CSCI 1200 Data Structures<<<<<<<<<<<<<<<<<<

- **Ziniu Yu**

People<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- **Professors**: William Thompson, Elsa Gonsiorowski
- **TAs**: Andrew Yale, Partha Sarathi Mukherjee, Srinivasan Iyer, Hendrik Weideman, Jassiem Ifill
- **Mentors**: Michael DiBuduo, Cameron Root, Isabella Siu, Renjie Xie, Wilson Gregory

Array<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- **Arrays** are fixed size, and each array knows **NOTHING** about its own size. The programmer must keep track of the size of each array.
- **Arrays** may be sorted using std::sort, just like vectors. Pointers are used in place of iterators. For example, if a is an array of doubles and there are n values in the array, then here's how to **sort the values in the array** into increasing order:

```
std::sort( a, a+n );
```

String<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- **construct a string**
  By default to create an empty string: std::string my_string_var;
  With a specified number of instances of a single char:
  std::string my_string_var2(10, ' ');
  From another string: std::string my_string_var3(my_string_var2);

Vector<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- **push back** is a vector function to append a value to the end of the vector, increasing its size by one. This is an O(1) operation (on average).
- There is **NO** automatic checking of subscript bounds in vector.
- **constructs a vector**
  This constructs a vector of 100 doubles, each entry storing the value 3.14. New entries can be created using push_back, but these will create entries 100, 101, 102, etc.

```
int n = 100;
std::vector<double> b( 100, 3.14 );
```

  This constructs a vector of 10,000 ints, but provides no initial values for these integers. Again, new entries can be created for the vector using push_back. These will create entries 10000, 10001, etc.

```
std::vector<int> c( n*n );
```

  This constructs a vector that is an exact copy of vector b.
  std::vector<double> d(b);
- **Sort the vector**
  std::sort(my_vec.begin(),my_vec.end(),optional_compare_function);
- **Erase** invalidates **all iterators after** the point of erasure in vectors; **push back** and **resize** invalidate **ALL iterators** in a vector The value of any associated vector iterator must be re-assigned / re-initialized after these operations.
- STL vectors / arrays **allow "random-access"** / indexing / [] subscripting. We can immediately jump to an arbitrary location within the vector / array.

List<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- **Sort the list** my_lst.sort(optional_compare_function);
- The **erase** member function (for STL vector and STL list) takes in a single argument, an iterator pointing at an element in the container. It removes that item, and the function returns an iterator pointing at the element after the removed item.
- Similarly, there is an **insert** function for STL vector and STL list that takes in 2 arguments, an iterator and a new element, and adds that element immediately before the item pointed to by the iterator. The function returns an iterator pointing at the newly added element.
- Even though the erase and insert functions have the same syntax for vector and for list, the vector versions are O(n), whereas the list versions are O(1).
- Iterators positioned on an STL vector, at or after the point of an erase operation, are invalidated. Iterators positioned anywhere on an STL vector may be invalid after an insert (or push back or resize) operation.
- Iterators attached to an STL list are **not invalidated** after an insert or erase (except iterators attached to the erased element!) or push back/push front.
- STL lists have **no subscripting operation** (we can't use [] to access data). The only way to get to the middle of a list is to follow pointers one link at a time.

Map<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- **Map erase**
  void erase(iterator p) — erase the pair referred to by iterator p.
  void erase(iterator first, iterator last) — erase all pairs from the map starting at first and going up to, but not including, last.
  size_type erase(const key_type& k) — erase the pair containing key k, returning either 0 or 1, depending on whether or not the key was in a pair in the map
- **Map find:** m.find(key)
  where m is the map object and key is the search key. It returns a map iterator: If the key is in one of the pairs stored in the map, find returns an iterator referring to this pair. If the key is not in one of the pairs stored in the map, find returns m.end().
- **Map insert:** m.insert(std::make_pair(key, value));
  returns a pair of a map iterator and a bool: std::pair<map<key_type, value_type>::iterator, bool> The insert function checks to see if the key being inserted is already in the map. If so, it does not change the value, and returns a (new) pair containing an iterator referring to the existing pair in the map and the bool value false. If not, it enters the pair in the map, and returns a (new) pair containing an iterator referring to the newly added pair in the map and the bool value true.
- **Map erase**
  void erase(iterator p) erase the pair referred to by iterator p.
  void erase(iterator first, iterator last) erase all pairs from the map starting at first and going up to, but not including, last.
  size_type erase(const key_type& k) erase the pair containing key k, returning either 0 or 1, depending on whether or not the key was in a pair in the map
- we can use any class we want as long as it has an **operator<** defined on it.

Set<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- **Set insert**
  There are two different versions of the insert member function. The first version inserts the entry into the set and returns a pair. The first component of the returned pair refers to the location in the set containing the entry. The second component is true if the entry wasn't already in the set and therefore was inserted. It is false otherwise. The second version also inserts the key if it is not already there. The iterator pos is a "hint" as to where to put it. This makes the insert faster if the hint is good.

```
pair<iterator,bool> set<Key>::insert(const Key& entry);
iterator set<Key>::insert(iterator pos, const Key& entry);
```

- The **find function** returns the end iterator if the key is not in the set:

```
const_iterator set<Key>::find(const Key& x) const;
```

- The keys are constant. This means you **can't change** a key while it is in the set. You must remove it, change it, and then reinsert it.
- **Set erase**
  There are three versions of erase. The first erase returns the number of entries removed (either 0 or 1). The second and third erase functions are just like the corresponding erase functions for maps. Note that the erase functions do not return iterators. This is different from the vector and list erase functions.

```
size_type set<Key>::erase(const Key& x);
void set<Key>::erase(iterator p);
void set<Key>::erase(iterator first, iterator last);
```

Tree<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- In-order, pre-order, and post-order are all examples of **depth-first tree traversals**.
- Although iterator increment looks expensive in the worst case for a single application of operator++, it is fairly easy to show that iterating through a tree storing n nodes requires O(n) operations overall.
- **Red-black tree**
  Each node is either red or black.
  The NULL child pointers are black.
  Both children of every red node are black. Thus, the parent of a red node must also be black.
  All paths from a particular node to a NULL child pointer contain the same number of black nodes.

