ADT, OOP & Algorithm, Design Patterns, API testing, Software testing Review C++, Python, C#

Ron Wu

Last update: 3/11/16

Table of Contents

Most Python codes in the notes are from https://nbviewer.ipython.org/github/jmportilla/Python-for-Algorithms--Data-Structures--and-Interviews/tree/master/

Most C++ codes in the notes are from *C++ Primer*

Array

1. String

- String is immutable, use list

```
# Run-length encoding
def string_compression(s):
    l =  len(s)
    if l == 0:
        return ""
    output = [s[0], "1"]

    for i in range(1,l):
        if output[-2]==s[i]:
            j = int(output[-1])+1
            output[-1] = str(j)
        else:
            output.append(s[i])
            output.append("1")
    return "".join(output)

string_compression("AAAbbb")  #A3b3
```

2. Dynamic Array

Indexing, appending – O(1) , iterate – O(n), sort – O(n*log*n)

```
#include <iostream>
#include <list>
#include <vector>
using namespace std;

int main()
{
       int * array = new int[] {1,2,3};
       list<int> listInt;
    return 0;
```

```
}
```

- Create array with twice capacity

```
import sys

l = list(range(10000))

data = []

for i in range(10):
    a = len(data)
    b = sys.getsizeof(data)

    print('Length: {0:3d}; size in bytes: {1:4d}'.format(a,b)) #1bit = 8 bytes
    data.append(10)

import ctypes

class DynamicArray(object):

    def __init__(self):
        self.n = 0
        self.capacity = 1
        self.A = self.make_array(self.capacity)

    def __len__(self):
        return self.n

    def __getitem__(self,k):
        if not 0 <= k <self.n:
            return IndexError('K is out of bounds!')

        return self.A[k]

    def append(self, ele):
        if self.n == self.capacity:
            self._resize(2*self.capacity) #when full, double in size

        self.A[self.n] = ele
        self.n += 1

    def _resize(self,new_cap):

        B = self.make_array(new_cap)

        for k in range(self.n):
            B[k] = self.A[k]

        self.A = B
        self.capacity = new_cap

    def make_array(self,new_cap):
        return (new_cap * ctypes.py_object)()
```

3. Hash Table

- Fetch – O(1), iterate – O(n), load factor
- Folding method, mid-square method, ordinal value, open address (linear probing, quadric), rehashing, chaining

```
class HashTable(object):

    def __init__(self,size):
        self.size = size
        self.slots = [None] * self.size
        self.data = [None] * self.size

    def put(self,key,data):
        hashvalue = self.hashfunction(key,len(self.slots))

        if self.slots[hashvalue] == None:
            self.slots[hashvalue] = key
            self.data[hashvalue] = data

        else:

            if self.slots[hashvalue] == key:
                self.data[hashvalue] = data

            else:

                nextslot = self.rehash(hashvalue,len(self.slots))

                while self.slots[nextslot] != None and self.slots[nextslot] != key:
                    nextslot = self.rehash(nextslot,len(self.slots))

                if self.slots[nextslot] == None:
                    self.slots[nextslot]=key
                    self.data[nextslot]=data

                else:
                    self.data[nextslot] = data

    def hashfunction(self,key,size):
        return key%size

    def rehash(self,oldhash,size):
        return (oldhash+1)%size


    def get(self,key):

        startslot = self.hashfunction(key,len(self.slots))
        data = None
        stop = False
        found = False
        position = startslot

        while self.slots[position] != None and not found and not stop:
```

```
            if self.slots[position] == key:
                found = True
                data = self.data[position]

            else:
                position=self.rehash(position,len(self.slots))
                if position == startslot:

                    stop = True
        return data

    def __getitem__(self,key):
        return self.get(key)

    def __setitem__(self,key,data):
        self.put(key,data)
```

4. Set

- insert, find – O(1)

```
aset = set([1,2,2,2]) # (1,2)


def unique_char(s):
       return len(set(s)) == len(s)

unique_char("sds")
```

- dictionary to test anagram

```
d = {'k1':1,'k2':2}

def anagram2(s1,s2):
    s1 = s1.replace(" ", "").lower()
    s2 = s2.replace(" ", "").lower()

    if len(s1)!=len(s2):
        return False

    count = {}

    for letter in s1:
        if letter in count:
            count[letter] +=1
        else:
            count[letter] = 1

    for letter in s2:
        if letter in count:
            count[letter] -=1
        else:
```

```
            count[letter] = 1

    for k in count:
        if count[k] != 0:
            return False

    return True
```

Stack & Queue & Deques

5. Stack - LIFO

- Push, pop, peek

```
class Stack(object):

    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if self.size() > 0:
            return self.items.pop()

    def peek(self):
        if self.size() > 0:
            return self.items[-1]
        else:
            return "emtpy stack"

    def size(self):
        return len(self.items)
```

- Close parenthese

```
def balance_parentheses(s):
    stack = Stack()
    match = set([('(',')'),('[',']'),('{','}')])
    for ch in s:
        if ch in ['{','(','[']:
            stack.push(ch)
        elif ch in ['}', ']',')']:
            if (stack.pop(),ch) not in match:
                return False
    return stack.size() == 0
```

6. Queue - FIFO

- Enquene, dequeue

```
class Queue(object):
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items==[]

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        if self.size() > 0:
            return self.items.pop()

    def size(self):
        return len(self.items)
```

- Implement a queue using two stacks

```
class Queue_byTwoStacks(object):
    def __init__(self):
        self.Stack1 = Stack()
        self.Stack2 = Stack()

    def _moveStack1ToStock2s(self, s1, s2):
        while s1.size()>0:
            s2.push(s1.pop())

    def isEmpty(self):
        return self.Stack1.isEmpty() & self.Stack2.isEmpty()

    def enqueue(self, item):
        if self.isEmpty:
            self.Stack1.push(item)
        elif self.Stack1.isEmpty():
            self._moveStack1ToStock2s(self.Stack2, self.Stack1)
            self.Stack1.push(item)
        else:
            self.Stack1.push(item)

    def dequeue(self):
        if self.isEmpty():
            return
        elif self.Stack1.isEmpty():
            return self.Stack2.pop()
        else:
            self._moveStack1ToStock2s(self.Stack1, self.Stack2)
            return self.Stack2.pop()

    def peek(self):
        if self.isEmpty():
            return
        elif self.Stack1.isEmpty():
            return self.Stack2.peek()
        else:
            self._moveStack1ToStock2s(self.Stack1, self.Stack2)
```

```
            return self.Stack2.peek()

    def size(self):
        return self.Stack1.size() + self.Stack2.size()


q = Queue_byTwoStacks()
q.enqueue("1")
q.enqueue("2")
q.enqueue("3")

#more elegant
class Queue_byTwoStacks(object):
    def __init__(self):
        self.Stack1 = Stack()
        self.Stack2 = Stack()

    def isEmpty(self):
        return self.Stack1.isEmpty() & self.Stack2.isEmpty()

    def enqueue(self, item):
        self.Stack1.push(item)

    def dequeue(self):
        if self.Stack2.isEmpty():
            while self.Stack1.size():
                self.Stack2.push(self.Stack1.pop())
        return self.Stack2.pop()

    def peek(self):
        if self.Stack2.isEmpty():
            while self.Stack1.size():
                self.Stack2.push(self.Stack1.pop())
        return self.Stack2.peek()

    def size(self):
        return self.Stack1.size() + self.Stack2.size()
```

7. Deque

- addFront, addRear, removeFront, removeRear

```
class Deque(object):
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items==[]

    def addFront(self, item):
        self.items.insert(0, item)

    def addRear(self, item):
        self.items.append(item)
```

```
    def removeFront(self):
        if self.size() > 0:
            return self.items.pop(0)

    def removeRear(self):
        if self.size() > 0:
            return self.items.pop()

    def size(self):
        return len(self.items)
```

Linked List

8. Single / double / circular linked list

-

```
 class Node(object):
    def __init__(self,value):
        self.value = value
        self.nextnode = None

a = Node(1)
b = Node(2)
c = Node(3)

a.nextnode = b
b.nextnode = c
c.nextnode = None
```

- Linked list reversal

```
def reverse(head):
    current = head
    prv = None
    nextnode =None

    while current != None:
        nextnode = current.nextnode
        current.nextnode = prv
        prv = current
        current = nextnode

    return prv
```

2.

Trees

9. Binary tree

- Preorder, in-order, post-order

```
 class BinaryTree(object):
```

```
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftchild = None
        self.rightchild = None

    def insertLeft(self, newNode):
        if self.leftchild == None:
            self.leftchild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.leftchild = self.leftchild
            self.leftchild = t

    def insertRight(self, newNode):
        if self.rightchild == None:
            self.rightchild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightchild = self.rightchild
            self.rightchild = t

    def getRightChild(self):
        return self.rightchild

    def getLeftChild(self):
        return self.leftchild

    def setRootVal(self, obj):
        self.key = obj

    def getRootVal(self):
        return self.key

    def preorder(self):
        print(self.key)
        if self.leftchild:
            self.leftchild.preorder()
        if self.rightchild:
            self.rightchild.preorder()

    def postorder(self):
        if self.leftchild:
            self.leftchild.postorder()
        if self.rightchild:
            self.rightchild.postorder()
        print(self.key)


    def inorder(self):
        if self.leftchild:
            self.leftchild.inoder()
        print(self.key)
        if self.rightchild:
            self.rightchild.inoder()

bb = BinaryTree('a')
bb.insertLeft('b')
bb.preorder()
```

