2.1 Structure Definition

In C programming, structures are derived data types. A structure contains a collection of related variables under one name. The variables can be different data types, eg. char, int, long, float, double, etc. or arrays and pointers.

For example,

Structure Definition

```
struct node {
    int age;
    float height;
    char firstName[20];
    char* address;
};
```

The keyword **struct** introduces a structure definition. The identifier *node* is the structure tag. This new structure type is named **struct node**. Variables declared within the braces of the structure definition are the structure's **members**.The members must have unique names. Each structure definition always ends with a semicolon.

To declare a variable of type *struct node*, you can do as the following:

Declaration of Structure Variables

```
struct node {
    int age;
    float height;
    char firstName[20];
    char* address;
}student1, student2;

struct node studentA[20];
```

It is noted that the structure tag name (eg. *node*) is optional. If it does not include in a structure definition, variables of the structure type only can be declared with the structure definition.

Another keyword **typedef** is used for creating synonyms or aliases for the data types. To have a shorter type name of **struct node**, we can

Creating Synonyms

```
typedef struct node {
    int age;
    float height;
    char firstName[20];
    char* address;
}student_t;

student_t student1;
```

Two operators, the structure member operator – a.k.a the dot operator (.) and the structure pointer operator – a.k.a the arrow operator ($->$), can be used to access structure members.

Operators for Accessing Structure Members

```
student_t student1;
student_t *s1Ptr;

student1.age = 15;
s1Ptr = &student1;

printf("%d %d",student1.age, s1Ptr->age);
```

2.2 Dynamic Memory Allocation

When we declare an array, the size of the array must be known at compile time. This kind of data structure can be considered as a static data structure. The allocated memory size is fixed at the compile time. We are not able to change the size at execution time. To avoid the overflow issue, we usually declare the size as large as possible. However, the memory space will not be fully utilized by doing so.

If we would like to have a better utilization and flexibility on the memory usage, dynamic memory allocation can let us to obtain more memory space at execution time and to release the space when it is no longer needed. We can use functions **malloc()** and **free()** to dynamically allocate and deallocate the memory.

Function **malloc()** takes an argument of the number of bytes to be allocated and returns a pointer of type **void *** (pointer to void) to the allocated memory. In practice, we will use operator **sizeof** to determine the size of a type in bytes. The allocated memory is not initialized.

Function **free()** takes a pointer and deallocates memory location where the pointer points to. It is noted that the pointer will not be a NULL pointer after the memory is deallocated. **free()** only returns the memory space back to the system so that it can be reallocated in the future. We need to carefully use the pointer or assign NULL to it.

An Example of malloc() and free()

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

int main(void)
{
  int i;
  double* item;
  char* string;
  item = (double *) malloc(10*sizeof(double));
  string = (char *) malloc(10*sizeof(char));

  for(i=0;i<10;i++)
  {
    scanf("%lf",&item[i]);
  }
  scanf("%*c");

  i=0;
      char *stringP=string;
      while(i++<9)
    scanf("%c",stringP++);
  *stringP='\0';
  printf("%s\n",string);

  double* itemP=item;
  for(i=0;i<10;i++,itemP++)
    printf("%.2lf ",*itemP);
  printf("\n");

  free(item);
  free(string);
  return 0;
}
```

These two functions are under the general utilities library **stdlib.h**. The library also provides two other functions for dynamic memory allocation – **calloc** and **realloc**.

the prototype of calloc() and realloc()

```c
    void *calloc(size_t nmemb, size_t size);
    void *realloc(void *ptr, size_t size);
```

The main difference between malloc and calloc is that calloc clears the memory it allocates and malloc does not. Its two arguments are the number of elements (nmemb) and the size of each element (size).

The realloc() is used to change the size of the allocated memory where *ptr* points to. The contents in the original space are not modified provided that the new amount of memory allocated is larger. Otherwise, the contents are unchanged up to the new size.

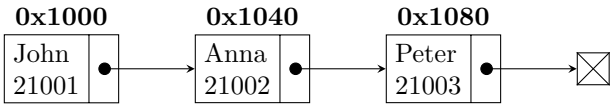


Figure 2.1: The illustration of a link list. The link pointer **nextPtr** consist of the memory address of the next struct node. In this example, the nextPtr of the first node is **0x1040**. Thus, we can find the memory location of the second node from the link of the first node.

2.3 Self-Referential Structures

We can have a pointer member that points to a structure of the same structure type. It is known as a *self-referential* structure.

Self-referential Structure

