Object-oriented Programming

Data Structures

YoungWoon Cha

Computer Science and Engineering

INTRODUCTION

In previous chapters, we have learned how handle objects in C++ individually.

In computer science, objects are often collections.

This means that we need to handle a collection of objects instead of individual objects.

The techniques we have learned so far can be applied to each object in the collection, but we also need to think about the collection as an object of objects.

We need to learn how to insert an object, how to erase an object, and how to access an object in a collection.

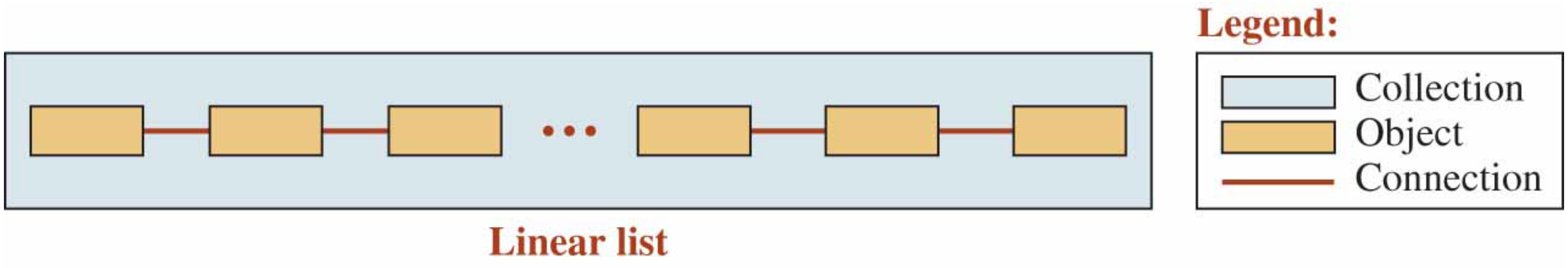
Objects Relationship Part 1

Operations on a collection, as an object of objects, depends on the relationship between the objects in the collection.

We normally encounter two general relationships in a collection: linear and non-linear.

Linear Collection

A collection may impose a linear relationship between objects in a collection which means that each object is somehow connected to the previous and next object.



Objects Relationship Part 2

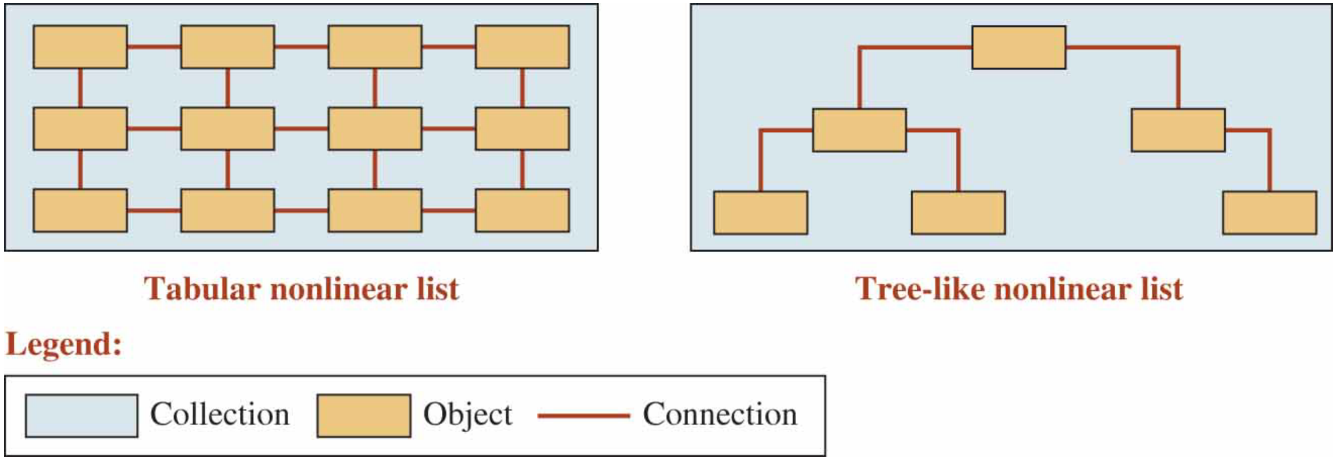
Non-Linear Collection

In a non-linear collection, each object can be connected to a group of objects.

The objects can be related to each other in a tabular relationship.

We can also have a collection in which the objects are related to each other the way branches in a tree are connected.

We can have a root and branches, each with a number of objects, in an up-side-down tree.



Tabular nonlinear list

Tree-like nonlinear list

Chapter Goal

Our main goal in this chapter is to learn how to implement some simple collections.

Although, several collections have been implemented in the STL library for us, if we learn how to create our own collections, it will be easier for us to understand the relationship between object in STL collections and how to derive our own collection from the STL.

Singly Linked List

Stack

Queue

Binary Search Tree

Singly Linked List

SINGLY LINKED LIST

We develop a linear implementation collection of objects using a singly linked list.

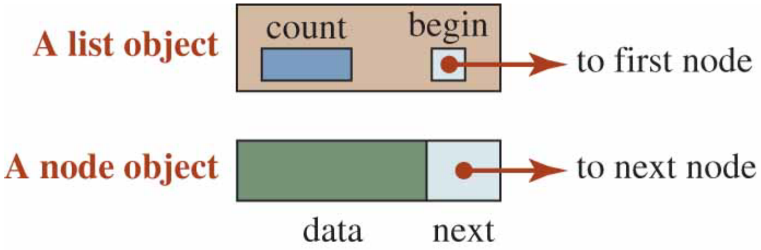
In this implementation, each object in the collection is related only to the next object in the collection (thus the term singly).

The object in a singly linked list is called a node which is made of two parts: data and a pointer.

The data section defines the value of the object; the pointer section points to the next node.

In a singly linked list we can go to the next node from any node, but we cannot go back to the previous node.

To design a container as a singly linked list, we use two different object types: *list* and *node*.

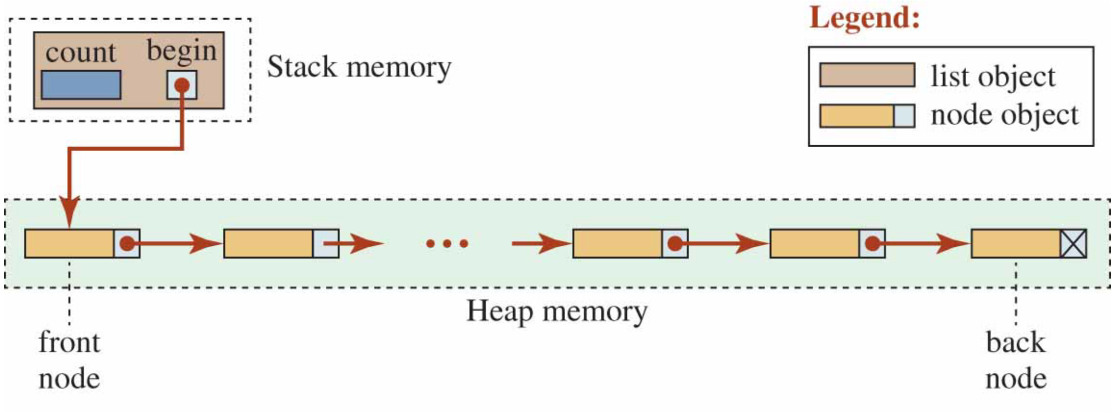


Note:
We refer to the link in a singly linked list as *next*.

Design

Creation of Objects

The list object is created in stack memory because we use only one instance of it; the nodes are created in heap memory because the number of nodes changes as we insert or erase nodes.



Note:
The variable *begin* is a pointer to the front node.

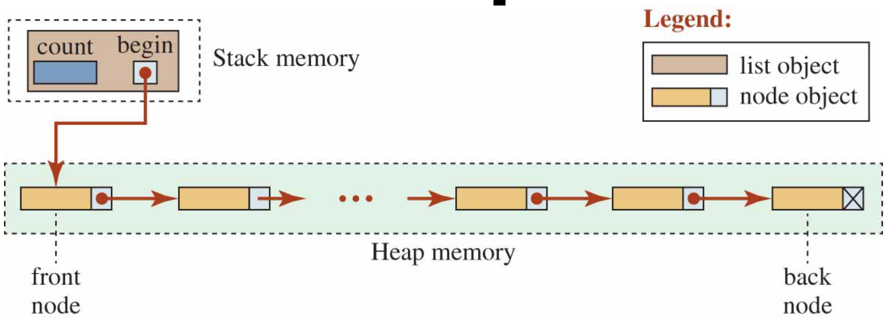
In this design, the list object has a pointer, *begin*, and an integer data type, *count*, that holds the number of nodes in the linked list.

To be able to access the *data* and the *next* members of a node from the list object, we define a node as a *struct*, instead of a *class* because we can access its data member from the list class directly; the members of a *struct* are public by default.

Interface File for a List Class

```
#ifndef LIST_H
#define LIST_H
#include <iostream>
#include <cassert>
using namespace std;

// Definition of the Node as a struct
template <typename T>
struct Node
{
        T data;
        Node <T>* next;
};
// Definition of the class List
template <typename T>
class List
{
        private:
                Node <T>* begin;
                int count;
                Node <T>* makeNode (const T& value);
        public:
                List ();
                ~List ();
                void insert (int pos, const T& value);
                void erase (int pos);
                T& get (int pos) const;
                void print () const;
                int size () const;
};
#endif
```

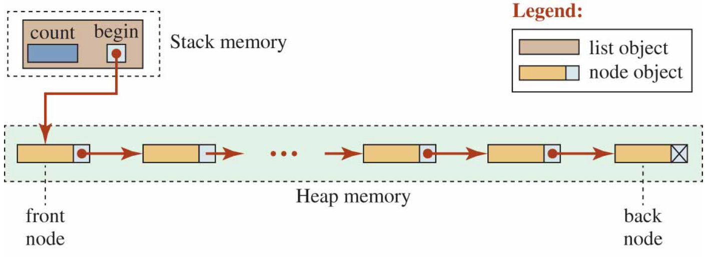
```
#ifndef LIST_CPP
#define LIST_CPP
#include "List.h"

// Constructor
template <typename T>
List <T> :: List ()
:begin (0), count (0)
{
}
// Destructor
template <typename T>
List <T> :: ~List ()
{
        Node <T>* del = begin;
        while (begin)
        {
                begin = begin -> next;
                delete del;
                del = begin;
        }
}
```



Note:
The variable *begin* is a pointer to the front node.