Stack<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- Stacks allow access, insertion and deletion from only one end called the top
  There is no access to values in the middle of a stack.
  Stacks may be implemented efficiently in terms of vectors and lists, although vectors are preferable.
  All stack operations are O(1)

Queue<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- Queues allow insertion at one end, called the back and removal from the other end, called the front
  There is no access to values in the middle of a queue.
  Queues may be implemented efficiently in terms of a list.
  Using vectors for queues is also possible, but requires more work to get right.
  All queue operations are O(1)

Leftist Heaps<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- **The null path length** (NPL) of a tree node is the length of the shortest path to a node with 0 children or 1 child. The NPL of a leaf is 0. The NPL of a NULL pointer is -1.
- **A leftist tree** is a binary tree where at each node the null path length of the left child is greater than or equal to the null path length of the right child.
- **The right path** of a node (e.g. the root) is obtained by following right children until a NULL child is reached. In a leftist tree, the right path of a node is at least as short as any other path to a NULL child. The right child of each node has the lower null path length.
- A leftist tree with r > 0 nodes on its right path has at least 2^r – 1 nodes. This can be proven by induction on r.
- A leftist tree with n nodes has a right path length of at most log(n + 1) = O(log n) nodes.
- **A leftist heap** is a leftist tree where the value stored at any node is less than or equal to the value stored at either of its children.
- Merge requires O(log n + log m) time, where m and n are the numbers of nodes stored in the two heaps, because it works on the right path at all times.

Polymorphic List of Pointers<<<<<<<<<<<<<<<<

```
for (std::list<Polygon*>::iterator i = polygons.begin(); i!
=polygons.end(); ++i) {
  Quadrilateral *q = dynamic_cast<Quadrilateral*> (*i);
  if (q) std::cout << "diagonal: " <<  q->LongerDiagonal() <<
std::endl;
}
```

Inheritance<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- With **public inheritance**, the member functions and variables do not change their public, protected or private status.
- With **protected inheritance**, public members becomes protected and other members are unchanged
- With **private inheritance**, all members become private.
- Once a function is redefined it is not possible to call the base class function, unless it is explicitly called as in SavingsAccount::compound.
- Destructors for classes which have derived classes **must be marked virtual** for this chain of calls to happen.

Operator<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- If we wanted to make one of these stream operators a regular member function, it would have to be a member function of the ostream class because this is the first argument (left operand). We cannot make it a member function of the Complex class. This is why **stream operators are never member functions.**
- Stream operators are either ordinary **non-member functions** (if the operators can do their work through the public class interface) or **friend functions** (if they need non public access).

Hash Table<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- **Separate Chaining**
  This works well when the number of items stored in each list is small, e.g., an average of 1. Other data structures, such as binary search trees, may be used in place of the list, but these have even greater overhead considering the (hopefully, very small) number of items stored per bin.
- **Open Addressing**
  Slows dramatically when the table is nearly full (e.g. about 80% or higher). This is particularly problematic for linear probing.
  Fails completely when the table is full.
  Cost of computing new hash values. Might require rebuilding the table.

Garbage Collection<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- **Reference Counting**
  Attach a counter to each Node in memory.
  When a new pointer is connected to that Node, increment the counter.
  When a pointer is removed, decrement the counter.
  Any Node with counter == 0 is garbage and is available for reuse.
- **Stop and Copy**
  Split memory in half (working memory and copy memory).
  When out of working memory, stop computation and begin garbage collection.
  Place scan and free pointers at the start of the copy memory.
  Copy the root to copy memory, incrementing free. Whenever a node is copied from working memory, leave a forwarding address to its new location in copy memory in the left address slot of its old location.
  Starting at the scan pointer, process the left and right pointers of each node. Look for their locations in working memory. If the node has already been copied (i.e., it has a forwarding address), update the reference. Otherwise, copy the location (as before) and update the reference.
  Repeat until scan == free.
  Swap the roles of the working and copy memory.
- **Mark-Sweep**
  Add a mark bit to each location in memory.
  Keep a free pointer to the head of the free list.
  When memory runs out, stop computation, clear the mark bits and begin garbage collection.
  Mark
  Start at the root and follow the accessible structure (keeping a stack of where you still need to go).
  Mark every node you visit.
  Stop when you see a marked node, so you don't go into a cycle.
  Sweep
  Start at the end of memory, and build a new free list.
  If a node is unmarked, then it's garbage, so hook it into the free list by chaining the left pointers.

Garbage Collection Comparison<<<<<<<<<<<

- **Reference Counting:**
  + fast and incremental
  – can't handle cyclical data structures!
  ? requires ~33% extra memory (1 integer per node)
- **Stop & Copy:**
  – requires a long pause in program execution
  +can handle cyclical data structures!
  – requires 100% extra memory (you can only use half the memory)
  + runs fast if most of the memory is garbage (it only touches the nodes reachable from the root) + data is clustered together and memory is "de-fragmented"
- **Mark-Sweep:**
  – requires a long pause in program execution
  + can handle cyclical data structures!
  + requires ~1% extra memory (just one bit per node)
  – runs the same speed regardless of how much of memory is garbage. It must touch all nodes in the mark phase, and must link together all garbage nodes into a free list.

