CSCI-1200 Data Structures — Spring 2019
Test 2 — Thursday, February 28th 6-7:50pm

Xinhao Luo	luox6@rpi.edu lab section: 11
room: Darrin 308 zone: RED row: 3 seat: 9	6-7:50pm

Write the name of three of the undergraduate mentors for your lab section.

Anshui
Eli
Trevor

- This exam has 4 problems worth a total of 100 points (including the cover sheet).
- This packet contains 8 pages of problems numbered 1-8. Please count the pages of your exam and raise your hand if you are missing a page.
- The packet contains 2 blank pages. If you use a blank page to solve a problem, make a note in the original box and clearly label which problem you are solving on the blank page.
- This test is closed-book and closed-notes except for the .pdf notes you (optionally) uploaded to Submitty by last night. These notes are the last 2 pages of your exam packet.
- **DO NOT REMOVE THE STAPLE OR SEPARATE THE PAGES OF YOUR EXAM. DOING SO WILL RESULT IN A -10 POINT PENALTY!**
- You may have pencils, eraser, pen, tissues, water, and your RPI ID card on your desk. Place everything else on the floor under your chair. Electronic equipment, including computers, cell phones, calculators, music players, smart watches, cameras, etc. is not permitted and must be turned "off" (not just vibrate).
- Please read each question carefully. Raise your hand if you have a question.
- Please state clearly any assumptions that you made in interpreting a question. Unless otherwise stated you may use any technique that we have discussed in lecture, lab, or on the homework.
- Please write neatly. If we can't read your solution, we can't give you full credit for your work.
- You do not need to write `#include` statements for STL libraries. Writing `std::` is optional.

1 Checking It Thrice [/ 22]

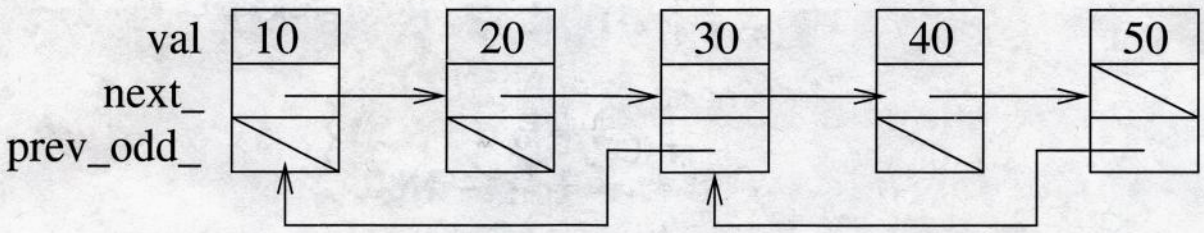
Each row in the table below contains two statements labeled A and B. Put a checkmark ☑ in up to three boxes per row. Each correct checkmark is worth +1, each blank checkbox is worth +0, and to discourage random guessing each incorrect checkmark is worth −2 points. Your score on this problem will not go below 0. Only two *A* statements are true because of their corresponding *B* statements.

A is		B is		A is true	Statemtents
True	False	True	False	**because** of B	
☑	☐	☐	☑	☐	**A**: STL vector erase() may invalidate an iterator. **B**: For programs that need to do a lot of erasing, you should use an STL list instead of an STL vector.
☑	☐	☐	☑	☐	**A**: A memory debugger like Valgrind or Dr. Memory can help find leaked memory. **B**: A memory debugger like Valgrind or Dr. Memory shows you the line you should have written delete/delete[] on to fix the memory leak.
☑	☐	☑	☐	☐	**A**: Vec push_back() on average takes O(n) **B**: Vec push_back() sometimes has to allocate a new array that's 2*m_alloc
☑	☐	☑	☐	☑	**A**: If std::list<int>::iterator *itr* points to a valid location in list *l1*, writing `l1.erase(itr)++` may cause a memory error. **B**: Incrementing the end() iterator in any STL list has undefined behavior
☑	☐	☑	☐	☐	**A**: STL lists / dslist do not have operator[] defined **B**: Using operator[] on a pointer is the same as using pointer arithmetic and then dereferencing the result.
☐	☑	☑	☐	☐	**A**: Assuming std::vector<int>::iterator *itr* points to a valid location in vector *v1*, writing `v1.insert(itr,5); std::cout << *itr;` will always cause a memory error. **B**: std::vector<T>::insert() returns an iterator because insert() may invalidate the iterator passed in.
☑	☐	☐	☑	☐	**A**: If std::vector<int> *v1* is an empty vector, `v1.end()--` will result in undefined behavior. **B**: Decrementing the end() iterator in any STL vector has undefined behavior.
☐	☑	☑	☐	☐	**A**: If a recursive function does not use array indexes or pointers, and there's a segmentation fault in that function, it means you must have infinite recursion. **B**: Infinite recursion can result in a segmentation fault.
☑	☐	☑	☐	☑	**A**: Writing `int* x=5;` will result a compiler error. **B**: Memory addresses are not numbers, so you cannot store a number in a pointer.
☐	☑	☑	☐	☐	**A**: Writing `dslist<int> x; x.push_back(5); dslist<int> y = x;` will not result in calling `y.operator=(x)` even though there is an = sign in the last statement. **B**: `operator=` can only be called on objects that are already constructed.

