3

Data Structures in Games Development

In chapter, the following recipes will be covered:

* Using more advanced data structures
* Using linked-List to store data
* Using stack to store data
* Using queue to store data
* Using tree to store data
* Using graph to store data
* Using STL : List to store data
* Using STL : Map to store data
* Using Hash-Tables to store data

# Introduction

Data structures are used in video games industry to organize code into more cleaner and manageable chunks. An average video game will have about 20,000 lines of code at the minimum. If we do not use an effective storage system and structure to manage that code, it will become very difficult to debug code. Also we may end up writing the same code multiple times.

Data structures are also very useful for searching elements. If we have a large data set. Let us consider that we are making a MMO. From the thousands of players online playing the game, we need to isolate a player who has the maximum points on a certain day. If we have not organized the user data into some meaningful data structure, this might take a long time. On the other hand, using a suitable data structure can help us achieve this within seconds.

# Using more advanced data structures

1. In this recipe we will see how to use advanced data structures. The main task of a programmer is to choose the correct data structure based on the need so that the time taken to achieve the task is minimized. Sometimes the choice of a correct data structure is more important than selecting a suitable algorithm to solve a problem.

## Getting ready

To step through this recipe, you will need a machine running Windows. No other prerequisites are required. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe we will see how easy it is to use advanced data structures and why we should use it. If we organise data into suitable structures, it becomes faster to access data and easier to apply complex algorithms on them.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp, LinkedList.h/cpp and HashTables.h/cpp
5. Add the following lines of code.

**Souce.cpp**

#include "HashTable.h"

#include <conio.h>

int main()

{

// Create 26 Items to store in the Hash Table.

Item \* A = new Item{ "Enemy1", NULL };

Item \* B = new Item{ "Enemy2", NULL };

Item \* C = new Item{ "Enemy3", NULL };

Item \* D = new Item{ "Enemy4", NULL };

Item \* E = new Item{ "Enemy5", NULL };

Item \* F = new Item{ "Enemy6", NULL };

Item \* G = new Item{ "Enemy7", NULL };

Item \* H = new Item{ "Enemy8", NULL };

Item \* I = new Item{ "Enemy9", NULL };

Item \* J = new Item{ "Enemy10", NULL };

Item \* K = new Item{ "Enemy11", NULL };

Item \* L = new Item{ "Enemy12", NULL };

Item \* M = new Item{ "Enemy13", NULL };

Item \* N = new Item{ "Enemy14", NULL };

Item \* O = new Item{ "Enemy15", NULL };

Item \* P = new Item{ "Enemy16", NULL };

Item \* Q = new Item{ "Enemy17", NULL };

Item \* R = new Item{ "Enemy18", NULL };

Item \* S = new Item{ "Enemy19", NULL };

Item \* T = new Item{ "Enemy20", NULL };

Item \* U = new Item{ "Enemy21", NULL };

Item \* V = new Item{ "Enemy22", NULL };

Item \* W = new Item{ "Enemy23", NULL };

Item \* X = new Item{ "Enemy24", NULL };

Item \* Y = new Item{ "Enemy25", NULL };

Item \* Z = new Item{ "Enemy26", NULL };

// Create a Hash Table of 13 Linked List elements.

HashTable table;

// Add 3 Items to Hash Table.

table.insertItem(A);

table.insertItem(B);

table.insertItem(C);

table.printTable();

// Remove one item from Hash Table.

table.removeItem("Enemy3");

table.printTable();

// Add 23 items to Hash Table.

table.insertItem(D);

table.insertItem(E);

table.insertItem(F);

table.insertItem(G);

table.insertItem(H);

table.insertItem(I);

table.insertItem(J);

table.insertItem(K);

table.insertItem(L);

table.insertItem(M);

table.insertItem(N);

table.insertItem(O);

table.insertItem(P);

table.insertItem(Q);

table.insertItem(R);

table.insertItem(S);

table.insertItem(T);

table.insertItem(U);

table.insertItem(V);

table.insertItem(W);

table.insertItem(X);

table.insertItem(Y);

table.insertItem(Z);

table.printTable();

// Look up an item in the hash table

Item \* result = table.getItemByKey("Enemy4");

if (result!=nullptr)

cout << endl<<"The next key is "<<result->next->key << endl;

\_getch();

return 0;

}

**LinkedList.h**

#ifndef LinkedList\_h

#define LinkedList\_h

#include <iostream>

#include <string>

using namespace std;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// List items are keys with pointers to the next item.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

struct Item

{

string key;

Item \* next;

};

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Linked lists store a variable number of items.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class LinkedList

{

private:

// Head is a reference to a list of data nodes.

