Vasiuk, Iurii Alekseevich CS 235 week13

makefile

```
###############################################################
# Program:
#     Week 13, Genealogy
#     Brother Ercanbrack, CS235
# Author:
#     Yurii Vasiuk
# Summary:
#     Read, process, write file.  Build the tree and print out the data from it
###############################################################

##############################################################
# The main rule
##############################################################
a.out: week13.o
       g++ -o a.out week13.o -g
       tar -cf week13.tar *.h *.cpp makefile

##############################################################
# The individual components
#      week13.o     : the driver program
#      level.o      : the level-order traversal program
##############################################################
week12.o: week13.cpp list.h
       g++ -c week13.cpp -g
```

118/150

Commented [ES1]: Some differences in alphabetic listing. You added birthdays to names without a birthday.   -5

No pedigree output   - 28

list.h

```
/***********************************************************************
* Header:
*    List
* Summary:
*    Custom made List analogous to the std::list
*    The class will use Node and ListIterator classes
* Author
*    Yura Vasiuk
************************************************************************/

#ifndef LIST_H
#define LIST_H

#include <iostream>
using namespace std;

template <typename T>
class ListIterator;

/************************************************
* PERSON
* The class to be used as data for the Node
***********************************************/
class Person
{
public:
   // default constructor
   Person() : _firstName(), _familyName(), _birthDate(), _birthYear(), _id() {}
   // non-default constructor
   Person(string firstName, string familyName, string bD, string bY, string id)
   {
      _firstName = firstName;
      _familyName = familyName;
      _birthDate = bD;
      _birthYear = bY;
      _id = id;
   }

   // is this person smaller than the rhs person?
   bool operator<(Person rhs)
   {
      // compensate for the low case letters
      string tempFN;
      string rhsTempFN;
      // this family name is low case
      if (islower(_familyName[0]))
      {
         _familyName[0] = toupper(_familyName[0]);
         tempFN = _familyName;
         _familyName[0] = tolower(_familyName[0]);
      }
      else
         tempFN = _familyName;
      // rhs family name is low case
      if (islower(rhs._familyName[0]))
      {
         rhs._familyName[0] = toupper(rhs._familyName[0]);
         rhsTempFN = rhs._familyName;
         rhs._familyName[0] = tolower(rhs._familyName[0]);
      }
      else
         rhsTempFN = rhs._familyName;
      // the end of low case compensation

      if (tempFN < rhsTempFN)
      {
         return true;
      }
      else if (_familyName == rhs._familyName && _firstName < rhs._firstName)
         return true;
      else if (_firstName == rhs._firstName && _familyName == rhs._familyName
         && (_birthYear < rhs._birthYear))
         return true;
      else
         return false;
   }

   string _firstName;
   string _familyName;
   string _birthDate;
   string _birthYear;
   string _id;
};

/************************************************
* NODE
* A class to be used in LinkedList
***********************************************/
template <typename T>
class Node
{
public:
   T data;
   Node<T> * pNext;
   Node<T> * pPrev;
   Node<T> * pMother;
   Node<T> * pFather;

   Node() : pNext(NULL), pPrev(NULL), pMother(NULL), pFather(NULL) {}
   Node(T data)
   {
      this->data = data;
      this->pNext = NULL;
      this->pPrev = NULL;
      this->pMother = NULL;
      this->pFather = NULL;
   }
   Node(T data, Node<T> * pNext, Node<T> * pPrev, Node<T> * pMother, Node<T> * pFather)
   {
      this->data = data;
      this->pNext = pNext;
      this->pPrev = pPrev;
      this->pMother = pMother;
      this->pFather = pFather;
   }
};

/************************************************
```

```
* LIST
* Custom made List, analogous to the std::list
***********************************************/
template <typename T>
class List
{
public:
   // default constructor
   List() : numItems(0), pHead(NULL), pTail(NULL) {}    // done

   // copy constructor : copy it
   List(const List<T> & rhs) throw (const char *);        // done

   // destructor
   ~List() { clear(); }                                   // done

   // assignment operator
   List<T> & operator=(const List<T> & rhs) throw (const char *);  // done

   // check if empty
   bool empty() const { return numItems == 0; }                 // done

   // what is the number of items in the list
   int size() const { return numItems; }                        // done

   // empy the list of all the items
   void clear();                                          // done

   // add an item to the back of the list
   void push_back(T t) throw (const char *);              // done

   // add an item to the front of the list
   void push_front(T t) throw (const char *);             // done

   // returnt the element at the front of the list
   T & front() throw (const char *);                      // done

   // return the element at the back of the list
   T & back() throw (const char *);                       // done

   // return the interator to the front of the list