Smart Pointer<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

- Smart pointers **do not** alleviate the need to master pointers, basic memory allocation & dealloca- tion, copy constructors, destructors, assignment operators, and reference variables.
- With thoughtful use, smart pointers make it easier to follow the principles of RAII and make code exception safe. In the **auto_ptr** example above, if DoSomething throws an exception, the memory for object p will be properly deallocated when we leave the scope of the foo function! This is not the case with the original version.
- The STL **shared_ptr** flavor implements reference counting garbage collection.

Concurrency And Asynchronous<<<<<<<<<<

- Once one thread has acquired the mutex (locking the resource), no other thread can acquire the mutex until it has been released.

Binary Search<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
template <class T>
bool binsearch(const std::vector<T> &v, int low, int high, const T &x) {
  if (high == low) return x == v[low];
  int mid = (low+high) / 2;
  if (x <= v[mid]) return binsearch(v, low, mid, x);
  else return binsearch(v, mid+1, high, x);
}
template <class T>
bool binsearch(const std::vector<T> &v, const T &x) {
   return binsearch(v, 0, v.size()-1, x);
}
```

Merge Sort<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
using namespace std;
template <class T> void mergesort(vector<T>& values) {
  vector<T> scratch(values.size());
  mergesort(0, int(values.size()-1), values, scratch);
}
template <class T> void mergesort(int low, int high, vector<T>& values, vector<T>& scratch) {
  cout << "mergesort:  low = " << low << ", high = " << high << endl;
  if (low >= high) return; //  intervals of size 0 or 1 are already sorted!
  int  mid = (low + high) / 2;
  mergesort(low, mid, values, scratch);
  mergesort(mid+1, high, values, scratch);
  merge(low, mid, high, values, scratch); // O(n)
}
template <class T> void merge(int low, int mid, int high, vector<T>& values, vector<T>& scratch) {
  cout << "merge:  low = " << low << ", mid = " << mid << ", high = " << high << endl;
  int i=low;        // "top" of pile a  [low -> mid]
  int j = mid+1; // "top" of pile b  [mid+1 -> high]
  int k=low;      // the next slot in the sorted
  for ( ; k <= high ; k++) { // result currently in scratch
    if (i <= mid && (j > high II values[i] < values [j])) {
      scratch[k] = values[i];
      i++;
    }
    else {
      scratch[k] = values[j];
      j++;
    }
  }
  for (k=low ; k <= high ; k++) values[k] = scratch[k];
}
```

Nonlinear Word Search<<<<<<<<<<<<<<<<<<<<<<<<<

```
bool on_path(loc pos, std::vector<loc> const& path) {
  for (unsigned int i=0; i<path.size(); ++i)
    if (pos == path[i]) return true;
  return false;
}
bool search_from_loc(loc pos, const std::vector<std::string>& bd, const std::string& word, std::vector<loc>& path ) {
  path.push_back(pos);
  if (path.size() == word.size()) return true;
  for (int i = std::max(pos.row-1, 0); i < std::min(int(bd.size()), pos.row+2); ++i) {
    for (int j = std::max(pos.col-1, 0); j < std::min(int(bd[i].size()), pos.col+2); ++j) {
      if (on_path(loc(i,j), path)) continue;
      if (bd[i][j] == word[path.size()]) {
        if (search_from_loc(loc(i,j), bd, word, path)) return true;
      }
  }
  path.pop_back();
  return false;
}
```

Quick Sort<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
int quickSort(vector<double>& array, int start, int end);
int partition(vector<double>& array, int start, int end, int& swaps) {
  int mid = (start + end)/2;
  double pivot = array[mid];
}
int quickSort(vector<double>& array, int start, int end) {
  int swaps = 0;
  if(start < end) {
    int pIndex = partition(array, start, end, swaps);
    swaps += quickSort(array, start, pIndex-1);
    swaps += quickSort(array, pIndex+1, end);
  }
  return swaps;
}
```

Tree Traversal Using A Stack<<<<<<<<<<<<<<<<<

```
#include <stack>
void Preorder(Node* root) {
  stack<Node*> s;
  if (root != NULL) s.push(root);
  while (!s.empty()) {
    Node* p = s.top();
    s.pop();
    Visit(p);   // process the node
    if (p->right != NULL) s.push(p->right);
    if (p->left != NULL) s.push(p->left);
  }
}
void Inorder(Node* root) {
  stack<Node*> s;
  Node* p = root;
  while (p != NULL) {
    s.push(p);
    p = p->left;
  }
  while (!s.empty()) {
    p = s.top();
    s.pop();
    Visit(p);   // process the node
    p = p->right;
    while (p != NULL) {
      s.push(p);
      p = p->left;
    }
  }
}
```

Breadth First Traversal Using A Queue<<

```
void BreadthFirst(Node* root) {
  queue<Node*> q;
  q.push(root);
  while(! q.empty) {
    Node* n = q.front();
    Visit(n);   // process the node
    if (n->left != NULL) q.push(n->left);
    if (n->right != NULL) q.push(n->right);
    q.pop();
  }
}
```

Merge Code with Leftist Heaps<<<<<<<<<<<<<

```
template <class T>
LeftNode<T>* merge(LeftNode<T> *H1,LeftNode<T> *H2) {
  if (!h1) return h2;
  else if (!h2) return h1;
  else if (h2->value > h1->value)
    return merge_helper(h1, h2);
  else return merge_helper(h2, h1);
}
template <class T>
LeftNode<T>* merge_helper(LeftNode<T> *h1, LeftNode<T> *h2) {
  if (h1->left == NULL) h1->left = h2;
  else {
    h1->right = merge(h1->right, h2);
    if(h1->left->npl < h1->right->npl) swap(h1->left, h1->right);
    h1->npl = h1->right->npl + 1;
  }
  return h1;
}
```

Functor<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
void float_print (float f) { std::cout << f << std::endl; }
std::for_each(my_data.begin(), my_data.end(), float_print);
std::for_each(my_data.begin(), my_data.end(), [](float f){ std::cout << f
                    << std::end; });
class between_values {
private:
  float low, high;
public:
  between_values(float l, float h) : low(l), high(h) {}
  bool operator() (float val) { return low <= val && val <= high; }
};
between_values two_and_four(2,4);
if (std::find_if(my_data.begin(), my_data.end(), two_and_four) !=
                    my_data.end()) {
  std::cout << "Found a value greater than 2 & less than 4!" << std::endl;
}
std::vector<float>::iterator itr;
itr = std::find_if(my_data.begin(), my_data.end(), between_values(2,4));
if (itr != my_data.end()) {
  std::cout << "my_data contains " << *itr
  << ", a value greater than 2 & less than 4!" << std::endl;
}
```