Item \* head;

// Length is the number of data nodes.

int length;

public:

// Constructs the empty linked list object.

// Creates the head node and sets length to zero.

LinkedList();

// Inserts an item at the end of the list.

void insertItem(Item \* newItem);

// Removes an item from the list by item key.

// Returns true if the operation is successful.

bool removeItem(string itemKey);

// Searches for an item by its key.

// Returns a reference to first match.

// Returns a NULL pointer if no match is found.

Item \* getItem(string itemKey);

// Displays list contents to the console window.

void printList();

// Returns the length of the list.

int getLength();

// De-allocates list memory when the program terminates.

~LinkedList();

};

#endif

**LinkedList.cpp**

#include "LinkedList.h"

// Constructs the empty linked list object.

// Creates the head node and sets length to zero.

LinkedList::LinkedList()

{

head = new Item;

head->next = NULL;

length = 0;

}

// Inserts an item at the end of the list.

void LinkedList::insertItem(Item \* newItem)

{

if (!head->next)

{

head->next = newItem;

length++;

return;

}

Item \* p = head;

Item \* q = head;

while (q)

{

p = q;

q = p->next;

}

p->next = newItem;

newItem->next = NULL;

length++;

}

// Removes an item from the list by item key.

// Returns true if the operation is successful.

bool LinkedList::removeItem(string itemKey)

{

if (!head->next) return false;

Item \* p = head;

Item \* q = head;

while (q)

{

if (q->key == itemKey)

{

p->next = q->next;

delete q;

length--;

return true;

}

p = q;

q = p->next;

}

return false;

}

// Searches for an item by its key.

// Returns a reference to first match.

// Returns a NULL pointer if no match is found.

Item \* LinkedList::getItem(string itemKey)

{

Item \* p = head;

Item \* q = head;

while (q)

{

p = q;

if ((p != head) && (p->key == itemKey))

return p;

q = p->next;

}

return NULL;

}

// Displays list contents to the console window.

void LinkedList::printList()

{

if (length == 0)

{

cout << "\n{ }\n";

return;

}

Item \* p = head;

Item \* q = head;

cout << "\n{ ";

while (q)

{

p = q;

if (p != head)

{

cout << p->key;

if (p->next) cout << ", ";

else cout << " ";

}

q = p->next;

}

cout << "}\n";

}

// Returns the length of the list.

int LinkedList::getLength()

{

return length;

}

// De-allocates list memory when the program terminates.

LinkedList::~LinkedList()

{

Item \* p = head;

Item \* q = head;

while (q)

{

p = q;

q = p->next;

if (q) delete p;

}

}

**HashTables.cpp**

#include "HashTable.h"

// Constructs the empty Hash Table object.

// Array length is set to 13 by default.

HashTable::HashTable(int tableLength)

{

if (tableLength <= 0) tableLength = 13;

array = new LinkedList[tableLength];

length = tableLength;

}

// Returns an array location for a given item key.

int HashTable::hash(string itemKey)

{

int value = 0;

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

value += itemKey[i];

return (value \* itemKey.length()) % length;

}

// Adds an item to the Hash Table.

void HashTable::insertItem(Item \* newItem)

{

int index = hash(newItem->key);

array[index].insertItem(newItem);

}

// Deletes an Item by key from the Hash Table.

// Returns true if the operation is successful.

bool HashTable::removeItem(string itemKey)

{

int index = hash(itemKey);

return array[index].removeItem(itemKey);

}

// Returns an item from the Hash Table by key.

// If the item isn't found, a null pointer is returned.

