DATA STRUCTURE

Author: Xingwei Chen

数据结构 JAVA

---

1 Java Primer

1.1 Base type

boolean	a boolean value: true or false
char	16-bit Unicode character
byte	8-bit signed two's complement integer
short	16-bit signed two's complement integer
int	32-bit signed two's complement integer
long	64-bit signed two's complement integer
float	32-bit floating-point number (IEEE 754-1985)
double	64-bit floating-point number (IEEE 754-1985)

Example for define a base type:

```
1  boolean flag = true;
2  boolean verbose, debug;              // two variables declared, but not yet initialized
3  char grade = 'A';
4  byte b = 12;
5  short s = 24;
6  int i, j, k = 257;                   // three variables declared; only k initialized
7  long l = 890L;                       // note the use of "L" here
8  float pi = 3.1416F;                  // note the use of "F" here
9  double e = 2.71828, a = 6.022e23;    // both variables are initialized
```

1.2 Object

Classes are known as **reference types** in Java, and a variable of that type is known as a **reference variable**. A reference variable is capable of storing the location (i.e., **memory address**) of an object from the declared class.

NOTE: A reference variable can also store a special value **null** 当有多个reference variable指向同一个对象时, 当其中一个改变了对象的值时, 所有的reference variable也会改变。

如果再constructor方法中没有指定变量初始化的值则将变量初始化成default values
The default values are null for reference variables and 0 for all base types except Boolean variables (which are false by default).

一个method的signature是有由method本身的名字和输入的变量个数和种类共同决定的。

The new operator returns a **reference** to the newly created instance

1.2.1 区分public, protected, private

Public class modifier designates that all classes may access the defined aspect.
Protected class modifier designates that access to the defined aspect is only granted to subclass 和同一个package下的classes.
Private class modifier designates that access to a defined member of a class be granted only to code within that class. Subclass和同一package下的class也不能
当没有特别指定时, 自动设为package-private class level

1.2.2 Static

Static variables用来存储 ' global' 的变量(即整个class都可以使用的变量)
Static method可以被直接调用，如Math.sqrt(2)
NOTE: Static methods can be useful for providing utility behaviors related to a class that need not rely on the state of any particular instance of that class.

1.2.3 Abstract

在使用抽象类时需要注意几点:

1. 抽象类不能被实例化，实例化的工作应该交由它的子类来完成，它只需要有一个引用即可。
2. 抽象方法必须由子类来进行重写。
3. 只要包含一个抽象方法的抽象类，该方法必须要定义成抽象类，不管是否还包含有其他方法。
4. 抽象类中可以包含具体的方法，当然也可以不包含抽象方法。
5. 子类中的抽象方法不能与父类的抽象方法同名。
6. abstract不能与final并列修饰同一个类。
7. abstract 不能与private、static、final或native并列修饰同一个方法.

Example:
定义一个抽象动物类Animal，提供抽象方法叫cry()，猫、狗都是动物类的子类，由于cry()为抽象方法，所以Cat、Dog必须要实现cry()方法。

```
1.  public abstract class Animal {
2.      public abstract void cry();
3.  }
4.
5.  public class Cat extends Animal{
6.
7.      @Override
8.      public void cry() {
9.          System.out.println("猫叫：喵喵...");
10.     }
11. }
12.
13. public class Dog extends Animal{
14.
15.     @Override
16.     public void cry() {
17.         System.out.println("狗叫:汪汪...");
18.     }
19.
20. }
21.
22. public class Test {
23.
24.     public static void main(String[] args) {
25.         Animal a1 = new Cat();
26.         Animal a2 = new Dog();
27.
28.         a1.cry();
```

```
29.          a2.cry();
30.      }
31. }
32.
33. ---------------------------------------------------------------------
34. Output:
35. 猫叫：喵喵...
36. 狗叫:汪汪...
```

抽象类在java语言中所表示的是一种继承关系，一个子类只能存在一个父类，但是可以存在多个接口。

1.2.4 Final

使用final修饰的variable不能再被赋予新的值
如果是一个base type，就变成了一个constant
如果是一个reference variable，它始终会指向同一个状态下的object. 即使其他的reference variable改变了object的值, 他所指向的object还是原来的值
一般成员变量不会定义成final, 因为这种情况下它被定义成static是一样的功能，而定义成final会浪费内存.
如果method用final修饰，则不能被overridden
如果class用final修饰，则不能被继承

1.2.5 定义variable和method, constructor

1.2.5.1 定义variable:

```
[modifiers] type identifier_1[=initialValue_1], identifier_2[=initialValue_2]...
;
```

1.2.5.2 定义method:

```
 [modifiers] returnType methodName(type_1, param_1, ... , type_n param_n){
    // method body ...
 }
```

1. 返回值类型
   当returnType为void时，可以不返回。其余情况要返回一个跟returnType一致的variable或value
   Java method只能返回一个值，如果需要返回多个value, 要将values复合成一个object
2. 参数parameters
   Java中的parameters passed by value，所以方法中的parameters是copy的， 所以在method中改变是无法影响original variable的值的。即使是reference variable同样是copied the reference

```
public static void badRest(Counter c){
    c = new Counter(); //reassigns local name c to a new counter
}
```

```
4.
5.  public static void goodReset(Counter c){
6.      c.reset;    //resets the couter sent by the caller
7.  }
```

两个method都无法初始化c

1.2.6 定义Constructor

```
1.  modifiers name(type_0 parameter_0, ..., type_n parameter_n){
2.      //constructor body ...
3.  }
```

Constructor不能用static, abstract, final修饰

1.2.7 this关键字

To store the reference in a variable, or send it as a parameter to another method that expects an instance of that type as an argument.
Example:

```
1.  public Counter(int count){
2.      this.count = count; // set the instance variable equal to parameter
3.  }
4.
5.  public Counter(){
6.      this(0);    // invoke one-parameter constructor with value zero
7.  }
```

相当于指向了当前的class

1.2.8 main Method

The primary control for an application in Java must begin in some class with the execution of a special method named main.

```
1.  public static void main(String[] args){
2.      // main method body ...
3.  }
4.
5.  /*
6.  The args parameter is an array of String objects, that is, a collection of index
    ed strings, with the first string being args[0], the second being args[1], and
    so on.
7.  */
```

Example:

```
java Aquarium 45 // 执行了main method of a class named Aquarium, args[0]存储了string "45".
```

1.3 String, Wrappers, Arrays and Enum Types

1.3.1 String:

Char类型用单引号, 如 'G'.
String类型用双引号，如 "Data Structures & Algorithms in Java".
String类型的character是可以Indexing的，string类型是不可以改变的(immutable)
Concatenation对于string类型是自动的这点和python是一致的
Example:

```
String answer = mess + " will cost me " + 5 + " hours!";
>>> "carpetspot will cost me 5 hours!"
```

- Java的String包学习笔记
  - String间的比较要用`stt.equals(b)`;
    忽略大小写`equalsIgnoreCase(String otherstr);`
    Character间可以直接用==
  - String是否为空`str.isEmpty();`
  - String的截取`str.substring(start, end);`
    若读取到string结尾则end可省略
  - String对应索引的`characterstr.charAt(index);`
  - 按字典顺序比较两个字符串`str.compareTo(String otherstr);`
  - 字母大小写转换`str.toLowerCase();`和`str.toUpperCase();`
  - 字符串分割`str.split(String sign);`
    - 没有统一的对字符串进行分割的符号，如果想定义多个分割符，可使用符号“|”。例如，“,|=”表示分割符分别为“，”和“=”。
    - `str.split(String sign, int limit);`根据给定的分割符对字符串进行拆分，并限定拆分的次数。
  - 判断字符串的开始与结尾`startsWith(String prefix);`和`endsWith(String prefix);`
  - 去除空格`Str.trim();`
    返回字符串的副本，忽略前导空格和尾部空格。
  - 字符串替换`replace(oldChar, newChar);`
  - 字符串查找`str.indexOf(char)`

1.3.2 Wrapper types

Base Type	*Class Name*	*Creation Example*	*Access Example*
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

Java 自动包装和解包
Examples:

```
1.  int j = 8;
2.  Integer a = new Integer(12);
3.  int k = a; // implicit call to a.intValue()
4.  int m = j + a; // a is automatically unboxed before the addition
5.  a = 3 * m; // result is automatically boxed before assignment
6.  Integer b = new Integer("-135"); //constructor accepts a String
7.  int n = Interger.parseInt("2013"); // using static method of Integer class
```

对于line 3, Java automatically boxes the int, with an implicit call to new Integer(k). Integer value a can be given in which case Java automatically unboxes it with an implicit call to v.intValue( ).

1.3.3 Array

Creating an array:

```
1.  elementType[] arrayName = {initialValue_0, initialValue_1, ..., initialValue_n};
    //方法1
2.
3.  int[] primes = {2, 3, 5, 7, 11, 13};    //Example for method_1
4.
5.  double[] measurements = new double[1000]; //方法2
```

1.3.4 Enum type

```
1.  modifier enum name{valueName_0, valueName_1, ..., valueName_n};
2.
3.  public enum Day{MON, TUE, WED, THU, FRI, SAT, SUN};
4.
5.  // 调用
6.  Day today;
```

```
today = Day.Tue;
```

1.4 Expression:

1.4.1 Special character constants:

```
\n  new line
\t  tab
\b  backspace
\r  return
\f  form feed
\'  single quote
\\  backslash
\"  double quote
```

1.4.2 ++ and −−:

```
int i = 8;
int j = i++;    //j become 8 and then i becomes 9
int k = ++i;    // i becomes 10 and then k becomes 10
int m = i--;    // m becomes 10 and then i becomes 9
int n = 9+--i;  // i becomes 8 and then n becomes 17
```

1.4.3 Conditional:

```
!       not(prefix)
&&      conditional and
||      conditional or
```

1.4.4 Bitwise operator

```
~       bitwise complement(prefix unary operator)
&       bitwise and
|       bitwise or
^       bitwise exclusive-or
<<      shift bits left, filling in with zeros
>>      shift bits right, filling in with sign bit
>>>     shift bits right, filling in with zeros
```

- exclusive-or异或：

A	B	⊕
F	F	F
F	T	T
T	F	T
T	T	F

1.4.5 Operator precedence:

Operator Precedence		
	Type	**Symbols**
1	array index method call dot operator	[ ] ( ) .
2	postfix ops prefix ops cast	*exp*++ *exp*−− ++*exp* −−*exp* +*exp* −*exp* ~*exp* !*exp* (*type*) *exp*
3	mult./div.	* / %
4	add./subt.	+ −
5	shift	<< >> >>>
6	comparison	< <= > >= instanceof
7	equality	== !=
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	|
11	and	&&
12	or	||
13	conditional	*booleanExpression* ? *valueIfTrue* : *valueIfFalse*
14	assignment	= += −= *= /= %= <<= >>= >>>= &= ^= |=

1.4.6 Type Conversions

1.4.6.1 Double和integer间的转换

```
1.  double d1 = 3.2;
2.  double d2 = 3.9999;
3.  int i1 = (int) d1;
4.  int i2 = (int) d2;
5.  double d3 = (double) i2;
```

1.4.6.2 String和其他类型间的转换

```
String s1 = "2014";
int i1 = Integer.parseInt(s1);      // i1 gets value 2014
int i2 = -35;
String s2 = Integer.toString(i2);   // s2 gets value -35
```

以上的是Explicit casting，下面是Implicit Casting

1.4.6.3 Double和integer

```
int i1 =42;
double d1 = i1;     // d1 gets value 42.0
i1 = d1;            // compile error: possible loss of precision
```

(double) 7 / 4 = 1.75, because operator precedence dictates that the cast happens first, as ((double) 7) / 4, and thus 7.0 / 4, which implicitly becomes 7.0 / 4.0.** Note however that the expression, (double) (7 / 4) produces the result 1.0.**

1.4.6.4 String和其他类型间的转换

无论合适string与任意对象或基本数据类型串联是，它们都会被转成string类型，但是直接将他们强制转换为string是不行的

```
String s = 22;                          // this is wrong
String t = (String) 4.5;                // this is wrong
String u = "Value = " + (String) 13;    // this is wrong

String s = Integer.toString(22);    // this is good
String t = "" + 4.5;                //correct, but poor style
String u = "Value = " + 13;         // this is good
```

1.5 Control flow

1.5.1 switch

```
switch(d){
    case MON:
        System.out.println("This is tough.");
        bresk;
    case TUE:
        System.out.println("This is gettring better.");
        bresk;
    case WED:
        System.out.println("Half way there.");
        bresk;
    case TUE:
        System.out.println("I can see the light.");
```

```
            bresk;
        case FRI:
            System.out.println("Now we are talking.");
            bresk;
        default:
            System.out.println("Day off!");
    }
```

1.5.2 两种for:

```
// Type 1
for(initialization; booleanCondition; increment)
    loopBody

// Type 2
for(elementType name: container)
    loopBody
```

1.6 Java 输入

```
import java.util.Scanner;

public class InputExample{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        System.out.println("Enter your age in years: ")
        double age = input.nextDouble();
        System.out.print("Enter tour maximum heart rate: ")
        double rate = input.nextDouble();
        double fb = (rate - age) * 0.65;
        System.out.println("Your ideal fat-burning heart rate is " + fb);
    }
}

// 输出如下
>>>
Enter your age in years: 21
Enter tour maximum heart rate: 220
Your ideal fat-burning heart rate is 129.35
```

The Scanner class reads the input stream and divides it into tokens, which are strings of characters separated by delimiters. A delimiter is a special separating string, and the default delimiter is whitespace.

hasNext( ):	Return **true** if there is another token in the input stream.
next( ):	Return the next token string in the input stream; generate an error if there are no more tokens left.
hasNext*Type*( ):	Return **true** if there is another token in the input stream and it can be interpreted as the corresponding base type, *Type*, where *Type* can be Boolean, Byte, Double, Float, Int, Long, or Short.
next*Type*( ):	Return the next token in the input stream, returned as the base type corresponding to *Type*; generate an error if there are no more tokens left or if the next token cannot be interpreted as a base type corresponding to *Type*.

Additionally, Scanner objects can process input line by line, ignoring delimiters and even look for patterns within lines while doing so.

hasNextLine( ):	Returns **true** if the input stream has another line of text.
nextLine( ):	Advances the input past the current line ending and returns the input that was skipped.
findInLine(String *s*):	Attempts to find a string matching the (regular expression) pattern *s* in the current line. If the pattern is found, it is returned and the scanner advances to the first character after this match. If the pattern is not found, the scanner returns **null** and doesn't advance.

1.7 Packages and imports

- 打包:

```
1.  package packageName;
```

- 导入包:
  需要注意的是 import packageName.*;
  如果两个包都导入所有的方法和变量那么如果存在冲突时，调用冲突的变量和方法还是需要清楚声明包，相当于两者都没有被导入。

---

2 Object-Oriented Design

2.1 Inheritance

继承的特性:

- 子类继承父类非private的属性，方法，另外可以通过super调用父类的constructor
- 子类可以拥有自己的属性和方法，即子类可以对父类进行扩展。
- 子类可以用自己的方式实现父类的方法。
- Java的继承是单继承，但是可以多重继承.
- 提高了类之间的耦合性（继承的缺点，耦合度高就会造成代码之间的联系）
  Subclass constructors
  如果没有特定的无参constructor, JAVA会自行调用父类的无参constuctor.
  NOTE:
  父类有的属性在子类中重新定义后，构建子类的对象时会出现两个变量比如父类有String a, 子类也有String a， 那么新的子类对象会有两个a变量。这时候调用父类的方法时默认的全局变量a为父类的a，子类的方法默认的a为子类的a这就会出现一些问题
  - Example:
    03/12/2016在写Double Linked List时犯了如下错误:
    MyDList extends DList
    再DList中有size即链表节点个数，在MyDList中又定义了size这个时候调用DList中的isEmpty()函数，它只能是Empty，应为构建MyDList的对象时，他拥有两个size一个是父类的一个是子类的(父类的size会被无参constructor进行构造)，这时候调用父类的size()时它所用的size变量就是父类的。要注意这种情况

2.2 Interface

An interface is a collection of method declarations with no data and no bodies.
当一个class implements一个interface时，必须实现这个interface中所有声明的方法。

```
/** interface for objects that can be sold. */
public interface Sellable{
    /** Returns a description of the object. */
    public String description();

    /** Returns the list price in cents. */
    public int listPrice();

    /** Returns the lowest price in cents we will accept. */
    public int lowestPrice();
}
```

```
/** Class for photo that can be sold. */
public class Pholograph implements Sellable{
    public class String descript;   // discription of this photo
    private int price;              // the price we are setting
    private boolean color;          // true if photo is in color

    public Photograph(String desc, int p, boolean c){
        descript = desc;
        pric = p;
        color = c;
    }

    public String description(){return descript;}
    public int listPrice() {return price;}
    public int lowestPrice() {retrun price/2;}
    public boolean isColor() {return color;}
}
```

- 接口实现多遗传：

```
public interface Insurable extend Sellable, Transportable{
    /** Returns insured value in cents. */
    public int insuredValue();

    ...
}

public class BoxedItem2 implements Insurable{
    // same code as class BoxedItem
}
```

2.3 Abstract class

在JAVA中, abstract class扮演了介于一般class和接口之间的角色。Abstract method是哪些定义在抽象类中的没有实体的method. Abstract class区别于interface的地方是他可以包含concrete method. Abstract class可以继承其他class也可以被其他class继承.

对于使用Abstract class和interface, 明显的是接口Abstract class功能更强大因为它可以包涵concrete method而interface不行。但是Abstract class和普通的class一样只能继承一个class, 因此当需要同时继承多个class时就只能用interface.

```
public abstract class AbstractProgression{
    prottected long current;
    public AbstractProgression() {this(0);}
    public AbstractProgression(long start) {current = start;}
```

```
5.
6.      public long nextValue(){        // this is a concrete method
7.          long answer = current;
8.          advance();  // this protected call is responsible for advacing the curre
    nt value
9.          return answer;
10.     }
11.
12.     public void printProgression(int n){        // this a concrete method
13.         System.out.print(nextValue());        //print first value without leading
    space
14.         for(int j=1; j<n; j++)
15.             System.out.print(" " + nextValue());// print leading space before oth
    ers
16.         System.out.println();   // end the line
17.     }
18.
19.     protected abstract void advance();      // notice the lack of method body
20. }
```

2.4 Exception

在Java中, exceptions是可以根据code被抛出的对象

2.4.1 Try-catch statement处理Exceptions:

```
1.  // A typical syntax for a try-catch statement in Java is as follows:
2.  try{
3.      guardedBody
4.  }catch(exceptionType_1 variable_1){
5.      remedyBody_1
6.  }catch(exceptionType_1 variable_1){
7.      remedyBody_2
8.  }...
9.
10. ...
```

在上述代码的最后可以加上finally语句，finally语句中的内容将会被执行无论是否发生异常，this can be useful, for example, to close a file before proceeding onward.

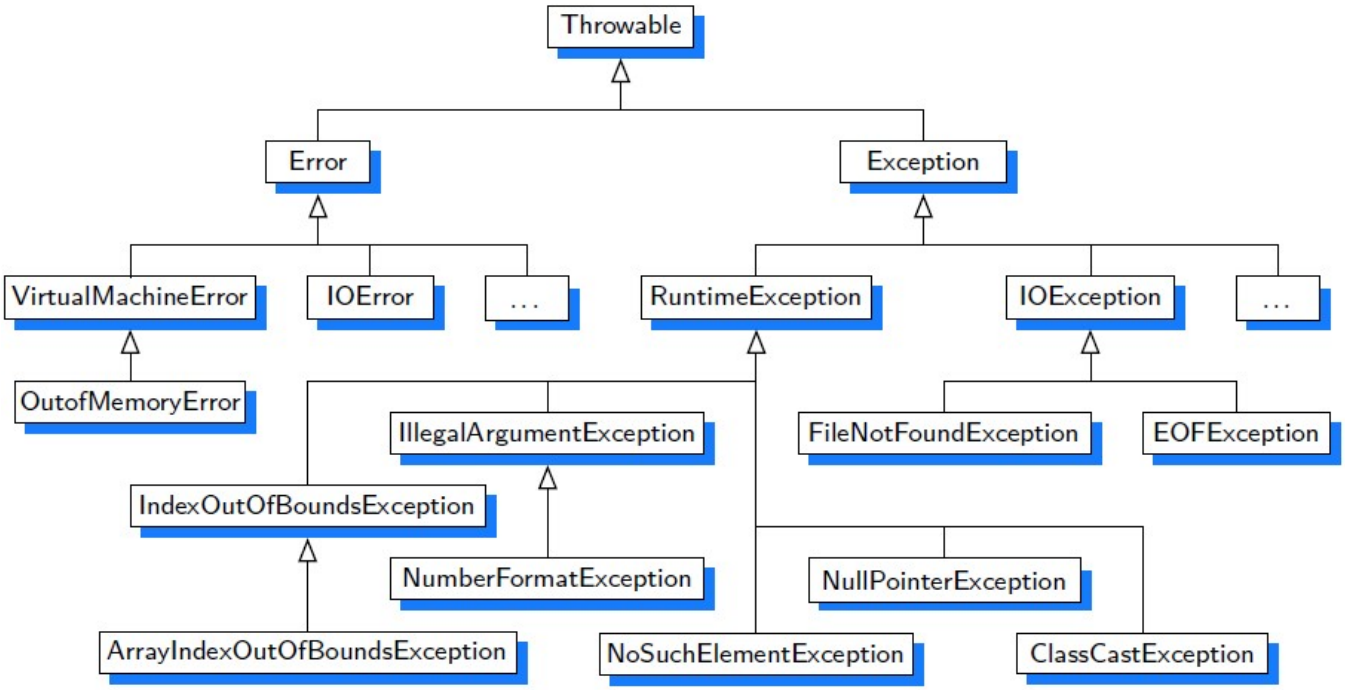
```
1.  ...}catch(ArrayIndexOutOfBoundsException e){
2.      System.out.println("No argument specified for n. Using default.")
3.  }catch(...
```

2.4.2 Throwing Exceptions:

```
1.  throw new exceptionType(parameters);
```

- Example:

```java
// Type 1
public void ensurePositive(int n){
    if (n < 0){
        throw new IllegalArgumentException("That's not positive!");
    }
    ...
}

// Type 2
public static int parseInt(String s) throws NumberFormatException{}
//需要在注释中用@throws进行说明，这样调用方法的时候就会注意到这些错误
```



2.5.1 Casting

```java
CreditCard card = new PredatoryCreditCard(...);     //widening
PredatoryCreditCard pc = (PredatoryCreditCard) card;    //narrowing
```

向上转型widening：从子类转成父类，不用casting
Father father-son = new son();
Father-son中的变量是等于father class中的
Father class中没有的son中有的method这时候后不能被调用
调用Father和son都有的method输出的是son版本的

向上转型narrowing：从父类转成子类，要casting
Son son-son = son() father-son;
Son-son中的变量和方法都等于son class中的

另外子类间不能互相转换，只能在父子间进行转换，为了避免出现这个问题可以是用instanceof进行检测
Example： Double和Integer都是n的自己

```
Number n;
Integer i;
n = new Integer(3);
i = (Integer) n;         // This is legal
n = new Double(3.1415);
i = (Integer) n;         // This is illegal
```

```
Number n;
Integer i;
n = new Integer(3);
if(n instanceof Integer)
    i = (Integer) n;     // This is legal
n = new Double(3.1415);
if(n instanceof Integer)
    i = (Integer) n;     // This will not be attempted
```

在construct variable时，如果constructor中的变量类型未知，可以用以下格式:

```
public class Pair<A, B>{
    A first;
    B second;
    public Pair(A a, B b){
        fist = a;
        second = b;
    }
    public A getFirst() {return first;}
    public B getSecond() {retrun second;}
}
bid = new Pair<String, Double>("ORCL", 32.07);
```

需要注意的是构建array时:

```
Pair<String, Double>[] holdings;
holdings = new Pair<String, Double>[25];    illegal; compile error
holding = new Pair[25]; // correct, but warning about unchecked cast
holding[0] = new Pair<>("ORCL", 32.07); // vaild element assignment

public class Protfolio<T>{
```

```
    T[] data;
    public Portfolio(int capacity){
        data = new T[capacity]; // illegal; compiler error
        data = (T[])new Object[capacity];   // legal, but compiler warning
    }
    public T get(int index){return data[index]};
    public void set(int index, T element){data[inde] = element;}
}
```

2.6 Nested class

The nested class is formally a member of the outer class, and its fully qualified name is OuterName.NestedName.
Nested class可以是static或nonstatic, static nested class像一般的类，它的instance与outer class没有联系，nonstatic可以通过OuterName.this来调用outer class中的methods和variables.

---

3 Fundamental Data Structures

3.1 Array

3.1.1 Array常用的方法 java.util.Arrays

equals(A, B): Returns true if and only if the array A and the array B are equal. Two arrays are considered equal if they have the same number of elements and every corresponding pair of elements in the two arrays are equal. That is, A and B have the same values in the same order.

fill(A, x): Stores value x in every cell of array A, provided the type of array A is defined so that it is allowed to store the value x.

copyOf(A, n): Returns an array of size n such that the first k elements of this array are copied from A, where $k = \min\{n, A.\text{length}\}$. If $n > A.\text{length}$, then the last $n - A.\text{length}$ elements in this array will be padded with default values, e.g., 0 for an array of **int** and **null** for an array of objects.

copyOfRange(A, s, t): Returns an array of size $t - s$ such that the elements of this array are copied in order from $A[s]$ to $A[t-1]$, where $s < t$, padded as with copyOf( ) if $t > A.\text{length}$.

toString(A): Returns a String representation of the array A, beginning with [, ending with], and with elements of A displayed separated by string ", ". The string representation of an element $A[i]$ is obtained using String.valueOf($A[i]$), which returns the string "null" for a **null** reference and otherwise calls $A[i]$.toString( ).

sort(A): Sorts the array A based on a natural ordering of its elements, which must be comparable. Sorting algorithms are the focus of Chapter 12.

binarySearch(A, x): Searches the *sorted* array A for value x, returning the index where it is found, or else the index of where it could be inserted while maintaining the sorted order. The binary-search algorithm is described in Section 5.1.3.

另外有一个有用的string method: S.toCharArray();

```
S=”bird”;
S.toCharArray();

>>> A={'b', 'i', 'r', 'd'}
```

3.1.2 Random库的常用方法 java.util.Random

Java中random得到的数是pseudorandom number, 它们是基于next = (a * cur + b) % n所生成的这里

的a, b和n是合适的选择的整数，产生的随机数是服从正态分布的

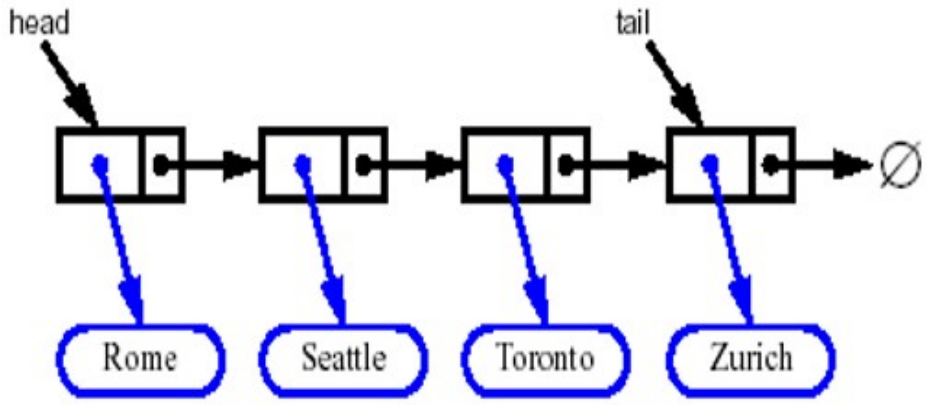
nextBoolean( ):	Returns the next pseudorandom **boolean** value.
nextDouble( ):	Returns the next pseudorandom **double** value, between 0.0 and 1.0.
nextInt( ):	Returns the next pseudorandom **int** value.
nextInt(*n*):	Returns the next pseudorandom **int** value in the range from 0 up to but not including *n*.
setSeed(*s*):	Sets the seed of this pseudorandom number generator to the **long** *s*.

Example:

```
1.  import java.util.Arrays;
2.  import java.util.Random;
3.  /** Program showing some array uses. */
4.  public class ArrayTest {
5.      public static void main(String[ ] args) {
6.          int data[ ] = new int[10];
7.          Random rand = new Random( ); // a pseudo-random number generator
8.          rand.setSeed(System.currentTimeMillis( )); // use current time as a seed
9.          // fill the data array with pseudo-random numbers from 0 to 99,
    inclusive
10.         for (int i = 0; i < data.length; i++)
11.         data[i] = rand.nextInt(100); // the next pseudo-random number
12.         int[ ] orig = Arrays.copyOf(data, data.length); // make a copy of the dat
    a array
13.         System.out.println("arrays equal before sort: "+Arrays.equals(data, orig)
    );
14.         Arrays.sort(data); // sorting the data array (orig is unchanged)
15.         System.out.println("arrays equal after sort: " + Arrays.equals(data, orig
    ));
16.         System.out.println("orig = " + Arrays.toString(orig));
17.         System.out.println("data = " + Arrays.toString(data));
18.     }
19. }
20.
21. //输出
22. /*
23. We show a sample output of this program below:
24. arrays equal before sort: true
25. arrays equal after sort: false
26. orig = [41, 38, 48, 12, 28, 46, 33, 19, 10, 58]
27. data = [10, 12, 19, 28, 33, 38, 41, 46, 48, 58]
28.
```

```
In another run, we got the following output:
arrays equal before sort: true
arrays equal after sort: false
orig = [87, 49, 70, 2, 59, 37, 63, 37, 95, 1]
data = [1, 2, 37, 37, 49, 59, 63, 70, 87, 95]
*/
```

3.2 Singly Linked Lists



Example:

```
public class Singly_LinkedList<E> {

    public static class Node<E> {
        private E element;
        private Node<E> next;

        public Node(E e, Node<E> n){
            element = e;
            next = n;
        }

        public E getElement(){
            return element;
        }

        public Node<E> getNext(){
            return next;
        }

        public void setNext(Node<E> n){
            next = n;
        }
    }

    private Node<E> head = null;
```

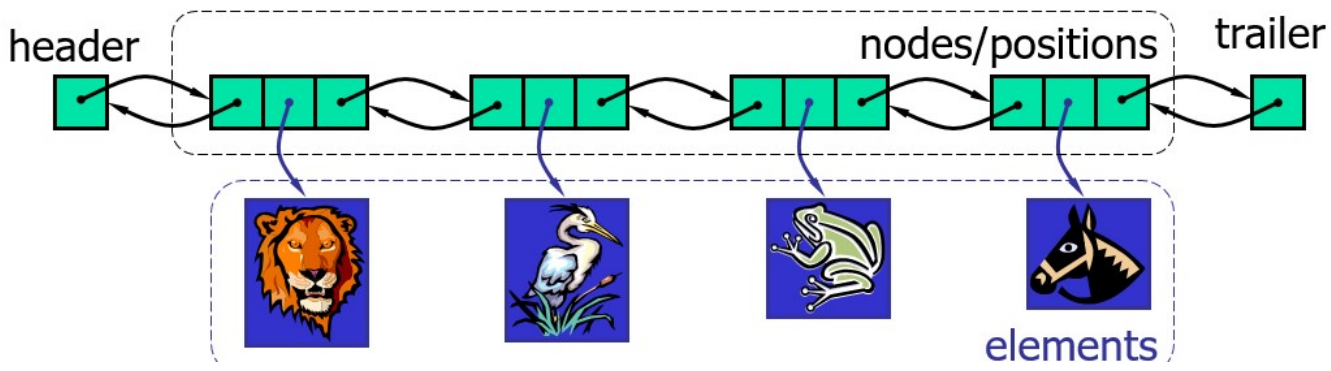
```
26.      private Node<E> tail = null;
27.      private int size = 0;
28.
29.
30.      public Singly_LinkedList(){}
31.
32.      public int size(){
33.          return size;
34.      }
35.
36.      public boolean isEmpty(){
37.          return size == 0;
38.      }
39.
40.      public E first(){
41.          if(isEmpty())
42.              return null;
43.          return tail.getElement();
44.      }
45.
46.      public void addFirst(E e){
47.          head = new Node<>(e, head);
48.          if (size == 0)
49.              tail = head;
50.          size++;
51.      }
52.
53.      public void addLast(E e){
54.          Node<E> newest = new Node<>(e, null);
55.          if(isEmpty())
56.              head = newest;
57.          else
58.              tail.setNext(newest);
59.          tail = newest;
60.          size++;
61.      }
62.
63.      public E removeFirst(){
64.          if(isEmpty())
65.              return null;
66.          E answer = head.getElement();
67.          head = head.getNext();
68.          size--;
69.          if(size == 0)
70.              tail = null;
71.          return answer;
72.      }
73.
74.  }
```

3.3 Circularly Linked Lists

```java
package fundamental;

public class CircularlyLinkedList<E> {

    public static class Node<E>{
        private E element;
        private Node<E> next;

        public Node(E e, Node<E> n){
            element = e;
            next = n;
        }

        public void setNext(Node<E> n){
            next = n;
        }

        public E getElement(){
            return element;
        }

        public Node<E> getNext(){
            return next;
        }
    }

    private Node<E> tail = null;    //we store tail(but not head)
    private int size = 0;

    public CircularlyLinkedList(){}

    public boolean isEmpty(){
        return size == 0;
    }

    public E first(){
        if(isEmpty())
            return null;
        return tail.getNext().getElement();
    }

    public E last(){
        if(isEmpty())
            return null;
        return tail.getElement();
    }

```

```
    public void rotate(){
        if(tail != null)
            tail = tail.getNext();
    }

    public void addFirst(E e){
        if(size == 0){
            tail = new Node(e, null);
            tail.setNext(tail);
        }else{
            Node<E> newest = new Node<>(e, tail.getNext());
            tail.setNext(newest);
        }
        size++;
    }

    public void addLast(E e){
        addFirst(e);
        tail = tail.getNext();
    }

    public E removeFirst(){
        if(isEmpty())
            return null;
        Node<E> head = tail.getNext();
        if (head == tail)
            tail = null;
        else
            tail.setNext(head.getNext());
        size--;
        return head.getElement();
    }
}
```

3.4 Doubly Linked Lists



- 详见`DList.java, MyDList`

3.5 Equivalence Testing

3.5.1 Equivalence Testing with Arrays

a == b: Tests if a and b refer to the same underlying array instance.

a.equals(b): Interestingly, this is identical to a == b. Arrays are not a true class type and do not override the Object.equals method.

Arrays.equals(a,b): This provides a more intuitive notion of equivalence, returning **true** if the arrays have the same length and all pairs of corresponding elements are "equal" to each other. More specifically, if the array elements are primitives, then it uses the standard == to compare values. If elements of the arrays are a reference type, then it makes pairwise comparisons a[k].equals(b[k]) in evaluating the equivalence.

Arrays.deepEquals(a,b): Identical to Arrays.equals(a,b) except when the elements of a and b are themselves arrays, in which case it calls Arrays.deepEquals(a[k],b[k]) for corresponding entries, rather than a[k].equals(b[k]).

3.5.2 Equivalence Testing with Linked Lists

```
1.  public boolean equals(Object o) {
2.      if (o == null)
3.          return false;
4.      if (getClass( ) != o.getClass( ))
5.          return false;
6.      SinglyLinkedList other = (SinglyLinkedList) o; // use nonparameterized type
7.      if (size != other.size)
8.          return false;
9.      Node walkA = head; // traverse the primary list
10.     Node walkB = other.head; // traverse the secondary list
11.     while (walkA != null) {
12.         if (!walkA.getElement( ).equals(walkB.getElement( )))
13.             return false; //mismatch
14.         walkA = walkA.getNext( );
15.         walkB = walkB.getNext( );
16.     }
17.     return true; // if we reach this, everything matched successfully
18. }
```

3.6 Clone

3.6.1 Cloning Array

```
Backup = data.clone();
```

当data这个array中存储的是reference variable时，clone是没有用的
Java中Array自带的clone是浅复制，当复制reference variable时，首先这个object对应的class要是clonable，而后进行如下操作, 对每一个reference variable进行clone：

```
Person[ ] guests = new Person[contacts.length];
for (int k=0; k < contacts.length; k++)
    guests[k] = (Person) contacts[k].clone( ); // returns Object type
```

多维数组的clone:

```
public static int[ ][ ] deepClone(int[ ][ ] original) {
    int[ ][ ] backup = new int[original.length][ ]; // create top-level array of arrays
    for (int k=0; k < original.length; k++)
    backup[k] = original[k].clone( ); // copy row k
    return backup;
}
```

3.6.2 Cloning Linked Lists

```
public SinglyLinkedList<E> clone( ) throws CloneNotSupportedException {
    // always use inherited Object.clone() to create the initial copy
    SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone( ); // safe cast
    if (size > 0) { // we need independent chain of nodes
        other.head = new Node<>(head.getElement( ), null);
        Node<E> walk = head.getNext( ); // walk through remainder of original list
        Node<E> otherTail = other.head; // remember most recently created node
        while (walk != null) { // make a new node storing same element
            Node<E> newest = new Node<>(walk.getElement( ), null);
            otherTail.setNext(newest); // link previous node to this one
            otherTail = newest;
            walk = walk.getNext( );
        }
    }
    return other;
}
```

4 Algorithm analysis

4.1 The Seven Functions Used in This Book

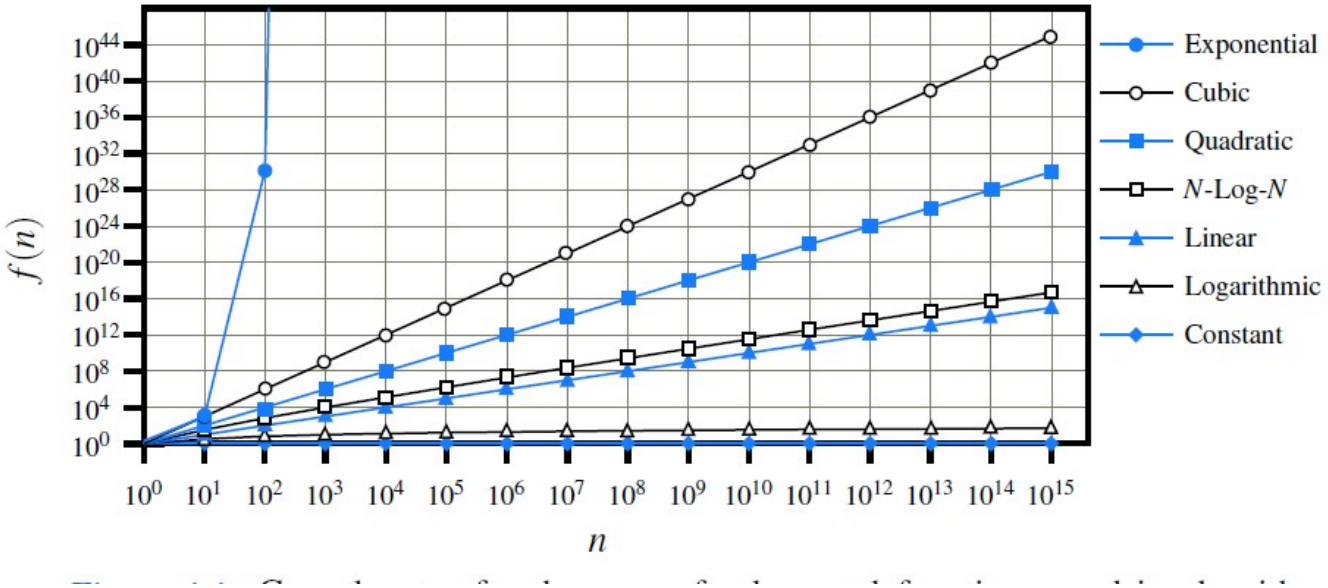
- The Constant Function: $f(n) = c$
- The Logarithm Function: $f(n) = log_b^n$

  This base is so common that we will typically omit it from the notation when it is 2.
- The Linear Function: $f(n) = n$
- The N-Log-N Function: $f(n) = nlogn$
- The Quadratic Function: $f(b) = n^2$
- The cubic Function: $f(n) = n^3$
  - Other Polynomials: $f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots + a_d n^d$
- The Exponential Function: $f(n) = b^n$

4.2 Comparing Growth Rates

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Table 4.2: Seven functions commonly used in the analysis of algorithms. We recall that $\log n = \log_2 n$. Also, we denote with a a constant greater than 1.



