COMP9024 18x1 Data Structures and Algorithms

Week 06b Problem Set
Random Numbers, Real Balanced Trees

1. (Random numbers)

   a. A program that simulates the tosses of a coin is as follows:

```
#define NTOSSES 20
srandom(time(NULL));
int i, count = 0;
for (i=0; i<NTOSSES; i++) {
   int toss = random() % 2;     // toss = 0 or 1
   if (toss == 0) {
      putchar('H');
      count++;
   } else {
      putchar('T');
   }
}
printf("\n%d heads, %d tails\n", count, NTOSSES-count);
```

   Sample output is:

```
HHTTTTHHTTTHHTHHHHHH
12 heads, 8 tails
```

   1. What would an analogous program to simulate the repeated rolling of a die look like?
   2. What could be a sample output of this program?

   b. If you were given a string, say "hippopotamus", and you had to select a random letter, how would you do this?

   c. If you have to pick a random number between 2 numbers, say *i* and *j* (inclusive), how would you do this? (Assume *i*<*j*.)

Answer:

a. There are 6 outcomes of dice rolling: the numbers 1 to 6.
- So we'll need a fixed array *count[0..5]* of length 6 to count how many of each.
- This needs to be initialised to all zeros.

We roll the die *NROLLS* times, just like we tossed the coin.
- The outcome each time will be *roll = 1 + random()%6* .
- This is a number between 1 and 6. (*Easy to see?*)

We increment the count for this *roll*:
- *count[roll-1]++*

We conclude by printing the contents of the *count* array.

Output such as the following would be possible (for 20 rolls of the die):

```
25426251423232651155
Counts = {3,6,2,2,5,2}
```

- The list of digits shows the sequence of numbers that are rolled.
- The count shows how often each number 1, 2, …, 6 appeared.

b. We pick a random number between 0 and 11 (as there are 12 letters in the given string, and they will be stored at indices 0 .. 11), and then print that element in the character array.

```
srandom(time(NULL));             // an arbitrary seed
char *string = "hippopotamus";
int size = strlen(string);
int ran = random() % size;       // 0 ≤ ran ≤ size-1
printf("%c\n", string[ran]);
```

c. We pick a random number between 0 and *j-i*, and add that number to *i*. Note that we must compute *random()* modulo (*j-i+1*) because the number *j-i* should be included.

```
srandom(time(NULL));            // an arbitrary seed
int ran = random()%(j-i+1);     // 0 ≤ ran ≤ j-i
int num = i + ran;              // i ≤ num ≤ j
printf("%d\n", num);
```

2. (Splay trees)

a. Show how a Splay tree would be constructed if the following values were inserted into an initially empty tree in the order given:

```
5 3 8 7 4
```

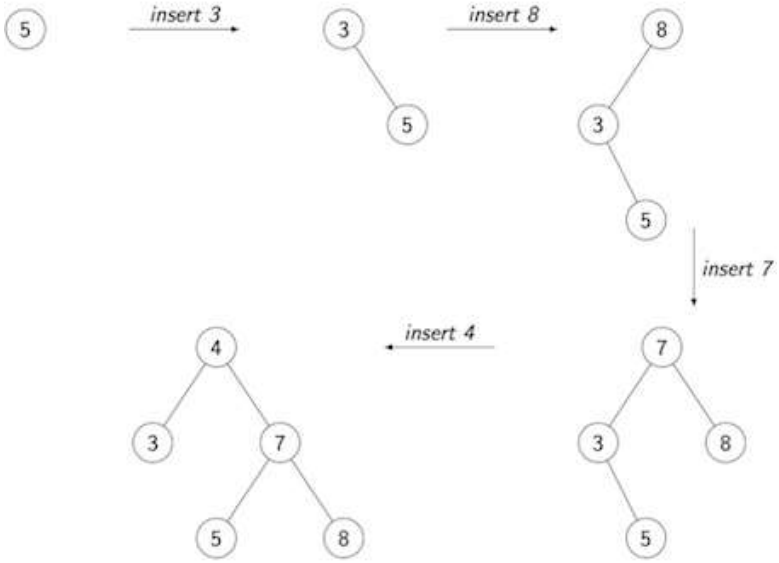
b. Let `t` be your answer to question a., and consider the following sequence of operations:

```
SearchSplay(t,7);
SearchSplay(t,3);
SearchSplay(t,8);
SearchSplay(t,6);
```