10. Binary Heap

- Make a balanced tree -> complete binary tree, space, search O(n), Delete, insert O(log(n)), Peek O(1)

```
 class Binheap(object):
    def __init__(self):
        self.heapList=[0]
        self.currentSize = 0

    def percUp(self, i):
        while i // 2 >0:
            if self.heapList[i] < self.heapList[i//2]:
                self.heapList[i],self.heapList[i//2]  = self.heapList[i//2],
self.heapList[i]
            i = i//2

    def minChild(self, i):
        if i*2 +1 > self.currentSize:
            return i * 2
        if self.heapList[i*2]<self.heapList[i*2+1]:
            return i * 2
        return i*2+1

    def percDown(self,i):
        while (i*2)<=self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i]>self.heapList[mc]:
                self.heapList[i], self.heapList[mc] = self.heapList[mc],
self.heapList[i]
            i = mc

    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize -= 1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def buildHeap(self,alist):
        i = len(alist) //2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while i>0:
            self.percDown(i)
            i =  i-1

bb = Binheap()
bb.buildHeap([31,1,36,3])
bb.heapList
bb.delMin()
bb.heapList
```

11. Binary Search Tree (bst)

- Inorder, yield for state function, generator

```
def __iter__(self):
    if self:
        if self.hasLeftChild():
            for elem in self.leftChild:
                yield elem
        yield self.key
        if self.hasRightChild():
            for elem in self.rightChild:
                yield elem
```

- verify bst

```
def isBST(tree):
       if tree == None:
              return True
       return isBST(tree.LeftMaxVal) <= tree.Value <= isBST(tree.RightMinVal)
```

- Traversal -> Breath First

```
output = []
def BTreeBreathFirst(tree, output, levelcount = 0):
    if not tree:
        return
    output[levelcount] += [tree.value]
    BTreeBreathFirst(tree.getLeftChild(), output, levelcount + 1)
    BTreeBreathFirst(tree.getRightChild(), output, levelcount + 1)

print(output)

##Or use queue, iterations

def BTreeBreathFirst_rec(tree):
    levelcount = 0
    if not tree:
        return
    nodes=collections.deque([tree])
    currentCount, nextCount = 1, 0
    while len(nodes)!=0: # empty queue
        currentNode=nodes.popleft() #dequeue from the left
        currentCount-=1
        output[levelcount] += [currentNode.val]
        if currentNode.left:
            nodes.append(currentNode.left)  #enqueue from the right
            nextCount+=1
        if currentNode.right:
            nodes.append(currentNode.right)
            nextCount+=1
```

```
        if currentCount==0:
            levelcount +=1
            currentCount, nextCount = nextCount, currentCount

print(output)
```

- Trim bst

```
 def TrimBSTLeft(tree, min_val):
    '''assume tree root is > min_val'''
    note = tree
    while note:
        if note.value == min_val:
            note.LeftChild == None
            break

        if note.hasLeftChild & note.getLeftChild().value < min_val:
            note.LeftChild = note.getLeftChild().getRightChild()

        note = note.getLeftChild()
```

12.2-3 tree – O(log n)

- C++ implement

```
//by RON WU
#include <iostream>

using namespace std;

struct node
{
       node(int);
       ~node();
       int * small, * big;
       node * left, * middle, * right;
};

node::node(int in)
{
      small=new int;  (*small)=in;
      big=NULL;  left=NULL; middle=NULL; right=NULL;
}

node::~node()
{
      delete small;  delete big;
}

class tree23
{
public:
      tree23();
      ~tree23();
      void display();
```

```
      void insert(int);
      int determheight();
private:
      int insert(node *&, int);
      void display(node*);
      int determheight(node*);
      void delnode(node*&);
      node* treeroot;
};

tree23::tree23()
{
    treeroot=NULL;
}

tree23::~tree23()
{
    delnode(treeroot);
}

void tree23::delnode(node*&root)
{
    if(root)
    {
        if(!root->left)   //a leaf
        {
           delete root;
           root=NULL;
        }
        else
        {
           delnode(root->left);
           delnode(root->middle);
           delnode(root->right);
        }
    }
}

void tree23::display()
{
      display(treeroot);
}

void tree23::display(node*root)
{
    if(root)
    {
        display(root->left);
        cout<<*(root->small)<<" ";
        display(root->middle);
        if(root->big)  //2-node
        {
             cout<<*(root->big)<<" ";
             display(root->right);
        }
    }
}
```

```
int tree23::determheight()
{
    return determheight(treeroot);
}

int tree23::determheight(node*root)
{
    if (!root)
       return 0;
    else if(!root->big)
    {
       int heigh_left=1+determheight(root->left);
       int heigh_middle=1+determheight(root->middle);
       if (heigh_left>heigh_middle)
          return heigh_left;
       else
          return heigh_middle;
    }
    else
    {
        int heigh_left=1+determheight(root->left);
        int heigh_middle=1+determheight(root->middle);
        int heigh_right=1+determheight(root->right);
        if (heigh_left>heigh_middle)
        {
            if(heigh_left>heigh_right)
                return heigh_left;
            else
                return heigh_right;
        }
        else
        {
            if(heigh_right>heigh_middle)
                return heigh_right;
            else
                return heigh_middle;
        }
    }
}


void tree23::insert(int in)
{
     insert(treeroot,in);
}

int tree23::insert(node*&root,int in)
{
     if(!root)  //very first time insertion
     {
         root=new node(in);
         return 1;
     }
     if(!root->left)
     {
         if(!root->big)  //1-node leaf
         {
            if(in>*(root->small))
```

```
            {
                root->big=new int;
                *(root->big)=in;
            }
            else
            {
               root->big=root->small;
               root->small=new int;
               *(root->small)=in;
            }
            return 1;
        }
        //2-node leaf
        if(in<*(root->small))
        {
            int temp=*(root->small);
            *(root->small)=in;
            in=temp;
        }
        else if (in>*(root->big))
        {
            int temp=*(root->big);
            *(root->big)=in;
            in=temp;
        }

        node* temp=root;
        root=new node(in);
        root->left=temp;
        root->middle=new node(*(root->left->big));
        delete root->left->big;
        root->left->big=NULL;

        return 0;
    }
    else //not a leaf
    {
       if(!root->big)
       {
         if(in<*(root->small))
         {
            if(!insert(root->left,in))
            {
                root->right=root->middle;
                node*temp=root->left;
                root->middle=temp->middle;
                root->left=temp->left;
                root->big=root->small;
                root->small=new int;
                *(root->small)=*(temp->small);
                delete temp;
            }
         }
         else
         {
             if(!insert(root->middle,in))
             {
                node*temp=root->middle;
```

```
                root->middle=temp->left;
                root->right=temp->middle;
                root->big=new int;
                *(root->big)=*(temp->small);
                delete temp;
            }
        }
        return 1;
    }
    else  //2-node
    {
        if(in<*(root->small))
        {
            if(!insert(root->left,in))
            {
                node*temp=root->middle;
                root->middle=new node(*(root->big));
                delete root->big;
                root->big=NULL;
                root->middle->left=temp;
                root->middle->middle=root->right;
                root->right=NULL;
                return 0;
            }
        }
        else if(in>*(root->big))
        {
            if(!insert(root->right,in))
            {
                node*temp=root;
                root=new node(*(temp->big));
                root->left=temp;
                root->middle=temp->right;
                delete temp->big;
                temp->big=NULL;
                temp->right=NULL;
                return 0;
            }
        }
        else
        {
            if(!insert(root->middle,in))
            {
                node *temp=root;
                root=new node(*(root->middle->small));
                root->left=temp;
                root->middle=new node(*(temp->big));
                delete temp->big;
                temp->big=NULL;
                root->middle->left=temp->middle->middle;
                root->middle->right=temp->right;
                temp->right=NULL;
                node*cur=temp->middle;
                temp->middle=cur->left;
                delete cur;
                return 0;
            }
        }
```

```
            return 1;
        }
     }
}

int main()
{
    tree23 m;
    int p[9]={50, 30, 20, 65, 35, 25, 5, 11, 70};

    for (int i=0; i<9;++i)
    {
         m.insert(p[i]);
    }

    m.display();
    cout<<"\nheight: "<<m.determheight()<<endl;
    cin.get();
}
```

