3.6 SIMULATING POINTERS

In most applications, we can implement the desired linked and indirect addressing representations using dynamic allocation and C++ pointers. At times, however, it is more convenient and efficient to use an array of nodes and simulate C++ pointers by integers that are indexes into this array.

Suppose we use an array `node`, each element of which has the two fields `data` and `link`. The nodes are `node[0]`, `node[1]`, ···, `node[NumberOfNodes-1]`. We shall refer to `node[i]` as node `i`. Now if a chain `c` consists of nodes 10, 5, and 24 (in that order), we shall have `c` = 10 (pointer to first node on the chain `c` is of type `int`); `node[10].link` = 5 (pointer to second node on chain); `node[5].link` = 24 (pointer to next node); and `node[24].link` = −1 (indicating that node 24 is the last node on the chain). When drawing the chain, the links are drawn as arrows in the same way as when C++ pointers are used (Figure 3.12).

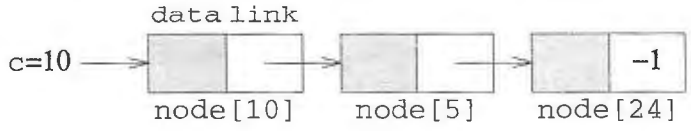

Figure 3.12 Chain using simulated pointers

To complete the simulation of pointers, we need to design procedures to allocate and deallocate a node. Nodes that are presently not in use will be kept in a **storage pool**. Initially, this pool contains all the nodes `node[0:NumberOfNodes-1]`. `Allocate` takes nodes out of this pool, one at a time. `Deallocate` puts nodes into this pool one at a time. Hence `Allocate` and `Deallocate`, respectively, perform deletes and inserts on the storage pool and are equivalent to the `delete` and `new` functions of C++. These functions can be performed efficiently if the storage pool is set up as a chain of nodes (as in Figure 3.13). This chain is called the **available space list**. It contains all nodes that are currently free. `first` is a variable of type `int`

that points to the first node on this chain. Additions to and deletions from this chain are made at the front.

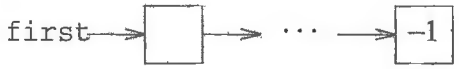

Figure 3.13 Available space list

To implement a simulated pointer system, we define the classes `SimNode` and `SimSpace` as in Program 3.27.

```
template <class T>
class SimNode {
   friend SimSpace<T>;
   private:
      T data;
      int link;
};

template <class T>
class SimSpace {
   public:
      SimSpace(int MaxSpaceSize = 100);
      ~SimSpace() {delete [] node;}
      int Allocate(); // allocate a node
      void Deallocate(int& i); // deallocate node i
   private:
      int NumberOfNodes, first;
      SimNode<T> *node; // array of nodes
};
```

Program 3.27 Class definition for simulated pointers

3.6.1 `SimSpace` Operations

Since all nodes are initially free, the available space list contains `NumberOfNodes` nodes at the time it is created. Program 3.28 initializes the available space list. Programs 3.29 and 3.30 perform the `Allocate` and `Deallocate` operations.

```
template<class T>
SimSpace<T>::SimSpace(int MaxSpaceSize)
{// Constructor.
   NumberOfNodes = MaxSpaceSize;
   node = new SimNode<T> [NumberOfNodes];
   // initialize available space list
   // create a chain of nodes
   for (int i = 0; i < NumberOfNodes-1; i++)
      node[i].link = i+1;
   // last node of chain
   node[NumberOfNodes-1].link = -1;
   // first node of chain
   first = 0;
}
```

Program 3.28 Initialize available space list

```
template<class T>
int SimSpace<T>::Allocate()
{// Allocate a free node.
   if (first == -1) throw NoMem();
   int i = first;          // allocate first node
   first = node[i].link;   // first points to next
                           // free node
   return i;
}
```

Program 3.29 Allocate a node using simulated pointers

```
template<class T>
void SimSpace<T>::Deallocate(int& i)
{// Free node i.
   // make i first node on avail list
   node[i].link = first;
   first = i;
   i = -1;
}
```

