	Vector	Singled-linked list	STL list
size	√ O(1)	√ O(n)	O(1)
push_back	√ O(1)	√ O(n)	O(1)
push_front		√ O(1)	O(1)
pop_back	√ O(1)	√ O(n)	O(1)
pop_front		√ O(1)	O(1)
erase	√ O(n)	√ O(1)	O(1)
insert	√ O(n)	√ O(1)	O(1)

use of uninitialized memory	用了没有被分配的空间
mismatched new/delete/delete[]	Delete 用的不对
memory leak	没删干净
already freed memory	Delete 了已经被删过的东西
invalid write	书写不规范

get a backtrace	不知道问题在那里，去找 function
add a breakpoint	知道问题大概在哪里，在 crash 前设断点去找
use step or next	爬虫找
add a watchpoint	某一个变量在运行中变了
examine different frames of the stack	recursive function 的时候用
use Dr Memory or Valgrind to locate the leak	Slow down Memory problem
examine variable values in gdb or lldb	got an order-of-operations error or a divide-by-zero error.

```
//O(1)
void complexity(int n){
  return;
}

//O(n^2)
void complexity(int n){
  for (int i=0; i<n; i++){ //n
    for (int j=0; j<n; j++){ //n
      continue;
    }
  }
}

//O(log n)
void complexity(int n){
  if(n == 0){
    return;
  }
  complexity(n/2);
}
```

binary search：
O（log n）

merge sort:
O(n*log n)

constructor	构造方法	Stairs(int s, const T& val);
destructor	destroy	~Stairs();

merge sort：（每个 Node 一个数）

```
template <class T>
void merge(const Node<T>* list1,
     const Node<T>* list2,
     Node<T>*& merged_list) {
 if (list1 == NULL && list2 == NULL)
   return;

 if (list1 == NULL) {
   Node<T>* tmp = new Node<T>;
   tmp->value = list2->value;
   tmp->next = NULL;
   merged_list = tmp;
   merge(list1, list2->next, merged_list->next);
   return;
 }

 if (list2 == NULL) {
   Node<T>* tmp = new Node<T>;
   tmp->value = list1->value;
   tmp->next = NULL;
   merged_list = tmp;
   merge(list1->next, list2, merged_list->next);
   return;
 }

 if (list1->value <= list2->value) {
   Node<T>* tmp = new Node<T>;
   tmp->value = list1->value;
   tmp->next = NULL;
   merged_list = tmp;
   merge(list1->next, list2, merged_list->next);
 } else {
   Node<T>* tmp = new Node<T>;
   tmp->value = list2->value;
   tmp->next = NULL;
   merged_list = tmp;
   merge(list1, list2->next, merged_list->next);
 }
}
```

```
// NODE CLASS
template <class T>
class Node {
public:
  Node() : next_(NULL), prev_(NULL) {}
  Node(const T& v) : value_(v), next_(NULL), prev_(NULL) {}

  // REPRESENTATION
  T value_;
  Node<T>* next_;
  Node<T>* prev_;
};

// A "forward declaration" of this class is needed
template <class T> class dslist;
```

```
template <class T>
class dslist {
public:
  // default constructor, copy constructor, assignment operator, & destructor
  dslist() : head_(NULL), tail_(NULL), size_(0) {}
  dslist(const dslist<T>& old) { copy_list(old); }
  dslist& operator= (const dslist<T>& old);
  ~dslist() { destroy_list(); }

  typedef list_iterator<T> iterator;

  // simple accessors & modifiers
  unsigned int size() const { return size_; }
  bool empty() const { return head_ == NULL; }
  void clear() { destroy_list(); }

  // read/write access to contents
  const T& front() const { return head_->value_;  }
  T& front() { return head_->value_; }
  const T& back() const { return tail_->value_; }
  T& back() { return tail_->value_; }

  // modify the linked list structure
  void push_front(const T& v);
  void pop_front();
  void push_back(const T& v);
  void pop_back();

  iterator erase(iterator itr);
  iterator insert(iterator itr, const T& v);
  iterator begin() { return iterator(head_); }
  iterator end() { return iterator(NULL); }

private:
  // private helper functions
  void copy_list(const dslist<T>& old);
  void destroy_list();

  //REPRESENTATION
  Node<T>* head_;
  Node<T>* tail_;
  unsigned int size_;
};
```

```
template <class T>
class list_iterator {
public:
  // default constructor, copy constructor, assignment operator, & destructor
  list_iterator(Node<T>* p=NULL) : ptr_(p) {}
  // NOTE: the implicit compiler definitions of the copy constructor,
  // assignment operator, and destructor are correct for this class

  // dereferencing operator gives access to the value at the pointer
  T& operator*()  { return ptr_->value_;  }

  // increment & decrement operators
  list_iterator<T>& operator++() { // pre-increment, e.g., ++iter
    ptr_ = ptr_->next_;
    return *this;
  }
  list_iterator<T> operator++(int) { // post-increment, e.g., iter++
    list_iterator<T> temp(*this);
    ptr_ = ptr_->next_;
    return temp;
  }
  list_iterator<T>& operator--() { // pre-decrement, e.g., --iter
    ptr_ = ptr_->prev_;
    return *this;
  }
  list_iterator<T> operator--(int) { // post-decrement, e.g., iter--
    list_iterator<T> temp(*this);
    ptr_ = ptr_->prev_;
    return temp;
  }
  // the dslist class needs access to the private ptr_ member variable
  friend class dslist<T>;

  // Comparions operators are straightforward
  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:
  // REPRESENTATION
  Node<T>* ptr_;    // ptr to node in the list
};
```

