Abstract Data Types I - Lists & Linked Lists

Doan Nhat Quang

doan-nhat.quang@usth.edu.vn
University of Science and Technology of Hanoi
ICT department

Today Objectives

- Introduce the basic of Linked Lists: declaration, initialization, and use.
- Learn different functions, operations with Linked Lists: add, remove, search, etc.
- Implement examples in C/C++.

Data Structures

- Data structures study the organization of data in computers
  - the (abstract) data types (definition and representation)
  - relationship between elements of this type
  - operations on data types: retrieve, manipulate, compute, etc.
- Algorithms
  - methods to operate on data structures: efficiency and simplicity
  - **program = data structures + algorithm**

Data Types

- A data type consists of
  - a collection of data elements (a type)
  - a set of operations on these data elements
- Data types in programming languages
  - **pre-defined**: any language defines a group of pre-defined data types such as int, char, float, double in C
  - **user-defined**: any language allows users to define their own (new) data types such as struct, union in C

Data Types

- Pre-defined data types:
  - type: int
  - elements: ...., -2, -1, 0, 1, 2, ...
  - operations: +, - , *, /, %, ...
- User-defined data types:
  - type: complex
  - elements: 1+3i, -15 + 93i, ...
  - operations: add, distance, square, etc.

Abstract Data Type

- An abstract data type (ADT) is a set of objects together with a set of operations. These operations can be divided into two types:
  - Access allowing to identify objects in the set.
  - Manipulation allowing to control, modify, manipulate objects.
- ADTs are precisely specified independent of any particular implementation.
- Abstract means that the implementation of operations are not specified in ADT definition.
- Example: List, Stack, Queue, String, Tree, etc.

Abstract Data Type

Elementary Data type

- **EDT**: char, int, float, double, enum, etc.
- EDT operations are built-in functions in programming languages
- EDT operations are addition, multiplication, division, etc.

Abstract Data Type

- **ADT**: List Stack, Queue, String, Tree, Graph
- ADT operations are often defined by users
- ADT operations can be: add, remove, search, display, etc.

Structures

Structure is user defined data type available in C programming, which allows to combine one or more variables, possibly of different types, grouped together under a single name for convenient handling.

```
1 struct [structure tag]{
2     member definition;
3     member definition;
4     ...
5     member definition;
6 } [structure name];
```

Structures

- Container for related data for easy access
- May have empty gaps between members
- Useful in creating data structures such as linked lists, queues, trees, etc.

Structures

```
1  struct Student{
2      int age;
3      char name[50];
4      unsigned char gender;
5  };
6  struct Student s1, s2;
```

Abstract Data Type

Example

- Consider a data type to represent natural numbers:
  - a data type called *nat*
  - elements: 0, 1, 2, 3, ...
  - operations: new, add, sub, mul, ...
- How to represent this abstract data type in C?

Abstract Data Type

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  struct nat {
4      int val;
5  };
6
7  struct nat* newN(int i){
8      struct nat *n = (struct nat *)malloc(sizeof(struct nat));
9      n->val = i;
10     return n;
11 }
12 struct nat* addN(struct nat *n1, struct nat *n2){
13    struct nat *n = (struct nat *)malloc(sizeof(struct nat));
14    n->val = n1->val + n2->val;
15    return n;
16 }
```

Abstract Data Type

