

CSC 427: Data Structures and Algorithm Analysis

Fall 2004

Lists and iterators

- existing kinds of lists: vector, stack, queue
- list class
- implementation
- iterators
- list-based applications

1

Lists in C++

recall from previous lectures

- we defined various classes for storing lists of items
- storing and accessing items in a list is very common in CS applications

C++ provides several useful classes for different types of lists

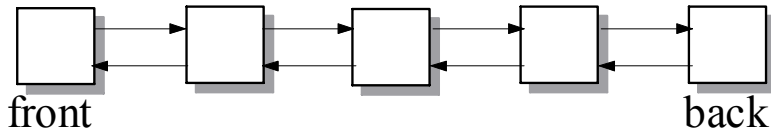
- **vector**: contiguous sequence of items, accessible via indexing
implemented using a dynamic array
 - ✓ access is $O(1)$
 - ✓ add at end is $O(1)^*$, add at beginning/middle is $O(N)$
 - ✓ remove from end is $O(1)$, remove from beginning/middle is $O(N)$
- **stack**: list at which access/add/remove all occur at same end
implemented using a linked list (?)
 - ✓ access/add/remove are all $O(1)$
- **queue**: list at which add and access/remove occur at opposite ends
implemented using a linked list (?)
 - ✓ access/add/remove are all $O(1)$

2

list class

C++ provides a more general list class with the following operations

- access/add/remove from either end in $O(1)$ time
- access/add/remove from the middle in $O(N)$ time
- traverse the entire list in $O(N)$ time
- implemented using a doubly-linked list, with a pointer to each end



3

list class

the `list` class is defined in the `<list>` library: `#include <list>`

- constructors include:
 - `list();` default constructor, creates empty list
 - `list(int size, TYPE & init = TYPE());` can specify initial size and values (if no value specified, default is used)

e.g., `list<string> words;`
`list<int> counts(10, 0);`

- has member functions for accessing/adding/removing from either end

```
TYPE & front();  
void push_front(const TYPE & newValue);  
void pop_front();  
  
TYPE & back();  
void push_back(const TYPE & newValue);  
void pop_back();  
  
bool empty() const; // true if no items  
void clear(); // removes all items
```

front, pop_front,
back, and pop_back
all assume that the
list is nonempty

4

DeckOfCards revisited

the list class would have been handy for the war game

- wanted to be able to remove from one end (top), add to other end (bottom)
- using a list, could do both of these in $O(1)$

```
class DeckOfCards
{
public:
    DeckOfCards();
    void Shuffle();
    Card DrawFromTop();
    bool IsEmpty() const;
protected:
    list<Card> cards;
};

Card DeckOfCards::DrawFromTop()
{
    Card topCard = cards.back();
    cards.pop_back();
    return topCard;
}

bool DeckOfCards::IsEmpty() const
{
    return cards.empty();
}
```

```
class PileOfCards : public DeckOfCards
{
public:
    PileOfCards();
    void AddToBottom(const Card & c);
};

PileOfCards::PileOfCards()
{
    cards.clear();
}

void PileOfCards::AddToBottom(const Card & c)
{
    cards.push_front(c);
}
```

what about Shuffle?

5

Implementing the list class

need to be able to represent a node with previous and next links

```
template <class Data> class Node
{
private:
    Data data; // data field
    Node<Data> * previousPtr; // pointer to previous node
    Node<Data> * nextPtr; // pointer to next node

public:
    Node(const Data & val, Node<Data> * previous, Node<Data> * next)
    {
        data = val;
        previousPtr = previous;
        nextPtr = next;
    }

    void setData(const Data & newValue)
    {
        data = newValue;
    }

    void setPrevious(Node<Data> * previous)
    {
        previousPtr = previous;
    }

    void setNext(Node<Data> * next)
    {
        nextPtr = next;
    }

    Data & getData()
    {
        return data;
    }

    Node<Data> * & getPrevious()
    {
        return previousPtr;
    }

    Node<Data> * & getNext()
    {
        return nextPtr;
    }
};
```

