**EEE2020-03** **Spring 2021**

# Assignment 3: Linked List

***Due: Sunday, May 2, 2021, 11:59PM***

## 1 Introduction

- The objective of this assignment is to implement a linked list class, or simply list.
- In particular, you will have to complete the missing parts of `class list_t` based on the lecture slides of `3_lists.pdf`.
- In Ubuntu or Mac terminal, use `wget` to download a copy of skeleton code compressed into `list.tar`.

```
$ wget https://icsl.yonsei.ac.kr/wp-content/uploads/list.tar
```

- Decompress the `tar` file, and go to the `list/` directory. Then, type `ls` to find the following list of files.

```
$ tar xf list.tar
$ cd list/
$ ls
data.h   func.h  input.cc  list.cc  main.cc   node.h
func.cc  input   input.h   list.h   Makefile  tar.sh
```

- From the listed files, `list.cc` is the only file you will have to work on in this assignment.
- You are allowed to change other files for your own testing and validation, but they will revert to the original state when your assignment is graded.
  - `func.*` files define several functions used in `main()`.
  - `input.*` files define `class input_t` that handles file operations. This class loads a file named `input` that contains 100 random integer numbers. The numbers will be used in `main()` to test various functions of linked list.
  - `list.*` are linked list files.
  - `main.cc` has the `main()` function.
  - `node.h` defines a linked list node, `class node_t`.
- To compile the code, you can simply type `make` in the terminal. `Makefile` that comes along with the skeleton code has automated all the compiling scripts for you.

```
$ make
g++ -Wall -Werror -g -o list.o  -c list.cc
g++ -Wall -Werror -g -o main.o  -c main.cc
g++ -Wall -Werror -g -o input.o -c input.cc
g++ -Wall -Werror -g -o func.o  -c func.cc
g++ -o list list.o main.o input.o func.o
```

- Executing the skeleton code should print the following output.

```
$ ./list
size()=3
list=[ 17 8 11 ]

size()=3
```

```
list=[ 17 8 11 ]

size()=3
list=[ 17 8 11 ]

size()=0
list=[ ]

size()=0
list=[ ]

size()=0
list=[ ]

size()=0
list=[ ]
```

- To clean up the directory and rebuild the program, you can type `make clean` and then `make` again.

```
$ make clean
rm -f list.o main.o input.o func.o list

$ make
g++ -Wall -Werror -g -o list.o  -c list.cc
g++ -Wall -Werror -g -o main.o  -c main.cc
g++ -Wall -Werror -g -o input.o -c input.cc
g++ -Wall -Werror -g -o func.o  -c func.cc
g++ -o list list.o main.o input.o func.o

$ ./list
size()=3
list=[ 17 8 11 ]

...
```

## 2 Implementation

- In `node.h`, you will find the definition of `class node_t`.
- The node implementation is self-contained in the header and not split to a source file. The header is complete, and you do not have to touch anything in this file.
- `class node_t` is a set of `*next`, `*prev`, and `data`. The first two are pointers to the next and previous nodes in a linked list, and the last variable is the actual data stored in the node.
- `class node_t` has a number of operators to facilitate node traversals in the linked list.
- The dereference and address operators (i.e., `operator*` and `operator&`) must be called when the node is properly connected to the linked list. Otherwise, the code throws a floating node exception for broken links.

```
/* node.h */

#ifndef __NODE_H__
#define __NODE_H__

#include <iostream>
#include "data.h"