渐进确定上下界:

Example:

```java
/** Uses repeated concatenation to compose a String with n copies of character c
. */
public static String repeat1(char c, int n) {
    String answer = "";
    for (int j=0; j < n; j++)
        answer += c;
    return answer;
}

/** Uses StringBuilder to compose a String with n copies of character c. */
public static String repeat2(char c, int n) {
    StringBuilder sb = new StringBuilder( );
    for (int j=0; j < n; j++)
        sb.append(c);
    return sb.toString( );
}
```

这种情况下对复杂度的分析要特别注意，在repeat1中，第一次是1，第二次是2...所以复杂度为n2(String不能修改，因此每次都要创建一个新的string相当于里面增加了一层

```java
 for(int j=0;j<i;j++)
```

而repeat2的stringBuilder可以被修改，每次只需增加一个character，因此复杂度为n

NOTE: 建一个n维数组的复杂度为n

5 IO

5.1 读写的路径

默认的读写路径从当前project文件开始
Example：
E:\Develop\WorkSpace\Java_class\src\assignment_1\myfile.txt这里的Java_class是当前project的名字，这个myfile.txt的相对路径为 src\assignment_1\myfile.txt

5.2 判断文件是否存在

```java
File file = new File(filename);
if(file.exist()){
}
```

5.3 Java按空格读写文件

```
...
Scanner input = new Scanner(new FileInputStream("src\\a_1\\myfile.txt"));
while(input.hasNext()){
    System.out.println(input.next());
}
input.close();
...
```

NOTE: Scanner的对象用完了用input.close()

6. Stacks， Queues and Deques

6.1 Stacks

JAVA有Stack的包， `java.util.Stack`
包含`size(); empty(); push(e); pop(); peek();...`
需要注意的是 methods pop and peek of the `java.util.Stack` class throw a custom EmptyStackException if called when the stack is empty否则返回一个null

6.1.1 Stack接口

```
public interface Stack<E> { 11
    /**
    * Returns the number of elements in the stack.
    * @return number of elements in the stack
    */
    int size();

    /**
    * Tests whether the stack is empty.
    * @return true if the stack is empty, false otherwise
    */
    boolean isEmpty();

    /**
    * Inserts an element at the top of the stack.
    * @param e the element to be inserted
    */
    void push(E e);

    /**
    * Returns, but does not remove, the element at the top of the stack.
```

```
     * @return top element in the stack (or null if empty)
     */
     E top();

     /**
     * Removes and returns the top element from the stack.
     * @return element removed (or null if empty)
     */
     E pop();
}
```

6.1.2 Implement Stack with Array

- 缺点是这样的Stack他的长度是固定的

```
public class ArrayStack<E> implements Stack<E> {
    public static final int CAPACITY=1000; // default array capacity
    private E[ ] data; // generic array used for storage
    private int t = -1; // index of the top element in stack
    public ArrayStack( ) { this(CAPACITY); } // constructs stack with default
capacity
    public ArrayStack(int capacity) { // constructs stack with given capacity
    data = (E[ ]) new Object[capacity]; // safe cast; compiler may give warning
    }
    public int size( ) { return (t + 1); }
    public boolean isEmpty( ) { return (t == -1); }
    public void push(E e) throws IllegalStateException {
        if (size( ) == data.length)
            throw new IllegalStateException("Stack is full");
        data[++t] = e; // increment t before storing new item
    }
    public E top( ) {
        if (isEmpty( ))
            return null;
        return data[t];
    }
    public E pop( ) {
        if (isEmpty( ))
            return null;
        E answer = data[t];
        data[t] = null; // dereference to help garbage collection
        t--;
        return answer;
    }
}
```

6.1.3 Implement Stack with Linked list

- 利用Linked list可以解决长度固定的问题

```java
public class LinkedStack<E> implements Stack<E> {
    private SinglyLinkedList<E> list = new SinglyLinkedList<>( ); // an empty list
    public LinkedStack( ) { } // new stack relies on the initially empty list
    public int size( ) { return list.size( ); }
    public boolean isEmpty( ) { return list.isEmpty( ); }
    public void push(E element) { list.addFirst(element); }
    public E top( ) { return list.first( ); }
    public E pop( ) { return list.removeFirst( ); }
}
```

6.1.4 Reversing an Array Using a Stack

```java
public static <E> void reverse(E[ ] a) {
    Stack<E> buffer = new ArrayStack<>(a.length);
    for (int i=0; i < a.length; i++)
        buffer.push(a[i]);
    for (int i=0; i < a.length; i++)
        a[i] = buffer.pop( );
}
```

6.1.5 运用Stack

- Matching Parentheses and HTML Tags，这里为HTML Tags的match为例给出源码

```java
public static boolean isHTMLMatched(String html) {
    Stack<String> buffer = new LinkedStack<>( );
    int j = html.indexOf('<'); // find first '<' character (if any)
    while (j != -1) {
        int k = html.indexOf('>', j+1); // find next '>' character
        if (k == -1)
            return false; // invalid tag
        String tag = html.substring(j+1, k); // strip away < >
        if (!tag.startsWith("/")) // this is an opening tag
            buffer.push(tag);
        else { // this is a closing tag
            if (buffer.isEmpty( ))
                return false; // no tag to match
            if (!tag.substring(1).equals(buffer.pop( )))
                return false; // mismatched tag
        }
        j = html.indexOf('<', k+1); // find next '<' character (if any)
    }
    return buffer.isEmpty( ); // were all opening tags matched?
}
```

6.2 Queue

6.2.1 java.util.Queue

Interface java.util.Queue	
throws exceptions	returns special value
add(e)	offer(e)
remove( )	poll( )
element( )	peek( )
size( )	
isEmpty( )	

6.2.2 Queue 接口

```
public interface Queue<E> {
    /** Returns the number of elements in the queue. */
    int size( );
    /** Tests whether the queue is empty. */
    boolean isEmpty( );
    /** Inserts an element at the rear of the queue. */
    void enqueue(E e);
    /** Returns, but does not remove, the first element of the queue (null if empty). */
    E first( );
    /** Removes and returns the first element of the queue (null if empty). */
    E dequeue( );
}
```

6.2.3 Implement Queue with Array

```
public class ArrayQueue<E> implements Queue<E> {
    // instance variables
    private E[ ] data; // generic array used for storage
    private int f = 0; // index of the front element
    private int sz = 0; // current number of elements
    // constructors
    public ArrayQueue( ) {this(CAPACITY);} // constructs queue with default capacity
    public ArrayQueue(int capacity) { // constructs queue with given capacity
        data = (E[ ]) new Object[capacity]; // safe cast; compiler may give warning
    }
    // methods
    /** Returns the number of elements in the queue. */
    public int size( ) { return sz; }
    /** Tests whether the queue is empty. */
    public boolean isEmpty( ) { return (sz == 0); }
```

```
    /** Inserts an element at the rear of the queue. */
    public void enqueue(E e) throws IllegalStateException {
        if (sz == data.length)
            throw new IllegalStateException("Queue is full");
        int avail = (f + sz) % data.length; // use modular arithmetic
        data[avail] = e;
        sz++;
    }
    /** Returns, but does not remove, the first element of the queue (null if empty). */
    public E first( ) {
        if (isEmpty( ))
            return null;
        return data[f];
    }
    /** Removes and returns the first element of the queue (null if empty). */
    public E dequeue( ) {
        if (isEmpty( ))
            return null;
        E answer = data[f];
        data[f] = null; // dereference to help garbage collection
        f = (f + 1) % data.length;
        sz--;
        return answer;
    }
}
```

- 再dequeue()中f = (f + 1) % data.length;它对data.length取余是因为如data.length = 10到最后一个数时索引应该为9，然后9+1 %10 = 0，然而0已经被重新赋值为null因此，正好可以避免9+1 = 10导致f超出array范围的情况

6.2.4 Implement Queue with LinkedList

```
public class LinkedQueue<E> implements Queue<E> {
    private SinglyLinkedList<E> list = new SinglyLinkedList<>( ); // an empty list
    public LinkedQueue( ) { } // new queue relies on the initially empty list
    public int size( ) { return list.size( ); }
    public boolean isEmpty( ) { return list.isEmpty( ); }
    public void enqueue(E element) { list.addLast(element); }
    public E first( ) { return list.first( ); }
    public E dequeue( ) { return list.removeFirst( ); }
}
```

6.2.5 A Circular Queue

6.3 Double-Ended Queues

6.3.2 Double-Ended Queue java.util.Deque

Interface java.util.Deque	
throws exceptions	returns special value
getFirst( )	peekFirst( )
getLast( )	peekLast( )
addFirst(e)	offerFirst(e)
addLast(e)	offerLast(e)
removeFirst( )	pollFirst( )
removeLast( )	pollLast( )
size( )	
isEmpty( )	

6.3.2 Deque interface

```
1.  public interface Deque<E> {
2.      /** Returns the number of elements in the deque. */
3.      int size( );
4.      /** Tests whether the deque is empty. */
5.      boolean isEmpty( );
6.      /** Returns, but does not remove, the first element of the deque (null if em
    pty). */
7.      E first( );
8.      /** Returns, but does not remove, the last element of the deque (null if emp
    ty). */
9.      E last( );
10.     /** Inserts an element at the front of the deque. */
11.     void addFirst(E e);
12.     /** Inserts an element at the back of the deque. */
13.     void addLast(E e);
14.     /** Removes and returns the first element of the deque (null if empty). */
15.     E removeFirst( );
16.     /** Removes and returns the last element of the deque (null if empty). */
17.     E removeLast( );
18. }
```

6.3.3 Implement Deque with Doubly Linked List

```
1.  public class DLinkedDeque<E> implements Deque<E> {
2.      private DoublyLinked<E> list = new SinglyLinkedList<>( ); // an empty list
3.      public DLinkedDeque( ) { } // new stack relies on the initially empty list
4.      public int size( ) { return list.size( ); }
5.      public boolean isEmpty( ) { return list.isEmpty( ); }
```

```
    public E first( ) { return list.first() };
    public E last( ) { return list.last( )}
    public void addFirst(E element) { list.addFirst(element); }
    public void addLast(E element) { list.addLast(element); }
    public E removeFirst( ) { return list.removeFirst( ); }
    public E removeLast( ) { return list.removeLast( ); }
}
```

7. List and Iterator ADTs

7.1 List

A list is a collection of elements of the same type that are stored in a certain linear order.

- An element can be accessed, inserted or removed.
  - Array lists
    - 可以通过索引进行访问，删除，插入
    - 超出索引或是负值索引会抛出exception
    - 其中的`get`和`set`方法的复杂度为 $O(1)$
    - `add`和`delete`方法的复杂度为 $O(n)$
    - 当List满时，新建一个两倍大的ArrayList，将原来list中的data拷贝到新的ArrayList中
    - 翻倍机制下，从capacity为1的list开始插入n个元素的$O(n)$
  - Node lists

7.1.1 java.util.List

Java自带的list包中包含`size(); isEmpty(); get(i); set(i,e); add(i,e); remove(i); ...`等方法

7.1.2 List interface

```
public interface List<E> {
    /** Returns the number of elements in this list. */
    int size( );

    /** Returns whether the list is empty. */
    boolean isEmpty( );

    /** Returns (but does not remove) the element at index i. */
    E get(int i) throws IndexOutOfBoundsException;

    /** Replaces the element at index i with e, and returns the replaced element
. */
    E set(int i, E e) throws IndexOutOfBoundsException;

```

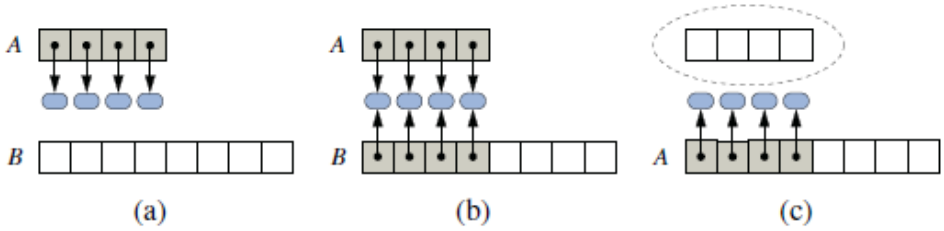
```
    /** Inserts element e to be at index i, shifting all subsequent elements later. */
    void add(int i, E e) throws IndexOutOfBoundsException;

    /** Removes/returns the element at index i, shifting subsequent elements earlier. */
    E remove(int i) throws IndexOutOfBoundsException;
    }
```

7.1.3 Implement list with array

```
public class ArrayList<E> implements List<E> {
    // instance variables
    public static final int CAPACITY=16; // default array capacity
    private E[ ] data; // generic array used for storage
    private int size = 0; // current number of elements
    // constructors
    public ArrayList( ) { this(CAPACITY); } // constructs list with default capacity
    public ArrayList(int capacity) { // constructs list with given capacity
        data = (E[ ]) new Object[capacity]; // safe cast; compiler may give warning
    }
    // public methods
    /** Returns the number of elements in the array list. */
    public int size( ) { return size; }
    /** Returns whether the array list is empty. */
    public boolean isEmpty( ) { return size == 0; }
    /** Returns (but does not remove) the element at index i. */
    public E get(int i) throws IndexOutOfBoundsException {
        checkIndex(i, size);
    return data[i];
    }
    /** Replaces the element at index i with e, and returns the replaced element. */
    public E set(int i, E e) throws IndexOutOfBoundsException { checkIndex(i, size);
        E temp = data[i];
        data[i] = e;
        return temp;
    }
    /** Inserts element e to be at index i, shifting all subsequent elements later. */
    public void add(int i, E e) throws IndexOutOfBoundsException, IllegalStateException {
        checkIndex(i, size + 1);
        if (size == data.length) // not enough capacity
            throw new IllegalStateException("Array is full");
        for (int k=size-1; k >= i; k--) // start by shifting rightmost
            data[k+1] = data[k];
```

```
            data[i] = e; // ready to place the new element
            size++;
        }
        /** Removes/returns the element at index i, shifting subsequent elements ear
    lier. */
        public E remove(int i) throws IndexOutOfBoundsException {
            checkIndex(i, size);
            E temp = data[i];
            for (int k=i; k < size-1; k++) // shift elements to fill hole
            data[k] = data[k+1];
            data[size-1] = null; // help garbage collection
            size--;
            return temp;
        }
        // utility method
        /** Checks whether the given index is in the range [0, n-1]. */
        protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
        if (i < 0 || i >= n)
            throw new IndexOutOfBoundsException("Illegal index: " + i);
        }
    }
```

7.1.4 Implementing a Dynamic Array




```
/** Resizes internal array to have given capacity = 2 * size. */
protected void resize() {
    E[ ] temp = (E[ ]) new Object[2 * data.length]; // safe cast; compiler may gi
ve warning
    for (int k=0; k < size; k++)
        temp[k] = data[k];
    data = temp; // start using the new array
}
```

7.2 Node List

7.2.1 Position

- The Position ADT models the notion of place within a data structure where a single object is stored
- Postion interface

```java
public interface Position<E> {
    /**
    * Returns the element stored at this position.
    *
    * @return the stored element
    * @throws IllegalStateException if position no longer valid
    */
    E getElement( ) throws IllegalStateException;
}
```

7.2.2 Node

- The Node List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  size(), isEmpty()
- Accessor methods: `first(); last(); prev(p); next(p);`
- Update
  methods: `set(p, e); addBefore(p, e); addAfter(p, e); addFirst(e); addLast(e); remove(p)`

7.2.3 Position List interface

```java
/** An interface for positional lists. */
public interface PositionalList<E> {
    /** Returns the number of elements in the list. */
    int size( );

    /** Tests whether the list is empty. */
    boolean isEmpty( );

    /** Returns the first Position in the list (or null, if empty). */
    Position<E> first( );

    /** Returns the last Position in the list (or null, if empty). */
    Position<E> last( );

    /** Returns the Position immediately before Position p (or null, if p is fir
st). */
    Position<E> before(Position<E> p) throws IllegalArgumentException;

    /** Returns the Position immediately after Position p (or null, if p is last
). */
    Position<E> after(Position<E> p) throws IllegalArgumentException;

    /** Inserts element e at the front of the list and returns its new Position.
*/
    Position<E> addFirst(E e);

```

```
    /** Inserts element e at the back of the list and returns its new Position. */
    Position<E> addLast(E e);

    /** Inserts element e immediately before Position p and returns its new Position. */
    Position<E> addBefore(Position<E> p, E e) throws IllegalArgumentException;

    /** Inserts element e immediately after Position p and returns its new Position. */
    Position<E> addAfter(Position<E> p, E e) throws IllegalArgumentException;

    /** Replaces the element stored at Position p and returns the replaced element. */
    E set(Position<E> p, E e) throws IllegalArgumentException;

    /** Removes the element stored at Position p and returns it (invalidating p). */
    E remove(Position<E> p) throws IllegalArgumentException;
}
```

7.2.3 Implement PositionList with DNode

```
public class DNode<E> implements Position<E> {
    private DNode<E> prev, next; // References to the nodes before and after
    private E element; // Element stored in this position

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

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

// Accessor methods
    public DNode<E> getNext() { return next; }
    public DNode<E> getPrev() { return prev; }
    // Update methods
    public void setNext(DNode<E> newNext) { next = newNext; }
    public void setPrev(DNode<E> newPrev) { prev = newPrev; }
    public void setElement(E newElement) { element = newElement; }
 }
```

7.2.3 Implement with Doubly Linked List

```
public class NodePositionList<E> implements PositionList<E> {
    protected int numElts; // Number of elements in the list
    protected DNode<E> header, trailer; // Special sentinels
    /** Constructor that creates an empty list; O(1) time */
    public NodePositionList() {
        numElts = 0;
        header = new DNode<E>(null, null, null); // create header
        trailer = new DNode<E>(header, null, null); // create trailer
        header.setNext(trailer); // make header and trailer point to each other
    }
    /** Checks if position is valid for this list and converts it to DNode if it
is valid; O(1) time */
    protected DNode<E> checkPosition(Position<E> p) throws
InvalidPositionException {
    if (p == null) throw new InvalidPositionException ("Null position passed to
NodeList");
    if (p == header) throw new InvalidPositionException ("The header node is not
a valid position");
    if (p == trailer) throw new InvalidPositionException ("The trailer node is
not a valid position");
    try {
        DNode<E> temp = (DNode<E>) p;
        if ((temp.getPrev() == null) || (temp.getNext() == null))
            throw new InvalidPositionException ("Position does not belong to a
valid NodeList");
        return temp;
    } catch (ClassCastException e) {
        throw new InvalidPositionException ("Position is of wrong type for this
list");
    }
    }

    /** Returns the number of elements in the list; O(1) time */
    public int size() { return numElts; }

    /** Returns whether the list is empty; O(1) time */
    public boolean isEmpty() { return (numElts == 0); }

    /** Returns the first position in the list; O(1) time */
    public Position<E> first() throws EmptyListException {
        if (isEmpty())
            throw new EmptyListException("List is empty");
        return header.getNext();
    }

    /** Returns the position before the given one; O(1) time */
    public Position<E> prev(Position<E> p) throws InvalidPositionException, Bound
aryViolationException {
```

```
41.         DNode<E> v = checkPosition(p);
42.         DNode<E> prev = v.getPrev();
43.         if (prev == header)
44.             throw new BoundaryViolationException ("Cannot advance past the
    beginning of the list");
45.         return prev;
46.     }
47.
48.     /** Insert the given element before the given position, returning the new po
    sition; O(1) time */
49.     public void addBefore(Position<E> p, E element) throws
    InvalidPositionException {
50.         DNode<E> v = checkPosition(p);
51.         numElts++;
52.         DNode<E> newNode = new DNode<E>(v.getPrev(), v, element);
53.         v.getPrev().setNext(newNode);
54.         v.setPrev(newNode);
55.     }
56.
57.     /** Insert the given element at the beginning of the list, returning  the ne
    w position; O(1) time */
58.     public void addFirst(E element) {
59.         numElts++;
60.         DNode<E> newNode = new DNode<E>(header, header.getNext(), element);
61.         header.getNext().setPrev(newNode);
62.         header.setNext(newNode);
63.     }
64.
65.     /**Remove the given position from the list; O(1) time */
66.     public E remove(Position<E> p) throws InvalidPositionException{
67.         DNode<E> v = checkPosition(p);
68.         numElts--;
69.         DNode<E> vPrev = v.getPrev();
70.         DNode<E> vNext = v.getNext();
71.         vPrev.setNext(vNext);
72.         vNext.setPrev(vPrev);
73.         E vElem = v.element();
74.         // unlink the position from the list and make it invalid
75.         v.setNext(null);
76.         v.setPrev(null);
77.         return vElem;
78.     }
79.
80.     /** Replace the element at the given position with the new element and retur
    n the old element; O(1) time */
81.     public E set(Position<E> p, E element) throws InvalidPositionException{
82.         DNode<E> v = checkPosition(p);
83.         E oldElt = v.element();
84.         v.setElement(element);
85.         return oldElt;
```

```
86.         }
```

- 在用doubly linked list实现List的过程中
  - The space used by a list with n elements is O(n).
  - The space used by each position of the list is O(1).
  - All the operations of the List ADT run in O(1) time.
  - Operation element() of the Position ADT runs in O(1) time.

7.3 Iterator

7.3.1 Iterators in JAVA

- Methods of the Iterator ADT:
  - boolean hasNext(): Test whether there are elements left in the iterator.
  - object next(): Return the next element in the iterator.

```
1.  public interface PositionList <E> extends Iterable <E> {
2.      // ...all the other methods of the list ADT ...
3.      /** Returns an iterator of all the elements in the list. */
4.      public Iterator<E> iterator();
5.  }
6.
7.  /** Returns a textual representation of a given node list */
8.  public static <E> String toString(PositionList<E> l) {
9.      Iterator<E> it = l.iterator();
10.     String s = "[";
11.     while (it.hasNext()) {
12.         s += it.next(); // implicit cast of the next element to String
13.         if (it.hasNext())
14.             s += ", ";
15.     }
16.     s += "]";
17.     return s;
18. }
```

7.4.2 Implement Iterator

- Two notions of iterator:
  - snapshot: freezes the contents of the data structure at a given time
    - 支持连续的访问， 他会将每一个数据存储到独立的数据结构中
    - 使用标记来记录当前位置
    - 根据标记创建一个新的Iterator
    - next() returns the next element, if any 再将标记移动到下一个位置

- dynamic: follows changes to the data structure
  - This approach iterates over the data structure directly. No separate copy of the data structure is needed.

```
public class ElementIterator<E> implements Iterator<E> {
    protected PositionList<E> list; // the underlying list
    protected Position<E> cursor; // the next position
    /** Creates an element iterator over the given list. */
    public ElementIterator(PositionList<E> L) {
        list = L; cursor = (list.isEmpty())? null : list.first();
    }
    public boolean hasNext() { return (cursor != null); }
    public E next() throws NoSuchElementException {
        if (cursor == null)
            throw new NoSuchElementException("No next element");
        E toReturn = cursor.element();
        cursor = (cursor == list.last())? null : list.next(cursor);
        return toReturn;
    }
}
```

7.4.2 Example of Using Iterator

```
import java.util.*;

public class TestIterator {
    public static void main(String[] args) {
        List list=new ArrayList();
        Map map=new HashMap();
        for(int i=0;i<10;i++){
            list.add(new String("list"+i) );
            map.put(i, new String("map"+i));
        }
        Iterator iterList= list.iterator(); //List接口实现了Iterable接口
        while(iterList.hasNext()){
            String strList=(String)iterList.next();
            System.out.println(strList.toString());
        }
        Iterator iterMap=map.entrySet().iterator();
        while(iterMap.hasNext()){
            Map.Entry  strMap=(Map.Entry)iterMap.next();
            System.out.println(strMap.getValue());
        }
    }
}
```

- forEach不是关键字,关键字还是for,语句是由iterator实现的，使用for each循环语句的优势在于更加

简洁，更不容易出错，不必关心下标的起始值和终止值。
，他们**最大的不同之处**就在于``remove(`方法上。

一般调用删除和添加方法都是具体集合的方法，例
如：`List list = new ArrayList(); list.add(...); list.remove(...);`
但是，如果在循环的过程中调用集合的remove()方法，就会导致循环出错，因为循环过程中list.size()的大小变化了，就会导致错误。所以，如果想在循环语句中删除集合中的某个元素，就要用迭代器iterator的remove()方法，因为它的remove()方法不仅会删除元素，还会维护一个标志，用来记录目前是不是可删除状态，例如，你不能连续两次调用它的remove()方法，调用之前至少有一次next()方法的调用。

forEach就是为了让用iterator循环访问的形式简单，写起来更方便。当然功能不太全,所以但如有删除操作，还是要用它原来的形式。

7.4.3 使用for循环与使用迭代器iterator的对比

- 效率上的各有优势
  = 采用ArrayList对随机访问比较快，而for循环中的get()方法，采用的即是随机访问的方法，因此在ArrayList里，for循环较快
  - 采用LinkedList则是顺序访问比较快，iterator中的next()方法，采用的即是顺序访问的方法，因此在LinkedList里，使用iterator较快
- 数据结构角度分析
  - for循环适合访问顺序结构,可以根据下标快速获取指定元素.
  - Iterator 适合访问链式结构,
    因为迭代器是通过next()和Pre()来定位的.可以访问没有顺序的集合.
- 使用 Iterator 的好处在于可以使用相同方式去遍历集合中元素，而不用考虑集合类的内部实现（只要它实现了 java.lang.Iterable 接口）
  - Example:
    - 使用 Iterator 来遍历集合中元素，一旦不再使用 List 转而使用 Set 来组织数据，那遍历元素的代码不用做任何修改，
    - 使用 for 来遍历，那所有遍历此集合的算法都得做相应调整
      因为List有序, Set无序, 结构不同, 他们的访问算法也不一样.

8. Tree

8.1 General Trees

8.1.1 A Tree Interface in Java

```java
public interface Tree<E> extends Iterable<E> {
    /* Returns the position of the root of the tree
(or null if empty). */
    Position<E> root( );
    /* Returns the position of the parent of position p (or null if p is the roo
t). */
    Position<E> parent(Position<E> p) throws IllegalArgumentException;
    /* Returns an iterable collection containing the children of position p (if
any). */
    Iterable<Position<E>> children(Position<E> p) throws
IllegalArgumentException;
    /* Returns the number of children of position p. */
    int numChildren(Position<E> p) throws IllegalArgumentException;

// Query method

/* Returns true if position p has at least one child. */
    boolean isInternal(Position<E> p) throws IllegalArgumentException;
    /* Returns true if position p does not have any children. */
    boolean isExternal(Position<E> p) throws IllegalArgumentException;
    /* Returns true if position p is the root of the tree. */
    boolean isRoot(Position<E> p) throws IllegalArgumentException;

// General method

    /* Returns the number of positions (and hence elements)
that are contained in the tree.*/
    int size( );
    /* Returns true if the tree does not contain any positions(and thus no eleme
nts). */
    boolean isEmpty( );
    /* Returns an iterator for all elements in the tree
(so that the tree itself is Iterable). */
    Iterator<E> iterator( );
    /* Returns an iterable collection of all positions of the tree. */
    Iterable<Position<E>> positions( );
}
```

8.1.1 Depth and height

- Depth 指的是某一个节点的深度

```java
/** Returns the number of levels separating Position p from the root. */
public int depth(Position<E> p) {
    if (isRoot(p))
        return 0;
    else
        return 1 + depth(parent(p));
}
```

- Height 指的是整棵树的最大深度， Height = max(Depth_sets)

```
/** Returns the height of the tree. */
private int heightBad( ) { // works, but quadratic worst-case time
    int h = 0;
    for (Position<E> p : positions( ))
        if (isExternal(p)) // only consider leaf positions
            h = Math.max(h, depth(p));
    return h;
}
```

这个方法的复杂度很高为$O(n^2)$， 高效的计算整棵树的height的方法如下

```
/** Returns the height of the subtree rooted at Position p. */
public int height(Position<E> p) {
    int h = 0; // base case if p is external
    for (Position<E> c : children(p))
        h = Math.max(h, 1 + height(c));
    return h;
}
```

这里利用了动态规划的思想从下往上计算每棵树的子节点的最大值，基于这个最大值进行后续的计算，这样的算法复杂度为$O(n)$

8.2 Binary Tree

8.2.1 BinaryTree interface

```
/** An interface for a binary tree, in which each node has at most two children.
*/
public interface BinaryTree<E> extends Tree<E> {
    /** Returns the Position of p's left child (or null if no child exists). */
    Position<E> left(Position<E> p) throws IllegalArgumentException;
    /** Returns the Position of p's right child (or null if no child exists). */
    Position<E> right(Position<E> p) throws IllegalArgumentException;
    /** Returns the Position of p's sibling (or null if no sibling exists). */
    Position<E> sibling(Position<E> p) throws IllegalArgumentException;
}
```

8.2.2 Binary Tree 的抽象类

- AbstractBinaryTree继承了Abstract的方法并对部分方法进行了重写， 另外还实现了BinaryTree的接口

```
public abstract class AbstractBinaryTree<E> extends AbstractTree<E> implements B
inaryTree<E> {
```

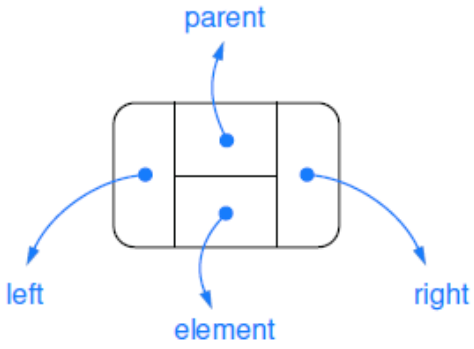
```
    /** Returns the Position of p's sibling (or null if no sibling exists). */
    public Position<E> sibling(Position<E> p) {
        Position<E> parent = parent(p);
        if (parent == null)
            return null; // p must be the root
        if (p == left(parent)) // p is a left child
            return right(parent); // (right child might be null)
        else // p is a right child
            return left(parent); // (left child might be null)
    }
    /** Returns the number of children of Position p. */
    public int numChildren(Position<E> p) {
        int count=0;
        if (left(p) != null)
            count++;
        if (right(p) != null)
            count++;
        return count;
    }
    /** Returns an iterable collection of the Positions representing p's childre
n. */
    public Iterable<Position<E>> children(Position<E> p) { List<Position<E>> snap
shot = new ArrayList<>(2); // max capacity of 2
        if (left(p) != null)
            snapshot.add(left(p));
        if (right(p) != null)
            snapshot.add(right(p));
        return snapshot;
    }
}
```

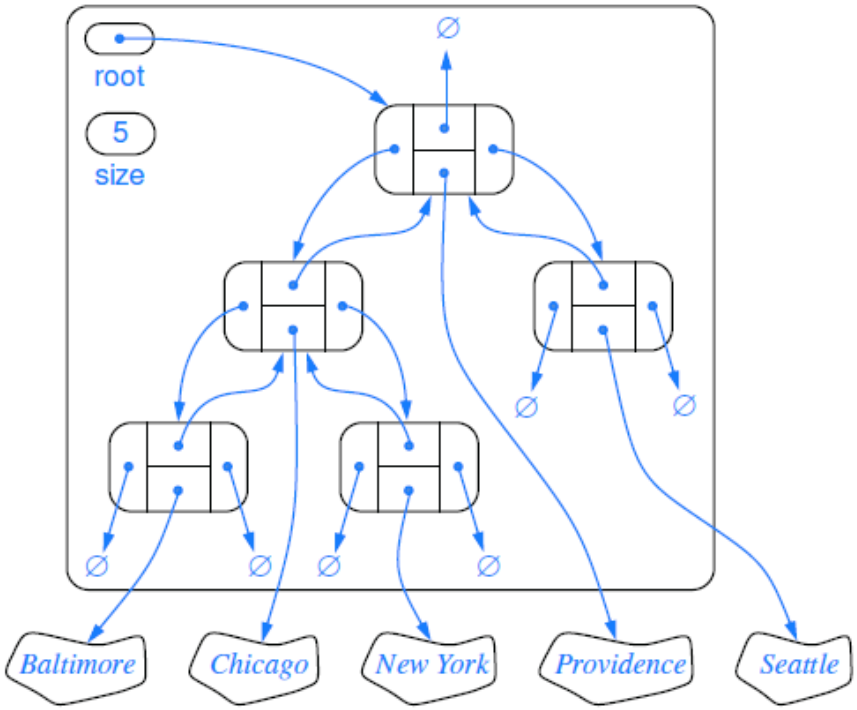
8.2.3 Binary Tree 的性质

- T是一棵非空Binary Tree, n, n_E, n_I, h 分别代表节点总数， 叶子节点数， 中间节点数和树的高度。
  - $h+1 \leq n \leq 2^{h+1}-1$
  - $1 \leq n_E \leq 2^h$
  - $h \leq n_I \leq 2^h-1$
  - $log(n+1)-1 \leq h \leq n-1$
- 当T为一棵完全二叉树时，它具有如下的性质
  - $2h+1 \leq n \leq 2^{h+1}-1$
  - $h+1 \leq n_E \leq 2^h$
  - $h \leq n_I \leq 2^h-1$
  - $log(n+1)-1 \leq h \leq \frac{n-1}{2}$

8.3 Implement Trees

8.3.1 Implementing Binary Tree






```java
/** Concrete implementation of a binary tree using a node-based, linked structure. */
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {
//---------------- nested Node class ----------------
    protected static class Node<E> implements Position<E> {
        private E element; // an element stored at this node
        private Node<E> parent; // a reference to the parent node (if any)
        private Node<E> left; // a reference to the left child (if any)
        private Node<E> right; // a reference to the right child (if any)
        /** Constructs a node with the given element and neighbors. */
        public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild){
            element = e;
            parent = above;
            left = leftChild;
            right = rightChild;
        }

        // accessor methods

        public E getElement( ) { return element; }
        public Node<E> getParent( ) { return parent; }
        public Node<E> getLeft( ) { return left; }
        public Node<E> getRight( ) { return right; }

        // update methods
        public void setElement(E e) { element = e; }
        public void setParent(Node<E> parentNode) { parent = parentNode; }
```

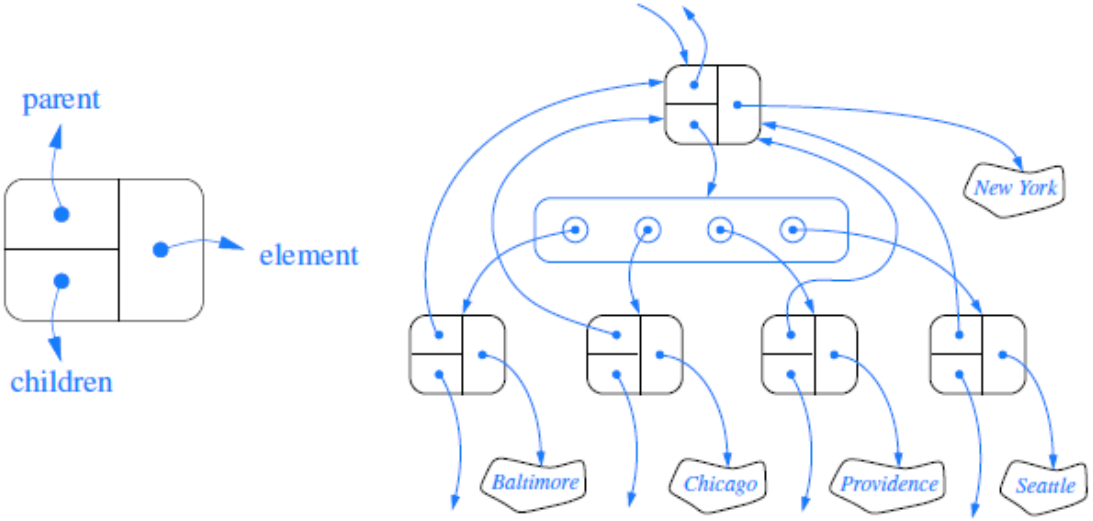
```
27.         public void setLeft(Node<E> leftChild) { left = leftChild; }
28.         public void setRight(Node<E> rightChild) { right =    rightChild; }
29.     }
30.
31.     //----------- end of nested Node class -----------
32.
33.     /** Factory function to create a new node storing element e. */
34.     protected Node<E> createNode(E e, Node<E> parent, Node<E> left, Node<E> right
    ) {
35.         return new Node<E>(e, parent, left, right);
36.     }
37.
38.     // LinkedBinaryTree instance variables
39.     protected Node<E> root = null; // root of the tree
40.     private int size = 0; // number of nodes in the tree
41.
42.     // constructor
43.     public LinkedBinaryTree( ) { } // constructs an empty binary tree
44.     // nonpublic utility
45.     /** Validates the position and returns it as a node. */
46.     protected Node<E> validate(Position<E> p) throws IllegalArgumentException {
47.         if (!(p instanceof Node))
48.             throw new IllegalArgumentException("Not valid position type");
49.         Node<E> node = (Node<E>) p; // safe cast
50.         if (node.getParent( ) == node) // our convention for defunct node
51.             throw new IllegalArgumentException("p is no longer in the tree");
52.         return node;
53.     }
54.
55.     // accessor methods (not already implemented in AbstractBinaryTree)
56.     /** Returns the number of nodes in the tree. */
57.     public int size( ) {
58.         return size;
59.     }
60.
61.     /** Returns the root Position of the tree (or null if tree is empty). */
62.     public Position<E> root( ) {
63.         return root;
64.     }
65.     /**Returns the Position of p's parent (or null if p is root). */
66.     public Position<E> parent(Position<E> p) throws IllegalArgumentException {
67.         Node<E> node = validate(p);
68.         return node.getParent( );
69.     }
70.
71.     /** Returns the Position of p's left child (or null if no child exists). */
72.     public Position<E> left(Position<E> p) throws IllegalArgumentException {
73.         Node<E> node = validate(p);
74.         return node.getLeft( );
75.     }
```

```
76.     /** Returns the Position of p's right child (or null if no child exists). */
77.     public Position<E> right(Position<E> p) throws IllegalArgumentException {
78.         Node<E> node = validate(p);
79.         return node.getRight( );
80.     }
81.
82.     // update methods supported by this class
83.     /** Places element e at the root of an empty tree and returns its new Positi
    on. */
84.     public Position<E> addRoot(E e) throws IllegalStateException {
85.         if (!isEmpty( )) throw new IllegalStateException("Tree is not empty");
86.             root = createNode(e, null, null, null);
87.             size = 1;
88.         return root;
89.     }
90.
91.     /** Creates a new left child of Position p storing element e; returns its Po
    sition. */
92.     public Position<E> addLeft(Position<E> p, E e) throws
    IllegalArgumentException {
93.         Node<E> parent = validate(p);
94.         if (parent.getLeft( ) != null)
95.             throw new IllegalArgumentException("p already has a left child");
96.         Node<E> child = createNode(e, parent, null, null);
97.         parent.setLeft(child);
98.         size++;
99.         return child;
100.    }
101.    /** Creates a new right child of Position p storing element e; returns its P
    osition. */
102.    public Position<E> addRight(Position<E> p, E e) throws
    IllegalArgumentException {
103.        Node<E> parent = validate(p);
104.        if (parent.getRight( ) != null)
105.            throw new IllegalArgumentException("p already has a right child");
106.        Node<E> child = createNode(e, parent, null, null);
107.        parent.setRight(child);
108.        size++;
109.        return child;
110.    }
111.
112.    /** Replaces the element at Position p with e and returns the replaced eleme
    nt. */
113.    public E set(Position<E> p, E e) throws IllegalArgumentException {Node<E> nod
    e = validate(p);
114.        E temp = node.getElement( );
115.        node.setElement(e);
116.        return temp;
117.    }
118.
```

```
119.     /** Attaches trees t1 and t2 as left and right subtrees of external p. */
120.     public void attach(Position<E> p, LinkedBinaryTree<E> t1, LinkedBinaryTree<E>
    t2) throws IllegalArgumentException {
121.         Node<E> node = validate(p);
122.         if (isInternal(p))
123.             throw new IllegalArgumentException("p must be a leaf");
124.         size += t1.size( ) + t2.size( );
125.         if (!t1.isEmpty( )) { // attach t1 as left subtree of node
126.             t1.root.setParent(node);
127.             node.setLeft(t1.root);
128.             t1.root = null;
129.             t1.size = 0;
130.         }
131.         if (!t2.isEmpty( )) { // attach t2 as right subtree of node
132.             t2.root.setParent(node);
133.             node.setRight(t2.root);
134.             t2.root = null;
135.             t2.size = 0;
136.         }
137.     }
138.     /** Removes the node at Position p and replaces it with its child, if any. *
    /
139.     public E remove(Position<E> p) throws IllegalArgumentException {
140.         Node<E> node = validate(p);
141.         if (numChildren(p) == 2)
142.             throw new IllegalArgumentException("p has two children");
143.         Node<E> child = (node.getLeft( ) != null ? node.getLeft( ) : node.getRigh
    t( ) );
144.         if (child != null)
145.             child.setParent(node.getParent( )); // child's grandparent becomes
    its parent
146.         if (node == root)
147.             root = child; // child becomes root
148.         else {
149.             Node<E> parent = node.getParent( );
150.             if (node == parent.getLeft( ))
151.                 parent.setLeft(child);
152.             else
153.                 parent.setRight(child);
154.         }
155.         size--;
156.         E temp = node.getElement( );
157.         node.setElement(null); // help garbage collection
158.         node.setLeft(null);
159.         node.setRight(null);
160.         node.setParent(node); // our convention for defunct node
161.         return temp;
162.     }
163. }
164. //----------- end of LinkedBinaryTree class -----------
```

- 复杂度除了depth和height，为$O(d_p+1)$和$O(n)$，其他都为$O(1)$

8.3.2 Implement of GeneralTree

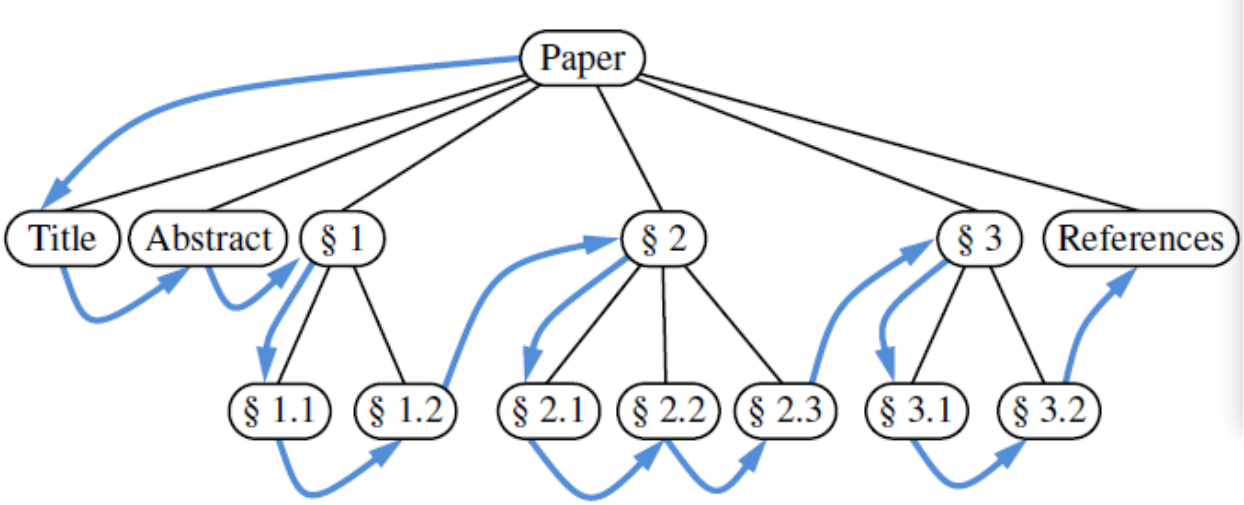


Method	Running Time
size, isEmpty	$O(1)$
root, parent, isRoot, isInternal, isExternal	$O(1)$
numChildren(p)	$O(1)$
children(p)	$O(c_p+1)$
depth(p)	$O(d_p+1)$
height	$O(n)$

