

CSC 427: Data Structures and Algorithm Analysis

Fall 2004

Dynamic memory structures

- vectors of pointers
- sequentialSearch, binarySearch, mergeSort, List, SortedList, LazyList revisited

Matrix class

- encapsulates 2-dimensional vector
- Bitmap class (pbm format)
- Pixmap class (pbm, pgm, ppm formats)

1

Sorting revisited...

recall from lectures

- sorting involves some # of inspections and some # of swaps
- the amount of time required for a swap is dependent on the size of the values

```
vector<int> nums(10000);    selectionSort(nums);  
  
vector<string> names(10000);    selectionSort(names);  
  
vector<IRSTaxRecord> audits(10000);    selectionSort(audits);
```

selection sort: $O(N^2)$ comparisons + $O(N)$ swaps, where each swap = 3 assignments

merge sort: $O(N \log N)$ comparisons + $O(N \log N)$ swaps, where each swap = 2 assigns

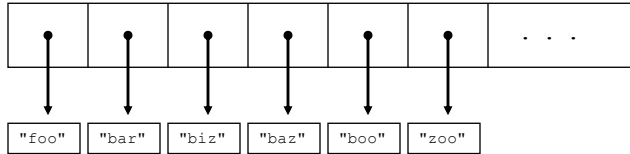
- if the items being sorted are large, then swap time can add up!

POSSIBLE WORKAROUNDS?

2

Lists of pointers

idea: instead of storing the (potentially large) items in the vector, store them elsewhere and keep pointers (addresses)



can sort as before

- when comparing values, have to follow the pointers to the actual data
- when swapping, copy the pointers, not the data values
pointers are addresses (integers), so copying is fast

note: since all pointers are same size, swap time is independent of data size

3

Tracing code...

```
template <class Type> void Swap(Type & x, Type & y)
{
    Type temp = x;
    x = y;
    y = temp;
}
```

```
int a = 5, b = 9;
Swap(a, b);
cout << a << " " << b << endl;
```

```
int * p1 = new int;
*p1 = 33;
int * p2 = new int;
*p2 = 45;
cout << a << " " << b << endl;
```

```
Swap(*p1, *p2);
cout << a << " " << b << endl;
cout << *p1 << " " << *p2 << endl;
```

```
Swap(p1, p2);
cout << a << " " << b << endl;
cout << *p1 << " " << *p2 << endl;
```

4

Reimplementing merge sort

```
template <class Comparable> void mergePtr(vector<Comparable *> & items, int low, int high)
{
    vector<Comparable *> copy;
    int size = high-low+1, middle = (low+high+1)/2; // middle rounds up if even
    int front1 = low, front2 = middle;
    for (int i = 0; i < size; i++) {
        if (front2 > high || (front1 < middle && *(items[front1]) < *(items[front2]))) {
            copy.push_back(items[front1]);
            front1++;
        }
        else {
            copy.push_back(items[front2]);
            front2++;
        }
    }
    for (int k = 0; k < size; k++) {
        items[low+k] = copy[k];
    }
}

template <class Comparable> void mergeSortPtr(vector<Comparable *> & items, int low, int high)
{
    if (low < high) {
        int middle = (low + high)/2;
        mergeSortPtr(items, low, middle);
        mergeSortPtr(items, middle+1, high);
        mergePtr(items, low, high);
    }
}

template <class Comparable> void mergeSortPtr(vector<Comparable *> & items)
{
    mergeSortPtr(items, 0, items.size()-1);
}
```

vector stores pointers
must dereference before
comparing elements
copy pointers, not values

5

Comparing the sorts

```
#include <iostream>
#include <ctime>
#include <vector>
#include <string>
#include "Die.h"
#include "Sorts.h"
using namespace std;

const int NUM_WORDS = 1000;
const int STRING_LENGTH = 8;

string randomString(int len);

int main()
{
    vector<string> words1;
    vector<string *> words2;
    for (int i = 0; i < NUM_WORDS; i++) {
        string str = randomString(STRING_LENGTH);
        words1.push_back(str);
        words2.push_back(new string(str));
    }

    clock_t start = clock();
    mergeSort(words1);
    clock_t stop = clock();
    cout << "Merge sort required " << stop-start << " milliseconds" << endl;

    start = clock();
    mergeSortPtr(words2);
    stop = clock();
    cout << "Merge sort with pointers required " << stop-start << " milliseconds" << endl;

    return 0;
}
```

can compare the two versions of merge sort by constructing two vectors (containing strings & pointers to strings)

which should be faster?

rate of growth?

