Data Structures Set 3

Sherri Shulman
Data Structures Set 3

Nov 5, 2013

3.1
```
template <typename Object>
void printLots(list <Object> L, list<int> P) {
   typename list < int > ::const_iterator pIter ;
   typename list < Object >::const_iterator lIter ;
   int start = 0;
   lIter = L.begin();
   for (pIter=P.begin(); pIter != P.end() && lIter != L.end();
      pIter++) {
      while (start < *pIter && lIter != L.end()) {
         start++;
         lIter++;
      }
   if (lIter !=L.end())
      cout<<*lIter<<endl;
   }
}
```

This code runs in time P.end()–, or largest number in P list. (I've also implemented these, but include them here for convenience.)

3.2 Here is the code **for** single linked lists:
```
// beforeP is the cell before the two adjacent cells that are to
    be
// swapped
// Error checks are omitted for clarity
void swapWithNext(Node * beforep) {
   Node *p , *afterp;
   p = before->next;
   afterp = p->next; // both p and afterp assumed not NULL
   p->next = afterp-> next;
   beforep ->next = afterp;
   afterp->next = p;
}
```

Here is the code **for** doubly linked lists:
```
// p and afterp are cells to be switched.
Error checks as before
{
   Node *beforep, *afterp;
   beforep = p->prev;
```

```
    afterp = p->next;
    p->next = afterp->next;
    beforep->next = afterp;
    afterp->next = p;
    p->next->prev = p;
    p->prev = afterp;
    afterp->prev = beforep;
}
```

3.4 *// Assumes both input lists are sorted*

```
template <typename Object>
list<Object> intersection( const list<Object> & L1, const list<
    Object> & L2) {
    list<Object> intersect;
    typename list<Object>:: const_iterator iterL1 = L1.begin();
    typename list<Object>:: const_iterator iterL2= L2.begin();
    while(iterL1 != L1.end() && iterL2 != L2.end()) {
        if (*iterL1 == *iterL2) {
            intersect.push_back(*iterL1);
            iterL1++;
            iterL2++;
        } else if (*iterL1 < *iterL2)
            iterL1++;
        else
            iterL2++;
    }
    return intersect;
}
```

3.5 *// Assumes both input lists are sorted*

```
template <typename Object>
list<Object> listUnion( const list<Object> & L1, const list<
    Object> & L2) {
    list<Object> result;
    typename list<Object>:: const_iterator iterL1 = L1.begin();
    typename list<Object>:: const_iterator iterL2= L2.begin();
    while(iterL1 != L1.end() && iterL2 != L2.end()) {
        if (*iterL1 == *iterL2) {
            result.push_back(*iterL1);
            iterL1++;
            iterL2++;
        } else if (*iterL1 < *iterL2) {
            result.push_back(*iterL1);
            iterL1++;
        } else {
            result.push_back(*iterL2);
            iterL2++;
        }
    }
    return result;
```

```
}
```

3.6 This is a standard programming project. The algorithm can be sped up by setting M = M mod N , so that the hot potato never goes around the circle more than once. If M ¿ N/2, the potato should be passed in the reverse direction. This requires a doubly linked list. The worst case running time is clearly O(N min(M , N)), although when the heuristics are used, and M and N are comparable, the algorithm might be significantly faster. If M = 1, the algorithm is clearly linear.

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    int i,j, n, m, mPrime, numLeft;
    list <int > L;
    list<int>::iterator iter;
    //Initialization
    cout<<"enter N (# of people) & M (# of passes before
        elimination):";
    cin>>n>>m;
    numLeft = n;
    mPrime = m % n;
    for (I =1 ; I <= n; i++)
       L.push_back(i);
    iter = L.begin();
    // Pass the potato
    for (I = 0; I < n; i++) {
       mPrime = mPrime % numLeft;
       if (mPrime <= numLeft/2) // pass forward
          for (j = 0; j < mPrime; j++) {
             iter++;
             if (iter == L.end())
                iter = L.begin();
          }
       else // pass backward
          for (j = 0; j < mPrime; j++) {
             if (iter == L.begin())
                iter = --L.end();
             else
                iter--;
          }
       cout<<*iter<<" ";
       iter= L.erase(iter);
       if (iter == L.end())
          iter = L.begin();
    }
    cout<<endl;
    return 0;
}
```

3.9 All the aforementioned functions may require the creation of a new array to hold the data.

When this occurs all the old pointers (iterators) are invalid.

3.10 The changes are the const_iterator class, the iterator class and changes to all Vector functions that use or return iterators. These classes and functions are shown in the following three code examples. Based on the Vector defined in the text, only changes includes begin() and end().