Item \* HashTable::getItemByKey(string itemKey)

{

int index = hash(itemKey);

return array[index].getItem(itemKey);

}

// Display the contents of the Hash Table to console window.

void HashTable::printTable()

{

cout << "\n\nHash Table:\n";

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

{

cout << "Bucket " << i + 1 << ": ";

array[i].printList();

}

}

// Returns the number of locations in the Hash Table.

int HashTable::getLength()

{

return length;

}

// Returns the number of Items in the Hash Table.

int HashTable::getNumberOfItems()

{

int itemCount = 0;

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

{

itemCount += array[i].getLength();

}

return itemCount;

}

// De-allocates all memory used for the Hash Table.

HashTable::~HashTable()

{

delete[] array;

}

**HashTables.h**

#ifndef HashTable\_h

#define HashTable\_h

#include "LinkedList.h"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Hash Table objects store a fixed number of Linked Lists.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class HashTable

{

private:

// Array is a reference to an array of Linked Lists.

LinkedList \* array;

// Length is the size of the Hash Table array.

int length;

// Returns an array location for a given item key.

int hash(string itemKey);

public:

// Constructs the empty Hash Table object.

// Array length is set to 13 by default.

HashTable(int tableLength = 13);

// Adds an item to the Hash Table.

void insertItem(Item \* newItem);

// Deletes an Item by key from the Hash Table.

// Returns true if the operation is successful.

bool removeItem(string itemKey);

// Returns an item from the Hash Table by key.

// If the item isn't found, a null pointer is returned.

Item \* getItemByKey(string itemKey);

// Display the contents of the Hash Table to console window.

void printTable();

// Returns the number of locations in the Hash Table.

int getLength();

// Returns the number of Items in the Hash Table.

int getNumberOfItems();

// De-allocates all memory used for the Hash Table.

~HashTable();

};

#endif

## How it works...

We have created this class to store different enemies using a hash table and then search for a particular enemy from the hash table using a key. The hash table in turn is created using a linked list.

In the LINKEDLIST file, we have defined a struct to store a key and a pointer to the next value in the hash table. The main class contains a pointer reference of the struct called ITEM. Apart from that the class contains length of the number of data number and member functions to insert an item, remove an item, finding an element, displaying the entire list and to find the length of the list.

In the HASHTABLE file, a hashtable is created using a linked list. A reference of linked list is created along with the length of the hashtable array and a private function to return an array location of a particular item in the hashtable array. Apart from that the hashtable has similar functionalities like the linked list such as inserting an item, removing an item and displaying the hash table.

From the driver program, an object of the struct is created to initialize the items to be pushed into the hashtable. Then an object of hashtable is created and the items are pushed to the table and displayed. An item is also deleted from the table. Finally a particular item called Enemy4 is searched and the next key is displayed.

# Using linked list to store data

In this recipe we will see how we can use linked list to store and organise data. The main advantage of a linked list in the games industry is that it is a dynamic data structure. This means we can assign memory to this data structure at run time. In games, most of the things are created, destroyed and updated at run time. So using linked list makes it very suitable. Linked lists can also be used to create linear data structures such as stack and queues which are equally important in game programming.

## Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe we will see how easy it is to use linked lists. Linked lists are a great way to store data and is used as a base mechanics for other data structures.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

#include <iostream>

#include <conio.h>

using namespace std;

typedef struct LinkedList {

int LevelNumber;

LinkedList \* next;

} LinkedList;

int main() {

LinkedList \* head = NULL;

int i;

for (i = 1; i <= 10; i++) {

LinkedList \* currentNode = new LinkedList;

currentNode->LevelNumber = i;

currentNode->next = head;

head = currentNode;

}

while (head) {

cout << head->LevelNumber << " ";

head = head->next;

}

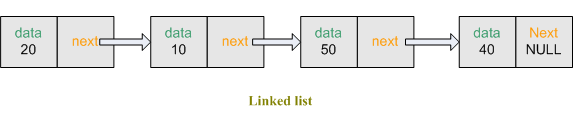
\_getch();

return 0;

## }

## How it works...

1. A linked list is used to create a data structure which stores data and a field which contains the address of the next node. A linked list is made up of nodes.