Implementation File for a List Class Part 1

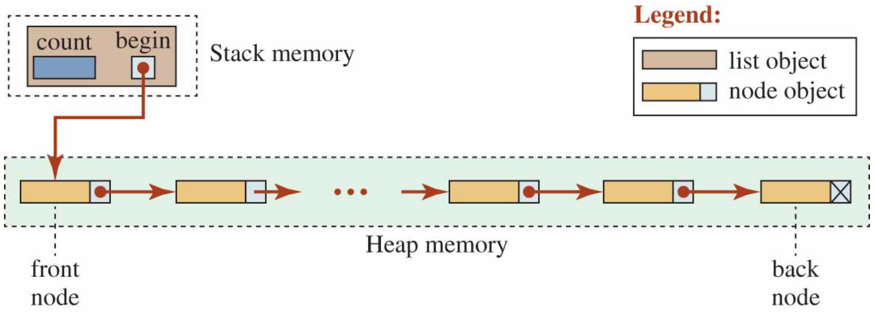
```
// Insert member function
template <typename T>
void List <T> :: insert (int pos, const T& value)
{
        if (pos < 0 || pos > count)
        {
              cout << "Error! The position is out
              of range." << endl;
              return;
        }
        Node <T>* add = makeNode (value);
        if (pos == 0)
        {
              add -> next = begin;
              begin = add;
        }
        else
        {
              Node <T>* cur = begin;
              for (int i = 1; i < pos; i++)
              {
                        cur = cur -> next;
              }
              add -> next = cur -> next;
              cur -> next = add;
        }
        count++;
}
// MakeNode member function (private)
template <typename T>
Node <T>* List <T> :: makeNode (const T& value)
{
        Node <T>* temp = new Node <T>;
        temp -> data = value;
        temp -> next = 0;
        return temp;
}
```



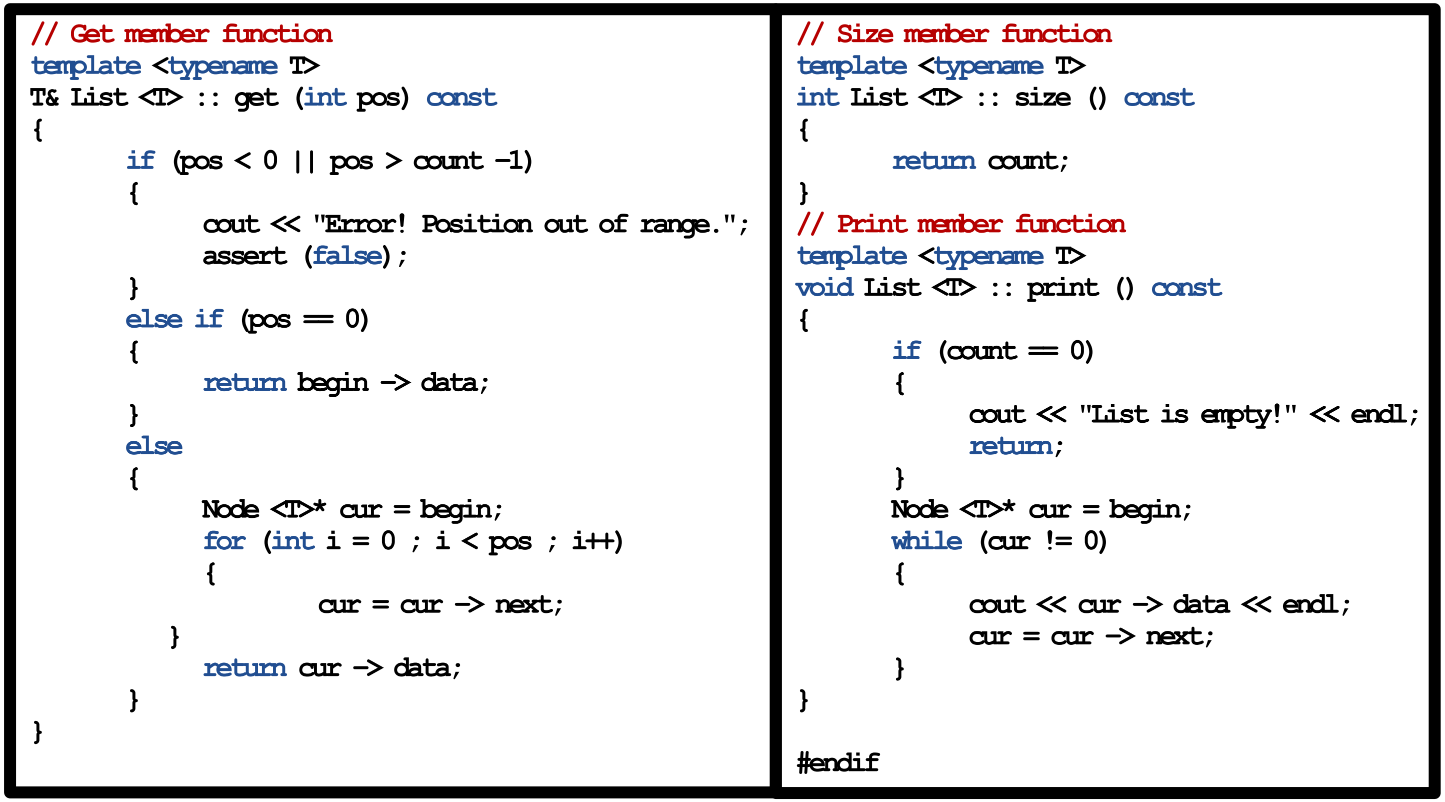
Note:
The variable *begin* is a pointer to the front node.

```
// Erase member function
template <typename T>
void List <T> :: erase (int pos)
{
        if (pos < 0 || pos > count - 1)
        {
              cout << "Error! The position is out of range." << endl;
              return;
        }
        if (pos == 0)
        {
              Node <T>* del = begin;
              begin = begin -> next;
              delete del;
        }
        else
        {
              Node <T>* cur = begin;
              for (int i = 0; i < pos - 1; i++)
              {
                        cur = cur -> next;
              }
              Node <T>* del = cur -> next;
              cur -> next = cur -> next -> next;
              delete del;
        }
        count--;
}
```

Implementation File for a List Class Part 2



Note:
The variable *begin* is a pointer to the front node.

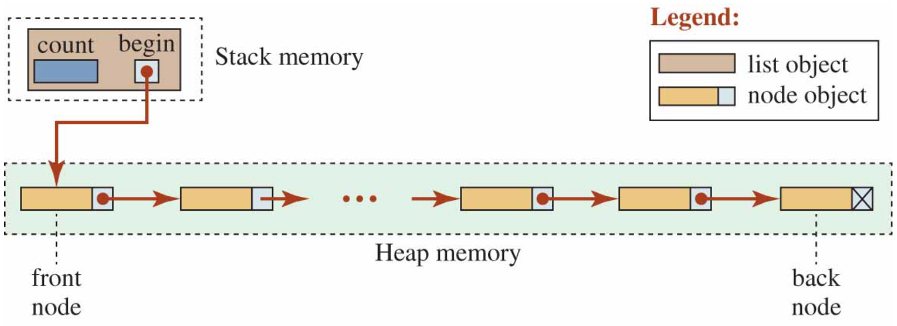


```
// Get member function
template <typename T>
T& List <T> :: get (int pos) const
{
        if (pos < 0 || pos > count -1)
        {
              cout << "Error! Position out of range.";
              assert (false);
        }
        else if (pos == 0)
        {
              return begin -> data;
        }
        else
        {
              Node <T>* cur = begin;
              for (int i = 0 ; i < pos ; i++)
              {
                        cur = cur -> next;
           }
              return cur -> data;
        }
}
```

```
// Size member function
template <typename T>
int List <T> :: size () const
{
        return count;
}
// Print member function
template <typename T>
void List <T> :: print () const
{
        if (count == 0)
        {
              cout << "List is empty!" << endl;
              return;
        }
        Node <T>* cur = begin;
        while (cur != 0)
        {
              cout << cur -> data << endl;
              cur = cur -> next;
        }
}

#endif
```

Application File for a List Class

```
#include "list.cpp"
#include <string>

int main ( )
{
        // Instantiation of a list object
        List <string> list;
        // Inserting six nodes in the list
        list.insert (0, "Michael");
        list.insert (1, "Jane");
        list.insert (2, "Sophie");
        list.insert (3, "Thomas");
        list.insert (4, "Rose");
        list.insert (5, "Richard");
        // Printing the values of nodes
        cout << "Printing the list" << endl;
        list.print ();
        // Printing the values of three nodes
        cout << "Getting data in some nodes" << endl;
        cout << list.get (0) << endl;
        cout << list.get (3) << endl;
        cout << list.get (5) << endl;
        // Erasing three nodes from the list
        cout << "Erasing some nodes and printing after
        erasures" << endl ;
        list.erase (0);
        list.erase (3);
        list.print ();
        // Printing the list after erasures
        cout << "Checking the list size" << endl ;
        cout << "List size: " << list.size () ;
        return 0;
}
```



Note:
The variable *begin* is a pointer to the front node.

```
Run:
Printing the list
Michael
Jane
Sophie
Thomas
Rose
Richard
Getting data in some nodes
Michael
Thomas
Richard
Erasing some nodes and printing after erasures
Jane
Sophie
Thomas
Richard
Checking the list size
List size: 4
```

