```
#include <stdlib.h>

#include "letterParser.h"


static inline int isUpperAlpha(letterType letter)
{
      return ((letter > 64 && letter < 91) ?  1 :  0);
}

static inline letterType toLowerAlpha(letterType letter)
{
      return ((isUpperAlpha(letter)) ?  (letter + 32) : letter);
}

int parser_parse(const char *filename, PARSER_LETTER_T
**toBeParsedListHead)
{
      FILE *fp = NULL;
      if( (fp = fopen(filename,"r")) == NULL )
      {
            /* LOG OPEN ERROR */
            printf("File open Error\n");
            return 1;
      }

      PARSER_LETTER_T *parsedListHead = *toBeParsedListHead;
      letterType parsedChar;
      int ret = fread(&parsedChar,sizeof(letterType),1,fp);
      while(ret == 1)
      {

            parsedListHead = parser_insert(parsedListHead,
toLowerAlpha(parsedChar));

            ret = fread(&parsedChar,sizeof(letterType),1,fp);
      }
      if(feof(fp))
      {
            /*LOG EOF */
            printf("END OF FILE\n");
      }
      else if(ferror(fp))
      {
            /* LOF ERROR */
            printf("FILE ERROR\n");
            return 1;
      }

      *toBeParsedListHead = parsedListHead;
      return 0;
}


PARSER_LETTER_T* parser_insert(PARSER_LETTER_T *parsedListHead,
letterType letter)
{
      /*We go the a list head having no elements. So we initialized the
head with the new letter */
      if(NULL == parsedListHead)
      {
            /* Creating the linked ist */
            parsedListHead = (PARSER_LETTER_T
*)malloc(sizeof(PARSER_LETTER_T));
            parsedListHead->letterElement = letter;
            parsedListHead->letterCount = 1;
            LIST_HEAD_INIT(&parsedListHead->selfNode);

            return parsedListHead;
      }
      else
      {
            /*
            * We traverse the list and find the occurence of the letter.
            * If found, we increment the letterCount of that node
            * else we add a new node at the end of list
            */
            PARSER_LETTER_T *list_itr = parsedListHead;
            uint8_t found = 0;
            LIST_FOR_EACH_ENTRY(list_itr, &list_itr->selfNode, selfNode)
            {
                  if(list_itr->letterElement == letter)
                  {
                        list_itr->letterCount++;
                        found = 1;
                        break;
                  }

            }
            /*    Improve - If not found, we are already at the end of the
list, so we can just append the new node after
            *     list_itr used above. Expand the scope of that iterator
and we are good to go to add the new node
            *     But can improve after wards. Going with the first
intuition.
            */
            if(!found)
            {
                  PARSER_LETTER_T *newListNode =
(PARSER_LETTER_T*)malloc(sizeof(PARSER_LETTER_T));

                  newListNode->letterElement = letter;
                  newListNode->letterCount = 1;
                  newListNode->selfNode.next = NULL;
                  newListNode->selfNode.prev = NULL;


                  /* Using insert at beginning as to avoid traversing to
the end */
                  return
GET_LIST_CONTAINER(insert_at_beginning(&parsedListHead-
>selfNode,&newListNode->selfNode),PARSER_LETTER_T,selfNode);
            }
            else
                  return parsedListHead;
```

```
	}

}