Operator<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
Complex& Complex::operator+= (Complex const& rhs) {
  real_ += rhs.real_;
  imag_ += rhs.imag_;
  return *this;
}
bool operator==(const Complex& c1, const Complex& c2) {
  return c1.Real() == c2.Real() && c1.Imaginary() == c2.Imaginary();
}
bool Complex::operator== (Complex const& rhs) {
  return real_ == rhs.real_ &&  imag_ == rhs.imag_;
}
bool operator!=(const Complex& c1, const Complex& c2) {
  return ! (c1.Real() == c2.Real() && c1.Imaginary() == c2.Imaginary());
}
boo Complex::operator!= (Complex const& rhs) {
  return ! (real_ == rhs.real_ &&  imag_ == rhs.imag_);
}
Complex Complex::operator* (Complex const& rhs) const {
  double re = (real_ * rhs.real_) - (imag_ * rhs.imag_);
  double im = (real_ * rhs.imag_) + (imag_ * rhs.real_);
  Complex tmp(re,im);
  return tmp; //Complex(re, im);
}
bool Complex::operator< (Complex const& rhs) const {
  return Magnitude() < rhs.Magnitude();
}
```

Exception<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
int my_func(int a, int b) throw(double,bool) {
  if (a > b) throw 20.3;
  else throw false;
}
int main() {
  try my_func(1,2);
  catch (double x) std::cout << " caught a double " << x << std::endl;
  catch (…) std::cout << " caught some other type " << std::endl;
}
```

STL Exception Class<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

STL provides a base class std::exception in the <exception> header file. You can derive your own exception type from the exception class, and overwrite the what() member function

```
class myexception: public std::exception {
  virtual const char* what() const throw() {
    return "My exception happened";
  }
};
int main () {
  myexception myex;
  try throw myex;
  catch (std::exception& e) {
    std::cout << e.what() << std::endl;
  }
  return 0;
}
```

Smart Pointer<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
template <class T> class auto_ptr {
public:
  explicit auto_ptr(T* p = NULL) : ptr(p) {}
  ~auto_ptr() { delete ptr; }
  T& operator*() { return *ptr; }
  T* operator->() { return ptr; }
private:
  T* ptr;
};
void foo() {
  auto_ptr<Polygon> p(new Polygon(/* stuff */);
  p->DoSomething();
}
std::vector<shared_ptr<Polygon> > polys;
polys.push_back(shared_ptr<Polygon>(new Triangle(/*...*/)));
polys.push_back(shared_ptr<Polygon>(new Quad(/*...*/)));
polys.clear();  // cleanup is automatic!
```

Concurrency And Asynchronous<<<<<<<<<