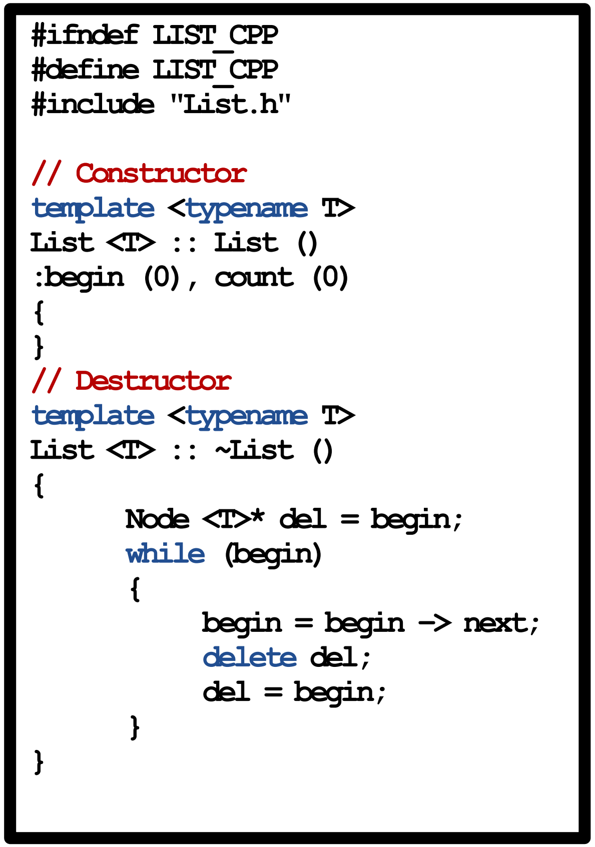
Singly Linked List Operations

Constructor

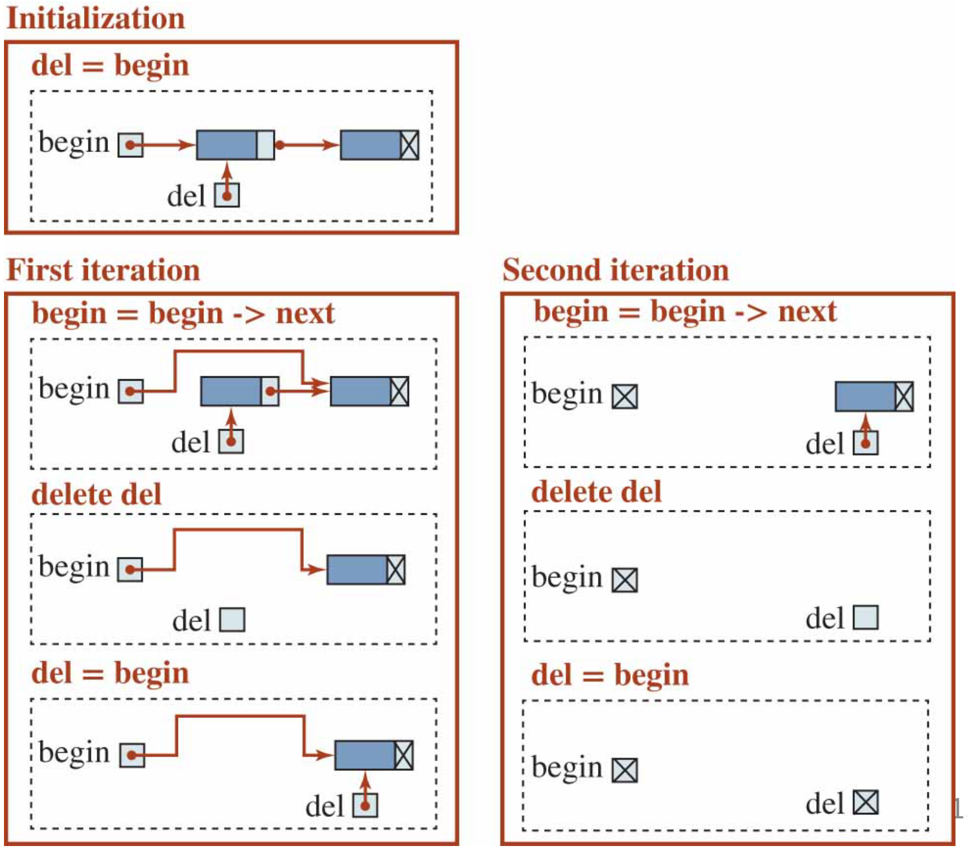
We have used a default constructor by storing 0 in the *count* member and a null pointer (0) in the *begin* member.

Destructor

The destructor deletes all nodes one by one and frees the heap memory.



```
#ifndef LIST_CPP
#define LIST_CPP
#include "List.h"

// Constructor
template <typename T>
List <T> :: List ()
:begin (0), count (0)
{
}
// Destructor
template <typename T>
List <T> :: ~List ()
{
        Node <T>* del = begin;
        while (begin)
        {
                begin = begin -> next;
                delete del;
                del = begin;
        }
}
```



Singly Linked List Operations

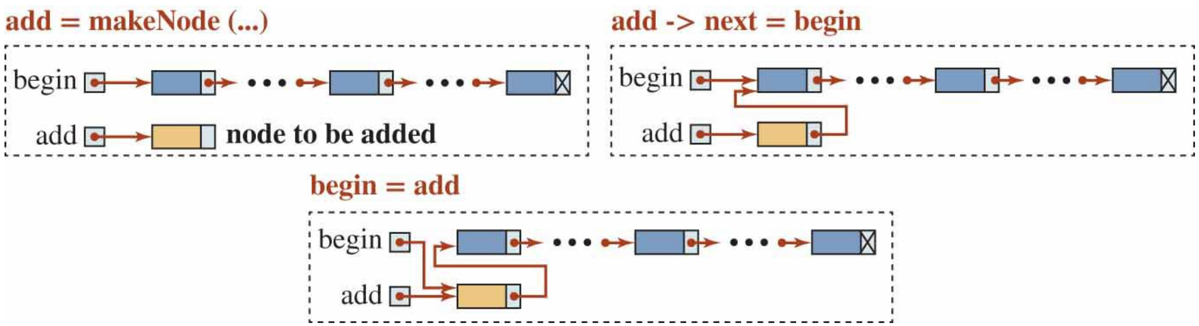
Insertion

Insertion is done at a specified location.

We can have three cases: insertion at the front, insertion at the middle, or insertion at the end.

Insertion at the beginning can be done using two operations (after making a node).

```
// Insert member function
template <typename T>
void List <T> :: insert (int pos, const T&
value)
{
        if (pos < 0 || pos > count)
        {
                cout << "Error! The position is
                out of range." << endl;
                return;
        }
        Node <T>* add = makeNode (value);
        if (pos == 0)
        {
                add -> next = begin;
                begin = add;
        }
        else
        {
                Node <T>* cur = begin;
                for (int i = 1; i < pos; i++)
                {
                            cur = cur -> next;
                }
                add -> next = cur -> next;
                cur -> next = add;
        }
        count++;
}
```



add = makeNode (...)

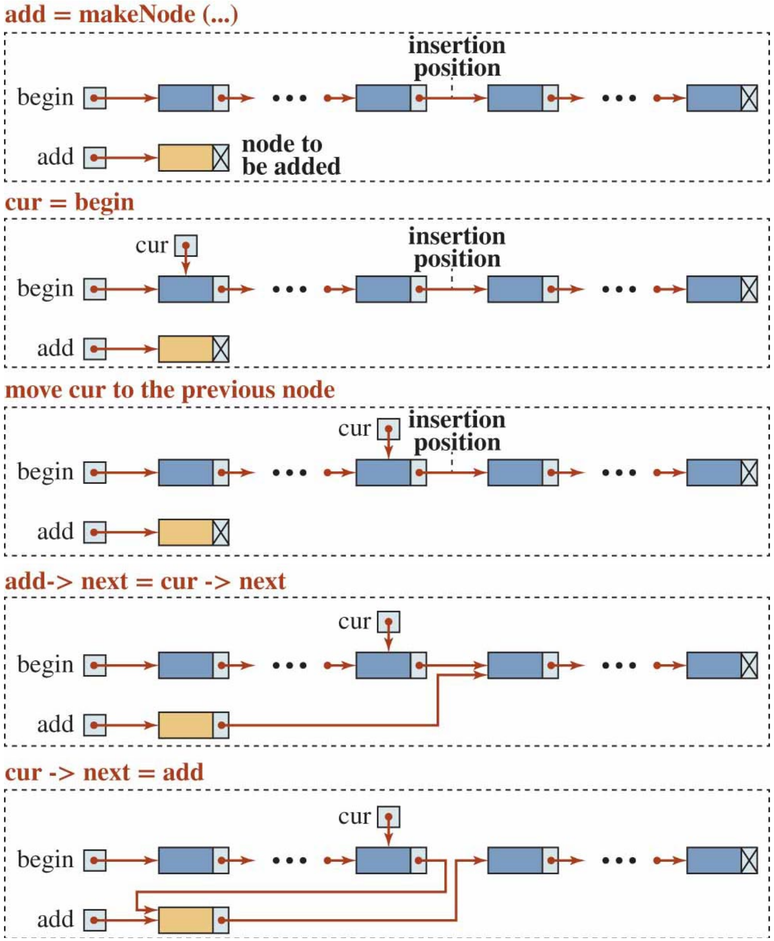
add -> next = begin

begin = add

```
// MakeNode member function (private)
template <typename T>
Node <T>* List <T> :: makeNode (const T& value)
{
        Node <T>* temp = new Node <T>;
        temp -> data = value;
        temp -> next = 0;
        return temp;
}
#endif
```

Singly Linked List Operations

For insertion at the middle, we need to have a *cur* pointer and move it to the node located before the position of insertion. We then can insert the node.



add = makeNode (...)

cur = begin

move cur to the previous node

add-> next = cur -> next

cur -> next = add

```
// Insert member function
template <typename T>
void List <T> :: insert (int pos, const T&
value)
{
        if (pos < 0 || pos > count)
        {
                cout << "Error! The position is
                out of range." << endl;
                return;
        }
        Node <T>* add = makeNode (value);
        if (pos == 0)
        {
                add -> next = begin;
                begin = add;
        }
        else
        {
                Node <T>* cur = begin;
                for (int i = 1; i < pos; i++)
                {
                        cur = cur -> next;
                }
                add -> next = cur -> next;
                cur -> next = add;
        }
        count++;
}
```

Insertion at the end is a special case of insertion at the middle in which the *cur* pointer should move to the last node.

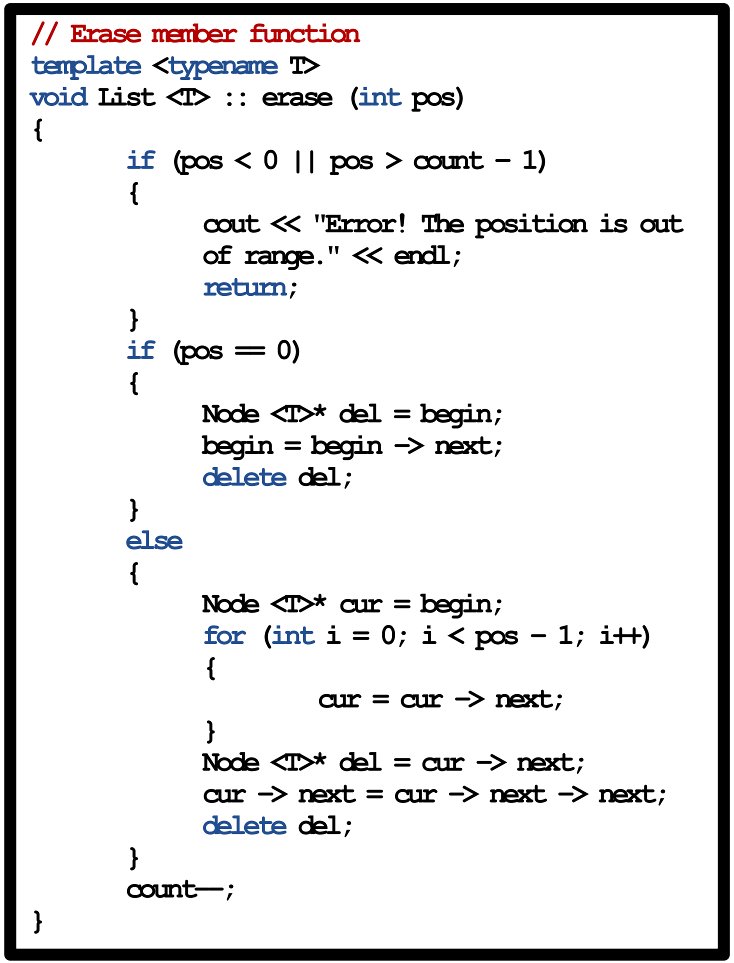
The new node is inserted after the last node.

