Chapter 3
Data Representation and Linear Structures

We begin the study of data structure with data representation, i.e., different ways to store data in computer memory. In this chapter, we will study how to represent data with *linear structure.*

A *data type* is a set of values. For example,

1. *Boolean* = {*true, false*}.
2. *integer* = $\{0, \pm 1, \pm 2, \cdots\}$.
3. *Character* $=\{a, b, \cdots, z, A, B, \cdots, Z\}$.
4. *String* $=\{a, b, \cdots, aa, ab, \cdots\}$.

A data type can be either *primitive,* or *composite,* for which every value is composed of values of other data types. For example, *Character* is primitive, but *String* is composite.

Data structures

The instances of a data type are usually related. For example, for the *Character* type, a is the first element, b is the second, c is the next one, etc.. In the natural number 675, 6 is the most significant digit, 7 is the next, and 5 is the least significant digit.

For each data type, there usually exists a collection of *functions* that transform one instance into another, or simply create a new instance based on some existing ones. For example, the *addition* function.

A *data structure* consists of a data type, the interrelationship among its values, together with a set of functions defined on its values.

Data types in C++

C++ comes with some of the most frequently used data objects and their frequently used functions such as *integer*(`int`,) *Real* (`float`,) *Character*(`char`,) etc. All other data types we may use can be composed based on the standard types, with the help of C++'s enumeration and grouping facilities. For example, the *String* type can be represented by using a character set array `s` as follows:

```
char s[MaxSize];
```

The linear structures

In general, a *linear list* is a data object whose values are of the form $(e_1, e_2, \cdots, e_n)$, where e_i terms are the elements of the list, and n, a finite number, is its length.

When $n = 0$, the list is *empty.* Otherwise, e_1 is the *first* element, and e_n is the *last* one. For any other i, e_i *precedes* e_{i+1}. This is called the *precedence* relation for the linear list type.

The *linear list* type is used a lot in real life, e.g., an alphabetized list of students, a list of test scores, etc..

End of the beginning

We immediately have the following list of functions, thinking about these applications: *create* a list; determine if a list is *empty;* find out the *length* of a list; *find out the* k^{th} *element* in a list; *Search* for a given element; *delete* an element in a list; and *insert* another element in a list, etc..

Any data type can be specified as an *Abstract Data Type,* which is independent of any representation. All representation must satisfy this specification.

The *List* ADT

Instances: ordered finite collection of zero or more elements.

Operations: `Create()` constructs an empty list.

`Destroy()` erases the list.

`Is_empty()` returns `true` if the list is empty, otherwise, returns `false.`

`Length()` returns the *size* of the list.

`Find(k, x)` returns the kth element of the list, and put it in `x`; returns `false`, if the size is smaller than k.

`void Search(x)` returns the position of `x`.

`Delete(k, x)` deletes the kth element of the list, and put it in `x`; returns the modified list.

`Insert(k, x)` adds `x` just after k^{th} element.

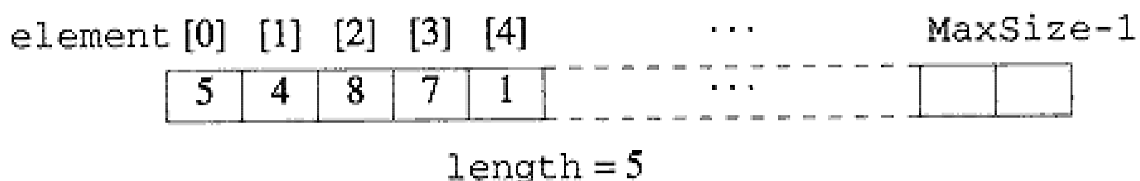
`Output(out)` puts the list into the output stream `out`.

Formula-based representation

A *formula-based* representation uses an array to represent the instances of an object. Each position of the array, called a *cell* or a *node,* holds one element that makes up an instance of that object. Individual elements of an instance are located in the array, based on a mathematical formula, e.g., a simple and often used formula is

$$Location(i) \;=\; i - 1,$$

which says the ith element of the list is in position $i - 1$. We also need two more variables, *length* and *MaxSize,* to completely characterize the list type.



