2011 BIOSTAT 615/815 Homework #2 - Solutions

Problem 1 - Pivoting in quickSort algorithms

1. 190
2. 190
3. Should vary, but much smaller than 190
4. $190 = 1 + 2 + ... + 20$
5.

```
int pivIdx = (A[r] > A[p]) ? ( A[p] > A[m] ? p : ( A[r] > A[m] ? m : r ) )
            : ( A[r] > A[m] ? r : ( A[p] > A[m] ? m : p ) );
```

   The numbers changes to 54, 60, and some value between them

Problem 2. Radixsort Optimization

1. 10-12 radix are usually optimal
2. May vary by implementations : radixsort is usually fasters, quicksort and std::sort are similar, mergeSort is slow.
3. For std::sort, (b) is more efficient. For mergedSort and quickSort (a) is more efficient because the algorithm does not handle ties efficiently

Problem 3 - Elementary Data Structures

1. Implement `mySortedArray`, `myList`, and `myTree` data structure from lecture note. You must type them on your own. Note that, because you're using templates, all the class body needs to be included in header (.h) file and you won't need .cpp file for each class. Using the `main()` function below, submit the full set of source code to the instructor by E-mail. You may assume that all input values are unique (i.e. no need to specially handle duplicated values)
2. Use 50,000 random inputs (such as input files posted in the web page) to evaluate the running time of insertion ad search of each data structure, and report your outcome.
3.

```
#include <iostream>

template<class T>
class myDListNode {
public:
  T value;
  myDListNode<T>* prev;
  myDListNode<T>* next;

  myDListNode(const T& v, myDListNode<T>* p, myDListNode<T>* n) : value(v), prev(p), next(n) {}
  ~myDListNode() {
    if ( next != NULL) { delete next; }
  }
  int search(const T& x, int curPos) {
    if ( value == x ) {
      return curPos;
    }
    else if ( next == NULL ) {
      return -1;
    }
    else {
      return next->search(x, curPos+1);
    }
```

```
  }
  myDListNode<T>* remove(const T& x) {
    if ( value == x ) {
      if ( prev == NULL ) {
        std::cerr << "FATAL ERROR" << std::endl; // This should not happen

      }
      else {
   prev->next = next;
      }

      if ( next != NULL ) {
   next->prev = prev;
      }

      prev = next = NULL;
      return this;
    }
    else if ( next == NULL ) {
      return NULL;
    }
    else {
      return next->remove(x);
    }
  }
  template <class TT> friend class myDList;
};

template <class T>
class myDList {
 public:
  myDListNode<T>* head;
  myDList(myDList& a) {};    // prevent copying
public:      // abstract interface visible to outside
  myDList() : head(NULL) {}
  ~myDList() { if ( head != NULL ) delete head; }

  void insert(const T& x) {
    head = new myDListNode<T>(x, NULL, head);
    if ( head->next != NULL ) {
      head->next->prev = head;
    }
  }

  int search(const T& x) {
    if ( head == NULL ) {
      return -1;
    }
    else {
      return head->search(x, 0);
    }
  }

  bool remove(const T& x) {
    if ( head == NULL ) {
      return false;
    }
    else if ( head->value == x ) {
      myDListNode<T>* tmp = head;
      head = head->next;
```

```
      tmp->next = NULL;
      delete tmp;
      return true;
    }
    else {
      myDListNode<T>* p = head->remove(x);
      if ( p == NULL ) {
   return false;
      }
      else {
        delete p;
        return true;
      }
    }
  }
};
```