letterType* parser_getMaxThreeElements(PARSER_LETTER_T *parsedListHead)
{
	PARSER_LETTER_T *list_itr = parsedListHead;
//GET_LIST_CONTAINER(parsedListHead->selfNode.next, PARSER_LETTER_T ,
selfNode);

	static letterType max_arr[3] = {0};

	letterType max1_E = parsedListHead->letterElement;
	letterType max2_E = 0;
	letterType max3_E = 0;

	uint32_t max1_C = parsedListHead->letterCount;
	uint32_t max2_C = 0;
	uint32_t max3_C = 0;

	max_arr[0] = 0;
	max_arr[1] = 0;
	max_arr[2] = 0;

	LIST_FOR_EACH_ENTRY(list_itr, &list_itr->selfNode, selfNode)
	{
		if(list_itr->letterCount > max1_C)
		{
			max1_C = list_itr->letterCount;
			max2_C = max1_C;
			max3_C = max2_C;

			max1_E = list_itr->letterElement;
			max2_E = max1_E;
			max3_E = max2_E;

		}
		else if(list_itr->letterCount > max2_C)
		{
			max2_C = list_itr->letterCount;
			max3_C = max2_C;

			max2_E = list_itr->letterElement;
			max3_E = max2_E;
		}
		else if(list_itr->letterCount > max3_C)
		{
			max3_C = list_itr->letterCount;

			max3_E = list_itr->letterElement;
		}

	}

	max_arr[0] = max1_E;
	max_arr[1] = max2_E;
	max_arr[2] = max3_E;
```

```
      return max_arr;

}

size_t get_occurenceN_letters(PARSER_LETTER_T *parsedListHead, letterType
**inout_elemArray, uint32_t occurenceN)
{
      PARSER_LETTER_T *list_itr = parsedListHead;
      size_t i = 0;
      if(NULL == *inout_elemArray)
      {
            *inout_elemArray =
(letterType*)malloc(sizeof(letterType)*10);
            if(NULL == *inout_elemArray)
            {
                  /*LOG ERROR*/
                  printf("MALLOC ERROR\n");
                  return 0;
            }
      }

      LIST_FOR_EACH_ENTRY(list_itr, &list_itr->selfNode, selfNode)
      {
            if(list_itr->letterCount == occurenceN)
            {
                  *(*inout_elemArray+i) = list_itr->letterElement;
                  i++;
            }
      }

      return i;
}

void cleanup_parser(PARSER_LETTER_T *parsedListHead)
{
      LIST_FOR_EACH_ENTRY(parsedListHead, &parsedListHead->selfNode,
selfNode)
      {
            free(parsedListHead);
      }
}


#include <pthread.h>
#include <time.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <unistd.h>
#include <semaphore.h>

#include "letterParser.h"
#include "time.h"
#include "log_macros.h"

#define TEXT_FILENAME  "Valentinesday.txt"

void* callBack_thread0(void* params)
{
      struct threadParams *inParams = (struct threadParams*)params;
      pthread_t self_pthreadId = pthread_self();
      pid_t process_id = getpid();
      pid_t linux_threadID = syscall(SYS_gettid);

      LOG_INIT(inParams->filename);
      if(!GET_LOG_HANDLE())
            printf("File open error\n");

      char timeString[40] = {0};
      if(get_time_string(timeString) == 0)
            LOG("[ENTRY TIME] %s\n",timeString);
      else
            LOG("[ERROR] Gettimeofday().\n");

      LOG("Setup of Thread0 done\n");


      /* TO DO - Add functions for parsing Valentines.txt */
      PARSER_LETTER_T *letter_list = NULL;
      int ret = parser_parse(TEXT_FILENAME,&letter_list);
      if(ret == 0)
      {
            letterType *inout_elemArray = NULL;
            size_t numofElems = get_occurenceN_letters(letter_list,
&inout_elemArray, 3);

            LOG("Found %u chars with 3 occurence.\n",numofElems);
            PRINT_THREAD_IDENTIFIER();
            printf("Found %u chars with 3 occurence.\n",numofElems);

