In the following problems, we will explore two different data structures and associated algorithms. Both data structures have similar interfaces but very different implementations. These data structures could be used in an operating system to manage programs running on a server. Every program is given a unique process ID, and every program is also associated with the username of the user running the program. For example, the following commands will display your username and the process IDs of all programs you are currently running on ec2linux:

```
% whoami
cb535
% ps
  PID  TTY     TIME    CMD
16734 pts/0  00:00:00 bash
17018 pts/0  00:00:00 ps
```

We will explore data structures to store the username and process ID for each program currently running on a system. The data structures should support queries by the operating system. The operating system should be able to query which user is running the program with a given process ID, and the operating system should also be able to list all process IDs associated with a given username. For this problem, you can assume process IDs range from zero to 31.

**Problem 1. Singly List-Based Program Info Data Structure**

Our first data structure will be similar in spirit to the singly linked list data structure discussed in lecture. The data structure is named `slist_pinfo`, since it is a list meant to store information about programs. We are using a list-based instead of a vector-based implementation, because we anticipate needing to frequently insert and remove entries as programs are started and then finish. The interface for `slist_pinfo` is as follows.

```c
typedef struct 
{
    // implementation defined
} slist_pinfo_t;

void slist_pinfo_construct ( slist_pinfo_t* this );
void slist_pinfo_destruct ( slist_pinfo_t* this );
void slist_pinfo_add ( slist_pinfo_t* this, int pid, char* uname );
void slist_pinfo_remove ( slist_pinfo_t* this, int pid );
char* slist_pinfo_get_uname ( slist_pinfo_t* this, int pid );
void slist_pinfo_print_pids ( slist_pinfo_t* this, char* uname );
```
Part 1.1A Implementing slist_pinfo_add

The implementation for the slist_pinfo_t struct, slist_pinfo_construct function, and slist_pinfo_add function are shown below. Notice that we always add the new entry to the end of the linked list. Draw the state diagram that corresponds to the execution of this C program.

```c
typedef struct _slist_pinfo_entry_t {
    int pid;
    char* uname;
    struct _slist_pinfo_entry_t* next_p;
} slist_pinfo_entry_t;

typedef struct {
    slist_pinfo_entry_t* head_p;
    slist_pinfo_entry_t* tail_p;
} slist_pinfo_t;

void slist_pinfo_construct( slist_pinfo_t* this )
{
    this->head_p = NULL;
    this->tail_p = NULL;
}

void slist_pinfo_add( slist_pinfo_t* this, int pid, char* uname )
{
    slist_pinfo_entry_t* new_entry_p = malloc( sizeof(slist_pinfo_entry_t) );
    new_entry_p->pid = pid;
    new_entry_p->uname = uname;
    new_entry_p->next_p = NULL;
    // Handle case where slist is empty
    if ( this->tail_p == NULL ) {
        this->head_p = new_entry_p;
        this->tail_p = new_entry_p;
    } else {
        this->tail_p->next_p = new_entry_p;
        this->tail_p = new_entry_p;
    }
}

int main( void )
{
    slist_pinfo_t pinfo;
    slist_pinfo_construct( &pinfo );
    char u0[] = "cfb";
    slist_pinfo_add( &pinfo, 1, u0 );
    char u1[] = "nfc";
    slist_pinfo_add( &pinfo, 31, u1 );
    return 0;
}
```
Part 1.B Implementing `slist_pinfo_get_uname`

Develop an implementation for `slist_pinfo_get_uname` function. The function should return the username corresponding to the given process ID (pid). If the pid is not present in the data structure, then the function should return a NULL pointer to indicate an error.

```c
Curr_t *slist_pinfo_get_uname (slist_pinfo_t **this, int pid)
{
    slist_pinfo_entry_t *curr_p = this->head_p;

    while (curr_p != NULL)
    {
        if (curr_p->pid == pid)
        {
            return curr_p->uname;
            curr_p = curr_p->next_p;
        }
    }

    return NULL;
}
```
Problem 2. Lookup-Table Program Info Data Structure

Our second data structure uses a new technique called a lookup table. A lookup table is an array where we can transform what we are searching for into an array index; then we can directly use the array index to access the corresponding entry. The data structure is named lut_pinfo; lut stands for lookup-table. The interface for lut_pinfo is as follows.

```c
typedef struct {
    // implementation defined
} lut_pinfo_t;
void lut_pinfo_construct ( lut_pinfo_t* this );
void lut_pinfo_destruct ( lut_pinfo_t* this );
void lut_pinfo_add ( lut_pinfo_t* this, int pid, char* uname );
void lut_pinfo_remove ( lut_pinfo_t* this, int pid );
char* lut_pinfo_get_uname ( lut_pinfo_t* this, int pid );
void lut_pinfo_print_pids ( lut_pinfo_t* this, char* uname );
```
Part 2.A Implementing `lut_pinfo_add`

The implementation for the `lut_pinfo_construct` and `lut_pinfo_add` functions are shown below. Notice that we explicitly initialize all entries in the `pid2uname` array to be NULL. We will use a NULL pointer to indicate whether or not the corresponding entry is valid. **Draw the state diagram that corresponds to the execution of this C program.**

```c
typedef struct {
   // This is an array of 32 char pointers
   char* pid2uname[32];
} lut_pinfo_t;

void lut_pinfo_construct( lut_pinfo_t* this )
{
   for ( int i = 0; i < 32; i++ )
      this->pid2uname[i] = NULL;
}

void lut_pinfo_add( lut_pinfo_t* this, int pid, char* uname )
{
   this->pid2uname[pid] = uname;
}

int main( void )
{
   lut_pinfo_t pinfo;
   lut_pinfo_construct( &pinfo );
   char u0[] = "cfb";
   lut_pinfo_add( &pinfo, 1, u0 );
   char u1[] = "nfc";
   lut_pinfo_add( &pinfo, 31, u1 );
   return 0;
}
```
Part 2.B Implementing `lut_pinfo_get_uname`

Develop an implementation for `lut_pinfo_get_uname` function. The function should return the username corresponding to the given process ID (pid). If the pid is not present in the data structure, then the function should return a NULL pointer to indicate an error.

```c
char * lut_pinfo_get_uname ( struct lut_pinfo * this, int pid )
{
    return this->uid2uname [ pid ];
}
```

Part 2.C Implementing `lut_pinfo_print_pids`

Develop an implementation for `lut_pinfo_print_pids` function. The function should use printf to print every process ID corresponding to the given username (uname). You can use the strcmp function we developed in the previous discussion section. This function compare two strings and returns one if they are equal and zero if they are not equal.

```c
void lut_pinfo_print_pids ( struct lut_pinfo * this, char * uname )
{
    for ( int i = 0; i < 31; i++ )
    {
        if ( strcmp ( this->uid2uname [ i ], uname )
            printf ( "%d
", i );
    }
}
```
Problem 3. Comparative Analysis

In this problem, you should perform a comparative analysis of the two implementations explored in Problems 1 and 2, with the ultimate goal of making a compelling argument for which implementation will perform better across a large number of usage scenarios. While you are free to use whatever approach you like, we recommend you structure your response in several paragraphs. The first paragraph might discuss the performance of both implementations using time complexity analysis. Remember that asymptotic big-O time complexity analysis is not the entire story; it is just the starting point for understanding execution time. The second paragraph might discuss the space requirements of both implementations using space complexity analysis. Remember that asymptotic big-O space complexity analysis is not the entire story; it is just the starting point for understanding space usage. The third paragraph can conclude by making a compelling argument for which implementation will perform better in the general case, or if you cannot strongly argue for either implementation explain why.