YSC2229 Introductory Data Structures and Algorithms

Midterm Project Report

Memory Allocator and dependent data structures

Author: Zhu Tianrui

Team Members: Li Wandan, Guo Ziting

Acknowledgement

This is a group project by Linda, Ziting and myself. My contributions to the project are: alloc function and some allocator tests, SLL and tests, SLL-based queue tests.

Introduction

At the fundamental level, this project builds a memory allocator that allows for storing and retrieving integers, strings, and pointers in the memory without using OCaml reference. Based on this memory allocator, we define data structures such as Doubly-linked list (DLL), Singly-linked list (SLL), DLL-based queue, and SLL-based queue and implement their associated functions.

Memory Allocator from Scratch

The data type of the memory is called **heap**. We choose to use **array** as the heap because we can retrieve any element in constant time given its index. We initially build three arrays for storing pointers, integers, and strings respectively, but after learning OCaml's algebraic data type, we decide to have only one array whose elements belong to a variant type. This way, we maximize the use of memory spaces in the heap.

To create an array where different types can be stored, we define a variant type for the array's elements. The type has five constructors: *Free, Init, P of ptr, I of in*t, and *S of string*. First, when the heap is just created, all elements are filled with *Free*, indicating that the cells in the heap are available for storing data. The memory cells can also take the form of an algebraic data type *P of ptr, I of int*, or *S of string*, signifying that the cell is currently being used to store the corresponding data type, which are pointer, integer, or string.

The **pointer** is a data type that specifies a location in the memory. Since the memory takes the form of an array, the pointer either carries the index of an array element or nothing. Therefore, we give it the type **int option** (Figure 1). With pointers, data in the memory can be linked as they can point to other data.

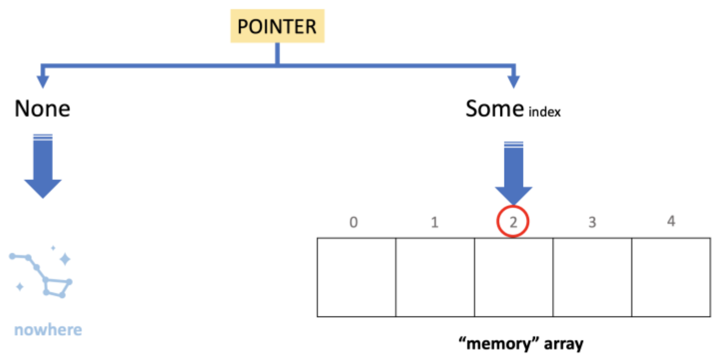

Figure 1. Pointer overview

Because data often come in a group (e.g. a key-value pair) where a specific number of values have to be stored next to each other for easy retrieval, we need to first traverse the entire heap to ensure there are consequtive memory cells that can be used to store these grouped data. Once this is done, these cells are initialized and filled with *Init*, the fifth constructor of the variant type of the heap elements.

The initialization of memory cells is done through the alloc function. The difficulty of its implementation is in finding a number of **consequtive** memory cells in the entire heap that are all marked *Free* so we do not waste any available cells. To do so, we implement two recursive funcitons. First, find traverses the heap and locates the first *Free* memory. Immediately after, count checks if the specific number of memory cells after the first find are all *Free*, and if so, returns the index of the first find. If it is not the case, to save time traversing the list, count passes the index of the first element within its radar that is not *Free* to find, and find will continue traversing the heap from this location onwards. Once a specific number of consequtive *Free* memory cells are found, all of them are marked *Init* and the location of the first *Init* cell in the sequence is returned in the form of a pointer (Fig. 2). This is to say, the heap, starting from the index where this pointer points to, is ready to store that specific number of values in sequence. One the other hand, an error is raised when alloc finds no consequtive memory cells of particular size (Fig. 3). We implement negative tests to check this behaviour of alloc by allocating more memory than the number of consequtive *Free* cells.