```

```
class node_t{
friend class list_t;
public:
    // Node constructors
    node_t(void) : next(this), prev(this) { }
    node_t(const data_t &m_data) : next(0), prev(0), data(m_data) { }

    // Dereference operator
    data_t& operator*(void)  { return this_ptr()->data; }
    // Address operator
    node_t* operator&(void)  { return this_ptr(); }
    // Prefix increment operator
    node_t& operator++(void) { return *this = *next; }
    // Prefix decrement operator
    node_t& operator--(void) { return *this = *prev; }
    // Postfix increment operator
    node_t& operator++(int)  { *this = *next; return *prev; }
    // Postfix decrement operator
    node_t& operator--(int)  { *this = *prev; return *next; }
    // Forward operator
    node_t& operator+(int v) { node_t *n = this; while(v--) { n = n->next; }
                               return *n; }
    // Backward operator
    node_t& operator-(int v) { node_t *n = this; while(v--) { n = n->prev; }
                               return *n; }
    // Equal operator
    bool operator==(const node_t &m_node) {
        return (m_node.next == next) && (m_node.prev == prev);
    }
    // Not-equal operator
    bool operator!=(const node_t &m_node) {
        return (m_node.next != next) || (m_node.prev != prev);
    }

private:
    // Get the pointer of this node.
    node_t* this_ptr(void) {
        try{
            if(!next || !prev || (*(next->prev) != *this) ||
              (*(prev->next) != *this)) {
                throw "Floating node exception: broken node links";
            }
        } catch(const char *msg) { std::cerr << msg << std::endl; exit(1); }
        return next->prev;
    }

    node_t *next;       // Pointer to the next node
    node_t *prev;       // Pointer to the previous node
    data_t data;        // Node data
};

#endif
```

- The next shows `list.h` that defines `class list_t`. This header file is also complete, and you do not have to touch anything in it.
- `class list_t` has two member variables, `*sentinel` and `num_elements`. The first variable is a pointer to the sentinel node of linked list, and the second indicates how many nodes there are in the list not including the sentinel node.

```
/* list.h */

#ifndef __LIST_H__
#define __LIST_H__

#include "node.h"

class list_t {
public:
    // Constructor
    list_t(void);
    // Copy constructor
    list_t(const list_t &m_list);
    // Destructor
    ˜list_t(void);

    // Get the number of elements in the list.
    size_t size(void) const;
    // Get the first node in the list.
    node_t begin(void) const;
    // Get the next of last node in the list.
    node_t end(void) const;
    // Assign contents to the list, and replace the existing list.
    list_t& operator=(const list_t &m_list);
    // Remove all elements in the list.
    void clear(void);
    // Add a new element at the end of list.
    void push_back(const data_t m_data);
    // Remove the last element in the list.
    void pop_back(void);
    // Add a new element at the beginning of list.
    void push_front(const data_t m_data);
    // Remove the first element in the list.
    void pop_front(void);
    // Insert a new element at the given location.
    void insert(node_t m_node, const data_t m_data);
    // Remove an element at the given location.
    void erase(node_t m_node);
    // Merge two lists. m_list becomes empty.
    void merge(list_t &m_list);

private:
    node_t *sentinel;       // Pointer to the sentinel node
    size_t num_elements;    // Number of elements in the list
};

#endif
```

- The next shows the contents of `list.cc` that has the actual implementation of `class list_t`.
- In the skeleton code, class constructor, copy constructor, and destructor are already implemented. The constructor sets `num_elements = 0` and creates a sentinel node. The copy constructor creates a sentinel node and pushes back the data copied from the other list, since the list has to be created as a copy of the other list. The destructor deletes all nodes including the sentinel node.
- In addition, several simple functions such as `size()`, `begin()`, and `end()` are also provided. For your reference, `push_back()` and `erase()` functions are also implemented in the skeleton code. Take a close look at these functions to get ideas about how to implement other missing functions.

