**XII Computer Science**

**Final Project :**

**Graph Coloring**

**Name :** Varad Mahashabde

**Class :** XII **Sec :** A

**Roll no.:** 29

**Acknowledgement**

I would like to sincerely and profusely thank my Computer Science teacher **Mr. Pankaj Kumar** for his helpful guidance and support in completing this project.

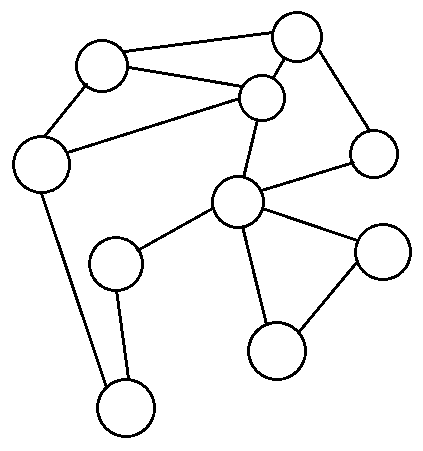
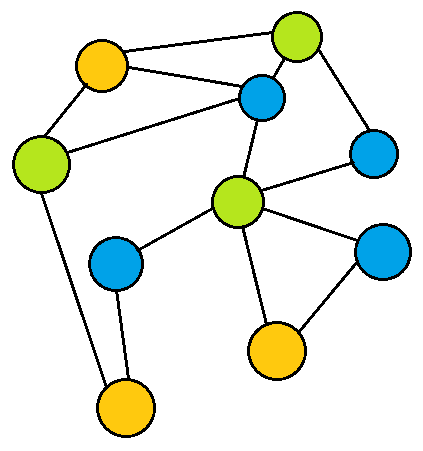
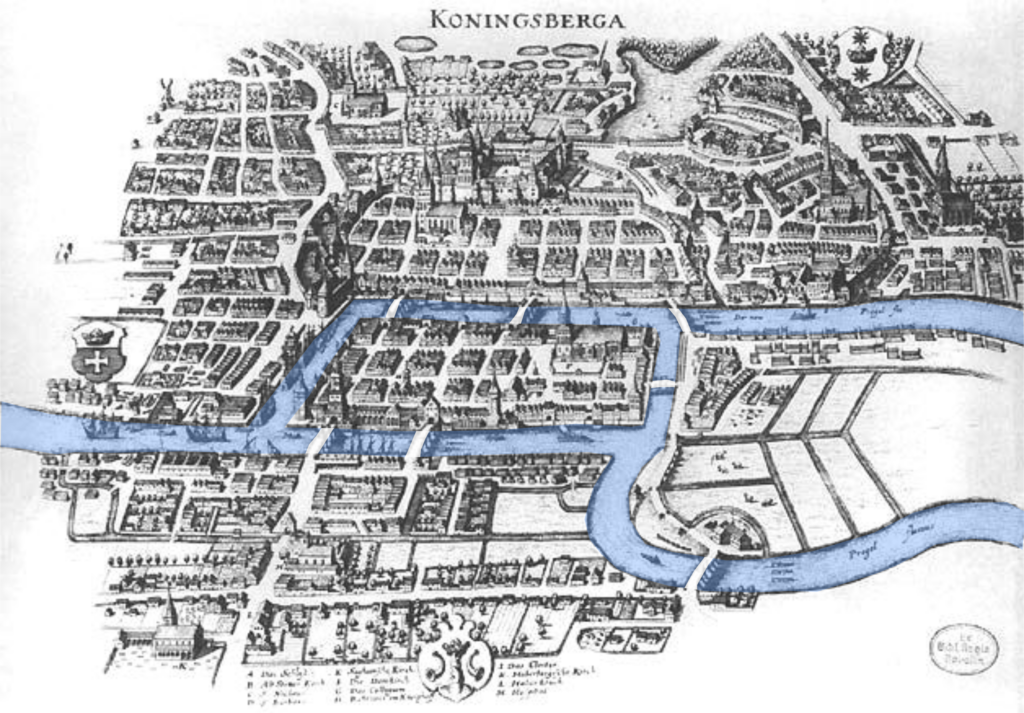
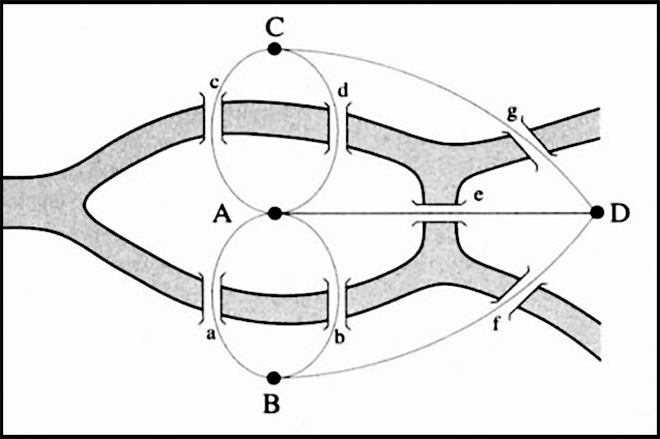
I would also like to extend my gratitude to the principal of our school **Mrs. Malini Narayanan** for providing me the facilities and resources to develop our curiosity and enable me to conduct this investigatory project.

**Certificate**

This document hereby certifies that **Varad Mahashabde**, a student of class XII has successfully completed the research on the investigatory project on the subject of **Graph Coloring** and has satisfactorily performed the required practicals under the guidance of **Mr. Pankaj Kumar** during the academic year **2019-2020** in partial fulfillment of the Computer Science Practical Examination of the AISSCE conducted under the aegis of CBSE, New Delhi.

Name of the Computer Science Teacher  
**Mr. Pankaj Kumar**

Signature  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



E  
U  
L  
E  
R

Graph Coloring

**Introduction**Graph theory can be summarized as the study of ‘graphs’, structures consisting of ‘vertices’ or nodes, which are interconnected to each other represented by ‘edges’. Graphs may ‘undirected’ or they may ‘directed’ i.e. the edges may simply connect two vertices or they may ‘go’ from vertex to another like arrows. Graphs find application in a diverse and unexpectedly large range of fields, as obvious as networks such as neural nets, communication networks, power grids, roads, and human relationships, to as obscure as to study language syntax and semantics, representing a molecule, or predicting protein interactions.  
The field was first conceived by the legendary mathematician Leonhard Euler in 1736, to resolve the problem known as Seven Bridges of Königsberg.