```
class Chalkboard {
public:
  Chalkboard() { student_done = true; }
  void write(Drawing d) {
    while (1) {
      board.lock();
      if (student_done) {
        drawing = d;
        student_done = false;
        board.unlock();
        return;
      }
      board.unlock();
    }
  }
  Drawing read() {
    while (1) {
      board.lock();
      if (!student_done) {
        Drawing answer = drawing;
        student_done = true;
        board.unlock();
        return answer;
      }
      board.unlock();
    }
  }
private:
  Drawing drawing;
  std::mutex board;
  bool student_done;
};
class Professor {
public:
  Professor(Chalkboard *c) { chalkboard = c; }
  virtual void Lecture(const std::string &notes) {
    chalkboard->write(notes);
  }
protected:
  Chalkboard* chalkboard;
};
class Student {
public:
  Student(Chalkboard *c) { chalkboard = c; }
  void TakeNotes() {
    Drawing d = chalkboard->read();
    notebook.push_back(d);
  }
private:
  Chalkboard* chalkboard;
  std::vector<Drawing> notebook;
};
#define num_notes 10
void student_thread(Chalkboard *chalkboard) {
  Student student(chalkboard);
  for (int i = 0; i < num_notes; i++) {
    student.TakeNotes();
  }
}
int main() {
  Chalkboard chalkboard;
  Professor prof(&chalkboard);
  std::thread student(student_thread, &chalkboard);
  for (int i = 0; i < num_notes; i++) {
    prof.Lecture("blah blah");
  }
  student.join();
}
class CautiousLecturer : public Professor {
public:
  CautiousLecturer(Chalkboard *c) : Professor(c) {}
  void Lecture() {
    chalkboard->textbook.lock();
    Drawing d = FromBookDrawing();
    chalkboard->chalk.lock();
    Professor::Lecture(d);
    chalkboard->chalk.unlock();
    chalkboard->textbook.unlock();
  }
};
void checkDrawing(const Drawing &d) {}
class BrashLecturer : public Professor {
public:
  BrashLecturer(Chalkboard *c) : Professor(c) {}
  void Lecture() {
    chalkboard->chalk.lock();
    Drawing d = FromMemoryDrawing();
    Professor::Lecture(d);
    chalkboard->textbook.lock();
    checkDrawing(d);
    chalkboard->textbook.unlock();
    chalkboard->chalk.unlock();
  }
};
```

VECTOR<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
template <class T> class Vec {
public:
  typedef T* iterator;
  typedef const T* const_iterator;
  typedef unsigned int size_type;
  Vec() { this->create(); }
  Vec(size_type n, const T& t = T()) { this->create(n, t); }
  Vec(const Vec& v) { copy(v); }
  Vec& operator=(const Vec& v);
  ~Vec() { delete [] m_data; }
  T& operator[] (size_type i) { return m_data[i]; }
  const T& operator[] (size_type i) const { return m_data[i]; }
  void push_back(const T& t);
  iterator erase(iterator p);
  void resize(size_type n, const T& fill_in_value = T());
  void clear() { delete [] m_data;  create(); }
  bool empty() const { return m_size == 0; }
  size_type size() const { return m_size; }
  iterator begin() { return m_data; }
  const_iterator begin() const { return m_data; }
  iterator end() { return m_data + m_size; }
  const_iterator end() const { return m_data + m_size; }
private:
  void create();
  void create(size_type n, const T& val);
  void copy(const Vec<T>& v);
  T* m_data;
  size_type m_size;
  size_type m_alloc;
};
template <class T>  void Vec<T>::create() {
  m_data = NULL;
  m_size = m_alloc = 0;
}
template <class T> void Vec<T>::create(size_type n, const T& val) {
  m_data = new T[n];
  m_size = m_alloc = n;
  for (T* p = m_data; p != m_data + m_size; ++p) *p = val;
}
template <class T> Vec<T>& Vec<T>::operator=(const Vec<T>& v) {
  if (this != &v) {
    delete [] m_data;
    this -> copy(v);
  }
  return *this;
}
template <class T> void Vec<T>::copy(const Vec<T>& v) {
  this->m_alloc = v.m_alloc;
  this->m_size = v.m_size;
  this->m_data = new T[this->m_alloc];
  for (size_type i = 0; i < this->m_size; ++i)
    this -> m_data[ i ] = v.m_data[ i ];
}
template <class T> void Vec<T>::push_back(const T& val) {
  if (m_size == m_alloc) {
    m_alloc *= 2;
    if (m_alloc < 1) m_alloc = 1;
    T* new_data = new T[ m_alloc ];
    for (size_type i=0; i<m_size; ++i) new_data[i] = m_data[i];
    delete [] m_data;
    m_data = new_data;
  }
  m_data[m_size] = val;
  ++ m_size;
}
template <class T> typename Vec<T>::iterator Vec<T>::erase(iterator
p) {
  for (iterator q = p; q < m_data+m_size-1; ++q) *q = *(q+1);
  m_size --;
  return p;
}
template <class T> void Vec<T>::resize(size_type n, const T&
fill_in_value) {
  if (n <= m_size) m_size = n;
  else {
    if (n > m_alloc) {
      m_alloc = n;
      T* new_data = new T[m_alloc];
      for (size_type i=0; i<m_size; ++i) new_data[i] = m_data[i];
      delete [] m_data;
      m_data = new_data;
    }
    for (size_type i = m_size; i<n; ++i) m_data[i] = fill_in_value;
    m_size = n;
  }
}
```

