Algorithm in coursera

Week 3

merge sort and quick sort

Interview questions

1. Merging with smaller auxiliary array. Suppose that the subarray **a[0]**$a[0]$ to **a[n − 1]**$a[n-1]$ is sorted and the subarray **a[n]**$a[n]$ to **a[2 ∗ n − 1]**$a[2*n-1]$ is sorted. How can you merge the two subarrays so that **a[0]**$a[0]$ to **a[2 ∗ n − 1]**$a[2*n-1]$ is sorted using an auxiliary array of length nn (instead of 2n2n)?

```
copy only the left half into the auxiliary array
```

2. Shuffling a linked list

```
design a linear-time subroutine that can take two uniformly shuffled linked
lists of sizes n1 and n2 and combined them into a uniformly shuffled linked
lists of size n1 + n2
```

Assignment3: Patter Recognition

> Computer vision: Computer vision involves analyzing patterns in visual images and reconstructing the real-world objects that produced them. The process is often broken up into two phases: feature detection and pattern recognition. Feature detection involves selecting important features of the image; pattern recognition involves discovering patterns in the features. We will investigate a particularly clean pattern recognition problem involving points and line segments. This kind of pattern recognition arises in many other applications such as statistical data analysis.

[The question]

- How to find the maximum line(nunmber of points on a line >= 5) in Fast program?

  > Only add the linesegment when the original point is less than other points

---

The code

- Point
- Brute
- Fast

Week 3

Quick sort

- Basic plan
  - recursion before the sort (merge is recursion after the sort)
  - partition (put an entry to a right place)
  - sort piece of front of the entry and tail of the entry recursively
- Step (choose the first one of the array as the entry)
  - i from left+1 to right, j from right to left + 1
  - until arr[i]>arr[left] && arr[j]<arr[left]
  - exchange arr[i] & arr[j]
  - repeat 1-3 until j < i

```
for(int i = left + 1, j = right; i < = j; ){

  if(arr[i] <= arr[left])
    i++ ;
  if(arr[j] > arr[left])
    j-- ;
  if(arr[i] > arr[left]&&arr[j]<=arr[left]){
    swap(arr[i],arr[j]);
  }
}
```

- Notice
  - stay in the bound (test needed):
  - [] (j==left)
  - [x] (i==right)
  - Shuffle the array before sort: for performance(quadratic case)
  - Equal keys: when duplicates are present, it is better to stop on keys equals to the partitioning item's key
- Properties
  - in-place:

    > Partition: constant extra place Depth of recursion
    >
    > Depth of recursion: logarithmic extra space

  - not stable

- improvements
    - Insertion of small arrays(10 items)
    - median of sample: choose the median as a entry(swap median and left)

Selection

- Goal: Given an array of N items, find the k-th largest
- Use theory as a guide:
    - Upper bound: NlogN
    - Upper bound for k = 1 ,2 ,3.:N How?

> go through the array

-
    - Lower bound : N Why?
- Quick selection

> Partition array until j == the k order you want (again shuffle before the partition)

- proposition: Linear time on average

Duplicate Keys

- large array but small # of key values
- Mergesort : Always between 1/2*NlgN and NlgN compares
- QuickSort: quadratic times unless patition stops at equal keys
    - recommend: stops scans on items equal to the partitioning item
- 3-way partitioning
    - Entries between lt and gt: equal

```
v=a[lo];lt=lo;gt=hi;i=lo+1;
(a[i]<v) swap(a[lt++],a[i++]]);
(a[i]>v) swap(a[gt--],a[i]]);
(a[i]==v) i++;
until i>= gt
```

System sort

- java system sorts: Array.sort();
    - different methods for each primitive type
    - a method for data types that implements Comparable
    - a method uses COmparator

  - Uses tuned quicksort fpr primitive types, tuned mergesort for objects(stable, space is not that important)
  - import java.util.arrays
- Tukey's ninther

  > small arrays: middle entry
  >
  > medium arrays: median of 3(3 ways partitioning)
  >
  > large arrays: Tukey's ninther

  - pick 9 items out of the array
  - take the median of the mediums as a ninther

  ```
  9 samples: RAM GXK BJE

  medians: MKE

  ninther: K
  ```

- Interview questions:
- Decimal dominants. Given an array with nn keys, design an algorithm to find all values that occur more than n/10n/10 times. The expected running time of your algorithm should be linear.

  ```
  Hint: determine the (n/10)th largest key using quickselect and check if it occurs more than n/10 times.

  Alternate solution hint: use 9 counters.
  ```

- Selection in sorted array: 2 arrays a[] b[], size n1, n2 find kth largest key. log(n1+n2) algorithm

  ```
  Approach A: Compute the median in a[] and the median in b[]. Recur in a subproblem of roughly half the size.

  Approach B: Design a constant-time algorithm to determine whether a[i] is the kth largest key. Use this subroutine and binary search.
  ```

Week 4

1. Priority Queues

APIs and Elemnentary Implementations

- Collections: Insert and delete items

> Stack: Remove the items most recently added
>
> Queue: Remove the items least recently added
>
> Randomized Queue: Remove a random item
>
> Priority queue: Remove the largest/smallest item

- Find largest M items in a stream of N items(large M frauds in N transactions)

```java
MinPQ<Transaction> pq = new MinPQ<Transaction>();
while(StdIn.hasNextLine()){
    String line = StdIn.readLine();
    Transaction item = new Transaction(Line);
    pq.insert(item);
    //delete items not topM
    if(pq.size()>M)
      pq.delMin();
}
```

implementation	time	space
sort	N logN	N
elementary PQ	M N	M
binary heap	N log M	M
best in theory	N	M

- unordered array implementation

```java
public class UnorderedMaxPQ<Key extends Comparable<Key>>{//key is comparable
    private Key[] pq;
    private int N;
    public UnorderedMaxPQ (int capacity){
        pq = (Key[]) new Comparable[capacity];
    }
    public boolean isEmpty(){
        return N==0;
    }
    public void insert(Key x){
        pq[N++] = x;
    }
    public Key delMax(){
```

```
            int max = 0;
            for(int i=1;i<n;i++)
                if(less(max,i)) max = i;
            //put to the end return and reduce the size
            swap(max,N-1);
            return pq[--N];
        }
        public boolean less(int x,int y){
            return pq[x].toCompare(pq[y])<0;
        }
        public void swap(int x,int y){
            Key temp = pq[x];
            pq[x] = pq[y];
            pq[y] = temp;
        }
}
```

impelementation	insert	del max	max
unoredered array	1	N	N
ordered array	N	1	1
goal	logN	logN	logN

- Use binary search to insert a key into ordered array, but when shifting we still need linearly array accesses.

Binary Heaps

- Complete binary tree
  - Binary tree: empty or node with links to left and right binary trees (only two branches for each nodes)
  - Complete tree: perfectly balanced, except for bottom level (except the bottom level, all of the level must be full)
  - Property: Height of complete tree with N nodes is [lg N]
  - Pf: Height only increases when N is a power of 2
- Binary heap: array representation of a heap-ordered complete binary tree
  - keys in nodes
  - parent's key no smaller than children's keys
- Array representation
  - Indices start at 1(the largest key)

    - Takes nodes in level order
    - No explicit links needed
    - Parent of node at k: k/2
    - Children of node at k: 2k and 2k + 1
- Scenario: Child's key becomes larger key than its parent's key
    - exchange key in child with key in parent
    - repeat until heap order restored

```
private void swim(int k){
    while(k>1 && less(k/2,k)){
        swap(k,k/2);
        k=k/2;
    }
}
// logN+1 compares
public void insert(Key x){
    pq[++N]=x;
    swim(N);
}
```

- Scenario: parent is smaller than its child
    - exchange key with larger child

```
private void sink(int k){
    while(2*k<=N){
        int j = 2*k;
        // find max child
        if(j<N && less(j,j+1)) j++; //here are N entries, start from 1 so N is the last
        if(!less(k,j)) break;
        swap(k,j);
        //check next
        k = j;
    }
}
public Key delMax(){
    Key max = pq[1];
    swap(1,N--);
    pq[N+1] = null;
    sink(1);
    return max;
}
```

```
public class MaxPQ<Key extends Comparable<Key>>{
    private Key[] pq;
```

```java
    private int N;
    public MaxPQ(int capacity){
        pq = (Key[]) new Comparable[capacity+1];
    }
    public boolean isEmpty(){return N==0;}
    public void insert(Key key);
    public Key delMax();
    public void swim(int k);
    public void sink(int k);
     public boolean less(int x,int y){
        return pq[x].toCompare(pq[y])<0;
    }
    public void swap(int x,int y){
        Key temp = pq[x];
        pq[x] = pq[y];
        pq[y] = temp;
    }
}
```

- Immutability of keys
    - Assumption: client doesn't change keys while they're on the PQ
    - Best practice: use immutable keys
    - Data type: set of values and operations on those values
    - Immutable data type: Cannot change the data type value once created (String Integer Double Color Vector...)

