Assignment 7

CSCI 2132: Software Development

Due April 8, 2019

Assignments are due on the due date before 23:59. All assignments must be submitted electronically via the course SVN server. Plagiarism in assignment answers will not be tolerated. By submitting your answers to this assignment, you declare that your answers are your original work and that you did not use any sources for its preparation other than the class notes, the textbook, and ones explicitly acknowledged in your answers. Any suspected act of plagiarism will be reported to the Faculty's Academic Integrity Officer and possibly to the Senate Discipline Committee. The penalty for academic dishonesty may range from failing the course to expulsion from the university, in accordance with Dalhousie University's regulations regarding academic integrity.

General Instructions: How to Submit Your Work

You must submit your assignment answers electronically:

- Change into your subversion directory on bluenose: `cd ~/csci2132/svn/CSID`.
- Create a directory `a7` for the current assignment.
- Change into your assignment directory: `cd a7`.
- Create files inside the a7 directory as instructed in the questions below and put them under Subversion control using `svn add <filename>`. **Only add the files you are asked to add!**
- Once you are done answering all questions in the assignment (or the ones that you are able to answer—hopefully all), the contents of your a7 directory should look like this:

```
a7
├─ permutations.c
├─ linked_list.c
├─ linked_list.h
├─ node_pool.c
├─ node_pool.h
└─ stringset.c
```

  (You will also have executable programs and potentially some data files in this directory, but you should not add them to SVN.) Submit your work using `svn commit -m"Submit Assignment 7"`.

(Q1) Recursion

Consider the sequence of the first n positive integers, $\langle 1, 2, \ldots, n \rangle$. A *permutation* of this sequence is any sequence $\langle x_1, x_2, \ldots, x_n \rangle$ that contains each of the numbers $1, 2, \ldots, n$ exactly once. So, $\langle 3, 4, 2, 1 \rangle$ is a permutation of $\langle 1, 2, 3, 4 \rangle$ but $\langle 1, 3, 2, 3 \rangle$ and $\langle 1, 5, 3, 2 \rangle$ are not.

A permutation $\langle x_1, x_2, \ldots, x_n \rangle$ is *lexicographically less* than another permutation $\langle y_1, y_2, \ldots, y_n \rangle$ if and only if there exists an index $1 \le k \le n$ such that

- $x_i = y_i$ for all $1 \le i < k$ and
- $x_k < y_k$.

So, $\langle 2, 1, 3, 4 \rangle$ is lexicographically less than $\langle 2, 3, 1, 4 \rangle$ because $x_1 = y_1 = 2$ and $x_2 = 1 < y_2 = 3$. If the elements in the sequence were letters rather than integers, you could interpret the sequence of letters as a word and the ordering defined here would simply be the one in which the words would occur in a dictionary.

Your task is to write a C program `permutations` that takes a command line argument n and outputs all permutations of the sequence $\langle 1, 2, \ldots, n \rangle$ in lexicographic order. The output should consist of $n!$ lines, each containing one permutation. The numbers in each permutation must be printed in order, separated by exactly one space between each pair of consecutive numbers. Thus, when running `permutations 3`, you should see the following output on your screen:

```
$ ./permutations 3
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

`permutations 4` should produce

```
$ ./permutations 4
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 1 3
2 4 3 1
3 1 2 4
3 1 4 2
```