LIST<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
template <class T> class Node {
public:
  Node() : next_(NULL), prev_(NULL) {}
  Node(const T& v) : value_(v), next_(NULL), prev_(NULL) {}
  T value_;
  Node<T>* next_;
  Node<T>* prev_;
};
template <class T> class dslist;
template <class T> class list_iterator {
public:
  list_iterator() : ptr_(NULL) {}
  list_iterator(Node<T>* p, std::string type, Node<T>* q) {
    ptr_ = p;
    end_ = q;
    type_ = type;
    if (type_ == "end") ptr_ = NULL;
  }
  list_iterator(const list_iterator<T>& old) : ptr_(old.ptr_),
type_(old.type_), end_(old.end_) {}
  list_iterator<T>& operator=(const list_iterator<T>& old) {
    ptr_ = old.ptr_; type_ = old.type_; end_ = old.end_; return *this; }
  ~list_iterator() {}
  T& operator*()  { return ptr_->value_;  }
  list_iterator<T>& operator++() { // pre-increment, e.g., ++iter
    if (type_ == "itr") ptr_ = ptr_->next_;
    if (ptr_ == NULL) type_ = "end";
    return *this;
  }
  list_iterator<T> operator++(int) { // post-increment, e.g., iter++
    list_iterator<T> temp(*this);
    if (type_ == "itr") ptr_ = ptr_->next_;
    if (ptr_ == NULL) type_ = "end";
    return temp;
  }
  list_iterator<T>& operator--() { // pre-decrement, e.g., --iter
    if (type_ == "itr") ptr_ = ptr_->prev_;
    if (type_ == "end") ptr_ = end_; type_ = "itr";
    return *this;
  }
  list_iterator<T> operator--(int) { // post-decrement, e.g., iter--
    list_iterator<T> temp(*this);
    if (type_ == "itr") ptr_ = ptr_->prev_;
    if (type_ == "end") ptr_ = end_; type_ = "itr";
    return temp;
  }
  friend class dslist<T>;
  bool operator==(const list_iterator<T>& r) const {
    return ptr_ == r.ptr_; }
  bool operator!=(const list_iterator<T>& r) const {
    return ptr_ != r.ptr_; }
private:
  Node<T>* ptr_;
  Node<T>* end_;
  std::string type_;
};
template <class T> class dslist {
public:
  dslist() : head_(NULL), tail_(NULL), size_(0) {}
  dslist(const dslist<T>& old) { this->copy_list(old); }
  dslist& operator= (const dslist<T>& old) {
    if (&old != this) {
      this->destroy_list();
      this->copy_list(old);
    }
    return *this;
  }
  ~dslist() { this->destroy_list(); }
  unsigned int size() const { return size_; }
  bool empty() const { return head_ == NULL; }
  void clear() { this->destroy_list(); }
  const T& front() const { return head_->value_;  }
  T& front() { return head_->value_; }
  const T& back() const { return tail_->value_; }
  T& back() { return tail_->value_; }
  void push_front(const T& v);
  void pop_front();
  void push_back(const T& v) {
    Node<T>* newp = new Node<T>(v);
    if (!tail_) head_ = tail_ = newp;
    else {
      newp->prev_ = tail_;
      tail_->next_ = newp;
      tail_ = newp;
    }
    ++size_;
  }
  void pop_back();
  typedef list_iterator<T> iterator;
  iterator erase(iterator itr) {
    --size_;
    iterator result(itr.ptr_->next_);
    if (itr.ptr_ == head_ && head_ == tail_) {
      head_ = tail_ = 0;
    }
    else if (itr.ptr_ == head_) {
      head_ = head_->next_;
      head_->prev_ = 0;
    }
    else if (itr.ptr_ == tail_) {
      tail_ = tail_->prev_;
      tail_->next_ = 0;
    }
    else {
      itr.ptr_->prev_->next_ = itr.ptr_->next_;
      itr.ptr_->next_->prev_ = itr.ptr_->prev_;
    }
    delete itr.ptr_;
    return result;
  }
  iterator insert(iterator itr, const T& v) {
    ++size_ ;
    Node<T>* p = new Node<T>(v);
    p->prev_ = itr.ptr_->prev_;
    p->next_ = itr.ptr_;
    itr.ptr_->prev_ = p;
    if (itr.ptr_ == head_) head_ = p;
    else p->prev_->next_ = p;
    return iterator(p);
  }
  iterator begin() { return iterator(head_, "itr", tail_); }
  iterator end() { return iterator(tail_, "end", tail_); }
private:
  void copy_list(const dslist<T>& old) {
    size_ = old.size_;
    if (size_ == 0) {
      head_ = tail_ = 0;
      return;
    }
    head_ = new Node<T>(old.head_->value_);
    tail_ = head_;
    Node<T>* old_p = old.head_->next_;
    while (old_p) {
      tail_->next_ = new Node<T>(old_p->value_);
      tail_->next_->prev_ = tail_;
      tail_ = tail_->next_;
      old_p = old_p->next_;
    }
  }
  void destroy_list() {
    if (head_ == NULL) return;
    while (head_ != NULL) {
      Node<T>* tmp = head_;
      head_ = head_->next_;
      delete tmp;
    }
  }
  Node<T>* head_;
  Node<T>* tail_;
  unsigned int size_;
};
```

TREE<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
template <class T> class TreeNode {
public:
  TreeNode() : left(NULL), right(NULL) {}
  TreeNode(const T& init) : value(init), left(NULL), right(NULL) {}
  T value;
  TreeNode* left;
  TreeNode* right;
};
template <class T> class ds_set;
template <class T> class tree_iterator {
public:
  tree_iterator() : ptr_(NULL) {}
  tree_iterator(TreeNode<T>* p) : ptr_(p) {}
  tree_iterator(const tree_iterator& old) : ptr_(old.ptr_) {}
  ~tree_iterator() {}
  tree_iterator& operator=(const tree_iterator& old) { ptr_ = old.ptr_;
return *this; }
  const T& operator*() const { return ptr_->value; }
  bool operator== (const tree_iterator& rgt) { return ptr_ == rgt.ptr_; }
  bool operator!= (const tree_iterator& rgt) { return ptr_ != rgt.ptr_; }
private:
  TreeNode<T>* ptr_;
};
template <class T> class ds_set {
public:
  ds_set() : root_(NULL), size_(0) {}
  ds_set(const ds_set<T>& old) : size_(old.size_) {
    root_ = this->copy_tree(old.root_); }
  ~ds_set() { this->destroy_tree(root_);  root_ = NULL; }
  ds_set& operator=(const ds_set<T>& old) {
    if (&old != this) {
      this->destroy_tree(root_);
      root_ = this->copy_tree(old.root_);
      size_ = old.size_;
    }
    return *this;
  }
  typedef tree_iterator<T> iterator;
  int size() const { return size_; }
  bool operator==(const ds_set<T>& old) const { return (old.root_ ==
this->root_); }
  iterator find(const T& key_value) { return find(key_value, root_);}
  std::pair< iterator, bool > insert(T const& key_value) { return
insert(key_value, root_); }
  int erase(T const& key_value) { return erase(key_value, root_); }
  friend std::ostream& operator<< (std::ostream& ostr, const
ds_set<T>& s) {
    s.print_in_order(ostr, s.root_);
    return ostr;
  }
  void print_as_sideways_tree(std::ostream& ostr) const {
    print_as_sideways_tree(ostr, root_, 0); }
  iterator begin() const {
    if (!root_) return iterator(NULL);
    TreeNode<T>* p = root_;
    while (p->left) p = p->left;
    return iterator(p);
  }
  iterator end() const { return iterator(NULL); }
private:
  TreeNode<T>* root_;
  int size_;
  TreeNode<T>* copy_tree(TreeNode<T>* old_root) {
    TreeNode<T>* new_root;
    if (old_root != NULL) {
        new_root = new TreeNode<T>(old_root->value);
        new_root->left = copy_tree(old_root->left);
        new_root->right = copy_tree(old_root->right);
    }
    else return NULL;
    return new_root;
  }
  void destroy_tree(TreeNode<T>* p) {
    if (p != NULL) {
      destroy_tree(p->left);
      destroy_tree(p->right);
      delete p;
      p = NULL;
      size_ = 0;
    }
  }
  iterator find(const T& key_value, TreeNode<T>* p) {
    if (!p) return iterator(NULL);
    if (p->value > key_value) return find(key_value, p->left);
    else if (p->value < key_value) return find(key_value, p->right);
    else return iterator(p);
  }
  iterator find(const T& key_value, TreeNode<T>* p) {
    if (!p) return iterator(NULL);
    while (p != NULL) {
      if (p->value == key_value) break;
      else if (p->value > key_value) p = p->left;
      else p = p->right;
    }
    return p;
  }
  std::pair<iterator,bool> insert(const T& key_value, TreeNode<T>*&
p) {
    if (!p) {
      p = new TreeNode<T>(key_value);
      this->size_++;
      return std::pair<iterator,bool>(iterator(p), true);
    }
    else if (key_value < p->value) return insert(key_value, p->left);
    else if (key_value > p->value) return insert(key_value, p->right);
    else return std::pair<iterator,bool>(iterator(p), false);
  }
  bool erase(T const& key_value, TreeNode<T>* &p) {
    if (!p) return false;
    if (p->value < key_value) return erase(key_value, p->right);
    else if (p->value > key_value) return erase(key_value, p->left);
    if (!p->left && !p->right) {
      delete p;
      p=NULL;
      this->size_--;
    }
    else if (!p->left) {
      TreeNode<T>* q = p;
      p=p->right;
      p->parent = q->parent;
      delete q;
      this->size_--;
    }
    else if (!p->right) {
      TreeNode<T>* q = p;
      p=p->left;
      assert (p->parent == q);
      p->parent = q->parent;
      delete q;
      this->size_--;
    }
    else {
      TreeNode<T>* q = p->left;
      while (q->right)
      q = q->right;
      p->value = q->value;
      bool check = erase(q->value, p->left);
      assert (check);
    }
    return true;
  }
  void print_in_order(std::ostream& ostr, const TreeNode<T>* p)
const {
    if (p) {
      print_in_order(ostr, p->left);
      ostr << p->value << "\n";
      print_in_order(ostr, p->right);
    }
  }
  void print_as_sideways_tree(std::ostream& ostr, const
TreeNode<T>* p, int depth) const {
    if (p) {
      print_as_sideways_tree(ostr, p->right, depth+1);
      for (int i=0; i<depth; ++i) ostr << "    ";
      ostr << p->value << "\n";
      print_as_sideways_tree(ostr, p->left, depth+1);
    }
  }
};
```