6

SimpleList class

```
template <class TYPE> class SimpleList
{
private:
    template <class Data> class Node { . . . } // hide Node definition inside SimpleList

    Node<TYPE> * frontPtr; // pointer to first item in the list (NULL if empty)
    Node<TYPE> * backPtr; // pointer to last item in the list (NULL if empty)

public:
    SimpleList() // default constructor - creates empty list
    {
        frontPtr = NULL;
        backPtr = NULL;
    }

    TYPE & front() // returns first item in the list (assumes not empty)
    {
        return frontPtr->getData();
    }

    TYPE & back() // returns last item in the list (assumes not empty)
    {
        return backPtr->getData();
    }

    . . .
}
```

7

SimpleList class (cont.)

```
void push_front(const TYPE & newValue) // adds newValue to front of list
{
    if (frontPtr == NULL) {
        frontPtr = new Node<TYPE>(newValue, NULL, NULL);
        backPtr = frontPtr;
    }
    else {
        frontPtr->setPrevious( new Node<TYPE>(newValue, NULL, frontPtr) );
        frontPtr = frontPtr->getPrevious();
    }
}

void push_back(const TYPE & newValue) // adds newValue to back of list
{
    if (backPtr == NULL) {
        backPtr = new Node<TYPE>(newValue, NULL, NULL);
        frontPtr = backPtr;
    }
    else {
        backPtr->setNext( new Node<TYPE>(newValue, backPtr, NULL) );
        backPtr = backPtr->getNext();
    }
}

. . .
```

8

SimpleList class (cont.)

```
void pop_front()          // removes first item in list (assumes list is not empty)
{
    if (frontPtr->getNext() == NULL) {
        delete frontPtr;
        frontPtr = NULL;
        backPtr = NULL;
    }
    else {
        frontPtr = frontPtr->getNext();
        delete frontPtr->getPrevious();
        frontPtr->setPrevious(NULL);
    }
}

void pop_back()          // removes last item in list (assumes list is not empty)
{
    if (backPtr->getPrevious() == NULL) {
        delete backPtr;
        backPtr = NULL;
        frontPtr = NULL;
    }
    else {
        backPtr = backPtr->getPrevious();
        delete backPtr->getNext();
        backPtr->setNext(NULL);
    }
}

. . .
```

9

SimpleList class (cont.)

```
bool empty()             // returns true if list is empty, else false
{
    return (frontPtr == NULL);
}

void clear()             // removes all items from the list
{
    while (!empty()) {
        pop_front();
    }
}

};
```

the list class also has the following member function

```
int size();              // returns number of items in the list
```

- how might we implement this? efficiently?

10

SimpleList destructor

whenever a class has dynamic data fields (i.e., allocated using `new`), should define a *destructor* to deallocate dynamic memory when done

- a destructor has the same name as the class, but preceded by a tilde
- for this class, the destructor is trivial – simply call `clear` to remove & deallocate nodes

```
~SimpleList()
{
    clear();
}
```

a copy constructor defines how to create a copy of an existing object

- the default copy constructor for a class does a field-by-field copy
- similarly, the default assignment operator does a field-by-field copy
- with dynamic fields, this does not do what you want
if you copy `frontPtr`, you just get another pointer to the same linked list!