   ListIterator<T> begin() const { return ListIterator<T>(pHead); }   // done

   // return the interator to the back of the list
   ListIterator<T> rbegin() const { return ListIterator<T>(pTail); }  // done

   // return the iterator to the past-the-front of the list
   ListIterator<T> rend() const { return NULL; }                // done

   // return the iterator to the past-the-back of the list
   ListIterator<T> end() const { return NULL; }                 // done

   // insert the passed item before the passed pointer
   ListIterator<T> insert(ListIterator<T> pInsertBefore, T t) throw (const char *);

   // remove the item at the passed poiter
   void remove(ListIterator<T> pRemoveHere) throw (const char *);

   // I will need it for BigNumber operator=
   Node<T> * & getHead() { return pHead; }

private:
   int numItems;
   Node<T> * pHead;
   Node<T> * pTail;
};

/***************************************************
* LIST :: COPY CONSTRUCTOR
* Create a new List and copy the data into it
**************************************************/
template <typename T>
List<T> ::List(const List<T> & rhs) throw (const char *)
{
   pHead = NULL;
   pTail = NULL;
   numItems = 0;

   // nothing to do
   if (rhs.pHead == NULL)
      return;

   // axiliary pointers
   Node<T> * pTraverseOld = NULL;
   Node<T> * pTraverseNew = NULL;
   Node<T> * pTemp = NULL;

   // make the first node
   pHead = new Node<T>;
   pTail = pHead;
   // assign pointers
   pHead->pNext = NULL;
   pHead->pPrev = NULL;
   // assign traverses
   pTraverseNew = pHead;
   pTraverseOld = rhs.pHead;

   while (pTraverseOld->pNext != NULL)
   {
      // fill the new node data
      pTraverseNew->data = pTraverseOld->data;
      // making a one more node, fill the new node address
      pTraverseNew->pNext = new Node<T>;
      // temp
      pTemp = pTraverseNew;
      // move the traverses
      pTraverseOld = pTraverseOld->pNext;
      pTraverseNew = pTraverseNew->pNext;
      // assign pPrev of the current last node and increase numItems
      pTraverseNew->pPrev = pTemp;
      numItems++;
   }
   // fill the last node data and increase numItems
   pTraverseNew->data = pTraverseOld->data;
   numItems++;
   // assign the pTail
   pTail = pTraverseNew;
}

/****************************************************
* LIST :: ASSIGNMENT OPERATOR
* Copy the data into the list
***************************************************/
template <typename T>
List<T> & List<T> :: operator=(const List<T> & rhs) throw (const char *)
{
   // clear the current list
   this->clear();

   // nothing to do
   if (rhs.pHead == NULL)
      return *this;

   // axiliary pointers
   Node<T> * pTraverseOld = NULL;
   Node<T> * pTraverseNew = NULL;
   Node<T> * pTemp = NULL;

   // make the first node
   this->pHead = new Node<T>;
   this->pTail = this->pHead;
   // assign pointers
   this->pHead->pNext = NULL;
   this->pHead->pPrev = NULL;
   // assign traverses
   pTraverseNew = this->pHead;
   pTraverseOld = rhs.pHead;

   while (pTraverseOld->pNext != NULL)
   {
      // fill the new node data
      pTraverseNew->data = pTraverseOld->data;
      // making a one more node, fill the new node address
      pTraverseNew->pNext = new Node<T>;
      // temp
      pTemp = pTraverseNew;
      // move the traverses
      pTraverseOld = pTraverseOld->pNext;
      pTraverseNew = pTraverseNew->pNext;
      // assign pPrev of the current last node and increase numItems
      pTraverseNew->pPrev = pTemp;
      numItems++;
   }
   // fill the last node data and increase numItems
   pTraverseNew->data = pTraverseOld->data;
   numItems++;
   // assign the pTail
   this->pTail = pTraverseNew;

   return *this;
}

/****************************************************
* LIST :: CLEAR
* Empty the list of all the items
***************************************************/
template <typename T>
void List<T> ::clear()
{
   // there is nothing to delete
   if (pHead == NULL)
      return;

   // axiliary pointers
   Node<T> * pDelete = pHead;
   Node<T> * pTraverse = pHead->pNext;

   while (pTraverse != NULL)
   {
      delete pDelete;
      pDelete = pTraverse;
      pTraverse = pTraverse->pNext;
   }
   // delete the last node and set the head and tail to NULL
   delete pDelete;
   pHead = NULL;
   pTail = NULL;
   // last thing to do
   numItems = 0;
}

/****************************************************
* LIST :: PUSH_BACK
* Add an item to the back of the list
***************************************************/
template <typename T>
void List<T> ::push_back(T t) throw (const char *)
{
   // attempt to allocate a new node
   try
   {
      if (empty())
      {
         pHead = new Node<T>();
         pTail = pHead;
      }
      else
         pTail->pNext = new Node<T>();
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: unable to allocate a new node for a list";
   }

   // case 1) only one node in the list
   if (empty())
   {
      pHead->data = t;
      numItems++;
   }
   // case 2) reassign pTail, assign pPrev, fill with data, add numItems
   else
   {
      Node<T> * temp = pTail;
      pTail = pTail->pNext;
      pTail->pPrev = temp;
      pTail->data = t;
      numItems++;
   }
}


/***************************************************
* LIST :: PUSH_FRONT
* Add an item to the front of the list
**************************************************/
template <typename T>
void List<T> ::push_front(T t) throw (const char *)
{
   // temporary pointer
   Node<T> * temp;