```
1
2  void display(struct nat* n){
3  printf("%d",n->val);
4  }
5  int main(){
6  struct nat *n3;
7  struct nat *n1 = newN(13);
8  struct nat *n2 = newN(14);
9  n3 = addN(n1,n2);
10 display(n3);
11 }
```

Lists

Definition

A list is a collection with finite number of data objects that has the following properties:

- It is homogenoeus, i.e. the elements are all of the same type.
- It has finite length.
- Its elements are arranged in sequential (linear) order.

Lists

- Polynomial functions
  - $13 + 4x^2 + 79x^3 + 100x^10 + 16x^58$
- Unbounded Integers
  - 456913018892132115479910488793126548976l3
- Text
  - "This is a sample of text!!"

Lists

Guild Wars 2 : heart of thorns par NCsoft — EUR 44,50
Jeu informatique
Il ne reste plus que 4 exemplaire(s) en stock.
Vendu par Century
Options de cadeau pas disponibles. En savoir plus
Supprimer | Mettre de côté

Starcraft 2 : legacy of the void par Blizzard — EUR 29,99
Jeu vidéo
En stock
Éligible à l'expédition GRATUITE
Ceci sera un cadeau En savoir plus
Supprimer | Mettre de côté

Kindle Paperwhite, Écran Haute Résolution 6" (15 cm) 300 ppp avec éclairage intégré et Wi-Fi - Avec offres spéciales par Amazon — EUR 129,99
En stock
Éligible à l'expédition GRATUITE
Ceci sera un cadeau En savoir plus
Supprimer | Mettre de côté

Sony DSC-RX100.CEE8 Appareil photo numérique 20,2 Mpix Zoom optique 3,6x Noir par Sony — EUR 345,24
En stock
Éligible à l'expédition GRATUITE
Ceci sera un cadeau En savoir plus
Supprimer | Mettre de côté

Samsung Galaxy Grand Prime Value Edition Gold par Samsung — EUR 136,00
En stock
Vendu par Tecnosell
Options de cadeau pas disponibles. En savoir plus
Supprimer | Mettre de côté

Lists

A **data structure** is a group of **data** elements grouped together under one name. These **data** elements, known as members, can have different types and different lengths. **Data structures** can be declared in C++ using the following syntax: struct type_name {

Data structures - C++ Tutorials - Cplusplus.com
www.cplusplus.com/doc/tutorial/**structures**/
You visited this page.

About this result • Feedback

Data structure - Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/**Data_structure** ▾ Wikipedia ▾
In computer science, a **data structure** is a particular way of organizing **data** in a computer so that it can be used efficiently.
List of data structures · Abstract data type · Linked · Array

Data Structure and Algorithms (DSA) Tutorial
www.tutorialspoint.com/**data_structures**_algorithms/ ▾
Data Structures are the programmatic way of storing **data** so that **data** can be used efficiently. Almost every enterprise application uses various types of **data structures** in one or other way.

Data structures - C++ Tutorials - Cplusplus.com
www.cplusplus.com/doc/tutorial/**structures**/ ▾
A **data structure** is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different ...
You visited this page.

Data Structures Archives - GeeksforGeeks
www.geeksforgeeks.org/**data-structures**/ ▾
Average time Taken to Cover All this post. Beginners - 8 MONTHS.(Learnt already C/C++/Java.) Intermediate - 4 Months(Read and Execute Mode). Intermediate ...

Data structure

In computer science, a data structure is a particular way of organizing data in a computer so that it can be used efficiently. Wikipedia

Feedback

See results about

Algorithms + Data Structures = Programs (Book by Nikl...
Originally published: 1976
Author: Niklaus Wirth

Lists

Common different approaches to implement a List ADT

- Static array: arrays can be simply used to manipulate collections of elements.
- Dynamic array: using **malloc()** is capable of representing a list to avoid the fixed-size list
- Linked list: A very flexible mechanism for dynamic memory management is provided by pointers.

Lists

Basic operations are often defined

1. init(): create an empty list
2. isEmpty(): check if the list is empty
3. insert(): add new item in a list
4. remove(): remove an item from a list

Lists

Basic operations are often defined

1. init(): create an empty list
2. isEmpty(): check if the list is empty
3. insert(): add new item in a list
4. remove(): remove an item from a list

Some other possible operations can be used:

1. length(): return the length of a list
2. search(): search a specific element in a list
3. display(): display a list
4. sort(): sort all items in a list

Static Array-based Lists

The idea is to store the list in a fixed size static array.

```
1 struct _List {
2     int size;
3     int data[CAPACITY];
4 };
5 typedef struct _List List;
```

Static Array-based Lists

- Before the use, a list must have to be initialized with no element.

```
1  void init (List *l) {
2      l->size = 0;
3  }
```

8	13	2	1	30	15	91	24	7	10

Static Array-based Lists

- Several operations require to verify whether a list is not empty or else further operations will be treated.