- It must be obvious in the skeleton code to find which functions are incomplete. The function bodies are marked with `/* Assignment */`. Fill in the functions to complete the homework.

```
/* list.cc */

#include <cstdlib>
#include "list.h"

// Constructor
list_t::list_t(void) :
    num_elements(0) {
    // Create a sentinel node.
    sentinel = new node_t();
}

// Copy constructor
list_t::list_t(const list_t &m_list) :
    num_elements(0) {
    // Create a sentinel node.
    sentinel = new node_t();
    // Copy data.
    for(node_t *node = m_list.sentinel->next;
        node != m_list.sentinel; node = node->next) {
        push_back(node->data);
    }
}

// Destructor
list_t::~list_t(void) { clear(); delete sentinel; }

// Get the number of elements in the list.
size_t list_t::size(void) const { return num_elements; }

// Get the first node in the list.
node_t list_t::begin(void) const { return *(sentinel->next); }

// Get the next of last node in the list.
node_t list_t::end(void) const { return *sentinel; }

// Add a new element at the end of list.
void list_t::push_back(const data_t m_data) {
    // Create a new node.
    node_t *node = new node_t(m_data);
    num_elements++;
    // Add the node before the sentinel.
    node->next = sentinel;
    node->prev = sentinel->prev;
    sentinel->prev->next = node;
    sentinel->prev = node;
}

// Remove an element at the given location.
void list_t::erase(node_t m_node) {
    node_t *m_node_ptr = &m_node;
    // In case of empty list, never remove the sentinel node.
    if(m_node_ptr != sentinel) {
        // Remove the node from the list.
        m_node_ptr->prev->next = m_node_ptr->next;
        m_node_ptr->next->prev = m_node_ptr->prev;
        // Delete the disconnected node.
        delete m_node_ptr;
        num_elements--;
    }
}





/************************
 * EEE2020: Assignment 3 *
 ************************/

// Assign contents to the list, and replace the existing list.
list_t& list_t::operator=(const list_t &m_list) {
    /* Assignment */
    return *this;
}

// Remove all elements in the list.
void list_t::clear(void) {
    /* Assignment */
}

// Remove the last element in the list.
void list_t::pop_back() {
    /* Assignment */
}

// Add a new element at the beginning of list.
void list_t::push_front(const data_t m_data) {
    /* Assignment */
}

// Remove the first element in the list.
void list_t::pop_front(void) {
    /* Assignment */
}

// Insert a new element at the given location.
void list_t::insert(node_t m_node, const data_t m_data) {
    /* Assignment */
}

// Merge two lists. m_list becomes empty.
void list_t::merge(list_t &m_list) {
    /* Assignment */
}

/********************
 * End of Assignment *
 ********************/
```

- The first incomplete function in `list.cc` is `operator=`. The operator assigns new contents to the list as a copy of another list, `m_list`. If the list previously had a list of nodes, they must be deleted. After the assignment, the list will have the same set of elements as the other list.

- `clear()` deletes all element nodes in the list but the sentinel node.

