C++

Book 1 : Learning
Book 2 : Implementation
Book 3 : Magic & Madness

This documentation is for reference purpose only and is for those who have attended the classroom sessions at Thinking Machines.

- **During your classroom session appropriate theory needs to be written against each example.**
- **You are required to bring this book daily for your classroom sessions.**
- **Some examples won't compile. They have been written to explain some rules.**
- **If you try to understand the examples without attending theory sessions then may god help you.**

Note : Book Three Of C++ is only for full course students.

S.No.	Topic	Page
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		
28		
29		
30		
31		

Implementing Collections using templates
tmstl.h

```
#include <iostream>
#define true 1
#define TRUE 1
#define false 0
#define FALSE 0
#define bool int
#define boolean int
namespace tmstl
{
// Singly Linked List Starts
template<class T>
class SinglyLinkedListNode;
template<class T>
class SinglyLinkedListIterator;
template<class T>
class SinglyLinkedList;
template<class T>
class SinglyLinkedListNode
{
private:
T data;
SinglyLinkedListNode *next;
SinglyLinkedListNode(T);
friend class SinglyLinkedList<T>;
friend class SinglyLinkedListIterator<T>;
};
template<class T>
SinglyLinkedListNode<T>::SinglyLinkedListNode(T data)
{
this->data=data;
this->next=NULL;
}
template<class T>
class SinglyLinkedListIterator
{
private:
SinglyLinkedListNode<T> *node;
SinglyLinkedListIterator(SinglyLinkedListNode<T> *);
public:
boolean hasNext();
T next();
friend class SinglyLinkedList<T>;
};
template<class T>
SinglyLinkedListIterator<T>::SinglyLinkedListIterator(SinglyLinkedListNode<T> *node)
```

```
{
this->node=node;
}
template<class T>
boolean SinglyLinkedListIterator<T>::hasNext()
{
return this->node!=NULL;
}
template<class T>
T SinglyLinkedListIterator<T>::next()
{
if(this->node==NULL) return 0;
T data;
data=this->node->data;
this->node=this->node->next;
return data;
}
template<class T>
class SinglyLinkedList
{
private:
SinglyLinkedListNode<T> *start;
SinglyLinkedListNode<T> *end;
int size;
public:
SinglyLinkedList();
SinglyLinkedList(const SinglyLinkedList<T> &);
virtual ~SinglyLinkedList();
SinglyLinkedList<T> & operator=(SinglyLinkedList<T>);
void add(T);
void insert(int,T);
T remove(int);
void clear();
T get(int);
int getSize();
SinglyLinkedListIterator<T> * getIterator();
};
template<class T>
SinglyLinkedList<T>::SinglyLinkedList()
{
this->size=0;
this->start=NULL;
this->end=NULL;
}
template<class T>
SinglyLinkedList<T>::SinglyLinkedList(const SinglyLinkedList<T> &otherSinglyLinkedList)
{
```

```
this->size=0;
this->start=NULL;
this->end=NULL;
SinglyLinkedListNode<T> *node;
node=otherSinglyLinkedList.start;
while(node!=NULL)
{
this->add(node->data);
node=node->next;
}
}
template<class T>
SinglyLinkedList<T>::~SinglyLinkedList()
{
this->clear();
}
template<class T>
SinglyLinkedList<T> & SinglyLinkedList<T>::operator=(SinglyLinkedList<T>
otherSinglyLinkedList)
{
this->clear();
SinglyLinkedListNode<T> *node;
node=otherSinglyLinkedList.start;
while(node!=NULL)
{
this->add(node->data);
node=node->next;
}
return *this;
}
template<class T>
void SinglyLinkedList<T>::add(T data)
{
SinglyLinkedListNode<T> *node;
node=new SinglyLinkedListNode<T>(data);
if(this->start==NULL)
{
this->start=node;
this->end=node;
}
else
{
end->next=node;
end=node;
}
this->size++;
}
```

```
template<class T>
T SinglyLinkedList<T>::get(int index)
{
if(index<0 || index>=size) return 0;
SinglyLinkedListNode<T> *node;
node=this->start;
int x;
x=0;
while(x<index)
{
node=node->next;
x++;
}
return node->data;
}
template<class T>