    ```java
    public final class Vector{
        private final int N;
        private final double[] data;
        public Vector(double[] data){
            this.N = data.length;
            this.data = new double[N];
            for(int i=0;i<N;i++)
                this.data[i]=data[i];
        }
    }
    ```

- Underflow and overflow(exception)
    - throw exception if deleting from emty
    - add no-arg constructor and use resizing array
- Minimum-oriented priority queue
    - Replace less() with greater()

- Implement greater()
- Other operations
  - Remove an arbitrary item
  - change the priority of an item //sink() and swim()

Heapsort

- Basic plan for in-place sort
  - Create max-heap with all N-keys
  - Repeatedly remove the maximum key
- Heap construction
  - Build max heap using bottom-up method (check all the 3-nodes heap from bottom)
- remoce maximum
  - swap(1, N--) then the largest one to the tail of the heap (array)

```
public static void sort(Comparable[] pq){
  int N = pq.length;
  //from the second last layer to sink make the heap in order
  for(int k = N/2; k >= 1; k--)
      sink(arr,k,N);
  // remove the maximum, one at a time
  while(N>1){
      swap(arr,1,N--);
      sink(arr,1,N);
  }
}
// N*logN(<=2N compares and exchanges in heap construction,<= 2NlogN comparaes and exchanges for heapsort) In-place sorting with NlogN worst case
```

- bottom line (heapsort is not often used)
  - Inner loop longer than quicksort's
  - Makes poor use of cache memory(高速缓存)
  - not stable

Event-Driven Simulation

- Goal: simulate the motion of N moving particles that behave according to the laws of elastic collision
- Hard disc model
  - moving particles via elastic collisions with each other and walls
  - each particle is a disc(圆盘) with known position, velocity, mass and radius
  - No other forces

```
public class BouncingBalls{
```

```java
    public static void main(String[] args){
    int N = Interger.parceInt(arg[0]);
        Ball[] balls = new Ball[N];
        for(int i=0;i<N;i++)
            balls[i]=new Ball();
        while(true){
            StdDraw.clear();
            for(int i=0;i<N;i++){
                balls[i].move(0.5);
                balls[i].draw();
            }
            StdDraw.show(50);
        }
    }
}
public class Ball{
    private double rx,ry;
    private double vx,vy;
    private final double radius;
    public Ball(){/*initialize position and velocity*/}
    public void move(double dt){
        //check collision with walls
        if((rx+vx*dt<radius)||(rx+vx*dt>1.0-radius)){vx=-vx;}
        if((ry+vy*dt<radius)||(ry+vy*dt>1.0-radius)){vy=-vy;}
        rx+=(vx*dt);
        ry+=(vy*dt);
    }
    public void draw(){
        StdDraw.filledCircle(rx,ry,radius);
    }
}
```

- Missing : collision between each other
- Time-driven simulation
  - Discretize time in quanta of size ***dt***
  - Update the position of each particle after every ***dt*** units of time and check overlap
  - If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation
  - Drawbacks: quadratic slow
- Even-driven (Change state only when something happened)
  - Between collisions, particles move in straight-line trajectories(弹道)
  - Focus only on times when collisions occur
  - Maintain PQ of collision events, prioritized by time
  - Remove the min = get next collision

> Prediction: at time ***t*** use position velocity and radius

> Resolution: at time ***t + dt*** change the veocity

```java
public class Particle{
    private double rx,ry;
    private double vx,vy;
    private final double radius;
    private final double mass;
    private int count;
    public Particle(...){}
    public void move(double dt){}
    public void draw(){}
    //predict collision with particle or wall
    public double timeToHit(Particle that){}
    public double timeToHitVerticalWall(){}
    public double timeToHitHorizontalWall(){}
    //resolve collision with particle or wall
    public void bounceOff(Particle that){}
    public void bounceOffVerticalWall(){}
    public void bounceOffHorizontalWall(){}
}
```

- Initialization
  - Fill PQ with all potential particle-wall collisions
  - Fill PQ with all potential particle-particle collisions
- Main loop
  - delete the impending(即将到来的) event from PQ
  - If the event has been invalidated, ignore it
  - Advance all particles to time t, on a straight-line trajectory
  - Update the velocities of the colliding particles
  - Predict future particle-wall and particle-particle collisions involving the colliding particles and insert events onto PQ

```java
private class Event implements Comparable<Event>{
    private double time;
    private Particle a,b;
    private int countA,countB;
    public Event(double t,Particle a,Particle b){}
    public int comparaTo(Event that){
        return this.time-that.time;
    }
    public boolean isValid(){}
}
public class CollisionSystem{
    private MinPQ<Event> pq;
    private double t = 0.0;
    private Particle[] particles;
    public CollisionSystem(Particle[] particles){}
```

```java
    private void predict(Particle a){
        if(a == null) return;
        for(int i=0;i<N;i++){
      double dt = a.timeToHit(particles[i]);
            pq.insert(new Event(t+dt,a,particles[i]));
        }
        pq.insert(new Event(t+a.timeToHitVerticalWall()),a,null);
        pq.insert(new Event(t+a.timeToHitHorizontalWall()),null,a);
    }
    private void redraw(){}
    public void simulate(){
        pq = new MinPQ<Event>();
        for(int i=0;i<N;i++)predict(particles[i]);
        pq.insert(new Event(0,null,null));
        while(!pq.isEmpty()){
            Event event = pq.delMin();
            if(!event.isValid())continue;
            Particle a = event.a;
            Particle b = event.b;
            for(int i=0;i<N;i++)
                particles[i].move(event.time-t)
            t=event.time;
            if(a!=null&&b!=null)a.bounceOff(b);
            else if(a!=null&&b==null)a.bounceOffVerticalWall();
            else if(a==null&&b!=null)a.bounceOffHorizontalWall();
            else if(a==null&&b==null)redraw();
            //predict new event
            predict(a);
            predict(b);
        }
    }
}
```

Interview questions

- Dynamic median(insert and remove in logarithmic, find the median in constant)
    - two heap one is max-oriented another is min-oriented
- Taxocab numbers: find all numbers can be represneted in two forms of cubic: a^3+b^3=c^3+d^3=n
    - form the sums a^3+b^3 and sort
    - use a min-oriented priority queue with n items

8 Puzzle

2. Elementary symbol tables

Symbol tables API

(1) Key - value pair abstraction

- Insert a value with specified key
- Given a key, search for the corresponding value.

```
public ckass ST<Key,Value>{
    ST(); //create a symbol table
    void put(Key key, Value val); //put key-value pair into the table like arr[key] = value
    Value get(Key key); //like int val = arr[key]
    void delete(Key key){
        put(key,null);
        size--;
    }
    boolean contains(Key key){
        return get(key)!=null;
    }
    boolean isEmpty();
    int size();
    Iterable<Key> keys();
}
```

(2) Conventions

- Values are not `null`
- Method `get()` returns `null` if key not present
- Method `put()` overwrites old value with new value

(3) Keys and Values

- Values: Any generic type
- Keys:
  - Assume keys are `Comparable` , use `compareTo()`.
    - more efficient implements by using the ordering of the keys to find ways around the data structure
    - support broader symbol table operations
  - Assume keys are any generic type, use `eauals()` to test equality

- Assume keys are any generic type, use `eauals()` to test equality, use `hashCode()` to test scarmble(争夺) key.