*The 18th century town Könisberg in East Prussia (now Kaliningrad, Russia) was built over the river Pregel. The town, splits into two banks and two islands, was connected by seven bridges. During summer, the people of the town liked to take evening strolls across the bridges. But they found out that in order to cross all the bridges at least once, they have to cross at least one bridge once. They wondered, was it actually possible to cross all the seven bridges without backtracking?*

* *Glen Vecchione, Challenging Math Puzzles, paraphrased*

Euler solved it by converting the landmasses into vertices, and the bridges into edges. He noticed that all vertices had an odd number of edges, however only the start and endpoints of a journey may have an odd no. of vertices, making a ‘one-stroke’ journey of the town impossible. He also made the observation that a graph may only have an even number of oddly-connected vertices.

**Graph Coloring**A graph coloring is an assignment of ‘colors’ to the vertices of the graph. Mostly this means a proper coloring i.e. a coloring such that each pair of connected vertices does not have a common color. If colors can be used to color a graph, it is , and the minimum for which this is possible is known as the **chromatic** number of the graph, which is .  
The term ‘coloring’ arose due to efforts of translating problems regarding the coloring of contiguous regions on maps into graphs. This lead to the famous Four Color Theorem, stating that for any plane partitioned into contiguous regions, all regions can colored using only 4 colors such that any two regions sharing an boundary line are colored differently.  
The ‘colors’ used to color a graph can be used to represent other properties as well, such as scheduling jobs on the processor which require common resources, assigning landing times to airplanes arriving in the same time intervals, solving Sudoku puzzles (as all Sudoku grids are 9-numerable/colorable), or as is explored in this project, the scheduling of exams for various courses and subjects which may share students.

**Code**

* **list.h**

#if !defined(VARAD\_LIST\_H\_INCLUDED)  
 #define VARAD\_LIST\_H\_INCLUDED  
   
 namespace list\_h {  
 const int check\_num = 0x1234abcd;  
   
 // Make type const if it is a pointer  
 template <class T>  
 struct AddConstToType {  
 typedef T type;  
 };  
 template <class T>  
 struct AddConstToType<T\*> {  
 typedef const T\* type;  
 };  
   
 template <class T>  
 class BinaryFunctor {  
 public :  
 virtual bool function(const typename AddConstToType<T>::type& a, const typename AddConstToType<T>::type& b) const = 0;  
 inline bool operator()(const typename AddConstToType<T>::type& a, const typename AddConstToType<T>::type& b) const {  
 return function(a, b);  
 }  
 };  
 }  
   
   
 template <class T>  
 class List {  
 protected :  
 class Node {  
 public :  
 T data;  
 Node\* next;  
 Node\* previous;  
 const int check;  
   
 Node() : check(list\_h::check\_num) {}  
 // Weird cast of const T new\_data to T so that it can be copied to data  
 Node(const typename list\_h::AddConstToType<T>::type& new\_data, Node\* prev, Node\* nxt = nullptr)  
 : check(list\_h::check\_num), data((T)new\_data), previous(prev), next(nxt) {}  
 Node(Node\* prev) : check(list\_h::check\_num), previous(prev), next(nullptr) {}  
 Node(const Node& old\_node, Node\* prev = nullptr) : Node(old\_node.data, prev) {}  
 ~Node() {  
 data.~T();  
 next = previous = nullptr;  
 }  
   
 bool isOk() const {  
 return check == list\_h::check\_num;  
 }  
 Node\* makeNext(const typename list\_h::AddConstToType<T>::type& new\_data) {  
 Node\* new\_next = new Node(new\_data, this);  
 if (new\_next != nullptr) next = new\_next;  
 return new\_next;  
 }  
 Node\* makeNext() {  
 Node\* new\_next = new Node(this);  
 if (new\_next != nullptr) next = new\_next;  
 return new\_next;  
 }  
 };  
   
 Node\* first;  
 Node\* last;  
 unsigned int length;  
   
 Node\* getNode(int index) const {  
 if (length == 0)  
 index = 0;  
 else if (index < 0)  
 index = index % length + length;  
 else  
 index = index % length;  
   
 Node\* cur;  
 if (2 \* index < length) {  
 cur = first;  
 for (int i = 1; i <=index; i++) {  
 cur = cur->next;  
 }  
 } else {  
 cur = last;  
 for (int i = length - 2; i >= index; i--) {  
 cur = cur->previous;  
 }  
 }  
 return cur;  
 }  
   
 void reCalculateLength() {  
 length = 0;  
 for(Node\* read\_head = first; read\_head != nullptr; read\_head = read\_head->next, ++length);  
 }  
   
 public:   
 class Iterator {  
 friend class List;  
   
 const List& parent\_list;  
 Node\* current;  
   
 public :  
 int position;  
  
 private :  
 // Custom Constructor for exact copying  
 Iterator(const List& pops, Node\* cur, int pos) : parent\_list(pops), current(cur), position(pos) {}  
   
 public :  
 // Basic functions  
 void first() {  
 current = parent\_list.first;  
 position = 0;  
 }  
 void last() {  
 current = parent\_list.last;  
 position = parent\_list.length - 1;  
 }   
 Iterator(const List& papa, bool start\_at\_first = true) : parent\_list(papa) {  
 if (start\_at\_first)  
 first();  
 else   
 last();  
 };  
 // Constructors  
 Iterator(const Iterator& i) : parent\_list(i.parent\_list), current(i.current), position(i.position) {}  
 Iterator& operator=(const Iterator& i) {  
 if (&parent\_list == &(i.parent\_list)) {  
 current = i.current;  
 position = i.position;  
 }/\* else {  
 ~Iterator();  
 new(this) Iterator(i);  
 }\*/  
 return \*this;  
 }  
   
 bool hasEnded() const {  
 return current == nullptr or position >= parent\_list.length or position < 0;  
 }  
 bool operator==(const Node\* const node) const {  
 return current == node;  
 }  
 bool operator==(const Iterator& i) const {  
 return current == i.current;  
 }  
 bool operator!=(const Node\* const node) const {  
 return not current == node;  
 }  
 bool operator!=(const Iterator& i) const {  
 return not current == i.current;  
 }  
 Iterator& operator++() {  
 if (current != nullptr) current = current->next;  
 ++position;  
 return \*this;  
 }  
 Iterator& operator++(int) {  
 return ++(\*this);  
 }  
 inline Iterator& operator--() {  
 if (current != nullptr) current = current->previous;  
 --position;  
 return \*this;  
 }  
 inline Iterator& operator--(int) {  
 return --(\*this);  
 }  
 Iterator operator+(int index) {  
 Node\* new\_current = current;  
 if (index < 0)  
 for(int i = index; i < 0; i++) {  
 new\_current = new\_current->previous;  
 if (new\_current == nullptr)  
 break;  
 }  
 else   
 for (int i = index; i > 0; i--) {  
 new\_current = new\_current->next;  
 if (new\_current == nullptr)  
 break;  
 }  
 return Iterator(parent\_list, new\_current, position + index);  
 }  
 inline Iterator operator-(int i) {  
 return \*this + (-i);  
 }  
   
