1. Print your name on your scantron in the space labeled **NAME**.
2. Print **CMSC 2613** in the space labeled **SUBJECT**.
3. Print the test number and version, **T3/V1**, in the space labeled **TEST NO.**
4. Print the date, **5-5-2017**, in the space labeled **DATE**.
5. Print your **CRN** number, **20975**, in the space labeled **PERIOD**.
6. This is a closed-book examination. No reference materials are permitted. No notes are permitted.
7. You may not consult your neighbors, colleagues, or fellow students to answer the questions on this test.
8. Cellular phones are prohibited. The possessor of a cellular phone will receive a ***zero*** (***0***) if the phone rings or is visible during the test.
9. You may use your personal calculator on this test. You are prohibited from loaning your calculator or borrowing a calculator from another person enrolled in this course.
10. Mark the best selection that satisfies the question. If selection **b** is better that selections **a** or **d**, then mark selection **b**. Mark only ***one*** selection.
11. Darken your selections completely. Make a heavy black mark that completely fills your selection.
12. Answer all **50** questions.
13. Record your answers on SCANTRON form **882-E (It is green!).**
14. When you have completed the test, place your scantron, face up, between pages 2 and 3 of your questionnaire and submit both the questionnaire and your scantron to your instructor.

1. What is the time complexity of function *Deq* for a queue whose elements are individually stored in dynamically allocated elements as shown in the Figure below?

   a. $O(n)$
   b. $O(1)$
   c. $O(n \log_2 n)$
   d. $O(\log_2 n)$

```
struct QueueException{
    QueueException(char* m)
    {   cout<<endl<<"I am the Queue and I am " << m << "." << endl;
    }
};
template <class T>
class Queue {
    struct Element {
        Element* newer;                    //Points to the next newer element
        T data;                            //Value of type T stored in this element
        Element(Element* n,T d):newer(n),data(d){}//Constructor
        Element(T t):newer(0),v(t){}       //Constructor
        Element(){}                        //Constructor
    };
    Element* oldest;                       //Points to the oldest element
    Element* newest;                       //Points to the newest element
    int count;                             //Records the number of elements
    void Kill(Element* e){...}             //Reclaims storage for all elements
public:
    Queue();                               //Constructor
    ~Queue();                              //Destructor
    bool IsEmpty(void){...}                //Is the Queue empty?
    bool IsFull(void){...}                 //Is the Queue full?
    void Enq(T v){...}                     //Insert a value, v, on the Queue
    T Deq(void){...}                       //Remove the oldest value on the Queue
    int Count(void){...}                   //Return the number of elements on the
    Queue
};
```

Figure 1. **class** *Queue*

2. What is the time complexity of function *Insert* from **class** *Heap* shown in the Figure below?

   a. $O(n \log_2 n)$
   b. $O(\log_2 n)$
   c. $O(1)$
   d. $O(n)$

```
void Insert(int v)
{   if (IsFull()) throw HeapFullException();
    int a=++count;
    while(H[a/2]>v){
        H[a]=H[a/2];
        a/=2;
    }
    H[a] = v;
}
```

Figure 2. Function *Insert*

3. What is the time complexity of member function *Sort* in **class** *Heap* as shown in the Figure below?

   a. $O(1)$
   b. $O(n \log_2 n)$
   c. $O(n)$
   d. $O(\log_2 n)$

```
void Sort(void)
{   int* J=new int[size];
    J[0]=INT_MIN;
    int a=1;
    while (!IsEmpty()) {
        int v=Remove();
        J[a++]=v;
    }
    if (H) delete[] H;
    H=J;
    count=a-1;
}
```

Figure 3. Function *Sort*

4. Which of the following is a valid implementation of function *IsFull* for a stack whose individual elements are allocated dynamically as shown in the Figure below?

   a. **bool** *IsFull*(**void**)
   ```
   {   Element* e=new Element;
       bool v=(bool)(!e);
       if (e) delete e;
       return v;
   }
   ```
   b. **bool** *IsFull*(**void**)
   ```
   {   Element* e=new Element;
       bool v=(bool)( e);
       if (e) delete e;
       return v;
   }
   ```
   c. **bool** *IsFull*(**void**){**return** *tos*!=0;}
   d. **bool** *IsFull*(**void**){**return true;**}

```
struct StackException{
    StackException(char* m)
    {   cout<<endl<<"I am the Stack and I am " << m << "." << endl;
    }
};
template <class T>
class Stack {
    struct Element {
        Element* prev;                          //Points to the previous element on the
        Stack
        T data;                                 //Data of type T stored in this element
        Element(Element* p,T d):prev(p),data(d){}   //Constructor
        Element(T d):data(d){}                  //Constructor, initialize
        Element(){}                             //Constructor, initialize no member data
    };
    Element* tos;                               //Points to the element on top of the Stack
    void Kill(Element* e) {...}                     //Reclaims storage for all elements on
    the
                                                //Stack
public:
    Stack():...{...}                                //Constructor
    ~Stack(){}
    bool IsEmpty(void){...}
    bool IsFull(void) {...}
    void Push(T d) {...}
    T Pop(void) {...}
};
```

Figure 4. class *Stack*

5. Which of the following is a valid implementation of function *Push* for a stack whose elements are stored in a dynamically allocated array as shown in the Figure below?

   a. **void** *Push*(*T v*){**if** (*IsFull*()) **throw** *StackException*(**"full"**);*S*[*tos*]=*v*;*tos*++;}
   b. **void** *Push*(*T v*){**if** (*IsFull*()) **throw** *StackException*(**"full"**);*S*[++*tos*]=*v*;}
   c. **void** *Push*(*T v*){**if** (*IsFull*()) **throw** *StackException*(**"full"**);*S*[*tos*]=*v*;++*tos*;}
   d. **void** *Push*(*T v*){**if** (*IsFull*()) **throw** *StackException*(**"full"**);*S*[*tos*++]=*v*;}

```
struct StackException{
    StackException(char* m)
    {   cout<<endl<<"I am the Stack and I am " << m << "." << endl;
    }
};
template <class T>
class Stack {
    int size;              //Number of elements available
    int tos;               //Index of the element on top of the Stack
    T* S;                  //Points to an array of type T containing the elements of the
    Stack
public:
    Stack(int sz=100);     //Constructor, initialize member data and allocate storage
    ~Stack();              //Destructor, reclaim storage used to implement the Stack
    bool IsEmpty(voiv);    //Is the Stack empty?
    bool IsFull(void);     //Is the Stack full?
    void Push(T v);        //Insert a value, v, on top of the Stack.
    T Pop(void);           //Remove and return the element on top of the Stack
};
```

Figure 5. class *Stack*

6. Which of the following is a valid implementation of the constructor, *Queue*, for a queue whose elements are individually stored in dynamically allocated elements as shown in the Figure below?