Singly Linked List Operations

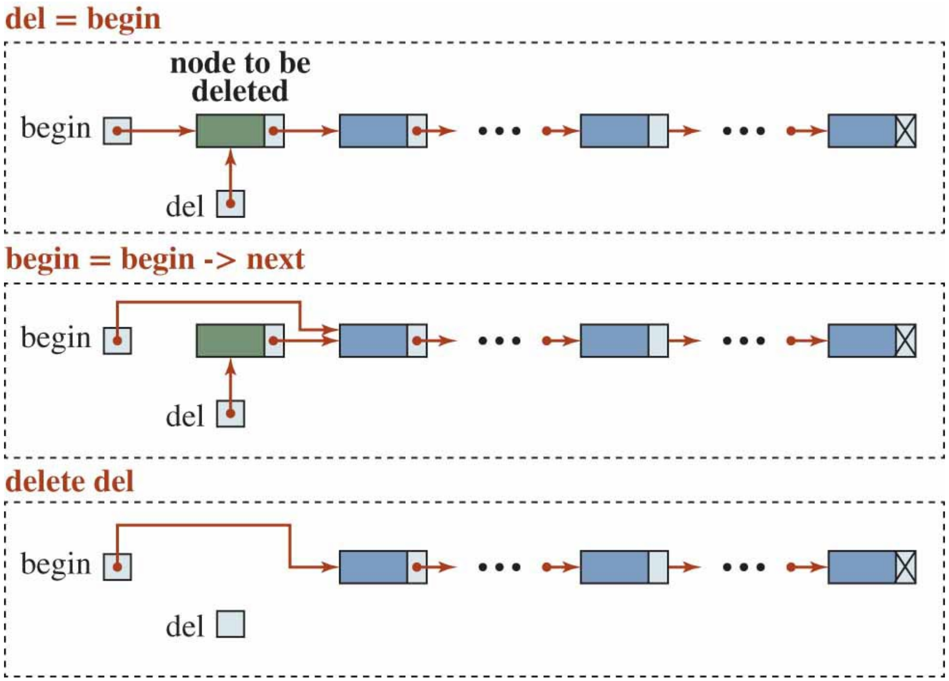
Erasure

Erasure is applied to a specific node.

If the list is not empty, we have three cases: erasure of the first node, erasure of a middle node, or erasure of the last node.



```
// Erase member function
template <typename T>
void List <T> :: erase (int pos)
{
        if (pos < 0 || pos > count - 1)
        {
              cout << "Error! The position is out
              of range." << endl;
              return;
        }
        if (pos == 0)
        {
              Node <T>* del = begin;
              begin = begin -> next;
              delete del;
        }
        else
        {
              Node <T>* cur = begin;
              for (int i = 0; i < pos - 1; i++)
              {
                        cur = cur -> next;
              }
              Node <T>* del = cur -> next;
              cur -> next = cur -> next -> next;
              delete del;
        }
        count--;
}
```



Singly Linked List Operations

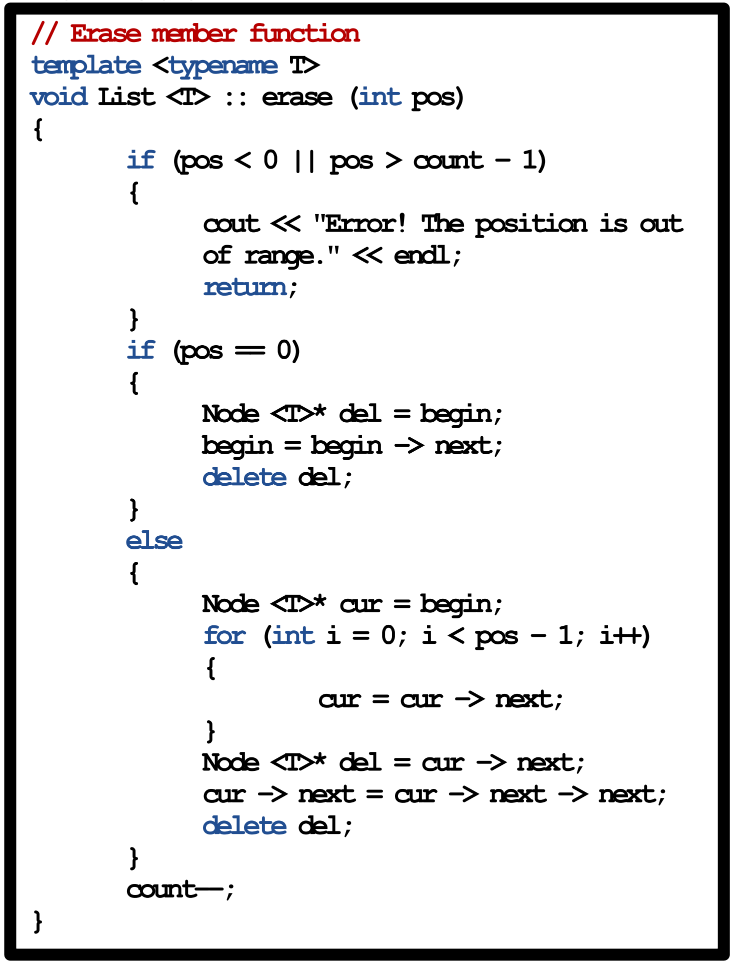
The erasure of a middle node is more involved.

We need to have a *cur* pointer to point to the node before the one to be deleted.

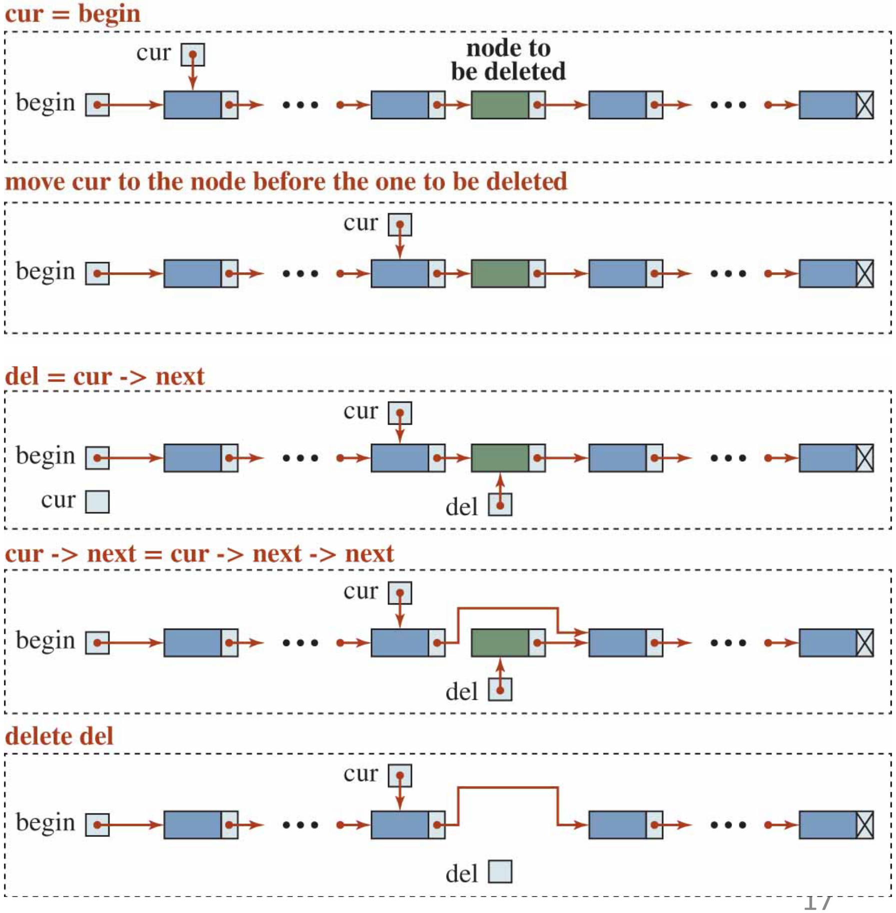
We can then use another pointer, *del*, to point to the node to be erased.

We can then erased the node.

The eraser at the end is similar, but we need to move the *cur* pointer to the node before the last node.



```
// Erase member function
template <typename T>
void List <T> :: erase (int pos)
{
        if (pos < 0 || pos > count - 1)
        {
              cout << "Error! The position is out
              of range." << endl;
              return;
        }
        if (pos == 0)
        {
              Node <T>* del = begin;
              begin = begin -> next;
              delete del;
        }
        else
        {
              Node <T>* cur = begin;
              for (int i = 0; i < pos - 1; i++)
              {
                        cur = cur -> next;
              }
              Node <T>* del = cur -> next;
              cur -> next = cur -> next -> next;
              delete del;
        }
        count--;
}
```



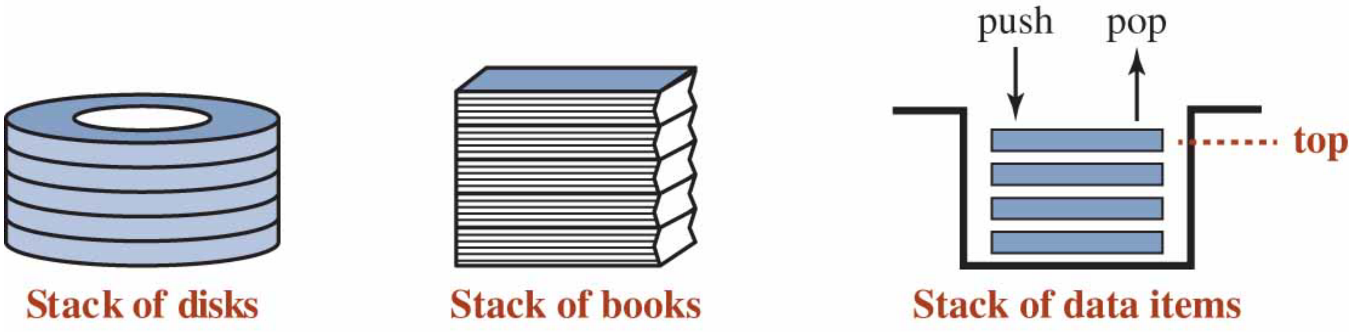
Stack

Stacks Part 1

A stack is a container implemented as a linear list in which all additions and deletions are restricted to one end, called the *top*.

If we insert data items into a stack and then remove them, the order of the data items would be reversed.

Data input as {5, 10, 8, 20} would be removed as {20, 8, 10, 5}. This reversing attribute is why stacks are known as a *last in–first out* (*LIFO*) data structure.



