CSC 222: Computer Programming II

Spring 2004

Searching and efficiency
- sequential search
- big-Oh, rate-of-growth
- binary search

Class design
- templated classes
- inheritance, virtual, protected

Searching a list

suppose you have a list, and want to find a particular item, e.g.,
- lookup a word in a dictionary
- find a number in the phone book
- locate a student's exam from a pile

searching is a common task in computing
- searching a database
- checking a login password
- lookup the value assigned to a variable in memory

if the items in the list are unordered (e.g., added at random)
- desired item is equally likely to be at any point in the list
- need to systematically search through the list, check each entry until found

➔ sequential search

Sequential search

sequential search traverses the list from beginning to end
- check each entry in the list
- if matches the desired entry, then FOUND
- if traverse entire list and no match, then NOT FOUND

```
bool IsStored(const vector<string> & words, string desired)
// postcondition: returns true if desired in words, else false
{
    for(int k=0; k < words.size(); k++) {
        if (words[k] == word) {
            return true;
        }
    }
    return false;
}
```

recall `WordList` class (from the word frequency program)
- stored words and frequencies in parallel vectors
- `Locate` member function used sequential search to locate a word in the vector

3

How efficient is sequential search?

for this algorithm, the dominant factor in execution time is checking an item
- the number of checks will determine efficiency

in the worst case:
- the item you are looking for is in the last position of the list (or not found)
- requires traversing and checking every item in the list

- if 100 or 1,000 entries → NO BIG DEAL
- if 10,000 or 100,000 entries → NOTICEABLE

in the average case?

in the best case?

4

Big-Oh notation

to represent an algorithm's performance in relation to the size of the problem, computer scientists use *Big-Oh* notation

an algorithm is O(N) if the number of operations required to solve a problem is proportional to the size of the problem

sequential search on a list of N items requires *roughly* N checks (+ other constants)
➔ O(N)

for an O(N) algorithm, doubling the size of the problem requires double the amount of work (in the worst case)

- if it takes 1 second to search a list of 1,000 items, then
  it takes 2 seconds to search a list of 2,000 items
  it takes 4 seconds to search a list of 4,000 items
  it takes 8 seconds to search a list of 8,000 items
  . . .

Searching an ordered list

when the list is unordered, can't do any better than sequential search
- but, if the list is ordered, a better alternative exists

e.g., when looking up a word in the dictionary or name in the phone book
- can take ordering knowledge into account
- pick a spot – if too far in the list, then go backward; if not far enough, go forward

binary search algorithm
- check midpoint of the list
- if desired item is found there, then DONE
- if the item at midpoint comes after the desired item in the ordering scheme, then repeat the process on the left half
- if the item at midpoint comes before the desired item in the ordering scheme, then repeat the process on the right half

Binary search

```
bool IsStored(const vector<string> & words, string desired)
// postcondition: returns true if desired in words, else false
{
    int left = 0, right = items.size()-1; // maintain bounds on where item must be
    while (left <= right) {
        int mid = (left+right)/2;
        if (item == items[mid]) {          // if at midpoint, then DONE
            return true;
        }
        else if (item < items[mid]) {      // if less than midpoint, focus on left half
            right = mid-1;
        }
        else {                             // otherwise, focus on right half
            left = mid + 1;
        }
    }
    return false;                          // if reduce to empty range, NOT FOUND
}
```

note: each check reduces the range in which the item can be found by half
- see www.creighton.edu/~davereed/csc107/search.html for demo

7

How efficient is binary search?

again, the dominant factor in execution time is checking an item
- the number of checks will determine efficiency

in the worst case:
- the item you are looking for is in the first or last position of the list (or not found)

  start with N items in list
  after 1st check, reduced to N/2 items to search
  after 2nd check, reduced to N/4 items to search
  after 3rd check, reduced to N/8 items to search
  . . .
  after $\lceil \log_2 N \rceil$ checks, reduced to 1 item to search

in the average case?

in the best case?

8

Big-Oh notation

an algorithm is O(log N) if the number of operations required to solve a problem is proportional to the logarithm of the size of the problem

> binary search on a list of N items requires *roughly* $\log_2 N$ checks (+ other constants)
> ➔ O(log N)

for an O(log N) algorithm, doubling the size of the problem adds only a constant amount of work

- if it takes 1 second to search a list of 1,000 items, then
  searching a list of 2,000 items will take time to check midpoint + 1 second
  searching a list of 4,000 items will take time for 2 checks + 1 second
  searching a list of 8,000 items will take time for 3 checks + 1 second
  . . .

Comparison: searching a phone book

Number of entries in phone book	Number of checks performed by sequential search	Number of checks performed by binary search
100	100	7
200	200	8
400	400	9
800	800	10
1,600	1,600	11
…	…	…
10,000	10,000	14
20,000	20,000	15
40,000	40,000	16
…	…	…
1,000,000	1,000,000	20

to search a phone book of the United States (~280 million) using binary search?

to search a phone book of the world (6 billion) using binary search?

