Data structure design

Jordi Cortadella
Department of Computer Science

Data structure design

- Up to now, designing a program (or a procedure or a function) has meant designing an algorithm. The structure of the data on which the algorithm operates was part of the problem statement.

However, when we create a program, we often need to design data structures to store data and intermediate results.

- The design of appropriate data structures is often critical:
  - to be able to solve the problem
  - to provide a more efficient solution

Sports tournament

- Design a program that reads the participants in a knockout tournament and the list of results for each round. The program must write the name of the winner.

- Assumptions:
  - The number of participants is a power of two.
  - The list represents the participation order, i.e. in the first round, the first participant plays with the second, the third with the fourth, etc. In the second round, the winner of the first match plays against the winner of the second match, the winner of the third match plays against the winner of the fourth match, etc. At the end, the winner of the first semi-final will play against the winner of the second semi-final.

- The specification of the program could be as follows:

  // Pre: the input contains the number of players, the players and the results of the tournament.
  // Post: the winner has been written at the output.

Sports tournament

- Input (example):

  8 Nadal Murray Berdych Soderling Federer Ferrer Djokovic Roddick
  2 0 3 1 3 1 3 2 3 1 2 3 3 0
A convenient data structure that would enable an efficient solution would be a vector with $2^n-1$ locations ($n$ is the number of participants):

- The first $n$ locations would store the participants.
- The following $n/2$ locations would store the winners of the first round.
- The following $n/4$ locations would store the winners of the second round, etc.
- The last location would store the name of the winner.

The algorithm could run as follows:

- First, it reads the number of participants and their names. They will be stored in the locations 0…$n$-1 of the vector.
- Next, it fills up the rest of the locations. Two pointers might be used. The first pointer ($j$) points at the locations of the players of a match. The second pointer ($k$) points at the location where the winner will be stored.

```plaintext
// Inv: players[n..k-1] contains the
//      winners of the matches stored
//      in players[0..j-1]
```

Input:

```
8
Nadal Murray Berdych
Soderling Federer Ferrer
Djokovic Roddick
```

First round

```
2 0 3 1 3 1 3 2 3 1 2 3 3 0
```

Second round

Third round

Winner
int main() {
    int n;
    cin >> n; // Number of participants
    vector<string> players(2*n - 1);
    // Read the participants
    for (int i = 0; i < n; ++i) cin >> players[i];
    int j = 0;
    // Read the results and calculate the winners
    for (int k = n; k < 2*n - 1; ++k) {
        int score1, score2;
        cin >> score1 >> score2;
        if (score1 > score2) players[k] = players[j];
        else players[k] = players[j + 1];
        j = j + 2;
    }
    cout << players[2*n - 2] << endl;
}

• Exercise:

Modify the previous algorithm using only a vector with n strings, i.e.,

vector<string> players(n)

• Problem: design a function that reads a text and reports the most frequent letter in the text and its frequency (as a percentage). The letter case is ignored. That is:

struct Res {
    char letter; // letter is in ‘a’..'z'
    double freq; // 0 <= freq <= 100
};

// Pre: the input contains a text
// Returns the most frequent letter in the text
// and its frequency, as a percentage,
// ignoring the letter case
Res most_freq_letter();

• The obvious algorithm is to sequentially read the characters of the text and keep a record of how many times we have seen each letter.

• Once we have read all the text, we compute the letter with the highest frequency, and report it with the frequency divided by the text length * 100.

• To do this process efficiently, we need fast access to the number of occurrences of each letter seen so far.
**Most frequent letter**

- **Solution:** keep a vector of length \(N\), where \(N\) is the number of distinct letters. The \(i\)-th component contains the number of occurrences of the \(i\)-th letter so far.