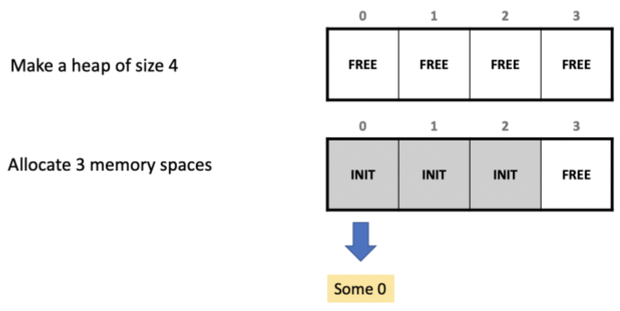

Figure 2. Make a heap and initialize memory

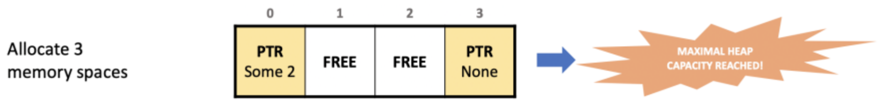

Figure 3. Alloc does not find adequate space

To assign values to memory cells, one provides the assign function with the pointer returned by alloc and an **offset** specifying how many cells after the first value is to store the next value (Fig.4). By the same logic, to retrive a value, one only needs to provide the deref function with a pointer and an offset (Fig.4). Because of the “pointer + offset” combination that is essential to all data storage and retrieval, the continuous chunk of memory cells that alloc finds in the first place works as a unit where all the data within this chunck is connected through their relative positions to the location of the first value. The location of the first value, i.e. the index of the element, is therefore functioning as the “identity index” of the entire the chunk of values. This property is important to the implementation of DLL and SLL as we can then construct linked nodes which contains pointers connecting itself to other nodes.

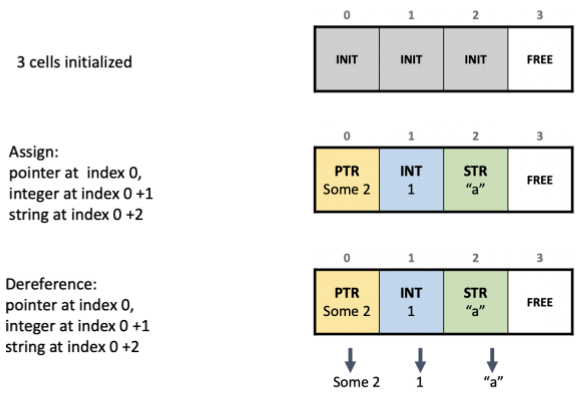

Figure 4. Assign values and dereference values.

Another important function is free. When the user decides a certain group of data is no longer needed, the free function is used to make these memory cells available for storage again. To free a chunk of data, one needs to provide the "identity index" of the first value in pointer form and the size of that chunk. The chunk will be marked *Free* so that the alloc function may identify it as a possible storage for data of the same size. In this way, we reuse these memory cells.

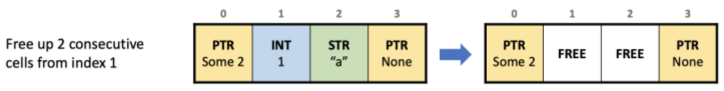

Figure 5. Free memory

Doubly-Linked List

The doubly-linked list takes full advantage of the **pointer** data type. In each node of a DLL, there are two pointers, one pointing to the previous nodes' "identity index" (i.e. index of the first node value) and the other pointing to that of the next node. Apart from the first node in the list which points to no previous node, and the last element pointing to no next node, all the other nodes in the list are connected to their previous and next neighbours. Creating a DLL that stores key-value pairs means first initializing chunks of 4 in the memory, assigning values to the memory heap, and connecting the nodes with their neighbours by changing their pointer values (Fig. 6).