- `pop_back()` removes the last element node in the linked list. This function must leave the sentinel node even when the list becomes empty. The sentinel node will be deleted only by the class destructor.
- `insert()` adds a new element at the position specified by `m_node`. Since `m_node` is simply a copy of node, you must call the memory operator `&` of `node_t` to retrieve a pointer to the node, such as `node_t *m_node_ptr = &m_node`. Refer to the provided `erase()` function as an example. The rest of function is about reconnecting the pointers of neighboring nodes so that the new node gets inserted into the list.
- `merge()` combines two linked lists. In particular, element nodes of `m_list` are moved to the end of current list. This function does not delete or recreate nodes. It simply plays with node pointers to combine the lists. After merging, `m_list` will become an empty list only left with its sentinel node.
- Lastly, `main()` looks as follows, and it tests various functions of `class list_t`.
- Your are allowed to change `main()` to further test and validate your code between the lines of `* TEST YOUR CODE HERE FOR VALIDATION *` and `* END OF TESTING *`. But, note that this file will revert to the original state when your assignment is graded.

```
/* main.cc */

#include "input.h"
#include "func.h"
#include "list.h"

int main(int argc, char **argv) {
    // Define a linked list.
    list_t list;

    /**************************************
     * TEST YOUR CODE HERE FOR VALIDATION *
     **************************************/
    list.push_back(17);
    list.push_back(8);
    list.push_back(11);
    list.push_front(-82);
    list.push_front(-179);
    list.push_front(-41);
    print(list);

    list.insert(list.begin(), 2);
    list.insert(list.begin()+1, 27);
    list.insert(list.begin()+4, 19);
    list.insert(list.end(), -35);
    list.insert(list.end()-1, -94);
    list.insert(list.end()-3, 101);
    print(list);

    list.pop_back();
    list.pop_back();
    list.pop_front();
    print(list);

    list.erase(list.begin());
    list.erase(list.begin()+2);
    list.erase(list.end()-1);
    list.erase(list.end()-3);
    print(list);

    list_t cp = list;
```

```
print(cp);

cp.clear();
list = cp;
print(list);

cp.push_back(0);
cp.push_front(57);
list.insert(list.begin(), 90);
list.insert(list.end(), -4);
cp.merge(list);
print(cp);

/******************
 * END OF TESTING *
 ******************/




/* *************************************************
 * WARNING: DO NOT MODIFY THE CODE BELOW THIS LINE *
 *************************************************/

// Proceed?
if(argc != 2) { return 0; }

// Empty the linked list.
debug("Test #1: clear()");
list.clear();
std::cout << "list:" << std::endl; print(list);

// Add elements read from the input file to the linked list.
input_t input(argv[1]);
debug("Test #2: push_back()");
for(size_t i = 0; i < input.size(); i++) { list.push_back(input[i]); }
std::cout << "list:" << std::endl; print(list);

// Split the list into positive- and negative-number lists.
list_t plist, nlist;
debug("Test #3: insert(), erase()");
split(list, plist, nlist);
print(list, plist, nlist);

// Use merge functions to combine the lists.
debug ("Test #4: merge(), operator=, clear()");
nlist.merge(list);
list = nlist;
nlist.clear();
list.merge(plist);
print(list, plist, nlist);

// Split the list again to positive- and negative-number lists.
split(list, plist, nlist);

// Move negative nubmers from nlist to list.
debug("Test #5: pop_back(), push_front()");
while(nlist.size()) {
        data_t val = *(--(nlist.end()));
        nlist.pop_back();
        list.push_front(val);
    }
    print(list, plist, nlist);

    // Move positive numbers from plist to list.
    debug("Test #6: pop_front(), push_back()");
    while(plist.size()) {
        data_t val = *(plist.begin());
        plist.pop_front();
        list.push_back(val);
    }
    print(list, plist, nlist);

    return 0;
}
```

- Your assignment is graded based on what you get from the latter half part of `main()`. Results before the line of `* WARNING: DO NOT MODIFY THE CODE BELOW THIS LINE *` will be not considered for grading.
- **Test #1:** The first part of the test code clears `list` by calling `list.clear()`.
- **Test #2:** Then, the code reads a file (i.e., `input` in the `list/` directory) that contains 100 random integer numbers between -100 and 100. File handling is already implemented, so you can disregard how it works. The numbers in the file are pushed back to the list using `push_back()` function. Since `push_back()` is already implemented in the skeleton code, **Test #2** must show the correct result regardless of your implementations.
- **Test #3:** It calls the `split()` function implemented in `func.cc`. This function moves the numbers in `list` to two other lists, `plist` and `nlist`. After split, the first list will contain only positive numbers, and the second one will have only negative numbers. The original list (i.e., `list`) will be left only with one value, i.e., zero. The `split()` function uses `insert()` and `erase()` functions to move the numbers. If `insert()` is correctly implemented, `plist` and `nlist` will have the numbers in sorted manner.
- **Test #4:** This part of code tests `merge()` and `operator=`. If correctly implemented, all numbers are moved back to `list`, and the numbers will show up in ascending order.
- **Test #5:** It splits `list` once again to `plist` and `nlist`. `pop_back()` and `push_front()` functions are used to move all negative numbers to `list`. `nlist` will become empty, but `plist` still has the positive numbers.
- **Test #6:** It uses `pop_front()` and `push_back()` functions to move all positive numbers to `list`. Finally, `list` will have 100 integer numbers in ascending order, and `plist` and `nlist` will become empty.
- If `class list_t` is correctly implemented, executing the code will produce the following output.

```
$ ./list
size()=6
list=[ -41 -179 -82 17 8 11 ]

size()=12
list=[ 2 27 -41 -179 19 -82 17 8 101 11 -94 -35 ]

size()=9
list=[ 27 -41 -179 19 -82 17 8 101 11 ]

size()=5
list=[ -41 -179 -82 8 101 ]

size()=5
```

```
list=[ -41 -179 -82 8 101 ]

size()=0
list=[ ]

size()=4
list=[ 57 90 0 -4 ]
```

