Lists

1. Lists

List definition.

A list is the most basic data structure that we will study. List means a set of items. For example, the following is a list of fruit names: apple, orange, mango, banana, and pear. The following are some more examples of lists:

- Saima, Jamil, Sebuti, Raihan, and Salma.
- 1, 3, 5, 7, 11, 17, 19, …
- Apple, mango, orange, banana, pear, …
- 0, 1, 1, 2, 3, 5, 8, 13, ...

The first one above is a list of names. The second one is a list of integer numbers, the third one is a list of fruit names, and the last one is a list of Fibonacci numbers.

Storing a list in a program.

Suppose, you will write a program and want to input a list from the user. You will require storing them in your program for later processing. Now, how will you store a list in a program. The first choice that comes to a programmer's mind is ***Array***. Obviously, there are other better choices such as linked-lists, but you must understand the basic array based list to learn how it really works. We will understand the linked-list based lists later.

Array.

An array is a variable that allows us to store a number of items. An array can be accessed by an index. It is very fast when we access items using index. So whenever we use arrays to store a list, we may call it an ***ArrayList***. An array is declared in C in the following way:

```
int A[100]; //an array of integers
```

An array has the following important characteristics:

- The size of array must be fixed at the compile time. Once an array is declared, then its size cannot be changed. *This is the major disadvantage of using arrays that you must know how many items will finally be stored.*
- In C programming, index start with 0. So, the first item will go to index 0.
- Items in an array are accessed using in index n. For example, $A[n]$ will give you the value of the $(n+1)$-th item stored in the list.

2. Static ArrayList.

Static ArrayList construction.

An ***ArrayList*** is a list implemented using static arrays. It may also be called static ***ArrayList*** as opposed to the dynamic ***ArrayList*** that we will show later. Below, we show sample program fragments that makes an ArrayList. An array is declared basically that will be used for storing list items. We have created the list to store *integer* items. If we want other types of lists, we will need to change the type of variable *list* to the desired type.

C Codes for declaring ***ArrayList***

```
#define LIST_MAX_SIZE 100
#define NULL_VALUE -99999
#define SUCCESS_VALUE 99999

int list[LIST_MAX_SIZE];
int length;
```

There are two variables that will be used to make a list: $list$ and $length$. The variable $list$ will store the actual items whereas the variable $length$ will be used to track the index position where the last item was stored. *Our list will be contiguous; it will not keep empty space between two items.*

Now that we have declared an ***ArrayList***, we need to define functions that can be used to perform operations on the list. Usually, a data structure should allow us with following three basic operations:

- *Insert an item* – list will be used to insert a new item
- *Delete an item from the list* – sometimes, we will require to delete items from the list
- *Get the nth item from the list* – Getting the n-th item from the list.

- *Update an item in the list* – sometimes we will require to modify existing items
- *Search an item in the list* – search an item to find its existence in the list

More advanced operations can be defined as follows:

- Find the minimum item – finds the minimum item in the list
- Find the maximum item – finds the minimum item in the list
- Insert at item after some specific item – inserts an item at a position

Basic operations on ArrayList

Initialize list before using it.

The *initializeList* function must be called before using the list. It basically sets the*length* variable to 0. So, items will be stored beginning from index 0 in the array.

C code for initializing static array list

```
void initializeList()
{
	length = 0 ;
}
```

Insert an item.

The *insertItem* function inserts at item in the list at the next empty position, maintained by *length* variable. The *length* variable is then increased by 1 to advance to the next empty position. The function also checks whether the *length* reached the maximum size of the array. In that case, it returns an indicator value, *NULL_VALLUE*, to notify the caller that insertion has been unsuccessful.

C code for inserting an item at the beginning of static array list

```
int insertItem(int item)
{
	if (length == LIST_MAX_SIZE) return NULL_VALUE ;
	list[length] = item ;
	length++ ;
	return SUCCESS_VALUE ;
}
```

Get the n-th item.

C code for retrieving the ith item of static array list

```
int getItem(int position)
{
	if( position < 0 || position >= length ) return NULL_VALUE;
	return list[position] ;
}
```

Search an item.

The *seachItem* function shown below searches the list to find the input *item*. If it finds the item, then it returns the index position of them item in the array. Otherwise, it returns *NULL_VALUE*.

C code for searching an item in the array list

```
int searchItem(int item)
{
	int i = 0;
	for (i = 0; i < length; i++)
	{
		if( list[i] == item ) return i;
	}
	return NULL_VALUE;
}
```

Delete an item at some position.

