## Linked List (5th Lab)

### Singly Linked List

- **feature**
  - Easy to insert/delete at intermediate node
  - Able to control the list size dynamically
  - Don’t locate at consecutive memory space

![Linked List Diagram](image)

- int array[5]의 크기는 4*5byte
  - Int형을 가진 5개 노드는 (4+포인터 크기)*5byte
  - 같은 수의 데이터를 표현 할때 배열보다 더 많은 데이터 필요

### Singly Linked List Example

```c
#include <stdio.h>
#include <stdlib.h>

struct node{
    int data;
    struct node *next;
}; // node structure composing list

static struct node *head, *tail, *walking;
// head: a pointer that points first node in list,
// tail: a pointer that points last node in list
// walking: when inserted, memory allocated, when deleted, a pointer finds node to be removed
```
void insert(int n);
void delete(int n);
void displayList();

int main()
{
    insert(5);
    insert(7);
    insert(1);
    insert(8);
    displayList();
    delete(7);
    delete(5);
    displayList();
    return 0;
}

\squarespace
\begin{tabular}{@{}c@{}}\quad Insert case 1: if Head is null (insert at the front of list) \end{tabular}

\includegraphics{algorithm.png}

\squarespace
\begin{tabular}{@{}c@{}}\quad Insert case 2: if Head is not null \end{tabular}

\begin{verbatim}
\begin{verbatim}
\void insert(int n)
{
    walking = malloc(sizeof(struct node)); // allocate memory to new node
    printf("%d is added in list\n", n);
    walking->data = n; // assign n to data member of node
\}
\end{verbatim}
\end{verbatim}
walking->next=NULL; // initialize next member

if(head==NULL) //case 1
{
    head=walking; //head points new node
    tail=walking; //tail points new node
    return;
}
else //case 2
{
    tail->next=walking; //tail points new node 노드를 가리킴
    tail=walking; //new node becomes tail
}

▽Delete case 1: delete first node

▽Delete case 2: delete middle node

▽Delete case 3: delete last node
void delete(int n) 
{
    if(head==NULL) return;

    walking=head; //in order to linear search for node to be removed, decide
    location of head at the walking
    struct node *fixer; //fixer always points the next of walking
    while(walking){ //linear search
        if(walking->data==n){
            printf("%d is deleted in list\n",n);
            if(walking==head){ //case 1
                head=walking->next; // head points the next node of node
                to be removed
            }
            if(walking==tail){ //case 2
                tail=fixer; //designate tail to fixed because node to be removed
                is last
            }
            //case 3 fixer, the previous node of node to be removed
            fixer->next=walking->next;
            free(walking); //free memory
            break; //if node is found, then break while break loop
        }
        fixer=walking;
        walking=walking->next; //fixer has walking value, walking has next of walking
    }
}

void displayList() 
{
    printf("Display all element:");
    printf("head=");
    walking=head;
```c
while(walking) //linear search
{
    printf("%d->",walking->data);
    walking=walking->next; // proceed one
}
printf("=tail\n");
```

- Practice
  implement Doubly Linked List by modifying above singly linked list.
  (to get the source linked_list.c, cp /home/comp-ta/dyseo/linked_list.c )

Doubly Linked List : the variation of singly linked list has prev pointer. size of node is getting bigger, but can reference previous node, so that the profit also is getting much mode bigger.

**Requirement**

- add struct node *prev member to struct node

- implement doubly linked list by modifying void insert(),void delete() (*prev should points previous node, if it points head, then *prev has NULL, when use delete(), in order to previous node information, singly linked list used *fixer, now *prev exists and *fixed can be removed)

- void displayList() function prints the data in the list from head to last node of list. doubly liked list is able to prints data from tail reversely, so change the displayList function to print reversely.