- Best practices : immutable types for symbol table keys(String Integer Double java.io.File)

(4) Equality test

- Reflexive: x.equals(x) is true
- Symmetric: x.equals(y) iff y.equals(x)
- Transitive: if x.equals(y) and y.equals(z), then x.equals(z)
- Non-null: x.equals(null) is false

```
public class Date implements Comparable<Date>{
  private final int month;
    private final int day;
    private final int year;
    public boolean equals(Object y){
        if(y==this) return true;  //if the reference is the same
        if(y==null) return false; //if y is null
        if(y.getClass()!=this.getClass()) //same class?
            return false;
        Date that = (Date) y;
        return (this.day==that.day)&&(this.month==that.month)&&
(this.year==that.year);
    }
}
```

(5) "Standard" recipe for user-defined types

- Optimization for reference equality (use `getClass()`)
- Check against null
- Check two objects are of the same type and cast
- Compare each significant field

> if field is a primitive type use ==
>
> If field is an object, use equals()
>
> if field is an array, apply to each entry use `Arrays.equals(a,b)` or `Array.deepEquals(a,b)`

- Best practices
  - No need to use calculated fields(dependent fields)
  - Compare fields most likely to differ first
  - Make `compareTo()` consistent with `equals()` (use first one if it is a Comparable type)

```
// example of find most frequent word using Symbol table
public class FrequencyCounter{
    public static void main(String[] args){
        int minlen = Integer.parseInt(args[0]);
```

```java
        ST<String, Integer> st = new ST<String,Integer>();
        while(!StdIn.isEmpty()){
            String word = StdIn.readString();
            if(word.length()<minlen)continue;
            if(!st.contains(word))st.put(word,1);
            else st.put(word,st.get(word)+1);
        }
        String max="";
        st.put(max,0);
        for(String word:st.keys())
            if(st.get(word)>st.get(max))
                max=word;
        StdOut.println(max+" "+st.get(max));
    }
}
```

Elementary Implementations

- Linked list
  - Data structure: linked list with key-value pairs
  - Search: Scan through all keys until find a match
  - Insert: Scan through all keys until find a match; if no add to front
  - for keys only have interface `equals()`
  - Linearly time
- Binary Search
  - ordered array
  - for keys have interface `compareTo()`;
  - Log search and linear insert

```java
private Key[] keys;
private Value[] vals;
private int N;
public Value get(Key key){
    if(isEmpty())return null;
    int i = rank(key);
    if(i<N && keys[i].compareTo(key)==0) return vals[i];
    else return null;
}
private int rank(Key key){
    int left = 0, right = N - 1;
    while(left<=right){
        int mid = (left+right)/2;
        int cmp = key.compareTo(keys[mid]);
        if(cmp<0) right=mid-1;
        else if(cmp>0) left = mid + 1;
        else return mid;
```

```
    }
    return left;
}
public void insert(Key key,Value value){
    if(isEmpty) {
        keys[N++]=key;
        vals[N++]=value;
    }
    else if(get(key)==null){
      int i = rank(key);
        for(int j = N-1;j>=i;j--){
            vals[j+1]=vals[j];
            keys[j+1]=keys[j];
        }
        vals[i]=value;
        keys[i]=key;
    }
    else{
        int i = rank(key);
        vals[i]=value;
    }
}
```

Ordered symbol table

API

```
public class ST<Key extends Comparable<Key>,Value>{
    ST
    //...
    Key min();
    Key max();
    Key floor(Key key);
    Key ceiling(Key key);

}
```

Binary Search Trees

(1) Defination

- Binary Heap: implicit representation of trees with an array
- Binary search tree: explicit binary search tree in symmetric order
- ***Symmetric*** order: each node has a key, and every node's key is:
    - **Larger** than all keys in its left subtree
    - **Smaller** than all keys in its right subtree

- Java: A BST is a reference to a root Node
- A Node is comprised of four fields:
  - A Key and a Value
  - A reference to the left and right subtree

```java
public class BST<Key extends Comparable<key>,Value>{
    private Node root;  //root of BST
  private class Node{
      private Key key;
      private Value val;
      private Node left,right;
      public Node(Key key,Value val) {
          this.key = key;
          this.val=val;
      }
  }

    /*public void put(Key key,Value val){
    Node x = root;
        while(x!=null){
            int cmp = key.compareTo(x.key);
            if(cmp<0) x = x.left;
            else if(cmp>0) x = x.right;
            else x.val = val;
        }
        if(x==null) x = new Node(key,val);
    }*/

    //recursive implementation
    public void put(Key key,Value val){
        root = put(root,key,val);
    }
    private Node put(Node x,Key key,Value val){
        if(x==null) return new Node(key,val);//for the new node
        int cmp = key.compareTo(x.key);
        if(cmp<0)
            x.left = put(x.left,key,val);
        else if(cmp>0)
            x.right = put(x.right,key,val);
        else
            x.val = val;
        return x; //every time return the processed same node
    }

    public Value get(Key key){
        Node x = root;
        while(x!=null){
            int cmp = key.compareTo(x.key);
```

```
                if(cmp<0) x = x.left;
                else if(cmp>0) x = x.right;
                else return x.val;
            }
            return null;
        }

        public void delete(Key key){

        }

        public Iterable<Key> iterator(){

        }
    }

```

(2) mathematical analysis

- Proposition: If N distinct keys are inserted into a BST in random order, the expected number of compares for a search/insert is ~2 lnN
- Pf: 1-1 correspondence with quicksort partitioning
- But for ordered key the time would be linear

(3) other methods

- Min: `while(x!=null) x=x.left;`
- Max: `while(x!=null) x=x.right;`
- Floor: largest key <= given key
  - Case1: [key equals the key at root] -> the floor of k is k
  - Case2: [key is less than the key at root] -> The floor of k is in the left subtree
  - Case3: [key is larger than the key at root] -> The floor of k is in the right otherwise the root

```
public Key floor(Key key){
    Node x = floor(root,key);
    if(x==null) return null;
    return x.key;
}
private Node floor(Node x,Key key){
    if(x==null) return null;
    int cmp = key.compareTo(x.key);
    if(cmp==0) return x;
    if(cmp<0) return floor(x.left,key);
    // right otherwise the root
    Node t = floor(x.right, key);
    if(t!=null) return t;
    else    return x;
}
```

- Ceiling: smallest key >= given key
- `rank()`, `select()` what is the rank of a given key, what is the key of the given rank
- `size()` return the count at the root

```
private class Node{
    private Key key;
    private Value val;
    private Node left,right;
    private int count;
    public Node(Key key,Value val,int cnt) {
        this.key = key;
        this.val=val;
        count = cnt;
    }
    public int size(){
        return size(root);
    }
    private int size(Node x){
        if(x==null) return 0;
        return x.count;
    }
}
public void put(Key key,Value val){
  root = put(root,key,val);
}
private Node put(Node x,Key key,Value val){
  if(x==null) return new Node(key,val,1);//for the new node
    int cmp = key.compareTo(x.key);
    if(cmp<0)
       x.left = put(x.left,key,val);
    else if(cmp>0)
       x.right = put(x.right,key,val);
    else
       x.val = val;
    x.count = 1 + size(x.left) + size(x.right); //1 is the node itself
    return x; //every time return the processed same node
}
public int rank(Key key){
    return rank(key,root);
}
private int rank(Key key,Node x){
    if(x==null) return 0;
    int cmp = key.compareTo(x.key);
    if(cmp<0) return rank(key,x.left);
    //the current node and left nodes all less than the key
    else if(cmp>0) return 1 + size(x.left) + rank(key,x.right);
    else return size(x.left);
```

```
}
```

