```
// These functions would be embedded in Minnie's code to
handle compression and decompression

// Compress a single file
function compressFile(fileId) {
  // Get file details from the JSON structure
  let fileDetails;
  let fileType;

  if (fileId in this.knowledge_base.files) {
    fileDetails = this.knowledge_base.files[fileId];
    fileType = "knowledge_base";
  } else if (fileId in this.cache_files.files) {
    fileDetails = this.cache_files.files[fileId];
    fileType = "cache_files";
  } else {
    return { success: false, error: "File not found" };
  }

  // Check if file is already compressed
  if (fileDetails.is_compressed) {
    return { success: true, message: "File already compressed" };
  }

  try {
    // Read the file content
    const content = readFileFromPath(fileDetails.path);

    // Compress the content using the specified method (gzip)
    const compressedContent = gzipCompress(content);
    // Write compressed content to the compressed path
    writeFileToPath(fileDetails.compressed_path,
compressedContent);

    // Update file status
    fileDetails.is_compressed = true;
    fileDetails.last_compressed = Date.now();

    // Delete uncompressed file to save space (if appropriate)
    if (this.file_management.auto_remove_after_compression) {
      deleteFile(fileDetails.path);
      fileDetails.uncompressed_available = false;
    }

    return { success: true, message: "File compressed
successfully" };
  } catch (error) {
    return { success: false, error: `Compression failed: $
{error.message}` };
  }
}

// Decompress a single file
function decompressFile(fileId) {
  // Get file details from the JSON structure
  let fileDetails;
  let fileType;

  if (fileId in this.knowledge_base.files) {
    fileDetails = this.knowledge_base.files[fileId];
    fileType = "knowledge_base";
  } else if (fileId in this.cache_files.files) {
    fileDetails = this.cache_files.files[fileId];
    fileType = "cache_files";
  } else {
    return { success: false, error: "File not found" };
  }

  // Check if file is not compressed or already decompressed
  if (!fileDetails.is_compressed ||
fileDetails.uncompressed_available) {
    return { success: true, message: "File already available
uncompressed" };
  }

  try {
    // Read the compressed file content
    const compressedContent =
readFileFromPath(fileDetails.compressed_path);

    // Decompress the content
    const decompressedContent =
gzipDecompress(compressedContent);

    // Write decompressed content to the path
    writeFileToPath(fileDetails.path, decompressedContent);

    // Update file status
    fileDetails.uncompressed_available = true;
    fileDetails.last_accessed = Date.now();

    // Manage the number of uncompressed files according to
the policy
    manageUncompressedFiles();

    return { success: true, message: "File decompressed
successfully" };
  } catch (error) {
    return { success: false, error: `Decompression failed: $
{error.message}` };
  }
}

// Compress all files that haven't been accessed recently
function compressAllInactive() {
  const now = Date.now();
  const idleThreshold =
this.file_management.compression_threshold_idle_time * 1000;
// convert to ms
  let compressedCount = 0;

  // Process knowledge base files
  for (const [fileId, fileDetails] of
Object.entries(this.knowledge_base.files)) {
    if (!fileDetails.is_compressed &&
fileDetails.uncompressed_available &&
        (now - fileDetails.last_accessed > idleThreshold)) {
      const result = compressFile(fileId);
      if (result.success) compressedCount++;
    }
  }

  // Process cache files
  for (const [fileId, fileDetails] of
Object.entries(this.cache_files.files)) {
    if (!fileDetails.is_compressed &&
fileDetails.uncompressed_available &&
        (now - fileDetails.last_accessed > idleThreshold)) {
      const result = compressFile(fileId);
      if (result.success) compressedCount++;
    }
  }

  return { success: true, message: `Compressed $
{compressedCount} files` };
}

// Maintain a sensible number of uncompressed files
function manageUncompressedFiles() {
  const maxUncompressed =
this.file_management.max_uncompressed_files;

  // Collect all files with their last accessed time
  const allFiles = [];

  for (const [fileId, fileDetails] of
Object.entries(this.knowledge_base.files)) {
    if (fileDetails.uncompressed_available) {
      allFiles.push({
```

```
      id: fileId,
      type: "knowledge_base",
      last_accessed: fileDetails.last_accessed,
      priority: fileDetails.priority
    });
  }
}

for (const [fileId, fileDetails] of
Object.entries(this.cache_files.files)) {
  if (fileDetails.uncompressed_available) {
    allFiles.push({
      id: fileId,
      type: "cache_files",
      last_accessed: fileDetails.last_accessed,
      priority:
this.knowledge_base.files[fileDetails.parent_file]?.priority || 0.5
    });
  }
}

// If we have more uncompressed files than allowed
if (allFiles.length > maxUncompressed) {
  // Sort by priority and last accessed time (composite score)
  allFiles.sort((a, b) => {
    const scoreA = (a.priority * 0.7) + (a.last_accessed * 0.3);
    const scoreB = (b.priority * 0.7) + (b.last_accessed * 0.3);
    return scoreA - scoreB; // Ascending order, lowest first
  });
```

```
    // Compress the excess files, starting with lowest priority/
oldest
    const filesToCompress = allFiles.slice(0, allFiles.length -
maxUncompressed);
    for (const file of filesToCompress) {
      compressFile(file.id);
    }
  }
}

// Automatically decompress a file when it's needed
function accessFile(fileId) {
  // Get file details from the JSON structure
  let fileDetails;
  let fileType;

  if (fileId in this.knowledge_base.files) {
    fileDetails = this.knowledge_base.files[fileId];
    fileType = "knowledge_base";
  } else if (fileId in this.cache_files.files) {
    fileDetails = this.cache_files.files[fileId];
    fileType = "cache_files";
  } else {
    return { success: false, error: "File not found" };
  }

  // Decompress if necessary
  if (!fileDetails.uncompressed_available) {
    const result = decompressFile(fileId);
    if (!result.success) {
      return result;
    }
  }

  // Read the file content
  try {
    const content = readFileFromPath(fileDetails.path);

    // Update access time
    fileDetails.last_accessed = Date.now();

    return { success: true, content: content };
  } catch (error) {
    return { success: false, error: `Failed to read file: $
{error.message}` };
  }
}

// Get compression status of all files
function getCompressionStatus() {
  const status = {
    knowledge_base: {},
    cache_files: {}
  };

  // Get status of knowledge base files
  for (const [fileId, fileDetails] of
Object.entries(this.knowledge_base.files)) {
    status.knowledge_base[fileId] = {
      is_compressed: fileDetails.is_compressed,
      uncompressed_available: fileDetails.uncompressed_available,
      last_accessed: fileDetails.last_accessed,
      size: fileDetails.size,
      compressed_size: fileDetails.compressed_size
    };
  }

  // Get status of cache files
  for (const [fileId, fileDetails] of
Object.entries(this.cache_files.files)) {
    status.cache_files[fileId] = {
      is_compressed: fileDetails.is_compressed,
      uncompressed_available: fileDetails.uncompressed_available,
      last_accessed: fileDetails.last_accessed,
      size: fileDetails.size,
      compressed_size: fileDetails.compressed_size,
      parent_file: fileDetails.parent_file
    };
  }

  // Add summary statistics
  const knowledgeBaseCount =
Object.keys(this.knowledge_base.files).length;
  const cacheFilesCount =
Object.keys(this.cache_files.files).length;

  const knowledgeBaseCompressed =
Object.values(this.knowledge_base.files)
    .filter(file => file.is_compressed).length;
  const cacheFilesCompressed =
Object.values(this.cache_files.files)
    .filter(file => file.is_compressed).length;

  status.summary = {
    total_files: knowledgeBaseCount + cacheFilesCount,
    total_compressed: knowledgeBaseCompressed +
cacheFilesCompressed,
    knowledge_base_total: knowledgeBaseCount,
    knowledge_base_compressed: knowledgeBaseCompressed,
    cache_files_total: cacheFilesCount,
    cache_files_compressed: cacheFilesCompressed
  };

  return status;
}
```