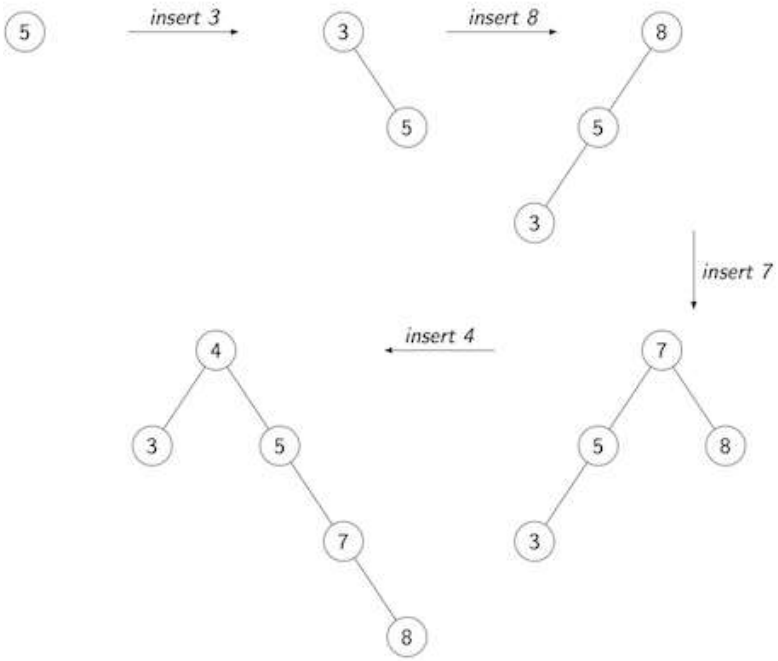
Show the tree after each operation.

Answer:

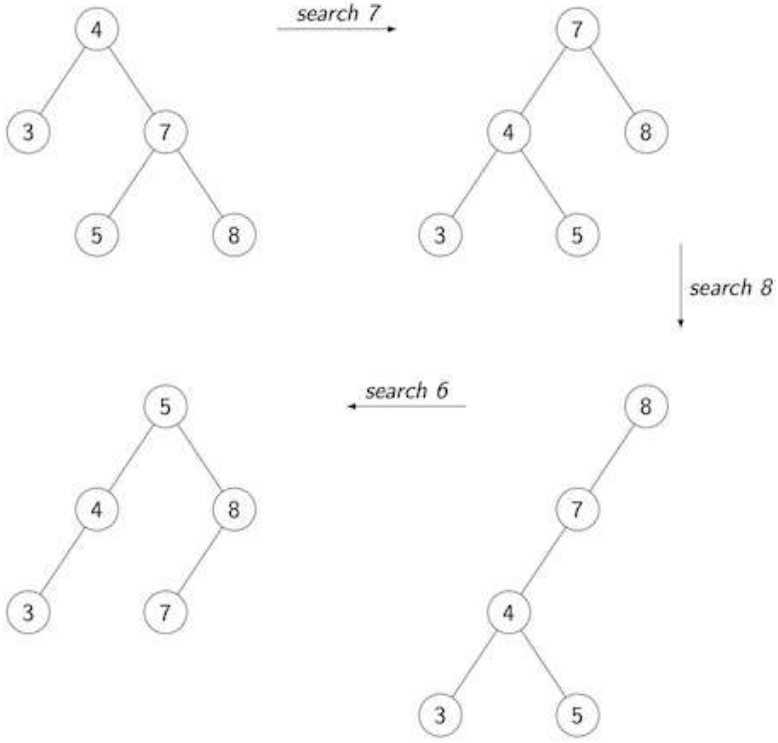
a. The following diagram shows how the tree grows based on the pseudocode given in the lecture:



This is how the tree would grow if root-root rotation were used for the left-left-child and right-right-child cases, as in the standard implementation of Splay trees:



b. The following diagram shows how the tree changes with each search operation:



3. (AVL trees)

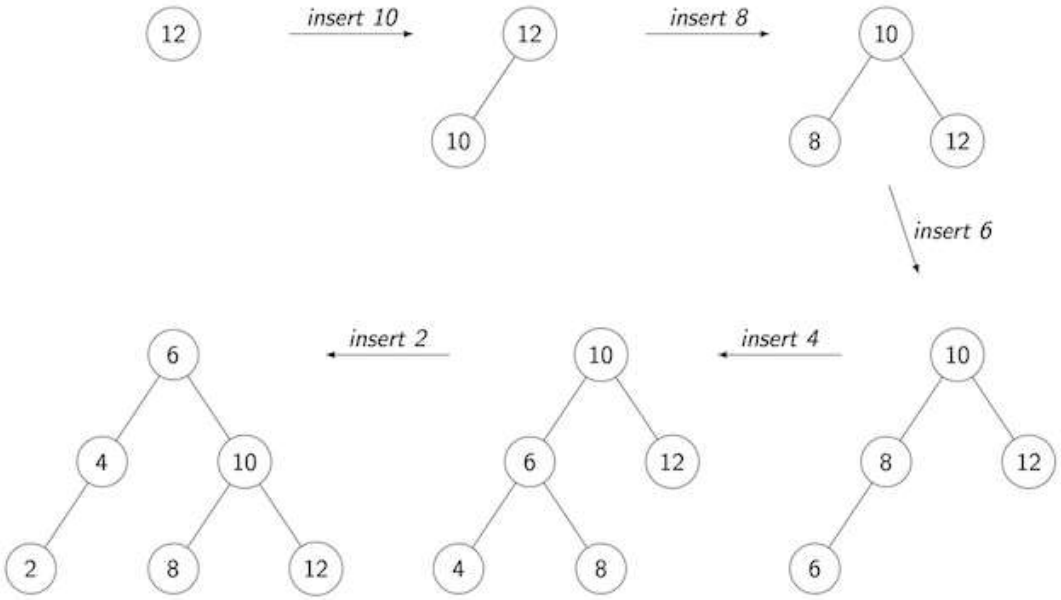
Answer the following question without the help of the `treeLab` program from the lecture.

Show how an AVL tree would be constructed if the following values were inserted into an initially empty tree in the order given:

```
12 10 8 6 4 2
```

Answer:

The following diagram shows how the tree grows:



4. (2-3-4 trees)

   Show how a 2-3-4 tree would be constructed if the following values were inserted into an initially empty tree in the order given:

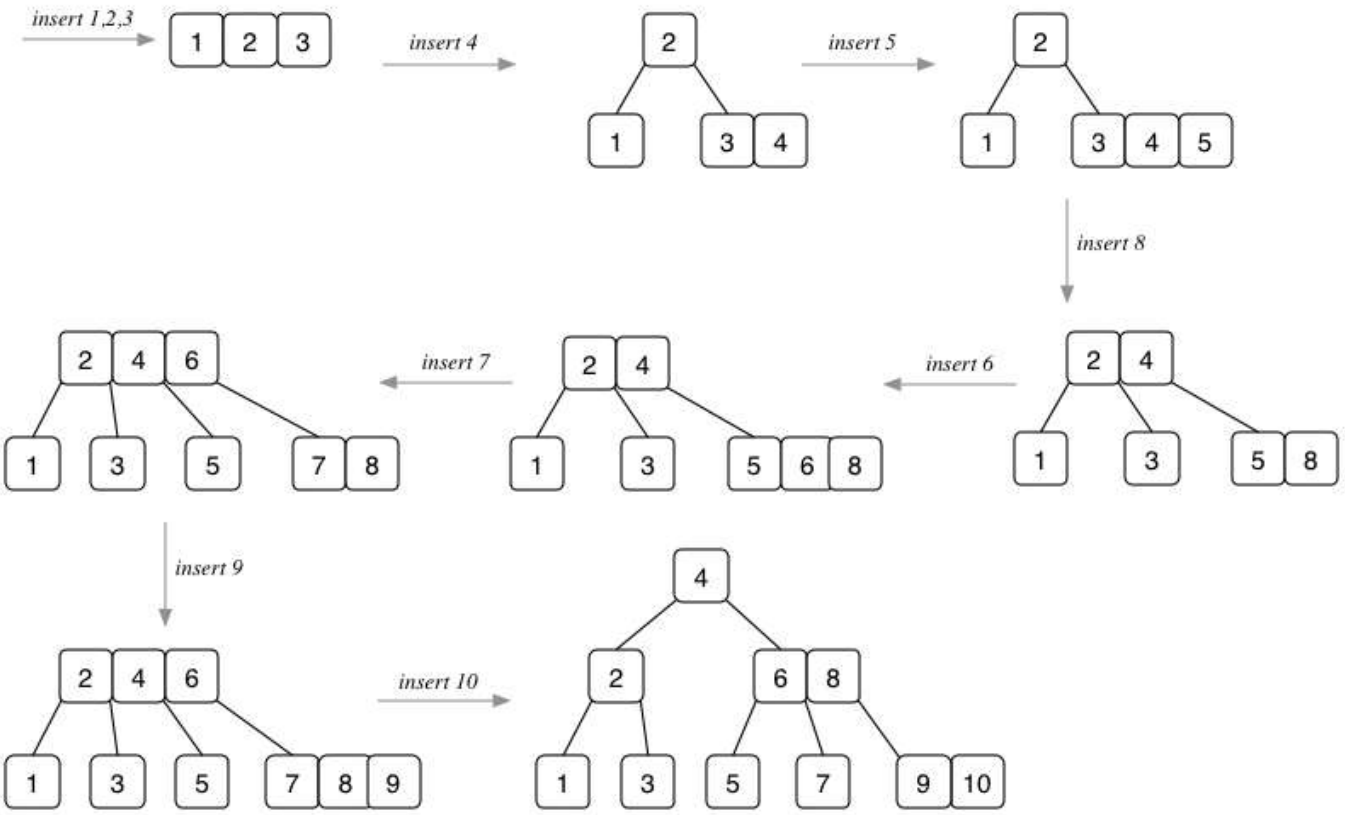
   ```
   1 2 3 4 5 8 6 7 9 10
   ```

   Once you have built the tree, count the number of comparisons needed to search for each of the following values in the tree:

   ```
   1  7  9  13
   ```

   Answer:

   The following diagram shows how the tree grows:



Search costs (for tree after insertion of 10):

- search(1): cmp(4),cmp(2),cmp(1) ⇒ cost = 3
- search(7): cmp(4),cmp(6),cmp(8),cmp(7) ⇒ cost = 4
- search(9): cmp(4),cmp(6),cmp(8),cmp(9) ⇒ cost = 4
- search(13): cmp(4),cmp(6),cmp(8),cmp(9),cmp(10) ⇒ cost = 5

5. **Challenge Exercise**

Extend the BSTree ADT from the lecture (`BSTree.h`, `BSTree.c`) by an implementation of the function

```
deleteAVL(Tree t, Item it)
```

to delete an element from an AVL tree while maintaining balance.

Answer:

```
Tree AVLrepair(Tree t) {
   int hL = TreeHeight(left(t));
   int hR = TreeHeight(right(t));
   if ((hL - hR) > 1)
      t = rotateRight(t);
   else if ((hR - hL) > 1)
      t = rotateLeft(t);
   return t;
}

Tree deleteAVL(Tree t, Item it) {
   if (t != NULL) {
      if (it < data(t)) {
         left(t) = TreeDelete(left(t), it);
         t = AVLrepair(t);
      } else if (it > data(t)) {
         right(t) = TreeDelete(right(t), it);
         t = AVLrepair(t);
      } else {
```

```
            Tree new;
            if (left(t) == NULL && right(t) == NULL)
               new = NULL;
            else if (left(t) == NULL)    // if only right subtree, make it the new root
               new = right(t);
            else if (right(t) == NULL)   // if only left subtree, make it the new root
               new = left(t);
            else {                       // left(t) != NULL and right(t) != NULL
               new = joinTrees(left(t), right(t));
               t = AVLrepair(new);
            }
            free(t);
            t = new;
         }
      }
      return t;
}
```