```
class const_iterator {
public:
   //const_iterator( ) : current( NULL )
   // { } Force use of the safe constructor
   const Object & operator* ( ) const {
      return retrieve( );
   }
   const_iterator & operator++ ( ) {
      current++;
      return *this;
   }
   const_iterator operator++ ( int ) {
      const_iterator old = *this;
      ++( *this );
      return old;
      }
   bool operator== ( const const_iterator & rhs ) const {
      return current == rhs.current;
   }
   bool operator!= ( const const_iterator & rhs ) const {
      return !( *this == rhs );
   }
protected:
   Object *current;
   const Vector<Object> *theVect;
   Object & retrieve( ) const {
      assertIsValid();
      return *current;
   }
   const_iterator( const Vector<Object> & vect, Object *p )
      :theVect (& vect), current( p ) { }
   void assertIsValid() const {
      if (theVect == NULL || current == NULL )
         throw IteratorOutOfBoundsException();
   }
   friend class Vector<Object>;
};

class iterator : public const_iterator {
public:
   //iterator( )
   // { } Force use of the safe constructor
   Object & operator* ( ) {
      return retrieve( );
```

```
    }
    const Object & operator* ( ) const {
       return const_iterator::operator*( );
    }
    iterator & operator++ ( ) {
       cout<<"old "<<*current<<" ";
       current++;
       cout<<" new "<<*current<<" ";
       return *this;
    }
    iterator operator++ ( int ) {
       iterator old = *this;
       ++( *this );
       return old;
       }
protected:
    iterator(const Vector<Object> & vect, Object *p )
       : const_iterator(vect, p )
          { }
    friend class Vector<Object>;
}

// Changes to Vector class:

    iterator begin( ) {
       return iterator(*this ,&objects[ 0 ]);
    }
    const_iterator begin( ) const {
       return const_iterator(*this,&objects[ 0 ]);
    }
    iterator end( ) {
       return iterator(*this, &objects[ size( ) ]);
    }
    const_iterator end( ) const {
       return const_iterator(*this, &objects[ size( ) ]);
    }
```

3.11 (This version, unlike the one adapted from the doubly linked list) uses null to terminate the list, and doesn't include an iterator.)

```
template <typename Object>
struct Node {
    Object data;
    Node * next;
    Node ( const Object & d = Object(), Node *n = NULL )
       : data(d) , next(n) {}
};
template <typename Object>
class singleList {
public:
    singleList( ) {
```

```
    init();
}
~singleList() {
    eraseList(head);
}
singleList( const singleList & rhs) {
    eraseList(head);
    init();
    *this = rhs;
}
bool add(Object x) {
    if (contains(x))
        return false;
    else {
        Node<Object> *ptr = new Node<Object>(x);
        ptr->next = head->next;
        head->next = ptr;
        theSize++;
    }
    return true;
}
bool remove(Object x) {
    if (!contains(x))
        return false;
    else {
        Node<Object>*ptr = head->next;
        Node<Object>*trailer;
        while(ptr->data != x) {
            trailer = ptr;
            ptr=ptr->next;
        }
        trailer->next = ptr->next;
        delete ptr;
        theSize--;
    }
    return true;
}
int size() {
    return theSize;
}
void print() {
    Node<Object> *ptr = head->next;
    while (ptr != NULL) {
        cout<< ptr->data<<" ";
        ptr = ptr->next;
    }
    cout<<endl;
}
bool contains(const Object & x) {
    Node<Object> * ptr = head->next;
```