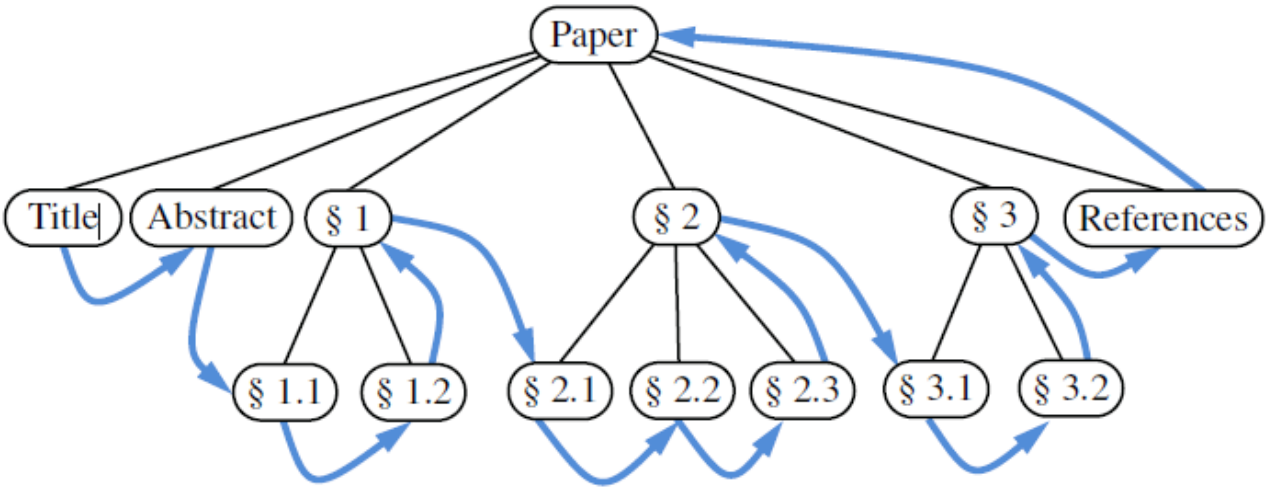
8.4 Tree Traversal Algorithms

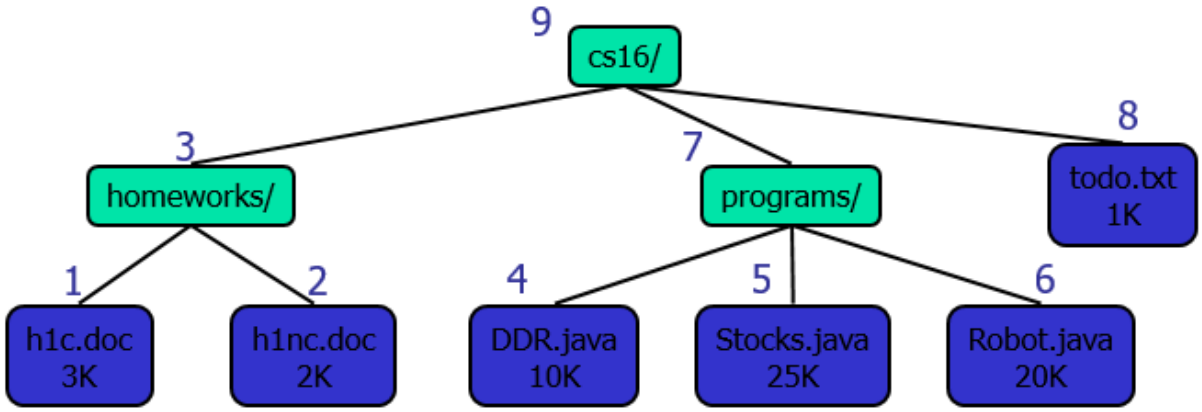
8.4.1 Preorder and Postorder Traversals

- Preorder(先序)

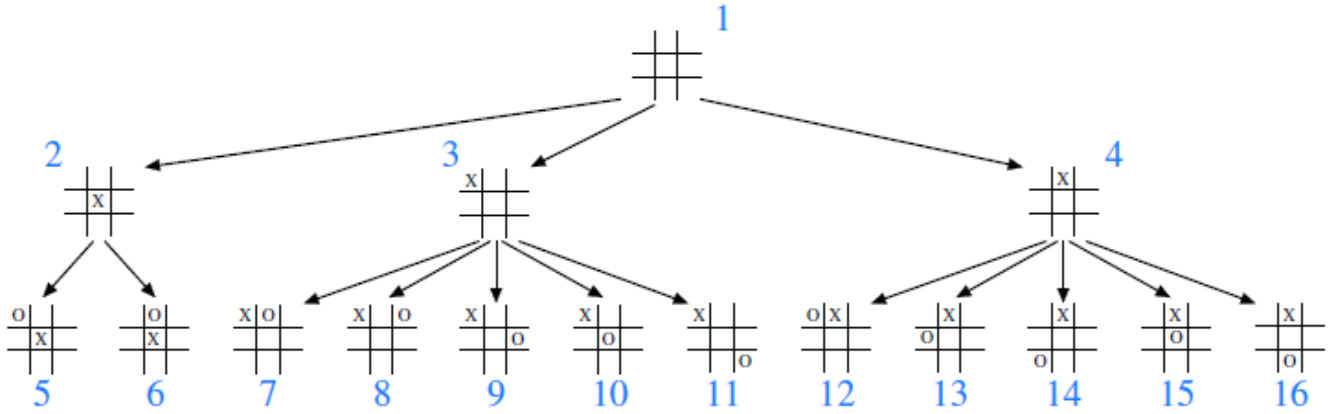


- Postorder(后序)



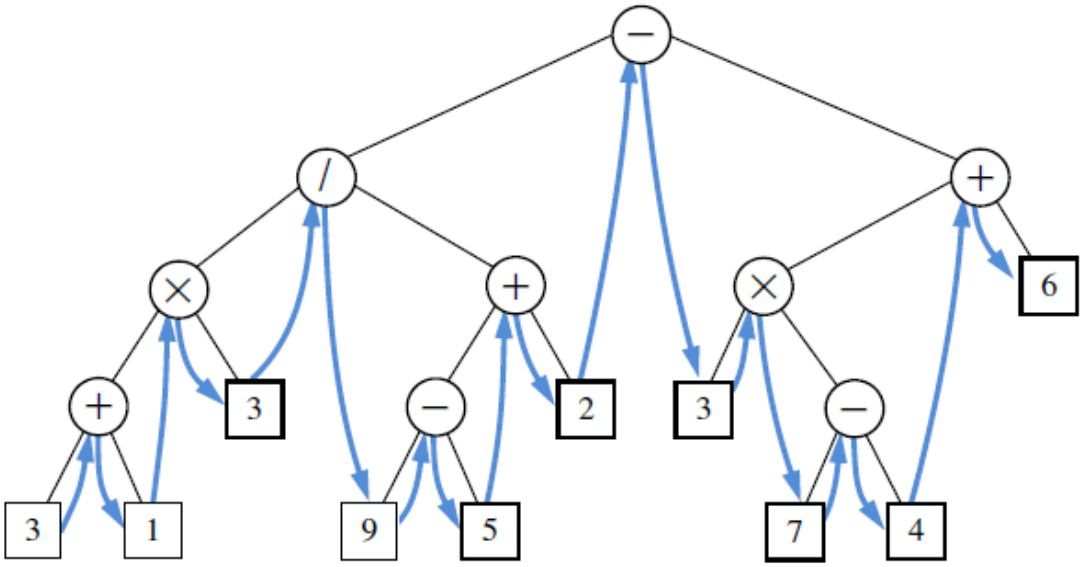


8.4.1 Breadth-First traversal (广度优先)




```
Algorithm breadthfirst( ):
    Initialize queue Q to contain root( )
    while Q not empty do
        p = Q.dequeue( ) { p is the oldest entry in the queue }
        perform the "visit" action for position p
        for each child c in children(p) do
            Q.enqueue(c) { add p's children to the end of the queue for later vis
its }
```

8.4.3 Inorder Traversal




```
Algorithm inorder(p):
    if p has a left child lc then
        inorder(lc) { recursively traverse the left subtree of p }
    perform the "visit" action for position p
    if p has a right child rc then
        inorder(rc) { recursively traverse the right subtree of p }
```

8.4.4 Implement Tree Traversals

```
//---------------- nested ElementIterator class ----------------
/* This class adapts the iteration produced by positions() to return elements. *
/
private class ElementIterator implements Iterator<E> {
    Iterator<Position<E>> posIterator = positions( ).iterator( );
    public boolean hasNext( ) { return posIterator.hasNext( ); }
    public E next( ) { return posIterator.next( ).getElement( ); } // return
element!
    public void remove( ) { posIterator.remove( ); }
}

/** Returns an iterator of the elements stored in the tree. */
    public Iterator<E> iterator( ) { return new ElementIterator( );
}

// 例子
public Iterable<Position<E>> positions( ) { return preorder( ); }
```

- Preorder Traveral

```
/** Adds positions of the subtree rooted at Position p to the given snapshot. */
private void preorderSubtree(Position<E> p, List<Position<E>> snapshot) {
```

```
        snapshot.add(p); // for preorder, we add position p before exploring subtree
s
        for (Position<E> c : children(p))
            preorderSubtree(c, snapshot);
    }

    /** Returns an iterable collection of positions of the tree, reported in preorde
r. */
    public Iterable<Position<E>> preorder( ) {
        List<Position<E>> snapshot = new ArrayList<>( );
        if (!isEmpty( ))
            preorderSubtree(root( ), snapshot); // fill the snapshot recursively
        return snapshot;
    }
```

- Postorder Traversal

```
    /** Adds positions of the subtree rooted at Position p to the given snapshot. */
    private void postorderSubtree(Position<E> p, List<Position<E>> snapshot) {
        for (Position<E> c : children(p))
            postorderSubtree(c, snapshot);
        snapshot.add(p); // for postorder, we add position p after exploring subtree
s
    }

    /** Returns an iterable collection of positions of the tree, reported in postord
er. */
    public Iterable<Position<E>> postorder( ) {
        List<Position<E>> snapshot = new ArrayList<>( );
        if (!isEmpty( ))
            postorderSubtree(root( ), snapshot); // fill the snapshot recursively
        return snapshot;
    }
```

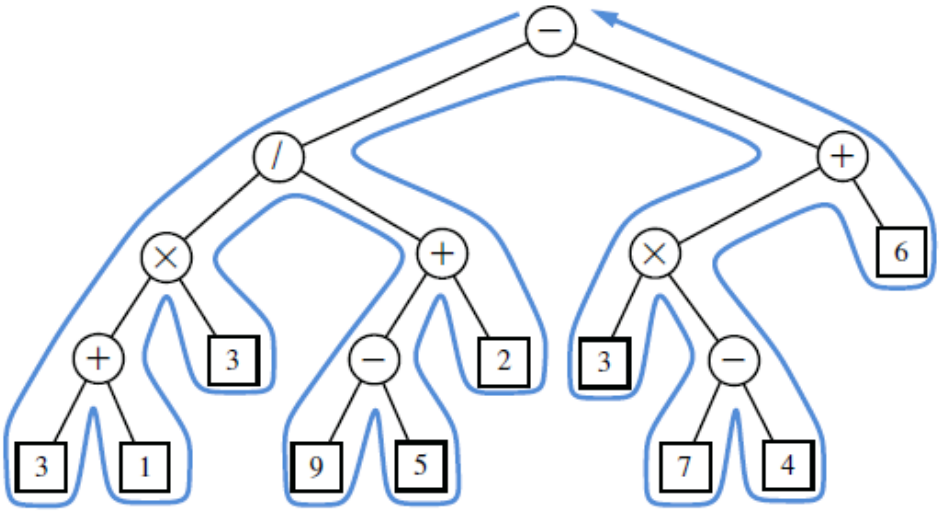
- Breadth-First traversal

```
    /** Returns an iterable collection of positions of the tree in breadth-first ord
er. */
    public Iterable<Position<E>> breadthfirst( ) {
        List<Position<E>> snapshot = new ArrayList<>( );
        if (!isEmpty( )) {
            Queue<Position<E>> fringe = new LinkedQueue<>( );
            fringe.enqueue(root( )); // start with the root
            while (!fringe.isEmpty( )) {
                Position<E> p = fringe.dequeue( ); // remove from front of the queue
                snapshot.add(p); // report this position
                for (Position<E> c : children(p))
                    fringe.enqueue(c); // add children to back of queue
            }
```

```
        }
        return snapshot;
    }
```

- Inorder Traversal
- Tree Traversals的应用
  - Table of Contents (目录)
  - Computing Disk Space

8.4.5 Euler tour



- 和中序遍历Inorder Traversal是一致的

9. Priority Queue

9.1 Priority Queue

9.1.1 Definition and Property

- Priority Queue储存了是entry(key, value)的集合
  - 两个不同的entries可以有相同的key
  - 在Priority Queue中key可以是任意的对象，但是它是有序的, 是可比较的，即实现了java.lang.Comparable接口，a.compareTo(b)会返回一个整数值i
    - $i < 0$ 表示$a < b$
    - $i = 0$ 表示$a = b$
    - $i > 0$ 表示$a > b$
    - Example：

```java
// Example_1:
public class StringLengthComparator implements Comparator<String> {
    /** Compares two strings according to their lengths. */
    public int compare(String a, String b) {
        if (a.length( ) < b.length( )) return -1;
        else if (a.length( ) == b.length( )) return 0;
        else return 1;
    }
}

// // Example_2: 实现按自然顺序进行排列
public class DefaultComparator<E> implements Comparator<E> {
    public int compare(E a, E b) throws ClassCastException {
        return ((Comparable<E>) a).compareTo(b);
    }
}
```

- 主要的方法有:
  - insert(k, x): Inserts an entry with key k and value x.
  - removeMin(): Removes and returns the entry with smallest key.
  - min(): returns, but does not remove, an entry with smallest key
  - size()
  - isEmpty()

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

- Applications:
  - Standby flyers(后备飞行员)
  - Auctions(拍卖)

- Stock market(证券市场)

- entry：

```
public interface  Entry<K, V> {
    // return the key for this entry
    public K key();
    // return the value for this entry
    public V value();
}

/** Interface for the priority queue ADT. */
public interface PriorityQueue<K,V> {
    int size( );
    boolean isEmpty( );
    Entry<K,V> insert(K key, V value) throws IllegalArgumentException;
    Entry<K,V> min( );
    Entry<K,V> removeMin( );
}
```

9.1.2 Implement PriorityQueue with abstract class

```
/** An abstract base class to assist implementations of the PriorityQueue interface.*/
public abstract class AbstractPriorityQueue<K,V> implements PriorityQueue<K,V> {
    //---------------- nested PQEntry class ----------------
    protected static class PQEntry<K,V> implements Entry<K,V> {
        private K k; // key
        private V v; // value
        public PQEntry(K key, V value) {
            k = key;
            v = value;
        }
        // methods of the Entry interface
        public K getKey( ) { return k; }
        public V getValue( ) { return v; }
        // utilities not exposed as part of the Entry interface
        protected void setKey(K key) { k = key; }
        protected void setValue(V value) { v = value; }
    }
    //----------- end of nested PQEntry class -----------

    // instance variable for an AbstractPriorityQueue
    /* The comparator defining the ordering of keys in the priority queue. */
    private Comparator<K> comp;
    /** Creates an empty priority queue using the given comparator to order keys. */
    protected AbstractPriorityQueue(Comparator<K> c) { comp = c; }
    /** Creates an empty priority queue based on the natural ordering of its keys. */
```

```
26.     protected AbstractPriorityQueue( ) { this(new DefaultComparator<K>( )); }
27.
28.     /** Method for comparing two entries according to key */
29.     protected int compare(Entry<K,V> a, Entry<K,V> b) {
30.         return comp.compare(a.getKey( ), b.getKey( ));
31.     }
32.
33.     /** Determines whether a key is valid. */
34.     protected boolean checkKey(K key) throws IllegalArgumentException {
35.         try {
36.             return (comp.compare(key,key) == 0); // see if key can be compared to
    itself
37.         } catch (ClassCastException e) {
38.             throw new IllegalArgumentException("Incompatible key");
39.         }
40.     }
41.     /** Tests whether the priority queue is empty. */
42.     public boolean isEmpty( ) { return size( ) == 0; }
43. }
```

9.1.3 Implement priorityQueue with Sorted List and unsorted list

Method	Unsorted List	Sorted List
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

- unsorted list

```
1.  /** An implementation of a priority queue with an unsorted list. */
2.  public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3.      /** primary collection of priority queue entries */
4.      private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>( );
5.      /** Creates an empty priority queue based on the natural ordering of its key
    s. */
6.      public UnsortedPriorityQueue( ) { super( ); }
7.      /** Creates an empty priority queue using the given comparator to order keys
    . */
8.      public UnsortedPriorityQueue(Comparator<K> comp) { super(comp); }
9.      /** Returns the Position of an entry having minimal key. */
10.     private Position<Entry<K,V>> findMin( ) {
11.         // only called when nonempty
12.         Position<Entry<K,V>> small = list.first( );
13.         for (Position<Entry<K,V>> walk : list.positions( )){
14.             if (compare(walk.getElement( ), small.getElement( )) < 0)
```

```
                small = walk; // found an even smaller key
            }
            return small;
        }
        /** Inserts a key-value pair and returns the entry created. */
        public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
            checkKey(key); // auxiliary key-checking method (could throw exception)
            Entry<K,V> newest = new PQEntry<>(key, value);
            list.addLast(newest);
            return newest;
        }
        /** Returns (but does not remove) an entry with minimal key. */
        public Entry<K,V> min( ) {
            if (list.isEmpty( ))
                return null;
            return findMin( ).getElement( );
        }
        /** Removes and returns an entry with minimal key. */
        public Entry<K,V> removeMin( ) {
            if (list.isEmpty( ))
                return null;
            return list.remove(findMin( ));
        }
        /** Returns the number of items in the priority queue. */
        public int size( ) { return list.size( ); }
    }
```

- sorted list

```
/** An implementation of a priority queue with a sorted list. */
public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
    /** primary collection of priority queue entries */
    private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>( );

    /** Creates an empty priority queue based on the natural ordering of its keys. */
    public SortedPriorityQueue( ) { super( ); }
    /** Creates an empty priority queue using the given comparator to order keys. */
    public SortedPriorityQueue(Comparator<K> comp) { super(comp); }
    /** Inserts a key-value pair and returns the entry created. */
    public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
        checkKey(key); // auxiliary key-checking method (could throw exception)
        Entry<K,V> newest = new PQEntry<>(key, value);
        Position<Entry<K,V>> walk = list.last( );
        // walk backward, looking for smaller key
        while (walk != null && compare(newest, walk.getElement( )) < 0)
            walk = list.before(walk);
        if (walk == null)
            list.addFirst(newest); // new key is smallest
```

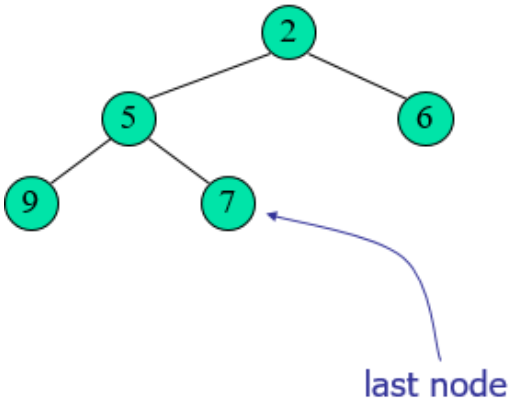
```
            else
            list.addAfter(walk, newest); // newest goes after walk
            return newest;
        }
        /** Returns (but does not remove) an entry with minimal key. */
        public Entry<K,V> min( ) {
            if (list.isEmpty( ))
                    return null;
            return list.first( ).getElement( );
        }
        /** Removes and returns an entry with minimal key. */
        public Entry<K,V> removeMin( ) {
            if (list.isEmpty( ))
                return null;
            return list.remove(list.first( ));
        }
        /** Returns the number of items in the priority queue. */
        public int size( ) { return list.size( ); }
    }
```

9.2 Heap

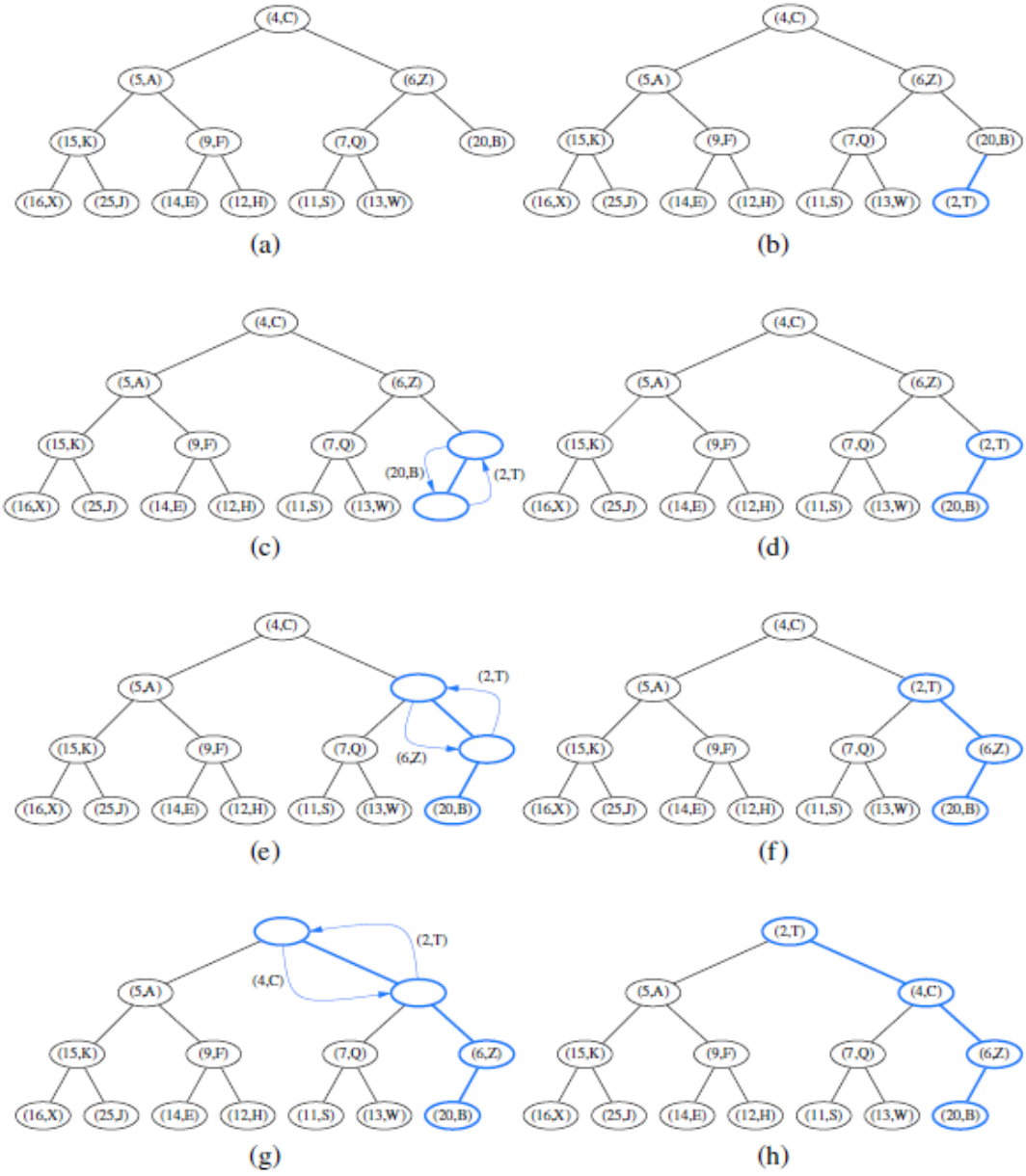
Heap is a kind of specific PriorityQueue

9.2.1 Definition and Property

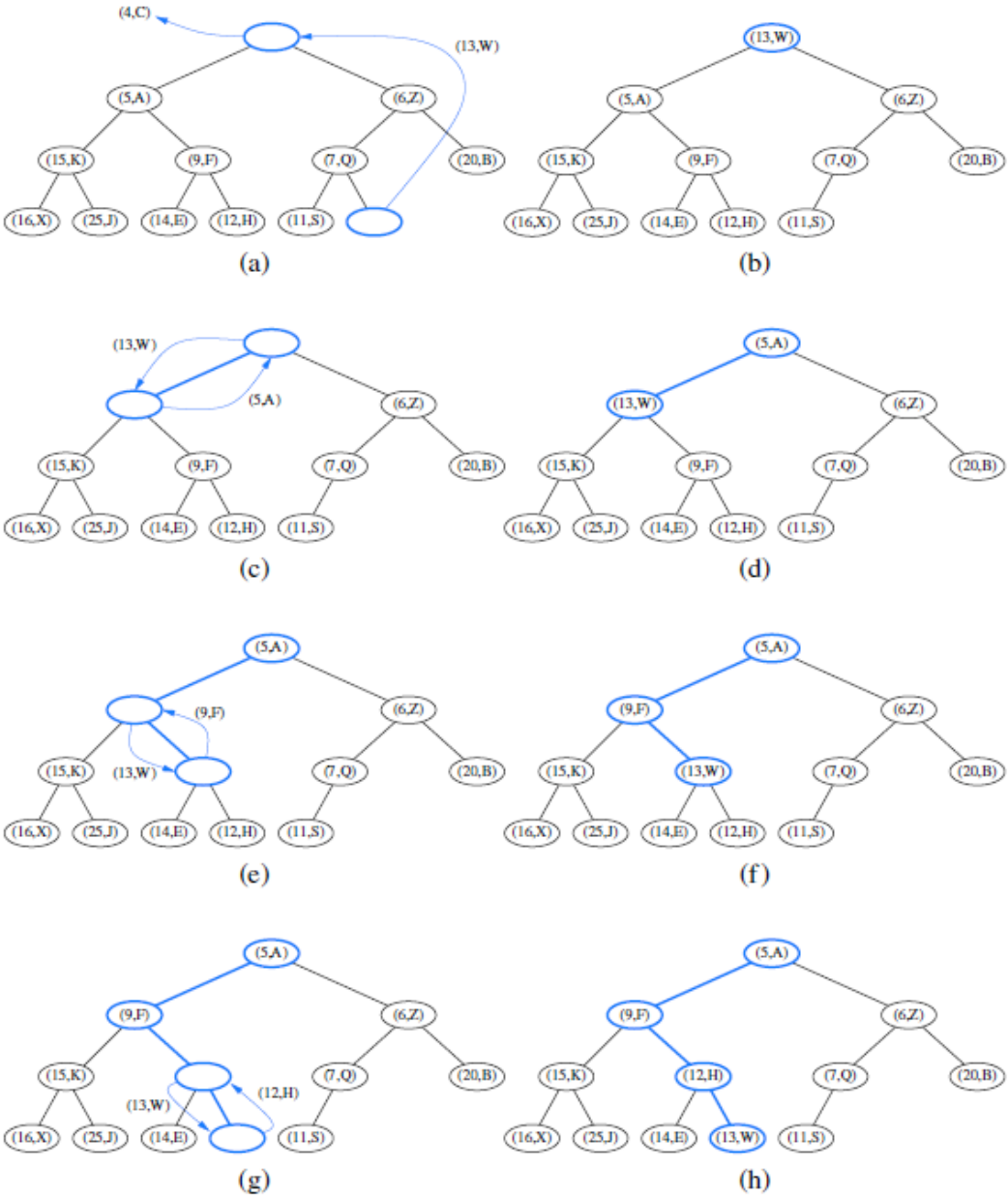
- Heap是一棵满足如下性质的存储了(key, value)的完整二叉树
  - Heap-Order: $key(v) \geq key(parent(v))$



- Heap的高度$hlog(n)$, n为节点总数
- Heap插入节点



- Heap删除节点



Method	Running Time
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)^*$
removeMin	$O(\log n)^*$

9.2.2 Implement heap with array

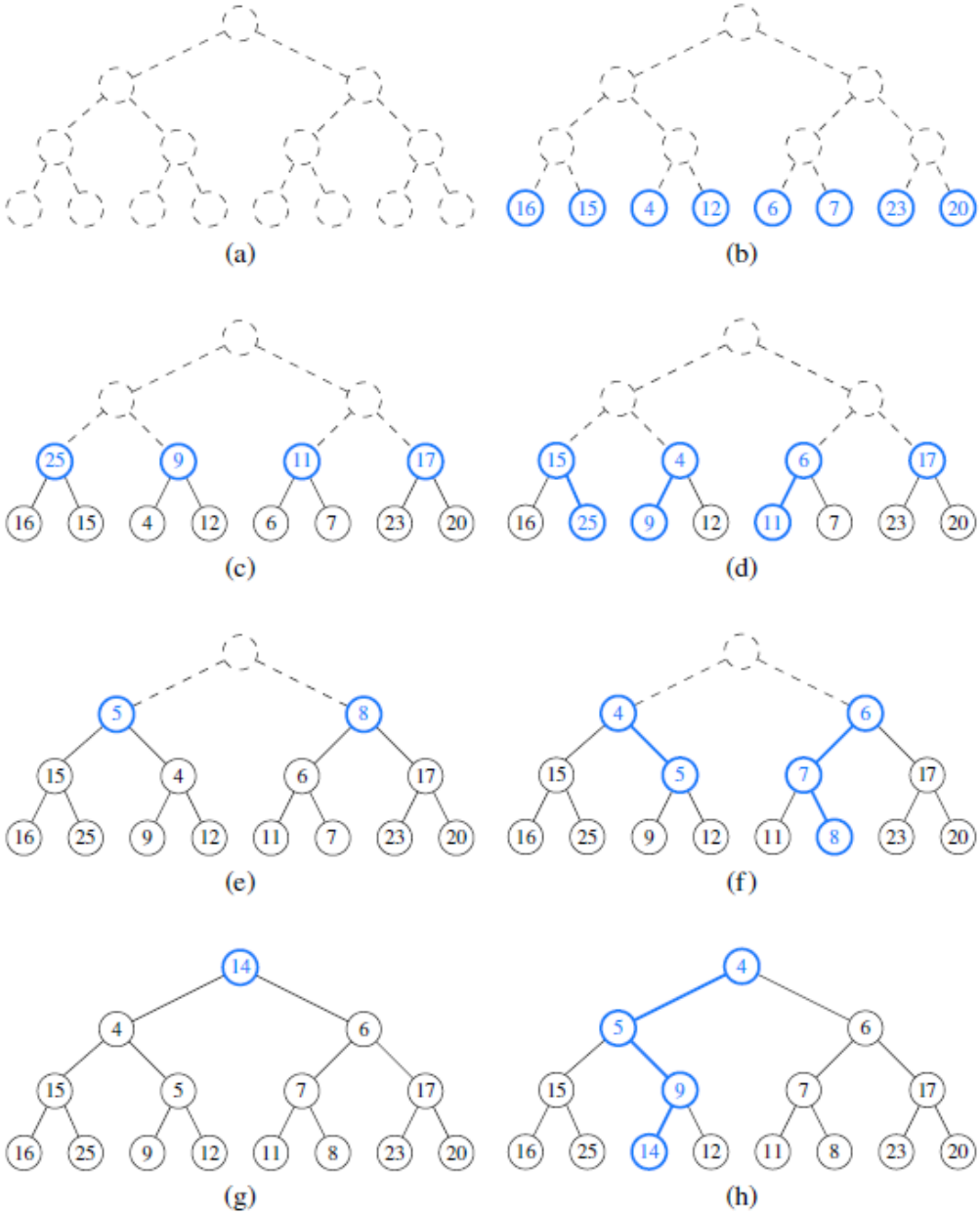
```
1.  /** An implementation of a priority queue using an array-based heap. */
2.  public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3.      /** primary collection of priority queue entries */
4.      protected ArrayList<Entry<K,V>> heap = new ArrayList<>( );
5.      /** Creates an empty priority queue based on the natural ordering of its key
    s. */
6.      public HeapPriorityQueue( ) { super( ); }
7.      /** Creates an empty priority queue using the given comparator to order keys
```

```
. */
    public HeapPriorityQueue(Comparator<K> comp) { super(comp); } // protected utilities
    protected int parent(int j) { return (j-1) / 2; } // truncating division
    protected int left(int j) { return 2*j + 1; }
    protected int right(int j) { return 2*j + 2; }
    protected boolean hasLeft(int j) { return left(j) < heap.size( ); }
    protected boolean hasRight(int j) { return right(j) < heap.size( ); }
    /** Exchanges the entries at indices i and j of the array list. */
    protected void swap(int i, int j) {
        Entry<K,V> temp = heap.get(i);
        heap.set(i, heap.get(j));
        heap.set(j, temp);
    }
    /** Moves the entry at index j higher, if necessary, to restore the heap property. */
    protected void upheap(int j) {
        while (j > 0) {
            // continue until reaching root (or break statement)
            int p = parent(j);
            if (compare(heap.get(j), heap.get(p)) >= 0) break; // heap property verified
                swap(j, p);
            j = p; // continue from the parent's location
        }
    }
    /** Moves the entry at index j lower, if necessary, to restore the heap property. */
    protected void downheap(int j) {while (hasLeft(j)) {
        // continue to bottom (or break statement)
        int leftIndex = left(j);
        int smallChildIndex = leftIndex; // although right may be smaller
        if (hasRight(j)) {
            int rightIndex = right(j);
            if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
        smallChildIndex = rightIndex; // right child is smaller
        }
        if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
        break; // heap property has been restored
        swap(j, smallChildIndex);
        j = smallChildIndex; // continue at position of the child
        }
    }
    // public methods
    /** Returns the number of items in the priority queue. */
    public int size( ) { return heap.size( ); }

    /** Returns (but does not remove) an entry with minimal key (if any). */
    public Entry<K,V> min( ) {
        if (heap.isEmpty( ))
```

```
            return null;
        return heap.get(0);
    }
    /** Inserts a key-value pair and returns the entry created. */
    public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
        checkKey(key); // auxiliary key-checking method (could throw exception)
        Entry<K,V> newest = new PQEntry<>(key, value);
        heap.add(newest); // add to the end of the list
        upheap(heap.size( ) - 1); // upheap newly added entry
        return newest;
    }
    /** Removes and returns an entry with minimal key (if any). */
    public Entry<K,V> removeMin( ) {
        if (heap.isEmpty( ))
            return null;
        Entry<K,V> answer = heap.get(0);
        swap(0, heap.size( ) - 1); // put minimum item at the end
        heap.remove(heap.size( ) - 1); // and remove it from the list;
        downheap(0); // then fix new root
        return answer;
    }
}
```

9.2.3 Bottom-Up Heap Construction

用insert构建一个n节点的heap的复杂度为$O(nlog(n))$, 但是利用Bottom-Up来创建一个n节点的heap的复杂度为$O(n)$




```
1.  /** Creates a priority queue initialized with the given key-value pairs. */
2.  public HeapPriorityQueue(K[ ] keys, V[ ] values) {
3.      super( );
4.      for (int j=0; j < Math.min(keys.length, values.length); j++)
5.          heap.add(new PQEntry<>(keys[j], values[j]));
6.      heapify( );
7.      }
8.      /** Performs a bottom-up construction of the heap in linear time. */
9.      protected void heapify( ) { int startIndex = parent(size( )-1); // start at
    PARENT of last entry
10.     for (int j=startIndex; j >= 0; j--) // loop until processing the root
11.         downheap(j);
12. }
```

9.2.3 java.util.PriorityQueue

Java自带的PriorityQueue实现了heap的接口

* `add(new Entry(k, v))`插入节点$O(log(n))$
* `peek()`得到key值最小的节点
* `remove()`移除节点$O(n)$
* `size()`
* `isEmpty()`

9.3 sort with PriorityQueue and Heap

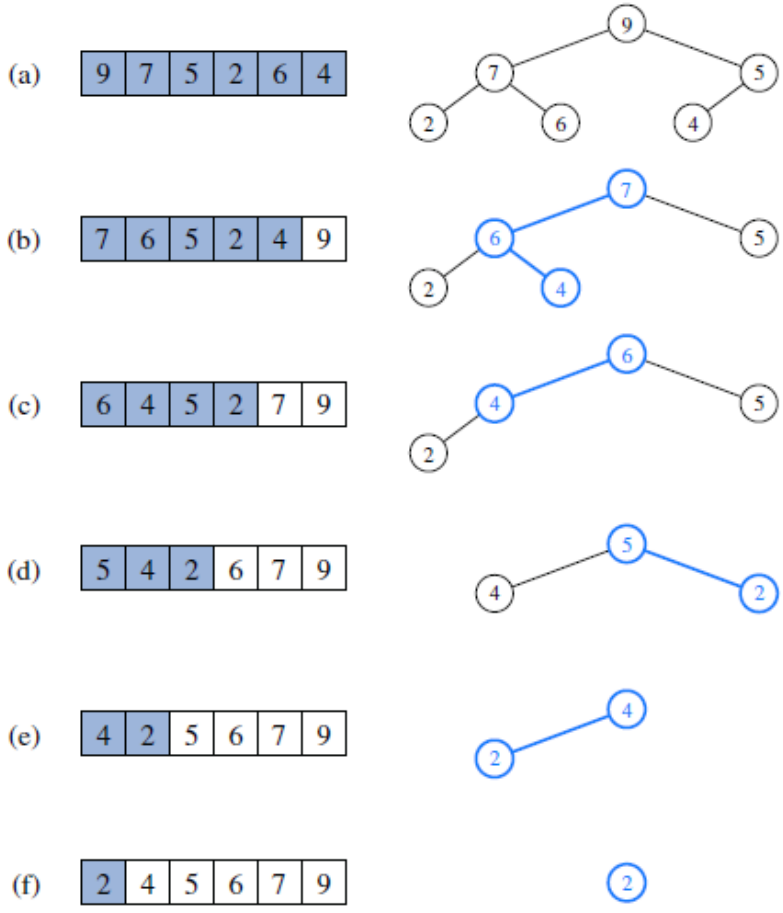
- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted list.$O(n^2)$

	List S	***Priority Queue*** P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	.. ..	
.	. .	
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted list.$O(n^2)$

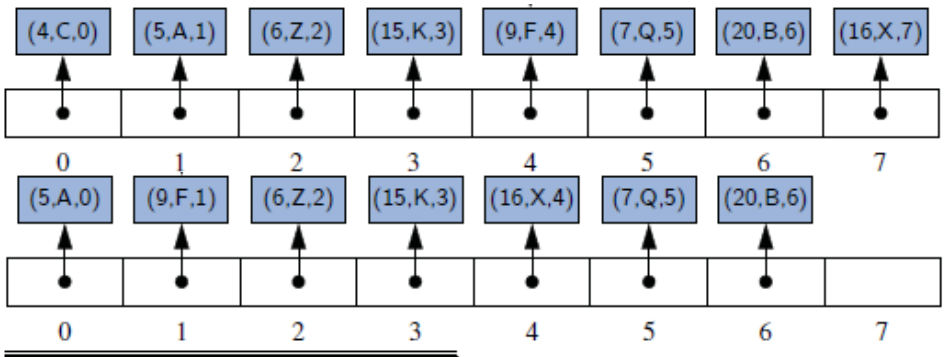
	List S	*Priority queue P*
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
.	.	.
(g)	(2,3,4,5,7,8,9)	()

- Heap-sort $O(nlog(n))$



9.4 Adaptable Priority Queues

Example(Location Aware Entries):



Method	Running Time
size, isEmpty, min	$O(1)$
insert	$O(\log n)$
remove	$O(\log n)$
removeMin	$O(\log n)$
replaceKey	$O(\log n)$
replaceValue	$O(1)$

- implementation

```
/** An implementation of an adaptable priority queue using an array-based heap. */
public class HeapAdaptablePriorityQueue<K,V> extends HeapPriorityQueue<K,V>
implements AdaptablePriorityQueue<K,V> {
    //---------------- nested AdaptablePQEntry class ----------------
    /** Extension of the PQEntry to include location information. */
    protected static class AdaptablePQEntry<K,V> extends PQEntry<K,V> {
    private int index; // entry's current index within the heap
    public AdaptablePQEntry(K key, V value, int j) {
        super(key, value); // this sets the key and value
        index = j; // this sets the new field
    }
    public int getIndex( ) { return index; }
    public void setIndex(int j) { index = j; }
    }
    //----------- end of nested AdaptablePQEntry class -----------

    /** Creates an empty adaptable priority queue using natural ordering of keys. */
    public HeapAdaptablePriorityQueue( ) { super( ); }
    /** Creates an empty adaptable priority queue using the given comparator. */
    public HeapAdaptablePriorityQueue(Comparator<K> comp) { super(comp);}
    // protected utilites
    /** Validates an entry to ensure it is location-aware. */
    protected AdaptablePQEntry<K,V> validate(Entry<K,V> entry) throws IllegalArgumentException {
        if (!(entry instanceof AdaptablePQEntry))
            throw new IllegalArgumentException("Invalid entry");
        AdaptablePQEntry<K,V> locator = (AdaptablePQEntry<K,V>) entry; // safe
        int j = locator.getIndex( );
```

```
27.         if (j >= heap.size( ) || heap.get(j) != locator)
28.         throw new IllegalArgumentException("Invalid entry");
29.         return locator;
30.     }
31.     /** Exchanges the entries at indices i and j of the array list. */
32.     protected void swap(int i, int j) {
33.         super.swap(i,j); // perform the swap
34.         ((AdaptablePQEntry<K,V>) heap.get(i)).setIndex(i); // reset entry's index
35.         ((AdaptablePQEntry<K,V>) heap.get(j)).setIndex(j); // reset entry's index
36.     }
37.     /** Restores the heap property by moving the entry at index j
    upward/downward.*/
38.     protected void bubble(int j) {
39.         if (j > 0 && compare(heap.get(j), heap.get(parent(j))) < 0)
40.             upheap(j);
41.         else
42.             downheap(j); // although it might not need to move
43.     }
44.     /** Inserts a key-value pair and returns the entry created. */
45.     public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
46.         checkKey(key); // might throw an exception
47.         Entry<K,V> newest = new AdaptablePQEntry<>(key, value, heap.size( ));
48.         heap.add(newest); // add to the end of the list
49.         upheap(heap.size( ) - 1); // upheap newly added entry
50.         return newest;
51.     }
52.     /** Removes the given entry from the priority queue. */
53.     public void remove(Entry<K,V> entry) throws IllegalArgumentException {
54.         AdaptablePQEntry<K,V> locator = validate(entry);
55.         int j = locator.getIndex( );
56.         if (j == heap.size( ) - 1) // entry is at last position
57.             heap.remove(heap.size( ) - 1); // so just remove it
58.         else {
59.             swap(j, heap.size( ) - 1); // swap entry to last position
60.             heap.remove(heap.size( ) - 1); // then remove it
61.             bubble(j); // and fix entry displaced by the swap
62.         }
63.     }
64.     /** Replaces the key of an entry. */
65.     public void replaceKey(Entry<K,V> entry, K key) throws
    IllegalArgumentException {
66.         AdaptablePQEntry<K,V> locator = validate(entry);
67.         checkKey(key); // might throw an exception
68.         locator.setKey(key); // method inherited from PQEntry
69.         bubble(locator.getIndex( )); // with new key, may need to move entry
70.     }
71.     /** Replaces the value of an entry. */
72.     public void replaceValue(Entry<K,V> entry, V value) throws IllegalArgumentExc
    eption {
73.         AdaptablePQEntry<K,V> locator = validate(entry);
```

```
            locator.setValue(value); // method inherited from PQEntry
    }
```