int SinglyLinkedList<T>::getSize()
{
return this->size;
}
template<class T>
void SinglyLinkedList<T>::insert(int position,T data)
{
if(position<0) position=0;
if(position>this->size) position=this->size;
if(position==this->size)
{
this->add(data);
return;
}
SinglyLinkedListNode<T> *node;
node=new SinglyLinkedListNode<T>(data);
if(position==0)
{
node->next=this->start;
this->start=node;
}
else
{
SinglyLinkedListNode<T> *t,*k;
int x;
t=this->start;
x=0;
while(x<position)
{
k=t;
t=t->next;
x++;
}
node->next=t;
k->next=node;
}
this->size++;
}
template<class T>
T SinglyLinkedList<T>::remove(int position)
{
if(position<0 || position>=this->size) return 0;
T data;
SinglyLinkedListNode<T> *t;
if(this->size==1)
{
t=this->start;
data=t->data;
this->start=NULL;
this->end=NULL;
delete t;
this->size=0;
return data;
}
SinglyLinkedListNode<T> *k;
t=this->start;
int x;
x=0;
while(x<position)
{
k=t;
t=t->next;
x++;
}
if(this->start==t)
{
this->start=this->start->next;
data=t->data;
delete t;
this->size--;
return data;
}
if(this->end==t)
{
end=k;
end->next=NULL;
data=t->data;
delete t;
```

```
this->size--;
return data;
}
k->next=t->next;
data=t->data;
delete t;
this->size--;
return data;
}
template<class T>
void SinglyLinkedList<T>::clear()
{
while(this->size>0) this->remove(0);
}
template<class T>
SinglyLinkedListIterator<T> * SinglyLinkedList<T>::getIterator()
{
return new SinglyLinkedListIterator<T>(this->start);
}
// Singly Linked List Complete

// Stack starts
template<class T>
class Stack
{
private:
SinglyLinkedList<T> list;
public:
Stack();
Stack(const Stack &);
virtual ~Stack();
Stack & operator=(Stack);
void push(T);
T pop();
boolean isEmpty();
T elementAtTop();
int getSize();
void clear();
};
template<class T>
Stack<T>::Stack()
{
// no code required
}
template<class T>
Stack<T>::Stack(const Stack<T> &otherStack)
{
```

```
this->list=otherStack.list;
}
template<class T>
Stack<T>::~Stack()
{
// no code required
}
template<class T>
Stack<T> & Stack<T>::operator=(Stack<T> otherStack)
{
this->list=otherStack.list;
}
template<class T>
void Stack<T>::push(T data)
{
list.insert(0,data);
}
template<class T>
T Stack<T>::pop()
{
if(this->isEmpty()) return 0;
T data;
data=list.remove(0);
return data;
}
template<class T>
boolean Stack<T>::isEmpty()
{
return this->list.getSize()==0;
}
template<class T>
T Stack<T>::elementAtTop()
{
if(this->isEmpty()) return 0;
T data;
data=list.get(0);
return data;
}
template<class T>
int Stack<T>::getSize()
{
return this->list.getSize();
}
template<class T>
void Stack<T>::clear()
{
this->list.clear();
```

```
}
// Stack ends
// Queue starts
template<class T>
class Queue
{
private:
SinglyLinkedList<T> list;
public:
Queue();
Queue(const Queue &);
virtual ~Queue();
Queue & operator=(Queue);
void add(T);
T remove();
boolean isEmpty();
T elementAtTop();
int getSize();
void clear();
};
template<class T>
Queue<T>::Queue()
{
// no code required
}
template<class T>
Queue<T>::Queue(const Queue<T> &otherQueue)
{
this->list=otherQueue.list;
}
template<class T>
Queue<T>::~Queue()
{
// no code required
}
template<class T>
Queue<T> & Queue<T>::operator=(Queue<T> otherQueue)
{
this->list=otherQueue.list;
}
template<class T>
void Queue<T>::add(T data)
{
list.add(data);
}
template<class T>
T Queue<T>::remove()
```

```
{
if(this->isEmpty()) return 0;
T data;
data=list.remove(0);
return data;
}
template<class T>
boolean Queue<T>::isEmpty()
{
return this->list.getSize()==0;
}
template<class T>
T Queue<T>::elementAtTop()
{
if(this->isEmpty()) return 0;
T data;
data=list.get(0);
return data;
}
template<class T>
int Queue<T>::getSize()
{
return this->list.getSize();
}
