CMPSC 130A
DATA STRUCTURES AND ALGORITHMS
Programming Assignment
First Turnin Day: July 21, 2014 (1:45pm)
You need to tun in electronically what you have on July 21st.
You will receive extra credit, but it depend on how much work you have completed by this time.
DUE DATE: July 28, 2014 (1:45 pm)
Remember: You will LOSE 8% of the total points EACH DAY (rounded up) your assignment is late.
Deadline: July 29, 2014 (1:45 pm).
Remember: Projects will not be accepted after the Deadline.
You must work on this project individually.
You may use any of the code in either of the textbooks, if it fits the project.
You may use any of the code discussed in class, if it fits the project.
Total Points: 160
PRELIMINARY VERSION

1 Introduction

For this programming assignment you need to write and implement a set of C++ Classes and Member Functions for the multiset of positive integers ADT. The ADT consists of a multiset of positive integers whose values are between 1 and 2,000,000,000. In Section 2 we discuss the user interface commands. The ADT must be represented by Sorted Linked List (SLL) or a Weight-Biased Leftist Tree (WBLT), depending on the input and the number of elements stored as defined in Section 3. In Section 4 we give you instructions on the coding. Later on we will post in the `map.html` page instructions for the electronic turnin of your project, an example, and sample input and output files.

The objective of this project is for you to practice the implementation of data structures, and use C++, rather than trying to make you an expert in C++. We are not looking for you to come up with an Object-Oriented master piece; however, your implementation must follow our guidelines and your implementation should be object-oriented. Note that you are **NOT ALLOWED to use STL** (except `string`) in any portion of your implementations. Though, you may use STL for debugging purposes only. STL (except `string`) should NOT be part of the final code.

The ADT consists of a multiset of positive integers. Initially the multiset is empty. Note that since the mathematical object is a multiset, we do keep multiple copies of elements in the multiset. In this project you will be manipulating many multisets and each multiset will have a name. The names of all the multisets will NOT be unique. When we have two or more multisets with the same name, we will be operating on the version of the multiset that was last defined and has not yet been deleted. The name of each multiset consists of a (nonempty) sequence of at most 80 (lower case) letters from the English alphabet. The names of the multisets will be stored in a Trie similar (but not identical) to the one discussed in class. Note that an element may be in more than one of the multisets being represented, and in each multiset there may be more than one copy of an element.

Note that when objects are no longer required by the program, **YOU MUST free them** via the destructor which is used by the `C++ delete` command. For complex objects, they must be deleted one sub-object at a time. The procedures must be efficient. This will be discussed later on. Your procedures are NOT allowed to change representation just before an operation is to be performed, then perform the operation on a simple structure, and then transform back the resulting data into the original

representation.

2 User Interface

Your main program must read in an arbitrary sequence of commands each in one line and ending with a finish-up command of the form given below. When we write $< s >$ or $< t >$ below, we mean any of the multisets (whose names consist of a **nonempty** sequence of lower-case English letters). **Note that the empty string of letters is NOT the name of a multiset.** The multisets $< s >$ or $< t >$ may or may not be defined. Below we explain how to interpret these conditions. The symbol x (or y) represents a positive integer (value is between 1 and 2,000,000,000). The symbols k represent a non-negative integer. You may assume that the input is free from FORMAT ERRORS. Remember that all the valid elements in a multiset are integers in the range $[1, 2000000000]$. Your program will process a sequence of the following commands:

Create $< s >$
: If multiset $< s >$ is undefined, then define an empty multiset with name $< s >$. If the multiset was defined already, then you we have a new version of the multiset which is initially empty. After this operation the multiset $< s >$ will be defined.

Merge $< s > < t >$
: If at least one of the multisets $< s >$ or $< t >$ is not defined, the operation will be a no-op. Otherwise, the current version of multiset $< s >$ will get all the elements in the current version of multiset $< t >$. The current multiset $< t >$ will remain defined, but it will become an empty multiset. Do NOT implement this operation as a series of DeleteMins and Inserts.