10. Maps, Hash Tables and Skip Lists

10.1 Maps

- Map是可搜索的键值对集合
  - Map中的key是唯一的
  - 主要operations有搜索，插入，删除

10.1.1 java.util.Map

- get(k): 如果map M中有k这个key则返回对应的值，否则返回null
- put(k, v): 向map M中插入键值对(k, v)如果k原本不存在则返回null, 否则返回v'，k在map M中原本对应的
- remove(k): 如果map M中含有键k，则将对应的键值对(k,v)移除，返回对应的v；否则返回null
- size(), isEmpty()
- keySet().iterator(): 返回一个包含M所有键k的迭代器
- values().iterator(): 返回一个包含M所有值v的迭代器

Operation	***Output***	***Map***
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
put(2,*C*)	**null**	(5,*A*),(7,*B*),(2,*C*)
put(8,*D*)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(4)	**null**	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(2)	*E*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
size()	4	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
remove(5)	*A*	(7,*B*),(2,*E*),(8,*D*)
remove(2)	*E*	(7,*B*),(8,*D*)
get(2)	**null**	(7,*B*),(8,*D*)
isEmpty()	**false**	(7,*B*),(8,*D*)

10.1.2 java的Map接口

```
public interface Map<K,V> {
    int size( );
    boolean isEmpty( );
    V get(K key);
```

```
    V put(K key, V value);
    V remove(K key);
    Iterable<K> keySet( );
    Iterable<V> values( );
    Iterable<Entry<K,V>> entrySet( );
}
```

10.1.3 Implement Map with unsorted list

- We can efficiently implement a map using an unsorted list
  - put$O(1)$
  - get $O(n)$
  - remove $O(n)$

```
public class UnsortedTableMap<K,V> extends AbstractMap<K,V> {
    /** Underlying storage for the map of entries. */
    private ArrayList<MapEntry<K,V>> table = new ArrayList<>( );

    /** Constructs an initially empty map. */
    public UnsortedTableMap( ) { }
    // private utility
    /** Returns the index of an entry with equal key, or -1 if none found. */
    private int findIndex(K key) {
        int n = table.size( );
        for (int j=0; j < n; j++)
            if (table.get(j).getKey( ).equals(key))
                return j;
        return -1; // special value denotes that key was not found
    }

    /** Returns the number of entries in the map. */
    public int size( ) { return table.size( ); }

    /** Returns the value associated with the specified key (or else null). */
    public V get(K key) {
        int j = findIndex(key);
        if (j == -1) return null; // not found
            return table.get(j).getValue( );
    }
    /** Associates given value with given key, replacing a previous value (if an
y). */
    public V put(K key, V value) {
        int j = findIndex(key);
        if (j == -1) {
            table.add(new MapEntry<>(key, value)); // add new entry
            return null;
        } else // key already exists
            return table.get(j).setValue(value); // replaced value is returned
    }
```

```
    /** Removes the entry with the specified key (if any) and returns its value. */
    public V remove(K key) {
        int j = findIndex(key);
        int n = size( );
        if (j == -1)
            return null; // not found
        V answer = table.get(j).getValue( );
        if (j != n - 1)
            table.set(j, table.get(n-1)); // relocate last entry to 'hole' created by removal
        table.remove(n-1); // remove last entry of table
        return answer;
    }// Support for public entrySet method...
    private class EntryIterator implements Iterator<Entry<K,V>> {
        private int j=0;
        public boolean hasNext( ) { return j < table.size( ); }
        public Entry<K,V> next( ) {
            if (j == table.size( ))
                throw new NoSuchElementException( );
            return table.get(j++);
        }
        public void remove( ) { throw new UnsupportedOperationException( ); }
    }
    private class EntryIterable implements Iterable<Entry<K,V>> {
        public Iterator<Entry<K,V>> iterator( ) { return new EntryIterator( ); }
    }
    /** Returns an iterable collection of all key-value entries of the map. */
    public Iterable<Entry<K,V>> entrySet( ) { return new EntryIterable( ); }
}
```

10.1.4 Example of Using Map

```
/** A program that counts words in a document, printing the most frequent. */
public class WordCount {
    public static void main(String[ ] args) {
        Map<String,Integer> freq = new ChainHashMap<>( ); // or any concrete map
        // scan input for words, using all nonletters as delimiters
        Scanner doc = new Scanner(System.in).useDelimiter("[^a-zA-Z]+");
        while (doc.hasNext( )) {
            String word = doc.next( ).toLowerCase( ); // convert next word to lowercase
            I9 nteger count = freq.get(word); // get the previous count for this word
            if (count == null)
                count = 0; // if not in map, previous count is zero
            freq.put(word, 1 + count); // (re)assign new count for this word
        }
        int maxCount = 0;
        String maxWord = "no word";
```

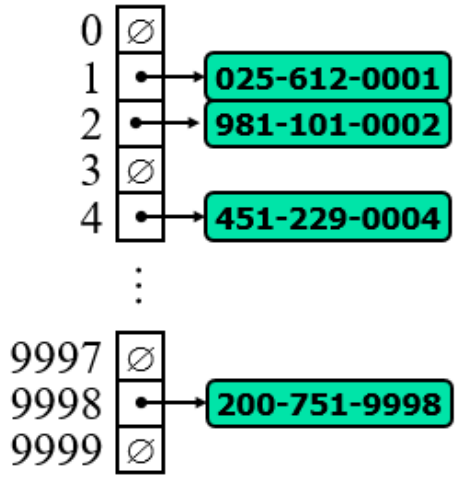
```
            for (Entry<String,Integer> ent : freq.entrySet( )) // find max-count word
                if (ent.getValue( ) > maxCount) {
                    maxWord = ent.getKey( );
                    maxCount = ent.getValue( );
                }
            System.out.print("The most frequent word is '" + maxWord);
            System.out.println("' with " + maxCount + " occurrences.");
        }
    }
```

10.2 Hash Tables

10.2.1 Definition and Property

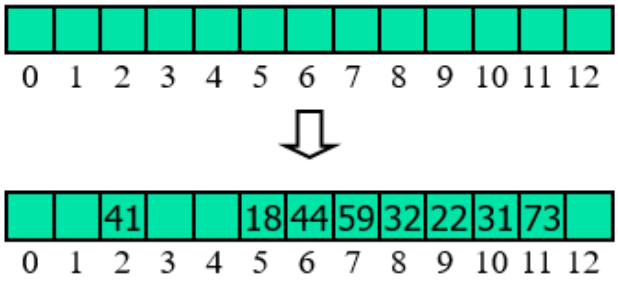
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
    When implementing a map with a hash table, the goal is to store item (k, o) at index i = h(k)
- Hash function: 是一个函数$h(x)$将给定的keys转换成$[0, N-1]$范围内的一个整数
  - Example: $h(x) = x mod N$ is a hash function for integer keys, the integer $h(x)$ is called the hash value of key x
  - Hash function常由两部分组成h：
    - h_1 keys $\rightarrow$ integers
      - Memory Address
        - 所有java object默认的hash code类型
        - 除了数值类型和string类型外都很不错
      - Integer Cast
        - 截取integer固定长度的一部分
        - 适用于byte，short，int和float
      - Component sum
        - 将key拆成固定长度的许多小部分， 再将他们讲起来
        - 特别适用于转换long和double类型
      - Polynomial accumulation
        - 先将key拆分成固定长度的成分，再利用$p(z) = a_0 + a_1 z + a_2 z^2 + \cdots a_{n-1} z^{n-1}$，z为一个固定常量，忽略溢出的情况
        - 多项式可以在$O(n)$复杂度下完成计算
          - $p_0(z) = a_{n-1}$
          - $p_i(z) = a_{n-i-1} + z p_{i-1}$ $(i = 1, 2, \cdots, n)$

        - 迭代计算，这样zhi'x
      - 特别适用于将String转换为integer(选择z=33)
    - Example：用大小为10000的array来存储(SSN, Name)的键值对， SSN为10位整数，$h(x)$ =x的最后四位数



  - h_2 integers $\rightarrow [0, N-1]$
    - Division:$h_2(y) = y \ mod \ N$, N一般取素数
    - Multiply， Add and Divide (MAD)
      - $h_2(y) = (ay + b) \ mod \ N$, a和b是非负整数，否则就无法取余了

- 冲突解决
  - collision: 当使用的function进行转化时， 常会出现多个值对应同一个key
  - 解决方法：
    - Linear Probing: 插入如果当前节点已经被占， 则插入后面的节点， 依次后移
      - Example：
        - $h(x) = x \bmod 13$
        - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order




```
Algorithm get(k)
```

```
{  i = h(k);
   p = 0;
   repeat
     { c = A[i];
        if ( c = null )
           return null;
        else if ( c.key () = k )
           return c.element();
        else
          { i = (i + 1) mod N; //不断移动位置，直到找到应该对应这个位置的(k,n)
            p = p + 1; //记录移动的次数 }
     }
until  ( p = N ); //遍历整个Table，没有找到则返回null
   return null;
 }
```

- remove(k)：先找k对应的(k,v)，如果找到对应的(k,v)则用AVAILABLE标签来替代他们，并返回v，否则返回null
- put(k, v):
  - table满时，throw exception
  - 从cell$h(k)$开始， 逐个检测cell直到发现一个空cell，或存储了AVAILABLE的cell， 将这个(k,v)存储在那里，否则throw exception

- Separate Chaining: 让这个cell指向一个linked list. 这中方法的缺陷是需要额外的空间

```
Algorithm get(k)
    Output: The value associated with the key k in the map, or null if there is no entry with key equal to k in the map
  {
    return A[h(k)].get(k);  // delegate the get to the list-based map at A[h(k)]
  }

Algorithm put(k,v)
    Output: If there is an existing entry in our map with key equal to k, then we return its value (replacing it with v); otherwise, we return null
  {
      t = A[h(k)].put(k,v);   // delegate the put to the list-based map at A[h(k)]
      if ( t = null )           // k is a new key
       n = n + 1;
      return t ;
  }

  Algorithm remove(k)
    Output: The (removed) value associated with key k in the map, or nu
```

```
    ll if there is no entry with key equal to k in the map
18.   {
19.      /* delegate the remove to the list-based map at A[h(k)] */
20.      t = A[h(k)].remove(k);
21.      if ( t ≠ null )      // k was found
22.       n = n - 1;
23.      return t ;
24.    }
```

- Double Hashing: $(i + j * d(x))\ mod\ N\ \ j = 0, 1, \cdots, N-1$
  - 这里的d(k)不能为0
    - 常用的$d(k) = q - k\ mod\ q$, $q < N$, q是素数
  - table size N是素数
  - Example：
    ![10-4.png-27.6kB][1]
- Performance of hashing
  - worst case for search, insert and remove is $O(n)$, 这种情况只在所有的keys都发生冲突时出现
  - The load factor $\alpha = \frac{n}{N}$会影响hash table的效率，最好大于1/2，接近100%
  - hash tables的应用:
    - 小型数据库
    - 编译器
    - 浏览器缓存

10.2.2 Implement hash table with linear probing and Map

```
1.  /** A hash table with linear probing and the MAD hash function */
2.  public class HashTable implements Map {
3.      protected static class HashEntry implements Entry {
4.          Object key, value;
5.          HashEntry () { /* default constructor */ }
6.          HashEntry(Object k, Object v) { key = k; value = v; }
7.          public Object key() { return key; }
8.          public Object value() { return value; }
9.          // set a new value, returning old
10.         protected Object setValue(Object v) {
11.             Object temp = value;
12.             value = v;
13.             return temp;  // return old value
14.         }
15.     }
16.     /** Nested class for a default equality tester */
17.     protected static class DefaultEqualityTester implements EqualityTester {
18.         /* default constructor */
19.         DefaultEqualityTester() { }
20.         /** Returns whether the two objects are equal.  */
```

```
21.         public boolean isEqualTo(Object a, Object b) { return a.equals(b); }
22.     }
23.
24.     protected static Entry AVAILABLE = new HashEntry(null, null); // empty
    marker
25.     protected int n = 0;     // number of entries in the dictionary
26.     protected int N;         // capacity of the bucket array
27.     protected Entry[] A;         // bucket array
28.     protected EqualityTester T; // the equality tester
29.     protected int scale, shift;   // the shift and scaling factors
30.
31.     /** Creates a hash table with initial capacity 1023. */
32.     public HashTable() {
33.         N = 1023; // default capacity
34.         A = new Entry[N];
35.         T = new DefaultEqualityTester(); // use the default equality tester
36.         java.util.Random rand = new java.util.Random();
37.         scale = rand.nextInt(N-1) + 1;
38.         shift = rand.nextInt(N);
39.     }
40.
41.     /** Creates a hash table with the given capacity and equality tester. */
42.     public HashTable(int bN, EqualityTester tester) {
43.         N = bN;
44.         A = new Entry[N];
45.         T = tester;
46.         java.util.Random rand = new java.util.Random();
47.         scale = rand.nextInt(N-1) + 1;
48.         shift = rand.nextInt(N);
49.     }
50.
51.     /** Determines whether a key is valid. */
52.     protected void checkKey(Object k) {
53.         if (k == null)
54.             throw new InvalidKeyException("Invalid key: null.");
55.     }
56.     /** Hash function applying MAD method to default hash code. */
57.     public int hashValue(Object key) {
58.         return Math.abs(key.hashCode()*scale + shift) % N;
59.     }
60.     /** Returns the number of entries in the hash table. */
61.     public int size() { return n; }
62.     /** Returns whether or not the table is empty. */
63.     public boolean isEmpty() { return (n == 0); }
64.
65.     /** Helper search method - returns index of found key or -index-1,
66.     * where index is the index of an empty or available slot. */
67.     protected int findEntry(Object key) throws InvalidKeyException {
68.         int avail = 0;
69.         checkKey(key);
```

```
70.         int i = hashValue(key);
71.         int j = i;
72.         do {
73.             if (A[i] == null)
74.                 return -i - 1;  // entry is not found
75.             if (A[i] == AVAILABLE) {    // bucket is deactivated
76.             avail = i;      // remember that this slot is available
77.             i = (i + 1) % N;    // keep looking
78.             }else if (T.isEqualTo(key,A[i].key())){  // we have found our entry
79.                 return i;
80.             }else{ // this slot is occupied--we must keep looking
81.                 i = (i + 1) % N;
82.             }
83.         } while (i != j);
84.         return -avail - 1;  // entry is not found
85.     }
86.
87.     /** Returns the value associated with a key. */
88.     public Object get (Object key) throws InvalidKeyException {
89.         int i = findEntry(key);  // helper method for finding a key
90.         if (i < 0)
91.             return null;  // there is no value for this key
92.         return A[i].value();     // return the found value in this case
93.     }
94.
95.     /** Put a key-value pair in the map, replacing previous one if it exists. */
96.     public Object put (Object key, Object value) throws InvalidKeyException {
97.         if (n >= N/2)
98.             rehash(); // rehash to keep the load factor <= 0.5
99.         int i = findEntry(key); //find the appropriate spot for this entry
100.        if (i < 0) {
101.            // this key does not already have a value
102.            A[-i-1] = new HashEntry(key, value); // convert to the proper index
103.            n++;
104.            return null;    // there was no previous value
105.        }else       // this key has a previous value
106.            return ((HashEntry) A[i]).setValue(value); // set new value & return
    old
107.    }
108.    /** Doubles the size of the hash table and rehashes all the entries. */
109.    protected void rehash() {
110.        N = 2*N;
111.        Entry[] B = A;
112.        A = new Entry[N]; // allocate a new version of A twice as big as before
113.        java.util.Random rand = new java.util.Random();
114.        scale = rand.nextInt(N-1) + 1;      // new hash scaling factor
115.        shift = rand.nextInt(N);        // new hash shifting factor
116.        for (int i=0; i&ltB.length; i++)
117.            // if we have a valid entry
118.            if ((B[i] != null) && (B[i] != AVAILABLE)) {
```

```
                int j = findEntry(B[i].key());  // find the appropriate spot
                A[-j-1] = B[i];                 // copy into the new array
            }
    }
    /** Removes the key-value pair with a specified key. */
    public Object remove (Object key) throws InvalidKeyException {
        int i = findEntry(key);     // find this key first
        if (i < 0)
            return null;    // nothing to remove
        Object toReturn = A[i].value();
        A[i] = AVAILABLE;       // mark this slot as deactivated
        n--;
        return toReturn;
    }
    /** Returns an iterator of keys. */
    public java.util.Iterator keys() {
        List keys = new NodeList();
        for (int i=0; i&ltN; i++)
            if ((A[i] != null) && (A[i] != AVAILABLE))
                keys.insertLast(A[i].key());
        return keys.elements();
    }
} // ... values() is similar to keys() and is omitted here ...
```

10.3 Sorted Maps

- Implement Sorted Map

```
public class SortedTableMap<K,V> extends AbstractSortedMap<K,V> {
    private ArrayList<MapEntry<K,V>> table = new ArrayList<>( );
    public SortedTableMap( ) { super( ); }
    public SortedTableMap(Comparator<K> comp) { super(comp); }
    /** Returns the smallest index for range table[low..high] inclusive storing
an entry with a key greater than or equal to k (or else index high+1, by convent
ion). */
    private int findIndex(K key, int low, int high) {
        if (high < low) return high + 1; // no entry qualifies
        int mid = (low + high) / 2;
        int comp = compare(key, table.get(mid));
        if (comp == 0)
            return mid; // found exact match
        else if (comp < 0)
            return findIndex(key, low, mid - 1); // answer is left of mid (or
possibly mid)
        else
            return findIndex(key, mid + 1, high); // answer is right of mid
    }
    /** Version of findIndex that searches the entire table */
    private int findIndex(K key) { return findIndex(key, 0, table.size( ) - 1); }
```

```
19.     /** Returns the number of entries in the map. */
20.     public int size( ) { return table.size( ); }
21.     /** Returns the value associated with the specified key (or else null). */
22.     public V get(K key) {
23.         int j = findIndex(key);
24.         if (j == size( ) || compare(key, table.get(j)) != 0)
25.             return null; // no match
26.         return table.get(j).getValue( );
27.     }
28.     /** Associates the given value with the given key, returning any overridden
    value.*/
29.     public V put(K key, V value) {
30.         int j = findIndex(key);
31.         if (j < size( ) && compare(key, table.get(j)) == 0) // match exists
32.             return table.get(j).setValue(value);
33.         table.add(j, new MapEntry<K,V>(key,value)); // otherwise new
34.         return null;
35.     }
36.     /** Removes the entry having key k (if any) and returns its associated value
    . */
37.     public V remove(K key) {
38.         int j = findIndex(key);
39.         if (j == size( ) || compare(key, table.get(j)) != 0)
40.             return null; // no match
41.         return table.remove(j).getValue( );
42.     }
43.     /** Utility returns the entry at index j, or else null if j is out of bounds
    . */
44.     private Entry<K,V> safeEntry(int j) {
45.         if (j < 0 || j >= table.size( ))
46.             return null;
47.         return table.get(j);
48.     }
49.     /** Returns the entry having the least key (or null if map is empty). */
50.     public Entry<K,V> firstEntry( ) { return safeEntry(0); }
51.     /** Returns the entry having the greatest key (or null if map is empty). */
52.     public Entry<K,V> lastEntry( ) { return safeEntry(table.size( )-1); }
53.     /** Returns the entry with least key greater than or equal to given key (if
    any). */
54.     public Entry<K,V> ceilingEntry(K key) { return safeEntry(findIndex(key)); }
55.     /** Returns the entry with greatest key less than or equal to given key (if
    any). */
56.     public Entry<K,V> floorEntry(K key) {
57.         int j = findIndex(key);
58.         if (j == size( ) || ! key.equals(table.get(j).getKey( )))
59.             j--; // look one earlier (unless we had found a perfect match)
60.         return safeEntry(j);
61.     }
62.     /** Returns the entry with greatest key strictly less than given key (if any
    ). */
```

```
63.     public Entry<K,V> lowerEntry(K key) {
64.         return safeEntry(findIndex(key) - 1); // go strictly before the ceiling
    entry
65.     }
66.     /** Returns the entry with least key strictly greater than given key (if any
    ). */
67.     public Entry<K,V> higherEntry(K key) {
68.         int j = findIndex(key);
69.         if (j < size( ) && key.equals(table.get(j).getKey( )))
70.             j++; // go past exact match
71.         return safeEntry(j);
72.     }
73.     // support for snapshot iterators for entrySet() and subMap() follow
74.     private Iterable<Entry<K,V>> snapshot(int startIndex, K stop) {
75.         ArrayList<Entry<K,V>> buffer = new ArrayList<>( );
76.         int j = startIndex;
77.         while (j < table.size( ) && (stop == null || compare(stop, table.get(j))
    > 0))
78.             buffer.add(table.get(j++));
79.         return buffer;
80.     }
81.     public Iterable<Entry<K,V>> entrySet( ) { return snapshot(0, null); }
82.     public Iterable<Entry<K,V>> subMap(K fromKey, K toKey) {
83.             return snapshot(findIndex(fromKey), toKey);
84.     }
85. }
```

Method	Running Time
size	$O(1)$
get	$O(\log n)$
put	$O(n)$; $O(\log n)$ if map has entry with given key
remove	$O(n)$
firstEntry, lastEntry	$O(1)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s+\log n)$ where s items are reported
entrySet, keySet, values	$O(n)$

- Example of using sorted map:
  - Flight Databases
  - trade-off problem

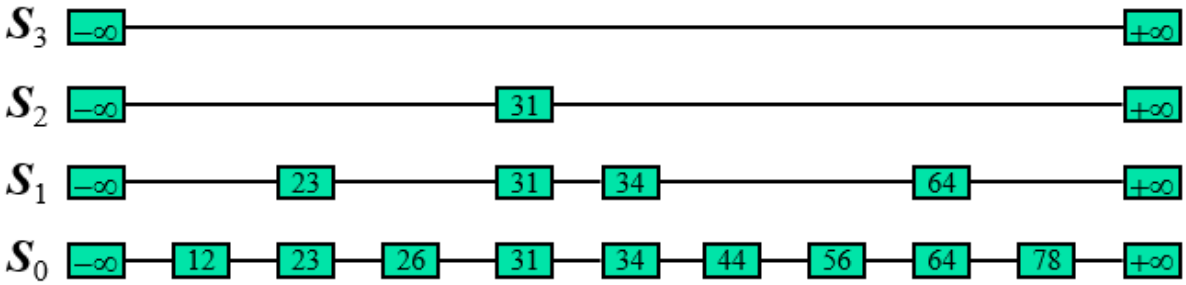
10.4 Skip List

Skip List的意义在于查找在Linked List中的效率很低， 但链表不能随机访问，因此不能实现二分查找，所有就有了Skip List通过牺牲一定的存储空间， 来实现更快的查找

形象化地说，SkipList就是额外保存了二分查找的中间信息。不过SkipList中含有随机化，生成的结构不会像上面那样完美

10.4.1 Defination and Property

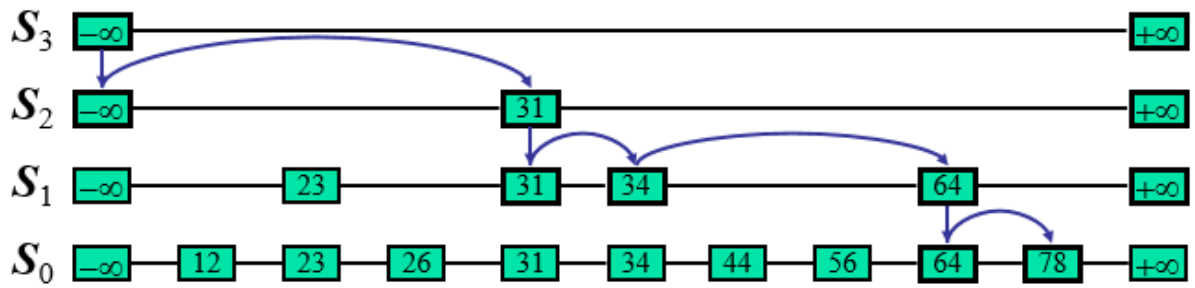
- 一个由互不相同的(key, element)元素组成的集合的Skip list是一系列lists $S_0, S_1, \cdots, S_h$满足:
  - 每个list S_i 包含特殊的键$-\infty$和$+\infty$
  - List S_0 包含原始序列的所有节点，并按非降序排列
  - $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_h$
  - List S_h 只包含特殊的键$-\infty$和$+\infty$



- 我们使用随机函数来选择每一层的节点，在每个节点进行一次随机选择，随着level的上升，节点被选中的概率下降

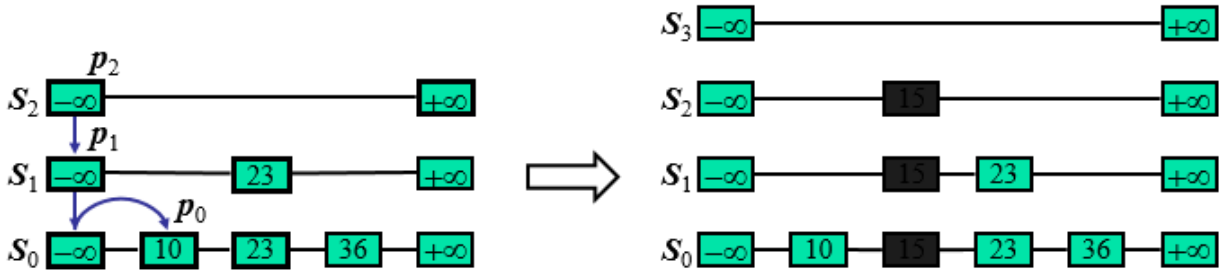
10.4.2 Function of Skip List

- Search
  - 我们在如下所示的skip list中搜索key值为x的键值对:
    - 从最顶层的最左端开始搜索
    - 再节点p， 我们比较x和$y = key(next(p))$
      - x = y: 返回element(next(p))
      - x > y: 向前
      - x < y: 下降
    - 当需要从S_0下降时，返回null
  - Example: search for 78

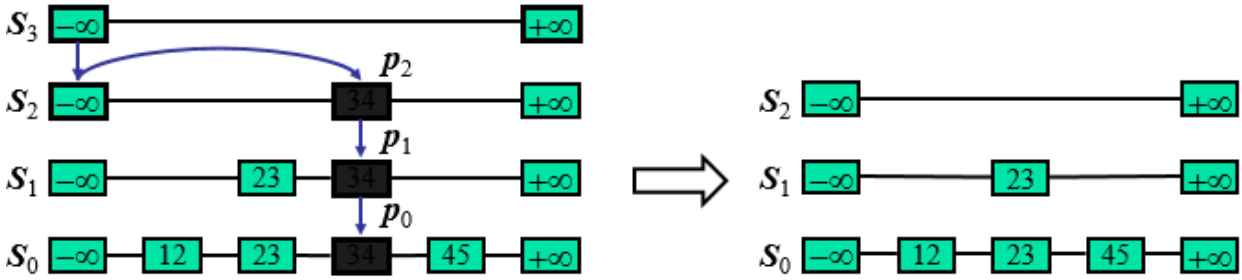


- Insertion

- 将(k,v)插入skip list, 我们需要用到随机函数
  - 我们重复的生成0，1随机值，直到得到0，记录下在次之前得到1的次数i
  - 如果i比目前的h大，则再skip list中添加层数$S_{h+1}, \cdots, S_{i+1}$, 每一个都包含连个特殊的值
  - 在新的i个Skip list中找到比k小的最大key值， 在它后面插入(k,v)
  - Example: insert key 15, with i = 2

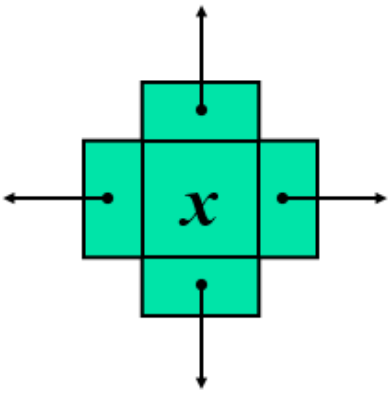


- Deletion
  - 从所有包含(k,v)的skip list中移除(k,v)
  - 最后保留一条只包含$-\infty$和$+\infty$的list



10.4.3 Implementation

- 用四向node来实现, 同时还需定义两个特殊节点 $-\infty$和$+\infty$



- 空间复杂度最差为$O(n)$, 搜索和更新等操作的时间复杂度为 $O(log(n))$

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

10.5 Sets, Multisets and Multimaps

10.5.1 defination and Property

- 我们用一个不重复的序列来表示一个set
- 集合可以进行的操作有
  - union
  - intersection
  - subtraction
  - 这些操作的复杂度最大为$O(n_A + n_B)$
- The Java Collections Framework 包含如下set的实现， 和map的数据结构相似:
  - java.util.HashSet
    - provides an implementation of the (unordered) set ADT with a hash table.
  - java.util.concurrent.ConcurrentSkipListSet
    - provides an implementation of the sorted set ADT using a skip list.
  - java.util.TreeSet
    - provides an implementation of the sorted set ADT using a balanced search tree.
- Set具有与Collection完全一样的接口，因此没有任何额外的功能。实际上Set就是Collection,只 是行为不同。(这是继承与多态思想的典型应用：表现不同的行为。)Set不保存重复的元素(至于如何判断元素相同则较为负责)
- Set : 存入Set的每个元素都必须是唯一的，因为Set不保存重复元素。加入Set的元素必须定义equals()方法以确保对象的唯一性。Set与Collection有完全一样的接口。Set接口不保证维护元素的次序。
  - HashSet：为快速查找设计的Set。存入HashSet的对象必须定义hashCode()。
  - TreeSet： 保存次序的Set, 底层为树结构。使用它可以从Set中提取有序的序列。
  - LinkedHashSet：具有HashSet的查询速度，且内部使用链表维护元素的顺序(插入的次序)。于是在使用迭代器遍历Set时，结果会按元素插入的次序显示。

10.5.2 List based set

Each set is stored in a sequence represented with a linked-list

- Union： 通常将较小的set中的元素移到大的中，复杂度为$O(nlog(n))$

10.5.3 Tree based set

- Each element is stored in a node, which contains a pointer to a set name
- Each set is a tree, rooted at a node with a self-referencing set pointer
- Union

11. Search Tree

11.1 Binary Search Tree

- searching

```
Algorithm TreeSearch(p, k):
    if p is external then
        return p {unsuccessful search}
    else if k == key(p) then
        return p {successful search}
    else if k < key(p) then
        return TreeSearch(left(p), k) {recur on left subtree}
    else {we know that k > key(p)}
    return TreeSearch(right(p), k) {recur on right subtree}
```

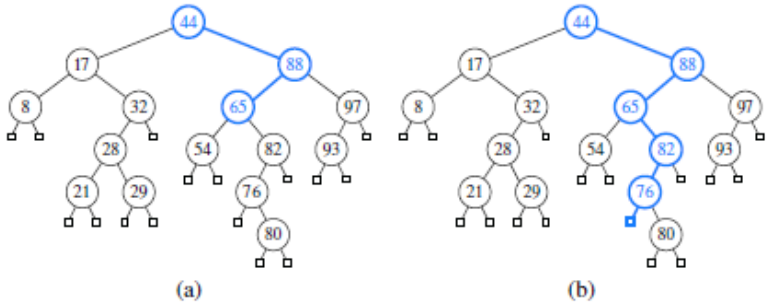


Figure 11.2: (a) A successful search for key 65 in a binary search tree; (b) an unsuccessful search for key 68 that terminates at the leaf to the left of the key 76.

- Insertions

```
Algorithm TreeInsert(k, v):
    Input: A search key k to be associated with value v
        p = TreeSearch(root( ), k)
        if k == key(p) then
            Change p's value to (v)
        else
            expandExternal(p, (k,v))
```

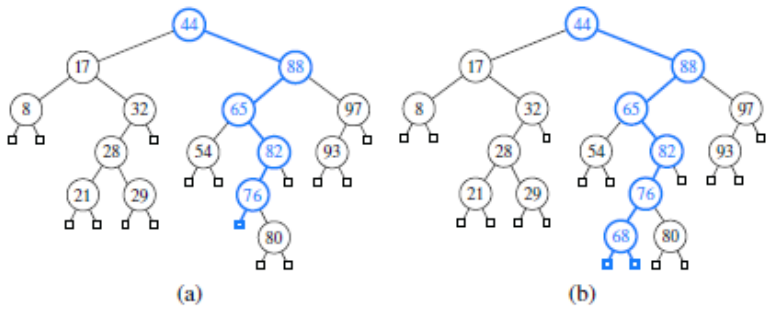


Figure 11.4: Insertion of an entry with key 68 into the search tree of Figure 11.2. Finding the position to insert is shown in (a), and the resulting tree is shown in (b).

- Deletions
  - 进行删除操作时
    - 如果被删除的节点没有右子节点，有左子节点，则将左侧的子节点上移；
    - 如果左右子节点都没有，则直接删除
    - 如果左右子节点都存在，则将左子节点的右子节点删除后，填补到该位置，同时对左子节点的右子节点进行同样的删除操作

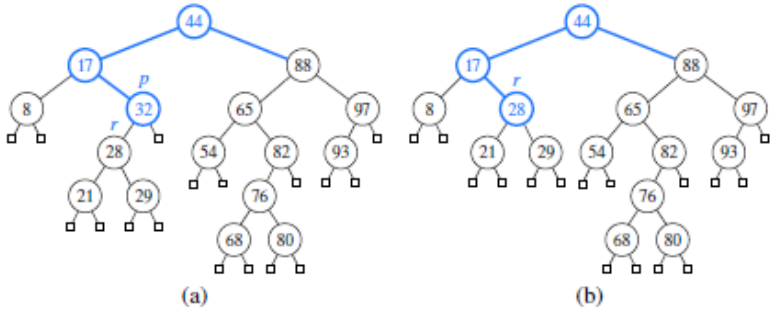


Figure 11.5: Deletion from the binary search tree of Figure 11.4b, where the entry to delete (with key 32) is stored at a position p with one child r: (a) before the deletion; (b) after the deletion.

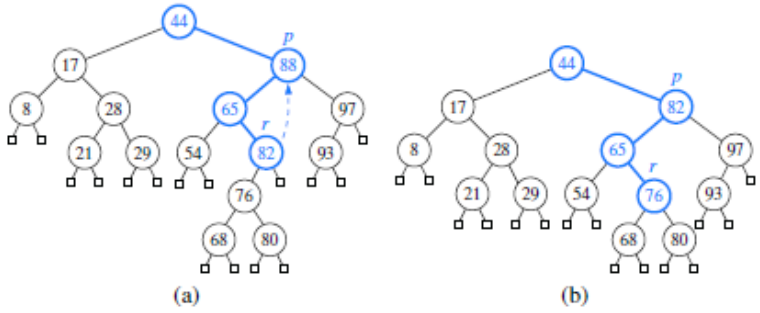


Figure 11.6: Deletion from the binary search tree of Figure 11.5b, where the entry to delete (with key 88) is stored at a position p with two children, and replaced by its predecessor r: (a) before the deletion; (b) after the deletion.

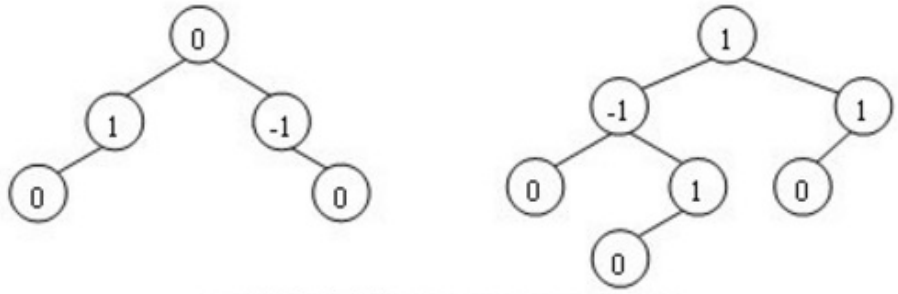
Binary search Tree相应的算法复杂度

Method	Running Time
size, isEmpty	$O(1)$
get, put, remove	$O(h)$
firstEntry, lastEntry	$O(h)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(h)$
subMap	$O(s+h)$
entrySet, keySet, values	$O(n)$

11.2 AVL Tree

11.2.1 Balanced Search Tree

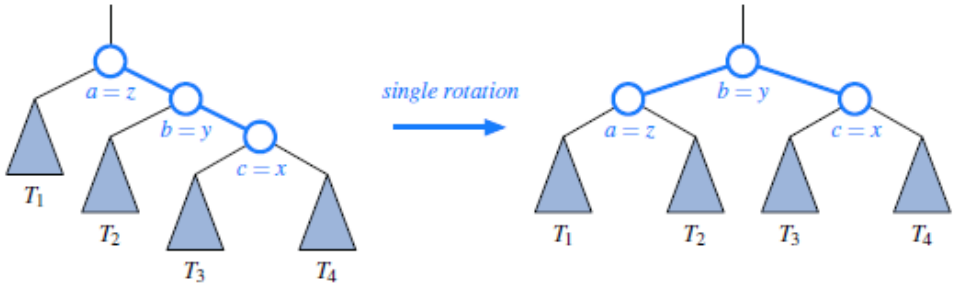
- 为了解决树出现极端偏移造成的低效数据结构， 我们引入了树的旋转， 当两个子节点的深度相差在1以上时， 就对树进行旋转。



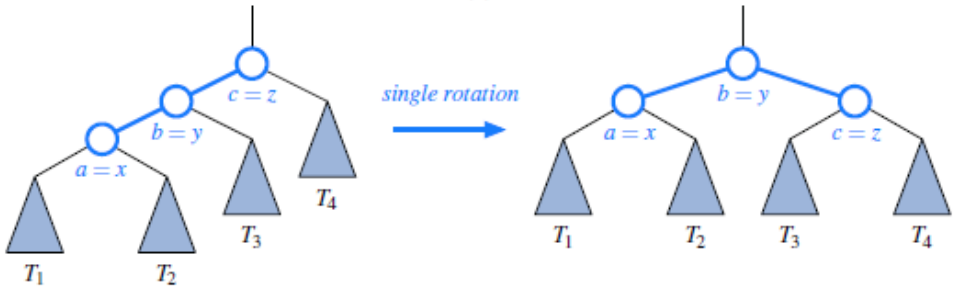
(a) 平衡二叉树 cgbluesky.blog.163.com

(b) 非平衡二叉树，灰色结点为不平衡因子

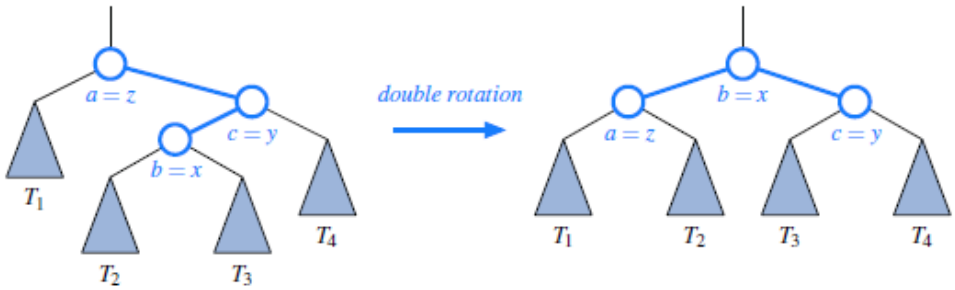
- Restructuring(旋转)的四种情况



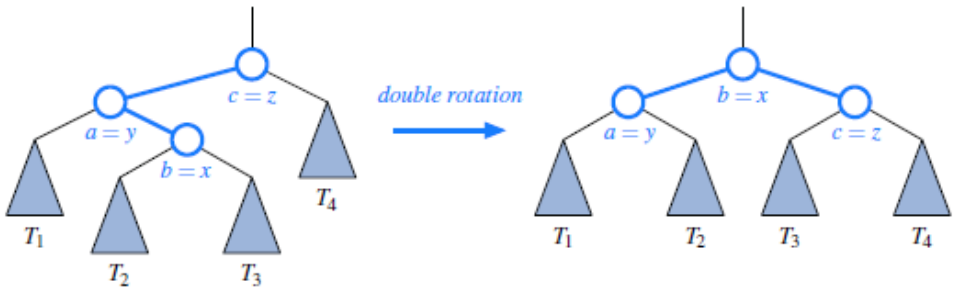
(a)



(b)



(c)



(d)

- a) z < y < x, 因此y成了父节点
- b) z > y > x, 因此y成了父节点
- c) z < x < y, 因此x成了父节点
- d) z > x > y, 因此x成了父节点

- Implement balance tree

```
/** A specialized version of LinkedBinaryTree with support for balancing. */
protected static class BalanceableBinaryTree<K,V> extends LinkedBinaryTree<Entry
<K,V>> {
    //-------------- nested BSTNode class --------------
    // this extends the inherited LinkedBinaryTree.Node class
```

```
5.      protected static class BSTNode<E> extends Node<E> {
6.          int aux=0;
7.          BSTNode(E e, Node<E> parent, Node<E> leftChild, Node<E> rightChild) {
8.          super(e, parent, leftChild, rightChild);
9.          }
10.         public int getAux( ) { return aux; }
11.         public void setAux(int value) { aux = value; }
12.     }
13.     //--------- end of nested BSTNode class ---------
14.
15.     // positional-based methods related to aux field
16.     public int getAux(Position<Entry<K,V>> p) {
17.         return ((BSTNode<Entry<K,V>>) p).getAux( );
18.     }
19.     public void setAux(Position<Entry<K,V>> p, int value) {
20.         ((BSTNode<Entry<K,V>>) p).setAux(value);
21.     }
22.     // Override node factory function to produce a BSTNode (rather than a Node)
23.     protected Node<Entry<K,V>> createNode(Entry<K,V> e, Node<Entry<K,V>> parent,
    Node<Entry<K,V>> left, Node<Entry<K,V>> right) {
24.         return new BSTNode<>(e, parent, left, right);
25.     }
26.
27.     /** Relinks a parent node with its oriented child node. */
28.     private void relink (Node<Entry<K,V>> parent, Node<Entry<K,V>> child,
    boolean makeLeftChild) {
29.         child.setParent(parent);
30.         if (makeLeftChild)
31.             parent.setLeft(child);
32.         else
33.             parent.setRight(child);
34.     }
35.     /** Rotates Position p above its parent. */
36.     public void rotate(Position<Entry<K,V>> p) {
37.         Node<Entry<K,V>> x = validate(p);
38.         Node<Entry<K,V>> y = x.getParent( ); // we assume this exists
39.         Node<Entry<K,V>> z = y.getParent( ); // grandparent (possibly null)
40.         if (z == null) {
41.             root = x; // x becomes root of the tree
42.             x.setParent(null);
43.         } else
44.             relink(z, x, y == z.getLeft( )); // x becomes direct child of z
45.             // now rotate x and y, including transfer of middle subtree
46.             if (x == y.getLeft( )) {
47.                 relink(y, x.getRight( ), true); // x's right child becomes y's le
    ft
48.                 relink(x, y, false); // y becomes x's right child
49.             } else {
50.                 relink(y, x.getLeft( ), false); // x's left child becomes y's rig
    ht
```

```
                relink(x, y, true); // y becomes left child of x
            }
        }
        /** Performs a trinode restructuring of Position x with its
    parent/grandparent. */
        public Position<Entry<K,V>> restructure(Position<Entry<K,V>> x) {
            Position<Entry<K,V>> y = parent(x);
            Position<Entry<K,V>> z = parent(y);
            if ((x == right(y)) == (y == right(z))) { // matching alignments
                rotate(y); // single rotation (of y)
                return y; // y is new subtree root
            } else { // opposite alignments
                rotate(x); // double rotation (of x)
                rotate(x);
                return x; // x is new subtree root
            }
        }
    }
```

