Bare Bones Winamp

EECS 281 – Project 2

Assigned: Tuesday, October 12
Project 2 Due Date: Thursday, November 11, 11:55pm

1 Introduction

Bare Bones Winamp (BBW) is a stripped down MP3 playlist manager. BBW strives to provide an easier way to build multiple playlists by reusing a central store of songs. This central store is loaded into BBW from an input file presented at the command line. This input file lists all of your songs. Using the central store, which is built at start-up, BBW can index music data and allows users to build playlists in a variety of ways.

1.1 Learning Objectives

Often one needs to index the same set of data in several different ways to support a variety of efficient operations. Doing this while maintaining strict interfaces between individual data structures is a technique that is valuable in practice. An important skill that you can take away from this class is the ability to manage multiple, interacting data structures efficiently. This project is designed to give you a real life application of data structures and algorithms that you have been learning in EECS 281.

Now that you have written some of the basic data structures (lists, arrays, dynamic arrays, stacks, queues, etc.), it’s time to use them at a higher level of abstraction. Part of this project will rely on the use of a data structure provided to you by the Standard Template Library (STL).

1.2 Pre-Project Hints

This project is even more complex than the previous project. At first, the spec may seem daunting due to its length and content. However, proper planning and good design can help your group conquer this project. We STRONGLY recommend that groups spend a significant amount of time planning out the project rather than immediately trying to code a solution. Robust designs should exploit the reuse of code amongst the different features of BBW. Another essential task is deciding how to divide up the work in this project (See Section 5.1). Try to break down the various features of BBW into reusable components and functions.

1.3 Summary of Due Dates

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Deliverable(s)</th>
<th>Place to Turn-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weds. October 20</td>
<td>6:00 pm</td>
<td>Group Contract</td>
<td>2420 EECS</td>
</tr>
<tr>
<td>Tues. October 26</td>
<td>6:00 pm</td>
<td>Design Plan</td>
<td>2420 EECS</td>
</tr>
<tr>
<td>Thurs. November 11</td>
<td>11:55 pm</td>
<td>Source Code w/ Makefile</td>
<td>submit281</td>
</tr>
</tbody>
</table>
| Fri. November 12   | 6:00 pm | Testing Analysis
Group Evaluation | 2420 EECS |

2 Major Data Structures and General Functionality

This section provides detailed descriptions of the major data structures that are used in BBW. In this project you must index your song data in several different data structures. Each one of these data structures will perform specific tasks with specified complexities.
2.1 Central Store

On startup, you will read in MP3 specifications from an input text file (see section 3.1 for details). The file will contain the name, artist, length, bitrate and file size of each song. This collection is immutable; it will not and cannot change during the runtime of the application.

Your first step will be storing the song data in the Central Store. You should create a class named `music_file`. The class contains fields for artist, title, length (in seconds), bitrate (in KHz), file size (in KB), and song id. Unique, integer song id’s should be assigned in your startup code in order that the songs are read in. A song id is the index into your central store. In other words, given a song id, you should be able to access its associated `music_file` object in O(1) time.

2.2 Display Store

The central store allows the user to display the entire list of MP3s sorted by the song id. You will also want to be able to display the central store data in a sorted order. Since the central store is immutable, this necessitates another data structure which is called the display store. The display store should not contain copies of `music_file` objects. Instead, it should contain pointers to `music_file` objects. At no time should there be more than one copy of the same `music_file` object in your application. You will instead manipulate only pointers to your objects. Your program will provide a number of options for sorting and displaying the contents of the display store. All of these sorts should preserve the relative ordering of duplicate keys; that is, the display store is ‘stable’. Whenever a request is made to print all available songs, the order is determined by the current state of the display store.

2.3 Searching

Although sorting is useful, this interface is still not particularly useful by itself. The data in your central store must also be searchable.

2.3.1 By Artist or Title: Binary Search Trees

The first data structure allows indexing by artist. This data structure should have O(lg N) access time, where N is the number of songs in the central store. In addition you should be able to traverse this sorted data structure in linear time. This calls for a Binary Search Tree (BST). This will allow the user to search for and display all the songs in a given range of artist names in O(lg N + M * lg M) time, where N is the number of total songs in the central store and M is the number of songs in the given range. All the songs by a particular artist should be printed out sorted by title in alphabetical order. You can deal with the rare case of two different files occurring with both identical artist and title fields as you wish; we will not test this condition.