The `List` class

```
template<class T>
class LinearList {
 public:
  LinearList(int MaxListSize = 10);
  ~LinearList() {delete [] element;}
  bool IsEmpty() const {return length == 0;}
  int Length() const {return length;}
  bool Find(int k, T& x) const;
  int Search(const T& x) const;
  LinearList<T>& Delete(int k, T& x);
  LinearList<T>& Insert(int k, const T& x);
  void Output(ostream& out) const;
 private:
  int length;
  int MaxSize;
  T *element; // dynamic 1D array
};
```

Implement operations

The `Create()` operation is implemented as a class constructor.

```
LinearList<T>::LinearList(int MaxListSize){
   MaxSize = MaxListSize;
   element = new T[MaxSize];
   length = 0;
}
```

The following line creates a linear list `y` of integers with its length being 100.

```
LinearList<int> y (100);
```

The `Destroy()` operation is similarly implemented as a class destructor.

The find and search operations are implemented as follows:

```
bool LinearList<T>::Find(int k, T& x) const
{// Set x to the k'th element of the list.
 // Return false if no k'th; true otherwise.
   if (k < 1 || k > length) return false;
   x = element[k - 1];
   return true;
}

int LinearList<T>::Search(const T& x) const
{// Locate x.  Return position of x if found.
 // Return 0 if x not in list.
   for (int i = 0; i < length; i++)
      if (element[i] == x) return ++i;
   return 0;
 }
```

To delete an element from a list, we have to make sure that it is in the list; if that is the case, we will copy it over. Otherwise, we will throw back an exception.

```
LinearList<T>&LinearList<T>::Delete(int k,T& x){
 // Set x to the k'th element and delete it.
 // Throw an exception if no k'th element.
 if (Find(k, x)) {//move elements down
   for (int i = k; i < length; i++)
     element[i-1] = element[i];
     length--;
     return *this;
   }
 else throw OutOfBounds();
 return *this;  // visual needs this
}
```

When there is no kth element, it takes $\Theta(1)$; Otherwise, it takes $\Theta(length - k)$.

Similarly, when we insert one more element into a list after the kth position, we have to check first if there is enough space; if there is, we have to move up all the element from the kth position on, then copy the new element into the kth position.

Below are the codes for output.

```
void LinearList<T>::
 Output(ostream& out) const{
  for (int i = 0; i < length; i++)
    out << element[i] << "  ";
}

ostream& operator<<(ostream& out,
            const LinearList<T>& x)
   {x.Output(out); return out;}
```

Work with lists

```
#include <iostream.h>
#include "llist.h"
#include "xcept.h"

void main(){
  LinearList<int> L(5);
  cout << "Length = " << L.Length() << endl;
  cout << "IsEmpty = " << L.IsEmpty() << endl;
  L.Insert(0,2).Insert(1,6);
  cout << "List is " << L << endl;
  cout << "IsEmpty = " << L.IsEmpty() << endl;
  int z;
  L.Find(1,z);
  cout << "First element is " << z << endl;
  cout << "Length = " << L.Length() << endl;
  L.Delete(1,z);
  cout << "Deleted element is " << z << endl;
  cout << "List is " << L << endl;
}
```

Performance evaluation

The formula-based representation leads to simple C++ functions, and low time complexities. On the negative side, it leads to inefficient use of space. For example, if we have to maintain three lists, for which we know that they will not contain more than 5,000 elements at any time.

However, since it is quite possible that any of those lists can contain 5,000 elements at one time, we have to assign 5,000 nodes for all the three lists, 15,000 nodes in total.

Homework

3.1. A shortcoming of the `LinearList` class is the need to predict the maximum possible size of the list. One possible improvement is to start the `MaxSize` with 1, then double the size whenever a need arises, create a new list with the doubled size, copy over the values, finally delete the original list. A similar action is performed when the size is reduced to a quarter of the current value of `MaxSize.` (i) Obtain a new implementation of the list by applying this idea. (ii) Consider any sequence of n operations starting with an empty list, assuming that the total step count of the original implementation takes $f(n$, show that under the new implementation, the state count will be $cf(n,)$ for some constant c.