            for(int  i = 0; i < numofElems && (inout_elemArray+i); i++)
            {
                  LOG("Char: %c\n",inout_elemArray[i]);
                  printf("[%c]",inout_elemArray[i]);
            }

            cleanup_parser(letter_list);
      }
      else
      {
            LOG("[ERROR} PARSING\n");
            PRINT_THREAD_IDENTIFIER();
            printf("[ERROR] PARSING\n");

      }

      LOG("Waiting for SIGUSR.\n");
      PRINT_THREAD_IDENTIFIER();
      printf("Waiting for SIGUSR.\n");

      sem_wait(&gotSignal_sem);


      LOG("Exiting Thread 0\n");
      PRINT_THREAD_IDENTIFIER();
      printf("Exiting thread 0.\n");
      if(get_time_string(timeString) == 0)
            LOG("[EXIT TIME] %s\n",timeString);
      else
            LOG("[ERROR] Gettimeofday().\n");

      if(GET_LOG_HANDLE())
            LOG_CLOSE();

      sem_post(&gotSignal_sem);
}

#include "posixTimer.h"

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <sys/types.h>
#include <string.h>

int register_timer(timer_t *timer_id, void (*timer_handler)(union
sigval), void *handlerArgs)
{

      if(NULL == timer_id)
            return -1;

      struct sigevent sige;

      /*SIGEV_THREAD will call the handler as if it was a new thread */
      sige.sigev_notify = SIGEV_THREAD;
      sige.sigev_notify_function = timer_handler;
//    sige.sigev_value.sival_ptr = timer_id;
      sige.sigev_value.sival_ptr = handlerArgs;
      sige.sigev_notify_attributes = NULL;

      int ret = timer_create(CLOCK_REALTIME, &sige, timer_id);

      return ret;
}

int start_timer(timer_t timer_id , uint32_t time_usec, uint8_t oneshot)
{
      if(NULL == timer_id)
            return -1;

      struct itimerspec ts;

      ts.it_value.tv_sec = time_usec / MICROSEC;
      ts.it_value.tv_nsec = (time_usec % MICROSEC) * 1000;
      if(1 == oneshot)
      {
            ts.it_interval.tv_sec = 0;
            ts.it_interval.tv_nsec = 0;
      }
      else
      {
            ts.it_interval.tv_sec = ts.it_value.tv_sec;
            ts.it_interval.tv_nsec = ts.it_value.tv_nsec;
      }

      int ret = timer_settime(timer_id, 0, &ts, 0);

      return ret;
}

int stop_timer(timer_t timer_id)
{
      if(NULL == timer_id)
            return -1;

      struct itimerspec ts;

      ts.it_value.tv_sec = 0;
      ts.it_value.tv_nsec = 0;
      ts.it_interval.tv_sec = 0;
      ts.it_interval.tv_nsec = 0;

      int ret = timer_settime(timer_id, 0, &ts, 0);

      return ret;
}


int delete_timer(timer_t timer_id)
{
      if(NULL == timer_id)
            return -1;

      int ret = timer_delete(timer_id);

      return ret;


}






#if 0

struct thread_cleanup{

      FILE *fp;
      void *heapMemArray;

};

void thread1_cleanup(void *arg)
{
      /* We need to clear the dynamic memory and file pointers */
      struct thread_cleanup *cleanup_mem = (struct thread_cleanup*)arg;
      if(cleanup_mem->fp)
      {
            fclose(cleanup_mem->fp);
            cleanup_mem->fp = NULL;
      }

      LOG("Exiting Thread 1 from Cleanup\n");
      PRINT_THREAD_IDENTIFIER();
      printf("Exiting thread 1 from Cleanup.\n");

      /* TO DO -Free any heap memory */

}

#endif




void* callBack_thread1(void* params)
{
      struct threadParams *inParams = (struct threadParams*)params;
      pthread_t self_pthreadId = pthread_self();
      pid_t process_id = getpid();
      pid_t linux_threadID = syscall(SYS_gettid);