template<class T>
void Queue<T>::clear()
{
this->list.clear();
}
// Queue ends
// Binary Search Tree Starts
template<class T1,class T2>
class BinarySearchTreeNode;
template<class T1,class T2>
class  BinarySearchTree;
template<class T1,class T2>
class BinarySearchTreeIterator;
template<class T1,class T2>
class BinarySearchTreeNode
{
private:
T1 data;
BinarySearchTreeNode<T1,T2> *left;
BinarySearchTreeNode<T1,T2> *right;
BinarySearchTreeNode(T1);
friend class BinarySearchTree<T1,T2>;
friend class BinarySearchTreeIterator<T1,T2>;
};
template<class T1,class T2>
BinarySearchTreeNode<T1,T2>::BinarySearchTreeNode(T1 data)
{
this->data=data;
this->left=NULL;
this->right=NULL;
}
template<class T1,class T2>
class BinarySearchTreeIterator
{
private:
BinarySearchTreeNode<T1,T2> *node;
int iteratorType;
Stack<BinarySearchTreeNode<T1,T2> *> stack;
BinarySearchTreeIterator(BinarySearchTreeNode<T1,T2> *,int);
public:
boolean hasNext();
T1 next();
static int IN_ORDER;
friend class BinarySearchTree<T1,T2>;
};
template<class T1,class T2>
int BinarySearchTreeIterator<T1,T2>::IN_ORDER=1;
template<class T1,class T2>
BinarySearchTreeIterator<T1,T2>::BinarySearchTreeIterator(BinarySearchTreeNode<T1,T2>
*node,int iteartorType)
{
this->node=node;
this->iteratorType=iteratorType;
if(this->node!=NULL)
{
while(this->node!=NULL)
{
stack.push(this->node);
this->node=this->node->left;
}
this->node=stack.pop();
}
}
template<class T1,class T2>
boolean BinarySearchTreeIterator<T1,T2>::hasNext()
{
return this->node!=NULL;
}
template<class T1,class T2>
T1 BinarySearchTreeIterator<T1,T2>::next()
```

```
{
if(this->node==NULL) return 0;
T1 d;
d=this->node->data;
this->node=this->node->right;
while(true)
{
if(this->node!=NULL)
{
stack.push(this->node);
this->node=this->node->left;
}
else
{
if(stack.isEmpty()) break;
this->node=stack.pop();
break;
}
}
return d;
}
template<class T1,class T2>
class BinarySearchTree
{
private:
int (*entityComparator)(T1,T1);
int (*keyComparator)(T2,T2);
T2 (*keyGetter)(T1);
BinarySearchTreeNode<T1,T2> *start;
int size;
int getHeight(BinarySearchTreeNode<T1,T2> *);
// following function is for testing
void processLevel(BinarySearchTreeNode<T1,T2> *,int);
public:
// following function is for testing
void levelOrderTraversal();
BinarySearchTree(int (*)(T1,T1),int (*)(T2,T2),T2 (*)(T1));
BinarySearchTree(const BinarySearchTree<T1,T2> &);
BinarySearchTree<T1,T2> & operator=(BinarySearchTree<T1,T2>);
virtual ~BinarySearchTree();
int getSize();
boolean add(T1);
void traverseInOrder(void (*)(T1));
BinarySearchTreeIterator<T1,T2> * getIterator(int);
boolean contains(T2);
void get(T2,boolean *,T1 &);
void remove(T2,boolean *,T1 &);
void clear();
};
template<class T1,class T2>
BinarySearchTree<T1,T2>::BinarySearchTree(int (*entityComparator)(T1,T1),int (*keyComparator)
(T2,T2),T2 (*keyGetter)(T1))
{
this->start=NULL;
this->size=0;
this->entityComparator=entityComparator;
this->keyComparator=keyComparator;
this->keyGetter=keyGetter;
}
template<class T1,class T2>
BinarySearchTree<T1,T2>::BinarySearchTree(const BinarySearchTree<T1,T2>
&otherBinarySearchTree)
{
this->start=NULL;
this->size=0;
this->entityComparator=otherBinarySearchTree.entityComparator;
this->keyComparator=otherBinarySearchTree.keyComparator;
this->keyGetter=otherBinarySearchTree.keyGetter;
// level order without recursion
Queue<BinarySearchTreeNode<T1,T2> *> queue;
BinarySearchTreeNode<T1,T2> *t;
t=otherBinarySearchTree.start;
while(t!=NULL)
{
this->add(t->data);
if(t->left!=NULL)queue.add(t->left);
if(t->right!=NULL)queue.add(t->right);
t=queue.remove();
}
}
template<class T1,class T2>