3.2. Extend the `LinearList` class by adding a function *reverse,* which, given a list, will send back its reverse.

3.3. Extend the `LinearList` class by adding a private member *Current*, which indicates the current position in the list. The functions to be added include *Reset*, setting `Current` to 1; *Current*, returning the current element; *End*, returning *true* iff `Current` is at the end of the list, and *Front*, *Next* and *Previous*.

3.4. Assume that we use $location(i) = i$ to represent a linear list, modify the declaration and implementation of the linear list class.

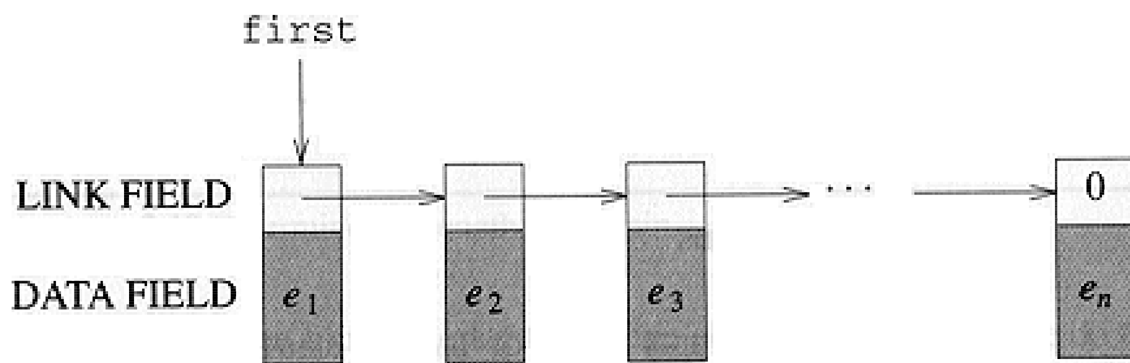
For all the above homework, you need to submit source code and sample output.

Linked lists

One way to overcome the inefficiency problem of the previous approach is to assign space on a need-only base. No space will be assigned if there is no need; and whenever there is a need, another piece of space will be assigned to an element. Since, we can't guarantee all the pieces of spaces assigned at different times will be physically adjacent, besides the space assigned for the elements, we also have to keep track of the location information of previously assigned pieces.

Hence, in a *linked* representation, each element of an instance is presented in a cell or node, which also contains a *pointer* that keeps information about the location of another node.

More specifically, let $L = (e_1, \cdots, e_n)$ be a linear list. In a linked representation of L, every e_i is represented in a separate node. A node also contains a link field that is used to locate the next element in the list. Thus, for every $1 \leq i < n, e_i$ is linked to e_{i+1}. A variable `first` locates e_1, and as e_n has no node to link to, its link field contains `NULL`. Such a structure is called a *singly linked list*, or a *chain.*



The *linked list* class

```
class ChainNode{
 friend Chain<T>l
 private:
  T: data;
  ChainNode<T>* link;
}

class Chain {
 friend ChainIterator<T>;
 public:
  Chain() {first = 0;}
  ~Chain();
  bool IsEmpty() const {return first == 0;}
  int Length() const;
  bool Find(int k, T& x) const;
  int Search(const T& x) const;
  Chain<T>& Delete(int k, T& x);
  Chain<T>& Insert(int k, const T& x);
  void Output(ostream& out) const;
 private:
  ChainNode<T> *first;
};
```

Operations

We can create an empty list of integers in the following way:

```
Chain<int> L;
```

The following code destroys a linked list.

```
Chain<T>::~Chain(){
  ChainNode<T> *next;  // next node
  while (first) {
    next = first->link;
    delete first;
    first = next;
  }
}
```

Its complexity is $\Theta(n)$, where n is the current length.

Implement `Search(x)`

The following codes implement the search functions:

```
int Chain<T>::Search(const T& x) const{
 // Locate x.  Return position of x if found.
 // Return 0 if x not in the chain.
 ChainNode<T> *current = first;
 int index = 1;  // index of current
 while (current && current->data != x) {
   current = current->link;
   index++;
 }
 if (current) return index;
 return 0;
}
```

It is to see that its complexity is $O(n)$, where n is the current length.