      pthread_t self = pthread_self();

      LOG_INIT(inParams->filename);
      if(!GET_LOG_HANDLE())
            printf("File open error\n");

      char timeString[40] = {0};
      if(get_time_string(timeString) == 0)
            LOG("[ENTRY TIME] %s\n",timeString);
      else
            LOG("[ERROR] Gettimeofday().\n");


      LOG("Setup of Thread1 done\n");

      //LOG("Registering cleanup function\n");

      //struct thread_cleanup cleanup_struct = { .fp = GET_LOG_HANDLE() ,
.heapMemArray = NULL  };
      //pthread_cleanup_push(thread1_cleanup,(void*)&cleanup_struct);

      /* TO DO - Create and start 100ms timer with callback which prints
CPU utilization */

      LOG("Waiting for SIGUSR.\n");
      PRINT_THREAD_IDENTIFIER();
      printf("Waiting for SIGUSR.\n");

      while(1)
      {
            if(sem_trywait(&gotSignal_sem) == 0)
            {
                  PRINT_THREAD_IDENTIFIER();
                  printf("Got semaphore from try wait.\n");
                  break;
            }
            LOG_CPU_UTILIZATION();
            sleep(5);
            //nanosleep(100000);
      }

      /* Waiting for SIGUSR1 or SIGUSR2. Which releases the semaphore */
      //sem_wait(&gotSignal_sem);


      LOG("Exiting Thread 1\n");
      PRINT_THREAD_IDENTIFIER();
      printf("Exiting thread 1.\n");

      if(get_time_string(timeString) == 0)
            LOG("[EXIT TIME] %s\n",timeString);
      else
            LOG("[ERROR] Gettimeofday().\n");

      if(GET_LOG_HANDLE())
            LOG_CLOSE();

      /* Release the semaphore to be used by other thread */
      sem_post(&gotSignal_sem);
}
#include <sys/time.h>
#include <time.h>
#include <string.h>
#include <stdio.h>

#include "my_time.h"

#define GET_TIMEOFDAY(x,y)   gettimeofday(x,y)
      //syscall(__sys_gettimeofday,x,y)

int get_time_string(char *timeString)
{
      struct timeval tv;
      //struct tm* ptm;
      char time_string[40] = {0};

      /* Obtain the time of day using the system call */
      unsigned long ret = GET_TIMEOFDAY(&tv,NULL);
      if(ret != 0)
      {
            memset(timeString,0,1);
            return 1;
      }
      snprintf(time_string,sizeof(time_string),"%ld.%ld",tv.tv_sec,tv.tv_
usec);
      //ptm = localtime (&tv.tv_sec);
      /* Format the date and time. */
      //strftime (time_string, sizeof (time_string), "%Y-%m-%d %H:%M:%S",
ptm);
      //strftime (time_string, sizeof (time_string), "%X", ptm);
    memcpy(timeString,time_string,40);
    return 0;
}
#include <pthread.h>
#include <time.h>
#include <sys/time.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <unistd.h>
#include <semaphore.h>
#include <signal.h>


#include "letterParser.h"
#include "my_time.h"
#include "my_signals.h"
#include "threadManager.h"
#include "posixTimer.h"

#include "log_macros.h"


#define LOG_FILENAME   "Homework3.log"
#define TEXT_FILENAME  "Valentinesday.txt"

sem_t gotSignal_sem;
sem_t gotTimerSignal_sem;

struct threadParams{

    pthread_t threadId;
    char *info;
    char *filename;
};

static void signal_handler(int signal)
{
      switch (signal)
      {

            case SIGUSR1:
                  STDOUT_LOG("\n[SIGNAL] SIGUSR1 signal.\n");
                  sem_post(&gotSignal_sem);
                  break;
            case SIGUSR2:
                  STDOUT_LOG("\n[SIGNAL] SIGUSR2 signal.\n");
                  sem_post(&gotSignal_sem);
