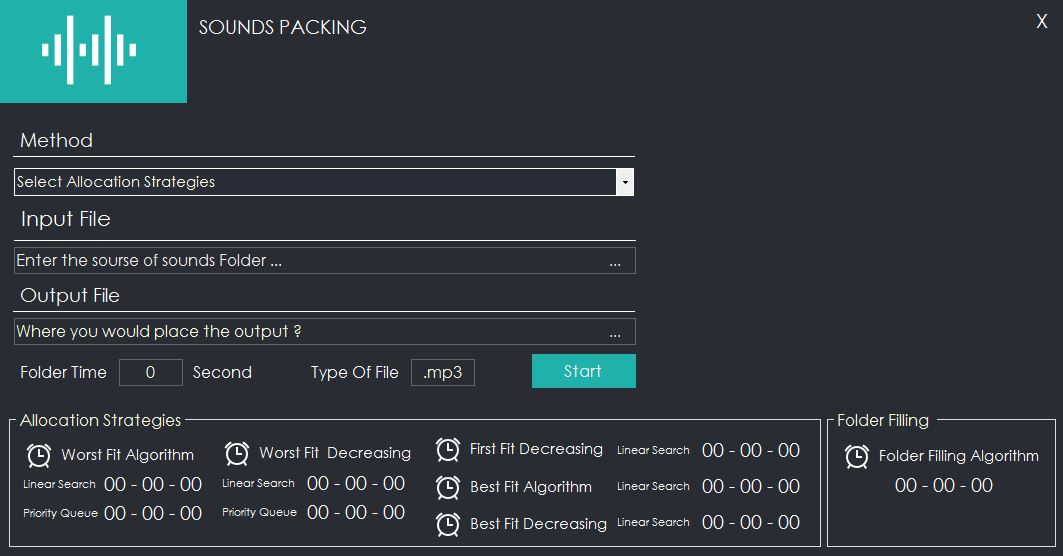
Sound Packing Project Documentation:



5

4

3

2

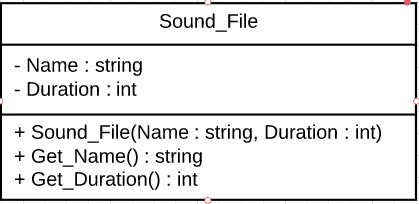
1

1. Dropdown list to select what type of allocation strategy to use:
   1. **Worst Fit Algorithm (Linear Search)**
   2. **Worst Fit Algorithm (Priority Queue)**
   3. **Worst Fit Decreasing Algorithm (Linear Search)**
   4. **First Fit Decreasing Algorithm (Priority Queue)**
   5. **First fit Decreasing**
   6. **Folder Filling Algorithm**
2. The location of the file to be sorted
3. The location of that the files will be generated
4. Desired time of each folder

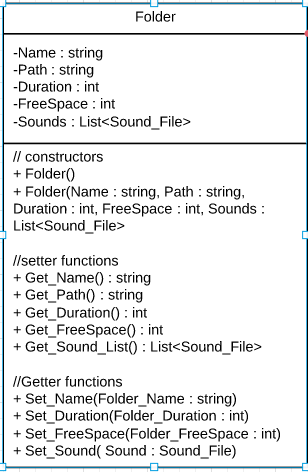
The project is divided into folders for clarity, with each class in a different file. The code is also arranged into functions for clarity and reusability, the code is fully commented for ease of navigation. List of folders:

* Class
* Main Form

**Starting with the class folder, which comprises of 4 Classes, each of which will be discussed in detail:**

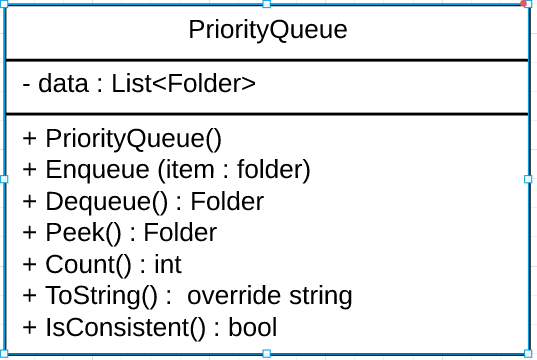
1. The sound file class that represent the audio file. It contains a default constructor, a parametrized constructor and getter function for its attributes.

All functions take constant time [O(1)] to perform its task.



1. The second class represent the folder. It contains a default constructor, a parametrized constructor, setter and getter functions for its attributes.

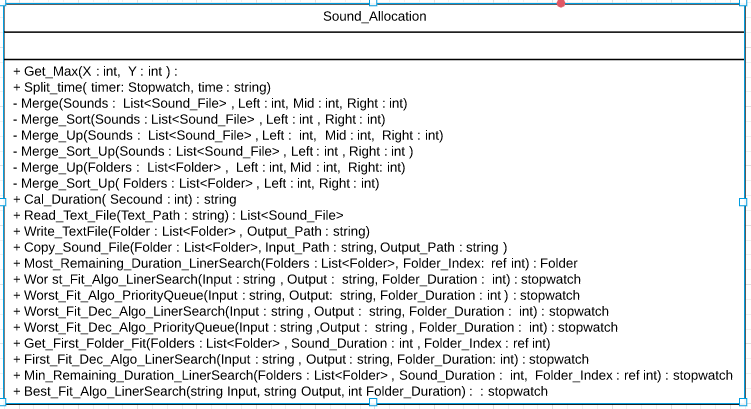
All functions take constant time [O(1)] to perform its task.



1. The 3rd class is the Priority Queue class that has the implementation of Priority Queue using lists. Adding and removing form the list takes linear time of O(Log(N)).

Now we will discuss the complexity of some of the functions in the Priority queue along with the pseudo code:

|  |  |  |
| --- | --- | --- |
| **Function** | **Code** | **Complexity** |
| Constructor: | Constructor PriorityQueue  {  data = new List of Folder  } | O(1) |
| Enqueue -> adding an item to the list. | Func Enqueue(Folder item)  {  data.Add(item);  ci = data.Count - 1 // child index; start at end  while (ci > 0)  {  int pi = (ci - 1) / 2 // parent index  if (data[ci].Get\_FreeSpace() >= data[pi].Get\_FreeSpace())  break;  Folder tmp = data[ci];  data[ci] = data[pi];  data[pi] = tmp;  ci = pi;  }  } | mostly O(1) (look at figure 1)  O(Log(n))  O(1)  O(1)  O(1)  O(1)  O(1) |
| Function Dequeue -> removes and returns the last element is the list. | Func Folder Dequeue()  {  last\_index = data.Count - 1 // last index (before removal)  Folder frontItem = data[0] // fetch the front  data[0] = data[last\_index]  data.RemoveAt(last\_index)  last\_index – 1 // last index (after removal)  int Parent\_Index = 0  while (true)  {  int Child\_Index = Parent\_Index \* 2 + 1  if (Child\_Index > last\_Index)  break  int Right\_Child = Child\_Index + 1  if (Right\_Child <= last\_child && data[right\_child].Get\_FreeSpace()< data[ci].Get\_FreeSpace())  Child\_Index = right\_Child  if (data[Parent\_Child].Get\_FreeSpace() <= data[ci].Get\_FreeSpace())  break  Folder tmp = data[Parent\_Index]  data[Parent\_Index] = data[Child\_Index]  data[Child\_Index] = tmp // swap parent and child  Parent\_Index = Child\_Index  }  return frontItem  } | O(1)  O(1)  O(1)  O(1)  Direct access to remove the last index(no shifting) therefore O(1)  O(1)  Each time we increment by multiplying by 2 therefore O(Log(n))  O(1)  O(1)  O(1)  O(1)  O(1)  O(1) |
| Returns the top element. | Func Folder Peek  {  Folder front\_Item = data[0]  return front\_Item  } | O(1)  O(1) |
| Returns no. of elements | Func int Count  return data.Count | O(1) |

1. The 4th class is the allocation strategies (Sound\_Allocation) class that has all the implementation of the allocations strategies and helping functions.

Now we will discuss the complexity of some of the functions in the Priority queue along with the pseudo code:

|  |  |
| --- | --- |
| **Function/Code segment** | **Time Complexity** |
| Merge Sort | O(NLog(N)) |
| Merge Sort Up for sound | O(NLog(N)) |
| Merge Sort Up for folder | O(MLog(M)) |
| Cal Duration | O(1) |
| Get Max | O(1) |
| Split | 0(N) |
| Parse | O(1) |
| Read text file | 0(N^2) |
| Write text file | O(N^2) |
| Copy Sound file | O(N) |
| Get First Folder Fit | O(M) |
| Most Remaining Duration Liner Search | O(N) |
| Worst Fit Decreasing using Liner Search | O(M\*N) |
| Worst Fit Decreasing using Priority Queue | O(N\*Log(N)) |
| First Fit Decreasing Algorithm Liner Search | O(M\*N) |
| Folder Filling Algorithm | O(N^2) \* O(D) |

**Worst fit Decreasing Algorithm Using Liner Search**:

public stopwatch Worst\_Fit\_Dec\_Algo\_LinerSearch(string Input, string Output, int Folder\_Duration)

{

string Output\_File = Output + @"\OUTPUT"; O(1)

if (!Directory.Exists(Output\_File)) O(1)

Directory.CreateDirectory(Output\_File); O(1)

string StrategiesName\_File = Output\_File+ @"\[2]WorstFit Decreasing"; O(1)

if (!Directory.Exists(StrategiesName\_File)) O(1)

Directory.CreateDirectory(StrategiesName\_File); O(1)

string Input\_TextFile\_Path = Input + @"\AudiosInfo.txt"; O(1)

List<Sound\_File> Sounds = Read\_Text\_File(Input\_TextFile\_Path); O(n^2)

Merge\_Sort(Sounds, 0, Sounds.Count() - 1); O(nlog(n))

List<Folder> Folders = new List<Folder>(); O(1)

Folder Folder = new Folder(); O(1)

Stopwatch s = new Stopwatch(); O(1)

s.Start(); O(1)

// Worst Fit Decreasing using Liner Search Algo starts here

if (Folders.Count() == 0) O(1)

{

Folder.Set\_Name("F1");

Folder.Set\_Duration(0);

Folder.Set\_FreeSpace(Folder\_Duration);

Folders.Add(Folder);

}

for (int i = 0; i < Sounds.Count(); i++) O(n)

{

int Folder\_Index = 0; O(1)

Folder Max\_Folder = Most\_Remaining\_Duration\_LinerSearch(Folders, ref Folder\_Index); O(m)

if (Max\_Folder.Get\_FreeSpace() >= Sounds[i].Get\_Duration()) O(1)

{

int New\_Folder\_Duration = Max\_Folder.Get\_Duration() + Sounds[i].Get\_Duration(); O(1)

int New\_Folder\_FreeSpace = Max\_Folder.Get\_FreeSpace() - Sounds[i].Get\_Duration(); O(1)

Max\_Folder.Set\_Sound(Sounds[i]); O(1)

Max\_Folder.Set\_Duration(New\_Folder\_Duration); O(1)

Max\_Folder.Set\_FreeSpace(New\_Folder\_FreeSpace); O(1)

Folders.RemoveAt(Folder\_Index); O(1)

Folders.Add(Max\_Folder); O(1)

}

else

{

Folder New\_Folder = new Folder(); O(1)

New\_Folder.Set\_Name("F" + (Folders.Count() + 1) + ""); O(1)

New\_Folder.Set\_Duration(Sounds[i].Get\_Duration()); O(1)

New\_Folder.Set\_FreeSpace(Folder\_Duration - Sounds[i].Get\_Duration()); O(1)

New\_Folder.Set\_Sound(Sounds[i]); O(1)

Folders.Add(New\_Folder); O(1)

}

}

// Worst Fit Decreasing using Liner Search Algo ends here

s.Stop(); O(1)

string Output\_TextFile = StrategiesName\_File; O(1)

Write\_TextFile(Folders, Output\_TextFile); O(N^2)

string Output\_SoundFile = StrategiesName\_File; O(1)

string Input\_SoundFile = Input + @"\Audios"; O(1)

Copy\_Sound\_File(Folders, Input\_SoundFile, Output\_SoundFile); O(n)

return s; O(1)

}

**Total time complexity of the algorithm => O(M\*N) + O(1)**

**=> O(M\*N)**

**Worst Fit Decreasing Algorithm Using Priority Queue :**

public Stopwatch Worst\_Fit\_Dec\_Algo\_PriorityQueue(string Input,string Output, int Folder\_Duration)

{

string Output\_File = Output + @"\OUTPUT"; O(1)

if (!Directory.Exists(Output\_File)) O(1)

Directory.CreateDirectory(Output\_File); O(1)

string StrategiesName\_File = Output\_File + @"\[2]WorstFit Decreasing"; O(1)

if (!Directory.Exists(StrategiesName\_File)) O(1)

Directory.CreateDirectory(StrategiesName\_File); O(1)

string Input\_TextFile\_Path = Input + @"\AudiosInfo.txt"; O(1)

List<Sound\_File> Sounds = Read\_Text\_File(Input\_TextFile\_Path); O(n^2)

Merge\_Sort(Sounds, 0, Sounds.Count() - 1); O(N\*Log(N))

PriorityQueue Folders = new PriorityQueue(); O(1)

Folder Folder = new Folder(); O(1)

Stopwatch s = new Stopwatch(); O(1)

s.Start(); O(1)

// Worst Fit Decreasing using Priority Queue starts here

if (Folders.Count() == 0) O(1)

{

Folder.Set\_Name("F1"); O(1)

Folder.Set\_Duration(0); O(1)

int New\_Free\_Space = (-1) \* Folder\_Duration; O(1)

Folder.Set\_FreeSpace(New\_Free\_Space); O(1)

Folders.Enqueue(Folder); O(1)

}

for (int i = 0; i < Sounds.Count(); i++) O(N)

{

Folder Max\_Folder = Folders.Peek(); O(1)

if ((Max\_Folder.Get\_FreeSpace()) \* (-1) >= Sounds[i].Get\_Duration()) O(1)

{

int New\_Folder\_Duration = Max\_Folder.Get\_Duration() + Sounds[i].Get\_Duration(); O(1)

int New\_Folder\_FreeSpace = (Max\_Folder.Get\_FreeSpace() \* (-1)) - Sounds[i].Get\_Duration(); O(1)

Max\_Folder.Set\_Sound(Sounds[i]); O(1)

Max\_Folder.Set\_Duration(New\_Folder\_Duration); O(1)

int New\_Free\_Space = ((-1) \* New\_Folder\_FreeSpace); O(1)

Max\_Folder.Set\_FreeSpace(New\_Free\_Space); O(1)

Folders.Dequeue(); O(Log(M))

Folders.Enqueue(Max\_Folder); O(Log(M))

}

else

{

Folder New\_Folder = new Folder(); O(1)

New\_Folder.Set\_Name("F" + (Folders.Count() + 1) + ""); O(1)

New\_Folder.Set\_Duration(Sounds[i].Get\_Duration()); O(1)

int New\_Free\_Space = (Folder\_Duration - Sounds[i].Get\_Duration()) \* (-1); O(1)

New\_Folder.Set\_FreeSpace(New\_Free\_Space); O(1)

New\_Folder.Set\_Sound(Sounds[i]); O(1)

Folders.Enqueue(New\_Folder); O(1)

}

}

// Worst Fit Decreasing using Priority Queue ends here

s.stop();

int Size = Folders.Count(); O(1)

List<Folder> List\_Folders = new List<Folder>(); O(1)

for (int i = 0; i < Size; i++) O(N)

{

Folder folder = Folders.Dequeue(); O(Log(M))

int FreeSpace = (folder.Get\_FreeSpace() \* (-1)); O(1)

folder.Set\_FreeSpace(FreeSpace); O(1)

List\_Folders.Add(folder); O(1)

}

string Output\_TextFile = StrategiesName\_File; O(1)

Write\_TextFile(List\_Folders, Output\_TextFile); O(N^2)

string Output\_SoundFile = StrategiesName\_File; O(1)

string Input\_SoundFile = Input + @"\Audios"; O(1)

Copy\_Sound\_File(List\_Folders, Input\_SoundFile, Output\_SoundFile); O(N)

Return s;

}

**Total time complexity of the algorithm => O(N\*Log(N)) + O(1)**

**=> O(N\*Log(N))**

**First Fit Decreasing Algorithm Using Liner Search:**

public Stopwatch First\_Fit\_Dec\_Algo\_LinerSearch(string Input, string Output, int Folder\_Duration)

{

string Output\_File = Output + @"\OUTPUT"; O(1)

if (!Directory.Exists(Output\_File)) O(1)

Directory.CreateDirectory(Output\_File); O(1)

string StrategiesName\_File = Output\_File + @"\[3] Firstfit Decreasing"; O(1)

if (!Directory.Exists(StrategiesName\_File)) O(1)

Directory.CreateDirectory(StrategiesName\_File); O(1)

string Input\_TextFile\_Path = Input + @"\AudiosInfo.txt"; O(1)

List<Sound\_File> Sounds = Read\_Text\_File(Input\_TextFile\_Path); O(n^2)

Merge\_Sort(Sounds, 0, Sounds.Count() - 1); O(N\*Log(N))

List<Folder> Folders = new List<Folder>(); O(1)

Folder Folder = new Folder(); O(1)

if (Folders.Count() == 0) O(1)

{

Folder.Set\_Name("F1"); O(1)

Folder.Set\_Duration(0); O(1)

Folder.Set\_FreeSpace(Folder\_Duration); O(1)

Folders.Add(Folder); O(1)

}

Stopwatch s = new Stopwatch(); O(1)

s.Start(); O(1) // first Fit Decreasing using Liner Search ends here

for (int i = 0; i < Sounds.Count(); i++) O(N)

{

int Folder\_Index = -1; O(1)

Get\_First\_Folder\_Fit(Folders, Sounds[i].Get\_Duration(), ref Folder\_Index); O(M)

if (Folder\_Index != -1) O(1)

{ O(1)

Folder = Folders[Folder\_Index]; O(1)

int New\_Folder\_Duration = Folder.Get\_Duration() + Sounds[i].Get\_Duration(); O(1)

int New\_Folder\_FreeSpace = Folder.Get\_FreeSpace() - Sounds[i].Get\_Duration(); O(1)

Folder.Set\_Sound(Sounds[i]); O(1)

Folder.Set\_Duration(New\_Folder\_Duration); O(1)

Folder.Set\_FreeSpace(New\_Folder\_FreeSpace); O(1)

Folders.RemoveAt(Folder\_Index); O(1)

Folders.Add(Folder); O(1)

}

else

{

Folder New\_Folder = new Folder(); O(1)

New\_Folder.Set\_Name("F" + (Folders.Count() + 1) + ""); O(1)

New\_Folder.Set\_Duration(Sounds[i].Get\_Duration()); O(1)

New\_Folder.Set\_FreeSpace(Folder\_Duration - Sounds[i].Get\_Duration()); O(1)

New\_Folder.Set\_Sound(Sounds[i]); O(1)

Folders.Add(New\_Folder); O(1)

}

}

s.stop(); O(1)

string Output\_TextFile = StrategiesName\_File; O(1)

Write\_TextFile(Folders, Output\_TextFile); O(N^2)

string Output\_SoundFile = StrategiesName\_File; O(1)

string Input\_SoundFile = Input + @"\Audios"; O(1)

Copy\_Sound\_File(Folders, Input\_SoundFile, Output\_SoundFile); O(N)

return s;

}

**Total time complexity of the algorithm => O(M\*N) + O(1)**

**=> O(M\*N)**

**Folder Filling Algorithm:**

string Output\_File = Output + @"\OUTPUT"; o(1)

if (!Directory.Exists(Output\_File)) o(1)

{

Directory.CreateDirectory(Output\_File); o(1)

}

string StrategiesName\_File = Output\_File + @"\[6] FolderFilling"; o(1)

if (!Directory.Exists(StrategiesName\_File)) o(1)

{

Directory.CreateDirectory(StrategiesName\_File); o(1)

}

string Input\_TextFile\_Path = Input + @"\AudiosInfo.txt"; o(1)

List<Sound\_File> Sounds = Read\_Text\_File(Input\_TextFile\_Path); o(N^2)

Timer = new Stopwatch(); o(1)

Timer.Start(); o(1)

List<Folder> Folders = new List<Folder>(); o(1)

int File\_Name = 0; o(1)

while(Sounds.Count() > 0) o(1)

{

List<int> Sound\_INT = new List<int>();

for (int j = 0; j < Sounds.Count(); j++) o(N)

{

Sound\_INT.Add(Sounds[j].Get\_Duration()); o(1)

}

int[,] DB = new int[Sounds.Count() + 1, Folder\_Duration + 1]; o(1)

int i, w; o(1)

int Sound\_Wight = Folder\_Duration; o(1)

int Number\_Sound = Sounds.Count(); o(1)

for (i = 0; i <= Number\_Sound; i++) o(N)

{

for (w = 0; w <= Sound\_Wight; w++) o(M)

{

if (i == 0 || w == 0) o(1)

DB[i, w] = 0;

else if (Sound\_INT[i - 1] <= w) o(1)

DB[i, w] = Get\_Max(Sound\_INT[i - 1] + DB[i - 1, w - Sound\_INT[i - 1]], DB[i - 1, w]); o(1)

else

DB[i, w] = DB[i - 1, w]; o(1)

}

}

i = Number\_Sound; o(1)

w = Sound\_Wight; o(1)

Folder Folder = new Folder(); o(1)

Folder.Set\_FreeSpace(Folder\_Duration);

Folder.Set\_Duration(0); o(1)

Folder.Set\_Name("F" + (File\_Name + 1) + ""); o(1)

List<int> Remove = new List<int>(); o(1)

while (w != 0 && i != 0) O(i) => O(n)

{

if (Sound\_INT[i - 1] > w || (DB[i - 1, w] > Sound\_INT[i - 1] + DB[i - 1, w -Sound\_INT[i - 1]])) o(1)

i--;

else

{

Folder.Set\_Duration(Folder.Get\_Duration() + Sounds[i - 1].Get\_Duration()); o(1)

Folder.Set\_FreeSpace(Folder.Get\_FreeSpace() - Sounds[i - 1].Get\_Duration()); o(1)

Folder.Set\_Sound(Sounds[i - 1]); o(1)

Remove.Add(i - 1); o(1)

w = w - Sound\_INT[i - 1]; o(1)

1. O(1)

}

}

Folders.Add(Folder);

for (int y = 0; y < Remove.Count(); y++) o(n)

{

Sounds.RemoveAt(Remove[y]); o(1)

Sound\_INT.RemoveAt(Remove[y]); o(1)

}

File\_Name++; o(1)

}

Timer.Stop(); o(1)

string Output\_TextFile = StrategiesName\_File;

Write\_TextFile(Folders, Output\_TextFile); o(1)

string Output\_SoundFile = StrategiesName\_File;

string Input\_SoundFile = Input + @"\Audios"; o(1)

Copy\_Sound\_File(Folders, Input\_SoundFile, Output\_SoundFile, file\_type); o(1)

return Timer; o(1)

**O(N^2) \* O(D)**

**Comparison between all methods in term of number of folders and execution time**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Sample (1) | | Sample (2) | | Sample (3) | |
| Sound Allocation | # Folder | Time | # Folder | Time | # Folder | Time |
| Worst Fit | 3 | 5ms | 4 | 4ms | 3 | 5ms |
| Worst Fit (Priority Queue) | 3 | 0ms | 4 | 8ms | 3 | 0ms |
| Worst Fit Decreasing | 3 | 3ms | 3 | 4ms | 2 | 5ms |
| Worst Fit Decreasing (P.Q) | 3 | 0ms | 3 | 4ms | 2 |  |
| First Fit Decreasing | 2 | 1ms | 3 | 1ms | 2 | 1ms |
| Best Fit | 2 | 5ms | 4 | 8ms | 3 | 8ms |
| Best Fit Decreasing | 2 | 0ms | 3 | 0ms | 2 | 0ms |
| Folder Filling | 2 | 42ms | 3 | 5ms | 2 | 3ms |