Delete $< s >$
: If multiset $< s >$ is not defined, then it will be a no-op. Otherwise delete the current version of multiset $< s >$. I.e., delete all its elements. It this was the only version of multiset $< s >$ then delete the name of the multiset from the current multisets available and multiset $< s >$ will become undefined. On the other hand, if there is another version of multiset $< s >$ then the multiset $< s >$ that was created last and has not yet been deleted will become the current multiset $< s >$ from now on and multiset $< s >$ continues to be defined. Do NOT implement this as series of DeleteMins.

DeleteAll $< s >$
: If multiset $< s >$ is not defined, then it will be a no-op. Otherwise delete all the versions of multiset $< s >$. I.e., delete all the elements in all the multisets named $< s >$. Multiset $< s >$ will become undefined after this operation. Do NOT implement this as sequence of Delete $< s >$ or DeleteMins.

Insert $< s >$ x
: If the multiset $< s >$ is not defined, then it will be a no-op. Otherwise insert element x in the current version of multiset $< s >$.

DeleteMin $< s >$
: If the multiset $< s >$ is not defined or it is defined and empty, then it will be a no-op. Otherwise delete the smallest element from the current multiset $< s >$. The multiset will continue to be defined, even if it becomes empty.

PrintMin $< s >$

If the multiset $< s >$ is not defined or it is defined and the current multiset $< s >$ is empty, then it will be a no-op and nothing will be printed. Otherwise, the operation will print the smallest element in the current multiset $< s >$ and the contents of the current multiset $< s >$ will not change.

PrintMax $< s >$

If the multiset $< s >$ is not defined or it is defined and the current multiset $< s >$ is empty, then it will be a no-op and nothing will be printed. Otherwise, the operation will print the largest element in the current multiset $< s >$ and the contents of the current multiset $< s >$ will not change. Do NOT implement this as a sequence of DeleteMins and Inserts.

PrintNum $< s >$

If the multiset $< s >$ is not defined, then it will be a no-op and nothing will be printed. Otherwise, it will count and print the number of elements in the current multiset $< s >$. If the multiset is empty, then it will print 0.

Dist $< s > k$

If the multiset $< s >$ is not defined, then it will be a no-op and nothing will be printed. Otherwise, count and then print the number of elements in the current multiset $< s >$ with value at least m and at most $m + k$, where m is the smallest element stored in the current multiset $< s >$. If the multiset is empty, then it will print 0.

PrintKth $< s > k$

If the multiset $< s >$ is not defined, then it will be a no-op and nothing will be printed. Otherwise, find and print the k^{th} smallest element in the current multiset $< s >$, when there are k or more elements in the current multiset $< s >$. If there are fewer than k elements, then nothing should be printed. It is normally the case that k is very small compared to the number of elements in the multiset. You are allowed to create an array with $O(k)$ elements at the beginning of this operation, but you must delete it at the end of this operation. DO NOT implement as series of DeleteMins and Inserts.

DeleteKth $< s >$ k

If the multiset $< s >$ is not defined, then it will be a no-op and nothing will be deleted. Otherwise, delete the kth smallest element in the current version of multiset $< s >$, when there are k or more elements in the current multiset $< s >$. If there are fewer than k elements, then nothing should be deleted. It is normally the case that k is very small compared to the number of elements in the multiset. You are allowed to create an array with O(k) elements at the beginning of this operation, but you must delete it at the end of this operation. DO NOT implement as series of Delete-Mins and Inserts.

CountN

Count and print the number of multisets currently defined. If there are none defined, then print 0.

CountNT

Count and print the total number of multisets including multiple copies of the multisets. If there are no multisets defined, then print 0.

PrintNumSF $< s > < t >$

Print the **total** number of multisets with names (in alphabetical order) after name $< s >$ and

before $< t >$. **Include in the count all the versions of all the multiset.** Note that $< s >$ and $< t >$ will not be included in the count. Note that $< s >$ and $< t >$ may or may not be defined multisets. If there are no names between these limits, then print 0.