   a. *Queue*(){*count*=0;*oldest*=*newest*=0;}
   b. *Queue*():*count*(0),*oldest*(nil),*newest*(nil){}
   c. *Queue*():*count*(0){*Element** *e*=**new** *Element*; *oldest*=*newest*=*e*; *e*->*newer*=*e*;}
   d. *Queue*():*count*(0){*newest*=*oldest*;}

```
struct QueueException{
    QueueException(char* m)
    {   cout<<endl<<"I am the Queue and I am " << m << "." << endl;
    }
};
template <class T>
class Queue {
    struct Element {
        Element* newer;                          //Points to the next newer element
        T data;                                  //Value of type T stored in this element
        Element(Element* n,T d):newer(n),data(d){}//Constructor
        Element(T t):newer(0),v(t){}             //Constructor
        Element(){}                              //Constructor
    };
    Element* oldest;                             //Points to the oldest element
    Element* newest;                             //Points to the newest element
    int count;                                   //Records the number of elements
    void Kill(Element* e){...}                   //Reclaims storage for all elements
public:
    Queue();                                     //Constructor
    ~Queue();                                    //Destructor
    bool IsEmpty(void){...}                      //Is the Queue empty?
    bool IsFull(void){...}                       //Is the Queue full?
    void Enq(T v){...}                           //Insert a value, v, on the Queue
    T Deq(void){...}                             //Remove the oldest value on the Queue
    int Count(void){...}                         //Return the number of elements on the
    Queue
};
```

Figure 6. class *Queue*

7. Select the expressions in suffix notation equivalent to the expression tree in the Figure below.

    a. **(1-2)/3*4*5+6/7**
    b. **1 2 – 3 / 4 5 * 6 7 / + ***
    c. **1-2/3*4*5+6/7**
    d. *** / - 1 2 3 + * 4 5 / 6 7**

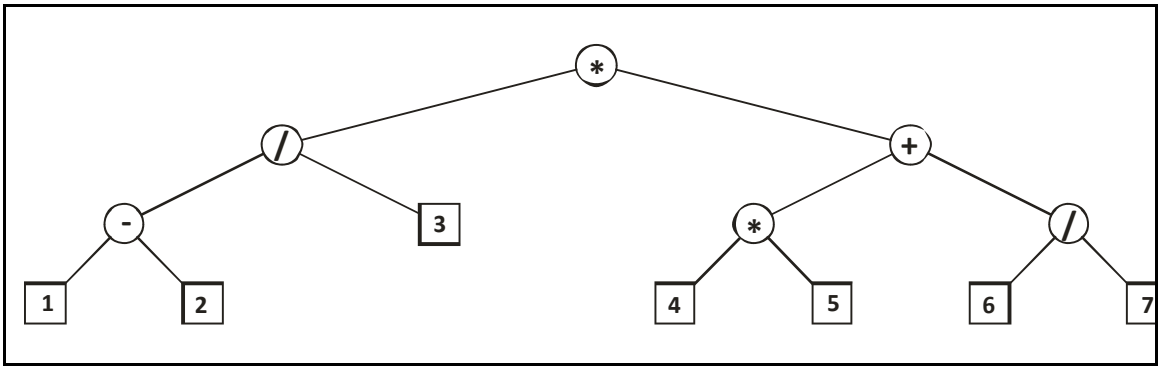

Figure 7. An expression tree

8. What is the time complexity of function *Enq* for a queue whose elements are stored in a dynamically allocated array as shown in the Figure below?

   a. $O(1)$
   b. $O(n \log_2 n)$
   c. $O(n)$
   d. $O(\log_2 n)$

```
struct QueueException{
    QueueException(char* m)
    {   cout<<endl<<"I am the Queue and I am " << m << "." << endl;
    }
};
template <class T>
class Queue {
    int size;                   //Number of available elements to store values on the Queue

    int count;                  //Number of elements stored on the Queue
    T* Q;                       //Points to a dynamically allocated array used to implement the
                                // Queue
    int oldest;                 //Index of the oldest element on the Queue
    int newest;                 //Index of the newest element on the Queue
public:
    Queue(int sz=100){...}      //Constructor, initialize member data and allocate storage for
    the
                                //Queue
    ~Queue(){...}               //Destructor, reclaim storage
    bool IsEmpty(void){...}     //Is the Queue empty?
    bool IsFull(void){...}      //Is the Queue full?
    void Enq(T d){...}          //Insert a value, d, on the Queue
    T Deq(void){...}            //Remove and return the oldest value on the Queue
};
```

Figure 8. class *Queue*

9. Which of the following is a valid implementation of the function *Kill* for a queue whose individual elements are allocated dynamically as shown in the Figure below?

a.
```
void Kill(Element* e)
{   if (e) {
        delete e;
        Kill(e->newer);
    }
}
```

b.
```
void Kill(Element* e)
{   if (e) return;
    Kill(e->newer);
    delete e;
}
```

c.
```
void Kill(Element* e)
{   if (!e){
        delete e;
        Kill(e->newer);
    }
}
```

d.
```
void Kill(Element* e)
{   if (!e) return;
    Kill(e->newer);
    delete e;
}
```

```
struct QueueException{
    QueueException(char* m)
    {   cout<<endl<<"I am the Queue and I am " << m << "." << endl;
    }
};
template <class T>
class Queue {
    struct Element {
        Element* newer;                          //Points to the next newer element
        T data;                                  //Value of type T stored in this element
        Element(Element* n,T d):newer(n),data(d){}     //Constructor
        Element(T t):newer(0),v(t){}             //Constructor
        Element(){}                              //Constructor
    };
    Element* oldest;                             //Points to the oldest element
    Element* newest;                             //Points to the newest element
    int count;                                   //Records the number of elements
    void Kill(Element* e){…}                     //Reclaims storage for all elements
public:
    Queue();                                     //Constructor
    ~Queue();                                    //Destructor
    bool IsEmpty(void){…}                        //Is the Queue empty?
    bool IsFull(void){…}                         //Is the Queue full?
    void Enq(T v){…}                             //Insert a value, v, on the Queue
    T Deq(void){…}                               //Remove the oldest value on the Queue
    int Count(void){…}                           //Return the number of elements on the
    Queue
};
```

Figure 9. class *Queue*

10. What is the time complexity of function *Index* for a list whose elements are stored in a dynamically allocated array? Function *Index* is given in Figure below.

    a. $O(n)$
    b. $O(\log_2 n)$
    c. $O(1)$
    d. $O(n \log_2 n)$

```
int Index(T key)
{    int lo=1,hi=count;
     while (lo<=hi) {
          int m=(lo+hi)/2;
          if (key==L[m]) return m;
          if (key<L[m]) hi=m-1; else lo=m+1;
     }
     return 0;
}
```

Figure 10. Function *Index*

11. Which of the following is a valid implementation of the constructor, *Stack*, for a stack whose elements are stored in a dynamically allocated array as shown in the Figure below?

    a. `Stack(int sz=100):size(sz),tos(0){S=new T[sz];}`
    b. `Stack(int size=100):size(sz),tos(-1){S=new T[sz];}`
    c. `Stack(int size=100):size(100),tos(0){S=new T[size];}`
    d. `Stack(int sz=100){size=sz;tos=-1;S=new T[size];}`

```
struct StackException{
     StackException(char* m)
     {    cout<<endl<<"I am the Stack and I am " << m << "." << endl;
     }
};
template <class T>
class Stack {
     int size;              //Number of elements available
     int tos;               //Index of the element on top of the Stack
     T* S;                  //Points to an array of type T containing the elements of the
     Stack
public:
     Stack(int sz=100);     //Constructor, initialize member data and allocate storage
     ~Stack();              //Destructor, reclaim storage used to implement the Stack
     bool IsEmpty(void);    //Is the Stack empty?
     bool IsFull(void);     //Is the Stack full?
     void Push(T d);        //Insert a value, v, on top of the Stack.
     T Pop(void);           //Remove and return the element on top of the Stack
};
```