11

SimpleList copy constructor

```
SimpleList(const SimpleList & copy) // copy constructor
{
    frontPtr = NULL;
    backPtr = NULL;

    for (Node<TYPE> * step = copy.frontPtr; step != NULL; step = step->getNext()) {
        push_back(step->getData());
    }
}

SimpleList<TYPE> & operator=(const SimpleList & rhs) // assignment operator
{
    if (this != &rhs) {
        clear();

        for (Node<TYPE> * step = rhs.frontPtr; step != NULL; step = step->getNext()) {
            push_back(step->getData());
        }
    }
    return *this;
}
```

12

Accessing items in the list

we *could* use indexing to access elements, as with vectors

```
TYPE & operator[](int index)
{
    Node<TYPE> * step = frontPtr;
    for (int i = 0; i < index; i++) {
        step = step->getNext();
    }
    return step->getData();
}

const TYPE & operator[](int index) const
{
    Node<TYPE> * step = frontPtr;
    for (int i = 0; i < index; i++) {
        step = step->getNext();
    }
    return step->getData();
}
```

how efficient is accessing an item?

```
cout << nums[nums.size()-1];
```

how efficient is traversing the list?

```
for (int i = 0; i < nums.size(); i++) {
    cout << nums[i] << endl;
}
```

13

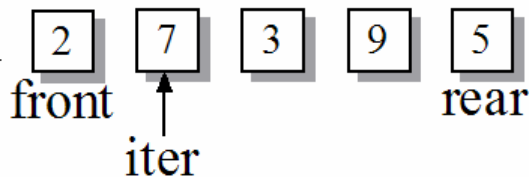
Iterators

using indexing, traversing a list of N items is $O(N^2)$

- $1 + 2 + 3 + \dots + N = N(N+1)/2$

better approach, use an *iterator* (i.e., a pointer) to step through the list

- given a pointer to a specific element, can
 - ✓ dereference to access the element value
 - ✓ move forward to the next element
 - ✓ move back to the previous element
 - ✓ also, need to be able to recognize the ends



14

Iterators

`iterator` is a supporting class defined for lists (and other collections)

- the list class has methods that return iterators

```
iterator begin();          const_iterator begin();
```

- returns an iterator that references the first position (front) of the list.
- if the list is empty, the iterator value `end()` is returned.

```
iterator end();          const_iterator end();
```

- returns an iterator pointing to a location immediately out of the range of actual elements.
- note: a program must not dereference the value of `end()`.

given an iterator, can access elements and step through a list

`*iter` accesses the value of the item currently pointed to by the iterator `itr`.

`iter++` moves the iterator to the next element in the list

`iter--` moves the iterator to the previous element in the list

`==` `!=` can be used to compare iterator values

15

List iterator examples

```
list<string> words;
words.push_back("foo");
words.push_front("boo");
words.push_back("zoo");

list<string>::iterator front = words.begin();
cout << *front << endl;

front++;
cout << *front << endl;

front--;
if (front == words.begin()) {
    cout << "Home again." << endl;
}
```

output:

```
boo
foo
Home again.
```

```
list<int> nums;
. . .

for (list<int>::iterator step = nums.begin(); step != nums.end(); nums++;) {
    cout << *step << endl;
}
```

iterates through each element & prints

16

Inserting into a list

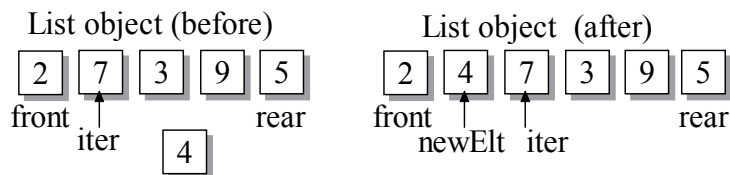
iterator `insert(iterator pos, const TYPE & value);`

- inserts value before pos, and return an iterator pointing to the position of the new value in list
- note: the operation does not affect any existing iterators

```
list<int> nums;
nums.push_back(2);
nums.push_back(7);
nums.push_back(3);
nums.push_back(9);
nums.push_back(5);

list<int>::iterator iter = nums.begin();
iter++;

nums.insert(iter, 4);
```