13.2-3-4 tree

```
//by RON WU
#include <iostream>

using namespace std;

struct node
{
     node(int);
     ~node();
     int * data[3];
     node * pt[4];
};

node::node(int in)
{
     data[0]=new int (in);
     data[1]=data[2]=data[3]=NULL;
     pt[0]=pt[1]=pt[2]=pt[3]=NULL;
}

node::~node()
{
      delete data[0];
      delete data[1];
      delete data[2];
}

class tree234
{
public:
```

```
      tree234();
      ~tree234();
      void insert(int in);
      int display();
      int determheigh();
private:
      void delnode(node*&);
      void display(node*);
      int insert(node*&, int);
      int determheigh(node*);
      node * treeroot;
};

tree234::tree234():treeroot(NULL)
{}

tree234::~tree234()
{
    delnode(treeroot);
}

void tree234::delnode(node*& root)
{
    if(root)
    {
      for (int i=0; i<4; ++i)
       delnode(root->pt[i]);
      delete root;
      root=NULL;
    }
}

int tree234::display()
{
   if(!treeroot)
     return 0;
   display(treeroot);
   return 1;
}

void tree234::display(node*root)
{
     if(root!=NULL)
     {
        display(root->pt[0]);
        cout<<*root->data[0]<<" ";
        display(root->pt[1]);
        if(root->data[1])
        {
           cout<<*root->data[1]<<" ";
           display(root->pt[2]);
           if (root->data[2])
           {
              cout<<*root->data[2]<<" ";
              display(root->pt[3]);
           }
        }
     }
```

```
}

int tree234::determheigh()
{
    return determheigh(treeroot);
}

int tree234::determheigh(node*root)
{
    if(!root)
       return 0;
    int heigh[4];
    for (int i=0; i<4;++i)
    {
        heigh[i]=1+determheigh(root->pt[i]);
    }

    int max=heigh[0];
    for (int i=1; i<4; ++i)
    {
        if(max<heigh[i])
          max=heigh[i];
    }
    return max;
}

void tree234::insert(int in)
{
     insert(treeroot, in);
}

int tree234::insert(node*&root, int in)
{
     if(!root)
     {
        root=new node(in);
        return 1;
     }
     if(!root->pt[0])
     {
        if(!root->data[2])
        {
            if(!root->data[1])
            {
               if(in<*(root->data[0]))
               {
                  root->data[1]=root->data[0];
                  root->data[0]=new int (in);
               }
               else
               {
                   root->data[1]=new int (in);
               }
            }
            else
            {
               if(in<*root->data[0])
               {
```

```
                root->data[2]=root->data[1];
                root->data[1]=root->data[0];
                root->data[0]=new int (in);
            }
            else if(in<*root->data[1])
            {
                root->data[2]=root->data[1];
                root->data[1]=new int (in);
            }
            else
            {
                root->data[2]=new int (in);
            }
        }
        return 1;
    }
    //3-node
    node*temp=root;
    root=new node(*temp->data[1]);
    root->pt[0]=temp;
    root->pt[1]=new node(*temp->data[2]);
    delete temp->data[1];
    delete temp->data[2];
    temp->data[1]=temp->data[2]=NULL;

    if(in<*temp->data[0])
    {
       temp->data[1]=temp->data[0];
       temp->data[0]=new int (in);
    }
    else if (in<*root->data[0])
    {
         temp->data[1]=new int(in);
    }
    else if (in<*root->pt[1]->data[0])
    {
         root->pt[1]->data[1]=root->pt[1]->data[0];
         root->pt[1]->data[0]=new int(in);
    }
    else
    {
         root->pt[1]->data[1]=new int(in);
    }
    return 0;
}
//not a leaf
if (!root->data[2])
{
    if(!root->data[1])
    {
        if (in<*root->data[0])
        {
           if(!insert(root->pt[0],in))
           {
                 node* temp=root->pt[0];
                 root->data[1]=root->data[0];
                 root->data[0]=new int(*temp->data[0]);
                 root->pt[2]=root->pt[1];
```

```
                    root->pt[1]=temp->pt[1];
                    root->pt[0]=temp->pt[0];
                    delete temp;
                }
            }
            else
            {
                if(!insert(root->pt[1],in))
                {
                    node*temp=root->pt[1];
                    root->pt[1]=temp->pt[0];
                    root->pt[2]=temp->pt[1];
                    root->data[1]=new int(*temp->data[0]);
                    delete temp;
                }
            }
        }
        else //2-node
        {
            if(in<*root->data[0])
            {
                if(!insert(root->pt[0],in))
                {
                    node* temp=root->pt[0];
                    root->data[2]=root->data[1];
                    root->data[1]=root->data[0];
                    root->data[0]=new int(*temp->data[0]);
                    root->pt[3]=root->pt[2];
                    root->pt[2]=root->pt[1];
                    root->pt[1]=temp->pt[1];
                    root->pt[0]=temp->pt[0];
                    delete temp;
                }
            }
            else if(in<*root->data[1])
            {
                if(!insert(root->pt[1],in))
                {
                    node*temp=root->pt[1];
                    root->data[2]=root->data[1];
                    root->data[1]=new int (*temp->data[0]);
                    root->pt[3]=root->pt[2];
                    root->pt[2]=temp->pt[1];
                    root->pt[1]=temp->pt[0];
                    delete temp;
                }
            }
            else
            {
                 if(!insert(root->pt[2],in))
                 {
                    node*temp=root->pt[2];
                    root->data[2]=new int (*temp->data[0]);
                    root->pt[3]=temp->pt[1];
                    root->pt[2]=temp->pt[0];
                    delete temp;
                 }
            }
```

```
            }
        }
        else//3-node
        {
            node*temp=root;
            root=new node(*temp->data[1]);
            root->pt[0]=temp;
            root->pt[1]=new node(*temp->data[2]);
            delete temp->data[1];
            delete temp->data[2];
            temp->data[1]=temp->data[2]=NULL;
            root->pt[1]->pt[0]=temp->pt[2];
            root->pt[1]->pt[1]=temp->pt[3];
            temp->pt[2]=temp->pt[3]=NULL;
            insert(root,in);
            return 0;
        }
        return 1;
}

int main()
{
    tree234 m;
    int p[5]={1,3,4,2,6};
    for(int i=0;i<5;++i)
    {
      m.insert(p[i]);
    }
    m.display();
    cout<<"\nHeight: "<<m.determheigh();
    cin.get();
}
```

Graph

14. Adjacency List

- Use dictionary for connections

```
class Vertex:
    def __init__(self,key): #key unique
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self,nbr,weight=0): #nbr <- id
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in
self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()
```

```
    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices  += 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost) #one direction

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())



g = Graph()
g.addEdge(0,1,2)

for vertex in g:
    print (vertex)
    print (vertex.getConnections())
    print ('\n')

#0 connectedTo: [1]
#1 connectedTo: []
```

15. Graph

- Node, label for isVisited …

```
from enum import Enum
from collections import OrderedDict

class State(Enum):
    unvisited = 1
    visited = 2
    visiting = 3


class Node:

    def __init__(self, num):
        self.num = num
        self.visit_state = State.unvisited
        self.adjacent = OrderedDict()  # key = node, val = weight

    def __str__(self):
        return str(self.num)


class Graph:

    def __init__(self):
        self.nodes = OrderedDict()  # num = node id, val = node

    def add_node(self, num):
        node = Node(num)
        self.nodes[num] = node
        return node

    def add_edge(self, source, dest, weight=0):
        if source not in self.nodes:
            self.add_node(source)
        if dest not in self.nodes:
            self.add_node(dest)
        self.nodes[source].adjacent[self.nodes[dest]] = weight

g = Graph()
g.add_edge(0, 1, 5)
g.nodes
```

16. Breath First Search

- Connected component

```
def BFS_ConnectedComp(graph, start):
    visited = []
    q = queue()
    q.enqueue(start)

    while q:
```

```
        vex = q.dequeue();
        if not vex in visited:
            visited.add(vex)
            q.enqueue(graph[vex].node-visited)
    return visited
```

- All path and the Shortest path

```
 def BFS_path(graph, start, end):
    q = queue()
    q.enqueue([start,[start]])

    while q:
        (vex, path) = q.dequeue()
        for nxt in graph[vex]-set(path):
            if nxt == end:
                yield path + [nxt]
            else:
                q.enqueue([start, path + [nxt]])
```

17. Depth first Search

- Connected component

```
def DFS_ConnectedComp(graph, start):
    visited = []
    s = stack
    s.push(start)

    while s:
        vex = s.pop();
        if not vex in visited:
            visited.add(vex)
            s.push(graph[vex].node-visited)
    return visited
```

- paths

```
def DFS_path(graph, start, end):
    s = stack()
    s.push([start,[start]])

    while q:
        (vex, path) = s.pop()
        for nxt in graph[vex]-set(path):
            if nxt == end:
                yield path + [nxt]
            else:
                s.push([start, path + [nxt]])
```

OOP

18. Good C++ practices

- Separate header file (.h), implementation file(.cpp) unless use INLINE functions

```
 //employee.h class interface
class employee
{
public:
	void read();	//read from stdin to employee object
	void write();	//write employee object to stdout
	long get_salary(); //get salary

private:

	char name[32];
	long salary;
};
```

```
//employee.cpp class implementation
#include <iostream>
using namespace std;
#include "employee.h"

void employee::read() {
	char delim;
	cin >> delim;
	cin.getline(name, 32, ':');
	cin >> salary >> delim;
}

void employee::write() {
	cout << "\nEmployee's name:  " << name;
	cout << "\nSalary:           $" << salary;
}

long employee::get_salary() {

	return (salary);

}
```

```
//sorted.h class interface
typedef long KEY_TYPE; //type of search key for sorted list
class sorted {
public:
	sorted();
	~sorted();
	void insert(void*, char, KEY_TYPE); //insert object
	void* next(char &); //get next object

private:
	struct node {	//linked list node structure
```

```
                KEY_TYPE key;       //search key in *data_ptr
                char data_type;     //type of object in *data_ptr
                void* data_ptr;     //pointer to object
                node* link;         //pointer to next node in list

        };
        node* list_head;    //list head
        node* cur_ptr;      //current node for next
};
```

```
//sorted.cpp class implementation
#include "sorted.h"
sorted::sorted() {
        cur_ptr = list_head = 0;
}

sorted::~sorted() {
        node* node_ptr
        while (node_ptr = list_head) {
                list_head = node_ptr->link;
                delete node_ptr;
        }
}

void sorted::insert(void* data_ptr, char data_type,     KEY_TYPE key) {
        node* node_ptr = new node;
        node_ptr->data_type = data_type;
        node_ptr->data_ptr = data_ptr;
        node_ptr->key = key;

        node* prev_ptr = 0;
        node* next_ptr = list_head;

        while (next_ptr && (key > next_ptr->key)) {
                prev_ptr = next_ptr;
                next_ptr = next_ptr->link;
        }

        node_ptr->link = next_ptr;

        if (prev_ptr)
                prev_ptr->link = node_ptr;
        else
                list_head = node_ptr;
}

void* sorted::next(char &type) {
        cur_ptr = cur_ptr ? cur_ptr->link : list_head;
        type = cur_ptr ? cur_ptr->data_type : ' ';
        return (cur_ptr ? cur_ptr->data_ptr : 0);
}
```

```
//Client Program
#include <iostream>
using namespace std;
#include "employee.h"
#include "sorted.h"
```

```
//display employees and managers from the sorted list
void write_sorted_list(sorted &list) {
	char type; //specifies type of data
	void* ptr; //pointer to employee or manager

	while (ptr = list.next(type))
	{
		static_cast<employee*>(ptr)->write();
	}
}

int main() {
	cout << "\nEmployees Sorted By Salary\n";
	write_sorted_list(sal_list);
    return (0);
}
```