- To run the program with the `input` file, add `input` to the command as follows.

```
$ ./list input

...

----------------
Test #1: clear()
----------------
list:
size()=0
list=[ ]

--------------------
Test #2: push_back()
--------------------
list:
size()=100
list=[ -41 35 -75 98 39 -31 0 -88 89 73 -46 65 -87 66 -39 -38 44 32 -82 -7 -18
-22 8 -78 -20 -6 -83 81 31 -11 -30 -36 -70 -37 15 14 -65 11 -15 29 -72 -86 21
-33 57 -29 -9 -96 77 91 42 87 -90 -67 -51 -63 99 36 -91 94 -8 -13 -23 -25 18 17
43 45 -49 41 -35 -5 3 -52 90 -95 70 -50 6 -28 -59 -54 71 -79 -57 38 85 34 27 16
72 2 -1 10 47 19 -14 -3 -85 -77 ]

--------------------------
Test #3: insert(), erase()
--------------------------
list:
size()=1
list=[ 0 ]

plist:
size()=45
list=[ 2 3 6 8 10 11 14 15 16 17 18 19 21 27 29 31 32 34 35 36 38 39 41 42 43 44
45 47 57 65 66 70 71 72 73 77 81 85 87 89 90 91 94 98 99 ]

nlist:
size()=54
list=[ -96 -95 -91 -90 -88 -87 -86 -85 -83 -82 -79 -78 -77 -75 -72 -70 -67 -65
-63 -59 -57 -54 -52 -51 -50 -49 -46 -41 -39 -38 -37 -36 -35 -33 -31 -30 -29 -28
-25 -23 -22 -20 -18 -15 -14 -13 -11 -9 -8 -7 -6 -5 -3 -1 ]

------------------------------------
Test #4: merge(), operator=, clear()
------------------------------------
list:
size()=100
list=[ -96 -95 -91 -90 -88 -87 -86 -85 -83 -82 -79 -78 -77 -75 -72 -70 -67 -65
-63 -59 -57 -54 -52 -51 -50 -49 -46 -41 -39 -38 -37 -36 -35 -33 -31 -30 -29 -28
-25 -23 -22 -20 -18 -15 -14 -13 -11 -9 -8 -7 -6 -5 -3 -1 0 2 3 6 8 10 11 14 15 16
17 18 19 21 27 29 31 32 34 35 36 38 39 41 42 43 44 45 47 57 65 66 70 71 72 73 77
81 85 87 89 90 91 94 98 99 ]

plist:
size()=0
list=[ ]

nlist:
size()=0
list=[ ]

---------------------------------
Test #5: pop_back(), push_front()
---------------------------------
list:
size()=55
list=[ -96 -95 -91 -90 -88 -87 -86 -85 -83 -82 -79 -78 -77 -75 -72 -70 -67 -65
-63 -59 -57 -54 -52 -51 -50 -49 -46 -41 -39 -38 -37 -36 -35 -33 -31 -30 -29 -28
-25 -23 -22 -20 -18 -15 -14 -13 -11 -9 -8 -7 -6 -5 -3 -1 0 ]

plist:
size()=45
list=[ 2 3 6 8 10 11 14 15 16 17 18 19 21 27 29 31 32 34 35 36 38 39 41 42 43 44
45 47 57 65 66 70 71 72 73 77 81 85 87 89 90 91 94 98 99 ]

nlist:
size()=0
list=[ ]

---------------------------------
Test #6: pop_front(), push_back()
---------------------------------
list:
size()=100
list=[ -96 -95 -91 -90 -88 -87 -86 -85 -83 -82 -79 -78 -77 -75 -72 -70 -67 -65
-63 -59 -57 -54 -52 -51 -50 -49 -46 -41 -39 -38 -37 -36 -35 -33 -31 -30 -29 -28
-25 -23 -22 -20 -18 -15 -14 -13 -11 -9 -8 -7 -6 -5 -3 -1 0 2 3 6 8 10 11 14 15 16
17 18 19 21 27 29 31 32 34 35 36 38 39 41 42 43 44 45 47 57 65 66 70 71 72 73 77
81 85 87 89 90 91 94 98 99 ]

plist:
size()=0
list=[ ]

nlist:
size()=0
list=[ ]
```

