

Examples on Dynamic Memory Management

Q1)

```
template <class T>
class MM
{
private:
    T x;
    MM<T>* m;
    T* t;
public:
    MM(T Xin, MM<T>* Min=NULL)
    {
        x=Xin;
        m=Min;
        if(Min==NULL)
            t=new T(Xin);
        else
            t=new T(Min->x);
    };
    MM(const MM<T>& MMin);
    ~MM();
}
```

(a) Draw a diagram that shows the data structures created by the following statements.

```
MM<int> p(3), *q;
```

```
q = new MM<int>(5,&p);
```

(b) Implement the copy constructor and destructor.

Q2)

```
1- #include <iostream.h>
2-
3- class MyClass{
4-     private:
5-         char *c;
6-     public:
7-         MyClass(const int& n);
8-         char& Put(const int n);
9-         char Get(const int n);
10-    };
11-
12- MyClass::MyClass(const int& n)
13-    { c= new char[n] };
14-
15- char& MyClass::Put(const int& n)
16-    { return c[n]; };
17-
18- char& MyClass::Get(const int& n)
19-    { return c[n]; };
20-
21- void MyFn(MyClass& m1)
22-    {
23-        MyClass *mc;
24-
25-        mc = new MyClass(10);
26-        mc->Put(3) = 'a';
27-        m1.Put(3) = mc->Get(3);
28-    };
29-
30- main()
31- {
32-     int n;
33-     MyClass m1(20);
34-
35-     cin >> n;
36-     MyFn(m1);
37-     cout << m1.Get(3);
38- };
```

What is the major programming error related to dynamic memory usage in this program?
Indicate how you would correct this error by giving the line numbers of the lines you would delete, if necessary, and adding new code, if necessary.

Q3) Consider the following C++ class declaration:

```
class Z
{
private:
    int *z1; int *z2;
public:
    void Z(const int x1, x2);
    void Z(const Z &x);
    int *first (void) {return z1};
    int *second (void) {return z2};
}
```

(a) Assuming that a complete implementation of this class, exactly as it is declared here, is available, draw the constructed data structures after the following program sequence is executed:

```
...
Z *zp;
zp = new Z(3,5);
Z a(6, *(zp->first() ) ), b=a, c(0,0);
c = *zp;
delete zp;
```

(b) Give an appropriate implementation for the destructor function for this class.

Q4) Consider the following C++ class declaration:

```
template <class T>
class MT2
{
private: int n; T *p;
public: MT2(const T &m1, int nn=0)
    {
        n=nn;
        if (n>0) p= new T[n];
        for (int i=0; i<n; i++) *(p+i) =m1;
    };
    MT2(const MT2<T> &m);
    ~MT2(void);
    MT2<T> &operator= (const MT2<T> &mt2obj);
}
```

a) Draw a diagram that show the data structures created by the following C++ statements:

```
MT2<int> A(1), *q;
q=new MT2(7,5);
```

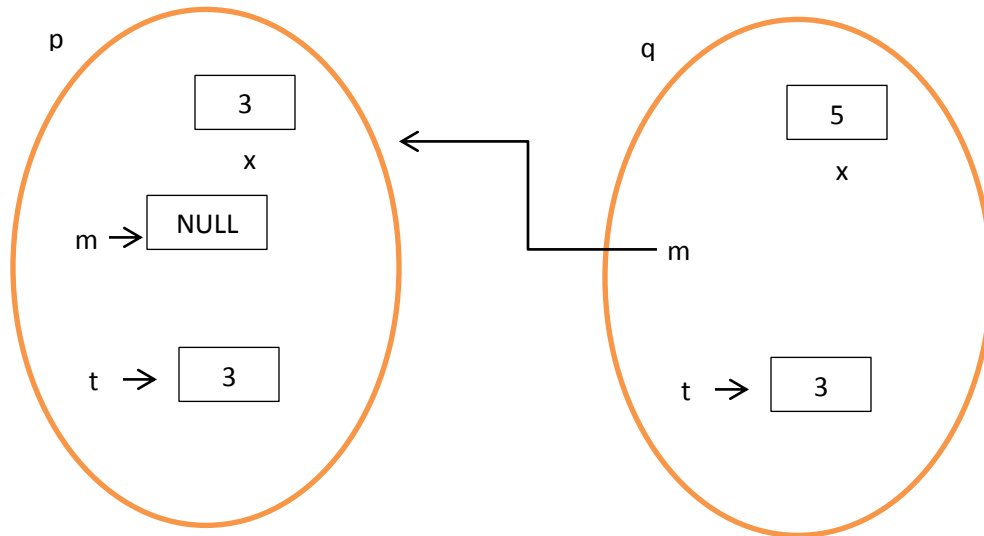
b) Implement the copy constructor and destructor functions of this class.

c) Implement the overloaded assignment operator for this class so that, regardless of the original contents of the left hand side, after assignment, it will be a copy of the right hand side.