DeleteSF $< s > < t >$
Delete all the CURRENT multisets with names (in alphabetical order) after name $< s >$ and before $< t >$. Note that $< s >$ and $< t >$ will not be deleted. Note that $< s >$ and $< t >$ may or may not be defined multisets. DO NOT implement operation as a sequence of Delete and Insert operations. The implementation should be an "integrated" operation.

Check $< s >$
Appendix 3 has the code for this operation. If multiset $< s >$ is not defined then the operation will do nothing. If the multiset $< s >$ is defined and the current multiset is empty, the print "True 0 1". Let $j = 1$ if the current multiset $< s >$ is represented by a SLL and $j = 2$ if it is represented by a WBLT. The procedure checks to see if the current multiset $< s >$ is a valid representation for the multiset. If it is valid, then print "True" followed by the number of elements in the multiset and then the value of j. If it is not valid, then it prints "False 0" followed by the value for j.

CheckTrie
Appendix 3 has the code for this operation. This procedure checks to see if the Trie is valid. If it is valid, then print "True" followed by the number of multisets with different names. Otherwise, prints "False 0".

Quit
Finish up. Return all the space used by your objects and end the program.

Again, do not implement the above operations as a sequence of DeleteMins and Inserts. An example as well as its input file and corresponding output file for this example will be added to the `map.html` web page.

3 Internal Representation

The names of the multisets will be represented by a `Trie`. The `Trie` is similar to the one defined in class. The nodes in the Trie will be of two types: `BasicTrieNodes` and `TrieNodes`. All the leaf nodes in the `Trie` will be `BasicTrieNodes` and all internal nodes will be TrieNodes. Note that this will save space as all the `BasicTrieNodes` do not have an array of pointers. The `ptr2ms` points to the multiset that stores all the elements in the multiset corresponding to the name stored at the `Trie` that ends at this `BasicTrieNode` or `TrieNode`. Note that `StrEnds` is not needed as one can compute it by checking whether or not `ptr2ms` is `Null` or not.

```
const int StrMaxElem = 81;
const int TrieMaxElem = 26;

class BasicTrieNode {
private:
    MultiSet   *ptr2ms;
public:
    virtual int WhoAmI() {return(0);}
    virtual bool CheckTrie(int*);    // Changed 7/14
```

```
};

class TrieNode:  public BasicTrieNode{
private:
    BasicTrieNode *ptr[TrieMaxElem];
public:
    int WhoAmI() {return(1);}
    bool CheckTrie(int*);    // Changed 7/14
};

class Trie {
private:
    BasicTrieNode *root;
};
```

We will be using either Sorted Linked Lists (SLL) or Weight-Biased Leftist Trees (WBLT) (discussed in class) to represent multisets. Two input variables, `useSLL` and `useWBLT`, such that $0 \leq useSLL < useWBLT$, will be used to decide the representation to be used. The representation is dynamic and the one used depends on the number of elements in the multiset. When a multiset has a number of elements at most $useSLL$ it is represented by a SLL, but if it has at least $useWBLT$ elements, then we represent the set by a WBLT. If the number of elements is more than $useSLL$ and less than $useWBLT$ then either representation may be used. Note that as the number of elements stored in the `MultiSet` changes, you may need to change representation.

When $useSLL$ (or `useWBLT`) is a very large value, you will be using a SLL as long as then number of elements in the multiset is less than that large value. When a set when created from scratch will be represented by a SLL, unless $useSLL$ is zero. When $useSLL$ is zero you will always represent the multiset by a WBLT.

A good explanation of basic weight-biased leftist trees is given in the in the textbook [Sa] (Section 12.5). There is also an explanation in wikipedia, but I am not sure it is correct. It will also be discussed in class and the slides will be in the `map.html` web page.