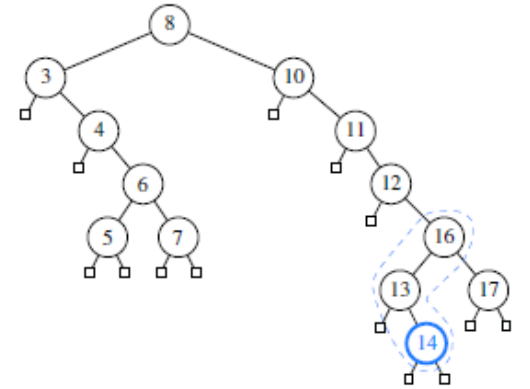
11.2.2 AVL Tree

- The height of an AVL tree storing n entries is O(logn).
- a single restructure is O(1)
  - using a linked-structure binary tree
- find is O(log n)
  - height of tree is O(log n), no restructures needed
- insert is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- remove is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

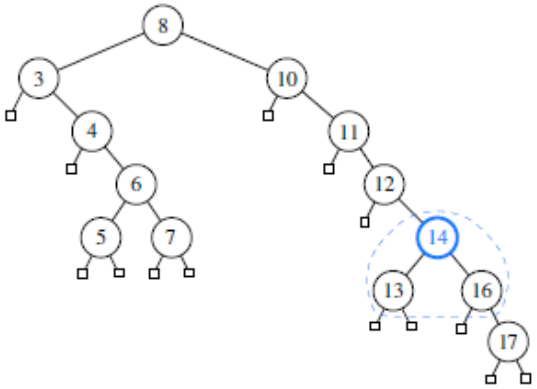
11.3 Splay Trees

- 什么是伸展树（Splay Tree）
  - 假设想要对一个二叉搜索树执行一系列的查找操作。为了使整个查找时间更小，被查频率高的那些条目就应当经常处于靠近树根的位置。于是想到设计一个简单方法，在每次查找之后对树进行重构，把被查找的条目搬移到离树根近一些的地方。splay tree应运而生。splay tree是一种自调整形式的二叉搜索树，它会沿着从某个节点到树根之间的路径，通过一系列的旋转把这个节点搬移到树根去。

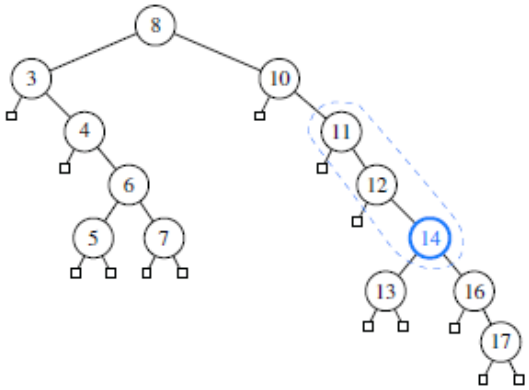
11.3.1 Restructuring


8
3
10
4
11
6
12
5
7
16
13
17
14

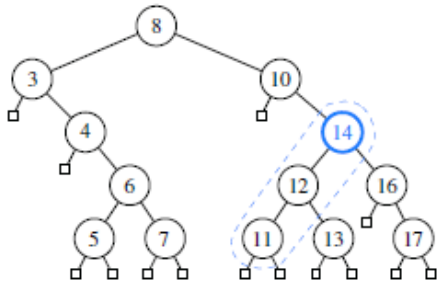
(a)


8
3
10
4
11
6
12
5
7
14
13
16
17

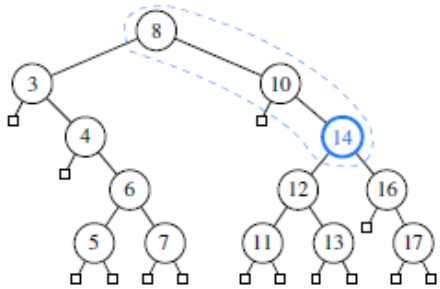
(b)


8
3
10
4
11
6
12
5
7
14
13
16
17

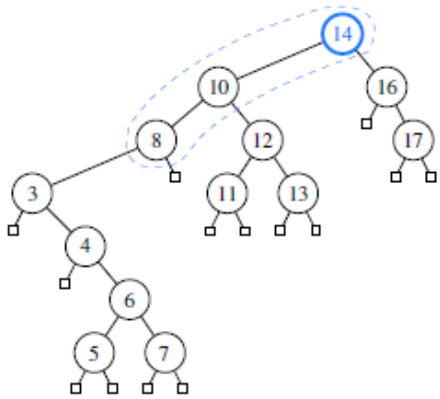
(c)



(d)



(e)



(f)

- 旋转的规则：
  - 当节点a为b的左子节点, 则旋转后b为a的右子节点， a原本的右子节点成为了b的左子节点
  - 当节点a为b的右子节点, 则旋转后b为a的左子节点， a原本的左子节点成为了b的右子节点
  - 重复选择操作直到a称为了根节点
- 旋转的时机：
  - 当搜索了a以后，就对a进行旋转
  - 当插入节点a后，对a进行旋转
  - 当删除节点a后，对a的父节点进行旋转

11.3.2 Implement Splay Tree

```java
/** An implementation of a sorted map using a splay tree. */
public class SplayTreeMap<K,V> extends TreeMap<K,V> {
    /** Constructs an empty map using the natural ordering of keys. */
    public SplayTreeMap( ) { super( ); }
    /** Constructs an empty map using the given comparator to order keys. */
    public SplayTreeMap(Comparator<K> comp) { super(comp); }
    /** Utility used to rebalance after a map operation. */
    private void splay(Position<Entry<K,V>> p) {
        while (!isRoot(p)) {
            Position<Entry<K,V>> parent = parent(p);
            Position<Entry<K,V>> grand = parent(parent);
            if (grand == null) // zig case
                rotate(p);
            else if ((parent == left(grand)) == (p == left(parent))) { // zig-zig case
                rotate(parent); // move PARENT upward
                rotate(p); // then move p upward
            } else { // zig-zag case
                rotate(p); // move p upward
                rotate(p); // move p upward again
            }
        }
    }
    // override the various TreeMap rebalancing hooks to perform the appropriate splay
    protected void rebalanceAccess(Position<Entry<K,V>> p) {
        if (isExternal(p))
            p = parent(p);
        if (p != null)
            splay(p);
    }
    protected void rebalanceInsert(Position<Entry<K,V>> p) {
        splay(p);
    }
    protected void rebalanceDelete(Position<Entry<K,V>> p) {
        if (!isRoot(p)) splay(parent(p));
    }
}
```

- all these operations are $O(log(n))$

11.4 (2,4) Tree

(2,4) Tree又叫2-3-4 Tree

11.4.1 Definition and Property

- (2,4) Tree的属性：

- 一个Node最多有4个节点
- 一个Node中最多有3个值或键值对

11.4.2 Insertion

参考资料

- 复杂度为$O(\log n)$

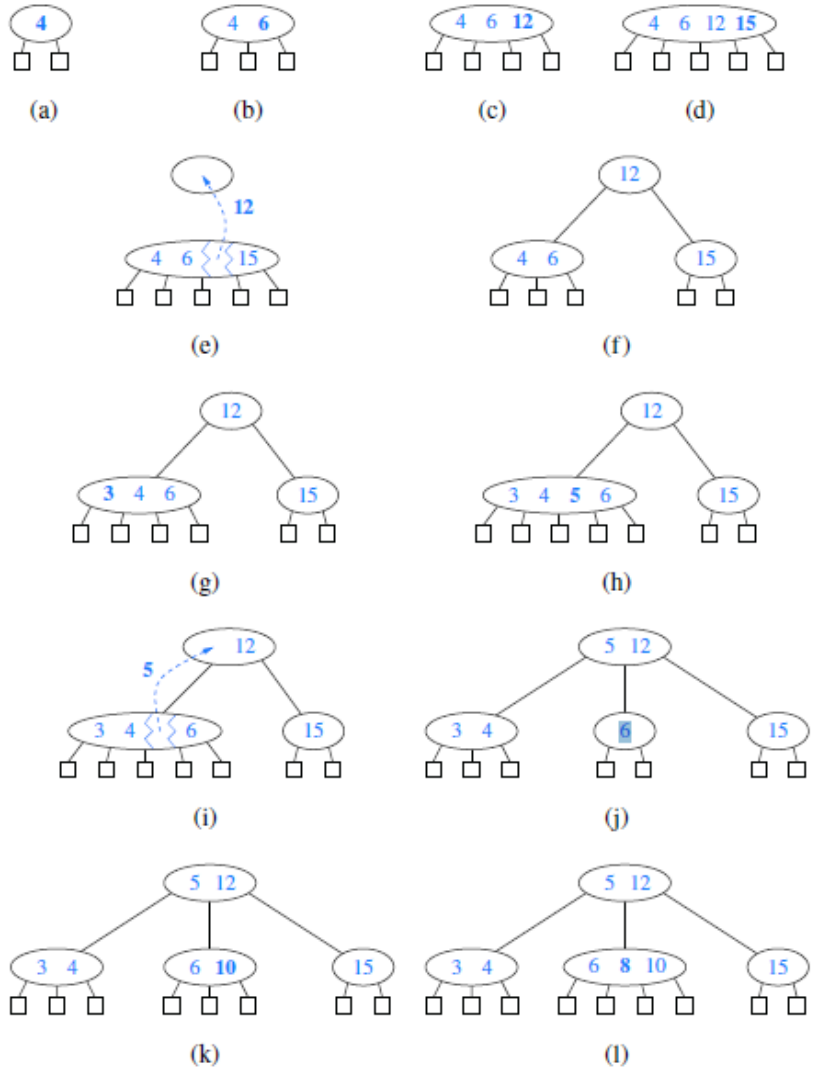


Figure 11.26: A sequence of insertions into a $(2,4)$ tree: (a) initial tree with one entry; (b) insertion of 6; (c) insertion of 12; (d) insertion of 15, which causes an overflow; (e) split, which causes the creation of a new root node; (f) after the split; (g) insertion of 3; (h) insertion of 5, which causes an overflow; (i) split; (j) after the split; (k) insertion of 10; (l) insertion of 8.

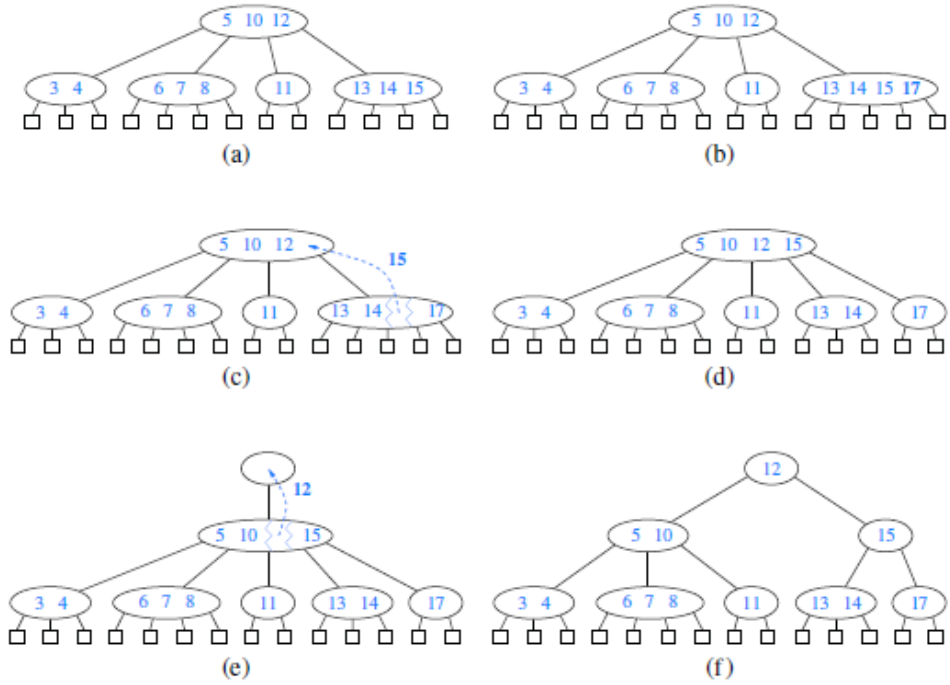


Figure 11.25: An insertion in a $(2,4)$ tree that causes a cascading split: (a) before the insertion; (b) insertion of 17, causing an overflow; (c) a split; (d) after the split a new overflow occurs; (e) another split, creating a new root node; (f) final tree.

- Insert共有两种一种是top-down, 一种是bottom-up, 区别在于
  - top-down将要前往的Node 不管是不是它最终停留的Node只要这个Node已经存储的entry达到了三个则讲中间值pop上去，将最大和最小值spilt成两个新的节点
  - Bottom-up则是先到达最终的Node，如果这个Node存储这个新的insert进来的值后，包含4个entry，则对齐进行pop和spilt处理
  - top-down在效率和存储上都比bottom-up要好一点

11.4.2 Deletion

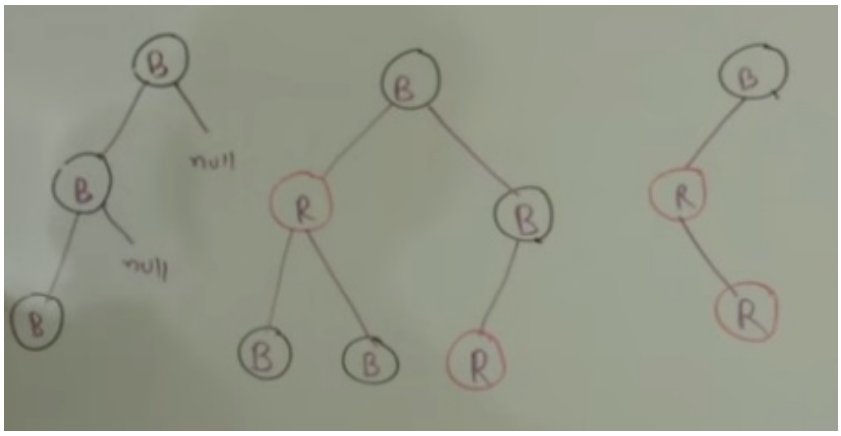
- Find key you want to remove and replace it with next higher key
- If Sibling have greater than one key, steal an adjacent key, make that the Parent and bring down the current parent
- If no adjacent sibling has greater than one key, steal key from parent
- If parent is the root and contain only one key, sibling has only one key fuse it into a key node and make it the new root.

11.5 Red Black Trees

参考资料

- 红黑树的三条规则

- Root节点永远是黑的
- 不能有red-red的parent-child节点
- 从根节点出发到所有child node小于2的节点的黑节点个数相同(也可以看作到所有null的黑节点个数相同)
- Example

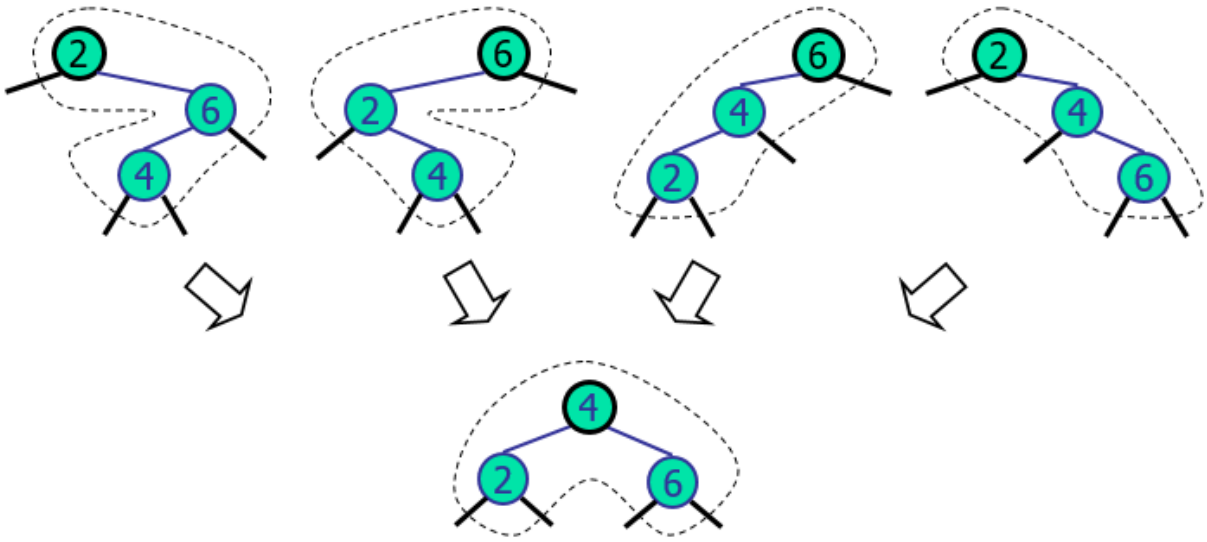


  - 第一个，不是违反了第三条规则，黑节点的个数不相同
  - 第二个，是红黑树
  - 第三个不是红黑树，违反了规则2，存在red-red

- 红黑树最长的枝干的深度不会超过最短枝干的两倍
  - 红黑树的insert，find，delete的complexity都是$O(\log n)$

11.5.1 insertion

- 如果是空树， 则创建一个黑色的root节点
- 否则插入红色的叶子节点
  - 如果parent是黑的，则insert完成
  - 如果parent是红的
    - 如果parent的兄弟节点也是红的
      - 将parent和他的兄弟节点改成黑，将parent的parent改成红(如果parent-parent是root则不改)，如果新树的parent是黑的则done，否则继续按情况向上修改节点颜色，知道满足红黑树的三条规则为止
    - 如果parent的兄弟节点是null或者黑，则对其进行rotation, 重复操作直到最后树满足红黑树的三条规则




```
if ( empty )
    Black Node
else{
    create red leaf node
    if (parent is black)
        done
    else{
        if (parent sibling is red)
            recolor & recheck
        else{
            // rotate
            // LL是从R-parent的父节点到R-child的路径， rotate R享有旋转一次对应上图中的三，R-child的位置最终向右上移动了一位
            if( LL ) rotate R
            // 类似上述
            if( LR ) rotate R,L
            if( RL ) rotate L,R
            if( RR ) rotate L
        }
    }
}
```

11.5.3 Deletion

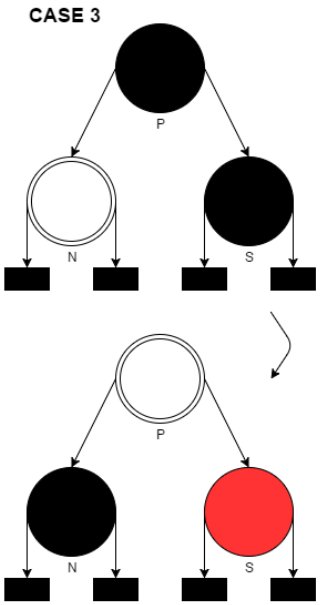
- 找到要删除的节点，将其删除
  - 如果这个节点有两个非null子节点，找到右子树中的最小节点(叫做inorder successor)，（即右子树的最左节点，他一定只有0-1个子节点），用它来代替这个被删除的节点，并将这个节点删除。把他转换成节点只有0-1个子节点的情况进行处理
  - 如果这个node只有0-1个子节点
    - 如果这个node是红色，则直接删除
    - 如果这个node是黑色，他有1个红色子节点，则将子节点改为黑色后上移

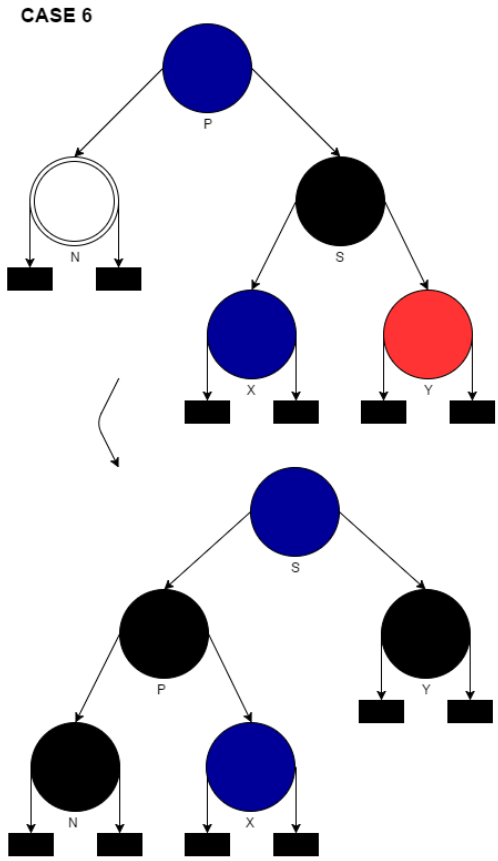
- 如果这个node是黑色的，他没有红色的子节点，则分以下六种情况进行讨论

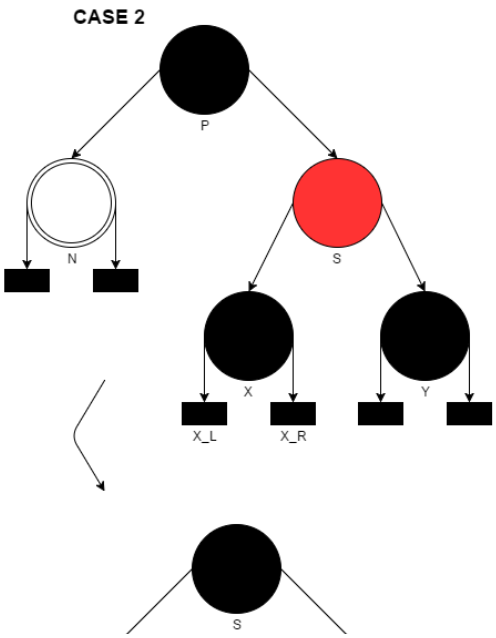
Double **Black** Node: 这个node再count path中的black node时候少了1

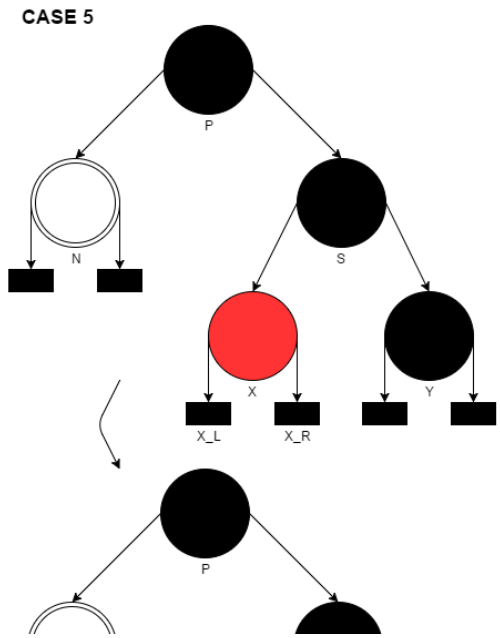
Rectangle: subtree 或 null node, 但是满足他是黑色的

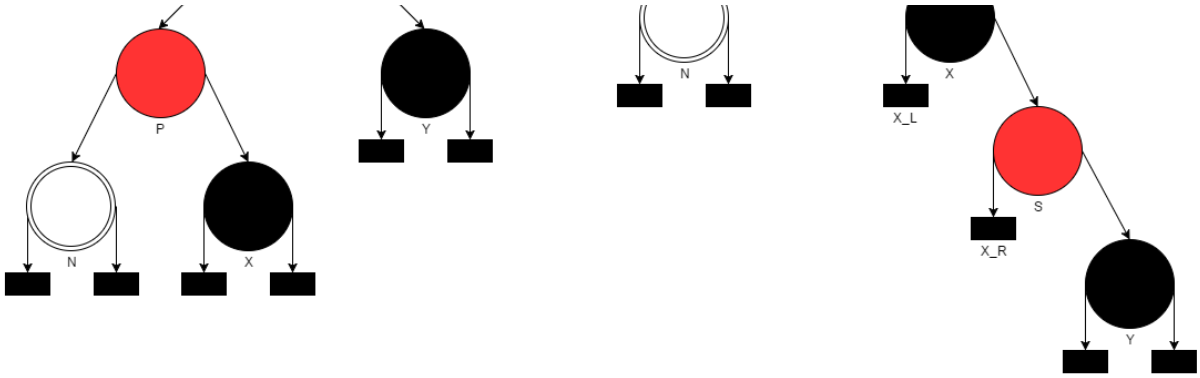
Purple node: 它的颜色可以是红也可以是黑, rotate之后这些node颜色跟之前一样

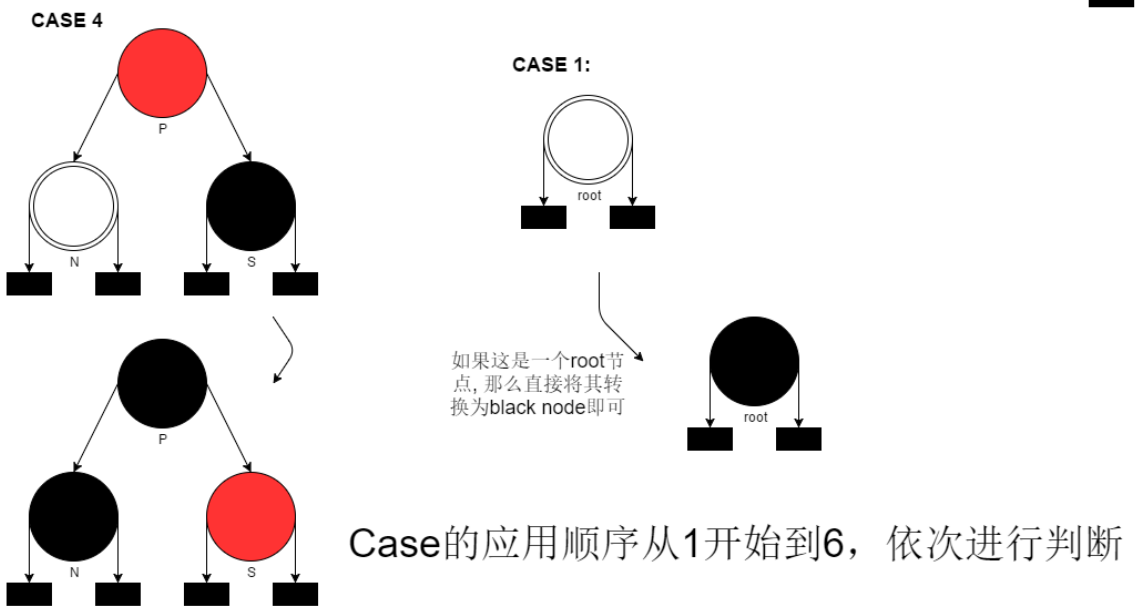












Case的应用顺序从1开始到6，依次进行判断

- 情况1，4，6是终结情形，2，3，5是中间情形

12. Sort

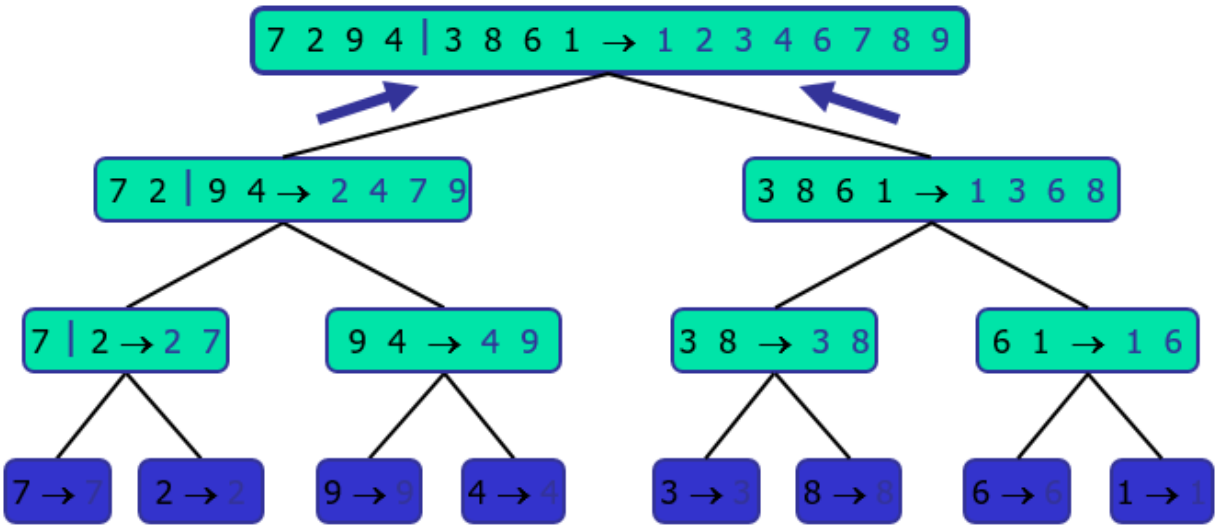
Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	fast in-place for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	fast sequential data access for huge data sets (> 1M)

12.1 Merge Sort

- 分治法是一种常用的算法paradigm(模型):
  - Divide: 将输入数据S分成两个互不相交的子集S1 and S2(通常分到$size$为0或1为止)
  - Recur: 解决S1和S2对应的子问题
  - Conquer: 将S1和S2解决的子问题合并起来，最终得到S
- Merge-Sort是一种基于分治法的算法
  - 与Heap-Sort的相似之处
    - 用到了comparator
    - 时间复杂度为 $n \log n$
  - 于Heap-Sort的差别在于
    - 没有用到priority Queue
    - 连续的获取数据，适用于对disk上的data进行排序

```
Algorithm mergeSort(S, C)
    Input sequence S with    n  elements, comparator C
    Output sequence S sort  according to C
  {
      if ( S.size() > 1 )
        {
           (S1, S2) = partition(S, n/2);
           mergeSort(S1, C);
           mergeSort(S2, C);
           S = merge(S1, S2);
         }
```

```
    }

Algorithm merge(A, B)
     Input sequences A and B with n/2 elements each
     Output sorted sequence of A ^ B
    {
              S = empty sequence;
       while ( not A.isEmpty()  and not B.isEmpty() )
            if   ( A.first().element() < B.first().element() )
                S.insertLast(A.remove(A.first()));
            else
                S.insertLast(B.remove(B.first()));
       while  ( not A.isEmpty()
            S.insertLast(A.remove(A.first()));
       while  ( not B.isEmpty() )
            S.insertLast(B.remove(B.first()));
       return S;
    }
```



- Implement Merge-Sort Without recursive

```
public static void mergeSort(Object[] orig, Comparator c) {
    Object[] in = new Object[orig.length]; // make a new temporary array
    System.arraycopy(orig,0,in,0,in.length); // copy the input
    Object[] out = new Object[in.length]; // output array
    Object[] temp; // temp array reference used for swapping
    int n = in.length;
    for (int i=1; i < n; i*=2) { // each iteration sorts all length-2*i runs
        for (int j=0; j < n; j+=2*i)  // each iteration merges two length-i pairs
            merge(in,out,c,j,i); // merge from in to out two length-i runs at j
        temp = in; in = out; out = temp; // swap arrays for next iteration
    }
    // the "in" array contains the sorted array, so re-copy it
    System.arraycopy(in,0,orig,0,in.length);
```

```
}

// merge in[start..start+inc-1] and in[start+inc..start+2*inc-1]
protected static void merge(Object[] in, Object[] out, Comparator c, int start,
int inc) {
    int x = start; // index into run #1
    int end1 = Math.min(start+inc, in.length); // boundary for run #1
    int end2 = Math.min(start+2*inc, in.length); // boundary for run #2
    int y = start+inc; // index into run #2 (could be beyond array boundary)
    int z = start; // index into the out array
    while ((x < end1) && (y < end2)){
        if (c.compare(in[x],in[y]) <= 0)
            out[z++] = in[x++];
        else
            out[z++] = in[y++];
    }
    if (x < end1) // first run didn't finish
        System.arraycopy(in, x, out, z, end1 - x);
    else if (y < end2) // second run didn't finish
        System.arraycopy(in, y, out, z, end2 - y);
}
```

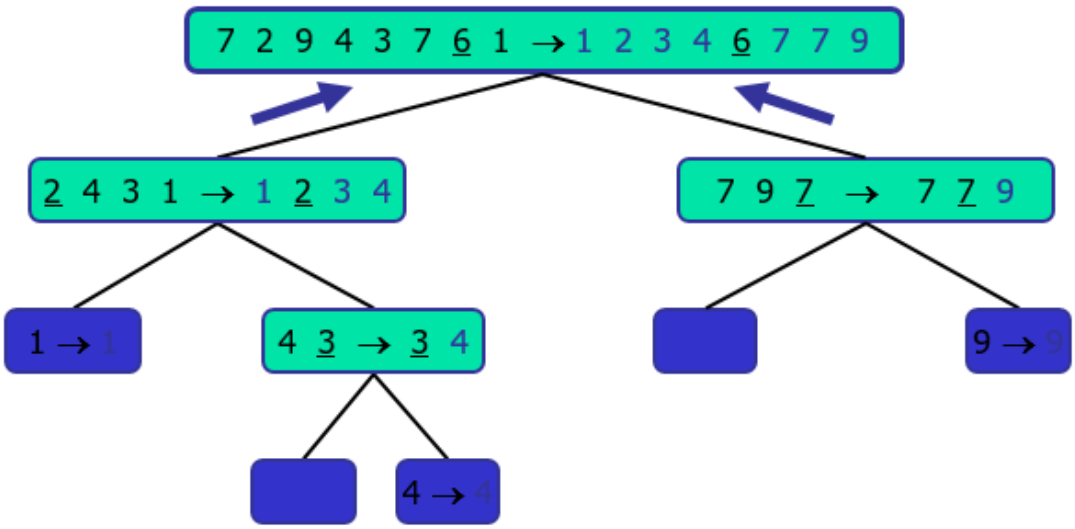
12.2 Quick-Sort

快排原理
快排复杂度

- 快速排序也是一种基于分治法的排序模型：
  - Divide:随机选择一个叫做中心点的元素x，由这个元素x将S分割成L,E,G三段
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: 对L和G进行排序
  - Conquer: 合并L, E和G
- 时间复杂度为 $O(n \log n)$, worst case is $O(n^2)$, 由于这个worst case是由于随机生成的x导致的，所以这里要对其概率进行讨论。

```
// partition的时间复杂度为O(n)
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the elements of S less than, equal to, or gre
ater than the pivot, resp.
  { L, E, G = empty sequences;
      x = S.remove(p);
      while  ( ¬S.isEmpty() )
```

```
          { y = S.remove(S.first());
           if  ( y < x )
                 L.insertLast(y);
             else if  ( y = x )
                 E.insertLast(y);
             else  //  y > x
                 G.insertLast(y);}
        return L, E, G;
    }

Algorithm inPlaceQuickSort(S, l, r)
    Input sequence S, ranks l and r
    Output sequence S with the elements of rank between l and r rearranged in in
creasing order
 {   if  ( l >= r )
          return;
      i = a random integer between l and r;
      x = S.elemAtRank(i);
      p = inPlacePartition(x);
      inPlaceQuickSort(S, l, p-1);
      inPlaceQuickSort(S, p+1, r);
  }
```



- 为了提高效率，避免worst case的出现， 我们选用两个指针从两端往中间进行扫描
- Implement Quick Sort

```
public static void quickSort (Object[] S, Comparator c) {
    if (S.length < 2)
        return; // the array is already sorted in this case
    quickSortStep(S, c, 0, S.length-1); // recursive sort method
}

private static void quickSortStep (Object[] S, Comparator c, int leftBound, int
```

```
rightBound ) {
    if (leftBound >= rightBound) return; // the indices have crossed
    Object temp;  // temp object used for swapping
    Object pivot = S[rightBound];
    int leftIndex = leftBound;     // will scan rightward
    int rightIndex = rightBound-1; // will scan leftward
    while (leftIndex <= rightIndex) { // scan right until larger than the pivot
        while ( (leftIndex <= rightIndex) && (c.compare(S[leftIndex], pivot)<=0)
)
            leftIndex++;
      // scan leftward to find an element smaller than the pivot
        while ( (rightIndex >= leftIndex) && (c.compare(S[rightIndex], pivot)>=0)
)
            rightIndex--;
        if (leftIndex < rightIndex) { // both elements were found
            temp = S[rightIndex];
            S[rightIndex] = S[leftIndex]; // swap these elements
            S[leftIndex] = temp;
        }
    } // the loop continues until the indices cross
    temp = S[rightBound]; // swap pivot with the element at leftIndex
    S[rightBound] = S[leftIndex];
    S[leftIndex] = temp; // the pivot is now at leftIndex, so recurse
    quickSortStep(S, c, leftBound, leftIndex-1);
    quickSortStep(S, c, leftIndex+1, rightBound);
}
```

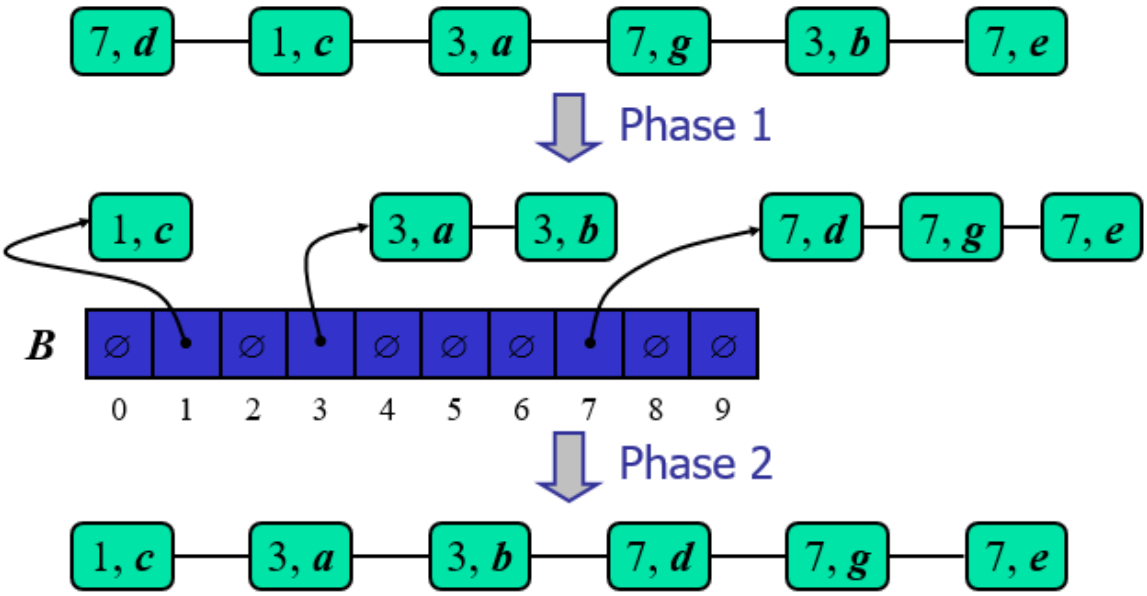
12.3 Bucket-Sort

- Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)
  - Phase 1: Empty sequence S by moving each entry (k, o) into its bucket B[k]
  - Phase 2: For i = 0, …, N - 1, move the entries of bucket B[i] to the end of sequence S
- Analysis:
  - Phase 1 takes O(n) time
  - Phase 2 takes O(n + N) time
  - Bucket-sort takes O(n + N) time

```
Algorithm bucketSort(S, N)
    Input sequence S of (key, element) items with keys in the range [0, N - 1]
    Output sequence S sorted by increasing keys
   { B = array of N empty sequences;
     while ( not S.isEmpty() )
         {  f = S.first();
            (k, o) = S.remove(f);
            B[k].insertLast((k, o)); }
       for ( i = 0; i++; i <=N - 1)
```

```
            while ( not B[i].isEmpty() )
                {     f = B[i].first();
                    (k, o) = B[i].remove(f);
                    S.insertLast((k, o)); }
      }
```

- Key range [0, 9]



12.4 Radix-Sort

12.4.1 Lexicographic-Sort

用于String的排序，先将string每个字母转换成integer，对这个tuple进行排序，顺序从左到右字母升序

```
Algorithm lexicographicSort(S)
    Input sequence S of  d-tuples
    Output sequence S sorted in lexicographic order
  {  for ( i = d; i>=1; i--; )
          stableSort(S, Ci);
   }
```

- Example:
  (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)

12.4.2 Radix-Sort

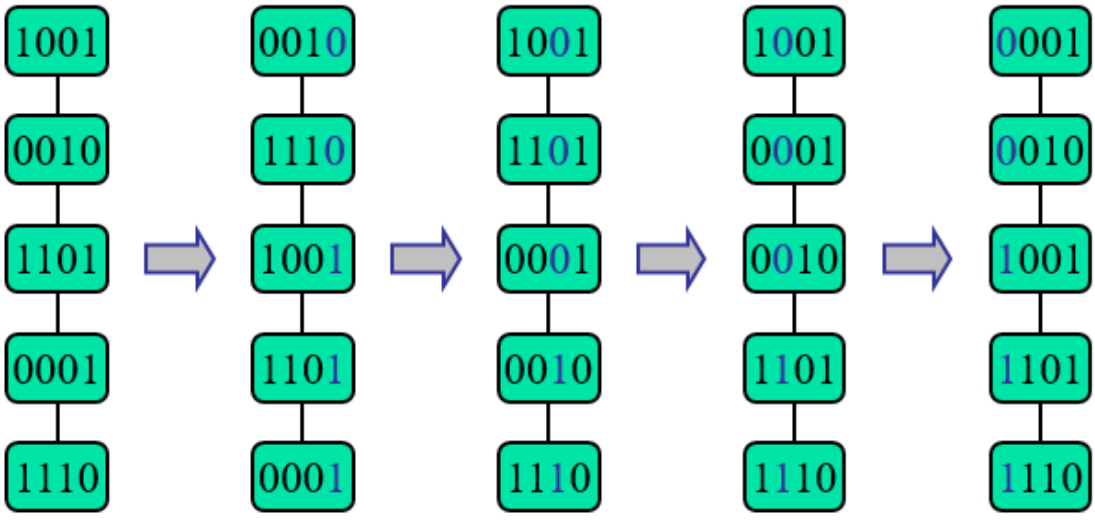
Radix-Sort是特殊的Lexicographic-Sort, 它是基于bucket-sort算法实现的

```
Algorithm radixSort(S, N)
    Input sequence S of  d-tuples such
        that (0, …, 0) ≠ (x1, …, xd) and (x1, …, xd) (N - 1, …, N - 1) for each t
uple (x1, …, xd) in S
    Output sequence S sorted in lexicographic order
  {  for ( i = d; i>=1; i-- )
         bucketSort(S, N);
  }
```