- Inline function: compiler will attempt to generate inline code. When member functions are defined outside of the class interface, we must qualify them with the class name and the scope resolution operator. If we want such a member function to be compiled as inline code, we must use the keyword inline in front of the member function and define the function in the same file as the class interface

```
class tree { //binary tree class
public:
	tree();
	tree(const tree &);
	~tree();
	class node;   //partial declaration
private:

	node* root;
	static void insert(const employee &emp);

};

inline tree::tree() :

	root(0) {

}
inline void tree::insert(const employee &emp) {
	node* new_node = new node(emp);
	root = add(root, new_node);

}


//because the new operator in insert
//must include class node in tree.cpp
//because it needs a complete definition of a node to allocate memory.
class node {
public:
	node() :
		left(0),
		right(0) {
```

```
        }

        node(const employee &obj) :
                emp(obj),
                left(0),
                right(0) {
        }

        employee emp;
        node* left;                 //left child node
        node* right;                //right child node
};
```

19. Encapsulation: Class scope

- Construction list

```
class_name{
public:
        //a constructor declaration with an initialization list
        //initialization list must be with constructor definition
        class_name(arg_list) :
                data_member_1(initial_value),
                data_member_2(initial_value) {

                //constructor implementation

        }

        ...

};
```

- If an exception occurs while constructing an object, the destructor is automatically invoked to correctly destroy a partially constructed object.
- allocating dynamic memory inside of class-- > must provide destructor, call by value v.s. call by reference

```
 #include <cstring>

using namespace std;

class name {

public:
        name(char* = "");         //default constructor
        name(const name &);       //copy constructor
        ~name();
        name &operator=(name &); //assignment op for deep copy
        const char* get_name();  //get pointer to name

private:
        char* ptr;  //pointer to name
        int length; //length of name
};
```

```
name::name(char* name_ptr) {
       length = strlen(name_ptr) + 1;
       ptr = new char[length];      //dynamically allocate space
       strcpy(ptr, name_ptr);
}

name::~name() {
       delete[] ptr;
}

const char* name::get_name() { //get pointer to name
       return(ptr);
}

name::name(const name &obj) {  //copy constructor
       length = obj.length;
       ptr = new char[length];
       strcpy(ptr, obj.ptr);
}

int main() {

       name blank;              //default constructor used
       name* ptr_name;          //pointer to object of type name
       ptr_name = new name;     //default constructor
       name smith("Sue Smith"); //one arg constructor used
       name clone_smith(smith); //copy constructor is called
       function_pass_by_val(const smith); //smith.prt is copied and the
                                          //memory to the name ptr will be
                                          // deallocated
       function_pass_by_ref(const smith); //always good choice. avoided calling
                                          //constructor, destructor
                                           // but don't return ref of a local
                                           // variable created inside function

       return (0);

}
```

20. polymorphism -- Operator Overloading

- Operators must come from the built-in operators.
- Operators maintain their precedence and associativity.
- Operators must be overloaded to expect the correct number of operands. Unary operators expect one operand and binary operators expect two operands.
- Operators can only be overloaded when at least one of the operands is an object of a class. No redefine the operation of operators on built-in types.
- Operators cannot have default arguments.

```
 //complex.h
class complex {
```

```
public:
	explicit complex(double r = 0, double i = 0) : //explicit prevent
                                                   // implicit conversion
		real(r), imag(i) {
	}

	double real; //real component
	double imag; //imaginary component
};

//function prototypes
complex operator- (const complex &, const complex &);

//////////////
//complex.cpp
//////////////
#include "complex.h"

//define overloaded operator- function for complex class
complex operator-(const complex &c1, const complex &c2) {
	complex temp;
	temp.real = c1.real - c2.real;
	temp.imag = c1.imag - c2.imag;
	return (temp);
}

//main.cpp

#include <iostream>
using namespace std;
#include "complex.h"

int main() {
	complex c1(4.0, 3.0);
	complex c2(1.0, 2.0);
	complex c;

	c = c1 - c2;
	// or just call c = operator-(c1, c2);
	return (0);
}
```

- There are twelve operators that cannot be overloaded, ::, ., .*, ?:, etc
- We prefer overload operator inside the class. For non-member, declare FRIEND to access private data.

```
class complex {

public:
	explicit complex(double r = 0, double i = 0) :
		real(r), imag(i) {
	}
	complex operator-() const;
private:
	double real;
```

```
        double imag;

};


//unary member function
inline complex complex::operator-() const {
        complex temp;
        temp.real = -real;
        temp.imag = -imag;
        return (temp);
}


//Overloading Operators as Non - members
class complex {
        friend complex operator-(const complex &); //unary

public:
        explicit complex(double r = 0, double i = 0) :
                real(r), imag(i) {
        }

private:
        double real;
        double imag;
};

//unary non-member function
inline complex operator-(const complex &c) {
        complex temp;
        temp.real = -c.real;
        temp.imag = -c.imag;
        return (temp);
}
```

- Overloading the iostream

```
 class complex {

        friend std::istream &operator>>(std::istream &, complex &);
        friend std::ostream &operator<<(std::ostream &,  const complex &);

public:
        complex(double r = 0, double i = 0) : real(r), imag(i) { }

private:
        double real;
        double imag;
};

istream &operator>>(istream &i, complex &c) {
        i >> c.real;
        i >> c.imag;
        return (i);
}
```

```
ostream &operator<<(ostream &o, const complex &c) {
	o << c.real;
	o << " ";
	o << c.imag;
	return (o);
}
```

- Dynamic Binding (overriding): inheritance hierarchy must use public derivation
- Overriding v.s. overloading: Overloading requires unique signatures whereas overriding requires the same signature and return type. Secondly, overloading requires that each overloaded version of the function be specified within the same scope whereas overriding requires each overridden version be specified within the scope of each derived class.
- Overriding v.s. hiding: Hiding member function in the base class has no requirements on the signatures. And hiding is static.
- Rules: virtual functions cannot be static member functions; the signature and return type must be the same for all implementations of the virtual function. If different, the virtual function will be invoked as an inherited function. Once a member function is declared to be virtual, it remains virtual for all derived classes. The derived class implementations of the overridden function do not need to repeat the use of that keyword.
- VTable v.s. Vpointer: One way that dynamic binding is often implemented is to create an array of member function pointers for all functions that are declared to be virtual. Each derived class has its own unique array of member function pointers. Functions that are inherited result in pointers to direct or indirect base class member functions. Only one table exists per class that is shared by all objects created from this class. Each derived class object contains its own pointer to this virtual table for its class. This pointer is sometimes called a virtual pointer.
- Virtual Friend is a non-member function that uses a virtual member function to implement its operation. This is done when functions and overloaded operators that cannot be implemented as members so we use dynamic binding by invoking a virtual helper member function that actually performs the required operation
- Whenever there is virtual function, always declare the destructor to be virtual. so that derived class destructors will be called when the object is deallocated

```
class account {

	friend std::ostream &operator<<(std::ostream &, account &);

public:
	account(const char* = "none", float = 0);
	virtual ~account();

protected:
	virtual void statement(std::ostream &);

private:
	char name[32];
```

```
        float balance;

};

class checking : public account
{
public:
        checking(const char* = "none", float = 0, float = 5);
        ~checking();
protected:
        void statement(std::ostream &);

private:
        float charges;

};


account::account(const char* n, float b) :

        balance(b) {
        strncpy(name, n, 32);
        name[31] = '\0';

}

void account::statement(ostream &o) {

        o << "Account Statement" << endl;
        o << "  name = " << name << endl;
        o << "  balance = " << balance << endl;

}

checking::checking(const char* n, float b, float c) :
        account(n, b),
        charges(c) {

}

void checking::statement(ostream &o) {

        o << "Checking ";
        account::statement(o);
        o << "  charges = " << charges << endl;

}

ostream &operator<<(ostream &o, account &a) {

        a.statement(o);
        return (o);
}


int main() {

        checking reed("Kyle Reed", 5000);
```

```
        account &ra(reed);  //ref or
        account* pa(&reed); //pointer

        cout << ra << endl;
        cout << *pa << endl;

        return (0);
}
```

- Run Time Type Identification and Downcasting: static_cast is unsafe, use dynamic_cast or typeid. They rely on information stored in an object by the compiler whenever a direct or indirect base class contains a virtual function.

```
class account {

};

class checking : public account {

};


checking c;
checking* pc;
account* pa = &c;
account a;

#include <typeinfo>
//Downcasting
pc = static_cast<checking*>(pa); //result is a valid pointer
pc = dynamic_cast<checking*>(pa); //result is a valid pointer

pa = &a;
pc = static_cast<checking*>(pa); //result is a bad pointer, it will compile
                                 //it has wrong type
pc = dynamic_cast<checking*>(pa); //this gives 0 pointer

if (typeid(*pc) == typeid(checking))
        cout << "*ps is an checking object" << endl
```

21. Inheritance

- Single, derived class into its own header, implement files
- Private Derivation: A base class' public members are no longer visible to a derived class' clients and neither public nor protected members are visible to descendants, beyond the immediate derived class.

