Constructors:

```
Storage(const Storage &s);
Storage& operator=(const Storage &s);
~Storage();
```

```
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; }
```

```
typedef list<Order> OrderList;
typedef list <Item> KitchenList;
```

```
for (KitchenList::iterator i = food_cooking.begin(); i != food_cooking.end(); i++)
```

Erase ->next available value Insert->value inserted

```
Storage::~Storage() {
  for (int w = 0; w < width; w++) {
    for (int d = 0; d < depth; d++) {
      for (int h = 0; h < height; h++) {
        if (data[w][d][h] != NULL) {
          remove(data[w][d][h],w,d,h);
        }
      }
      delete [] data[w][d];
    }
    delete [] data[w];
  }
  delete [] data;
}
```

Solution:

```
void Storage::remove(Box *b, int w, int d, int h) {
  for (int i = w; i < w+b->width; i++) {
    for (int j = d; j < d+b->depth; j++) {
      for (int k = h; k < h+b->height; k++) {
        assert (data[i][j][k] == b);
        data[i][j][k] = NULL;
      }
    }
  }
  delete b;
}
```

destructor

```
template <class T> Node<T>* FindSumStart(Node<T>* n) {
  if (n == NULL) {
    return NULL;
  }
  int total = 0;
  Node<T>* tmp = n;
  while (tmp != NULL) {
    if (total == tmp->value) {
      return n;
    }
    total += tmp->value;
    tmp = tmp->next;
  }
  return FindSumStart(n->next);
}
```

recursive method

Solution:

```
const Lamp& Lamp::operator=(const Lamp &l) {
  if (this != &l) {
    for (int i = 0; i < max_bulbs; i++) {
      if (installed[i] != NULL) {
        delete installed[i];
      }
    }
    delete [] installed;
    max_bulbs = l.max_bulbs;
    recommended = l.recommended;
    installed = new Bulb*[max_bulbs];
    for (int i = 0; i < max_bulbs; i++) {
      if (l.installed[i] == NULL) {
        installed[i] = NULL;
      } else {
        installed[i] = new Bulb(*l.installed[i]);
      }
    }
  }
  return *this;
}
```

Overloaded =

Solution:

```
template <class T> void swap(T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}
```

Now implement `transpose`, as it would appear out:

Solution:

```
template <class T> void Grid<T>::transpose() {
  Node<T> *row = upper_left;
  while (row != NULL) {
    Node<T> *next_row = row->down;
    Node<T> *element = row;
    while (element != NULL) {
      Node<T> *next_element = element->right;
      swap(element->up,element->left);
      swap(element->right,element->down);
      element = next_element;
    }
    row = next_row;
  }
```

```
const int NUM_ELEMENTS_PER_NODE = 6;
//------------------------------------
// NODE CLASS
template <class T>
class Node
{
public:
    //CONSTRUCTORS
    Node()
    {
        next_ = NULL;
        prev_ = NULL;
        values_ = new T[NUM_ELEMENTS_PE
        num_elements = 0;
    }
    Node(const T& v)
    {
        next_ = NULL;
        prev_ = NULL;
        values_ = new T[NUM_ELEMENTS_PE
        values_[0] = v;
        num_elements = 1;
    }
    Node(T* const v, unsigned int size)
    {
        next_ = NULL;
        prev_ = NULL;
        values_ = v;
        num_elements = size;
    }
    // REPRESENTATION
    T *values_;
    Node<T>* next_;
    Node<T>* prev_;
    unsigned int num_elements;
};
```

Node class ex.

```
// NODE CLASS
template <class T>  <T> Provide sample template arguments for IntelliSense
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_;
};
```