17

Inserting in order

once you have an iterator pointing to a spot in the list, inserting in the middle is an $O(1)$ operation

e.g., inserting into a sorted list \rightarrow traverse to the correct spot $O(N)$ + insert $O(1)$ $\rightarrow O(N)$

```
template <class Comparable>
void InsertInOrder(list<Comparable> & ordered, Comparable newItem)
{
    list<Comparable>::iterator step = ordered.begin();
    while (step != ordered.end() && newItem > *step) {
        step++;
    }

    ordered.insert(step, newItem);
}
```

18

Removing from a list

```
void erase(iterator pos);
```

- erases the element pointed to by pos

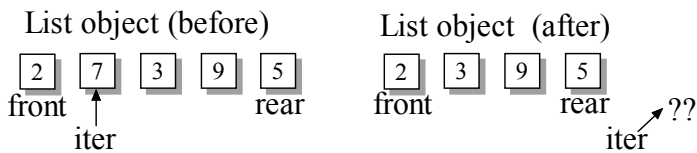
```
void erase(iterator first, iterator last);
```

- erases all list elements within the iterator range [first, last]

```
list<int> nums;
nums.push_back(2);
nums.push_back(7);
nums.push_back(3);
nums.push_back(9);
nums.push_back(5);

list<int>::iterator iter = nums.begin();
iter++;

nums.remove(iter);
```



19

Predefined algorithms

the `algorithm` library contains functions that apply to lists

```
#include <algorithm>
```

- `iterator find(iterator left, iterator right, const TYPE & value)`
- `bool binary_search(iterator left, iterator right, const TYPE & value)`
- `int count(iterator left, iterator right, const TYPE & value)`
- `const TYPE & min_element(iterator left, iterator right)`
- `const TYPE & max_element(iterator left, iterator right)`
- `void random_shuffle(iterator left, iterator right)`
- `void sort(iterator left, iterator right)`
- `void stable_sort(iterator left, iterator right)`
-
-
-

20

Example: word frequencies

suppose we wanted to analyze a file (e.g., a program) to determine the relative frequencies of words

- would need to store all words & associate a count with each word
- as read in a file, store each word in the list
 - if a new word, add to the list with a count of 1
 - if an existing word, increment its count
- display all words and their counts (in alphabetical order)

we could do it with a pair of parallel vectors (1 for words, 1 for counts)

we could do it with a vector of [word+frequency] objects

- in either case, searching and inserting are non-trivial tasks
- if the vector(s) unsorted, then
 - searching to see if a word is stored is $O(N)$; inserting a new word is $O(1)$
- if the vector(s) maintained in sorted order, then
 - searching to see if a word is stored is $O(\log N)$; inserting a new word is $O(N)$

21

Lists of word frequencies

we can abstract away some details using `list` and the `<algorithm>` library

- define a `WordFreq` class that encapsulates a word and a count
- utilize a list of `WordFreq` objects
- can use `find` to see if stored $O(N)$, `push_back` and `insert` to add elements $O(1)$
- if we wanted to display the words sorted by count, could use `sort`

```
class WordFreq
{
public:
    WordFreq(string str = "");
    string getWord() const;
    int getFreq() const;
    void increment();
    bool operator==(const string & str) const;
private:
    string word;
    int freq;
};
```

== operator that compares a WordFreq object with a string will prove useful

22

WordFreq implementation

```
WordFreq::WordFreq(string str)
{
    word = str;
    freq = 1;
}

string WordFreq::getWord() const
{
    return word;
}

int WordFreq::getFreq() const
{
    return freq;
}

void WordFreq::increment()
{
    freq++;
}

bool WordFreq::operator==(const string & str) const
{
    return (str == getWord());
}
```

initially, a `WordFreq` object has a count of 1

– WHY NOT 0?