Node<T>* p = new Node<T>();

#include <iostream> //reading & writing from keyboard

#include <cmath> //the square root function & absolute value

#include <string>//when use string, include this

#include <vector>//when use vector, include this

#include “.h” //the class head file

#include <fstream> //read and write file

```
std::list<int>::reverse_iterator ri;
for( ri = a.rbegin(); ri != a.rend(); ++ri )
  cout << *ri << endl;
```

读取文件

std::ifstream in_str(argv[3]);（读取）

while (in_str >> my_variable) {

// do something with my_variable

}

```
if (!in_str.good()) {
  std::cerr << "Can't open " << argv[3] << " to read.\n";
  exit(1);
}
```

```
unsigned int v_it, ret_it;
for(v_it=0,ret_it=0 ; v_it<v.size(); v_it++){
  if(i< v[v_it].size()){
    ret[i][ret_it] = v[v_it][i];
    ret_it++;
  }
}
```

word search:

```
// Simple class to record the grid location.
class loc {
public:
  loc(int r=0, int c=0) : row(r), col(c) {}
  int row, col;
};

bool operator== (const loc& lhs, const loc& rhs) {
  return lhs.row == rhs.row && lhs.col == rhs.col;
}

//  Prototype for the main search function
bool search_from_loc(loc position, const vector<string>& board, const string& word, vector<loc>& path);
```

```
bool search_from_loc(loc position, //  current position
                     const vector<string>& board,
                     const string& word,
                     vector<loc>& path)  //  path up to the current pos
{

  // DOUBLE CHECKING OUR LOGIC: the letter at the current board
  // position should equal the next letter in the word
  assert (board[position.row][position.col] == word[path.size()]);

  // start by adding this location to the path
  path.push_back(position);

  // BASE CASE: if the path length matches the word length, we're done!
  if (path.size() == word.size()) return true;

  //  search all the places you can get to in one step
  for (int i = position.row-1; i <= position.row+1; i++) {
    for (int j = position.col-1; j <= position.col+1; j++) {

      // don't walk off the board though!
      if (i < 0 || i >= int(board.size())) continue;
      if (j < 0 || j >= int(board[0].size())) continue;
      // don't consider locations already on our path
      if (on_path(loc(i,j),path)) continue;

      // if this letter matches, recurse!
      if (word[path.size()] == board[i][j]) {
        // if we find the remaining substring, we're done!
        if (search_from_loc (loc(i,j),board,word,path))
          return true;
      }
    }
  }
}
```

```
template <class T>
void dslist<T>::copy_list(const dslist<T>& old) {
  size_ = old.size_;
  // Handle the special case of an empty list.
  if (size_ == 0) {
    head_ = tail_ = 0;
    return;
  }
  // Create a new head node.
  head_ = new Node<T>(old.head_->value_);
  // tail_ will point to the last node created and therefore will move
  // down the new list as it is built
  tail_ = head_;
  // old_p will point to the next node to be copied in the old list
  Node<T>* old_p = old.head_->next_;
  // copy the remainder of the old list, one node at a time
  while (old_p) {
    tail_->next_ = new Node<T>(old_p->value_);
    tail_->next_->prev_ = tail_;
    tail_ = tail_->next_;
    old_p = old_p->next_;
  }
}
```

```
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)  //  intervals of size 0 or 1 are already sorted!
    return;

  int  mid = (low + high) / 2;
  mergesort(low, mid, values, scratch);
  mergesort(mid+1, high, values, scratch);