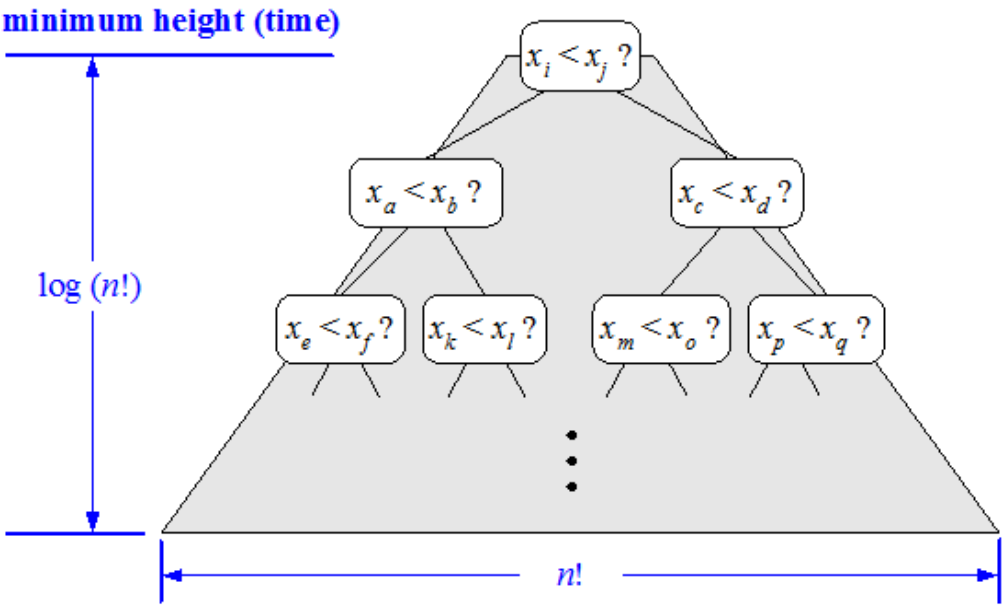
Example:

```
Algorithm binaryRadixSort(S)
    Input sequence S of b-bit integers
    Output sequence S sorted replace each element x of S with the item (0, x)
  { for ( i = 0; i<= b-1; i++ )
        {  replace the key k of each item (k, x) of S            with bit xi of x
;
        bucketSort(S, 2);
         }
   }
```

- Sorting a sequence of 4-bit integers



12.5 Lower bound



- 任何基于比较的排序算法的时间复杂度 $\log(n!)$
- Therefore, any such algorithm takes time at least
  $log(n!) \geq \log(\frac{n}{2})^{\frac{n}{2}} = \frac{n}{2}\log(\frac{n}{2})$
- 因此任何基于比较的排序算法的时间复杂度至少为$O(n \log n)$
  ![12-7.jpg-9.9kB](http://static.zybuluo.com/Detachment/bi6llvsyyz01tmfajhq07ryv/12-7.jpg

13 Selection

13.1 Quick-Selection

- 时间复杂度为$O(n)$
- 思想类似于quick-sort, 不同点在于每次partition后， 我们只需要关注所要查询的值所在的那一部分

```
Algorithm quickSelect(S,k):
    Input: Sequence S of n comparable elements, and an integer k ∈ [1,n]
    Output: The k th smallest element of S
    if n == 1 then
        return the (first) element of S.
    pick a random (pivot) element x of S and divide S into three sequences:
        • L, storing the elements in S less than x
        • E, storing the elements in S equal to x
        • G, storing the elements in S greater than x
    if k ≤ |L| then
        return quickSelect(L,k)
    else if k ≤ |L|+|E| then
        return x //{each element in E is equal to x}
    else
        return quickSelect(G,k-|L|-|E|) //{note the new selection parameter}
```

- 还有一种叫做Deterministic Selection的方法
  - 他的不同之处在于选择x不是任意的而是，取序列的中间那个数

14. Text Processing

14.1 Pattern Matching

14.1.1 Definition

- String是由一些列的characters组成的
  - Example：
    - Java program
    - HTML document
    - DNA sequence
    - Digitized image
- Pattern Matching 就是找到text T中等于pattern P的substring
- 应用：
  - Text editors
  - Search engines
  - Biological reser

14.1.2 Brute-Force Pattern Matching

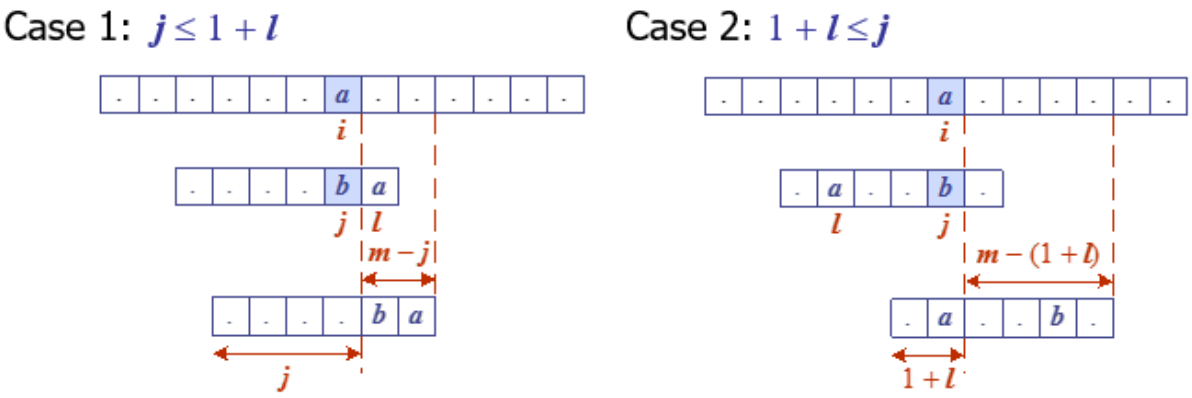
```
Algorithm BruteForceMatch(T, P)
   Input text T of size n and pattern
             P of size m
   Output starting index of a substring of
             T equal to P or -1 if no such
              substring exists
  { for ( i = 0; i < n - m+1; i++ )
        { // test shift i of the pattern
            j = 0;
            while ( j < m ⊣ T[i + j] = P[j] )
                  j = j + 1;
            if   ( j = m )
                  return  i;  // match at i
          }
       return  -1  // no match anywhere
    }
```

- 复杂度为$O(n^2)$

14.1.3 Boyer-Moore Pattern Matching

参考资料

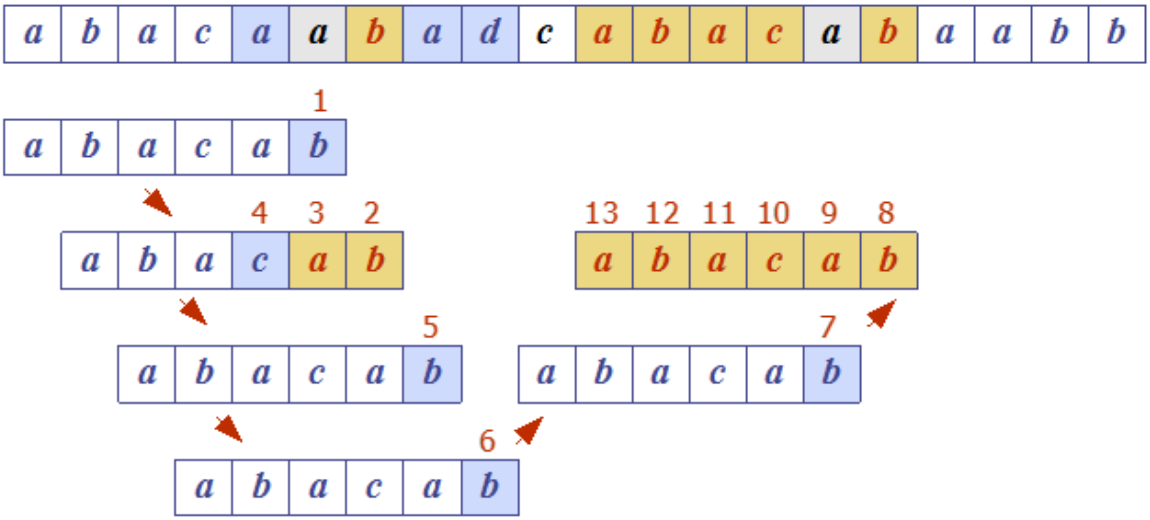
- Boyer-Moore Pattern Matching基于以下两点
  - 从后往前比较P(Pattern)和T(text)
  - 当出现mismatch时T[i]=c
    - 如果P包含c，shift P中最后一个c(P[j] = c)满足P[j] = T[i]
    - 如果不包含，shift 这个pattern P直到(P[0] with T[i+1])



- 这里的case1是当最后一个a的索引大于当前节点索引j时，这时候将pattern shift 1位

```
Algorithm BoyerMooreMatch(T, P, S)
     { L = lastOccurenceFunction(P, S )
      i = m - 1
      j = m - 1
      repeat
    if ( T[i] = P[j] )
           { if  ( j = 0 )
            return  i  // match at i
          else
            { i = i - 1;
              j = j - 1; }
            }
    else  // character-jump
        { l = L[T[i]];
          i = i + m - min(j, 1 + l);
          j = m - 1; }
  until  ( i > n - 1)
  return  -1  //  no match
}
```

- Example:



- 获取character对应最后一个节点的函数返回character的所有，如果没有找到则返回-1
- 算法的时间复杂度是$O(mn)$

14.1.4 The Knuth-Morris-Pratt Algorithm

参考资料

最好看看这个参考资料讲的很详细

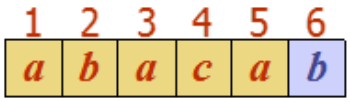
- 时间复杂度为$O(m+n)$
- 基本特点：
  - 从左往右进行匹配
  - 发生不匹配，跳过已经比较过的部分，避免多余的比较
- 基本步骤：
  - 先求Pattern部分的匹配值
  - 发生冲突时根据对应的匹配值，查找新的起点
  - 移动位数 = 当前位置 - 对应的匹配值

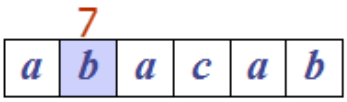
```
Algorithm KMPMatch(T, P)
   {    F = failureFunction(P);
    i = 0;
    j = 0;
    while ( i < n )
         if   ( T[i] = P[j] )
             { if   ( j = m - 1 )
                      return  i - j ; // match
                else
                      { i = i + 1; j = j + 1;}
              }
         else
             if  ( j > 0 )
                   j = F[j - 1];
```

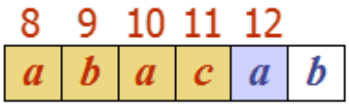
```
                else
                        i = i + 1;
            return  -1;  // no match
        }
```

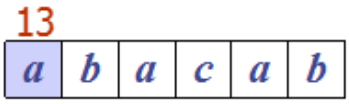
- 利用failure Function求匹配值
  - 前缀： 除了最后一个字符外的全部头部组合
  - 后缀： 除了第一个字符外的全部尾部组合
  - F(j): 前缀后后最共有的最长组合的长度
  - 足以要从第二位才会计算这个即F[1], 即F[0] = 0

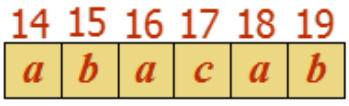
```
j : 0 1 2 3 4 5
P[j]: a b a a b a
F(j): 0 0 1 1 2 3

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

T: abaab[x]
P: abaab[a]
```











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

```
Algorithm failureFunction(P)
    Input:the pattern P with m elements
    Output: F=failureFunction(P)
    {
```

```
        F[0]=0;//前缀[],后缀[], 相当于忽略掉F[0]这种情况

        //多于一个元素：初始化前缀pre后缀sur位置
        int pre=0;
        int sur=1;
        while(sur<m){
            if(P[pre]==P[sur]){//前头==后尾
                F[sur]=pre+1;
                pre++;
                sur++;
            }else if(pre>0){
                pre=F[pre-1];
            }else{//pre==0且头尾元素不等
            sur++;
            }
        }
    }
```

- Failure function算法分析:
  - pre=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,pre++;
- Example : abacabb
  - sur=4,pre=0 → 出现匹配 → pre++,sur++,F[4]=1
  - sur=5,pre=1 → 出现匹配 → pre++,sur++,F[5]=2
  - sur=6,pre=2 → 不匹配, 且pre>0 → pre=F[1]=0 → 头尾不匹配 → F[6]=0,sur++

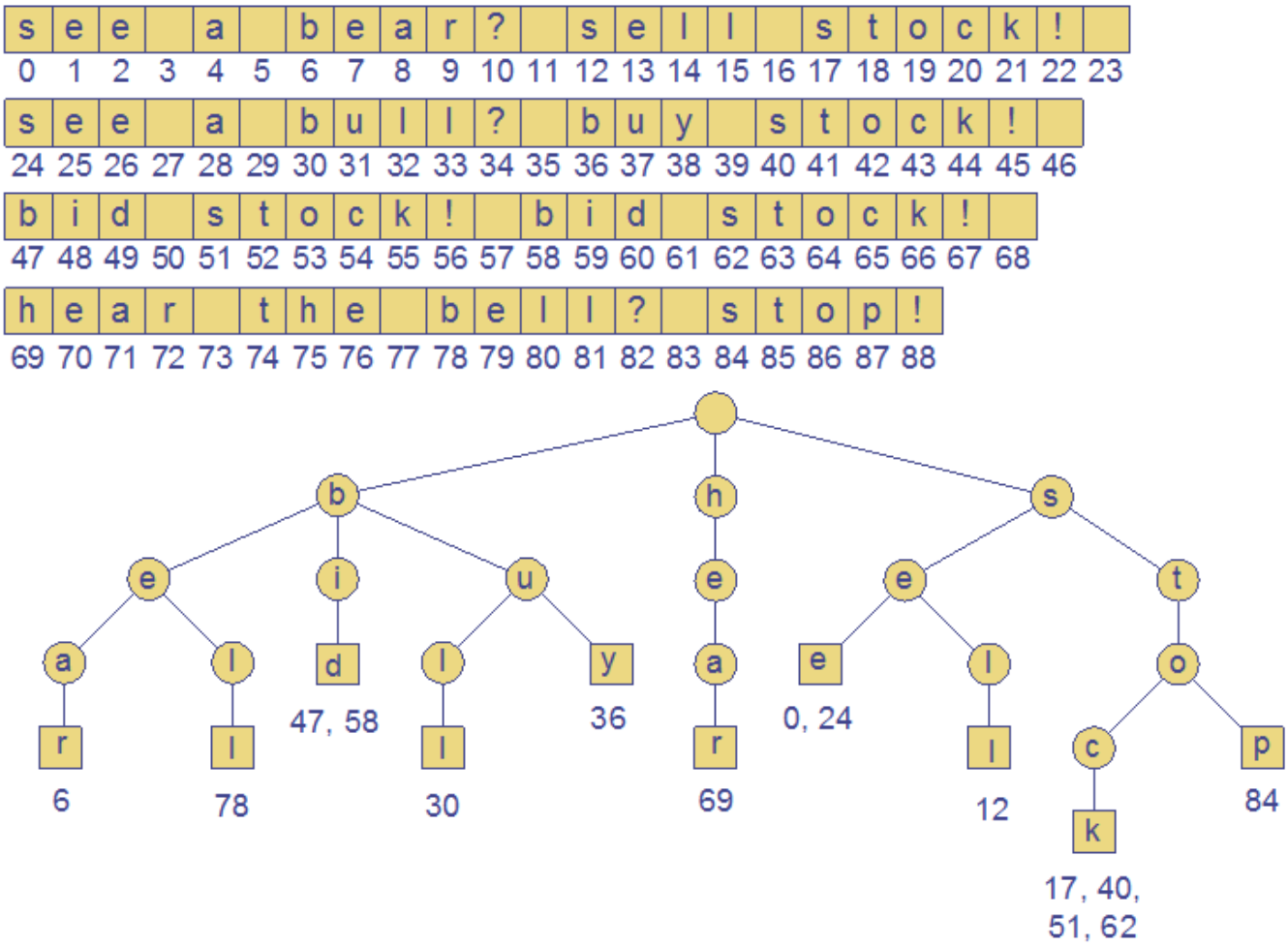
14.2 Tries

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$O(m)$

14.2.1 Standard Tries

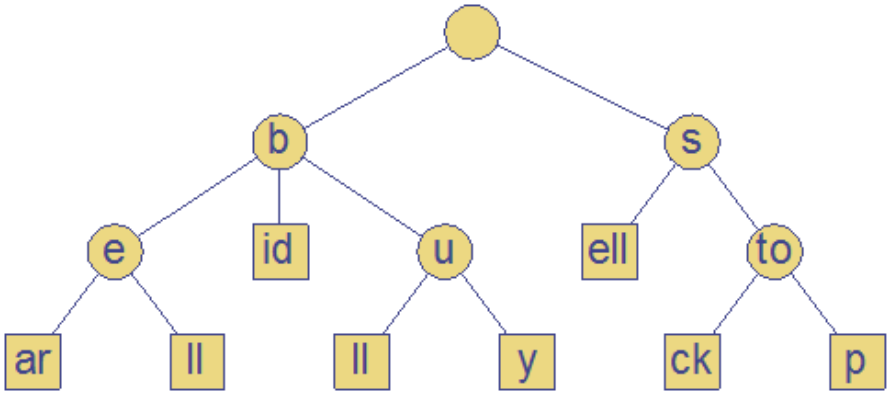
- 基本构造:
  - 除了root外每个node都配有一个字母标签, root代表空格;
  - node的子树按字母表顺序排序, root的子结点为单词首字母;
  - 从root到external的路径构成S中一个字符串
- 对于长度为n，存储了s个字符串的标准字典树T
  - n : 总长度;
  - m : 每次操作动用字符串参数的数量;
  - d : 字母表的size;
  - T的高度为s个字符串中最长的那个的长度;
  - 空间复杂度为$O(n)$, 查找，插入和删除的事件复杂度为$O(dm)$

- 使用字典树进行文本匹配
  - 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首字母的位置;

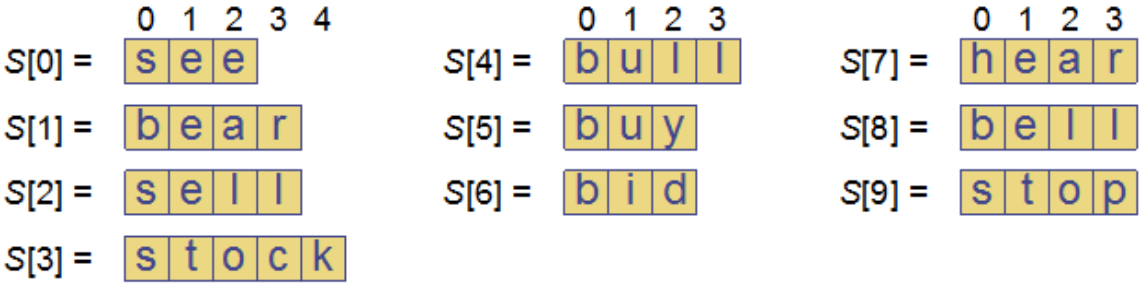


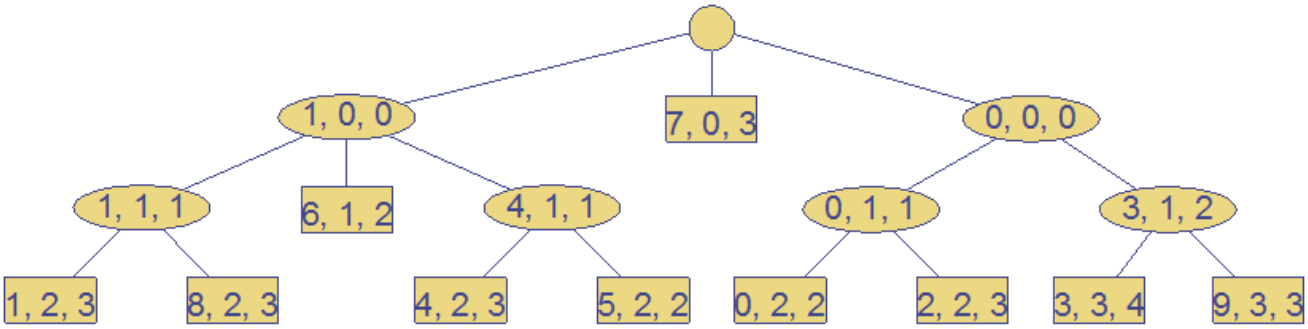
14.2.2 Compressed Tries

- 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
- 空间复杂度为O(s), 其中S为array中的字符串数量, 和标准字典树相比优化了存储空间



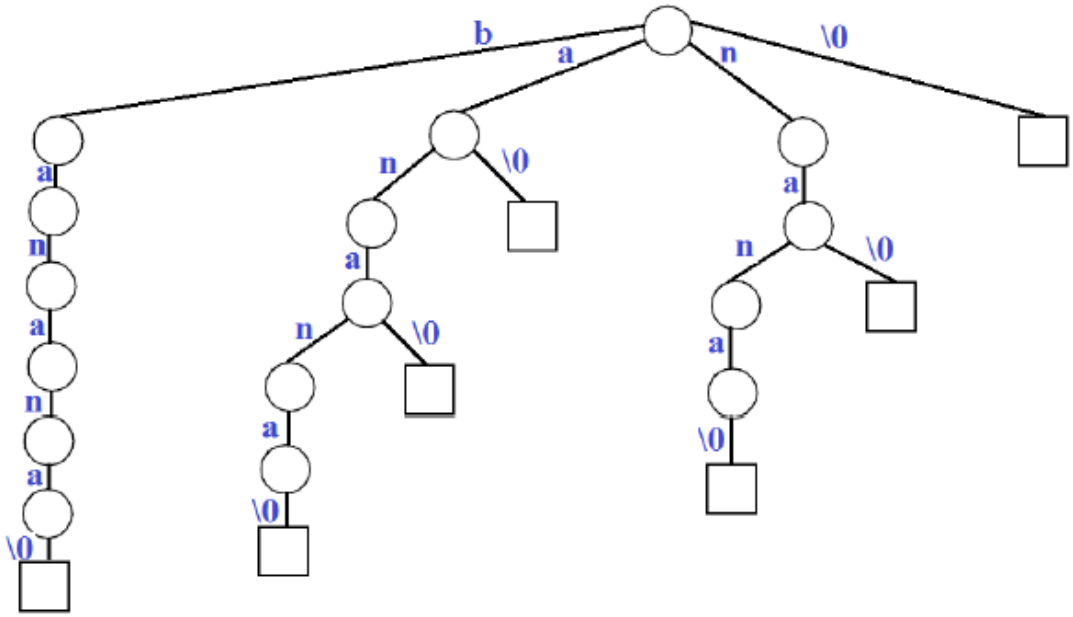
- 压缩的表达
  - 存储了node的位置以及里面字符的range，如7,(0,3)表示hear

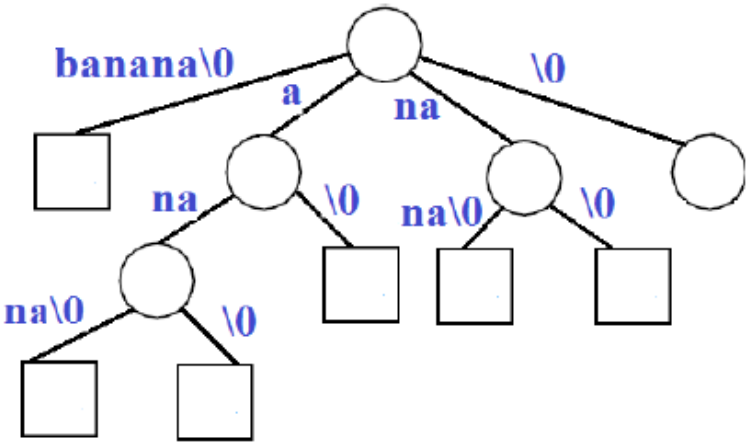




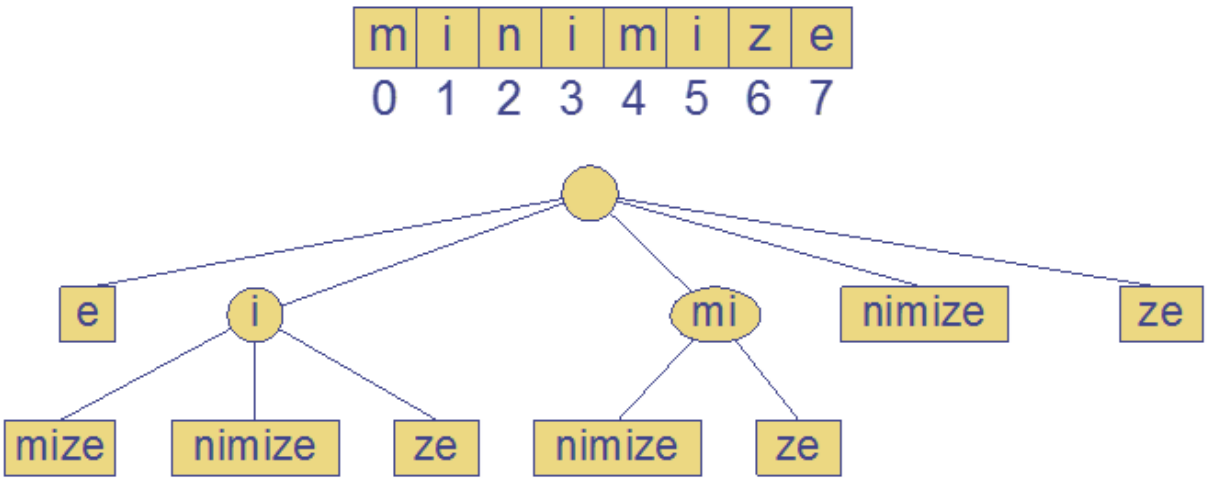
14.2.3 Suffix Trie

- 后缀Trie构建: 以"banana为例"
  - 首先获得当前字符串的全部后缀
  - 将所有后缀根据其公共前缀构建成前缀树: 例如以"b"为前缀的后缀只有"anana",以"a"为前缀的后缀有"na"和"nana"...





- 使用坐标表示后缀, 优化空间复杂度
- 课件中的栗子: "minimize"
  - 后缀: "e","ze","ize","mize","imize","nimize","inimize","minimize"
  - 用后缀构建前缀trie:
    - "inimize","ize","imize"都从"i"开始, 进行压缩
    - "minimize","mize"都从"m-i"开始, 进行压缩
    - 剩下的"e","ze","nimize"都是单链, 进行压缩



- 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$O(n)$
    - 长度为n的串X的后缀总长度为n(n+1)/2
    - 若显式保存空间复杂度是平方级
    - 这里使用隐式保存(坐标), 优化了空间复杂度
  - 字符匹配操作$O(dm) \rightarrow O(m)$
  - 构建字典树$O(n)$, 但相对不容易构建, 如果是显式构建的话和空间复杂度一样都是平方级别
- 和之前几种匹配算法的对比: Trie缺点是需要提前构建

	Preprocess Pattern	Preprocess Text	Space	Search Time
Brute Force			O(1)	O(mn)
Boyer Moore	O(m+d)		O(d)	O(n) *
Suffix Trie		O(n)	O(n)	O(m)

  - n = text size
  - m = pattern size
  - * on average
- 应用:
  - 文本匹配
    - 标准Trie树只适合前缀匹配和全字匹配，并不适合后缀和子串匹配
    - 后缀Trie进行匹配的原理: 若sb在S中，则sb必然是S的某个后缀的前缀
  - 求指定字符串T在字符串S中的重复次数

    - 方案: 用S+'
      '构造后缀树，搜索T节点下的叶节点数目即为重复次数
      * 原理: 如果T在S中重复了两次，则S应有两个后缀以T为前缀，重复次数就自然统计出来了
      * 字符串S中的最长重复子串
      * 原理同前, 找到最深的非叶节点(从root所经历过的字符个数), 最深非叶节点所经历的字符串起来就是最长重复
      子串
      * 两个字符串S1，S2的最长公共部分
      * 将S1/S2作为字符串压入后缀树，找到最深的非叶节点，且该节点的叶节点既有#也有(无#)
- 应用举例: 使用后缀字典树进行文本匹配, 原理见前
    - 从根结点root出发，遍历所有的根的孩子结点：N1,N2,N3...
    - 如果所有孩子结点中的关键字的第一个字符都和P的第一个字符不匹配(只要是子字符串肯定是其中一个后缀的前
      缀)，则判定不存在该子字符串，return -1
    - 假如某个结点N3的关键字K3第一个字符与P的相同，则匹配K3和P
    - P包含在K3里(K3.length>=P.length): 若P是K3的子字符串(K3.subString(0,P.length-1)==P), 则匹配成功
    - K3包含在P里(K3.length<=P.length): 首先判断整个K3能否和P匹配, 然后取出P中排除K3之后的子串P1, 以
      N3为根结点继续重复之前的步骤对P1进行匹配, 直到匹配完所有字符
- Pattern Matching Using Suffix Trie

```
Algorithm suffixTrieMatch
    Input: Compact suffix trie T for a text X and pattern P;
    Output: 若完全匹配, 给出X中对应P开头的位置, 否则返回-1;
    {
        p=P.length;
        j=0;
        v=T.root();
        repeat{
            f=true;
            for (each child w of v){
                //已经匹配了j+1个字符
                i=start(w); //w的起始index(首字母)
                if(P[j]==T[i]){//对子树w进行处理
                    x=end(w)-i+1;//end(w)->end index of w
                    if(p<=x){
                        //判断pattern是否是当前结点关键字的子字符串
                        if(P[j:j+p-1]==X[i:i+p-1]) return i-j;
                            else return -1;
```

```
                        }
                        else{
                        //先判断当前结点关键字是否是pattern的子字符串
                            if(P[j:j+x-1]==X[i:i+x-1]){
                                p=p-x;//更新后缀长度
                                j=j+x;//更新后缀起始index
                                v=w;
                                f=false;
                                break for loop;//进行下一次repeat循环
                            }
                        }
                    }
                }
            }until(f=true || isExternal(v))
                return -1;
        }
```

14.3 Greedy Method and Text Compression

14.3.1 Greedy method technique

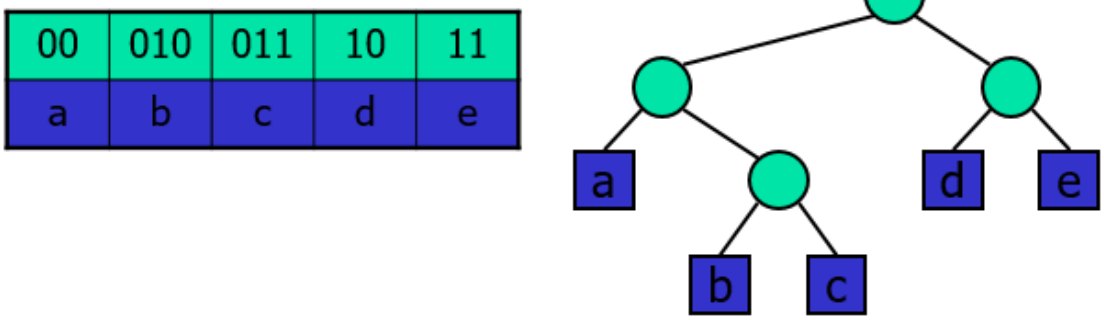
- configurations: different choices/collections/values to find;
- objective function : 分配给前面的结构,取决于取最大或最小值;
- Example: 总金额一致,如何尽可能让买到的商品总价值更高;

14.3.2 Text compression

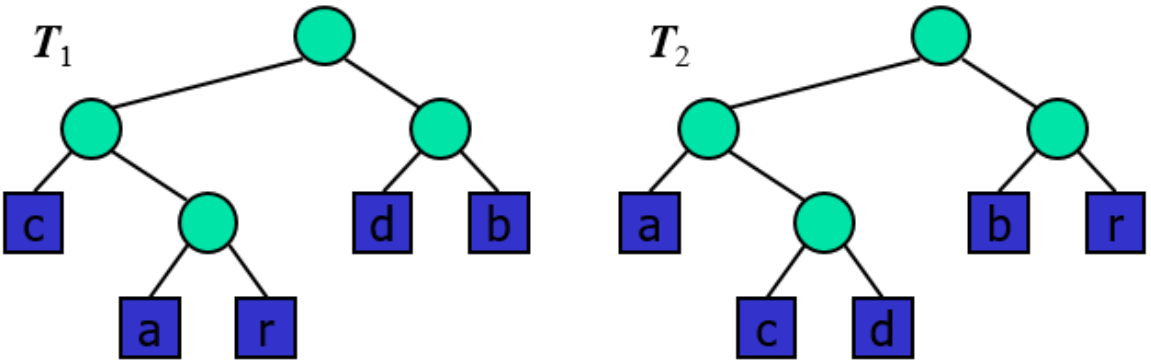
将Given String X编码为更小的String Y , 节省memory, bandwidth

14.3.2.1 Huffman encoding

- 基本思路
  - 计算每个字母c的出现频率f(c);
  - 高频字母的code word尽可能短;
  - 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;
- 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root某个字母的路径是左-右-左,则该字母的codeword为"010"



- 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;
  - Example: T2优于T1;
    - X = abracadabra
    - T_1 encodes X into 29 bits
    - T_2 encodes X into 24 bits



- Huffman Algorithm
  - 构建optimal encoding trie : 高频浅低频深
  - 基于Heap(反正只要是优先队列就好咯):
    - 首先将全部元素插入Heap, key为频率
    - 每次removeMin()两次得到两个低频元素,将其归并作为这俩元素的父结点再插入Heap, 重复这一过程直到堆里就剩下一个键值对
    - 最后这个键值对的key就是优化后的总size, 使用removeMin()输出

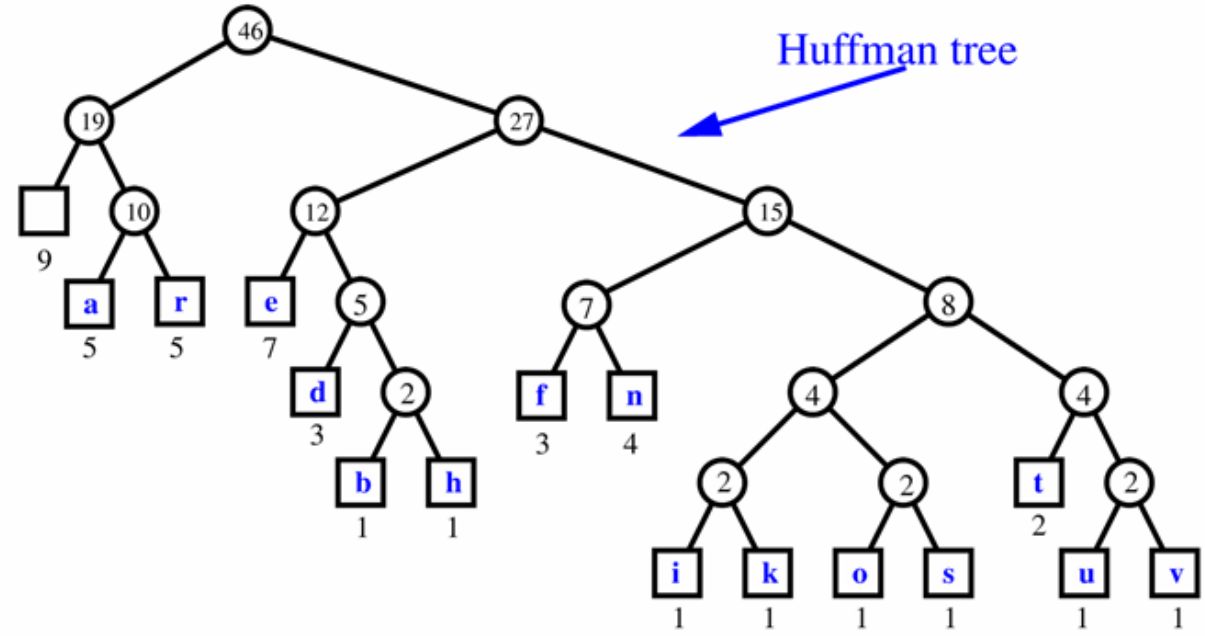
```
Algorithm HuffmanEncoding(X)
    Input string X of size n
    Output optimal encoding trie for X//最小化编码后的size
    {
    C = distinctCharacters(X);//构建字母表
    computeFrequencies(C, X);//计算字母表中每个元素的频率
    Q = new empty heap;
    for all c in C {
        T = new single-node tree storing c;
```

```
            Q.insert(getFrequency(c), T);
        }//先将全部元素及其频率以键值对形式插入Heap

        while ( Q.size() > 1 ){
            f1 = Q.minKey();
            T1 = Q.removeMin();
            f2 = Q.minKey();
            T2 = Q.removeMin();
            //每次移出并归并两个频率最低的键值对得到一个新键值对作为父结点(key为这两个键值对key的和)
            T = join(T1, T2);
            Q.insert(f1 + f2, T);//将新键值对再插入Q中
        }
        return Q.removeMin();
        }
```

- 算法分析:
  - 时间复杂度为$O(n + d\log d)$,d为字母表size;
  - 基于heap PQ:
    - 先全塞进去
    - 每次拿出来两个放回去一个直到堆里就剩下最后一个

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

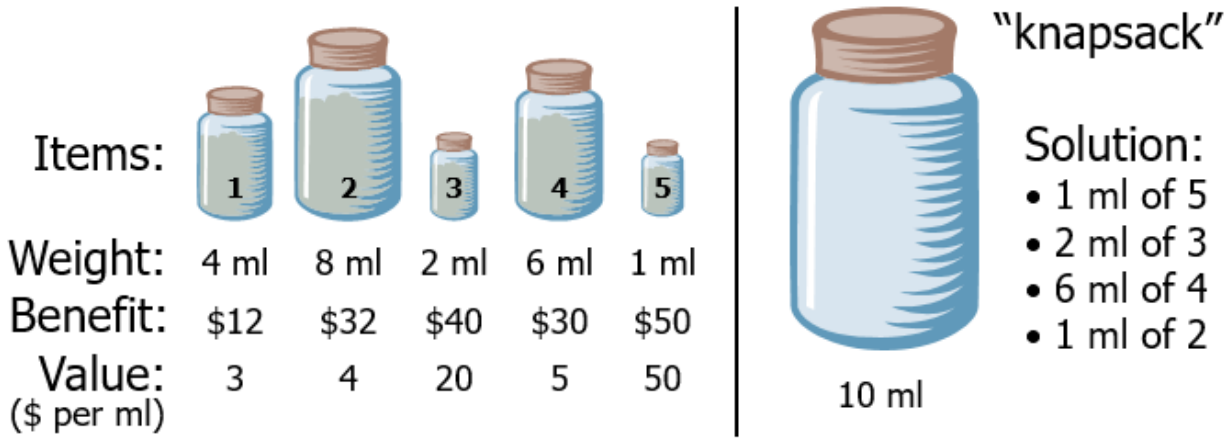


14.3.2.2 Fractional Knapsack Problem

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

```
Algorithm fractionalKnapsack(S, W)
    Input: set S of items with benefit b_i and weight w_i; max weight W;
    Output: 返回得到最优化的xi;
    {   for each item i in S{//初始化xi,value
            x_i = 0;
            v_i = b_i  / w_i ;  // value
        }
        w = 0;              // total weight
        while ( w < W ){
            remove item i with highest vi
            x_i = min{w_i , W - w};
            w = w  + min{w_i , W - w};
        }
    }
```

- Example



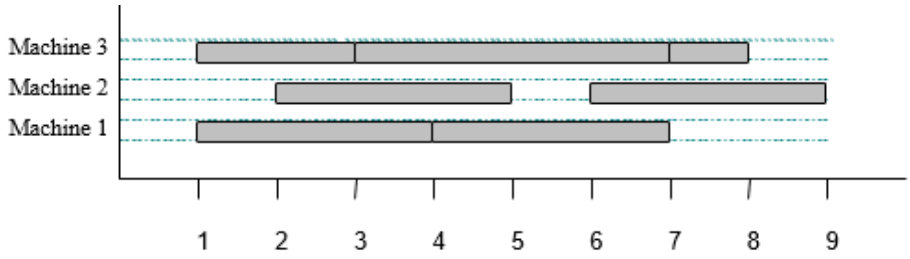
- 时间复杂度为$O(n \log n)$

14.3.2.3 Task Scheduling

- 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

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



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

14.4 Dynamic Programming

14.4.1 Analysis and Property

- 几种经典算法适用情况对比

算法	适用情况
递推	每个阶段只有一个状态
贪心	每个阶段的最优状态都是由上一个阶段的最优状态得到的
搜索	每个阶段的最优状态是由之前所有阶段的状态的组合得到的
动态规划	每个阶段的最优状态可以从之前某个阶段的某个或某些状态直接得到而不管之前这个状态是如何得到的

- 分治-动态规划-贪心

算法	分治	动态规划	贪心
适用情况	通用问题	优化问题	优化问题
子问题	子问题各不相同	子问题不独立	只有一个子问题
最优子结构	不需要	必须满足	必须满足
需要解决的子问题数量	全部	全部	一个
子问题在最优解里	全部	部分	部分
选择/求解子问题次序	选择→求解	求解→选择, 从底向上	选择→求解, 从上向下
依赖	-	有待做出的最优选择	已经得到的最优选择
思想	将大问题切分为多个小问题各个击破	将待求解的问题分解为若干个子问题，按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息	贪心算法采用的是逐步构造最优解的方法,先得到当前最优解再进行下一步求解
缺点	子问题需要完全独立	空间复杂度大	局部最优解不一定是整体最优解