a `WordFreq` object will `==` a string if its word field `==` the string

23

WordFreqList class

```
class WordFreqList
{
public:
    WordFreqList();
    WordFreqList(istream & istr);
    void addWord(string str);
    int getFreq(string str) const;
    void displayAlphabetical(ostream & ostr = cout) const;
private:
    list<WordFreq> words;
    string Normalize(const string & str) const;
};

////////////////////////////////////

WordFreqList::WordFreqList()
{
    // does nothing (list is empty)
}

WordFreqList::WordFreqList(istream & istr)
{
    string str;
    while (istr >> str) {
        addWord(str);
    }
}
```

note: two different constructors

- default constructor creates empty list
- constructor with `istream` reads and stores strings from that stream

24

WordFreqList class (cont.)

```
void WordFreqList::addWord(string str)
{
    str = Normalize(str);

    list<WordFreq>::iterator step = words.begin();
    while (step != words.end() && str > step->getWord()) {
        step++;
    }

    if (step == words.end() || str < step->getWord()) {
        words.insert(step, WordFreq(str));
    }
    else {
        step->increment();
    }
}

string WordFreqList::Normalize(const string & word) const
{
    string copy;
    for (int i = 0; i < word.length(); i++) {
        if (!ispunct(word[i])) {
            copy += tolower(word[i]);
        }
    }
    return copy;
}
```

addWord first normalizes
the string

iterates through the list
to either find &
increment the word, or
else add it in it correct
position

WHY NOT USE find?

25

WordFreqList class (cont.)

```
int WordFreqList::getFreq(string str) const
{
    str = Normalize(str);

    list<WordFreq>::const_iterator locate = find(words.begin(), words.end(), str);

    if (locate == words.end()) {
        return 0;
    }
    else {
        return locate->getFreq();
    }
}

void WordFreqList::displayAlphabetical(ostream & ostr) const
{
    for (list<WordFreq>::const_iterator step = words.begin(); step != words.end(); step++) {
        ostr << step->getWord() << " " << step->getFreq() << endl;
    }
}
```

getFreq uses the find function to locate the word (note use of operator==)
how efficient is find?

26

freq.cpp

using the `WordFreqList` class, processing a file is trivial

- while not the most efficient, development was easy due to the abstractions provided in `<algorithm>`

```
#include <iostream>
#include <fstream>
#include "WordFreqList.h"
using namespace std;

void OpenFile(ifstream & myin);

int main()
{
    ifstream ifstr;
    OpenFile(ifstr);

    WordFreqList words(ifstr);
    words.displayAlphabetical();

    return 0;
}

void OpenFile(ifstream & myin)
// Returns: input stream opened to user's file
{
    string filename;
    cout << "Enter the text file name: ";
    cin >> filename;
    myin.open( filename.c_str() );

    while (!myin) {
        cout << "File not found. Try again: ";
        cin >> filename;
        myin.clear();
        myin.open( filename.c_str() );
    }
}
```

27

Iterators and vectors

`WordFreqList` was easy to develop because of the predefined abstractions

- not the most efficient, since list iterators do not provide direct access
- `addWord` and `getFreq` are both $O(N)$

it turns out, iterators can be applied to vectors as well

→ you need never write another sort again!

```
vector<string> words; // CREATE VECTOR

words.push_back("foo"); // PUSH WORDS ONTO BACK OF VECTOR
words.push_back("biz");
words.push_back("baz");
words.push_back("oof");

sort( words.begin(), words.end() ); // SORT WORDS INTO INCREASING ORDER

for (int i = 0; i < words.size(); i++) { // DISPLAY SORTED WORDS
    cout << words[i] << endl; // baz, biz, foo, oof
}
```

```
for (vector<string>::iterator step = words.begin(); step != words.end(); step++) {
    cout << *step << endl;
}
```

28

Advanced sorting

the sort function uses the < operator defined for the element type

- if you want to sort by a different criteria, can provide a comparison function
- function should return true when 1st element comes before 2nd in the ordering

```
bool compare(const string & s1, const string & s2)
// Comparison function that defines reverse order of two strings
{
    return (s1 > s2);
}

////////////////////////////////////

sort( words.begin(), words.end(), compare ); // IF SPECIFY COMPARISON FUNCTION,
// WILL SORT USING IT

for (int i = 0; i < words.size(); i++) { // DISPLAYS SORTED WORDS
    cout << words[i] << endl; // oof, foo, biz, baz
}
```