```
        while (ptr != NULL) {
            if (x == ptr->data)
                return true;
            else
                ptr = ptr-> next;
        }
        return false;
    }
    void init() {
        theSize = 0;
        head = new Node<Object>;
        head-> next = NULL;
    }
    void eraseList(Node<Object> * h) {
        Node<Object> *ptr= h;
        Node<Object> *nextPtr;
        while(ptr != NULL) {
            nextPtr = ptr->next;
            delete ptr;
            ptr= nextPtr;
        }
    };
private:
    Node<Object> *head;
    int theSize;
};
```

3.22
```
double evalPostFix( ) {
    stack<double> s;
    string token;
    double a, b, result;
    cin>> token;
    while (token[0] !=    =   ) {
        result = atof (token.c_str());
        if (result != 0.0 )
            s.push(result);
        else if (token == "0.0")
            s.push(result);
        else
        switch (token[0]) {
        case    +    : a = s.top(); s.pop(); b = s.top();
                  s.pop(); s.push(a+b); break;
        case    -    : a = s.top(); s.pop(); b = s.top();
                  s.pop(); s.push(a-b); break;
        case    *    : a = s.top(); s.pop(); b = s.top();
                  s.pop(); s.push(a*b); break;
        case    /    : a = s.top(); s.pop(); b = s.top();
                  s.pop(); s.push(a/b); break;
        case    ^    : a = s.top(); s.pop(); b = s.top();
                  s.pop(); s.push(exp(a*log(b))); break;
```

```
        }
        cin>> token;
    }
    return s.top();
}
```

3.23 (a, b) This function will read in from standard input an infix expression of single lower case characters and the operators, +, , /, ,^a nd(,), and output a postfix expression.

```
void inToPostfix() {
    stack<char> s;
    char token;
    cin>> token;
    while (token !=     =    ) {
        if (token >=     a     && token <=     z    )
            cout<<token<<"␣";
        else
        switch (token) {
        case     )     : while(!s.empty() && s.top() !=     (    ) {
                        cout<<s.top()<<"␣"; s.pop();
                    }
                    s.pop();
                    break;
        case     (     : s.push(token); break;
        case     ^     : while(!s.empty() && !(s.top()==     ^     ||
            s.top() ==     (    )) {
                        cout<<s.top(); s.pop();
                    }
                    s.push(token); break;
        case     *     :
        case     /     : while(!s.empty() && s.top() !=     +
                    && s.top() !=     -     && s.top() !=     (    ) {
                        cout<<s.top(); s.pop();
                    }
                    s.push(token); break;
        case     +     :
        case     -     : while(!s.empty() && s.top() !=     (     ) {
                        cout<<s.top()<<                ; s.pop();
                    }
                    s.push(token); break;
        }
        cin>> token;
    }
    while (!s.empty()) {
        cout<<s.top()<<                ; s.pop();
    }
    cout<<    = \n  ;
}
```

(c) The function converts postfix to infix with the same restrictions as above.

```
string postToInfix() {
```

```
    stack<string> s;
    string token;
    string a, b;
    cin>>token;
    while (token[0] !=    =   ) {
       if (token[0] >=    a    && token[0] <=    z    )
          s.push(token);
       else
       switch (token[0]) {
       case    +    : a = s.top(); s.pop(); b = s.top(); s.pop();
                s.push("("+ a+" + "+ b+")"); break;
       case    -    : a = s.top(); s.pop(); b = s.top(); s.pop();
                s.push("("+a+" - "+ b+")"); break;
       case    *    : a = s.top(); s.pop(); b = s.top(); s.pop();
                s.push("("+a+" * "+ b+")"); break;
       case    /    : a = s.top(); s.pop(); b = s.top(); s.pop();
                s.push("("+a+" / " + b+")"); break;
       case    ^    : a = s.top(); s.pop(); b = s.top(); s.pop();
                s.push("("+a+" ^ " + b+")"); break;
       }
       cin>> token;
    }
    return s.top();
}
//Converts postfix to infix
```

3.29 Reversal of a singly linked list can be done recursively using a stack, but this requires O (N) extra space. The following solution is similar to strategies employed in garbage collection algorithms (first represents the first node in the non-empty node in the non-empty list). At the top of the while loop the list from the start to previousPos is already reversed, whereas the rest of the list, from currentPos to the end is normal. This algorithm uses only constant extra space.

```
//Assuming no header and that first is not NULL
Node * reverseList(Node *first) {
    Node * currentPos, *nextPos, *previousPos;
    previousPos = NULL;
    currentPos = first;
    nextPos = first->next;
    while (nextPos != NULL) {
       currentPos -> next = previousPos;
       perviousPos = currentPos;
       currentPos = nextPos;
       nextPos = nextPos -> next;
    }
    currentPos->next = previousPos;
    return currentPos;
}
```