Linked lists in C

Ying Pei Lin

Fall 2024

A linked list

The following code snippet implements the methods of linked list in C with the defined structure, `cell` and `linked`.

The function `linked_add` adds an item to the first of the linked list, we create a temporary cell, assign the original first cell to the new cell's tail, and then assign the new cell to the first cell of the linked list.

```
void linked_add(linked *lnk, int item) {
  cell *new = (cell*)malloc(sizeof(cell));
  new->value = item;
  new->tail = lnk->first;
  lnk->first = new;
}
```

The function `linked_length` returns the length of the linked list. The linked list we used in this assignment is singly linked, therefore we can know if the list ends by checking whether the cell's tail is `NULL`. This method is very handy and will be used in many other linked list operations in the assignment. In the loop, we increment the length and move the current cell to the next cell until the current cell is `NULL`.

```
int linked_length(linked *lnk) {
  int len = 0;
  cell *curr = lnk->first;
  while (curr != NULL) {
    len++;
    curr = curr->tail;
  }
  return len;
}
```

The function `linked_find` returns a boolean value indicating whether the item is in the linked list. We iterate through the linked list in the same way as the `linked_length` function, and return when we find the item.

```c
bool linked_find(linked *lnk, int item) {
  cell *curr = lnk->first;
  while (curr != NULL) {
    if (curr->value == item)
      return true;
    curr = curr->tail;
  }
  return false;
}
```

Removing an item from the linked list is a bit more complicated, because we need to relink the rest of the list to the previous cell when we remove the current cell. Also, we need to handle the case when the first cell is the one to be removed, because the first cell has no previous tail, which we need to assign the rest of the list.

So, the precedure is to iterate through the linked list, if the current cell is the target, we check if the previous cell is `NULL`, before we replace the current cell with its tail.

```c
void linked_remove(linked *lnk, int item) {
  cell *curr = lnk->first;
  cell *prv = NULL;
  while (curr != NULL) {
    if (curr->value != item) {
      prv = curr;
      curr = curr->tail;
    } else {
      if(prv != NULL) {
        prv->tail = curr->tail; // Link the previous cell to the next cell.
      } else {
        lnk->first = curr->tail; // If the first cell is the one to be removed,
      }
      cell *tmp = curr->tail;
      free(curr);
      curr = tmp; // Move the current tail to the current cell.
    }
  }
}
```

Benchmarks of linked list operations

In this section, I measure the time taken to append a linked list to the end of another linked list. There are two set of experiments, one with the different size of the linked list to be appended to a fixed size of linked list, and the other with the fixed size of linked list to be appended to a different size of linked list.

The results are shown in the Figure 1. We can see that the time taken to append a linked list to the end of another linked list is affected much more by the size of the linked list to be appended than the size of the linked list to be appended to. In theorey, the time complexity should be $O(n)$, where n is the size of the linked list to be appended, because we need to iterate through the first linked list to find the end of the list.