The *deleteItemAt* function shown below deletes an item from a given index position as its parameter. Observe that when you delete an item from a position, that position becomes empty in the array. Since we will always keep items contiguously in the array, we will need to fill up this position by some other items. To achieve this we can do either of the following:

- We will move all items in the right one position left. Finally, we reduce the *counter* variable by one as now we have one less items. Note that this operation will take a long time if there are many items in the right of the deleted item.

- We can move the last item to the empty position. In this case, the order of entry will not be preserved. However, this technique will be very fast. *This version is not suitable when we keep the data in some specific order, for example, we keep the integer items sorted.*

C code for deleting an item from a given position

```
int deleteItemAt(int position) //version 1, preserve order of items
{
        if ( position < 0 || position >= length ) return NULL_VALUE;
        int i;
        for (i = position + 1; i < length; i++)
        {
                list[i-1] = list[i] ; //move item left by one slot
        }
        length-- ;
        return SUCCESS_VALUE ;
}
```

C code for deleting an item from a given position

```
int deleteItemAt(int position) //version 2, do not preserve order of items
{
        if (position < 0 || position >= length) return NULL_VALUE;
        list[position] = list[length-1] ;
        length-- ;
        return SUCCESS_VALUE ;
}
```

Delete given item.

The $deleteItem$ function shown below deletes a given item. First, we need to search the item to find its position. Then we will delete it by calling the $deleteItemAt$ function.

C codes for deleting an item from an array list

```
int deleteItem(int item)
{
        int position;
        position = searchItem(item) ;
        if ( position == NULL_VALUE ) return NULL_VALUE;
        deleteItemAt(position) ;
        return SUCCESS_VALUE ;
}
```

Disadvantages of static ArrayList.

Some of the major disadvantages of **ArrayList** are-

- Array size is fixed at compile time. An array is declared before the program is run. So you must know the amount of data that you will write in the array. *This is not suitable for most of the modern day applications where size of data is not limited and keeps growing over time.*
- When you delete an item from an *order-preserved ArrayList* shown above, all items to the right must be moved left by one slot. This requires a significant amount of time, proportional to the number of items remaining.

Below, we show some improved data structure that can avoid the problems of array based lists.

3. Dynamic ArrayList

A dynamic **ArrayList** is similar to an **ArrayList** except that it uses a pointer rather than a fixed array. Using pointer instead of an array allows us following advantages-

- *The size of the list need not be pre-fixed, i.e., must be known at compile time*. Since memory for a pointer can be allocated during run-time, so the size of the list may be determined during run-time.
- *The size may even be changed later*. When we find that the array is full, we can create new memory for the array, thus allowing the size to grow over time. However, this operation requires copying all existing items of the list to a new memory location.

ArrayListDyn construction.

This time we will use a pointer rather than an array to store the data. Below, we show the variable declaration program fragment. This time, we have three variables: $listMaxSize$, $list$, and $length$. The $listMaxSize$ variable will store the size of the memory allocated to the $list$ variable at any point of time, the $list$ pointer variable that will be allocated space, and the $length$ variable as before. Note also that the previous $LIST\_MAX\_SIZE$ definition has been changed to $LIST\_INIT\_SIZE$.

C Codes for declaring ***ArrayListDyn***

```
#define LIST_INIT_SIZE 100
#define NULL_VALUE -99999
#define SUCCESS_VALUE 99999

int listMaxSize;
int *list;
int length;
```

Initializing of the Dynamic ArrayList.

In the initialization, this time we will require to allocate space for the pointer. Initially, we will allocate memory space for storing *LIST_INIT_SIZE* number of items.

C codes for initializing dynamic array list

```
void initializeList()
{
      listMaxSize = LIST_INIT_SIZE;
      list = (int*)malloc(sizeof(int)*listMaxSize) ;
      length = 0 ;
}
```

The *malloc* function in C library allocates memory and returns the address of the allocated memory. *Note that sometime, this allocation becomes unsuccessful. So, our code should have handled that case. But, for simplicity, we are not writing it here.*

Operations on the ArrayListDyn.

All operations will have codes similar to basic array based **ArrayList**. However, during insertion of an item, if we see that the *length* has reached maximum size of the array, we will allocate new memory for the *list* variable to create more space. This is shown in the modified *insertItem* function below.

```
   C codes for inserting item at the end of a dynamic array list
1  int insertItem(int item)
2  {
3         if (length == listMaxSize)
4         {
5                int * tempList ;
6                //allocate new memory space for tempList
7                listMaxSize = 2 * listMaxSize ;
8                tempList = (int*) malloc (listMaxSize*sizeof(int)) ;
9                int i;
10               for( i = 0; i < length ; i++ )
11               {
12                      tempList[i] = list[i] ; //copy all items
13               }
14               free(list) ; //free the memory allocated before
15               list = tempList ; //make list to point to new memory
16        }
17        list[length] = item ; //store new item
18        length++ ;
19        return SUCCESS_VALUE ;
20 }
```

Note the following in the above *inserItem* function.

