**DYNAMIC ARRAY**

.CPP

#include"dynamicarray.h"

DynamicArray :: DynamicArray(int size)

{

data = new int[size];

}

void DynamicArray:: insertItem (int index , int item)

{

data [ index ] = item;

}

int DynamicArray :: getItem (int index)

{

return data [index];

}

DynamicArray:: ~DynamicArray ()

{

**delete** [] data;

}

.H

#ifndef DYNAMICARRAY\_H\_INCLUDED

#define DYNAMICARRAY\_H\_INCLUDED

class DynamicArray{

private :

int\* data;

public :

DynamicArray (int);

~DynamicArray ();

void insertItem(int , int );

int getItem(int);

};

#endif *// DYNAMICARRAY\_H\_INCLUDED*

**SORTEDTYPE**

.CPP

#include "sortedtype.h"

#include <iostream>

**using** **namespace** std;

**template** <class ItemType >

SortedType <ItemType > :: SortedType()

{

length = 0;

currentPos = -1;

}

**template** <class ItemType >

void SortedType <ItemType> :: MakeEmpty()

{

length = 0;

}

**template** <class ItemType >

**bool** SortedType <ItemType> :: IsFull()

{

return (length == MAX\_ITEMS);

}

**template** <class ItemType >

int SortedType <ItemType> :: LengthIs()

{

return length;

}

**template** <class ItemType >

void SortedType <ItemType> :: ResetList()

{

currentPos = -1;

}

**template** <class ItemType >

void SortedType <ItemType> :: GetNextItem(ItemType& item)

{

currentPos++;

item = info [currentPos];

}

**template** <class ItemType >

void SortedType <ItemType> :: InsertItem(ItemType item)

{

int location = 0;

**bool** moreToSearch = (location < length );

while (moreToSearch)

{

if (item > info [location] )

{

location++;

moreToSearch = false;

}else if (item < info [location])

moreToSearch = false;

}

for (int index = length ; index > location ; index--)

{

info [index] = info [ index- 1];

}

info [ location ] = item;

length++;

}

**template** <class ItemType >

void SortedType <ItemType> :: DeleteItem (ItemType item)

{

int location = 0;

while (item != info [location])

{

location++;

}

for (int index = location +1 ; index < length ; index ++)

info [index-1 ] = info [index];

length --;

}

**template** <class ItemType >

void SortedType <ItemType> :: RetrieveItem (ItemType& item , **bool**& found)

{

int midPoint , first = 0, last = length -1 ;

**bool** moreToSearch = (first <= last);

found= false ;

while (moreToSearch && !found )

{

midPoint= (first + last) /2;

if (item < info[midPoint])

{

last = midPoint -1 ;

moreToSearch = (first <= last);

} else if(item > info [midPoint])

{

first = midPoint +1;

moreToSearch = (first <= last);

} else

{

found = true ;

item = info [midPoint];

}

}

}

.H

#ifndef SORTEDTYPE\_H\_INCLUDED

#define SORTEDTYPE\_H\_INCLUDED

#include <iostream>

**using** **namespace** std;

const int MAX\_ITEMS = 5;

**template** <class ItemType >

class SortedType

{

public:

SortedType ();

void MakeEmpty();

**bool** IsFull ();

int LengthIs();

void InsertItem (ItemType);

void DeleteItem( ItemType);

void RetrieveItem(ItemType& , **bool**&);

void ResetList();

void GetNextItem(ItemType&);

private:

int length;

ItemType info [MAX\_ITEMS];

int currentPos;

};

#endif *// SORTEDTYPE\_H\_INCLUDED*

MAIN

#include"sortedtype.cpp"

#include <iostream>

**using** **namespace** std;

int main()

{

SortedType <int> list1;

cout << "Length of the list is: " << list1.LengthIs() <<endl ;

*// inserting Item*

list1.InsertItem(5);

list1.InsertItem(7);

list1.InsertItem(4);

list1.InsertItem(2);

list1.InsertItem(1);

cout << "Length of the list is: " << list1.LengthIs() <<endl ;

*// Printing List*

cout<< "List: " <<endl;

for (int i = 0 ; i< list1.LengthIs() ; i++)

{

int temp;

list1.GetNextItem( temp);

cout << temp << " " ;

}

list1.ResetList();

cout<<endl;