- Inorder traversal
  - Traverse left subtree
  - Enqueue key
  - Traverse right subtree

```java
public Iterable<Key> keys(){
    Queue<Key> q = new Queue<Key>();
    inorder(root,q);
    return q;
}
private void inorder(Node x, Queue<Key> q){
    if(x==null) return;
    inorder(x.left,q);
    q.enqueue(x.key);
    inorder(x.right,q);
}
```

- Delete key-value pairs from table
  - Set its value to null
  - Leave key in tree to guide searches (but don't consider it equal in search)
  - ~lnN -> memory overload
- Deleting the minimum
  - Go left until finding a node with a null left link
  - Replace that node by its right link
  - Update subtree counts

```java
public void deleteMin(){
    root = deleteMin(root); //root = root; x=x recursion pattern
}
private Node deleteMin(Node x){
    // if it is null replace by its right link
    if(x.left == null) return x.right;
    x.left = deleteMin(x.left);
    // recalculate the size of the BST
    x.count = 1+size(x.left)+size(x.right);
    return x;
}
```

- Hubbard deletion: To delete a node with key ***k*** : search for node ***t*** containing key ***k***
  - Case 0: [0 children] Delete ***t*** by setting parent link to null (`return null;`)

        - Case 1: [1 child] Delete ***t*** by replacing parent link (`return children;`)
        - Case 2: [2 children]
            - Find successor ***x*** of ***t***
            - Delete the minimum in t's right subtree
            - put ***x*** in ***t***'s spot (replace x by t)
            - i.e. swap x and the min of right subtree, delete x

```java
public void delete(Key key){
    root = delete(root,key);
}
private Node delete(Node x,Key key){
  if(x==null)return null;
    int cmp = key.compareTo(x.key);
    if(cmp<0) delete(x.left,key);
    else if(cmp>0) delete(x.right,key);
    else{
     //if it has only one child or zero child, or no child can also handle
        if(x.right==null) return x.left;
        else if(x.left==null) return x.right;
        Node t = x; //now t is the one should be deleted
        //find the minimum of the right tree
        x = min(t.right);
        //assign right of x as the right tree deleted x
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    x.count = size(x.left)+size(x.right)+1;
    return x;
}
```

        - Problem: lack of symmetric -> aways get right hand tree -> delete time $N^{\frac{1}{2}}$

---

week 5

---

1. Balanced Search Trees -- control logN symbol table operations

(1) 2-3 Search Trees

- Properties
    - Idea: Allow 1 or 2 keys per node
        - 2-node: one key, two children
        - 3-node: two keys, three children (1 child less,1 child between , 1 child larger than the two keys)
    - Perfect balance: Every path from root to null link has same length

    - Symmetric order: Inorder traversal yields keys in ascending order
- Search
    - compare search key against keys in node
    - find interval containing search key
    - Follow associated link (recursively)
- Insert
    - Insert into a 2-node at bottom
        - Search for key, as usual
        - Replace 2-node with 3-node
    - Insert into a 3-node at bottom
        - Add new key to 3-node to create temporary 4-node
        - Move middle key in 4-node into parent
        - Repeate until there are no 4-node
- Performance
    - Worst: lgN (all 2-nodes)
    - Best: log_3N
    - Between 12 and 20 for a million nodes
    - Between 18 and 30 for a billlion nodes.
    - Guaranteed logarithmic time

---

(2) Left-leaning Red-Black BSTs

- Property
    - Represent 2-3 tree as a BST
    - Use "internal" left-leaning links as "glue" for 3-nodes
    - No node has two red links connected to it
    - Every path from root to null link has the same number of black links
    - Red links lean left
- **Search** is the same as for elementary BST

    > Because it is more balanced it will be faster

```
public Val get(Key key){
    Node x = root;
    while(x!=null){
        int cmp = key.compareTo(x.key);
        if(cmp<0)      x = x.left;
        else if(cmp>0)  x = x.right;
        else       return x.val;
    }
    return null;
}
```

- Implementation

```java
private static final boolean RED = true;
private static final boolean BLACK = false;
private class Node{
    Key key;
    Value val;
    Node left,right;
    boolean color;  //color represent the link from its parent to it
}
private boolean isRed(Node x){
    if(x!=null) return false;
    return x.color == RED;
}
```

- **Left(right) rotation**: Orient a (temporarily) right-leaning red link to lean left (when insertion we need to rotate it right and get it back)

> maintains symmetric order and perfect black balance

```java
private Node rotateLeft(Node h){
    assert isRed(h.right);
    //store x and it will be the parent , h will be the left child
    Node x = h.right;
    // send the one between h and x(x.left) to h.right
    h.right = x.left;
    x.left = h;
    //keep the original color with parent
    x.color = h.color;
    // change color of h
    h.color = RED;
    return x;
}
private Node rotateRight(Node h){
    assert isRed(h.left);
    //store x and it will be the parent , h will be the left child
    Node x = h.left;
    // send the one between h and x(x.left) to h.right
    h.left = x.right;
    x.right = h;
    //keep the original color with parent
    x.color = h.color;
    // change color of h
    h.color = RED;
    return x;
}
```

- **Color flip** : Recolor to split a (temporary) 4-node.

```java
private void flipColors(Node h){
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

- **Insert**
  - Warmup 1: Insert into a tree with exactly 1 node
    - Left -> set the child color to be RED
    - Right -> set the child color to be RED then rotate left
  - Case 1: Insert into a 2-node at the bottom
    - Do standard BST insert; color new link red
    - if new red link is a right link, rotate left
  - Warmup 2: Insert into a tree with exactly 2 nodes
    - Larger than the 2 nodes -> attached new node with red link -> flipped to BLACK
    - Smaller than the 2 nodes -> attach to left of the child of the 2 nodes -> rotated the child of the 2 nodes right -> flipped to BLACK
    - Between -> attach to right of the child of the 2 nodes -> rotated the new node left and then right -> flipped to BLACK
  - Case 2: insert into a 3-node at the bottom
    - Do standard BST insert; color new link red
    - Rotate to balance the 4-node (if needed)
    - Flip colors to pass red link up one level
    - Rotate to make lean left (if needed)
    - Repeat case 1 or case 2 up the tree (if needed)
  - **Same code handles all cases**
    - Right child red, left child black: rotate left
    - Left child , left-left child both red: rotate right
    - Both cild red: flip colors

```java
private Node put(Node h,Key key, Value val){
    if(h == null) return new Node(key,val,RED);
    int cmp = key.compareTo(h.key);
    // standard BST insert
    if(cmp<0)   h.left = put(h.left,key,val);
    else if(cmp>0)  h.right = put(h.right,key,val);
    else      h.val = val;
    // rotate and flip -> only for recursive implementation
    if(!isRed(h.left)&&isRed(h.right))    h = rotateLeft(h);
    if(isRed(h.left)&&isRed(h.left.left)) h = rotateRight(h);
    if(isRed(h.right)&&isRed(h.left))   h = flipColors(h);
    return h;
}
```

---

(3) B-Trees

- Background -> file system
  - Page: Contiguous block of data
  - Probe: First access to a page(e.g. from disk to memory)
  - Property : Time required for a probe is much larger than time to access data within a page
  - Cost model: number of probes (find the first page)
  - Goal: Access data using minimum number of probes
- Generalize 2-3 trees by allowing up to M -1 key-link pairs(a lot of keys) per node (M is choosed by user,can be larger than 3 -> M-1,M tree)
  - At least 2 key-link pairs at root
  - At least M/2 key-link pairs in other node
  - if full split it
- Proposition
  - In a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes
  - Because all internal nodes have between M/2 and M-1 links
  - In practice: Number of probes is at most 4
  - Optimization: Always keep root page in memory
- Libray
  - Java: java.util.TreeMap, java.util.TreeSet
  - C++ STL: map, multimap, multiet

---

2. Geometirc Application of BSTs

---

(1) 1 d range search

- Extension of ordered symbol table
  - Range search: find all keys between k_1 and k_2
  - Range count: find number of keys between k_1 and k_2
- Geometrix interpretation
  - Keys are point on a line
  - Find/count points in a given 1 d interval
- Two elementary operation
  - Unordered array : fast insert , slow search
  - Ordered array : slow insert , binary search
- In BST