 T& operator()() const {  
 return current->data;  
 }  
 };  
 // Basic Functions  
 inline unsigned int len() const {  
 return length;  
 }  
 T& operator[] (int index) const {  
 return getNode(index)->data;  
 }  
 bool append(const typename list\_h::AddConstToType<T>::type& new\_data) {  
 if (last == nullptr) {  
 last = new Node(new\_data, nullptr);  
 first = last;  
 if (last == nullptr)  
 return false;  
 } else {  
 Node\* new\_last = last->makeNext(new\_data);  
 if (new\_last == nullptr) return false;  
 last = new\_last;  
 }  
 ++length;  
 return true;  
 }  
 bool prepend(const typename list\_h::AddConstToType<T>::type& new\_data) {  
 if (last == nullptr) {  
 first = last = new Node(new\_data, nullptr);  
 if (first == nullptr)  
 return false;  
 } else {  
 Node\* new\_node = new Node(new\_data, nullptr);  
 if (new\_node == nullptr)  
 return false;  
 new\_node->next = first;  
 first->previous = new\_node;  
 first = new\_node;  
 }  
 ++length;  
 return true;  
 }  
 private :  
 void removeByPointer(Node\* node) {  
 if (node == nullptr)  
 return;  
 // Neighbour management  
 if (node->next != nullptr) node->next->previous = node->previous;  
 if (node->previous != nullptr) node->previous->next = node->next;  
 // Boundary conditions  
 if (node == first)  
 first = node->next;  
 if (node == last)  
 last = node->previous;  
 // Guillotine  
 delete node;  
 --length;  
 }  
 public :  
 void remove(unsigned int index) {  
 removeByPointer(getNode(index));  
 }  
 // Constructors  
 List() : first(nullptr), last(nullptr), length(0) {}  
 List(const T\* const data\_arr, unsigned int len) : List() {  
 for (int i = 0; i < len; ++i)  
 if (not append(data\_arr[i]))  
 break;  
 }  
 List(unsigned int len) : List() {  
 length = len;  
 first = new Node(nullptr);  
 if (first == nullptr)  
 return;  
 last = first;  
 for (int i = 1; i < length; ++i) {  
 Node\* new\_last = last->makeNext();  
 if (new\_last == nullptr)  
 i = length;  
 else  
 last = new\_last;  
 }  
 }  
 List& operator=(const List& old\_list) {  
 length = old\_list.length;  
 if (length != 0) {  
 for (Iterator i(old\_list); not i.hasEnded(); ++i)  
 if (not append(i()))  
 break;  
 } else {  
 first = last = nullptr;  
 }  
 return \*this;  
 }  
 List(const List& old\_list) : List() {  
 \*this = old\_list;  
 }  
 ~List() {  
 if (length != 0) {  
 for (Iterator i(\*this, false); not i.hasEnded(); ) {  
 Node\* current = i.current;  
 --i;  
 delete current;  
 }  
 first = last = nullptr;  
 }  
 }  
   
 bool insert(unsigned int index, const typename list\_h::AddConstToType<T>::type& new\_data) {  
 if (index == length)   
 return append(new\_data);  
 else if (index == 0)   
 return prepend(new\_data);  
   
 Node\* cur = getNode(index);  
 if (cur == nullptr)  
 return false;  
 Node\* new\_node = new Node(new\_data, nullptr);  
 if (new\_node == nullptr)  
 return false;  
   
 new\_node->next = cur;  
 new\_node->previous = cur->previous;  
 if (new\_node->previous != nullptr) new\_node->previous->next = new\_node;  
 cur->previous = new\_node;  
 ++length;  
 }  
 void truncate(int end) {  
 // Domain folding  
 if (length == 0)  
 end = 0;  
 else if (end < 0)  
 end = end % length + length;  
 else  
 end = end % length;  
   
 Iterator i(\*this);  
 for ( i.last(); not i.hasEnded() and i.position >= end; ) {  
 Node\* current = i.current;  
 --i;  
 delete current;  
 }  
 // Loose ends management  
 last = i.current;  
 if (i.current != nullptr) i.current->next = nullptr;  
 length = end;  
 }  
 void slice(int start, int end, unsigned int step = 1) {  
 // Domain folding  
 if (length == 0)  
 end = 0;  
 else if (end < 0)  
 end = end % length + length;  
 else  
 end = end % length;  
 if (length == 0)  
 start = 0;  
 else if (start < 0)  
 start = start % length + length;  
 else  
 start = start % length;  
   
 if (start >= end)  
 return;  
 truncate(end);  
   
 Iterator i(\*this);  
 for ( i.first(); not i.hasEnded() and i.position < start; ) {  
 Node\* current = i.current;  
 ++i;  
 delete current;  
 }  
 // Loose ends management  
 first = i.current;  
 if (i.current != nullptr) i.current->prev = nullptr;  
 length -= start;  
 // No iterators here because the list is dynamically changing  
 if (step != 1 or step != 0) {  
 for (int i = 0, offset = 0; i < length; ++i) {  
 if ((i + offset) % step != 0) {  
 remove(i);  
 ++offset; --i;  
 }  
 }  
 reCalculateLength();  
 }  
 }  
 void swap(unsigned int i\_1, unsigned int i\_2) {  
 Node\* temp;  
 // Ensure that i\_1 <= i\_2   
 if (i\_1 > i\_2) {  
 unsigned int temp = i\_1;  
 i\_1 = i\_2;  
 i\_2 = temp;  
 }  
 if (i\_1 == i\_2) return;  
 Node\* node\_1 = getNode(i\_1);  
 Node\* node\_2 = getNode(i\_2);  
 if (node\_1 == nullptr or node\_2 == nullptr)   
 return;  
   