BinarySearchTree<T1,T2> & BinarySearchTree<T1,T2>::operator=(BinarySearchTree<T1,T2>
otherBinarySearchTree)
{
this->clear();
// level order without recursion
Queue<BinarySearchTreeNode<T1,T2> *> queue;
BinarySearchTreeNode<T1,T2> *t;
t=otherBinarySearchTree.start;
while(t!=NULL)
{
this->add(t->data);
if(t->left!=NULL)queue.add(t->left);
if(t->right!=NULL)queue.add(t->right);
t=queue.remove();
}
}
template<class T1,class T2>
BinarySearchTree<T1,T2>::~BinarySearchTree()
{
this->clear();
}
template<class T1,class T2>
void BinarySearchTree<T1,T2>::clear()
{
Stack<BinarySearchTreeNode<T1,T2> *> stack;
Queue<BinarySearchTreeNode<T1,T2> *> queue;
BinarySearchTreeNode<T1,T2> *t;
t=this->start;
while(t!=NULL)
{
stack.push(t);
if(t->left!=NULL)queue.add(t->left);
if(t->right!=NULL)queue.add(t->right);
t=queue.remove();
}
while(stack.isEmpty()==false)
{
delete stack.pop();
}
this->start=NULL;
this->size=0;
}
template<class T1,class T2>
int BinarySearchTree<T1,T2>::getSize()
{
return this->size;
}
template<class T1,class T2>
boolean BinarySearchTree<T1,T2>::add(T1 data)
{
Stack<BinarySearchTreeNode<T1,T2> *> stack;
BinarySearchTreeNode<T1,T2> *node;
if(this->start==NULL)
{
node=new BinarySearchTreeNode<T1,T2>(data);
this->start=node;
this->size++;
return true;
}
int weight;
```

```
BinarySearchTreeNode<T1,T2> *t;
t=this->start;
while(true)
{
weight=entityComparator(data,t->data);
if(weight==0)
{
return false;
}
stack.push(t);
if(weight<0)
{
if(t->left==NULL)
{
node=new BinarySearchTreeNode<T1,T2>(data);
t->left=node;
break;
}
else
{
t=t->left;
}
}
else
{
if(t->right==NULL)
{
node=new BinarySearchTreeNode<T1,T2>(data);
t->right=node;
break;
}
else
{
t=t->right;
}
}
}
this->size++;
BinarySearchTreeNode<T1,T2> *root;
BinarySearchTreeNode<T1,T2> *parentOfRoot;
BinarySearchTreeNode<T1,T2> *leftChild;
BinarySearchTreeNode<T1,T2> *rightChild;
BinarySearchTreeNode<T1,T2> *grandChild;
BinarySearchTreeNode<T1,T2> **pointerToRootPointer;
int leftChildHeight,rightChildHeight;
int heightDiff;
while(stack.isEmpty()==false)
```

```
{
root=stack.pop();
if(stack.isEmpty()==true)
{
pointerToRootPointer=&start;
}
else
{
parentOfRoot=stack.elementAtTop();
if(parentOfRoot->left==root)
{
pointerToRootPointer=&(parentOfRoot->left);
}
else
{
pointerToRootPointer=&(parentOfRoot->right);
}
}
leftChild=root->left;
rightChild=root->right;
leftChildHeight=getHeight(leftChild);
rightChildHeight=getHeight(rightChild);
heightDiff=leftChildHeight-rightChildHeight;
if(heightDiff>=-1 && heightDiff<=1) continue;
if(heightDiff>1)
{
// left  side is heavy
if(getHeight(leftChild->right)>getHeight(leftChild->left))
{
// left is right heavy
grandChild=leftChild->right;
leftChild->right=grandChild->left;
grandChild->left=leftChild;
root->left=grandChild;
grandChild=leftChild;
leftChild=root->left;
}
// left is left heavy
root->left=leftChild->right;
leftChild->right=root;
*pointerToRootPointer=leftChild;
}
else
{
// right side is heavy
if(getHeight(rightChild->left)>getHeight(rightChild->right))
{
```

```
// right is left heavy
grandChild=rightChild->left;
rightChild->left=grandChild->right;
grandChild->right=rightChild;
root->right=grandChild;
grandChild=rightChild;
rightChild=root->right;
}
// right is right heavy
root->right=rightChild->left;
rightChild->left=root;
*pointerToRootPointer=rightChild;
}
}
return true;
}
template<class T1,class T2>
void BinarySearchTree<T1,T2>::traverseInOrder(void (*processor)(T1))
{
Stack<BinarySearchTreeNode<T1,T2> *> stack;
BinarySearchTreeNode<T1,T2> *t;
t=start;
while(true)
{
if(t!=NULL)
{
stack.push(t);
t=t->left;
}
else
{
if(stack.isEmpty()) break;
t=stack.pop();
processor(t->data);
t=t->right;
}
}
}
template<class T1,class T2>