```
public int size(Key left,Key right){
    if(contains(right)) return rank(right)-rank(left)+1;
    else         return rank(right)-rank(left);
}
```

---

(2) Line segament intersection

- Given N horizontal and vertical line segments, find all intersections
- Method
  - Quadratic algorithm : check all pairs of line segments
  - Nondegeneracy assumption: All x- and y-coordinates are distinct
- Sweep-line algorithm : Sweep vertical line from left to right
  - x-coordinates define events
  - h-segment (hit left endpoint) : insert y-coordinate into BST
  - h-segment (hit right endpoint) : remove y-coordinate from BST
  - v-segment(hit vertical line): range search for interval of y-endpoints (search for the point in the range) **using method above**
- Time: NlogN + Rx
  - Put x-coordinates on a PQ(or sort): NlogN
  - Insert y-coordinates into BST : NlogN
  - Remove y-coordinates from BST :NlogN
  - Range searches in BST : NlogN + R(number of intersections)

---

(3) Kd-trees

a. backgroud and silly method

- Extension of ordered symbol table
  - Range search: find all keys in 2d range
  - Range count: find number of keys in to
- Geometrix interpretation

- Keys are point on a plane
- Find/count points in a given h-v rectangle

- First metod
  - Grid implementation
    - Divide space into M-by-M grid of squares
    - Create list of points contained in each square
    - Use 2d array to directly index relevant square
    - Insert: add(x,y) to list for corresponding square
    - Range search: examine only squares that intersect 2d range query
  - Space-time trade off
    - Space: $M^2 + N$
    - Time: $1 + N/M^2$ per square examined, on average
  - Choose M
    - To small: waste Time
    - To big: waste space
    - Rule of thumb: $M = \sqrt{N}$
  - Running time(Randomly distributed points)
    - Initialize data structure: N
    - Insert point: 1
    - Range search : 1 per point in range
  - Problem: clustering
    - Lists are too long, even though average length is short
    - Need data structure that adapts gracefully to data

b. Kd-trees

- 2d tree : recursively divide space into two halfplanes
  - divide the plane into two subplanes by using the vertical line through the points (left plane become left child)
  - similar but use horizontal line in the second interation (up plane become right child)
- Find point in rectangle
  - Goal: find all the points in a query axis-aligned rectangle
    - check if point in node lies in given rectangle
    - Recursively search left/bottom (if any could fall in rectangle)
    - Recursively search right/top(if any could fall in rectangle)
  - Cost
    - Typical case: R + logN
    - worst case : R + $\sqrt{N}$
- Find closest point to query point
  - Check distance from current point with query point
  - Recursively search left/bottom (if any could be closer)
    - first **go down** the point that is on the **same side** of the splitting line as the **query point** as, and compare with previous point and replace if smaller (To maintain the performance, same

side is more likely to be the answer)
    - when **go out** of the recursive loop compare with the whole vertical/horizontal line, to check is it possible to have a closer point in that area
  - Recursively search right/top(if any could be closer)
  - Organize method so that it begins by searching for query point
  - Cost
    - logN
    - Worst: N
- Kd-tree: recursively partition k-dimensional space into 2 half spaces
- N-body simulation
  - Simulate the motion of N particles, mutually affected by gravity
  - Brute force: For each pair of particles, compute force : $F = \frac{Gm_1m_2}{r^2}$

---

(4) Interval search

a. 1 d interval search

- Extension of symbol table
  - Insert an interval (left,right)
  - Search for an interval (left,right)
  - Delete an interval (left,right)
  - Interval intersection query: given an interval (left,right), find all intervals(or one interval) in data structure that intersects (left,right)
- API

```
public class IntervalST<Key extends Comparable<key>,Value>{
    public IntervalST();
    public void put(Key left,Key right,Value val);
    public Value get(Key left,Key right);
    public void delete(Key left,Key right);
    Iterable<Value> intersects(Key left,Key right);
}
```

- Interval search tree
  - insert
    - Use **left point** as BST key
    - Store the **largest end point** in subtree rooted at node (largest end point of the whole subtree, **it is for the search of intersect -> if largest one is smaller than the targets' left, it won't be checked**)
  - Search : To search for **any one** interal that intersects query interval (left,right)
    - if interval in node intersects query interval, return it
    - else if left subtree is null, go right
    - else if max end point in left subtree is less than *left*, go right

```java
Node x = root;
while(x!=null){
    if     (x.interval.intersects(left,right)) return x.interval;
    else if (x.left==null||x.left.max<lo)   x = x.right;
    else                          x = x.left;
}
return null;
```

- Proof:
  - If search goes right, then no itersectionb in left (max<$left_{target}$)
    - If search goes left, then there is either an intersection in left subtree or no intersections in either (当目标区间卡在左边子树的某两个区间之间时，无解) ->(此时$right_{target} < c$)
    - Suppose no intersection in the left
    - Since went leftm we have $left_{target}$<= max
    - Then for any interval (a,b) in right subtree of x.

      $right_{target} < c <= a$ (where c is left value of interval (c,max) who provide the max end point)
- Implementation: Use a red-black BST to mentain performance

---

(5) Rectangle Intersection

- Background
  - Goal: Find all intersections among a set of N orthogonal rectangles
  - Quadratic algorithm: Check all pairs of rectangles for intersection
  - Non-degeneracy assumption : all x- and y- coordinates are distinct
- Sweep-line algorithm
  - x-coordinates of left and right endpoints define events (when hit add or move the y-interval)
  - Maintain set of rectangles that intersect the weep line in an interval search tree (using y-interval of rectangle)
  - Left endpoint: interval search for y-interval of rectangle; insert y-interval
  - right endpoint: remove y-interval

Week6

---

1. Hash Table

---

(1) Hash table

- To Begin with
  - Basic idea : Save items in a key-indexed table (use an index as the key of an array) (index is a function of the key).
  - Hash function : Method for computing array index from key.
- Issue:
  - Computing the hash function
  - Equality test: Method for checking whether two keys are equal
  - Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index
- Classic space-time tradeoff
  - No space limitation : trivial hash function with key as index
  - No time limitation : trival collision resolution with sequential search
  - Space and time limitation : hashing (the real world)
- Requirement of hashfunction
  - Efficiently computable
  - Each table index **equally likely** for each key
- Java's hash code conventions : all java classes inherit a method `hashcode()` , which returns a 32-bit `int`.
  - Requirement: If `x.equals(y)`,then `(x.hashCode()==y.hashcode())` (反过来可能不行)
  - Highly desirable:(for collision) If `!x.equals(y)`,then `(x.hashCode()!=y.hashcode())`
  - Default implementation : Memory address of x.
  - Legal (but poor) implementation : Always return 17

```
// method for String
public int hashcode(){
  int hash = 0;
    for(int i=0;i<length();i++)
        hash = s[i] + (31*hash);
    return hash;
}
//optimization : Cache the hash value in an instance variable return the cached value (String is immutable)
private int hash = 0;
public int hashcode(){
    int h = hash;
    if(h!=0)return h; // compute only one time
    for(int i=0;i<length();i++)
        h = s[i] + (31*h);
    hash = h;
    return h;
}

