Unit 5.Trees and Graphs

5.1 Trees

With this module, the course introduces trees and the tree based STL containers.

Readings:

- **Required:**

  > **Weiss, chapter 18, sections 7.7 - 7.9, 19.1.**

  Remark:

  > **Remember that this book supplements the course's online material. You will be asked questions based on this material.**

- **Required:**

  > **Schildt, chapters 33 through 37. Remark: Remember that this book serves as a general reference to the C++ language, not a course textbook. Therefore, you should browse through the assigned sections in order to get a sense of what they have to offer and where and how they treat important topics. Do not study the sections assigned in this book as you would assignments from a textbook: your goal here should be familiarity, not mastery.**

5.1.1 Introduction to Trees

A Tree as a Data Structure

A tree is a data structure that generally supports element insertion, access, and removal in less than linear time. Trees consist of an arrangement of nodes such that the following conditions hold.

- Only one node has no predecessor: the root.
- Every node other than the root has a unique predecessor.
- Starting at any node, one can reach the root by repeatedly stepping from a node to its predecessor.

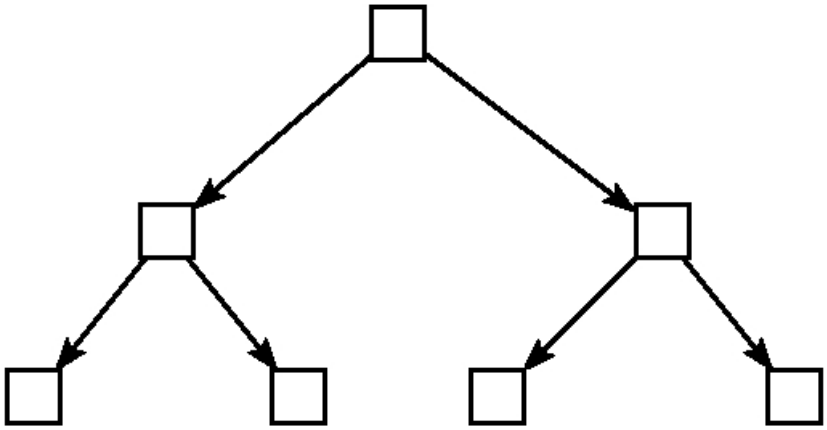

Figure 1 A basic tree

All nodes other than the root node have a predecessor, or parent node. These nodes are said to be child nodes of the parent. Nodes that have no children are leaf nodes. The height of a tree is the number of levels of nodes, including the root node, which exist in the tree. The tree in Figure 1 has a height of three and contains four leaf nodes.

Trees have many applications in Computer Science. The representation of categories and sub-categories for an online bookstore, for instance, is a common application of trees. Programmers use trees to represent file systems. Trees are also used in the implementation of compilers and in file compression algorithms.

Types of Trees

Tree implementations can be categorized into different types. This categorization process takes into account several properties and characteristics of trees. A property that categorizes a tree implementation is the maximum number of children that can exist in any given node. If the maximum number of children in any given node is not limited in an implementation, we consider the tree to be a general hierarchy.

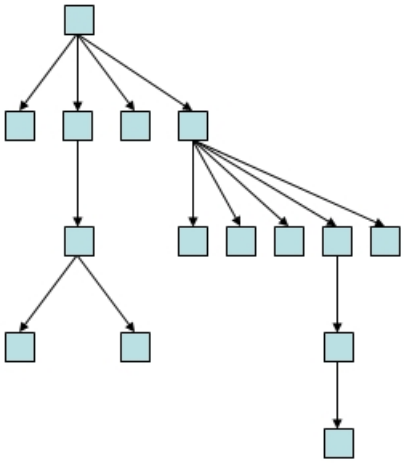

Figure 2 A general tree structure

Tree implementations that limit the number of child nodes for any given node form a separate class of trees. The most common type of these implementations limits the number of nodes to two. These types of trees are known as binary trees. Less commonly used implementations that limit the number of child nodes to three is known as a ternary trees.

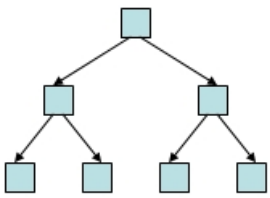

Figure 3 A binary tree

Binary trees can be further categorized based on the ordering of elements within the tree. A heap is a binary tree that maintains elements in either increasing or decreasing order. At each level in a heap, the value at a node is either less than or greater than the values at all nodes below it. Two types of heaps exist: min heaps and max heaps. A min heap contains elements that always have increasingly larger values at each level in the tree. This means the root node of the tree contains the smallest element in the tree. Max heaps have their largest element stored at the root and store their remaining elements in decreasing order.

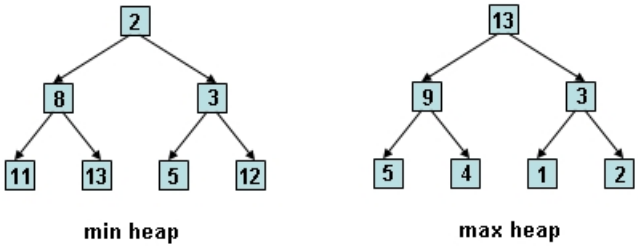

Figure 4 A min heap and a max heap

Trees as Search Structures

Binary search trees are another type of binary tree characterized by the ordering of the elements with the tree. Binary search trees maintain elements in a sorted order. Because of this, binary search trees effectively support searching for individual elements.