- **Observation:** the problem specification did not mention any vectors. We introduce one to solve the problem efficiently.

```cpp
const int N = int('z') - int('a') + 1;
vector<int> occs(N, 0);
int n_letters;

// Inv: n_letters is the number of letters read so far, occs[i] is the number of occurrences of letter 'a' + i in the text read so far

Res most_freq_letter() {
const int N = int('z') - int('a') + 1;
vector<int> occs(N, 0);
int n_letters = 0;
char c;

while (cin >> c) {
  if (c >= 'A' and c <= 'Z') c = c - 'A' + 'a';
  if (c >= 'a' and c <= 'z') {
    ++n_letters;
    ++occs[int(c) - int('a')];
  }
}

int imax = 0;
// imax is the index of the highest value in occs[0..i-1]
for (int i = 1; i < N; ++i) {
  if (occs[i] > occs[imax]) imax = i;
}
Res r;
r.letter = 'a' + imax;
if (n_letters > 0) r.freq = double(occs[imax])*100/n_letters;
else r.freq = 0; // 0% if no letters in the text
return r;
}
```

**Pangrams**

- A pangram is a sentence containing all the letters in the alphabet.

An English pangram:

_The quick brown dog jumps over the lazy fox_

A Catalan pangram:

_Jove xef, porti whisky amb quinze glaçons d’hidrogen, coi!

- **Problem:** design a function that reads a sentence and says whether it is a pangram. That is,

```cpp
bool is_pangram();
```

**Pangrams**

- The algorithm is similar to previous the problem:
  - Use a vector with one position per letter as a data structure.
  - Read the sentence and keep track of the number of occurrences of each letter.
  - Then check that each letter appeared at least once.
Pangrams

```cpp
bool is_pangram() {
    const int N = int('z') - int('a') + 1;
    vector<bool> appear(N, false);
    char c;

    // Inv: appear[i] indicates whether the letter
    //      'a' + i has already appeared in the text.
    while (cin >> c) {
        if (c >= 'A' and c <= 'Z') c = c - 'A' + 'a';
        if (c >= 'a' and c <= 'z') appear[int(c) - int('a')] = true;
    }

    // Check that all letters appear
    for (int i = 0; i < N; ++i) {
        if (not appear[i]) return false;
    }
    return true;
}
```

Brackets

A number of characters go in pairs, one used to “open” a part of a text and the other to “close” it. Some examples are:

( ) (parenthesis),
[ ] (square brackets)
{ } (curly brackets)
⟨ ⟩ (angle brackets)
¿ ? (question marks - Spanish)
¡ ! (exclamation marks - Spanish)
“ ” (double quotes)
‘ ’ (single quotes)

The correct use of brackets can be defined by three rules:

1. Every opening bracket is followed in the text by a matching closing bracket of the same type – though not necessarily immediately.

2. Vice versa, every closing bracket is preceded in the text by a matching opening bracket of the same type.

3. The text between an opening bracket and its matching closing bracket must include the closing bracket of every opening bracket it contains, and the opening bracket of every closing bracket it contains (It’s ok if you need to read this more than once)

Exercise: design a function that reads a sequence of bracket characters of different kinds, and tells whether the sequence respects the bracketing rules.

```cpp
// Example test cases
(([])[]) gives: true
(([])[[{}]][]) gives: false
(([])[{]}) gives: false
([[{}]][]) gives: false
(([]){}) gives: true
```
Brackets

That is, we want:

// Pre: the input contains a nonempty sequence of bracket chars
// Returns true if the sequence is correctly bracketed, and false otherwise.
bool brackets();

Suppose we use the following functions:

bool is_open_br(char c); // Is c an opening bracket?
bool is_clos_br(char c); // Is c a closing bracket?
char match(char c);      // Returns the match of c

Brackets

// Pre: the input contains a nonempty sequence of bracket chars
// Returns true if the sequence is correctly bracketed, and false otherwise.
bool brackets()
{
    vector<char> unclosed;
    char c;
    while (cin >> c) {
        if (is_open_br(c)) unclosed.push_back(c);
        else if (unclosed.size() == 0) return false;
        else if (match(c) != unclosed[unclosed.size()-1]) return false;
        else unclosed.pop_back();
    }

    // Check that no bracket has been left open
    return unclosed.size() == 0;
}

Summary

• As programming problems become complex, it is important to first define the data structures required to represent data.

• When defining the data structures, think of the operations you will have to perform. If necessary, change the data structure.

Strategy: keep a vector of unclosed open brackets.

When we see an opening bracket in the input, we store it in unclosed (its matching closing bracket should arrive later).

When we see a closing bracket in the input, either its matching opening bracket must be the last element in unclosed (and we can remove both), or we know the sequence is incorrect.

At the end of the sequence unclosed should be empty.