   // attempt to allocate a new node, switch the pointers
   try
   {
      if (empty())
      {
         pHead = new Node<T>();
         pTail = pHead;
      }
      else
      {
         temp = pHead;
         pHead = new Node<T>();
         pHead->pNext = temp;           // the first node pointer
         pHead->pNext->pPrev = pHead;   // the second node pointer
      }
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: unable to allocate a new node for a list";
   }

   // finally, fill the data and increment the numItems
   pHead->data = t;
   numItems++;
}

/***************************************************
* LIST :: FRONT
* Return the element at the front of the list
**************************************************/
template <typename T>
T & List<T> ::front() throw (const char *)
{
   if (empty())
      throw "ERROR: unable to access data from an empty list";
   else
      return pHead->data;
}

/***************************************************
* LIST :: BACK
* Return the element at the back of the list
**************************************************/
template <typename T>
T & List<T> ::back() throw (const char *)
{
   if (empty())
      throw "ERROR: unable to access data from an empty list";
   else
      return pTail->data;
}

/***************************************************
* LIST :: INSERT
* Insert the passed element before the passed pointer
* and return the pointer to the inserted element
**************************************************/
template <typename T>
ListIterator<T> List<T> ::insert(ListIterator<T> pInsertBefore, T t) throw (const char *)
{
   // convert the pointer from ListIterator to Node type
   Node<T> * pNodeInsertBefore = pInsertBefore.p;

   Node<T> * temp;
   // attempt to allocate a new node
   try
   {
      temp = new Node<T>(t);
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: unable to allocate a new node for a list";
   }

   // special cases
   // 1) insert after the list
   if (pNodeInsertBefore == NULL)
   {
      push_back(t);
      return NULL;
   }
   // 2) insert before the list
   if (pNodeInsertBefore->pPrev == NULL)
   {
      push_front(t);
      return NULL;
   }

   // usual cases
   // case 1) insert into empty list
   if (pHead == 0)
   {
      pHead = temp;
      pTail = temp;
   }
   // case 2) insert before the last node
   else if (pNodeInsertBefore == pTail)
   {
      pTail->pPrev->pNext = temp;
      temp->pPrev = pTail->pPrev;
      temp->pNext = pTail;
      pTail->pPrev = temp;
   }
   // case 3, 4) the rest: insert in the middle, insert before the first node
   else
   {
      temp->pNext = pNodeInsertBefore->pPrev->pNext;
      temp->pPrev = pNodeInsertBefore->pPrev;
      pNodeInsertBefore->pPrev = temp;
      // before the first, repoint the pHead
      if (pNodeInsertBefore == pHead)
         pHead = temp;
      else
         temp->pPrev->pNext = temp;
   }

   // the last thing to do
   numItems++;

   return NULL;
}

/***************************************************
* LIST :: REMOVE
* Remove the element at the passed pointer
**************************************************/
template <typename T>
void List<T> ::remove(ListIterator<T> pRemoveHere) throw (const char *)
{
   // nothing to remove
   if (pRemoveHere == end())
      throw "ERROR: unable to remove from an invalid location in a list";

   // convert the pointer from ListIterator to Node type
   Node<T> * pNodeRemoveHere = pRemoveHere.p;

   // reassign the pointers
   // case 1) remove the fist node
   if (pHead == pNodeRemoveHere)
   {
      pNodeRemoveHere->pNext->pPrev = NULL;
      pHead = pHead->pNext;
   }
   // case 2) remover the last node
   else if (pTail == pNodeRemoveHere)
   {
      pTail = pTail->pPrev;
      pTail->pNext = NULL;
   }
   // case 3) the rest, remove in the middle
   else
   {
      pNodeRemoveHere->pPrev->pNext = pNodeRemoveHere->pNext;
      pNodeRemoveHere->pNext->pPrev = pNodeRemoveHere->pPrev;
   }
   // delete the node
   delete pNodeRemoveHere;

   // last thing to do
   numItems--;
}

/**************************************************
* LIST ITERATOR
* An iterator through List
*************************************************/
template <typename T>
class ListIterator
{
public:
   // default constructor
   ListIterator() : p(0x00000000) {}

   // initialize to direct p to some item
   ListIterator(Node<T> * p) : p(p) {}

   // copy constructor
   ListIterator(const ListIterator<T> & rhs) { *this = rhs; }

   // assignment operator
   ListIterator<T> & operator = (const ListIterator<T> & rhs)
   {
      this->p = rhs.p;
      return *this;
   }

   // not equals operator
   bool operator != (const ListIterator<T> & rhs) const
   {
      return rhs.p != this->p;
   }

   // equals operator
   bool operator == (const ListIterator<T> & rhs) const
   {
      return rhs.p == this->p;
   }

   // dereference operator
   T & operator * ()
   {
      return p->data;
   }

   // prefix increment
   ListIterator<T> & operator++()
   {
      //p++;
      p = p->pNext;
      return *this;
   }

   // prefix decrement
   ListIterator<T> & operator--()
   {
      //p++;
      p = p->pPrev;
      return *this;
   }

   // these two functions will need the access to the iterator's private *p
   friend ListIterator<T> List<T>::insert(ListIterator<T> pInsertBefore, T t);
   friend void List<T>::remove(ListIterator<T> pRemoveHere);
```

```
private:
   Node<T> * p;
};

#endif // LIST_H
```