2 Hop Lists [/ 37]

In this problem we will consider "hop lists", a made-up data structure based on "skip lists". A hop list is a linked list using the Node class listed below. For simplicity we will not make it templated and our Nodes will only contain integers. The *next_* pointer works the same way as in the singly and doubly-linked lists we have studied so far. However, the *prev_odd_* pointer is `NULL` for the 1^{st} element in the list, and for all other elements in an odd position (3^{rd} in the list, 5^{th} in the list, and so on), the *prev_odd_* pointer points two elements back. There is a diagram below to illustrate how this would look for a list with five nodes containing the values `10, 20, 30, 40, 50` in that order.

```
class Node{
public:
  Node(int v) : val(v), next_(NULL), prev_odd_(NULL) {}
  Node(int v, Node* next) : val(v), next_(next), prev_odd_(NULL) {}
  int val;
  Node* next_;
  Node* prev_odd_;
};
```



In the following sections you will write two different versions of `AddPrevOddPointers`. Both versions take a Node pointer called *head* and a Node pointer called *tail*. After the function is done, all of the *prev_odd_* pointers should be set correctly, and *tail* should point to the last element (furtherst from *head*) that is in an odd position. If there are not at least 3 Nodes in the list, then *tail* should be `NULL`.

Here are two examples:

```
Printing forward list: 10 20 30 40 50 60 70 80 90 100 110
Value at tail: 110
Printing backwards every-other list: 110 90 70 50 30 10

Printing forward list: 31 32 33 34 35 36 37 38 39 301
Value at tail: 39
Printing backwards every-other list: 39 37 35 33 31
```

2.1 `AddPrevOddPointers` (Recursive Version) [/ 18]

Write a version of `AddPrevOddPointers` that uses recursion.

```
* Add... Function is On the back Page, sorry!
this is the helper function.
void recursive(Node * head, Node *& tail, bool & flag){
  if (head == NULL) {
    tail = NULL;
  } else {
    Node * next = head->next_;
    if (next == NULL) {
      if (!flag){
        tail = NULL;
      }
    } else {
      Node * odd = next->next_;
      if (odd == NULL){
        if (!flag) {
          tail = NULL;
        }
      } else {
        odd->prev_odd = head;
        tail = odd;
        AddPrevOddPointers(odd, tail);
        flag = true;
      }
    }
  }
```

sample solution: 23 line(s) of code

2.2 `AddPrevOddPointers` (Iterative Version) [/ 15]

Write a version of `AddPrevOddPointers` that does not use any recursion.

```
void AddPrevOddPointers(Node * head,
                        Node * &tail) {
  tail = NULL;
  Node * s = head;
  while (s != NULL) {
    Node * next = head->next_;
    if (next == NULL) {
      break;
    } else {
      Node * odd = next->next_;
      if (odd == NULL) {
        break;
      } else {
        odd->prev_odd_ = s;
        tail = odd;
        s = odd;
      }
    }
  }
}
```

sample solution: 20 line(s) of code

2.3 `AddPrevOddPointers` Complexity [/ 4]

If the hop list has n elements, what is the running time order notation of the iterative version of `AddPrevOddPointers`? What about for the recursive version?

iterative: $O(n)$

recursive: $O(n)$

3 Shape Overlays [/ 14]

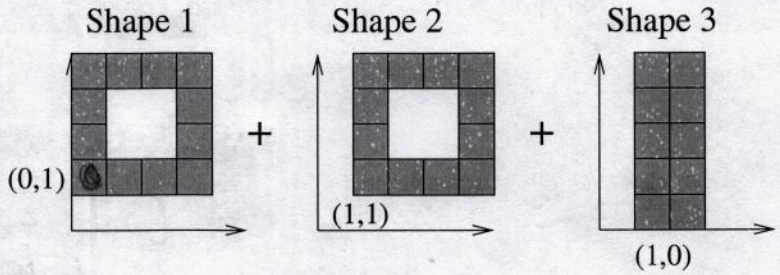
Given a series of Shape objects in a vector<Shape> *shapes*, the code below should allocate and fill a 2D dynamic array of integers, *overlay* that represents an overlay of all the shapes. If all of the shapes were to be put on top of each other, an overlay would show for every position (x,y) how many shapes had a pixel on at that position. You can assume that *shapes* is not empty and each shape is at least 1x1 in size. The coordinates use the Cartesian coordinate system, where (0,0) is the origin, $x = 1$ is to the right of the origin, and $y = 1$ is above the origin. You can also assume non-negative x,y coordinates for everything and that the overlay starts with the bottom left corner at (0,0). The example shapes and overlay output have (0,0) in the bottom left corner for easy visualization.

```
Shape 1 (outline rectangle)
Printing 4 x 4 overlay:
XXXX
X..X
X..X
XXXX

Lower left is at (0, 1)
```

