Faculty of Computer Science, Dalhousie University *19-Nov-2018*

CSCI 2132 — Software Development

Lecture 29: Linked Lists

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

Previous Lecture

- Structures, arrow operator
- Dynamic memory allocation: malloc and free
- Heap (Free Store) started

23.1 Heap (Free Store)

When we talked about the functions `malloc` and `free`, we mentioned some low-level details about how the blocks of dynamic memory are allocated and freed on the heap. By the way, you will or may have already learned about a data structure called *heap*. It is useful to have in mind that the *heap* we discuss here, which is the heap memory of a process, and the other heap, which is the heap data structure, are very different concepts.

Let us look again in more details on the heap memory. As we previously learned, each process is given the four parts of memory: code, data, stack, and heap. We learned what is stored in code, data, and stack. The heap memory is used to store dynamically allocated objects, i.e., the data allocated with the `malloc` function or another similar function, and freed with the `free` function. We must provide the function `free` with the exact pointer to the block allocated by `malloc`. The function `free` knowns the size of the block to free because this size is recorded by the function `malloc`, and it is usually recorded at the memory address just before the starting address of the allocated block.

The heap memory is also called *free store*. A process starts usually with one block of heap memory. Whenever the function `malloc` is executed, a part of heap memory, called block, is marked as being used and its address is given to the process. When the function `free` is executed the block is returned to the pool of free blocks, which are typically maintained as a linked list of blocks. After many executions of malloc and free, the free memory on heap may become very fragmented, and `malloc` may need to spend significant time in searching for a free block of sufficient size. There are different strategies how to make this more efficient. During execution of a process, all heap memory may be used up, or there may just be no free block of sufficient size when `malloc` is called. In this case, `malloc` normally makes a system call to the operating system kernel to obtain more heap memory. This request usually asks for larger chunks of memory, called memory pages. The function `malloc` then adds these pages to the pool of free blocks and completes request. If even after this system call `malloc` cannot find a sufficiently large block, it will return `NULL`.

There are advantages and disadvantages of using memory from the heap. First, since the heap is a large pool of memory, large data, such as large arrays and structures, should be allocated in the heap. The stack may not have enough memory for these large data requests.

Second, dynamically allocated memory stays allocated until it is deallocated explicitly or the process terminates. Thus, a function can return a pointer to a dynamically allocated memory block, and its caller can make use of this memory space. While this is useful, it may also lead to memory leaks if the memory is not deallocated when it is no longer in use.

Finally, heap allocation is slower than stack allocation. All stack allocation for a function happens automatically when the function is called. Even variable length arrays are very efficiently allocated simply by taking more memory on the stack. Release of the memory is also fast, all stack memory is released when we return from a function and the stack-top register is moved back. On the other hand, heap over time of execution becomes frequently fragmented with free blocks linked in a linked list. Searching for an available free block may take some time, and may involve even a system call to request more heap memory. Freeing memory is relatively efficient since the free block can be just added to the beginning of the linked list of free blocks. However, if we want to avoid heap memory fragmentation, sometimes freeing memory involves execution of more instructions.

We should usually try to avoid allocating very small objects, like characters (type `char`) on the heap. This may lead to a lot of fragmentation and even waste of memory. Namely, first, remember that with every block allocated with `malloc` some extra memory is used for storing block size, and sometimes even some other internal information. Second, due to typical computer architecture, larger basic types, such as `int` and `double`, must be aligned properly in memory. For example, if a `double` uses 8 bytes, it usually means that it must start from an address divisible by 8. Otherwise, the hardware will generate a "Bus error" if we try to access it. In order to avoid these kind of errors, `malloc` typically allocates only memory sizes that are multiples of 8. This means that `malloc` may need to round up the actual size that we need. For example, if we ask for 1 byte for a character, `malloc` will allocate 8 bytes. This leads to something called internal memory fragmentation.