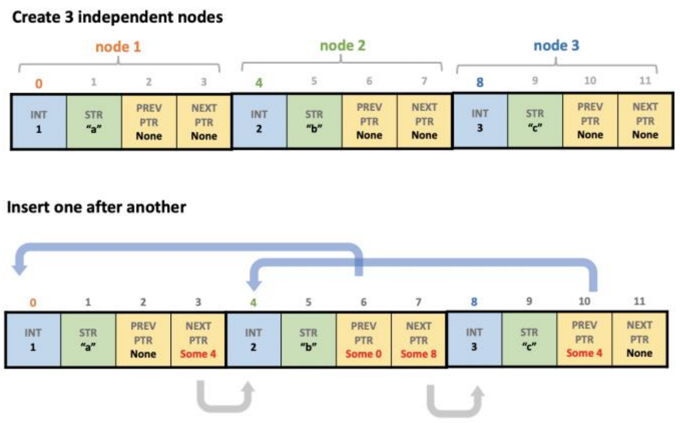

Figure 6. creating a doubly-linked list

To remove a node, one only needs to provide the “indentity index” of the node. the function remove utilizes Allocator functions free to clear the corresponding memory cells and assign to reconnect the “loose ends” (Fig.7). Because the alloc function is able to pick up any free spaces in the memory heap, the nodes of a DLL do not have to be stored in the order they are created (Fig.7).

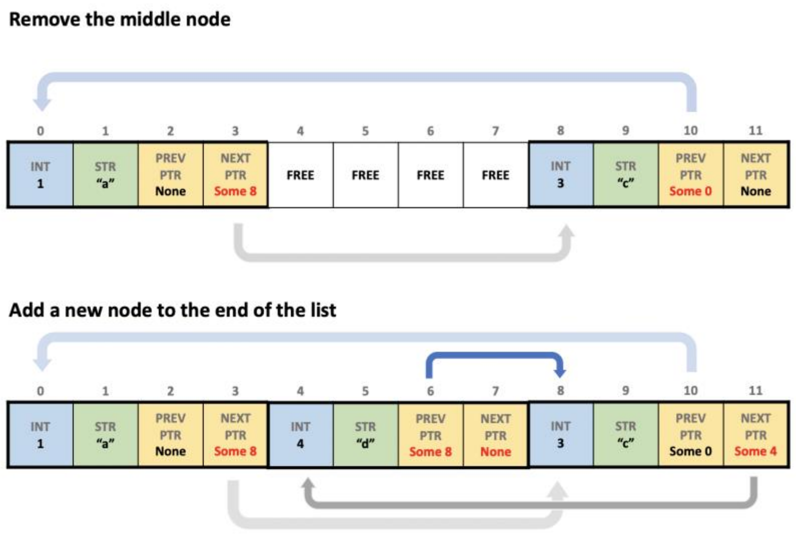

Figure 7. removing nodes and adding new node

DLL-based Queue

The DLL-based Queue makes use of the DLL to store elements in a way that is easy to retrieve (“dequeue”) the element that is put inside the queue first. This property requires the queue data structure to keep track of the head node (i.e. the earliest node). Thus, one cell in the heap is dedicated to storing a pointer that points to the “identity index” of the head node. Meanwhile, it is also necessary to keep track of the last node because the process of enqueueing requires modifying the new node’s “previous-pointer” to have the index of the last node in order to connect it to the body of the list. Therefore, we dedicate another cell in the heap to storing a pointer to the tail node. One can see the DLL-based queue as having three parts: a head pointer, a tail pointer , and a body made of a DLL. Thus, enqueueing and dequeueing essentially mean removing the head node in a DLL and adding a new node respectively.

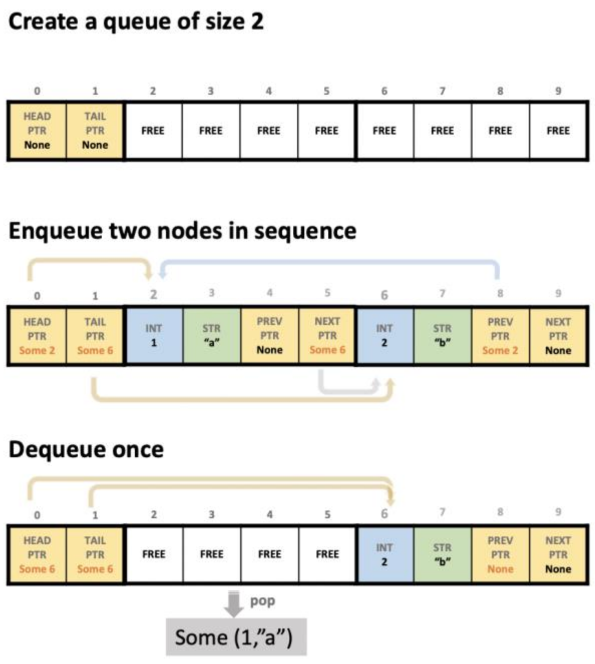

Figure 8. DDL-based queue operations

Singly-Linked List

The SLL is similar to DLL but with only a “next-pointer” in each node instead of a next and a previous pointer. The deletion and addition of nodes also work similar to DLL but without having to modify previous-pointers.