# strings	words1	words2
1000	370 msec	100 msec
2000	601 msec	200 msec
4000	1272 msec	451 msec
...		

6

SortedList & LazyList revisited

recall the SortedList and LazyList classes

- SortedList: stores a vector of items
add operation inserts item in order
isStored performs binary search
- LazyList: stores a vector of items
add operation inserts item at end
isStored performs merge sort if not already sorted, then binary search

we could improve the performance of the SortedList and LazyList classes by storing pointers in the vector

- note: member functions are unchanged, so change is transparent to client
- underlying data structure changes, and member functions that access it

7

ListPtr.h

```
template <class ItemType>
class List
{
public:
    List<ItemType>()
    {
        // does nothing
    }

    virtual void add(const ItemType & item)
    {
        items.push_back(new ItemType(item));
    }

    virtual bool isStored(const ItemType & item)
    {
        return (sequentialSearch(items, item) >= 0);
    }

    int numItems() const
    {
        return items.size();
    }

    void displayAll() const
    {
        for (int i = 0; i < items.size(); i++) {
            cout << *(items[i]) << endl;
        }
    }

protected:
    vector<ItemType *> items;    // storage for pointers
};
```

add must allocate space for a copy,
then add a pointer to that copy to the
vector

isStored must use the pointer
version of sequential search

displayAll must dereference
the pointers when displaying

items stores pointers

8

Reimplementing the searches

```
template <class Type>
int sequentialSearch(const vector<Type *> & items, Type desired)
// precondition: returns index of desired in items, else -1 if not found
{
    for(int k=0; k < items.size(); k++) {
        if (*(items[k]) == desired) {
            return k;
        }
    }
    return -1;
}
```

note: don't need new names since parameter types differentiate from existing functions

```
template <class Comparable>
int binarySearch(const vector<Comparable *> & items, Comparable desired)
// precondition: returns index of desired in items, else -1 if not found
{
    int left = 0, right = items.size()-1; // maintain bounds on where item must be
    while (left <= right) {
        int mid = (left+right)/2;
        if (desired == *(items[mid])) { // if at midpoint, then DONE
            return mid;
        }
        else if (desired < *(items[mid])) { // if less than midpoint, focus on left half
            right = mid-1;
        }
        else { // otherwise, focus on right half
            left = mid + 1;
        }
    }
    return -1; // if reduce to empty range, NOT FOUND
}
```

9

SortedListPtr.h

```
template <class ItemType>
class SortedList : public List<ItemType>
{
public:
    SortedList<ItemType>()
    {
        // does nothing
    }

    void add(const ItemType & item)
    {
        ItemType * newPtr = new ItemType(item);
        items.push_back(newPtr);

        int i;
        for (i = items.size()-1; i > 0 && *(items[i-1]) > item; i--) {
            items[i] = items[i-1];
        }
        items[i] = newPtr;
    }

    bool isStored(const ItemType & item)
    {
        return (binarySearch(items, item) >= 0);
    }
};
```

sortedList inherits the vector of pointers

to add new item, allocate space and shift pointers (not data) to insert in order

isStored calls the pointer version of binarySearch

10

LazyListPtr.h

```
template <class ItemType>
class LazyList : public SortedList<ItemType>
{
public:
    LazyList<ItemType>()
    // constructor, creates an empty list
    {
        isSorted = true;
    }

    void add(const ItemType & item)
    // Results: adds item to the list in order
    {
        List<ItemType>::add(item);
        isSorted = false;
    }

    bool isStored(const ItemType & item)
    // Returns: true if item is stored in list, else false
    {
        if (!isSorted) {
            mergeSortPtr(items);
            isSorted = true;
        }
        return SortedList<ItemType>::isStored(item);
    }

private:
    bool isSorted;
};
```

since LazyList calls member functions from the parent classes, no major changes needed

isStored does need to call the pointer version of merge sort

11

Timing the different versions

using random 20-letter words:

List size: 1000

SortedList: 1913 msec
LazyList: 390 msec

SortedList with pointers: 1212 msec
LazyList with pointers: 110 msec

List size: 2000

SortedList: 6640 msec
LazyList: 881 msec

SortedList with pointers: 4517 msec
LazyList with pointers: 270 msec

List size: 4000

SortedList: 25908 msec
LazyList: 1832 msec

SortedList with pointers: 17826 msec
LazyList with pointers: 481 msec

List size: 8000

SortedList: 105331 msec
LazyList: 4086 msec

SortedList with pointers: 73285 msec
LazyList with pointers: 1022 msec

12

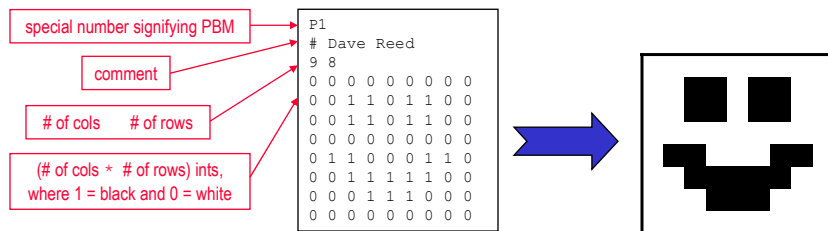
2-dimensional vectors

for HW1, many of you wanted a 2-D vector to store player scores

- got messy since data structure details got mixed up with algorithmic details
- data abstraction is a GOOD THING
- still, there are times when a 2-D array/vector is the right thing

for example, consider image formatting:

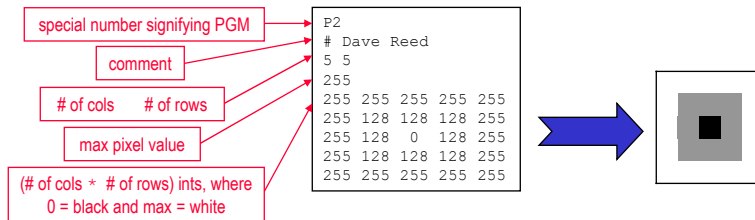
- the simplest image formats use ASCII characters to represent pixels in the image
Portable Bit Map (PBM), Portable Grayscale Map (PGM), Portable Pixel Map (PPM)



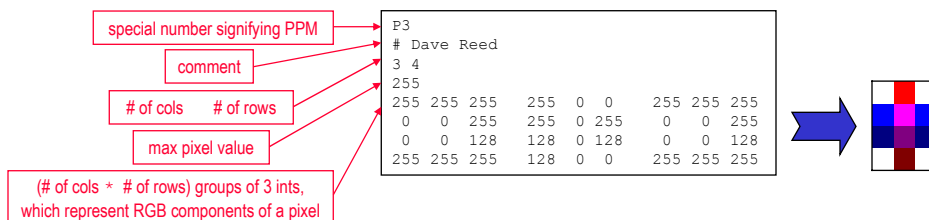
13

Portable image formats

PGM (portable grayscale map) similarly formats grayscale images



PPM (portable pixel map) similarly formats color images



14

Storing portable image maps

for HW3, you will need to be able to store PBM, PGM, and PPM images

- perform various operations on the images (e.g., sharpen, blur, rotate)

we will define a class hierarchy for pixels (monochrome, grayscale, color)

each type of image stores a matrix of pixels

- we could declare a vector of vectors (or even work with a 1-D vector?!?!?)
- better to define a new class for storing and manipulating matrices (2-D vectors)

15

Matrix class

```
template <class Item> class Matrix
{
private:
    vector< vector<Item> > mat;    // the matrix of items
    int numRows;                // # of rows (capacity)
    int numCols;                // # of cols (capacity)

public:
    Matrix(int rows = 0, int cols = 0)
    // constructor -- creates matrix w/ dimensions rows x cols
    {
        resize(rows, cols);
    }

    void resize(int newRows, int newCols)
    // resizes matrix to specified dimensions
    {
        numRows = newRows;
        numCols = newCols;
        mat.resize(numRows);
        for (int k=0; k < numRows; k++)
        {
            mat[k].resize(numCols);
        }
    }
    . . .
}
```