 // Check if list markers need to be changed  
 if (i\_1 == 0)  
 first = node\_2;  
 else if (i\_1 == length - 1)  
 last = node\_2;  
 if (i\_2 == 0)  
 first = node\_1;  
 else if (i\_2 == length - 1)  
 last = node\_1;  
 // Alternate behaviour for adjacent swaps  
 if (i\_1 + 1 == i\_2) {  
 node\_1->next = node\_2->next;  
 node\_2->previous = node\_1->previous;  
   
 node\_1->previous = node\_2;  
 node\_2->next = node\_1;  
   
 if (node\_1->next != nullptr) node\_1->next->previous = node\_1;  
 if (node\_2->previous != nullptr) node\_2->previous->next = node\_2;  
   
 return;  
 }  
 // Standard swap behaviour  
 temp = node\_1->previous;  
 node\_1->previous = node\_2->previous;  
 node\_2->previous = temp;  
   
 temp = node\_1->next;  
 node\_1->next = node\_2->next;  
 node\_2->next = temp;  
 // Correct neighbours addresses  
 if (node\_1->next != nullptr) node\_1->next->previous = node\_1;  
 if (node\_1->previous != nullptr) node\_1->previous->next = node\_1;  
   
 if (node\_2->next != nullptr) node\_2->next->previous = node\_2;  
 if (node\_2->previous != nullptr) node\_2->previous->next = node\_2;  
 }  
 void sort(const list\_h::BinaryFunctor<T>& func) {  
 class Dummy {  
 public :   
 void quickSort(List& arr, unsigned int start, unsigned int end, const list\_h::BinaryFunctor<T>& func) {  
 if (end - start <= 1)   
 return;  
 int pivotIndex = start;  
 for (int i = start; i < end - 1; ++i) {  
 if (func(arr[i], arr[end - 1])) {  
 arr.swap(i, pivotIndex);  
 ++pivotIndex;  
 }  
 }  
 arr.swap(pivotIndex, end - 1);  
 quickSort(arr, start, pivotIndex, func);  
 quickSort(arr, pivotIndex + 1, end, func);  
 }  
 } dummy\_dum\_dum;  
 dummy\_dum\_dum.quickSort(\*this, 0, length, func);  
 }  
 int includes(const typename list\_h::AddConstToType<T>::type& search\_data) const {  
 for (Iterator i(\*this); not i.hasEnded(); i++)   
 if (i() == search\_data and i.position < length)  
 return i.position;  
   
 return -1;  
 }  
 };  
#endif

* **set.h**

#if ! defined(VARAD\_SET\_H\_INCLUDED)  
 #define VARAD\_SET\_H\_INCLUDED  
   
 namespace set\_h {  
 const int check\_num = 0x1234abcd;  
   
 // Add const to type if it is a pointer  
 template <class T>  
 struct AddConstToType {  
 typedef T type;  
 };  
 template <class T>  
 struct AddConstToType<T\*> {  
 typedef const T\* type;  
 };  
   
 template <class T>  
 class UnaryFunctor {  
 public :  
 virtual bool function(const typename AddConstToType<T>::type& a) const = 0;  
 inline bool operator()(const typename AddConstToType<T>::type& a) const {  
 return function(a);  
 }  
 };  
 }  
   
 template <typename T>  
 class Set {  
 protected :  
 class Node {  
 public :  
 T data;  
 Node\* next;  
 Node\* previous;  
 const int check;  
   
 Node() : check(set\_h::check\_num) {}  
 // Weird cast of const T new\_data to T so that it can be copied to data  
 Node(const typename set\_h::AddConstToType<T>::type& new\_data, Node\* prev, Node\* nxt = nullptr)  
 : check(set\_h::check\_num), data((T)new\_data), previous(prev), next(nxt) {}  
 Node(Node\* prev) : check(set\_h::check\_num), previous(prev), next(nullptr) {}  
 Node(const Node& old\_node, Node\* prev = nullptr) : Node(old\_node.data, prev) {}  
 ~Node() {  
 data.~T();  
 next = previous = nullptr;  
 }  
   
 bool isOk() const {  
 return check == set\_h::check\_num;  
 }  
 Node\* makeNext(const typename set\_h::AddConstToType<T>::type& new\_data) {  
 Node\* new\_next = new Node(new\_data, this);  
 if (new\_next != nullptr) next = new\_next;  
 return new\_next;  
 }  
 Node\* makeNext() {  
 Node\* new\_next = new Node(this);  
 if (new\_next != nullptr) next = new\_next;  
 return new\_next;  
 }  
 };  
   
 Node\* first;  
 Node\* last;  
 unsigned int length;  
   
 Node\* getNode(int index) const {  
 if (length == 0)  
 index = 0;  
 else if (index < 0)  
 index = index % length + length;  
 else  
 index = index % length;  
   
 Node\* cur;  
 if (2 \* index < length) {  
 cur = first;  
 for (int i = 1; i <=index; i++) {  
 cur = cur->next;  
 }  
 } else {  
 cur = last;  
 for (int i = length - 2; i >= index; i--) {  
 cur = cur->previous;  
 }  
 }  
 return cur;  
 }  
   
 public:   
 class Iterator {  
 friend class Set;  
   
 const Set& parent\_set;  
 Node\* current;  
   
 public :  
 int position;  
   
 private :  
 Iterator(const Set& pops, Node\* cur, int pos) : parent\_set(pops), current(cur), position(pos) {}  
   
 public :  
 // Basic Functions  
 Iterator first() {  
 current = parent\_set.first;  
 position = 0;  
 }  
 Iterator last() {  
 current = parent\_set.last;  
 position = parent\_set.length - 1;  
 }  
 // Constructors  
 Iterator(const Set& papa, bool start\_at\_first = true) : parent\_set(papa) {  
 if (start\_at\_first)  
 first();  
 else   
 last();  
 };  
 Iterator(const Iterator& i) : parent\_set(i.parent\_set), current(i.current), position(i.position) {}  
 Iterator& operator=(const Iterator& i) {  
 if (&parent\_set == &(i.parent\_set)) {  
 current = i.current;  
 position = i.position;  
 }  
 return \*this;  
 }  
   
 bool hasEnded() const {  
 return current == nullptr or position >= parent\_set.length or position < 0;  
 }  
 bool operator==(const Node\* const node) const {  
 return current == node;  
 }  
 bool operator==(const Iterator& i) const {  
 return current == i.current;  
 }  
 bool operator!=(const Node\* const node) const {  
 return not current == node;  
 }  
 bool operator!=(const Iterator& i) const {  
 return not current == i.current;  
 }  
 Iterator& operator++() {  
 if (current != nullptr) current = current->next;  
 ++position;  
 return \*this;  
 }  
 Iterator& operator++(int) {  
 return ++(\*this);  
 }  
 inline Iterator& operator--() {  
 if (current != nullptr) current = current->previous;  
 --position;  
 return \*this;  
 }  
 inline Iterator& operator--(int) {  
 return --(\*this);  
 }  
 Iterator operator+(int index) {  
 Node\* new\_current = current;  
 if (index < 0)  
 for(int i = index; i < 0; i++) {  
 new\_current = new\_current->previous;  
 if (new\_current == nullptr)  
 break;  
 }  
 else   
 for (int i = index; i > 0; i--) {  
 new\_current = new\_current->next;  
 if (new\_current == nullptr)  
 break;  
 }  
 return Iterator(parent\_set, new\_current, position + index);  
 }  
 inline Iterator operator-(int i) {  
 return \*this + (-i);  
 }  
   