- Protected Derivation: A base class' public members are no longer visible to a derived class' clients but both public and protected members are visible to descendants.

```
class account
{
public:
	account();
	~account();

	account(const char* , float = 0);

	void statement();

private:
	char name[32];  //account owner
	float balance;  //account balance

};

class checking : public account {

public:
	checking();
	~checking();

	checking(const char* , float = 0, float = 5);

	void statement(); //this makes statement() in account hidden
	float get_charges();

private:
	float charges;  //charges for current month
};

account::account() :

	balance(0) {
	strncpy(name, "none", 32);
	name[31] = '\0'; //force terminating nul

}

account::account(const char* n, float b) :
	balance(b) {
	strncpy(name, n, 32);
	name[31] = '\0';
}


void account::statement() {

	cout << "Account Statement" << endl;
	cout << "  name = " << name << endl;
	cout << "  balance = " << balance << endl;
}
```

```
checking::checking() :
	charges(5) {
}

checking::checking(const char* n, float b, float c) :
	account(n, b), //invoke base class constructor
	charges(c) {

}

void checking::statement() {

	cout << "Checking ";
	account::statement(); //reuse base class function
	cout << "  charges = " << charges << endl;

}

float checking::get_charges() {
	return (charges);
}

void print_account(account* p) { //pointer

	p->statement();
}

void print_account(account &r) { //reference

	r.statement();
}

int main() {

	checking c; //calls account constructor than calls checking constructor
	checking c("Sue Smith", 1000.0);


	// UPCASTING

	//Assignment of Derived Class Object to Base Class Object.
	account a;
	a = c; //same effect as account a(c);
		//this is called Static binding

	//one can also use pointer or reference

	//Assigning Pointers to Derived Class Object to Base Class Pointer

	//this is the only place that C++ allows implicit cast operation, because
	//a pointer to a derived class object points to a direct or indirect
	//base class object as well

	account* pa = &c;

	pa->statement();
```

```
        print_account(&c); //pass by pointer
        print_account(c);  //pass by reference

        //the result is the same -> the derived class member funtions
        //'statement()' are lost, because of static binding.

        //To reach to the derived class member, we need dynamic binding
        //just add virtual to class account definition

        //say we have virtual void statement(); in the definition,
        // if we don't want dynamics binding, we do

        account a;
        a.statement();            //statically bind
    //or
        pa->account::statement(); //statically bind

       //we can also force the client application to use dynamic binding
       //by putting protect or private on checking::statement, then to invoke
       // checking::statement, one has to use dynamics binding.


        return (0);

} //checking destructor is called then call account
```

- Friends declared within a base class are not inherited by a derived class. Neither the copy constructor nor the assignment operator are inherited.
- Copy derived class will implicitly copy base class object as well as long as there IS an implicit default copy constructor, but if we have an assignment operator in the derived class, then we have to explicitly call the copy constructor in the parent class, so we need to do *THIS

```
class account {

public:
        account(const char* = "none", float = 0);
        ~account();
        void statement();

private:
        char* name;
        float balance;
};

class checking : public account {

public:
        checking(const char* = "none", float = 0, float = 5);
        checking &operator=(const checking &);
        void statement();

private:
        float charges;
};
```

```
account::account(const char* n, float b) :
	balance(b) {
	name = new char[strlen(n) + 1];
	strcpy(name, n);
}

account::~account() {
	delete[] name;
}

checking::checking(const char* n, float b, float c) :
	account(n, b),
	charges(c) {
}


checking &checking::operator=(const checking &c) {
	if (this != &c) {
		static_cast<account &>(*this) = c; //call assign op
	}
	return(*this);
}
```

- Multiple inheritance
- The definition cannot be cyclical. In addition, a direct base class cannot be specified more than once for any given derived class.
- A class derived from two or more base classes that have a virtual base class in common must override all virtual functions declared in the common base class if it is overridden in more than one of its direct base class branches. Also dominance rule: A class derived from two or more base classes that have a virtual base class in common and where only one of the base classes has overridden a virtual function from the common base class is allowed. The virtual function that is overridden in the one base class will dominate and will be used.

```
class account {

public:

	account(const char* = "none", float = 0);
	const char* get_name();
	float get_balance();

private:

	char name[32];
	float balance;

};



class savings : public account {
```

```
public:

	savings(const char* = "none", float = 0);
	float get_interest();

private:
	float interest;

};

class equity {

public:

	equity(const char* = "none", float = 0);
	float get_holdings();

private:

	char name[32];
	float holdings;

};

class assets : public savings, public equity {

public:
	assets(const char* = "none", float = 0, float = 0);
	float get_total();

private:
	float total;
};

class assets : public savings, public equity {  //gives the order of invoking
                                                //constr
public:
	assets(const char* n, float s) :
		savings(n, s),
		equity(n, s) {
	}
};
```

- Virtual inheritance – extension of multiple inheritance, used for inheritance hierarchies that share a common base class. Using virtual inheritance prevents duplicate base class, therefore virtual inheritance has no use within a single inheritance hierarchy. Virtual base classes are constructed before any of their derived classes. They are also constructed before any non virtual base classes. And, destructors are still invoked in the reverse order of constructors.

```
class account {

public:
	account(const char* = "none", float = 0);
```

```
        const char* get_name();
        float get_balance();

private:
        char name[32];
        float balance;
};


class checking : virtual public account {

public:
        checking(const char* = "none", float = 0, float = 5);
        float get_charges();

private:
        float charges;
};


class savings : virtual public account {

public:
        savings(const char* = "none", float = 0);
        float get_interest();

private:
        float interest;
};


class ibc : public checking, public savings {

public:
        ibc(const char* = "none", float = 0, float = 1000);
        float get_minimum();

private:
        float minimum;

};
```

- Abstract Class: no objects can be instantiated.
- A base class becomes an abstract class either by 1. making its constructor(s) protected or by 2. declaring a virtual function to be pure.

```
class account {
public:
        virtual void statement() = 0;
};
```

Advanced C++

22. Type Conversion

- Steps that compiler will try: trivial conversion -> promotion conversion -> built-in type conversion - > user defined type conversion -> error
- the lifetime of a temporary conversion helping object is from the time it is created until the end of the statement in which it was created
- `Using the Constructor as a Type Conversion.`

```
class name {

public:

	name(char* = "");	// change it to explicit constructor
	~name();
	const char* get_name(); //get pointer to name

private:
	char array[32];
};

void function(name obj) {
	cout << obj.get_name() << endl;
}

int main() {
	name obj;
	obj = "sue smith";	//implicitly convert char* to name
	cout << obj.get_name() << endl;

	function("sue smith"); //implicitly convert char* to name

	//if we change to explicit constructor, both above will fail
	//so we need Type Conversion Function or explicit conversion

	obj = (name)"sue smith";
	cout << obj.get_name() << endl;

	obj = static_cast<name>("sue smith");
	cout << obj.get_name() << endl;

	return (0);
}


//if we choose to do a user-defined type Conversion Function

typedef const char* pchar;
//make single identifier,  must be in the global namespace

class name {

public:
	explicit name(char* = "");
```

```
        ~name();

        operator pchar();           //conversion (name to char*)

        const char* get_name();     //get pointer to name

private:
        char array[32];
};

name::operator pchar() {        //conversion function
        cout << "conversion function pchar called" << endl;
        return array;
}

int main() {

        name obj("sue smith");
        const char* p;

        p = (pchar)obj;
        cout << p << endl;

        p = pchar(obj);
        cout << p << endl;

        p = static_cast<pchar>(obj);
        cout << p << endl;

        return (0);

}
```

23. Relationship among different classes

- Friends: allow non-member functions or member functions of other classes access to private data

```
 class node { //node for binary tree

        friend tree; //only tree class can create and access nodes

        //we can add just add friend member functions
        friend void tree::copy(node* &, const node*);
        friend void tree::destroy(node*);
        friend void tree::insert(const employee &);
        friend node* tree::add(node*, node*);
        friend void tree::traverse(const node*) const;

private:
        node() :                    //default constructor
                left(0),
                right(0) {
```

```
        }

        node(const employee &obj) : //one arg constructor
                emp(obj),
                left(0),
                right(0) {
        }

        employee emp;               //employee object
        node* left;                 //left child node
        node* right;                //right child node

};

class employee { //employee class

        friend class tree; //allow tree private access to employee class
        //this makes node class and employee to have mutual friend TREE

};

// implement
void tree::insert(const employee &emp) { //insert employee

        node* new_node = new node(emp);
        root = add(root, new_node);

}

node* tree::add(node* node_ptr, node* new_node) {

        if (node_ptr) {
                if (new_node->emp.salary < node_ptr->emp.salary)
                        node_ptr->left = add(node_ptr->left, new_node);
                else
                        node_ptr->right = add(node_ptr->right, new_node);
                return (node_ptr);
        }

        else
                return (new_node);
}
```

- Nested: A class defined inside another class' definition. Nesting restricts the scope of a class' name and significantly reduce the global namespace pollution. It is not an alternative to Friend. A nested class has no special access to any of the outer class' data or member functions. And, an outer class has no special access to a nested class' data or member functions.

```
//nest the node class within the tree class

class tree { //binary tree class
```

```
public:

	tree();
	tree(const tree &);
	~tree();

	void insert(const employee &);    //insert employee
	void write() const;               //write employee info

private:
	class node { //node for binary tree

	public:                          //make access public
		node() :                         //default constructor
			left(0),
			right(0) {
		}

		node(const employee &obj) :   //one arg constructor
			emp(obj),
			left(0),
			right(0) {
		}

		employee emp;                 //employee object
		node* left;                   //left child node
		node* right;                  //right child node

	};

	node* add(node*, node*);          //add in sorted order
	void traverse(const node*) const; //inorder traversal
	node* root;                       //root node of tree
};

void tree::insert(const employee &emp) { //insert employee

	node* new_node = new node(emp);
	root = add(root, new_node);
}

//use scope resolution operator & put node in scope of tree
tree::node* tree::add(node* node_ptr, node* new_node) {
	if (node_ptr) {
		if (new_node->emp.get_salary() <
			node_ptr->emp.get_salary())
			node_ptr->left = add(node_ptr->left, new_node);
		else
			node_ptr->right = add(node_ptr->right, new_node);
		return (node_ptr);
	}
	else
		return (new_node);
}
```

- Utility Operations as Non-member Static Functions, so they are invoked not through an object and they don't have to access to object data (because there is no this pointer associated with static functions). For good practices, implement them as static functions and place their definitions in the class implementation file, (not declaration in header file) then member function can invoke a non-member static utility function with a function call. For non-utility functions simply implement them as member static functions

```
class tree { //binary tree class
  public:
    tree();
    tree(const tree &);
    ~tree();

  private:
       class node {
         public:
                node() :
                       left(0),
                       right(0) {
                }

                node(const employee &obj) :
                       emp(obj),
                       left(0),
                       right(0) {
                }

                employee emp;
                node* left;                 //left child node
                node* right;                //right child node
         };

    node* root;
       friend static void copy(node* &, const node*);
       friend static void destroy(node*);
       friend static tree::node* add(node*, node*);
       friend static void traverse(const node*);

};

//always need scope because static has no THIS
static void copy(tree::node* &new_node, const tree::node* old_node) {
       if (old_node) {
              new_node = new tree::node(old_node->emp);
              copy(new_node->left, old_node->left);
              copy(new_node->right, old_node->right);
       }
}

tree::tree(const tree &tree) :  //copy constructor
       root(0) {
       copy(root, tree.root);
}
```