  - 动态规划与贪心算法的一个重要区别: 动态规划算法的每一步决策给出的不是唯一结果，而是一组中间结果，而且这些结果在以后各步可能得到多次引用，只是每走一步使问题的规模逐步缩小，最终得到问题的一个结果
- 贪心算法
  - 贪心算法适用情况: 求解每一步的最优解，进行迭代，从局部最优能得到整体最优
  - 并不能优化计算量
- 递归算法
  - 得到最小问题问题 + 递归表达式 设计递归程序
  - 适用情况: 二分搜索/大整数乘法/矩阵乘法/棋盘覆盖/合并排序/快速排序/线性时间选择/最接近点对问题/循环赛日程表/汉诺塔
- 动态规划

- 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
- 核心代码为状态转移方程例如$d_p(n) = d_p(n-1) + 9$
- 特点是以空间换时间

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

        for(b=1;b<n;b++){
            for(i=0;i<n-b;i++){
                j = i+b;
                N_ij += infinity;
            }
            for(k=i;k<j;k++){
                N_ij = min{N_ij , N_ik +N_(k+1)j +d_i dk+1 dj+1};
            }
        }
    }
```

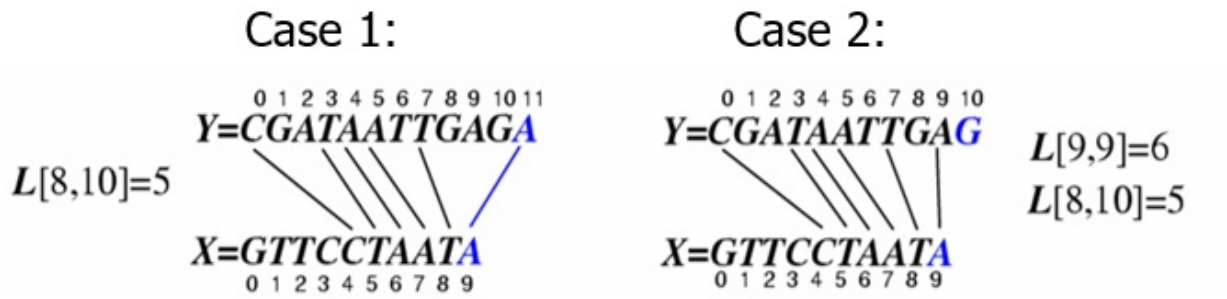
- 算法分析:
  - 先从最小的$N_{0,0}$开始,然后然后逐步计算到大矩阵；
  - $N_{i,j}$的值源于之前i行j列的entries;
  - 核心代码为状态转移方程(normal equation);
  - 时间复杂度:$O(n^3)$, 至少强于指数级；
- Example: 从三藩到纽约的最短路径
  - 首先计算从三藩出发到每一个第一站驿站的路径和距离
  - 接下来计算从三藩出发到每一个第二站驿站的最短路径（分别经过某一个相同或者不同的第一站驿站)
  - 然后计算第三站（这时候只用知道到第二站的最短路径）
  - 第四站（这时候只用知道到第三站的最短路径）
  - 不断重复上述过程直到到达纽约，即可得到从三藩出发到纽约的最短路径

14.4.2 Longest Commom Subsequence(LCS) Problem

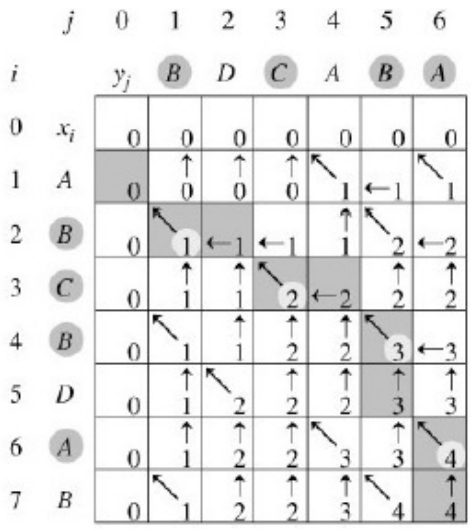
- Subsequences:
  - 格式$x_{i_1}, x_{i_2}, \cdots, x_{i_k}$ $(i_j < i_{j+1})$为的字符串;
  - 和substring的主要区别在于子序列是有序的
  - Example:'DFGHK'是子序列而'DAGH'不是子序列
- 最长共有子序列
  - Given : strings X,strings Y;

- Goal : 找到XY共有的最长的子序列;
- Example: 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;
  - 时间复杂度$O(2^n)$
- 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);
  - 若xi==yj (match), $L[i,j] = L[i-1,j-1]+1$, 否则 $L[i,j] = max\{L[i-1,j]+L[i,j-1]\}$




```
Algorithm LCS(X,Y)
    Input: Strings X(n),Y(m);
    Output: LCS的长度, L[i,j] is the LCS of X[0:i+1] and Y[0,j+1] ;
    {
        //先初始化L=0
        int[][] L= new int[n+1][m+1]

        //开始匹配
        for(i=0;i<n;i++){
            for(j=0;j<m;j++){
                if (x[i]==y[j]){
                    L[i+1][j+1]=L[i][j]+1;
                }else{
                    L[i+1][j+1]=max{L[i+1][j],L[i][j+1]};
                }
            }
        }
        return L;
    }
```



- 求取LCS序列: 基于前面的LCS长度算法

```
1.  //返回LCS Table L
2.  public static char[] reconstructLCS(char[] X, char[] Y,int[][] L){
3.      StringBuilder solution = new StringBuilder( );
4.      int i=X.length;
5.      int j=Y.length;
6.      while(L[i][j]>0){
7.          if(X[i-1]==Y[j-1]){
8.              solution.append(X[i-1]);
9.              i--;
10.             j--;
11.         }else if(L[i-1][j]>=L[i][j-1]){
12.             i--;
13.         }else{
14.             j--;
15.         }
16.     }
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	2	2	2	2	2	2	2	2	2
3	0	0	1	1	2	2	2	3	3	3	3	3	3
4	0	1	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	2	2	2	3	3	3	3	3	3
6	0	1	1	1	2	2	2	3	4	4	4	4	4
7	0	1	1	2	2	3	3	3	4	4	5	5	5
8	0	1	1	2	2	3	4	4	4	4	5	5	6
9	0	1	1	2	3	3	4	5	5	5	5	5	6
10	0	1	1	2	3	4	4	5	5	5	6	6	6

15 Graphs

图是由点集合和线集合组成的，他们分别被称作vertices和edges，分别代表了位置和存储的元素

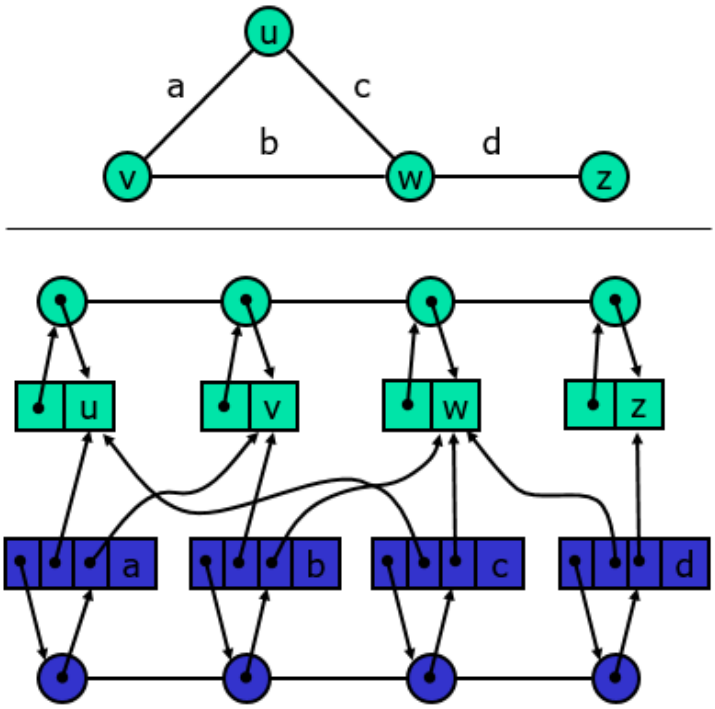
- Edge的种类
  - 由有方向的箭头连接起来的图叫做有向图，他的所有edge都是有方向的
  - 有无方向的线段连接起来的图叫做无向图，他的所有edge都是没方向的

15.1 Direct and Undirect Graph

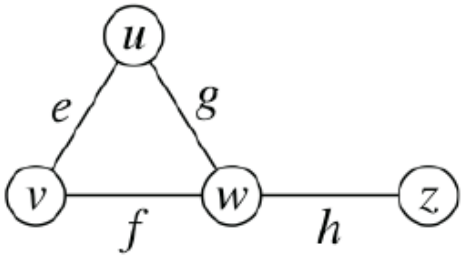
15.1.1 Undirect Graph

- 无向图的术语
  - End vertices of an edge: edge的两个端点
  - Edge incident on a vertex: 与某个vertex相连的所有edge都附属于这个vertex
  - adjacent vertices： 联通的两个节点
  - Degree of a vertex: 某个vertex拥有的edge的数量
  - Parallel edges： 两个vertex之间存在两条线段，则称这两条线段为Parallel ed
  - self-loop: 一条edge的两个端点是同一个点，则称它为self-loop
  - Path: 从某一个vertex通过edge到达另一个vertex的一系列vertices和edges叫做path
    - 如果path中的定点和边都不重复则称其为Simple path
    - cycle: path的出发的vertex和终点的vertexs是同一个vertex
      - 如果cycle中的定点和边都不重复则称其为Simple cycle
- 无向图所有节点的总度数等于edge数量的两倍
- 如果一个无向图没有self-loop和Parallel edges则边的总数$m \leq n(n-1)/2$, n为vertex的总个数
- Main Methods
  - Vertices and edges
    - positions and store elements
  - Accessor methods
    - endVertices(e): 有edge的两个端点组成的数组
    - opposite(v,e): v关于e的另外一个端点
    - areAdjacent(v, w): true, iff v和w是连通的
    - replace(v,x): replace element at vertex v with x
    - replace(e,x): replace element at edge e with x
  - Update Methods
    - insertVertex(o): 插入一个存储了element o 的vertex
    - insertEdge(v,w,o): 插入一个存储了element o 的edge(v,w)

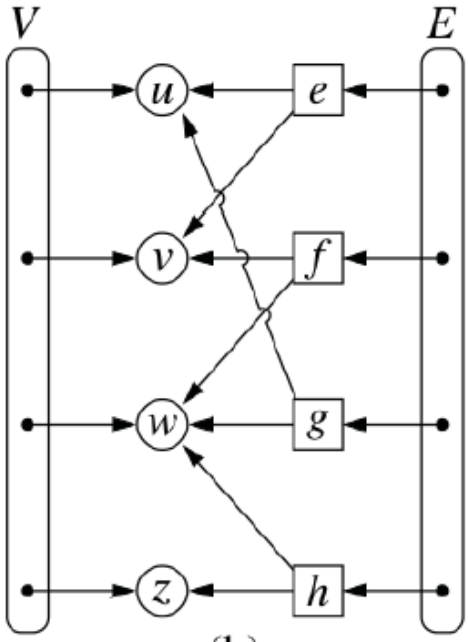
        - reomveVertex(v): 删除vertex v和附属于v的所有edge
        - removeEdge(e): 删除edge e
    - Iterator methods
        - incidentEdges(v): 返回所有依赖于vertex v的edges
        - vertices(): 返回图中所有的vertices
        - edges(): 返回图中所有的edges
- Edge List Structure
    - Vertex object
        - element
        - reference to position in vertex sequence
    - Edge object
        - element
        - origin vertex object
        - destination vertex object
        - reference to position in edge sequence
    - Vertex sequence
        - sequence of vertex objects
    - Edge sequence
        - sequence of edge objects



- reference to position in list E

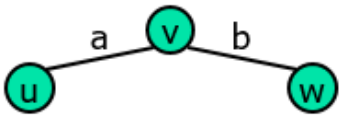


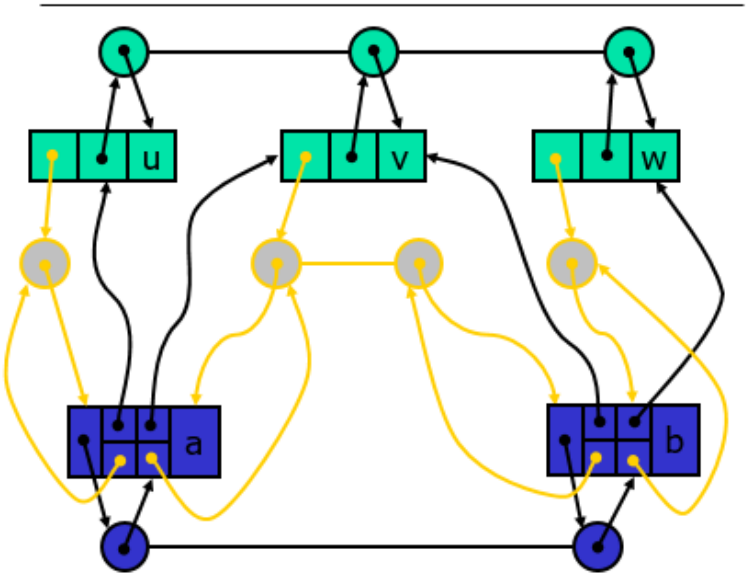
(a)

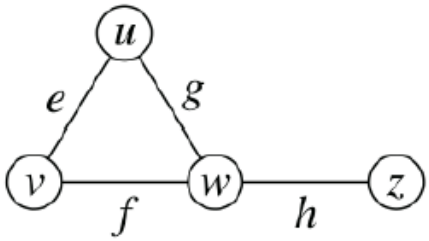


(b)

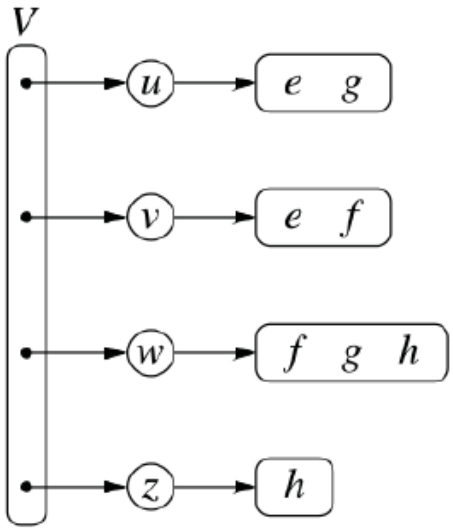
- ◦ Adjacency List Structure
  - ◦ Vertex list structure
  - ◦ Object Vertex v
    - reference to element x, to support getElement()
    - reference to position in list V
    - reference to collection of adjacent edges of this vertex, list.length=deg(v)





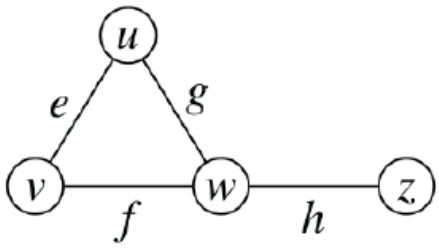


(a)

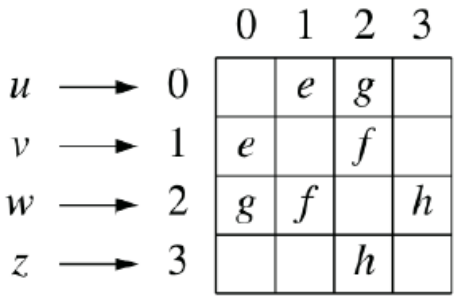


(b)

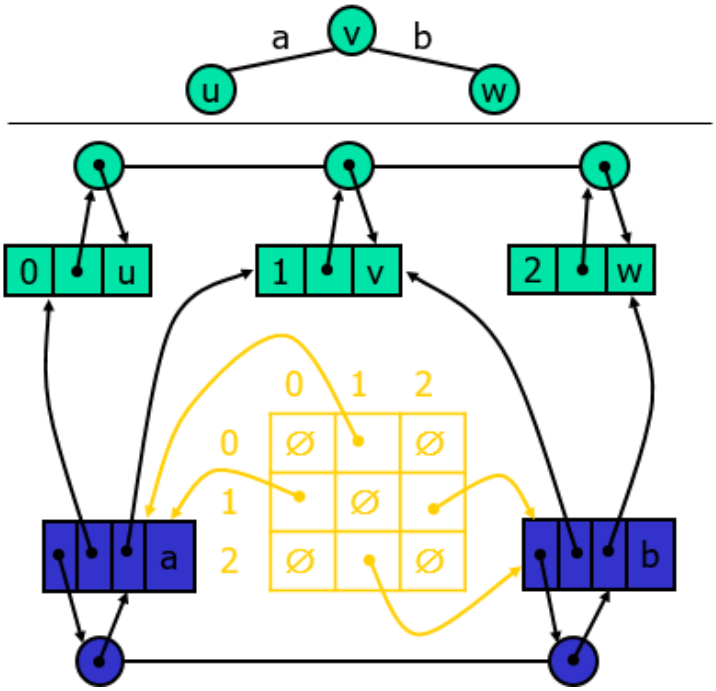
- Adjacency Matrix Structure
  - A[i][j] 储存了到边e(u,v)的链接
  - A[i][j]=A[j][i], 完全对称
  - A[i][j] is true iff e(u,v) exists



(a)



(b)



- 三种结构的性能对比

n点m边, 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)	$O(1)$	$O(1)$	$O(1)$

15.1.2 Direct graph

- Reachability : 给定端点v,经过有向路径可到达的端点数量
- 四种边 : DISCOVERY/BACK/FORWARD/CROSS
- 图的邻接矩阵

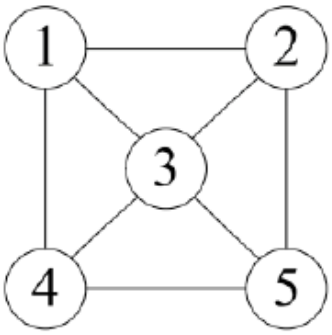


图8-1 无向图G_1

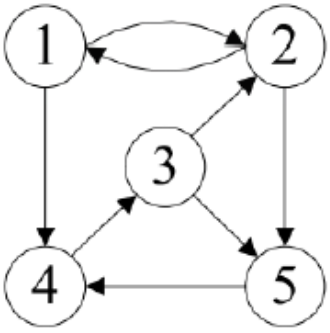


图8-2 有向图G_2

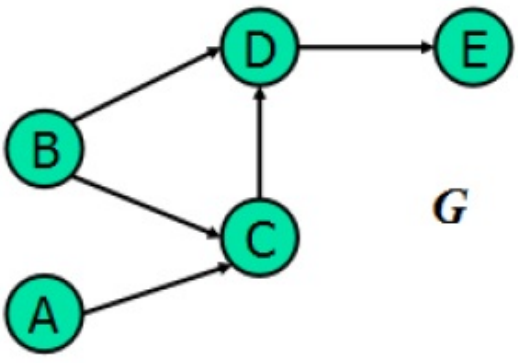
$$A_1=\begin{bmatrix}0&1&1&1&0\\1&0&1&0&1\\1&1&0&1&1\\1&0&1&0&1\\0&1&1&1&0\end{bmatrix}\qquad A_2=\begin{bmatrix}0&1&0&1&0\\1&0&0&0&1\\0&1&0&0&1\\0&0&1&0&0\\0&0&0&1&0\end{bmatrix}$$

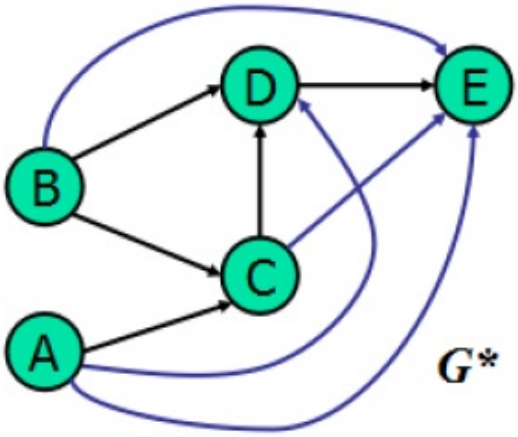
图8-12 邻接矩阵

- 无向图的邻接矩阵必然是对称的, 有向图的邻接矩阵不一定

15.1.3 Transitive Closure

```
* 传递闭包: 某两个端点之间是否存在一条路径
* 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)
    * 如果uv为G中的连通端点,则在G*中(u $\to$ v)为一条有向边
```





- 使用矩阵表示传递闭包:

- 传递闭包: 若两点i,j存在大于0的路径, 则D[i,j]=1, 否则为0, 注意因为是有向图, 不一定是对称的
- 自反传递闭包: 考虑了自反的情况, 若两点存在不小于0的路径, 则D[i,j]=1

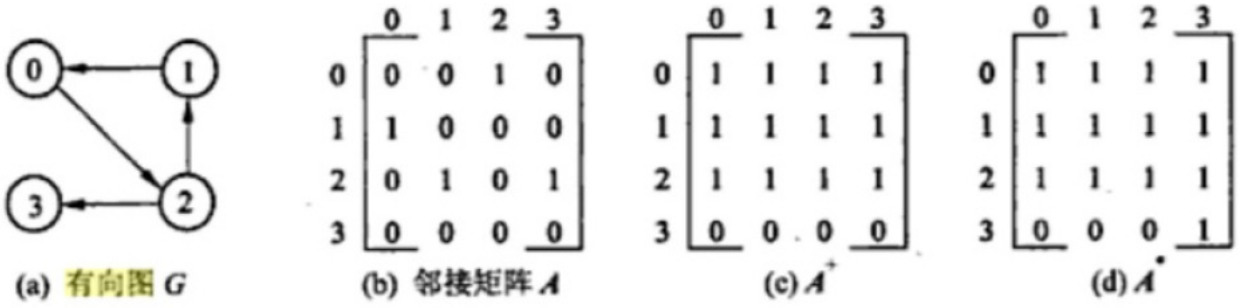


图 6.28 图 G 及其邻接矩阵 A, A^+和 A^*

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

11.1.4 Floyd-Warshall Algorithm

参考资料

- 两种应用: 求传递闭包, 求最小距离
- 使用动态规划求解
- 运行时间$O(n^3)$,仅与端点数有关
- 若两个端点是连通的, 从一个端点到另一个端点有两种可能:
  - 两端点相邻, 可以直接到达
  - 从一个端点开始, 经过若干端点到另一个端点

```
Algorithm FloydWarshall(G)
    Input:digraph G;
    Output:transitive closure G* of G;
    {
        //初始化path和distance矩阵
        for (i in [1,n]){
            for (j in [1,n]){
                if (i and j connected)
                    distance[i][j] = distance of i, j
                    path[i][j] = i
            }
        }
        G0 = G;
        //以节点k为中转点
        for (k in [1,n]){
            // 遍历所有节点连接
            for (i in [1,n]) and (i!=k){
                for (j in [i,n]) and (j!= i,k){
                    if (distance[i][k] == INF || distance[k][j] == INF){
                        continue;
```

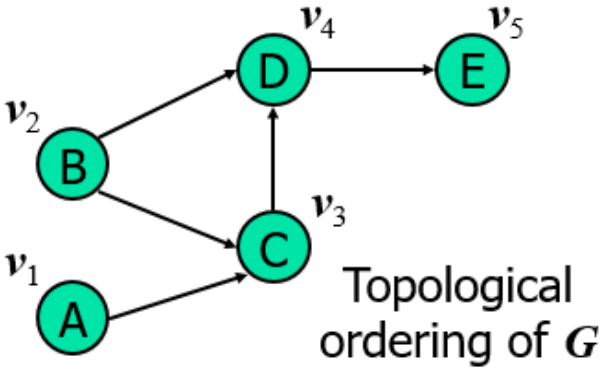
```
                        }

                        if(distance[i][j] > distance[i][k] + distance[k][j]){
                            distance[i][j] = distance[i][k] + distance[k][j];
                            //记录两点是通过k连通的
                            path[i][j] = k;
                        }
                    }
                }
            }
            return Gn;//返回最后一个图
        }
```

- NOTE: 如果distance的初始矩阵的对角线上有复数, 则无法使用该算法
- 算法思路
  - 首先给所有点编号
  - 初始化每对点之间的距离: 若两点相邻, 则距离就是边的长度, 若两点不直接相邻, 距离初始化为无穷大，并初始化path矩阵，两点相邻记录path[i][j] = i, 否则则为-1
  - 对于每对端点(i,j),查看是否存在一个中间端点k,令Dis[i,k]+Dis[k,j] < Dis[i][j]，如果存在则将k记录再path矩阵中
  - 当我们需要获得从i到j的path时, 找到path矩阵中path[i][j]的值, 若path[i][j] != i则, j= path[i][j], 然后repeat这一过程

15.1.5 DAGs and Topological Ordering

参考资料

- 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



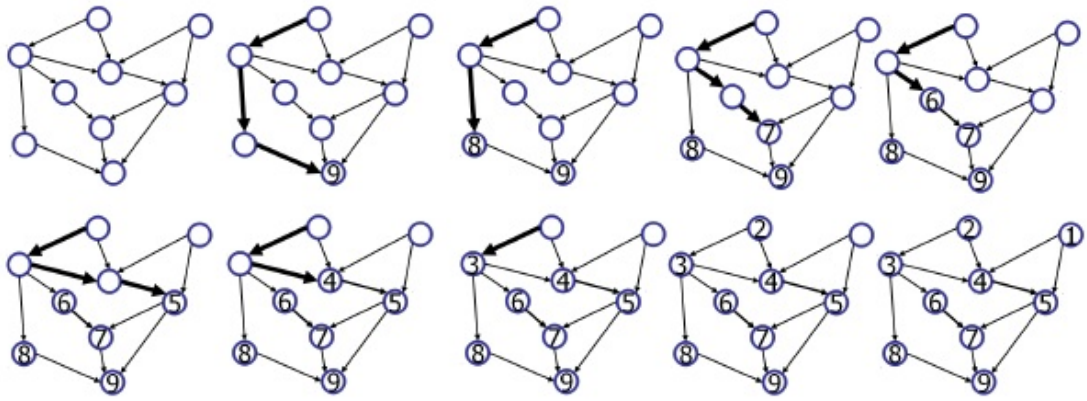
- 拓扑有序: 给端点编号,对于每条边$(v_i, v_j), i < j$;
- 有且仅有DAG才能实现拓扑有序
- 拓扑排序算法 $O(n+m)$

```
Method TopologicalSort(G)
    {
        H = G;  // Temporary copy of G
        n = G.numVertices();
        while H is not empty{
            Let v be a vertex with no outgoing edges;
            Label v = n;
            n = n - 1;
            Remove v from H;
        }
    }
```

- DFS实现拓扑排序
  - 在一个结点的所有子结点未被编号前,不给这个结点编号
  - 类似于树的pre-order traversal

```
Algorithm topologicalDFS(G)
    Input: dag G
    Output: topological ordering of G
    {
        n = G.numVertices();
        for u in G.vertices(){
            setLabel(u, UNEXPLORED);
        }

        for e in G.edges(){
            setLabel(e, UNEXPLORED);
        }

        for v in G.vertices(){
            if( getLabel(v) = UNEXPLORED ){
                topologicalDFS(G, v);
            }
        }
    }


Algorithm topologicalDFS(G, v)
    Input: graph G and a start vertex v of G
    Output: 给vertices所有的connected component加上标签
    {
        setLabel(v, VISITED);
        for e in G.incidentEdges(v){
            if( getLabel(e) = UNEXPLORED ){
                w = opposite(v,e);
                if( getLabel(w) = UNEXPLORED ){
                    setLabel(e, DISCOVERY);
                    topologicalDFS(G, w);
                }
```

```
            }
        }
        Label v with topological number n;//v的所有端点都处理完了, 赋予v编号n
        n = n - 1;
        return;
    }
```

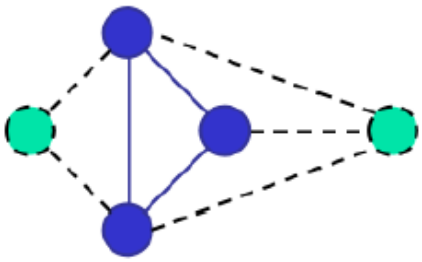


15.2 Depth-First Search

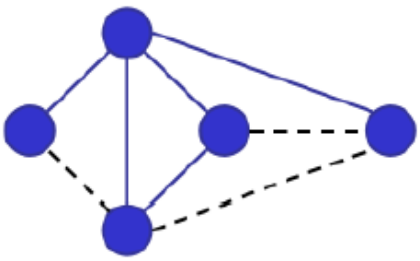
- Applications:
  - path finding;
  - cycle finding

15.2.1 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

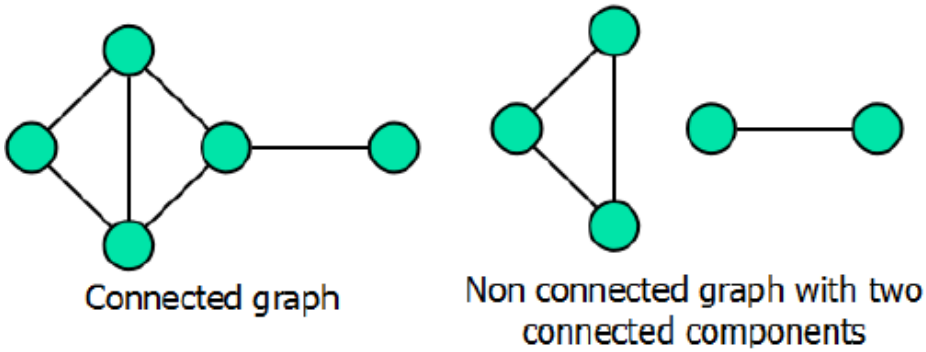


Spanning subgraph

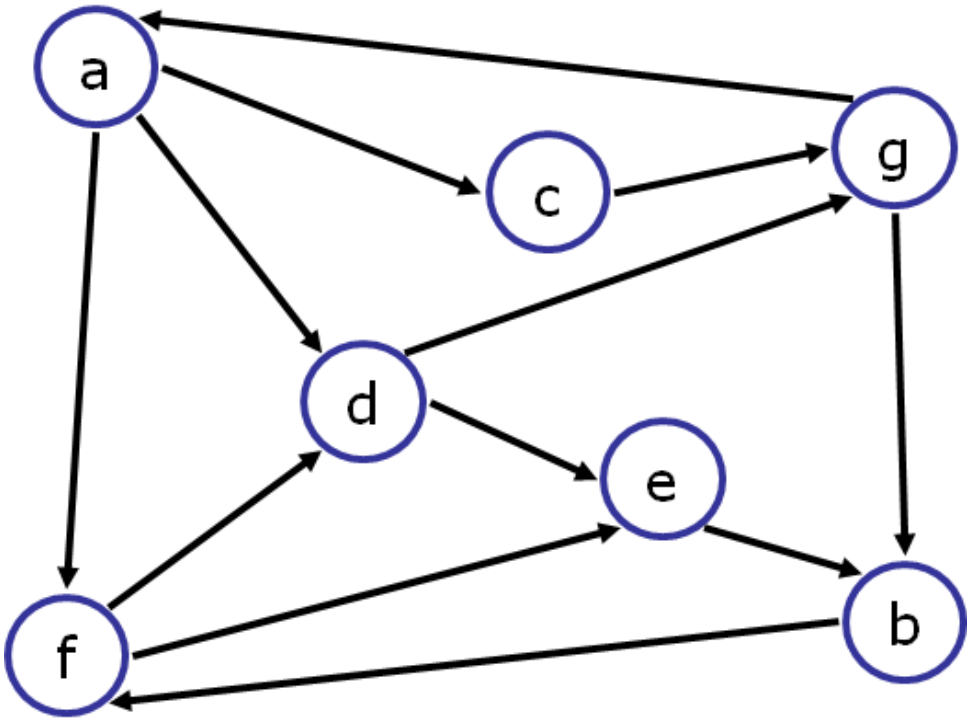
15.2.2 Connectivity

- 任意两个端点都能有路径相连，则connected

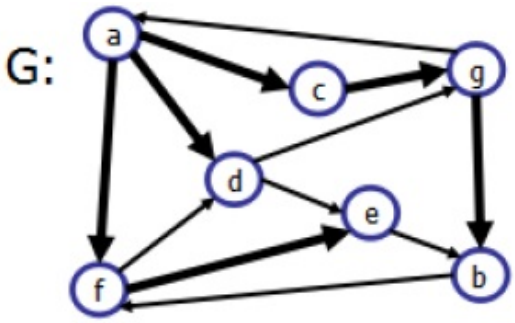
- Connected component: G中connected subgraph的数量;

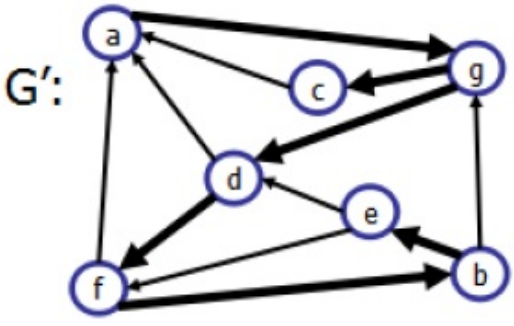


- Strong Connectivity
  - 每个端点都可以到达其余所有端点



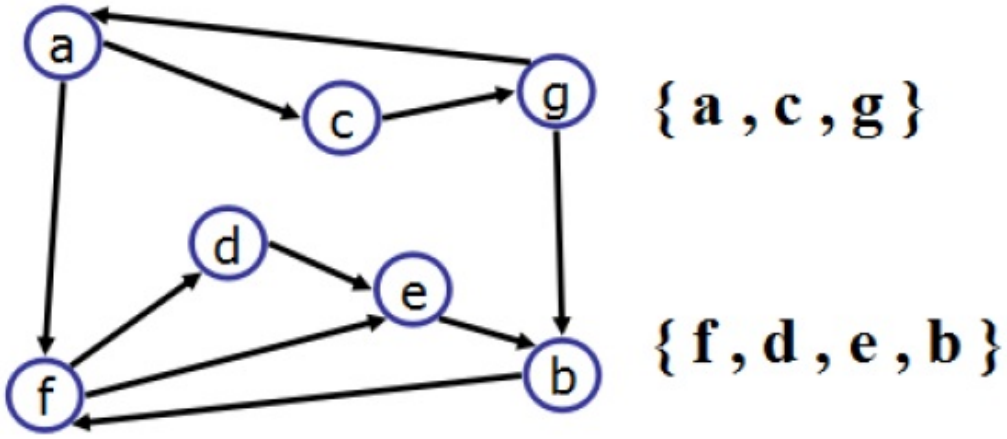
- 随机选取一个点开始DFS遍历所有的点, 如果无法遍历所有节点则不是Strong Connected，选择最深路径的终点作为起点，进行反相遍历，如果不能遍历所有节点则不是Strong Connected，如果满足两种情况则为Strong Connected





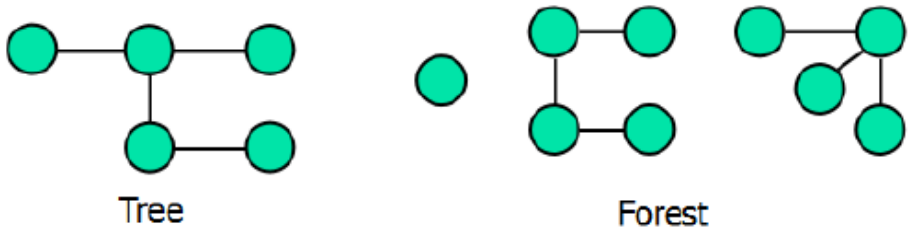
- Strong Connected Components:

- 保证Strong connectivity的前提下得到的最大subgraph
- 同样可以通过DFS得到



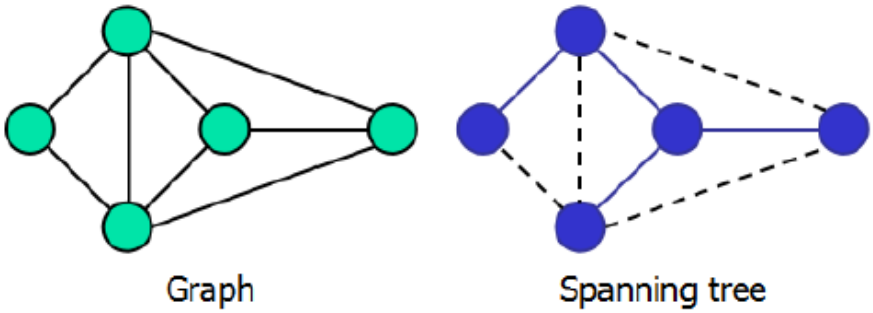
15.2.3 Trees and Forests

- 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



15.2.4 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

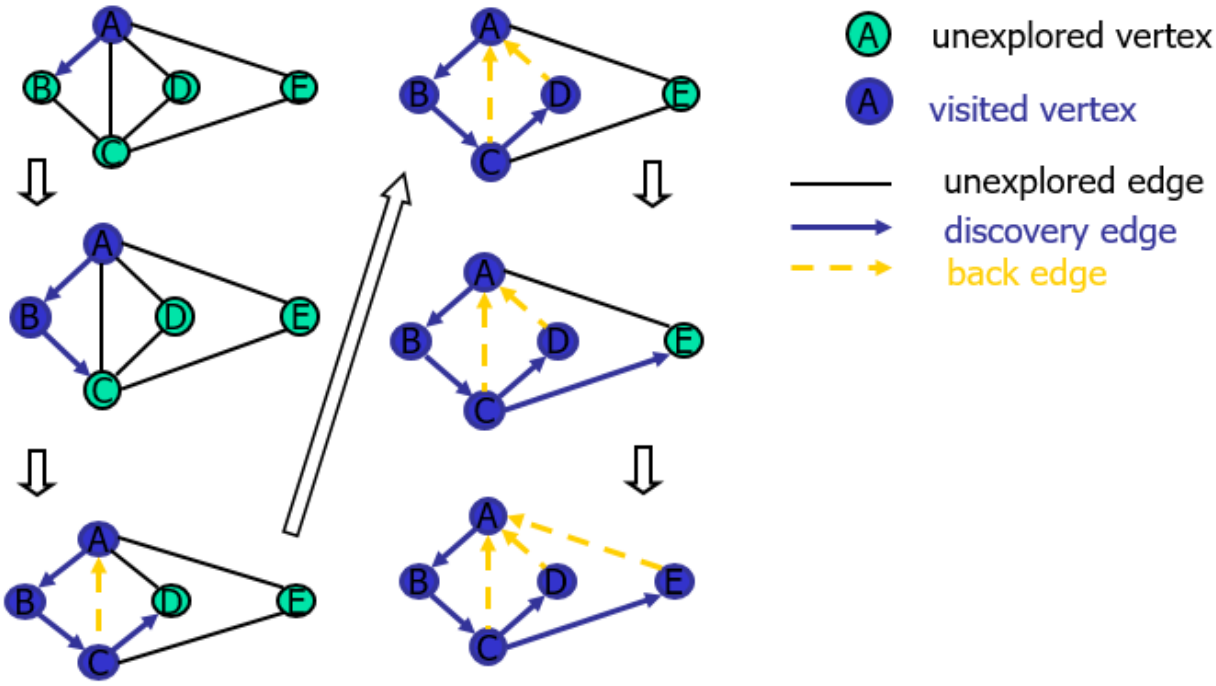


15.2.5 Depth-First Search(DFS)

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

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

```
// Part I
Algorithm DFS(G,v)
    Input: graph G and a start vertex v;
    Output: 给以v为端点的连通元素的几条边做标记,前进边DISCOVERY,后退边BACK;
    {
        setLabel(v,VISITED);//将起始端点设为VISITED
        for e in G.incidentEdge(v){//循环遍历以v为端点的所有未被访问过的边
            if(getLabel(e)==UNEXPLORED){
                w=opposite(v,e); //找到另一个端点w
                if(getLabel(w)==UNEXPLORED){
                    //若该端点也没有被探索过-->将边标签更改DISCOVERY
                    setLabel(e,DISCOVERY);
                    DFS(G,w);//以w为起点递归这一过程直到遇到的新w已经被探索-->进入else
                }else{ //将e标签设置为BACK-->返回边
                    setLabel(e,BACK);
                }
            }
        }
    }

// Part II
Algorithm DFS(G)
    Input:graph G
    Output: 深度优先遍历, 将G的标签更改为合适的状态
    {
        //初始化所有端点和边的标签为"UNEXPLORED"
        for u in G.vertices(){
            setLabel(u,UNEXPLORED);
        }
        for e in G.edges(){
            setLabel(e,UNEXPLORED);
        }

        //对每一个未被探索过的端点, 调用之前的Part I算法
        //这样做的原因是图不一定是完全连通的, 从一个端点不一定能探索所有的端点和边
        for v in G.vertices(){
            if(getLabel(v)==UNEXPLORED) DFS(G,v);
        }
    }
```



- Properties:
  - 第一个算法DFS(G,v)访问给定端点v能到达的所有边和端点,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()
  - 时间复杂度$O(m+n)$
- Application : Path Finding
  - 给定端点v,z,得到vz之间的path(只要有一条就行)
  - 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, 以栈的形式返回路径;

```
Algorithm pathDFS(G,v,z)
    Input: Graph G, start vertex v, end vertex z;
    Output: the path from v to z(elements in Stack S);
    //和普通的DFS相比在处理边的时候多了入栈和出栈的过程
    {
        setLabel(v,VISITED);
        S.push(v);//v入栈作为起点
        if(v==z) return S.elements(); //递归到最后就会得到这个结果
            for e in G.incidentEdges(v){
```

```
                if(getLabel(e)==UNEXPLORED){
                    w=opposite(v,e);
                    if(getLabel(w)==UNEXPLORED){
                        setLabel(e,DISCOVERY);
                        S.push(e);//把e加入栈
                        pathDFS(G,w,z);//递归执行, 最后结果是所有需要的边都入栈了
                        S.pop(e);
                    }else
                        setLabel(e,BACK);
                }
            }
        S.pop(v);
    }
```