*// Searching for 6*

**bool** found;

int itemInput;

cout<<"Enter search: ";

cin >> itemInput;

list1.RetrieveItem(itemInput, found);

if(!found)

{

cout<< "Item is not found"<<endl;

}else

{

cout<< "Item is found"<<endl;

}

*//Searching for 5*

cout<<"Enter search: ";

cin >> itemInput;

list1.RetrieveItem(itemInput, found);

if(!found)

{

cout<< "Item is not found"<<endl;

}else

{

cout<< "Item is found"<<endl;

}

*// List is full or not*

if (list1.IsFull())

cout << "List is full" << endl;

else

cout << "List is not full" << endl;

list1.DeleteItem( 1);

cout << "Length of the list after deletion is: " << list1.LengthIs() <<endl ;

*// Printing List*

cout << "New List: " << endl;

for (int i = 0 ; i< list1.LengthIs() ; i++)

{

int temp;

list1.GetNextItem( temp);

cout << temp << " " ;

}

list1.ResetList();

cout<<endl;

*// List is full or not*

if (list1.IsFull())

cout << "List is full" << endl;

else

cout << "List is not full" << endl;

list1.DeleteItem( 1);

cout << "Length of the list after deletion is: " << list1.LengthIs() <<endl ;

}

**UNSORTEDTYPE**

.CPP

#include"unsortedtype.h"

#include <iostream>

**using** **namespace** std;

**template** < class ItemType>

UnsortedType <ItemType> :: UnsortedType()

{

length = 0;

currentPos = -1;

}

**template** < class ItemType>

**bool** UnsortedType <ItemType> :: isFull()

{

return (length == MAX\_ITEMS);

}

**template** < class ItemType>

int UnsortedType <ItemType> :: LengthIs()

{

return length;

}

**template** < class ItemType>

void UnsortedType <ItemType> :: ResetList()

{

currentPos = -1;

}

**template** < class ItemType>

void UnsortedType <ItemType> :: GetNextItem (ItemType& item)

{

currentPos++;

item = info [currentPos];

}

**template** < class ItemType>

void UnsortedType <ItemType> :: RetrieveItem (ItemType& item , **bool**& found)

{

int location = 0;

**bool** moreToSearch = (location <length);

found = false;

while ( moreToSearch && !found)

{

if (item == info [location])

{

found = true;

item = info [location];

}

else

{

location++;

moreToSearch = (location <length);

}

}

}

**template** < class ItemType>

void UnsortedType <ItemType> :: InsertItem (ItemType item)

{

info [length] = item;

length++;

}

**template** < class ItemType>

void UnsortedType <ItemType> :: DeleteItem (ItemType item)

{

int location = 0 ;

while ( item != info [ location ])

location++;

info[location] = info[length - 1];

length --;

}

.H

#ifndef UNSORTEDTYPE\_H\_INCLUDED

#define UNSORTEDTYPE\_H\_INCLUDED

const int MAX\_ITEMS = 5;

**template** <class ItemType>

class UnsortedType

{

public:

UnsortedType();

void MakeEmpty();

**bool** isFull();

int LengthIs();

void InsertItem (ItemType);

void DeleteItem (ItemType);

void RetrieveItem (ItemType& , **bool**&);

void ResetList ();

void GetNextItem (ItemType&);

private:

int length;

ItemType info[MAX\_ITEMS];

int currentPos;

};

#endif *// UNSORTEDTYPE\_H\_INCLUDED*

MAIN

#include <iostream>

#include"unsortedtype.cpp"

**using** **namespace** std;

int main()

{

UnsortedType<int>u;

u.InsertItem(5);

u.InsertItem(7);

u.InsertItem(6);

u.InsertItem(9);

int temp;

cout << "List: " ;

for (int i = 0 ; i < u.LengthIs() ; i++){

u.GetNextItem(temp);

cout << temp << " " ;

}

u.ResetList();

cout <<endl;

cout <<"Length of the list is: " <<u.LengthIs()<<endl;

u.InsertItem(1);

*//temp= 0;*

cout << "List: " ;

for (int i = 0 ; i < u.LengthIs() ; i++){

u.GetNextItem(temp);

cout << temp << " " ;

}

u.ResetList();

**bool** found;

int itemInput;

cout<<"Enter search: ";

cin >> itemInput;

u.RetrieveItem(itemInput, found);

if(!found)