24. Exception handling

- Try-catch, exception specification

```
try {
	if (c != 'x')
		throw c;  //throw exception of type char
	if (i != 42)
		throw i;  //throw exception of type int
}

catch (char) { //catch char exception
}

catch (...) {  //catch all other exceptions
}
```

- Use library function set_terminate. It must not take any arguments, must not return data, it can only terminate by calling exit or abort, and it is not allowed to throw an exception.

```
include<exception>
void user_terminate() {  //user supplied terminate catch unexpected
	cout << "user terminate function calling exit" << endl;
	exit(1);			//abnormal program exit
}

int main() {
	int i;
	set_terminate(user_terminate); //install user function
	cout << "Enter an integer: ";
	cin >> i;

	if (i != 42)			//detect error condition
		throw i;			//throw an exception
	cout << "no throw was executed" << endl;

	return(0);
}
```

- Out of memory exception

```
#include <new>
void out_of_mem();		//user new handler callback

int main() {
	std::set_new_handler(out_of_mem); //install user new handler
	while (true)
		int *p = new int[1024];  //gobble up memory till gone
	return (0);
}
```

```
void out_of_mem() {
	cout << "programmer supplied new handler called" << endl;
	//free up space & return, throw bad_alloc,
	//or exit(1)
}
```

- Exception with class: check new bad_allc, index out of bound

```
 //version 1 -- no throw all handled
class dyn_a1 {
public:
	explicit dyn_a1(INDEX) throw(); //1D array of size i
	~dyn_a1() throw();
	int &operator[](INDEX) throw(); //subscript operator
private:
	dyn_a1(const dyn_a1 &);            // copy ctor
	dyn_a1 &operator=(const dyn_a1 &); // assign
	INDEX d1;                 //# of elements in 1D array
	int* a0;                  //base address of all elements
	int dummy;                //for out of bounds reference
};

//Implementation of dyn_a1 constructor and destructor
inline dyn_a1::dyn_a1(INDEX i) throw() :
	d1(i),                      //# of 1D array elements
	dummy(0) {
	a0 = new(nothrow) int[i]; //total # elements for 1D array
	if (a0 == 0) {            //check if new failed
		cerr << "new failed in class dyn_a1" << endl;
		d1 = 0;                 //set # elements to zero
	}
}
inline dyn_a1::~dyn_a1() throw() {
	delete[] a0;              //deallocate all array elements
}
//Implementation of subscript operator
inline int &dyn_a1::operator[](INDEX i) throw() {
	if (i<0 || i >= d1) {       //check if out of bounds
		cerr << "out of bounds at index " << i << endl;
		return (dummy);         //reference to dummy element
	}
	return (a0[i]);           //ith element in 1D array
}

//version 2 -- throw upto user to handle
struct bad_index {          //bad index exception type
	long index;
};

class dyn_a1 {
public:
	explicit dyn_a1(INDEX) throw(bad_alloc); //constructor
	~dyn_a1() throw();
	int &operator[](INDEX) throw(bad_index); //subscript op
private:
	dyn_a1(const dyn_a1 &);
	dyn_a1 &operator=(const dyn_a1 &);
```

```
        INDEX d1;
        int* a0;
};

//Implementation of dyn_a1 constructor and destructor
inline dyn_a1::dyn_a1(INDEX i) throw(bad_alloc) :
        d1(0) {                    //set to 0 in case of exception
        a0 = new int[i];           //total # elements for 1D array
        d1 = i;                    //# of 1D array elements
}
inline dyn_a1::~dyn_a1() throw() {
        delete[] a0;
}

//Implementation of subscript operator
inline int &dyn_a1::operator[](INDEX i) throw() {
        if (i<0 || i >= d1) {       //check if out of bounds
                bad_index e;           //create bad_index object
                e.index = i;           //save bad index
                throw(e);              //throw bad_index exception
        }
        return (a0[i]);           //ith element in 1D array
}
```

25. Template Class, Template Function

- Function Template v.s. Template Function: A function template is the definition of a parameterized family of functions; a template function is a particular instance of this family.
- Local types, types with no linkage, unnamed types (anonymous) or compound types with these constructs cannot be used as the actual template arguments for a template function being called. A string literal may also not be used as an actual template argument because it has no internal linkage.

```
//function template declaration
template <class TYPE_ID>
//or
template <typename TYPE_ID>
void function(TYPE_ID formal_arg);

//function call
int i; float f; double* ptr_k;
function(i);     //TYPE_ID is deduced to be an integer
function(f);     //TYPE_ID is deduced to be a float
function(ptr_k); //TYPE_ID is deduced as a pointer

//or
//function template declaration with return, with build-in type
template <class TYPE_ID>
TYPE_ID function(int formal_arg);

//function call
int i; float f; double* ptr_k;
```

```
i = function<int>(i);            //return type is int
f = function<float>(i);          //return type is a float
ptr_k = function <double*>(i);   //return type is a pointer

//The data type of non-type identifiers must be an integral type,
//an enumeration type, a pointer to an object, a reference to an
//object, a pointer to a function, a reference to a function, or a
//pointer to a member. They cannot be void or a floating point type.
template <data_type nontype_identifier>

//example
template <class TYPE1, class TYPE2>
void array_copy(TYPE1 dest[], TYPE2 source[], int size) {

       for (int i = 0; i < size; ++i)
              dest[i] = source[i];
}

//this will replace the above general template for the
//specified argument types. This is called a specialized template
//function. If we leave the "template<>" part out, it will be a
//regular function. It can too be used to hide
//the above general template, but then the
//regular function must declare before the general template
template<>
void array_copy(char* dest[], char* source[], int size) {

       for (int i = 0; i < size; ++i) {
              dest[i] = new char[strlen(source[i]) + 1];
              strcpy(dest[i], source[i]);
       }
}


//client can call this function
int int_array[100];
float real_array[100];

array_copy(real_array, int_array1, 100);
//we could say or even lever out “int[]”
array_copy<float[],int[]>(real_array, int_array1, 100);
```

- Good practices: use export to separate files. Without export, the definition of the template function must be included in the file where we invoke the function, i.e. main.cpp. And it will become an inline function.

```
//main.cpp

#include "t_func.h"

main() {
       t_funct<int, float>();  //template function call
}
```

```
//t_func.h

//declarations of the function template(s)

template<class TYPE_ID1, class TYPE_ID2>
void t_funct(TYPE_ID1, TYPE_ID2);


//t_func.cpp

//implementation of the function template(s)
export template<class TYPE_ID1, class TYPE_ID2>
void t_funct(TYPE_ID1 arg_1, TYPE_ID2 arg_2) { ... }
```

- Class Template v.s. template class:
- Specialized Template Class v.s. Partially Specialized Template Class: Specialized class templates are customized versions of a template class that work for situations.

```
 //class template declaration
template <char non_type, class TYPE_ID>
class t_class {

public:
	TYPE_ID function(int TYPE_ID);
};

//one can have Template Classes as Formal Arguments
template <template <actual_arguments> class_identifier> class something;

//with two default types
template <class TYPE1, int sz = 100, template<TYPE1>class TYPE2 = stack> class
list;

//then the client
list <int> object; //does two things: instantiate list class and
                        //created a list object

//Member functions can either be defined inside of a class template
//as inline members or separated from the class' definition.
//if outside must follow this: rewrite the complete template header
template <class TYPE1, int sz, template<TYPE1> class TYPE2>
list & list<TYPE1, sz, TYPE2>::operator = (const list &) {

	//function's definition

}

//with static member
template <class TYPE> class stack {
	static int data;
};

template <class TYPE>
int stack<TYPE>::data = 100;
//just like usual static member, each object of a particular template
//class instantiation shares the same static data members' memory.
```

```
//so 100 will apply to all stacks regardless their type
// Therefore, a more common approach is to supply the size of
//the stack as an argument to the constructor instead of as
//an argument to the template's argument list.
template class stack<int>;  //explicit instantiation


//for nest class, one can have Member Templates of a Class Template
template <class TYPE1, int sz, template<TYPE1> class TYPE2>class list {

	//a member template follows
	template <class TYPE3>
	int function(data_type arg);
};

//then use scope with double header
template <class TYPE1, int sz, template<TYPE1> class TYPE2>
template <class TYPE3>
int list<TYPE1, sz, TYPE2>::function(data_type arg) {

	//definition of the template function

}


//one can do Implicit Instantiation
template <class TYPE, int sz>
class stack {

public:
	stack();
	int push(const TYPE & data);
	TYPE & pop();

private:

	//this will not instantiated the list class
	//until data member is first used
	list <int, 100, stack> list_object;

};

//Using Operators with Class Templates, and
//node is a class template
list_object->template node<int> * ptr = new node<int>;

//Now we implement push() and use list template
template <class TYPE, int sz>
int stack<TYPE, sz>::push(const TYPE & data) {

	//if this is the first usage of the list_object member,
	//then the list class is instantiated
}
```

- Customizing Class Templates

- Partially specialized class templates are semi-customized versions of a template class where some of the type and non-type dependencies remain unspecified. When all formal arguments are given customized types and/or values, we call this an explicit specialization.

```
template <class TYPE>
class stack {

private:
	TYPE * stack_array;
	const int stack_size;
	int stack_index;

public:
	stack(int size = 100) : stack_size(size), stack_index(0) {
		stack_array = new TYPE[size];

	}
	push(TYPE item);
	TYPE pop(void);
};


//A template member function
template <class TYPE>
void stack<TYPE>::push(TYPE item) {

	if (stack_index < stack_size) {
		stack_array[stack_index] = item;
		++stack_index;
	}
}

//An explicit specialization of a member function
template <>
void stack <char *>::push(char * item) {

	if (stack_index < stack_size) {
		stack_array[stack_index] = new char[strlen(item) + 1];
		strcpy(stack_array[stack_index], item);
		++stack_index;
	}
}

//or write a partial specialization of that class:
template <class TYPE>
class stack <char *> {
private:

	char ** stack_array;
	int stack_index;

public:
	stack(int size = 100) : stack_index(0) {

		stack_array = new char *[size];
	}
```

```
        void push(const char * item) {

                stack_array[stack_index] = new char[strlen(item) + 1];
                strcpy(stack_array[stack_index], item);
                ++stack_index;
        }
};
```