Implement the Delete(k, x)

```
Chain<T>& Chain<T>::Delete(int k, T& x){
  if (k < 1 || !first)
    throw OutOfBounds(); // no k'th

  ChainNode<T> *p = first;

  if (k == 1) // p already at k'th
    first = first->link; // remove
  else { // use q to get to k-1'st
        ChainNode<T> *q = first;
        for (int index = 1;
             index < k - 1 && q; index++)
          q = q->link;
        if (!q || !q->link)
          throw OutOfBounds();
        p = q->link; // k'th
        q->link = p->link;}

  x = p->data;
  delete p;
  return *this;
}
```

A chain iterator

We often need to go through the whole list, visiting all the nodes one by one. The following iterator will be handy.

```
class ChainIterator {
  public:
   T* Initialize(const Chain<T>& c){
       location = c.first;
       if (location) return &location->data;
         return 0;
      }

   T* Next(){
       if (!location) return 0;
        location = location->link;
        if (location) return &location->data;
        return 0;
      }

 private:
  ChainNode<T> *location;
};
```

Then, when printing out elements, instead of using the following $\Theta(n^2)$ segment:

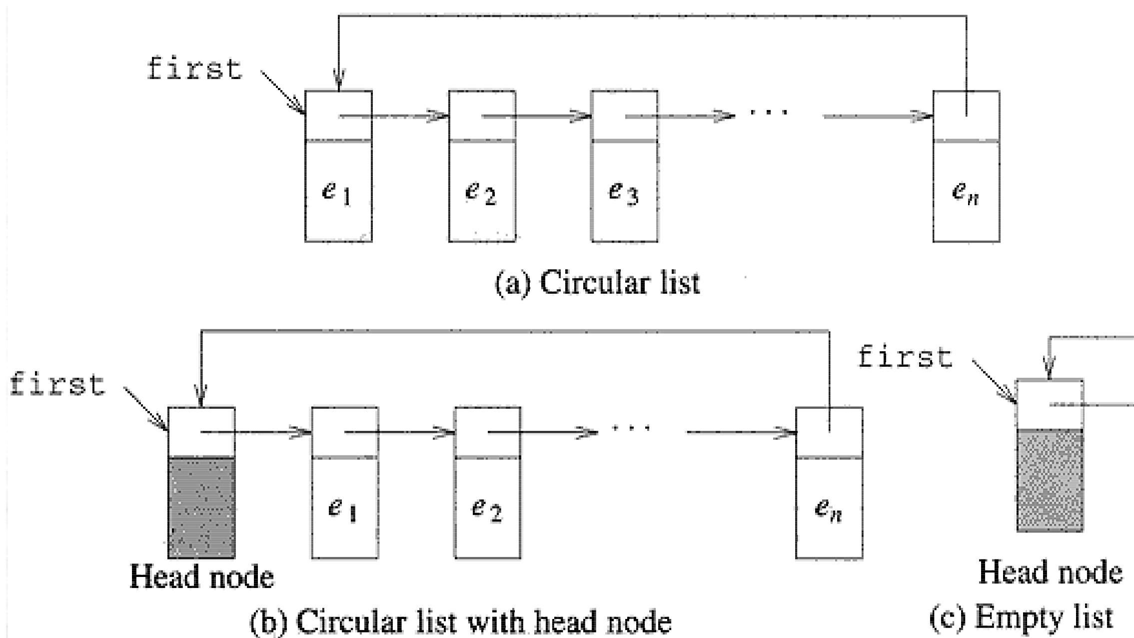
```
int len=X.length();
for(int i=1; i<=len; i++){
  X.Find(i, x);
  cout << x << ' ';
}
```

we can use the following $\Theta(n)$ segment:

```
int *x;
ChainIterator<int> c;
x=c.Initialize(X);
while(x){
  cout << *x << ' ';
  x=c.Next();
}
cout << endl;
```

Circular list

Some application might be simpler, or run faster, by representing a list as a *circular list*, and/or adding a *Head node*, at the front.



An example

Below gives code for search in a circular list.

```
template<class T>
int CircularList<T>::Search(const T& x) const{
  ChainNode<T> *current = first->link;
  int index = 1; // index of current
  first->data = x; // put x in head node

  while (current->data != x) {
    current = current->link);
    index++;
  }

  return ((current == first) ? 0 : index);
}
```

It does not run faster, but it is simpler.