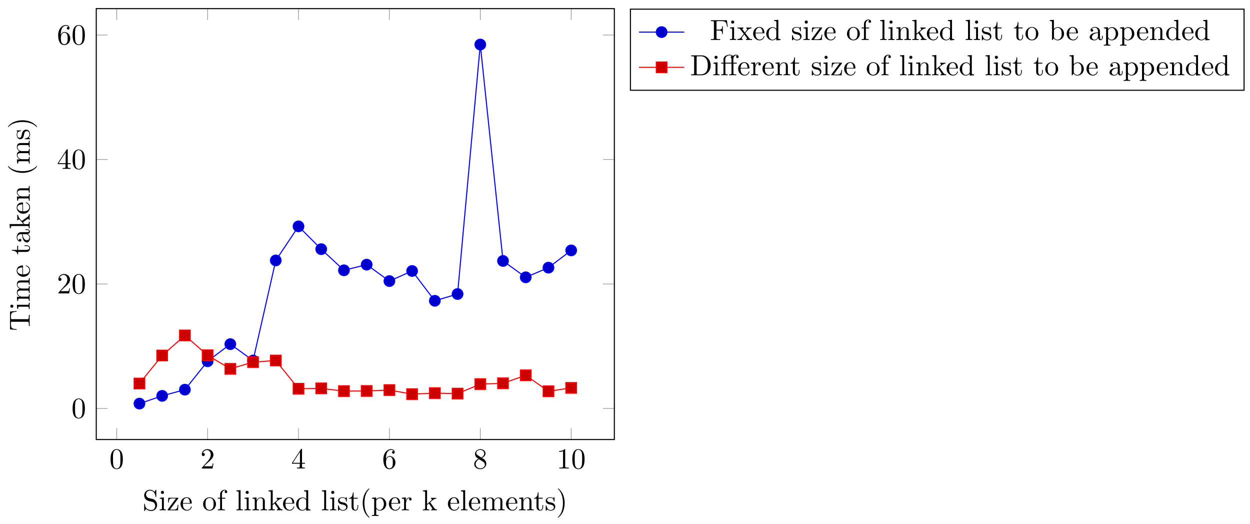

Figure 1: Time taken to append a linked list to the end of another linked list

Compare to array

To compare the performance of linked list with array, I first implement the same append operation with array. The function `array_append` takes two arrays and their sizes, and copy both the elements of both arrays to a new array.

```c
void array_append(int **a, int *size_a, int *b, int size_b) {
  int new_size = *size_a + size_b;
  int *new_array = (int*)malloc(new_size * sizeof(int));

  // Copy elements from array a
  for (int i = 0; i < *size_a; i++) {
    new_array[i] = (*a)[i];
  }
  // Copy elements from array b
  for (int i = 0; i < size_b; i++) {
    new_array[*size_a + i] = b[i];
  }
}
```

The results are shown in the Figure 2. We can see that it is much faster to append an array than a linked list, because we can directly access the memory location of the array, while we need to iterate through the linked list.

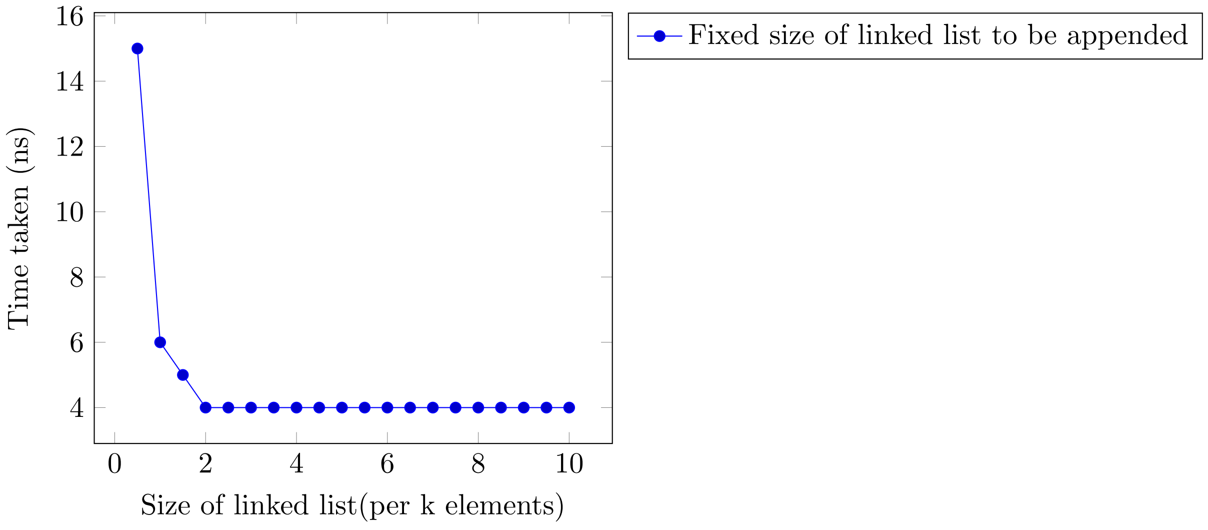

Figure 2: Time taken to append a array to another array

Stack using linked list

The code snippet below shows the implementation of the stack using linked list. The key difference between using linked list and array is that we can dynamically allocate memory for the linked list, and also we can access the memory location of the linked list in constant time.

When popping and pushing an item to the stack, both operations are $O(1)$, because we only need to access the first cell of the linked list. But when we need to use the stack dynamically, the resize operation of the array method is $O(n)$, while the linked list method is still $O(1)$.

```
stack *new_stack() {
  stack *stk = (stack*)malloc(sizeof(stack));
  stk->list = linked_create();
  return stk;
}
```

```c
void push(stack *stk, int val) {
  linked_add(stk->list, val);
}

int pop(stack *stk) {
  if (stk->list->first == NULL) {
    printf("Underflow\n");
    return -1;
  } else {
    int val = stk->list->first->value;
    cell *temp = stk->list->first;
    stk->list->first = stk->list->first->tail;
    free(temp);
    return val;
  }
}
```