Binary search trees maintain items in a sorted order. All elements less than the element at the root of the tree are stored in the tree rooted at the root's left child. All elements greater than the root are stored in the tree rooted at the root's right child. This principle is applied recursively to all nodes in the tree. For any node in a binary search tree, all elements in the sub-tree in the left child are less than the element in the node. Also, all elements in the sub-tree in the right child are greater than the element in the node.

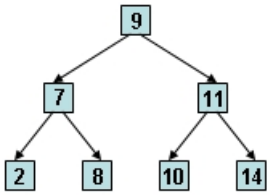

Figure 5 A binary search tree

Not all binary search trees have all leaf nodes at the same level in the tree. Binary search trees can be unbalanced. An unbalanced tree is a tree that has left and right sub-trees whose heights differ by more than one level. Binary search trees become unbalanced when elements of either increasing or decreasing size are continually added to the structure. The performance of a binary search tree approaches linear time as the structure becomes too unbalanced.

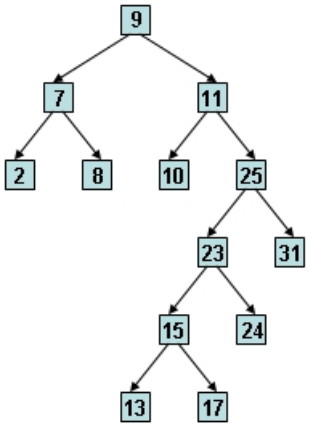

Figure 6 An unbalanced binary search tree

The nodes in a binary search tree do not have to contain two child nodes. A binary search tree limits, but does not require the number of child nodes to be two. A node in a binary search tree can contain only one child. A binary search tree that contains one or more of these types of nodes is an incomplete tree.

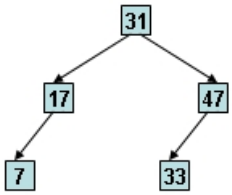

Figure 7 An incomplete binary search tree

5.1.2 Using Tree Structures

Using Binary Search Trees

The BSTree Class Interface

A binary search tree is a tree based data structure that maintains an ordered collection of items. Because of how they maintain the ordering of their stored items, binary search trees can perform certain tasks more efficiently than other data structures. For example, traversing items in sorted order (or in reverse sorted order) is an efficient operation in a binary search tree. This is not the case for a hash table. The ability to search for a range of items is another task easily accomplished using a binary search tree, but not using a hash table.

The implementation of a binary search tree class is covered in chapter 19 of Weiss. This page focuses on the use of binary search trees to solve problems relating to storing a collection of items. As such, we examine the interface of an existing binary search tree class.

```
1  template <class T>
2  class BSTree {
3
4  protected:
5   BSTNode<T> *root;    // root of tree
6   int count;         // size of tree
7
8  public:
9   BSTree() : root(NULL), count(0) {}
10  BSTree(const BSTree&);
11  virtual ~BSTree() {if (root) delete root;}
12
13  virtual int size() const { return count; }
14  virtual bool insert( const T& x );
15  virtual const T* const search( const T& x );
16  virtual bool remove( const T& x );
17
18 protected:
19  virtual BSTNode<T>* copy_tree(BSTNode<T>* nodep);
20  virtual bool insert_helper(BSTNode<T> *&nodep, const T &x);
21  virtual const T* const search_helper(BSTNode<T> * nodep, const T &x);
22  virtual bool remove_helper(BSTNode<T> *&nodep, const T &x);
```

```
23   virtual BSTNode<T>* remove_leftmost_child(BSTNode<T> *&nodep);
24
25 };
```

Listing 1 The `BSTree` interface

Listing 1 contains the declaration of class `BSTree`. This is a binary search tree class that supports operations for element insertion, access, and removal. It also contains a method that returns the number of elements currently stored in the binary search tree.

Class `BSTree` is a template class. This allows us to create instances of the class that can store different data types. There is an important consideration we must make before using this class. Internally, some of the methods of the class use operators `==`, `<`, and `>` to perform the element insertion, access, and removal tasks. Thus, any object we store in this binary search tree class must provide these operators.

```
1  template <class T>
2  const T* const BSTree<T>::search_helper(BSTNode<T> *nodep, const T &x)
   {
3   if (nodep == 0) {
4       return NULL;
5   }
6
7   if (x == nodep->data) {
8       return &(nodep->data);
9   }
10
11  if (x < nodep->data) {
12      return search_helper(nodep->left, x);
13  }
14  else {
15      return search_helper(nodep->right, x);
16  }
17 }
```

Listing 2 Internal use of operators of type `T`

The public data members of class `BSTree` that we use the most have rather straightforward usage. We use the `insert` and `remove` operations to add and delete elements to and from the binary search tree. Both of these functions return a `bool` value. The `insert` function returns `true` when the element is successfully added and `false` when the element already exists. The `remove` function returns `true` if the element is successfully removed and `false` if the element is not found in the binary search tree. Function `size` returns the count of the number of elements in the binary search tree.