Program 3.30 Deallocate a node with simulated pointers

We readily see that the three functions have time complexity Θ(NumberOfNodes), $\Theta(1)$, and $\Theta(1)$, respectively. We can reduce the run time of the constructor (Program 3.28) by maintaining two available space lists. One contains all free nodes that haven't been used yet. The second contains all free nodes that have been used at least once. Whenever a node is deallocated, it is put onto the second list. When a new node is needed, we provide it from the second list in case this list is not empty. Otherwise, we attempt to provide it from the first list. Let `first1` and `first2`, respectively, point to the front of the first and second space lists. Because of the way nodes are allocated, the nodes on the first list are `node[i]`, `first1` $\leq$ `i` < `NumberOfNodes`. The code to deallocate a node differs from Program 3.30 only in that all occurences of the variable `first` are replaced by `first2`. The new constructor and allocation codes are given in Programs 3.31 and 3.32. For these codes to work, we make the integer variables `first1` and `first2` private members of `SimSpace`.

```
template<class T>
SimSpace<T>::SimSpace(int MaxSpaceSize)
{// Dual available list constructor.
   NumberOfNodes = MaxSpaceSize;
   node = new SimNode<T> [NumberOfNodes];
   // initialize available space lists
   first1 = 0;
   first2 = -1;
}
```

Program 3.31 Initialization of dual available space list

```
template<class T>
int SimSpace<T>::Allocate()
{// Allocate a free node.
   if (first2 == -1) {// 2nd list empty
      if (first1 == NumberOfNodes) throw NoMem();
      return first1++;}
   // allocate first node of chain
   int i = first2;
   first2 = node[i].link;
   return i;
}
```

Program 3.32 Dual available space list version of `Allocate`

We expect the dual available space list of Programs 3.31 and 3.32 to provide better performance than the single space list version in most applications. We make the following observations:

- Program 3.32 takes the same time as does Program 3.29 except when the node is to be provided from the first list. This exception occurs at most `NumberOfNodes` times. The extra time spent on these cases is balanced by the savings during initialization. In fact, we will frequently need fewer than `NumberOfNodes` nodes (especially during debugging runs and in software designed to handle problems with widely varying instance characteristics), and the dual scheme will be faster.
- The reduction in the initialization time is very desirable in an interactive environment. The startup time for the program is significantly reduced.
- When the single list scheme is in use, chains can be built without explicitly setting the link fields in any but the last node because the appropriate link values are already present in the nodes (see Figure 3.13). This advantage can also be incorporated into the dual available space list scheme by writing a function `Get(n)` that provides a chain with `n` nodes on it. This function will explicitly set links only when nodes are taken from the first list.
- Chains can be disposed more efficiently using either of these schemes than when C++ pointers are used. For instance, if we know the front `f` and end `e` of a chain, all nodes on it are freed by the following statements:

```
node[e].link = first; first = f;
```

- If `c` is a circular list, then all nodes on it are disposed in $\Theta(1)$ time using Program 3.33. Figure 3.14 shows the link changes that take place.

```
template<class T>
void SimSpace<T>::DeallocateCircular(int& c)
{// Deallocate the circular list c.
   if (c != -1) {
      int next = node[c].link;
      node[c].link = first;
      first = next;
      c = -1;
      }
}
```

Program 3.33 Deallocate a circular list

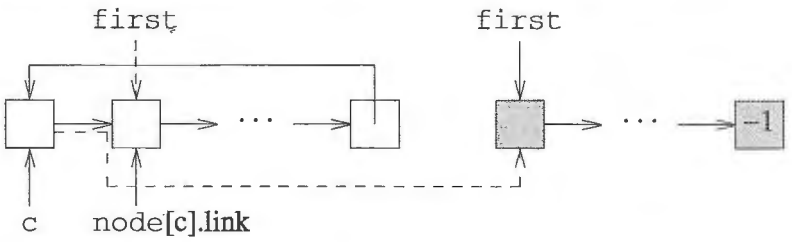

Figure 3.14 Deallocating a circular list

3.6.2 Chains Using Simulated Pointers

We may define a class for chains using the simulated space `S` (see Program 3.34). `S` is declared as a `static` member so that all simulated chains of the same type `T` share the same simulated space. Programs 3.35 to 3.38 give the code for the public methods other than `Search` and `Output`. The code assumes that `SimChain` has been declared a friend of both `SimNode` and `SimSpace`. Notice the similarity between these codes and the codes for the corresponding members of `Chain`. Program 3.39 gives a sample program that uses a simulated chain. In this program `simul.h` and `schain.h` are files that contain the codes for `SimSpace` and `SimChain`, respectively.

```
template<class T>
class SimChain {
   public:
      SimChain() {first = -1;}
      ~SimChain() {Destroy();}
      void Destroy(); // make list null
      int Length() const;
      bool Find(int k, T& x) const;
      int Search(const T& x) const;
      SimChain<T>& Delete(int k, T& x);
      SimChain<T>& Insert(int k, const T& x);
      void Output(ostream& out) const;
   private:
      int first;  // index of first node
      static SimSpace<T> S;
};
```