16

Matrix class (cont.)

```
int rows() const // returns row capacity of matrix
{
    return numRows;
}

int cols() const // returns column capacity of matrix
{
    return numCols;
}

vector<Item> & operator[](int index) // returns reference to a row
{
    return mat[index];
}

const vector<Item> & operator[](int index) const // returns const reference to a row
{
    return mat[index];
}

friend ostream & operator<<(ostream & ostr, const Matrix<Item> & Rhs)
// friend function that displays the matrix contents to ostr
{
    for (int r = 0; r < Rhs.rows(); r++) {
        for (int c = 0; c < Rhs.cols(); c++) {
            ostr << Rhs.mat[r][c] << " ";
        }
        ostr << endl;
    }
    return ostr;
}
};
```

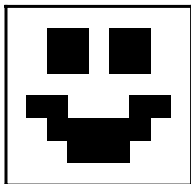
a "friend" function is a stand-alone function that is trusted with access to private data

17

Bitmap.h

for starters, let us only consider monochrome images (PBM)

```
P1
# Dave Reed
9 8
0 0 0 0 0 0 0 0 0
0 0 1 1 0 1 1 0 0
0 0 1 1 0 1 1 0 0
0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 1 1 0
0 0 1 1 1 1 1 0 0
0 0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0
```



```
#include <string>
#include "Matrix.h"
using namespace std;

class Bitmap{

public:
    Bitmap();
    void Read(istream &); // read image from file
    void HorizReflect(); // reflect on x-axis
    void VertReflect(); // reflect on y-axis
    void Invert(); // convert to mirror image
    void Write(ostream &); // write image to file

private:
    string kind; // P1, P2, etc.
    string creator; // program that created it
    int numRows, numCols; // dimensions of image
    Matrix<short int> image; // 2-d array of "pixels"
};
```

18

Bitmap.cpp

```
Bitmap::Bitmap()
// constructor -- initializes an empty pixel-map
{
    kind = "unknown";
    creator = "unknown";
    numRows = 0; numCols = 0;
}
```

```
void Bitmap::Read(istream & input)
// Assumes: input file contains a pixel-map
// Results: initializes empty pixel-map
{
    if (input){
        getline(input, kind);           // read P1, P2, P1, etc
        if (kind == "P1"){
            getline(input, creator);    // read comment (creating program)

            input >> numCols >> numRows; // read number of rows and cols
            image.resize(numRows, numCols);

            for(int j=0; j < numRows; j++){ // read individual pixels
                for(int k=0; k < numCols; k++){
                    input >> image[j][k];
                }
            }
        }
        else {
            cout << "UNKNOWN PICTURE TYPE" << endl;
        }
    }
    else {
        cout << "FILE NOT FOUND" << endl;
    }
}
```

19

Bitmap.cpp (cont.)

```
void Bitmap::HorizReflect()
// Results: reflects picture along x-axis
{
    for (int row = 0; row < numRows/2; row++) {
        for (int col = 0; col < numCols; col++) {
            short int temp = image[row][col];
            image[row][col] = image[numRows-row-1][col];
            image[numRows-row-1][col] = temp;
        }
    }
}
```

```
void Bitmap::VertReflect()
// Results: reflects picture along y-axis
{
    for (int row = 0; row < numRows; row++) {
        for (int col = 0; col < numCols/2; col++) {
            short int temp = image[row][col];
            image[row][col] = image[row][numCols-col-1];
            image[row][numCols-col-1] = temp;
        }
    }
}
```

```
void Bitmap::Invert()
// Results: inverts picture by setting pixel = maxPixel-pixel
{
    for (int row = 0; row < numRows; row++) {
        for (int col = 0; col < numCols; col++) {
            image[row][col] = 1 - image[row][col];
        }
    }
}
```

20

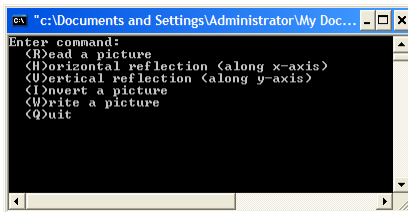
Bitmap.cpp (cont.)

```
void Bitmap::Write(ostream & output)
// Assumes: output is open for writing
// Results: Bitmap information written to output
{
    output << kind << endl;
    output << "# CREATOR: reedd pix program v 2.2" << endl;
    output << numCols << " " << numRows << endl;

    for (int j=0; j < numRows; j++){
        for (int k=0; k < numCols; k++){
            output << image[j][k] << " ";
        }
        output << endl;
    }
}
```

we now need a main program for reading/manipulating/writing images

- many commercial products exist for this; most utilize nice GUI
- we will build a simple, command-line interface