The `search` function provides a way to access elements that exist in the binary search tree. If the requested element is not found in the binary search tree, the function returns the null pointer. Otherwise, the function returns a constant pointer to a constant version of the element. The pointer must be constant to prevent a user from attempting to gain access to another portion of memory through pointer arithmetic. The element the pointer points to must be constant to ensure that the integrity of the tree is maintained. Changing the value of a stored element could possibly change where the element should be stored in the binary search tree. The correct process of updating a stored element involves first removing the element, then updating a copy of the element, then inserting the updated copy.

An Application: Customer Database

The following files comprise a program that manages a list of customer contacts. A binary search tree is a good choice to use in this implementation since it allows for efficient element insertion and access.

- `main.cpp` - Main application source code
- `Customer.h` - Declaration of a customer class
- `Customer.cpp` - Implementation of class `Customer`
- `CustomerDatabase.h` - Declaration of a customer database class
- `CustomerDatabase.cpp` - Implementation of class `CustomerDatabase`
- `bst.h` - A binary search tree implementation
- `contacts.dat` - A sample data file

Using Heaps

Heaps are often used to provide efficient access to either the minimum or maximum of a set of values. One use of this functionality is the implementation of a priority queue. A priority queue is a data structure that behaves similar to regular queue in that it provides `push`, `pop`, `size`, and `empty` methods. Priority queues differ from regular queues since the element that a priority queue `pop` removes depends on an assigned priority, and not the "First-In First-Out" principle. Priority queues are used often enough that STL provides a `priority_queue` adapter. The `priority_queue` adapter is covered in detail in 5.1.3 Using the Tree Based STL Containers

5.1.3 Using the Tree Based STL Containers

- Sets
- Maps
- Priority Queues

Sets

The STL `set` container stores unique items in an ordered collection. The tasks of element addition, removal, and access are guaranteed by the STL standard to take logarithmic time in a `set`. Inserting items in sorted order will not degrade these operations to linear time, as is the case with binary search trees.

Declaring objects of class `set` is similar to declaring other STL containers. A programmer specifies the data type that the set can contain as a template parameter. To access the `set` container, a programmer must include the `<set>` library.

```
set<int> s1;
set<int> s2;
```

Listing 1 Declaring `set` objects

Adding items to a `set` is done using the `insert` member function.

```
set<int> s1;
set<int> s2;

for (int i = 0; i < 20; i++) {
 s1.insert(i);
 s2.insert(30 - i);
}
```

Listing 2 Adding items to a set

The member function `size` returns the number of items stored in a set.

```
set<int> s1;
set<int> s2;

for (int i = 0; i < 20; i++) {
 s1.insert(i);
 s2.insert(30 - i);
}

cout << "size of s1: " << s1.size() << endl;
cout << "size of s2: " << s2.size() << endl;
```

Listing 3 Reporting the number of items stored in a set

The `find` function searches for an item in a `set`. This function returns an iterator to the found item. If the item is not found, function `find` returns an iterator equal to the iterator returned by function `end`.

```
if (s1.find(10) != s1.end()) {
 cout << "s1 contains 10\n";
}
```

Listing 4 Searching for an item in a set

The STL `set` container can be used in conjunction with the STL generic algorithms. Four of the STL generic algorithms are geared specifically for use with sets. These are functions `set_intersection`, `set_union`, `set_difference`, and `set_symmetric_difference`. The following listing demonstrates the use and effect of these functions. It is important to understand that these four functions can also be used on other STL containers (such as `vectors` and `deques`). They are more efficient, however, when used with sets. This is because the STL `set` container maintains items in sorted order.

```
ostream_iterator<int> out(cout, " ");

// Set intersection
cout << "\nset intersection: ";
set_intersection(s1.begin(), s1.end(),
                 s2.begin(), s2.end(),
                 out);

// Set union
cout << "\nset union: ";
set_union(s1.begin(), s1.end(),
          s2.begin(), s2.end(),
          out);

// Set difference
cout << "\nset difference: ";
set_difference(s1.begin(), s1.end(),
               s2.begin(), s2.end(),
               out);

// Set symmetric difference
```

```
cout << "\nset symmetric difference: ";
set_symmetric_difference(s1.begin(), s1.end(),
                         s2.begin(), s2.end(),
                         out);
```

Listing 5 The set STL algorithms

Maps

The STL `map` container is an associative structure that stores data in key-value pairs. The `map` container is considered associative since it maps, or associates, one piece of data (a key) with another piece of data (a value).