  merge(low, mid, high, values, scratch);
}

// Non-recursive function to merge two sorted intervals (low..mid & mid+1..high)
// of a vector, using "scratch" as temporary copying space.
template <class T>
void merge(int low, int mid, int high, vector<T>& values, vector<T>& scratch) {

  // some output so we can watch how merge sort works
  cout << "merge:  low = " << low << ", mid = " << mid << ", high = " << high << endl;
  int i=low, j=mid+1, k=low;
  // int p;
  /*
  cout << "LOW INTERVAL:     ";
  for (int p = low; p <= mid; p++)
    cout << values[p] << " ";
  cout << endl << "HIGH INTERVAL:    ";
  for (int p = mid+1; p <= high; p++)
    cout << values[p] << " ";
  cout << endl;
  */

  // while there's still something left in one of the sorted subintervals...
  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;
  }

  // Copy the remainder of the interval that hasn't been exhausted
  // Note: only one of for loops will do anything (have a non-zero index range)
  for ( ; i<=mid; ++i, ++k ) scratch[k] = values[i];  // low interval
  for ( ; j<=high; ++j, ++k ) scratch[k] = values[j]; // high interval

  // Copy from scratch back to values
  for ( i=low; i<=high; ++i ) values[i] = scratch[i];

  // observe how the interval has been sorted correctly
  /*
  cout << "SORTED INTERVAL:  ";
  for (int p = low; p <= high; p++)
    cout << values[p] << " ";
  cout << endl;
  */

}
```

```
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);
*/

//Code for exercise 8.19
if ( x < v[mid] )
return binsearch( v, low, mid-1, x );
else
return binsearch( v, mid, high, x );

}
template <class T>
bool binsearch(const std::vector<T> &v, const T &x) {
return binsearch(v, 0, v.size()-1, x);
}
```

```
template <class T>
void dslist<T>::push_back(const T& v) {
  Node<T>* newp = new Node<T>(v);
  // special case: initially empty list
  if (!tail_) {
    head_ = tail_ = newp;
  } else {
    // normal case: at least one node already
    newp->prev_ = tail_;
    tail_->next_ = newp;
    tail_ = newp;
  }
  ++size_;
}
```

```
// do these lists look the same (length & contents)?
template <class T>
bool operator== (dslist<T>& left, dslist<T>& right) {
  if (left.size() != right.size()) return false;
  typename dslist<T>::iterator left_itr = left.begin();
  typename dslist<T>::iterator right_itr = right.begin();
  // walk over both lists, looking for a mismatched value
  while (left_itr != left.end()) {
    if (*left_itr != *right_itr) return false;
    left_itr++; right_itr++;
  }
  return true;
}

template <class T>
bool operator!= (dslist<T>& left, dslist<T>& right){ return !(left==right); }
template <class T>
typename dslist<T>::iterator dslist<T>::erase(iterator itr) {
  assert (size_ > 0);
  --size_;
  iterator result(itr.ptr_->next_);
  // One node left in the list.
  if (itr.ptr_ == head_ && head_ == tail_) {
    head_ = tail_ = 0;
  }
  // Removing the head in a list with at least two nodes
  else if (itr.ptr_ == head_) {
    head_ = head_->next_;
    head_->prev_ = 0;
  }
  // Removing the tail in a list with at least two nodes
  else if (itr.ptr_ == tail_) {
    tail_ = tail_->prev_;
    tail_->next_ = 0;
  }
  // Normal remove
  else {
    itr.ptr_->prev_->next_ = itr.ptr_->next_;
    itr.ptr_->next_->prev_ = itr.ptr_->prev_;
  }
  delete itr.ptr_;
  return result;
}


template <class T>
typename dslist<T>::iterator dslist<T>::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);
}
```