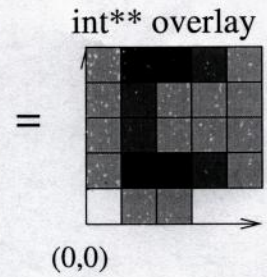
```
Shape 2 (outline rectangle)
Printing 4 x 4 overlay:
XXXX
X..X
X..X
XXXX

Lower left is at (1, 1)
```

```
Shape 3 (filled in rectangle)
Printing 5 x 2 overlay:
XX
XX
XX
XX
XX

Lower left is at (1, 0)
```

```
Printing 5 x 5 overlay:
13321
12111
12111
13321
01100
```




```
class Point{
public:
  Point(int x=0, int y=0) : m_x(x), m_y(y) {}
  int m_x, m_y;
};

class Shape{
public:
  bool pixelOn(int x, int y) const;
  void getDimensions(int& width, int& height, Point& corner) const;
...
};
```




```
void MakeOverlay(int**& overlay, int& height, const std::vector<Shape>& shapes){
  int shape_height, shape_width, width;
  height = width = 0; //Start by assuming a 0x0 overlay
  Point shape_corner; //Holds the lower left corner of the shape

  //Cycle through each shape and find the further-upper-right point
  for(unsigned int i=0; i<shapes.size(); i++){
    shapes[i].getDimensions(shape_width, shape_height, shape_corner);
    height = std::max(height, shape_corner.m_y + shape_height);
    width = std::max(width, shape_corner.m_x + shape_width);
  }

  //STUDENT PORTION #1: ALLOCATE OVERLAY, first box on next page goes here

  //STUDENT PORTION #2: FILL IN OVERLAY, second box on next page goes here

  PrintOverlay(overlay, width, height); //DO NOT WRITE THIS FUNCTION
}
```

Xinhao Luo luox6@rpi.edu

3.1 #1: Allocate Overlay [/ 8]

Write the code that should go under the "STUDENT PORTION #1" comment in `MakeOverlay`.

```
overlay = new int * [height];
for (unsigned int i=0; i <= height; i++) {
    overlay[i] = new int[width];
    for (int j=0; j <= width; j++){
        overlay[i][j] = 0;
    }
}
```

sample solution: 7 line(s) of code

3.2 #2: Fill In Overlay [/ 6]

Write the code that should go under the "STUDENT PORTION #2" comment in `MakeOverlay`.

```
for (unsigned int i=0; i < shapes.size(); i++){
    int s_h, s_w;
    Point c;
    shapes[i].getDimensions(s_w, s_h, c);
    int x = c.m_x;
    int y = c.m_y;
    for (unsigned int j=y; j <= y+s_h;
         j++ ){
        for (unsigned int k=x; k <= x+s_w
             k++ ){
            if (shapes[i].pixelOn(k,y)){
                overlay[j][k] += 1;
            }
        }
    }
}
```

sample solution: 13 line(s) of code

4 Pivoting Lists [/ 24]

Write a void function `MovePivots` that takes an STL `list<string>` *a* and a constant STL `list<string>` *pivots*. The function should rearrange *a* so that all instances of strings that are in both *a* and *pivots* are at the front of *a*. The order of other elements in *a* should not be changed, and the elements that were moved to the front should be in the order they were found in originally in *a*. The elements in *a* are not guaranteed to be unique. Do not create any new lists/vectors/arrays. Do not use anything from <algorithm>.

Here are some examples of before and after the function, with five strings in *pivots*: `ant bear cat dog eel`

```
Before List (size 3): bob ant cod
After List (size 3): ant bob cod
Before List (size 3): eel ant cat
After List (size 3): eel ant cat
Before List (size 9): bob blob cod eel cod ant eel eel ant
After List (size 9): eel ant eel eel ant bob blob cod cod
```

4.1 `MovePivots` Implementation [/ 20]

```
void MovePivots(list<string> & a, const list<string> & pivots)
{
    int size = a.size();
    int s = 0;
    list<string>::iterator e = a.end();
    e--;
    while (s < size) {
        if (flag) {
            string t = *e;
            e = a.erase(e);
            a.push_front(t);
        }
        e--;
        s++;
    }
    list<string>::const_iterator x = pivots.begin();
    bool flag = false;
    while (x != pivots.end()) {
        if (*e == *x) {
            flag = true;
            break;
        }
        x++;
    }
}
```

sample solution: 18 line(s) of code

4.2 `MovePivots` Complexity [/ 4]

If there are *p* strings in *pivots*, and *w* strings in *a*, what is the running time order notation for `MovePivots`? Explain your analysis.