 T& operator()() const {  
 return current->data;  
 }  
 };  
 // Basic Functions  
 inline unsigned int size() const {  
 return length;  
 }  
 bool contains(const typename set\_h::AddConstToType<T>::type& search\_data) const {  
 for (Iterator i(\*this); not i.hasEnded(); i++)   
 if (i() == search\_data and i.position < length)  
 return true;  
   
 return false;  
 }  
 bool add(const typename set\_h::AddConstToType<T>::type& new\_data) {  
 if (last == nullptr) {  
 last = new Node(new\_data, nullptr);  
 first = last;  
 if (last == nullptr)  
 return false;  
 ++length;  
 } else if (not contains(new\_data)) {  
 Node\* new\_last = last->makeNext(new\_data);  
 if (new\_last == nullptr) return false;  
 last = new\_last;  
 ++length;  
 }  
 return true;  
 }  
 void remove(const typename set\_h::AddConstToType<T>::type& new\_data) {  
 Iterator i(\*this);  
 for (; not (i.hasEnded() or new\_data == i()); (new\_data == i()) ? i : ++i);  
 if (i.hasEnded())  
 return;  
   
 Node\* cur = i.current;  
 if (cur == nullptr)  
 return;  
 if (cur->next != nullptr) cur->next->previous = cur->previous;  
 if (cur->previous != nullptr) cur->previous->next = cur->next;  
   
 if (cur == first)  
 first = cur->next;  
 if (cur == last)  
 last = cur->previous;  
   
 delete cur;  
 --length;  
 }  
 // Constructors  
 Set() : first(nullptr), last(nullptr), length(0) {}  
 Set(const T\* const data\_arr, unsigned int len) : Set() {  
 for (int i = 0; i < len; ++i)  
 add(data\_arr[i]);  
 }  
 Set& operator=(const Set& old\_set) {  
 length = old\_set.length;  
 if (length != 0) {  
 for (Iterator i(old\_set); not i.hasEnded(); ++i)  
 if (not add(i()))  
 break;  
 } else {  
 first = last = nullptr;  
 }  
 return \*this;  
 }  
 Set(const Set& old\_set) : Set() {  
 \*this = old\_set;  
 }  
 ~Set() {  
 if (length != 0) {  
 for (Iterator i(\*this, false); not i.hasEnded(); ) {  
 Node\* current = i.current;  
 --i;  
 delete current;  
 }  
 first = last = nullptr;  
 }  
 }  
   
 bool isASubset(const Set& maybe\_subset) const {  
 for (Iterator i(maybe\_subset); not i.hasEnded(); i++)  
 if (not contains(i()))  
 return false;  
 return true;  
 }  
 bool operator==(const Set& B) const {  
 if (not isASubset(B))  
 return false;  
 return size() == B.size();  
 }  
 bool isAProperSubset(const Set& maybe\_subset) const {  
 if (not isASubset(maybe\_subset))  
 return false;  
 return size() != maybe\_subset.size();  
 }  
 Set subset(const set\_h::UnaryFunctor<T>& func) const {  
 Set subset;  
 for (Iterator i(\*this); not i.hasEnded(); i++)  
 if (func(i()))  
 subset.add(i());  
 return subset;  
 }  
 Set intersection(const Set& B) const {  
 class Dummy : public set\_h::UnaryFunctor<T> {  
 const Set& other;  
 Dummy(const Set& another) : other(another) {}  
 bool function (const typename set\_h::AddConstToType<T>::type& a) const {  
 return other.contains(a);  
 }  
 } dummy\_dum(B);  
 return subset(dummy\_dum);  
 }  
 Set join(const Set& B) const {  
 Set join(B);  
 for (Iterator i(\*this); not i.hasEnded(); i++)  
 join.add(i());  
 return join;  
 }  
 bool disjoint(const Set& B) const {  
 for (Iterator i(\*this); not i.hasEnded(); i++)  
 if (B.contains(i()))  
 return false;  
 return true;  
 }  
 Set difference(const Set& B) const {  
 class Dummy : public set\_h::UnaryFunctor<T> {  
 const Set& other;  
 Dummy(const Set& another) : other(another) {}  
 bool function (const typename set\_h::AddConstToType<T>::type& a) const {  
 return not other.contains(a);  
 }  
 } dummy\_dum(B);  
 return subset(dummy\_dum);  
 }  
 Set symmetric\_difference(const Set& B) const {  
 Set sym\_diff(B);  
 for (Iterator i(\*this); not i.hasEnded(); i++)  
 if (not B.contains(i()))  
 sym\_diff.add(i());  
 for (Iterator i(B); not i.hasEnded(); i++)  
 if (not contains(i()))  
 sym\_diff.add(i());  
 return sym\_diff;  
 }  
 };  
#endif

* **graph.h**

#if !defined(VARAD\_GRAPH\_H\_INCLUDED)  
 #define VARAD\_GRAPH\_H\_INCLUDED  
   
 #include "list.h"  
 #include <iostream>  
 #include <iomanip>  
   
 namespace graph\_h {  
 // Add const to type if it is a pointer  
 template <class T>  
 struct AddConstToType {  
 typedef T type;  
 };  
 template <class T>  
 struct AddConstToType<T\*> {  
 typedef const T\* type;  
 };  
   
 template <class T>  
 class BinaryFunctor {  
 public :  
 virtual bool function(const typename AddConstToType<T>::type& a, const typename AddConstToType<T>::type& b) const = 0;  
 inline bool operator()(const typename AddConstToType<T>::type& a, const typename AddConstToType<T>::type& b) const {  
 return function(a, b);  
 }  
 };  
 }  
   