We use many different types of stacks in our daily lives.

We often talk of a stack of coins or a stack of books.

Any situation in which you can only add or remove an object at the top is a stack.

Stacks Part 2

Stack Operations

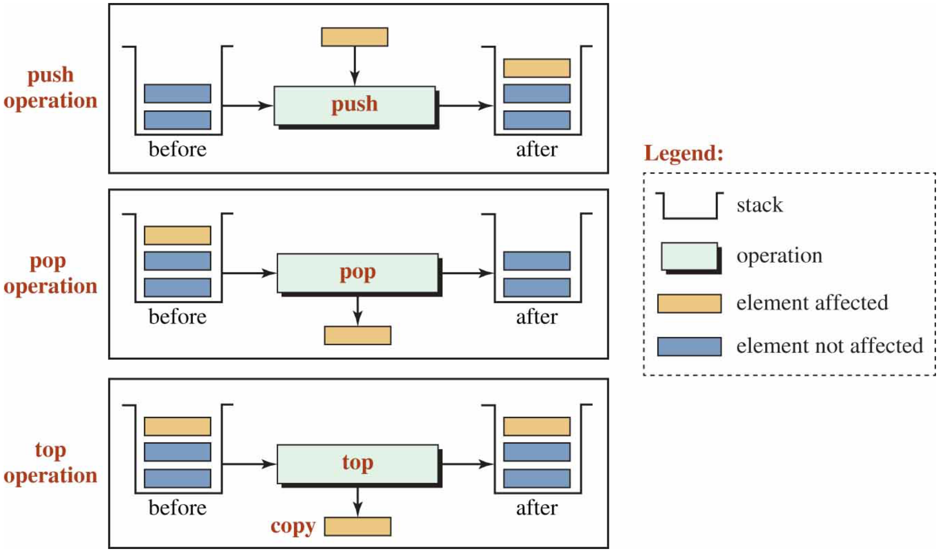
We normally encounter three basic operations for a stack data structure.

Stack Implementation

A stack can be implemented either as an array or a linked-list.

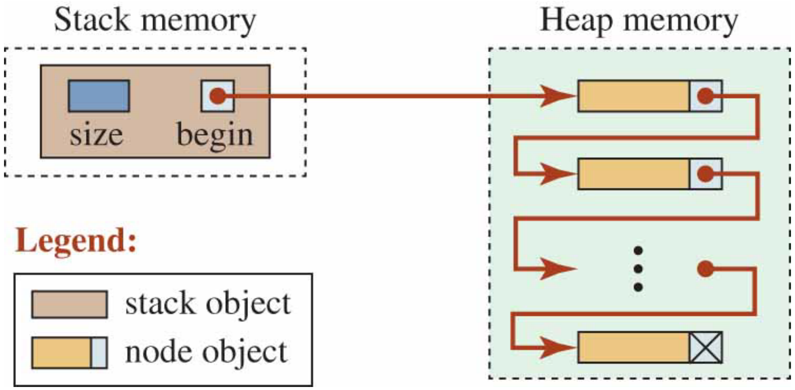
In the array implementation is easier, the size of the array needs to be fixed at compilation time.

A linked-list implementation allows the size of the stack to grow and shrink dynamically.

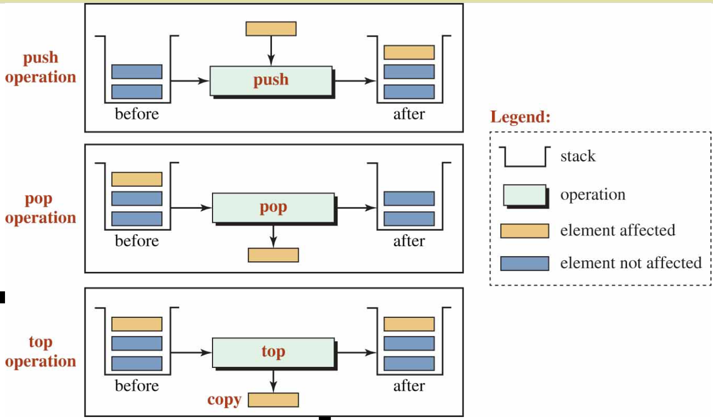


In the figure, we have one stack object and a number of node objects.

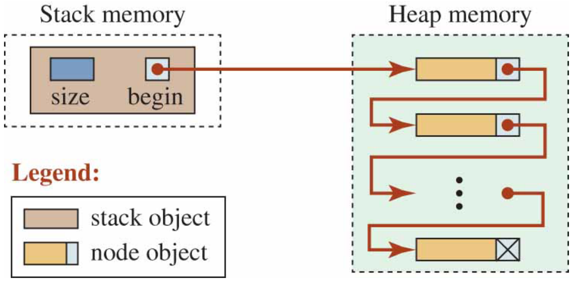
The nodes are created in the heap because the numbers of nodes grows with each push operation and shrinks with each pop operation.



Implementation for the Stack class



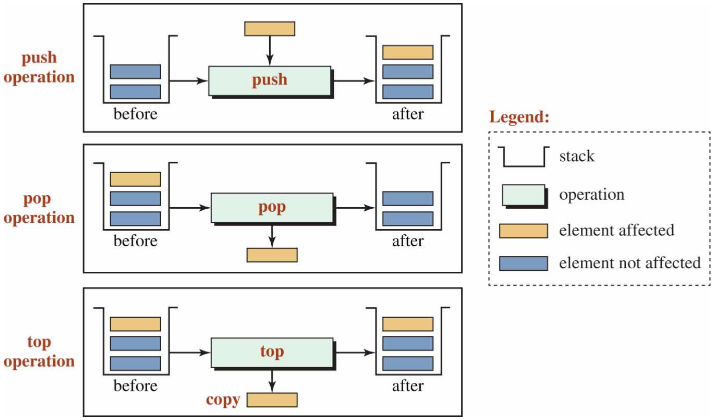
```
#ifndef STACK_H
#define STACK_H
#include "list.cpp"

// Stack class definition composing a list object
template <typename T>
class Stack
{
        private:
                List <T> list;
        public:
                void push (const T& data);
                void pop ();
                T& top() const;
                int size() const;
};
#endif
```



```
#ifndef STACK_CPP
#define STACK_CPP
#include "stack.h"

// Definition of the push member function
template <typename T>
void Stack <T> :: push (const T& value)
{
        list.insert (0, value);
}
// Definition of the pop member function
template <typename T>
void Stack <T> :: pop ()
{
        list.erase (0);
}
// Definition of the top member function
template <typename T>
T& Stack <T> :: top () const
{
        return list.get(0);
}
// Definition of the size member function
template <typename T>
int Stack <T> :: size () const
{
        return list.size ();
}
#endif
```

Simple Application To Test The Stack Class

```
#include "stack.cpp"

int main ( )
{
        // Instantiation of a Stack object
        Stack <string> stack;
        // Pushing four nodes into the stack
        stack.push ("Henry");
        stack.push ("William");
        stack.push ("Tara");
        stack.push ("Richard");
        // Testing the size of the stack after four push
        cout << "Stack size: " << stack.size () << endl;
        // Continuously get the value of the top node and pop it from the stack
        while (stack.size () > 0)
        {
              cout << "Node value at the top: " << stack.top () << endl;
              stack.pop ();
        }
        // Recheck the size after all elements are popped out
        cout << "Stack size: " << stack.size ();
        return 0;
}
```



```
Run:
Stack size: 4
Node value at the top: Richard
Node value at the top: Tara
Node value at the top: William
Node value at the top: Henry
Stack size: 0
```

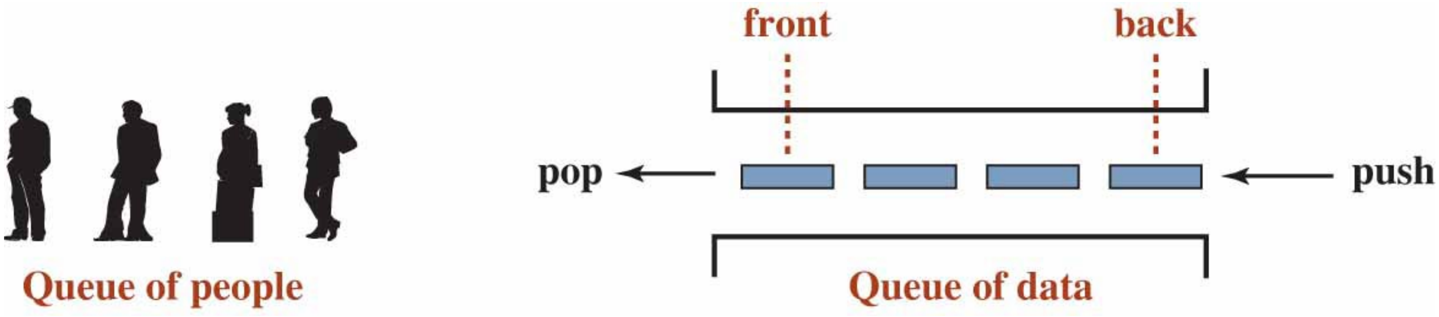
Queue

Queues Part 1

A *queue* is a linear list in which data can only be inserted at one end, called *back*, and deleted from the other end, called *front*.

These restrictions ensure that the data are output through the queue in the order in which they are input.

In other words, a queue is a *first in–first out* (*FIFO*) structure.



A *queue* is the same as a line.

A line of people waiting for the bus in a bus station is a *queue*; a list of calls put on hold to be answered by a telephone operator is a *queue*; and a list of waiting jobs to be processed by a computer is a *queue*.