- Good practices, use export

```
//t_class.h
//declarations of the function template(s)
template<class TYPE_ID1, class TYPE_ID2>
class t_class
{
public:
        t_class();
        t_class(const t_class &);
        ~t_class();
        void t_member(TYPE_ID1, TYPE_ID2);

private:
        TYPE_ID1 data_1;
        TYPE_ID2* ptr_2;
};


//t_class.cpp
//implementation of the member function
export template<class TYPE_ID1, class TYPE_ID2>
t_class() { }

export template<class TYPE_ID1, class TYPE_ID2>
t_class(const t_class & source) { }

export template<class TYPE_ID1, class TYPE_ID2>
~t_class() { }

export template<class TYPE_ID1, class TYPE_ID2>
void t_member(TYPE_ID1 arg_1, TYPE_ID2 arg_2) { }
```

Advanced C#

Examples from *Exam Ref 70-483 Programming in C#*, By Wouter de Kort

26. Multithreading

- Task Parallel library--TPL

```
Task<int> t = Task.Run(() => { return 42; }).ContinueWith
                ((i) => { return i.Result* 2; });

TaskFactory tf = new TaskFactory
```

```
                    (TaskCreationOptions.AttachedToParent,
                    TaskContinuationOptions.ExecuteSynchronously);

Parallel.For(0, 10, i => { Thread.Sleep(1000); });

ThreadPool.QueueUserWorkItem((s) => { Console.WriteLine(
              "Working on a thread from threadpool"); });
```

- use async and await

```
public static async Task<string> DownloadContent()
        {
            using (HttpClient client = new HttpClient())
            {
                string result = await client.GetStringAsync(
                    "http://www.microsoft.com");
                return result;
            }
        }
```

- implement locking;

```
object lockA = new object();
            object lockB = new object();
            lock (lockB)
            {
                lock (lockA)
                {
                }
            }
```

27. Encryption

- Choose an appropriate encryption algorithm; manage and create certificates; implement key management

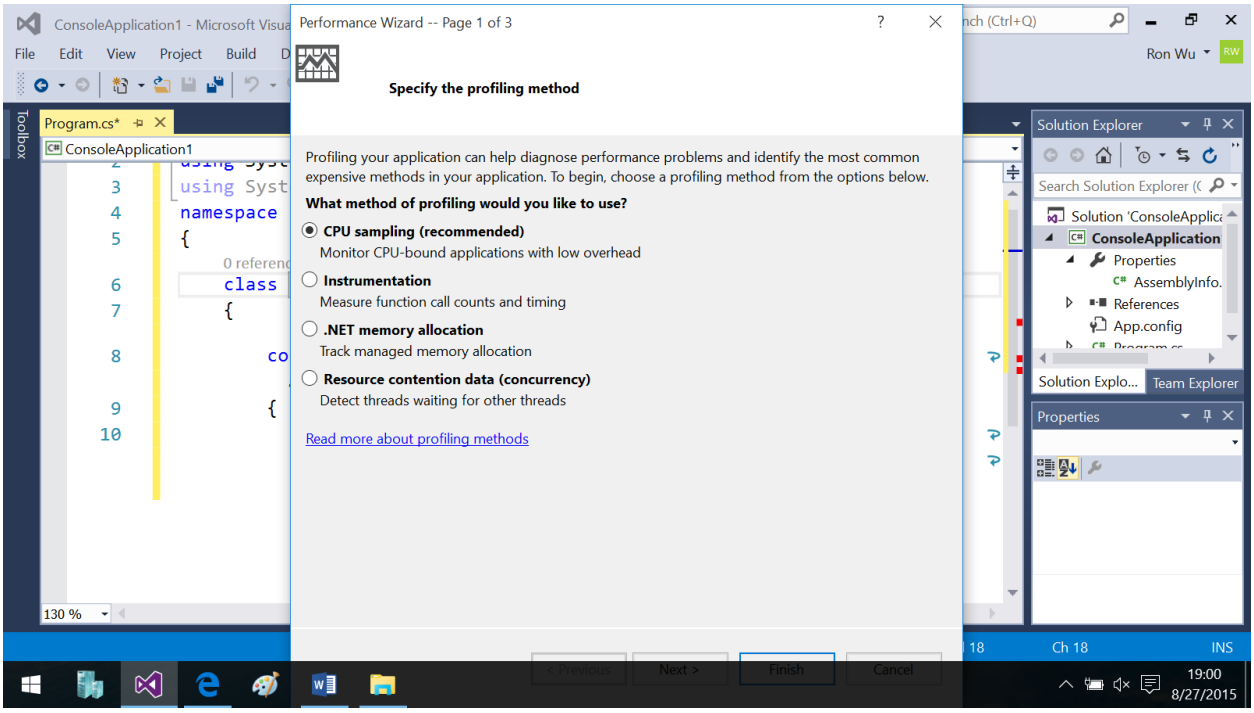
```
using System.Security.Cryptography;
using System.Security.Cryptography.X509Certificates;

public static void SignAndVerify()
{
    string textToSign ="Test paragraph";
    byte[] signature = Sign(textToSign,"cn = WouterDeKort");
    Console.WriteLine(Verify(textToSign, signature)); }
static byte[] Sign(string text, string certSubject)
{
    X509Certificate2 cert = GetCertificate();
    var csp = (RSACryptoServiceProvider)cert.PrivateKey;
```

```
    byte[] hash = HashData(text);
    return csp.SignHash(hash, CryptoConfig.MapNameToOID("SHA1"));
}
static bool Verify(string text, byte[] signature)
{
    X509Certificate2 cert = GetCertificate();
    var csp = (RSACryptoServiceProvider)cert.PublicKey.Key;
    byte[] hash = HashData(text);
    return csp.VerifyHash(hash, CryptoConfig.MapNameToOID("SHA1"), signature);
}
private static byte[] HashData(string text)
{
    HashAlgorithm hashAlgorithm = new SHA1Managed();
    UnicodeEncoding encoding = new UnicodeEncoding();
    byte[] data = encoding.GetBytes(text);
    byte[] hash = hashAlgorithm.ComputeHash(data);
    return hash;
}
private static X509Certificate2 GetCertificate()
{
    X509Store my = new X509Store("testCertStore", StoreLocation.CurrentUser);
    my.Open(OpenFlags.ReadOnly);
    var certificate = my.Certificates[0];
    return certificate;
}
```

- hash

```
UnicodeEncoding byteConverter = new UnicodeEncoding();
SHA256 sha256 = SHA256.Create();
string data ="A paragraphof text";
byte[] hashA = sha256.ComputeHash(byteConverter.GetBytes(data));
data = "A paragraphof changedtext";
byte[] hashB = sha256.ComputeHash(byteConverter.GetBytes(data));
data = "A paragraphof text";
byte[] hashC = sha256.ComputeHash(byteConverter.GetBytes(data));
Console.WriteLine(hashA.SequenceEqual(hashB)); // Displays: false
Console.WriteLine(hashA.SequenceEqual(hashC)); // Displays: true
```

28.Profiling



diagnostics in an application

- PerformanceCounters

```
        if (CreatePerformanceCounters()) {
            Console.WriteLine("Createdperformancecounters");
            Console.WriteLine("Pleaserestartapplication");
            Console.ReadKey();
            return;
        }
        var totalOperationsCounter = new PerformanceCounter(
              "MyCategory",
              "# operations executed",
              "",
              false);
```

29.Serialize and deserialize data

- JavaScript Object Notation (JSON) and Extensible Markup Language (XML) data

```
//xml sample
<? xmlversion="1.0" encoding="utf-8" ?>
<people>
      <person firstName ="John" lastName="Doe">
        <contactdetails>
```

```
            <emailaddress>john @unknown.com</emailaddress>
        </contactdetails>
      </person>
      <person firstName ="Jane" lastName="Doe">
        <contactdetails>
             <emailaddress>jane@unknown.com</emailaddress>
             <phonenumber>001122334455</phonenumber>
        </contactdetails>
      </person>
 </people>

//xmlread
using (StringReader stringReader = new StringReader(xml))
{
    using (XmlReader xmlReader = XmlReader.Create(stringReader,
           new XmlReaderSettings() {IgnoreWhitespace = true }))
    {
        xmlReader.MoveToContent();
        xmlReader.ReadStartElement("People");
        string firstName = xmlReader.GetAttribute("firstName");
        string lastName = xmlReader.GetAttribute("lastName");
        Console.WriteLine("Person: {0} {1}", firstName, lastName);
        xmlReader.ReadStartElement("Person");
        Console.WriteLine("ContactDetails");
        xmlReader.ReadStartElement("ContactDetails");
        string emailAddress = xmlReader.ReadString();
        Console.WriteLine("Email address: {0}", emailAddress);
    }
}

//xmlwriter
StringWriter stream = new StringWriter();
using (XmlWriter writer = XmlWriter.Create(stream,
           new XmlWriterSettings() { Indent = true }))
{
    writer.WriteStartDocument();
    writer.WriteStartElement("People");
    writer.WriteStartElement("Person");
    writer.WriteAttributeString("firstName", "John");
    writer.WriteAttributeString("lastName", "Doe");
    writer.WriteStartElement("ContactDetails");
    writer.WriteElementString("EmailAddress", "john@unknown.com");
    writer.WriteEndElement();
    writer.WriteEndElement();
    writer.Flush();
}
Console.WriteLine(stream.ToString());
```

