COMP 152H: OOP and Data Structures (Honors Track)
Spring Semester 2011
Final Exam
Instructor: Chi Keung Tang
Saturday, May 21, 2011
16:30-19:30
Room 2407

This is a **CLOSED-BOOK-CLOSED-NOTES** exam consisting of **eight** (8) problems. Follow the instructions carefully. Please write legibly in the space provided. Keep the exam booklet stapled.

K E Y

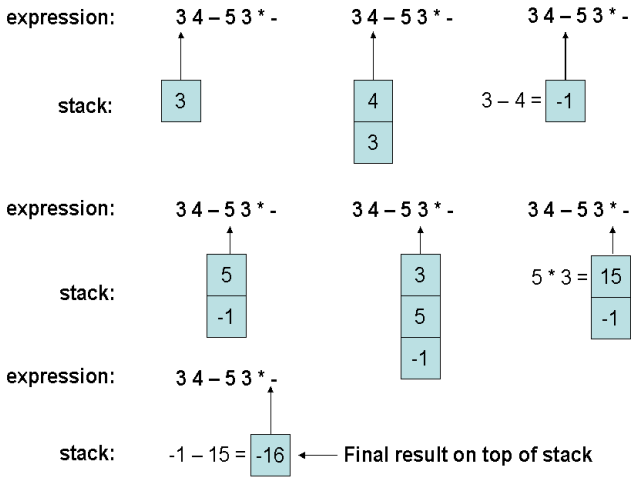
Problem	Points	your score
1 INTERFACE VS IMPLEMENTATION	10	
2 OBJECTS CONSTRUCTION & DESTRUCTION	10	
3 FUNCTION OBJECTS	10	
4 ANALYSIS (HEAP, AVL, BST)	15	
5 SORTING, COMPLEXITY	15	
6 GRAPH	10	
7 TREE	15	
8 HASH TABLE	15	
Total	100	

1 Interface vs Implementation

Write a program to evaluate expressions in *Reverse Polish Notation (RPN)* or *postfix* notation, using the `stack` class and other classes provided for you. The evaluation rule is:

1. Evaluate the given expression from left to right.
2. At each occurrence of an operator, apply it to the two operands to the immediate left and replace the sequence of two operands and one operator by the resulting value.

An example is shown in the figure below.



To simplify the problem, we make use of the following `Token` class and the associated facilities. A tokenizer is a program that reads the input, and tokenizes the input string into a set of tokens which are of the following types:

```
enum tokenType { operandToken, operatorToken, eolToken, eofToken, badToken };
```

Each token will be one of the enumerated types. An operand will be an `operandToken`; an operator will be an `operatorToken`. The end of a line and end of the input file will be signaled by the special `eolToken` and `eofToken`. Finally, if the tokenizer cannot interpret the input, it will mark it as a `badToken`, leaving it to the client to respond appropriately.
The operators are classified using the following declaration:

```
enum operatorType { none, add, subtract, multiply, divide };
```

The `none` type is used if the client mistakenly requests the operator type for a token that's not an operator.
The code below contains the class definition that defines the public interface for the `Token` class:

```
#ifndef TOKEN_H
#define TOKEN_H

class Token {
public:
  tokenType     nextToken();   // get next token from the input stream
  operatorType  getOperator(); // return operator
  double        getOperand();  // return operand

// other declarations...
};

#endif
```

The `Stack` class provided is a class template:

```
#ifndef STACK_H
#define STACK_H

template <class ElementType>
class Stack {
public:
  Stack ();
  void push (ElementType e);
  ElementType pop();     // pop an element
  ElementType top();     // get the top element; stack does not change
  bool isEmpty();        // check stack emptiness
  void clear();          // clear stack
private:
  // private data structures...
};

#endif
```

You must use the following skeleton and complete the program. Use the functions provided to keep your code simple – writing complex and buggy codes are not recommended.

```
#include "utilities.h" // enumerated types definitions
#include "stack.h"     // Stack class
#include "token.h"     // token class

int main()
{
  Token t;
  Stack <double> s;
  double op1, op2;
  bool done(false);

  while (!done) {
    switch (t.nextToken()) {
      case operandToken:









      case operatorToken:
        // op2 is the top of the stack, op1 is the next value down
        // first, have to make sure there are two items to pop
```

```
    case eofToken:
      done = true;   // break intentionally omitted here
    case eolToken:   // if there's something in stack, display it
      if (!s.isEmpty())
        fprintf (stdout, "--> %f\n",  s.pop());
        s.clear();  // clear stack for the next line
      break;
    case badToken:
      fprintf (stderr, "Input error!\n");
      break;
  }
}
return 0;
```

