

CS 14: Data Structures and Algorithms

Lab 9: Binary Search Trees

Peter H. Fröhlich
phf@cs.ucr.edu

Wagner Truppel
wagner@cs.ucr.edu

Out on: March 3, 2003
Due by: March 9, 2003

Instructions: Remember to follow the instructions and policies for assignments given on the course website. This week’s assignment consists of **two** separate parts:

- an **in lab** programming exercise (1)
- an **at home** programming exercise (2)

If available, your TA **may** provide skeleton code or test drivers, but otherwise you are on your own. The “in lab” part is due at the **end** of your **lab section**. The “at home” part is due **before 8:00 pm** on the date given above. You have to use the department’s electronic turnin service for **everything**.

1 In-Lab: Sorted Lists as Sets

Your task for this week is to implement the ADT Set using linked lists. Your implementation should be in the form of a **template** class `ListSet` in a file `listset`. To make things a little more interesting, you have to support operations to access and remove the minimal and maximal elements in $O(1)$ time as well. Your implementation should provide the following `public` operations:

```
void add( const T &t );
void rem( const T &t );
bool has( const T &t ) const;
```

```
T min() const;
void rem_min();
T max() const;
void rem_max();
bool empty() const;
```

Operations that can **fail** under certain conditions should throw **appropriate** exceptions.

We recommend using the class `DoubleList` from Lab 5, but you are free to use `std::list`, or you can implement the list from scratch (if you are sure you have enough time). To achieve the required performance for `min`, `max`, `rem_min`, and `rem_max` you should keep the list in **sorted** order. Make sure your implementation constrains the type parameter `T` as **little** as possible.

2 At-Home: Basic BSTs

Your next task is to implement the **binary search tree** (BST) data structure discussed in the lecture. Your implementation should be in the form of a **template** class `BST` in a file `bstbasic`. You **can** reuse both `DoubleList` from Lab 5 and `BinaryTree` from Lab 6 for this part, but you can also start from scratch if you prefer.

Since a BST maintains a **set** of elements, it should adhere to ADT `Set` and have the following `public` operations:

```

void add( const T &t );
void rem( const T &t );
bool has( const T &t ) const;
bool empty() const;

```

In addition, you have to implement an **iterator** class to allow clients to process elements one-by-one. The BST operation

```

Iterator all() const;

```

should return an instance of `BST::Iterator`, a nested class supporting the following operations:

```

bool valid();
T get() const;
void next();

```

As for list iterators, `valid` returns `true` as long as `get` can return an element and as long as there is (potentially) a next element. The order in which elements are returned in a sequence of `next()`; `get()`; operations is **undefined**, so you can choose the order most **convenient** for your implementation. However, you must return **each** element of the BST **exactly** once.

Note that there is **no a-priori limit** on the size of a BST, so your constructor should not take any arguments and your implementation should be able to “grow and shrink” as necessary. Finally, your implementation should constrain the type parameter `T` as **little** as possible.

Hints: A simple way to implement the iterator is to traverse the tree, build an intermediate list, and iterate over **it** instead. Of course, you can also run the iterator over the tree directly, but you will probably need to do some “research” in your text book. . .

3 Optional: Balanced BSTs

This part of the assignment is **entirely optional**, and there is **no extra credit** for it either. However, if you want to make **sure** you understand how AVL-balanced BSTs work (to prepare for the final exam

maybe?), actually implementing them is one of the “easiest” ways.

Extend `bstbasic` to `bstbalanced` and implement BSTs that “automatically” maintain their balance according to the AVL approach. The goal is to keep the height of a BST with n elements bounded by $O(\log n)$ to achieve the time complexity of $O(\log n)$ for each operation.

You should add code to maintain a `int balance` field in each `Node` of the tree. When elements (and thus nodes) are added or removed, the balance might “leave” the range of $[-1, 0, 1]$, and you need to perform an appropriate sequence of **rotations** to re-establish it.