Insert Image B04929\_03\_01.png

In our example we have created a linked list using struct and used an iteration to populate the linked list. The main concept of a linked list as explained above is that it contains data of some kind and contains address information of the next node. In our example we have created a linked list to store the current level number and the address of the next level to be loaded. This kind of structure is really important to store the levels we want to load. Just by traversing the linked list, we can load the levels in the correct order. Even checkpoints in the game can be programmed in similar manner.

# Using stacks to store data

Stacks are an example of linear data structure in C++. In this type of data structure, the order in which the data is entered into the data structure is very important. The last data to be entered is the first data to be deleted. That is why, this is sometimes also referred to as the (LIFO) Last In First Out data structure. The process of entering data into a stack is called push and the process of deleting data is called pop. Sometimes we just want to print the value at the top of the stack, without deleting or popping. The stack is used in variety of areas in the games industry but especially with creating an UI system for a game.

## Getting ready

1. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe, we will find out how easy it is to use the stack data structure. Stack is one of the easiest data structures to implement and used in multiple areas.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

#include <iostream>

#include <conio.h>

#include <string>

using namespace std;

class Stack

{

private:

string UI\_Elements[10];

int top;

public:

Stack()

{

top = -1;

}

void Push(string element)

{

if (top >= 10)

{

cout << "Some error occured";

}

UI\_Elements[++top] = element;

}

string Pop()

{

if (top == -1)

{

cout << "Some error occured";

}

return UI\_Elements[top--];

}

string Top()

{

return UI\_Elements[top];

}

int Size()

{

return top + 1;

}

bool isEmpty()

{

return (top == -1) ? true : false;

}

};

int main()

{

Stack \_stack;

if (\_stack.isEmpty())

{

cout << "Stack is empty" << endl;

}

// Push elements

\_stack.Push("UI\_Element1");

\_stack.Push("UI\_Element2");

// Size of stack

cout << "Size of stack = " << \_stack.Size() << endl;

// Top element

cout << \_stack.Top() << endl;

// Pop element

cout << \_stack.Pop() << endl;

// Top element

cout << \_stack.Top() << endl;

\_getch();

return 0;

}

## How it works...

In this example, we have used the STACK data structure to push the various UI elements into the stack. The STACK itself is created with the help of an array. . While pushing an element, we need to check whether the stack is empty or it has already some elements present in the stack. While popping elements, we need to delete the element which is at the top of the stack and change the pointer address accordingly. While printing the UI elements of the stack, we traverse the entire stack and display them from the top. Let us consider a game with the following levels: Main Menu, Chapter Select, Level Select, and Play Game. Now when we want to quit the game, we want the user to select the levels in the reverse order. So the first level should be Play Game (Pause State) followed by Level Select, Chapter Select and finally Main Menu. This can be easily achieved by stack as explained in the example above.

# Using queue to store data

Queue is example of a dynamic data structure. This means the size of the queue can be changed at runtime. This is a huge advantage when it comes to programming in games. Queues are enqeued/inserted from the rear of the data structure and dequeued/deleted/pushed out from the front of the data structure. This makes it a FIFO (First in First Out) data structure. Imagine in a game, we have an inventory but we want to let the player currently use the first item he has picked up unless he manually changes to a different item. This can be easily achieved by a queue. If we want to design such that the current item switches to the most powerful item in the inventory, we can use a priority queue for that purpose.

## Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

In this recipe we will implement the queue data structure using linked list. It is very easy to implement a queue and it is a very robust data structure to use.

1. Open Visual Studio
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp
5. Add the following lines of code

**Source.cpp**

#include <iostream>

#include <queue>

#include <string>

#include <conio.h>

using namespace std;

int main()

{

queue <string> gunInventory;

gunInventory.push("AK-47");

gunInventory.push("BullPup");

gunInventory.push("Carbine");

cout << "This is your weapons inventory" << endl << endl;

cout << "The first gun that you are using is "

<< gunInventory.front() << endl << endl;

gunInventory.pop();

cout << "There are currently " << gunInventory.size()

<< " more guns in your inventory. " << endl << endl

<< "The next gun in the inventory is "

<< gunInventory.front() << "." << endl << endl

<< gunInventory.back() << " is the last gun in the inventory."