Declaring an object of type `map` involves specifying both the key and value data types. The key is specified with the first template parameter, and the value is specified with a second template parameter. For instance, the following listing declares two `map` objects. The first is a map of type `string` to `string`. The second is a map of type `string` to `int`.

```
// A map of strings to strings
map<string, string> m1;

// A map of strings to ints
map<string, int> m2;
```

Listing 6 Declaring STL `map` objects

Inserting elements into a map requires both the key and value. The following listing inserts two key-value pairs into a map. In this listing, the keys are "apple" and "orange". They map to the values "a small red fruit" and "a small orange fruit", respectively. Note that the version of the `insert` method used here takes only one parameter. To use this method correctly, we have to place the key and value into an object of type `pair<string, string>`. If we were inserting data into a map of type `int` to `double`, for instance, we would have to place our keys and values in objects of type `pair<int, double>` to use the `insert` method.

```
// A map of strings to strings
map<string, string> m1;

// A map of strings to ints
map<string, int> m2;

m1.insert(pair<string, string>("apple", "a small red fruit"));
m1.insert(pair<string, string>("orange", "a small orange fruit"));
```

Listing 7 Using the `insert` function

Using the `insert` method of the STL `map` container is an effective but cumbersome method to add data to a map. An alternative mechanism for adding items to a map is the overloaded double bracket operator. The following listing uses this operator to insert an item into a map.

```
m1["banana"] = "a long yellow fruit";
```

Listing 8 A second way to add items to a map

The overloaded double bracket operator also provides a way to access values based on their keys.

```
cout << m1["apple"] << endl;
cout << m1["orange"] << endl;
cout << m1["banana"] << endl;
```

Listing 9 Accessing items in a map

Similar to the `set` container, the `map` container provides `size`, `find`, and `count` member functions. A programmer can also iterate through the items stored in a map. The implementation of a map iteration warrants examination since it is slightly more complex than a standard container iteration.

```
map<string, string>::iterator it = m1.begin();
for ( ; it != m1.end(); it++) {

 cout << it->first << ": " << it->second << endl;
}
```

Listing 10 Traversing the items in a map

The above listing iterates through the elements of the map object `m1`. The important aspect to notice is how the key and value of each object is accessed. Internally, a map stores the key and value for each data item in an object of type `pair<key, value>`. Therefore, an iterator pointing to an item in a map points to an object of the same `pair<key, value>` type. To access an item's key through an iterator, one must dereference the iterator and then use the `pair` object's `first` data member. The data member `second` provides access to the value.

Keys stored in a `map` must be unique. Put another way, a key can only map to one value. There is another STL container, a `multimap`, which allows keys to map to more than one value.

Priority Queues

The STL `priority_queue` adapter provides to programmers an interface suitable for as a priority queue. As their name suggests, priority queues behave similar to regular queues. Priority queues, however, allow access to only the item with the highest priority.

```
#include <iostream>
#include <cstdlib>
#include <queue>

using namespace std;

int main(int argc, char* argv[]) {

 priority_queue<int> pq;

 pq.push(1);
 pq.push(4);
 pq.push(2);

 cout << pq.top() << endl; // outputs '4'
 pq.pop();
 cout << pq.top() << endl; // outputs '2'
 pq.pop();
 cout << pq.top() << endl; // outputs '1'

 cout << pq.size() << endl; // outputs '1'
```

```
22
23   return EXIT_SUCCESS;
24  }
```

Listing 11 Basic use of the `priority_queue`

The `priority_queue` adapter requires some mechanism to determine the relative priority of elements. By default, the `priority_queue` adapter uses the `<` operator of the elements to determine priority. Many data types and classes define this operator, but many classes, especially user-defined classes, do not. To use a `priority_queue` with objects that define the < operator is straightforward. The above listing does exactly this. As long as the semantics of the supplied `<` make sense for the application at hand, the implementation is relatively painless.

There is an alternative mechanism that a programmer can implement to use a `priority_queue` with objects that do not provide a `<` operator. This mechanism involves providing the `priority_queue` adapter with a compare function. This compare function dictates how the `priority_queue` adapter should determine the priority of the stored items.

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <string>
4  #include <queue>
5
6  using namespace std;
7
8  class StringCompare {
9  public:
10   bool operator()(const string& s1, const string& s2) {
11
12       if (s1.length() < s2.length()) {
13           return true;
14       }
15       else {
16           return false;
17       }
18   }
19 };
20
21 int main(int argc, char* argv[]) {
22
23   // Create a priority queue that assigns longer
24   // strings a higher priority.
25   priority_queue<string, vector<string>, StringCompare> pq;
26
27   pq.push("small string");
28   pq.push("a slightly longer string");
29   pq.push("another small string");
30
31   cout << pq.top() << endl;
32   pq.pop();
33
34   cout << pq.top() << endl;
35   pq.pop();
36
37   cout << pq.top() << endl;
38
```