PRIORITY QUEUE<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
template <class T> class PriorityQueue {
 public:
  PriorityQueue() {}
  unsigned int size() const { return m_heap.size(); }
  T top() const  { return m_heap[0]; }
  bool exist(T element) const {
    typename std::map<T,int>::const_iterator itr =
locations.find(element);
    if (itr != locations.end()) return true;
    return false;
  }
  void push(T element) {
    typename std::map<T,int>::iterator itr = locations.find(element);
    if (itr != locations.end()) {
      std::cout << “ERROR! already exists" << element << std::endl;
    }
    m_heap.push_back(element);
    locations[element] = m_heap.size()-1;
    this->percolate_up(int(m_heap.size()-1));
  }
  void pop() {
    int success = locations.erase(m_heap[0]);
    m_heap[0] = m_heap.back();
    m_heap.pop_back();
    this->percolate_down(0);
  }
  void remove(T element) {
    if (exist(element)) {
      int loc = locations[element];
      locations.erase(element);
      m_heap[loc] = m_heap.back();
      locations[m_heap.back()] = loc;
      m_heap.pop_back();
      update_position(m_heap[loc]);
    }
  }
  void update_position(T element) {
    typename std::map<T,int>::iterator itr = locations.find(element);
    this->percolate_up(itr->second);
    this->percolate_down(itr->second);
}
  void print_heap(std::ostream & ostr) const {
  for (int i=0; i<(int)m_heap.size(); ++i)
    ostr << "[" << std::setw(4) << i << "] : "
           << std::setw(6) << m_heap[i]->getPriorityValue()
           << " " << *m_heap[i] << std::endl;
}
private:
  int last_non_leaf() const { return ((int)size()-1) / 2; }
  int get_parent(int i) const { assert (i > 0 && i < (int)size()); return (i-1) /
2; }
  bool has_left_child(int i) const { return (2*i)+1 < (int)size(); }
  bool has_right_child(int i) const { return (2*i)+2 < (int)size(); }
  bool has_parent(int i) const { return (i-1)/2 >= 0; }
  int get_left_child(int i) const { assert (i >= 0 && has_left_child(i));
return 2*i + 1; }
  int get_right_child(int i) const { assert (i >= 0 && has_right_child(i));
return 2*i + 2; }
  void percolate_up(int i) {
    while(i > 0) {
      if (m_heap[i]->getPriorityValue() < m_heap[get_parent(i)]-
>getPriorityValue()) {
        locations[m_heap[i]] = get_parent(i);
        locations[m_heap[get_parent(i)]] = i;
        std::swap(m_heap[i], m_heap[get_parent(i)]);
        i = get_parent(i);
      }
      else break;
    }
  }
  void percolate_down(int i) {
    while (has_left_child(i)) {
      int child = 0;
      if (has_right_child(i) && m_heap[get_right_child(i)]-
>getPriorityValue() <
        m_heap[get_left_child(i)]->getPriorityValue())
        child = get_right_child(i);
      else child = get_left_child(i);
      if (m_heap[child]->getPriorityValue() < m_heap[i]-
>getPriorityValue()) {
        locations[m_heap[child]] = i;
        locations[m_heap[i]] = child;
        std::swap(m_heap[child], m_heap[i]);
        i = child;
      }
      else break;
    }
  }
  std::vector<T> m_heap;
  std::map<T,int> locations;
};
```

HASH TABLE<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

