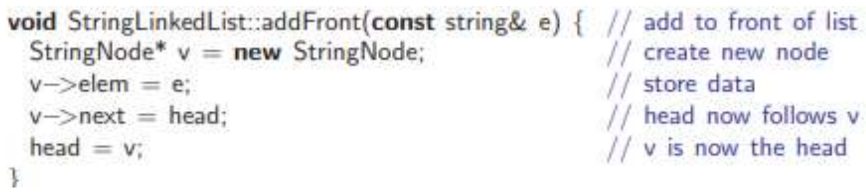
Content:

```
/*-----------------------Linked List----------------------*/
#include <iostream>
using namespace std;
typedef struct Node
{
    int data;
    Node* next;
} *nodeptr;
class LinkedList
{
    private:
        nodeptr head;
        nodeptr curr;
        nodeptr temp;
    public:
        LinkedList();
        int getNode(int pos);
        void addNode(int d);
        void insertNode(int p, int d);
        void deleteNodeByPos(int p);
        void deleteNodeByData(int d);
        void printList();
};
LinkedList::LinkedList()
{
    head = NULL;
    curr = NULL;
    temp = NULL;
}
void LinkedList::addNode(int d)
{
    nodeptr n = new Node;
    n->data = d;
    n->next = nullptr;
    if(head != NULL)
    {
        curr= head;
        while(curr->next != NULL)
        {
            curr = curr->next;
        }
        curr->next = n;
    }
    else
    {
        cout<<"List is empty..."<<endl;
        head = n;
    }
}
```

Another way: the one written above adds node to the end of the list, the one written below, adds a node to the front of the list. Use this one below in linked stack so that all operations would be O(N).



```
void StringLinkedList::addFront(const string& e) {   // add to front of list
  StringNode* v = new StringNode;                    // create new node
  v->elem = e;                                       // store data
  v->next = head;                                    // head now follows v
  head = v;                                          // v is now the head
}
```

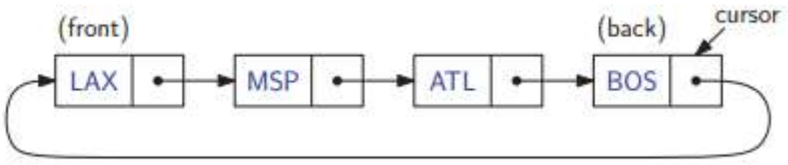
```
int LinkedList::getNode(int pos)
{
    if(head != NULL)
    {
        curr= head;
        for(int i = 0; i<pos; i++)
        {
            if(curr->next != NULL)
                curr = curr->next;
            else
            {
                cout<<"List does not have a node in position "<<pos<<endl;
                return 0;
            }
        }

        return curr->data;
    }
    else
    {
        cout<<"List is empty!"<<endl;
        return 0;
    }
}

void LinkedList::insertNode(int p, int d)
{
    nodeptr n = new Node;
    n->data = d;

    if(head != NULL)
    {
        curr= head;
        for(int i = 0; i<p-1; i++)
        {
            if(curr->next != NULL)
                curr = curr->next;
            else
            {
                cout<<"List does not have a node in position "<<p<<endl;
                break;
            }
        }
        n->next = curr->next;
        curr->next = n;

    }
    else if(head == NULL && p==1)
    {
        cout<<"List is empty..."<<endl;
        head = n;
    }
    else
        cout<<"List does not have a node in position "<<p<<endl;

}
```

```
void LinkedList::deleteNodeByPos(int p)
{

    if(head != NULL)
    {
        nodeptr delptr;
        curr= head;

        for(int i = 0; i<p-2; i++)
        {
            if(curr->next != NULL)
                curr = curr->next;
            else
            {
                cout<<"List does not have a node in position "<<p<<endl;
                break;
            }
        }
        temp = curr->next;
        curr->next = temp->next;
        delptr = temp;

        delete delptr;
    }
    else
        cout<<"List does not have a node in position "<<p<<endl;


}

void LinkedList::printList()
{
    if(head != NULL)
    {
        curr= head;

        while(curr->next != NULL)
        {
            cout<<curr->data<<"  ";
            curr = curr->next;
        }
        cout<<curr->data;
    }
    else
        cout<<"List is empty!"<<endl;

}
```

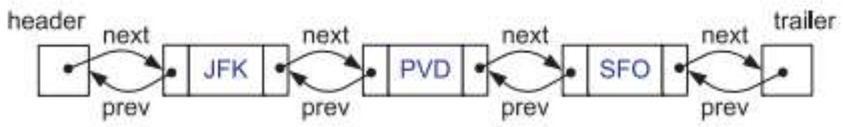
```
void LinkedList::deleteNodeByData(int d)
{
    nodeptr delptr;
    bool dFlag=0;
    curr = head;
    temp = curr;
    if(curr == NULL)
    {
        cout<<"Linked List is empty!";
        return;
    }
    else if(curr!=NULL && curr->data == d)
    {
        dFlag = 1;
        delptr = curr;
        delete delptr;
    }
    else
    {
        while(curr !=NULL)
        {
            if(curr->next != NULL)
                curr=curr->next;
            if(curr->data == d)
            {
                delptr = curr;
                curr=curr->next;
                temp->next = curr;
                dFlag = 1;
                delete delptr;
            }
            temp = curr;
        }
    }
    if(!dFlag)
        cout<<"No node with data ("<<d<<") was in the list!\n";
    else
        cout<<"Data ("<<d<<") deleted successfully!\n";
}
int main()
{
    LinkedList L1;
    L1.printList();
    L1.addNode(1);
    L1.addNode(2);
    L1.addNode(3);
    L1.addNode(4);
    L1.addNode(5);
    L1.addNode(6);
    L1.deleteNodeByData(6);
    L1.printList();
    return 0;
    /*Output:
        List is empty!
        List is empty...
        Data (6) deleted successfully!
        1 2 3 4 5
    */
}

/*------------------------End of Linked List-----------------------*/
```

```
/*---------------------------Circular Linked List------------------------*/
#include <iostream>
using namespace std;

template<typename T>
class CLinkedList
{
    private:
        typedef struct Node
        {
            T data;
            Node* next;
        } *nodeptr;
        nodeptr cursor;
    public:
        CLinkedList();
        bool empty();
        const T& front();
        const T& back();
        void advance();
        void remove();
        void addNode(const T& d);
};
template<typename T>
CLinkedList<T>::CLinkedList()
{
    cursor = NULL;
}
template<typename T>
void CLinkedList<T>::addNode(const T& d)
{
    nodeptr n = new Node;
    n->data = d;
    if(cursor == NULL)
    {
        n->next = n;
        cursor = n;
    }
    else
    {
        n->next = cursor->next;
        cursor->next = n;
    }
}
template <typename T>
bool CLinkedList<T>::empty()
{
    return (cursor==NULL);
}
template <typename T>
const T& CLinkedList<T>::back()
{
    return (cursor->data);
}
template <typename T>
const T& CLinkedList<T>::front()
{
    return (cursor->next->data);
}
```

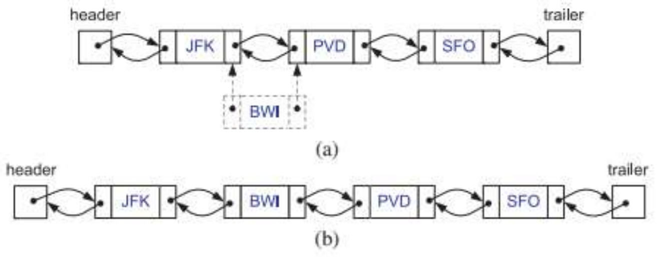
```
template <typename T>
void CLinkedList<T>::advance()
{
    cursor=cursor->next;
}

template <typename T>
void CLinkedList<T>::remove()
{
    nodeptr old=cursor->next;
    if(old == cursor)
    {
        cursor = NULL;
    }
    else
    {
        cursor->next = old->next;
    }
    delete old;
}
```



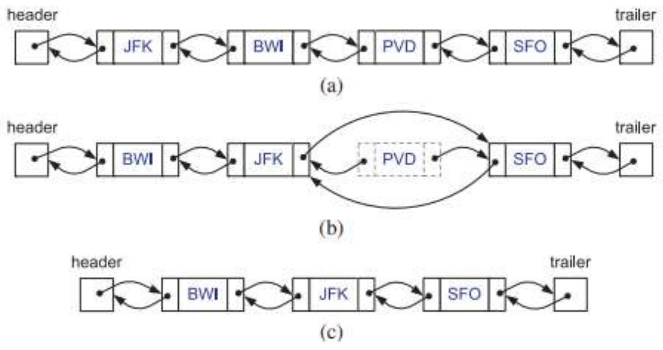
```
/*---------------------------End of Circular Linked List------------------------*/

/*--------------------Doubly linked list--------------------*/
#include <iostream>
using namespace std;

typedef int Elem;
class DLinkedList
{
    private:
        typedef struct Node
        {
            Elem data;
            Node* next;
            Node* prev;
        }*nodeptr;
        nodeptr header;
        nodeptr trailer;

    protected:
        void add(nodeptr n, const Elem& d); //add node before n
        void remove(nodeptr n);             //remove n
    public:
        DLinkedList(); // constructor
        ~DLinkedList(); // destructor
        bool empty() const; // is list empty?
        const Elem& front() const; // get front element
        const Elem& back() const; // get back element
        void addFront(const Elem& e); // add to front of list
        void addBack(const Elem& e); // add to back of list
        void removeFront(); // remove from front
        void removeBack(); // remove from back
};
```



```
DLinkedList::DLinkedList()
{
    header = new Node;
    trailer = new Node;
    header->next = trailer;
    trailer->prev = header;
}
DLinkedList::~DLinkedList()
{
    while(!empty())
    {
        removeFront();
    }
    delete header;
    delete trailer;
}
void DLinkedList::add(nodeptr n, const Elem& d)
{
    nodeptr n_new = new Node;
    n_new->data = d;
    n_new->prev = n->prev;
    n_new->next = n;
    n->prev->next = n_new;
    n->prev =n_new;
}
void DLinkedList::remove(nodeptr n)
{
    nodeptr temp = n;
    n->prev->next = n->next;
    n->next->prev = n->prev;
    delete temp;
}
void DLinkedList::addFront(const Elem& e) // add to front of list
{
    add(header->next, e);
}
void DLinkedList::addBack(const Elem& e) // add to back of list
{
    add(trailer, e);
}
void DLinkedList::removeFront() // remove from front
{
    remove(header->next);
}
void DLinkedList::removeBack() // remove from back
{
    remove(trailer->prev);
}
bool DLinkedList::empty() const
{
    return (header->next == trailer);
}
const Elem& DLinkedList::front() const
{
    return header->next->data;
}
const Elem& DLinkedList::back() const
{
    return trailer->prev->data;
}
```