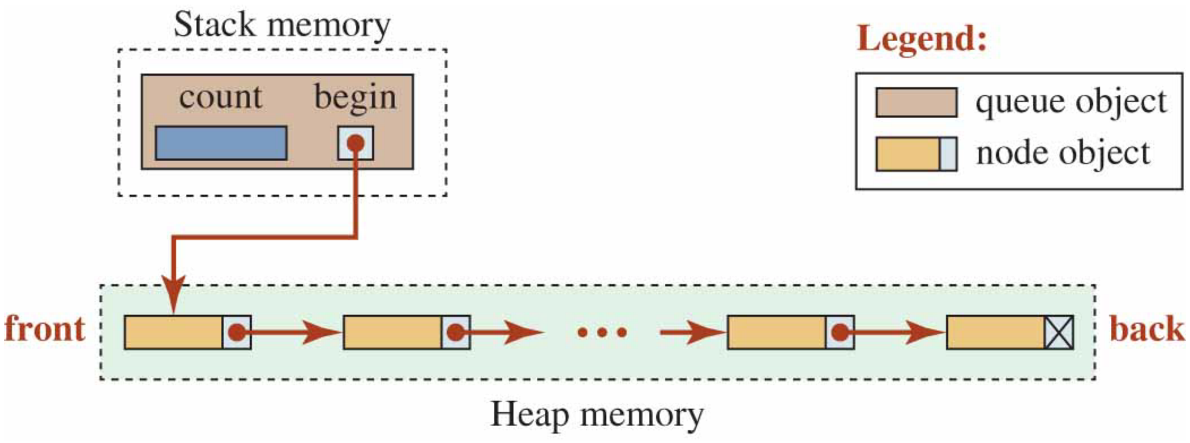
Queues Part 2

Implementation

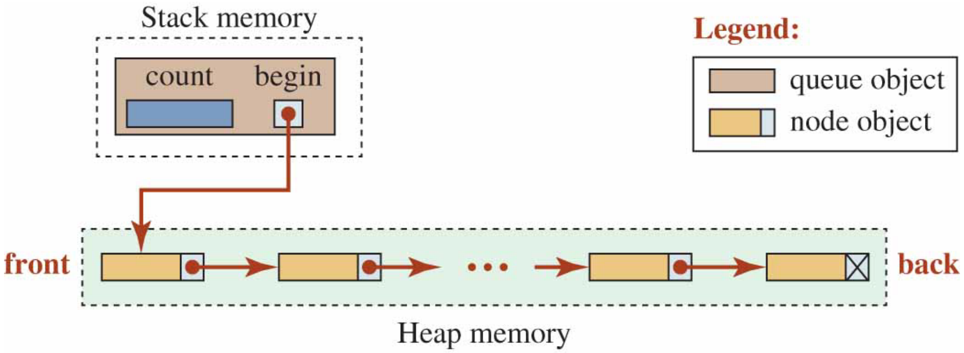
A *queue* can be implemented either as an array or a linked-list.

Although the array implementation is easier, the size of the array needs to be fixed at compilation time.

A linked-list implementation allows the size of the queue to grow and shrink dynamically.



Implementation for the Queue Class

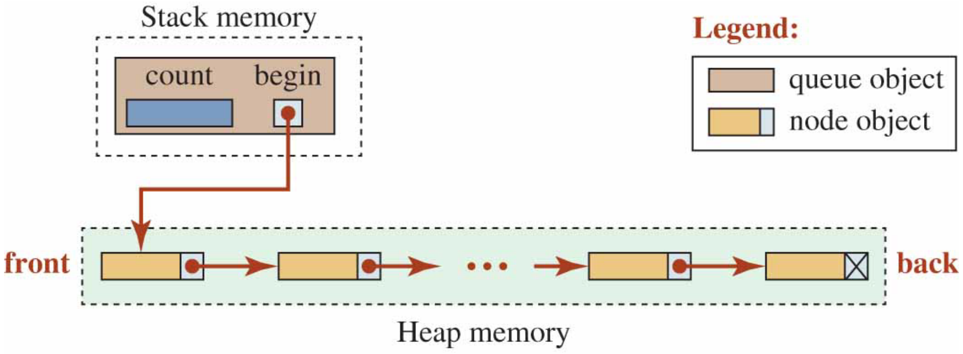


```cpp
#ifndef QUEUE_H
#define QUEUE_H
#include "list.cpp“

template <class T>
class Queue
{
        private:
                List <T> list;
        public:
                void push (const T& data);
                void pop ();
                T& front() const;
                T& back() const;
                int size() const;
                void print() const;
};
#endif
```

```cpp
#ifndef QUEUE_CPP
#define QUEUE_CPP
#include "queue.h"
#include "list.cpp"

// Definition of push operation
template <typename T>
void Queue <T> :: push (const T& value)
{
        list.insert (list.size (), value);
}
// Definition of pop operation
template <typename T>
void Queue <T> :: pop ()
{
        list.erase (0);
}
// Definition of print operation
template <typename T>
void Queue <T> :: print () const
{
        list.print ();
}
```

```cpp
// Definition of front operation
template <typename T>
T& Queue <T> :: front () const
{
        return list.get( 0);
}
// Definition of back operation
template <typename T>
T& Queue <T> :: back () const
{
        return list.get (list.size () – 1);
}
// Definition of size operation
template <typename T>
int Queue <T> :: size () const
{
        return list.size ();
}

#endif
```

Application File to Test Queue Class



```
#include "queue.cpp"

int main ( )
{
       // Instantiation of a queue object
       Queue <string> queue;
       // Pushing four nodes into the queue
       queue.push ("Henry");
       queue.push ("William");
       queue.push ("Tara");
       queue.push ("Richard");
       // Checking the element at the front and the back of the queue
       cout << "Checking front and back elements";
       cout << "after four push operations:" << endl;
       cout << "Element at the front: " << queue.front () << endl;
       cout << "Element at the back: " << queue.back () << endl << endl;
       // Popping two elements from the queue
       queue.pop ();
       queue.pop ();
       // Checking the front and the back node after two pop operations
       cout << "Checking front and back elements";
       cout << "after two pop operations:" << endl;
       cout << "Element at the front: " << queue.front () << endl;
       cout << "Element at the back: " << queue.back () << endl;
       return 0;
}
```

```
Run:
Checking front and back elements after four push
operations:
Element at the front: Henry
Element at the back: Richard

Checking front and back elements after two pop
operations:
Element at the front: Tara
Element at the back: Richard
```

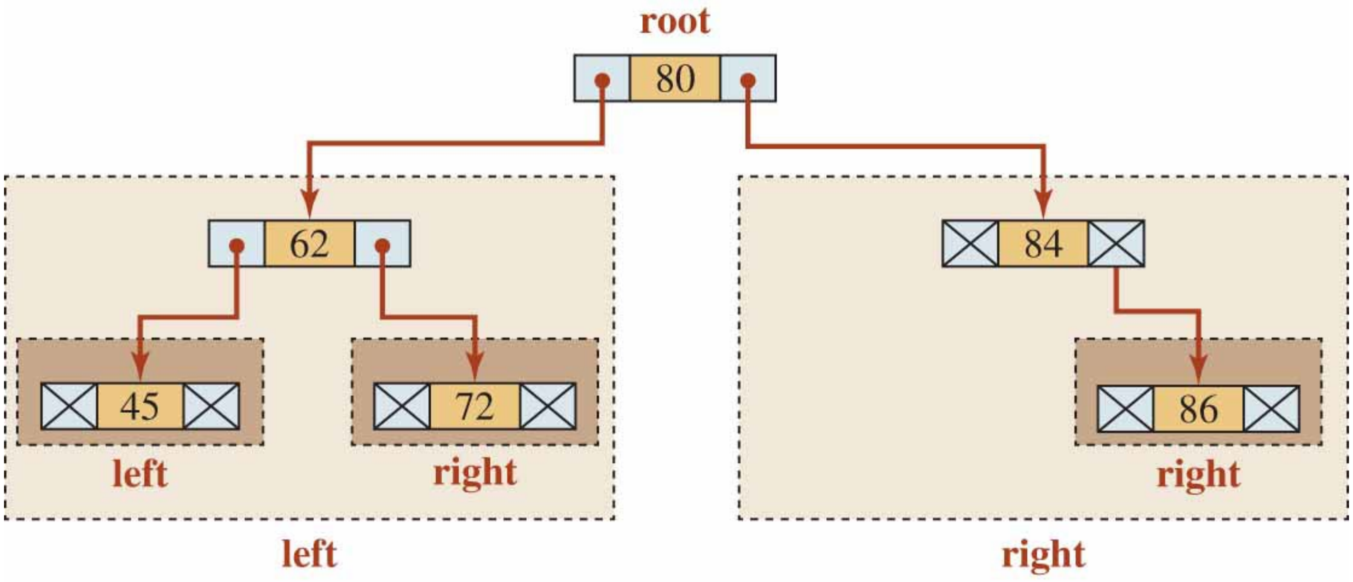
Binary Search Tree

BINARY SEARCH TREES

Linked lists, stacks, and queues are linear collections; a *tree* is a non-linear one.

In a general tree, each node can have two or more links to other nodes.

Although general trees have many applications in computer science (such as directories) we encounter more *binary trees*, trees with a maximum of two subtrees.



A special case of a binary tree is called a *binary search tree* in which the values in the nodes in the left subtree are less than the value in the root and values in the right subtree are greater that the value in the root.

Traversals

A binary search tree traversal requires that each node of the tree be processed once and only once in a predetermined sequence.

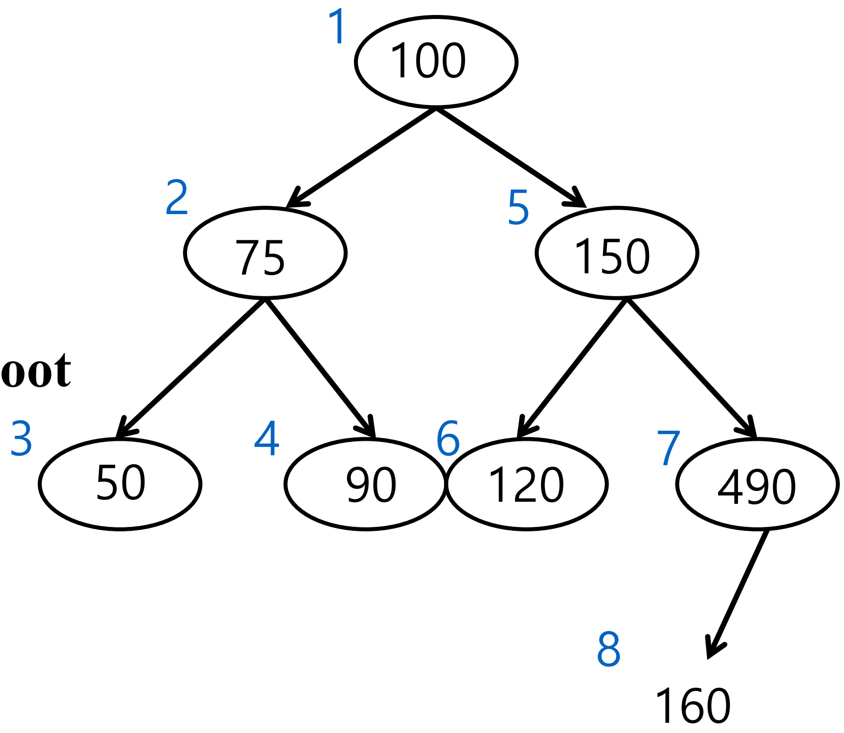
We discuss three common traversals: *pre-order*, *in-order*, and *post-order*.

Pre-order Traversal: Mother, First Child, Second Child

In the *pre-order traversal*, the root node is processed first, followed by all the nodes in the left subtree traversed in pre-order, and then all the nodes in the right subtree traversed in pre-order.

Note that the root comes at the beginning.

The pre-order traversal is useful whenever we want to access the root in the tree or in the subtree first.