Solutions:

Q1)

(a)



(b)

```
template <class T>
MM::MM(const MM<T>& MMin)
{
    x=MMin.x;
    m=MMin.m;
    t=new T(*MMin.t);
};

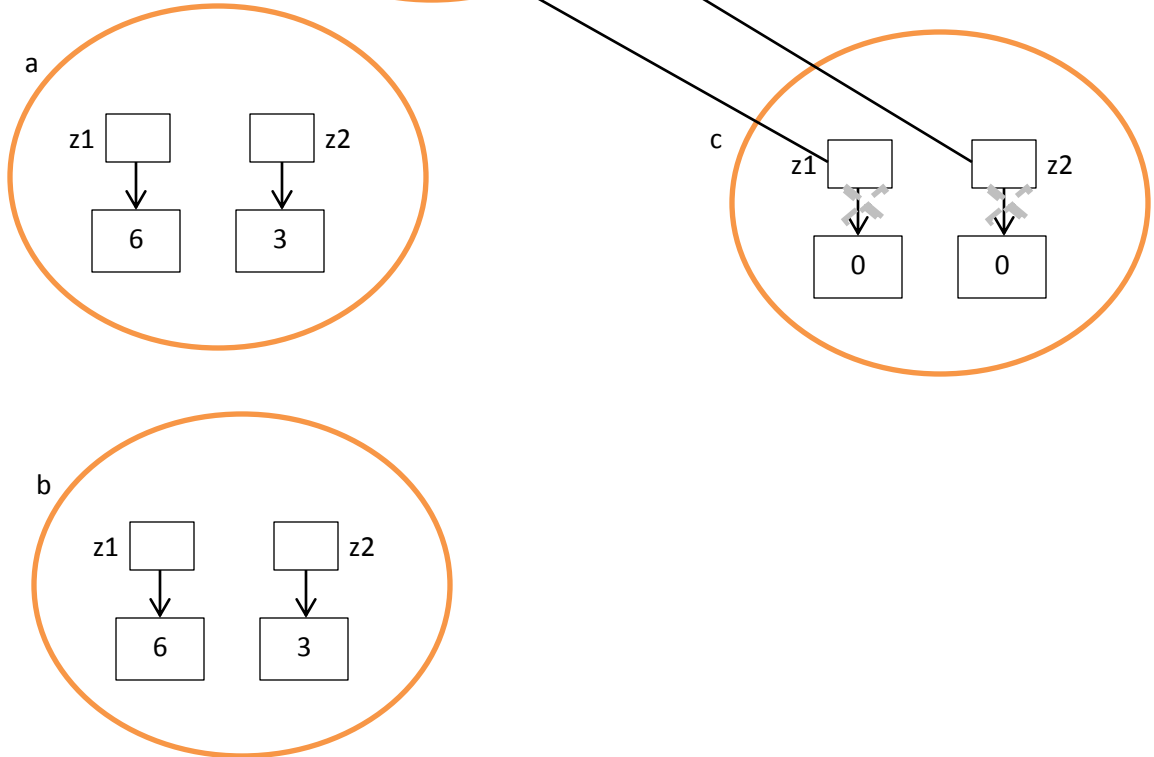
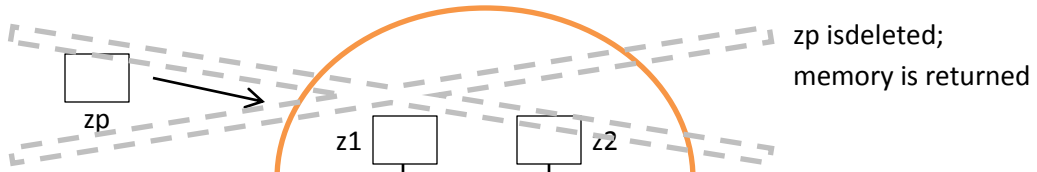
template <class T>
~MM(void)
{
    delete t;
};
```

Q2) Destructor is missing because the class uses dynamic memory to allocate memory space. This space must be returned to the system memory manager.

```
3- class MyClass{
4-     private:
5-         char *c;
6-     public:
7-         MyClass(const int& n);
8-         char& Put(const int n);
9-         char Get(const int n);
10-         ~MyClass();
11- };
```

```
12- MyClass::~~MyClass()
13- { delete []c; };
```

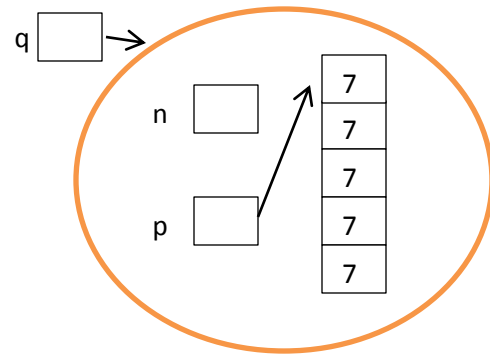
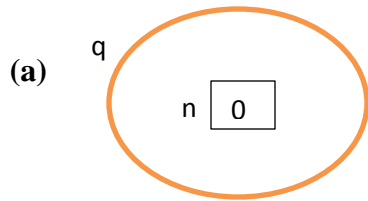
Q3)
(a)



(b)

```
Z::~~Z(void)
{ delete z1; delete z2; }
```

Q4)



(b)

```
template <class T>
MT2<T>::MT2(const MT2<T> &m)
{
    n=m.n;
    if (n) p=new T[n];
    for (int i=0; i<n; i++)
        *(p+i)=*(m.p+i);
}
```

```
template <class T>
MT2<T>::~~MT2(void)
{ if(n) delete p[n]; }
```

(c)

```
template <class T>
MT2<T>&MT2<T>::operator=(const MT2<T> &mt2obj)
{
    if (n) delete p[n];
    n=mt2obj.n;
    if (n) p=new T[n];
    for (int i=0; i<n; i++)
        *(p+i)=*(mt2obj.p+i);
    return *this;
}
```

LINKED LIST

Question 1)

You are given a modified version of the Node Class covered in the lectures in which access to the next pointer is public.

```
template <class T>
class Node
{
    public:
        T data;
        Node<T> *next;

        Node (const T& item, Node<T>* ptrnext = NULL);

        void InsertAfter(Node<T> *p);
        Node<T> *DeleteAfter(void);

        Node<T> *NextNode(void) const;
};
```

Consider the SwapNodes global function template is given as follows:

```
template <class T>
void SwapNodes(Node<T>* head, T data1, T data2)
```

SwapNodes function:

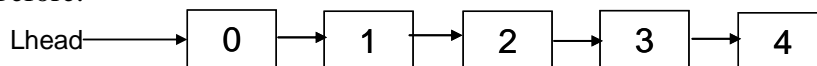
- Takes a linked list pointed by the head pointer. **All of the nodes of this linked list were created dynamically before.**
- Swaps the nodes whose data fields have values of data1 and data2.

