Iterators and Sequences

Sections 6.2.5 – 6.4

with an introduction to trees

© 2019 Shermer, based on Goodrich, Tamassia, Mount

Iterators and Sequences
STL Containers

- The STL container classes are
  - vector
  - deque
  - list
  - stack
  - queue
  - priority queue
  - set (and multiset)
  - map (and multimap)

- Each container type C supports iterators:
  - C::iterator – read/write iterator type
  - C::const_iterator – read-only iterator type
  - C.begin(), C.end() – return start/end iterators
STL Iterators

- Operators defined for iterators:
  - \(*p\): access current element
  - \(++p, --p\): advance to next/previous element

- Common STL vector operations using iterators:
  - \(\text{assign}(p, q)\): replace the vector’s contents with contents referenced by the iterator range \([p, q)\) (from \(p\) up to, but not including, \(q\))
  - \(\text{insert}(p, e)\): insert \(e\) prior to position \(p\)
  - \(\text{erase}(p)\): remove element at position \(p\)
  - \(\text{erase}(p, q)\): remove elements in the iterator range \([p, q)\)
Using STL Iterators

- A common use case of iterators is to iterate through all of the elements of a collection (for instance, a vector).

```cpp
int vectorSum2(vector<int> V) {
    typedef vector<int>::iterator Iterator;
    int sum = 0;
    for (Iterator p = V.begin(); p != V.end(); ++p)
        sum += *p;
    return sum;
}
```
STL Container Algorithms

- STL provides algorithms that operate on general containers. To use them, you must
  ```
  #include <algorithm>
  ```
- In the following, p and q are iterators over a base type, and e is an object of this base type.
  - `sort(p, q)`
  - `random_shuffle(p, q)`
  - `reverse(p, q)`
  - `find(p, q, e)`
  - `min_element(p, q)`
  - `max_element(p, q)`
  - `for_each(p, q, f)`
Sequence ADT

- The **Sequence ADT** is the union of the Vector and List ADTs
- Elements accessed by
  - Index, or
  - Position
- Generic methods:
  - `size()`, `empty()`
- Vector-based methods:
  - `at(i)`, `set(i, o)`, `insert(i, o)`, `erase(i)`
- List-based methods:
  - `begin()`, `end()`
  - `insertFront(o)`, `insertBack(o)`
  - `eraseFront()`, `eraseBack()`
  - `insert(p, o)`, `erase(p)`
- Bridge methods:
  - `atIndex(i)`, `indexOf(p)`
Applications of Sequences

- The Sequence ADT is a basic, general-purpose data structure for storing an ordered collection of elements

- Direct applications:
  - Generic replacement for stack, queue, vector, or list
  - small database (e.g., address book)

- Indirect applications:
  - Building block of more complex data structures
Linked List Implementation

- A doubly linked list provides a reasonable implementation of the Sequence ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes
- Position-based methods run in constant time
- Index-based methods require searching from header or trailer while keeping track of indices; hence, run in linear time
Array-based Implementation

- We use a circular array storing positions
- A position object stores:
  - Element
  - Index
- Indices $f$ and $l$ keep track of first and last positions

© 2010 Goodrich, Tamassia
### Comparing Sequence Implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, empty</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>atIndex, indexOf, at</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>begin, end</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>set(p,e)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>set(i,e)</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>insert(i,e), erase(i)</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>insertBack, eraseBack</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>insertFront, eraseFront</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>insert(p,e), erase(p)</td>
<td>n</td>
<td>1</td>
</tr>
</tbody>
</table>
Trees

- A tree is a mathematical object that models hierarchical structures and acyclic relations.
- A tree consists of a set of vertices (a.k.a. nodes) and a set of edges (a.k.a. arcs).
- The vertices can be anything. We typically draw a vertex as a dot, a circle, or a box.
- The edges are equivalent to pairs of vertices. We typically draw an edge as a line segment between two vertices.
- The tree must be connected and have no cycles.
Some Drawings of Trees

Make Money Fast!

1. Motivations
   1.1 Greed
   1.2 Avidity

2. Methods
   2.1 Stock Fraud
   2.2 Ponzi Scheme
   2.3 Bank Robbery

References
Some Not-Trees

- Left tree has a cycle
- Right tree has a three-way edge
- The graph is not connected

1. Motivations
   - 1.1 Greed
   - 1.2 Avidity

2. Methods
   - 2.1 Stock Fraud
   - 2.2 Ponzi Scheme
   - 2.3 Bank Robbery

References
Recursive Definition of Trees

- An empty vertex set is a tree (the **empty tree**).
- A single vertex with no edges is a tree.
- Any object created by starting with a tree $T$, selecting one vertex $u$ of $T$, and adding a new vertex $v$ to $T$ along with the edge $uv$ is a tree.
Tree Terminology

- Two vertices that share an edge are said to be adjacent (and are called neighbors). Here, u and v are neighbors.
- The degree of a vertex is the number of edges that include it. Here, u has degree 4 and w has degree 1.
- A vertex with degree 1 is called a leaf. Here, w is a leaf, and so are the four unlabeled vertices.
- The vertices with degree higher than 1 are called internal nodes. Here, u and v are internal.
- The distance between two nodes is the number of edges on the path in the tree between the nodes. Here, v and w have distance 2.
Rooted Trees

- In computing, when we say tree we often mean what mathematicians call a rooted tree.
- A rooted tree is a tree with a special vertex called the root.
- Typically we draw a rooted tree with the root at the top, and the other vertices at a height denoting their distance from the root.
We use family terms for the relations amongst nodes in a rooted tree.

The **parent** of a node is the neighbor that is closer to the root (i.e. above). The root has no parent. Other nodes have one parent. Here the parent of $v$ is $p$.

A **child** of a node is any neighbor that is farther from the root (i.e. below). Here the children of $v$ are $y$ and $z$. 
Rooted Tree Terminology

- The **grandparent** of a node is that node’s parent’s parent. Here the grandparent of \(v\) is \(q\).
- A **grandchild** of a node is any of the node’s children’s children. Here \(v\) and \(w\) are the grandchildren of \(q\).
- A **sibling** of a node is a node that has the same parent. Here \(w\) is a sibling of \(v\); we also say that \(v\) and \(w\) are siblings.
Rooted Tree Terminology

- The **ancestors** of a node $v$ are all nodes on the path to the root. Here the ancestors of $v$ are $p$, $q$, and the root.

- The **descendants** of a node $v$ are all nodes whose path to the root includes $v$. Here the descendants of $q$ are $p$, $v$, $w$, $y$, and $z$.

- Sometimes $v$ is included in the ancestors and descendants; be careful which definition you work with.
Rooted Tree Terminology

- The **subtree** of a node \( v \) is a tree that consists of \( v \), the descendants of \( v \), and any tree edges inbetween them. Here the subtree of \( p \) is circled.

- The **depth** of a node is the number of edges required to go from that node to the root. Here the depth of \( p \) is 2, and the depth of \( w \) is 3.

- The **height** of the tree is the maximum depth of any node (here 4).