DESIGN SPECIFICATIONS

DATA STRUCTURES USED :

1) Element

Attributes :

1. ptr _next, _prev [pointers to next and previous elements]
2. int _data [data stored in the element]

2) List

Attributes :

1. ptr head [pointer to the head of the list]
2. int size [size of the list]
3. vector<ptr> occupiedPages [vector of pages occupied by the list]

3) Frame

Attributes :

1. int id [id of the frame]
2. unordered_set<string> localListNames [set of list names declared in frame]

GLOBAL VARIABLES / DATA STRUCTURES :

1. void* mem [stores the starting location of the memory]
2. queue<ptr> freePages [stores the starting location of free pages]
3. unordered_map<string, list> page_table [stores list name and list object]
4. stack<frame> stack_frame [stores the frames]
5. int f_counter [counter for the frame id]

KEY METHODS USED :

1. int createList(string lname, int size) [creates a list of size 'size']
2. int assignVal(string lname, int index, int val) [assigns value 'val' to element at index 'index' in list 'lname']
3. int freeElem(string lname) [frees the list 'lname']
4. int getVal(string lname, int index, int &val) [gets the value of element at index 'index' in list 'lname' and stores it in 'val']
5. void push_frame() [pushes a new frame on the stack]
6. void pop_frame() [pops the top frame from the stack]
7. string generateLName(string lname) [generates scope-based unique name]
8. string findLName(string lname) [disambiguates name based on scope]

PAGING SCHEME :

- Each element is of size 12 bytes (one int value, two int pointers).
- Each page is defined to contain a maximum of 128 such elements.
- For a memory allocation of 250 MB, it translates to roughly 170k free pages.

PAGE TABLE STRUCTURE :

Design :

- Used to store the mapping of list names with the corresponding linked list objects.
- Defined as an unordered map of <string, list> tuples, where list names are strings and corresponding linked lists are stored as “list” objects.
- Each “list” object stores a vector of starting pointers for the pages allocated to it from the memory segment.

Justification :

- The mapping of list names with the corresponding list objects is stored in the page table to allow the library user to refer to the linked lists directly using names.
- The vector of starting pointers to allocated pages alleviates the need for traversal through the whole linked list and allows faster element-wise access using the page offset and corresponding element offset in the page.
- Storing pointers to the allocated pages also enables us to implement non-contiguous memory allocation using the paging scheme described previously.

SCOPE HANDLING :

- A separate “frame” data structure for each new scope is created to store names of linked lists belonging to it.
- When a new scope is created, the user should call the “push_frame()” function to create a new frame and push it to the global stack frame (stack_frame).
- Whenever a new linked list is created and initialized in the scope, it is added to a set of linked list names local to that scope.
- To distinguish between two linked lists with the same name but part of different scopes, the “generateLName()” function is used to create a unique name for each such list depending on the scope it belongs to, which is added to the page table. This naming scheme prevents name conflicts in case of future reference.
- To disambiguate between two lists named similarly by the user in different scopes, the “findLName” function is used to reconstruct the unique scope-specific name for each list, which is then used for page table referencing.
- If the linked list used in a particular scope is not found in the current scope’s set of linked lists, the set of linked lists for the parent scope is referred to. If the name is not found even after multiple such scope resolutions, it is checked in the set of linked lists for the global scope as a last resort.

- Finally, when the scope ends, the corresponding frame is popped from the stack frame and all local lists in the scope are freed.

IMPACT OF “freeElem()” FOR MERGE SORT :

- freeElem() is essential because it allows us to free the small-sized linked lists created during Merge Sort.
- Without this, the **memory footprint would be persistently high** as the smaller sized lists aren’t cleared after usage, but it incurs **lesser time overhead** for clearing lists each time the scope ends.

  NOTE : For a list of 50k elements, the number of leaf nodes = 50000 , and size of each element = 12 bytes.

  Hence, total size for leaf nodes only = 12 * 50000 = 600 KB !

  (Actual memory usage is much higher as each element is allocated a single page, so actual memory footprint = 128 * 600 KB = **75 MB** (~**30% !**))

  For 100 iterations,

  Avg Time taken: 2.53125s
  Max page usage: 52942 (31%)

- If freeElem() is called after each list is finished being used, the **memory footprint decreases** but **more time is taken** for each such list clearing.

  For 100 iterations,

  Avg Time taken: 2.63414s
  Max page usage: 391 (0.23%)

- Such a memory management implementation would have **maximum performance** if the code structure **doesn't involve a large number of scope definitions**, while the **performance would be hampered if the implementation involves multiple scope definitions** (typical of recursive functions). This is because more time overhead would be required for maintaining scopes using stack frames and then deleting local lists when the scopes end.
- Our design **does not involve locks**, because we don't make use of multiple processes/threads that simultaneously access the shared memory segment for allocation or other purposes.