The List Type

Lecture 9

Hartmut Kaiser

hkaiser@cct.lsu.edu

http://www.cct.lsu.edu/~hkaiser/spring_2015/csc1254.html
Programming Principle of the Day

• KISS (Keep it simple, stupid!)
  • Simplicity (and avoiding complexity) should always be a key goal. Simple code takes less time to write, has fewer bugs, and is easier to modify.
  • The principle is best exemplified by the story of Kelly Johnson (lead engineer at Lockheed) handing a team of design engineers a handful of tools, with the challenge that the jet aircraft they were designing must be repairable by an average mechanic in the field with only these tools.

Abstract

• Performance of our example is fine for small inputs, but quickly degrades for a larger number of students. Why?

• Inserting/deleting records in the middle of vectors requires moving big chunks of memory to preserve the random access property. Our program has a time complexity of $O(N^2)$.

• Different data structure is required
The List Type

• We rewrote code to remove reliance on indices
  • Now: change data structure allowing to efficiently delete elements from the middle of the sequence

• Common requirement, therefore: \texttt{std::list<>}
  • Vectors are optimized for fast random access
  • List are optimized for fast insert and delete at any point
    • Generally, slower than vectors
    • In heavy insertion and deletion scenarios, faster
  • Choose data structure depending on use case
The List Type

- List and vector share many common ideas
  - Can store almost any data type
  - Share almost all operations
- Converting our program to use lists is surprisingly simple
  - Main change is swapping container types
- List do not support indexing operations
  - But using iterators allows to get away without those
Using Vector Type

// version 3: iterators but no indexing
vector<student_info> extract_fails(vector<student_info>& students)
{
    vector<student_info> fail;
    auto iter = students.begin();
    while (iter != students.end()) {
        if (fail_grade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);   // watch out!
        } else
        
            ++iter;
    }
    return fail;
}
Using List Type

// version 4: using lists
list<student_info> extract_fails(list<student_info>& students)
{
    list<student_info> fail;
    auto iter = students.begin();
    while (iter != students.end()) {
        if (fail_grade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);  // watch out!
        } else
            ++iter;
    }
    return fail;
}
Iterator Invalidation

• For vectors, any insert/remove operation potentially invalidated all iterators.
  • Reallocation, data movement
  • Watch out when storing iterators (i.e. end())!

• For lists, no iterators are invalidated
  • Well, except for the one pointing to an erased element
  • But lists are not random access, that mean that iterators do not support random access operations
Sorting a List

• Standard sort algorithm is usable for random access iterators only
  • std::sort not usable for lists:
    
    ```cpp
    list<student_info> students;
    student.sort(compare);
    ```

• Instead of:
  
    ```cpp
    vector<student_info> students;
    sort(students.begin(), students.end(), compare);
    ```
Performance Data

<table>
<thead>
<tr>
<th>File size [records]</th>
<th>List [s]</th>
<th>Vector [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>700</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>7000</td>
<td>0.8</td>
<td>6.7</td>
</tr>
<tr>
<td>70000</td>
<td>8.8</td>
<td>597.1</td>
</tr>
</tbody>
</table>

- ‘Right’ data structure is not a once and for all decision
  - Performance might not even matter
  - But depending on use case, performance might have a profound influence
Doubly-Linked List

Given a pointer to a node in a doubly-linked list, we can remove the node in $O(1)$ time.

This isn’t possible in a singly-linked list, since we must have a pointer to the node in front of the one we want to remove.
Doubly-Linked List

Each node is made from a struct that looks something like this.

```c
struct DLNode {
    DataType info;
    DLNodeIter next;
    DLNodeIter back;
};
```
Doubly-Linked List
Doubly-Linked List

\[\text{start} \quad \cdots \quad \text{ptr}\]
Doubly-Linked List

\[
\text{ptr->back->next} = \text{ptr->next}; \\
\text{ptr->next->back} = \text{ptr->back}; \\
\text{delete ptr;}
\]
Doubly-Linked List

```
ptr->back->next = ptr->next;
ptr->next->back = ptr->back;
delete ptr;
```
Doubly-Linked List

\[
\text{start} \\
\begin{array}{c}
\vdots \\
\end{array} \\
\text{ptr} \\
\begin{array}{c}
\vdots \\
\end{array} \\
\text{ptr}\rightarrow\text{back}\rightarrow\text{next} = \text{ptr}\rightarrow\text{next} \\
\text{ptr}\rightarrow\text{next}\rightarrow\text{back} = \text{ptr}\rightarrow\text{back} \\
delete \text{ptr};
\]
Doubly-Linked List

```
ptr->back->next = ptr->next;
ptr->next->back = ptr->back;
delete ptr;
```
Doubly-Linked List

\[ \text{start} \]

\[ \text{ptr} \]

\[ \text{ptr->back->next = ptr->next;} \]
\[ \text{ptr->next->back = ptr->back;} \]
\[ \text{delete ptr;} \]
Doubly-Linked List

\[ \text{start} \]

\[ \text{ptr} \]

\[ \text{ptr->back->next} = \text{ptr->next}; \]
\[ \text{ptr->next->back} = \text{ptr->back}; \]
\[ \text{delete ptr}; \]
Doubly-Linked List

\[
\text{start} \quad \rightarrow \quad \ldots \quad \rightarrow \quad \text{ptr} \quad \rightarrow \quad \text{ptr}'s \text{ next} \quad = \quad \text{ptr}'s \text{ next} \\
\text{ptr}'s \text{ back} \rightarrow \text{ptr}'s \text{ back} \quad = \quad \text{ptr}'s \text{ back} \\
delete \text{ ptr};
\]