BinarySearchTreeIterator<T1,T2> * BinarySearchTree<T1,T2>::getIterator(int iteratorType)
{
if(iteratorType!=BinarySearchTreeIterator<T1,T2>::IN_ORDER) return NULL;
return new BinarySearchTreeIterator<T1,T2>(this->start,iteratorType);
}
template<class T1,class T2>
boolean BinarySearchTree<T1,T2>::contains(T2 searchKey)
{
```

```
BinarySearchTreeNode<T1,T2> *t;
T2 key;
int weight;
t=this->start;
while(t!=NULL)
{
key=this->keyGetter(t->data);
weight=keyComparator(searchKey,key);
if(weight==0) return true;
if(weight<0)
{
t=t->left;
}
else
{
t=t->right;
}
}
return false;
}
template<class T1,class T2>
void BinarySearchTree<T1,T2>::get(T2 searchKey,boolean *exists,T1 &d)
{
BinarySearchTreeNode<T1,T2> *t;
T2 key;
int weight;
t=this->start;
while(t!=NULL)
{
key=this->keyGetter(t->data);
weight=keyComparator(searchKey,key);
if(weight==0)
{
*exists=true;
d=t->data;
return;
}
if(weight<0)
{
t=t->left;
}
else
{
t=t->right;
}
}
*exists=false;
```

```
d=0;
}
template<class T1,class T2>
int BinarySearchTree<T1,T2>::getHeight(BinarySearchTreeNode<T1,T2> *t)
{
if(t==NULL) return 0;
int leftHeight=getHeight(t->left);
int rightHeight=getHeight(t->right);
if(leftHeight>=rightHeight) return leftHeight+1;
else return rightHeight+1;
}
// the following is for testing only
template<class T1,class T2>
void BinarySearchTree<T1,T2>::processLevel(BinarySearchTreeNode<T1,T2> *t,int level)
{
if(t==NULL) return;
if(level==1)
{
std::cout<<t->data<<" ";
}
else
{
if(level>1)
{
processLevel(t->left,level-1);
processLevel(t->right,level-1);
}
}
}
template<class T1,class T2>
void BinarySearchTree<T1,T2>::levelOrderTraversal()
{
int h=getHeight(start);
for(int x=1;x<=h;x++)
{
processLevel(start,x);
std::cout<<std::endl<<"--------------------------------"<<std::endl;
}
}
template<class T1,class T2>
void BinarySearchTree<T1,T2>::remove(T2 searchKey,boolean *exists,T1 &d)
{
Stack<BinarySearchTreeNode<T1,T2> *> stack;
BinarySearchTreeNode<T1,T2> *t;
T2 key;
int weight;
t=this->start;
```

```
while(t!=NULL)
{
key=this->keyGetter(t->data);
weight=keyComparator(searchKey,key);
if(weight==0)
{
break;
}
stack.push(t);
if(weight<0)
{
t=t->left;
}
else
{
t=t->right;
}
}
if(t==NULL)
{
*exists=false;
d=0;
return;
}
*exists=true;
d=t->data;
// logic to delete and balance to the top
BinarySearchTreeNode<T1,T2> **pointerToParentPointer;
BinarySearchTreeNode<T1,T2> *parent=NULL;
if(stack.isEmpty()==true)
{
pointerToParentPointer=&start;
}
else
{
parent=stack.elementAtTop();
if(parent->left==t)
{
pointerToParentPointer=&(parent->left);
}
else
{
pointerToParentPointer=&(parent->right);
}
}
if(t->left==NULL && t->right==NULL)
{
```

```
*pointerToParentPointer=NULL;
} else if(t->left==NULL)
{
*pointerToParentPointer=t->right;
} else if(t->right==NULL)
{
*pointerToParentPointer=t->left;
}
else
{
BinarySearchTreeNode<T1,T2> *f;
Queue<BinarySearchTreeNode<T1,T2> *> queue;
BinarySearchTreeNode<T1,T2> *parent2;
parent2=t;
f=t->right;
while(f->left!=NULL)
{
queue.add(f);
parent2=f;
f=f->left;
}
stack.push(f);
while(queue.getSize()>0)
{
stack.push(queue.remove());
}
if(parent2->left==f)
{
parent2->left=f->right;
}
else
{
parent2->right=f->right;
}
f->left=t->left;
f->right=t->right;
*pointerToParentPointer=f;
delete t;
this->size--;
BinarySearchTreeNode<T1,T2> *root;
BinarySearchTreeNode<T1,T2> *parentOfRoot;
BinarySearchTreeNode<T1,T2> *leftChild;
BinarySearchTreeNode<T1,T2> *rightChild;
BinarySearchTreeNode<T1,T2> *grandChild;
BinarySearchTreeNode<T1,T2> **pointerToRootPointer;
int leftChildHeight,rightChildHeight;
int heightDiff;
```

```
while(stack.isEmpty()==false)
{
root=stack.pop();
if(stack.isEmpty()==true)
{