<< endl;

\_getch();

return 0;

## }

## How it works...

We have used STL: queue to create the queue structure or rather use the queue structure. Queue structure as we know is important when we need to use the FIFO data structure. Like in a First Person Shooter, we may want the user to use the first gun he picks up and the remaining guns to be put in the inventory. This is an ideal case for queue as explained in the example. The front of the queue structure holds the first gun he picked up or the most current gun and the remaining guns are stored in the inventory in the order in which they are picked up. Sometimes we do want in our game that if we pick up a gun more powerful than the one we are using, it should automatically swap to that. In such a case, we can use a more specialized form of the queue called a priority queue where we need to use the same queue structure but just specify on what parameters the queue is to get sorted.

# Using tree to store data

A tree is an example of non-linear data structure, unlike array and linked list which are linear. A tree is often used in games which require hierarchy. Imagine a car with many parts and all the parts are functional, upgradable and can be interacted with. In this case, we will create the entire class of the car using a tree data structure. A tree uses a parent child relationship to traverse between all the nodes.

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

In this recipe we will be implementing a binary tree. There are many variations of the binary tree. We will be creating the most basic binary tree. It is very easy to add new logic to the binary tree to implement a balanced binary or AVL tree and so on.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called CTree.cpp
5. Add the following lines of code.

// Initialise the node with a value and pointers

// to left child

// and right child

struct node

{

string data\_value;

node \*left;

node \*right;

};

class Binary\_Tree

{

public:

Binary\_Tree();

~Binary\_Tree();

void insert(string key);

node \*search(string key);

void destroy\_tree();

private:

void destroy\_tree(node \*leaf);

void insert(string key, node \*leaf);

node \*search(string key, node \*leaf);

node \*root;

};

Binary\_Tree::Binary\_Tree()

{

root = NULL;

}

Binary\_Tree::~Binary\_Tree()

{

destroy\_tree();

}

void Binary\_Tree::destroy\_tree(node \*leaf)

{

if (leaf != NULL)

{

destroy\_tree(leaf->left);

destroy\_tree(leaf->right);

delete leaf;

}

}

void Binary\_Tree::insert(string key, node \*leaf)

{

if (key< leaf->key\_value)

{

if (leaf->left != NULL)

insert(key, leaf->left);

else

{

leaf->left = new node;

leaf->left->key\_value = key;

leaf->left->left = NULL;

leaf->left->right = NULL;

}

}

else if (key >= leaf->key\_value)

{

if (leaf->right != NULL)

insert(key, leaf->right);

else

{

leaf->right = new node;

leaf->right->key\_value = key;

leaf->right->left = NULL;

leaf->right->right = NULL;

}

}

}

node \*Binary\_Tree::search(string key, node \*leaf)

{

if (leaf != NULL)

{

if (key == leaf->key\_value)

return leaf;

if (key<leaf->key\_value)

return search(key, leaf->left);

else

return search(key, leaf->right);

}

else return NULL;

}

void Binary\_Tree::insert(string key)

{

if (root != NULL)

insert(key, root);

else

{

root = new node;

root->key\_value = key;

root->left = NULL;

root->right = NULL;

}

}

node \*Binary\_Tree::search(string key)

{

return search(key, root);

}

void Binary\_Tree::destroy\_tree()

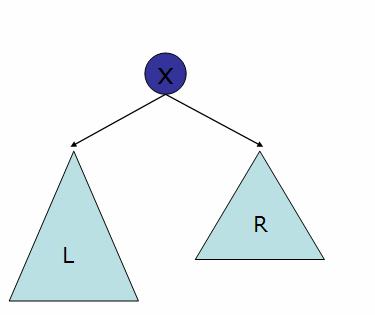
{

destroy\_tree(root);

}

## How it works...

We use a struct to store the value and a pointer to the left child and the right child. There is no particular rule as to which elements should be your left child and which elements should be part of the right child. We can decide if we want so that all elements lower than the root element to be on the left and all elements greater that the root are to the right.