The next data structure is also a BST, indexed by title instead of by artist, which prints ranges of songs. The complexity requirements for this data structure are identical. When songs with identical titles are printed, the results should be ordered by artist. Just like the artist BST, we will not test the case with identical artists and titles.

All comparisons must be lexicographical when you are inserting or searching in your BSTs. Also, you must eliminate all trailing and leading whitespace (trim the string) before inserting and searching. For example, “Paul Oakenfold” becomes “Paul Oakenfold” (see section 3.5 – Helper Functions).

2.3.2 Searching by Keyword : Hashing

Although these data structures are certainly helpful for searching, they are still not entirely adequate. Say that you had “All Along The Watchtower” by “Jimi Hendrix” in your song database. If you typed “Hendrix Watchtower” into a search prompt the program should be able to infer that you want this song in your search results. You will accomplish this by creating a single hash table that maps a given string, call it string B, to a set of songs in your database. That set of songs is every song in the central store that contains string B in the artist or title field. In the previous example the entry in the keyword hash for “Hendrix” would map to all the songs in the database that contain “Hendrix” in the artist or title. “Watchtower” would map to another set of songs that contain Watchtower in the artist or title. The search result you return is the intersection between these two sets.

The search can contain any number of keywords. Given N keywords, you will have N sets of songs. You must perform an N-way set intersection of these songs. For example:
Song Database
song 0: artist = “Pearl Jam”, title = “Ten – Even Flow”
song 1: artist = “Pearl Jam”, title = “Ten – Alive (Live)”
song 2: artist = “Dream Theater”, title = “Dream and Day Unite – Yste Jam (Live)”
song 3 artist = “Eric Clapton Layla”, title = “Unplugged (LIVE)”

Search: “jam ten live”

Each string in the search field maps to a set of strings (ignore the fact that Live is in between parentheses, this will be explained a little later.)
jam:
song 0: artist = “Pearl Jam”, title = “Ten – Even Flow”
song 1: artist = “Pearl Jam”, title = “Ten – Alive (Live)”
song 2: artist = “Dream Theater”, title = “Even Dream and Day Unite – Yste Jam (Live)”
ten:
song 0: artist = “Pearl Jam”, title = “Ten – Even Flow”
song 1: artist = “Pearl Jam”, title = “Ten – Alive (Live)”
live:
song 1: artist = “Pearl Jam”, title = “Ten – Alive (Live)”
song 2 artist = “Dream Theater”, title = “Dream and Day Unite – Yste Jam (Live)”
song 3: artist = “Eric Clapton Layla”, title = “Unplugged (LIVE)”

Then a set intersection would be performed on these three sets. Since only Pearl Jam’s Alive is in all 3 sets, only that song would be returned.

Search Returns:
file 1: artist = “Pearl Jam” title = “Ten – Alive (Live)”

Given N sets containing M songs, your algorithm must accomplish this set intersection in O(MN) time and make at most one pass through each set. N is the number of keywords and M is the number of songs returned by each keyword.

If you noticed above, “Live” still matched a group of songs even though the actual word in every song was actually “(Live)”. This type of naming convention is typical. We account for this when first processing the data to populate the hash table. Apply the following rules while building up this data structure at startup.

1. Process each artist and title word by word. (A word is defined as a sequence of non-whitespace characters, i.e., a string extracted from a >> operator of a stream).

2. Two passes are made for each artist and title. The first pass hashes every word of the raw string. The second replaces all special characters with spaces, and then hashes every word again.

To make this process more concrete, here is a more thorough example of how the preceding process works.

Artist = “Pearl Jam”
Title = “Ten – Alive (Live)”

Pass 1:
Add entry to “Pearl” set
Add entry to “Jam” set
Add entry to “Ten” set
Add entry to “-” set
Add entry to “Alive” set
Add entry to “(Live)” set

**Convert all special characters to spaces:**
Processed Artist = “Pearl Jam”
Processed Title = “Ten   Alive  Live ” ← all special characters replaced

**Pass 2:**
Add entry to “Pearl” set – redundant
Add entry to “Jam” set – redundant
Add entry to “Ten” set – redundant
Add entry to “Alive” set – redundant
Add entry to “Live” set

You need the function `elim_spec()` to eliminate special characters from a string (see section 3.5 – Helper Functions).