Now the elements in the multiset will be represented by objects of the classes `SLL` or `WBLT` which are derived from the class `MultiSet`. The integer `number` is used for the number of elements in the multiset, and `ptrpreviousVersion` points to previous version of the multiset. The pointer `first` is used to point the first node in the SLL and `root` points to the root node of the `WBLT`. The meaning of the private variables for the `SLLNodes` and `WBLTNodes` is straight forward. The classes are defined as follows.

```
class MultiSet{
private:
    int            number;
    MultiSet*      ptr2previousVersion;
public:
    virtual int WhoAmI() {}
};

class SLL: public MultiSet{
private:
     SLLNode*      first;
```

```
public:
int      WhoAmI(){return 2;}
};

class WBLT: public MultiSet{
private:
     WBLTNode*      root;
public:
int      WhoAmI(){return 3;}
};


class SLLNode {
private:
   int  data;
   SLLNode *next;
public:
  void Insert( int );
  int DeleteMin( int );
};

class WBLTNode {
private:
   int  data;
   int  w;
   WBLTNode *leftchild;
   WBLTNode *rightchild;
public:
  void Insert( int );
  int DeleteMin( int );
};
```

In Appendix 1 you will find an example of the structure for the classes defined above. All the classes are assembled together in Appendix 2.

4 CODING

We will provide you with some test files later on, but you may create (individually or in a group) test files to test your programs. You may interchange your test files with other students in the class and compare results. But each student must debug their code independently. This way you will further develop your debugging skills.

If you have a test file that you feel is robust, turn them in when you turnin your code and we might use it to test all the programs. **Write a one page summary that describes which operations have been implemented correctly and which have not. Explain what were he major hurdles you encounter while doing this project. Explain how would you go about the next time you have a project like this.** Your write-up must be turned in to the CS Mailbox in HFH 2108 when you turnin your code or you may create a PDF file and turn electronically with your code. The

programs should also be turned in electronically via the turnin program (we will give instructions about this later on).

INPUT: The first input line that your program must read has the value for `useSLL` followed by `useWBLT`. Then you will find a sequence of operations (one per line) ending with the operation "**Quit**". It is assumed that `useSLL` < `useWBLT`.

The program must be in C++, but it is not required to be an Object Oriented work of art. But you must use the C++ stuff discussed in class and the classes given in the Appendix 2. In "principle" you must use the same structures defined in Appendix 2. Though you need to add a bunch of member functions, but no additional data can be included in the classes. The names for variables may be different. But if you change the names of variables, you may need to change the code we will provide for `Check` and `CheckTrie`. Leave the procedures recursive, but the code must be efficient. Nodes that do not need to be visited in the operations should not be visited. **YOU MUST USE makefiles**, split your files into .H and .C files. Each Class must have a .H file for the class declarations and a .C file for all the code for the Class functions. You must turn in electronically your code. To grade it, we will save it in a directory and type "make", followed by "./executeit < data.1 > output.1 ", "./executeit < data.2 > output.2 ", ..., where "data.1", "data.2", ... are files we will be creating, and the answers you program computes is stored in output.1, output.2, etc. This means is that you must leave the executable in a file called `executeit`. Note that the above execution is by redirecting the input/output so your program should do all the input/out through the standard input (`cin`) and standard output (`cout`). After everyone turns in their project we will make the test files available to you. Your program MUST work in the CSIL PCs running Linux using the `g++` compiler..

The program MUST work in the CSIL PCs under Linux. Note that your grade depends on how well you program works on the examples we will use to test your program. These files include small, large and very large files. Partial credit will be given if not all of the code is working correctly. The partial credit will depend on how well your code works on the test files. You will be deducted points if your **program takes too long to execute and/or does not use the structures we define here** (in either of these cases **you may loose most of the points**).

Implement your procedures as efficient as possibles, BUT leave the procedures RECURSIVE. Nodes that do not need to be visited in operations should not be visited. Try to use the minimum amount of additional space. But do not optimize to the last bit. Your procedures must perform logical sub-functions. Your code must be readable and there should be appropriate comments all over the code describing its behavior. Points will be deducted if your code is unreadable and/or the comments in your program are not enough and/or inappropriate.

