**Faculty of Computer Science, Dalhousie University** *21-Nov-2018*

# CSCI 2132 — Software Development

# Lecture 30: Merge Sort with Linked Lists

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

**Previous Lecture**

- Heap (Free Store) continued
  - Efficient use of heap
  - Additional allocation functions
- Linked lists
- list.c example (student database) started

## 24.2 Merge Sort with Linked Lists

The previous code example shows how to implement some typical linked list operations in C: inserting nodes at the beginning of the list, deleting a node, and traversing the list.

We will not implement a more complex algorithm on the list: sorting the linked list. When we talked about the mergesort algorithm and compared it to quicksort, we mentioned that quicksort is in practice faster on arrays, but mergesort is faster on linked lists. We will now show how to implement the mergesort algorithm on linked lists using the same student database example. We will modify the code and add one more option to the original code, which is to sort the linked list of students according to the student number.

The following are the main steps of performing mergesort on a linked list:

1. Divide: Divide the n-element linked list to be sorted into two sub-lists of n/2 elements each
2. Conquer: Sort the two sub-lists recursively using mergesort
3. Combine: Merge the two sorted sub-lists to produce the sorted answer

An interesting issue is how to perform Step 1. This is not a problem at all with arrays since we can perform random access on arrays. How to divide a linked list into two halves? There are two solutions:

Solution 1:

1. Use one loop to count the number, n, of elements in the linked list
2. Start from the head, and follow the next pointer n/2 times to reach the head for the second sub-list

Solution 2:

1. Declare two pointers that both initially point to the head of the linked list
2. Move the second pointer towards the tail twice while moving the first pointer once
3. Repeat Step 2 until the second pointer reaches the tail. At this time, the first pointer points to a node in the middle of the list

Which solution is superior? The first solution performs two loops. The first loop iterates through the entire list, while the second loop iterates through half the list. In the second solution, one pointer iterates through the entire

list while another pointer iterates through half the list. There does not seem to be any major difference in the total number of nodes visited. However, in the first solution, we need perform arithmetic computations such as counting and comparing integers, while the second solution avoids arithmetic calculation. Thus, the second solution is likely superior. We cannot be sure since memory caching may be better used in the first solution, depending on the memory position of the nodes of the list. In the code example that we will see, we will use the Solution 2.

The fill-in-the-blanks code is available at:

```
~prof2132/public/sortlist.c-blanks
```

The program shares a lot of code with the previous program `list.c.`, so we will share here only the parts relevant to the addition of the merge sort:

```
/* Program: sortlist.c
   An example used in CSCI 2132 to illustrate the mergesort algorithm
   on linked lists.  It bulids on the example list.c.
*/
...

struct node {
  int number;
  char name[NAME_LEN+1];
  struct node* next;
};

...

/* Merge sort */
struct node* mergesort     (struct node* student_list);
struct node* merge         (struct node* list1, struct node *list2);

...

int main() {
  int option;
  struct node* student_list = NULL;

  for (;;) {
    printf("\n-- OPTIONS MENU -----------------\n");
    printf("1: Add a student\n");
    printf("2: Search for a student by number\n");
    printf("3: Delete a student by number\n");
    printf("4: Display all students\n");
    printf("5: Sort students by number\n");
    printf("0: Exit\n");
    printf("\n");

    printf("Enter an option: ");
    if ( scanf("%d", &option) != 1 ) {
      if ( feof(stdin) ) break;
      printf("Invalid option: "); line_copy(); pause();
      continue;
    }
```

```
    /* Read the rest of the line after option number.  Usually, it is
       just one new-line character */
    line_skip();

    if (option == 0) break;

    switch(option) {
    case 1: student_list = insert(student_list);    break;
    case 2: search(student_list);                   break;
    case 3: student_list = delete(student_list);    break;
    case 4: print_list(student_list);               break;
    case 5: student_list = mergesort(student_list); break;
    default:
      printf("Incorrect option: %d\n", option); pause();
    }
  }

  delete_list(student_list); /* Not necessary in this example */
  printf("Bye!\n");
  return 0;
}

struct node* mergesort(struct node* student_list) {
  struct node* list1 = student_list;
  struct node* list2 = student_list;

  if (student_list == NULL || student_list->next == NULL)
    return student_list;

  while ((list2 = list2->next) != NULL && (list2 = list2->next) != NULL)
    list1 = list1->next;

  list2 = list1->next;

  list1->next = _____________ ;
  list1 = student_list;

  list1 = mergesort(list1);
  list2 = mergesort(list2);

  return merge(list1, list2);
}

struct node* merge(struct node* list1, struct node* list2) {
  struct node *list, *prev;

  if (list1 == NULL) return list2;
  if (list2 == NULL) return list1;

  if (list1->number <= list2->number) {
    list = list1; list1 = list1->next;
```

```
  } else {
    list = list2; list2 = list2->next;
  }

  prev = list;

  while (list1 != NULL && list2 != NULL) {
    if (list1->number <= list2->number) {
      prev->next = list1;
      list1 = list1->next;
    } else {
      prev->next = list2;
      list2 = list2->next;
    }
    prev = _____________ ;
  }

  if (list1 != NULL)
    prev->next = list1;
  else
    prev->next = list2;

  return _____________;
}

...
```