### 2.4 Playlists

You will also maintain an array data structure that tracks an arbitrary number of playlists. You will be able to create, edit, insert songs into, and print playlists. The most basic operation on this data structure is creating an empty playlist. A playlistid is assigned whenever a new playlist is created. The playlistid is the array index of a playlist. Once this is done, you must be able to add songs to it. This is done in the edit playlist menu. The most straightforward way to add a single song is by song id. However, this is not entirely usable. To remedy this, all of the searching data structures will also be available to use from this menu. The playlist editor will utilize the searching features detailed above – code reuse is essential here. In addition, you will provide a command that will allow a user to add the entire contents of the last search to a playlist (see section 3.3.3 for the specific command). Also, no one song can appear more than once in a playlist; your program must enforce this rule. Any insertions that attempt to insert a song that already exists in the playlist are ignored. All insertions and deletions from a given playlist should run in linear time or better.

All playlists are also sortable by the same indices as the display store: artist, title, length, bitrate, file size and song id. Just like sorting in the display store, the playlists are stable. Playlists can be indexed in O(1) time, using their playlistid.

### 2.5 Summary of Structures and Complexities

The following list contains a summary of the structures described in this section.

- **Central Store**
  - O(1) access time given a Song id
  - Container: Array

- **Display Store**
  - O(N²) time sortable by Artist, Title, Length, Bitrate, and Filesize
  - All sorts should preserve the relative ordering of duplicate keys
  - Container: Array

- **Artist and Title Indexes**
  - O(lg N) Access Time
  - Linear traversal for ranges of artists and titles
  - O(lg N + M * lg M) where N is number of songs and M is the number of songs by an artist
  - During searches by artist, songs are printed out in sorted order by song artist and then by song title when there are duplicates. Always sort in alphabetical order.
  - During searches by title, songs should be printed out in sorted order by song title and then by artist when there are duplicates. All sorts sort in alphabetical order.
  - Trim all strings before insertion of search
  - Containers: Binary Search Trees
• Keyword Hash
  - Amortized O(1) Access Time
  - O(MN) cost for set intersection where \( N \) is number of sets and \( M \) is number of elements per set
    - Note that partial credit will be given for \( O(MN \times \log M) \) cost algorithms and \( O(MN) \) that make more than one pass through each set.
    - More formally, given a set of \( n \) sets \( \{S_1, S_2, \ldots, S_n\} \), your algorithm should compute the set \( S = \bigcap_{i=1}^{n} S_i \) in \( O(\sum_{i=1}^{n} |S_i|) \) time where \( |S_i| \) is the size of set \( S_i \).
  - Follow processing rules for insertions into the hash map
  - Container: STL hash_map where key = string and value = set

• Playlists
  - Individual Playlists
    - O(\( N \)) cost for insertion and find where \( N \) is the number of items in the playlist
      - Cost is O(\( N \)) for insertion because duplicate entries are not allowed. Since the playlist could be in many different orders, perform a find and then an insert
      - Cost is O(\( N \)) for find because the playlist can be any order
    - Sortable by Artist, Title, Length, Bitrate, or Filesize
    - All sorts should preserve the relative ordering of duplicate keys
    - Container: Array of music_file*
  - Container: Array of playlists

3 Major Data Structures and General Functionality

This section describes all detailed aspects of the program. This includes all features supported by BBW.

3.1 class music_file

Create a class called music_file to store information about MP3 files. Do not include mutator functions; the central store is immutable! You will use the constructor to create the music_file objects as you read them from the input file. The input file will have the following format:

```
# of songs
<artist> <title> <minutes>:<seconds> <bitrate> <filesize>
<artist> <title> <minutes>:<seconds> <bitrate> <filesize>
```

etc.

For example: songs.txt

```
5
Ben_Folds Brick 4:18 128 4130
Ben_Folds Army 3:23 192 3100
Guster Happier 3:51 128 5300
Maroon_5 She_Will_Be_Loved 4:18 192 5910
Great_Big_Sea Consequence_Free 3:11 192 4380
```

3.2 Main Menu

From the main menu you can execute a number of commands. Output the menu prompt before reading a command from stdin. You should then read a single line of input from stdin, and take action based on that input. The C++ library call getline() is helpful here. You may deal with any error conditions as you wish; we will not test your handling of them. However, we suggest handling them in a robust manner to ease your manual testing. The following list details the commands you must implement in your main menu.
This project specification formally defines the syntax of commands using the following notation. We hope the notation used is fairly intuitive. Curly braces represent a set of words, where words can be a sequence of characters and/or numbers with no intermediate white space. If you see {“word1”,”word2”} in a command specification that means that you will see either “word1” or “word2” in that part of the command. Words or sets of words enclosed in brackets [] represent optional parameters. Arguments enclosed in < >, represent an argument to the command. In all formalizations of command syntax, the helper characters ””{[<,>]} are not to be included by the user.