Insert Image B04929\_03\_02.png

Both insertion and deletion in the tree data structure is done in a recursive way. To insert elements, we traverse the tree and check if it is empty. If it is empty, we create a new node and add all the corresponding nodes recursively by checking if the new node’s value is greater than or less than the root node. Searching for an element is also done in a similar way. If the element to be searched has a value lower than the root node, we can ignore the entire right hand section of the tree as we can see in our Search function and keep searching recursively. This reduces the search space considerably and optimizes our algorithm. This means searching for an item at runtime will be faster. Let us say we are creating a game where we need to implement a procedural terrain. After the scene is loaded, we can use a binary tree to divide the entire level into sections based on which appear left or right. If this information is correctly stored in the tree, then the game camera can use this information to decide which section is rendered and which is not. This also created a great level of culling optimization. If the parent does not get rendered, we can neglect checking the remaining of tree for rendering.

# Using graph to store data

In this recipe we will see how easy it is to store data using the graph data structure. The graph data structure is extremely useful if we have to create a system like facebook to sell and share our game with friends and friends of friends. A graph can be implemented in few ways. The most used method is by using edges and nodes.

## Getting ready

To step through this recipe, you will need a machine running Windows. No other prerequisites are required. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe, we will see how we can implement graphs. Graphs are a very good data structure for interconnecting various states and data together with edge conditions. Any social networking algorithm uses the graph data structure in some way or the other.

1. Open Visual Studio.
   1. Create a new C++ project
   2. Select a win32 console application
   3. Add the following files: CGraph.h/cpp
   4. Add the following lines of code.

**CGraph.h**

#include <iostream>

#include <vector>

#include <map>

#include <string>

using namespace std;

struct vertex

{

typedef pair<int, vertex\*> ve;

vector<ve> adj; //cost of edge, destination vertex

string name;

vertex(string s)

{

name = s;

}

};

class graph

{

public:

typedef map<string, vertex \*> vmap;

vmap work;

void addvertex(const string&);

void addedge(const string& from, const string& to, double cost);

};

**CGraph.cpp**

void graph::addvertex(const string &name)

{

vmap::iterator itr = work.begin();

itr = work.find(name);

if (itr == work.end())

{

vertex \*v;

v = new vertex(name);

work[name] = v;

return;

}

cout << "\nVertex already exists!";

}

void graph::addedge(const string& from, const string& to, double cost)

{

vertex \*f = (work.find(from)->second);

vertex \*t = (work.find(to)->second);

pair<int, vertex \*> edge = make\_pair(cost, t);

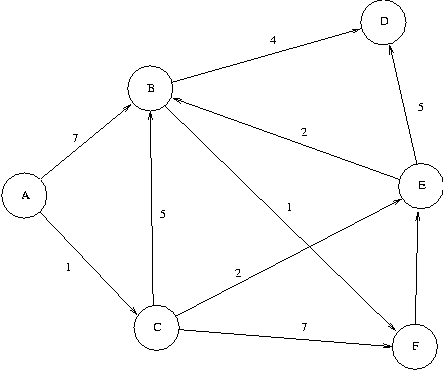
f->adj.push\_back(edge);

}

## How it works...

A graph is comprised of edges and nodes. So the first thing to do while implementing a graph data structure is to create a struct to store the node and vertex information. The diagram below has 6 nodes and 7 edges. To implement a graph we need to understand the cost of each edge to go from one node to another. These are called adjacency costs. To insert a node, we create a node. To add an edge to the node, we need to supply the information about the two nodes that needs to be connected and the cost of the edge.

After we get that information we create a pair using the cost of the edge and one of the nodes and push that edge information to the other node.



Insert Image B04929\_03\_03.png

# Using STL List to store data

STL is a standard template library which contains a lot of implementation of the basic data structures, which means we can directly use them for our purpose. The list is internally implemented as doubly linked lists, which means insertion and deletion can happen at both ends.

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

In this recipe we will see how we can easily use the inbuilt template library provided to us by C++ to create complex data structures. After the complex data structure is created, we can easily use it to store data and access it.