Insertion



Deletion

```
int main()
{
    DLinkedList DL;
    DL.addBack(1);
    cout<<DL.front();
    return 0;
    /*Output:
      1
    */

}
/*--------------------End of Doubly linked list--------------------*/

/*------------------Array Stack----------------------*/
#include <iostream>
using namespace std;

template <typename T>
class ArrayStack
{
    enum{DEF_CAPACITY = 100};
    public:
        ArrayStack(int cap = DEF_CAPACITY);
        const int size();
        bool empty();
        const T& top();
        void push(const T& d);
        void pop();
    private:
        T* S;
        int capacity;
        int t;

};

template <typename T>
ArrayStack<T>::ArrayStack(int cap) : S(new T[cap]), t(-1), capacity(cap)
{}

template <typename T>
const int ArrayStack<T>::size()
{
    return t+1;
}

template <typename T>
bool ArrayStack<T>::empty()
{
    return (t==-1);
}

template <typename T>
const T& ArrayStack<T>::top()
{
    try
    {
        if(t==-1)
        {
            throw("Stack is empty!");
        }
        return S[t];
    }
    catch(const char* cerr)
    {
        cout<<cerr<<endl;
    }
    return 0;

}

template <typename T>
void ArrayStack<T>::push(const T& d)
{
    try
    {
        if(size() == capacity)
        {
            throw("Stack is full!");
        }
        t++;
        S[t] = d;
    }
    catch(const char* cerr)
    {
        cout<<cerr<<endl;
    }

}

template <typename T>
void ArrayStack<T>::pop()
{
    try
    {
        if(empty())
        {
            throw("Stack is empty!");
        }
        t--;
    }
    catch(const char* cerr)
    {
        cout<<cerr<<'\n';
    }

}

int getMax_recursive(ArrayStack<int> S)
{
    int top = S.top();
    S.pop();
    if(S.empty())
    {
        S.push(top);
        return top;
    }
    else
    {
        int max = getMax_recursive(S);
        if(top>max)
        {
            int temp = max;
            max = top;
            S.push(temp);
            S.push(max);
        }
        else
        {
            S.pop();
            S.push(top);
            S.push(max);
        }
        return max;
    }
}
int main()
{
    ArrayStack<int>my_stack;
    my_stack.pop();
    cout<<"the stack size: "<<my_stack.size()<<endl;
    cout<<"is the stack empty? "<<my_stack.empty()<<endl;
    my_stack.push(12);
    my_stack.push(134);
    cout<<"the stack size: "<<my_stack.size()<<endl;
    cout<<"is the stack empty? "<<my_stack.empty()<<endl;
    cout<<my_stack.top()<<endl;
    my_stack.pop();
    cout<<my_stack.top()<<endl;
    cout<<"\n";
    ArrayStack<int> A;
    A.push(7);
    A.push(13);
    cout << A.top() << endl; A.pop();
    A.push(9);
    cout << A.top() << endl;
    cout << A.top() << endl; A.pop();
    ArrayStack<string> B(10);
    B.push("Bob");
    B.push("Alice");
    cout << B.top() << endl;
    B.pop();
    B.push("Eve");
    cout << B.top() << endl;
    /*Output:
    -------------------
    Stack is empty!
    the stack size: 0
    is the stack empty? 1
    the stack size: 2
    is the stack empty? 0
    134
    12

    13
    9
    9
    Alice
    Eve
    */
}

/*------------------End of Array Stack----------------------*/
```

```
/*---------------------------Linked Stack-------------------------------*/
#include <iostream>
using namespace std;

template <typename T>
class TLinkedList
{
    private:
        typedef struct Node
        {
            T data;
            Node* next;
        } *nodeptr;
        nodeptr head;
        nodeptr curr;
        nodeptr temp;
    public:
        TLinkedList();
        T& Front();
        void addFront(T d);
        void removeFront();
        void printList();
};

template <typename T>
TLinkedList<T>::TLinkedList()
{
    head = NULL;
    curr = NULL;
    temp = NULL;
}
template <typename T>
void TLinkedList<T>::addFront(T d)
{
    nodeptr n = new Node;
    n->data = d;
    if(head == NULL)
    {
        head = n;
        head->next = NULL;
    }
    else
    {
        n->next = head;
        head = n;
    }
}
template <typename T>
void TLinkedList<T>::removeFront()
{
    nodeptr temp = head;
    head = temp->next;
    delete temp;
}
template <typename T>
T& TLinkedList<T>::Front()
{
        return head->data;
}
```

```
template <typename T>
void TLinkedList<T>::printList()
{
    if(head != NULL)
    {
        curr= head;

        while(curr->next != NULL)
        {
            cout<<curr->data<<"  ";
            curr = curr->next;
        }
        cout<<curr->data;
    }
    else
        cout<<"List is empty!"<<endl;
}

template <typename T>
class LinkedStack
{
    private:
        TLinkedList<T> LS;
        int n;

    public:
        LinkedStack();
        int size();
        bool empty();
        void push(const T& d);
        const T top();
        void pop();
};

template <typename T>
LinkedStack<T>::LinkedStack() : LS(), n(0)
{}

template <typename T>
int LinkedStack<T>::size()
{
    return n;
}

template <typename T>
bool LinkedStack<T>::empty()
{
    return (n==0);
}

template <typename T>
void LinkedStack<T>::push(const T& d)
{
    LS.addFront(d);
    n++;
}
```

```
template <typename T>
void LinkedStack<T>::pop()
{
    LS.removeFront();
    n--;
}


template <typename T>
const T LinkedStack<T>::top()
{
    return(LS.Front());
}

int main()
{
    cout<<"Testing template linked stack :\n";
    LinkedStack<int> L2;
    if(L2.empty())
        cout<<"Stack is empty\n";
    else
        cout<<"stack not empty\n";
    L2.push(2);
    L2.push(3);
    L2.push(4);
    L2.push(5);
    L2.push(6);
    L2.push(7);
    cout<<L2.top()<<endl;
    L2.pop();
    cout<<L2.top()<<endl;

    /*Output:
    Testing template linked stack :
    Stack is empty
    7
    6
    */
    return 0;
}

/*---------------------------End of Linked Stack-------------------------------*/
```

```cpp
/*--------------------Linked Queue----------------------*/
#include <iostream>
using namespace std;

template<typename T>
class CLinkedList
{
    private:
        typedef struct Node
        {
            T data;
            Node* next;
        } *nodeptr;
        nodeptr cursor;
    public:
        CLinkedList();
        bool empty();
        const T& front();
        const T& back();
        void advance();
        void remove();
        void addNode(const T& d);
};

template<typename T>
CLinkedList<T>::CLinkedList()
{
    cursor = NULL;
}

template<typename T>
void CLinkedList<T>::addNode(const T& d) //add to the front - after cursor
{
    nodeptr n = new Node;
    n->data = d;
    if(cursor == NULL)
    {
        n->next = n;
        cursor = n;
    }
    else
    {
        n->next = cursor->next;
        cursor->next = n;
    }
}

template <typename T>
bool CLinkedList<T>::empty()
{
    return (cursor==NULL);
}


template <typename T>
const T& CLinkedList<T>::back()
{
    return (cursor->data);
}
```

```cpp
template <typename T>
const T& CLinkedList<T>::front()
{
    return (cursor->next->data);
}

template <typename T>
void CLinkedList<T>::advance()
{
    cursor=cursor->next;
}

template <typename T>
void CLinkedList<T>::remove() //delete node after cursor - first node
{
    nodeptr old=cursor->next;
    if(old == cursor)
    {
        cursor = NULL;
    }
    else
    {
        cursor->next = old->next;
    }
    delete old;
}

template<typename T>
class LinkedQueue
{
    private:
        CLinkedList<T> CL; //Circular Linked List
        int n;
    public:
        LinkedQueue();
        int size();
        bool empty();
        const T& front();
        void enqueue(const T& d);
        void dequeue();
};

template<typename T>
LinkedQueue<T>::LinkedQueue() : CL(), n(0)
{}

template<typename T>
int LinkedQueue<T>::size()
{
    return n;
}

template<typename T>
bool LinkedQueue<T>::empty()
{
    return (n==0);
}
```

```
template<typename T>
const T& LinkedQueue<T>::front()
{
    return (CL.front());
}

template<typename T>
void LinkedQueue<T>::enqueue(const T& d)
{
    CL.addNode(d);
    CL.advance();
    n++;
}

template<typename T>
void LinkedQueue<T>::dequeue()
{
    CL.remove();
    n--;
}

int main()
{
    cout<<"\nTesting Linked Queue: \n";
    LinkedQueue<int> L1;
    if(L1.empty())
        cout<<"Queue is empty\n";
    L1.enqueue(1);
    L1.enqueue(2);
    L1.enqueue(3);
    cout<<L1.front()<<endl;
    cout<<"The size is : "<<L1.size()<<endl;
    L1.dequeue();
    L1.dequeue();
    cout<<L1.front()<<endl;
    L1.enqueue(4);
    L1.dequeue();
    cout<<L1.front();
    return 0;
    /*Output:
    Testing Linked Queue:
    Queue is empty
    1
    The size is : 3
    3
    4
    */
}

/*--------------------End of Linked Queue----------------------*/
```

```
/*---------------------------Queue using circular array--------------------------*/
#include <iostream>
using namespace std;

template<typename T>
class Queue
{
    enum{DEF_CAPACITY = 100};
    private:
        T* Q;
        int f,r,n;
    public:
        Queue(int cap = DEF_CAPACITY);
        int size();
        bool empty();
        const T& front();
        void enqueue(const T& d);
        void dequeue();
        void printQueue();
};

template<typename T>
Queue<T>::Queue(int cap) : Q(new T[cap]), f(0), r(0), n(0)
{}

template<typename T>
int Queue<T>::size()
{
    return n;
}

template<typename T>
bool Queue<T>::empty()
{
    return (n==0);
}
template<typename T>
const T& Queue<T>::front()
{
    if(!empty())
    {
        return Q[f];
    }
    else
    {
        cout<<"Queue is empty!\n";
        return NULL;
    }
}
template <typename T>
void Queue<T>::dequeue()
{
    if(!empty())
    {
        f = (f+1)%DEF_CAPACITY;
        n--;
    }
    else
        cout<<"Queue is empty!\n";

}
```

```
template <typename T>
void Queue<T>::enqueue(const T& d)
{
    if(size()<DEF_CAPACITY)
    {
        Q[r] = d;
        r=(r+1) % DEF_CAPACITY;
        n++;
    }
    else
    {
        cout<<"Queue is full!\n";
    }
}
template <typename T>
void Queue<T>::printQueue()
{
    if(!empty())
    {
        for(int i = 0; i<size(); i++)
        {
            cout<<Q[f+i]<<" ";
        }
        cout<<endl;
    }
    else
    {
        cout<<"Queue is full!\n";
    }
}

int main()
{
    Queue<int> Q1;
    Q1.enqueue(5);
    Q1.enqueue(3);
    cout<<Q1.front()<<endl;
    Q1.printQueue();
    cout<<"The size is "<<Q1.size()<<endl;
    Q1.dequeue();
    cout<<Q1.front()<<endl;
    Q1.printQueue();
    cout<<Q1.empty()<<'\n';
    return 0;
    /*Output:
    5
    5 3
    The size is 2
    3
    3
    0
    */
}

/*---------------------------End of Queue using circular array--------------------------*/
```

```
/*----------------------Deque------------------------*/
#include <iostream>
using namespace std;

typedef int Elem;
class DLinkedList
{
    private:
    typedef struct Node
    {
        Elem data;
        Node* next;
        Node* prev;
    }*nodeptr;
    nodeptr header;
    nodeptr trailer;
    protected:
        void add(nodeptr n, const Elem& d);
        void remove(nodeptr n);
    public:
        DLinkedList(); // constructor
        ~DLinkedList(); // destructor
        bool empty() ; // is list empty?
        const Elem& Lfront() ; // get front element
        const Elem& Lback() ; // get back element
        void LaddFront(const Elem& e); // add to front of list
        void LaddBack(const Elem& e); // add to back of list
        void LremoveFront(); // remove from front
        void LremoveBack(); // remove from back
};

DLinkedList::DLinkedList()
{
    header = new Node;
    trailer = new Node;
    header->next = trailer;
    trailer->prev = header;
}

DLinkedList::~DLinkedList()
{
    while(!empty())
    {
        LremoveFront();
    }
    delete header;
    delete trailer;
}

void DLinkedList::add(nodeptr n, const Elem& d)
{
    nodeptr u = new Node; u->data = d; // create a new node for e
    u->next = n; // link u in between n
    u->prev = n->prev; // . . .and n->prev
    n->prev->next = u;
    n->prev = u;
}
```

```
void DLinkedList::remove(nodeptr n)
{
    nodeptr u = n->prev; // predecessor
    nodeptr w = n->next; // successor
    u->next = w; // unlink n from list
    w->prev = u;
    delete n;
}

void DLinkedList::LaddFront(const Elem& e) // add to front of list
{
    add(header->next, e);
}
void DLinkedList::LaddBack(const Elem& e) // add to back of list
{
    add(trailer, e);
}
void DLinkedList::LremoveFront() // remove from front
{
    remove(header->next);
}
void DLinkedList::LremoveBack() // remove from back
{
    remove(trailer->prev);
}

bool DLinkedList::empty()
{
    return(header->next == NULL);
}

const Elem& DLinkedList::Lfront()
{
    return (header->next->data);
}

const Elem& DLinkedList::Lback()
{
    return(trailer->prev->data);
}

class LinkedDeque
{
    private:
        int n;
        DLinkedList DL;

    public:
        LinkedDeque();
        int size();
        bool empty();
        void front();
        void back();
        void insertFront(const Elem& d);
        void insertBack(const Elem& d);
        void removeFront();
        void removeBack();
};

LinkedDeque::LinkedDeque() : DL(), n(0)
{}
```

```
int LinkedDeque::size()
{return n;}
bool LinkedDeque::empty()
{
    return (n==0);
}
void LinkedDeque::front() //you can use return
{
    cout<<DL.Lfront()<<endl;
}
void LinkedDeque::back() //you can use return
{
    cout<<DL.Lback()<<endl;
}
void LinkedDeque::insertFront(const Elem& d)
{
    DL.LaddFront(d);
    n++;
}
void LinkedDeque::insertBack(const Elem& d)
{
    DL.LaddBack(d);
    n++;
}
void LinkedDeque::removeFront()
{
    DL.LremoveFront();
    n--;
}
void LinkedDeque::removeBack()
{
    DL.LremoveBack();
    n--;
}
int main()
{
    DLinkedList D1;
    D1.LaddFront(3);
    D1.LaddFront(5);
    cout<<D1.Lfront()<<endl;
    D1.LremoveFront();
    D1.LaddBack(7);
    cout<<D1.Lback()<<endl;
    LinkedDeque D2;
    D2.insertFront(3);
    D2.insertFront(5);
    D2.front();
    D2.removeFront();
    D2.insertBack(7);
    D2.back();
    return 0;
    /*Output:
    5
    7
    5
    7*/
}
/*----------------------End of Deque------------------------*/
```

```cpp
/*-------------------Array Vector---------------------*/
#include <iostream>
using namespace std;

typedef int Elem;
class ArrayVector
{
    private:
        int capacity;
        int n;
        Elem* A;
    public:
        ArrayVector();
        int size();
        bool empty();
        const Elem& operator[](int i);
        const Elem& at(int i);
        void insert(int i, const Elem& E);
        void erase(int i);
        void reserve(int N);
        void set(int i, const Elem& E);
};

ArrayVector::ArrayVector() : A(NULL), n(0), capacity(0)
{}
int ArrayVector::size()
{
    return n;
}
bool ArrayVector::empty()
{
    return (n==0);
}
const Elem& ArrayVector::operator[](int i)
{
    try
    {
        if(i<0 || i>=n)
            throw("index out of range!");

    }
    catch(const char* cerr)
    {
        cout<<cerr<<endl;
    }
    return A[i];
}
const Elem& ArrayVector::at(int i)
{
    try
    {
        if(i<0 || i>=n)
            throw("index out of range!");
    }
    catch(const char* cerr)
    {
        cout<<cerr<<endl;
    }
    return A[i];
}
```

```
void ArrayVector::insert(int i, const Elem& E)
{
    if(n>=capacity)
    {
        reserve(max(1,2*capacity));
    }
    for(int j = n-1; j>=i; j--)
    {
        A[j+1]=A[j];
    }
    A[i] = E;
    n++;
}
void ArrayVector::erase(int i)
{
    for(int j = i; j<n; j++)
    {
        A[j] = A[j+1];
    }
    n--;
}
void ArrayVector::reserve(int N)
{
    if(capacity>=N)
        return;
    Elem* B = new Elem[N];
    for(int i = 0; i<n; i++)
    {
        B[i] = A[i];
    }
    if(A!=NULL)
        delete [] A;
    A=B;
    capacity = N;
}


void ArrayVector::set(int i, const Elem& E)
{
    try
    {
        if(i<0 || i>=n)
            throw("index out of range!");
        A[i] = E;
    }
    catch(char* cerr)
    {
        cout<<cerr<<endl;
    }
}
int main()
{
    ArrayVector AV;
    AV.insert(0,7);
    AV.insert(0,4);
    cout<<AV.at(1)<<endl;
    cout<<AV[1];
    AV.insert(2,2);
    AV.erase(1);
    AV.insert(1,5);
    AV.insert(1,3);
    AV.insert(4,9);
    cout<<AV.at(2)<<endl;
    AV.set(3,8);
    for(int i=0; i<5; i++)
    {
        cout<<AV[i]<<" ";
    }
    /*Output:
    7
    75
    4 3 5 8 9
    */
    return 0;
}
/*-------------------End of Array Vector----------------------*/

/*--------------------------NodeList-------------------------*/
#include <iostream>
using namespace std;
typedef int T;
class NodeList
{
    private:
        typedef struct Node
        {
            Node* prev;
            Node* next;
            T Elem;
        } *nodeptr;
    public:
        class Iterator
        {
            private:
                nodeptr v;
                Iterator(nodeptr u);
            public:
                T& operator*();
                bool operator==(const Iterator& p);
                bool operator!=(const Iterator& p);
                Iterator& operator++();
                Iterator& operator--();
                friend class NodeList;
        };
    public:
        NodeList();
        int size();
        bool empty();
        Iterator begin();
        Iterator end();
        void insertFront(const T& e);
        void insertBack(const T& e);
        void insert(const Iterator& p, const T& e);
        void eraseFront();
        void eraseBack();
        void erase(const Iterator& p);
    private:
        int n;
        nodeptr header;
        nodeptr trailer;
};
```

```
NodeList::Iterator::Iterator(nodeptr u)
{
    v=u;
}

T& NodeList::Iterator::operator*()
{
    return (v->Elem);
}
bool NodeList::Iterator::operator==(const Iterator& p)
{
    return(v == p.v);
}
bool NodeList::Iterator::operator!=(const Iterator& p)
{
    return(v!=p.v);
}
NodeList::Iterator& NodeList::Iterator::operator++()
{
    v=v->next;
    return *this;
}
NodeList::Iterator& NodeList::Iterator::operator--()
{
    v=v->prev;
    return *this;
}

NodeList::NodeList()
{
    n = 0;
    header = new Node;
    trailer = new Node;
    header->next = trailer;
    trailer->prev = header;
}
int NodeList::size()
{
    return n;
}
bool NodeList::empty()
{
    return (n==0);
}
NodeList::Iterator NodeList::begin()
{

    return Iterator(header->next);
}
NodeList::Iterator NodeList::end()  // end position is just beyond last
{
    return Iterator(trailer);
}
void NodeList::insertFront(const T& e)
{
    insert(begin(), e);
}
void NodeList::insertBack(const T& e)
{
    insert(end(), e);
}
```

```
void NodeList::insert(const Iterator& p, const T& e) //insert before p
{
    n+=1;
    nodeptr v = new Node;
    v->Elem = e;
    nodeptr beforeP = (p.v)->prev;
    v->prev = beforeP;
    beforeP->next = v;
    v->next = p.v;
    p.v->prev = v;
}
void NodeList::eraseFront()
{
    erase(begin());
}
void NodeList::eraseBack()
{
    erase(--end());
}
void NodeList::erase(const Iterator& p)
{
    nodeptr beforeP = p.v->prev;
    nodeptr afterP = p.v->next;
    beforeP->next = afterP;
    afterP->prev = beforeP;
    delete p.v;
    n--;
}

int main()
{
    typedef NodeList::Iterator iterator;
    NodeList N1;
    N1.insertFront(8);
    for(iterator I = N1.begin(); I!=N1.end(); ++I)
    {
        cout<<*I<<" ";
    }
    cout<<endl;
    iterator p = N1.begin();
    cout<<"p: "<<*p<<endl;
    N1.insertBack(5);
    for(iterator I = N1.begin(); I!=N1.end(); ++I)
    {
        cout<<*I<<" ";
    }
    cout<<endl;
    iterator q = p;
    ++q;
    cout<<"q: "<<*q<<endl;
    cout<<(p==N1.begin())<<endl;
    N1.insert(q,3);
    for(iterator I = N1.begin(); I!=N1.end(); ++I)
    {
        cout<<*I<<" ";
    }
    cout<<endl;
    *q = 7;
    for(iterator I = N1.begin(); I!=N1.end(); ++I)
    {
        cout<<*I<<" ";
    }
    cout<<endl;
    N1.insertFront(9);
    for(iterator I = N1.begin(); I!=N1.end(); ++I)
    {
        cout<<*I<<" ";
    }
    cout<<endl;

    N1.eraseBack();
    for(iterator I = N1.begin(); I!=N1.end(); ++I)
    {
        cout<<*I<<" ";
    }
    cout<<endl;

    N1.erase(p);
    for(iterator I = N1.begin(); I!=N1.end(); ++I)
    {
        cout<<*I<<" ";
    }
    cout<<endl;

    N1.eraseFront();
    for(iterator I = N1.begin(); I!=N1.end(); ++I)
    {
        cout<<*I<<" ";
    }
    return 0;
    /*Output:
    8
    p: 8
    8 5
    q: 5
    1
    8 3 5
    8 3 7
    9 8 3 7
    9 8 3
    9 3
    3
    */
}
/*--------------------------End of NodeList-------------------------*/

/*---------------------------------Linked Binary Tree---------------------------------*/
#include <iostream>
#include <list>
#include <queue>
using namespace std;

typedef int Elem;
class BinarySearchTree
{
    protected:
        typedef struct Node
        {
            Node* par;
            Node* left;
            Node* right;
            Elem data;
            Node() : data(), par(NULL), left(NULL), right(NULL)
            {}
        }*nodeptr;
    public:
        class Position
        {
            private:
                nodeptr v;
            public:
                Position(Node* _v = NULL); /* constructor */
                Elem& operator*(); /* get element */
                Position left() const; /* get left child */
                Position right() const; /* get right child */
                Position parent() const; /* get parent */
                bool isRoot() const; /* root of the tree? */
                bool isExternal() const; /* an external node? */
                Position sibling() const; /* return Position to sibling */
                friend class BinarySearchTree; /* give tree access */
        };
        typedef std::list<Position> PositionList;
    private:
        nodeptr _root;
        int n;
    public:
        BinarySearchTree();
        int size();         //number of nodes
        bool empty();       //is the tree empty?
        Position root();    //get root
        PositionList positions() const; //list of nodes
        void addRoot();     //add root to empty tree
        void expandExternal(const Position& p);
        Position removeAboveExternal(const Position& p);
        /* added functions */
        void print_exp_recur(Position p) const; /* print expression recursively */
        float eval_recur(Position p) const ; /* evaluate expression recursively */
        bool isAncestor(Position p1, Position p2) const; /* return true if p1 is an ancestor
of p2 */
        bool isDescendant(Position p1, Position p2) const; /* return true if p1 is a
descendant of p2 */
        PositionList get_path(Position p1, Position p2) const; /* return PositionList
containing a path between p1 and p2 */
        bool isSibling(Position p1, Position p2) const; /* return true if p1 is sibling of p2
*/
        int depth_recur(Position p) const; /* calculates depth of a node recursively */
        int depth(Position p) const; /* calculate depth of a node */
        int height_recur(Position p) const; /* calculate tree height recursively */
        int height(Position p) const; /* calculate tree height */
    protected:
        void preorder(Node* v, PositionList& pl) const; // preorder utility
};

typedef std::list<BinarySearchTree::Position> PositionList;
/*-------------------------Position functions---------------------------------*/
BinarySearchTree::Position::Position(Node* _v) : v(_v)
{}

Elem& BinarySearchTree::Position::operator*()
{
    return v->data;
}
```

```
BinarySearchTree::Position BinarySearchTree::Position::left() const
{
    return Position(v->left);
}

BinarySearchTree::Position BinarySearchTree::Position::right() const
{
    return Position(v->right);
}

BinarySearchTree::Position BinarySearchTree::Position::parent() const
{
    return Position(v->par);
}

bool BinarySearchTree::Position::isRoot() const
{
    return (v->par == NULL);
}

bool BinarySearchTree::Position::isExternal() const
{
    return (v->left == NULL && v->right == NULL);
}

BinarySearchTree::Position BinarySearchTree::Position::sibling() const
{
    if(v->par->left == v)
        return Position(v->par->right);
    else
        return Position(v->par->left);
}

/*-----------------------------------------------------------------------------------*/
/*-------------------------BinarySearchTree functions---------------------------------*/

BinarySearchTree::BinarySearchTree() : _root(NULL), n(0)
{}

int BinarySearchTree::size()
{
    return n;
}
bool BinarySearchTree::empty()
{
    return n==0;
}
BinarySearchTree::Position BinarySearchTree::root()
{
    return Position(_root);
}

PositionList BinarySearchTree::positions() const
{
    PositionList pl;
    preorder(_root, pl);
    return PositionList(pl);
}
```

```
void BinarySearchTree::addRoot()
{
    _root = new Node;
    n=1;
}
void BinarySearchTree::expandExternal(const Position& p)
{
    nodeptr v = p.v;
    v->left = new Node;
    v->right = new Node;
    v->left->par = v;
    v->right->par = v;
    n+=2;
}
BinarySearchTree::Position BinarySearchTree::removeAboveExternal(const Position& p)
{
    Node* w = p.v;
    Node* v = w->par; // get p’s node and parent
    Node* sib = (w == v->left ? v->right : v->left);
    if (v == _root)   // child of root?
    {
        _root = sib; // . . .make sibling root
        sib->par = NULL;
    }
    else
    {
        Node* gpar = v->par; // w’s grandparent
        if (v == gpar->left) gpar->left = sib; // replace parent by sib
        else gpar->right = sib;
        sib->par = gpar;
    }
    delete w;
    delete v; // delete removed nodes
    n -= 2; // two fewer nodes
    return Position(sib);
}

void BinarySearchTree::preorder(Node* v, PositionList& pl) const
{
    pl.push_back(Position(v));
    if(v->left != NULL)
    {
        preorder(v->left, pl);
    }
    if(v->right != NULL)
    {
        preorder(v->right, pl);
    }
}

void BinarySearchTree::print_exp_recur(Position p) const
{
    if(p.isExternal())
    {
        cout<<*p - 48;
    }
    else
    {
        cout<<"(";
        print_exp_recur(p.left());
        if(*p == '+')
```

```
                cout<<"+";
            else if(*p == '-')
                cout<<"-";
            else if(*p == '*')
                cout<<"*";
            else if(*p == '/')
                cout<<"/";
            else if(*p == '%')
                cout<<"%";
            else if(*p>=48 && *p<58)
                cout<<*p-48;
            print_exp_recur(p.right());
            cout<<")";
        }
}

float BinarySearchTree::eval_recur(Position p) const
{
    if(p.isExternal())
    {
        return *p-48;
    }
    else
    {
        if(*p == '+')
            return eval_recur(p.left()) + eval_recur(p.right());
        else if(*p == '-')
            return eval_recur(p.left()) - eval_recur(p.right());
        else if(*p == '*')
            return eval_recur(p.left()) * eval_recur(p.right());
        else if(*p == '/')
            return eval_recur(p.left()) / eval_recur(p.right());
    }
    return 0;
}
bool BinarySearchTree::isAncestor(Position p1, Position p2) const
{
    /*the root node in a binary tree is not considered an ancestor,
    because it has no parent node. An ancestor is defined as any node
    that is on the path from the root node to a given node.
    Since the root node is at the top of the tree and has no parent node,
    it cannot be an ancestor of any other node in the tree. However,
    the root node is considered the "ultimate" ancestor of all the nodes
    in the tree, since all other nodes are descendants of the root node.*/
    nodeptr pp1 = p1.v;
    nodeptr pp2 = p2.v;
    while(pp2!=_root)
    {
        if(pp1 == pp2)
            return true;
        pp2=pp2->par;
    }
    return false;
}
bool BinarySearchTree::isDescendant(Position p1, Position p2) const
{
    if(isAncestor(p2,p1))
        return true;
    else
        return false;
}
```

```
PositionList BinarySearchTree::get_path(Position p1, Position p2) const
{
    PositionList list;
    Position p = p1;
    list.push_back(p);
    while (*p != *p2)
    {
        if(!isAncestor(p.left(), p2) && !isAncestor(p.right(), p2))
        {
            cout<<"No path between p1 and p2\n";
            list.pop_back();
            return list;
        }
        p = (isAncestor(p.left(), p2))? p.left():p.right();
        list.push_back(p);
    }
    return list;
}

bool BinarySearchTree::isSibling(Position p1, Position p2) const
{
    Position sib = p1.sibling();
    if(p2.v == sib.v)
        return true;
    else
        return false;
}

int BinarySearchTree::depth_recur(Position p) const
{
    if(p.isRoot())
    {
        return 0;
    }
    else
    {
        return (1 + depth_recur(p.parent()));
    }
}
int BinarySearchTree::depth(Position p) const
{
    int dep = 0;
    Position pp = p;
    while(!pp.isRoot())
    {
        pp = pp.parent();
        dep++;
    }
    return dep;
}
int BinarySearchTree::height_recur(Position p) const
{
    if(p.isExternal())
        return 0;
    int h = 0;

    h = max(h, height_recur(p.left()));
    h = max(h, height_recur(p.right()));
    return 1+h;
}
```

```
int BinarySearchTree::height(Position p) const
{
    int d = 0;
    queue<Position> posQueue;
    posQueue.push(p);
    while(!posQueue.empty())
    {
        int n = posQueue.size();
        for(int i = 0; i<n;i++)
        {
            Position node = posQueue.front();
            if(!node.isExternal())
            {
                posQueue.push(node.left());
                posQueue.push(node.right());
            }
            posQueue.pop();
        }
        if(posQueue.size())
            d++;
    }
    return d;
}


int main()
{
    BinarySearchTree BT;
    BT.addRoot();

    BinarySearchTree::Position p[15];
    p[0] = BT.root(); //get root position
    for(int i = 0; i<=6; i++)
    {
        BT.expandExternal(p[i]);
        p[i*2+1] = p[i].left();
        p[i*2+2] = p[i].right();
        *p[i] = i+1;
        *p[i*2+1] = i*2+2;
        *p[i*2+2] = i*2+3;
    }

    for(int i =0; i<15; i++) //printing tree level by level
    {
        static int j = 1;
        if(i == 0)
        {
            cout<<*p[i]<<endl;
        }
        else
        {
            j*=2;
            for(int k = 0; k<j; k++)
            {
                cout<<*p[i]<<" ";
                i++;

            }
            i--;
            cout<<endl;
        }
    }
    cout<<"The root is: "<<*BT.root()<<endl;
    cout<<"The depth of the node 15 recursively is: "<<BT.depth_recur(p[14])<<endl;
    cout<<"The depth of the node 15 non-recursively is: "<<BT.depth(p[14])<<endl;
    cout<<"The height of the tree recursively is: "<<BT.height_recur(BT.root())<<endl;
    cout<<"The height of the tree non-recursively is: "<<BT.height(BT.root())<<endl;

    /*Output:
    1
    2 3
    4 5 6 7
    8 9 10 11 12 13 14 15
    The root is: 1
    The depth of the node 15 recursively is: 3
    The depth of the node 15 non-recursively is: 3
    The height of the tree recursively is: 3
    The height of the tree non-recursively is: 3
    */
    return 0;
}
/*---------------------------------End of Linked Binary Tree-------------------------------*/

/*----------------------Euler Tour-------------------------*/
#include <iostream>
#include <list>
using namespace std;

template <typename T, typename R>
class EulerTour
{
    protected:
        struct Result
        {
            R leftResult;
            R rightResult;
            R finalResult;
        };
        typedef BinaryTree<T> BinaryTree;
        typedef typename BinaryTree::Position Position;

    protected:
        const BinaryTree* tree;
    public:
        void initiaize(const BinaryTree& T)
        {
            tree = &T;
        }
    protected:
        int eulerTour(const Position& p) const;
        virtual void visitExternal(const Position& p, Result& r) const{}
        virtual void visitLeft(const Position& p, Result& r) const{}
        virtual void visitRight(const Position& p, Result& r) const{}
        virtual void visitBelow(const Position& p, Result& r) const{}
        virtual Result initResult() const {return r.finalResult;}
        R result(const Result& r) const {return r.finalResult;}
};
```

```
template <typename T, typename R>
int EulerTour<T,R>::eulerTour(const Position& p) const
{
    Result r = initResult();
    if(p.isExternal())
    {
        visitExternal(p,r);
    }
    else
    {
        visitLeft(p,r);
        r.leftResult = eulerTour(p.left());
        visitBelow();
        r.rightResult = eulerTour(p.right());
        visitRight();
    }
    return result(r);
}

template<typename T, typename R>
class EvaluateExpressionTour : public EulerTour<T,R>
{
    protected:
        typedef typename EulerTour<T,R>::BinaryTree BinaryTree;
        typedef typename EulerTour<T,R>::Position Position;
        typedef typename EulerTour<T,R>::Result Result;
    public:
        void execute(const BinaryTree& T)
        {
            initiaize(T);
            cout<<"The value is: "<<eulerTour(T.root())<<"\n";
        }
    protected:
        virtual void visitExternal(const Position& p, Result& r) const
        {
            r.finalResult = *p;
        }
};
/*----------------------End of Euler Tour------------------------*/
/*----------------Priority Queue using sorted list-------------------*/
#include <iostream>
#include <list>
using namespace std;

//Comparators
template <typename T>
class BottomTop
{
    public:
        bool operator()(const T& p, const T& q)
        {
            return (p<q);
        }
};
//T: type of list elements
//C: type of comparator
template <typename T, typename C>
class ListPriorityQueue
{
    private:
        list<T> L;
        C isLess;           //less than comparator
    public:
        int LPQsize();
        bool empty();
        void insert(const T& e);
        T& min();
        void removeMin();
};
template <typename T, typename C>
int ListPriorityQueue<T,C>::LPQsize()
{
    return(L.size());
}
template <typename T, typename C>
bool ListPriorityQueue<T,C>::empty()
{
    return(L.empty());
}
template <typename T, typename C>
void ListPriorityQueue<T,C>::insert(const T& e)
{
    typename std::list<T>::iterator p;
    p = L.begin();
    while(p!=L.end() && (!isLess(e, *p)))
        ++p;
    L.insert(p,e);
}
template <typename T, typename C>
T& ListPriorityQueue<T,C>::min()
{
    return L.front();
}
template <typename T, typename C>
void ListPriorityQueue<T,C>::removeMin()
{
    L.pop_front();
}

int main()
{
    BottomTop<int> BT;
    ListPriorityQueue<int, BottomTop<int>> LPQ;
    LPQ.insert(3);
    LPQ.insert(55);
    cout<<"expected value is: 3 -- output value is: "<<LPQ.min()<<endl;
    LPQ.insert(1);
    LPQ.insert(2);

    LPQ.removeMin();
    cout<<"expected value is: 2 -- output value is: "<<LPQ.min()<<endl;

    return 0;
}
/*-----------------End of Priority Queue using sorted list-------------------*/
```