- First, we allocate new memory space for a temporary pointer variable *tempList*. In Line 7, we see that the new memory is twice the existing memory. So, our policy is to double the memory whenever it becomes full.
- All items from existing *list* variable are copied to *tempList* variable (Line 11-14).
- Then, we free the memory space previously allocated. This is a very important step; otherwise the previous memory will be leaked out. We will describe about this problem later.
- Finally, we make the list variable to point to new memory (Line 16).

Disadvantage of Dynamic ArrrayList.

Although, a dynamic array list is a better approach than a static array list, it has following drawbacks:

- First, to create more space, we need to move all elements from the current list to the new list. This takes a significant amount of running time, proportional to the length of the array.
- Second, creating space for a dynamic array, i.e., a pointer requires operating system to allocate contiguous memory location. Sometimes, such a large memory block may not be found, and thus memory allocation will not succeed.
- Third, to insert or delete a particular item from the list, we will need to move some existing items. This will require time proportional to the number of items copied.

To solve the problems stated above, we described below an alternate implementation of a list based on linked list.

4. **Linked List**

A Linked List is a dynamic List. The advantages of a Linked List are –

- It can grow its size according to the requirements. We do not need to know the amount of data that will be stored.
- A dynamic list also grows its memory for one item at a time. At each point in time, it consumes the required amount of memory to store only the current items. It does not allocate memory for future items.

Linked List construction.

A Linked List is actually list of **nodes** rather than a list of items. For each item stored, it manages a node. A node contains item as well as some other necessary information to connect the list. The nodes are connected by pointers. To construct a Linked List, we need to declare the node type first. Below we show the type $listNode$ that will be used to create the Linked List. The definition given below is also called a singly linked list as opposed to a doubly linked list that has two different pointers to locate next and previous nodes.

```
C Codes for declaring LinkedList
#define NULL_VALUE -99999
#define SUCCESS_VALUE 99999

struct listNode
{
	int item ; //will be used to store value
	struct node *next ; //will keep address of next node
} ;
struct listNode * list ;
```

In the above definition, we declared a new data type $listNode$ that contains two variables: an integer variable $item$ to keep the data and a pointer variable $next$ that keeps the address of next $listNode$ variable of the whole list. Finally, a variable $list$ is declared as a pointer to the $listNode$ type. This pointer will point to the first node of the list.

Initialization of the LinkedList.

This time for a LinkedList, we do not require much initialization codes except setting the pointer list to 0. In C programming language, 0 is called "NULL" value for a pointer. When a pointer is assigned a value of 0, it implies that the pointer does not point to a valid memory location. This is because no valid memory location can have the value 0.

```
C code for initializing linked list
void initializeList()
{
	list = 0 ; //set to NULL
}
```

Item insert in the linked list at the front.

When items are inserted in the list, the will be inserted at the beginning of the list. The code of insertion is shown below.

C Codes for inserting an item in the ***LinkedList***

```c
int insertItem(int item)
{
	struct listNode * newNode ;
	newNode = (struct listNode*) malloc (sizeof(struct listNode)) ;
	newNode->item = item ;
	newNode->next = list ;
	list = newNode ;
	return SUCCESS_VALUE ;
}
```

Search item in the list

The $searchItem$ function, shown below, searches for the given item in the Linked List. It returns the pointer to the node that contains the item; otherwise it returns 0, a value that indicate invalid pointer value in C.

C Codes for searching an item in a ***LinkedList***

```c
struct listNode* searchItem(int item)
{
	struct listNode *temp ;
	temp = list ; //start at the beginning
	while (temp != 0)
	{
		if (temp->item == item) return temp ;
		temp = temp->next ; //move to next node
	}
	return 0 ; //0 means invalid pointer in C, also called NULL value in C
}
```

Delete an item from the list.

The $deleteItem$ function, shown below, searches for the given item in the Linked List. Then it deletes the node containing the desired item. It also releases the memory occupied by the deleted node using $free$ function of C library. This is very important as C programs does not deletes memory automatically that are not being used. Suppose Y is the node to be deleted, X is the node previous to Y, and Z is the node after X. Either of X and Z can be epmty. The list takes one of the following two actions to delete the node X:

- If X is not empty, then we just need to set the $next$ pointer of X to point to node Z.

- If X is empty (Y is the first node of the list), then we just need to set the $list$ (the head pointer) to point to node Z.