Program 3.34 Class definition for simulated chains

```
template<class T>
void SimChain<T>::Destroy()
{// Deallocate chain nodes.
   int next;
   while (first != -1) {
      next = S.node[first].link;
      S.Deallocate(first);
      first = next;}
}
template<class T>
int SimChain<T>::Length() const
{// Return the number of elements in the chain.
   int current = first,  // chain node cursor
       len = 0;          // element counter
   while (current != -1) {
      current = S.node[current].link;
      len++;}
   return len;
}
```

Program 3.35 Destructor and length using simulated pointers

```
template<class T>
bool SimChain<T>::Find(int k, T& x) const
{// Set x to the k'th element of the chain.
 // Return false if no k'th; return true otherwise.
   if (k < 1) return false;
   int current = first,  // cursor for chain nodes
       index = 1;        // index of current node
   // move current to k'th node
   while (index < k && current != -1) {
      current = S.node[current].link;
      index++;}
   // verify that we got to the k'th node
   if (current != -1) {x = S.node[current].data;
                       return true;}
   return false; // no k'th element
}
```

Program 3.36 Find using simulated pointers

```
template<class T>
SimChain<T>& SimChain<T>::Delete(int k, T& x)
{// Set x to the k'th element and delete it.
 // Throw OutOfBounds exception if no k'th element.

   if (k < 1 || first == -1)
      throw OutOfBounds(); // no k'th

   // p will eventually point to k'th node
   int p = first;

   // move p to k'th & remove from chain
   if (k == 1) // p already at k'th
      first = S.node[first].link; // remove from chain
   else { // use q to get to k-1'st
      int q = first;
      for (int index = 1; index < k - 1 && q != -1;
                          index++)
         q = S.node[q].link;

      // verify presence of k'th element
      if (q == -1 || S.node[q].link == -1)
         throw OutOfBounds(); // no k'th

      // make p point to k'th element
      p = S.node[q].link;

      // remove k'th element from chain
      S.node[q].link = S.node[p].link;
      }

   // save k'th element and free node p
   x = S.node[p].data;
   S.Deallocate(p);
   return *this;
}
```

Program 3.37 Delete using simulated pointers

```
template<class T>
SimChain<T>& SimChain<T>::Insert(int k, const T& x)
{// Insert x after the k'th element.
 // Throw OutOfBounds exception if no k'th element.
 // Pass NoMem exception if inadequate space.

   if (k < 0) throw OutOfBounds();

   // define a cursor p that will
   // eventually point to k'th node
   int p = first;

   // move p to k'th node
   for (int index = 1; index < k && p != -1;
                       index++)
      p = S.node[p].link;

   // verify presence of k'th element
   if (k > 0 && p == -1)
      throw OutOfBounds();

   // prepare a new node for insertion
   int y = S.Allocate();
   S.node[y].data = x;

   // insert the new node into the chain
   // first check if the new node is to be the
   // first one in the chain
   if (k) {// insert after p
           S.node[y].link = S.node[p].link;
           S.node[p].link = y;}
   else {// insert as first element
         S.node[y].link = first;
         first = y;}

   return *this;
}
```

Program 3.38 Insert using simulated pointers

```
#include <iostream.h>
#include "schain.h"

SimSpace<int> SimChain<int>::S;

void main(void)
{
   int x;
   SimChain<int> c;
   cout << "Chain length is " << c.Length() << endl;
   c.Insert(0,2).Insert(1,6);
   cout << "Chain length is " << c.Length() << endl;
   c.Find(1,x);
   cout <<"First element is " << x << endl;
   c.Delete(1,x);
   cout << "Deleted " << x << endl;
   cout << "New length is " << c.Length() << endl;
   cout << "Position of 2 is " << c.Search(2) << endl;
   cout << "Position of 6 is " << c.Search(6) << endl;
   c.Insert(0,9).Insert(1,8).Insert(2,7);
   cout << "Current chain is " << c << endl;
   cout << "Its length is " << c.Length() << endl;
}
```

Program 3.39 Using a simulated chain

EXERCISES

67. Develop an iterator class `SimIterator` for the class `SimChain`. See Program 3.18 for the definition of an iterator class defined for `Chains` (Program 3.8). `SimIterator` should contain the same functions as does `ChainIterator`. Write and test your code.