pointerToRootPointer=&start;
}
else
{
parentOfRoot=stack.elementAtTop();
if(parentOfRoot->left==root)
{
pointerToRootPointer=&(parentOfRoot->left);
}
else
{
pointerToRootPointer=&(parentOfRoot->right);
}
}
leftChild=root->left;
rightChild=root->right;
leftChildHeight=getHeight(leftChild);
rightChildHeight=getHeight(rightChild);
heightDiff=leftChildHeight-rightChildHeight;
if(heightDiff>=-1 && heightDiff<=1) continue;
if(heightDiff>1)
{
// left  side is heavy
if(getHeight(leftChild->right)>getHeight(leftChild->left))
{
// left is right heavy
grandChild=leftChild->right;
leftChild->right=grandChild->left;
grandChild->left=leftChild;
root->left=grandChild;
grandChild=leftChild;
leftChild=root->left;
}
// left is left heavy
root->left=leftChild->right;
leftChild->right=root;
*pointerToRootPointer=leftChild;
}
else
{
// right side is heavy
if(getHeight(rightChild->left)>getHeight(rightChild->right))
{
// right is left heavy
grandChild=rightChild->left;
rightChild->left=grandChild->right;
grandChild->right=rightChild;
root->right=grandChild;
grandChild=rightChild;
rightChild=root->right;
}
// right is right heavy
root->right=rightChild->left;
rightChild->left=root;
*pointerToRootPointer=rightChild;
}
}
}
}
// Assignment : Write code for other traversals using iterator
// Binary Search Tree ends
// TMString starts
class TMStringPool
{
static int stringComparator(const char *leftOperand,const char *rightOperand)
{
return strcmp(leftOperand,rightOperand);
}
static const char * stringKeyGetter(const char *data)
{
return data;
}
static BinarySearchTree<const char *,const char *> *tree;
friend class TMString;
};
BinarySearchTree<const char *,const char *> * TMStringPool::tree=new BinarySearchTree<const char
*,const char
*>(TMStringPool::stringComparator,TMStringPool::stringComparator,TMStringPool::stringKeyGetter
);
class TMString
{
private:
const char *collection;
static TMStringPool pool;
public:
TMString()
{
collection=NULL;
}
```

```
TMString(const char *collection)
{
if(collection==NULL)
{
this->collection=NULL;
return;
}
const char *tmp;
boolean exists;
TMStringPool::tree->get(collection,&exists,tmp);
if(exists)
{
this->collection=tmp;
}
else
{
this->collection=new char[strlen(collection)+1];
strcpy((char *)this->collection,collection);
TMStringPool::tree->add(this->collection);
}
}
TMString(const TMString &other)
{
this->collection=other.collection;
}
TMString & operator=(TMString other)
{
this->collection=other.collection;
}
~TMString()
{
// do nothing
}
int operator==(TMString &other)
{
return this->collection==other.collection;
}
int operator<(TMString &other)
{
if(this->collection==other.collection) return false;
if(this->collection==NULL && other.collection!=NULL) return true;
if(this->collection!=NULL && other.collection==NULL) return false;
return strcmp(this->collection,other.collection)<0;
}
//Assignment : overload > <= >= <= and !=
TMString operator+(TMString &other)
{
```

```
if(other.collection==NULL) return TMString(this->collection);
if(this->collection==NULL) return TMString(other.collection);
const char *tmp=new char[strlen(this->collection)+strlen(other.collection)+1];
strcpy((char *)tmp,this->collection);
strcat((char *)tmp,other.collection);
const char *tmp2;
boolean exists;
TMStringPool::tree->get(tmp,&exists,tmp2);
if(exists)
{
delete [] tmp;
return TMString(tmp2);
}
TMStringPool::tree->add(tmp);
return TMString(tmp);
}
friend ostream & operator<<(ostream &,TMString &);
};
TMStringPool TMString::pool;
ostream & operator<<(ostream &cout,TMString &tmstring)
{
cout<<tmstring.collection;
return cout;
}
/*
Scale the TMString for other features as per your requirement.