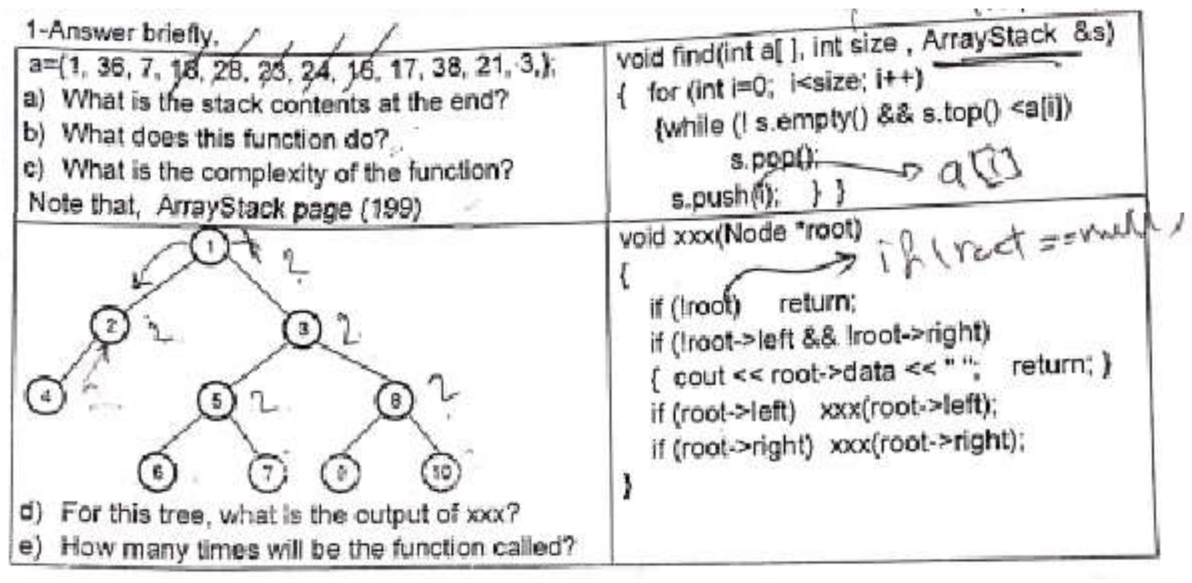
```
/*-----------------Priority Queue using unsorted list-------------------*/
#include <iostream>
#include <list>
using namespace std;
//Comparators
template <typename T>
class BottomTop
{
    public:
        bool operator()(const T& p, const T& q)
        {
            return (p<q);
        }
};
//T: type of list elements
//C: type of comparator
template <typename T, typename C>
class ListPriorityQueue
{
    private:
        list<T> L;
        C isLess;           //less than comparator
    public:
        int LPQsize();
        bool empty();
        void insert(const T& e);
        const T& min();
        void removeMin();
};
template <typename T, typename C>
int ListPriorityQueue<T,C>::LPQsize()
{
    return(L.size());
}
template <typename T, typename C>
bool ListPriorityQueue<T,C>::empty()
{
    return(L.empty());
}
template <typename T, typename C>
void ListPriorityQueue<T,C>::insert(const T& e)
{
    L.push_back(e);
}
template <typename T, typename C>
const T& ListPriorityQueue<T,C>::min()
{
    typename std::list<T>::iterator p;
    typename std::list<T>::iterator min;
    p = L.begin();
    min = p;
    while(p!=L.end())
    {
        if(isLess(*p, *min))
        {
            min = p;
        }
        ++p;
    }
    return *min;
}
```

```
template <typename T, typename C>
void ListPriorityQueue<T,C>::removeMin()
{
    typename std::list<T>::iterator p;
    p = L.begin();
    T min = *p;
    while(p!=L.end())
    {
        if(isLess(*p, min))
        {
            min = *p;
        }
        ++p;
    }
    L.remove(min);
}
int main()
{
    BottomTop<int> BT;
    ListPriorityQueue<int, BottomTop<int>> LPQ;
    LPQ.insert(3);
    LPQ.insert(55);
    cout<<"expected value is: 3 -- output value is: "<<LPQ.min()<<endl;
    LPQ.insert(1);
    LPQ.insert(2);
    LPQ.removeMin();
    cout<<"expected value is: 2 -- output value is: "<<LPQ.min()<<endl;
    return 0;
}
/*-----------------End of Priority Queue using unsorted list-------------------*/

/*--------------Vector complete tree------------*/
#include <iostream>
#include <vector>
using namespace std;

template <typename T>
class VectorCompleteTree
{
    private:
        vector<T> V;
    public:
        typedef typename vector<T>::iterator Position;
    protected:
        Position pos(int i)
        {
            return V.begin() + i;
        }

        int idx(const Position& p) const
        {
            return p - V.begin();
        }
    public:
        VectorCompleteTree() : V(1)
        {}
        int size() const
        {
            return(V.size()-1);
        }
        Position left(const Position& p)
        {
            return pos(idx(p)*2);
        }
        Position right(const Position& p)
        {
            return pos(idx(p)*2 + 1);
        }
        Position parent(const Position& p)
        {
            return pos(idx(p)/2);
        }
        bool hasLeft(const Position& p)
        {
            return (2*idx(p)<=size());
        }
        bool hasRight(const Position& p)
        {
            return ((2*idx(p) + 1)<=size());
        }
        bool isRoot(const Position& p)
        {
            return (idx(p)==1);
        }
        Position root()
        {
            return pos(1);
        }
        Position last()
        {
            return pos(size());
        }
        void addLast(const T& e)
        {
            V.push_back(e);
        }
        void removeLast()
        {
            V.pop_back();
        }
        void swap(const Position& p, const Position& q)
        {
            T e = *p;
            *p = *q;
            *q = e;
        }
};
int main()
{
    VectorCompleteTree<int> VCT;
    VCT.addLast(1);
    cout<<"Expected size is: 1 --- Output size is: "<<VCT.size()<<endl;
    cout<<"Expected output is: 1 --- Output  is: "<<*(VCT.root())<<endl;
    VCT.addLast(2);
    VCT.swap(VCT.root(), VCT.left(VCT.root()));
    cout<<"Expected size is: 2 --- Output size is: "<<VCT.size()<<endl;
    cout<<"Expected output is: 2 --- Output  is: "<<*(VCT.root())<<endl;
    return 0;
}
/*--------------End of Vector complete tree------------*/
```

```
/*-----------------Heap priority queue-------------------*/
#include <iostream>
#include <vector>
using namespace std;


//Comparators
template <typename T>
class BottomTop
{
    public:
        bool operator()(const T& p, const T& q)
        {
            return (p<q);
        }
};

template <typename T, typename C>
class HeapPriorityQueue
{
    private:
        VectorCompleteTree<T> CT;
        C isLess;
        typedef typename VectorCompleteTree<T>::Position Position;
    public:
        int size();
        bool empty();
        void insert(const T& e);
        const T& min();
        void removeMin();

};

template <typename T, typename C>
int HeapPriorityQueue<T,C>::size()
{
    return CT.size();
}
template <typename T, typename C>
bool HeapPriorityQueue<T,C>::empty()
{
    return (size() == 0);
}

template <typename T, typename C>
void HeapPriorityQueue<T,C>::insert(const T& e)
{
    CT.addLast(e);
    Position v = CT.last();
    while(!CT.isRoot(v))
    {
        Position par = CT.parent(v);
        if(isLess(*v,*par))
        {
            CT.swap(v,par);
            v = par;
        }
        else
            break;
    }
}
```

```
template <typename T, typename C>
const T& HeapPriorityQueue<T,C>::min()
{
    return *(CT.root());
}

template <typename T, typename C>
void HeapPriorityQueue<T,C>::removeMin()
{
    if(size() == 1)
    {
        CT.removeLast();
    }
    else
    {
        Position u = CT.root();
        CT.swap(u,CT.last());
        CT.removeLast();
        while(CT.hasLeft(u))
        {
            Position v = CT.left(u);
            if(CT.hasRight(u) && isLess(*(CT.right(u)), *v))
            {
                v = CT.right(u);
            }
            if(isLess(*v, *u))
            {
                CT.swap(u,v);
                u=v;
            }
            else
                break;
        }
    }

}
int main()
{

    HeapPriorityQueue<int, BottomTop<int> > HPQ;
    HPQ.insert(1);
    HPQ.insert(2);
    HPQ.insert(3);
    HPQ.insert(4);

    cout<<"Expected size is: 1 --- Output size is: "<<HPQ.min()<<endl;
    HPQ.removeMin();
    cout<<"Expected size is: 2 --- Output size is: "<<HPQ.min()<<endl;

    return 0;
}

/*-----------------End of Heap priority queue-------------------*/
```

```
/*-----------------------------2022 Exam------------------------------*/
//Q1:
//int a[] = {1,36,7,18,28,23,24,16,17,38,21,3};
void find(int a[], int size, ArrayStack<int> &s)
{
    for(int i =0; i<size; i++)
    {
        while(! s.empty() && s.top()<a[i])
        {
            s.pop();
        }
        s.push(a[i]);
    }
}

//a) What is the stack contents at the end?
//      s = {38,21,3}
//b) What does this function do?
//      مش عارفها
//c) What is the complexity of this function?
//      O(N^2)
//d) What is the output of xxx();
//      4 6 7 9 10
//e) How many times the function will be called?
//      10 times
```