Solution:

```
#include "utilities.h" // enumerated types definitions
#include "stack.h"     // Stack class
#include "token.h"     // token class
int main()
{
  Token t;
  Stack <double> s;
  double op1, op2;
  bool done(false);
  while (!done) {
    switch (t.nextToken()) {
      case operandToken:
        s.push(t.getOperand());
        break;
      case operatorToken:
        // op2 is the top of the stack, op1 is the next value down
        // first, have to make sure there are two items to pop
        if ( (s.isEmpty()) || (op2 = s.pop() && s.isEmpty() ))
          fprintf (stderr, "Not enough operands for operator!\n");
        else {  // get op1, then apply appropriate operator
          op1 = s.pop();
          switch (t.getOperator()) {
            case add:      s.push (op1 + op2); break;
            case subtract: s.push (op1 - op2); break;
            case multiply: s.push (op1 * op2); break;
            case divide:
              if (op2 == 0)
                fprintf (stderr, "Division by zero!\n");
              else
                s.push (op1 / op2);
              break;
           }
        }
        break;
      case eofToken:
        done = true;   // break intentionally omitted here
      case eolToken:   // if there's something in stack, display it
        if (!s.isEmpty())
          fprintf (stdout, "--> %f\n",  s.pop());
        // s.clear();  // clear stack for the next line
        break;
      case badToken:
        fprintf (stderr, "Input error!\n");
        break;
    }
  }
  return 0;
}
```

2 Objects Construction/Destruction

What is the output of the following program?

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

class Hand {
public:
   Hand() { cout << "H" << " ";}
   ~Hand() { cout << "~H" << " ";}
};

class Watch {
public:
   Watch() { cout << "W" << " ";}
   ~Watch() { cout << "~W" << " ";}
private:
   Hand h;
};

class Father {
public:
   Father() { cout << "F" << " ";}
   ~Father() { cout << "~F" << " ";}
private:
   Watch w;
};

class Son : public Father {
public:
   Son() { cout << "S" << " ";}
   Son(const Watch& iw) : w(iw) { cout << "Sw" << " "; };
   ~Son() { cout << "~S" << " ";}
private:
   Watch w;
};

int main ()
{
  Watch watch;
  cout << "son 1" << endl; Son s1;
  cout << "son 2" << endl; Son s2(watch);
  return 0;
}
```

Solution:

```
H W son 1
H W F H W S son 2
H W F Sw ~S ~W ~H ~F ~W ~H ~S ~W ~H ~F ~W ~H ~W ~H
```

3 Function Objects

1. Fill a `vector` of 100 elements with random numbers ranging from 1 to 50. In your answer, you are required to use STL container class `vector`, STL algorithm `generate`, and the `RandomNumber` function. The prototypes of `generate` and `rand` are as follows:

```
int RandomNumber() { return (rand()%50 + 1); }

template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

   Solution:

```
  vector<int> V(100);
  generate(V.begin(), V.end(), RandomNumber);
```

2. Define a class that allows for a function object to be instantiated, so that the following code, which sorts a vector of double by magnitude, i.e. ignoring the elements' signs, can compile and run. Assume all appropriate header files have been included.

```
    vector<double> V;
    ...
    sort(V.begin(), V.end(), less_mag());
```

   Solution:

```
class less_mag {
public:
  bool operator()(double x, double y) { return fabs(x) < fabs(y); }
};
```

3. Define a class that allows for a function object to be instantiated, so that the following code can compile and run. Assume all appropriate header files have been included.

```
vector<double> V;
...
adder result = for_each(V.begin(), V.end(), adder());
cout << "The sum is " << result.get_sum() << endl;

// note in the final result.sum is directly accessed which is bad because
// it implies that sum is public; so I made a change here
// if you made sum public it is OK in exam.
// if you made sum private it is OK (scores the same) despite the program won't
// compile; I'm giving the benefit of doubts here when grading.
```