See to it that its object works with cin for String input
*/
//TMString ends
} // tmstl ends
```

The Test Cases

Student.h

```
#ifndef _STRING_INCLUDED
#define _STRING_INCLUDED
#include <string.h>
#endif
class Student
{
private:
int rollNumber;
char *name;
public:
Student();
Student(const Student &);
Student & operator=(Student);
~Student();
void setRollNumber(int);
int getRollNumber();
void setName(const char *);
void getName(char * &);
};
Student::Student()
{
this->rollNumber=0;
this->name=NULL;
}
Student::Student(const Student &otherStudent)
{
this->rollNumber=0;
this->name=NULL;
this->setRollNumber(otherStudent.rollNumber);
this->setName(otherStudent.name);
}
Student & Student::operator=(Student otherStudent)
{
this->setRollNumber(otherStudent.rollNumber);
this->setName(otherStudent.name);
return *this;
}
Student::~Student()
{
if(this->name!=NULL) delete [] this->name;
}
void Student::setRollNumber(int rollNumber)
{
this->rollNumber=rollNumber;
}
int Student::getRollNumber()
{
return this->rollNumber;
}
void Student::setName(const char *name)
{
if(this->name!=NULL)
{
delete [] this->name;
this->name=NULL;
}
if(name==NULL) return;
this->name=new char[strlen(name)+1];
strcpy(this->name,name);
}
void Student::getName(char * &name)
```

```
{
if(this->name==NULL)
{
name=NULL;
return;
}
name=new char[strlen(this->name)+1];
strcpy(name,this->name);
}
```

SinglyTest.cpp

```
#include "student.h" // this is for testing our templates
#include "tmstl.h"
#include<iostream>
using namespace std;
using namespace tmstl;
int main()
{
SinglyLinkedList<int> s;
s.add(10);
s.insert(0,20);
s.add(30);
s.add(40);
s.insert(3,25);
SinglyLinkedListIterator<int> *iterator=s.getIterator();
while(iterator->hasNext())
{
cout<<iterator->next()<<endl;
}
delete iterator;
SinglyLinkedList<Student *> students;
Student *s1;
s1=new Student;
s1->setRollNumber(101);
s1->setName("Sameer");
Student *s2;
s2=new Student;
s2->setRollNumber(102);
s2->setName("Sudhir");
Student *s3;
s3=new Student;
s3->setRollNumber(103);
s3->setName("Mahesh");
students.add(s1);
students.insert(0,s2);
students.add(s3);
cout<<"After adding size : "<<students.getSize()<<endl;
SinglyLinkedListIterator<Student *> *siterator;
Student *stud;
char *n;
int r;
siterator=students.getIterator();
while(siterator->hasNext())
{
stud=siterator->next();
r=stud->getRollNumber();
stud->getName(n);
cout<<r<<","<<n<<endl;
delete [] n;
}
students.remove(1);
cout<<"After removal, size is : "<<students.getSize()<<endl;
siterator=students.getIterator();
while(siterator->hasNext())
{
stud=siterator->next();
r=stud->getRollNumber();
stud->getName(n);
cout<<r<<","<<n<<endl;
delete [] n;
}
students.clear();
cout<<"After clearing, size is : "<<students.getSize()<<endl;
delete s1;
delete s2;
delete s3;
return 0;
}
```

StackTest.cpp

```
#include<iostream>
using namespace std;
#include "student.h"
#include "tmstl.h"
using namespace tmstl;
int main()
{
Stack<int> s;
s.push(10);
s.push(20);
s.push(30);
s.push(40);
while(s.isEmpty()==false)
{
```

```
cout<<s.pop()<<endl;
}
Stack<Student *> students;
Student *s1;
s1=new Student;
s1->setRollNumber(101);
s1->setName("Sameer");
Student *s2;
s2=new Student;
s2->setRollNumber(102);
s2->setName("Sudhir");
Student *s3;
s3=new Student;
s3->setRollNumber(103);
s3->setName("Mahesh");
students.push(s1);
students.push(s2);
students.push(s3);
cout<<"After adding size : "<<students.getSize()<<endl;
Student *stud;
char *n;
int r;
while(!students.isEmpty())
{
stud=students.pop();
r=stud->getRollNumber();
stud->getName(n);
cout<<r<<","<<n<<endl;
delete [] n;
}
delete s1;
delete s2;
delete s3;
return 0;
}
```

QueueTest.cpp

```
#include<iostream>
using namespace std;
#include "student.h"
#include "tmstl.h"
using namespace tmstl;
int main()
{
Queue<int> q;
q.add(10);
q.add(20);
q.add(30);
q.add(40);
while(q.isEmpty()==false)
{
cout<<q.remove()<<endl;
}
Queue<Student *> students;
Student *s1;
s1=new Student;
s1->setRollNumber(101);
s1->setName("Sameer");
Student *s2;
s2=new Student;
s2->setRollNumber(102);
s2->setName("Sudhir");
Student *s3;
s3=new Student;
s3->setRollNumber(103);
s3->setName("Mahesh");
students.add(s1);
students.add(s2);
students.add(s3);
cout<<"After adding size : "<<students.getSize()<<endl;
Student *stud;
char *n;
int r;
while(!students.isEmpty())
{
stud=students.remove();
r=stud->getRollNumber();
stud->getName(n);
cout<<r<<","<<n<<endl;
delete [] n;
}
delete s1;
delete s2;
delete s3;
return 0;
}
```

BinaryTest.cpp

```
/*
the following file contains two functions named as
mainTest2 and main
First testcase is in main function, test it and then swap the names
of the two functions and run the second test case.