{

cout<< "Item is not found"<<endl;

}else

{

cout<< "Item is found"<<endl;

}

cout<<"Enter search: ";

cin >> itemInput;

u.RetrieveItem(itemInput, found);

if(!found)

{

cout<< "Item is not found"<<endl;

}else

{

cout<< "Item is found"<<endl;

}

cout<<"Enter search: ";

cin >> itemInput;

u.RetrieveItem(itemInput, found);

if(!found)

{

cout<< "Item is not found"<<endl;

}else

{

cout<< "Item is found"<<endl;

}

cout<<"Enter search: ";

cin >> itemInput;

u.RetrieveItem(itemInput, found);

if(!found)

{

cout<< "Item is not found"<<endl;

}else

{

cout<< "Item is found"<<endl;

}

if (u.isFull())

{

cout<< "List is full"<<endl;

}else{

cout<< "List is not full"<<endl;

}

return 0 ;

}

**UNSORTEDTYPE- LINKED LIST**

.CPP

#ifndef UNSORTEDTYPE\_CPP

#define UNSORTEDTYPE\_CPP

#include "UnsortedType.h"

UnsortedType::UnsortedType()

{

length = 0;

listData = **NULL**;

}

UnsortedType::~UnsortedType()

{

MakeEmpty();

}

int UnsortedType::GetLength() const

{

return length;

}

ItemType UnsortedType::GetItem(ItemType item,**bool**& found)

{

**bool** moreToSearch;

NodeType\* location;

location = listData;

found = false;

moreToSearch = (location != **NULL**);

while(moreToSearch && !found)

{

switch(item.ComparedTo(location->info))

{

case LESS:

case GREATER:

location = location->next;

moreToSearch=(location!=**NULL**);

break;

case EQUAL:

found = true;

item = location->info;

break;

}

}

return item;

}

void UnsortedType::PutItem(ItemType item)

{

NodeType\* location;

*//obtain and fill a node*

location = new NodeType;

location->info = item;

location->next = listData;

listData = location;

length++;

}

void UnsortedType:: PrintList()

{

ResetList();

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

{

GetNextItem().Print();

cout<<" ";

}

cout<<endl;

}

void UnsortedType::DeleteItem(ItemType item)

{

NodeType\* location = listData;

NodeType\* tempLocation;

if(item.ComparedTo(location->info)==EQUAL)

{

tempLocation = location;

listData = listData->next;

}

else

{

while(item.ComparedTo(location->next->info) != EQUAL)

location = location->next;

tempLocation=location->next;

location->next=location->next->next;

}

**delete** tempLocation;

length--;

}

void UnsortedType::MakeEmpty()

{

NodeType\* tempPtr;

while(listData != **NULL**)

{

tempPtr=listData;

listData=listData->next;

**delete** tempPtr;

}

length = 0;

}

ItemType UnsortedType::GetNextItem()

{

if(currentPos == **NULL**)

currentPos = listData;

else

currentPos = currentPos->next;

return currentPos->info;

}

void UnsortedType::ResetList()

{

currentPos = **NULL**;

}

**bool** UnsortedType::IsFull() const

{

NodeType\* location;

try

{

location = new NodeType;

**delete** location;

return false;

}

catch(bad\_alloc &exception)

{

return true;

}

}

#endif *// UNSORTEDTYPE\_CPP*

.H

#ifndef UNSORTEDTYPE\_H

#define UNSORTEDTYPE\_H

#include "ItemType.h"

#include <new>

**using** **namespace** std;

**struct** NodeType;

class UnsortedType

{

public:

UnsortedType();

~UnsortedType();

**bool** IsFull() const;

int GetLength() const;

ItemType GetItem(ItemType item,**bool**& found);

void PutItem(ItemType item);

void DeleteItem(ItemType item);

void MakeEmpty();

void ResetList();

ItemType GetNextItem();

void PrintList();

private:

NodeType\* listData;

int length;

NodeType\* currentPos;

};

**struct** NodeType

{

ItemType info;

NodeType\* next;

};

#endif *// UNSORTEDTYPE\_H*

**STACKTYPE**

.CPP

#include "StackType.h"

**template** <class ItemType>

StackType<ItemType>::StackType()

{

top = -1;