Assume that:

- It is guaranteed that the nodes whose data fields have values of data1 and data2 exist in the list
- data1 and data2 appear only once in the list
- Head node never contains data1 or data2

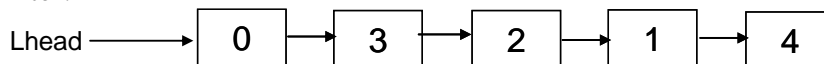
Example:

Before:



SwapNodes (Lhead, 1, 3);

After:



Part a) Implement the SwapNodes function. You must use only the existing nodes of the list. **Do not create any new nodes in your implementation even for temporary use.**

Part b) What is the complexity of the SwapNodes function in

$O()$

$\Omega()$

$\Theta()$

Justify your answers.

Solution (Compiled and works):

Part a)

```

template <class T>
void SwapNodes(Node<T>* head, T data1, T data2)
{
Node<int> * p , *q , *pA, *pB, *temp;

p=head;
//find node with data1
while (p->next->data!=data1)
p=p->next;
pA=p->next;

q=head;
//find node with data2
while (q->next->data!=data2)
q=q->next;
pB=q->next;

//update pointers and swap the nodes
p->next=pB;
q->next=pA;
temp=pA->next;
pA->next=pB->next;
pB->next=temp;

}

```

WONT WORK:

```

p->next=pB;
temp=pA->next;
q->next=pA;
pA->next=pB->next;
pB->next=temp;

```

```

temp=pA->next;
p->next=pB;
q->next=pA;
pA->next=pB->next;
pB->next=temp;

```

Part b) $O(N)$, N is the size of the list. We have to scan the list to find the items. $\Omega(1)$ No Θ

Question 2) For this question, use the following modified Node class definition:

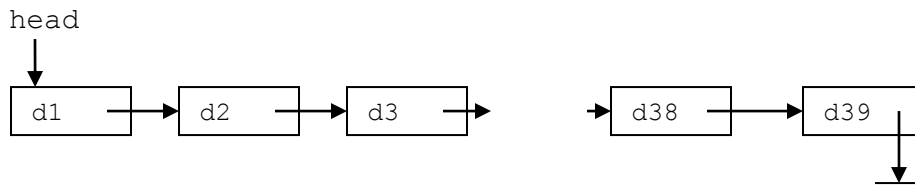
```
template <T>
```

```
class Node {public: T data; Node<T> *link; Node (T d, Node<T> p);}
```

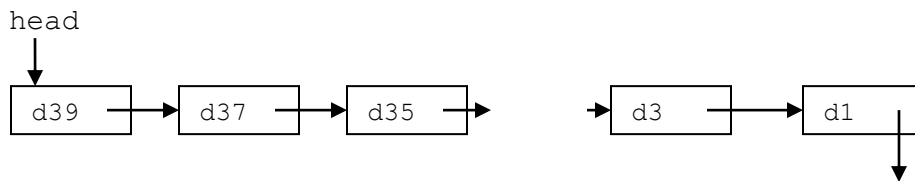
a) (5 pts.) Complete the InsertFront () function that will insert the node pointed by p to the front of the list whose first node is pointed by head.

```
template <class T>
void InsertFront (Node<T> *&head, Node<T> *p)
{if (p==0) return;
 else
     { ___ ; //link new node to list
       ___ ;} //link head to new node
}
```

b) (20 pts.) Now, complete the function Rev_odd () using InsertFront (). Rev_odd will return with head pointing to a new list whose items consist of only the items positioned at the odd numbered nodes of the input list, in reverse order. That is, for the following list:



The call Rev_odd(head); returns with:



```
template <class T>
void Rev_odd (___ head) //pointer to list

{if (___) return; //if input list empty
 else {
     Node<T> *cur = head; Node<T> *next;
     head=0;
     while (cur != 0)

         {___ ;//hold rest of list
           ___ ;//insert odd node
           ___ ;//exit if finished
           ___ ;//skip even node
           ___ ;//move on

         } //end while
     } //end else
 } // end function
```

SOLUTION. For this question, use the following modified Node class definition:

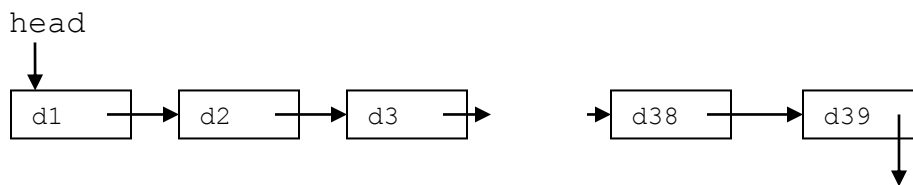
```
template <T>
```

```
class Node {public: T data; Node<T> *link; Node (T d, Node<T> p);}
```

a) (5 pts.) Complete the InsertFront () function that will insert the node pointed by p to the front of the list whose first node is pointed by head.

```
template <class T>
void InsertFront (Node<T> *&head, Node<T> *p)
{if (p==0) return;
 else
    {p -> link = head ;      //link new node to list
      head = p ;}           //link head to new node
}
```

b) (20 pts.) Now, complete the function Rev_odd () using InsertFront (). Rev_odd will return with head pointing to a new list whose items consist of only the odd items of the input list, in reverse order. That is, for the following list:



The call Rev_odd(head) ; returns with:



```
template <class T>
void Rev_odd (Node<T> *&head)           //pointer to list
{if (head == 0) return;                 //if input list empty
 else {   Node<T> *cur = head; Node<T> *next;
          head=0;
          while (cur != 0)

              {next = cur -> link;      //hold rest of list

                InsertFront (head, cur); //insert odd node

                if (next = 0) break;     //exit if finished

                next = next -> link;     //skip even node

                cur = next;              //move on

              } //end while
          } //end else
} // end function
```

Question 3) (18 pts)

You are given a modified version of the Node Class covered in the lectures as follows.

```
template <class T>
class Node
{
public:
    T data;
    Node<T> *next;
    Node (const T& item, Node<T>* ptrnext = NULL);
};
```

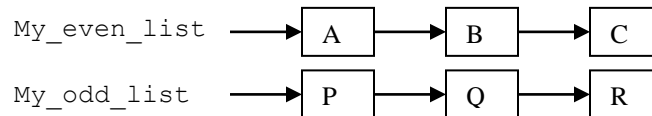
Write a global function

```
Template <class T>
```

```
void Merge(Node<T>*& even_list, Node<T>*& odd_list)
```

which takes two linked lists `even_list` and `odd_list` of the **same size** and merges `odd_list` into `even_list`. As the result of this operation, `even_list` is changed into a new list whose nodes at the odd positions are the nodes of the `odd_list` and nodes at the even positions are the nodes of the original `even_list`. See the figure for an example:

Before Merge (Node<T>*& My_even_list, Node<T>*& My_odd_list)



After Merge (Node<T>*& My_even_list, Node<T>*& My_odd_list)



Fill in the boxes and follow the program comments.

```
template <class T>
void Merge(Node<T>*& even_list, Node<T>*& odd_list)
{
    Node<T>* curr_even= even_list;
    Node<T>* curr_odd= odd_list;
    Node<T>* next_even = curr_even->next();
    Node<T>* next_odd = curr_odd->next();
    if(next_even==NULL) //the lists have one node each
```

```
    curr_even->next=curr_odd;
```

```
    else{
        while(next_even!=NULL) //process all of the nodes in the lists
```

```
        {
            curr_even->next= curr_odd;
            curr_even= next_even;
            next_even= next_even->next;

            curr_odd->next= curr_even;
            curr_odd= next_odd;
            next_odd= next_odd->next;
        }
```

```
    }
    odd_list=NULL;
}
```

Question 4) (25 pts)

Given the node class declaration in the lectures below:

```
template <class T>
class Node
{
private:
Node <T> *next; // next part is a pointer to nodes of this type
public :
T data; // data part is public
// constructor
Node (const T &item, Node<T>* ptrNext=0);

//Insert the node pointed by p after the current node
void InsertAfter(Node<T> *p);
//Delete the node after the current node
Node <T> *DeleteAfter(void);

//get address of next node
Node<T> *NextNode(void) const;
}
```

Part a) 5 points Implement the functions

```
void Node::InsertAfter(Node<T> *p)
{
```

```
    p->next=next
    next=p;
```

```

}
Node <T> * Node::DeleteAfter(void) /*delete node following the current node. Do
not dellocate the memory of the deleted node, just return its address */
{
```

```
    //save address of node to be deleted
    Node <T> *tempPtr=next;
    //if no successor, return NULL
    if (next==NULL)
        return NULL;
    //delete next node by copying its nextptr to the
    //nextptr of current node
    next=tempPtr->next;
    //return pointer to deleted node
    return tempPtr;
```

```
}
```

Part b) You are given a linked list that consists of integers and the function

`void Arrange2 (Node <T>* &head)` . This function rearranges the nodes of the linked list such that the nodes with even integers are at the beginning of the list, followed by the nodes with odd integers.

The order of occurrence among the group of nodes that give the same remainder should be maintained as in the original list.

Example:

Original linked list data: 12, 43, 36, 3, 90,14,2, 67

After calling Arrange2:

12, 36, 90, 14, 2, 43, 3, 67

Implement Arrange2 by filling in the blanks and boxes of the function below according to the given comments.

Hint: mod operation in C++

16 mod 2 : 16%2

template <class T>

void Arrange2 (Node <T>* &head)

{

//declare the variables required to move through the list and perform

//the rearrangement

Node<T> *currPtr=head, *prevPtr=head;

Node<T> *oddhead=NULL;

Node<T> *oddcptr=NULL;

//return if list is empty

if

(currPtr==NULL)

return;

//divide the list into 2, keep even items in the original list, move //odd items to another linked list

while

(currPtr !=NULL)

{

if (currPtr->data%2==1)

{

if (oddhead==NULL)

{

oddhead=GetNode (currPtr->data) ;

oddcptr=oddhead;

}

else

{

oddcptr->InsertAfter (GetNode (currPtr->data)) ;

oddcptr=oddcptr->NextNode () ;

}

if (currPtr==head)

head=head->NextNode () ;

else

prevPtr->DeleteAfter () ;

}

prevPtr=currPtr;

currPtr=currPtr->NextNode () ;

}

//Finalize the function to get the rearranged list

oddcptr=oddhead;

while (oddcptr !=NULL)

{

prevPtr->InsertAfter (GetNode (oddcptr->data)) ;

prevPtr=prevPtr->NextNode () ;

oddcptr=oddcptr->NextNode () ;

}

}

Examples on Linked List:

Q1)

You are given a Linked List that contains items of type T. Nodes are ordered such that stored items are ordered in increasing order.

