Biostatistics 615/815 Lecture 6: Elementary Data Structures

Hyun Min Kang

September 20th, 2012

Merge Sort

Divide and conquer algorithm

Divide Divide the n element sequence to be sorted into two subsequences of $n/2$ elements each

Conquer Sort the two subsequences recursively using merge sort

Combine Merge the two sorted subsequences to produce the sorted answer

Time complexity

- $\Theta(n \log n)$ algorithm in worst case
- Need additional memory for array copy
- In practice, slightly slower than other $\Theta(n \log n)$ algorithms due to overhead of array copy

Quicksort Algorithm

Algorithm QUICKSORT

Data: array A and indices p and r
Result: $A[p..r]$ is sorted
if $p < r$ **then**
    $q = \text{PARTITION}(A,p,r)$;
    $\text{QUICKSORT}(A,p,q-1)$;
    $\text{QUICKSORT}(A,q+1,r)$;
end

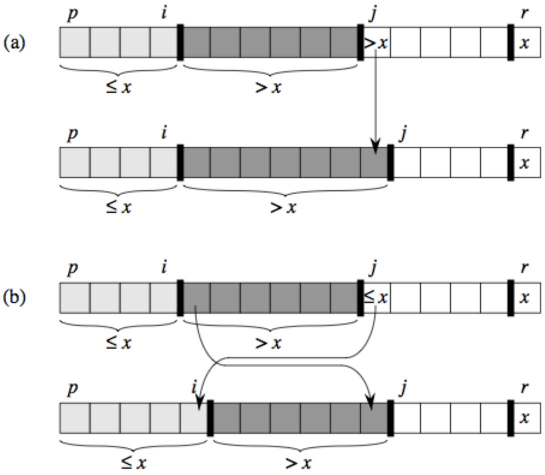
Quicksort Algorithm

Algorithm PARTITION

Data: array A and indices p and r
Result: Returns q such that $A[p..q-1] \leq A[q] \leq A[q+1..r]$

$x = A[r]$;
$i = p - 1$;
for $j = p$ **to** $r - 1$ **do**
    if $A[j] \leq x$ **then**
        $i = i + 1$;
        $\textsc{Exchange}(A[i], A[j])$;
    end
end
$\textsc{Exchange}(A[i+1], A[r])$;
return $i + 1$;

How PARTITION Algorithm Works



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

Designing a simple array - myArray.h

```
#include <iostream>
#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:
  myArray(); // default constructor
  ~myArray(); // destructor
  void insert(const T& x); // insert an element x, const means read-only
  bool search(const T& x);  // search for an element x and return its location
  bool remove(const T& x); // delete a particular element
  void print();  // print the content of array to the screen
};
```

Using a simple array - myArrayTest.cpp

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

int main(int argc, char** argv) {
  myArray<int> A;
  A.insert(10);          // {10}
  A.insert(5);           // {10,5}
  A.insert(20);          // {10,5,20}
  A.insert(7);           // {10,5,20,7}
  A.print();
  std::cout << "A.search(7) = " << A.search(7) << std::endl;   // true
  std::cout << "A.remove(10) = " << A.remove(10) << std::endl; // {5,20,7}
  A.print();
  std::cout << "A.search(10) = " << A.search(10) << std::endl; // false
  return 0;
}
```

Implementing a simple array in myArray.h

```
class myArray {
  // declarations of member variables and functions go here..
};

// If the function is not yet defined above, it can be defined as follows..
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.h : 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.h : search

```
template <class T>
bool myArray<T>::search(const T& x) {
  for(int i=0; i < size; ++i) { // iterate each element
    if ( data[i] == x ) {
      return true;
    }
  }
  return false;
}
```

myArray.h : remove

```
template <class T>
bool myArray<T>::remove(const T& x) {
  bool found = false;
  for(int i=0; i < size; ++i) { // iterate each element
    if ( data[i] == x ) { found = true; }
    if ( found && i < size-1 ) { data[i] = data[i+1]; }
  }
  if ( found ) --size;
  return found;
}
```

myArray.h : print

```
template <class T>
void myArray<T>::print() {
  if ( size > 0 ) {
    std::cout << "(" << data[0];
    for(int i=1; i < size; ++i) {
      std::cout << "," << data[i];
    }
    std::cout << ")" << std::endl;
  }
  else {
    std::cout << "(EMPTY ARRAY)" << std::endl;
  }
}
```

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) ) {
    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

Summary: Array

- Simplest container
- Constant time for insertion
- $\Theta(n)$ for search
- $\Theta(n)$ for remove
- Elements are clustered in memory, so faster than list in practice.
- Limited by the allocation size. $\Theta(n)$ needed for expansion

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

```
#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
   bool 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
   bool search(const T& x);  // search for an element x and return its location
   bool remove(const T& x); // delete a particular element
   void print();  // print the content of array to the screen
};
```

Implementation : mySortedArrayTest.cpp

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

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}
  A.print();
  std::cout << "A.search(7) = " << A.search(7) << std::endl;   // true (1)
  std::cout << "A.remove(10) = " << A.remove(10) << std::endl; // true (1)
  A.print();
  std::cout << "A.search(10) = " << A.search(10) << std::endl; // false (0)
  return 0;
}
```

Constructors and destructors

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

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>
bool mySortedArray<T>::search(const T& x) {
  return search(x, 0, size-1);
}

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

Implementation : mySortedArray::remove()

```
// same as myArray::remove()
template <class T>
bool mySortedArray<T>::remove(const T& x) {
  bool found = false;
  for(int i=0; i < size; ++i) { // iterate each element
    if ( data[i] == x ) { found = true; }
    if ( found && i < size-1 ) { data[i] = data[i+1]; }
  }
  if ( found ) --size;
  return found;
}
```

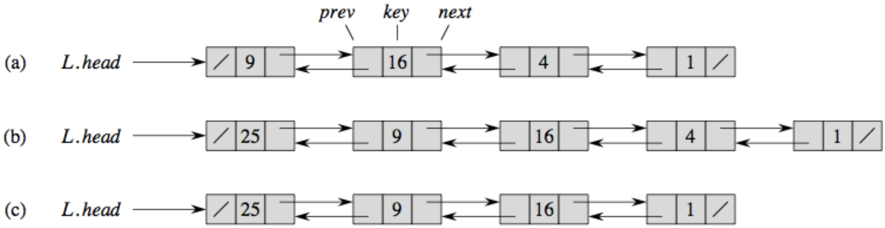
Summary: SortedArray

- $\Theta(n)$ for insertion
- $\Theta(\log n)$ for search
- $\Theta(n)$ for remove
- Optimal for frequent searches and infrequent updates
- Limited by the allocation size. $\Theta(n)$ needed for expansion

Linked List

- A data structure where the objects are arranged in linear order
- Each object contains the pointer to the next object
- Objects do not exist in consecutive memory space
  - No need to shift elements for insertions and deletions
  - No need to allocate/reallocate the memory space
  - Need to traverse elements one by one
  - Likely inefficient than `Array` in practice because data is not necessarily localized in memory
- Variants in implementation
  - (Singly-) linked list
  - Doubly-linked list

Example of a linked list



- Example of a doubly-linked list
- Singly-linked list if `prev` field does not exist

Implementation of singly-linked list

myList.h

```
#include "myListNode.h"
template <class T>
class myList {
protected:
  myListNode<T>* head; // list only contains the pointer to head
  myList(myList& a) {};    // prevent copying
public:
  myList() : head(NULL) {} // initially header is NIL
  ~myList();
  void insert(const T& x); // insert an element x
  bool search(const T& x);  // search for an element x and return its location
  bool remove(const T& x); // delete a particular element
  void print();  // print the content of array to the screen
};
```

List implementation : `class myListNode`

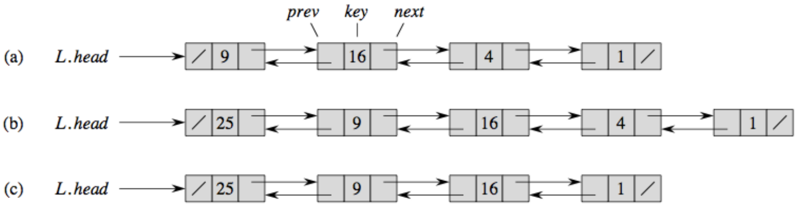
myListNode.h

```
// myListNode class is only accessible from myList class
template<class T>
class myListNode {
protected:
  T value;           // the value of each element
  myListNode<T>* next;  // pointer to the next element
  myListNode(const T& x, myListNode<T>* n) : value(x), next(n) {} // constructor
  ~myListNode();
  bool search(const T& x);
  myListNode<T>* remove(const T& x, myListNode<T>*& prevNext);
  void print(char c);
  template <class S> friend class myList; // allow full access to myList class
};
```

Inserting an element to a list

myList.h

```
template <class T>
void myList<T>::insert(const T& x) {
  // create a new node, and make them head
  // and assign the original head to head->next
  head = new myListNode<T>(x, head);
}
```



Destructor is required because new was used

myList.h

```
template <class T>
myList<T>::~myList() {
  if ( head != NULL ) {
    delete head;    // delete dependent objects before deleting itself
  }
}
```

myListNode.cpp

```
template <class T>
myListNode<T>::~myListNode() {
  if ( next != NULL ) {
    delete next;  // recursively calling destructor until the end of the list
  }
}
```