```
1  int isEmpty(List *l) {
2    return (l.size==0);
3  }
```

Static Array-based Lists

- This function provides basic information of a list.

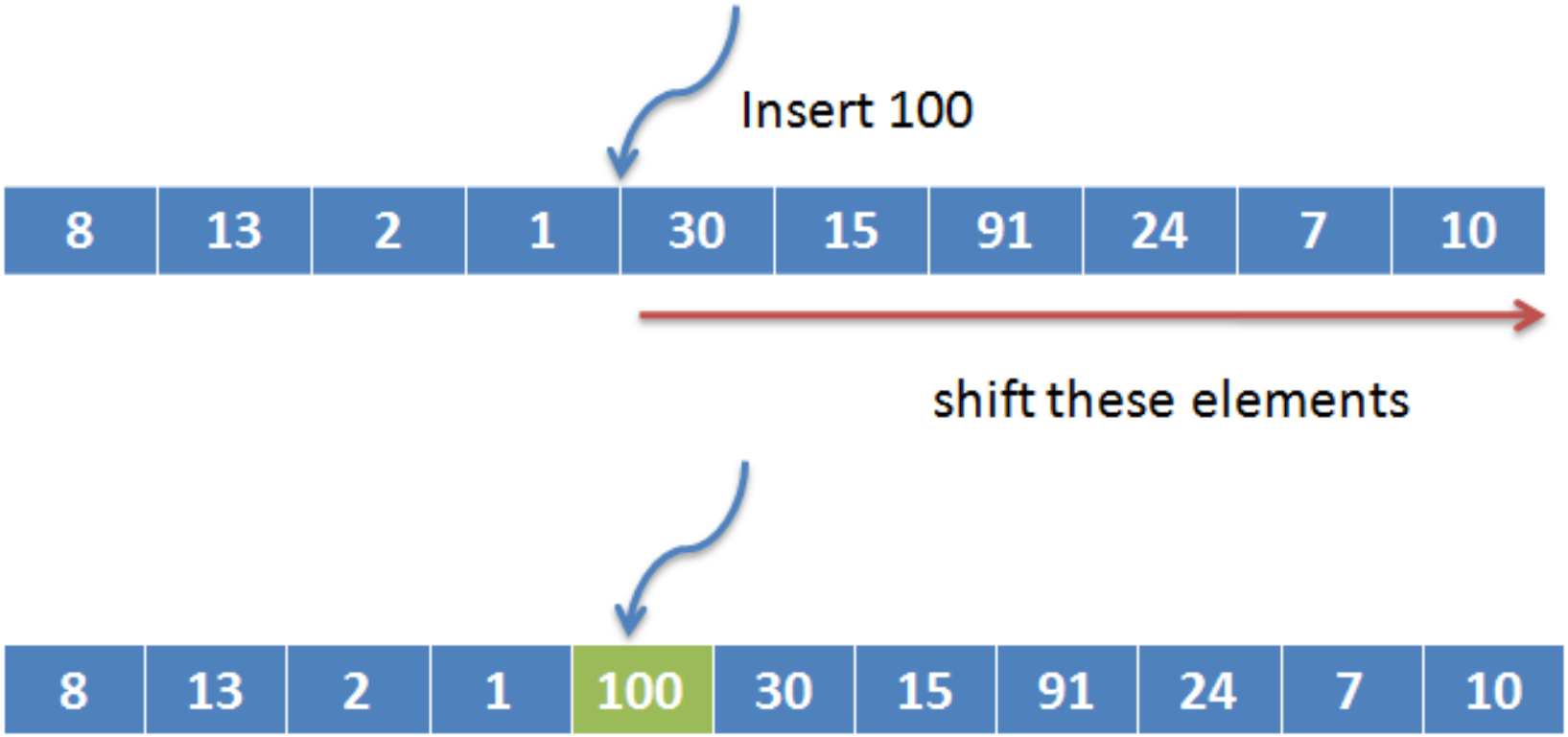
```
1  int length(List *l) {
2    return l.size;
3  }
```

- An array-based list allows to access directly the index of its elements. display() functions can be implemented to provide its information.