23.2 Additional Allocation Functions

There are two additional allocation functions, particularly designed for dynamic allocation of arrays: `calloc` and `realloc`. The function `calloc` is used to allocate memory for an array of objects, and its prototype is:

```
void *calloc(size_t nmemb, size_t size);
```

where `nmemb` is the number of elements in the array, and `size` is the size of one object of the array. An example of its use is:

```
a = calloc(n, sizeof(int));
```

We can notice that the same allocation is easily done with `malloc` simply by allocating `nmemb*size` memory; however, `calloc` does one more thing: it clears the memory by setting all elements to 0.

The second additional interesting function is `realloc`. This function changes the size of the allocated block. The values in the original block are saved, but the location of the original block may be changed. The prototype of the function `realloc` is:

```
void *realloc(void *ptr, size_t size);
```

where the pointer `ptr` is a pointer to currently allocated block by `malloc` or `calloc`, and `size` is the desired new size of the block. Generally, the `realloc` function will extend or shrink the block, and it will keep the values in the part which belongs to both the original and the new block. However, the location of the block may need to be changed, so if we have any pointers pointing to the elements of the old block, they need to be updated. The pointer to the new block is returned.

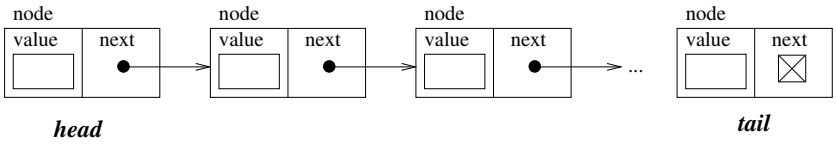
Here are some more details about `realloc`: If the block is expanded, the new part of the block is not initialized. If the block cannot be expanded, NULL is returned and the old block has not been changed or released. If the `NULL` pointer is given as an argument, than `realloc` behaves as the `malloc` function. If 0 is given as the second argument, then `realloc` behaves as the `free` function.

24 Linked Lists in C

Slide notes:

Linked Lists

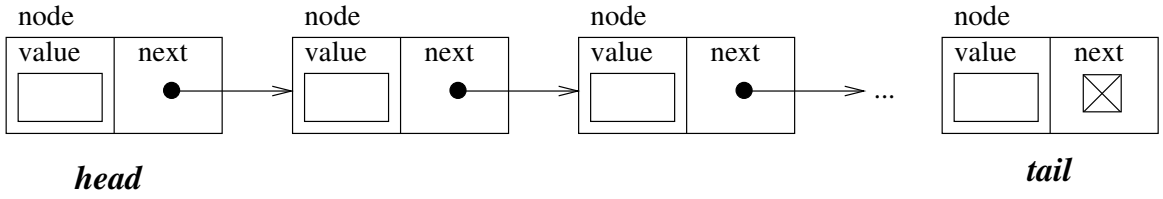
- A set of nodes linked in one direction
- Advantage over arrays:
  - Insertion and Deletion fast
- Disadvantage:
  - No random access
- Graphical representation:



- In C, use pointers to link elements

Let us use what we learned about structures and dynamic memory allocation to implement a linked list. A linked list is a data structure consisting of a set of nodes which represent a sequence together by distinguishing the first element, and each element containing a link to the next element. An array can also be used to represent a sequence. The main advantage of using a linked list is that the insertion and deletion of a data element in a linked list is fast, while the main advantage of arrays is that they provide fast random access to an element; i.e., an access based on the element's sequence number.

The following figure illustrates a linked list:



In C, we can use a pointer to the next node in each node to chain the nodes together. Hence, we implement pointers as C structures that contain a member that is the pointer to the next done, and some additional members that contain node content. The pointer member in the last node stores a NULL pointer. For example, if we store an `int` value in each node, we can declare the following node type:

```
struct node {
  int value;
  struct node *next;
};
```

To create an empty list, we can use the following definition:

```
struct node *list = NULL;
```

This list pointer will always point to the head of the linked list. This way our program can access each node, and we can avoid memory leaks.

24.1 Linked List Example: Student Database

To show how to implement fundamental algorithms on a linked list, let us study an example that uses a linked list to implement a simple student database.

