Announcements

A good and bad news
- Homework #2 will be announced in the next lecture

815 projects
- 5-6 team pairs in total
- Team assignment will be made during this week
- Each team should set up a meeting with the instructor to kick-start the project

Recap on sorting algorithms: Insertion sort

Algorithm description
1. For each \( j \in [2 \cdots n] \), iterate element at indices \( j-1, j-2, \cdots 1 \).
3. If \( A[i] \leq A[j] \), increase \( j \) and go to step 1

Insertion sort is loop invariant
At the start of each iteration, \( A[1 \cdots j-1] \) is loop invariant iff:
- \( A[1 \cdots j-1] \) consist of elements originally in \( A[1 \cdots j-1] \).
- \( A[1 \cdots j-1] \) is in sorted order.

Time complexity of Insertion sort
- Worst and average case time-complexity is \( \Theta(n^2) \)

Recap on sorting algorithms: Mergesort

Algorithm Mergesort
Data: array \( A \) and indices \( p \) and \( r \)
Result: \( A[p..r] \) is sorted
if \( p < r \) then
    \( q = \lfloor (p + r)/2 \rfloor \);
    Mergesort\( (A, p, q) \);
    Mergesort\( (A, q+1, r) \);
    Merge\( (A, p, q, r) \);
end

Time complexity of Mergesort
- Worst and average case time-complexity is \( \Theta(n \log n) \)
Recap on sorting algorithms: Quicksort

Algorithm QUICKSORT

Data: array $A$ and indices $p$ and $r$
Result: $A[p..r]$ is sorted

if $p < r$ then
    $q = \text{PARTITION}(A, p, r)$;
    QUICKSORT($A, p, q - 1$);
    QUICKSORT($A, q + 1, r$);
end

Time complexity of Quicksort

- Average case time-complexity is $\Theta(n \log n)$
- Worst case time-complexity is $\Theta(n^2)$, but practically faster than other $\Theta(n \log n)$ algorithms.

Performance of sorting algorithms in practice

Running example with 100,000 elements (in UNIX or MacOS)

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/mergeSort > /dev/null
real 0m0.898s
user 0m0.755s
sys 0m0.131s

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

How PARTITION algorithm works

Lower bounds for comparison sorting

CLRS Theorem 8.1

Any comparison-based sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case

An informal proof

- Any comparison sort algorithm can be represented as a binary decision tree, where each node represents a comparison. Each path from the root to leaf represents possible series of comparisons to sort a sequence.
- Each leaf of the decision tree represents one of $n!$ possible permutations of input sequences
- We have $n! \leq l \leq 2^h$, where $l$ is the number of leaf nodes, and $h$ is the height of the tree, equivalent to the $\#$ of comparisons.
- Then it implies $h \geq \log(n!) = \Theta(n \log n)$
Finding faster sorting methods

Sorting faster than $\Theta(n \log n)$
- Comparison-based sorting algorithms cannot be faster than $\Theta(n \log n)$
- Sorting algorithms NOT based on comparisons may be faster

Linear time sorting algorithms
- Counting sort
- Radix sort
- Bucket sort

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 Counting Sort
- Applicable to integers within a finite range

Example decision-tree representing Insertion Sort

A linear sorting algorithm: Counting sort

A restrictive input setting
- The input sequences have a finite range with many expected duplication.
- For example, each elements of input sequences is one digit number, and your input sequences are millions.

Key idea
1. Scan through each input sequence and count number of occurrences of each possible input value.
2. From the smallest to largest possible input value, output each value repeatedly by its stored count.
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) {
        // initialize each element of B as an 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);
    }
}
```

Bitwise operation examples

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

117 = 1110101
     110101

117>>3 = 110
mask = 1
     000

ret = 110
```

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

117 = 1110101
     110101

117>>3 = 110
mask = 111
     111

ret = 110
```

Radix sort in practice

```
user@host:~/time cat src/sample.input.txt | src/quickSort > /dev/null 
real 0m0.430s 
user 0m0.281s 
sys 0m0.149s 

user@host:~/time cat src/sample.input.txt | src/insertionSort > /dev/null 
real 0m1.824s 
user 0m0.285s 
sys 0m0.212s 

user@host:~/time cat src/sample.input.txt | src/quickSort > /dev/null 
real 0m0.447s 
user 0m0.343s 
sys 0m0.103s 

user@host:~/time cat src/sample.input.txt | src/quickSort > /dev/null 
real 0m0.437s 
user 0m0.334s 
sys 0m0.109s 
```
Elementary data structure

**Container**

A container $T$ is a genetic data structure which supports the following three operations for an object $x$.

- **SEARCH**$(T, x)$
- **INSERT**$(T, x)$
- **DELETE**$(T, x)$

**Possible types of container**

- Arrays
- Linked lists
- Trees
- Hashes

Average time complexity of container operations

<table>
<thead>
<tr>
<th></th>
<th>SEARCH</th>
<th>INSERT</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Tree</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>Hash</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

- 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

Implementing data structure on your own

**myArray.h**

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

**myArray.cpp**

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

**Main.cpp**

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

Individually compile and link

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

Or create a Makefile and just type 'make'

```bash
all: myArrayTest # binary name is myArrayTest

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

Main.o: Main.cpp myArray.h # compile to build an object file
  g++ -c Main.cpp

myArray.o: myArray.cpp myArray.h # compile to build an object file
  g++ -c myArray.cpp

clean:
  rm *.o myArrayTest
```

Using a simple array Main.cpp

```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

```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
  }
}
```

Designing a simple array - myArray.h

```cpp
// myArray.h declares the interface of the class, and the definition is in myArray.cpp
#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 genetic type
    int size; // number of elements in the container
    int nalloc; // # of objects allocated in the memory

class myArray {
  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
};
```

Hyun Min Kang
myArray.cpp : insert

```cpp
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

```cpp
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

```cpp
template <class T>
bool myArray<T>::remove(const T & x) {
    int i = search(x); // try to find the element
    if ( i > -1 ) { // if found
        for(int j=i; j < size-1; ++j) {
            data[j] = data[j+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

```cpp
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;
    public:
        myArray(myArray& a) {}; // do not allow copying object
        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

A practical advice in implementing a C++ class

- When there are already proven implementations, always utilize them
  - Standard Template Library for basic data structures
  - Boost Library for more sophisticated data types (e.g., Graphs)
  - Eigen package for matrix operations
  - Always check the license carefully, especially if do not want to release your source code
- If it is necessary to implement your own complex data types
  - Use STL (or other well-behaving) data types as member variables whenever possible
  - Keep the behavior simple and well-defined to reduce implementation overhead
  - However, if you spend your time to design your data type robust against many complex situations, your class will be very useful to others.

Overview of elementary data structures

- Sorted array
- Linked list
- Binary search tree
- Hash table

Reading materials

- CLRS Chapter 10
- CLRS Chapter 11
- CLRS Chapter 12