Appendix 1 has an example of the structures used. Appendix 2 has all the Classes and it is also in the `map.htm` web page. Appendix 3 has the code for Check and CheckTrie.

5 Time complexity of your procedures

The **time complexity for your procedures** must be as in the following table.

Operation	TC SLL	TC WBLT
Create	$O(\#s + c)$	$O(\#s + c)$
Merge	$O(\#s + \#t + m)$	$O(\#s + \#t + \log m)$
Delete	$O(\#s + n)$	$O(\#s + n)$
Insert	$O(\#s + n)$	$O(\#s + \log n)$
DeleteMin	$O(\#s + c)$	$O(\#s + \log n)$
PrintMin	$O(\#s + c)$	$O(\#s + c)$
PrintMax	$O(\#s + n)$	$O(\#s + n)$
PrintNum	$O(\#s + c)$	$O(\#s + c)$
Dist	$O(\#s + k)$	$O(\#s + k)$
PrintKth	$O(\#s + k)$	$O(\#s + k \log k)$
DeleteKth	$O(\#s + k)$	$O(\#s + k \log k + \log n)$
CountN	$O(\#s + N)$	$O(\#s + N)$
CountNT	$O(\#s + M + N)$	$O(\#s + M + N)$
PrintNumSF	$O(\#s + \#t + rh)$	$O(\#s + \#t + rh)$
DeleteSF	$O(\#s + \#t + r'h)$	$O(\#s + \#t + r'h)$
Quit	$O(q)$	$O(q)$

Table 1: Required time complexity for your implementation. Note that we do not include the time complexity needed to change representation. Note that the number of English letters is 26 which is just a constant.

Term	Meaning
c	Constant independent of the problem size.
$\#s$	Number of characters in $< s >$.
$\#t$	Number of characters in $< t >$.
m	Total elements in both multisets.
n	Number of elements in the multiset.
k	Value of k in operation.
N	Number of TrieNodes.
M	Total number of multisets (including all versions).
r	Value printed by operation (constant, if zero or nothing printed).
h	Height of Trie.
r'	Number of TrieNodes deleted in the operation.
q	Total amount of space allocated dynamically in your program.

Table 2: Terms.