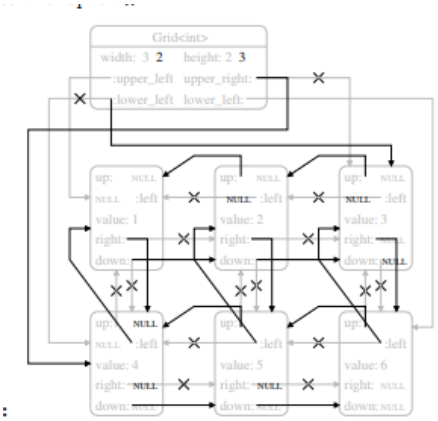
Solution: Calculating the max and min of an (unsorted) sequence of numbers requires only a linear scan/visit of the elements, comparing each element to the current min & max. n elements and $2n$ comparisons, so overall $= O(n)$. Ben's algorithm will also visit each element once, and at each recursive call it will do 2 comparisons. If we draw out the tree we see that we have n recursive calls. So the algorithms are basically equivalent in Big O Notation for performance / running time. However, function calls are expensive (more expensive than a simple loop), so in practice the running time of Ben's recursive algorithm will probably be slower (but it's not terrible).

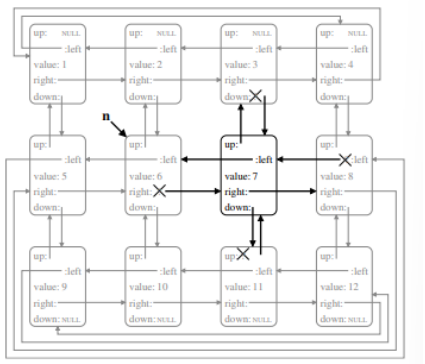
```
Interval compute_interval(const std::vector<float> &data, int i, int j) {
  // cannot compute an interval for no values
  assert (i <= j);
  if (i == j) return Interval(data[i],data[i]);
  int mid = (i+j)/2;
  Interval low = compute_interval(data,i,mid);
  Interval high = compute_interval(data,mid+1,j);
  if (low.min > high.min) low.min = high.min;
  if (low.max < high.max) low.max = high.max;
  return low;
}

Interval compute_interval(const std::vector<float> &data) {
  return compute_interval(data,0,data.size()-1);
}
```

Solution:

```
void organize_words(std::vector<std::list<std::string> > &words) {
  int count = 0;
  std::vector<std::list<std::string> >::iterator itr = words.begin();
  while (itr != words.end()) {
    std::list<std::string>::iterator itr2 = (*itr).begin();
    std::string last = "";
    while (itr2 != (*itr).end()) {
      std::string word = *itr2;
      if (word.size() != count || (last != "" && word < last)) {
        itr2 = (*itr).erase(itr2);
        place(words,word);
      } else {
        last = *itr2;
        itr2++;
      }
    }
    itr++;
    count++;
  }
}
```





2 while loops

Solution:

```
template <class T> void destroy_row(Node<T>* n) {
  if (n->right != NULL)
    destroy_row(n->right);
  delete n;
}

template <class T> void destroy_rows(Node<T>* n) {
  if (n->down != NULL)
    destroy_rows(n->down);
  n->left->right = NULL;
  destroy_row(n);
}

template <class T> void destroy_tube(Node<T>* n) {
  if (n->up != NULL)
    destroy_tube(n->up);
  else
    destroy_rows(n);
}
```

non-iterative destructor

```
warning: comparison of integers of different signs: 'int' and 'unsigned int'
```

Solution: This code is attempting to print a vector in reverse order. But it will go into an infinite loop because the condition is always true.

```
    int zero = 0;
*   for (unsigned int i = vec.size()-1; i >= zero; i--) {
      std::cout << vec[i] << std::endl;
    }
```

```
warning: control reaches / may reach end of non-void function
```

Solution: The function below does not handle the case when a == b. Essentially the return value is uninitialized in this case and could be anything.

```
    int larger_value (int a, int b) {
     if (a > b) return a;
     if (b > a) return b;
*   }
```

```
warning: variable is uninitialized when used here / in this function
```

Solution: We've forgotten to initialize the sum variable, so unfortunately all of the work is wasted. The final value of sum could be anything.

```
    int sum;
    for (int i = 0; i < vec.size(); i++) {
*     sum += vec[i];
    }
```