Forward Linked Lists

- This week, we will deal with another data structure named as *forward linked list*. It is an alternative of arrays. A linked list allows us to use the memory dynamically. It is called as a forward linked list, because each element of the list is linked to the next element.

- Each element of a linked list is referred to as a *node*. A schematic form of one node would be as follows:

member containing the data	address of the next node

- Arrays use *static* (*fixed*) memory allocation. We always have to know or guess the maximum number of elements (thus, array size) while we are declaring an array.

- However, linked lists use *dynamic* memory allocation. It means that, when we declare a linked list, we reserve memory location for only one node, and whenever we need to add a new element, we ask the computer to allocate a new node.

- The memory allocations for two nodes do not need to be adjacent, but each node must know where the next node is.

- Although an array can become full, if you could not decide its size properly while declaration, a linked list becomes full only if the memory of the computer is so full that no locations can be found for any other node, which happens very rarely.

- Therefore, because of the dynamic memory allocation property, a linked list is very much advantageous when compared with arrays. However, sometimes a linked list may use more memory than an array, because of the memory used to store the links (pointers).

- Linked lists have an important disadvantage when the time to reach an element is considered.

Definition, Declaration and Initialization of Linked Lists

- As I said before, a linked list consists of a series of nodes linked to each other. In order to define a linked list, we will first of all define a structure for a single node. As you know, a node is a structure with two members: a member to store the data, and a pointer to the next node. For instance, we will define a node to contain one integer value as follows:

```
typedef struct node_s {
     int data;
     struct node_s *next;
} node_t;
```

- `node_s` tag should be used to declare `next`. This is necessary because we cannot use `node_t *next` since the compiler has not yet seen the name `node_t`.

- Let's define another node to contain the id, name and grade of a student.

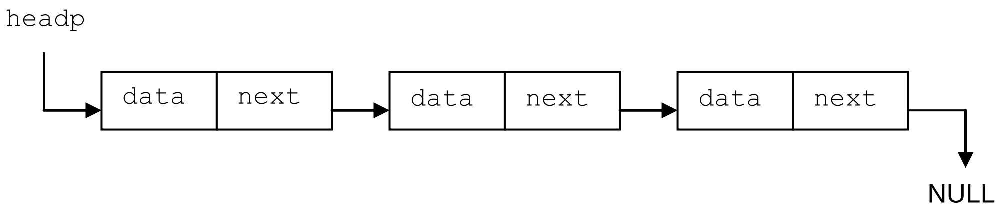
```
typedef struct {
     int    id;
     char   name[20];
     double grade;
}  student_t;

typedef struct node_s
{    student_t data;
     struct node_s *next;
} node_t;
```

- In order to specify the beginning of the list, we will declare a pointer, usually named as `headp`, which contains the address of the first node, as follows:

```
node_t *headp;
```

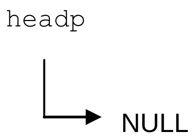
- Now, `headp` represents the name of our linked list.

- In order to specify the end of the list, we will assign the `next` member of the last node to NULL. Thus, if the `next` member of a certain node points to NULL, this means that this is the last node on the list. Therefore, a schematic picture of a linked list with three nodes can be drawn as below:



- How can we initialize an empty list?

```
headp = NULL;
```



Referring to Nodes

- With linked lists, we will refer to the member of a certain node with the notation

  <pointer>**->**<node member>

- For example, to refer to the grade of the first student in the list we defined above, we will say

  ```
  headp->data.grade
  ```

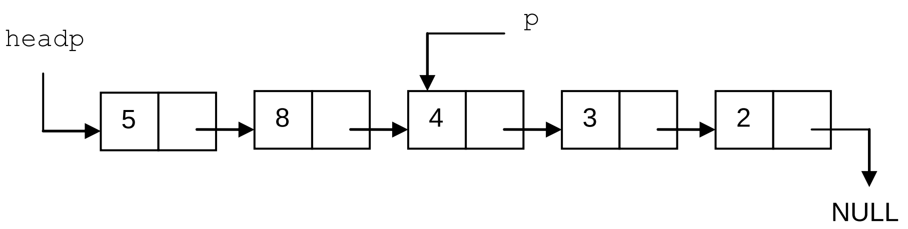
  meaning, "the grade part of the data in the node which `headp` is pointing to".

- Referring to the name of the student in the second node would be:

  ```
  headp->next->data.name
  ```

- Consider the list below containing integer data items, where a certain node is shown by a pointer `p`**.** Thus, `headp` and `p` are declared as follows:

  ```
  node_t *headp, *p;
  ```



- If we want to reach to
  - 4, we will write `p->data`,
  - 5, we will write `headp->data`,
  - 3, we will write `p->next->data`,
  - 8, we will write `headp->next->data`, and
  - 2, we will write `p->next->next->data`.
- Therefore, in order to refer to a certain data item in a linked list, you need to be able to explain the memory address of the correct node.

Forward Linked Lists - Functions

Example: Define a function that displays all the data in a given linked list.

```
void display_list(node_t *headp)
{
     node_t *p;

     /* if the list is empty */
     if (headp == NULL)
         printf("The List is EMPTY !!!");
     else /* if not */
     {
         /* start from the beginning of the list */
         p = headp;
         /* repeat until the end of the list is reached */
         while (p != NULL)
         {
             /* display the data in the current node */
             printf(" %d ", p->data);
             /* if it is not the last node */
             if (p->next != NULL)
                 printf("-->");
             /* pass to the next node */
             p = p->next;
         }
     }
     printf("\n");
}
```

- When you define a function making operations on a linked list, <u>you have to test it for three cases: for an empty list, for a list with one node, and for a list with more than one nodes</u>. Notice that, our function works correctly for all three cases.

- What will be the output, if we call it to display the data of the list above?

Home Exercise: Define a function that finds the number of items in a given linked list.

Solution:

```
int size_list(node_t *headp)
{
     node_t *p;
     int cnt = 0;

     /* start from the beginning of the list */
     p = headp;
     /* repeat until the end of the list is reached */
     while (p != NULL)
     {    /* count the current node */
          cnt++;
          /* pass to the next node */
          p = p->next;
     }
     return (cnt);
}
```

Example: Define a function that searches for an item in a list, and if it can find that item, it returns the address of the node containing it, otherwise it returns NULL.

```
node_t *search_node(node_t *headp, int item)
{
     node_t *p;

     p = headp; /* start from the beginning of the list */
     /* repeat until the beginning of the list is reached or
        the item is found */
     while (p != NULL && p->data != item)
          p = p->next; /* pass to the next node */
     return (p);
}
```

Example: Rewrite the above function recursively.

```
node_t *rec_search_node(node_t *headp, int item)
{
     if (headp == NULL || headp->data == item)
          return (headp);
     else
          return (rec_search_node(headp->next, item));
}
```

Home Exercise: Define a function that searches for an item in a sorted list.