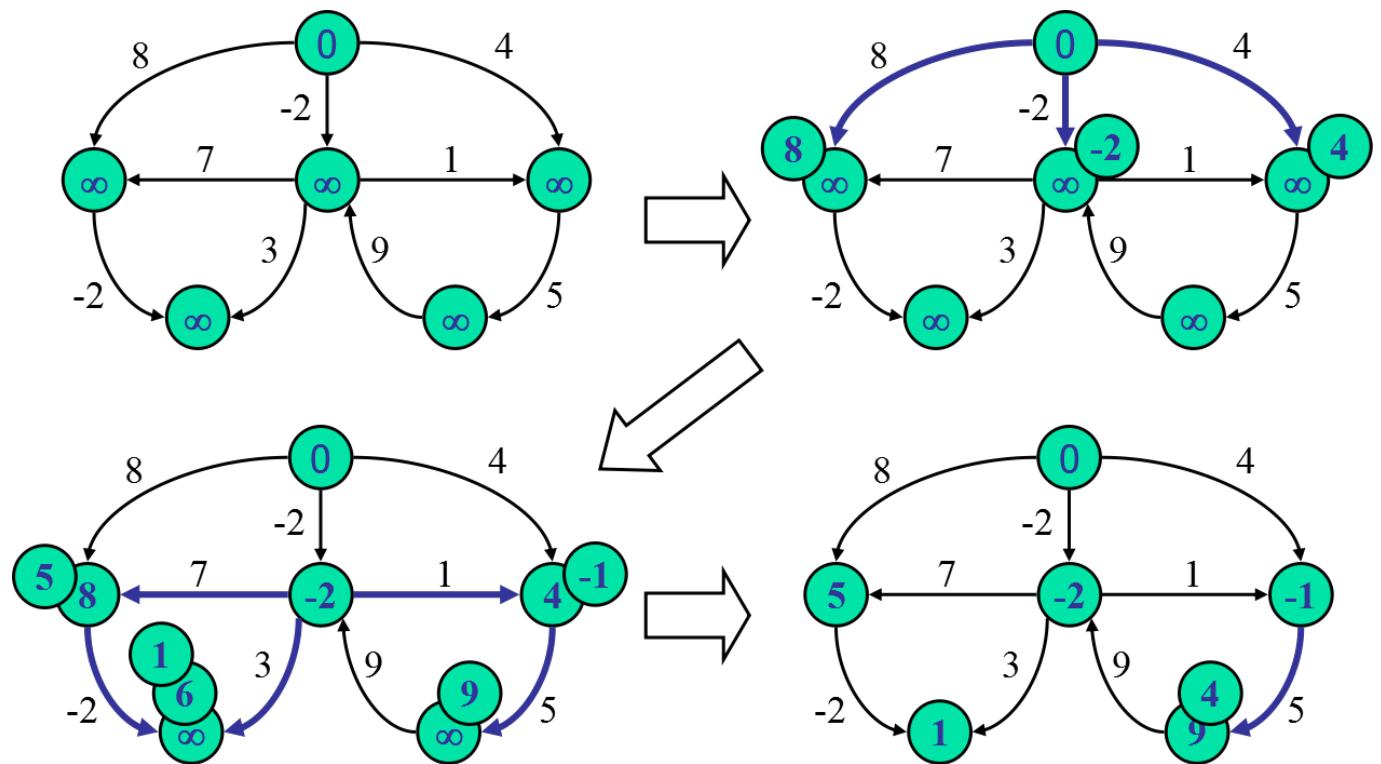
- 本课程不考察
- 即使边的权重是负的也能运行
- 要求G是有向图，运行时间 $O(mn)$ 长于Dijkstra，但可计算负权重

```

1. Algorithm BellmanFordDistance(G,s)
2. {
3.     for v in G.vertices() {
4.         if (v==s) D[v]=0;
5.         else D[v]=infinity;
6.     }
7.     for(i=1;i<n;i++) {
8.         for e in G.edges() {
9.             u=G.origin(e); //有向边的起始端点
10.            z=G.opposite(u,e); //该边的终止端点
11.            r=D[u]+weight(e);
12.            D[z]=min{D[z], r};
```

```

13.         }
14.     }
15. }
```