3.2.1 Create Playlist
- Command: “create”
- Shortcut: “c”
- Exact Syntax {“create”, “c”} <playlistname>
  o <playlistname> is the name of the playlist to be created. <playlistname> may contain spaces.
- Examples: Enter command: create playlist1 -or- Enter command: c playlist1
- Actions taken:
  1. Add an empty playlist to the playlist vector with the <playlistname> as the name. <playlistname> may be multiple words. The playlist id of this new playlist should match its index into the playlist vector.
  2. Go to the playlist menu with the newly created playlist in context.

3.2.2 Edit Playlist
- Command: “edit”
- Shortcut: “e”
- Exact Syntax: {“edit”, “e”} <playlistid>
  o <playlistid> is the playlistid of the playlist to be edited.
- Examples: Enter command: edit playlist1 -or- Enter command: e playlist1
- Action taken: Go to the Playlist Menu with the playlist indexed by playlistid in context.

3.2.3 Print Song
- Command: “song”
- Shortcut: “s”
- Exact Syntax {“song”, “s”} <songid>
  o <songid> is the song id of the song to be played.
- Examples: Enter command: song 6 -or- Enter command: s 6
- Action taken: Display the contents of the music_file for that song.

3.2.4 Print Playlist
- Command: “playlist”
- Shortcut: “y”
- Exact Syntax: {“playlist”, “y”} <playlistid>
  o <playlistid> is the playlistid of the list to be played.
- Examples: Enter command: playlist 2 -or- Enter command: y 2
- Action taken: Display the artist and title for each song in the playlist in its current order.

3.2.5 Print All Playlists
- Command: “print”
- Shortcut: “p”
- Exact Syntax: {“print”, “p”}
- Examples: Enter command: print -or- Enter command: p
- Action taken: Display the artist and title for each song in each playlist in its current order.

3.2.6 Print All Songs
- Command: “all”
- Shortcut: “a”
- Exact Syntax: {“all”, “a”}
- Examples: Enter command: all -or- Enter command: a
• Action taken: Display the contents of the music_file for each song in the central store. Be sure to display the songs in the current order of the display list.

3.2.7 Search Central Store
• Note: this command has many options. Read this section carefully!
• Command: “search”
• Shortcut: “s”
• Exact Syntax: \{“search”, “s”\}[<searchoption>][<listofwords>]
  o <searchoption> is optional
  o <searchoption> is in the set {“title”, “t”, “artist”, “a”}, then <listofwords> - <lower><upper>. The strings <lower> and <upper> represent a range of songs.
  o When <searchoption> is in the set {“keywords”, “k”}, then <listofwords> is a list of strings, all with length greater than 1 and separated by one or more spaces
• Examples: Enter command: search keywords Ben -or- Enter command: s t a d
• Actions Taken:
  1. If <searchoption> is omitted, all music files should be printed. The display store should be sent as the parameters of this function.
  2. When <searchoption> is in the set {“title”, “t”, “artist”, “a”}, use a the BST Search. The strings <lower> and <upper> represent a range of songs. The beginning of the range corresponds to the first element whose key is not less than <lower>. We will call the end of the range E. E is defined as the first element whose key is greater than <upper>. E is not included in the returned range, and is one iteration past the last element in the valid range. When <searchoption> is “title” or “t” the results should be sorted by title and duplicate titles should be sorted by artist. When <searchoption> is “artist” or “a” the results should be sorted by artists and duplicate artists should be sorted by title. All sorts use lexicographical comparisons and sort from least to greatest (a-z). The contents of the music_file for all results of the search should be printed.
  3. If <searchoption> is “keywords” or “k” then perform the keyword hash search. Each word in <listofwords> will map to a set of songs as described in Section 2.3.2. The resulting intersection of these sets should be stored in a container and the contents of the music_file for each song should be printed. This resulting container should be sorted by artist and then by title. The comparisons of artist and title should be lexicographical.