- Binary serialization

```
[Serializable]
public class Person
{
```

```
    public int Id { get; set; }
    public string Name { get; set; }
    private bool isDirty = false;
}
Person p = new Person { Id = 1, Name ="JohnDoe" };
IFormatter formatter = new BinaryFormatter();
using (Stream stream = new FileStream("data.bin", FileMode.Create))
{
    formatter.Serialize(stream, p);
}


[Serializable]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
}

XmlSerializer serializer = new XmlSerializer(typeof(Person));
string xml;
using (StringWriter stringWriter = new StringWriter())
{
    Person p = new Person { FirstName ="John", LastName ="Doe", Age = 42 };
    serializer.Serialize(stringWriter, p);
    xml = stringWriter.ToString();
}
Console.WriteLine(xml);
```

Search and Sorting

30.Sequential Search O(n), Binary Search O(log(n))

- Ordered list

```
def rec_binarySeach(arr, val):
    if len(arr)==0:
        return
    mid = len(arr)//2
    if arr[mid]==val:
        return True
    if arr[mid] > val:
        return rec_binarySeach(arr[:mid-1], val)
    rec_binarySeach(arr[mid+1:], val)
```

- Ordered list

```
def rec_binarySeach(arr, val):
    if len(arr)==0:
        return
    mid = len(arr)//2
    if arr[mid]==val:
```

```
        return True
    if arr[mid] > val:
        return rec_binarySeach(arr[:mid-1], val)
    rec_binarySeach(arr[mid+1:], val)
```

31. Bubble sort

- O(n^2)

```
def BubbleSort(arr):
    if len(arr)<2:
        return
    i = 0
    for i in range(len(arr)):
        if arr[i]>arr[i+1]:
            arr[i], arr[i+1] = arr[i+1], arr[i]
    BubbleSort(arr[:-1])

#second version
def BubbleSort(arr):
    for i in range(len(arr)-1,0,-1):
        for j in range(i):
            if arr[j]>arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

32. Selection sort

- O(n^2)

```
def SelectionSort(arr):
    for i in range(len(arr)-1,0,-1):
        maxIndex = i
        maxVal = arr[maxIndex]
        for j in range(i):
            if arr[j]>maxVal:
                maxVal, maxIndex = arr[j], j
        arr[i], arr[maxIndex] = arr[maxIndex], arr[i]
```

33. Counting sort -- special

- List of positive number and given the maxVal

```
 def CountingSort(list_n, maxVal):

    list_count = [0]*(maxVal+1)
    for n in list_n:
        list_count[n] += 1

    list_sorted=[]
    for i in range(maxVal+1):
```

```
        for j in range(list_count[i]):
            list_sorted += [i]
    return list_sorted
solution([2,3, 4,2,5, 4, 3], 5)
#[2, 2, 3, 3, 4, 4, 5]
```

34.Insertion sort

- O(n^2)

```
def InsertionSort(arr):
    for i in range(1,len(arr)):
        insertVal = arr[i]
        j = i
        while j > 0 and arr[j-1]>insertVal:
            arr[j]= arr[j-1]
            j -= 1
        arr[j] = insertVal
    return arr
```

35.Shell sort

-

```
def ShellSort(arr):

    '''#sublists size -> gap'''
    shell = [5,3,1]

    for i in shell:
        for j in range(i):
            k = j
            arr1= []
            while k < len(arr):
                arr1 += [arr[k]]
                k = k + i
            arr2 = InsertSort(arr1)

            k = j
            for k1 in range(len(arr2)):
                arr[k] = arr2[k1]
                k = k + i
    return arr


# or
def InsertionSort(arr, start = 0, gap = 1):
    for i in range(start+gap,len(arr)):
        insertVal = arr[i]
        j = i
        while j > start and arr[j-gap]>insertVal:
            arr[j]= arr[j-gap]
```

```
            j -= gap
        arr[j] = insertVal
    return arr

def ShellSort(arr):
    shell = [5,3,1]
    for p in shell:
        for i in range(p):
            InsertionSort(arr, i, p)
```

36. Merge sort

- O(nlog(n))

```
def InsertSort(arr, start = 0, end =-1, gap = 1):
    if end == -1:
        end = len(arr)-1
    for i in range(start,end+1):
        insertVal = arr[i]
        j = i
        while j > start and arr[j-gap]>insertVal:
            arr[j]= arr[j-gap]
            j -= gap
        arr[j] = insertVal
    return arr


def MergeSort(arr, start = 0, end = -1):
    if end == -1:
        end = len(arr) -1
    if end<=start + 1 :
        return

    MergeSort(arr, start, end - end//2)
    MergeSort(arr, start + end//2+1,end)

    InsertSort(arr, start, end)
```

37. Quick sort

- O(nlog(n))

```
def QuickSort(arr, start=0, end=-1):

    if end == -1:
        end = len(arr) -1
    if start>=end-1:
        return
    pivot = arr[start]
```

```
    left = start
    right = end

    while left <= right:
        while left <= right and arr[left] <= pivot:
            left += 1

        while left <= right and arr[right] >= pivot:
            right -= 1

        if left < right:
            arr[left], arr[right] = arr[right], arr[left]
    if right != start:
        arr[start], arr[right] = arr[right], arr[start]

    QuickSort(arr, start, right-1)
    QuickSort(arr, right + 1, end)
```

```
//Quick Sort uses Template
#include <iostream>
using namespace std;
#include "qsort.h"

int    ia[] = { 46,   28,   35,   44,   15,   22,   19 };
double da[] = { 46.5, 28.9, 35.1, 44.6, 15.3, 22.8, 19.4 };

int main() {

       const int isize = sizeof(ia) / sizeof(int);
       const int dsize = sizeof(da) / sizeof(double);

       qsort(ia, 0, isize - 1);       // integer qsort

       for (int i = 0; i < isize; ++i)
              cout << ia[i] << endl;

       qsort(da, 0, dsize - 1);       // double qsort
       for (i = 0; i < dsize; ++i)
              cout << da[i] << endl;

       return (0);
}

//qsort.h
template <class TYPE>
inline void swap(TYPE v[], int i, int j) {

       TYPE temp;
       temp = v[i];
       v[i] = v[j];
       v[j] = temp;
}

template <class TYPE>
void qsort(TYPE v[], int left, int right) {

       if (left < right) {
```

```
                swap(v, left, (left + right) / 2);
                int last = left;

                for (int i = left + 1; i <= right; ++i)
                        if (v[i] < v[left])
                                swap(v, ++last, i);

                swap(v, left, last);

                qsort(v, left, last - 1);
                qsort(v, last + 1, right);
        }
}
```

Algorithm

38. Recursion

- factorial

```
def factorial(n):
    if n==0 or n==1:
        return 1
    return n * factorial(n-1)
```

- Split words against the dictionary

```
def word_split(s, dic, output):
    if s in dic:
        output.append(s)
        return True
    if len(s)==1 & (s not in dic):
        del output[:]
        return False
    for i in range(1,len(s)):
        if s[:i] in dic:
            output.append(s[:i])
            if word_split(s[i:], dic, output):
                return True
            else:
                del output[:]
    del output[:]
    return False

#second solution
def word_split(s, dic, output):
    if len(s) == 0:
        return True

    for word in dic:
        if s.startswith(word):
            output.append(word)
            if word_split(s[len(word):], dic, output):
                return True
            else:
```

```
                output = []
    output = []
    return False


output = []
print(word_split('abcdcdes', ['abcd','abc','cd','es'], output))
print(output)

#True
#['abcd', 'cd', 'es']
```

- Permute string

```
 def permute( s, out, head = ''):
    if len(s)==1:
        out.append(head+s)
        return

    for i in range(len(s)):
        permute( s[:i]+s[i+1:], out, head + s[i] )

out = []
permute("abc", out)
print(out)
#['abc', 'acb', 'bac', 'bca', 'cab', 'cba']

# or simply
import itertools
for p in itertools.permutations("abc", 3):
    out += [''.join(p)]
```

39. Amortization

- E.g. Dynamic array

```
$$\text{Amortized Cost} = \frac{(1+1+...)+(1+2+4+...)}{n}  = O(1)$$
```

40. Divide-and-conquer

- Merge sort, shell sort, quick sort

41. The greedy method

- Dijkstra (Fibonacci heap), http://interactivepython.org/runestone/static/pythonds/Graphs/DijkstrasAlgorithm.html

42. Dynamic programming

- Recursion + memorization

```
factorial_memo = {}

def factorial(k):

    if k < 2:
        return 1

    if not k in factorial_memo:
        factorial_memo[k] = k * factorial(k-1)

    return factorial_memo[k]

#or WRAP the function
class Memoize:
    def __init__(self, f):
        self.f = f
        self.memo = {}
    def __call__(self, *args):
        if not args in self.memo:
            self.memo[args] = self.f(*args)
        return self.memo[args]

def factorial(k):
    if k < 2:
        return 1
    return k * factorial(k - 1)

factorial = Memoize(factorial)

#or use decorator
@Memoize
def factorial(k):
    for i in range(k+1):
        a = i*a
    return a

#use generator
a, i = 1, 0
def factorial():
    global a, i
    while True:
        i += 1
        a = a*i
        yield a, i

f = factorial()
next(f)
next(f)
next(f)
next(f)
```

Reference

43. Books

- Stephen Prata, *C++ Primer Plus*, (with Source Code) 6th Edition, Addison-Wesley 2011
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, (Examples in C# .Net, in Java) 1st ed Addison-Wesley 1994
- Thomas Cormen, Charles Leiserson, Ronald Rivest and Clifford Stein, *Introduction to Algorithms*, Third Edition MIT Press
- Wouter de Kort, *Exam Ref 70-483 Programming in C#*, Microsoft Press
- Gayle Laakmann McDowell*, Cracking the Coding Interview: 150 Programming Questions and Solutions*, Careerup 2011

44. Courses

- Erik Demaine, Srinivas Devadas, *Introduction to Algorithms*, MIT
- Tim Roughgarden, *Algorithms: Design and Analysis*, Part 1, Part 2, Stanford
- Josh Hug, *CS 61B Data Structures* (Java), Berkeley
- Jose Portilla, *Python for Data Structures, Algorithms, and Interviews* (with Py code), Udemy