You are given an item, you create a new node with this item and insert the new node in the correct position such that the list maintains its order.

What is the complexity of this operation in terms of N. (N is the # of items in the list)

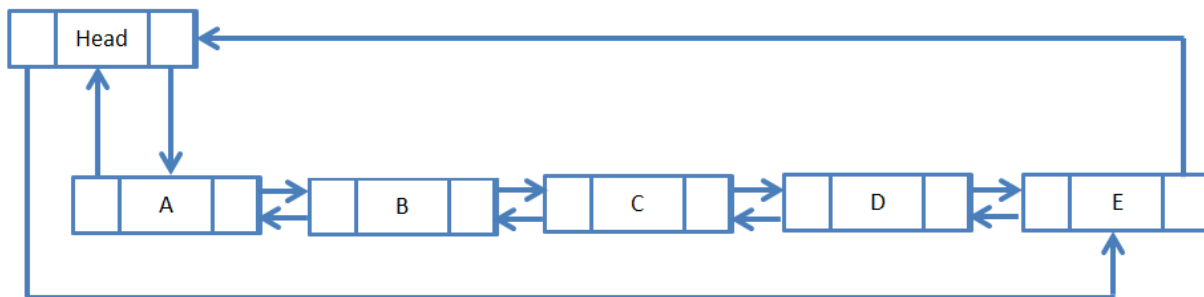
Q2)

```
template <class T>
class nodedouble
{
    private:
        nodedouble<T> *prev;
        nodedouble<T> *next;
    public:
        nodedouble(const T &item, nodedouble<T> *ptrPrev = NULL,
                    nodedouble<T> *ptrNext = NULL);
        void InsertAfter(nodedouble<T> *p);
        nodedouble<T> * DeleteAfter(void);
        nodedouble<T> * NextNode() const;
}
```

(a) Implement “*InsertAfter*” which inserts the node pointed by *p* after the node on which it is applied.

(b) Write a template function “*CreateNode*” that dynamically creates a node of *nodedouble* class and initialize it.

(c)



Write a piece of code that uses functions implemented in (a) and (b) in order to get the list structure given above.

(d) For the Linked List above, draw the resulting linked list of the following code.

```
nodedouble<char> head, *p;  
p = head.next;  
head.next->next->prev = &head;  
head.next = head.next->next;  
p->prev = NULL;  
p->next = NULL;
```

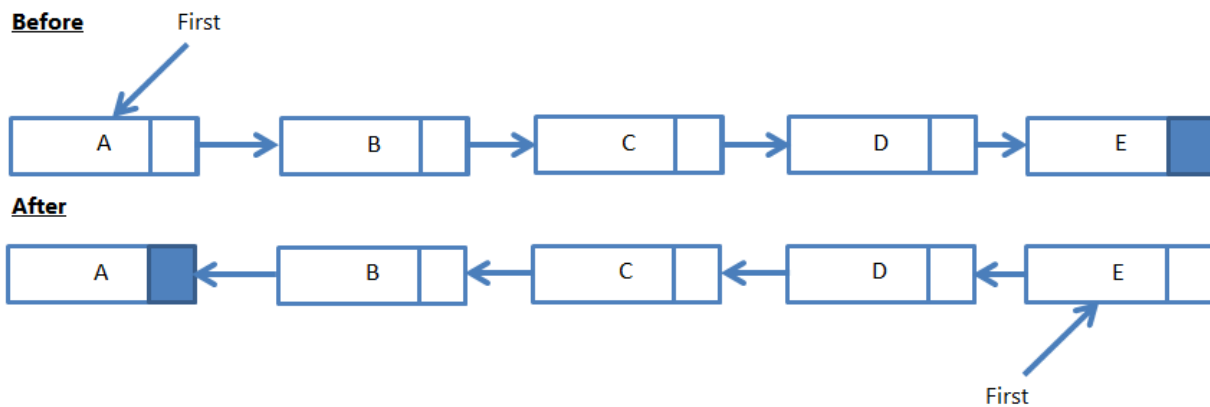
Q3)

Given the NodeClass; (note that Node<T>* next is public), implement a Global function

```
template <class T>
```

```
void Invert(Node<T>* &first);
```

which inverts a given linked list pointed by first by changing only the next pointers.



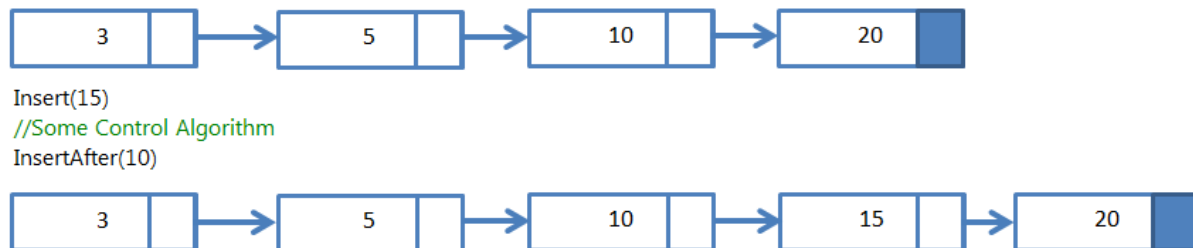
Solutions:

Q1)

The complexity of the operation is $O(N)$ because we can traverse the Linked List only beginning from the head node and progressing node by node.