                  break;
            case SIGINT:
                  STDOUT_LOG("\n[SIGNAL] SIGINT signal.\n");
                  sem_post(&gotSignal_sem);
                  break;
            case SIGTERM:
                  STDOUT_LOG("\n[SIGNAL] SIGTERM signal.\n");
                  sem_post(&gotSignal_sem);
                  break;
            case SIGTSTP:
                  STDOUT_LOG("\n[SIGNAL] SIGTSTP signal.\n");
                  sem_post(&gotSignal_sem);
                  break;
            default:
                  STDOUT_LOG("\n[SIGNAL] Invalid signal.\n");
                  break;
      }
}

void* callBack_thread0(void* params)
{
      struct threadParams *inParams = (struct threadParams*)params;
      pthread_t self_pthreadId = pthread_self();
      pid_t process_id = getpid();
      pid_t linux_threadID = syscall(SYS_gettid);

      LOG_INIT(inParams->filename);
      if(GET_LOG_HANDLE() == NULL)
      {
            STDOUT_LOG("[ERROR] File open error\n");
      }

      char timeString[40] = {0};
      if(get_time_string(timeString) == 0)
            LOG("[ENTRY TIME] %s\n",timeString);
      else
            LOG("[ERROR] Gettimeofday().\n");

      LOG("[INFO] Setup of Thread0 done\n");


      /* TO DO - Add functions for parsing Valentines.txt */
      PARSER_LETTER_T *letter_list = NULL;
      int ret = parser_parse(TEXT_FILENAME,&letter_list);
      if(ret == 0)
      {
            letterType *inout_elemArray = NULL;
            size_t numofElems = get_occurenceN_letters(letter_list,
&inout_elemArray, 3);

            LOG("[INFO] Found %u chars with 3 occurence. -",numofElems);
            STDOUT_LOG("Found %u chars with 3 occurence. -",numofElems);

            for(int  i = 0; i < numofElems && (inout_elemArray+i); i++)
            {
                  LOG_PLAIN("[%c]",inout_elemArray[i]);
                  STDOUT_LOG_PLAIN("[%c]",inout_elemArray[i]);
            }
            STDOUT_LOG_PLAIN("\n");
            LOG_PLAIN("\n");

            cleanup_parser(letter_list);
      }
      else
      {
            LOG("[ERROR} PARSING\n");
            STDOUT_LOG("[ERROR] PARSING\n");

      }
      LOG("[INFO] Waiting for SIGUSR.\n");
      STDOUT_LOG("[INFO] Waiting for SIGUSR.\n");

      sem_wait(&gotSignal_sem);


      LOG("[INFO] Exiting Thread 0\n");
      STDOUT_LOG("[INFO] Exiting thread 0.\n");

      if(get_time_string(timeString) == 0)
            LOG("[EXIT TIME] %s\n",timeString);
      else
            LOG("[ERROR] Gettimeofday().\n");

      if(GET_LOG_HANDLE())
            LOG_CLOSE();

      sem_post(&gotSignal_sem);
}

static void timer_handler(union sigval sig)
{
      sem_post(&gotTimerSignal_sem);
}

void* callBack_thread1(void* params)
{
      struct threadParams *inParams = (struct threadParams*)params;
      pthread_t self_pthreadId = pthread_self();
      pid_t process_id = getpid();
      pid_t linux_threadID = syscall(SYS_gettid);
      pthread_t self = pthread_self();

      sem_init(&gotTimerSignal_sem,0,0);

      LOG_INIT(inParams->filename);
      if(GET_LOG_HANDLE() == NULL)
      {
            STDOUT_LOG("[ERROR] File open error\n");
      }

      char timeString[40] = {0};
      if(get_time_string(timeString) == 0)
            LOG("[ENTRY TIME] %s\n",timeString);
      else
            LOG("[ERROR] Gettimeofday().\n");

      timer_t timer_id;

      if(register_timer(&timer_id, timer_handler,&timer_id) == -1)