Comparisons with arrays

First of all, we consider the *space* factor. As array must occupy adjacent locations in memory, the request for an array of 10,000 elements might fail, even there are 40,000 integer-sized locations available. On the other hand, the same request for that many linked nodes is more likely to be successful.

Assume that a pointer needs 4 bytes, an integer 2 bytes and each element d bytes. As we double the size of an array when needed, we assume the average size of any array to store N elements is $\frac{3}{2}N$. Thus, the array implementation needs $\frac{3}{2}Nd+8$ bytes. In contrast, a linked list version needs $N(d+4)+4$ bytes. This will lead to the conclusion that when $d \geq 8$, an array will use more space; when $d < 8$, an linked list uses more space;

The different implementation also has some impact on the running time. For example, in defining the operation of `InsertBefore`, if an array is used, some elements must be moved first, thus it needs $O(n)$ time; while it only takes $O(1)$ time to do it if a linked list is used. On the other hand, the destructor uses less time in an array than a linked list.

Finally, comprehensibility is also an issue. Array implementation is much easier to understand than the linked list implementation.

These consideration leads to the question of "what do you mean by 'best'?" It depends on what criterion must be regarded as the most important one. We are often faced with tradeoffs.

Homework

3.5. Extend the `Chain` class to include functions to convert a `LinearList` to `Chain` and vice versa.

3.7. Write a function `Alternate` to create a new chain `C`, given two chains `A` and `B`, beginning with the first element of `A`, followed by the first one in `B`, etc..

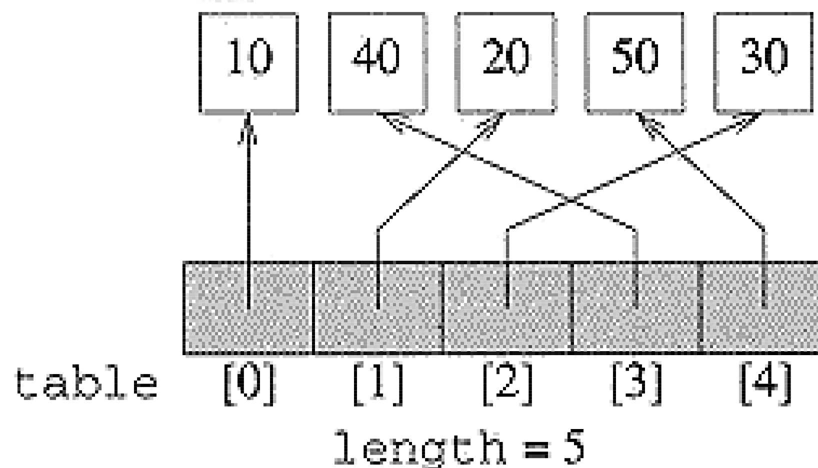
3.8. Write a function `Split` that does the opposite to the above, but `B` is to collect all the odd positioned elements of `A`, while `C` collects the even positioned ones.

For all the above homework, you need to submit source code and sample output.

Indirect addressing

This approach combines the formula-based approach and that of the linked representation. As a result, we can not only get access to elements in $\Theta(1)$ times, but also have the storage flexibility, elements will not be physically moved during insertion and/or deletion.

In indirect addressing, we use a table of pointers to get access to a list of elements, as shown in the following figure.



The indirect addressing class

The following declares the indirect addressing structure, when the elements are stored dynamically.

```
class IndirectList {
 public:
  IndirectList(int MaxListSize = 10);
  ~IndirectList();
  bool IsEmpty() const {return length == 0;}
  int Length() const {return length;}
  bool Find(int k, T& x) const;
  int Search(const T& x) const;
  IndirectList<T>& Delete(int k, T& x);
  IndirectList<T>& Insert(int k, const T& x);
  void Output(ostream& out) const;
 private:
  T **table; // 1D array of T pointers
  int length, MaxSize;
};
```

To create an indirect addressing list of no more than 20 integers, we can use the following code

```
IndirectList<int> x(20);
```

Below shows the constructor and/or destructor.

```
IndirectList<T>::IndirectList(int MaxListSize)
{
  MaxSize = MaxListSize;
  table = new T *[MaxSize];
  length = 0;
}

IndirectList<T>::~IndirectList(){
  for (int i = 0; i < length; i++)
    delete table[i];
  delete [] table;
}
```