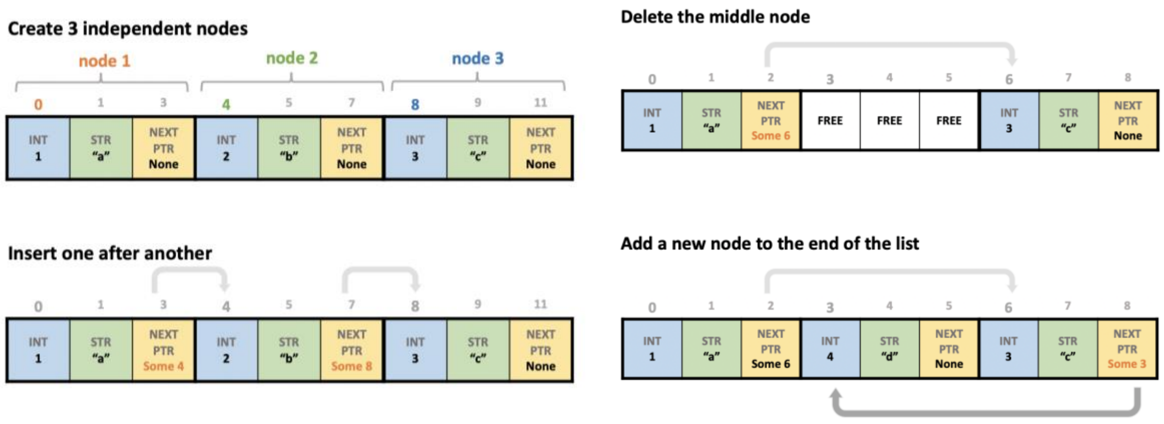

Figure 9. Left: creating a singly-linked list. Right: deleting and adding nodes.

SLL-based Queue

The implementation of SLL-based queue is different from DLL-based queue. Here we follow the instruction to implement it with two SLLs. We still use two memory cells for storing a head pointer and a tail pointer, but instead of pointing to the head node and the tail node of the entire queue, the **head pointer** now points to the root of a **head list** and the **tail pointer** to the root of a **tail list**. The tail list can be understood as the “body” of the queue where the latest node forms the **root** with a next-pointer pointing to the second latest node. Whenever we enqueue, we grow the tail list (Fig. 10). The head list, on the contrary, has the earliest node as its root, with a next-pointer pointing to the second earliest node. Whenever we dequeue, we reduce the head list. The purpose of the tail pointer is to keep track of the latest node, to which the nodes before it are to be connected to. The purpose of the head pointer is to keep track of the head node, the earliest node, which is to be poped out when we dequeue (Fig.11).

The dequeue operation first has to ensure there is a head list by checking if the head pointer is not None. Is it is None, then a head list has to be created by reversing the tail list (Fig. 11). This reversal will eliminate the tail list. But fear not, we can grow the tail list simply by enqueuing new nodes. Now with a head list, the dequeue function just needs to pop the root node which is the earliest node. The two SLL design can be confusing, but the purpose is really just to keep track of the head and tail.

One can imagine the tail list as a farmhouse where sheep are kept in a row of cubicles (tail list) where the oldest sheep is at the end of the row (tail node). When the demand for mutton is high, the sheep are taken out of the farmhouse, oldest first, and made to walk to the slaughterhouse in a line (head list). The oldest sheep (head node) is killed first (dequeue), followed by the second oldest, etc. Meanwhile, the farmer fills the empty cubicles (*Free* cells) with new lambs (enqueue), arranging them in a row by age (tail list).