1-Answer briefly.

a={1, 36, 7, 18, 28, 23, 24, 16, 17, 38, 21, 3,};

a) What is the stack contents at the end?

b) What does this function do?

c) What is the complexity of the function?

Note that, ArrayStack page (199)

```
void find(int a[ ], int size , ArrayStack &s)
{  for (int i=0;  i<size; i++)
   {while (! s.empty() && s.top() <a[i])
         s.pop();  → a[i]
      s.push(i);  } }
```

```
void xxx(Node *root)
{
  if (!root)     return;   → if(root==null)
  if (!root->left && !root->right)
  { cout << root->data << " ";    return; }
  if (root->left)   xxx(root->left);
  if (root->right)  xxx(root->right);
}
```

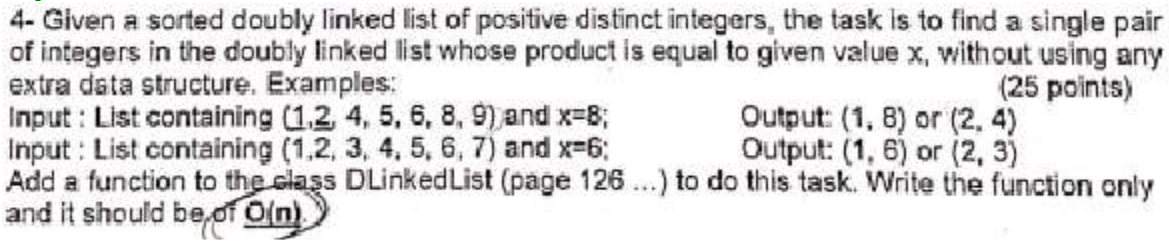
d) For this tree, what is the output of xxx?

e) How many times will be the function called?

```
/*Q2: Extend euler tour to get the sum of external nodes*/
template <typename E, typename R>
class SumOfExternalNodes : public EulerTour<E, R>
{
    protected: // shortcut type names
        typedef typename EulerTour<E, R>::BinaryTree BinaryTree;
        typedef typename EulerTour<E, R>::Position Position;
        typedef typename EulerTour<E, R>::Result Result;
    public:
        void execute(const BinaryTree& T){ // execute the tour
            initialize(T);
            std::cout << "The sum of external nodes is: " << eulerTour(T.root()) << "\n";
        }
    protected: // leaf: return value
        virtual void visitExternal(const Position& p, Result& r) const
        { r.finalResult = (*p).value(); }
        // internal: do operation
        virtual void visitRight(const Position& p, Result& r) const
        { r.finalResult = r.leftResult+ r.rightResult; }
};
/*Q3: Write a deque class without inheritance or objects from any other class or structure*/
// = Deque with array
#include <iostream>
using namespace std;
typedef unsigned int Elem;
class DequeArr
{
    private:
        Elem* arr;
        int capacity;
        int t;
    public:
        DequeArr(int cap);
        int size();
        bool empty();
        Elem front();
        Elem back();
        void insertFront(const Elem& d);
        void insertBack(const Elem& d);
        void removeFront();
        void removeBack();
        void PrintQueue();
};
DequeArr::DequeArr(int cap = 10) : capacity(cap), arr(new Elem[capacity]), t(-1)
{}
int DequeArr::size()
{
    return (t+1);
}
bool DequeArr::empty()
{
    return (t==-1);
}
Elem DequeArr::front()
{
  return arr[0];
}
Elem DequeArr::back()
{
    return arr[t];
}
```

```cpp
void DequeArr::insertFront(const Elem& d)
{
    if(t+1 < capacity)
    {
        for(int i=t+1; i>0; i--)
            arr[i] = arr[i-1];
        arr[0] = d;
        t++;
    }
}
void DequeArr::insertBack(const Elem& d)
{
    if(t+1 < capacity)
    {
        t++;
        arr[t] = d;
    }
}
void DequeArr::removeFront()
{
    for(int i = 0; i<t; i++)
        arr[i] = arr[i+1];
    t--;
}
void DequeArr::removeBack()
{
    t--;
}
void DequeArr::PrintQueue() //for testing
{
    for(int i =0; i<=t; i++)
        cout<<arr[i]<<" ";
}
int main()
{
    DequeArr D(10);
    D.insertFront(1);
    D.insertBack(2);
    D.insertBack(3);
    D.insertBack(4);
    D.insertBack(5);
    cout<<"the size is : "<<D.size()<<endl;
    D.PrintQueue();
    cout<<endl;
    D.removeBack();
    D.PrintQueue();
    cout<<endl;
    D.removeFront();
    D.PrintQueue();
    cout<<endl;
    return 0;
    /*Output:
    the size is : 5
    1 2 3 4 5
    1 2 3 4
    2 3 4
    */
}
```

```
//Q4
```



4- Given a sorted doubly linked list of positive distinct integers, the task is to find a single pair of integers in the doubly linked list whose product is equal to given value x, without using any extra data structure. Examples: (25 points)

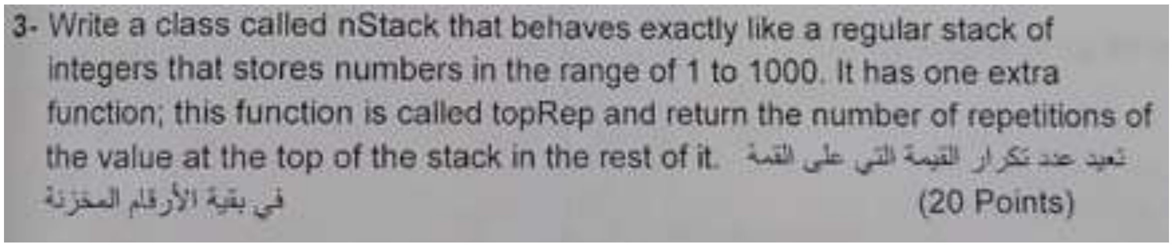
Input : List containing (1,2, 4, 5, 6, 8, 9) and x=8; Output: (1, 8) or (2, 4)

Input : List containing (1,2, 3, 4, 5, 6, 7) and x=6; Output: (1, 6) or (2, 3)

Add a function to the class DLinkedList (page 126 ...) to do this task. Write the function only and it should be of O(n).