week13.cpp

```
/***********************************************************************
* Program:
*    Week 13, Genealogy
*    Brother Ercanbrack, CS 235
* Author:
*    Yurii Vasiuk
* Summary:
*   This is a driver program for the Genealogy assignment
************************************************************************/
#include <fstream>         // file reading and writing
#include <iostream>        // for CIN and COUT
#include <iomanip>         // for SETW
#include <string.h>
#include "list.h"

using namespace std;

/**********************************************************************
* FILLLIST
* the function receives the list and the file name,
* and it fills the list with the data from the file in sorted order
***********************************************************************/
void fillList(List<Person> & people, string fileName)
{
   // these are the components for filling the list
   string givenName, familyName, birthDate, idNumber, birthYear;
   // I will need a mark for fishing out the bith date
   bool birthMark = false;
   // Another mark for having the full info for 1 person
   bool atFirst = true;

   // read and process the file
   string line, token;
   ifstream fin(fileName.c_str());

   if (fin.fail())
   {
      cout << "Could not read the file " << fileName << endl;
   }

   // all the job for parsing the file and filling the list of people
   // will be done in this while() {.........................................}

   // get a line from the file
   while (getline(fin, line))
   {
      //  (1 begin) getting the data for 1 individual----------------------
      // is it individual?
      token = line.substr(2, 2);
      if (token == "@I")   // yes, it is
      {
         if (!atFirst)
         {
            Person thePerson(givenName, familyName, birthDate, birthYear, idNumber);
            bool inserted = false;
            // the insertion
            for (ListIterator<Person> it = people.begin(); it != people.end(); ++it)
            {
               if (thePerson < *it)
               {
                  people.insert(it, thePerson);
                  inserted = true;
                  break;
               }
            }
            if (!inserted)
               people.push_back(thePerson);
            // the person has been inserted to the right place or just pushed on the back
         }
         // this will work for the operator<() in the Person class
         givenName = "";  familyName = ""; birthDate = ""; birthYear = ""; idNumber = "";
         // I am out of the atFirst now
         atFirst = false;
         // get the id
         idNumber = line.substr(4, line.rfind('@') - 4);
      }
      // get the given name
      token = line.substr(2, 4);
      if (token == "GIVN")
      {
         givenName = line.substr(7);
      }
      // get the family name
      token = line.substr(2, 4);
      if (token == "SURN")
      {
         familyName = line.substr(7);
      }
      // get the birth date
      // am I in bith?
      token = line.substr(2, 4);
      if (token == "BIRT")  // yes, I am
      {
         birthMark = true;
      }
      token = line.substr(2, 4);
      if (token == "DATE" && birthMark == true)
      {
         birthDate = line.substr(7);
         birthMark = false;
         birthYear = line.substr(line.rfind(' '));
      }
      //  (1 end) at this point I have the individul's data------------------
   }
   fin.close();

}

/**********************************************************************
* WRITEFILE
* the function receives the list and the filename, and it writes the list inot the sorted.dat file
***********************************************************************/
void writeFile(List<Person> & people, string fileToWrite)
{
   ofstream fout;

   fout.open(fileToWrite.c_str());
   for (ListIterator<Person> it = people.begin(); it != people.end(); ++it)
   {
      if ((*it)._firstName != "")
         fout << (*it)._firstName << " " << (*it)._familyName;
      else
         fout << (*it)._familyName;

      if ((*it)._birthDate != "")
         fout << ", b. " << (*it)._birthDate;
      fout << endl;
   }
   fout.close();
}

/**********************************************************************
* BUILDTREE
* the function receives the list and the file name, and it builds the family tree
***********************************************************************/
void buildTree(List<Person> & people, string fileName)
{

   // read and process the file
   string line, token;
   ifstream fin(fileName.c_str());

   if (fin.fail())
   {
      cout << "Could not read the file " << fileName << endl;
   }
   // all the job for parsing the family part of the file and building the tree
   // will be done in this while() {.......................................}

   string child, father, mother;
   // get a line from the file
   while (getline(fin, line))
   {
      //  getting the data for 1 individual---------------------
      // father
      token = line.substr(2, 4);
      if (token == "HUSB")
      {
         father = line.substr(8, line.rfind('@') - 8);
      }
      // mother
      token = line.substr(2, 4);
      if (token == "WIFE")
      {
         mother = line.substr(8, line.rfind('@') - 8);
      }
      // child
      token = line.substr(2, 4);
      if (token == "CHIL")
      {
         child = line.substr(8, line.rfind('@') - 8);
      }
      // I have 1 family ids at this point, I can assign pointers in the tree
      // build the tree (assigning mother and father pointers)
      for (ListIterator<Person> it = people.begin(); it != people.end(); ++it)
      {
         if ((*it)._id == child)
         {
            for (ListIterator<Person> itM = people.begin(); itM != people.end(); ++itM)
            {
               if ((*itM)._id == mother)
               {
                  // ???;
                  break;
               }
            }
            for (ListIterator<Person> itF = people.begin(); itF != people.end(); ++itF)
            {
               if ((*itF)._id == father)
               {
                  // ???;
                  break;
               }
            }
         }
      }
      // the end of assigning the family tree pointers
   }
   fin.close();
}

/***********************************************************************
* PRINTGENERATIONS
* the function receives the list and prints out the generations
***********************************************************************/
void printGenerations(List<Person> & people)
{

}

/***********************************************************************
* MAIN
* the driver function for the application
***********************************************************************/
int main(int argc, const char* argv[])
{
   string fileName = "";

   // get the file name using cin or from the shell
   if (argc < 2)
   {
      cout << "Usage: fileName" << endl;
      getline(cin, fileName);
      fileName = "/home/vas14001/CS235_Spring2016/week13_Genealogy/cameron.ged";
   }
```