```
39   return EXIT_SUCCESS;
40 }
```

Listing 12 A user defined compare function

Supplying a compare function to the `priority_queue` is also advantageous when an existing `<` operator is not suitable for use in determining priority. The above listing actually demonstrates this concept since class `string` does provide a `<` operator. If the listing did not supply the user-defined compare function, the priority queue would use the `<` operator. This would cause the program to assign the highest priority to the string that alphabetically precedes the other strings.

5.2 Graphs

With this module, the course introduces graphs and graph algorithms.

Readings:

- **Required:**

  > **Weiss, chapter 15.**

  Remark:

  > **Remember that this book supplements the course's online material. You will be asked questions based on this material.**

- **Required:**

  > **Schildt, chapters 33 through 37. Remark: Remember that this book serves as a general reference to the C++ language, not a course textbook. Therefore, you should browse through the assigned sections in order to get a sense of what they have to offer and where and how they treat important topics. Do not study the sections assigned in this book as you would assignments from a textbook: your goal here should be familiarity, not mastery.**

5.2.1 Introduction to Graphs

Graphs in the Real World

Many rich examples of graphs exist in the real world. You may be using one such example, the Internet, right now. The interconnected networks that make up the Internet are a common example of a graph. The computers form the nodes of the graph. Since not every machine is connected to every other machine, the links between machines also are part of the graph. Each such link between a pair of computers may be directed or undirected, depending on whether messages can be sent in only one direction or in both.

A similar example is traffic networks, such as railroad links between stations, or airplane flights connecting cities. The nodes here are railroad stations and cities, and the links are given by the rails and the flights.

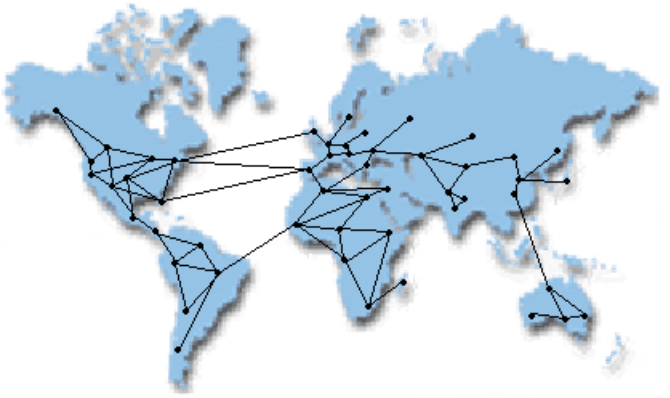

Figure 1 Flights connecting cities as a graph

Lastly, consider a company's organizational chart. In this case, the nodes are the employees, and links indicate the supervisor-subordinate relationship. You may be tempted to think that this type of hierarchy is actually a tree, but we have to consider the fact that an employee may have more than one supervisor. For example, in the following organizational chart the "QA Manager" reports to both the "V.P. of Sales" and the "V.P. of Technology." This prohibits a tree-based representation.

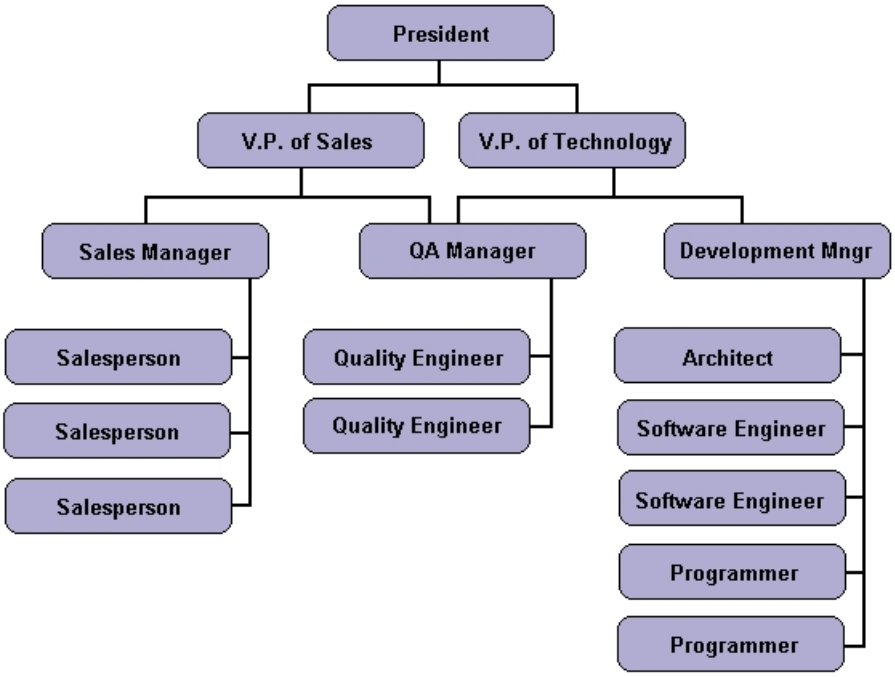

Figure 2 An organizational chart as a graph

Graphs Defined

A directed graph consists of a set `V` of vertices or nodes and a set `E` of edges or arcs. Each edge has a source node and a target node, and the edge can be traversed only from source to target.

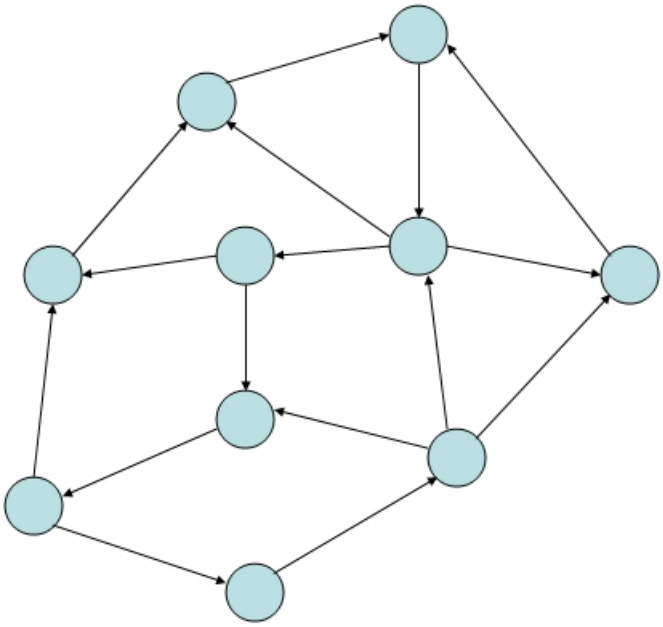

Figure 3 A directed graph

Many of the basic questions one can ask about a graph revolve around path existence. A path is any sequence of vertices connected by edges. The number of edges in a path is the length of the path.

Given two nodes `s` and `t` (the source and the target), we would like to determine whether there is a path from `s` to `t`. This is the single pair reachability problem. Closely related is the all pairs reachability problem, which asks whether there is a path between any two vertices in the graph. In terms of a computer network, this translates into the ability to send a message from any machine to any other machine (assuming some suitable forwarding mechanism).

For the sake of completeness, we mention that one often deals with undirected graphs where each edge can be traversed in either direction. An undirected edge can always be thought of as a pair of directed edges, so we will focus on the directed case.

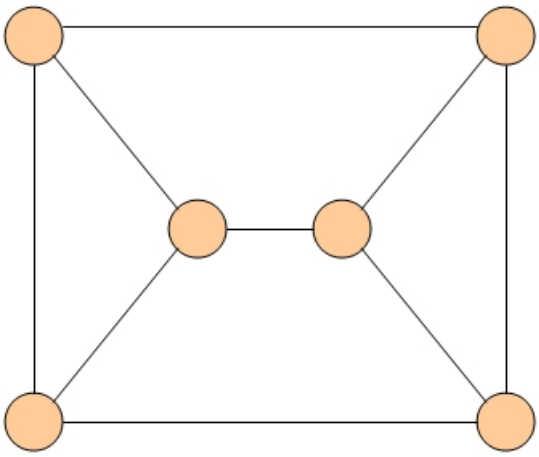

Figure 4 An undirected graph

