std::string string_name(string2); => copy constructor
std::string string_name(n, 'c') => 'c'*n string
STD string can be regarded as a vector a of chars, which can do all vector operations ([], insert, erase, etc.)
C-style string: char h[] = "hello";
C-style to STL: std::string s2(h);
STL to C-stle: char h[] = s1.str();

std::vector<double> a(100, 3.14) => 3.14 * 100 vector
std::vector<int> c(100) => 0 * 100 slots int vector
std::vector<int> c(a) => copy constructor

default sort func: from least to greatest.
customise sort function:
sort(vector.begin(), vector.end(), func);
in func: first > second: sort from greatest to smallest;

Vector implementation:

```
template <class T>
class Vec{
public:
  typedef unsigned int size_type;
  Vec() {this->create();} // default constructor;
  Vec(const Vec& v) {this->copy(v);}; // copy constructor
  Vec(const int a, const int b) {this->create(a, b);} // const
  ~Vec() {destroy();} // destructor;
  void push_back(const T& t);
  Vec& operator=(const Vec& v);
  T& operator[]   (size_type i) {return m_data[i]}; => = vector[i] = vector.operator[](7); (read-and-write function)
  const T& operator[] (size_type i) const {return m_data[i]} => (read-only get function) (const in return type refer that the return value is not allowed to modify either)
void push_back(const T& t); => as shown;
private:
  void create();
  void create(int a, int b);
  void destroy();
  T* m_data;
  size_type m_size;
  size_type m_alloc; }
```

Common seg fault:
1. Dereferencing a null pointer
2. Dereferencing an uninitialized pointer
3. access out-of-boundary memory on vector/list/…
4. writing a read-only memory.

```
template <class T>
void Vec<T>::push_back((const T& val){
        if (m_size == m_alloc){
                // copy the current array to the new &
                // size-doubled array, delete the old array.
        }
        m_data[m_size] = val;
        m_size++;}
template <class T>
void <T>::copy(const Vec<T>& v){
        // copy each slot & m_size & m_alloc;}
template <class T>
Vec<T>& Vec<T>::operator=(const Vec<T>& v){
        if (this != &v){
                this -> destroy();
                this->copy(v); }
        return *this;}
pop_back: remove the last element in the vector, size - 1. (NO return val)
best/avg/worst: O(1), (same for push_back);
erase_from_vector<unsigned int i, vector<std::string>& v){
        for (unsigned int j = i; j < v.size - 1; j++{
                v[j] = v[j+1];}
        v.pop_back();}
```

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);
```

Friend:
public:
friend Bar; (grant class Bar to have a full access)
friend Bar::some_func(); (grant private function some_function())
friend std::istream& operator >> (std::istream&, Complex& c);
(grant a certain function a full access to the current class)

Iterator:
vector<string>::const_iterator q; => can change q but cannot change the vector through q.
define iterator in a templated class:
typedef T* iterator;
iterator version:

```
erase_from_vector(std::vector<std::string> itr, vector<string> &v){
        std::vector<std::string::iterator> itr2 = itr;
        itr2++;
        while (itr2 != v.end(){
                (*itr) = (*itr2);
                itr++;
                itr2++;}
```

erase func:

```
template <class T>
typename Vec<T>::iterator Vec<T>::erase(iterator p){
        for (iterator q = p; q + 1 < m_data + m_size; ++q){
                (*q) = *(q+1);}
        m_size—;
        return p;
```

vector operates erase like above, O(n);
insert: all element after p, inclusive, will "shift" 1 backward.
v.insert(iterator p, element)
return: the pointer of the element being inserted.

LIst:
sort func: member function of list: list.sort(opt_condition); O(nLogn);

```
template <class T> void insert(Node<T>* &head, Node<T>* &pnt, const T& value){
        //create a new node, assign the value.
        //loop until find the head->pnt = pnt, change pointer pointing to the
                new node.
template <class T> Node<T>* erase(Node<T>* &head, Node<T>* &pnt){
        // consider the pop_front case
        // loop until find head->pnt = pnt, changing pointer
        // return the pointer to the erased node.
```

in doubly-linked list;

```
template <class T>
void erase(Node<T>* &p, Node<T>* &head, Node<T>* &tail){
        node<T>& prevNode = p->prev;
        node<T>& nextNode = p->next;
        if (head == p && nextNode == NULL){
                // erase the only node
        else if(head == p){
                // erase the first node and >1 elements
        else if(nextNode == NULL){
                // pop_back
        else{
                preNode->next = nextNode; nextNode->prev=prevNode;
                delete p;
        }
}
```

height of a tree:

```
unsigned int height (Node* p){
        if (!p){
                return 0;}
        return 1 + std::max(height (p->left), height (p->right));}
```

shortest path to leaf:

```
unsigned int shortest_path(Node* p){
        if (!p){
                return 0;}
        return 1 + std::min(height (p->left), height (p->right));}
```

erase from tree:
4 cases:
1. no children (leaf node): delete, remove pointer from its parent;
2. only left children: delete, merge whole left sub-tree to the current node;
3. only right children: delete, merge whole right sub-tree to the current node;

Set:
unique ordered key containers. O(logn) for access.
insert:
1. just like map
2. return a iterator to the inserted element by set.insert(pos, entry, pos=set_itr.

erase: just like map

Map:
std::map<key_type, value_type> var_name;
Map search/insert/erase: O(log(n))
- features: key in order, no duplicate, cannot change the key's val once defined. In a prototype class, have to define operator <

Pair: (std::pair), associated two members, accessed by pair.first & pair.second.
Constructors:
std::pair<int, double> p1(5, 7.5);
std::pair<int, double> p2 = std::make_pair(8, 9.5);
modify:
p1.first = p2.second; etc…

Map itr:
std::map<std::string, int>::iterator it = map.begin(); it != map.end(); it++){
access: it -> first (for key), it -> second (for value);

Find: map.find(key);
return: a iterator;
1. if the key is in the map, return an iterator to the pair in the map;
2. if the key is not in the map, return an iterator to the map.end();

Insert: map.insert(std::make_pair(key, value)); O(logn)
return: a pair: std::pair< map<key_type, value_type>::iterator, bool>
1. if the key is in the map: (not changing the map), return a iterator direct to the existing pair in the map, bool = false;
2. if the key is not in the map: (changing the map), return a iterator direct to the newly added pair, bool = true;

Erase: (3 versions)
1. erase(iterator p) => erase the (*p) pair in the map; O(1),
   1. return: an iterator point to the next pair
2. erase (iterator first, iterator last) => erase all pairs from first(inclusive) to last(exclusive), O(1)
   1. return: an iterator pointing to the next pair.
3. erase(const key_type& k) => erase the pair which key = k;
   1. return: size_type, 0 if not exist, 1 if exist and erased.

Merge sort
```
// driver function
template <class T>
void mergesort(std::vector<T>& values){
   std::vector<T> scratch(values.size());
   mergesort(0, int(values.size()-1), values, scratch);}
// recursive function
template <class T>
void mergesort(int low, int high, std::vector<T>& values, std::vector<T>&
scratch){
   std::cout << "mergesort: low = " << low << ", high = " << high << std::endl;
   if (low >= high) {return;}
   int mid = (low + high) / 2;
   mergesort(low, mid, values, scratch);
   mergesort(mid+1, high, values, scratch);
   merge(low, mid, high, values, scratch);}
// helper function of the recursive function
template <class T>
void merge(int low, int mid, int high, int value, std::vector<T> &scratch){
   int i = low;
   int j = mid + 1;
   k = low;