   Solution:

```
class adder {
public:
  adder() : sum(0) {}
  void operator()(double x) { sum += x; }
  double get_sum() { return sum; }
private:
  double sum;
};
```

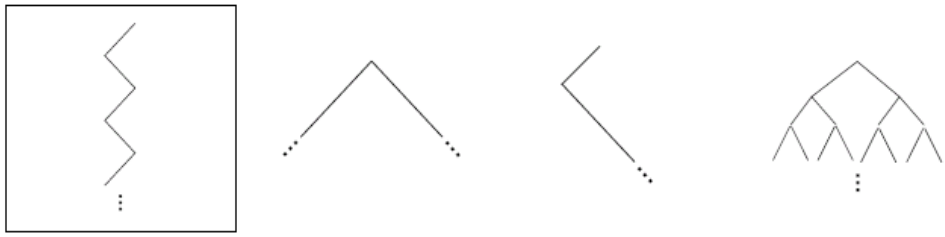
4 Analysis (Heap, AVL, BST)

1. Suppose we have a min-Heap A and a max-Heap Z, each of which store integers. Assume also that all the keys in A are less than all the keys in Z, and A and Z have the same number of keys. (Note: a max-Heap is just a Heap where all paths from root to leaf are decreasing.) Finally, assume that we have a Binary Search Tree class called `BST` with all the usual member functions.

   Consider this function `play`:

```
BST play(minHeap A, maxHeap Z) {
  BST B;
  while (!A.isEmpty && !Z.isEmpty) {
    B.insert(Z.removeMax());
    B.insert(A.removeMin());
  }
  return B;
}
```

   (a) Circle the illustration that *best* describes the `BST` returned by `play`.



   (b) Analyze (i.e., not just state) the total running time of `play` if each Heap contains n items.

   Solution:

   For simplifying the run-time analysis we can write the code as following:

```
BST play(minHeap A, maxHeap Z) {
  BST B;
  while(!A.isEmpty() && !Z.isEmpty()) {
    N=Z.removeMax();
    B.insert(N);
    N=Z.removeMin();
    B.insert(N);
  }
  return B;
}
```

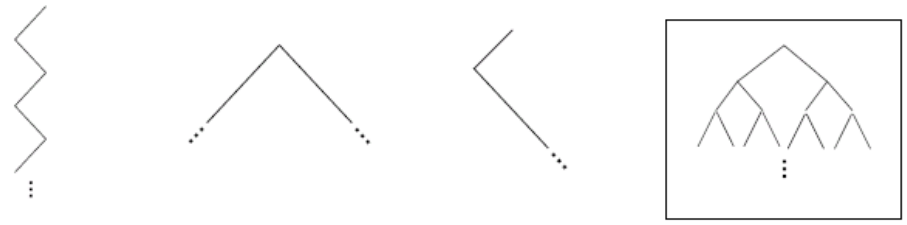
It is obvious that the while loop goes through all n nodes of `minHeap` and `maxHeap`. All the operations inside the loop are done sequentially, so we need to find the maximum order of the operations inside the loop and just multiply the maximum order by n for finding the total running time. `removeMax` and `removeMin` from heap each takes $O(\log(n))$. Insert into `BST` takes $O(n)$ in the worse case. Therefore the maximum order of operations inside the loop is $O(n)$. By multiplying $n * O(n)$ the order is $O(n^2)$.

2. Suppose we have a min-Heap A and a max-Heap Z, each of which store integers. Assume also that all the keys in A are greater than all the keys in Z, and A and Z have the same number of keys. (Note: a max-Heap is just a Heap where all paths from root to leaf are decreasing.) Finally, assume that we have an AVL tree class called `AVLTree` with all the usual member functions.

   Consider this function `work`:

```
AVLTree work (minHeap A, maxHeap Z) {
  AVLTree B;
  while (!A.isEmpty && !Z.isEmpty) {
    B.insert(Z.removeMax());
    B.insert(A.removeMin());
  }
  return B;
}
```

   (a) Circle the illustration that *best* describes the AVL Tree returned by `work`.