Figure 11. class *Stack*

12. Which of the following is a valid implementation of the function *Deq*, for a queue whose elements are stored in a dynamically allocated array as shown in the Figure below?

a.
```
T Deq(void)
{   if (IsEmpty()) throw QueueException("empty");
    oldest++;
    T v=Q[oldest];
    count--;
    return v;
}
```

b.
```
T Deq(void)
{   if (IsEmpty()) throw QueueException("empty");
    T v=Q[(oldest+1)%size];
    count--;
    return v;
}
```

c.
```
T Deq(void)
{   if (IsEmpty()) throw QueueException("empty");
    oldest=(oldest+1)%size;
    T v=Q[oldest];
    count--;
    return v;
}
```

d.
```
T Deq(void)
{   if (IsEmpty()) throw QueueException("empty");
    T v=Q[oldest];
    oldest=(oldest+1)%size;
    count--;
    return v;
}
```

```
struct QueueException{
    QueueException(char* m)
    {   cout<<endl<<"I am the Queue and I am " << m << "."<< endl;
    }
};
template <class T>
class Queue {
    int size;                   //Number of available elements to store values on the Queue

    int count;                  //Number of elements stored on the Queue
    T* Q;                       //Points to a dynamically allocated array used to implement the
                                // Queue
    int oldest;                 //Index of the oldest element on the Queue
    int newest;                 //Index of the newest element on the Queue
public:
    Queue(int sz=100){…}        //Constructor, initialize member data and allocate storage for
    the
                                //Queue
    ~Queue(){…}                 //Destructor, reclaim storage
    bool IsEmpty(void) {…}      //Is the Queue empty?
    bool IsFull(void) {…}       //Is the Queue full?
    void Enq(T d) {…}           //Insert a value, d, on the Queue
    T Deq(void){…}              //Remove and return the oldest value on the Queue
};
```

Figure 12. class *Queue*

13. What is the time complexity of function *Insert* for a list whose elements are stored in a dynamically allocated array? Function Insert is given in the Figure below.

    a. $O(n \log_2 n)$
    b. $O(1)$
    c. $O(n)$
    d. $O(\log_2 n)$

```
void Insert(T key)
{   if (IsMember(key)) return;
    if (IsFull()) throw ListIException("full");
    int i=++count;
    for (;key<L[i-1];i--) L[i]=L[i-1];
    L[i]=key;
}
```

Figure 13. Function *Insert*

14. What is the time complexity of function *Pop* for a stack whose elements are stored in a dynamically allocated array as shown in the Figure below?

    a. $O(n)$
    b. $O(n \log_2 n)$

c. $O(1)$
d. $O(\log_2 n)$

```
struct StackException{
    StackException(char* m)
    {   cout<<endl<<"I am the Stack and I am " << m << "." << endl;
    }
};
template <class T>
class Stack {
    int size;              //Number of elements available
    int tos;               //Index of the element on top of the Stack
    T* S;                  //Points to an array of type T containing the elements of the
    Stack
public:
    Stack(int sz=100);     //Constructor, initialize member data and allocate storage
    ~Stack();              //Destructor, reclaim storage used to implement the Stack
    bool IsEmpty(void);    //Is the Stack empty?
    bool IsFull(void);     //Is the Stack full?
    void Push(T d);        //Insert a value, v, on top of the Stack.
    T Pop(void);           //Remove and return the element on top of the Stack
};
```

Figure 14. class *Stack*

15. (31) Insert the strings "The" "cow" "is" "of" "the" "bovine" "ilk." into a stack, a queue, a list, and a binary search tree. Now, print the list according to the following instructions for each structure used. For the stack, pop and print elements from the stack. For the queue, dequeue and print elements from the queue. For the list, note that the list is stored in ascending order and print the list accordingly. For the tree, print elements using a post-order traversal. Match the lists printed for each structure against the lists in Table 15. Select the data structure from which the corresponding correct ordered list of strings is given.

Ordered list of strings	Abstract data type
"bovine" "ilk." "of" "is" "cow" "the" "The"	**class** *List*
"The" "cow" "is" "of" "the" "bovine" "ilk."	**class** *Queue*
"bovine" "cow" "ilk." "is" "of" "The" "the"	**class** *Stack*
"ilk." "bovine" "the" "of" "is" "cow" "The"	**class** *Tree*

Table 15. Ordered list of strings

a. Tree
b. List
c. Queue
d. Stack

16. Which of the following is a valid implementation of the function *IsFull* for a queue whose individual elements are allocated dynamically as shown in the Figure below?

a.
```
bool IsFull(void)
{   Element* e=new Element;
    bool v=(bool)e;
    if (e) delete e;
    return v;
}
```

b.
```
bool IsFull(void){return count>=size;}
```

c.
```
bool IsFull(void)
{   Element* e=new Element;
    bool v=(bool)(!e);
    if (e) delete e;
    return v;
}
```

d.
```
bool IsFull(void){return count<=0;}
```

```
struct QueueException{
    QueueException(char* m)
    {   cout<<endl<<"I am the Queue and I am " << m << "." << endl;
    }
};
template <class T>
class Queue {
    struct Element {
        Element* newer;                 //Points to the next newer element
        T data;                         //Value of type T stored in this element
        Element(Element* n,T d):newer(n),data(d){}//Constructor
        Element(T t):newer(0),v(t){}    //Constructor
        Element(){}                     //Constructor
    };
    Element* oldest;                    //Points to the oldest element
    Element* newest;                    //Points to the newest element
    int count;                          //Records the number of elements
    void Kill(Element* e){...}          //Reclaims storage for all elements
public:
    Queue();                            //Constructor
    ~Queue();                           //Destructor
    bool IsEmpty(void){...}             //Is the Queue empty?
    bool IsFull(void){...}              //Is the Queue full?
    void Enq(T v){...}                  //Insert a value, v, on the Queue
    T Deq(void){...}                    //Remove the oldest value on the Queue
    int Count(void){...}                //Return the number of elements on the Queue
};
```

Figure 16. class *Queue*

17. What is the maximum number of comparisons required to find a key in a complete binary tree having 9,754 nodes?

    a. 12
    b. 14
    c. 11
    d. 13

18. Which of the following is a valid implementation of function *IsFull* for a stack whose elements are stored in a dynamically allocated array as shown in the Figure below?

    a. `bool IsFull(void){return tos>size;}`
    b. `bool IsFull(void){return size>tos;}`
    c. `bool IsFull(void){return size-1<=tos;}`
    d. `bool IsFull(void){return tos>=size;}`

```
struct StackException{
    StackException(char* m)
    {   cout<<endl<<"I am the Stack and I am " << m << "." << endl;
    }
};
template <class T>
class Stack {
    int size;               //Number of elements available
    int tos;                //Index of the element on top of the Stack
    T* S;                   //Points to an array of type T containing the elements of the
    Stack
public:
    Stack(int sz=100);      //Constructor, initialize member data and allocate storage
    ~Stack();               //Destructor, reclaim storage used to implement the Stack
    bool IsEmpty(void);     //Is the Stack empty?
    bool IsFull(void);      //Is the Stack full?
    void Push(T d);         //Insert a value, v, on top of the Stack.
    T Pop(void);            //Remove and return the element on top of the Stack
};
```