}

**template** <class ItemType>

**bool** StackType<ItemType>::IsEmpty()

{

return (top == -1);

}

**template** <class ItemType>

**bool** StackType<ItemType>::IsFull()

{

return (top == MAX\_ITEMS - 1);

}

**template** <class ItemType>

void StackType<ItemType>::Push(ItemType newItem)

{

if( IsFull() )

throw FullStack();

top++;

items[top] = newItem;

}

**template** <class ItemType>

void StackType<ItemType>::Pop()

{

if( IsEmpty() )

throw EmptyStack();

top--;

}

**template** <class ItemType>

ItemType StackType<ItemType>::Top()

{

if (IsEmpty())

throw EmptyStack();

return items[top];

}

.H

#ifndef STACKTYPE\_H\_INCLUDED

#define STACKTYPE\_H\_INCLUDED

const int MAX\_ITEMS = 5;

class FullStack

*// Exception class thrown*

*// by Push when stack is full.*

{

};

class EmptyStack

*// Exception class thrown*

*// by Pop and Top when stack is emtpy.*

{

};

**template** <class ItemType> class StackType

{

public:

StackType();

**bool** IsFull();

**bool** IsEmpty();

void Push(ItemType);

void Pop();

ItemType Top();

private:

int top;

ItemType items[MAX\_ITEMS];

};

#endif *// STACKTYPE\_H\_INCLUDED*

**STACKTYPE- LINKED LIST**

.CPP

#ifndef STACKTYPELL\_CPP

#define STACKTYPELL\_CPP

#include "StackTypeLL.h"

**template**<class ItemType>

StackTypeLL<ItemType>::StackTypeLL()

{

cout<<"Constructing\n";

tos=**NULL**;

}

**template**<class ItemType>

StackTypeLL<ItemType>::~StackTypeLL()

{

cout<<"Destructing\n";

makeEmpty();

}

**template**<class ItemType>

void StackTypeLL<ItemType>::makeEmpty()

{

NodeType<ItemType> \*temp;

while(tos)

{

temp=tos;

tos=tos->next;

**delete** temp;

}

}

**template**<class ItemType>

void StackTypeLL<ItemType>::Push(ItemType item)

{

cout<<"Push::"<<item<<endl;

NodeType<ItemType> \*temp = new NodeType<ItemType>;

temp->data = item;

temp->next = tos;

tos = temp;

}

**template**<class ItemType>

void StackTypeLL<ItemType>::pop(ItemType &item)

{

NodeType<ItemType> \*temp;

temp = tos;

tos = tos->next;

item = temp->data;

**delete** temp;

}

**template**<class ItemType>

void StackTypeLL<ItemType>::printStack() const

{

NodeType<ItemType> \*temp = tos;

while(temp)

{

cout<<temp->data<<" ";

temp = temp->next;

}

}

**template**<class ItemType>

**bool** StackTypeLL<ItemType>::isEmpty() const

{

return (tos==**NULL**);

}

**template**<class ItemType>

**bool** StackTypeLL<ItemType>::isFull() const

{

NodeType<ItemType> \*temp;

try

{

temp = new NodeType<ItemType>;

**delete** temp;

return false;

}

catch(bad\_alloc e){

return true;

}

return (tos==**NULL**);

}

#endif *// STACKTYPELL\_CPP*

.H

#ifndef STACKTYPELL\_H

#define STACKTYPELL\_H

#include<iostream>

#include<new>

**using** **namespace** std;

**template**<class ItemType>

class NodeType

{

public:

int data;

NodeType \*next;

};

**template**<class ItemType>

class StackTypeLL

{

private:

NodeType<ItemType> \*tos;

public:

StackTypeLL();

~StackTypeLL();

void Push(ItemType);

void pop(ItemType &);

**bool** isFull () const;

**bool** isEmpty() const;

void makeEmpty();

void printStack() const;

};

#endif *// STACKTYPELL\_H*

#include "StackTypeLL.cpp"

**QUEUETYPE**

.CPP

#include <iostream>

#include "queuetype.h"

**using** **namespace** std;

**template**<class ItemType>

QueType<ItemType>::QueType(int max)

{

maxQue = max + 1;

front = maxQue - 1;

rear = maxQue - 1;

items = new ItemType[maxQue];

}

**template**<class ItemType>

QueType<ItemType>::QueType()