// method for a self defined type
```

```java
public final class Transaction implements Comparable<Transaction>{
    private final String who;
    private final Date when;
    private final double amount;
    public int hashCode(){
    int hash = 17 ;// some nonzero constant
        hash = 31*hash + who.hashCode();
        hash = 31*hash + when.hashCode();
        hash = 31*hash + (Double) amount.hashCode();
        return hash;
    }
}
```

- "Standard" recipe for user-defined types
  - Combine each significant field using the $31x + y$ rule
  - If field is a primitive type, use wrapper type `hashcode()`
  - if field is null, return 0
  - if field is a reference type, use `hashCode()`. (apply rule recursibely)
  - if field is an array, apply to each entry
  - In theory : Keys are bitstring; "universal" hash functions exist
- Modular hashing
  - Hash code. An `int` between -2^{31} and $2^{31}-1$
  - Hash function. An `int` between 0 and M-1(for array)

```java
private int hash(Key key){
    return (key.hashCode()&0x7fffffff)%M;
}
```

- Uniform hashing assumption: Each key is equally likely to hash to an integer between 0 and M-1 (throw N balls into M bins)

(2) Separate chaining

- Collisions : Two distinct keys hashing to same index
  - Birthday problem -> can't avoid collisions unless you have a ridiculous amount of memory
  - Coupon collector + load balancing -> collisions will be evenly distributed (均匀分布)
  - Challenge: Deal with collisions efficiently
- Separate chaining symbol table
  - Hash: map key to integer *i* between 0 and M-1
  - Insert: put at front of i^{th} chain (if not already there)
  - Search: need to search only i^{th} chain

```java
public class SeparateChainingHashST<Key,Value>{
  private int M = 97;          //number of chains
    private Node[] st = new Node[M];  //array of chains
    private static class Node{
```

```
            private Object key;
            private Object val;
            private Node next;
        }
        private int hash(Key key){
            return (key.hashCode()&0x7fffffff)%M;
        }
        public Value get(Key key){
            int i = hash(key);
            for(Node x = st[i];x!=null;x=x.next)
                if(key.equals(x.key)) return x.val;
        }
        public void put(Key key,Value val){
        int i = hash(key);
            for(Node x = st[i];x!=null;x = x.next)
                if(key.equals(x.key)){x.val = val;return;}
            st[i] = new Node(key,val,st[i]);
        }
}
```

- Analysis
    - The number of keys in a list is within a constant factor ofd N/M is extremely close to 1
    - Consequence: Number of probes for search/insert is proportional to N/M
        - M too large -> too many empty chains
        - M too small -> too long to search
        - Typical choice: array resizing

(3) Linear probing (探测)

- Collision resolution : open addressing

> When a new key collides, find next empty slot, and put it there

- Linear probing hash table demo
    - **Hash**: Map key to integer i between 0 and M-1
    - **Insert**: Put at table index i if free; if not try i+1, i+2 (if reaches the end go back to head of the array) etc
    - **Search**: similar as insert
    - Note: Array size M *must be* greater than number of key-value pairs N

```
public class LinearProbingHashST<Key,value>{
    private int M = 30001;
    private Value[] vals = (Value[]) new Object[M];
    private Key[] keys = (Key[]) new Object[M];
    private int hash(Key key{return(key.hashCode()&0x7fffffff)%M;}
    public void put(Key key,Value val){
    int i;
```

```
        for(i=hash(key);keys[i]!=null;i=(i+1)%M)
            if(keys[i].equals(key))
                break;
        keys[i] = key;
        vals[i] = val;
    }
    public Value get(Key key){
    for(int i=hash(key);keys[i]!=null;i=(i+1)%M)
            if(keys[i].equal(key))
                return vals[i];
        return null;
    }
}
```

- Clustering
    - Cluster : A contiguous block of items
    - Observation: New keys likely to hash into middle of big clusters
- Lnuth's parking problem
    - Model: Cars arrive at one-way street with M parking spaces (Each desire a random space i: if i is taken, try i+1,i+2,etc)
    - Question : what is mean displacement of a car
    - Half-full: with M/2 cars, mean displacement is ~ 3/2
    - Full: with M cars, mean displacement is ~ $\sqrt{\frac{\pi M}{8}}$
    - Method: use resizing array

(4) Context

- War story: String hashing in Java
    - String `hashCode()` in Java 1.1
    - Benefit: saves time in performing arithmetic

```
public int hashCode(){
    int hash = 0;
    int skip = Math.max(1,length()/8);
    for(int i=0;i<length;i+=skip)
        hash = s[i]+(37*hash);
    return hash;
}
```

- Separate chaining vs. Linear probing
    - Separate chaining
        - Easier to implement delete
        - Performance degrades gracefully
        - Clustering less sensitive to poorly-designed hash function
    - Linear probing

        - less wasted space
        - Better cache performance
    - Improved version
        - Two-probe hashing (separate-chaining variant)
            - Hash to two positions （two hash function），insert key in shorter of the two chains
            - Reduce expected length of the longest chain to $loglogN$
        - Double hasing (linear-probing variant)
            - Use linear probing, but skip a variable amount, not just 1 each time
            - Effectively eliminates clustering
            - Can allow table to become nearly full
            - More difficult to implement delete
        - Cukoo hashing (linear-probing variant)
            - Hash key to two positions (two hash function); insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur)
    - Hash tables vs balanced search tree
        - Simpler to code
        - No effective alternative for unordered key
        - Faster for simple keys
        - Better system support in java for strings
    - Balanced search tees
        - Stronger performance guarantee
        - SUpport for ordered ST operations
        - Easier to implement `compareTO()` correctly than `equals()` and `hashCode()`.

---

2. Symbol table Applications

---

(1) Sets

- Mathematical set: A collection of distinct keys
- Remove "Value" of any implementations

```java
public class SET<Key extends Comparable<Key>>{
  SET();
    void add(Key key);
    boolean contains(Key key);
    void remove(Key key);
    int size();
    Iterator<Key> iterator();
}
```

- Exception filter
    - Read in a list of words from one file
    - Print out all word from standard input that are {in , not in } the list

```
public class WhiteList{
  public static void main(String[] args){
    SET<String> set = new SET<String>();
        In in = new In(args[0]);
        while(!in.isEmpty())
            set.add(in.readString());
        while(!StdIn.isEmpty()){
      String word = StdIn.readString();
            if(set.contains(word))
                StdOut.println(word);
        }
    }
}
```

(2) Dictionary Clients

- Dictionary lookup
    - A comma-separated value (CSV) file (URL and IP address pair)
    - Key field
    - Value field

```
public class LookupCSV{
    public static void main(String[] args){
    In in = new In(args[0]);
        int keyField = Integer.parseInt(args[1]);
        int valField = Integer.parseInt(args[2]);
        ST<String,String> st = new ST<String,String>();
        while(!in.isEmpty()){
      String line = in.readLine();
            String[] tokens = line.split(",");
            String key = tokens[keyField];
            String val = tokens[valField];
            st.put(key,val);
        }
        while(!StdIn.isEmpty()){
            String s = StdIn.readString();
            if(!st.contains(s)) StdOut.println("Not found");
            else        StdOut.println(st.get(s));
        }
    }
}
```

(3) Indexing Clients

- File indexing : Use search key to get associating information (搜索引擎)
- Goal: Given a list of files specified, create an index so that you can efficiently find all files containing a given query string
- Solution : Key = query string; value = set of files containing that string

```
import java.io.File;
public class FileIndex{
    public static void main(String[] args){
    ST<String,SET<File>> st = new ST<String, SET<File>>();
        for(String filename:args){
      File file = new File(filename);
            In in = new In(file);
            while(!in.isEmpty){
                String key = in.readString();
                if(!st.contains(key))
                    st.put(key,new SET<File>());
                st.get(key).add(file);
            }
        }
        while(!StdIn.isEmpty()){
      String query = StdIn.readString();
            StdOut.println(st.get(query));
        }
    }
}
```

- Book index(Index for an e-book)
    - Preprocess a text corpus to support concordance queries: given a word, find all occurrences with their immediate contexts

(4) Sparse Vectors

- Matrix-vector multiplication (standard implementation)