Commented [ES2]: Tree doesn’t get built!

Commented [ES3]: Incomplete!

```
   else
   {
      if (strcmp(argv[1], "cameron.ged") == 0)
      {
         fileName = argv[1];
      }
      else
      {
         cout << "\nInvalid sort name" << endl;
      }
   }

   List<Person> people;

   // fill the list
   fillList(people, fileName);

   // write the list into the file
   string fileToWrite =
      "/home/vas14001/CS235_Spring2016/week13_Genealogy/sorted.dat";
   writeFile(people, fileToWrite);

   // build the family tree
   buildTree(people, fileName);

   return 0;
}

/*
void BTree::level()
{
  const int MAX = 100;
  BTree *queue[MAX];
  BTree *temp;
  int front = 0;
  int back = 0;

  queue[back++] = this;

  while (front != back)
  {
    temp = queue[front];
    front = (front + 1) % MAX;
    if (temp != NULL)
    {
      // visit
      cout.width(4);
      cout << temp->data << " ";
      // end Visit
      queue[back] = temp->left;
      back = (back + 1) % MAX;
      queue[back] = temp->right;
      back = (back + 1) % MAX;

    }
  }
}
*/
```

vas14001@byui.edu

Commented [ES4]: You can’t write the file to your directory and expect me to see it.

This shouldn’t be a full path