```
3 2 1 4
3 2 4 1
3 4 1 2
3 4 2 1
4 1 2 3
4 1 3 2
4 2 1 3
4 2 3 1
4 3 1 2
4 3 2 1
```

As the title of the question suggests, recursion is a natural strategy to implement this program. You should not have to generate the permutations and then sort them. Your code should be able to generate the permutations in the desired order.

The basic strategy is fairly simple: Your top-level invocation of the function that generates the permutation should iterate over all possible choices of the first element in the permutation, from smallest to largest. For each such choice, you make a recursive call that generates all permutations of the remaining $n-1$ elements. How do you do this? You iterate over all possible choices for the second element, from smallest to largest and, for each such choice, make a recursive call that generates all permutations of the remaining $n-2$ elements, and so on. The deepest recursive call is asked to generate a permutation of 0 elements, that is, you have already chosen all n elements in the permutation. This is the point at which you print the current permutation.

One issue you will have to deal with is how to keep track of the elements that can still be chosen as the second, third, fourth, ... element in the permutationu after choosing the first one, two, three, ... elements in the permutation. One easy way to do this is to initialize an array of size n that contains all the numbers from 1 to n. Whenever you choose an element from this array, you set its entry to 0 and, in subsequent recursive calls, you skip elements that are 0 because they were already chosen. Don't forget to restore each number x in this array of candidates when it is no longer part of the set of elements currently chosen as part of the permutation. This is not the most efficient way to do this, but efficiency is not your main concern in this question.

Implement your program in a single source file `permutations.c`. Do not submit the executable file obtained by compiling it. However, you should obviously compile and test your code.

(Q2) Structs, unions, and dynamic memory management

The ability to allocate and deallocate memory on the heap at arbitrary times is key to writing flexible programs, but every allocation or deallocation has an associated cost because the runtime system or operating system needs to perform a certain amount of bookkeeping to keep track of which memory locations on the heap are currently occupied and which ones are available for future allocation requests. As a result, in programs that create and destroy many small objects on the heap, the cost of heap management may become the main performance bottleneck. This is the case in object-oriented programs that manipulate complex collections of small objects and in the implementation of pointer-based data structures such as linked lists and binary search trees, where a naïve implementation represents each list or tree node as a separately allocated block on the heap.

A common technique to improve the performance of such programs substantially is to allocate larger chunks of memory at a time, large enough to hold hundreds or thousands of the manipulated objects (e.g., a chunk that is big enough to hold 1000 list nodes in a linked list implementation). Your code then has to manually track which of the slots in this large memory chunk are occupied or not, but this is a much simpler problem that can be solved much more efficiently than general heap management. The data structure that manages the allocation of individual list nodes from this array of list nodes is often referred to as a "node pool".

Here, your task is to implement a linked list that uses a node pool to efficiently manage the memory occupied by its nodes. As an application, you will use the same test program as in Lab 8, which needs a data structure to store a collection of strings. In Lab 8, you implement this collection as a splay tree in order to guarantee fast searches. Here, you do not worry about the speed of searching the data structure, only about the cost of heap management, so a linked list suffices as the data structure to store the collection of strings.[1]

Develop your code for this question in four steps:

1. Implement a basic linked list data structure. (You can use the discussion of linked lists from class as a guide.)
2. Implement the test application. (You can cut and paste most of this code from Step 7 in Lab 8.)
3. Implement the node pool.
4. Modify your list implementation from Step 1 to use the node pool from Step 3 to manage the allocation of list nodes.

Step 1: Implement a (singly) linked list

A *(singly) linked list* is a sequence of *nodes* linked together by successor pointers. The first node of the list is its *head*. The last node is its *tail*. Every node that is not the tail has exactly one successor. Every node that is not the head is the successor of exactly one node. Every node stores some data item, which we will call the *value* of the node here. This is illustrated in Figure 1.

[1]If you enjoy implementing data structures and are keen to practice your C programming skills, I encourage you to attempt the following project in your spare time: The splay tree implementation in Lab 8 allocates each node as its own block of memory on the heap. The linked list implementation you develop in this assignment avoids this but does not support fast searches. Try to implement a splay tree (or other *balanced* binary search tree if you want) and use the strategy from this assignment to manage the space occupied by its nodes efficiently.

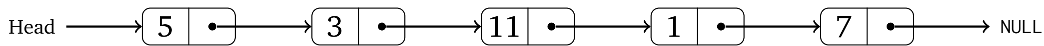

Figure 1: A linked list storing a sequence of integers.

Develop a linked list implementation that supports the following operations:

```
list_t make_list();
```

This creates a new linked list and returns a reference to it. Similarly to Lab 8, the list should be represented as a `struct _list_t` and `list_t` should be a pointer to a heap-allocated `struct _list_t`, that is, the `list_t` type should be defined using `typedef struct _list_t *list_t;`

```
void destroy_list(list_t);
```

This destroys the given linked list, that is, it frees all the memory the list occupies on the heap. This requires the deallocation of the list structure itself as well as of all node structures that are part of the list.