```
struct node{
    char Name[15];
    int matricNo;
    struct node *nextPtr;  //link
}
```

The last member of the given structure above is a pointer of struct node. We can refer it as a link. It can be used to "tie" a structure to another structure. This is a basic element of a linked list.

2.4 Linked List

We can take a variable of a self-referential structure as a node. Its pointer member with the same structure type as a pointer link to the next node. A collection of nodes will form a new data structure. We called it as a linked list. Since each node contains data members and a link, the linked list is also known as a singly linked list. The example showed in the previous section. The link contains the memory address of the next node. If we know the address of the first node, we can easily obtain the memory location of the subsequent nodes via the link of the previous nodes. The link of the last node is usually set to NULL to mark the end of the list.

The following example code shows the definition of a linked list structure and the declaration of a linked list with a node statically and dynamically.

Definition and Declaration of A Linked List)

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

struct _listnode
{
  int item;
  struct _listnode *next;
};
typedef struct _listnode ListNode;

int main(void)
{
//static node
  ListNode static_node;
  static_node.data = 50;
  static_node.next = NULL;

//dynamic node
  ListNode* dynamic_node= (ListNode*) malloc(sizeof(ListNode));
  dynamic_node->data = 50;
  dynamic_node->next = NULL;
  free(dynamic_node);

  return 0;
}
```

2.5 Utility Functions for A Linked List

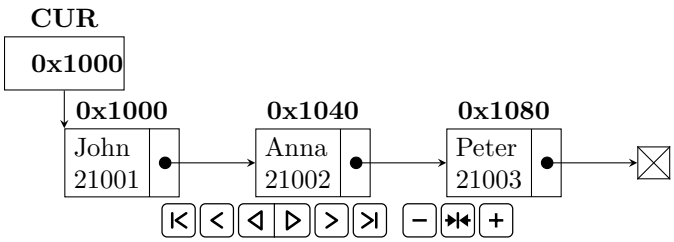
2.5.1 Display All Nodes of A Linked List: printList()

When we need to get the access to every node in a linked list, we just need to know the memory address of the first node. It is stored in a head pointer. We can simply assign it to a pointer. After we print out the data in the current node, you will update the pointer to the next node which its address can be obtained from the link in the current node. When we reach the last node which its link is a NULL pointer, we will stop the print job.

Print All Items in A Linked List)

```
struct _listnode
{
    char firstName[15];
  int num;
  struct _listnode *next;
};
typedef struct _listnode ListNode;

void printList(ListNode *cur){
    while (cur != NULL){
        printf("%s\n", cur->firstName);
        printf("%d\n", cur->item);
        cur = cur->next;
    }
}
```



2.5.2 Search The i^{th} Node: findNode()

The approach of searching is similar to the printlist(). The calling function should pass a head pointer of the linked list. It is the memory address of the first node.

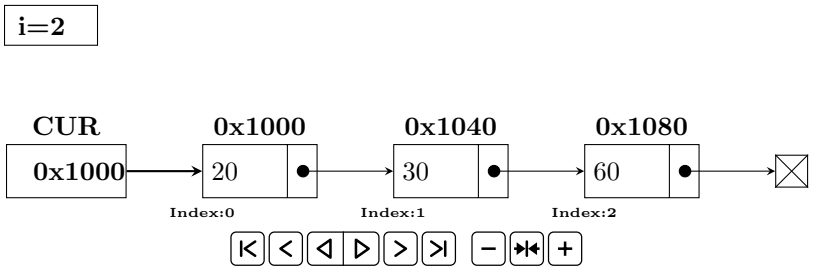
The head pointer is assigned to *cur*. Next, the *cur* moves along the linked list to the i^{th} node subject to availability. The node will be simply returned to the calling function.