Other operations

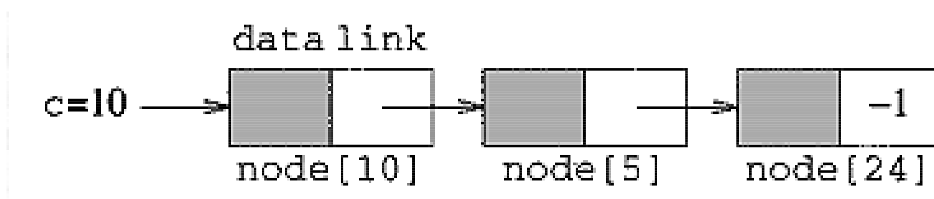
`Find(x, k)` simply returns the element pointed by `table[k-1]`. `Search(x)` simply goes through `table[0], table[1],...,table[MaxSize-1]` to look for `x`. Below gives the code for `Delete(x, k)`.

```
IndirectList<T>& IndirectList<T>::
               Delete(int k, T& x){
  if (Find(k, x)) {
    for (int i = k; i < length; i++)
      table[i-1] = table[i];
   length--;
   return *this;
  }
  else throw OutOfBounds();
  return *this;
}
```

Simulating pointers

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

Assume that we use an array, each element of which has two parts, `data` and `link`. Now, if a chain `c` consists of nodes 10, 5 and 25. We shall have `c=10`, `node[9].link=4`, `node[4].link = 24` and `node[24].link=-1`.



The SimNode class

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

class SimSpace {
 friend SimChain<T>;
 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
};
```

SimSpace operations

We first need to implement the constructor, which simply initialize the available space list, by applying for enough space, hooking up all the nodes, and returning 0 as the head address.

```
SimSpace<T>::SimSpace(int MaxSpaceSize){
  NumberOfNodes = MaxSpaceSize;
  node = new SimNode<T> [NumberOfNodes];
  for (int i = 0; i < NumberOfNodes-1; i++)
    node[i].link = i+1;
  node[NumberOfNodes-1].link = -1;
  first = 0;
}
```

Allocate and deallocate

The key operations are how to get a node assigned, and send it back when it is no longer needed.

```
int SimSpace<T>::Allocate(){
  if (first == -1) throw NoMem();
  int i = first;
  first = node[i].link;
  return i;
}

void SimSpace<T>::Deallocate(int& i){
  node[i].link = first;
  first = i;
  i = -1;
}
```

Comparisons

The following table compares the asymptotic complexities of various operations on a linear list, using various data representation mechanisms, where s and n refer to the size of the element and the length of the list.

Methods	`Find(k)`	`Delete(k)`	`Insert(k)`
Formula	$\Theta(1)$	$O((n-k)s)$	$O((n-k)s)$
Linked	$O(k)$	$O(k)$	$O(k+s)$
Indirect	$\Theta(1)$	$O(n-k)$	$O(n-k)$

In terms of space, indirect addressing uses about the same space as does a linked list. Both of them use more space than a formula-based mechanism.

When the lists are ordered, then with both formula-based and indirect addressing, we can carry out a search in $\Theta(\log n)$. Otherwise, such a search has to take $O(n)$.

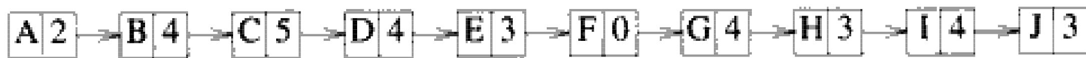
Bin sort

Assume we have maintained a list of students, together with their test scores. Assume that their average scores are integers between 0 and 100, we want to sort them according to their average scores. It will take $O(n^2)$ to do that, if we use some of the methods we have leant in the previous chapter.

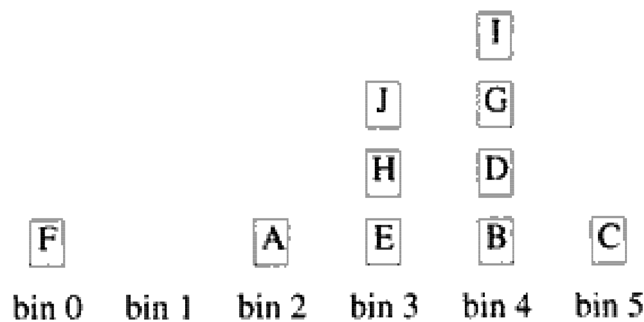
A faster way in this case is to use *bin sort,* in which we prepare 101 bins, each of which is labeled with a score. We then put all the nodes into bins according to their scores. We finally create a sorted list by combining those bins.

An example

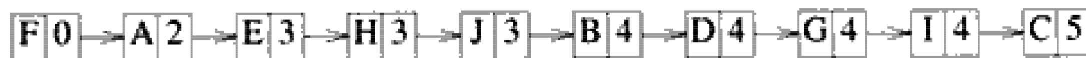
In this example, since there are only 6 different scores, we use just 6 bins.



(a) Input chain



(b) Nodes in bins



(c) Sorted chain

Operation overloading

In order to properly compare nodes, we have to overload such operations as $\neq$.