Searching example

for small N, the difference between O(N) and O(log N) may be negligible

consider the following large-scale application: a spell checker
- want to read in and store a dictionary of words
- then, process a text file one word at a time
  - if word is not found in dictionary, report as misspelled

- since the dictionary file is large (~60,000 words), the difference is clear

we could define a Dictionary class to store & access words
- constructor, Add, IsStored, NumStored, DisplayAll, ...

better yet, define a generic List class to store & access any type of data
- utilize template (as in vector) to allow for any type

```
List<string> words;          List<int> grades;
```

Templated classes

when defining a templated class
- begin class definition with
  template <class *ItemType*>
  which gives a name to the arbitrary type involved
- within the class definition, use ItemType for the type
- when an actual instance is declared, e.g.,
  List<string> words;
  the specified type will be substituted for ItemType

templated classes must be defined in one file

```cpp
template <class ItemType>
class List
{
  public:
    List<ItemType>()
    // constructor, creates an empty list
    {
      //does nothing
    }

    void Add(const ItemType & item)
    // Results: adds item to the end of the list
    {
        items.push_back(item);
    }

    bool IsStored(const ItemType & item) const
    // Returns: true if item is stored in list, else false
    {
        for (int i = 0; i < items.size(); i++) {
            if (item == items[i]) {
                      return true;
            }
        }
        return false;
    }

    int NumItems() const
    // Returns: number of items in the list
    {
        return items.size();
    }

    void DisplayAll() const
    // Results: displays all items in the list, one per line
    {
        for (int i = 0; i < items.size(); i++) {
           cout << items[i] << endl;
        }
    }
  private:
    vector<ItemType> items;
};
```

Spell checker (spell1.cpp)

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <cassert>
#include "List.h"
using namespace std;

const string DICTIONARY_FILE = "dict.txt";

void OpenFile(ifstream & myin);
void ReadDictionary(List<string> & dict);
string Normalize(string word);

int main()
{
  List<string> dictionary;

  ReadDictionary(dictionary);

  ifstream textFile;
  OpenFile(textFile);

  cout << endl << "MISSPELLED WORDS" << endl
               << "----------------" << endl;

  string word;
  while (textFile >> word) {
    word = Normalize(word);
    if (!dictionary.IsStored(word)) {
      cout << word << endl;
    }
  }

  return 0;
}
```

```cpp
void OpenFile(ifstream & myin)
// Results: myin is opened to the user's file
{
  // AS BEFORE
}

void ReadDictionary(List<string> & dict)
{
  ifstream myDict(DICTIONARY_FILE.c_str());
  assert(myDict);

  cout << "Please wait while file loads... ";

  string word;
  while (myDict >> word) {
    dict.Add(word);
  }
  myDict.close();

  cout << "DONE!" << endl << endl;
}

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

13

In-class exercise

copy the following files
- www.creighton.edu/~davereed/csc222/Code/List.h
- www.creighton.edu/~davereed/csc222/Code/spell1.cpp
- www.creighton.edu/~davereed/csc222/Code/dict.txt
- www.creighton.edu/~davereed/csc222/Code/gettysburg.txt

create a project and spell check the Gettysburg address
- is the delay noticeable?

now download www.creighton.edu/~davereed/csc222/doubledict.txt

and spell check the Gettysburg address with that
- doubledict.txt is twice as big (dummy value inserted between each word)
- does it take roughly twice as long?

14

SortedList class

could define another class to represent sorted lists

- `Add` must add items in sorted order

- `IsStored` can use binary search

- data field and other member functions identical to `List`

code duplication is troubling…

```cpp
template <class ItemType>
class SortedList
{
  public:
    SortedList<ItemType>() { /* DOES NOTHING */ }

    void Add(const ItemType & item)
    // Results: adds item to the list in order
    {
      items.push_back(item);
      int i;
      for (i = items.size()-1; i > 0 && items[i-1] > item; i--) {
        items[i] = items[i-1];
      }
      items[i] = item;
    }

    bool IsStored(const ItemType & item) const
    // Returns: true if item is stored in list, else false
    {
      int left = 0, right = items.size()-1;
      while (left <= right) {
        int mid = (left+right)/2;
        if (item == items[mid]) {
          return true;
        }
        else if (item < items[mid]) {
          right = mid-1;
        }
        else {
          left = mid + 1;
        }
      }
      return false;
    }

    int NumItems() const { /* AS BEFORE */ }

    void DisplayAll() const { /* AS BEFORE */ }

  private:
    vector<ItemType> items;  /* AS BEFORE */
};
```

15

Inheritance

C++ provides a better mechanism: inheritance
- if have an existing class, and want to define a new class that extends it
- can *derive* a new class from the existing (*parent*) class

- a *derived* class inherits all data fields and member functions from its *parent*

- can add new member functions (or reimplement existing ones) in the derived class