1. Open Visual Studio.
2. Create a new C++ project
3. Add a source file called Source.cpp
4. Add the following lines of code.
5. **Source.cpp**

#include <iostream>

#include <list>

#include <conio.h>

using namespace std;

int main()

{

list<int> possible\_paths;

possible\_paths.push\_back(1);

possible\_paths.push\_back(1);

possible\_paths.push\_back(8);

possible\_paths.push\_back(9);

possible\_paths.push\_back(7);

possible\_paths.push\_back(8);

possible\_paths.push\_back(2);

possible\_paths.push\_back(3);

possible\_paths.push\_back(3);

possible\_paths.sort();

possible\_paths.unique();

for (list<int>::iterator list\_iter = possible\_paths.begin();

list\_iter != possible\_paths.end(); list\_iter++)

{

cout << \*list\_iter << endl;

}

\_getch();

return 0;

}

## How it works...

We have the used the list to push back values of possible path costs of a certain AI player to reach a destination. Because we have used STL List, it comes in built with few functions which we can apply on the container. We use the sort function to sort the list in ascending order. We also have unique function to delete all duplicate values from the list. After sorting the list, we have the least path cost and accordingly we can apply that path to the AI player. Although the code size is reduced immensely and it is much easier to write, we should use STL with caution as we are never sure about the algorithm behind the inbuilt functions. For example the sort function, most likely uses QuickSort but we don’t know.

# Using STL Map to store data

Map is one of the associative containers of STL that store elements formed by a combination of a key value and a mapped value, following a specific order.Map is a part of the Standard Template Library provided to us by C++.

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

1. In this recipe we will see how we can easily use the inbuilt template library provided to us by C++ to create complex data structures. After the complex data structure is created, we can easily use it to store data and access it.
2. Open Visual Studio.
3. Create a new C++ project
4. Add a source file called Source.cpp
5. Add the following lines of code.
6. **Source.cpp**

#include <iostream>

#include <map>

#include <conio.h>

using namespace std;

int main()

{

map <string, int> score\_list;

score\_list["John"] = 242;

score\_list["Tim"] = 768;

score\_list["Sam"] = 34;

if (score\_list.find("Samuel") == score\_list.end())

{

cout << "Samuel is not in the map!" << endl;

}

cout << score\_list.begin()->second << endl;

\_getch();

return 0;

}

## How it works...

We have used the STL map to create a key and value pair to store the names of the players playing our game along with their high score. We can use any data type in a map. In our example we have used a string and an int. After creating the data structure, it is very easy to find whether a player exists in the data base and we can also sort the map and display the score associated with the player. The second field gives us the values whereas the first field gives us the key.

# Using STL HashTables to store data

The biggest difference between a map and a hash table is that while the map data structure is ordered, a hash table is unordered. Both use the same principle of key, value pairs. .The worst case search complexity for unordered\_map is O(N) as it is not ordered like a map which has a complexity of O( log N).

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

1. In this recipe we will see how we can easily use the inbuilt template library provided to us by C++ to create complex data structures. After the complex data structure is created, we can easily use it to store data and access it.
2. Open Visual Studio.
3. Create a new C++ project
4. Add a source file called Source.cpp
5. Add the following lines of code.
6. **Source.cpp**

#include <unordered\_map>

#include <string>

#include <iostream>

#include <conio.h>

using namespace std;

int main()

{

unordered\_map<string, string> hashtable;

hashtable.emplace("Alexander", "23ms");

hashtable.emplace("Christopher", "21ms");

hashtable.emplace("Steve", "55ms");

hashtable.emplace("Amy", "17ms");

hashtable.emplace("Declan", "999ms");

cout << "Ping time in milliseconds: " << hashtable["Amy"] << endl<<endl;

cout << "----------------------------------" << endl << endl;

hashtable.insert(make\_pair("Fawad", "67ms"));

cout << endl<<"Ping time of all player is the server" << endl;

cout << "------------------------------------" << endl << endl;

for (auto &itr : hashtable)

{

cout << itr.first << ": " << itr.second << endl;

}

\_getch();

return 0;

}

## How it works...

The program calculates the ping time of all players who are currently playing our game on the server. We create a hash table and store all their names and ping times using the emplace keyword. We can also insert a new player later on with his ping time by using the make\_pair keyword. After the hash table is created, we can easily display the ping time of a particular player or the ping time of all players in the server. We use an iterator to iterate through the hash table. The first parameter gives us the key and the second parameter gives us the value.