```
1  int display(List *l){
2       for (int i=0; i<length(l);i++){
3            printf(''Element %d: %d'',i,l->data[i]);
4       }
5  }
```

Static Array-based Lists

Before the insertion, several verifications on the index should be done. Elements next to the inserted index should be shifted to the right.

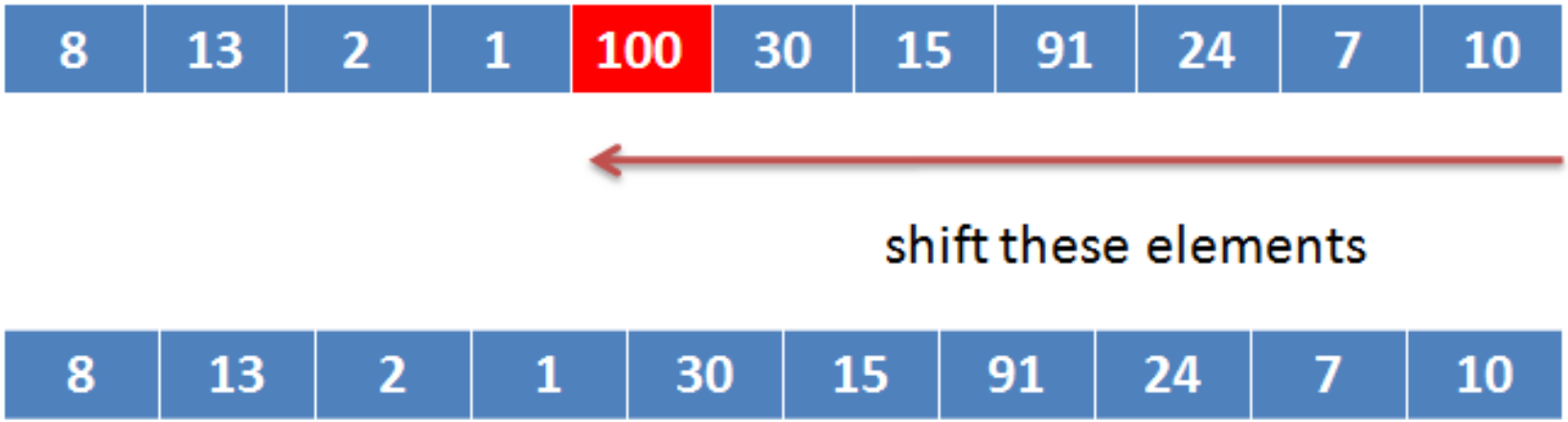


Static Array-based Lists

```
1  void add(List *l, int index, int _val){
2      if (length(l) == CAPACITY){
3        printf("Cannot add more items in the list!");
4        return;
5      }
6      if ((index < 0) || (index> length(s))){
7        printf("Illegal index!"); return;
8      }
9      else{
10         for (int i=length(l);i>index;i--)
11             l->val[i] = l->val[i-1];
12         l->val[index] = _val;
13         l->size++;
14     }
15 }
```

Static Array-based Lists

Removing an item with a specific index from a list requires shifting elements to the left.



Static Array-based Lists

```
1  void remove(List *l, int index){
2       if (isEmpty(l) == 1){
3           printf("List is empty!");
4           return;
5       }
6       if ((index < 0) || (index> length(l))){
7           printf("Illegal index!");
8           return;
9       }
10      else{
11          for (int i=index-1;i<length(l);i++)
12              l->val[i] = l->val[i+1];
13          l->size--;
14      }
15 }
```

Dynamic Array-based Lists

- We often don't know the size of the memory in the problem in advance.
- Instead of Static Lists, Dynamic Lists can be used.

```
1 struct _List{
2         int size;
3         int capacity;
4         int  *val;
5 }List;
```

Dynamic Array-based Lists

- The initialization will be implemented with the use of malloc function.