21

pix.cpp

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

char GetCommand();

int main()
{
    Bitmap pic;
    char command = '?';

    while (command != 'q') {
        command = GetCommand();

        switch (command) {
            case 'r': {
                string filename;
                cout << "Enter input file name: ";
                cin >> filename;
                ifstream istr(filename.c_str());

                pic.Read(istr);
            } break;
            case 'h': {
                pic.HorizReflect();
            } break;
            case 'v': {
                pic.VertReflect();
            } break;
        }
    }
}
```

```
case 'i': {
    pic.Invert();
}
break;
case 'w': {
    string filename;
    cout << "Enter output file name: ";
    cin >> filename;
    ofstream ostr(filename.c_str());

    pic.Write(ostr);
}
break;
case 'q': {
    cout << "Goodbye" << endl;
}
break;
default : {
    cout << "Unknown command." << endl;
}
break;
}
cout << endl;
}

return 0;
}

////////////////////////////////////
char GetCommand() { ... }
```

22

How do we generalize to PGM & PPM?

OPTION 1: define a separate class for each type of image

```
#include <string>
#include "Matrix.h"
using namespace std;

class Graymap{

public:
    Graymap();
    void Read(istream &);
    void HorizReflect();
    void VertReflect();
    void Invert();
    void Write(ostream &);

private:
    string kind;
    string creator;
    int numRows, numCols;
    int maxValue;
    Matrix<int> image;
};
```

```
#include <string>
#include "Matrix.h"
using namespace std;

enum Hue {RED, GREEN, BLUE};

class Color {
public:
    Color();
    Color(int R, int G, int B);
    int GetHue(Hue h)
    void SetHue(Hue h, int val);
private:
    int redValue, greenValue, blueValue;
};

class Colormap{

public:
    Colormap();
    void Read(istream &);
    void HorizReflect();
    void VertReflect();
    void Invert();
    void Write(ostream &);

private:
    string kind;
    string creator;
    int numRows, numCols;
    int maxValue;
    Matrix<Color> image;
};
```

23

Pixmap hierarchy

to be general, we could even link these three classes into a hierarchy

- define abstract Pixmap class from which Bitmap, Graymap, and Colormap are derived
- this would allow for general functions that operate on Pixmap
polymorphism would ensure that the correct method was applied to the map

```
void SaveMirror(Pixmap & img)
{
    img.Invert();
    img.Write("mirror.img");
}
```

not a very good solution

- lots of duplication in the code for the three classes
but enough differences to rule out inheriting entire methods
- when reading from a file, would need to read first line before know which image type

24

Better yet: Pixel inheritance

OPTION 2: have a single Pixmap class that can store any of the three

- note that methods are the same for all three classes
- only difference is type of value in the matrix (and maxValue for PGM and PPM)
- implementations of member functions will differ depending on the type

- define abstract Pixel class, with 3 derived classes: MonoPixel, GrayPixel, ColorPixel

- each derived class provides basic operations for that type of pixel
e.g., read from file
invert (given max value)
scale by some factor
add to another pixel
write to a file

```
#include <string>
#include "Matrix.h"
#include "Pixel.h"
using namespace std;

class Pixmap{
public:
    Pixmap();
    void Read(istream &);
    void HorizReflect();
    void VertReflect();
    void Invert();
    void Write(ostream &);

private:
    string kind;
    string creator;
    int numRows, numCols;
    int maxValue;
    Matrix<Pixel *> image;
};
```

SEE HW 2

25

Thursday: TEST 1

will contain a mixture of question types, to assess different kinds of knowledge

- quick-and-dirty, factual knowledge
e.g., TRUE/FALSE, multiple choice *similar to questions on quizzes*
- conceptual understanding
e.g., short answer, explain code *similar to quizzes, possibly deeper*
- practical knowledge & programming skills
trace/analyze/modify/augment code *either similar to homework exercises or somewhat simpler*

the test will contain several "extra" points

e.g., 52 or 53 points available, but graded on a scale of 50 (hey, mistakes happen ☺)

study advice:

- review lecture notes (if not *mentioned* in notes, will not be on test)
- read text to augment conceptual understanding, see more examples
- review quizzes and homeworks
- feel free to review other sources (lots of C++/algorithms tutorials online)

26