Figure 18. class *Stack*

19. Which of the following is a valid implementation of the function *Kill*, for a list whose individual elements are allocated dynamically as shown in the Figure below?

a.
```
void Kill(Element* e)
{   if (e==largest) return;
    Kill(e->larger);
    delete e;
}
```

b.
```
void Kill(Element* e)
{   if (e->key<MAX) return;
    Kill(e->larger);
    delete e;
}
```

c.
```
void Kill(Element* e)
{   if (e!=largest) return;
    Kill(e->larger);
    delete e;
}
```

d.
```
void Kill(Element* e)
{   if (e==largest){
        Kill(e->larger);
        delete e;
    }
```

```
        }
template <class T>
class List {
    struct Element {
        Element* smaller;          //References the next smaller element
        T key;                     //Key of type T
        Element* larger;           //References the next larger element
        Element(){}                //Default constructor
                                   //Constructor assigning all member data
        Element(Element* s,T k,Element *l):smaller(s),key(k),larger(l){}
    };
    Element* largest;              //Points to the sentinel
    Element* cursor;               //Points to the cursor
    void Kill(Element* e);         //Reclaims storage for all elements except the
    sentinel
protected:
    const T MAX;                   //Maximum element of type T stored in the sentinel
public:
    List(T m);                     //Constructor, m is the maximum value of type T
    ~List();                       //Destructor, reclaim storage for the list
    void Insert(T k);              //Insert element k
    void Remove(T k);              //Remove element  k
    void Print(ostream& o,char* title); //Print the list, title={i1,i2,...,in}
    void Scan(istream& i);         //Store values found in stream in the list
    void First(void);              //Position the cursor on the smallest element
    void Next(void);               //Position the cursor on the next larger element
    bool IsEol(void);              //Is the cursor just past the largest element on the
    list
    T Member(void);                //Return the value of the element on which the
    cursor
                                   //is positioned
    bool IsMember(T k);            //Is member k an element of the list
};
```

Figure 19. class *List*

20. Which of the following is a valid implementation of the constructor, *Queue*, for a queue whose elements are stored in a dynamically allocated array as shown in the Figure below?

    a. `Queue(int sz=100):size(sz),count(0),oldest(-1),newest(size-1){Q=new T[size];}`
    b. `Queue(int sz=100):size(sz),count(0),oldest(0),newest(-1){Q=new T[size];}`
    c. `Queue(int sz=100):size(sz),count(0),oldest(0),newest(0){Q=new T[size];}`
    d. `Queue(int sz=100):size(sz),count(0),oldest(-1),newest(0){Q=new T[size];}`

```
struct QueueException{
    QueueException(char* m)
    {   cout<<endl<<"I am the Queue and I am " << m << "." << endl;
    }
};
template <class T>
class Queue {
    int size;                   //Number of available elements to store values on the Queue

    int count;                  //Number of elements stored on the Queue
    T* Q;                       //Points to a dynamically allocated array used to implement the
                                // Queue
    int oldest;                 //Index of the oldest element on the Queue
    int newest;                 //Index of the newest element on the Queue
public:
    Queue(int sz=100){…}        //Constructor, initialize member data and allocate storage for
    the
                                //Queue
    ~Queue(){…}                 //Destructor, reclaim storage
    bool IsEmpty(void) {…}      //Is the Queue empty?
    bool IsFull(void) {…}       //Is the Queue full?
    void Enq(T d) {…}           //Insert a value, d, on the Queue
    T Deq(void){…}              //Remove and return the oldest value on the Queue
};
```

Figure 20. class *Queue*

21. What is the time complexity of the destructor, **~***Queue*, for a queue whose elements are stored in a dynamically allocated array as shown in the Figure below?

    a. $O(n)$
    b. $O(1)$
    c. $O(\log_2 n)$
    d. $O(n \log_2 n)$

```
struct QueueException{
    QueueException(char* m)
    {   cout<<endl<<"I am the Queue and I am " << m << "." << endl;
    }
};
template <class T>
class Queue {
    int size;                   //Number of available elements to store values on the Queue

    int count;                  //Number of elements stored on the Queue
    T* Q;                       //Points to a dynamically allocated array used to implement the
                                // Queue
    int oldest;                 //Index of the oldest element on the Queue
    int newest;                 //Index of the newest element on the Queue
public:
    Queue(int sz=100){...}      //Constructor, initialize member data and allocate storage for
    the
                                //Queue
    ~Queue(){...}               //Destructor, reclaim storage
    bool IsEmpty(void){...}     //Is the Queue empty?
    bool IsFull(void){...}      //Is the Queue full?
    void Enq(T d){...}          //Insert a value, d, on the Queue
    T Deq(void){...}            //Remove and return the oldest value on the Queue
};
```

Figure 21. class *Queue*

22. What is the time complexity of function *Insert* for a list whose elements are individually stored in dynamically allocated elements as shown in the Figure below?

    a. $O(n \log_2 n)$
    b. $O(\log_2 n)$
    c. $O(1)$
    d. $O(n)$

```
template <class T>
class List {
    struct Element {
        Element* smaller;              //References the next smaller element
        T key;                         //Key of type T
        Element* larger;               //References the next larger element
        Element(){}                    //Default constructor
                                       //Constructor assigning all member data
        Element(Element* s,T k,Element *l):smaller(s),key(k),larger(l){}
    };
    Element* largest;                  //Points to the sentinel
    Element* cursor;                   //Points to the cursor
    void Kill(Element* e);             //Reclaims storage for all elements except the
    sentinel
protected:
    const T MAX;                       //Maximum element of type T stored in the sentinel
public:
    List(T m);                         //Constructor, m is the maximum value of type T
    ~List();                           //Destructor, reclaim storage for the list
    void Insert(T k);                  //Insert element k
    void Remove(T k);                  //Remove element  k
    void Print(ostream& o,char* title); //Print the list, title={i1,i2,...,in}
    void Scan(istream& i);             //Store values found in stream in the list
    void First(void);                  //Position the cursor on the smallest element
    void Next(void);                   //Position the cursor on the next larger element
    bool IsEol(void);                  //Is the cursor just past the largest element on the
    list
    T Member(void);                    //Return the value of the element on which the
    cursor
                                       //is positioned
    bool IsMember(T k);                //Is member k an element of the list
};
```

Figure 22. class *List*

23. Which of the following is a valid implementation of function *IsEmpty* for a stack whose elements are stored in a dynamically allocated array as shown in the Figure below?

    a. `bool IsEmpty(void){return tos>0;}`
    b. `bool IsEmpty(void){return -1<=tos;}`
    c. `bool IsEmpty(void){return 0>=tos;}`
    d. `bool IsEmpty(void){return -1>=tos;}`

```
struct StackException{
    StackException(char* m)
    {   cout<<endl<<"I am the Stack and I am " << m << "." << endl;
    }
};
template <class T>
class Stack {
    int size;               //Number of elements available
    int tos;                //Index of the element on top of the Stack
    T* S;                   //Points to an array of type T containing the elements of the
    Stack
public:
    Stack(int sz=100);      //Constructor, initialize member data and allocate storage
    ~Stack();               //Destructor, reclaim storage used to implement the Stack
    bool IsEmpty(void);     //Is the Stack empty?
    bool IsFull(void);      //Is the Stack full?
    void Push(T d);         //Insert a value, v, on top of the Stack.
    T Pop(void);            //Remove and return the element on top of the Stack
};
```