*/
```

```
#include<iostream>
using namespace std;
#include "tmstl.h"
using namespace tmstl;
#include "student.h"
int studentComparator(Student *leftOperand,Student *rightOperand)
{
return leftOperand->getRollNumber()-rightOperand->getRollNumber();
}
int studentKeyGetter(Student *student)
{
return student->getRollNumber();
}
int studentKeyComparator(int leftOperand,int rightOperand)
{
return leftOperand-rightOperand;
}
int iComparator(int e,int t)
{
return e-t;
}
int iKeyGetter(int e)
{
return e;
}
void iProcessor(int num)
{
cout<<num<<endl;
}
int main()
{
Student *s1;
s1=new Student();
s1->setRollNumber(101);
s1->setName("Mahesh");

Student *s2;
s2=new Student();
s2->setRollNumber(102);
s2->setName("Lalita");

Student *s3;
s3=new Student();
s3->setRollNumber(103);
s3->setName("Rohit");

Student *s4;
```

```
s4=new Student();
s4->setRollNumber(102);
s4->setName("Rohan");

BinarySearchTree<Student *,int> bst1(studentComparator,studentKeyComparator,studentKeyGetter);
bst1.add(s1);
bst1.add(s2);
bst1.add(s3);
bst1.add(s4);
cout<<"Size of bst1 containing (Student *) is "<<bst1.getSize()<<endl;
BinarySearchTree<int,int> bst2(iComparator,iComparator,iKeyGetter);
bst2.add(10);
bst2.add(101);
bst2.add(59);
bst2.add(645);
bst2.add(59);
bst2.add(100);
bst2.add(13);
cout<<"Size of bst2 containing (int) is "<<bst2.getSize()<<endl;
bst2.traverseInOrder(iProcessor);
cout<<"Iterating over bst2"<<endl;
BinarySearchTreeIterator<int,int> *iterator;
iterator=bst2.getIterator(BinarySearchTreeIterator<int,int>::IN_ORDER);
int n;
while(iterator->hasNext())
{
n=iterator->next();
cout<<n<<endl;
}
boolean exists;
int rrr;
cout<<"Enter roll number to search : ";
cin>>rrr;
exists=bst1.contains(rrr);
if(exists)
{
cout<<"Student with roll number : "<<rrr<<" exists"<<endl;
}
else
{
cout<<"Student with roll number : "<<rrr<<" does not exist"<<endl;
}
int num;
cout<<"Enter number to search   : ";
cin>>num;
exists=bst2.contains(num);
if(exists)
```

```
{
cout<<num<<" exists"<<endl;
}
else
{
cout<<num<<" does not exist"<<endl;
}
cout<<"Testing get function"<<endl;
cout<<"Enter roll number ";
cin>>rrr;
Student *s;
char *j;
bst1.get(rrr,&exists,s);
if(exists)
{
s->getName(j);
cout<<"Name : "<<j<<endl;
delete [] j;
}
else
{
cout<<rrr<<" is invalid roll number"<<endl;
}
return 0;
}
int mainTest2()
{
BinarySearchTree<int,int> bst2(iComparator,iComparator,iKeyGetter);
bst2.add(600);
bst2.add(500);
bst2.add(700);
bst2.add(550);
bst2.add(800);
bst2.add(900);
bst2.add(950);
bst2.add(975);
bst2.add(980);
bst2.add(990);
bst2.add(1000);
bst2.add(1050);
bst2.add(1055);
bst2.add(400);
bst2.add(350);
bst2.add(325);
bst2.add(320);
bst2.add(310);
bst2.add(300);
```

```
bst2.add(250);
boolean exists;
int num;
bst2.remove(550,&exists,num);
if(exists)
{
cout<<"Deleted : "<<num<<endl;
}
BinarySearchTreeIterator<int,int> *iterator;
iterator=bst2.getIterator(BinarySearchTreeIterator<int,int>::IN_ORDER);
int n;
while(iterator->hasNext())
{
n=iterator->next();
cout<<n<<endl;
}
return 0;
}
```

StringTest.cpp

```
#include<string.h>
#include<iostream>
using namespace std;
#include "tmstl.h"
using namespace tmstl;
int main()
{
TMString g="Cool";
TMString t;
t=g;
TMString k;
k="Fool";
if(t==g)
{
cout<<"Same"<<endl;
}
else
{
cout<<"Not same"<<endl;
}
if(t==k)
{
cout<<"Same"<<endl;
}
else
{
cout<<"Not same"<<endl;
}
if(t<k)
{
cout<<t<<" is less than "<<k<<endl;
}
else
{
cout<<t<<" is not less than "<<k<<endl;
}
TMString m;
m=t+k;
cout<<m<<endl;
return 0;
}
```