68. (a) Modify the class `SimSpace` so that `Allocate` returns a pointer to `node[i]` rather than the index `i`. Similarly, `Deallocate` takes as input a pointer to the node that is to be deallocated.

    (b) Rewrite the code for `SimChain` using the `SimSpace` codes of (a). Notice the similarity between your new code and that for the class `Chain`.

69. (a) Modify the definition of the class `SimNode` so that it contains a static member `S` of type `SimSpace<T>`. All nodes of the type `SimSpace<T>` can now share the same simulated space. Overload

the functions `new` and `delete` so as to get/return `SimNodes` from/to the simulated space `S`.

(b) Suppose that `SimSpace` is implemented as in Exercise 68 and that `SimNode` is as in (a). Change the code for the class `Chain` (Program 3.8) so that it works properly using `SimNodes` in place of `ChainNodes`. Test your code and perform run-time measurements to determine which version of `Chain` is faster.

70. Assume that a chain is represented using simulated pointers. The nodes are of type `SimNode`.

(a) Write a procedure that uses the insertion sort method to sort the chain into nondecreasing order of the field `data`.

(b) What is the time complexity of your code? In case it isn't $O(n^2)$, where n is the chain length, rewrite the code to have this complexity.

(c) Test the correctness of your code.

71. Do Exercise 70 using selection sort.

72. Do Exercise 70 using bubble sort.

73. Do Exercise 70 using rank sort.

74. Calls to the functions `new` and `delete` are usually quite expensive and we can often improve the run time of our code by replacing the use of `delete` by a call to our own deallocating function which saves the deleted node on a chain of free nodes. Calls to `new` are replaced by calls to our own node allocator which invokes `new` only when the free node chain is empty. Modify the class `Chain` (Program 3.8) to operate in this way. Write functions to allocate and deallocate a node as described, and to initialize the chain of free nodes. Compare the run times of the two versions of the class `Chain`. Comment on the merits/demerits of the new implementation.

75. Consider the operation XOR (exclusive OR, also written as $\oplus$) defined as below (for i and j binary):

$$i \oplus j = \begin{cases} 0 & \text{if } i \text{ and } j \text{ are identical} \\ 1 & \text{otherwise} \end{cases}$$

The XOR of two binary strings i and j is obtained by take the XOR of corresponding bits of i and j. For example, if $i = 10110$ and $j = 01100$, then i XOR $j = i \oplus j = 11010$. Note that

$$a \oplus (a \oplus b) = (a \oplus a) \oplus b = b$$

and

$$(a \oplus b) \oplus b = a \oplus (b \oplus b) = a$$

This observation gives us a space-saving device for storing the right and left links of a doubly linked list. We assume that the available nodes are in an array `node` and that the node indexes are 1, 2, $\cdots$. So `node[0]` is not used. A `NULL` link can now be represented as a zero rather than as –1. Each node has two fields: `data` and `link`. If `l` is to the left of node `x` and `r` is to its right, then `link(x)` = `l` $\oplus$ `r`. For the left-most node `l` = 0, and for the right-most node `r` = 0. Let (`l`, `r`) be a doubly linked list represented in this way; `l` points to the left-most node and `r` points to the right-most node in the list.

(a) Write a function to traverse the doubly linked list `(l,r)` from left to right, listing out the contents of the `data` field of each node.

(b) Write a function to traverse the list from right to left, listing out the contents of the `data` field of each node.

(c) Test the correctness of your codes.

3.7 A COMPARISON

The table of Figure 3.15 compares the asymptotic complexity of performing various functions on a linear list, using each of the four representation methods discussed in this chapter. In this table s and n, respectively, denote `sizeof(T)` and the list length. Since the asymptotic complexity of the operations is the same when C++ pointers and simulated pointers are used, the table contains a single row for both.

Representation	Function		
	Find kth	Delete kth	Insert after kth
Formula (3.1)	$\Theta(1)$	$O((n-k)s)$	$O((n-k)s)$
Linked List (C++ & Simulated)	$O(k)$	$O(k)$	$O(k+s)$
Indirect	$\Theta(1)$	$O(n-k)$	$O(n-k)$

Figure 3.15 Comparison of four representation methods

Data Structures, Algorithms, and Applications in C++

Sartaj Sahni

University of Florida

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogota Caracas Lisbon London Madrid Mexico City Milan New Delhi Seoul
Singapore Sydney Taipei Toronto