Figure 23. class *Stack*

24. Which of the following is a valid implementation of the constructor, *List*, for a list whose individual elements are allocated dynamically as shown in the Figure below?

    a.
    ```
    List(T m):MAX(m)
    {   Element* n=new Element;
        n->key=m;
        n=cursor;
        n=largest; n=n->smaller;
        n=n->larger;
    }
    ```

    b.
    ```
    List(T m):MAX(m)
    {   Element* n=new Element(0,MAX,0);
        n=cursor=largest=n->smaller=n->larger;
    }
    ```

    c.
    ```
    List(T m):MAX(m)
    {   Element* n=new Element(0,MAX,0);
        cursor=largest=n->smaller=n->larger=n;
    }
    ```

    d.
    ```
    List(T m):MAX(m)
    {   MAX=m; Element* n=new Element;
        n->key=m;cursor=largest=n->smaller=n->larger=n;
    }
    ```

```
template <class T>
class List {
    struct Element {
        Element* smaller;          //References the next smaller element
        T key;                     //Key of type T
        Element* larger;           //References the next larger element
        Element(){}                //Default constructor
                                   //Constructor assigning all member data
        Element(Element* s,T k,Element *l):smaller(s),key(k),larger(l){}
    };
    Element* largest;              //Points to the sentinel
    Element* cursor;               //Points to the cursor
    void Kill(Element* e);         //Reclaims storage for all elements except the
    sentinel
protected:
    const T MAX;                   //Maximum element of type T stored in the sentinel
public:
    List(T m);                     //Constructor, m is the maximum value of type T
    ~List();                       //Destructor, reclaim storage for the list
    void Insert(T k);              //Insert element k
    void Remove(T k);              //Remove element  k
    void Print(ostream& o,char* title); //Print the list, title={i1,i2,...,in}
    void Scan(istream& i);         //Store values found in stream in the list
    void First(void);              //Position the cursor on the smallest element
    void Next(void);               //Position the cursor on the next larger element
    bool IsEol(void);              //Is the cursor just past the largest element on the
    list
    T Member(void);                //Return the value of the element on which the
    cursor
                                   //is positioned
    bool IsMember(T k);            //Is member k an element of the list
};
```

Figure 24. class *List*

25. Which of the following is a valid implementation of the function *IsEmpty*, for a queue whose elements are stored in a dynamically allocated array as shown in the Figure below?

    a. `bool IsEmpty(void){return 0<=count;}`
    b. `bool IsEmpty(void){return count!=0;}`
    c. `bool IsEmpty(void){return count>=0;}`
    d. `bool IsEmpty(void){return 0>=count;}`

```
struct QueueException{
    QueueException(char* m)
    {   cout<<endl<<"I am the Queue and I am " << m << "." << endl;
    }
};
template <class T>
class Queue {
    int size;                   //Number of available elements to store values on the Queue

    int count;                  //Number of elements stored on the Queue
    T* Q;                       //Points to a dynamically allocated array used to implement the
                                // Queue
    int oldest;                 //Index of the oldest element on the Queue
    int newest;                 //Index of the newest element on the Queue
public:
    Queue(int sz=100){…}        //Constructor, initialize member data and allocate storage for
    the
                                //Queue
    ~Queue(){…}                 //Destructor, reclaim storage
    bool IsEmpty(void) {…}      //Is the Queue empty?
    bool IsFull(void) {…}       //Is the Queue full?
    void Enq(T d) {…}           //Insert a value, d, on the Queue
    T Deq(void){…}              //Remove and return the oldest value on the Queue
};
```

Figure 25. class *Queue*

26. Which of the following sequences is an in-order traversal of the binary search tree in the Figure below?

    a. none of the other alternatives
    b. **A I F Q T M**
    c. **A F I M Q T**
    d. **M F A I T Q**

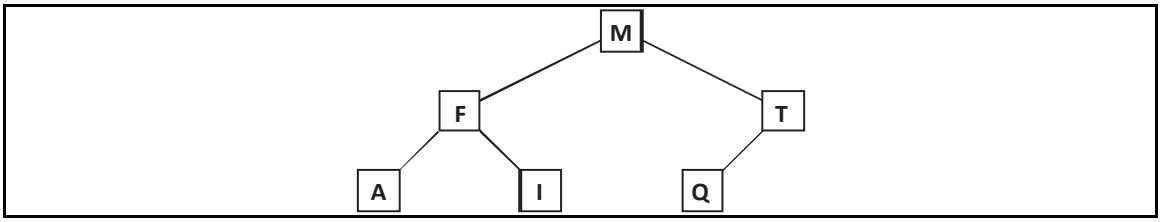

Figure 26. Binary tree

27. What is the time complexity of function *Kill* for a stack whose individual elements are allocated dynamically as shown in the Figure below?

    a. $O(n)$
    b. $O(1)$
    c. $O(\log_2 n)$
    d. $O(n \log_2 n)$

```
struct StackException{
    StackException(char* m)
    {   cout<<endl<<"I am the Stack and I am " << m << "." << endl;
    }
};
template <class T>
class Stack {
    struct Element {
        Element* prev;                          //Points to the previous element on the
        Stack
        T data;                                 //Data of type T stored in this element
        Element(Element* p,T d):prev(p),data(d){}   //Constructor
        Element(T d):data(d){}                  //Constructor, initialize
        Element(){}                             //Constructor, initialize no member data
    };
    Element* tos;                               //Points to the element on top of the Stack
    void Kill(Element* e) {…}                   //Reclaims storage for all elements on the
                                                //Stack
public:
    Stack():…{…}                                //Constructor
    ~Stack(){}                                  //Destructor
    bool IsEmpty(void){…}                       //Is the Stack empty?
    bool IsFull(void) {…}                       //Is the Stack full?
    void Push(T d) {…}                          //Push a value, d, on the Stack
    T Pop(void) {…}                             //Pop a value from the Stack
};
```

Figure 27. class *Stack*

28. Which of the following is a valid implementation of the function *Insert*, for a list whose individual elements are allocated dynamically as shown in the Figure below?