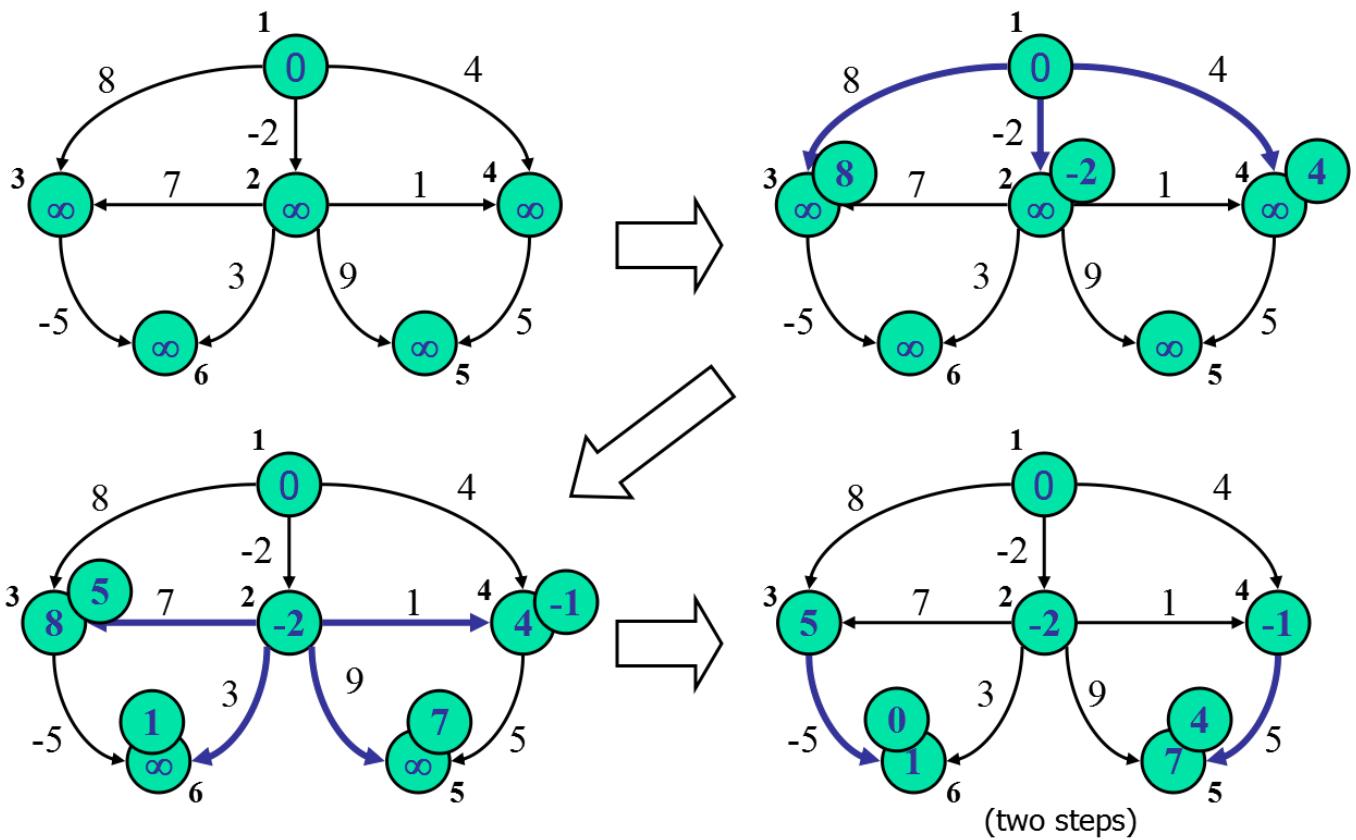


12.1.7 DAG-based Algorithm

- 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

```

1. Algorithm DagDistance(G, s)
2. {
3.     for v in G.vertices() {
4.         if (v==s) D[v]=0;
5.         else D[v]=infinity;
6.     }
7.     Perform a topological sort of the vertices;
8.     for(u=1;u<=n;u++) {
9.         z=G.opposite(u,e);
10.        r=D[u]+weight(e);
11.        D[z]=min{D[z], r};
12.    }
13. }
```