O(p·w) for each string in a, you may need to run w times comparisons in string list pivots.

Xinhao Luo luox6@rpi.edu

*Q2.1, main function here. ↓

(blank page)

```
void AddPrevOddPointers(Node* head, Node* &tail) {
    bool t = false;
    recursive(head, tail, t);
}
```

(blank page)

Xinhao Luo luox6@rpi.edu

```
 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);
}


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 dslist<T>::destroy_list() {
 while(head_ != NULL) {
  Node<T>* h = head_;
  head_ = head_->next_;
  delete h;
  size_--;
 }
}
```

DEBUG

A) get a backtrace	E) examine different frames of the stack
B) add a breakpoint	~~F) reboot your computer~~
C) use step or next	G) use Dr Memory or Valgrind to locate the leak
D) add a watchpoint	H) examine variable values in gdb or lldb

A) I'm unsure where the program is crashing.
B) Once you've found the general area of the problem, it can be helpful to add a breakpoint shortly before the crash, so you can examine the situation more closely.
C) Once you've decided the state of the program is reasonable, you can advance the program one line at a time using next or step into a helper function that may be causing problems.
D) I'm implementing software for a bank, and the value of a customer's bank account is changing in the middle of the month. Interest is only supposed to be added at the end of the month.
E) A complex recursive function seems to be entering an infinite loop, despite what I think are perfect base cases.
G) The program always gets the right answer, but when I test it with a complex input dataset that takes a long time to process, my whole computer slows down.
H) I've got some tricky math formulas and I suspect I've got an order-of-operations error or a divide-by-zero error.

TA: YingYi
Mentors:

Anshui
Eli
Trevor
Geogre

Text

```
// We split the vector in half, recursively sort each half, and
// merge the two sorted halves into a single sorted interval.
template <class T>
void mergesort(int low, int high, vector<T>& values, vector<T>& scratch) {
  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);
}


// 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) {
  int i=low, j=mid+1, k=low;

  // 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
  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];

}
```

Solution:

```
template <class T>
int erase_middles(std::list<T>& data, const T& val) {
  bool found_first = false;
  typename std::list<T>::iterator prev = data.end();
  typename std::list<T>::iterator itr = data.begin();
  int count = 0;
  while (itr != data.end()) {
    if (*itr == val) {
      if (prev != data.end()) {
        data.erase(prev);
        count++;
      }
      if (found_first == false) {
        found_first = true;
      } else {
        prev = itr;
      }
    }
    itr++;
  }
  return count;
}
```

BIG-O NOTATION

O(1) - a.k.a. CONSTANT: The number of operations is independent of the size of the problem. e.g., compute quadratic root.
O(log n) - a.k.a. LOGARITHMIC. e.g., dictionary lookup, binary search.
O(n) - a.k.a. LINEAR. e.g., sum up a list.
O(n log n) - e.g., sorting.
O(n^2), O(n^3), O(n^k) - a.k.a. POLYNOMIAL. e.g., find closest pair of points.
O(2^n), O(k^n) - a.k.a. EXPONENTIAL. e.g., Fibonacci, playing chess.

DS-LIST

```
// ------------------------------------------------------------------------
// 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_;
};
template <class T> class dslist;
// ------------------------------------------------------------------------
// LIST ITERATOR
template <class T>
class list_iterator {
public:
  // default constructor, copy constructor, assignment operator, & destructor
  list_iterator(Node<T>* p=NULL) : ptr_(p) {}

  // 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
};

// ------------------------------------------------------------------------
// LIST CLASS DECLARATION
// Note that it explicitly maintains the size of the list.
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_;
};
// ------------------------------------------------------------------------
// LIST CLASS IMPLEMENTATION
template <class T>
dslist<T>& dslist<T>::operator= (const dslist<T>& old) {
  // check for self-assignment
  if (&old != this) {
    destroy_list();
    copy_list(old);
  }
  return *this;
}

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


template <class T>
void dslist<T>::pop_front() {
  if(head_==tail_){
    destroy_list();
    size_ = 0;
  }else{
    Node<T>* x = head_;
    head_ = head_->next_;
    delete x;
    size_--;
    head_->prev_=NULL;
  }
}

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_;
}

template <class T>
void dslist<T>::pop_back() {
  if(head_==tail_){
    destroy_list();
    size_ = 0;
  }else{
    Node<T>* x = tail_;
    tail_ = tail_->prev_;
    delete x;
    size_--;
    tail_->next_=NULL;
  }
}

// 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_;
```