a.
```
void Insert(T k)
{   Element* e=largest->larger;
    while (k > e->key) e=e->larger;
    if (k == e->key) return;
    Element* n=new Element(e->smaller,k,e);
    e->smaller->larger=n;
    e->smaller=n;
}
```

b.
```
void Insert(T k)
{   Element* e=largest->larger;
    while (k > e->key) e=e->larger;
    if (k == e->key) return;
    Element* n=new Element(e,k,e->larger);
    e->smaller->larger=n;
    e->smaller=n;

}
```

c.
```
void Insert(T k)
{   Element* e=largest->larger;
    while (k > e->key) e=e->larger;
    if (k == e->key) return;
    Element* n=new Element(e->smaller,k,e);
    e->smaller=n;
    e->smaller->larger=n;
}
```

d.
```
void Insert(T k)
{   Element* e=largest->larger;
    while (k < e->key) e=e->larger;
    if (k == e->key) return;
    Element* n=new Element(e->smaller,k,e);
    e->smaller->larger=n;
    e->smaller=n;
}
```

```
template <class T>
class List {
    struct Element {
        Element* smaller;              //References the next smaller element
        T key;                         //Key of type T
        Element* larger;               //References the next larger element
        Element(Element* s,T k,Element *l):smaller(s),key(k),larger(l){}
    };
    Element* largest;                  //Points to the sentinel
    Element* cursor;                   //Points to the cursor
    void Kill(Element* e);             //Reclaims storage for all elements except the
    sentinel
protected:
    const T MAX;                       //Maximum element of type T stored in the sentinel
public:
    List(T m);                         //Constructor, m is the maximum value of type T
    ~List();                           //Destructor, reclaim storage for the list
    void Insert(T k);                  //Insert element k
    void Remove(T k);                  //Remove element  k
    void Print(ostream& o,char* title); //Print the list, title={i1,i2,...,in}
    void Scan(istream& i);             //Store values found in stream in the list
    void First(void);                  //Position the cursor on the smallest element
    void Next(void);                   //Position the cursor on the next larger element
    bool IsEol(void);                  //Is the cursor just past the largest element on the
    list
    T Member(void);                    //Return the value of the element on which the
    cursor
                                       //is positioned
    bool IsMember(T k);                //Is member k an element of the list
};
```

Figure 28. class *List*

29. What is the time complexity of the destructor, **~***List*, for a list whose individual elements are allocated dynamically as shown in the Figure below?

    a. $O(1)$
    b. $O(n \log_2 n)$
    c. $O(n)$
    d. $O(\log_2 n)$

```
template <class T>
class List {
    struct Element {
        Element* smaller;              //References the next smaller element
        T key;                         //Key of type T
        Element* larger;               //References the next larger element
        Element(){}                    //Default constructor
                                       //Constructor assigning all member data
        Element(Element* s,T k,Element *l):smaller(s),key(k),larger(l){}
    };
    Element* largest;                  //Points to the sentinel
    Element* cursor;                   //Points to the cursor
    void Kill(Element* e);             //Reclaims storage for all elements except the
    sentinel
protected:
    const T MAX;                       //Maximum element of type T stored in the sentinel
public:
    List(T m);                         //Constructor, m is the maximum value of type T
    ~List();                           //Destructor, reclaim storage for the list
    void Insert(T k);                  //Insert element k
    void Remove(T k);                  //Remove element  k
    void Print(ostream& o,char* title); //Print the list, title={i1,i2,...,in}
    void Scan(istream& i);             //Store values found in stream in the list
    void First(void);                  //Position the cursor on the smallest element
    void Next(void);                   //Position the cursor on the next larger element
    bool IsEol(void);                  //Is the cursor just past the largest element on the
    list
    T Member(void);                    //Return the value of the element on which the
    cursor
                                       //is positioned
    bool IsMember(T k);                //Is member k an element of the list
};
```

Figure 29. class *List*

30. Which of the following is a valid implementation of function *Push* for a stack whose individual elements are allocated dynamically as shown in the Figure below?

    a.
    ```
    void Push(T d)
    {   if (IsFull()) throw StackException("full");
        Element* e=new Element(tos,d);
    }
    ```

b.
```
void Push(T d)
{   if (IsFull()) throw StackException("full");
    Element* e=new Element(d);
    e->prev=tos;
    tos=e;
}
```

c.
```
void Push(T d)
{   if (IsFull()) throw StackException("full");
    Element* e=new Element;
    tos=e;
    e->prev=tos;
    e->data=d;
}
```

d.
```
void Push(T d)
{   if (IsFull()) throw StackException("full");
    Element* e=new Element(d,tos);
    tos=e;
}
```

```
struct StackException{
    StackException(char* m)
    {   cout<<endl<<"I am the Stack and I am " << m << "." << endl;
    }
};
template <class T>
class Stack {
    struct Element {
        Element* prev;                  //Points to the previous element on the Stack
        T data;                         //Data of type T stored in this element
        Element(Element* p,T d):prev(p),data(d){}   //Constructor
        Element(T d):data(d){}          //Constructor, initialize
        Element(){}                     //Constructor, initialize no member data
    };
    Element* tos;                       //Points to the element on top of the Stack
    void Kill(Element* e) {...}         //Reclaims storage for all elements on the  Stack
public:
    Stack():...{...}                    //Constructor
    ~Stack(){}                          //Destructor
    bool IsEmpty(void){...}             //Is the Stack empty?
    bool IsFull(void) {...}             //Is the Stack full?
    void Push(T d) {...}                //Push a value, d, on the Stack
    T Pop(void) {...}                   //Pop a value from the Stack
};
```

Figure 30. class *Stack*

31. Which of the following is a valid implementation of the constructor, *Stack*, for a stack whose individual elements are allocated dynamically as shown in the Figure below?

    a. *Stack*(**):***tos*(**~NULL**){}
    b. *Stack*(**):***tos*(**0**){}
    c. *Stack*(**):***tos*(**!0**){}
    d. *Stack*(**):***tos*(**1.602e-19**){}

```
struct StackException{
    StackException(char* m)
    {   cout<<endl<<"I am the Stack and I am " << m << "." << endl;
    }
};
template <class T>
class Stack {
    struct Element {
        Element* prev;                  //Points to the previous element on the Stack
        T data;                         //Data of type T stored in this element
        Element(Element* p,T d):prev(p),data(d){}   //Constructor
        Element(T d):data(d){}          //Constructor, initialize
        Element(){}                     //Constructor, initialize no member data
    };
    Element* tos;                       //Points to the element on top of the Stack
    void Kill(Element* e) {…}           //Reclaims storage for all elements on the  Stack
public:
    Stack():…{…}                        //Constructor
    ~Stack(){}                          //Destructor
    bool IsEmpty(void){…}               //Is the Stack empty?
    bool IsFull(void){…}                //Is the Stack full?
    void Push(T d) {…}                  //Push a value, d, on the Stack
    T Pop(void){…}                      //Pop a value from the Stack
};
```

Figure 31. class *Stack*

32. Which of the following is a valid implementation of the function *Enq* for a queue whose individual elements are allocated dynamically as shown in the Figure below?

    a.
    ```
    void Enq(T d)
    {   if (IsFull()) throw QueueException("full");
        Element* e=new Element(d);
        if (count==0) oldest=e; else newest->newer=e;
        newest=e;
        count++;
    }
    ```
    b.
    ```
    void Enq(T d)
    {   if (IsFull()) throw QueueException("full");
    ```

```
        Element* e=new Element(d);
        if (IsEmpty()) {oldest=e;} else {newest->newer=e; newest=e;}
        count++;
    }
```

c.
```
    void Enq(T d)
    {   if (IsFull()) throw QueueException("full");
        Element* e=new Element(d);
        if (!IsFull()) oldest=e; else newest->newer=e;
        newest=e;
        count++;
    }
```

d.
```
    void Enq(T d)
    {   if (IsFull()) throw QueueException("full");
        Element* e=new Element(d);
        newest=e;
        if (IsEmpty()) oldest=e; else newest->newer=e;
        count++;
    }
```

```
struct QueueException{
    QueueException(char* m)
    {   cout<<endl<<"I am the Queue and I am " << m << "." << endl;
    }
};
template <class T>
class Queue {
    struct Element {
        Element* newer;                          //Points to the next newer element
        T data;                                  //Value of type T stored in this element
        Element(Element* n,T d):newer(n),data(d){}//Constructor
        Element(T t):newer(0),v(t){}             //Constructor
        Element(){}                              //Constructor
    };
    Element* oldest;                             //Points to the oldest element
    Element* newest;                             //Points to the newest element
    int count;                                   //Records the number of elements
    void Kill(Element* e){...}                   //Reclaims storage for all elements
public:
    Queue();                                     //Constructor
    ~Queue();                                    //Destructor
    bool IsEmpty(void){...}                      //Is the Queue empty?
    bool IsFull(void){...}                       //Is the Queue full?
    void Enq(T v){...}                           //Insert a value, v, on the Queue
    T Deq(void){...}                             //Remove the oldest value on the Queue
    int Count(void){...}                         //Return the number of elements on the
    Queue
};
```