      {
            LOG("[ERROR] Register Timer\n");
            //exit (1);
      }
      else
            LOG("[INFO] Timer created\n");
      if(start_timer(timer_id , 500, 0) == -1)
      {
            LOG("[ERROR] Start Timer\n");
            //exit (1);
      }
      else
            LOG("[INFO] Timer started\n");

      LOG("[INFO] Setup of Thread1 done\n");


      LOG("[INFO] Waiting for SIGUSR.\n");
      STDOUT_LOG("INFO] Waiting for SIGUSR.\n");


      while(1)
      {
            if(sem_trywait(&gotSignal_sem) == 0)
            {
                  STDOUT_LOG("[INFO] Got semaphore from try wait.\n");
                  break;
            }
            if(sem_trywait(&gotTimerSignal_sem) == 0)
            {
                  LOG_CPU_UTILIZATION();
            }
      }


      /* Waiting for SIGUSR1 or SIGUSR2. Which releases the semaphore */
      //sem_wait(&gotSignal_sem);

      if(delete_timer(timer_id) == -1)
      {
            LOG("[ERROR] Delete Timer\n");
            //exit (1);
      }
      else
            LOG("[INFO] Timer deleted\n");

      sem_destroy(&gotTimerSignal_sem);

      if(get_time_string(timeString) == 0)
            LOG("[EXIT TIME][THREAD1] %s\n",timeString);
      else
            LOG("[ERROR] Gettimeofday().\n");

      if(GET_LOG_HANDLE())
            LOG_CLOSE();

      STDOUT_LOG("[INFO] Exiting thread 1.\n");

      /* Release the semaphore to be used by other thread */
      sem_post(&gotSignal_sem);
}
```

```
int threadManager_startThreads()
{
      pthread_t p_threads[2];
      struct sigaction sa;
    int ret;
    struct threadParams thread_info[2];

      sem_init(&gotSignal_sem,0,0);

      LOG_INIT(LOG_FILENAME);
      if(!GET_LOG_HANDLE())
      {
            STDOUT_LOG("[ERROR] Cannot open log\n");
            return 1;
      }

      LOG("[INFO] Log initialized.\n");

    /*Registering the signal callback handler*/
      register_signalHandler(&sa,signal_handler, REG_SIG_ALL);

      thread_info[0].threadId     = 0;
    thread_info[0].info           = "Thread0";
      thread_info[0].filename     = LOG_FILENAME;

    thread_info[1].threadId       = 1;
    thread_info[1].info                 = "Thread1";
      thread_info[1].filename     = LOG_FILENAME;

      LOG("[INFO] Creating children Threads.\n");

      ret = pthread_create(&p_threads[0], NULL, callBack_thread0,
(void*)&thread_info[0]);
      if(ret != 0)
      {
            LOG("[ERROR] Cannot create child thread 0\n");
            if(GET_LOG_HANDLE())
                  LOG_CLOSE();
            return 1;
      }

      ret = pthread_create(&p_threads[1], NULL, callBack_thread1,
(void*)&thread_info[1]);
      if(ret != 0)
      {
            LOG("[ERROR] Cannot create child thread 1\n");
            if(GET_LOG_HANDLE())
                  LOG_CLOSE();
            return 1;
      }

    LOG("[INFO] Thread created successfully\n");

      /* Waiting on child threads to complete */
      ret = pthread_join(p_threads[0],NULL);
      if(0 != ret)
      {
            LOG("[ERROR] Pthread JOIN error\n");
            STDOUT_LOG("[ERROR] Join Error Thread 0\n");
            if(GET_LOG_HANDLE())
                  LOG_CLOSE();
            return 1;
      }

      ret   = pthread_join(p_threads[1],NULL);
      if(0 != ret)
      {
            LOG("[ERROR] Pthread JOIN error\n");
            STDOUT_LOG("[ERROR] Join Error Thread 0\n");