- 分析两个pop()的作用:
  - 一开始根据递归将点和边加入栈$\{A, AB, B, BC, C, CD, D, \cdots\}$
  - S储存从A到某点的一条路径, 链状
  - 如果找到了如$A \to D$, 则返回S的所有元素
  - 如果遍历到这条链的头都没找到, 按相反顺序将点和边pop出S: $D \to CD \to C \to \cdots$, 留下起始点A, 对A的下一条链进行遍历(如果多条边联结A的话)
  - 还是需要注意递归这个过程
  - 全过程无环(BACK边不加入S)
- Application : Cycle Finding
  - 查找simple cycle(返回所有的环)
    - 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;
  - 路径查找基于DFS, 将DISCOVERY边加入栈, 而环查找基于路径查找, 当出现BACK边时, 将栈中包含当前起始点的环放入T

```
Algorithm cycleDFS(G,v,z){
    setLabel(v,VISITED);
    S.push(v);//v入栈作为起点
    if(v==z)
        return S.elements();
    for e in G.incidentEdges(v){
        if(getLabel(e)==UNEXPLORED){
            w=opposite(v,e);
            if(getLabel(w)==UNEXPLORED){
                setLabel(e,DISCOVERY);
                S.push(e);//把e加入栈
                pathDFS(G,w,z);//递归
                S.pop(e);//返回e,并将e移出栈
            }else{//可能找到了一个环
                T=new empty stack;
```

```
                    //检验是否为环
                    while(not S.isempty()){//S中的元素都从S中取出放入T，直到这一整个环的元素都被移动
                    o=S.pop();
                    T.push(o);
                    if(o==w)
                        return T.elements;
                    }
                }
            }
        }
        S.pop(v);
    }
```

- 找出所有连通的元素

```
Algorithm DFSComplete(Graph G){
    Set<Vertex<V>> known = new HashSet<>();
    Map<Vertex<V>,Edge<E>> forest = new ProbeHashMap<>();
    for (Vertex<V> u : g.vertices( )){
        if (!known.contains(u)){
            DFS(g, u, known, forest); // (re)start the DFS process at u
        }
    }
    return forest;
}
```

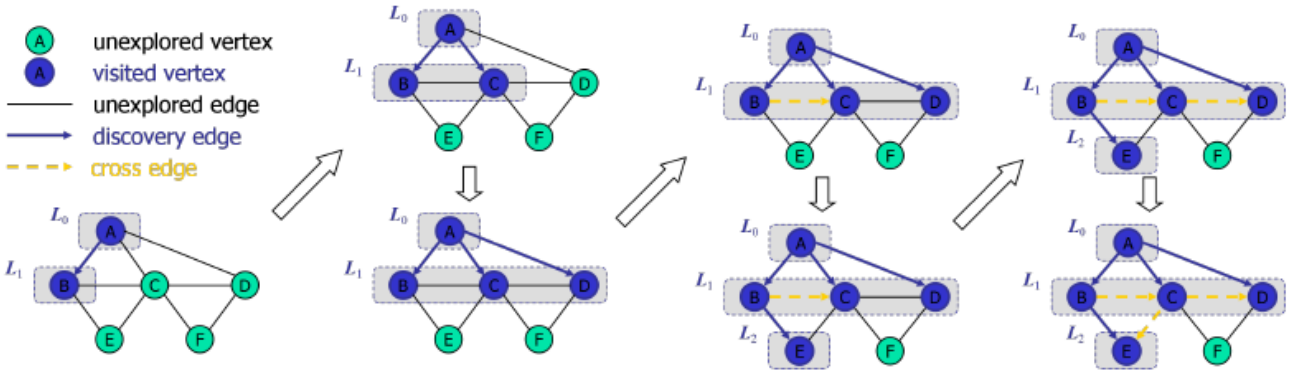
- 其他应用,如测试图的连通性…

15.3.1 Breadth-First Search

- Applications
  - 给定端点，返回最短路径;
  - 给出simple cycle;
- BFS Algorithm

```
//Part I
//每一层的端点放一层，通过遍历当前层的端点查找下一层的端点并加入新队列
Algorithm BFS(G,s){//基于队列，和DFS(G,v)相比不用递归
    L0=new empty sequence;
    L0.insertLast(s);
    setLabel(s,VISITED)//起始端点
    i=0;
    while(!Li.isEmpty()){//当前层为空说明已经遍历完毕(上一层为最底层)
        L(i+1)=new empty sequence;//构建一个新序列用于存放下一层的点
        for v in Li.elements(){
            for e in G.incidentEdges(v){
```

```
                    if(getLabel(e)==UNEXPLORED){//这说明是不同层的端点
                        w=opposite(v,e);
                        if(getLabel(w)==UNEXPLORED){
                            setLabel(e,DISCOVERY);
                            setLabel(w,VISITED); //和DFS的一个不同点：在这里给点加标签
                            L(i+1).insertLast(w); //将加好标签的点放进新序列(这个新序列在下个循环旧序列，然
                                后再创造序列存放更深层的点)
                        }else
                            setLabel(e,CROSS);//同一层的端点
                    }
                }
            }
            i++;
        }
    }

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



15.3.2 Properties and analysis

* 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一次
* 对于每个端点,调用一次incidentEdges()
* 运行时间$O(n+m)$

15.3.3 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";

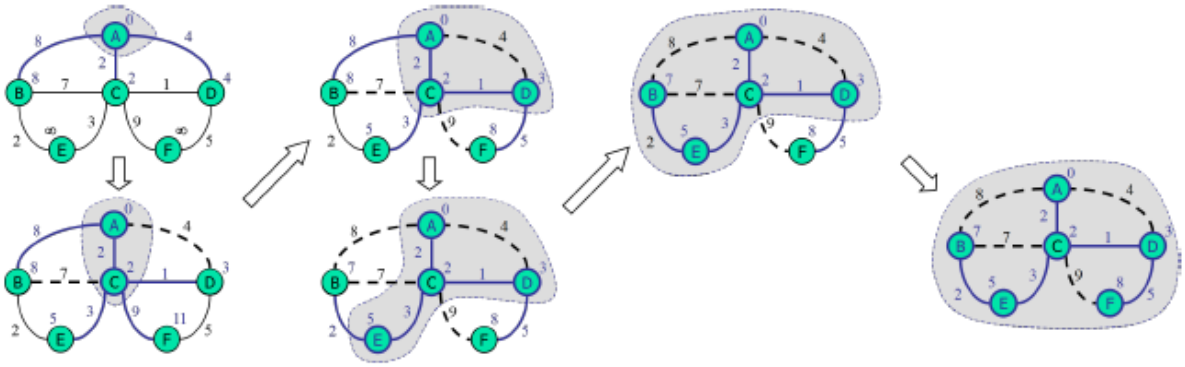
Applications	DFS	BFS
Spanning forest , connected components , paths , cycles	Y	Y
Biconnected components	Y	N
Shortest paths	N	Y,路径边尽可能少
Label	Back edge(v,w) : w是v的祖先	Cross edge(v,w) : w是v的同级或下级, 且w已被访问

16 Shortest Path

- 之前的路径算法包括BFS/DFS/Floyd-Warshall
- Weighted Graphs: 每条边都有权重(price, distance…)
- 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;
  - 从起始端点到其他任意端点的最短路径组成一棵树;

16.1 Dijkstra Algorithm

- distance(s,v):两端点之间的最短路径;
- 该算法的目标:给定起始点s,计算s到每个端点vi的距离label D(vi);
- 前提条件:
  - connected graph
  - undirected edges
  - nonnegative edge weight
- At each step:
  - 计算s到端点的距离d,加标签,并将该端点加入输出集合;
  - 计算与相邻且还未加入输出集合的点;
  - 基于edge relaxation
- Edge relaxation
  - edge e=(u,z) : u是最近被加入输出集合的端点,z还未被加入;
  - relaxation:以如下形式对D(z)进行更新
    $D(z) = min(D(z), D(u) + weight(e))$
  - 每个点都有初始距离参数，如果这个距离参数大于(前一个点+相邻边)，进行更新;
  - 可能有许多路径，找其中距离最短的,更新初始距离参数

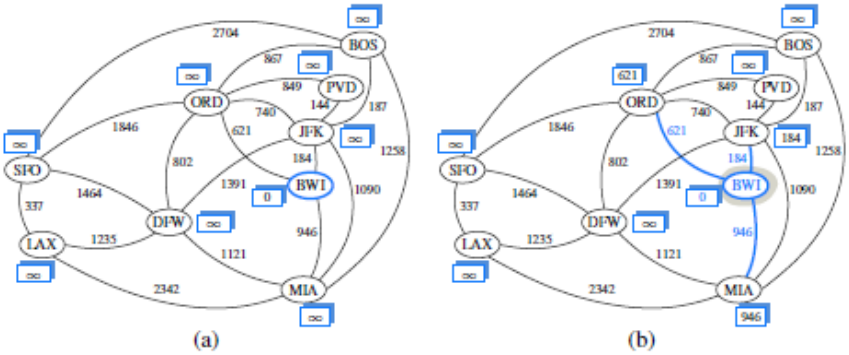


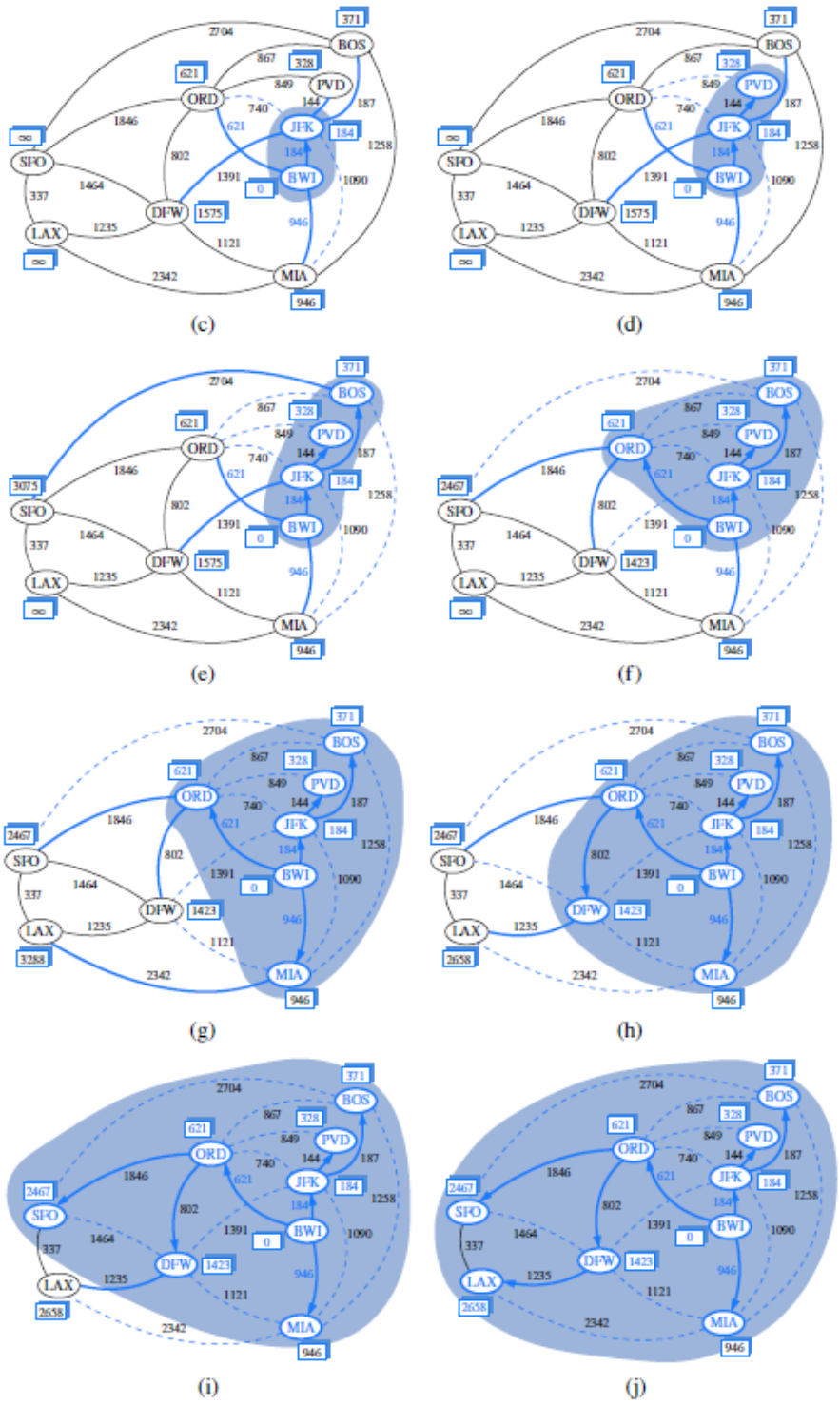
```
1.
```

- 算法分析:
  - 创建优先队列Q
    - adaptable PQ: $O(n \log n)$;
    - bottom-up heap constructure $O(n)$:
  - while循环的每一步:
    - Q.removeMin(): $O(\log n)$
    - relaxation:$O(deg(u) \log n)$，对于每个点比较$O(1)$, 在优先队列中更新并更改优先级 $O(\log n)$
    - while循环总共需要:$O((m+n) \log n)$
  - 整个算法:$O((m+n) \log n)$, 相当于优先队列中的每个端点都被更新了标签/移出

- 该算法基于贪婪：距离升序增加端点
- 不能有负的边权重

- Example
  - The start vertex is BWI.
  - A box next to each vertex v stores the label D[v]
  - Cloud:
    - 每次removeMin()操作, 将PQ中最小的端点拿出来进行计算, 直到PQ为空
    - 云外的端点为尚在优先队列中的
  - 最后的粗线为到每个点的最小路径





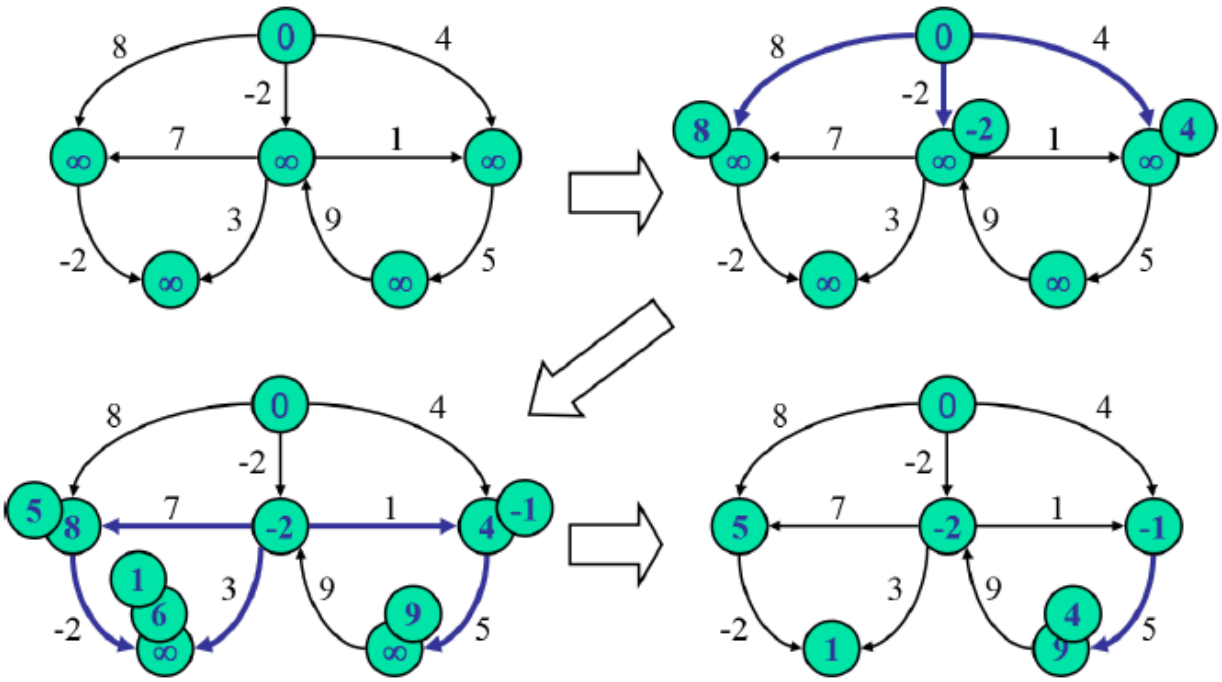
- Floyd算法和Dijkstra算法的对比

Item	Floyd	Dijkstra
Method	DP	Greedy
Core function	$Dist(i,j)=min(Dist(i,j),Dist(i,k)+Dist(k,j))$	$Label(v)=min(Label(u),Label(v)+d(u$
Time complexity	$O(n^3)$	$O((m+n)logn)$
Result	Min distance between each pair	Min from the start point
Negetive weight	Y	N

16.2 Bellman-Ford Algorithm

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

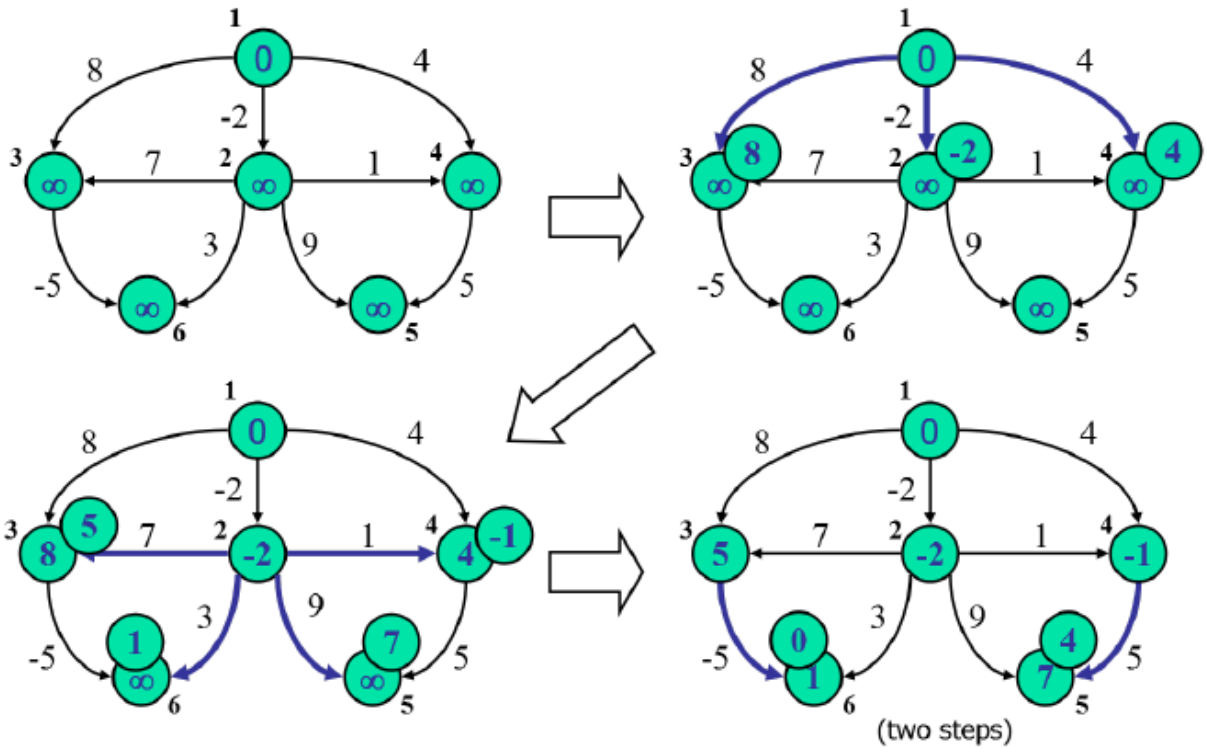
```
Algorithm BellmanFordDistance(G,s)
    {
        for v in G.vertices(){
            if(v==s)
                D[v]=0;
            else
                D[v]=infinity;
        }
        for(i=1;i<n;i++){
            for e in G.edges(){
                u=G.origin(e);//有向边的起始端点
                z=G.opposite(u,e);//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



* 与Dijkstra的区别: Dijkstra每次只是对当前最小端点的相邻端点进行比较更新， BF算法对所有端点进行比较更新

16.3 DAG-based Algorithm

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




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

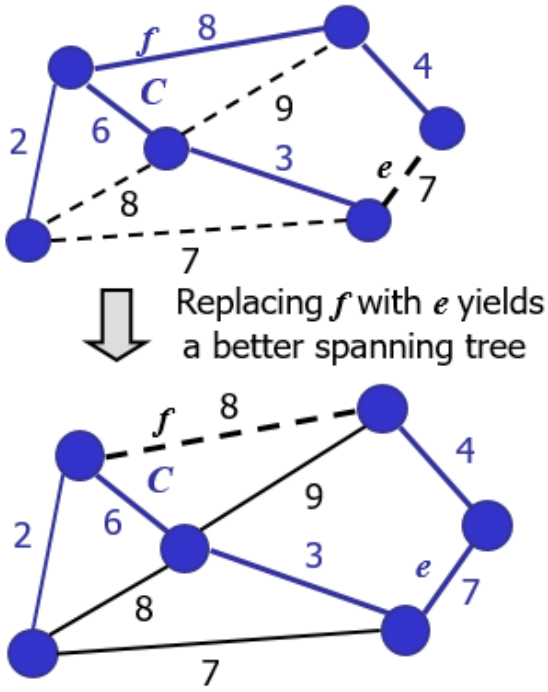
16.4 Minimum Spanning Trees(MST)

- 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
- MST的边的数量为: n-1, 两个端点一条边

16.4.1 Cycle property

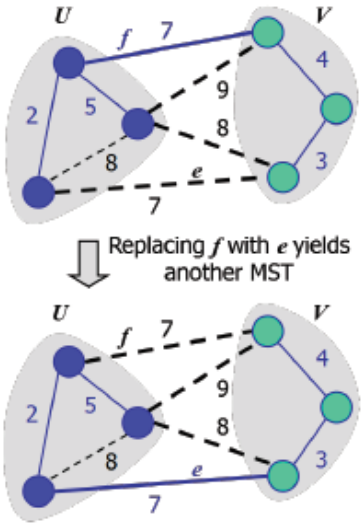
- T: 权重图G的MST

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



16.4.2 Partition Property

- 类似离散数学里的二分图
- 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



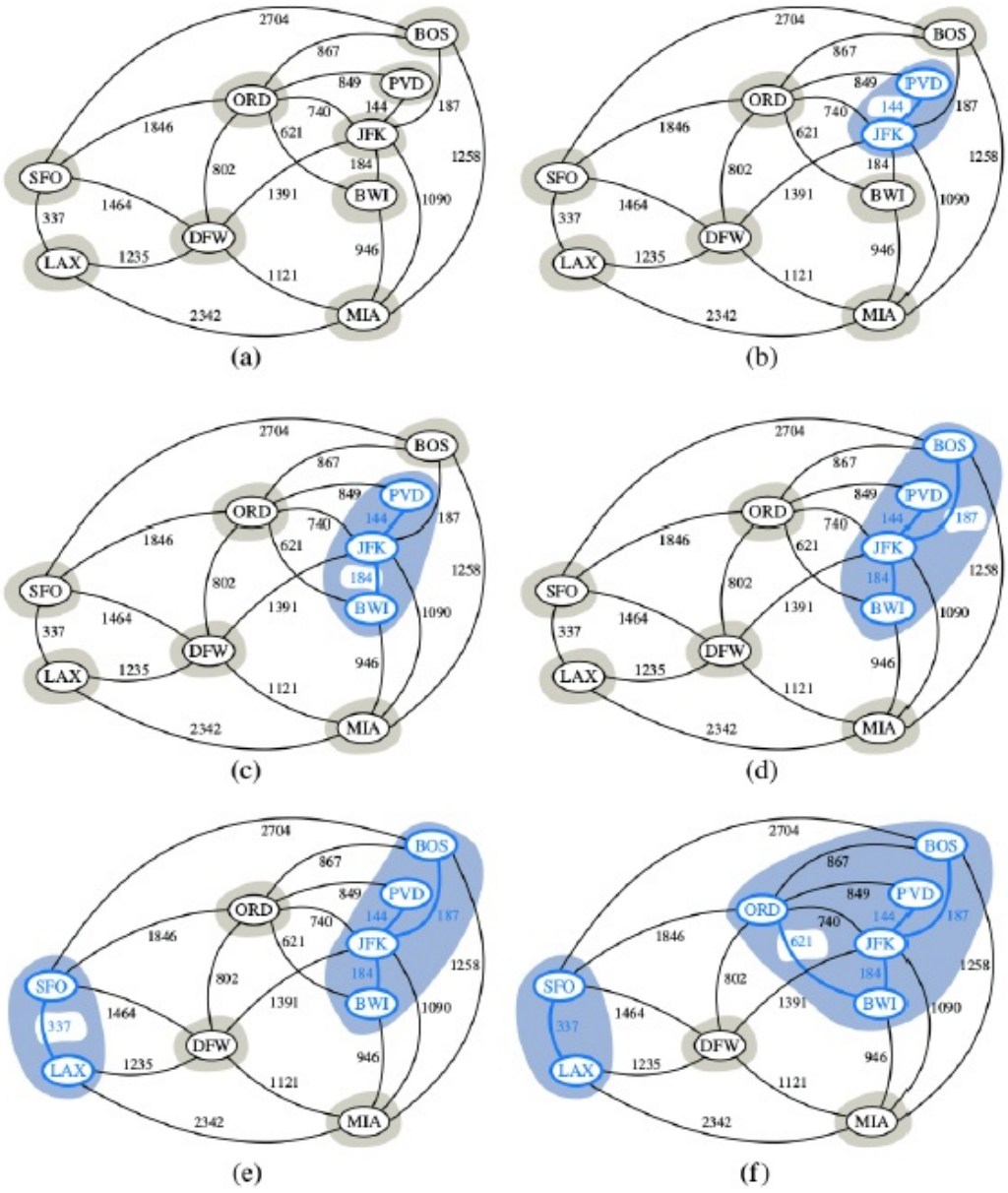
16.4.3 Kruskal Algorithm

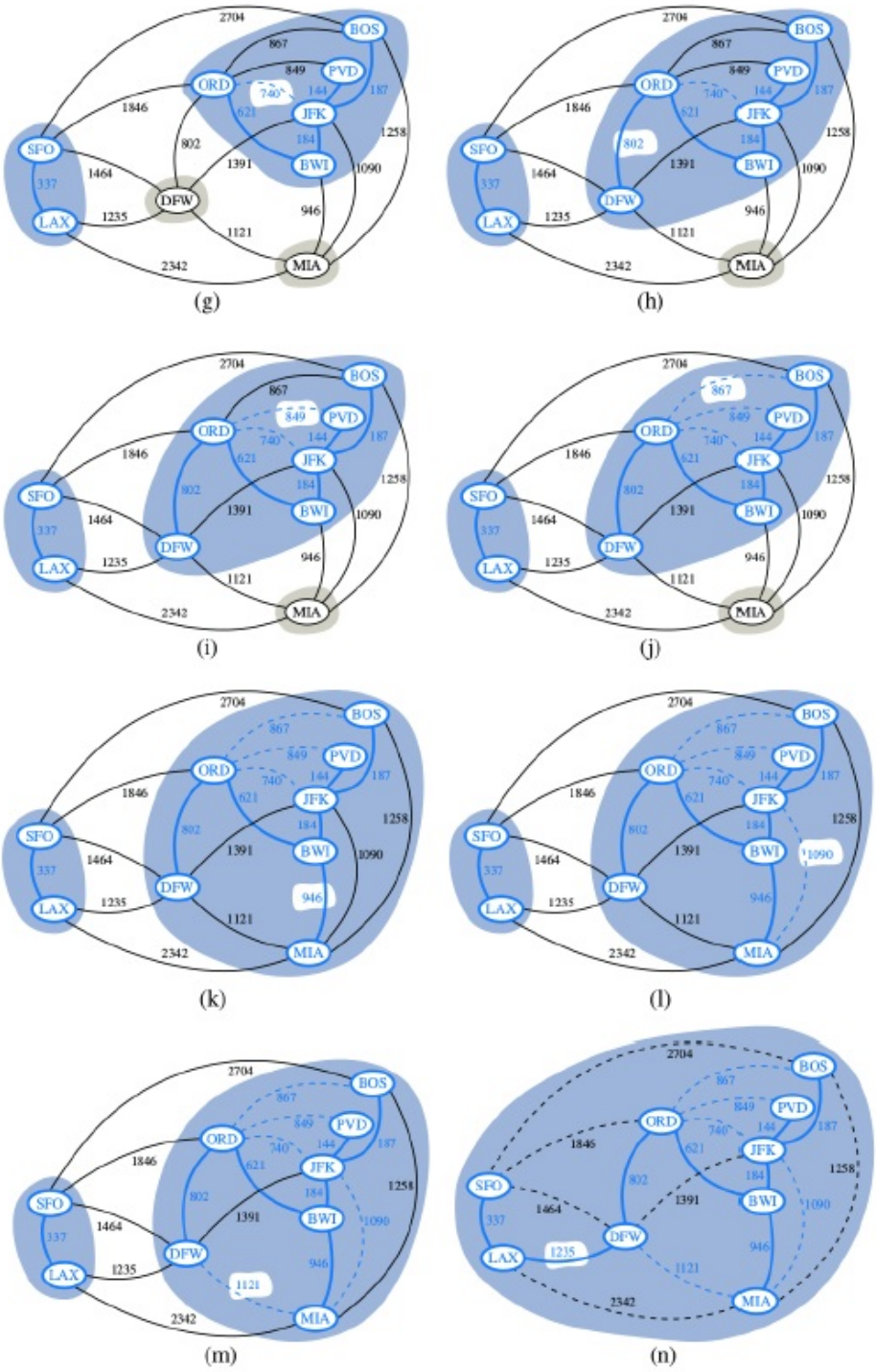
- 算法描述: 用于求取MST
  - 假设原图G有v个顶点, e条边
  - 新建图Gnew有v个顶点,但暂时不包括边
  - 将原图G中的e条边按权重进行排序
  - 在Gnew的所有点连通前, 从最小权重边e开始进行以下循环:
    - 若e的两个端点u,v在Gnew中不连通
    - 将e加入Gnew中
  - 最后返回Gnew

```
Algorithm KruskalMST(G){
    //每个cloud(v)代表v连通的点, 初始化为v自身
    for v in G.vertices(){
        define a Cloud(v) of {v};
    }

    T=null;//初始化最小生成树为0
    //创建一个优先队列Q, 和Prim/Dijkstra不同, 该优先队列储存边并按权重排序
    Create priority queue Q{
        element:edge e in G;
        key:weight of e;
    }

    while (T.edges().length()<n-1){//注意MST边数为n-1
        //移出权重最小的边并判断两端点是否在同一个类内
        edge e =T.removeMin();
        u,v=endpoints(e);
        //若不在同一个类内, 合并两个云, 相当于用e将两个云连通
        if(Cloud(v)!=Cloud(u){
            T.insertLast(e);
            Merge Cloud(v) and Cloud(u);
        }
        //否则就当这条边被扔了, 进入下一次循环
    }
    return T;//等到T储存的边数量达到了MST的边数n-1, 执行输出
}
```

- 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;
  - 可以通过使用并查集加以简化
    - find(u): 用于返回u所属的cloud, 并与find(v)进行比较
    - union(u,v) : 合并两个云集合
- 时间复杂度:
  - 对边进行排序: $O(m \log m)$

    - 对于简单图: $m = n^2 \rightarrow O(m \log n)$
  - 引入并查集:
    - find: 2m
    - union: n-1
    - 总时间: $O(m + n \log n)$
  - 对于连通图:**时间复杂度可归总为**$O(m \log m)$
- Example：



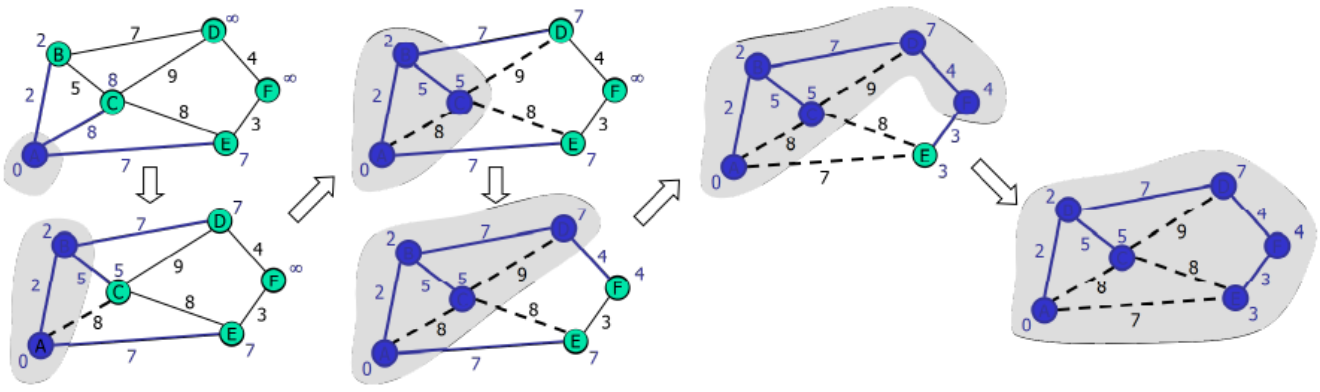


16.4.4 Prim-Jarnik Algorithm

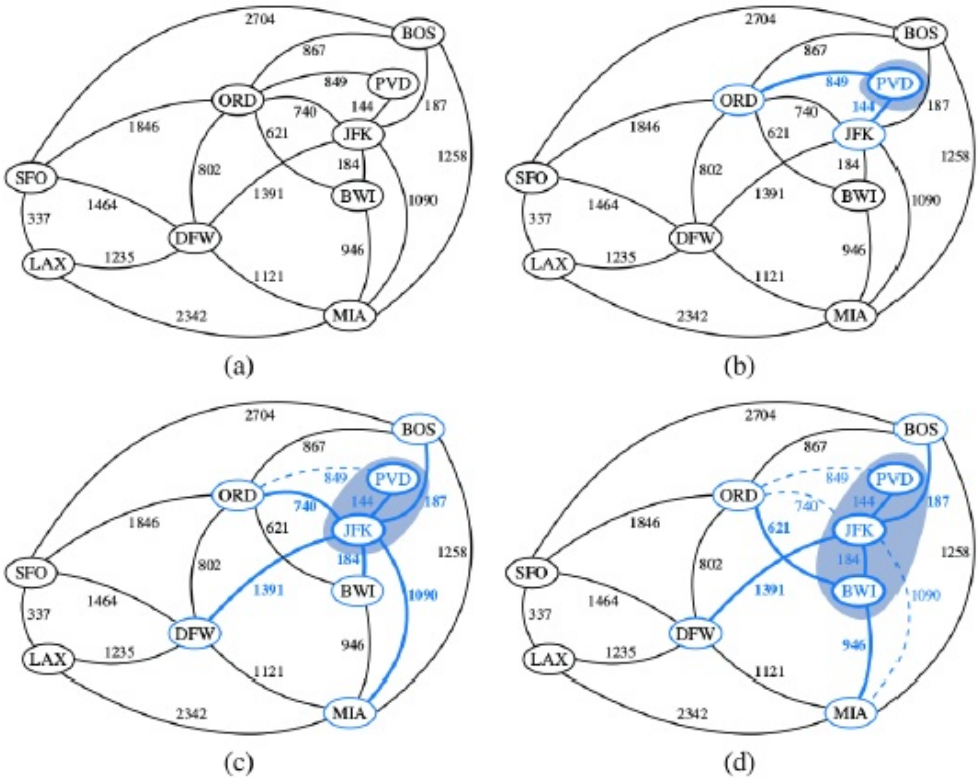
- 与Dijkstra算法类似，都是对连通图操作
- 算法描述: 同样用于求取MST
  - 随机选定一个端点v作为起始点, 放入Vnew={v},Enew={}
  - 在Vnew包含全部端点前, 循环执行以下步骤:
    - 寻找Vnew中端点的外接边
    - 将权重最小的边e加入Enew

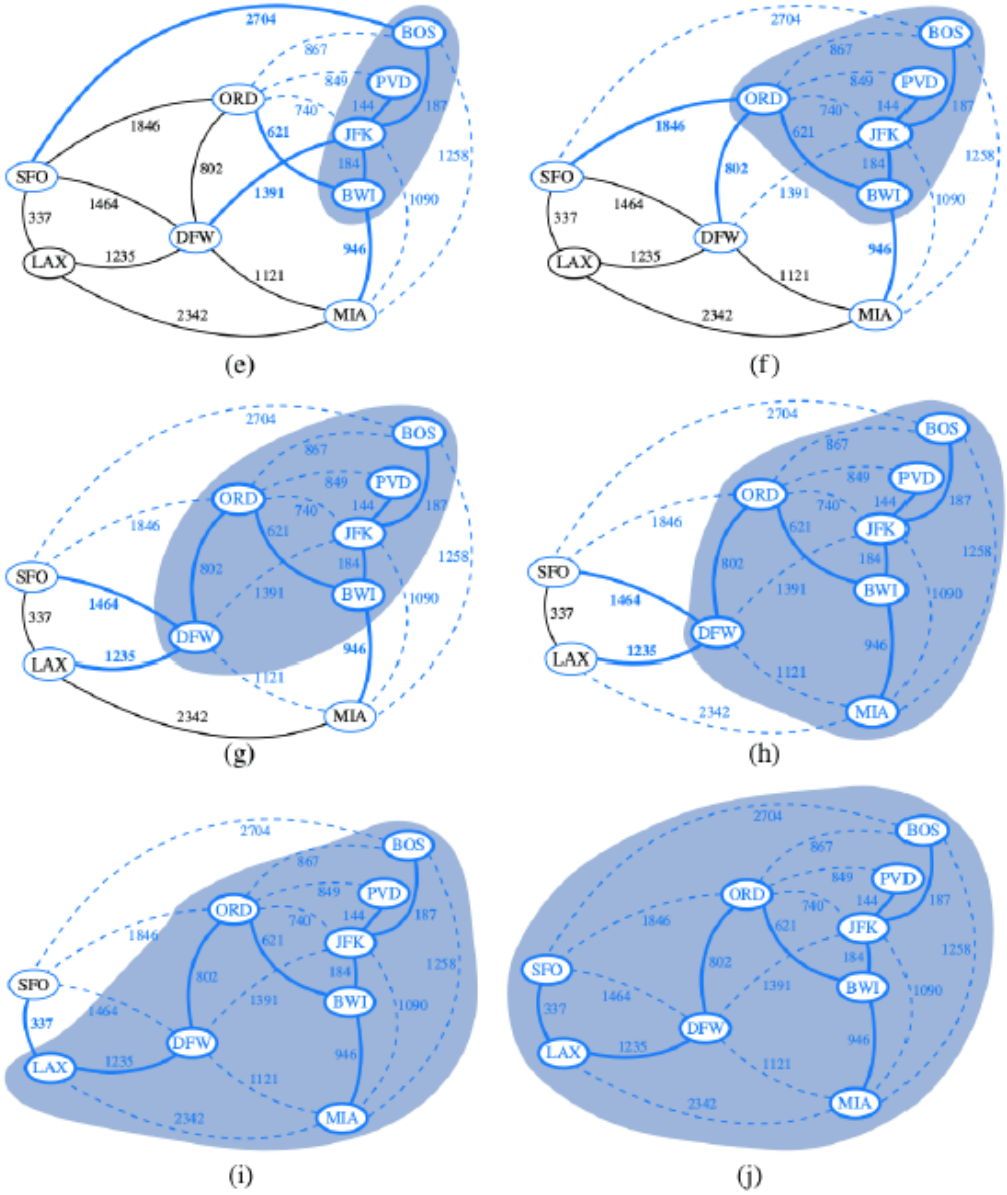
    - 将e边另一头的端点w加入Vnew
- 最后得到的Vnew和Enew集合即代表最小生成树

```
Algorithm PrimJarnikMST(G)
{
    //初始化起始点v, label=0
    //剩余点label=infinity
    //T用于存放加入MST的端点和边
    randomly pick any v of G;
    D[v]=0;
    for ((u in G.vertices()) && (u!=v))
        D[u]= infinity;
    T=null;

    //和Dijkstra一样创建一个PQ存放所有端点
    //element包含了端点以及边
    Create priority queue Q storing each vertex u{
        entry((D[u],u);
        element=(u,null);
        key=D[u];
    }

    while(!Q.isEmpty()){
        //从Q中取出标签最小的与T连通的端点加入云
        (u,e)=Q.removeMin();
        T.insertLast(vertex u,edge e);

        //寻找T中所有端点的外接边，并更新外接端点的label
        for (z in Q) && (areAdjacent(z,u){
            if(w(u,z)<D[z]){
                D[z]=w((u,z));
                z.setElement(z,(u,z));//更新端点z在Q中的element
                z.setKey(D[z]);//更新端点z在Q中的key
            }
        }
    }
    return T;
}
```



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





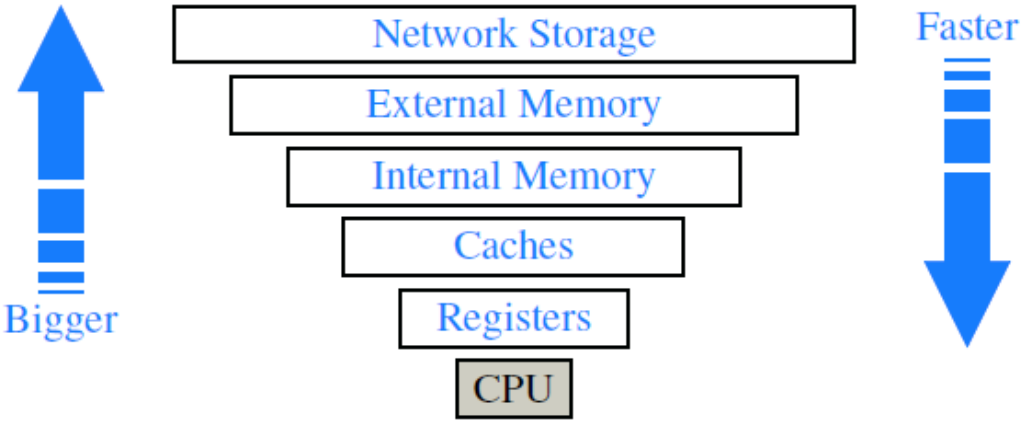
17 Memory Management

参考资料

- Application's memory有四个组成部分
  - Stack: 类似于data strucuture的stack，运行时，将调用方法， 压入stack，执行结束后，就将其pop掉(String, int, float等基本数据类型，以及object的对象都存储在这里)
  - Heap：存储复杂的object， Stack或global中的对象指向这里的object
    - 当heap中的object与stack或global中的对象不存在任何关系时，garbage collection就会将其回收

- Code(text): 存储了代码
- global： 全局变量和static变量

- Memory Systems



- Registers： 非常接近CPU, 但是空间很少
- Caches: 速度比Registers稍慢，但是空间要大一些
- Internal Memory： main memory/ core memory
- External Memory: consists of disks, CD drives, DVD drives and/or tapes.(very large)
- Network Storage: 非常大的存储空间， 但也是最慢的