- inheritance defines an *"IS A" relationship*: an instance of the desired class IS A instance of the parent class, just more specific

example: `List` → `SortedList`
- `List` & `SortedList` have the same data field (vector of `ItemType`)
- constructor, `NumItems`, `DisplayAll` are identical
- `Add` & `IsStored` differ (`List`: add at end, sequential search; `SortedList`: add in order, binary search)

- a `SortedList` IS A `List`

16

Inheritance

to define a derived class

- at the end of the first line of the class definition, add

  `: public ParentClass`

  which specifies the parent from which to inherit

- then, only define new data/functions, or functions being overridden

- parent class constructor is automatically called when constructing an object of derived class (but should always have constructor even if does nothing)

```
template <class ItemType>
class SortedList : public List<ItemType>
{
  public:
    SortedList<ItemType>()
    // constructor (note: List constructor implicitly called first)
    {
      // does nothing
    }

    void Add(const ItemType & item)
    // Results: adds item to the list in order
    {
      items.push_back(item);
      int i;
      for (i = items.size()-1; i > 0 && items[i-1] > item; i--) {
        items[i] = items[i-1];
      }
      items[i] = item;
    }

    bool IsStored(const ItemType & item) const
    // Returns: true if item is stored in list, else false
    {
      int left = 0, right = items.size()-1;
      while (left <= right) {
        int mid = (left+right)/2;
        if (item == items[mid]) {
          return true;
        }
        else if (item < items[mid]) {
          right = mid-1;
        }
        else {
          left = mid + 1;
        }
      }
      return false;
    }
};
```

17

Inheritance vs. reimplementation

using inheritance is preferable to reimplementing the class from scratch
- derived class is simpler/cleaner
- no duplication of code
- if change implementation of code in parent class, derived class automatically updated
- object of derived class is still considered to be an object of parent class
    e.g., could pass a `SortedList` to a function expecting a `List`

inheritance approach does require some modifications to List class
- private data is hidden from derived class as well as client program
    instead, use 3rd protection level: protected
    protected data is accessible to derived classes, but not client program

- member functions to be overridden should be declared `virtual` in parent class
    necessary if want to have a function that works on both `List` and `SortedList`

18

parent List class

protected specifies that the vector will be accessible to the SortedList class

virtual specifies that Add and IsStored can be overridden and handled correctly

- if pass a SortedList to a function expecting a List, should still recognize it as a SortedList

```cpp
template <class ItemType>
class List
{
  public:
    List<ItemType>()
    // constructor, creates an empty list
    {
      //does nothing
    }

    virtual void Add(const ItemType & item)
    // Results: adds item to the end of the list
    {
        items.push_back(item);
    }

    virtual bool IsStored(const ItemType & item) const
    // Returns: true if item is stored in list, else false
    {
        for (int i = 0; i < items.size(); i++) {
            if (item == items[i]) {
                        return true;
            }
        }
        return false;
    }

    int NumItems() const
    // Returns: number of items in the list
    {
        return items.size();
    }

    void DisplayAll() const
    // Results: displays all items in the list, one per line
    {
        for (int i = 0; i < items.size(); i++) {
            cout << items[i] << endl;
        }
    }
  protected:
    vector<ItemType> items;
};
```

19

Spell checker (spell2.cpp)

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <cassert>
#include "SortedList.h"
using namespace std;

const string DICTIONARY_FILE = "dict.txt";

void OpenFile(ifstream & myin);
void ReadDictionary(List<string> & dict);
string Normalize(string word);

int main()
{
  SortedList<string> dictionary;

  ReadDictionary(dictionary);

  ifstream textFile;
  OpenFile(textFile);

  cout << endl << "MISSPELLED WORDS" << endl
            << "----------------" << endl;

  string word;
  while (textFile >> word) {
    word = Normalize(word);
    if (!dictionary.IsStored(word)) {
      cout << word << endl;
    }
  }

  return 0;
}
```

```cpp
void OpenFile(ifstream & myin)
// Results: myin is opened to the user's file
{
  // AS BEFORE
}

void ReadDictionary(List<string> & dict)
{
  ifstream myDict(DICTIONARY_FILE.c_str());
  assert(myDict);

  cout << "Please wait while file loads... ";

  string word;
  while (myDict >> word) {
    dict.Add(word);
  }
  myDict.close();

  cout << "DONE!" << endl << endl;
}

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

20

In-class exercise

copy the following files
- www.creighton.edu/~davereed/csc222/Code/List.h
- www.creighton.edu/~davereed/csc222/Code/SortedList.h
- www.creighton.edu/~davereed/csc222/Code/spell2.cpp
- www.creighton.edu/~davereed/csc222/Code/dict.txt
- www.creighton.edu/~davereed/csc222/Code/gettysburg.txt

create a project and spell check the Gettysburg address
- is it noticeably faster than sequential search?

now download www.creighton.edu/~davereed/csc222/doubledict.txt

and spell check the Gettysburg address with that
- `doubledict.txt` is twice as big (dummy value inserted between each word)
- how much longer does it take?