Perhaps the most appealing feature of graphs is that they can be rendered visually, and pictures of graphs are often helpful in motivating and describing important properties. Graph algorithms usually are best understood in conjunction with a little drawing that shows what the algorithm does.

5.2.2 Fundamental Graph Algorithms

- Breadth-First Search
  - The Concept
  - An Implementation
- Depth-First Search
  - The Concept
  - An Implementation
- Shortest Path Calculation

Breadth-First Search

The Concept

A breadth-first search is an algorithm that explores the nodes in a graph in order of increasing distance from the starting node. A breadth-first search can be used to answer questions about reachability. In other words, we can perform a breadth-first search to determine which nodes can be reached from a starting node. This is an important application since graphs can contain nodes that are disconnected from other nodes.

Figures 1 through 7 demonstrate a breadth-first search of a simple graph.

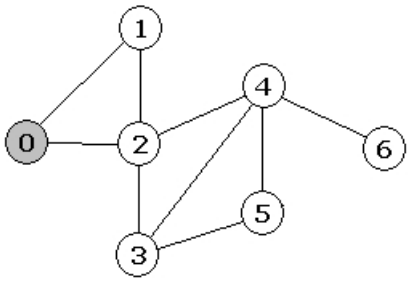

Figure 1 The start of a breadth-first search

The starting node in the breadth-first exploration is highlighted in Figure 1. The algorithm begins by examining each of the outgoing edges of this start node. New nodes are discovered when the algorithm examines these outgoing edges. These "discovered" nodes are placed into a queue. The breadth-first search algorithm then examines each discovered node in turn to determine if new nodes can be reached from the discovered node.

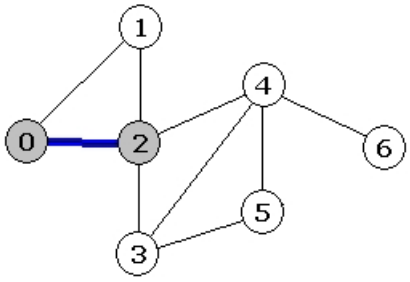

Figure 2 Node 2 discovered

In the figure above, we see that the algorithm has discovered node `2` by examining one of the edges connected to node `0`. The figure shades node `2` to denote that the node has been discovered. Once a node has been discovered, it is placed into a queue of nodes that have been discovered, but not explored. The algorithm then finishes exploring the starting node by examining its remaining edge. This edge leads to node `1`.

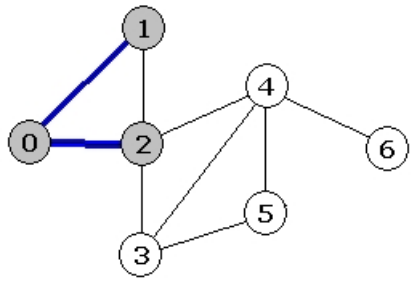

Figure 3 Node 1 discovered

A node is completely explored when all of its edges have been examined. When the exploration of a node is complete, the algorithm selects another node to explore. This selection is not arbitrary. The chosen node is the node at the front of the queue of discovered, but not explored nodes. At this point in this algorithm, this means that the algorithm considers node `2`. In Figures 4 and 5, we see the algorithm explore the edges this node.

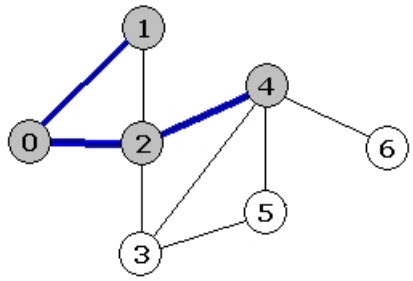

Figure 4 Node 4 discovered

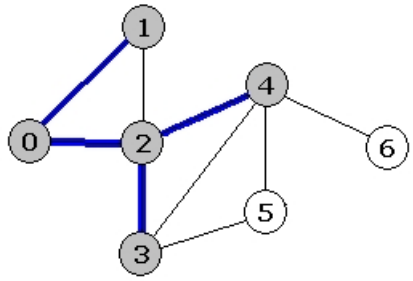

Figure 5 Node 3 discovered

After examining all the edges from node `2`, the algorithm considers the edges of node `1`. The edges connected to node `1`, however, all lead to nodes that have already been discovered. The algorithm moves on and considers the edges connected to node `4`.

Node `4` is connected to four edges. Two of these edges lead to nodes that have already been discovered. The other two edges lead to nodes `5` and `6`. Both of these nodes are marked as "discovered" and placed into the queue. The algorithm then moves on to examine the edges connected to node `3`.

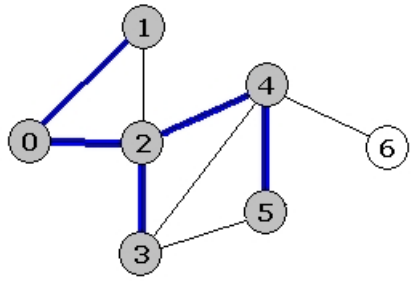

Figure 6 Node 5 discovered

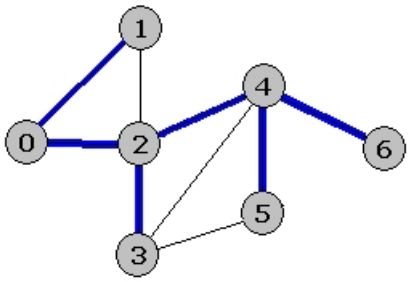

Figure 7 Node 6 discovered

At this point, the algorithm has discovered all the nodes in the graph. We can see this because all the nodes in Figure 7 are shaded. The algorithm, however, does not know this. It continues and examines the edges connected to nodes `3`, `5`, and `6`. All of these edges lead to nodes previously discovered. The algorithm finally terminates when it runs out of nodes that need to be explored.

An Implementation

An implementation of a breadth-first search appears in Listing 1. This program explores the graph in Figures 1 through 7. Because the nodes in this graph are referenced via integers, we can use a vector of lists of type `int` to maintain adjacency lists.