```
double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
for(int i=0;i<N;i++){
  sum = 0.0;
    for(int j = 0; j < N; j++)
        sum += a[i][j]*x[j];
    b[i] = sum;
}
```

- When there is a lot of 0 entries
    - x -> 1D array

- a -> Symbol table
  - Key = index, value = entry
  - Efficient iterator
  - Space proportional to number of nonzeros

```java
public class SparseVector{
    private HashST<Integer, Double> v;
    public SpareVector(){
        v = new HashST<Integer, Double>();
    }
    public void put(int i, double x){
        v.put(i,x);
    }
    public double get(int i){
    if(!v.contains(i)) return 0.0;
        else return v.get(i);
    }
    public Iterable<Integer> indices(){
        return v.keys();
    }
    public double dot(double[] that){
        double sum = 0.0;
        for(int i: indices())
            sum+=that[i]*this.get(i);
        return sum;
    }
}
```

Week7

---

1. Undirected graph

(1) Introduction

- Graph : Set of vertices connected pairwise by edges
- Graph-processing problems
  - Path : Is there a path between s and t
  - Shortest path : What is the shortest path between s and t
  - Cycle : Is there a cycle
  - Euler tour : Is there a cycle that uses each edge exactly once
  - Hamilton tour : Is thete a cycle that uses each vertex exactly once
  - Connectivily : Is there a way to connect all of the vertices
  - MST : What is the best way to connect all of the vertices
  - Biconnectivity : Is there a vertex whose removal disconnects the graph

- Planarity : Can you draw the graph in the plane with no corssing edges
- Graph isomorphism : Do two adjacency lists represent the same graph

(2) Graph API

- Graph representation
  - Graph drawing : Provides intuition about the structure of the graph
- Vertex representation
  - This lecture : use integers between 0 and V-1 (Allow use vertex indexed arrays)
  - Application : convert between names and intergers with symbol table

```
public class Graph{
    Graph(int V);
    Graph(In in){
    In in = new In(args[0]);  //create an empty graph with V vertices
        Graph G = new Graph(in);  //create a graph from input stream
        for(int v=0;v<G.V();v++)
            for(int w:G.adj(v))
                StdOut.println(v+"-"+w);
    }
    void addEdge(int v,int w);    //add an edge v-w
    Iterable<Integer> adj(int v); //vertices adjacent to v
    int V();            //number of vertices
    int E();            //number of edges
    String toString();        //string representation
    public static int degree(Graph G,int v){
        int degree = 0;
        for(int w:G.adj(v)) degree++;
        return degree;
    }
    public static int maxDegree(Graph G){
    int max = 0;
        for(int v=0;v<G.V();v++){
            int d = degree(G,v);
            if(d>max)
                max = d;
            return max;
        }
    }
    public static double averageDegree(Graph G){
        return 2.0*G.E()/G.V();
    }
    public static int numberOfSelfLoops(Graph G){
        int count = 0;
        for(int v=0; v < G.V(); v++)
            for(int w : G.adj(v))
                if(v==w) count++;
        return count/2;
```

```
    }
}
```

- Set-of-edges graph representation : maintain a list of the edges (linked list or array) => inefficient
- Adjacency-matrix graph representation
  - Maintain a two-dimentional V-by-V boolean array
  - for each edge $v - w$ in graph `adj[v][w]=adj[w][v]=true` (1 represent there is a connection)
- Adjacency-list graph representation : maitain vertex-indexed array of lists

```
public class Graph{
    private final int V;
    private Bag<Integer>[] adj; //adjacency lists(Using bag data type)
    public Graph(int V){
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for(int v = 0;v < V; v++)
            adj[v] = new Bag<Integer>();
    }
    public void addEdge(int v, int w){
        adj[v].add(w);
        adj[w].add(v);
    }
    public Iterable<Integer> adj(int v){
        return adj[v];
    }
}
```

- In practice : Use adjacency-lists representation
  - Algorithm based on iterating over vertices adjacent to v
  - Real-world graphs tend to be sparse (huge number of vertices, small average vertex degree)

(3) Depth-First Search

- Maze exploration
  - Vertex = intersection
  - Edge = passage
  - Goal : explore every intersection in the maze
- Tremaux maze exploration (avoid going to the same place twice)
  - Unrool a ball of string behind you
  - Mark each visited intersection and each visited passage
  - Retrace steps when no unvisited options
- Depth-first search
  - Goal : Systematically search through a graph
  - Idea : Mimic maze exploration
  - Typical applications

        - Find all vertices connected to a given sourse vertex
        - Find a path between two vertices
- Design pattern for graph processing => decouple(分离) graph data type from graph processing (easily to change implementation)
    - Create a Graph object
    - Pass the Graph to a graph-processing routine
    - Query the graph-processing routine for information

```java
public class Paths{
    Paths(Graph G, int s);
    boolean hasPathTo(int v);      //is there a path from s to v
    Iterable<Integer> pathTo(int v);  //all paths from s to v (if no return null)
}
```

- To visit a vertex v
    - Mark vertex v as visited
        - provide an array `marked[]` to represent vertex v is marked or not
        - provide a vertex indexed array of ints `edgeTo[v]` to store *where did you from*

```java
public class DepthFirstPaths{
    private boolean[] marked;
    private int[] edgeTo;
    private int s;
    public DepthFirstPaths(Graph G, int s){
        int num = G.V();
        marked = new boolean[num];
        edgeTo = new int[num];
        for(int i=0;i<num;i++)
            marked[i] = false;
        this.s = s;
        dfs(G,s);
    }
    private void dfs(Graph G, int v){
        marked[v] = true;
        for(int w : G.adj(v))
            if(!marked[w]){
                dfs(w);
                edgeTo[w] = v;
            }
    }
}
```

- Depth-first search properties
    - Proposition(主张， 命题)： DFS marks all vertices connected to s in time proportional to the sum of their degrees

- Pf
    - If w marked, then w connected to s
    - If w connected to s , then w marked
- Find the paths

```
public boolean hasPathTo(int v){
    return marked[v];
}
public Iterable<Integer> pathTo(int v){
    if(!hasPathTo(v)) return null;
    Stack<Integer> path = new Stack<Integer> ();
    for(int x = v; x != s; x = edgeTo[x])
        path.push(x);
    path.push(s);
    return path;
}
```

(4) Breadth-first search

- Repeat until queue is empty:
    - Remove vertex v from queue
    - Add to queue all unmarked vertices adjacent to v and mark them
- Compare
    - Depth-first search : Put unvisited vertices on a stack (using recursion)
    - Breadth-first search : Put unvisited vertices on a queue
    - Shortest path : Find path from s to t that uses fewest number of edges
- Implementation

```
public class BreadthFirstPaths{
    private boolean[] marked;
    private int[] edgeTo;
    private int[] dist;
    private void bfs(Graph G, int s){
        Queue<Integer> q = new Queue<Integer>();
        q.enqueue(s);
        marked[s] = true;
        dist[s]=0;
        while(!q.isEmpty()){
            int v = q.dequeue();
            for(int w:G.adj(v)){
                if(!marked[w]){
                    q.enqueue(w);
                    marked[w]=true;
                    edgeTo[w]=v;
                    dist[w] = dist[v]+1;
```

```
            }
        }
    }
}
```

(5) Connected Components

- Pre
  - Def :
    - Connect : Vertices v and w are connected if there is a path between them
    - Maximal set of connected vertices
  - Goal : Preprocess graph to answer queries of the form is v connected to w in constant time (union find can't fulfill)
  - Difference with union find : union find一边画图，一边计算； 后者被提供不变的图