NULL will be returned if the index i is out of the range or the linked list is an empty list.

The index here starts from 0 to the size of the linked list -1.

Searching the i^{th} node in the linked list

```
ListNode *findNode(ListNode* cur, int i)
{
   if (cur==NULL || i<0)
      return NULL;
   while(i>0){
      cur=cur->next;
      if (cur==NULL)
         return NULL;
      i--;
   }
   return cur;
}
```



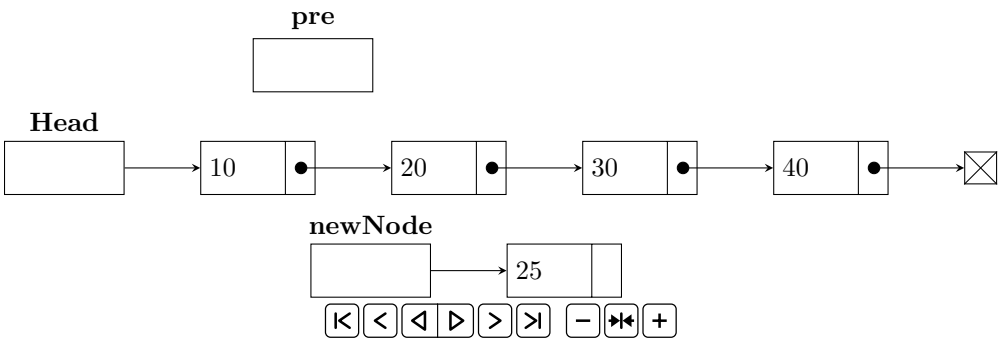
2.5.3 Insert A New Node: insertNode()

Two key utility functions of the linked list are the node insertion and the node removal. Here, we only discus about the node insertion. You can use the similar idea on the node removal.

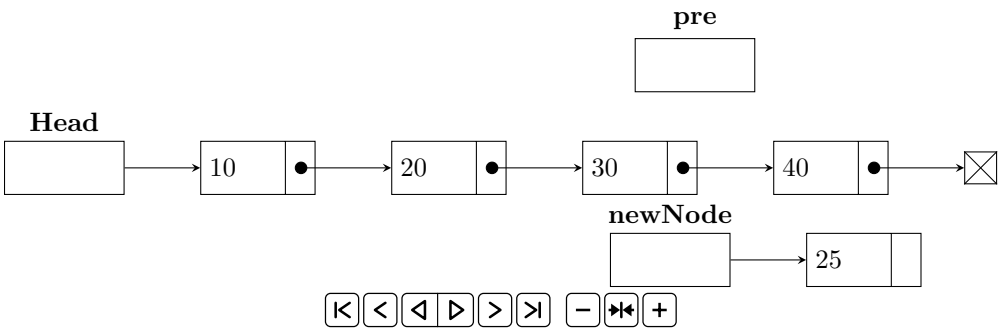
To insert a node, we need to consider three scenario:

1. Insert at the front of a linked list
2. Insert in the middle of a linked list
3. Insert at the back of a linked list

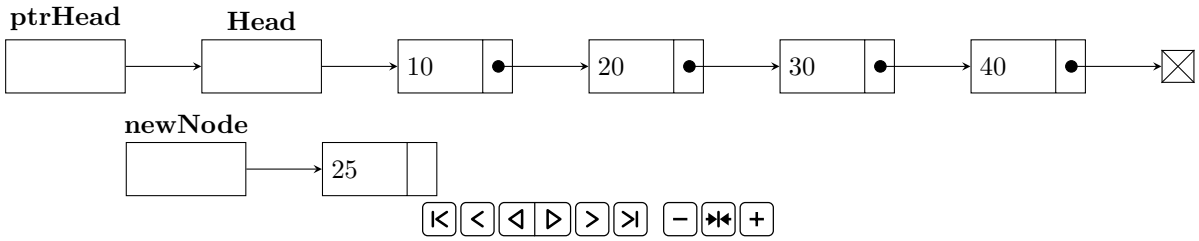
Let's discuss about the last two cases first. To insert a new node in the middle of a linked list, we first need to locate the node which will be the previous node of the new node. Let us denote it as *pre*. Next, the link of the new node is assigned to the link of *pre* node. After that, we can update the link of *pre* node to the new node. Once it is done, the new node is successfully inserted to the linked list. See the animation below.



Similarly, we can use the exactly same algorithm on inserting node at the back. The link of *pre* node is a NULL because it is the last node. The link of the new node is assigned to a NULL. Next, the link of *pre* is pointed to the new node.



Inserting a new node at the front is a bit difference from the previous two cases. When you insert a new node at the front, you need to modify the content of head pointer. To modify a variable in the calling function, we need to pass by reference (a pointer to the variable). If we need to modify a pointer in the calling function, we need to pass a pointer to the pointer. Thus, we need a pointer pointed to the head pointer (*ListNode **ptrHead*).



The sample code of node insertion is given below.

Insert a new node at the ith position in a linked list

```
int insertNode(ListNode **ptrHead, int i, int item){
    ListNode  *pre, *newNode;
    // If empty list or inserting first node, update head pointer
    if (i == 0){
        newNode = (ListNode  *) malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
        return 1;
    }
    // Find the nodes before and at the target position
    // Create a new node and reconnect the links
    else if ((pre = findNode(*ptrHead, i-1)) != NULL){
        newNode = (ListNode  *) malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = pre->next;
        pre->next = newNode;
        return 1;
    }
    return 0;
}
```

2.5.4 Find The Size of A Linked List: sizeList()

The size of a linked list is another similar approach. We just need to use a brute-force approach to visit every node in the link list and create a counter to count the number of nodes. When we reach the last node which is a null, the size is obtained.

Size of A linked List)

```
int sizeList(ListNode *head){

    int count = 0;

    while (head != NULL){
       count++;
       head = head->next;
    }

    return count;
}
```

2.5.5 Revised Definition of A Linked List Structure

Let us relook the sizeList(). If we check the size of a linked list, we always need to repeatedly visit every node and do a counting. It is too time consuming. To improve the efficiency, we can introduce a variable to store the size of the linked list. We can increase and decrease the variable when we insert and remove a node respectively. Hence, we define a new struct with a new member *size*.

Wrapped version of the linked list structure definition)

```
typedef struct _linkedlist{
     ListNode *head;
     int size;
} LinkedList;
```

The size of the linked list can be easily obtained. Of course, we need to introduce and update the size in the node insertion and node removal.

Utility Functions

```
int sizeList(LinkedList ll){
    return ll.size;
}


void printList(LinkedList ll){
    ListNode *temp = ll.head;

    while (temp != NULL){
        printf("%d\n", temp->item);
        temp = temp->next;
    }
}

ListNode *findNode(LinkedList ll, int i){
    ListNode *temp = ll.head;
    if (cur==NULL || i < 0|| i >ll.size)
        return NULL;

    while (i > 0){
        temp = temp->next;
        if (temp == NULL)
            return NULL;
        i--;
    }
    return temp;
}
```

2.5.6 Linked Lists VS Arrays

Arrays:

- Efficient on random access
- Difficult to expand, shrink and re-arrange
- When inserting/removing items in the middle or at the front, computation time scales with size of list. (you may need to create a new array)
- Generally a better choice when data is immutable

Linked Lists:

- "Random access" is hardly implemented
- Additional cost on storing links and it is only use internally for the linked list.
- Easy to expand, shrink and rearrange (but array-based linked list has a fixed size)
- Insert/remove operations only require fixed number of operations regardless of list size. no shifting