C Codes for deleting item from a ***LinkedList***

```c
int deleteItem(int item)
{
        struct listNode *temp, *prev ;
        temp = list ; //start at the beginning
        while (temp != 0)
        {
                if (temp->item == item) break ;
                prev = temp;
                temp = temp->next ; //move to next node
        }
        if (temp == 0) return NULL_VALUE ; //item not found to delete
        if (temp == list) //delete first node
        {
                list = list->next ;
                free(temp) ;
        }
        else
        {
                prev->next = temp->next ;
                free(temp);
        }
        return SUCCESS_VALUE ;
}
```

Disadvantages of Linked List.

Despite having some major advantages, a Linked list has several disadvantages that an **ArrayList**. Some are following:

- Unlike ArrayList, there is no way to access items in a LinkedList by index position. You cannot get the item at the n-th position in constant amount of time. You need to traverse the whole list to find the n-th item which takes long time, proportional to number of items in the list.
- Each memory is required to maintain the connections between nodes.

The singly linked list that we shown has several drawbacks. Some important operations on the list such as inserting item at the end and deleting item at the end will require traversing the whole list to locate the end point. This will require time proportional to number of items in the list. Our second version of the linked list called doubly linked list will solve this problem and will execute those operations in constant time.

5. Doubly Linked List.

You may have noticed that we needed to keep the previous node of the item that will be deleted. In a double linked list, we always keep a pointer to the previous node. So, during deletion, we will not anymore require to track the previous node manually by ourselves.

Definition of a doubly linked list

The definition of a double linked list is given below.

C Codes for declaring ***DoublyLinkedList***

```
#define NULL_VALUE -99999
#define SUCCESS_VALUE 99999

struct listNode
{
      int item ; //will be used to store value
      struct node *next ; //will keep address of next node
      struct node *prev ; //will keep address of previous node
} ;
struct listNode * list ;
struct listNode * tail ;
```

In the double linked list definition above, we have following additions to the singly linked list defition:

- The *listNode* now contains two pointers: *next* and *prev*. The *next* pointer will point to the next node of the list while the *prev* pointer will point to the previous node of the list.
- In addition to *list* global pointer variable, we declare another global variable *tail*. The *tail* will always point to the last node of the list. This will help us to locate the last node of the list in

constant time for implementing operations such as inserting item at the end and deleting item at the end of the list.

Initialization of the DoublyLinkedList.

For initialization, we will only require setting both the global pointers to 0.

C codes for initializing doubly linked list

```c
void initializeList()
{
      list = 0 ; //set to NULL
      tail = 0 ; //set to NULL
}
```

Insertion of an item in the doubly linked list.

The insertion and deletion from a doubly linked list will be simplified. The *insertItem* function is given below which inserts a new item at the beginning of the linked list.

C Codes for inserting an item in the ***DoublyLinkedList***

```c
int insertItem(int item)
{
      struct listNode *newNode ;
      newNode = (struct listNode*) malloc (sizeof(struct listNode)) ;
      newNode->item = item ;
      newNode->next = list ;
      list = newNode ;
      newNode->prev = 0 ; //no previous node as this is the first item
      if ( newNode->next != 0 )
            newNode->next->prev = newNode ;
      else
            tail = newNode ; //this is the first node in the list
      return SUCCESS_VALUE ;
}
```

Insertion of an item at the end of the doubly linked list.

The following function shows how to insert an item at the end of the list. This operation runs in constant time.

C Codes for inserting an item at the end of the ***DoublyLinkedList***

```c
int insertLast(int item)
{
        struct listNode *newNode ;
        newNode = (struct listNode*) malloc (sizeof(struct listNode)) ;
        newNode->item = item ;
        newNode->prev = tail ;
        tail = newNode ;
        newNode->next = 0 ; //no next node as this is the last node
        if ( newNode->prev != 0 )
                newNode->prev->next = newNode ;
        else
                list = newNode ;     //this is the first node in the list
        return SUCCESS_VALUE ;
}
```

Running times of different operations of an ArrayList and LinkedList.

The running time comparisons (worst case) for different operations of an array based list (dynamic), single linked list, and doubly linked list are given below:

Operation	**Array based list (dynamic)**	**Singly linked list**	**Doubly linked list**
Insert item at the beginning	$O(n)*$	$O(1)$	$O(1)$
Insert item at the end	$O(n)$*	$O(n)$	$O(1)$
Delete item at the beginning	$O(1)$**	$O(1)$	$O(1)$
Delete item at the end	$O(1)$	$O(n)$	$O(1)$
Insert item at ith position	$O(n)*$	$O(n)$	$O(n)$
Delete item at ith position	$O(1)$**	$O(n)$	$O(n)$
Get the ith item	$O(1)$	$O(n)$	$O(n)$

*For insertion, at some points of time, we need to copy all existing items of the list.

**For order preserving lists, these operations will run in $O(n)$ time.