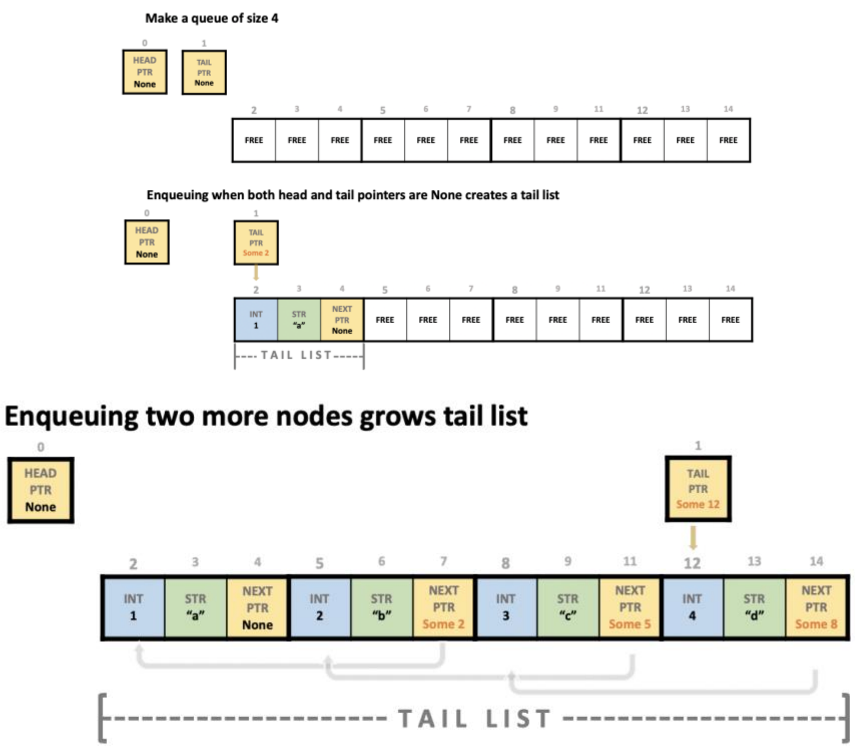

Figure 10. Making, enqueuing a SLL-based queue

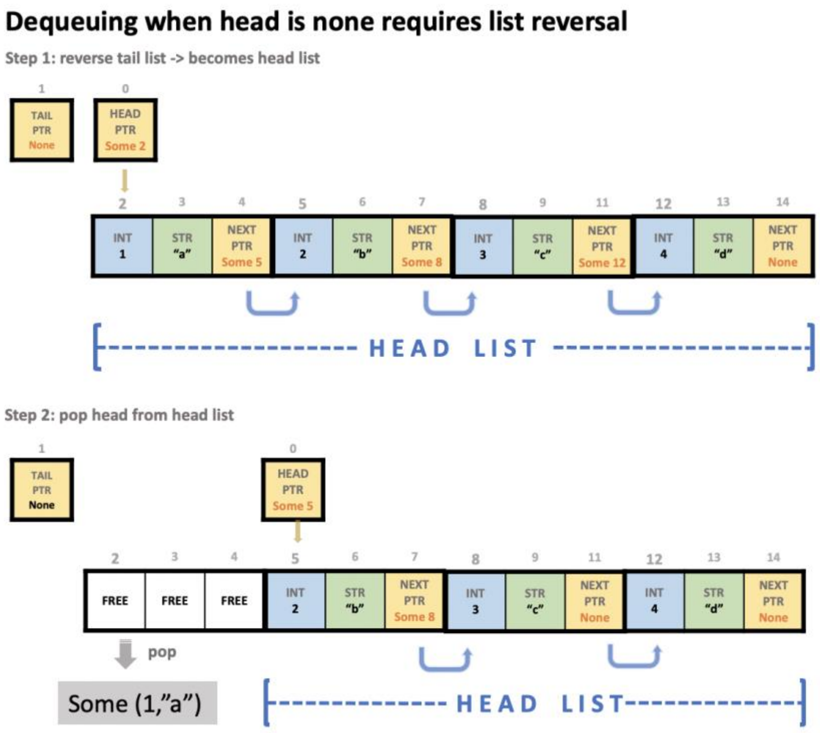

Figure 11. Dequeue when head list is non-existent