3.2.7 Sort Display Store
• Command: “sort”
• Shortcut: “o”
• Exact Syntax \{“sort”,”o”\}[<sortoption>]
  o <sortoption> is in the set {“a”, “artist”, “t”, “title”, “l”, “length”, “b”, “bitrate”, “f”, “filesize”, “i”, “songid”}
• Examples: Enter command: sort length -or- Enter command: o i
• Actions taken: (Note that all of the following sorts should be stable. All sorts should be from least to greatest.)
  1. If <sortoption> is “a” or “artist” then sort the display store by artist. The comparison should be lexicographical.
  2. If <sortoption> is “t” or “title” then sort the display store by title. The comparison should be lexicographical.
  3. If <sortoption> is “l” or “length” then sort by song length (in seconds).
  4. If <sortoption > is “b” or “bitrate” then sort by bitrate.
  5. If <sortoption> is “f” or “filesize” then sort by filesize.
  6. If <sortoption> is “i” or “songid” then sort by songid.

3.2.8 Quit
• Command: “quit”
• Shortcut: “q”
• Exact Syntax: \{“quit”, “q”\}
• Examples: Enter command: quit -or- Enter command: q
• Actions taken: Clean up and quit the application.

7
3.3 Playlist Menu

Editing a playlist requires the user to navigate to a new menu, the playlist menu. You will notice a great deal of similarity between many of the commands on this menu and the main menu. You should be able to reuse your code. Whenever the user navigates to this menu, a certain playlist is “in context.” This means that any commands executed from this playlist menu should operate on that playlist, the one in context. The following list summarizes the commands that must be implemented from the playlist menu.

3.3.1 Delete Song
- Command: “delete”
- Shortcut: “d”
- Exact Syntax: {“delete”, “d”} <songid>
  - <songid> is songid of a song in the playlists
- Examples: Enter command: delete 7 -or- Enter command: d 4
- Actions taken: Delete the file with <songid> in the playlist. All files with higher indices in the playlist are shifted to an index one lower.

3.3.2 Insert Song
- Command: “insert”
- Shortcut: “i”
- Exact Syntax: {“insert”, “i”} <songid>
  - <songid> is the songid of the file you want to insert
- Examples: Enter command: insert 6 -or- Enter command: i 9
- Actions taken:
  1. If the file already exists in the playlist, ignore this command.
  2. If the file does not exist, append it to the end of the playlist.

3.3.3 Insert Search
- Command: “insert_search”
- Shortcut: “n”
- Exact Syntax: {“insert_search”, “n”}
- Examples: Enter command: insert_search -or- Enter command: n
- Actions taken:
  1. Insert the contents of the last search made in this context in the order they were printed.
     (a) Whenever a “quit” command is processed, the last search results are cleared.
  2. Each individual insert should have the same semantics as the “insert” command; duplicate files should be ignored and new songs are appended to the playlist
  3. If this command is executed before a search command has been successfully executed in this context, no songs should be inserted.

3.3.4 Print Playlist
- Command: “print”
- Shortcut: “p”
- Exact Syntax: {“print”, “p”}
- Examples: Enter command: print -or- Enter command: p
- Actions Taken: Display the artist and title for each song in the playlist in its current order.

3.3.5 Sort and Search
- Sorts and searches are available from the playlist menu and are implemented as described in 3.3.6 and 3.3.7

3.3.6 Return to Main Menu
- Command: “main”
- Shortcut: “m”
• Exact Syntax (“main”, “m”)
• Examples: Enter command: main or Enter command: m
• Actions Taken:
  1. Exit playlist menu.
  2. Print main menu and prompt for user input.

3.4 Command Line Interface to BBW

BBW will be invoked from the command line with the following syntax:

```
proj2.exe -in <infile>
```

where `<infile>` specifies the text file that will determine the MP3 files in our central store.

3.5 Helper Functions

3.5.1 `void trim(string &)`

Trim takes a reference to a string and removes all leading and trailing spaces and tabs.

3.5.2 `void elim_spec(string &)`

This function replaces all “special characters” with spaces. You must call this to accurately process the artist and title and fields and insert them into the hash table.

4 STL Usage

You may use Standard Template Library (STL) for hash_map, string, and input/output functionality. However, you may not use the STL versions of BSTs, Vectors, List, Stacks or Queues. Doing so would result in a loss of points.

5 Project Logistics

5.1 Dividing Work

As mentioned in the introduction, the success of this project depends on effectively dividing up the work while also reusing code. The work in this project can be broken down into components. Give each group member responsibility over one or more of these components. One group member could become the "BST expert" whereas another could concentrate on learning the ins and outs of hash_map.

There are three major roles in this project: Project Manager, Hand-In Czar, and Documentation Specialist. Each of these roles applies to the work you will do outside of coding this project. It is expected that each member of the group will do roughly equivalent amounts of coding.