```
class Node {
 friend ostream& operator<<
     (ostream&, const Node &);
 friend void BinSort(Chain<Node>&, int);
 friend void main();
 public:
  int operator !=(Node x) const
     {return (score != x.score
            || name[0] != x.name[0]);}
  operator int() const {return score;}
 private:
  int score;
  char *name;
};

ostream& operator<<(ostream& out, const Node& x)
   {out << x.score << ' ' << x.name[0]
        << ' '; return out;}
```

The `bin sort`

If the input list is of `chain` type, we can implement bin sort by successively deleting nodes from the chain, adding it into corresponding bins, and then concatenate all the bins.

```
void BinSort(Chain<Node>& X, int range){
 int len = X.Length();
 Node x;
 Chain<Node> *bin;
 bin = new Chain<Node> [range + 1];
 for (int i = 1; i <= len; i++) {
   X.Delete(1,x);
   bin[x.score].Insert(0,x);}
 for (int j = range; j >= 0; j--)
   while (!bin[j].IsEmpty()) {
     bin[j].Delete(1,x);
     X.Insert(0,x);}
 delete [] bin;
}
```

A couple of issues

1. Notice that does not change the relative order of nodes that have the same score. Such a sort is called a *stable* sort.

2. For the first `for loop`, each node has to be deleted and inserted into a bin, there are $\Theta(n)$ work. For the second `for loop`, there are $\Theta(range)$ tests in the `while` loop, when the test succeeds, it will delete nodes and put them into the chain, there are exactly $\Theta(n)$ such nodes. Hence, the total time complexity is $\Theta(n + range)$.

3. We can also include bin sort into the `Chain` class so that we can use the same physical nodes no matter they are in bins or they are in the original chain. We also can eliminate the need for all the `new` and `delete` invocations.

Radix sort

The bin sort method may be extended further to *radix sort,* which can sort, in $\Theta(n)$ time, n integers in the range 0 through $n^c - 1$, where c is a constant. Notice that if we were to directly apply bin sort on this range, the time complexity would be $\Theta(n^c)$.

Assume that we want to sort 10 numbers in the range 0 through 999. If we directly apply bin sort on them, we have to take 1000 steps to initialize the 1000 bins, 10 steps to put them into the bins, and another 1000 steps to collect them; 2010 steps in total.

Another approach is to apply bin sort three times on their digits, from the rightmost to the leftmost.

More specifically, we go through the following steps:
1) Use bin sort to sort the 10 numbers by their least significant digit. As a result, the resulted sequence is sorted by the rightmost digit.

2) We apply bin sort again on the second digit. As a result, the sequence will be sorted by the second digit. Moreover, since bin sort is stable, among all the numbers that have the same second digit, they remain sorted by the third digit. Hence, now the numbers are sorted by the last two digits.

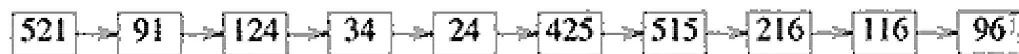
3) We finally bin sort on the first digit. Again, because of the stability, the numbers are completely sorted.

In terms of time complexity, since range=10 in all cases. The total complexity will not be more than 100. More generally, it will be $\Theta(cn)$.

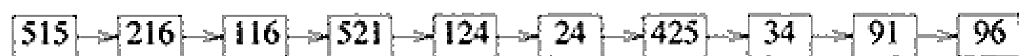
An example

Below gives a concrete example of radix sort.

(a) Input chain



(b) Chain after sorting on least significant digit



(c) Chain after sorting on 2nd least significant digit

(d) Chain after sorting on most significant digit

Equivalence relations

A *relation* is defined on a set S if for every pair of elements (a, b), $a, b \in S$, aRb is either true or false. If aRb is ture, then we say that a *is related to* b

An *equivalence relation* is a relation R that satisfies the following three properties:

1. *Reflexive:*

   aRb, for all $a \in S$.

2. *symmetric:*

   aRb if only if bRa.

3. *Transitive:*

   aRb and bRc implies that aRc.

Examples

The $\leq$ relation is not equivalent, as it is not symmetric. Electrical connectivity is equivalent. The relation satisfied by any pair of cities if they are located in the same country is equivalent, as well.

Assume $n = 14$ and $R = \{(1, 11), (7, 11), (2, 12),$ $(12, 8), (11, 12),(3, 13), (4, 13),(13, 14), (14, 9),$ $(5, 14), (6, 10)\}$. All the pairs in the form of (a, a) have been omitted because of the reflexivity. Also, since $(1, 11) \in R$, by symmetry, $(11, 1) \in R$, as well. Other omitted pairs can be obtained via the transitivity, e.g., since both $(7, 11)$ and $(11, 12)$ are in R, so should $(7, 12)$.

We say a and b are equivalent,iff $(a, b) \in R$. An *equivalent class* is the maximal set of equivalent elements. For example, in the above example, since 1 and 11, 11 and 12 are equivalent, elements 1, 11, and 12 are equivalent to each other. But, they don't form an equivalent class, since they are also equivalent to 7.

Actually, R defines three equivalent classes: $\{1, 2, 7, 8, 11\}$, $\{3, 4, 5, 8, 13, 14\}$, and $\{6, 10\}$.

Given an equivalence relation R and n,, the *equivalence class* problem is to determine the equivalent classes, i.e., given any two elements, whether or not they are related.

On line equivalent class problem

In this problem, we begin with n elements, each in a separate equivalent class. We can find out all the equivalent classes by carrying out a series of operations: 1) `combine(a, b)` is to combine the equivalent classes that contain a and b into a single class; and 2) `Find(e)` is to determine the class that contains e. The purpose for the latter operation is to decide if two elements belong to the same class.

Moreover, the `combine(a, b)` operation can be done in terms of `Find` and `Union`, which takes two classes and makes one:

```
i=Find(a); j=Find(b);
if(i !=j) Union(i, j);
```

What will we do?

With the help of `Union` and `Find`, we can add new relations to R. More specifically, before we add new relation (a, b), we use `Find(a, b)` to check if $(a, b) \in R$. If it is the case, then we don't need to add it to R, Otherwise, we will apply `Union(a, b)` to merge the two classes into one.

In this section, we will study the online version of the equivalent class problem, and develop some simple solutions. This problem has immediate application. For example, an electronic circuit consists of components, pins and wires. Two pins are electronically equivalent, iff they are either connected by a wire, or there is a sequence of wires that connect them. a net is a maximal set of electronically equivalent pins.

A simple solution

We use trees to represent equivalent classes, which is identified by its root. Actually, we can use an array, `S`, and let `S[i]` be the root of the tree that currently contains `e`. Finally, `S[i]==0` iff `i` is the root of a tree.

We immediately have the following functions:

```
S = new int [n + 1];

void Initialize(int n){
  for (int e = 1; e <= n; e++)
   S[e] = 0;
}
```

Wrapping up

```
int Find(int e){
   int temp=e;
   while (S[temp]<>0)
     temp=S[temp];
   return temp;
}

void Union(int i, int j){
  int r1=Find(i);
  int r2=Find(j);
  s[r2]=r1;
}
```

It is easy to see that all the functions take $O(n)$.

Possible improvements

One idea is to keep all the elements of the same equivalent class in a linked list. This saves time for the updating. But, this by itself won't cut the asymptotic time.

Another idea is that we also keep the size of every equivalent class, and when performing *union*s, we change the name of the smaller equivalence class to the larger, then the total time spent for $n-1$ merges is $O(nlog(n))$.