            if(GET_LOG_HANDLE())
                  LOG_CLOSE();
            return 1;
      }


      sem_destroy(&gotSignal_sem);
      LOG("[INFO] GoodBye!!\n");
      STDOUT_LOG("[INFO] GoodBye!!\n");

      if(GET_LOG_HANDLE())
            LOG_CLOSE();

    return EXIT_SUCCESS;


}
#include <stdio.h>

#include "my_signals.h"
#include "log_macros.h"

int register_signalHandler(struct sigaction *sa, void (*handler)(int),
REG_SIGNAL_FLAG_t signalMask)
{
      sa->sa_handler = handler;

      sa->sa_flags = SA_RESTART;

      sigfillset(&sa->sa_mask);

      int ret_error = 0;

      if ((signalMask & REG_SIG_USR1) && sigaction(SIGUSR1, sa, NULL) ==
-1)
      {
            ret_error++;
            PRINT_THREAD_IDENTIFIER();
            printf("Cannot handle SIGUSR1.\n");
      }

      if ((signalMask & REG_SIG_USR2) && sigaction(SIGUSR2, sa, NULL) ==
-1)
      {
            ret_error++;
            PRINT_THREAD_IDENTIFIER();
```

```
		printf("Cannot handle SIGUSR2.\n");
	}

	if ((signalMask & REG_SIG_INT) && sigaction(SIGINT, sa, NULL) == -
1)
	{
		ret_error++;
		PRINT_THREAD_IDENTIFIER();
		printf("Cannot handle SIGINT.\n");
	}

	if ((signalMask & REG_SIG_TSTP) && sigaction(SIGTERM, sa, NULL) ==
-1)
	{
		ret_error++;
		PRINT_THREAD_IDENTIFIER();
		printf("Cannot handle SIGTERM.\n");
	}

	if ((signalMask & REG_SIG_TSTP) && sigaction(SIGTSTP, sa, NULL) ==
-1)
	{
		ret_error++;
		PRINT_THREAD_IDENTIFIER();
		printf("Cannot handle SIGTSTOP.\n");
	}

	return ret_error;
}
/*
 * @File doublyLinkedList.c
 *
 * @Created on: 02-Feb-2018
 * @Author: Gunj Manseta
 */

#include "doublyLinkedList.h"

void LIST_HEAD_ALLOCATE(LIST_NODE_T **list_node)
{
	*list_node = (LIST_NODE_T*)malloc(sizeof(LIST_NODE_T));
}

/**
 * @brief Function to initialize the head node which takes head list
pointer
 * @param [in] LIST_NODE_T* list head
 * @return void
 **/
void LIST_HEAD_INIT(LIST_NODE_T *list_node)
{
	if(!list_node)
	{
		LIST_HEAD_ALLOCATE(&list_node);
	}

	list_node->prev = NULL;
	list_node->next = NULL;
```

```
}

/**
 * @brief Inserts the node at the beginning and the new node becomes the
head
 * @param [in] LIST_NODE_T* list head
 * @param [in] LIST_NODE_T* new node
 * @return LIST_NODE_T* new head
 */
LIST_NODE_T* insert_at_beginning(LIST_NODE_T *list_head, LIST_NODE_T
*new_node)
{
      if(list_head)
      {
            new_node->prev   = NULL;
            new_node->next   = list_head;
            list_head->prev = new_node;
      }
      else
      {
            LIST_HEAD_INIT(new_node);
      }

      return new_node;
}

/**
 * @brief Inserts the node at the end of the list
 * @param [in] LIST_NODE_T* list head
 * @param [in] LIST_NODE_T* new node
 * @return LIST_NODE_T* head
 */
LIST_NODE_T* insert_at_end(LIST_NODE_T *list_head, LIST_NODE_T *new_node)
{
      if(list_head)
      {
            LIST_NODE_T *list_itr = list_head;

            //while there no element in the list. i.e. traversing to the
end of the list
            while(list_itr->next)
                  list_itr = list_itr->next;

            new_node->prev = list_itr;
            new_node->next = list_itr->next;
            list_itr->next = new_node;

            return list_head;
      }
      else
      {
            LIST_HEAD_INIT(new_node);
            return new_node;
      }

}
```

```
/**
 * @brief Inserts the node at the position given from the base_node i.e.