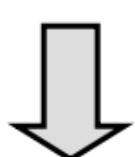
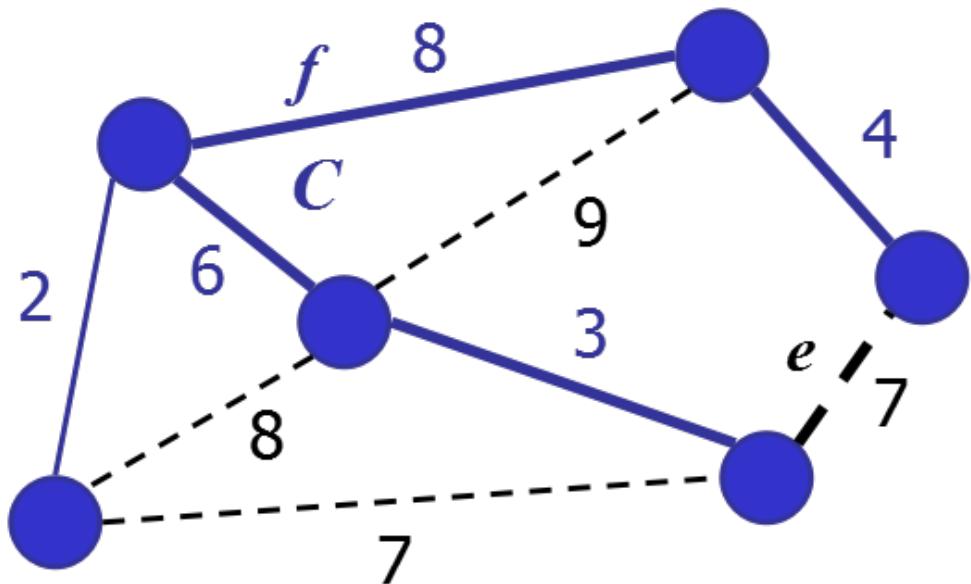


12.2 Minimum Spanning Trees(MST)

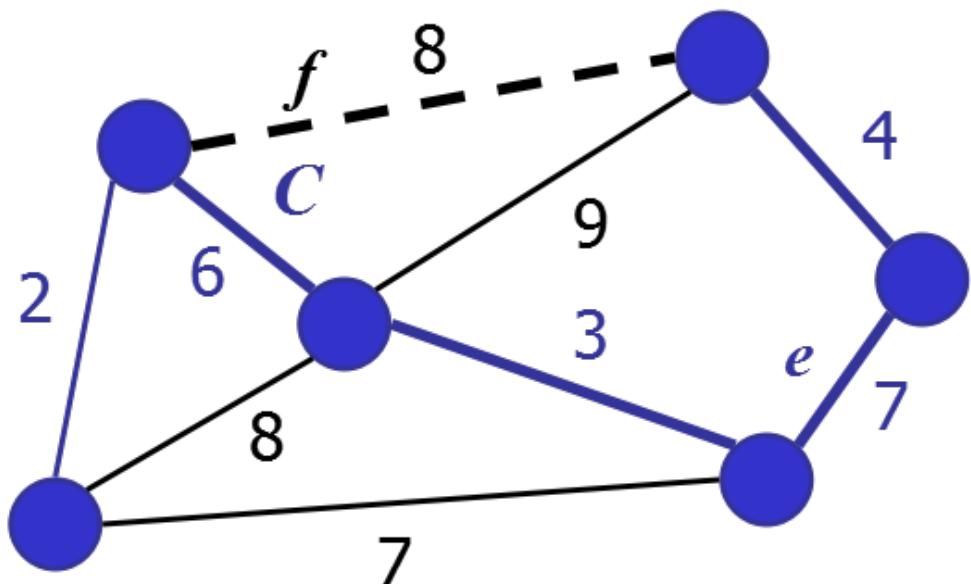
- Spanning tree: 包括了图中所有的点, 边组合可变
- MST: 找出边权重加和最小的spanning tree;
- Applications : 通讯网和运输网

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 $\text{weight}(f) \leq \text{weight}(e)$
 - 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