APPENDIX 1

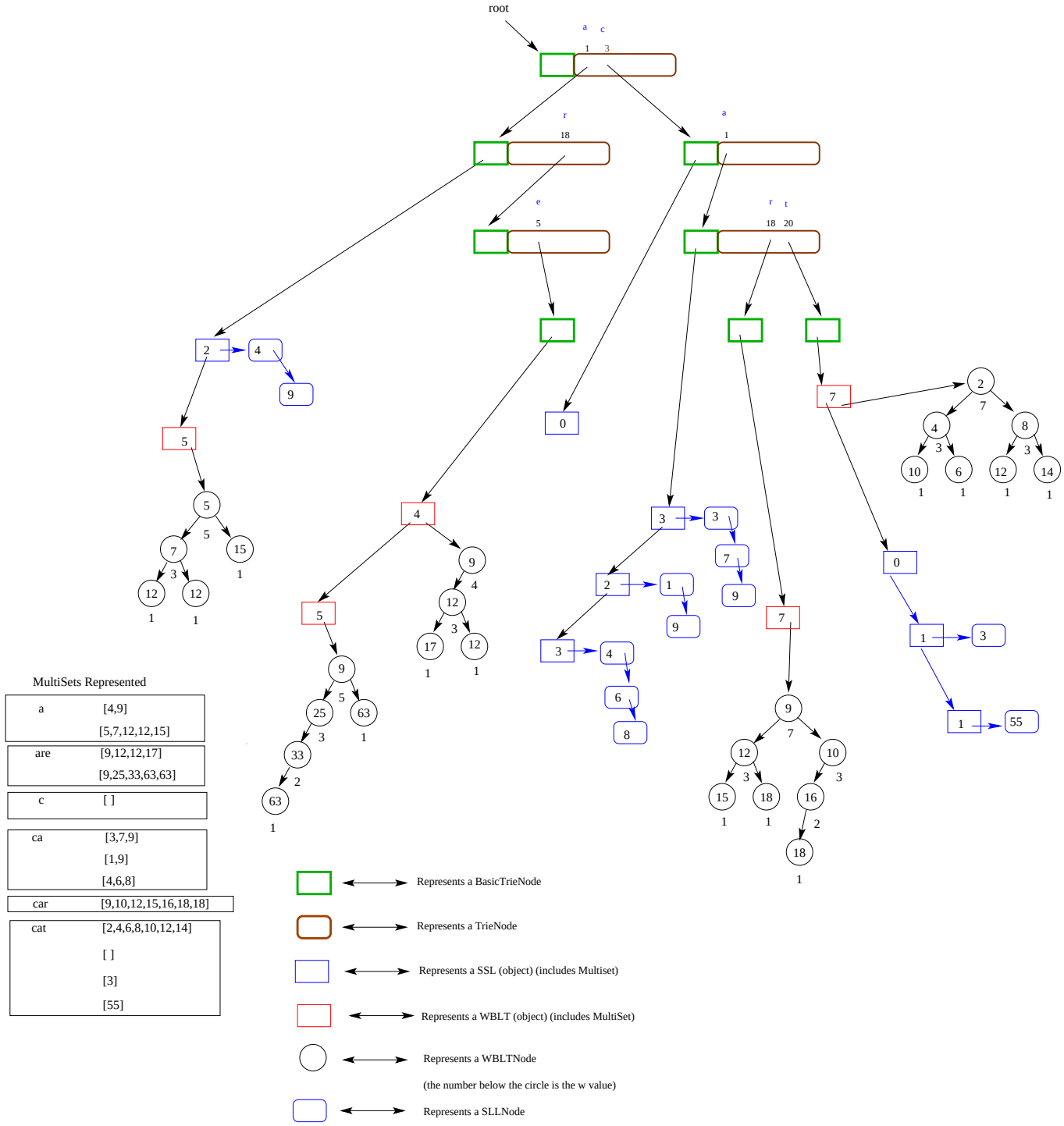

Figure 1: Example of the structure

Appendix 2

```
#include <iostream>
#include <string>

using namespace std;

// Global Variables
const int StrMaxElem = 81;
const int TrieMaxElem = 26;

class TrieNode;
class MultiSet;

class BasicTrieNode {
private:
    MultiSet   *ptr2ms;
public:
    virtual int WhoAmI() {return(0);}
    virtual bool CheckTrie(int*);    // Changed 7/14
};

class TrieNode:  public BasicTrieNode{
private:
    BasicTrieNode *ptr[TrieMaxElem];
public:
    int WhoAmI() {return(1);}
    bool CheckTrie(int*);    // Changed 7/14
};

class Trie {
private:
    BasicTrieNode *root;
};

class MultiSet{
private:
    int            number;
    MultiSet*      ptr2previousVersion;
public:
    virtual int WhoAmI() {}
};

class SLLNode;
class WBLTNode;

class SLL: public MultiSet{
private:
```

```
     SLLNode*      first;
public:
int      WhoAmI(){return 2;}
};

class WBLT: public MultiSet{
private:
     WBLTNode*      root;
public:
int      WhoAmI(){return 3;}
};

class SLLNode {
private:
   int  data;
   SLLNode *next;
public:
  void Insert( int );
  int DeleteMin( int );
};

class WBLTNode {
private:
   int  data;
   int  w;
   WBLTNode *leftchild;
   WBLTNode *rightchild;
public:
  void Insert( int );
  int DeleteMin( int );
};



int main(void)
{
  return 0;
}
```