If we want to insert something in the list, only insertion method for the node is InsertAfter method, as it is described in lecture notes.

i.e.



Q2)

(a) The solution is:

```
void nodedouble<T>::InsertAfter(nodedouble<T> *p)
{
    p->next = next;
    next = p;
    p->next->prev = p;
    p->previous = this;
}
```

OR

```
void nodedouble<T>::InsertAfter(nodedouble<T> *p)
{
    p->next = next;
    next->prev = p;
    p->prev = this;
    next = p;
}
```


(b) The solution is:

```
nodedouble<T>* CreateNode(const T &item, nodedouble<T> *nextPtr = NULL,
                          nodedouble<T> *prevPtr = NULL)
{
    nodedouble<T> *newNode;
    newNode = new nodedouble<T>(item, nextPtr, prevPtr);
    if(newNode==NULL)
    {
        cerr<<"No Memory";
        exit(1);
    }
    return newNode;
}
```

(c) The solution is:

```
nodedouble<char> *head;
head = CreateNode(' ',head,head);
a = CreateNode('A',head,head);
head->next = a;
head->prev = a;
b = CreateNode('B');
a->InsertAfter(b);
c = CreateNode('C');
b->InsertAfter(c);
d = CreateNode('D');
c->InsertAfter(d);
e = CreateNode('E');
d->InsertAfter(e);
e->next = head;
head->prev = e;
```

If this is not a public function, no direct access to *prev* and *next*.
Then;

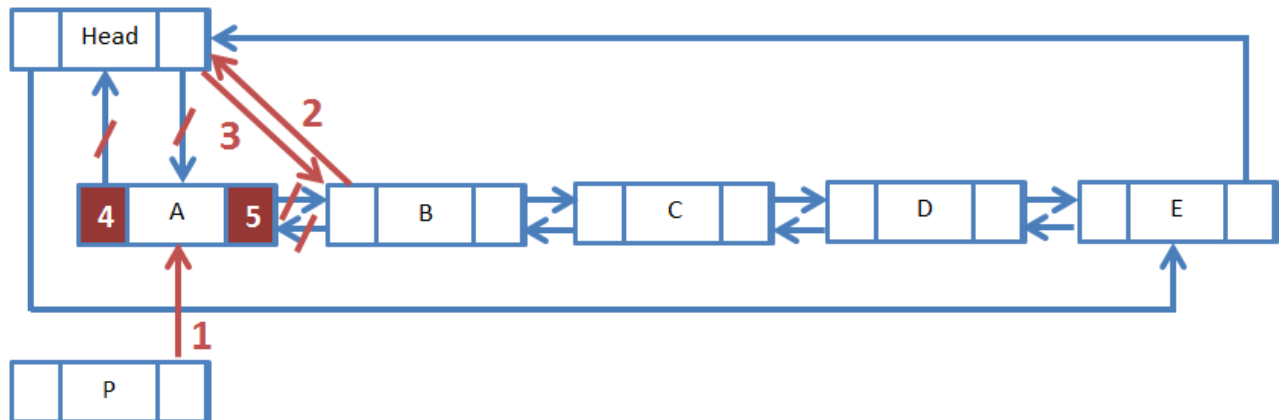
```
a = CreateNode('A');
b = CreateNode('B');
a->InsertAfter(b);
c = CreateNode('C');
b->InsertAfter(c);
d = CreateNode('D');
c->InsertAfter(d);
```

```

e = CreateNode('E');
d->InsertAfter(e);
head = CreateNode(' ');
head->InsertAfter(a);
e->InsertAfter(head);

```

(d) By doing so, we separated Node A from the rest of the linked list.



```

nodedouble<char> *p;
p = head->next; .....1
head->next->next->prev = &head; .....2
head->next = head->next->next; .....3
p->prev = NULL; .....4
p->next = NULL; .....5

```

Q3)

```

void Invert(Node<T>* &first)
{
    Node<T> *temp1, *temp2, *temp3;           //create temporary pointers
    temp1 = first;
    temp2 = NULL;
    temp3 = NULL;
    while(temp1->next != NULL)
    {
        temp3 = temp2;
        temp2 = temp1;
        temp1 = temp1->next;
        temp2->next = temp3;
    }
    temp1->next = temp2;
    first = temp1;
}

```

TREES

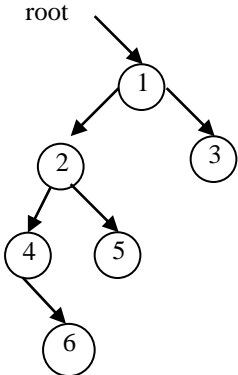
Question 1)

```
template <class T>
class TreeNode
{private:
    TreeNode<T> *left;
    TreeNode<T> *right;
public:
    T data;
    //constructor
        TreeNode(const T &item, TreeNode<T> *lptr=NULL, TreeNode<T>
                                *rptr=NULL);

    //access methods for the pointer fields
        TreeNode<T>* Left(void) const;
        TreeNode<T>* Right(void) const;
};

int treeFunc1( TreeNode<int> *t ) {
    if ( t == NULL )
        return 0;
    else {
        cout<< "(";
        int count = 1;
        count = count + treeFunc1(t->Left());
        count = count + treeFunc1(t->Right());
        cout <<"D:"<< t->data <<"C:"<<count<<" ";
        return count;
    }
}

int treeFunc2( TreeNode<int> *t ) {
    if ( t == NULL )
        return 0;
    else {
        int count,countL,countR;
        countL = treeFunc2(t->Left());
        countR = treeFunc2(t->Right());
        count=t->data+countL+countR;
        cout<<"D:"<<t-
>data<<"CL:"<<countL<<"CR:"<<countR<<"C:"<<count<<endl; //endl is new line
        return count;
    }
}
```