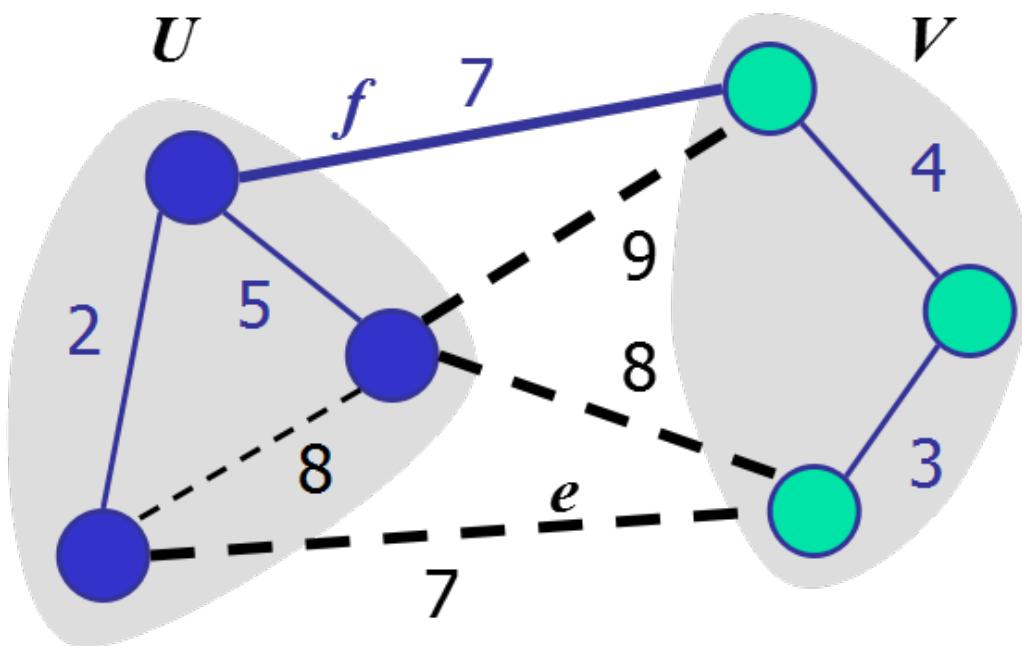


Replacing f with e yields
a better spanning tree

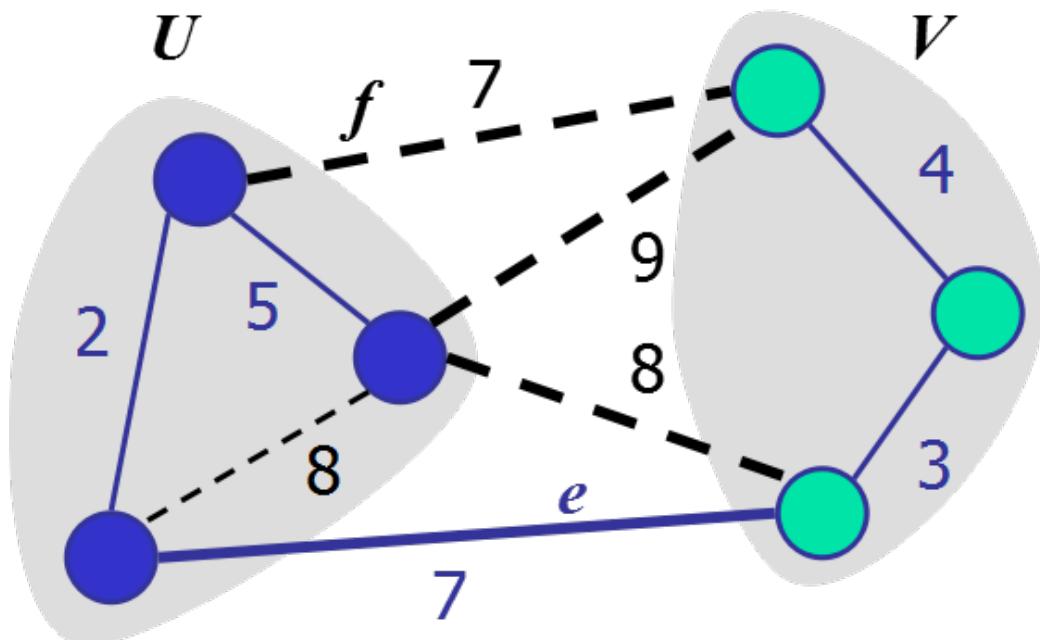


12.2.2 Partition Property

- 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 ;