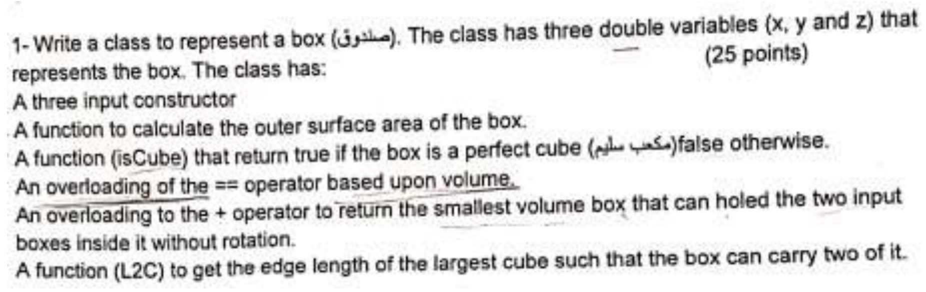
```
void DLinkedList::SinglePairsProductEqualToX(int x)
{
    nodeptr curr = header->next;
    nodeptr trailer_curr = trailer->prev;
    while(curr!=trailer_curr)
    {
        if(curr->data * trailer_curr->data == x)
        {
            cout<<"("<<curr->data<<", "<<trailer_curr->data<<")\n";
            trailer_curr=trailer_curr->prev;
            continue;
        }
        else if(curr->data * trailer_curr->data < x)
        {
            curr=curr->next;
        }
        else
        {
            trailer_curr = trailer_curr->prev;
        }
    }
}

//Q5
//(1)_isHeap()
#include <iostream>
using namespace std;

bool isHeap(int* arr, int sz)
{
    for(int i = 0; i<sz/2; i++)
    {
        if(arr[i]>arr[i*2+1] || arr[i]>arr[i*2+2])
            return false;
    }
    return true;
}
//(2)_BottomUpHeap()
void BottomUpHeap(int* arr, int sz)
{
    int temp;
    for(int i = (sz-2)/2; i>=0; i--)
    {

        int left = arr[i*2 + 1];
        int right = arr[i*2 + 2];
        if(arr[i]<left && arr[i]<right)
        {

            continue;
        }
        else
        {
            if( left < right)
            {
                temp = left;
                arr[i*2 + 1] = arr[i];
                arr[i] = temp;
            }
            else if(right<left)
            {
                temp = right;
                arr[i*2 + 2] = arr[i];
                arr[i] = temp;

            }
            else
                continue;
        }
    }
}
int main()
{

    int arr[] ={1,3,2,7,4};
    cout<<isHeap(arr, 5);
    cout<<endl;
    int arr2[]= {3,7,2,1,4};
    BottomUpHeap(arr2, 5);
    for(int i =0; i<5; i++)
    {
        cout<<arr2[i]<<" ";
    }
    return 0;
    /*Output:
    1
    1 3 2 7 4
    */
}


/*---------------------2019 exam--------------------*/
//Q1 - Triangle class
#include <iostream>
using namespace std;

class Tri
{
    private:
        double L1, L2, L3;
    public:
        Tri(double L1_, double L2_, double L3_);
        void printTr();
        bool operator==(Tri T2);
        bool isObtuseTriangle();

};

Tri::Tri(double L1_, double L2_, double L3_) : L1(L1_), L2(L2_), L3(L3_)
{}
```

```
void Tri::printTr()
{
    cout<<"L1 is: "<<L1<<endl;
    cout<<"L2 is: "<<L2<<endl;
    cout<<"L3 is: "<<L3<<endl;
}

bool Tri::operator==(Tri T2)
{
    if(L1 != T2.L1 && L1 !=T2.L2 && L1!= T2.L3)
        return false;
    if(L2 != T2.L1 && L1 !=T2.L2 && L1!= T2.L3)
        return false;
    if(L3 != T2.L1 && L1 !=T2.L2 && L1!= T2.L3)
        return false;
    return true;
}
bool Tri::isObtuseTriangle()
{
    double max = L1;
    double min1 = L2, min2=L3;
    if(L2>max)
    {
        max = L2;
        min1 = L1;
    }
    if(L3>max)
    {
        max = L3;
        min1 = L1;
        min2 = L2;
    }

    if((max*max)>((min1*min1)+(min2*min2)))
        return true;
    else
        return false;

}

int main()
{
    Tri T(3,4,5);
    Tri T2(5,3,4);
    Tri T3(3,4,6);
    cout<<"The triangles initialised are as follows:\nTriangle 1: \n";
    T.printTr();
    cout<<"Triangle 2: \n";
    T2.printTr();
    cout<<"Triangle 3: \n";
    T3.printTr();
    cout<<"T1: 3,4,5 --- T2: 5,3,4\nAre the two traingles identical?\nExpected Output = 
true(1) ----- Your output: "<<(T==T2)<<endl;
    cout<<"Is T1: (3,4,5) an obtuse triangle?\nExpected output: No (0) ----- Output: 
"<<T.isObtuseTriangle()<<endl;
    cout<<"Is T2: (5,3,4) an obtuse triangle?\nExpected output: No (0) ----- Output: 
"<<T2.isObtuseTriangle()<<endl;
    cout<<"Is T3: (3,4,6) an obtuse triangle?\nExpected output: Yes (1) ----- Output: 
"<<T3.isObtuseTriangle()<<endl;
    return 0;
    /*Output:
    The triangles initialised are as follows:
    Triangle 1:
    L1 is: 3
    L2 is: 4
    L3 is: 5
    Triangle 2:
    L1 is: 5
    L2 is: 3
    L3 is: 4
    Triangle 3:
    L1 is: 3
    L2 is: 4
    L3 is: 6
    T1: 3,4,5 --- T2: 5,3,4
    Are the two traingles identical?
    Expected Output = true(1) ----- Your output: 1
    Is T1: (3,4,5) an obtuse triangle?
    Expected output: No (0) ----- Output: 0
    Is T2: (5,3,4) an obtuse triangle?
    Expected output: No (0) ----- Output: 0
    Is T3: (3,4,6) an obtuse triangle?
    Expected output: Yes (1) ----- Output: 1
    */
}

//Q2: get the index of max element in arr
#include <iostream>
using namespace std;

template<typename T>
int getMaxElement(T* arr, int n)
{
    T max = arr[0];
    int indMax = 0;
    for(int i =1; i<n; i++)
    {
        if(arr[i]>max)
        {
            max = arr[i];
            indMax = i;
        }
    }
    return indMax;
}


int main()
{
    int arr[] = {9,2,3,4};
    cout<<getMaxElement<int>(arr, 4);
    return 0;
    //output: 0
}
```


3- Write a class called nStack that behaves exactly like a regular stack of integers that stores numbers in the range of 1 to 1000. It has one extra function; this function is called topRep and return the number of repetitions of the value at the top of the stack in the rest of it. تعيد عدد تكرار القيمة التي على القمة في بقية الأرقام المخزنة (20 Points)

```
//Q3
#include <iostream>
using namespace std;

typedef unsigned int Elem;
class nStack
{
    private:
        typedef struct Node
        {
            Elem data;
            Node* next;
        }*nodeptr;
        nodeptr header;
        int n;
        int arr[1000] = {0}; //using array of frequency 0->999
    public:
        int size();
        bool empty();
        void pop();
        void push(Elem d);
        Elem top();
        int topRep();
};

int nStack::size()
{
    return n;
}
bool nStack::empty()
{
    return (n==0);
}
void nStack::pop()
{
    int indTop = top();
    nodeptr old = header;
    header = header->next;
    delete old;
    n--;
    arr[indTop-1]--;
}
void nStack::push(Elem d)
{
    nodeptr v = new Node;
    v->data = d;
    if(empty())
    {
        header = v;
    }
    else
    {
        v->next = header;
        header = v;
    }
    arr[d-1]++;
    n++;
}
Elem nStack::top()
{
    return header->data;
}

int nStack::topRep()
{
    int indTop= top();
    return arr[indTop-1];
}
int main()
{
    nStack S;
    S.push(1);
    S.push(2);
    S.push(2);
    cout<<S.top()<<endl;
    cout<<S.topRep()<<endl;
    S.pop();
    cout<<S.top()<<endl;
    cout<<S.topRep()<<endl;
    return 0;
    /*Output:
    2
    2
    2
    1
    */
}

//Q4 – height non-recursive funtion
int BinarySearchTree::height2(Position p) const
{
    int d = 0;
    queue<Position> posQueue;
    posQueue.push(p);
    while(!posQueue.empty())
    {
        int n = posQueue.size();
        for(int i = 0; i<n;i++)
        {
            Position node = posQueue.front();
            if(!node.isExternal())
            {
                posQueue.push(node.left());
                posQueue.push(node.right());
            }
            posQueue.pop();
        }
        if(posQueue.size())
            d++;
    }
    return d;
}
```

```
//Q5 - extend euler tour class to test if the heap order property is satisfied
template <typename E, typename R>
class HeapOrderPropertyTest : public EulerTour<E, R>
{
    protected: // shortcut type names
        typedef typename EulerTour<E, R>::BinaryTree BinaryTree;
        typedef typename EulerTour<E, R>::Position Position;
        typedef typename EulerTour<E, R>::Result Result;
    public:
        void execute(const BinaryTree& T)
        { // execute the tour
            initialize(T);
            eulerTour(T.root()) << "\n";
        }
    protected: // leaf: return value
        virtual void visitExternal(const Position& p, Result& r) const
        { r.finalResult = (*p).value(); }
        // internal: do operation
        virtual void visitRight(const Position& p, Result& r) const
        {
            if((*p).value() < r.leftResult && (*p).value()< r.rightResult)
                r.finalResult = (*p).value();
            else
                cout<<"The heap order property is not satisfied!\n";
        }
};

/*-----------------2019 تخلفات ------------------*/
```



1- Write a class to represent a box (صندوق). The class has three double variables (x, y and z) that represents the box. The class has: (25 points)

A three input constructor

A function to calculate the outer surface area of the box.

A function (isCube) that return true if the box is a perfect cube (مكعب سليم)false otherwise.

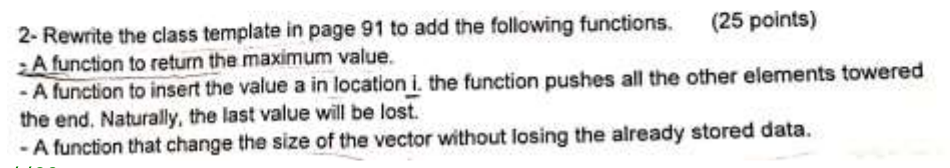
An overloading of the == operator based upon volume.

An overloading to the + operator to return the smallest volume box that can holed the two input boxes inside it without rotation.

A function (L2C) to get the edge length of the largest cube such that the box can carry two of it.

```
//Q1
class BOX
{
    private:
        double x,y,z;
    public:
        BOX(double x_, double y_, double z_);
        double GetOuterSurfaceArea();
        bool isCube();
        bool operator==(BOX B2);
        double operator+(BOX B2);
        double L2C();

};
BOX::BOX(double x_, double y_, double z_) : x(x_), y(y_), z(z_)
{}
double BOX::GetOuterSurfaceArea()
{
    return (2*x*y + 2*y*z + 2*x*z);
}
```

```
bool BOX::isCube()
{
    return (x==y && x==z);
}

bool BOX::operator==(BOX B2)
{
    return ((x*y*z) == (B2.x*B2.y*B2.z));
}
double BOX::operator+(BOX B2)
{
    double vol1 = 1.0;
    double vol2 = 1.0;
    double vol3 = 1.0;
    vol1*= (x+B2.x);
    vol1*=y>B2.y? y:B2.y;
    vol1*=z>B2.z? z:B2.z;

    vol2*= (y+B2.y);
    vol2*=x>B2.x? x:B2.x;
    vol2*=z>B2.z? z:B2.z;

    vol3*= (z+B2.z);
    vol3*=x>B2.x? x:B2.x;
    vol3*=z>B2.z? z:B2.z;
    double min = vol1;
    if(vol2 < min)
    {
        min = vol2;
    }
    if(vol3 < min)
    {
        min = vol3;
    }
    return min;
}
double BOX::L2C()
{
    double max = x;
    double max2 = y; double max3 = z;
    double temp;
    double edgeLength = 0.0;
    if(max2>max)
    {
        temp = max;
        max = max2;
        max2 = temp;
    }
    if(max3>max)
    {
        temp = max;
        max = max3;
        max3 = temp;
    }
    if(max3>max2)
    {
        temp = max2;
        max2 = max3;
        max3 = temp;
    }
    edgeLength = max/2;
    if(edgeLength<=max2 && edgeLength<=max3)
    {
        return edgeLength;
    }
    else
    {
        edgeLength = max2/2;
        if(edgeLength<=max3)
        {
            return edgeLength;
        }
        else
        {
            edgeLength = max3/2;
            return edgeLength;
        }
    }
}
```

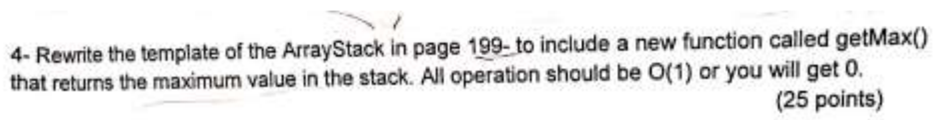


2- Rewrite the class template in page 91 to add the following functions. (25 points)

- A function to return the maximum value.
- A function to insert the value a in location i. the function pushes all the other elements towered the end. Naturally, the last value will be lost.
- A function that change the size of the vector without losing the already stored data.