29

More advanced sorting

better yet, can define operator< for a new class and make it automatic

- e.g. for sorting names, go by last name then first name

```
class Name // NAME CLASS, WITH FIRST & LAST NAMES
{
public:
    Name(string firstName, string lastName) { first = firstName; last = lastName; }
    void Display() { cout << first+" "+last << endl; }
    bool operator<(const Name & other) {
        return (last < other.last || (last == other.last && first < other.first));
    }
private:
    string first, last;
};

////////////////////////////////////

vector<Name> people; // CREATE VECTOR

people.push_back( Name("Dave", "Reed") ); // PUSH NAMES ONTO VECTOR
people.push_back( Name("Sue", "Reed") );
people.push_back( Name("Xavier", "Adams") );

sort( people.begin(), people.end() ); // SORT BASED ON operator< FOR Name CLASS

for (int i = 0; i < people.size(); i++) { // DISPLAY SORTED NAMES
    people[i].Display(); // Xavier Adams, Dave Reed, Sue Reed
}
```

30

Word frequency revisited

would a sorted vector be better than a sorted list for `WordFreqList`?

```
void WordFreqList::addWord(string str)
{
    str = Normalize(str);

    list<WordFreq>::iterator step = words.begin();
    while (step != words.end() && str > step->getWord()) {
        step++;
    }

    if (step == words.end() || str < step->getWord()) {
        words.insert(step, WordFreq(str));
    }
    else {
        step->increment();
    }
}
```

```
int WordFreqList::getFreq(string str) const
{
    str = Normalize(str);

    list<WordFreq>::const_iterator locate =
        find(words.begin(), words.end(), str);

    if (locate == words.end()) {
        return 0;
    }
    else {
        return locate->getFreq();
    }
}
```

31

Spell checker revisited

recall the task of spell checking a file

- read a dictionary of words from a file and store (e.g., `List`, `SortedList`, `LazyList`)
- read each word from a text file and search the dictionary
- if word not found, then report it as misspelled

instead of defining our own list types, we could use list or vector

- can then make use of `<algorithm>` functions, e.g., `insert`, `find`, `binary_search`
- if we use list, then adding a word is $O(N)$, searching is $O(N)$
- if we use vector, then adding a word is $O(N)$, searching is $O(\log N)$

additional feature: if word is not found, could ask user if they want it added

- when done processing the file, the dictionary file can be updated with new words

32

Dictionary class

we could use `list/vector` directly in the main program, but a dictionary is a useful structure

better to define a reusable class

```
class Dictionary
{
public:
    Dictionary();
    void read(istream & istr = cin);
    void write(ostream & ostr = cout);
    void addWord(const string & str);
    bool isStored(const string & str) const;
private:
    vector<string> words;
    string Normalize(const string & str) const;
};

////////////////////////////////////

Dictionary::Dictionary()
{
}

void Dictionary::read(istream & istr)
// Assumes: words stored in alphabetical order
{
    string word;
    while (istr >> word) {
        words.push_back(word);
    }
}

void Dictionary::write(ostream & ostr)
{
    for (int i = 0; i < words.size(); i++) {
        ostr << words[i] << endl;
    }
}
```