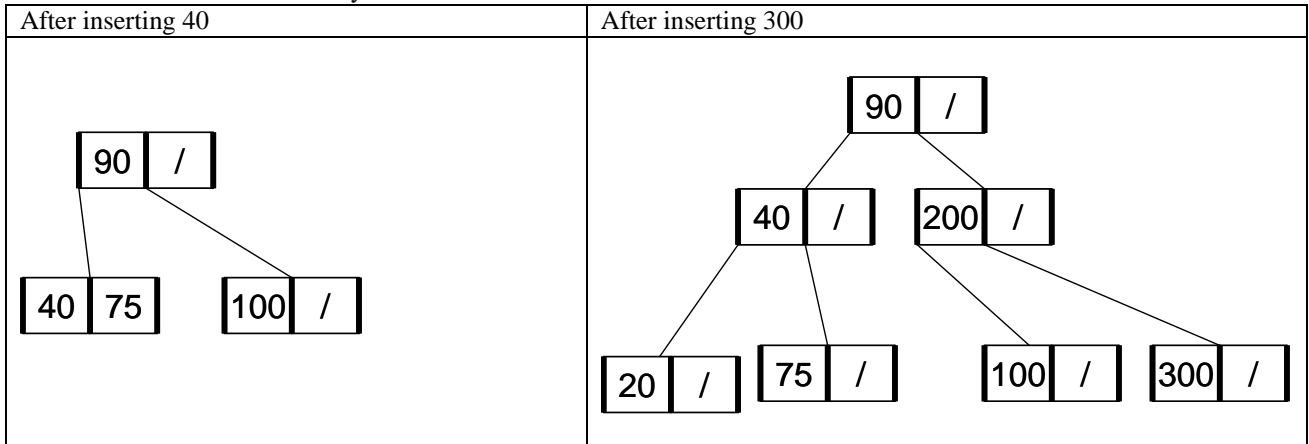
	<p>a) write the output produced when TreeFunc1 (root) is executed:</p> <p>(((D:6:C:1)D:4:C:2) (D:5:C:1)D:2:C:4) (D:3:C:1)D:1:C:6)</p> <p>b) write the output produced when TreeFunc2 (root) is executed:</p> <p>D:6CL:0CR:0C:6 D:4CL:0CR:6C:10 D:5CL:0CR:0C:5 D:2CL:10CR:5C:17 D:3CL:0CR:0C:3 D:1CL:17CR:3C:21</p>
---	--

Question 2) (25 pts)

a) (7 pts.) Insert the following keys in a B-Tree of order 3 in the order they are given.

100,75,90,40,20,200,300

Draw the state of the tree after you insert 40 and 300.



b) (18 pts.) Using the **TreeNode** class in Q3 complete the following **recursive** function (also complete the blank in the function header)

FindKey which searches a given key in a given **binary search tree** that stores integers. The root node of the tree is pointed by MyTree. If key is found, FindKey returns with the pointer to the node that stores the key in argument KeyPointer. If the key is not found, or if MyTree is empty, FindKey returns with NULL value in KeyPointer.

Assume that all items that are stored in the tree exist only once.

```
void FindKey(TreeNode<int> *MyTree, int key, _____KeyPointer)
{
    if (MyTree==NULL)
```

```
    {
        KeyPointer= NULL;
        return;
    }
```

```
    else{
        //stopping condition1: visit the node to check the key match
```

```
    If (MyTree ->data==key)
    {
        KeyPointer= MyTree;
        return;
    }
```

```
    else{
        //If there is no match
        // stopping condition2: if you reach a leaf node, stop the search
        unsuccessfully
```

```
        if(MyTree ->Left()==NULL&& MyTree ->Right()==NULL)
            { KeyPointer= NULL;
              return;}
```

```
    else
        //recursively descend to the correct direction according to the key
```

```
        { if(MyTree ->data>key)
            FindKey (MyTree ->Left(),key);
          else
            FindKey (MyTree ->Right(),key);
        }
```

```
    }
```

Question 3) (25 pts)

a) Given the TreeNode class covered in lectures:

```
template <class T>
class TreeNode
{private:
    TreeNode<T> *left;
    TreeNode<T> *right;
public:
    T data;
    TreeNode(const T &item, TreeNode<T> *lptr=NULL, TreeNode<T>
    *rptr=NULL);
    TreeNode<T>* Left(void) const;
    TreeNode<T>* Right(void) const;
};
```

Write a **recursive** function

```
template <class T>
void int BSTSearch(TreeNode<T> *t, _____level, T key)
```

which takes a given binary search tree and a key. If the key exists in the tree, the function returns the level of the node that the key is found. If the key does not exist, the function returns -1.