Figure 32. class *Queue*

33. What is the maximum number of comparisons required to find a key in a binary tree having 191 nodes and whose height is equal to 12?

    a. 11
    b. 14
    c. 13
    d. 12

34. Which of the following is a valid implementation of function *Pop* for a stack whose elements are stored in a dynamically allocated array as shown in the Figure below?

a.
```
T Pop(void)
{   if (IsEmpty()) throw StackException("empty");
    return S[tos];
    --tos;
}
```

b.
```
T Pop(void)
{   if (IsEmpty()) throw StackException("empty");
    T v=S[tos];
    --tos;
    return v;
}
```

c.
```
T Pop(void)
{   if (IsEmpty()) throw StackException("empty");
    return S[--tos];
}
```

d.
```
T Pop(void)
{   if (IsEmpty()) throw StackException("empty");
    return S[tos];
    tos--;
}
```

```
struct StackException{
    StackException(char* m)
    {   cout<<endl<<"I am the Stack and I am " << m << "." << endl;
    }
};
template <class T>
class Stack {
    int size;               //Number of elements available
    int tos;                //Index of the element on top of the Stack
    T* S;                   //Points to an array of type T containing the elements of the
    Stack
public:
    Stack(int sz=100);      //Constructor, initialize member data and allocate storage
    ~Stack();               //Destructor, reclaim storage used to implement the Stack
    bool IsEmpty(void);     //Is the Stack empty?
    bool IsFull(void);      //Is the Stack full?
    void Push(T d);         //Insert a value, v, on top of the Stack.
    T Pop(void);            //Remove and return the element on top of the Stack
};
```

Figure 34. class *Stack*

35. What is the time complexity of the destructor, **~***Queue*, for a queue whose individual elements are allocated dynamically as shown in the Figure below?

    a. $O(\log_2 n)$
    b. $O(1)$
    c. $O(n \log_2 n)$
    d. $O(n)$

```
struct QueueException{
    QueueException(char* m)
    {   cout<<endl<<"I am the Queue and I am " << m << "." << endl;
    }
};
template <class T>
class Queue {
    struct Element {
        Element* newer;                    //Points to the next newer element
        T data;                            //Value of type T stored in this element
        Element(Element* n,T d):newer(n),data(d){}//Constructor
        Element(T t):newer(0),v(t){}       //Constructor
        Element(){}                        //Constructor
    };
    Element* oldest;                       //Points to the oldest element
    Element* newest;                       //Points to the newest element
    int count;                             //Records the number of elements
    void Kill(Element* e){...}             //Reclaims storage for all elements
public:
    Queue();                               //Constructor
    ~Queue();                              //Destructor
    bool IsEmpty(void){...}                //Is the Queue empty?
    bool IsFull(void){...}                 //Is the Queue full?
    void Enq(T v) {...}                    //Insert a value, v, on the Queue
    T Deq(void){...}                       //Remove the oldest value on the Queue
    int Count(void){...}                   //Return the number of elements on the Queue
};
```

Figure 35. class *Queue*

36. Select the expressions in infix notation equivalent to the expression tree in the Figure below.

    a. **1-2/3*4*5+6/7**
    b. **(1-2)/3*(4*5+6/7)**
    c. *** / - 1 2 3 + * 4 5 / 6 7**
    d. **1 2 – 3 / 4 5 * 6 7 / + ***

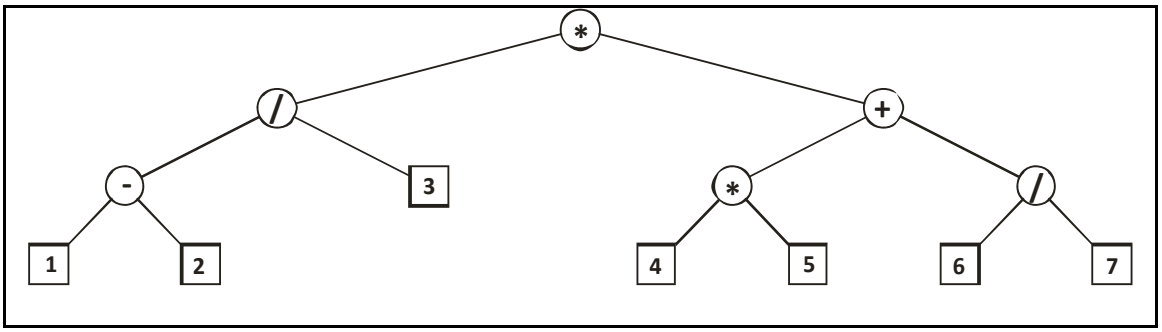

Figure 36. An expression tree

37. What is the time complexity of function *Push* for a stack whose individual elements are allocated dynamically as shown in the Figure below?

    a. $O(\log_2 n)$
    b. $O(1)$
    c. $O(n \log_2 n)$
    d. $O(n)$

```
struct StackException{
    StackException(char* m)
    {   cout<<endl<<"I am the Stack and I am " << m << "." << endl;
    }
};
template <class T>
class Stack {
    struct Element {
        Element* prev;              //Points to the previous element on the Stack
        T data;                     //Data of type T stored in this element
        Element(Element* p,T d):prev(p),data(d){}   //Constructor
        Element(T d):data(d){}      //Constructor, initialize
        Element(){}                 //Constructor, initialize no member data
    };
    Element* tos;                   //Points to the element on top of the Stack
    void Kill(Element* e) {...}     //Reclaims storage for all elements on the Stack
public:
    Stack():...{...}                //Constructor
    ~Stack(){}                      //Destructor
    bool IsEmpty(void){...}         //Is the Stack empty?
    bool IsFull(void) {...}         //Is the Stack full?
    void Push(T d) {...}            //Push a value, d, on the Stack
    T Pop(void) {...}               //Pop a value from the Stack
};
```

Figure 37. class *Stack*

38. Which of the following is a valid implementation of function *Kill* for a stack whose elements are individually stored in dynamically allocated elements as shown in the Figure below?

