Practice Solution

Final exam: Friday March 21 12:15-3:15pm
Kresge Aud (in the Law School)

Problem 1: Templates and callback functions

```
template <typename Type>
  Vector<string> KeysForMaxValue(Map<Type> & map,
                            int (cmp)(Type, Type) = OperatorCmp)
{
   Map<Type>::Iterator itr = map.iterator();
   Vector<string> keys;

   string key = itr.next();   // init max to first entry
   Type maxValue = map[key];
   keys.add(key);

   while (itr.hasNext()) {    // iterate to examine other entries
      string next = itr.next();
      Type val = map[next];
      if (cmp(val, maxValue) == 0)  // if tied for max
            keys.add(next);         // add to existing set
      else if (cmp(val, maxValue) > 0) {  // if new max found
            keys.clear();                 // clear set and init to this key
            keys.add(next);
            maxValue = val;
      }
   }
   return keys;
}

int CmpBySize(Vector<char> one, Vector<char> two)
{
   return one.size() - two.size(); // cheesy but effective, subtract for -,0,+
}

string MostFrequentSeed(Map<Vector<char> > & model)
{
   Vector<string> seeds = KeysForMaxValue(model, CmpBySize);
   return seeds[RandomInteger(0, seeds.size()-1)];
}
```

Problem 2: Linked lists

```
template <typename ElemType>
      bool Set<ElemType>::contains(ElemType elem)
{
      for (cellT *cur = head; cur != NULL; cur = cur->next) {
         int sign = cmp(cur->value, elem);
         if (sign == 0) return true;
         if (sign > 0) break;
      }
      return false;
}

template <typename ElemType>
       void Set<ElemType>::unionWith(Set &otherSet)
{
   cellT *prev = NULL;
   cellT *cur = head, *other = otherSet.head;

   while (other != NULL) {
      int sign = (cur == NULL ? 1 : cmp(cur->value, other->value));
      if (sign == 0) {           // case 1: this elem == other elem
          prev = cur;
          cur = cur->next;             // advance both this and other
          other = other->next;
      } else if (sign < 0) {     // case 2: this elem < other elem
         prev = cur;
         cur = cur->next;              // advance this
      } else {                   // case 3: this elem > other elem
         cellT *newOne = new cellT;
         newOne->value = other->value;
         newOne->next = cur;           // copy elem from other to this
         if (prev != NULL) prev->next = newOne;
         else head = newOne;
         prev = newOne;
         nElements++;
         other = other->next;          // advance other
      }
   }
}
```

Problem 3: Trees

```
void Rebalance(nodeT * &t)
{
   Vector<nodeT *> v;
   FillVector(t, v);
   t = BuildTree(v, 0, v.size()-1);
}

  void FillVector(nodeT *t, Vector<nodeT *> &v)
{
   if (t != NULL) {
      FillVector(t->left, v);
      v.add(t);
      FillVector(t->right, v);
   }
}
```

```
nodeT *BuildTree(Vector<nodeT *> &v, int start, int stop)
{
   if (start > stop) return NULL;
   int mid = (start + stop)/2;
   nodeT *t = v[mid];
   t->left = BuildTree(v, start, mid-1);
   t->right = BuildTree(v, mid+1, stop);
   return t;
}
```

Problem 4: Graphs and graph algorithms

```
int CmpByNumNeighbors(node *a, node *b)
{
   return a->connectedTo.size() - b->connectedTo.size();
}

Set<node *>FindSmallDomSet(Set<node *> &allNodes)
{
   Set<node *> result, dominated;
   PQueue<node *> pq(CmpByNumNeighbors);

   Set<node *>::Iterator itr = allNodes.iterator();
   while (itr.hasNext())            // load pq with all nodes
      pq.enqueue(itr.next());

   while (dominated.size() < allNodes.size()) {
      node *cur = pq.dequeueMax();
      Set<node *> neighbors = cur->connectedTo;
      if (!dominated.contains(cur) || !neighbors.isSubsetOf(dominated)) {
        result.add(cur);
        dominated.add(cur);
        dominated.unionWith(neighbors);
      }
   }
   return result;
}
```

Problem 5: Class design

There are other possible correct answers with suitable justification of space/time/ease priorities.

a) Scenario 1 can handled with a static-sized array of Vectors. One vector per day, indexed from 0 (January 1) to 364 (December 31). The events within a day are stored in sorted order by time.

```
Vector<Event> dates[365];
```

A static-array has no overhead and gives direct access to date by index. Since most days are in use, the empty slots are not a big factor. Array is perfect, no need for fancy Vector insert/remove operations. For the day's events, a Vector is low-overhead and provides easy grow/shrink as events added/removed. Keeping it sorted is essential to run displayByDate in O(N) time.

Scenario 2 can be served by a Vector of Vectors. Only non-empty days are stored, the vector is sorted by date.

```
struct day {
    Date date;
    Vector<Event> events;     // sorted by time
};
Vector<day> days; // sorted by date
```

Finding a specific day can be done in O(lgD) time using binary search. Same reasons as above for using sorted Vector for day's events.

b) Add an index that maps from word to events containing that word in the description:

```
Map<Vector<Event *> > index;
```

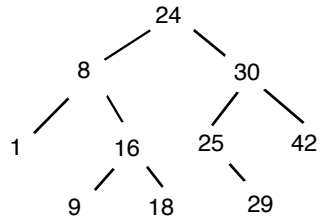
When a new event is added, the description is tokenized using Scanner. For each token, add pointer to this event into map. To list matching events, simply use word as key into table, retrieve previously stored Vector and print its contents. Map is used since it is the only structure that can provide O(1) lookup by word. Good idea to store pointers to Events in the map rather than copy the Events themselves (saves space and unifies update). Could use a Set to hold matches (avoid duplicate entries) but that adds overhead with little benefit.

c) One efficient strategy is to change data structures to store `Event*`, not `Event`, and then represent recurring event by one shared heap-allocated struct where multiple pointers point to it (ie each day for recurrent event has a pointer to the same struct). Editing that one shared copy updates all occurrences with no extra work.

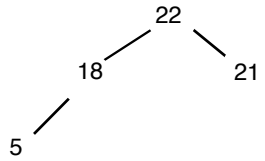
Problem 6: Short answer

a) Quicksort: better NlgN in practice and works in-place (no auxiliary storage). Mergesort: guaranteed upper bound on performance (always NlgN) and is a stable sorting algorithm.

b)



c) False. A pre-order traversal of heap shown below 22 18 5 21.



d) There are many to choose from: using a letter trie to unify word prefixes, extending to a dawg to unify suffixes, using a dynamically allocated array of children instead of a fixed 26-element array, flattening the trie into an array rather to avoid storing explicit pointers, mashing into bit fields to squish extra space, and so on.