- Even if your code compiles and runs to completion with correct results, you need to double-check the code via `valgrind` to confirm no memory leaks.

```
$ valgrind ./list input

...

==2020== All heap blocks were freed -- no leaks are possible

...
```

- In Mac OS, `valgrind` is not well supported. You may use `leaks` instead to test memory leaks.

```
$ leaks --atExit -- ./list input

...

Process 2020: 0 leaks for 0 total leaked bytes.

...
```

## 3 Submission

- When the assignment is done, execute the `tar.sh` script in the `list/` directory.
- It will compress the `list/` directory into a `tar` file named after your student ID such as `2020140000.tar`.

```
$ ./tar.sh
rm -f list.o main.o input.o func.o list
list/
list/list.h
list/main.cc
list/func.h
list/Makefile
list/list.cc
list/node.h
list/func.cc
list/data.h
list/input
list/input.cc
list/tar.sh
list/input.h

$ ls
2020140000.tar  func.cc  input     input.h  list.h   Makefile  tar.sh
data.h          func.h   input.cc  list.cc  main.cc  node.h
```

- Upload the tar file (e.g., `2020140000.tar`) on LearnUs. Do not rename the `tar` file or C++ files included in it.

## 4 Grading Rules

- The following is the general guideline for grading. 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change, and the grader may add a few extra rules for fair evaluation of students' efforts.

  **-3 points:** A submitted `tar` file is renamed and includes redundant tags such as `hw3`, student name, etc.

  **-5 points:** A program code does not have sufficient amount of comments. Comments in the skeleton code do not count. You must make an effort to clearly explain what each part of your code intends to do.

  **-5 points each:** There are total six sets of testing in `main()`, such as `Test #1: clear()`, ··· For each incorrect block, 5 points will be deducted.

  **-15 points:** The code compiles and runs, but it has memory leaks or crashes at the end with errors.

  **-25 points:** The code does not compile or fails to run (e.g., no or completely wrong results), but it shows substantial efforts to complete the assignment.

  **-30 points:** No or late submission. The following cases will also be regarded as no submissions.

    - ∗ Little to no efforts in the code (e.g., submitting nearly the same version of code as the skeleton code) will be regarded as no submission. Even if the code is incomplete, you must make substantial amount of efforts to earn partial credits.
    - ∗ Fake codes will be regarded as cheating attempts and thus not graded. Examples of fake codes are i) hard-coding a program to print expected outputs to deceive the grader as if the program is correctly running, ii) copying and pasting random stuff found on the Internet to make the code look as if some efforts are made. More serious penalties may be considered if students abuse the grading rules.

  **Final grade = F:** The submitted code is copied from someone else. All students involved in the incident will be penalized and given F for the final grades irrespective of assignments, attendance, etc.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect for any reasons, feel free to discuss your concerns with the TA. In case no agreement is made between you and the TA, elevate the case to the instructor to review your assignment. Refer to the course website for the contact information of TA and instructor: `https://icsl.yonsei.ac.kr/eee2020`

- Begging partial credits for no valid reasons will be treated as a cheating attempt, and such a student will lose all scores of the assignment.