33

Dictionary class

```
void Dictionary::addWord(const string & str)
{
    string normal = Normalize(str);

    if (normal != "") {
        vector<string>::iterator step = words.begin();
        while (step != words.end() && normal > *step) {
            step++;
        }

        if (step == words.end() || normal < *step) {
            words.insert(step, normal);
        }
    }
}

bool Dictionary::isStored(const string & word) const
{
    return binary_search(words.begin(), words.end(), Normalize(word));
}

string Dictionary::Normalize(const string & word) const
{
    string copy;
    for (int i = 0; i < word.length(); i++) {
        if (!ispunct(word[i])) {
            copy += tolower(word[i]);
        }
    }
    return copy;
}
```

to add a word, iterate to find the spot and insert (if not already there)

since a vector, `binary_search` is $O(\log N)$

34

speller.cpp

first, read and store dictionary

then, read from input file and search – if not in dictionary, ask user if add

finally, write the updated dictionary back to the file

```
#include <iostream>
#include <fstream>
#include <string>
#include "Dictionary.h"
using namespace std;

const string DICTIONARY_FILE = "dict.txt";

void OpenFile(ifstream & myin);

int main()
{
    Dictionary dict;

    ifstream dictIn(DICTIONARY_FILE.c_str());
    dict.read(dictIn);
    dictIn.close();

    ifstream textFile;
    OpenFile(textFile);

    string word;
    while (textFile >> word) {
        if (!dict.isStored(word)) {
            cout << word << ": add to the dictionary? (y/n) ";
            string response;
            cin >> response;
            if (response[0] == 'y' || response[0] == 'Y') {
                dict.addWord(word);
            }
        }
    }

    ofstream dictOut(DICTIONARY_FILE.c_str());
    dict.write(dictOut);
    dictOut.close();

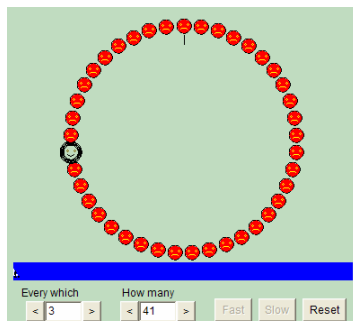
    return 0;
}
```

35

Josephus problem

consider the puzzle named after Josephus Flavius (1st century historian)

- During the Jewish-Roman war he got trapped in a cave with a group of 40 soldiers surrounded by romans.
- The legend has it that preferring suicide to capture, the Jews decided to form a circle and, proceeding around it, to kill every third remaining person until no one was left.
- Josephus, not keen to die, quickly found the safe spot in the circle and thus stayed alive.



see

www.cut-the-knot.org/recurrence/flavius.shtml

for a nice visualization

36

josephus.cpp

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

void Increment(list<int>::iterator & itr, list<int> & people);

int main()
{
    int numPeople, stepSize;
    cin >> numPeople >> stepSize;

    list<int> people;
    for (int i = 1; i <= numPeople; i++) {
        people.push_back(i);
    }

    list<int>::iterator counter = people.begin();
    while (people.size() > 1) {
        for (int i = 1; i < stepSize; i++) {
            Increment(counter, people);
        }

        list<int>::iterator temp = counter;
        Increment(counter, people);
        people.erase(temp);
    }

    cout << "With " << numPeople << " people, counting by " << stepSize
        << ", the survivor is in position " << *counter << "." << endl;

    return 0;
}

void Increment(list<int>::iterator & itr, list<int> & people)
// increments itr, wrapping around if necessary
{
    itr++;
    if (itr == people.end()) {
        itr = people.begin();
    }
}
```

note: this problem really calls for a circularly linked list – can simulate using Increment

In-class exercises:

- find the location where Josephus stood (41 people, skip 3)
- what if there were only 40 people?
- what if there were 41 people but skip 5?

add code to display the sequence of numbers after each removal

e.g., 5 people, skip 3:

```
1 2 3 4 5
1 2 4 5
2 4 6
2 4
4
```

modify your code so that sequences are displayed with current number first

e.g., 5 people, skip 3:

```
1 2 3 4 5
4 5 1 2
2 4 6
2 4
4
```

consider the following wrinkle (from a programming contest problem):

- when a person is killed, the person SKIP spots away buries them then takes their place