 template <class T>  
 class Graph {  
 public :  
 class Vertex {  
 public :   
 T data;  
 List<Vertex\*> edges;  
 // Vertex Constructors  
 Vertex() : data(), edges() {}  
 // Weird cast of const T new\_data to T so that it can be copied to data  
 Vertex(const typename graph\_h::AddConstToType<T>::type& new\_data) : data((T)new\_data), edges() {}  
 Vertex(const typename graph\_h::AddConstToType<T>::type& new\_data, Vertex\*\* edge\_list, unsigned int edge\_count)   
 : data((T)new\_data), edges(edge\_list, edge\_count) {}  
 Vertex(const Vertex& old\_vertex) : data(old\_vertex.data), edges(old\_vertex.edges) {}  
 // Equivalence checks are dependent on the Vertex data, not the edges  
 inline bool operator==(const Vertex& other\_vertex) const {  
 return data == other\_vertex.data;  
 }  
 inline bool operator!=(const Vertex& other\_vertex) const {  
 return not (\*this == other\_vertex.data) ;  
 }  
 // Functions for edge management  
 bool isConnected() const {  
 return edges.len() != 0;  
 }  
 bool isConnectedWithVertex(const Vertex\* const other\_vertex) const {  
 if (other\_vertex != nullptr and other\_vertex != this)  
 return edges.includes(other\_vertex) != -1;  
 return false;  
 }  
 bool isConnectedWithVertex(const Vertex& other\_vertex) const {  
 return isConnectedWithVertex(&other\_vertex);  
 }  
 void connectTo(const Vertex\* const other\_vertex) {  
 if (other\_vertex != nullptr and other\_vertex != this) {  
 if (edges.includes(other\_vertex) == -1)  
 edges.append(other\_vertex);  
 }  
 }  
 void disconnectFrom(const Vertex\* const other\_vertex) {  
 if (other\_vertex != nullptr and other\_vertex != this) {  
 int i = edges.includes(other\_vertex);  
 if (i != -1)  
 edges.remove(i);  
 }  
 }  
 void connectTo(const Vertex& other\_vertex) {  
 connectTo(&other\_vertex);  
 }  
 void disconnectFrom(const Vertex& other\_vertex) {  
 disconnectFrom(&other\_vertex);  
 }  
 // Vertex Destructor  
 ~Vertex() {  
 data.~T();  
 for (typename List<Vertex\*>::Iterator connected\_vertex(edges); not connected\_vertex.hasEnded(); connected\_vertex++)  
 connected\_vertex()->disconnectFrom(this);  
 edges.~List();  
 }  
 };  
   
 List<Vertex> vertices;  
   
 Graph() : vertices() {}  
 Graph(Graph& old\_graph) : vertices(old\_graph.vertices) {}  
   
 void makeVertex(const typename graph\_h::AddConstToType<T>::type& data) {  
 vertices.append(Vertex(data));  
 }  
 void removeVertex(unsigned int index) {  
 vertices.remove(index);  
 }  
   
 void connectVertices(unsigned int i\_1, unsigned int i\_2) {  
 Vertex\* v\_1 = &(vertices[i\_1]);  
 Vertex\* v\_2 = &(vertices[i\_2]);  
   
 if (v\_1 == v\_2 or v\_1 == nullptr or v\_2 == nullptr)  
 return;  
   
 v\_1->connectTo(v\_2);  
 v\_2->connectTo(v\_1);  
 }  
 void disconnectVertices(unsigned int i\_1, unsigned int i\_2) {  
 Vertex\* v\_1 = &(vertices[i\_1]);  
 Vertex\* v\_2 = &(vertices[i\_2]);  
   
 if (v\_1 == v\_2 or v\_1 == nullptr or v\_2 == nullptr)  
 return;  
   
 v\_1->disconnectFrom(v\_2);  
 v\_2->disconnectFrom(v\_1);  
 }  
   
 void makeConnections(const graph\_h::BinaryFunctor<T>& func) {  
 // Cycle through each possible vertex pair and connect them  
 // if the given function returns true  
 for (typename List<Vertex>::Iterator i(vertices); not i.hasEnded(); i++)   
 for (typename List<Vertex>::Iterator j = i + 1; not j.hasEnded(); j++)   
 if (func(i().data, j().data)) {  
 i().connectTo(j());  
 j().connectTo(i());  
 }  
 }  
   
 List<List<Vertex\*>> colorVertices() {  
 List<List<Vertex\*>> color\_list;  
   
 for (typename List<Vertex>::Iterator i(vertices); not i.hasEnded(); i++) {  
 typename List<List<Vertex\*>>::Iterator color(color\_list);  
 // Determine if vertex is connected to any other colored vertex  
 bool is\_connected = true;  
 // Come out color-checking loop if colors have run out or   
 // the current vertex is not connected to any vertex of the given color  
 //   
 // Don't increment color if none are connected  
 for (; not (color.hasEnded() or not is\_connected); is\_connected ? color++ : color)  
 // Come out of vertex-connection-checking loop  
 // if there are no more vertices to check or  
 // the current vertex is connected to any vertex of the given color  
 for (typename List<Vertex\*>::Iterator j( (is\_connected = false, color()) ); not (j.hasEnded() or is\_connected); ++j)  
 is\_connected |= i().isConnectedWithVertex(j());  
 // If current vertex was not connected to any in the last color,  
 // switch back to it  
 if (color.hasEnded() and not is\_connected)  
 color.last();  
 if (not color.hasEnded()) {  
 // Color the vertex if an unconnected color was found  
 color().append(&i());  
 } else {  
 // Make a new color if the vertex is connected to all the current colors  
 List<Vertex\*> new\_color;  
 new\_color.append(&i());  
 color\_list.append(new\_color);  
 }  
 }  
   
 return color\_list;  
 }  
 };  
   
#endif

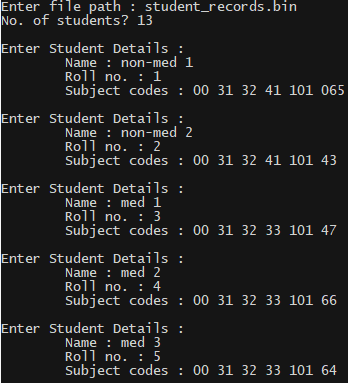
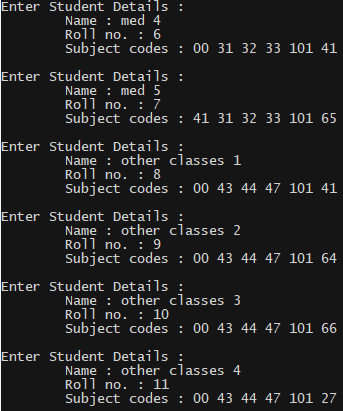
* **classes.h**

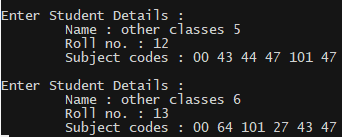
#if ! defined(VARAD\_CLASSES\_H\_INCLUDED)  
 #define VARAD\_CLASSES\_H\_INCLUDED  
   