Solution:

```
template <class T>
void BSTSearch(TreeNode<T> *t, int& level, T key)
{
    if (t!=NULL)
    {
        if(t->data==key)
            return;
        else
        {
            if(t->Left()==NULL&& t->Right()==NULL)
            {
                level= -1;
                return;
            }
            else if(t->data>key)
            {
                level++;
                BSTSearch(t->Left(),level, key);
            }
            else
            {
                level++;
                BSTSearch(t->Right(),level, key);
            }
        }
    }
}
```

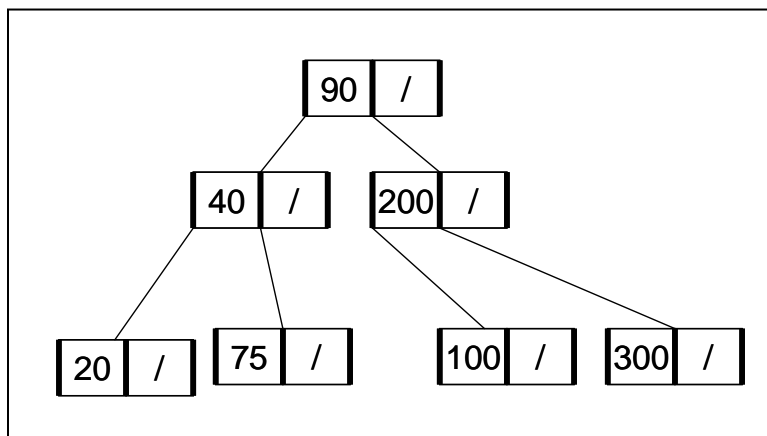
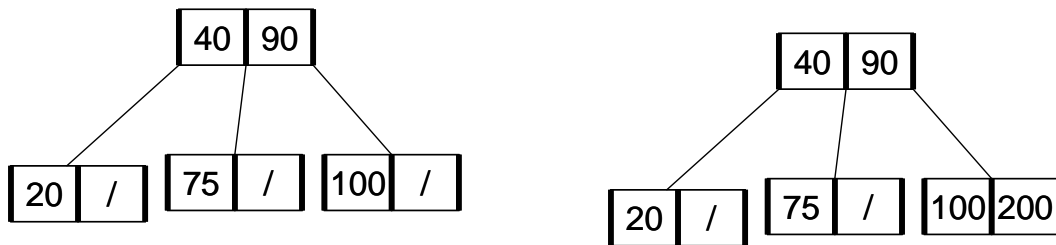
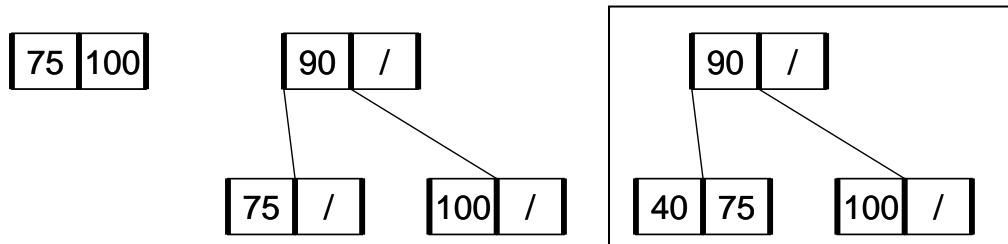
Question 4) (25 pts)

a) Insert the following keys in a B-Tree of order 3 in the order they are given.

100,75,90,40,20,200,300

Draw the state of the tree after you insert 40 and 300.

Solution:



Question 5)

Use the following `TreeNode` class covered in lectures

```
template <class T>
class TreeNode
{private:
    TreeNode<T> *left;
    TreeNode<T> *right;
public:
    T data;
    TreeNode(const T &item, TreeNode<T> *lptr=NULL, TreeNode<T>
    *rptr=NULL);
    TreeNode<T>* Left(void) const;
    TreeNode<T>* Right(void) const;
};
```

a) Write a **recursive** function

```
int TSum (TreeNode<int> *t)
```

which takes a given binary tree which stores positive integers (no 0 and no negative integers) and returns the sum of all stored integers in that tree. If the tree is empty the function returns a 0.

b) Complete the following **recursive** function

```
void TSumLess(TreeNode<int> *MyTree, int key, int& score)
{
    score++;
    ...
}
```

which searches a given key in a given binary search tree that stores positive integers as defined above. The root node of the tree is pointed by `MyTree`. If the key exists in the tree, the function prints the sum of all items that are smaller than or equal to the key on the screen. If the key does not exist, the function does not print anything. Assume that all integers stored in the tree are distinct.

Requirement: After the following function call `myscore` has the **minimum value possible** for the given pointer to `ThisTree` and given key `thiskey`.

```
int myscore=0;
TSumLess(ThisTree, thiskey, myscore);
```

Hint: You might use `TSum` in `TSumLess` (Even you could not solve part a)

SOLUTION

a)

```
template <class T>
int TSum(TreeNode<T> *t)
{
    int sumLeft, sumRight, sum;
    if (t==NULL)
        sum=0; // if empty, sum=0
    else
    {
        sumLeft=TSum(t->Left());
        sumRight=TSum(t->Right());
        sum=t->data+sumLeft+sumRight;
    }
    return sum;
}
```

b)

```
void TSumLess(TreeNode<int> *MyTree, int key, int& score)
{
    score++;
    if (MyTree!=NULL)
    {
        if(MyTree ->data==key)
        {
            cout<<TSum(t);
            return;
        }
        else
        {
            if(MyTree ->Left()==NULL&& MyTree ->Right()==NULL)
                return;
            else if(MyTree ->data>key)
                TSumLess(MyTree ->Left(),key, score);
            else
                TSumLess(MyTree ->Right(),key, score);
        }
    }
}
```