a.
```
void Kill(Element* e)
{   while (e) {
        e=e->prev;
        Element* p=e;
        delete p;
    }
}
```

b.
```
void Kill(Element* e)
{   while (e) {
        Element* p=e;
        e=e->prev;
        delete e;
    }
}
```

c.
```
void Kill(Element* e)
{   if (e) Kill(e->prev);
    delete e;
}
```

d.
```
void Kill(Element* e)
{   if (!e) return;
    Kill(e->prev);
    delete e;
}
```

```
struct StackException{
    StackException(char* m)
    {   cout<<endl<<"I am the Stack and I am " << m << "." << endl;
    }
};
template <class T>
class Stack {
    struct Element {
        Element* prev;              //Points to the previous element on the Stack
        T data;                     //Data of type T stored in this element
        Element(Element* p,T d):prev(p),data(d){}  //Constructor
        Element(T d):data(d){}      //Constructor, initialize
        Element(){}                 //Constructor, initialize no member data
    };
    Element* tos;                   //Points to the element on top of the Stack
    void Kill(Element* e) {...}     //Reclaims storage for all elements on the Stack
public:
    Stack():...{...}                //Constructor
    ~Stack(){}                      //Destructor
    bool IsEmpty(void){...}         //Is the Stack empty?
    bool IsFull(void){...}          //Is the Stack full?
    void Push(T d) {...}            //Push a value, d, on the Stack
    T Pop(void){...}                //Pop a value from the Stack
};
```

Figure 38. class *Stack*

39. Which of the following sequences is a pre-order traversal of the binary search tree in the Figure below?

    a. **M F A I T Q**
    b. **A I F Q T M**
    c. none of the other alternatives
    d. **A F I M Q T**

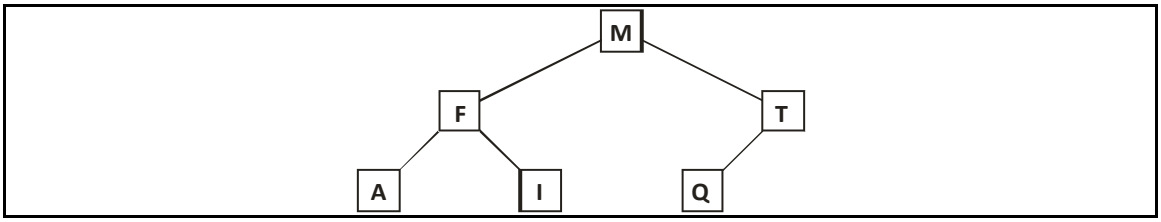

Figure 39. Binary tree

40. Which of the following sequences is a post-order traversal of the binary search tree in the Figure below?

    a. **A I F Q T M**
    b. **A F I M Q T**
    c. **M F A I T Q**
    d. none of the other alternatives

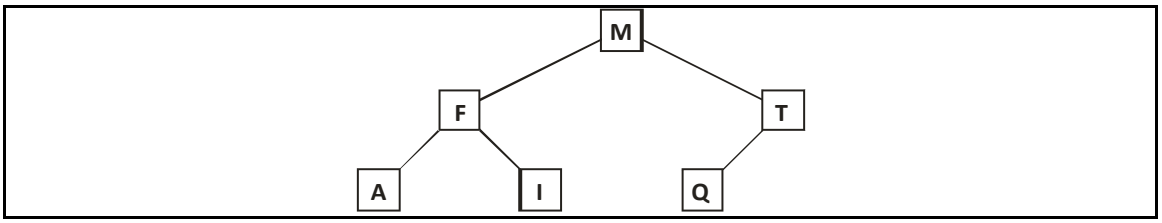

Figure 40. Binary tree

41. What is the time complexity function of function given in Figure below? You may complete the table provided to compute your answer.

    a. $\boldsymbol{T(n) = \frac{3}{2}n^2 + \frac{11}{2}n + 3}$
    b. $\boldsymbol{T(n) = 3n^2 + 5n + 3}$
    c. $\boldsymbol{T(n) = \frac{3}{2}n^2 + \frac{5}{2}n + 3}$
    d. $\boldsymbol{T(n) = 3n^2 + 11n + 3}$

```
int sum=0;
for (int a=0;a<n;a++)
    for (int b=0;b<a;b++)
        sum++;
```

Figure 41. Code Fragment

Line	Code	Cost
1	**int** *sum*=**0;**	
2	**int** *a*=**0;**	
3	**while (***a***<n) {**	
4	**int** *b*=**0;**	
5	**while (***b***<***a***) {**	
6	*sum***++;**	
7	*b***++;**	
8	**}**	
9	*a***++;**	
10	**}**	

42. What is the smallest integer value n_0 that makes $T(n) = O(n^2)$ where $T(n) = 3n^2 + 5n + 3$ and $c = 4$?

    a. 7
    b. 6
    c. 8
    d. 5

43. What is the time complexity function of function given in Figure below? You may complete the table provided to compute your answer.

    a. $\boldsymbol{T(n) = \begin{cases} n = 0, T(0) = 1 \\ n > 0, T(n) = 3\log_2(n) + 1 \end{cases}}$
    b. $\boldsymbol{T(n) = \begin{cases} n = 0, T(0) = 1 \\ n > 0, T(n) = 3(\log_2(n) + 1) + 1 \end{cases}}$
    c. $\boldsymbol{T(n) = \begin{cases} n = 0, T(0) = 1 \\ n > 0, T(n) = 3(\log_2(n) + 1) \end{cases}}$

d. $\boldsymbol{T(n)} = \begin{cases} \boldsymbol{n = 0, T(0) = 1} \\ \boldsymbol{n > 0, T(n) = 3\log_2(n)} \end{cases}$

```
while (n>0) n=n/2;
```

Figure 41. Code Fragment

Line	Code	Cost
1	while (n>0) {	
2	n=n/2;	
3	}	

44. What is the smallest integer value n_0 that makes $T(n) = O(\log_2(n))$ where $T(n) = 3\log_2(n) + 1$ and $c = 4$?

    a. 2
    b. 4
    c. 5
    d. 3

45. In what order do the keys Anna, Beatrice, Colette, Dee, Edith, Felicia, and Grace need to be inserted into a binary tree to guarantee that the tree has a height no greater than two (2)?

    a. Anna, Beatrice, Colette, Dee, Edith, Felicia, Grace
    b. Beatrice, Anna, Felicia, Colette, Edith, Grace, Dee
    c. Grace, Felicia, Edith, Dee, Colette, Anna
    d. Dee, Beatrice, Felicia, Anna, Colette, Edith, Grace

46. What is the best class to implement a page replacement policy such that the oldest page is replaced?

    a. Queue
    b. Ordered list
    c. Stack
    d. Heap

47. How many comparisons are required to insert the integer 16 into a list implemented by allocating one element for every integer, having links to both the smaller and larger elements, having a sentinel element containing an integer larger than any on the list, connecting all the links in both directions so that a circle is maintained? The list contains the following integer 1, 2, 4, 8, 32, and 64 excluding the sentinel element.

    a. 4
    b. 5
    c. 3
    d. 6

48. How many comparisons are required to insert the integer 20 into a binary tree of minimum height that contains the following integers: 4, 8, 12, 16, 24, and 28?

    a. 2
    b. 1
    c. 3
    d. 4

49. What is the maximum number of comparisons required to insert a key in a binary tree of minimum height that has 729 keys in the tree?

    a. 10
    b. 8
    c. 11
    d. 9

50. What is the length of the longest path from the root to a leaf is a tree containing 729 keys where each node can have up to four (4) children and the tree has the minimum height?

    a. 6
    b. 4
    c. 5
    d. 3