 #include <iostream>  
 #include <limits>  
   
 struct Subject {  
 char name[51];  
 unsigned short code;  
   
 Subject() : code(0) {name[0] = '\0';}  
 Subject(unsigned short c) : code(c) {name[0] = '\0';}  
 Subject(unsigned short c, std::string n) : code(c) {  
 int i;  
 for (std::string::iterator it = (i = 0, n.begin()); it != n.end() and i < 50; ++i, ++it)  
 name[i] = \*it;  
 name[i] = '\0';  
 }  
 Subject(unsigned short c, char\* n) : code(c) {  
 int i;  
 for (int i = 0 ; i < 50 and n[i] != '\0'; ++i)  
 name[i] = n[i];  
 name[i] = '\0';  
 }  
 Subject& operator=(const Subject& s) {  
 code = s.code;  
 int i;  
 for (i = 0; int(i) < 50 and s.name[i] != '\0'; ++i)  
 name[i] = s.name[i];  
 name[i] = '\0';  
 return \*this;  
 }  
 Subject(const Subject& s) {  
 \*this = s;  
 }  
   
 bool operator==(const Subject& s) const {  
 return code == s.code;  
 }  
 };  
  
 struct Student {  
 char name[50];  
 unsigned short roll\_no;  
 unsigned short subjects[6];  
   
   
 void display(std::ostream& out\_strm) {  
 out\_strm << "\n\tName : " << name;  
 out\_strm << "\n\tRoll no. : " << roll\_no;  
 out\_strm << "\n\tSubjects : ";  
 for (int i = 0; i < 5; ++i)  
 out\_strm << subjects[i] << ", ";  
 out\_strm << subjects[5];  
 }  
 void getData(std::istream& in\_strm, std::ostream& out\_strm) {  
 out\_strm << "\nEnter Student Details :";  
 in\_strm.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');  
 out\_strm << "\n\tName : "; in\_strm.getline(name, 20);  
 out\_strm << "\tRoll no. : "; in\_strm >> roll\_no;  
 out\_strm << "\tSubject codes : "; for (int i = 0; i < 6; ++i) in\_strm >> subjects[i];  
 }  
 };  
   
#endif

* **record\_maker.cpp**

#include <iostream>  
#include <fstream>  
#include "classes.h"  
  
int main() {  
 int n;  
 char file\_path[100];  
 Student stu;  
 std::cout << "Enter file path : "; std::cin.getline(file\_path, 100);  
 std::ofstream out(file\_path, std::ios::out | std::ios::binary | std::ios::trunc);  
 if (not (out.is\_open() and out.good())) {  
 std::cerr << "FileError : cannot open file \"" << file\_path << "\"";  
 return 1;  
 }  
 std::cout << "No. of students? "; std::cin >> n;  
 for (int i = 0; i < n; ++i) {  
 stu.getData(std::cin, std::cout);  
 out.write( (char\*)&stu, sizeof(Student));  
 }  
}

Output :

****

* **main.cpp**

#include <string.h>  
#include <iostream>  
#include <fstream>  
#include "graph.h"  
#include "set.h"  
#include "classes.h"  
  
namespace commands {  
 struct Command {  
 const char code;  
 const std::string description;  
 };  
 const Command display = {'d', "Display current subject sets"};  
 const Command add = {'a', "Add a subject set"};  
 const Command remove = {'r', "Remove a subject set"};  
 const Command name = {'n', "Add names to subjects"};  
 const Command connect = {'c', "Connect two subjects"};  
 const Command disconnect = {'k', "Disconnect two subjects"};  
 const Command procedurally\_connect = {'p', "Connect any two sets with a common subject"};  
 const Command read\_file = {'f', "Wipe current subject sets and read all subject sets from a student records file"};  
 const Command color\_graph = {'l', "Determine the coloring i.e. exam distribution of the subjects"};  
 const Command help = {'h', "Print this help message"};  
 const Command quit = {'q', "Quit the program"};  
   
 enum CommandEnumerator {  
 FIRST,  
 display\_num,  
 add\_num,  
 remove\_num,  
 name\_num,  
 connect\_num,  
 disconnect\_num,  
 procedurally\_connect\_num,  
 read\_file\_num,  
 color\_graph\_num,  
 help\_num,  
 quit\_num,  
 LAST  
 };  
   
 CommandEnumerator getCommandEnumerator(const char com) {  
 if (com == display.code)   
 return CommandEnumerator::display\_num;  
 else if (com == add.code)   
 return CommandEnumerator::add\_num;  
 else if (com == remove.code)   
 return CommandEnumerator::remove\_num;  
 else if (com == name.code)   
 return CommandEnumerator::name\_num;  
 else if (com == connect.code)   
 return CommandEnumerator::connect\_num;  
 else if (com == disconnect.code)   
 return CommandEnumerator::disconnect\_num;  
 else if (com == procedurally\_connect.code)   
 return CommandEnumerator::procedurally\_connect\_num;  
 else if (com == read\_file.code)  
 return CommandEnumerator::read\_file\_num;  
 else if (com == color\_graph.code)   
 return CommandEnumerator::color\_graph\_num;  
 else if (com == help.code)   
 return CommandEnumerator::help\_num;  
 else if (com == quit.code)   
 return CommandEnumerator::quit\_num;   
 }  
 const Command& getCommand(const CommandEnumerator com) {  
 if (com == CommandEnumerator::display\_num)   
 return display;  
 else if (com == CommandEnumerator::add\_num)   
 return add;  
 else if (com == CommandEnumerator::remove\_num)   
 return remove;  
 else if (com == CommandEnumerator::name\_num)   
 return name;  
 else if (com == CommandEnumerator::connect\_num)   
 return connect;  
 else if (com == CommandEnumerator::disconnect\_num)   
 return disconnect;  
 else if (com == CommandEnumerator::procedurally\_connect\_num)   
 return procedurally\_connect;  
 else if (com == CommandEnumerator::read\_file\_num)  
 return read\_file;  
 else if (com == CommandEnumerator::color\_graph\_num)   
 return color\_graph;  
 else if (com == CommandEnumerator::help\_num)   
 return help;  
 else if (com == CommandEnumerator::quit\_num)   
 return quit;   
 }   
}  
  
class Dummy : public graph\_h::BinaryFunctor<Subject> {  
 Set<Set<Subject>>& subject\_choices;  
 public :  
 Dummy(Set<Set<Subject>>& sc) : subject\_choices(sc) {}  
 bool function(const Subject& a, const Subject& b) const {  
 Set<Subject> a\_b\_set;  
 a\_b\_set.add(a), a\_b\_set.add(b);  
 for (Set<Set<Subject>>::Iterator i(subject\_choices); not i.hasEnded(); ++i)   
 if (i().isASubset(a\_b\_set))  
 return true;  
 return false;  
 }  
};  
  
int main () {  
 unsigned short code;  
 char user\_choice;  
   
