Biostatistics 615/815 Lecture 6: Elementary Data Structures

Hyun Min Kang

September 22nd, 2011

Recap Quiz : Time Complexity of Each Sorting Algorithm

- Insertion Sort :

Recap Quiz : Time Complexity of Each Sorting Algorithm

- Insertion Sort : $\Theta(n^2)$

Recap Quiz : Time Complexity of Each Sorting Algorithm

- Insertion Sort : $\Theta(n^2)$
- Merge Sort :

Recap Quiz : Time Complexity of Each Sorting Algorithm

- Insertion Sort : $\Theta(n^2)$
- Merge Sort : $\Theta(n \log n)$

Recap Quiz : Time Complexity of Each Sorting Algorithm

- Insertion Sort : $\Theta(n^2)$
- Merge Sort : $\Theta(n \log n)$
- Quick Sort :

Recap Quiz : Time Complexity of Each Sorting Algorithm

- Insertion Sort : $\Theta(n^2)$
- Merge Sort : $\Theta(n \log n)$
- Quick Sort : $\Theta(n^2)$ worst case, $\Theta(n \log n)$ amortized

Recap Quiz : Time Complexity of Each Sorting Algorithm

- Insertion Sort : $\Theta(n^2)$
- Merge Sort : $\Theta(n \log n)$
- Quick Sort : $\Theta(n^2)$ worst case, $\Theta(n \log n)$ amortized
- Counting Sort :

Recap Quiz : Time Complexity of Each Sorting Algorithm

- Insertion Sort : $\Theta(n^2)$
- Merge Sort : $\Theta(n \log n)$
- Quick Sort : $\Theta(n^2)$ worst case, $\Theta(n \log n)$ amortized
- Counting Sort : $\Theta(n)$ but requires $\Omega(|X|)$ memory for possible input set X.
- Radix Sort :

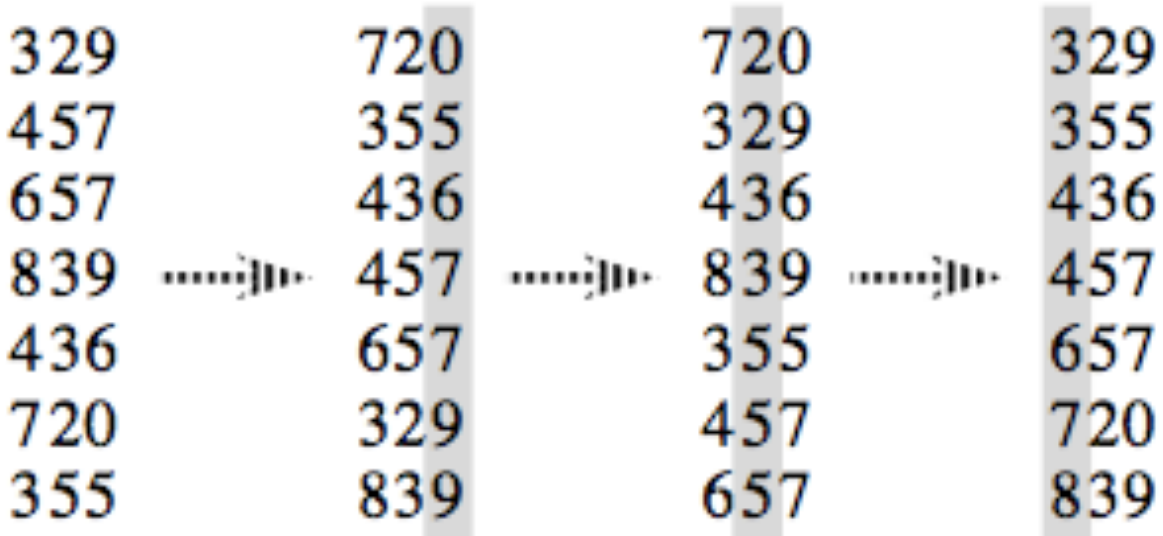
Recap Quiz : Time Complexity of Each Sorting Algorithm

- Insertion Sort : $\Theta(n^2)$
- Merge Sort : $\Theta(n \log n)$
- Quick Sort : $\Theta(n^2)$ worst case, $\Theta(n \log n)$ amortized
- Counting Sort : $\Theta(n)$ but requires $\Omega(|X|)$ memory for possible input set X.
- Radix Sort : $\Theta(nk)$ for k digits, with $\Omega(r)$ for radix r.

Another linear sorting algorithm : Radix sort

Key idea

- Sort the input sequence from the last digit to the first repeatedly using a linear sorting algorithm such as CountingSort
- Applicable to integers within a finite range



Understanding bitwise operator

```
int x = 0x7a;      // hexadecimal number, equivalent to 01111010 in binary
int y = (x >> 4); // move x by 4 bits to right, y = 0111 in binary
int z = (y << 4); // move y by 4 bits to left,  z = 01110000 in binary
int w = 1 << 4;    // 10000 = 2^4 = 16
int u = (1 << 4) - 1; // 1111
int v = (0x7a & ((1 << 4) -1));  // 01111010 & 00001111 = 00001010 = 0x0a (extract
```

Implementing radixSort.cpp

```
// use #[radixBits] bits as radix (e.g. hexadecimal if radixBits=4)
void radixSort(std::vector<int>& A, int radixBits, int max) {
  // calculate the number of digits required to represent the maximum number
  int nIter = (int)(ceil(log((double)max)/log(2.)/radixBits));
  int nCounts = (1 << radixBits); // 1<<radixBits == 2^radixBits == # of digits
  int mask = nCounts-1;           // mask for extracting #(radixBits) bits
  std::vector< std::vector<int> > B; // vector of vector, each containing
                // the list of input values containing a particular digit
  B.resize(nCounts);
  for(int i=0; i < nIter; ++i) {
    // initialze each element of B as a empty vector
    for(int j=0; j < nCounts; ++j) { B[j].clear(); }
    // distribute the input sequences into multiple bins, based on i-th digit
    radixSortDivide(A, B, radixBits*i, mask);
    // merge the distributed sequences B into original array A
    radixSortMerge(A, B);
  }
}
```

Implementing radixSort.cpp

```
// divide input sequences based on a particular digit
void radixSortDivide(std::vector<int>& A,
                     std::vector< std::vector<int> >& B, int shift, int mask) {
  for(int i=0; i < (int)A.size(); ++i) {
    // (A[i]>>shift)&mask takes last [shift .. shift+radixBits-1] bits of A[i]
    B[ (A[i] >> shift) & mask ].push_back(A[i]);
  }
}
// merge the partitioned sequences into single array
void radixSortMerge(std::vector<int>& A, std::vector< std::vector<int> >&B ) {
  for(int i=0, k=0; i < (int)B.size(); ++i) {
    for(int j=0; j < (int)B[i].size(); ++j) {
      A[k] = B[i][j]; // iterate each bin of digit and concatenate all values
      ++k;
    }
  }
}
```

Bitwise operation examples

shift=3, radixBits=1, A[i] = 117

```
117    = 1110101
----------------
117>>3 =    1110
mask   =       1
---------------
ret    =       0
```

Bitwise operation examples

shift=3, radixBits=1, A[i] = 117

```
117    = 1110101
---------------
117>>3 =    1110
mask   =       1
---------------
ret    =       0
```

shift=3, radixBits=3, A[i] = 117

```
117    = 1110101
---------------
117>>3 =    1110
mask   =     111
---------------
ret    =     110
```

Radix sort in practice

```
user@host:~/> time cat src/sample.input.txt | src/stdSort > /dev/null
real 0m0.430s
user 0m0.281s
sys 0m0.130s

user@host:~/> time cat src/sample.input.txt | src/insertionSort > /dev/null
real 1m8.795s
user 1m8.181s
sys 0m0.206s

user@host:~/> time cat src/sample.input.txt | src/quickSort > /dev/null
real 0m0.427s
user 0m0.285s
sys 0m0.129s

user@host:~/> time cat src/sample.input.txt | src/radixSort 8 > /dev/null
real 0m0.334s
user 0m0.195s
sys 0m0.129s
```

Elementary data structure

Container

A container T is a generic data structure which supports the following three operation for an object x.

- $\textsc{Search}(T, x)$
- $\textsc{Insert}(T, x)$
- $\textsc{Delete}(T, x)$

Possible types of container

- Arrays
- Linked lists
- Trees
- Hashes

Average time complexity of container operations

	SEARCH	INSERT	DELETE
Array	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
SortedArray	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
List	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Hash	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

- Array or list is simple and fast enough for small-sized data
- Tree is easier to scale up to moderate to large-sized data
- Hash is the most robust for very large datasets

Arrays

Key features

- Stores the data in a consecutive memory space
- Fastest when the data size is small due to locality of data

Using std::vector as array

```
std::vector<int> v;  // creates an empty vector
// INSERT : append at the end, O(1)
v.push_back(10);
// SEARCH : find a value scanning from begin to end, O(n)
std::vector<int>::iterator i = std::find(v.begin(), v.end(), 10);
if ( i != v.end() ) { std::cout << "Found " << (*i) << std::endl; }
// DELETE : search first, and delete, O(n)
if ( i != v.end() ) { v.erase(i); } // delete an element
```

Implementing data structure on your own

myArray.h

```
class myArray {
  int* data;
  int size;
  void insert(int x);
  ...
};
```

myArray.cpp

```
#include "myArray.h"
void myArray::insert(int x) { // function body goes here
...
```

Main.cpp

```
#include <iostream>
#include "myArray.h"
int main(int argc, char** argv) {
...
```

Building your progam

Individually compile and link

- Include the content of your `.cpp` files into `.h`
- For example, `Main.cpp` includes `myArray.h`

```
user@host:~/> g++ -o myArrayTest Main.cpp
```

Or create a Makefile and just type 'make'

```
all: myArrayTest  # binary name is myArrayTest

myArrayTest: Main.cpp  # link two object files to build binary
    g++ -o myArrayTest Main.cpp  # must start with a tab

clean:
    rm *.o myArrayTest
```

Designing a simple array - myArray.h

```
// myArray.h declares the interface of the class, and the definition is in myArray
#define DEFAULT_ALLOC 1024
template <class T> // template supporting a generic type
class myArray {
protected:     // member variables hidden from outside
   T *data;    // array of the generic type
   int size;   // number of elements in the container
   int nalloc; // # of objects allocated in the memory
public:      // abstract interface visible to outside
   myArray();          // default constructor
   ~myArray();         // destructor
   void insert(const T& x); // insert an element x
   int search(const T& x);  // search for an element x and return its location
   bool remove(const T& x); // delete a particular element
};
```

Using a simple array Main.cpp

```
#include <iostream>
#include "myArray.h"
int main(int argc, char** argv) {
  myArray<int> A;
  A.insert(10);                // insert example
  if ( A.search(10) > 0 ) {    // search example
    std::cout << "Found element 10" << std::endl;
  }
  A.remove(10);                // remove example
  return 0;
}
```

Implementing a simple array myArray.cpp

```
template <class T>
myArray<T>::myArray() {  // default constructor
  size = 0;              // array do not have element initially
  nalloc = DEFAULT_ALLOC;
  data = new T[nalloc];  // allocate default # of objects in memory
}

template <class T>
myArray<T>::~myArray() { // destructor
  if ( data != NULL ) {
    delete [] data;      // delete the allocated memory before destroying
  }                      // the object. otherwise, memory leak happens
}
```

myArray.cpp : insert

```
template <class T>
void myArray<T>::insert(const T& x) {
  if ( size >= nalloc ) {  // if container has more elements than allocated
    T* newdata = new T[nalloc*2];   // make an array at doubled size
    for(int i=0; i < nalloc; ++i) {
      newdata[i] = data[i];         // copy the contents of array
    }
    delete [] data;                 // delete the original array
    data = newdata;                 // and reassign data ptr
    nalloc *= 2;                    // double the allocation
  }
  data[size] = x;                   // push back to the last element
  ++size;                           // increase the size
}
```

myArray.cpp : search

```
template <class T>
int myArray<T>::search(const T& x) {
  for(int i=0; i < size; ++i) { // iterate each element
    if ( data[i] == x ) {
      return i;                 // and return index of the first match
    }
  }
  return -1;                    // return -1 if no match found
}
```

myArray.cpp : remove

```
template <class T>
bool myArray<T>::remove(const T& x) {
  int i = search(x);          // try to find the element
  if ( i > 0 ) {              // if found
    for(int j=i; j < size-1; ++j) {
      data[i] = data[i+1];    // shift all the elements by one
    }
    --size;                   // and reduce the array size
    return true;              // successfully removed the value
  }
  else {
    return false;             // could not find the value to remove
  }
}
```

Implementing complex data types is not so simple

```
int main(int argc, char** argv) {
  myArray<int> A;          // creating an instance of myArray
  A.insert(10);
  A.insert(20);
  myArray<int> B = A;      // copy the instance
  B.remove(10);
  if ( A.search(10) < 0 ) {
    std::cout << "Cannot find 10" << std::endl; // what would happen?
  }
  return 0;                // would to program terminate without errors?
}
```

Implementing complex data types is not so simple

```
int main(int argc, char** argv) {
  myArray<int> A;          // A is empty, A.data points an address x
  A.insert(10);            // A.data[0] = 10, A.size = 1
  A.insert(20);            // A.data[0] = 10, A.data[1] = 20, A.size = 2
  myArray<int> B = A;      // shallow copy, B.size == A.size, B.data == A.data
  B.remove(10);            // A.data[0] = 20, A size = 2 -- NOT GOOD
  if ( A.search(10) < 0 ) {
    std::cout << "Cannot find 10" << std::endl; // A.data is unwillingly modified
  }
  return 0;  // ERROR : both delete [] A.data and delete [] B.data is called
}
```

How to fix it

A naive fix : preventing object-to-object copy

```
template <class T>
class myArray {
protected:
   T *data;
   int size;
   int nalloc;
   myArray(myArray& a) {}; // do not allow copying object
public:
   myArray() {...};        // allow to create an object from scratch
```

How to fix it

A naive fix : preventing object-to-object copy

```
template <class T>
class myArray {
protected:
   T *data;
   int size;
   int nalloc;
   myArray(myArray& a) {}; // do not allow copying object
public:
   myArray() {...};        // allow to create an object from scratch
```

A complete fix

- `std::vector` does not suffer from these problems
- Implementing such a nicely-behaving complex object is NOT trivial
- Requires a deep understanding of C++ programming language

Sorted Array

Key Idea

- Same structure with `Array`
- Ensure that elements are sorted when inserting and deleting an object
- Insertion takes longer, but search will be much faster
  - $\Theta(n)$ for insert
  - $\Theta(\log n)$ for search

Sorted Array

Key Idea

- Same structure with `Array`
- Ensure that elements are sorted when inserting and deleting an object
- Insertion takes longer, but search will be much faster
  - $\Theta(n)$ for insert
  - $\Theta(\log n)$ for search

Algorithms

Sorted Array

Key Idea

- Same structure with `Array`
- Ensure that elements are sorted when inserting and deleting an object
- Insertion takes longer, but search will be much faster
  - $\Theta(n)$ for insert
  - $\Theta(\log n)$ for search

Algorithms

Insert Insert the element at the end, and swap with the previous element if larger

Sorted Array

Key Idea

- Same structure with `Array`
- Ensure that elements are sorted when inserting and deleting an object
- Insertion takes longer, but search will be much faster
  - $\Theta(n)$ for insert
  - $\Theta(\log n)$ for search

Algorithms

Insert Insert the element at the end, and swap with the previous element if larger

- Same as a single iteration of InsertionSort

Sorted Array

Key Idea

- Same structure with `Array`
- Ensure that elements are sorted when inserting and deleting an object
- Insertion takes longer, but search will be much faster
  - $\Theta(n)$ for insert
  - $\Theta(\log n)$ for search

Algorithms

Insert Insert the element at the end, and swap with the previous element if larger

- Same as a single iteration of InsertionSort

Search Use the binary search algorithm

Sorted Array

Key Idea

- Same structure with `Array`
- Ensure that elements are sorted when inserting and deleting an object
- Insertion takes longer, but search will be much faster
  - $\Theta(n)$ for insert
  - $\Theta(\log n)$ for search

Algorithms

Insert Insert the element at the end, and swap with the previous element if larger

- Same as a single iteration of InsertionSort

Search Use the binary search algorithm

Remove Same as the unsorted version of `Array`

Implementation : mySortedArray.h

```
// Exactly the same as myArray.h
#include <iostream>
#define DEFAULT_ALLOC 1024
template <class T> // template supporting a generic type
class mySortedArray {
protected:      // member variables hidden from outside
   T *data;     // array of the generic type
   int size;    // number of elements in the container
   int nalloc; // # of objects allocated in the memory
   mySortedArray(mySortedArray& a) {}; // for disabling object copy
   int search(const T& x, int begin, int end); // search with ranges
public:       // abstract interface visible to outside
   mySortedArray();          // default constructor
   ~mySortedArray();         // destructor
   void insert(const T& x); // insert an element x
   int search(const T& x);  // search for an element x and return its location
   bool remove(const T& x); // delete a particular element
};
```

Implementation : Main.cpp

```
int main(int argc, char** argv) {
  mySortedArray<int> A;
  A.insert(10);              // {10}
  A.insert(5);               // {5,10}
  A.insert(20);              // {5,10,20}
  A.insert(7);               // {5,7,10,20}
  std::cout << "A.search(7) = " << A.search(7) << std::endl;   // returns 1
  std::cout << "A.search(10) = " << A.search(10) << std::endl; // returns 2
  mySortedArray<int>& B = A; // copy is disallowed but reference is allowed
  std::cout << "B.search(10) = " << B.search(10) << std::endl; // returns 2
  return 0;
}
```

Implementation : mySortedArray::insert()

```
template <class T>
void mySortedArray<T>::insert(const T& x) {
  if ( size >= nalloc ) {  // if container has more elements than allocated
    T* newdata = new T[nalloc*2];   // make an array at doubled size
    for(int i=0; i < nalloc; ++i) {
      newdata[i] = data[i];         // copy the contents of array
    }
    delete [] data;                 // delete the original array
    data = newdata;                 // and reassign data ptr
    nalloc *= 2;                    // and double the nalloc
  }

  int i;  // scan from last to first until find smaller element
  for(i=size-1; (i >= 0) && (data[i] > x); --i) {
    data[i+1] = data[i];            // shift the elements to right
  }
  data[i+1] = x;                    // insert the element at the right position
  ++size;                           // increase the size
}
```

Implementation : mySortedArray::search()

```
template <class T>
int mySortedArray<T>::search(const T& x) {
  return search(x, 0, size-1);
}
template <class T>   // simple binary search
int mySortedArray<T>::search(const T& x, int begin, int end) {
  if ( begin > end )
    return -1;
  else {
    int mid = (begin+end)/2;
    if ( data[mid] == x )
      return mid;
    else if ( data[mid] < x )
      return search(x, mid+1, end);
    else
      return search(x, begin, mid);
  }
}
```

Implementation : mySortedArray::remove()

```
// same as myArray::remove()
template <class T>
bool mySortedArray<T>::remove(const T& x) {
  int i = search(x);  // try to find the element
  if ( i >= 0 ) {      // if found
    for(int j=i; j < size-1; ++j) {
      data[i] = data[i+1];  // shift all the elements by one
    }
    --size;             // and reduce the array size
    return true;        // successfully removed the value
  }
  else {
    return false;       // cannot find the value to remove
  }
}
```

Summary

Today

- Radix Sort
- Array
- Sorted array

Next Lectures

- Linked list
- Binary search tree
- Hash tables
- Dynamic Programming