@{base + position}
 * @param [in] LIST_NODE_T* base node
 * @param [in] LIST_NODE_T* new node
 * @param [in] int position
 * @return LIST_NODE_T* head
 */
LIST_NODE_T* insert_at_position(LIST_NODE_T *base_node, LIST_NODE_T
*new_node, int pos)
{
      if(base_node)
      {
            LIST_NODE_T *list_itr = base_node;
            while(((--pos) > 0) && list_itr->next)
            {
                  list_itr = list_itr->next;
            }

            //the list_itr points to the node after which new node should
be entered
            new_node->prev               = list_itr;
            new_node->next               = list_itr->next;
            list_itr->next ? list_itr->next->prev = new_node : 0;
            list_itr->next               = new_node;

            //traversing to the head
            list_itr = base_node;
            while(list_itr->prev)
                  list_itr = list_itr->prev;

            return list_itr;
      }
      else
      {
            LIST_HEAD_INIT(new_node);
            return new_node;
      }

}

/**
 * @brief Delete the node at the beginning of the list so the head gets
updated
 * The deleted node is head, the deleted node pointer is returned as it
is required
 * to free the containing structure using the GET_LIST_CONTAINER macro.
 * We are taking the pointer to the head pointer, so the list_head gets
updated
 * @param [in][out] LIST_NODE_T** address of head node pointer
 * @return LIST_NODE_T* deleted node
 */
LIST_NODE_T* delete_from_beginning(LIST_NODE_T **list_head)
{
      if(*list_head)
      {

                 (*list_head)->next->prev = NULL;
```

```
                LIST_NODE_T *deletedNode = *list_head;
                *list_head = (*list_head)->next;
                return deletedNode;
      }
      else
            return NULL;
}


/**
 * @brief Delete the node at the end of the list so the tail gets updated
 * The deleted node pointer is returned as it will be required to free
the entire list node as well
 * as the containing structure using the GET_LIST_CONTAINER macro.
 * @param [in] LIST_NODE_T* head node
 * @return LIST_NODE_T* The node that was deleted
 */
LIST_NODE_T* delete_from_end(LIST_NODE_T *list_head)
{
      if(list_head)
      {
            //as we are deleting the tail itself, we need to update the
next of tail->prev as null

            LIST_NODE_T *list_itr = list_head;
            //while there no element in the list. i.e. traversing to the
end of the list
            while(list_itr->next)
                  list_itr = list_itr->next;

            list_itr->prev->next = NULL;

            //returning the tail as it will be required by the callee to
free list_node as well the containing strucutre
            return list_itr;
      }
      else
            return NULL;
}

/**
 * @brief Delete the node at the specified pos from the base_node
 * The deleted node pointer is returned as it will be required to free
the entire list node as well
 * as the containing structure using the GET_LIST_CONTAINER macro.
 * @param [in] LIST_NODE_T* base node
 * @param [in] int position from base node
 * @return LIST_NODE_T* The node that was deleted
 */
LIST_NODE_T* delete_from_position(LIST_NODE_T *base_node, int pos)
{
      if(base_node)
      {
            LIST_NODE_T *list_itr = base_node->next;
            while(((--pos) > 0 ) && list_itr)
            {
                  list_itr = list_itr->next;
            }
            //the list_itr points to the node before the node should be 
deleted
            list_itr->prev->next = list_itr->next;
            list_itr->next->prev = list_itr->prev;

            //returning the tail as it will be required by the callee to 
free list_node as well the containing strucutre
            return list_itr;
      }
      else
            return NULL;
}

/**
 * @brief Gives the size of the list from the given node
 * @param [in] LIST_NODE_T* hnode
 * @return size_t The size of list from the given node
 */
size_t size(LIST_NODE_T *node)
{
      size_t list_size = 0;
      while(node)
      {
            list_size++;
            node = node->next;
      }

      return list_size;

}



#include "threadManager.h"

int main()
{
    int ret =  threadManager_startThreads();

    return ret;
}
```