```
1  void bfs(vector< list<int> >& adj_lists, int start_node) {
2
3   queue<int> not_yet_explored;
4   set<int> discovered;
5
6   // Mark the start node as being discovered,
7   // and place it in the queue of nodes to explore
8   not_yet_explored.push(start_node);
9   discovered.insert(start_node);
10
11  while (! not_yet_explored.empty()) {
12
13      // Get a node to explore.
14      int node_to_explore = not_yet_explored.front();
15      not_yet_explored.pop();
16
17      // Examine all the edges of the node
18      list<int>::iterator edges = adj_lists[node_to_explore].begin();
19      for ( ; edges != adj_lists[node_to_explore].end(); edges++) {
20
21          // See if the edge leads to a node that we
22          // have not yet discovered
23          if (discovered.count(*edges) == 0) {
24
25              // We have discovered a new node!
26              // Add this node to the queue of nodes
27              // to explore.
28              discovered.insert(*edges);
29              not_yet_explored.push(*edges);
30
31              cout << "Found " << *edges <<
32              " from " << node_to_explore << endl;
33
34          }
35      }
36  }
```

```
37
38 }
```

Listing 1 Breadth-first search implementation

Depth-First Search

The Concept

A depth-first search is an algorithm that explores the nodes in a graph in reverse order of increasing distance from the starting node. A depth-first search explores nodes deeper into a graph and then works its way back to the nodes that are close to the starting node.

To search deeper into a maze first, a depth-first search algorithm keeps track of the nodes to be explored in a stack rather than a queue. This causes newly discovered nodes to be explored before previously discovered nodes. In a breadth-first search, we explored nodes in the order that we discovered them. In a depth-first search, we explore nodes as we discover them.

Figures 8 through 14 illustrate a depth-first search of the same graph from the breadth-first search example.

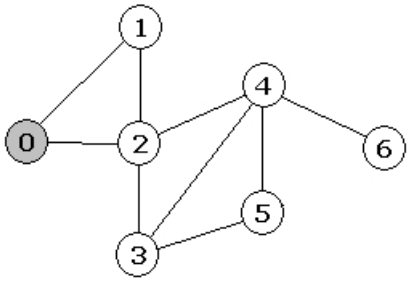

Figure 8 First step of a depth-first search

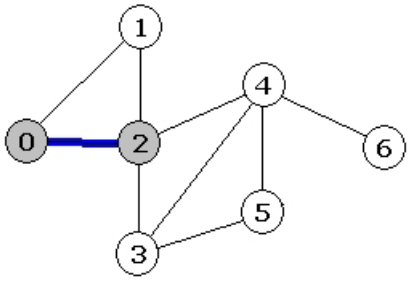

Figure 9 Second step of a depth-first search

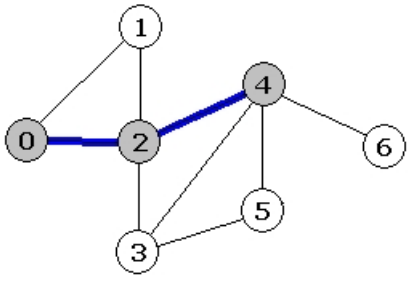

Figure 10 Third step of a depth-first search

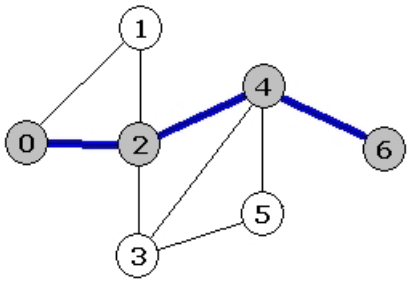

Figure 11 Fourth step of a depth-first search

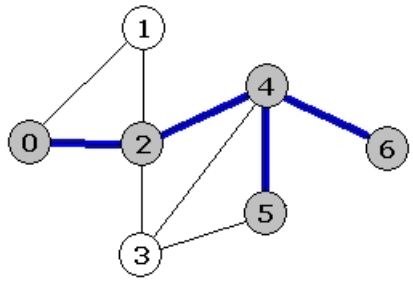

Figure 12 Fifth step of a depth-first search

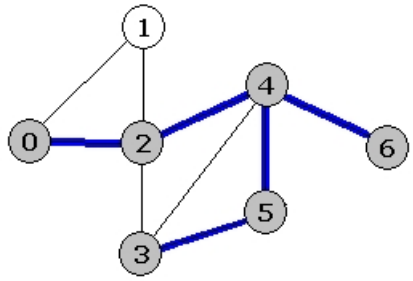

Figure 13 Sixth step of a depth-first search

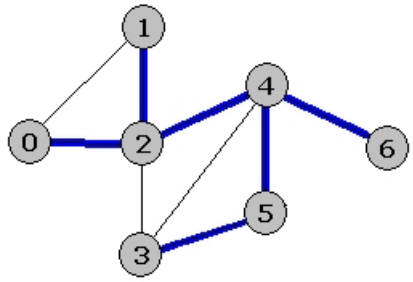

Figure 14 Final step of a depth-first search

An Implementation

A depth-first search implementation differs from a breadth-first search implementation only in the container used to store unexplored nodes. In a breadth-first search, we explore nodes in the order that they are discovered. Thus, a queue is used to maintain the discovered but not explored nodes. A depth-first search algorithm always explores the most recently discovered node. To facilitate this, a stack is used instead of a queue to store the unexplored nodes. In all actuality, though, we can leverage the fact that the run-time system maintains function calls in a stack, and code the routine recursively. This altogether eliminates the need to use a container to store unexplored nodes.