100
75
50
90
150
120
490
160

Traversals Part 2

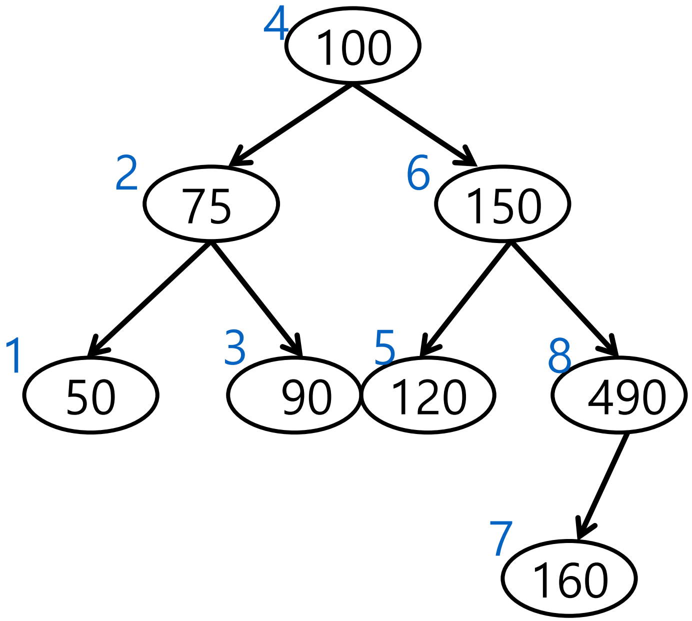
In-order Traversal: First Child, Mother, Second Child

In the *in-order traversal*, the processing of the root comes between the two subtree.

In other words, we need to traverse the whole left subtree first, then the root, then the right subtree.

Note that in the in-order traversal of a binary search tree, the values are processed in ascending sequence.

The root of the whole tree comes in the middle.



50
75
90
100
120
150
160
490

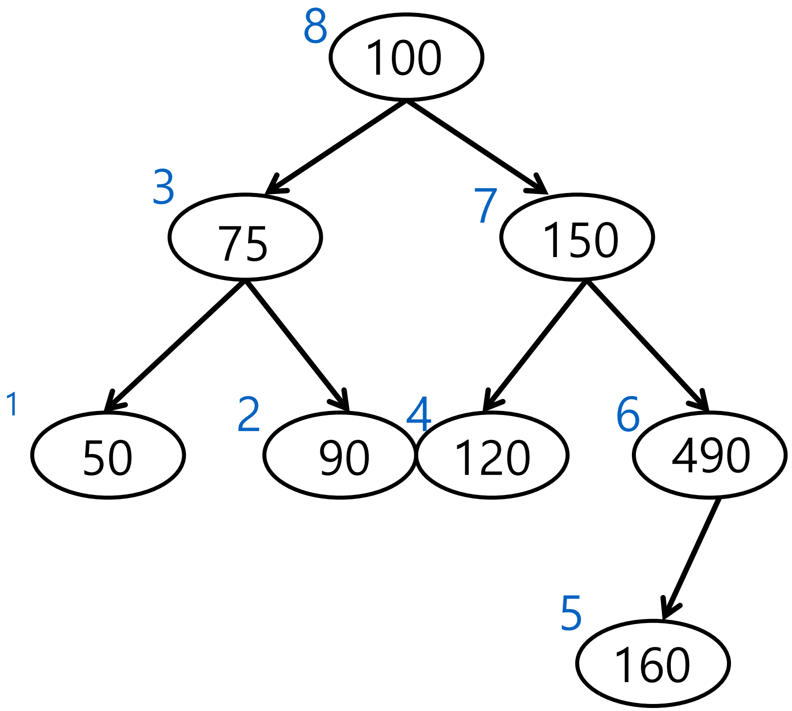
Traversals Part 3

Post-order Traversal: First Child, Second Child, Mother

In the *post-order traversal*, the processing of the root comes after the processing of two subtrees.

In other words, we need to traverse the left subtree first, then the right subtree, and finally the root.

Note that the root of the whole tree comes at the end.



50
90
75
120
160
490
150
100

BST Implementation

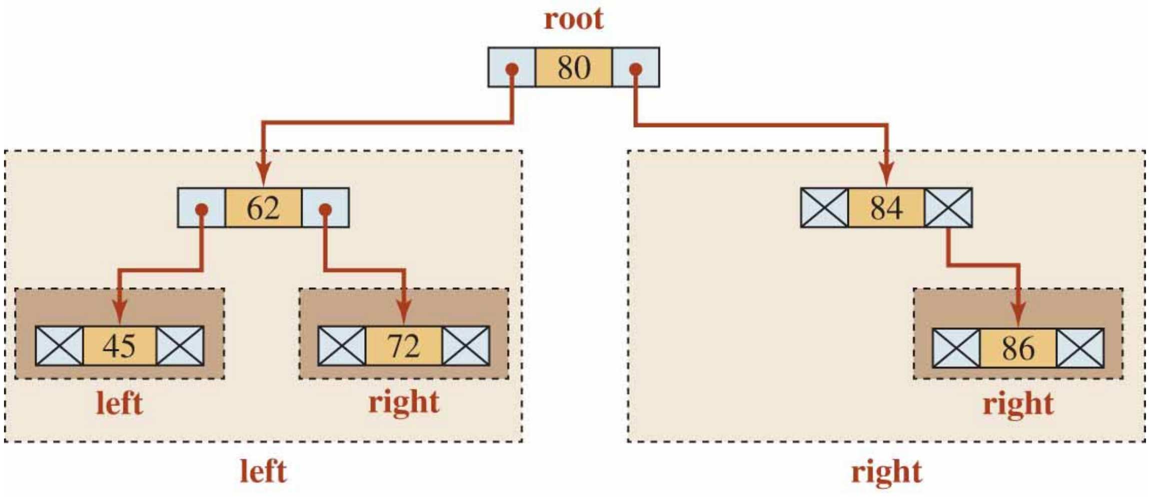
We discuss three traversals for a binary search tree.

The reason is that when a binary search tree is implemented correctly, the data values in the binary search tree are sorted, which means that we can search the tree easier.

This is the reason that this tree is called a binary search tree.

The three traversals help us build a binary search tree, search a binary search tree, and destroy a binary search tree.

However, for each activity, we need to use the correct traversal.



BST Implementation Part 2

Insertion

How do we insert nodes in a binary search tree?

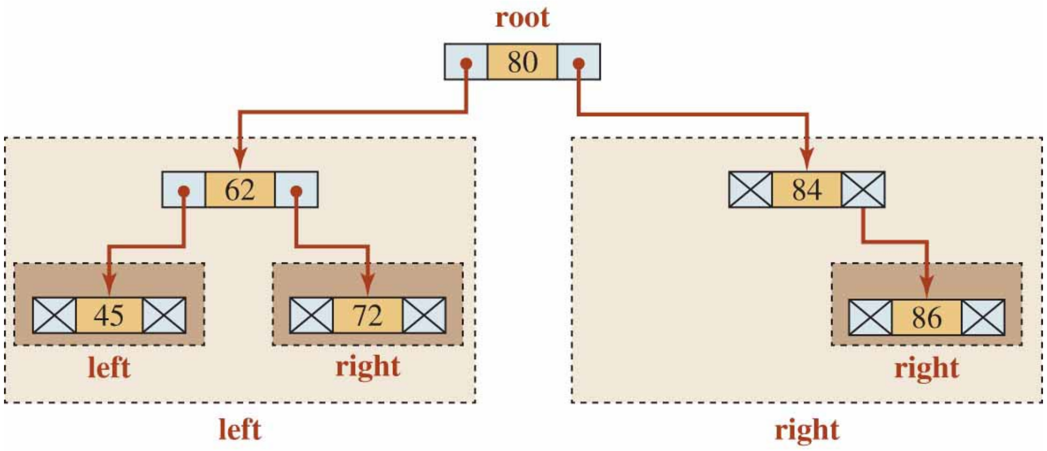
The answer is that we need to find the position of the node to be inserted.

The search for the position should be start from the root. We need to insert the root, then the left subtree or the right subtree.

In each subtree, we should do the same.

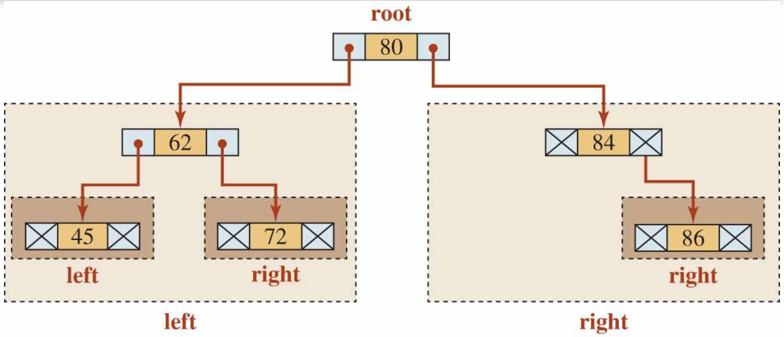
This means we can write a recursive algorithm as shown below.

```
If the tree is empty, insert as the root.                                   // Base case
Insert at the left subtree if value is less than root value.                // General case
Insert at the right subtree if value is greater than root value.            // General case
```



BST Implementation Part 3

Destruction



How do we destroy a binary search tree?

The answer is that we do it node by node.

However to delete a node, the left subtree and right subtree need to be empty.

This gives a clue that destroying a binary search tree needs to use the post-order traversal because the last node that needs to be destroyed is the root.

We need to destroy the left subtree, then destroy the right subtree, and finally destroy the root.

This means we can write a recursive algorithm as shown below.

Destroy the left subtree.	// General case
Destroy the right subtree.	// Base case
Delete data item in the root	// General case

BST Implementation Part 4

Printing

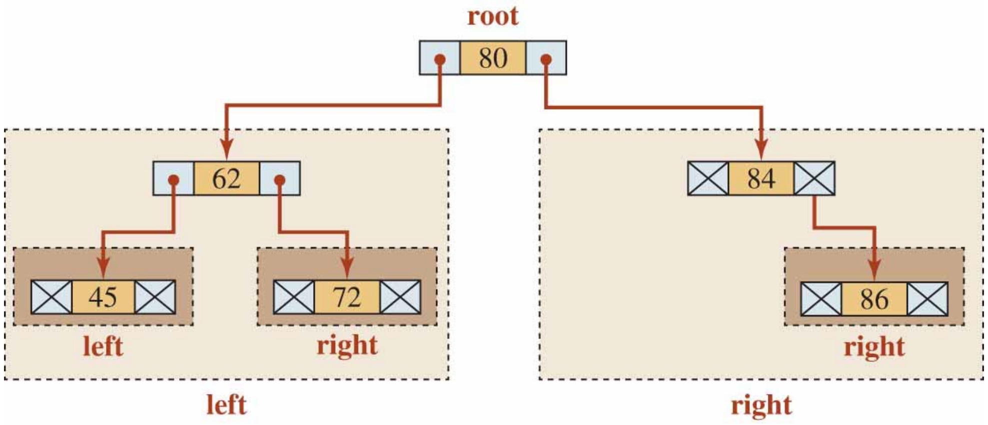
How do we print the items in a binary search tree to get a sorted list?