   // while there's still something left in one of the sorted sub-intervals:
   while (i <= mid && j <= high){
      // look at the top values, grab the smaller one, store it in the scratch
vector
      if (values[i] < values[j]){
         scratch[k] = values[i];   i++;
        }else{
         scratch[k] = values[j];   j++;}k++;}
   while (i <= mid){
      scratch[k] = values[i];i++;k++;}
   while (j <= high){
      scratch[k] = values[j];j++;k++;}
   // copy the scratch back to values
   for (l = low; l <= high; l++){
      values[l] = scratch[l];}}
```

Yanzhen Lu
DS final
TA: Kajsa
mentor: Anthony, Sean & Xujun
Prof.Jasmine P, Jidong Xiao
Partner: Stuait

Tree:
Leaf node: node that BOTH children are NULL.
Balanced Tree:
for every parent node, it has two children.
possible to create if only they have (2^n-1) nodes
number of leaf nodes: (n + 1) / 2
Balanced binary search tree: UNIQUE

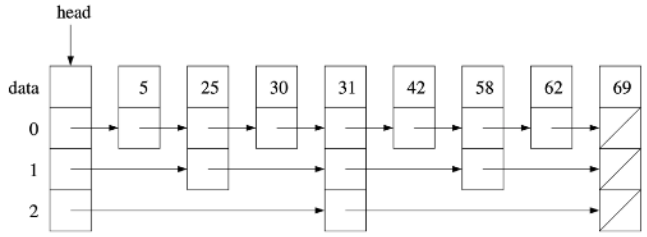
ds_set:
find smallest: all the way to the left until node->left = NULL;
operator++() : worst: O(logn), avg: O(1), best: O(1)
```
        TreeNode* curNode = ptr_;
        if (curNode -> right != NULL){
                // get the smallest node in the right subTree;
        else {
                TreeNode* curNode = ptr_;
                TreeNode* parNode = ptr -> parent;
                if (parNode == NULL){
                        ptr_ = NULL;
                        return *this;}
                while (parent -> right == current_node){
                        if (parent == NULL){
                                ptr_ = NULL;
                                return *this}
                        current_node = parent_node;
                        parent_node = current_node -> parent;}
                ptr_ = parent_node;
                return *this;}
        return *this;}
iterator find(const T& key_value, TreeNode* p){
        if (p == NULL){
                return iterator(NULL);}
        if (p -> value == key_value){
                return iterator(key_value);}
        if (key_value < p->value){
                return find(key_value, p->left)}
        if (key_value > p->value){
                return find(key_value, p->right);}}
std::pair<iterator, bool> insert(const T& key_value, TreeNode* &p){
        if (!p){
                // reached the leaf-level
                return std::make_pair<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::make_pair(iterator(p), false);}
```
In-order:
```
void inRec(treeNode<T>* root){
        if (root){
                inRec(root->left);
                std::cout << root -> value;
                inRec(root->right);}}
```
copy function:
```
TreeNode<T>* copy_tree(TreeNode<T>* old_root){
        if (old_root == NULL){
                return NULL;}
        T curVal = old_root -> val;
        TreeNode* newRoot = new TreeNode(curVal);
        newRoot -> left = copy_tree(old_root->left);
        new -> right = copy_tree(old_root -> right);
        return newRoot;
}
```
bread-first traversal: running time: O(n), memory: best(1), avg/worst: O(n)
```
void breadth_first_traverse(Node* root){
        if (root == NULL){
                return;}
        level = 0;
        std::vector<Node*> curLev;
        curLev.push_back(root);
        std::vector<Node*> nextLev;
        while (curLev.size() > 0){
                for (unsigned int i = 0; i < curLev.size(); i++){
                        if (curLev[i] -> left != NULL){
                                nextLev.push_back(curLev[i] -> left;}
                        if (curLev[i] -> right != NULL){
                                nextLev.push_back(curLev[i] ->
right;}}
                        level++;
                        curLev = nextLev;
```

Operator define:
in .h file:
bool operator<(const class_name& first);
in .cpp file:
bool operator<(const class_name& first, cons class_name second)
{ operating rule; }
Compared to non-member functions, private variable can be accessed by member function.
For "-" operator: both minus and negation operator have to define.
Return by reference: modify the variable outside the function.
Return by value: copy the variable that is created in function locally.
Stream operators: (non-member function):
(cannot defined as member, as it has defined in STL, overwriting it will produce an error.)
output stream:
define: std::ostream& operator<< (std::ostream& ostr, const Complex& c){}
use: cout << z3 = operator<<(cout, z3);
Unary operator: one parameter:
+ - * & + - * & ~ ! ++ -- -> ->*
Binary operator: two parameters:
+ - * / % ^ & | << >> += -= *= /= %= ^=
&= |= <<= >>= < <= > >= == != && || , [] () new new[] delete delete[]
All operator must return (*this) if not specified.
Order of implementation:
1) Non-member function
2) Member function
3) Friend function
Both sides protected in member function:
Complex Complex::operator+(const Complex& rhs) const{}
Both sides protected in non-member function:
Complex operator- (const Complex& lhs, const Complex& rhs) {}

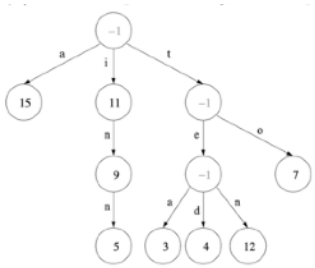
for each:
std::for_each(my_data.begin(), my_data.end(), custom_func);
in custom_func: no "()" is needed.

Variant data structure:
Unrolled linked list: Embedded a fix-length array in each node.
Its iterator should contain a pointer to the node, as well as the offset in the current node, in order to get the exact data in the sub-array.

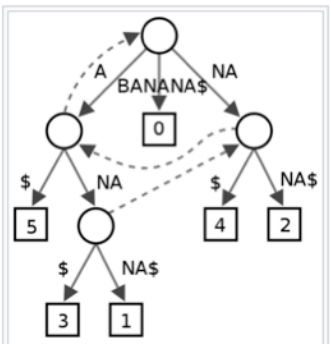
Skip list: store N pointers in the current node, pointing to the Nth node afterward.
Each level contains roughly half the nodes of the previous level, approximately every other node from the previous level.
Start from the highest level, lowering down when the target is lower than the pointer.



Trie / Prefix tree: (an alternative representation of hash table)
The stopping node of the tree is the corresponding value.



key	value
a	15
i	11
in	9
inn	5
tea	3
ted	4
ten	12
to	7



Suffix tree for the text `BANANA` . Each substring is terminated with special character `$` . The six paths from the root to the leaves (shown as boxes) correspond to the six suffixes `A$` , `NA$` , `ANA$` , `NANA$` , `ANANA$` and `BANANA$` . The numbers in the leaves give the start position of the corresponding suffix. Suffix links, drawn dashed, are used during construction.

Hash function:
```
class hash_string_obj{
public:
        unsigned int operator() (const std::string &key) const{
                // implementation of the hash function;
        }
};
```
define a hash table:
```
template <class KeyType, class HashFunc>
class ds_hashset{
        // implementations of hash_table.
private:  std::vector<std::list<KeyType> > m_table;  //actual table (use separate chaining)
        HashFunc m_hash;
        unsigned int m_size;
public:   std::pair<iterator, bool> insert(KeyType const& key){
                const float reloadSize = 1.25;
                if (m_size >= reloadSize * m_table.size()){
                        this -> resize_table( 2*m_table.size() + 1);
                unsigned int hashVal = m_hash(key);
                unsigned int index = hashVal & (this -> m_table).size();
                hash_list_itr itr;
                if (m_table[index].size() > 0{
                        itr = m_table[index].begin();
                        while (itr != (this -> m_table)[index].end()){
                                if ((*itr) == key){
                                        // have found)
                                }itr++;}}}
        // similar in find function
        void resize_table(unsigned int new_size){
                ds_hashset newSet(new_size);
                iterator itr = this -> begin();
                while (itr != this -> end()){
                        newSet.insert(*itr);
                        itr++;}
                (*this) = newSet;
```
Eliminate collision:
1, Separate chaining:
create a linked-list when there is a collision in a slot
2. Open spacing (Linear probing):
if i%N is occupied, store in (i+1)%N. or if (i+1)%N is occupied, store in (i+2)%N etc.
Finding in the hash table (top-level array):
when we reach a empty spaces after the that hash_value position, it is not exist.
3. Open spacing (Quadratic probing):
if i%N is occupied, stored in (i+1)%N, or if (i+1)%N is occupied, stored in (i+2*2)%N, (i+3*3)%N...
4. Open spacing (Secondary hashing):
Hash it again when 2 values are collided

Exception:
method 1: plan for the worst case: write if statement.
method 2: procrastination: assert().
Industrial solution:
```
try{
   if (..){
      throw std::string("....");}}
catch(std::string &error){
   string-type throw error info.}
e.g. catch from a function
int my_func(int a, int b) throw(double, bool){
   if (a>b){
      return 20.3;
   }else{
      return 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;}}
catch(int &error){
   int-type throw error info.}
catch(...){    // all types, but cannot recieve the variable.

}
..rest of the codes
```

Priority Queue:
Implementing pop: delete the root. (percolate down) worst/avg: O(logn), best:1
step 1: delete the root.
step 2: replace the root witht he right-most node in the bottom node(last leaf node).
step 3: implementing percolate down.
percolate down:

```
int curPos = 0;
while (true){
        int leftChildPos = 2 * curPos + 1;
        int rightChildPos = 2 * curPos + 2
        T leftChildVal = (this -> m_heap)[leftChildPos];
        T rightChildVal = (this -> m_heap)[rightChildPos];
        T temp = (this -> m_heap)[curPos];
        T minChild = std::min(leftChildVal, rightChildVal);
        if (temp > minChild){
                if (leftChildVal < rightChildVal){
                        std::swap((this -> m_heap)[leftChildPos], (this -> m_heap)[curPos]);
                        curPos = leftChildPos;}
                else{
                        std::swap((this -> m_heap)[rightChildPos], (this -> m_heap[curPos]);
                        curPos = rightChildPos;}
        else{ return}}
```

Implementing push: insert a new element. (percolate up) worst: O(logn), avg/ best; O(1): 50% chance as the leaf node.
percolate up:

```
(this -> m_heap).push_back(entry);
(this -> heap_size)++;
int curPos = (this -> m_heap).size() - 1;
while (curPos != 0){
        int parentPos = (curPos - 1) / 2;
        if ((this -> m_heap)[curPos] < (this -> m_heap)[parentPos]){
                std::swap((this -> m_heap)[curPos], (this -> m_heap)[parentPos]);
                curPos = parentPos;
        else{return;}}
```

Garbage collection:
Technique 1: reference counting: (fast) (doesnt handle cyclic)
Attach a counter variable to each node.
When the counter of a node become 0, add the node to the "reuse pool".

Technique 2: Stop & copy (extra memory required) (handle cyclic) (slow)
create a new memory space: copy memory, with same length as working memory.

1. place scan & free pointers at the start of the copy memory.
2. copy the root to copy memory, incrementing free. When the root is copied, leave a forward address in the left slot of the old memory.
3. Start scanning the copy memory, process left and right of each node. Check if their locations have already in copy memory. If already in, update the pointer pointing to new memory. Otherwise, beside modifying pointer in the “scanning node”, also leave a forward address in the old memory, add to the free slot and incrementing “free”.
4. stop until scan == free.
5. copy memory is now contain all the useful slot.
6.

Technique 3: Mark-sweep: (handle cyclic) (a little memory) (need to visit all memory) (slow)

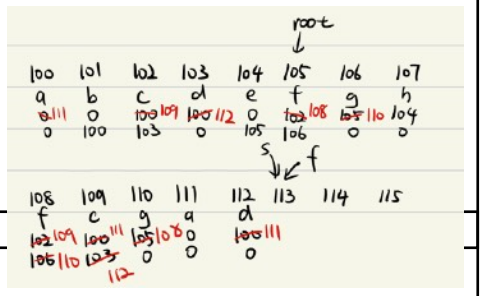
1. add a bit to each slot to mark if it has been visited.
2. From the root, as it has been visited, do nothing. Otherwise, add left and right pointer to the stack.
3. Poping each element from the stack to process each node.
4. As the stack is empty, all useful slots are marked.
5. Build a new list to join every unmarked nodes by their left pointers, start from the end (the left node of end is NULL).

Concurrency:

```
void student_thread(Chalkboard *chalkboard) {
        Student student(chalkboard);
        for (int i = 0; i < num_notes; i++){
                student.TakeNotes(); => contain chalkboard.read();}}
int main(){
        Chalkboard chalkboard;
        Professor prof(&chalkboard); =>&is important to use the same board
        std::thread student(student_thread, &chalkboard);
        for(int i = 0; i < num_notes. i++){ prof.Lecture(“aaa”) => contain chalkboard.write();
        student.join();
```

Must throw an object (non-void function) or an exception.

```
void tri(std::string &pts){
    if (pts.size() != 3){
        throw -1;
    }
}
try{
    tri(std::string("abc"));
}
catch(std::exception e){
    std::cout << e.what();
}
```



Inheritance:

```
class Account{
public:
        Account (double bal = 0.0): balance(bal) {}
        void deposit (double amt) {balance += amt; }
        double get_balance() const {return balance; }
protected: // private to all other classes but accessible for its children.
        double balance;}
class SavingsAccount: public Account{
public:
        SavingAccount(double bal = 0.0, double pct = 5.0): Account(bal), rate (pct / 100.0) {}
        double compound() {
                double interest = balance * rate;
                balance ++ interest;
                return interest;}
        double withdraw(double amt){
                if (amt > balance) { return 0.0;}
                balance -= amt;
                return amt;}}
class TimeAccount: public SavingsAccount {
public: ….
        double compound () {
                double interest = SavingsAccount::compound(); // call the function in the specific class. If not specified, call the closest one from itself to its parent, grandparent…)
```

Constructor: All parent classes will be called, from parent to children.
Destructor: Reverse sequence as the constructor.

virtual: do down the children to find a more specific redefined function.
without virtual, the closest function will be called.
Function with “virtual … (function define) … = 0” is a pure virtual function, and that class is called abstract. (Cannot create an object, but a pointer can create

```
std::list<Polygon*> polygons; // this list can all polygons & its children classes
Polygon* p_ptr = new Triangle();
```

assume: *i is a pointer to polygon
dynamic_cast<Quadrilateral*> (*i)
=> attempt to cast to quad. If not success, i = NULL

```
class Chalkboard {
public:    Chalkboard() {}
        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 ans = drawing;
                                student_done = true;
                                board.unlock();
                                return ans;}
                        board.unlock();}}}
private:
        Drawing drawing;
        std::mutex board;
        bool student_done;
```

Atomic function: process function are not interrupted, from its begin to end.