The source code can be found at:

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

and it is included here as well:

```
/* Program: list.c
   An example used in CSCI 2132 to illustrate a use of linked list to
   store student records, which include student number and student
   name.
*/
#include <stdio.h>
#include <stdlib.h>

#define NAME_LEN 25

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

/* The functions to manage the linked list.  The functions prompt the
   user and read the standard input if needed. */
struct node* insert        (struct node* student_list);
void         print_student (struct node* student);
void         print_list    (struct node* student_list);
void         search        (struct node* student_list);
struct node* delete        (struct node* student_list);
void         delete_list   (struct node* student_list);

                                    /* Auxiliary functions */
int read_line(char line[], int len); /* Read at most len characters
                                        from the standard input and
                                        ignore the rest of the line. */
int line_skip(); /* Read the standard input to the end of the line. */
int line_copy(); /* Read the standard input to the end of the line
                    and copy to the standard output. */
int pause();     /* Ask user to press Enter to continue. */

int main(void) {
  int option, ch;
  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("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;
    default:
      printf("Incorrect option: %d\n", option); pause();
    }
  }

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

struct node* insert(struct node* student_list) {
  struct node* student = malloc(sizeof(struct node));
  /* Why would it be incorrect to use "struct node student;" ? __ */

  if (student == _________ ) {
    printf("Out of memory for a new student!\n"); pause();
    return student_list;
  }

  printf("\nAdding a new student\n");
  printf("Enter student's number: ");
  if (scanf("%d", &student->number) != 1) {
    printf("Incorrect student number: ");
    line_copy(); pause();
    ___________ ;           /* ?? Warning: memory leak risk */
    return student_list;
  }
  line_skip();                  /* to skip the newline character */
```

```
  printf("Enter student's name: ");
  read_line(student->name, NAME_LEN);

  student->next = student_list;
  printf("Student %d added.\n", student->number); pause();

  return student;
}

void print_student(struct node* student) {
  printf("Number:%3d  Name: %s\n", student->number, student->name);
}

void print_list(struct node* student_list) {
  printf("\nStudent List:\n");
  while (student_list != NULL) {
    print_student(student_list);
    student_list = student_list->next;
  }
  pause();
}

void search(struct node* student_list) {
  int number;

  printf("Enter student number: ");
  if (scanf("%d", &number) != 1) {
    printf("Incorrect student number: ");
    line_copy(); pause();
    return;
  }
  line_skip();

  while (student_list != NULL && number != student_list->number)
    student_list = ____________________________;

  if (student_list == NULL)
    printf("Not found.\n");
  else
    print_student(student_list);
  pause();
}

struct node* delete(struct node* student_list) {
  int number;
  struct node *prev, *cur;

  printf("Enter student number: ");
  if (scanf("%d", &number) != 1) {
    printf("Incorrect student number: "); line_copy(); pause();
    return student_list;
  }
```

```
  line_skip();

  for (cur = student_list, prev = NULL;
       cur != NULL && cur -> number != number;
       prev = cur, cur = cur->next)
    ;

  if (cur == NULL) {
    printf("Student not found!\n"); pause();
    return student_list;
  }

  if (prev == NULL)
    student_list = student_list->next;
  else
    prev->next = _____________;

  free(cur);
  return student_list;
}

void delete_list(struct node* student_list) {
  struct node* temp;

  while (student_list != NULL) {
    temp = student_list;
    student_list = student_list->next;
    free(__________);
  }
}

/*************************************************************************/
/*                        Auxiliary functions                            */
/*************************************************************************/
int read_line(char line[], int len) {
  int ch, i = 0;

  while ((ch = getchar()) != '\n' && ch != EOF)
    if (i < len)
      line[i++] = ch;

  line[i] = '\0';

  return i;
}

int line_skip() {
  int ch;
  while ( (ch=getchar()) != '\n' && ch != EOF )
    ;
  return ch != EOF;
}
```

```
int line_copy() {
  int ch;
  while ( (ch=getchar()) != '\n' && ch != EOF )
    putchar(ch);
  putchar('\n');
  return ch != EOF;
}

int pause() {
  printf("Press Enter to continue...");
  return line_skip();
}

/************************** Program End **************************/
```