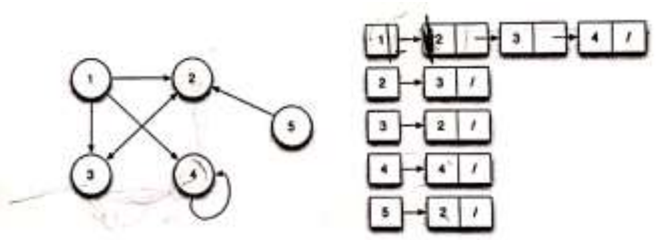
```
//Q2
template <typename T>
class BasicVector
{   // a simple vector class
    public:
        BasicVector(int capac = 10); // constructor
        T& operator[ ](int i) // access element at index i
        { return a[i]; }
        // . . . other public members omitted
        T& maxValue();
        void insert(T, int ind);
        void changeSize(int N);
    private:
        T* a; // array storing the elements
        int capacity; // length of array a
};

template <typename T> // constructor
BasicVector<T>::BasicVector(int capac) {
    capacity = capac;
    a = new T[capacity]; // allocate array storage
    for (int i = 0; i<capacity; i++)
    {
        a[i] = 0;
    }
}
```

```
template <typename T> // constructor
T& BasicVector<T>::maxValue() {
    T max = a[0];
    for(int i = 1; i<capacity; i++)
    {
        if(a[i]> max)
        {
            max = a[i];
        }
    }
    return max;
}


template <typename T> // constructor
void BasicVector<T>::insert(T data, int ind) {
    for(int i = capacity-1; i>ind; i--)
    {
        a[i] = a[i-1];
    }
    a[ind] = data;
}


template <typename T>
void BasicVector<T>::changeSize(int N) {
    if (capacity >= N)
        return; // already big enough
    T* B = new T[N]; // allocate bigger array
    for (int j = 0; j < n; j++) // copy contents to new array
        B[j] = a[j];
    if (a != NULL)
        delete [ ] a; // discard old array
    a = B; // make B the new array
    capacity = N; // set new capacity
}
```


4- Rewrite the template of the ArrayStack in page 199- to include a new function called getMax() that returns the maximum value in the stack. All operation should be O(1) or you will get 0.
(25 points)

```
//Q3
#include <iostream>
using namespace std;

template <typename T>
class ArrayStack
{
    enum{DEF_CAPACITY = 100};
    public:
        ArrayStack(int cap = DEF_CAPACITY);
        const int size();
        bool empty();
        const T& top();
        void push(const T& d);
        void pop();
        T& getMax();
    private:
        T* S;
        T* S_with_max;
        int capacity;
        int t;

};


template <typename T>
ArrayStack<T>::ArrayStack(int cap) : S(new T[cap]), S_with_max(new T[cap]),  t(-1),
capacity(cap)
{}

template <typename T>
const int ArrayStack<T>::size()
{
    return t+1;
}

template <typename T>
bool ArrayStack<T>::empty()
{
    return (t==-1);
}

template <typename T>
const T& ArrayStack<T>::top()
{
    try
    {
        if(t==-1)
        {
            throw("Stack is empty!");
        }
        return S[t];
    }
    catch(const char* cerr)
    {
        cout<<cerr<<endl;
    }
    return 0;
}

template <typename T>
void ArrayStack<T>::push(const T& d)
{
    try
    {
        if(size() == capacity)
        {
            throw("Stack is full!");
        }
        t++;
        S[t] = d;
        if(t==0)
        {
            S_with_max[t] = d;
        }
        else
        {
            if(d>S_with_max[t-1])
                S_with_max[t] = d;
            else
```

```
                S_with_max[t] = S_with_max[t-1];
            }
        }
        catch(const char* cerr)
        {
            cout<<cerr<<endl;
        }

    }

    template <typename T>
    void ArrayStack<T>::pop()
    {
        try
        {
            if(empty())
            {
                throw("Stack is empty!");
            }
            t--;
        }
        catch(const char* cerr)
        {
            cout<<cerr<<'\n';
        }

    }
    template <typename T>
    T& ArrayStack<T>::getMax()
    {
        return (S_with_max[t]);
    }

    int main()
    {
        ArrayStack<int>my_stack;
        my_stack.push(12);
        my_stack.push(134);

        my_stack.push(7);
        my_stack.push(13);
        cout<<my_stack.getMax();
    }
```



One way to represent a network of connections (communication, road, state) is through a structure called adjacency list. This list is an array of n linked lists, where n is the number of nodes in the network. Linked list number m contains the nodes connected to node m. Write a class to represent this structure. The class has

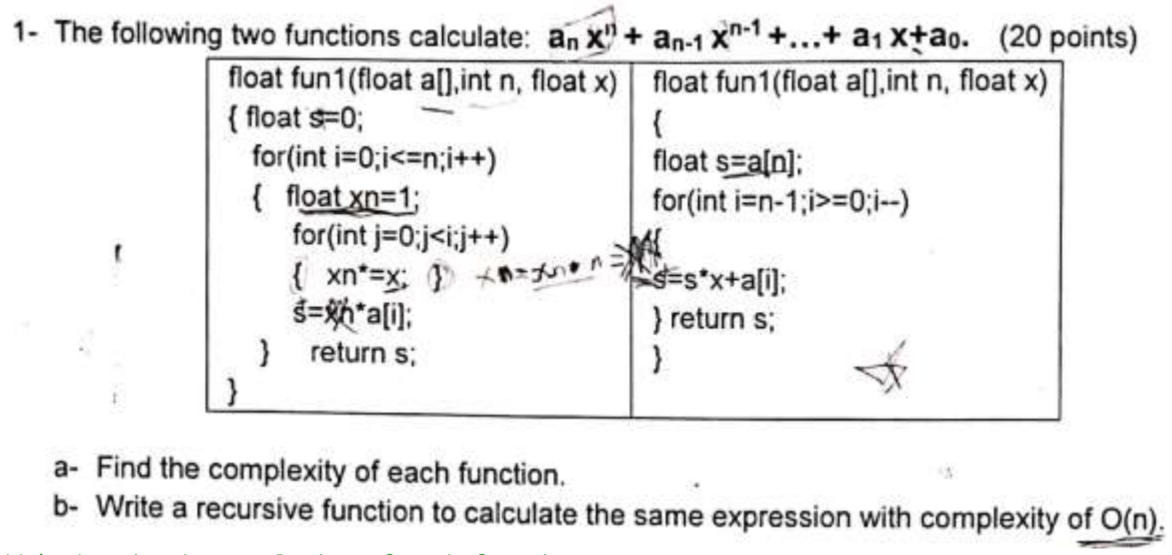
- A constructor that takes n as an input.
- A function that add a connection to the network. It takes the number of the two nodes representing a connection.
- A function that takes two nodes numbers and return true if there is a direct or indirect connection.

Do not use STL Classes. (25 points)

```
//Q4
#include <iostream>
using namespace std;

typedef int Elem;
typedef struct Node
{
    Elem data;
    Node* next;
}*nodeptr;


class AdjacencyList
{
    private:
        int n;
        nodeptr ADJ_L;
    public:
        AdjacencyList(int n_);
        void AddConnection(Elem a, Elem b);
        bool AreConnected(Elem a, Elem b);
};

AdjacencyList::AdjacencyList(int n_) : n(n_), ADJ_L(new Node[n])
{
    for(int i =0; i<n; i++)
    {
        ADJ_L[i].data = i;
        ADJ_L[i].next = NULL;
    }
}
```

```
void AdjacencyList::AddConnection(Elem a, Elem b)
{
    if((a>=0 && a<n) && (b>=0 && b<n))
    {
        nodeptr curr_a = &ADJ_L[a];
        nodeptr curr_b = &ADJ_L[b];
        while(curr_a->next!=NULL)
        {
            curr_a = curr_a->next;
        }

        curr_a->next = new Node;
        curr_a->next->data = b;
        curr_a = curr_a->next;
        curr_a->next = NULL;

        while(curr_b->next!=NULL)
        {
            curr_b = curr_b->next;
        }
        curr_b->next = new Node;
        curr_b->next->data = a;
        curr_b = curr_b->next;
        curr_b->next = NULL;


    }
}

bool AdjacencyList::AreConnected(Elem a, Elem b)
{
    int flag = 0;
    if((a>=0 && a<n) && (b>=0 && b<n))
    {
        for(int i = 0; i< n; i++)
        {
            nodeptr curr_a = &ADJ_L[i];
            while(curr_a!=NULL)
            {
                if(curr_a->data == a)
                {
                    flag++;
                }
                else if(curr_a->data == b)
                    flag++;
                curr_a = curr_a->next;
            }
            if(flag == 2)
            {
                return true;
            }
            else
            {
                flag = 0;
                continue;
            }

        }
        return false;
    }
}
```

```
int main()
{
    AdjacencyList AD(2);
    AD.AddConnection(0,1);
    cout<<AD.AreConnected(1,0);
    return 0;
}

/*--------------2018 exam------------*/
```



1- The following two functions calculate: $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$. (20 points)

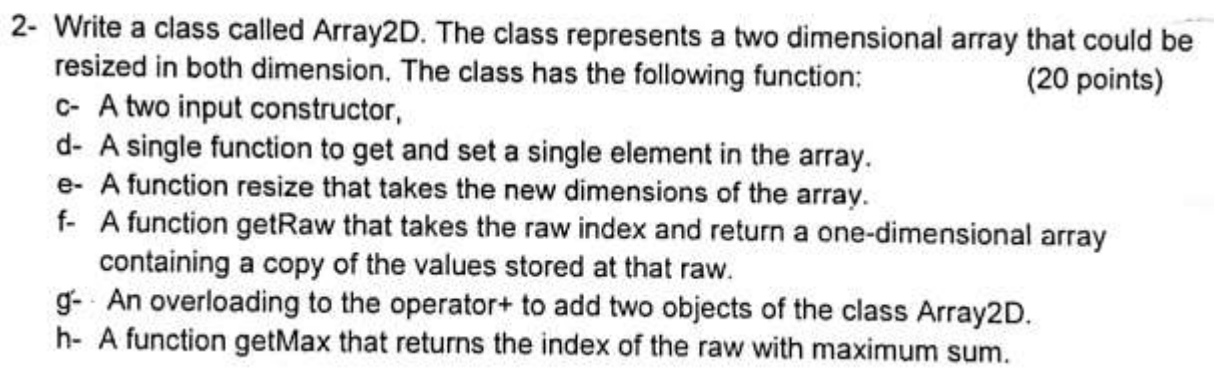
```
float fun1(float a[],int n, float x)
{ float s=0;
   for(int i=0;i<=n;i++)
   {  float xn=1;
       for(int j=0;j<i;j++)
       {  xn*=x;  }
       s=xn*a[i];
   }   return s;
}
```

```
float fun1(float a[],int n, float x)
{
float s=a[n];
for(int i=n-1;i>=0;i--)
{
s=s*x+a[i];
} return s;
}
```

a- Find the complexity of each function.

b- Write a recursive function to calculate the same expression with complexity of O(n).

```
//a) what is the complexity of each function:
//the function on the left: O(N^2), right: O(N)

#include <iostream>
using namespace std;
float fun1(int a[], int n, float x)
{
    float s=a[n];
    for(int i = n-1; i>=0; i--)
    {
        s=s*x+a[i];
    }
    return s;
}

float fun1_recursive(int a[], int n, float x)
{
    static int i = 0;
    if(i == n-1)
        return a[i];
    else
    {
        i++;
        float s = fun1_recursive(a, n,x);
        i--;
        return (x*s + a[i]);
    }
}
```



2- Write a class called Array2D. The class represents a two dimensional array that could be resized in both dimension. The class has the following function: (20 points)

c- A two input constructor,

d- A single function to get and set a single element in the array.