```
node_t list_add(list_t, void *);
```

This adds a data item referenced by a `void *` to the beginning of the list. The list should simply store the pointer to the data item in the new list node. The data should not be copied into the list. This is similar to the approach taken in Lab 8. See the discussion there to understand why this is useful. The return value is a pointer to the new list node, that is, the `node_t` type should be defined using `typedef struct _node_t *node_t;`

```
void *list_delete(list_t, node_t);
```

This deletes the given node from the given list and returns the data that was stored at that node. (Since the node only stores a `void *` to the data and the data is always kept externally from the list, you simply return the `void *` that was stored at the list node.) Again, this is similar to the binary tree implementation in Lab 8.

```
void *node_value(node_t);
```

This returns a pointer to the data element stored at the given node. (The node itself stores a `void *` to this data item. You should simply return this pointer.)

```
node_t list_head(list_t);
```

This returns (a pointer to) the first node of the list. (Recall that `node_t` is a pointer to a list node.)

```
node_t node_succ(node_t);
```

This returns the successor of the given node in the list, that is, the next node in the list after this node. If the given node is the last node of the list, this function returns `NULL`.

In this first implementation of your linked list data structure, it is okay to allocate the list and the nodes of the list as individual chunks of memory, that is, using one `malloc` call per list node.

For this step, create a header file `linked_list.h` that *declares* the above types and function prototypes and an implementation file `linked_list.c` that *defines* the above types and functions. Review Steps 1–6 of Lab 8 if you do not remember how to do this.

Step 2: Use the linked list to store a collection of strings

For this step, you can use the code from Step 7 in Lab 8, but you have to modify it slightly because the linked list structure you have developed in Step 1 does not support a search operation as the binary search tree developed in Lab 8 does.

Your goal is to implement a program that starts out with an empty collection of strings and repeatedly prompts you for input by printing "? ". Valid inputs are:

- `a <string>` to add a string to the collection.
- `d <pattern>` to delete a string from the collection. This operation deletes the first string it finds that starts with the given pattern. For example, `d Nor` may delete the string `Norbert` from the collection if it is currently in the collection. If there is no string in the collection that starts with the given pattern, this operation does not modify the current collection and prints `STRING NOT FOUND`.
- `f <pattern>` to find an arbitrary string in the collection that starts with the given pattern and print it on the screen. Similar to the delete operation, `f Nor` may report the string `Norbert` if this string is in the collection. Also similar to the delete operation, this find operation prints `STRING NOT FOUND` if there is no string in the collection that starts with the given prefix.
- `l` to list all strings currently in the collection. (In contrast to Lab 8, the strings do not need to be listed in sorted order. Thus, you can simply list them in the order they are stored in the list.)
- `q` to terminate the program.

Implement the required code (which you are free to copy from Lab 8 and modify appropriately) in a file `stringset.c`. If you copy this code from Lab 8, you need to make sure that you include `linked_list.h` instead of `tree.h` and that you replace the use of `tree_find` in the code from Lab 8 with appropriate custom code that uses `list_head`, `node_succ`, and `node_value` to search for a string with the given prefix as part of the delete and find operations. Additional small changes are required to replace other calls to tree functions with their appropriate counterparts in the linked list implementation.

Building an executable program `stringset` requires you to compile both the `stringset.c` and `linked_list.c` files and link the two resulting object files. Review Step 8 of Lab 8 if you do not remember how to do this. This executable file is not part of your assignment submission, but you should obviously build and run your code in order to check your code for syntax errors and test your code.

Step 3: Implement a node pool

This is the main creative part of this question. You will implement a node pool that allows you to manage the heap memory occupied by list nodes more efficiently than by using one call to `malloc` per list node.

The data structure you need to implement is implemented as a `struct _node_pool_t`. You will again define an alias `node_pool_t` for pointers to this struct:

```
typedef struct _node_pool_t *node_pool_t;
```

The structure supports exactly four operations:

```
node_pool_t make_node_pool();
```

This creates a new node pool data structure.

```
void destroy_node_pool(node_pool_t);
```

This destroys the given node pool, that is, it frees all memory it occupies on the heap.

```
node_t request_node(node_pool_t);
```

This requests a new linked list node from the node pool. In Step 4, you will modify your linked list implementation to replace calls to `malloc` to create new list nodes with calls to `request_node`.