Replacing f with e yields another MST



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;

```

1. Algorithm KruskalMST(G) {
2.     for v in G.vertices() {
3.         define a Cloud(v) of {v};
4.     }
5.
6.     Create priority queue Q://储存未被加入输出集的所有边
7.     element:edge e in G;
8.     key:weight;
9.
10.    T=null;//初始化输出集MST为0
11.
12.    while (T.edges().length()<n-1) {//注意MST边数为n-1
13.        edge e =T.removeMin();
14.        u,v=endpoints(v);
15.        if(Cloud(v) !=Cloud(u) {
16.            T.insertLast(e);
17.            Merge Cloud(v) and Cloud(u);
18.        }
19.    }
20.    return T;
21. }

```

- 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 $O(\log n)$
- 基于Partition的Kruskal算法实现: $O((m + n)\log n)$
- 这里先略过栗子

```

1. Algorithm KruskalMST(G)
2. Input: weighted graph G;
3. Output: MST T for G;
4. {

```

```

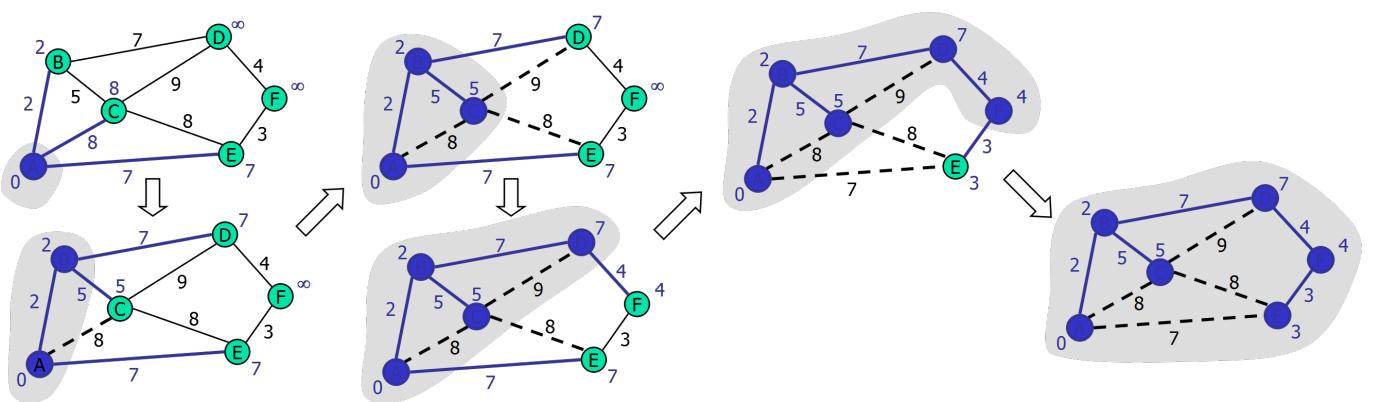
5.      P:partition of vertices in G,每个端点都是一个set;
6.      Q:PQ储存edges in G,sorted by key(weight);
7.      T:initially empty tree;
8.      while(!Q.isEmpty()) {
9.          (u,v)=Q.removeMin().element();
10.         if(P.find(u)!=P.find(v)) {
11.             Add e(u,v) to T;
12.             P.union(u,v);
13.         }
14.     }
15.     return T;
16. }
```

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;

```

1.  Algorithm PrimJarnikMST (G)
2.  {
3.      //初始化起始点v,剩余点u,T
4.      randomly pick any v of G;
5.      D[v]=0;
6.      for ((u in G.vertices()) && (u!=v)) D[u]= infinity;
7.      T=null;
8.      Create priority queue Q storing each vertex u:
9.          entry((u,null),D[u]);
10.         element=(u,null);
11.         key=D[u];
12.
13.      while(!Q.isEmpty()) {
14.          (u,e)=Q.removeMin();
15.          T.insertLast(vertex u,edge e); //每次拿出最小元素加入T
16.          for (z in Q) && (areAdjacent(z,u)) {
17.              if(w((u,z))<D[z]) {
18.                  D[z]=w((u,z));
19.                  z.setElement(z,(u,z)); //更新端点z在Q中的element
20.                  z.setKey(D[z]); //更新端点z在Q中的key
21.              }
22.          }
23.      }
24.      return T;
25. }
```



- 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: $O(\log n)$;
 - 端点w的key每次变化耗时 $O(\log n)$, 变化次数最多 $O(\deg(w))$;
- 总运行时间:
 - 如果G的结构是adjacency list structure : $O((n + m)\log n)$;
 - 如果G是连通图 : $O(m\log n)$;