   (b) Analyze (i.e., not just state) the total running time of `work` if each Heap contains n items.
   Solution: For simplifying the run-time analysis we can write the code as following:

```
AVLTree work(minHeap A, maxHeap Z) {
AVLTree B;
while(!A.isEmpty() && !Z.isEmpty()) {
  N=Z.removeMax();
  B.insert(N);
  N=Z.removeMin();
  B.insert(N);
}
return B;
}
```

   It is obvious that the while loop goes through all n nodes of `minHeap` and `maxHeap`. All the operations inside the loop are done sequentially, so we need to find the maximum order of the operations inside the loop and just multiply the maximum order by n for finding the total running time. `removeMax` and `removeMin` from heap each takes $O(\log(n))$. Insert

into `AVLTree` takes $O(\log n)$ in the worse case. Therefore the maximum order of operations inside the loop is $O(\log(n))$. By multiplying $n * O(\log n)$ the order is $O(n \log(n))$.

5 Sorting, Complexity

(a) Order these functions in order of asymptotic growth rate, with the most rapidly growing first. If two of them have the same asymptotic growth rate, state that fact. No proofs are needed.

$$\log(n^2) \qquad 0.000001n \qquad \log n \qquad 2^{2^n} \qquad 4^n \qquad n\log(n) \qquad \log(2^n)$$

Solution:

$$\underbrace{\log n \qquad \log(n^2)}_{\text{same growth rate}} \qquad \underbrace{0.000001n \qquad \log(2^n)}_{\text{same growth rate}} \qquad n\log(n) \qquad 4^n \qquad 2^{2^n}$$

(b) Simulate counting sort on the keys below.

$$10 \quad 3 \quad 6 \quad 4 \quad 8 \quad 7$$

Solution:

(c) Describe in one sentence in terms of the key size when using counting sort is a bad idea.

Solution:

When the key size is not linearly bounded by the number of elements.

OR

When the key size is larger than the number of elements (not precise but still score full points).

(d) Recall the complexity of running Merge Sort on an array of n values.

$$T(n) = \Theta(n \log n)$$

Consider a variant of the Merge Sort algorithm, where the array is split in three parts (instead of two) that are then sorted recursively before being merged back together. What is the complexity of merging three sorted arrays of size $n/3$ in one sorted array of size n? No derivation is needed.

Solution:

It is still linear (i.e. $\Theta(n)$). For each element of the final array, we have to do two comparisons to find the smallest elements from the front of each array.

(e) Give a recursive definition (or recurrence relation) of the complexity of this "3-part" Merge Sort algorithm.

Solution:

$$T(n) = 3T(n/3) + \Theta(n)$$

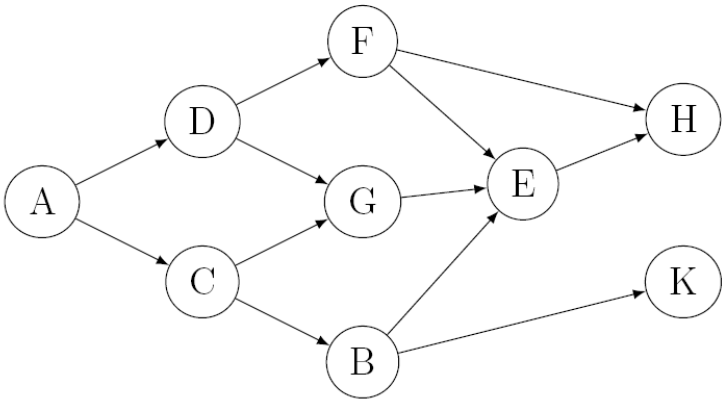
(f) What is the solution of this recursive equation. Show your derivation.

Solution:

$$T(n) = \Theta(n \log n)$$

6 Graph

Give 2 possible DFS traversals of the graph below, listing the nodes in the order they traversed. *A* should be the starting vertex.



Solution: Any two of the following exhaustive list of traversals is OK.

- ACBKEHGDF
- ACBEHKGDF
- ACGEHBKDF
- ADGEHFCBK
- ADFEHGCBK
- ADFHEGCBK

7 Trees

(a) Is the operation of insertion in a binary search tree commutative in the sense that inserting x and then y into a binary search tree gives the same tree as inserting y and then x? Argue why it is so (in no more than *five* lines) or give a counter-example.

