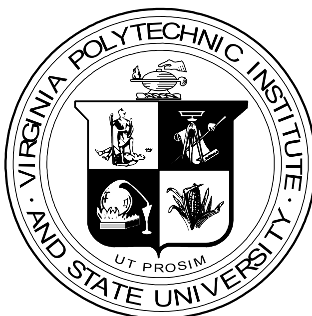


Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet. No calculators or other computing devices may be used.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 10 questions, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

Name Solution
printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

signed

1. [9 points] Determine whether each of the following statements is true or false. No justification is required.

a) **TRUE** $7n^2 + 150n + 1000$ is $\Theta(n^2)$

b) **TRUE** $7n^2 + 150n + 1000$ is $\Omega(150)$

c) **FALSE** $7n^2 + 150n + 1000$ is $O(n \log n)$

---

2. [8 points] Use the theorem about limits from the notes to either prove the statement below is true, or that it is false. (Be sure to state your conclusion.)

$$f(n) = n + 3n^2 \log n \text{ is } \Omega\left(n^3\right)$$

$$\lim_{n\to\infty} \frac{n + 3n^2 \log n}{n^3} = \lim_{n\to\infty} \frac{1 + 3n \log n}{n^2} = \lim_{n\to\infty} \frac{3\log n + 3n\frac{1}{n \ln 2}}{2n} = \lim_{n\to\infty}\left(\frac{3\log n}{2n} + \frac{3}{n \ln 2}\right)$$

$$= \lim_{n\to\infty}\left(\frac{3/n \ln 2}{2} + \frac{3}{n \ln 2}\right) = 0$$

This implies than f(n) is O(n^3), not Ω(n^3).

---

3. [10 points] A programmer must choose a data structure to store N records, which will be supplied to the program in random order, and to support random searches on those records. Suppose the programmer decides to use a skip list. As each element is received, it will be inserted into the skip list.

a) What is the expected Θ-complexity for putting all N elements into the list (assuming a "good" skip list)? Explain.

Insertion of the k-th element requires a search of the skiplist, which should average Θ(log (k-1)) if the skip list is "good". So, inserting all N of them should cost about:

$$\sum_{2}^{N} \log(k-1) \approx \log(N!)$$

which is actually Θ(N log N).

b) If $N = 2^{10}$, estimate the cost of performing 2^{20} searches on the data in the skip list? Your answer should be a number, perhaps expressed using powers of 2.

Again, if the skip list is "good", each search should cost about log (2^10), or 10. So, all of the searches should cost about 10*2^20.

4. [12 points] Assuming that each assignment, arithmetic operation, comparison, and array index costs one unit of time, analyze the complexity of the body of the following function that computes an approximation of the Euler number e, and give a simplified exact count complexity function T(N) and state its big-Θ category:

```
double Euler( int N ) {

   double Approx, Denom;                    // no cost
   Approx = 1.0;                            // 1
   Denom  = 1.0;                            // 1
   int Iter = 1;                            // 1

   while ( Iter < N ) {                     // 1 each pass, 1 more to exit

         Approx = Approx + 1.0 / Denom;     // 3 each pass
         ++Iter;                            // 1 each pass
         Denom = Denom * Iter;              // 2 each pass
   }
   return Approx;                           // 1
}
```

The complexity function would be:

$$T(N) = 3 + \sum_{Iter=1}^{N-1} (7) + 2 = 7N - 2$$

This is obviously Θ(N).

A quibble: The derivation above assumes that N is at least 1. T(0) would actually be 5.

5. [10 points] Referring to the binary tree shown below, write down the values in the order they would be visited if an enumeration was done using:

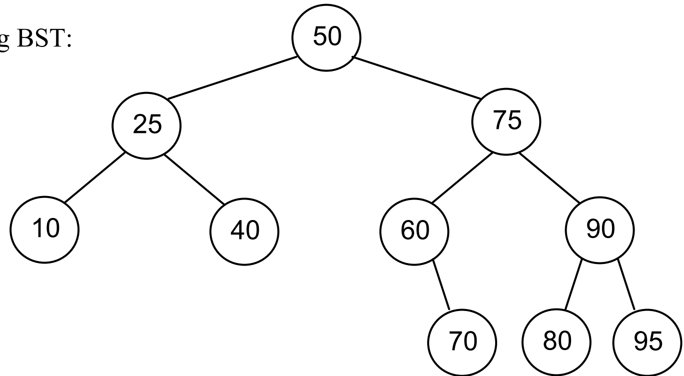
a) a preorder traversal

50 25 10 40 75 60 70 90 80 95

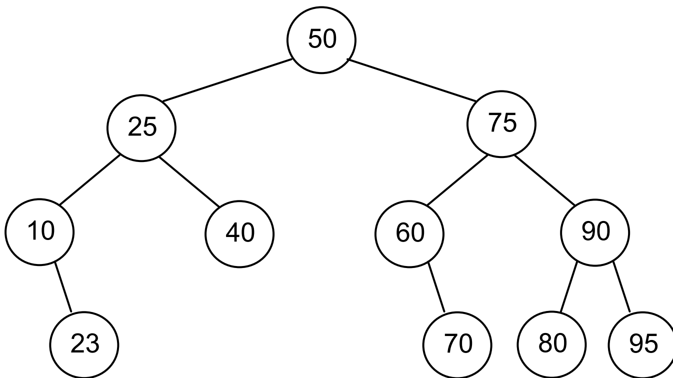
b) a postorder traversal

10 40 25 70 60 80 95 90 75 50

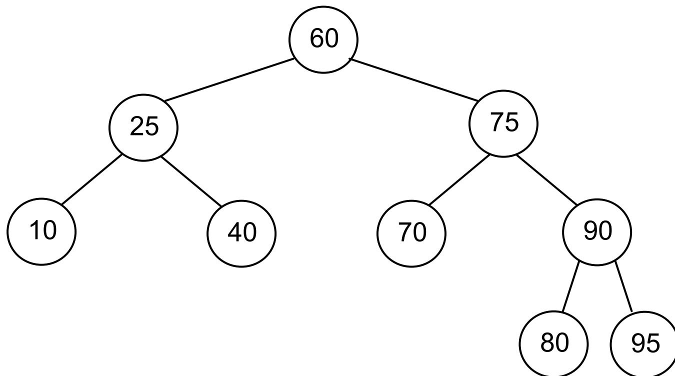
For <u>each</u> of the questions 5 and 6, <u>start</u> with the following BST:



6. [8 points] Draw the resulting BST if 23 is inserted.



7. [8 points] Draw the resulting BST if 50 is deleted.



For question 8, assume the following template declarations for an implementation of a doubly-linked list:

```
// DNodeT.h
//
. . .
template <typename T> class DNodeT {
public:
   T           Element;
   DNodeT<T>* Prev;
   DNodeT<T>* Next;
   // irrelevant members not shown
};
. . .
```

```
// DListT.h
//
. . .
template <typename T> class DListT {
private:
   DNodeT<T>* Head, Tail;    // pointers to first and last data nodes, if any
   DNodeT<T>* Fore, Aft;     // pointers to leading and trailing sentinels
public:
   // irrelevant members not shown
   iterator begin();              // return iterator to first data node (or end())
   iterator end();                //                       one-past-end
   const_iterator begin() const; // return const_iterator objects similarly
   const_iterator end() const;
   ~DListT();                     // destroy all dynamic content of the list
};
. . .
```

8. [10 points] Write an implementation of the assignment operator for the `DListT` template. You may not call any other member functions of the template, other than iterator-related ones if you wish, in your solution.

```
template <typename T> DListT<T>& DListT<T>::operator=(const DListT<T>& Source) {

   if ( this == &Source) return (*this);       // if self-copying, do nothing
   while ( Head != Aft ) {                     // delete current contents
      Fore->Next = Head->Next;
      delete Head;
      Head = Fore->Next;
   }
   Head = Tail = NULL;                          // restore to empty config
   Aft->Prev = Fore;
   DNodeT<T>* Curr = Source.Head;
   while ( Curr != Source.Aft ) {
      Aft->Prev = new DNodeT<T>(Curr->Element, Aft->Prev, Aft);
      if ( Fore->Next == Aft )
         Fore->Next = Aft->Prev;
      Curr = Curr->Next;
   }
   Head = Fore->Next;
   Tail = Aft->Prev;
   return (*this);
}
```

9. [10 points] Write an implementation of the following client function that reverses the order of the elements in a DListT object. You may call any public member functions of the template, but you should concentrate on producing an efficient solution. Be careful not to make any assumptions about the contents of the list; in particular, there is no guarantee the list will be non-empty, or that it will not contain duplicate values.

```
// Reverse() efficiently reverses the order of the elements in L.
//
template <typename T> void Reverse(DListT<T>& L) {

   if ( L.isEmpty() ) return;              // nothing to do if L is empty

   DListT<T>::iterator Front = L.begin();
   DListT<T>::iterator Back  = --L.end();

   while ( Front != Back ) {               // done if iterators meet
      T Temp = *Front;
      *Front = *Back;
      *Back  = Temp;

      Front++;
      if ( Front == Back ) return;         // done if iterators were adjacent
      Back--;
   }
}
```

For question 10, assume the following template declarations for an implementation of a binary tree:

```
template <typename T> class BinNodeT {
public:
   T             Element;
   BinNodeT<T>* Left;
   BinNodeT<T>* Right;
   // irrelevant members not shown
};
```

```
template <typename T> class BST {
protected:
   BinNodeT<T>* Root;
   // irrelevant members not shown

public:
   // irrelevant members not shown
};
```

10. [15 points] Write an implementation for the new BST member function described below. Your implementation should not need to call any other template member functions, except for any helper member functions you may wish to write.

```
// Less() uses a (modified) inorder traversal pattern, and prints all the
// values it finds that are strictly less than the parameter Max to cout.
// The function should not visit any tree nodes unnecessarily.
//
template <typename T> void BST<T>::Less(const T& Max) const {

   LessHelper(Max, Root);
}

template <typename T>
void BST<T>::LessHelper(const T& Max, BinNodeT<T>* sRoot) const {

   if ( sRoot == NULL ) return;

   LessHelper(Max, sRoot->Left);       // must always check left subtree

   if ( sRoot->Element < Max ) {
      cout << sRoot->Element << endl;
      LessHelper(Max, sRoot->Right);  // ONLY check left subtree in this case
   }
}
```