The answer is that we do it node by node.

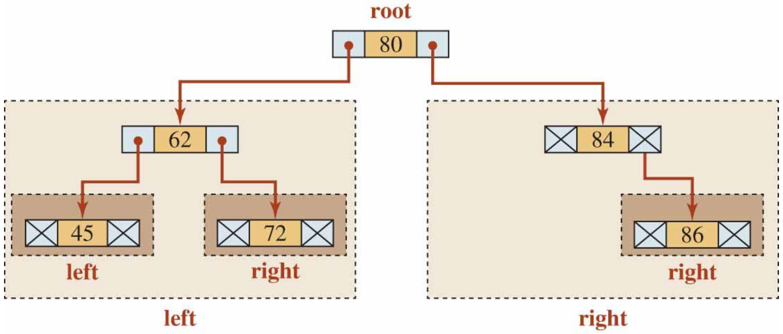
The left subtree is processed before the node and right subtree processed after the node. (in-order traversal)

This gives a clue that the printing the value of the all nodes is a recursive process as shown below.

Print items at the left subtree.	// Base case
Print the root.	// General case
Print the items in right subtree.	// General case



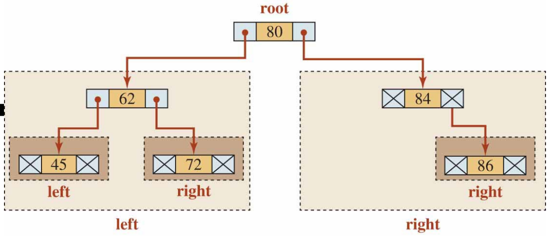
Interface File a Binary Search Tree



```
#ifndef BINARYSEARCHTREE_H
#define BINARYSEARCHTREE_H
#include <iostream>
#include <cassert>
using namespace std;

// Definition of Node struct
template <class T>
struct Node
{
        T data;
        Node <T>* left;
        Node <T>* right;
};
```

```
// Definition of BinarySearchTree class
template <class T>
class BinarySearchTree
{
        private:
                Node <T>* root;
                int count;
                Node <T>* makeNode (const T& value);
                void destroy (Node <T>* ptr); // Helper
                void insert (const T& value, Node <T>*& ptr); // Helper
                void inorder (Node <T>* ptr) const; // Helper
                void preorder (Node <T>* ptr) const; // Helper
                void postorder (Node <T>* ptr) const; // Helper
                bool search (const T& value, Node <T>* ptr) const; // Helper
        public:
                BinarySearchTree ();
                ~BinarySearchTree ();
                void insert (const T& value);
                void erase (const T& value);
                bool search (const T& value) const;
                void inorder () const;
                void preorder () const;
                void postorder () const;
                int size () const;
                bool empty () const;
};
#endif
```

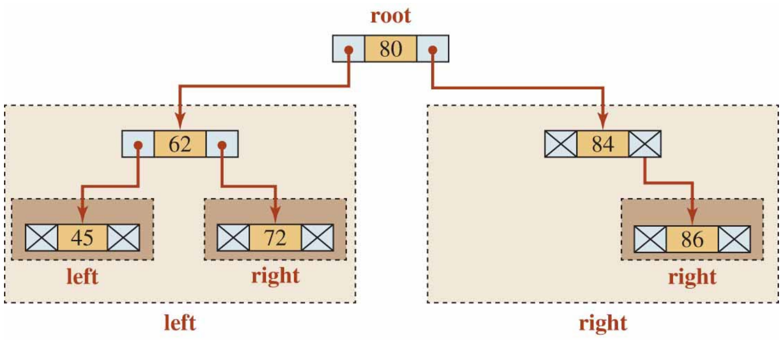
Implementation File for a Binary Search Tree Part 1



```
#ifndef BINARYSEARCHTREE_CPP
#define BINARYSEARCHTREE_CPP
#include "binarySearchTree.h"

// Constructor
template <class T>
BinarySearchTree <T> :: BinarySearchTree()
:root (0), count (0)
{
}
// Destructor
template <class T>
BinarySearchTree <T> :: ~BinarySearchTree ()
{
        destroy (root);
}
// Recursive helper member function called by the
destructor
template <class T>
void BinarySearchTree <T> :: destroy (Node <T>* ptr)
{
        if (!ptr)
        {
              return;
        }
        destroy (ptr -> left);
        destroy (ptr -> right);
        delete ptr;
}
//Size member function
template <class T>
int BinarySearchTree <T> :: size () const
{
        return count;
}
```

```
// Empty member function
template <class T>
bool BinarySearchTree <T> :: empty () const
{
        return (count == 0);
}
// Search member function
template <class T>
bool BinarySearchTree <T> :: search (const T& value) const
{
        return search (value, root);
}
// Recursive helper member function called by the search function
template <typename T>
bool BinarySearchTree <T> :: search (const T& value, Node <T>* ptr) const
{
        if (!ptr)
        {
              return false;
        }
        else if (ptr -> data == value)
        {
              return true;
        }
        else if (value < ptr -> data)
        {
              return search (value, ptr -> left);
        }
        else
        {
              return search (value, ptr -> right);
        }
}
```

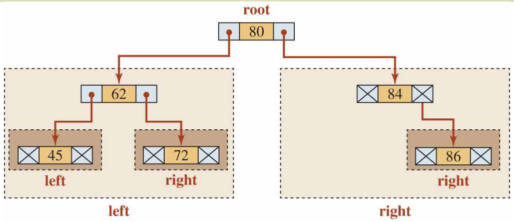
Implementation File for a Binary Search Tree Part 2



```
// Insert member function
template <class T>
void BinarySearchTree <T> :: insert (const T& value)
{
        insert (value, root);
        count++;
}
// Recursive helper member function called by the makeNode
function
template <typename T>
Node <T>* BinarySearchTree <T> :: makeNode (const T& value)
{
        Node <T>* temp = new Node <T>;
        temp -> data = value;
        temp -> left = 0;
        temp -> right = 0;
        return temp;
}
```

```
// Recursive helper member function called by insert member function
template <class T>
void BinarySearchTree <T> :: insert (const T& value, Node <T>*& ptr)
{
        if (!ptr)
        {
              ptr = makeNode (value);
              return;
        }
        else if (value < ptr -> data)
        {
              insert (value, ptr -> left);
        }
        else
        {
              insert (value, ptr -> right);
        }
}
```

Implementation File for a Binary Search Tree Part 3



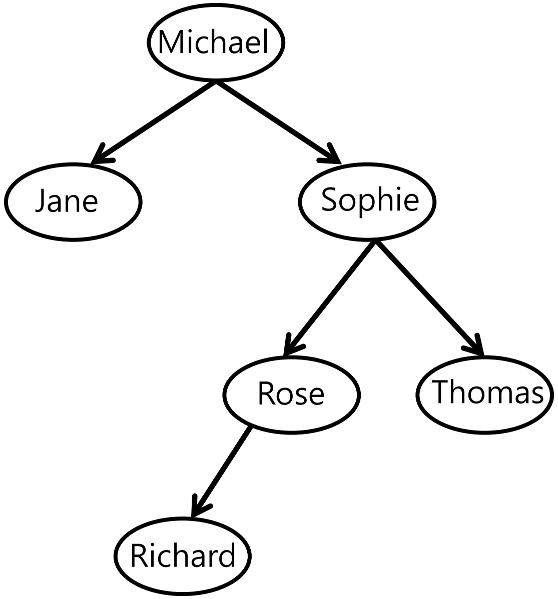
```
// Preorder traversal function
template <class T>
void BinarySearchTree <T> :: preorder () const
{
        preorder (root);
}
// Inorder traversal function
template <class T>
void BinarySearchTree <T> :: inorder () const
{
        inorder (root);
}
// Postorder traversal function
template <class T>
void BinarySearchTree <T> :: postorder () const
{
        postorder (root);
}
// Recursive helper member function called by preorder
function
template <typename T>
void BinarySearchTree <T> :: preorder (Node <T>* ptr) const
{
        if (!ptr)
        {
              return;
        }
        cout << ptr -> data << endl;
        preorder (ptr -> left);
        preorder (ptr -> right);
}
```

```
// Recursive helper member function called by the inorder
function
template <class T>
void BinarySearchTree <T> :: inorder (Node <T>* ptr) const
{
        if (!ptr)
        {
              return;
        }
        inorder (ptr -> left);
        cout << ptr -> data << endl;
        inorder (ptr -> right);
}
// Recursive helper member function called by the postorder
function
template <class T>
void BinarySearchTree <T> :: postorder (Node <T>* ptr) const
{
        if (!ptr)
        {
              return;
        }
        postorder (ptr -> left);
        postorder (ptr -> right);
        cout << ptr -> data << endl;
}
#endif
```

Application File to Test the Binary Search Tree

```
#include "binarySearchTree.cpp"

int main ( )
{
        // Instantiation of a binary search tree object
        BinarySearchTree <string> bct;
        // Inserting six nodes in the tree
        bct.insert ("Michael");
        bct.insert ("Jane");
        bct.insert ("Sophie");
        bct.insert ("Thomas");
        bct.insert ("Rose");
        bct.insert ("Richard");
        // Printing values using preorder traversal
        cout << "Using preorder traversal" << endl;
        bct.preorder ();
        cout << endl << endl;
        // Printing values using inorder traversal
        cout << "Using inorder traversal" << endl;
        bct.inorder ();
        cout << endl << endl;
        // Printing values using postorder traversal
        cout << "Using postorder traversal" << endl;
        bct.postorder ();
        cout << endl << endl;
        // Searching for a two values
        cout << "Searching: " << endl ;
        cout << "Is Sophie in the tree? " << boolalpha;
        cout << bct.search ("Sophie") << endl;
        cout << "Is Mary in the tree? " << boolalpha;
        cout << bct.search ("Mary") << endl;
        return 0;
}
```

```
Using preorder traversal
Michael
Jane
Sophie
Rose
Richard
Thomas


Using inorder traversal
Jane
Michael
Richard
Rose
Sophie
Thomas


Using postorder traversal
Jane
Richard
Rose
Thomas
Sophie
Michael


Searching:
Is Sophie in the tree? true
Is Mary in the tree? false
```



Summary

Highlights

- ❑ Singly Linked List
- ❑ Stack and Queue
- ❑ Binary Search Tree

What's Next? (Reading Assignment)

- ❑ Read Chap. 19. Standard Template Library (STL)
- ❑ Read Chap. 16. Input/Output Streams

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr