CS221-L Data Structures and Algorithms

Assignment # 02

Submitted by: **Shayan Rizwan**

Registration Number: **2024585**

Submitted to: Prof. Ahmar Rashid

Semester: 3rd

Faculty of Computer Science and Engineering

GIK Institute of Engineering Sciences and Technology

CODE:

```
#include <iostream>
using namespace std;

struct SuperBlock {
  int Blockid;
  int startIndex;
  int sizeOfMemoryBlock;
  string data;
  SuperBlock* next; // A pointer to the next superblock in the linked list

   SuperBlock(int start, int sz)
     : Blockid(0), startIndex(start), sizeOfMemoryBlock(sz), data(""), next(nullptr)
{}

};

// Global memory pool - accessible to all functions
char memoryPool[64];

// Global variables for management
SuperBlock* head = NULL;  // Head of linked list
SuperBlock* tail = NULL;
int nextBlockId = 1;         // For generating unique BlockIds

 void initializeMemoryPool() {
     // Each building block represents one character.
     for (int i = 0; i < 64; i++) {
      memoryPool[i] = 'E';
     }
     // At the beginning, all blocks will be marked as 'E' for empty.
  }

  // There is a size of the memory block, and initially the size of the memory block
is 64, which is initialized.

int findAvailableBlock(int size) {
  int currentStart = -1;  // Track start of current free sequence
  int currentLength = 0;  // Track length of current free sequence

  for (int i = 0; i < 64; i++) {
    if (memoryPool[i] == 'E') {
      // If this is the start of a new free sequence
```

```
      if (currentStart == -1) {
        currentStart = i;  // Mark the start
      }
      currentLength++;  // Increase the length of current sequence

      // Check if we've found a large enough block
      if (currentLength >= size) {
        return currentStart;  // Return the start of this sequence
      }
    } else {
      // Hit an occupied block - reset our search
      currentStart = -1;
      currentLength = 0;
    }
  }

  // If we finish the loop without finding a block
  return -1;
}

SuperBlock* Append(int startIndex, int size) {
  if (tail != 0) { // if the linked list is NOT empty;
    tail->next = new SuperBlock(startIndex, size);
    tail = tail->next;
  }
  else head = tail = new SuperBlock(startIndex, size);

  tail->Blockid = nextBlockId;  // Set the Blockid

   // INCREMENT for the next block that will be created
  nextBlockId++;

  return tail;


}

SuperBlock* allocateSuperBlockForString(const std::string &str) {
   // Step 1: Calculate how many blocks needed
  int counter = 0;
  for (int i = 0; i < str.size(); i++) {
    counter++;
  }

  // Step 2: Find available space using findAvailableBlock()
  int startIndex = findAvailableBlock(counter);
  if (startIndex == -1) {
```

```
    cout << "Error: Not enough contiguous memory!" << endl;
    return nullptr;
  }

  // Step 3: "Allocate" by writing string into memoryPool
  for (int i = 0; i < counter; i++) {
    memoryPool[startIndex + i] = str[i];
  }

  // Step 4: Create SuperBlock to track this allocation
  SuperBlock* newBlock = Append(startIndex, counter);
  newBlock->data = str;

  return newBlock;
}

// Function to display the entire memory pool;
// This shows what's actually stored in each of the 64 memory blocks:
void displayMemoryPool() {
  cout << "Memory Pool:" << endl;
  cout << "Index:   ";
  for (int i = 0; i < 64; i++) {
    if (i < 10) cout << " ";  // Alignment for single-digit numbers
    cout << i << " ";
  }
  cout << endl;

  cout << "Data:    ";
  for (int i = 0; i < 64; i++) {
    cout << " " << memoryPool[i] << " ";
  }
  cout << endl;
}

// Function to traverse the linked list and display all the metadata of the
superblock;
void DisplayLinkedList() {
  cout << "Traversing the List: " << endl;
  if (head == NULL) {
    cout << "  No allocated blocks" << endl;
    return;
  }

  SuperBlock* current = head;
  while (current != NULL) {
    int endIndex = current->startIndex + current->sizeOfMemoryBlock - 1;
    cout << "  " << current->Blockid << ": " << current->data
```

```
      << "    Indices: (" << current->startIndex << " -> " << endIndex
      << "), Size: " << current->sizeOfMemoryBlock << endl;
    current = current->next;
  }

}

void displayEverything() {
  cout << "****************************" << endl;
  displayMemoryPool();      // Show physical memory
  cout << endl;
  DisplayLinkedList();      // Show logical organization
  cout << "****************************" << endl;
}

void deallocateSuperBlock(int BlockId) {
  if (head == NULL) {
    cout << "Error: Empty list - no blocks to deallocate" << endl;
    return;
  }

  SuperBlock* current = head;
  SuperBlock* prev = NULL;

  // Search for the block with matching BlockId
  while (current != NULL && current->Blockid != BlockId) {
    prev = current;
    current = current->next;
  }

  // If block not found
  if (current == NULL) {
    cout << "Error: BlockId " << BlockId << " not found" << endl;
    return;
  }

  // Get startIndex BEFORE deleting the node
  int startIndex = current->startIndex;
  int size = current->sizeOfMemoryBlock;

  // FREE THE MEMORY in memoryPool
  for (int i = startIndex; i < startIndex + size; i++) {
    memoryPool[i] = '_';  // or 'E' - mark as free
  }

  // Remove from linked list
  if (prev == NULL) {
```

```
    // Deleting the head node
    head = current->next;
    if (head == NULL) tail = NULL;  // Update tail if list becomes empty
  } else {
    // Deleting a middle or tail node
    prev->next = current->next;
    if (current == tail) tail = prev;  // Update tail if deleting last node
  }

  delete current;
  cout << "Memory deallocated for Super-block id: " << BlockId << endl;
}

// Function 8: Deallocating PART of a superblock
void deallocatePartOfSuperBlock(int BlockId, int partSize) {
  if (head == NULL) {
    cout << "Error: Empty list - no blocks to deallocate" << endl;
    return;
  }

  // Find the super-block with matching BlockId
  SuperBlock* current = head;
  while (current != NULL && current->Blockid != BlockId) {
    current = current->next;
  }

  // If block not found
  if (current == NULL) {
    cout << "Error: BlockId " << BlockId << " not found" << endl;
    return;
  }

  // Check if partSize is valid
  if (partSize <= 0) {
    cout << "Error: partSize must be positive" << endl;
    return;
  }

  if (partSize >= current->sizeOfMemoryBlock) {
    cout << "Error: partSize exceeds or matches super-block size. Use
deallocateSuperBlock instead." << endl;
    return;
  }

  // Free the specified portion from start in memory pool
  int startIndex = current->startIndex;
  for (int i = startIndex; i < startIndex + partSize; i++) {
```

```
    memoryPool[i] = '_';
  }

  // Adjust the super-block's metadata
  current->startIndex = startIndex + partSize;
  current->sizeOfMemoryBlock = current->sizeOfMemoryBlock - partSize;

  // Update the data string (remove the deallocated part from the beginning)
  current->data = current->data.substr(partSize);

  cout << "Part of super-block deallocated: " << partSize << " blocks freed starting
from index " << startIndex << endl;
}




void deallocatePartOfSuperBlockAnywhere(int BlockId, int StartIndex, int partSize) {
  if (head == NULL) {
    cout << "Error: Empty list - no blocks to deallocate" << endl;
    return;
  }

  // Find the super-block with matching BlockId
  SuperBlock* current = head;
  while (current != NULL && current->Blockid != BlockId) {
    current = current->next;
  }

  // If block not found
  if (current == NULL) {
    cout << "Error: BlockId " << BlockId << " not found" << endl;
    return;
  }

  // Validate deallocation bounds
  int blockStart = current->startIndex;
  int blockEnd = blockStart + current->sizeOfMemoryBlock - 1;
  int deallocStart = StartIndex;
  int deallocEnd = StartIndex + partSize - 1;

  // Check if deallocation is within block bounds
  if (deallocStart < blockStart || deallocEnd > blockEnd) {
    cout << "Error: Deallocation exceeds the bounds of the super-block" << endl;
    return;
  }
```

```
  if (partSize <= 0) {
    cout << "Error: partSize must be positive" << endl;
    return;
  }

  // Free the specified portion in memory pool
  for (int i = deallocStart; i <= deallocEnd; i++) {
    memoryPool[i] = '_';
  }

  // Handle three cases based on deallocation position
  if (deallocStart == blockStart) {
    // Case 1: Deallocation from start
    if (partSize == current->sizeOfMemoryBlock) {
      // Entire block is deallocated - remove from linked list
      deallocateSuperBlock(BlockId);
    } else {
      // Partial deallocation from start - adjust current block
      current->startIndex = deallocEnd + 1;
      current->sizeOfMemoryBlock = current->sizeOfMemoryBlock - partSize;
      current->data = current->data.substr(partSize);
    }
  } else if (deallocEnd == blockEnd) {
    // Case 2: Deallocation from end - just shrink current block
    current->sizeOfMemoryBlock = current->sizeOfMemoryBlock - partSize;
    current->data = current->data.substr(0, current->data.length() - partSize);
  } else {
    // Case 3: Deallocation from middle - split into two blocks
    int firstPartSize = deallocStart - blockStart;
    int secondPartSize = blockEnd - deallocEnd;


    string secondPartData = current->data.substr(deallocStart - blockStart +
partSize);

    // Adjust current block to be the first part
    current->sizeOfMemoryBlock = firstPartSize;
    current->data = current->data.substr(0, firstPartSize);

    // Create new block for the second part
    int newBlockStart = deallocEnd + 1;

    SuperBlock* newBlock = new SuperBlock(newBlockStart, secondPartSize);
    newBlock->data = secondPartData;
    newBlock->Blockid = nextBlockId++;
```

```
    // Insert new block after current block in linked list
    newBlock->next = current->next;
    current->next = newBlock;
    if (tail == current) tail = newBlock;  // Update tail if needed
  }

  cout << "Part of super-block deallocated: " << partSize << " blocks freed starting
from index " << StartIndex << endl;
}

void userMemoryManagementInterface() {
  int choice;
  string inputString;
  int blockId, partSize, startIndex;

  initializeMemoryPool();  // Initialize memory pool at start

  do {
    // Display menu
    cout << "\n=== Memory Management System ===" << endl;
    cout << "1. Allocate memory for a string" << endl;
    cout << "2. Deallocate entire super-block" << endl;
    cout << "3. Deallocate part from start of super-block" << endl;
    cout << "4. Deallocate part from anywhere in super-block" << endl;
    cout << "5. Display current status" << endl;
    cout << "6. Exit" << endl;
    cout << "Enter your choice (1-6): ";
    cin >> choice;

    switch(choice) {
      case 1:
        // Allocate memory for string
        cout << "Enter string to allocate: ";
        cin.ignore(); // Clear input buffer
        getline(cin, inputString);
        allocateSuperBlockForString(inputString);
        displayEverything(); // Show updated state
        break;

      case 2:
        // Deallocate entire block
        cout << "Enter BlockId to deallocate entirely: ";
        cin >> blockId;
        deallocateSuperBlock(blockId);
        displayEverything(); // Show updated state
        break;
```

```
      case 3:
        // Deallocate part from start
        cout << "Enter BlockId: ";
        cin >> blockId;
        cout << "Enter number of blocks to deallocate from start: ";
        cin >> partSize;
        deallocatePartOfSuperBlock(blockId, partSize);
        displayEverything(); // Show updated state
        break;

      case 4:
        // Deallocate part from anywhere
        cout << "Enter BlockId: ";
        cin >> blockId;
        cout << "Enter start index for deallocation: ";
        cin >> startIndex;
        cout << "Enter number of blocks to deallocate: ";
        cin >> partSize;
        deallocatePartOfSuperBlockAnywhere(blockId, startIndex, partSize);
        displayEverything(); // Show updated state
        break;

      case 5:
        // Display current status
        displayEverything();
        break;

      case 6:
        cout << "Exiting Memory Management System..." << endl;
        break;

      default:
        cout << "Invalid choice! Please enter 1-6." << endl;
    }

  } while (choice != 6);
}


int main() {

  userMemoryManagementInterface();

  return 0;
}
```

Technical Explanation of Memory Management System

1. `initializeMemoryPool()`

Initializes the 64-byte character array `memoryPool` by setting all elements to 'E' (Empty). This establishes the initial state of contiguous memory blocks before any allocation operations.

2. `findAvailableBlock(int size)`

Implements first-fit contiguous memory allocation algorithm. Scans `memoryPool` linearly to locate the first sequence of `size` consecutive 'E' blocks. Returns starting index of available block or -1 if insufficient contiguous space exists. Time complexity: O(n) where n=64.

3. `Append(int startIndex, int size)`

Linked list insertion operation that creates a new `SuperBlock` node and appends it to the tail of the singly linked list. Maintains `head` and `tail` pointers for O(1) insertion. Assigns unique `Blockid` using atomic `nextBlockId` counter.

4. `allocateSuperBlockForString(const string &str)`

Memory allocation routine that:

- Calculates block requirement: `str.length()`
- Invokes `findAvailableBlock()` for contiguous space search
- Performs physical memory allocation by writing string characters to `memoryPool[startIndex + i]`
- Creates logical tracking via `Append()` and stores string metadata

5. `displayMemoryPool()` & `DisplayLinkedList()`

Dual-view memory state inspection:

- `displayMemoryPool()`: Raw dump of 64-byte memory array showing physical allocation map
- `DisplayLinkedList()`: Traversal of linked list displaying SuperBlock metadata including Blockid, data ranges, and size

6. `deallocateSuperBlock(int BlockId)`

Complete memory deallocation operation:

- Linear search through linked list for target `BlockId`
- Marks corresponding `memoryPool` blocks as '_' (freed)
- Performs linked list node deletion with edge case handling for head/tail nodes
- Updates `tail` pointer when deleting terminal node

7. `deallocatePartOfSuperBlock(int BlockId, int partSize)`

Partial deallocation from block start:

- Validates `partSize < current block size`
- Frees first `partSize` blocks in memory pool
- Adjusts SuperBlock metadata: `startIndex += partSize`, `size -= partSize`
- Updates data string via `substr(partSize)` to maintain consistency

8. `deallocatePartOfSuperBlockAnywhere(int BlockId, int StartIndex, int partSize)`

Arbitrary-range deallocation with three operational modes:

- **Start deallocation**: Adjusts block start pointer and size
- **End deallocation**: Reduces block size only
- **Middle deallocation**: Splits block into two nodes using linked list insertion, creating new SuperBlock for post-deallocation segment

9. `userMemoryManagementInterface()`

Command-line interface controller implementing finite state machine:

•Presents menu-driven options for all memory operations
•Maintains input buffer integrity using `cin.ignore()`
•Ensures state visibility via `displayEverything()` after each operation
•Implements graceful termination on exit command

Technical Explanation: Data String Update in Partial Deallocation

The Problem:

The SuperBlock's `data` string becomes **inconsistent** with the memory pool after partial deallocation. When you free the first `partSize` blocks from memory, those characters are physically removed from `memoryPool[]`, but the SuperBlock's `data` field still contains the original full string.

The Inconsistency:

```cpp
// After deallocating first 3 blocks of "Hello"
memoryPool[0] = '_'  // Freed
memoryPool[1] = '_'  // Freed
memoryPool[2] = '_'  // Freed
memoryPool[3] = 'l'  // Still allocated
memoryPool[4] = 'o'  // Still allocated

// But SuperBlock data still shows: "Hello" (WRONG!)
// Should show: "lo" (CORRECT!)
```

The Solution:

```cpp
current->data = current->data.substr(partSize);
```

How It Works:

•`data.substr(partSize)` creates a new string starting from index `partSize`
•For "Hello" with `partSize=3`: `"Hello".substr(3) → "lo"`
•This removes the deallocated portion from the beginning of the string

Why It's Necessary:

1.**Data Integrity**: The `data` field should reflect what's **actually allocated** in memory
2.**Display Consistency**: `DisplayLinkedList()` shows the `data` field - it must match what's physically in `memoryPool`
3.**Logical Coherence**: The SuperBlock metadata should accurately represent its current state, not its historical state

Technical Impact:

•**Memory**: Creates a new string (minimal overhead for small strings)
•**Correctness**: Ensures all system views (memory pool + linked list) remain synchronized
•**Debugging**: Prevents confusion during testing and verification

Bottom Line: The update maintains **system invariants** - the SuperBlock's `data` field must always represent the content of its allocated memory region in `memoryPool[]`.