```
void release_node(node_pool_t, node_t);
```

This releases a previously requested linked list node back to the pool. The node pool can reuse the memory occupied by this node to satisfy future node requests made by calling `request_node`. In Step 4, you will modify your linked list implementation to replace calls to `free` to release list nodes with calls to `release_node`.

Let us discuss a strategy for developing an efficient node pool structure that does not call `malloc` for every single list node that is created and instead requests space for list nodes in larger chunks.

Let us assume we will never create more than 1023 list nodes. (As a bonus part of this step, you may remove this assumption.) Then we can get away with a single call to `malloc` by allocating a block of memory of size `1023 * sizeof(struct _node_t)`, large enough to hold 1023 list nodes. This is essentially an array of 1023 list node structs. The node pool then returns pointers to available slots in this array in response to `request_node` calls.

But how does the node pool keep track of which array slots are available? It keeps a linked list of available array slots! The slots are used themselves to store the pointers necessary to form this linked list. Specifically, you should implement a slot as:

```
union slot_t {
    struct _node_t node;
    slot_t *next;
};
```

This allows you to use the memory of the slot to store either a list node (when it is reserved for this purpose by returning it to the user in response to a `request_node` call) or a pointer to the next available array slot if this slot is currently unoccupied, that is, it is part of the linked list of available array slots.

The `_node_pool_t` implementation could then look like this:

```
struct _node_pool_t {
    union slot_t slots[1023];
    slot_t *free_slots;
};
```

The `make_node_pool` function has to allocate memory for this structure (using `malloc`) and initialize the list of available slots by setting up pointers between them. The `destroy_node_pool` function simply frees the memory allocated to the node pool. The `request_node` function returns the first available slot in the list of slots starting with `free_slots` and updates `free_slots` to point to the next available slot in this list. You should return `NULL` if the list of available slots is empty. The `release_node` function makes the released node part of the list of free slots again by casting it to a `slot_t *`, setting its `next` pointer to point to the current first slot in the list of available slots, and updating `free_slots` to point to the returned slot.

For this step, create a header file `node_pool.h` that *declares* the above types and function prototypes and an implementation file `node_pool.c` that *defines* the above types and functions.

Bonus: For 25% bonus marks, you should support allocating an arbitrary number of list nodes, not just a fixed number. The strategy is as follows: Instead of allocating an array capable of holding 1023 list nodes, you start out with an empty node pool that does not have any space reserved for list nodes. The `free_slots` pointer should be `NULL` to indicate that currently no slots are available. Then, the `request_node` function should do one of two things:

- If `free_slots != NULL`, then it should reserve the next available slot as in the above implementation.
- If `free_slots == NULL`, which can happen at any time when all available slots are occupied, it should allocate a new array of 1023 slots, form a new list of available slots using these slots, and then return the first slot in this list.

This creates a small problem: When you destroy the pool, you need to free the memory occupied by all 1023-slot blocks you allocated. The strategy you should use to deal with this is to replace the `slots` array with a linked list of structs, each holding one of the 1023-slot blocks. You can then free the memory allocated by these blocks by traversing this linked list.

(This implementation has another problem: since you never free node blocks until the entire node pool is destroyed, even if none of the slots in a given block is occupied, the total space used by the node pool is the maximum number of nodes in the list over the lifetime of the list, rounded to the next

multiple of 1024. Thus, you may end up in a situation where a single peak usage that requires millions of nodes makes your list occupy a lot of space even though the list stores only a few hundred nodes most of the time. There are ways to mitigate this while keeping the cost per list operation constant, but this is well beyond the scope of this course.)

Step 4: Use the node pool in your linked list implementation

Modify your linked list implementation from Step 1 to use the node pool from Step 3 to manage the heap memory occupied by list nodes. This requires the following simple changes:

- You need to modify your `_list_t` structure so it also stores a reference of type `node_pool_t` to a node pool to be used by this list.
- Your `make_list` function should call `make_node_pool` to create a new node pool to be used by this list.
- Your `destroy_list` function also needs to destroy the node pool used by the given list by calling `destroy_node_pool`. If you think about it, you no longer need to release individual nodes as a result when destroying the list.
- All other operations that create or destroy list nodes should use `request_node` and `release_node` instead of `malloc` and `free` to create and destroy list nodes.