Solution:

No, insertion is not commutative in this sense. A counter-example would be insertion of the elements 6 and 7 into the tree containing only the element 5. The two trees depicted below shows the two possible trees resulting from this.

(b) Insert the following number sequence into an empty B^+ tree with $M = 5, L = 4$, according to the given order. Apply the insertion procedure described in the lecture notes. Draw the intermediate B^+ trees after inserting

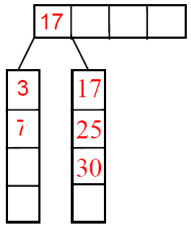
(i) 3,
(ii) 38,
(iii) 20,
(iv) 10

in the following sequence:

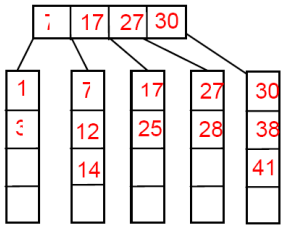
7 30 17 25 3 41 14 12 27 1 28 38 15 29 23 6 20 21 10

Solution:

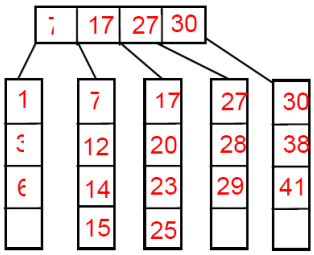
i) After inserting 3



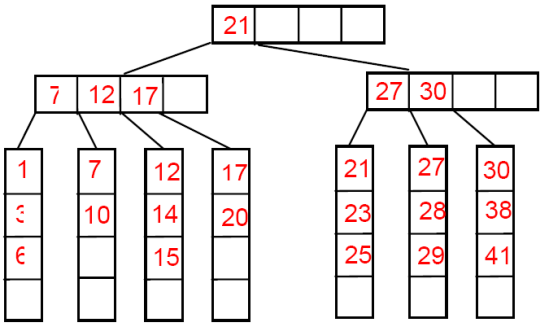
ii) After inserting 38



iii) After inserting 20



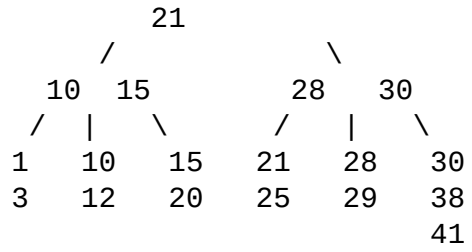
iv) After inserting 10



(v) Apply the deletion procedure described in the lecture notes to the B^+ tree you derived above. Draw the resulting B^+ tree after deleting all the following numbers in the given order:

17 14 7 27 23 6

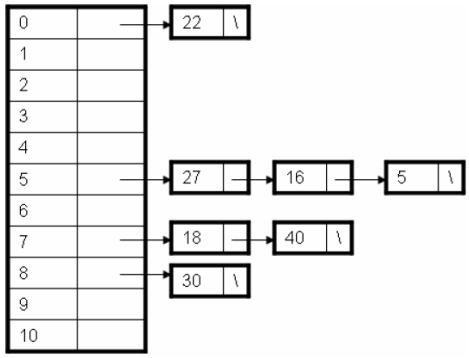
Solution:



8 Hash Table

Suppose you are given a table T of size 11 and a set $S = \{5, 40, 18, 22, 16, 30, 27\}$ to hash into the table, using the hash function $h(k) = k\%11$.

(a) Show T after the values from S are entered into it, using separate chaining.
Solution:



(b) Show T after the values from S are entered into it, using linear probing with the function $h(k, i) = (h(k) + 2i)\%11$.
Solution:

0	22
1	
2	16
3	
4	27
5	5
6	
7	40
8	30
9	18
10	

(c) For this particular T and S, which is the better choice, and why? Your answer should discuss the number of probes necessary to `Find` a particular key.
Solution:
Separate chaining is the better choice for this set S. The longest list in the separate chaining case contains 3 elements to search through, while linear probing requires as many as 6 probes to insert or find elements.

(d) Give an example of a set S with 7 elements for which the other method is the better choice. Explain briefly.
Solution:
A good set for linear probing is one that has no collisions, e.g., $\{1, 2, 3, 4, 5, 6, 7\}$. In this case, the extra space required for the lists of separate chaining is a waste of memory, as a find is equally fast for either method.