 Graph<Subject> subjects;  
 Set<Set<Subject>> subject\_choices;  
   
 Dummy dummy\_dum(subject\_choices);  
   
 do {  
 std::cout << "\n\nEnter your choice : "; std:: cin >> user\_choice;  
   
 switch ( commands::getCommandEnumerator(user\_choice) ) {  
 case commands::CommandEnumerator::display\_num :  
 std::cout << "\nIndex Code Name Connections (indices)";  
 std::cout << "\n===== ==== ================================================== =====================";  
   
 for (typename List<Graph<Subject>::Vertex>::Iterator i(subjects.vertices); not i.hasEnded(); ++i) {  
 std::cout << '\n' << std::setw(5) << i.position << " " << std::setw(4) << i().data.code << " " << std::setw(50) << i().data.name << " ";  
 for (typename List<Graph<Subject>::Vertex\*>::Iterator j(i().edges); not j.hasEnded(); ++j)  
 std::cout << subjects.vertices.includes(\*j()) << ", ";  
 }  
 break;  
 case commands::CommandEnumerator::add\_num : {  
 std::cout << "Enter the 6 subject codes in the set to be added (00 - null) : ";  
 Set<Subject> subject\_combo;  
 for (int i = 0; i < 6; ++i) {  
 std::cin >> code;  
 Subject sub(code);  
 subject\_combo.add(sub);  
 if ( subjects.vertices.includes(Graph<Subject>::Vertex(sub)) == -1)  
 subjects.makeVertex(sub);  
 }  
 subject\_choices.add(subject\_combo);  
 }  
 break;  
 case commands::CommandEnumerator::remove\_num : {  
 std::cout << "Enter the 6 subject codes in the set to be removed (00 - null) : ";  
 Set<Subject> subject\_combo;  
 for (int i = 0; i < 6; ++i) {  
 std::cin >> code;  
 Subject sub(code);  
 subject\_combo.add(sub);  
 int sub\_pos = subjects.vertices.includes(Graph<Subject>::Vertex(sub));  
 if (subjects.vertices[sub\_pos].edges.len() == 0)  
 subjects.removeVertex(sub\_pos);  
 }  
 subject\_choices.remove(subject\_combo);  
 }  
 break;  
 case commands::CommandEnumerator::name\_num : {  
 char c;  
 std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');  
 for (List<Graph<Subject>::Vertex>::Iterator i(subjects.vertices); not i.hasEnded(); ++i) {  
 std::cout << "Name of subject with code " << i().data.code << " : ";  
 std::cin.getline(i().data.name, sizeof(i().data.name));  
 }  
 }  
 break;  
 case commands::CommandEnumerator::connect\_num : {  
 int i, j;  
 std::cout << "\nEnter indices of the subjects to connect : "; std::cin >> i >> j;  
 subjects.connectVertices(i, j);  
 }  
 break;  
 case commands::CommandEnumerator::disconnect\_num : {  
 int i, j;  
 std::cout << "\nEnter indices of the subjects to disconnect : "; std::cin >> i >> j;  
 subjects.disconnectVertices(i, j);  
 }  
 break;  
 case commands::CommandEnumerator::procedurally\_connect\_num : {  
 std::cout << "Connecting any two subjects featuring in the same subject set...";  
 subjects.makeConnections(dummy\_dum);  
 }  
 break;  
 case commands::CommandEnumerator::read\_file\_num : {  
 char file\_name[51];  
 std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');  
 std::cout << "Enter the file name of the records file : "; std::cin.getline(file\_name, 51);  
   
 Student stu;  
 std::ifstream records(file\_name, std::ios::binary | std::ios::in);  
 if (not (records.is\_open() and records.good())) {  
 records.close();  
 std::cerr << "FileError : cannot open file stream \"" << file\_name << "\"";  
 break;  
 }  
 records.seekg(0, std::ios::beg);  
 // Wipe everything clean  
 subjects.vertices.truncate(0);  
 subject\_choices = Set<Set<Subject>>();  
 // Load all the subject sets one by one  
 while( records.read((char\*)(&stu), sizeof(Student))) {  
 Set<Subject> subject\_set;  
 for (int i = 0; i < 6; ++i)  
 subject\_set.add(Subject(stu.subjects[i]));  
 subject\_choices.add(subject\_set);  
 }  
   
 // Get a set of all subjects  
 Set<Subject> all\_subjects;  
 for (Set<Set<Subject>>::Iterator i(subject\_choices); not i.hasEnded(); ++i)   
 all\_subjects = all\_subjects.join(i());;  
 // Make a new one  
 for (Set<Subject>::Iterator i(all\_subjects); not i.hasEnded(); ++i)   
 subjects.makeVertex(i());  
 subjects.makeConnections(dummy\_dum);  
 // Connect vertices  
 subjects.makeConnections(dummy\_dum);  
 }  
 break;  
 case commands::CommandEnumerator::color\_graph\_num : {   
 List<List<Graph<Subject>::Vertex\*>> color\_scheme = subjects.colorVertices();  
 for (typename List<List<Graph<Subject>::Vertex\*>>::Iterator i(color\_scheme); not i.hasEnded(); ++i) {  
 std::cout << "\nDay " << i.position + 1 << " - ";  
 for (typename List<Graph<Subject>::Vertex\*>::Iterator j(i()); not j.hasEnded(); ++j)   
 std::cout << j()->data.code << ", ";  
 }  
 }  
 break;  
 case commands::CommandEnumerator::help\_num :  
 for (commands::CommandEnumerator com = commands::CommandEnumerator::FIRST; (com = commands::CommandEnumerator(com + 1)) != commands::CommandEnumerator::LAST;) {  
 const commands::Command& command = commands::getCommand(com);  
 std::cout << "\n\t" << command.code << " - " << command.description;  
 }  
 std::cout << '\n';  
 break;  
 case commands::CommandEnumerator::quit\_num :  
 std::cout << "\nBye!!\n";  
 break;  
 default :   
 std::cout << "\nInvalid Option. Enter \'" << commands::help.code << "\' for a help message";  
 break;  
   
 }  
 } while (user\_choice != commands::quit.code);  
}

Output :