```
template < typename KeyType, typename HashFunc >
class ds_hashset {
private:
  typedef typename std::list<KeyType>::iterator hash_list_itr;
public:
  class iterator {
  public:
    friend class ds_hashset;
  private:
    ds_hashset* m_hs;
    int m_index;
    hash_list_itr m_list_itr;
  private:
    iterator(ds_hashset * hs) : m_hs(hs), m_index(-1) {}
    iterator(ds_hashset* hs, int index, hash_list_itr loc)
      : m_hs(hs), m_index(index), m_list_itr(loc) {}
  public:
    iterator() : m_hs(0), m_index(-1)  {}
    iterator(iterator const& itr)
      : m_hs(itr.m_hs), m_index(itr.m_index), m_list_itr(itr.m_list_itr) {}
    iterator&  operator=(const iterator& old) {
      m_hs = old.m_hs;
      m_index = old.m_index;
      m_list_itr = old.m_list_itr;
      return *this;
    }
    const KeyType& operator*() const { return *m_list_itr; }
    friend bool operator== (const iterator& lft, const iterator& rgt)
    { return lft.m_hs == rgt.m_hs && lft.m_index == rgt.m_index &&
(lft.m_index == -1 || lft.m_list_itr == rgt.m_list_itr); }
    friend bool operator!= (const iterator& lft, const iterator& rgt)
    { return lft.m_hs != rgt.m_hs || lft.m_index != rgt.m_index ||
(lft.m_index != -1 && lft.m_list_itr != rgt.m_list_itr); }
    iterator& operator++() {
      this->next();
      return *this;
    }
    iterator operator++(int) {
      iterator temp(*this);
      this->next();
      return temp;
    }
    iterator & operator--() {
      this->prev();
      return *this;
    }
    iterator operator--(int) {
      iterator temp(*this);
      this->prev();
      return temp;
    }
  private:
    void next() {
      ++ m_list_itr;
      if (m_list_itr == m_hs->m_table[m_index].end()) {
        for (++m_index; m_index < int(m_hs->m_table.size()) &&
             m_hs->m_table[m_index].empty(); ++m_index) {}
        if (m_index != int(m_hs->m_table.size()))
          m_list_itr = m_hs->m_table[m_index].begin();
        else m_index = -1;
      }
    }
    void prev() {
      if (m_list_itr != m_hs->m_table[m_index].begin()) m_list_itr -- ;
      else {
        for (--m_index; m_index >= 0 && m_hs->
             m_table[m_index].empty(); --m_index) {}
        m_list_itr = m_hs->m_table[m_index].begin();
        hash_list_itr p = m_list_itr; ++p;
        for (; p != m_hs->m_table[m_index].end(); ++p, ++m_list_itr) {}
      }
    }
  };
private:
  std::vector<std::list<KeyType> > m_table;
  HashFunc m_hash;
  unsigned int m_size;
public:
  ds_hashset(unsigned int init_size = 10) : m_table(init_size), m_size(0) {}
  ds_hashset(const ds_hashset<KeyType, HashFunc>& old)
    : m_table(old.m_table), m_size(old.m_size) {}
  ~ds_hashset() {}
  ds_hashset& operator=(const ds_hashset<KeyType, HashFunc>&
old) {
    if (&old != this) {
      this->m_table = old.m_table;
      this->m_size = old.m_size;
      this->m_hash = old.m_hash;
    }
    return *this;
  }
  unsigned int size() const { return m_size; }
  std::pair< iterator, bool > insert(KeyType const& key) {
    const float LOAD_FRACTION_FOR_RESIZE = 1.25;
    if (m_size >= LOAD_FRACTION_FOR_RESIZE * m_table.size())
      this->resize_table(2*m_table.size()+1);
    unsigned int hash_value = m_hash(key);
    unsigned int index = hash_value % m_table.size();
    hash_list_itr p = std::find( m_table[index].begin(),
m_table[index].end(), key );
    if (p == m_table[index].end()) {
      m_table[index].push_front(key);
      iterator h_itr(this, index, m_table[index].begin());
      m_size ++ ;
      return std::make_pair(h_itr, true);
    }
    else {
      iterator h_itr(this, index, p);
      return std::make_pair(h_itr, false);
    }
  }
  iterator find(const KeyType& key) {
    unsigned int hash_value = m_hash(key);
    unsigned int index = hash_value % m_table.size();
    hash_list_itr p = std::find(m_table[index].begin(),
m_table[index].end(), key);
    if (p == m_table[index].end()) return this->end();
    else return iterator(this, index, p);
  }
  int erase(const KeyType& key) {
    iterator p = find(key);
    if (p == end()) return 0;
    else {
      erase(p);
      return 1;
    }
  }
  void erase(iterator p) { m_table[ p.m_index ].erase(p.m_list_itr); }
  iterator begin() {
    iterator p(this);
    for (p.m_index = 0; p.m_index<int(this->m_table.size());
         ++p.m_index) {
      if (!m_table[p.m_index].empty()) {
        hash_list_itr q = m_table[p.m_index].begin();
        p.m_list_itr = q;
        return p;
      }
    }
    p.m_index = -1;
    return p;
  }
  iterator end() {
    iterator p(this);
    p.m_index = -1;
    return p;
  }
  void print(std::ostream & ostr) {
    for (unsigned int i=0; i<m_table.size(); ++i) {
      ostr << i << ": ";
      for (hash_list_itr p = m_table[i].begin(); p != m_table[i].end(); ++p)
ostr << ' ' << *p;
      ostr << std::endl;
    }
  }
private:
  void resize_table(unsigned int new_size) {
    std::vector<std::list<KeyType> > old_table = m_table;
    m_table.clear();
    m_table.resize(new_size);
    for (unsigned int i = 0; i < old_table.size(); ++i) {
      for (hash_list_itr p = old_table[i].begin(); p!=old_table[i].end(); ++p) {
        unsigned int index = m_hash(*p) % new_size;
        m_table[index].push_front(*p);
      }
    }
  }
};
```