{

maxQue = 5;

front = maxQue - 1;

rear = maxQue - 1;

items = new ItemType[maxQue];

}

**template**<class ItemType>

QueType<ItemType>::~QueType()

{

**delete** [] items;

}

**template**<class ItemType>

void QueType<ItemType>::MakeEmpty()

{

front = maxQue - 1;

rear = maxQue - 1;

}

**template**<class ItemType>

**bool** QueType<ItemType>::IsEmpty()

{

return (rear == front);

}

**template**<class ItemType>

**bool** QueType<ItemType>::IsFull()

{

return ((rear+1)%maxQue == front);

}

**template**<class ItemType>

void QueType<ItemType>::Enqueue(ItemType newItem)

{

if (IsFull())

throw FullQueue();

else

{

rear = (rear +1) % maxQue;

items[rear] = newItem;

}

}

**template**<class ItemType>

void QueType<ItemType>::Dequeue(ItemType& item)

{

if (IsEmpty())

throw EmptyQueue();

else

{

front = (front + 1) % maxQue;

item = items[front];

}

}

.H

#ifndef QUETYPE\_H\_INCLUDED

#define QUETYPE\_H\_INCLUDED

class FullQueue

{};

class EmptyQueue

{};

**template**<class ItemType>

class QueType

{

public:

QueType();

QueType(int max);

~QueType();

void MakeEmpty();

**bool** IsEmpty();

**bool** IsFull();

void Enqueue(ItemType);

void Dequeue(ItemType&);

private:

int front;

int rear;

ItemType\* items;

int maxQue;

};

#endif *// QUETYPE\_H\_INCLUDED*

**QUEUETYPE – LINKED LIST**

.CPP

#ifndef QUEUETYPE\_LL\_CPP

#define QUEUETYPE\_LL\_CPP

#include "QueueTypeLL.h"

**template** <class ItemType>

QueueType<ItemType>::QueueType()

{

front = **NULL**;

rear = **NULL**;

}

**template** <class ItemType>

QueueType<ItemType>::~QueueType()

{

makeEmpty();

}

**template** <class ItemType>

void QueueType<ItemType>::makeEmpty()

{

NodeType<ItemType>\* tempPtr;

while (front != **NULL**){

tempPtr = front;

front = front->next;

**delete** tempPtr;

}

rear = **NULL**;

}

**template** <class ItemType>

**bool** QueueType<ItemType>::IsEmpty() const

{

return front == **NULL**;

}

**template** <class ItemType>

**bool** QueueType<ItemType>::IsFull() const

{

NodeType<ItemType>\* location;

try

{

location = new NodeType<ItemType>;

**delete** location;

return false;

}

catch(bad\_alloc e)

{

return true;

}

}

**template** <class ItemType>

void QueueType<ItemType>::enqueue(ItemType newItem)

{

if (IsFull())

throw FullQueue();

else

{

NodeType<ItemType>\* newNode;

newNode = new NodeType<ItemType>;

newNode->info = newItem;

newNode->next = **NULL**;

if(rear == **NULL**)

front = newNode;

else

rear->next = newNode;

rear = newNode;

}

}

**template** <class ItemType>

void QueueType<ItemType>::dequeue(ItemType& item)

{

if (IsEmpty())

throw EmptyQueue();

else

{

NodeType<ItemType>\* tempPtr;

tempPtr = front;

item = front->info;

front = front->next;

if (front == **NULL**)

rear = **NULL**;

**delete** tempPtr;

}

}

#endif *// QUEUETYPE\_LL\_CPP*

.H

#ifndef QUEUETYPE\_LL\_H

#define QUEUETYPE\_LL\_H

#include<iostream>

#include<new>

**using** **namespace** std;

class FullQueue

{};

class EmptyQueue

{};

**template** <class ItemType>

**struct** NodeType

{

ItemType info;

NodeType\* next;

};

**template** <class ItemType>

class QueueType

{

public:

QueueType();

~QueueType();

void makeEmpty();

void enqueue(ItemType);

void dequeue(ItemType&);

**bool** IsEmpty() const;

**bool** IsFull() const;

private:

NodeType<ItemType>\* front;

NodeType<ItemType>\* rear;

};

#endif *// QUEUETYPE\_LL\_H*

#include "QueueTypeLL.cpp"