template <class Type>

class StackType

{

public:

~StackType();// Destructor

StackType(int maxSize); // Parameter constructor

// 1. Copy constructor

StackType(const StackType<Type>& otherStack); // copy constructor

// 2. Assignment Operator overload

const StackType<Type>& operator = (const StackType<Type>& otherStack);

//const StackType<Type>& StackType<Type>::operator = (const StackType<Type>& otherStack)

void initializeStack();

void invertStack();

bool isEmptyStack() const;

bool isFullStack() const;

Type stackTop() const; // peek into stack to know the Top content of stack

void push(Type anItem);

Type pop();

private:

int stackMaxSize;

int curSize;

bool isFull, isEmpty;

Type \*list;

//CopyStack Function that facilitates 1. Copy constructor

// and the 2. Assignment Operator overload

void copyStack(const StackType<Type>& otherStack);

};

template<class Type>

StackType<Type>::~StackType()

{

delete[] list;

//cout << "Destructor activated .." << endl;

}

template<class Type>

StackType<Type>::StackType(int maxSize = 100)

{

if (maxSize <= 0)

{

stackMaxSize = 100;

}

else

{

stackMaxSize = maxSize;

}

curSize = 0;

list = new Type[stackMaxSize];

//cout << "Parameter constructor used .." << endl;

}

// 1. Copy constructor that will use the void copyStack Function

template<class Type>

StackType<Type>::StackType(const StackType<Type>& otherStack)

{

list = NULL; // to facilitate delete

copyStack(otherStack);

}

// 2. Assignment Operator Overload that will use the void copyStack Function

template<class Type>

const StackType<Type>& StackType<Type>::operator = (const StackType<Type>& otherStack)

{

if (this != &otherStack) // ie. pointers are not pointing to same Object

copyStack(otherStack);// avoid self copy

return \*this;// return contents at where the pointer points to

}

//CopyStack Function that facilitates 1. Copy constructor

// and the 2. Assignment Operator overload

template<class Type>

void StackType<Type>::copyStack(const StackType<Type>& otherStack)

{

delete[] list;

stackMaxSize = otherStack.stackMaxSize;

curSize = otherStack.curSize;

list = new Type[stackMaxSize];

for (int i = 0; i < curSize; i++)

{

list[i] = otherStack.list[i];

}

}

template<class Type>

void StackType<Type>::initializeStack()

{

curSize = 0;

}

template<class Type>

void StackType<Type>::invertStack()

{

StackType<Type> tmpStack(stackMaxSize);

if (!isEmptyStack())

{

tmpStack.curSize = curSize;

tmpStack.list = new Type[stackMaxSize];

for (int i = 0; i < curSize; i++)

{

tmpStack.list[i] = list[i];

}

for (int i = (curSize - 1); i >= 0; i--)

{

list[i] = tmpStack.list[i];

}

}

else

{

//cout << "Stack is Empty !!" << endl;

}

}

template<class Type>

bool StackType<Type>::isEmptyStack() const

{

return(curSize == 0);

}

template<class Type>

bool StackType<Type>::isFullStack() const

{

return (curSize == stackMaxSize);

}

template<class Type>

Type StackType<Type>::stackTop() const // peek into stack to know the Top content of stack

{

if (!isEmptyStack())

{

return list[curSize - 1];

}

else

{

//cout << "Stack is Empty !!" << endl;

//return NULL;

}

};

template<class Type>

void StackType<Type>::push(Type anItem)

{

if (!isFullStack())

{

list[curSize] = anItem;

curSize++;

}

else

{

cout << "Stack is Full, item cannot be pushed !!" << endl;

}

}

template<class Type>

Type StackType<Type>::pop()

{

Type tmp;

tmp = stackTop();

if (!isEmptyStack())

{

curSize--;

}

else

{

cout << "Stack is Empty, nothing to pop !!" << endl;

}

return tmp;

};

Include binaryheap class

Invlude stack class

------------

In main

Ask gate numbers from user

Create a stack object with gate numbers

A double FOR loop to create (24 \* 14 = 336) 336 arrival times with flight numbers starting from 1000 and insert it into an array or a vector

Create the binary heap object using the arrival time vector just created above

Start a while loop to execute till binary heap size becomes zero

{

Here get the minimum from heap, which should be taken into an event structure variable, and this minimum should be deleted from the heap, but should be stored back into a vector that will contain all arrivals as well as departure event structure

If departure then store back its gate number into gate stack, and store the event in the schedules vector as a departure event

If arrival flight

Check for a gate number availability

If gate number available

store the event in the schedules vector as a arrival event

allot a delay time and re insert into the heap as a departure flight

else airport full - abort

}

If all arrivals have been successfully allotted gate numbers then the new vector thus formed will have all arrival and departure information

Create a new heap using this vector which will sort all flights in proper order of time which can be printed

End main

------------

Additional functions that may be required

A Random number function for delay

Create a stack of gate numbers initially