```
public class CC{
    CC(Graph G);           //find connected components in G
    boolean connected(int v, int w);//are v and w connected
    int count();           //number of connected components
    int id(int v);           //component identifier for v
}
```

- Properties : The relation "is connected to" is an equivalence relation
  - Reflexive : v connected to v
  - Symmetric
  - Transitive
- Implementation
  - Goal : Partition vertices into connected components => int array id to make cluster

```
public class CC{
    private boolean marked[];
    private int[] id;
    private int count;

    public CC(Graph G){
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for(int v=0;v<G.V();v++){
            if(!marked[v]){
                dfs(G,v);
                count++;
            }
        }
```

```
    }
    public int count(){return count;}
    public int id(int v){return id[v];}
    private void dfs(Graph G,int v){
        marked[v] = true;
        id[v] = count;
        for(int w:G.adj(v))
            if(!marked[w])
                dfs(G,w);
    }
}
```

(6) Challenges

- Challenge 1 : Is a graph bipartite ?
  - Bipartite : divide vertices into two subsets that every connect one subset's point to another's
  - Use DFS
- Challenge 2 : Find a cycle
  - DFS
- Challenge 3 : Find a cycle use each edge once
  - Bridges of Konigsberg : Is there a cycle that uses each edge exactly once?
  - Answer : A connected graph is Eulerian iff all vertices have even degree
- Challenge 4 : Find a cycle that visits every vertex exactly once (Intractable)
- Challenge 5 : Are two graphs identical except for vertex names (we don't know)
- Challenge 6 : Lay out a graph in the plane without crossing edges (expert do this)

(7) Interview questions

- implement DFS without recursion => use stack
- Diameter and center of a tree
  - Diameter : pick a vertex s; run BFS from s; then run BFS again from the vertex that is furthest from s
  - Center : consider vertices on the longest path
- Euler circle (edge once)
  - A connected graph has an Euler cycle if and only if every vertex has even degree
  - Design a linear-time algorithm to determine whether a graph has an Euler cycle, and if so, find one
  - Hint : use DFS and piece together the cycles you discover

---

2. Directed Network

(1) Introduction to Digraphs

- Digraph : vertices connected by directed edges
- Problems
  - Path : Is there a directed path from s to t
  - Shortest Path : What is the shortest directed path form s to t
  - Topological sort : Can you draw a digraph so that all edges point upwards?
  - Strong connectivity : Is there a directed path between all pairs of vertices
  - Transitive closure : For which vertices v and w is there a path from v to w
  - PageRank : What is the importance of a web page
- Parallel edges : two or more edges connect two vertices

(2) Digraph API

```
public class Digraph{
    private final int V;
    private final Bag<Integer>[] adj;
    Digraph(int V){
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for(int i=0;i<V;v++)
            adj[i] = new Bag<Integer>();
    }
    Digraph(In in);
    void addEdge(int v,int w){
        adj[v].add(w);
    }
    Iterable<Integer> adj(int v){
        return adj[v];
    }
    int V();
    int E();
    Digraph reverse();        //reverse of this graph
    String toString();
}
```

- Maintain vertex-indexed array of lists

(3) Digraph search

- Reachability : Find all vertices reachable from s along a directed path
  - Same as undirected graph
  - DFS is a digraph algorithm
- Application
  - program control-flow analysis

- Mark-sweep garbage collector (Object(vertices) reference(edges))

- BFS in digraphs
  - Every undirected graph is a kind of digraph
  - BFS is a digraph algorithm
- Application
  - Crawl web, starting from some root web page (use implicit digraph)
    - Choose root web as source s
    - Maintain a Queue of websites to explore
    - Maintain a SET of discovered websites
    - Dequeue the next website and enqueue websites to which it links
    - why not DFS : new websites create new links, make it too deep

```java
Queue<String> queue = new Queue<String>();
SET<String> discovered = new SET<String>();
String root = "http://www.princeton.edu";
queue.enqueue(root);
discovered.add(root);
while(!queue.isEmpty()){
    String v = queue.dequeue();
    StdOut.println(v);

    In in = new In(v);
    String input = in.readAll();
    String regexp = "http://(\\w+\\.)*(\\w+)";
    Pattern pattern = Pattern.compile(regexp);
    Matcher matcher = pattern.matcher(input);

    while(matcher.find()){
        String w = matcher.group();
        if(!discovered.contains(w)){
            discovered.add(w);
            queue.enqueue(w);
        }
    }
}
```

(4) Topological Sorting

- Precedence scheduling
  - Goal : Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks
  - Digraph model : vertex = task edge = precedence constraint (There is no cycle in the graph i.e. acyclic graph)
  - Topological sort : Redraw the graph s.t. all edges point upwards

- Solve : DFS
  - Run DFS
  - Return vertices in reverse postorder (The order of solved)
- Implementation

```java
public class DepthFirstOrder{
    private boolean[] marked;
    private Stack<Integer> reversePost;
    public DepthFirstOrder(Digraph G){
        reversePost = new Stack<Integer>();
        marked = new boolean[G.V()];
        for(int i=0 ; i<G.V(); i++)
            if(!marked[i]) dfs(G,i);
    }
    private void dfs(Digraph G, int i){
        marked[i] = true;
        for(int w:G.adj(i))
            if(!marked(w)) dfs(G,w);
        reversePost.push(i);
    }
    public Iterable<Integer> reversePost(){
        return reversePost;
    }
}
```

- Consider an execution of depth-first search on a directed acyclic graph G which contains the edge $v \to w$. Which one of the following is impossible at the time $dfs(v)$ is called?
  - dfs(w) has already been called but not yet returned.
  - If $dfs(v)$ is called before $dfs(w)$ returns, then the function-call stack contains a directed path from w to v (as in the previous in-video quiz). Combining this path with the edge $v \to w$ yields a directed cycle, which is impossible since G is acyclic.
- Proposition : Reverse DFS postorder of a DAG(no cycle graph) is a topological order
  - Pf : Consider any edge v -> w. When dfs(v) is called
    - Case1: `dfs(w)` has already been called and returned, w has done before v
    - Case2: `dfs(w)` has not yet been called => it will be called directly or indirectly by `dfs(v)` , and will finish before it
    - Case3: impossible
- Directed cycle detection
  - Proposition : A digraph has a topological order iff no directed cycle
  - DFS

(5) Strong components

- Def : Vertices v and w are strongly connected if there is a directed path from v to w and a directed path from w to v
    - Equivalence relation
- Method used in Undirected(DFS) => wrong because the source node may not be the first one in topological order
- Kosaraju-Sharir algorithm
    - Reverse graph : Strong components in G are same as in G^R
    - Kernel DAG : Contract each strong component into a single vertex
    - Idea :
        - Compute topological order in kernel DAG
        - Run DFS, consider vertices in reverse topological order
        - Phase 1 : Compute reverse postorder in G^R
        - Phase2 : Run DFS in G, in the order of reverse postorder of G^R
    - Implementation

```
public class KosarajuSharirSCC{
    private boolean marked[];
    private int[] id;
    private int count;
    public  KosarajuSharirSCC(Digraph G){
        marked = new boolean[G.V()];
        id = new int[G.V()];
        count=0;
        DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
        for(int v:dfs.reversePost()){
            if(!marked[v]){
                dfs(G,v);
                count++;
            }
        }
    }
    private void dfs(G,v){
        marked[v] = true;
        id[v] = count;
        for(int w:G.adj(v))
            if(!marked[w])
                dfs(G,w);
    }
    public boolean stronglyConnected(int v,int w){
        return id[v]==id[w];
    }
}
```

- Actually, you only need to use DFS in phase 1, in phase 2 any search algorithm is avaliable

- Convex Hull
    - Quick Hull
        - divide the plane into two every time until there is still outside points n^2
    - Merge Hull
        - generate 1 convex Hull from 2 convex Hull $nlogn$
    - Graham Scan