```
1  void dfs_helper(vector< list<int> >& adj_lists, set<int>& discovered,
   int node) {
2
3   // Examine all the edges of the node
4   list<int>::iterator edges = adj_lists[node].begin();
5   for ( ; edges != adj_lists[node].end(); edges++) {
6
7       // See if the edge leads to a node that we
8       // have not yet discovered
9       if (discovered.count(*edges) == 0) {
10
11          // We have discovered a new node!
12          // Add this node to the queue of nodes
13          // to explore.
14          discovered.insert(*edges);
15          cout << "Found " << *edges <<
16          " from " << node << endl;
```

```
            dfs_helper(adj_lists, discovered, *edges);
        }
    }
}

void dfs(vector< list<int> >& adj_lists, int start_node) {

    // Mark the start node as being discovered
    set<int> discovered;
    discovered.insert(start_node);
    dfs_helper(adj_lists, discovered, start_node);
}
```

Listing 2 Depth-first search implementation

Shortest Path Calculation

Shortest path calculations are a family of algorithms that determine the shortest distance between nodes in a graph. Different shortest path algorithms exist because different types of graphs require slightly different approaches. In an unweighted graph, for instance, a simple breadth-first search can be used to calculate the shortest path between nodes. For weighted graphs, different and slightly more complex algorithms must be used. We can use Dijkstra's algorithm if all the weights in a weighted graph are positive. An algorithm known as the Bellman-Ford algorithm solves the shortest path problem for graphs that contain edges with negative weights.