Based on what we perceive to be a fair distribution of work, we provide the following suggestions for roles and responsibilities within the group. Remember that this list is just a suggestion.

<table>
<thead>
<tr>
<th>Project Manager:</th>
<th>Hand-In Czar:</th>
<th>Documentation Specialist:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-coding Responsibilities:</td>
<td>Non-Coding Responsibilities:</td>
<td>Non-Coding Responsibilities:</td>
</tr>
<tr>
<td>• Arrange meetings</td>
<td>• Consolidate Individual Code into Group Code</td>
<td>• Group Contract and Design Plan</td>
</tr>
<tr>
<td>• Initiate communication</td>
<td>• Only person who can submit on the behalf of the group</td>
<td>• Testing Analysis</td>
</tr>
<tr>
<td>• Track group progress</td>
<td></td>
<td>• Physically hand in project documentation</td>
</tr>
</tbody>
</table>

Coding Responsibilities:

<table>
<thead>
<tr>
<th>Project Manager:</th>
<th>Hand-In Czar:</th>
<th>Documentation Specialist:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Main Menu and Playlist Menu</td>
<td>• Hashing</td>
<td>• Binary Search Trees</td>
</tr>
<tr>
<td>• Simple Sorts</td>
<td>• Help develop menu command functions</td>
<td>• Help develop menu command functions</td>
</tr>
</tbody>
</table>

Based on what we perceive to be a fair distribution of work, we provide the following suggestions for roles and responsibilities within the group. Remember that this list is just a suggestion.
5.2 Handing In Project 3

Each group must designate their “Hand-in Czar” in their Group Contract by Wednesday October 20th. The Hand-in Czar is the only team member who can submit on behalf of the group. The only submission that will be graded is the final submission before the deadline.

To allow fairly unrestricted development of your code, we will allow you to turn in an arbitrary number of files when you submit. Since we must have a uniform build process, you must include a working Makefile for the project. The Makefile should build proj2.exe with the command “make”.

Along with the electronic submission of your project, your group is required to turn in a Group Evaluation Form to the hand-in box in 2420EECS by 6pm on Friday November 12. This form should contain a breakdown of each group member’s percentage of contribution to the project as well as a description of their contribution.

Since each submission erases earlier submissions, be sure that the final submission includes all of the required files. The Hand-In Czar should verify that the response that the code compiles correctly in g++ before submitting it.

5.3 Documentation

5.3.1 Group Contract:

This piece of your documentation should list the names of your group members and their roles in the project. It should detail how your group will communicate and work together. Get the Group Contract Template from the website.

Due: Wednesday, October 20th, 6pm.

5.3.2 Design Plan:

This piece of your documentation should describe the layout of your code in terms of major data structures and functions. Be sure to explain how your group approached the project as well as how project tasks were divided up among group members. We are expecting a description of your set-intersection algorithm. Please do not make documentation of your algorithm “read like code”. Be aware that we will compare your design plan with the actual design of your project. So make a plan and stick to it!

Due: Tuesday, October 26th, 6pm.

5.3.2 Testing Analysis:

This piece of your documentation should describe the ways in which you tested your program. Include a brief description of your testing strategy, including: the objective of your testing and your approach to testing. Then, list the bugs you tested for. This document should be about a page long.

Due: Friday, November 12th, 6pm.

5.4 Grading Criterion

<table>
<thead>
<tr>
<th></th>
<th>Project Implementation</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness</td>
<td>70%</td>
<td>0%</td>
</tr>
<tr>
<td>Style and Design</td>
<td>15%</td>
<td>-5% if not received</td>
</tr>
<tr>
<td>Total</td>
<td>85%</td>
<td>15%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.5   **Sample Runs**

We will be providing sample runs via the course homepage.

6   **Reference Section**

This section will provide you with references that we recommend. As always, google.com is an invaluable tool for finding references.

6.1   **Book List**

- The C++ Standard Template Library: A Tutorial and Reference – Nicholi Josuttis
  - We really recommend this one!
- The C++ Programming Language Special Edition – Bjarne Stroustrup
- Effective STL – Scott Myers
- Introduction to Algorithms – Cormen et. Al.

6.2   **WWW Resources**

- Standard Template Library Programmer’s Guide
- A Modest STL Tutorial
- Murnit’s STL Newbie Guide
  - [http://www.nanotech.wisc.edu/~khan/software/stl/STL.newbie.html](http://www.nanotech.wisc.edu/~khan/software/stl/STL.newbie.html)

7   **Specification’s Revision History**

Currently no revisions.