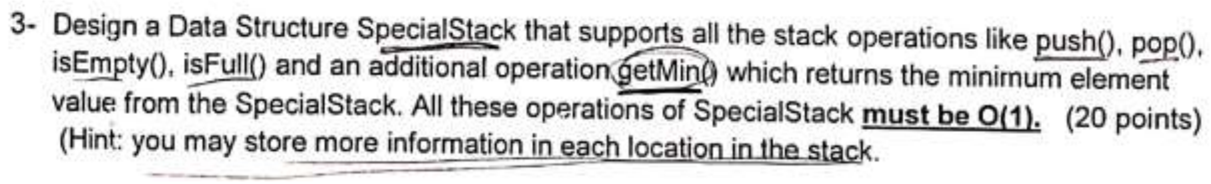
e- A function resize that takes the new dimensions of the array.

f- A function getRaw that takes the raw index and return a one-dimensional array containing a copy of the values stored at that raw.

g- An overloading to the operator+ to add two objects of the class Array2D.

h- A function getMax that returns the index of the raw with maximum sum.

```
#include <iostream>
using namespace std;

class Array2D
{

    private:
        int ** Arr;
        int rows;
        int cols;
    public:
        Array2D(int r, int c);
        int& operator()(int r, int c);
        void resize(int new_r, int new_c);
        int* getRow(int r);
        Array2D operator+(Array2D D2);
        int maxRowSum();
};
Array2D::Array2D(int r, int c) : rows(r), cols(c)
{
    Arr = new int* [rows];
    for(int i =0; i<rows; i++)
    {
        Arr[i] = new int[cols];
    }
}
int& Array2D::operator()(int r, int c)
{
    return Arr[r][c];
}
void Array2D::resize(int new_r, int new_c)
{
    int** b;
    b = new int* [new_r];
    for(int i = 0;i<new_r; i++)
        b[i] = new int [new_c];

    for(int i = 0; i<new_r; i++)
    {
        for(int j =0; j<new_c; j++)
        {
            b[i][j] = Arr[i][j];
        }
    }
    int ** old = Arr;
    Arr = b;
    for(int i =0; i<rows; i++)
    {
        delete[] old[i];
    }
    delete[] old;
    rows = new_r;
    cols = new_c;
}

int* Array2D::getRow(int r)
{
    int * row_arr = new int[cols];
    for(int i = 0; i<cols; i++)
    {
        row_arr[i] = Arr[r][i];
    }
    return row_arr;
}
Array2D Array2D::operator+(Array2D D2)
{
    if(rows != D2.rows || cols!=D2.cols)
    {
        Array2D result(0,0);
        return result;
    }

    Array2D result(rows, cols);
    for(int i =0; i<rows; i++)
    {
        for(int j=0; j<cols; j++)
        {
            result.Arr[i][j] = Arr[i][j] + D2.Arr[i][j];
        }
    }
    return result;

}

int Array2D::maxRowSum()
{
    int max = 0;
    int rowSum = 0;
    int maxRowIndex;
    for(int i = 0; i<rows; i++)
    {
        for(int j = 0; j<cols; j++)
        {
            rowSum += Arr[i][j];
        }
        if (rowSum>max)
        {
            max = rowSum;
            maxRowIndex = i;
        }
    }
    return maxRowIndex;
}
```


3- Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), isEmpty(), isFull() and an additional operation getMin() which returns the minimum element value from the SpecialStack. All these operations of SpecialStack **must be O(1).** (20 points) (Hint: you may store more information in each location in the stack.

```
class SpecialStack
{
    private:
        struct Elem
        {
            int data;
            int min;
        };
        int capacity;
        Elem* AS;
        int t;
    public:
        SpecialStack(int cap = 100);
        bool isEmpty();
        bool isFull();
        void push(int d);
        int& pop();
        int& getMin();
        int& top();
    };

SpecialStack::SpecialStack(int cap = 100) : capacity(cap), AS(new Elem [capacity]), t(-1)
{}
bool SpecialStack::isEmpty()
{
    return (t==-1);
}
bool SpecialStack::isFull()
{
    return (t==capacity-1);
}
void SpecialStack::push(int d)
{
    Elem e;
    e.data = d;
    if(isEmpty())
    {
        e.min = d;
    }
    else
    {
        if(AS[t].min>d)
        {
            e.min = d;
        }
        else
        {
            e.min = AS[t].min;
        }
    }
    t++;
    AS[t] = e;
}
```

```
int& SpecialStack::pop()
{
    t--;
}
int& SpecialStack::getMin()
{
    return AS[t].min;
}

int& SpecialStack::top()
{
    return AS[t].data;
}
```

4- Cunsider the following list of integers then (20 points)

18	17	8	6	11	5	2	13	16	1	3	4	7	12	9

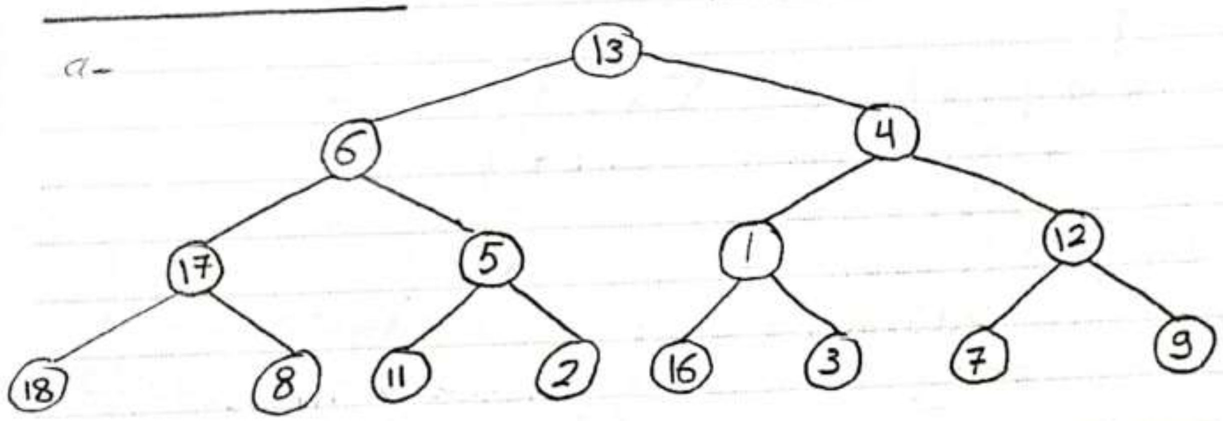
a- If the above list is the output of printing the inorder Travers of a complete binary tree, draw the tree.

b- Draw a binary search tree by inserting data from left to right.

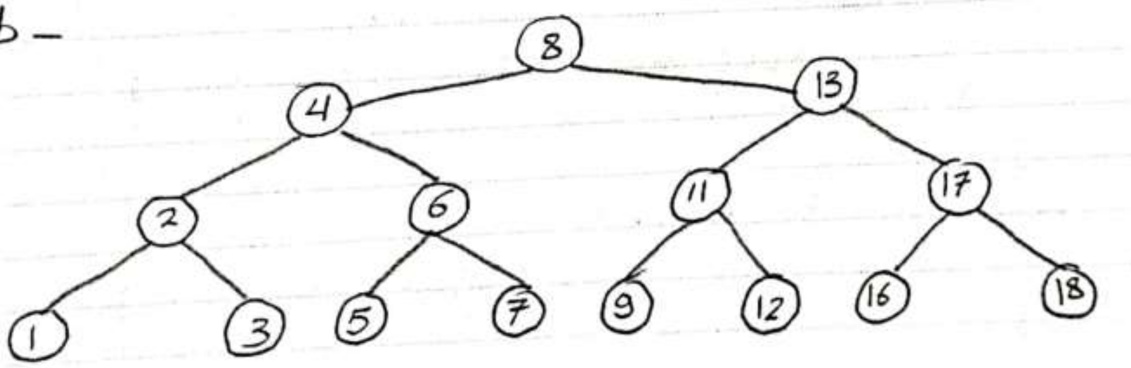
c- Draw the bottom-Up Heap construction steps like (figure 8.10) (make small drawings)

Question $\boxed{4}$:-

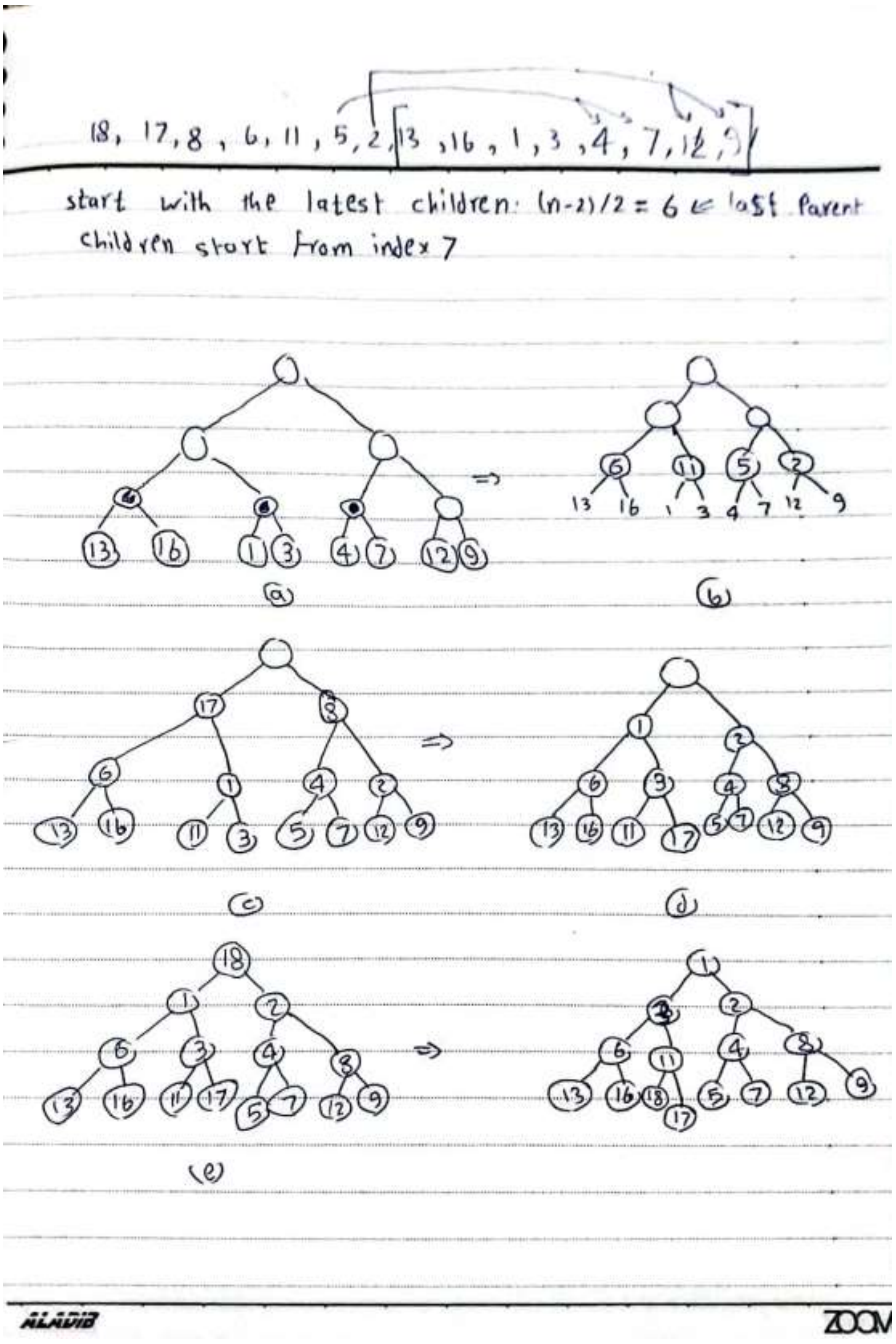
a-



b-



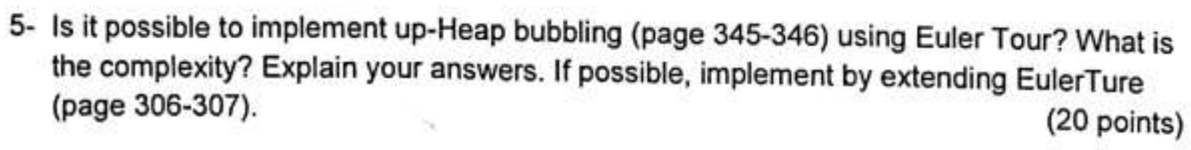
C:



18, 17, 8, 6, 11, 5, 2, 13, 16, 1, 3, 4, 7, 12, 9

start with the latest children: (n-2)/2 = 6 ⇐ last Parent

children start from index 7



5- Is it possible to implement up-Heap bubbling (page 345-346) using Euler Tour? What is the complexity? Explain your answers. If possible, implement by extending EulerTure (page 306-307).

(20 points)

No, it is not possible to implement up-heap bubbling using Euler tour.

Euler tour is a technique used for tree traversal that involves visiting each node in the tree exactly twice, once when entering the subtree and once when leaving the subtree. However, up-heap bubbling involves swapping a node with its parent node until the heap property is satisfied, which cannot be accomplished by simply visiting each node twice.

Up-heap bubbling is a bottom-up operation, where a new element is inserted at the end of the heap and then moved up the tree until the heap property is satisfied. This operation requires access to the parent node of the newly inserted element, which is not possible with Euler tour.

Therefore, up-heap bubbling cannot be implemented using Euler tour. Instead, it typically involves repeatedly swapping a newly inserted element with its parent until the heap property is satisfied. This operation can be implemented efficiently using a while loop and arithmetic calculations to find the parent node of a given element.