```
1 void init (List *l, int N) {
2     l->size = 0;
3     l->capacity = N;
4     l->val = (int *)malloc(l->capacity);
5 }
```

- All other operations are the same as the ones of Static Array-based Lists.

Array-based Lists

Array-based list implementation:

Pros

- easy to understand and simple to implement.
- able to access to any element.

Cons

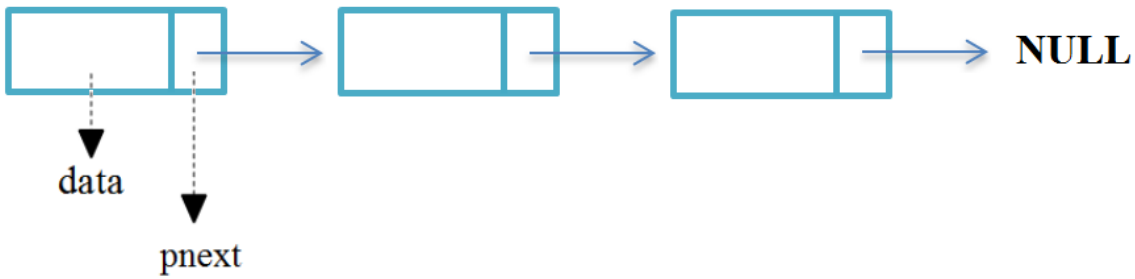
- the size has to be fixed beforehand.
- inserting or deleting an element are very difficult because we have to shift position of large number of elements.

Linked Lists

Definition

In Linked Lists, each item is placed together with the link to the next item, resulting in a simple component called a node:

- A data part stores an element value of the list.
- A next part contains a link (or pointer) that indicates the location of the node containing the next list element. If link does not point a node, then its value set to NULL (a special C++ constant in stdlib.h).



Linked Lists

- An array is a static data structure. This means the length of array cannot be altered at run time. While, a linked list is a dynamic data structure.
- In an array, all the elements are kept at consecutive memory locations while in a linked list the elements (or nodes) may be kept at any location but still connected to each other.

Linked Lists

- Successive elements are lniked by pointers and the last element points to NULL.
- The size grows or shrinks during execution of a program, it does not waste memory space like Array-based List ADT.
- Linked lists provide flexibility in allowing the items to be rearranged efficiently for inserting or deleting an element.

Lists

Basic operations are often defined

1. init(): create an empty list
2. isEmpty(): check if the list is empty
3. insert(): add new item in a list
4. remove(): remove an item from a list

Lists

Basic operations are often defined

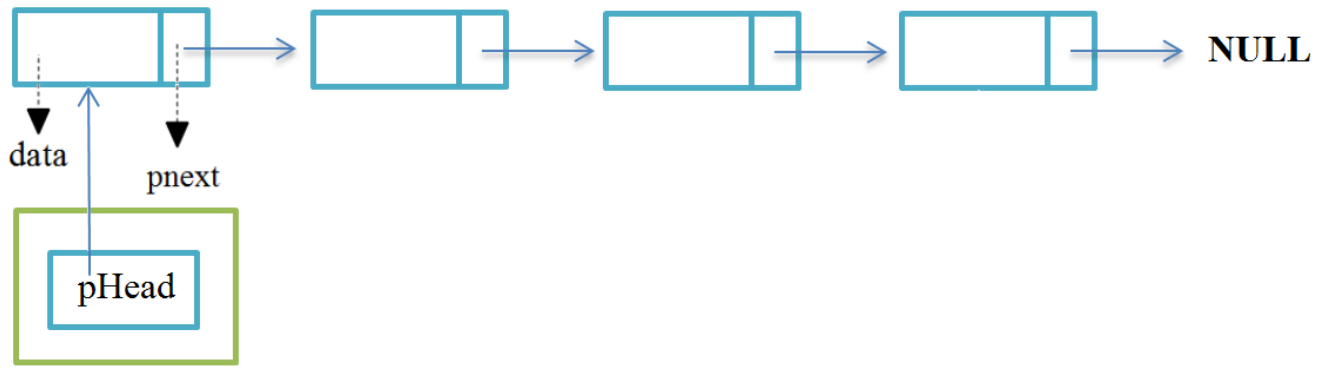
1. init(): create an empty list
2. isEmpty(): check if the list is empty
3. insert(): add new item in a list
4. remove(): remove an item from a list

Some other possible operations can be used:

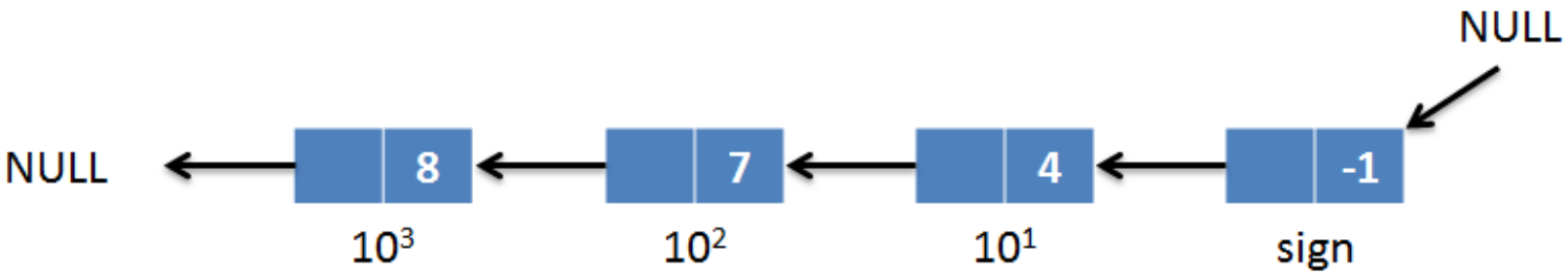
1. length(): return the length of a list
2. search(): search a specific element in a list
3. display(): display a list
4. sort(): sort all items in a list

Linked Lists

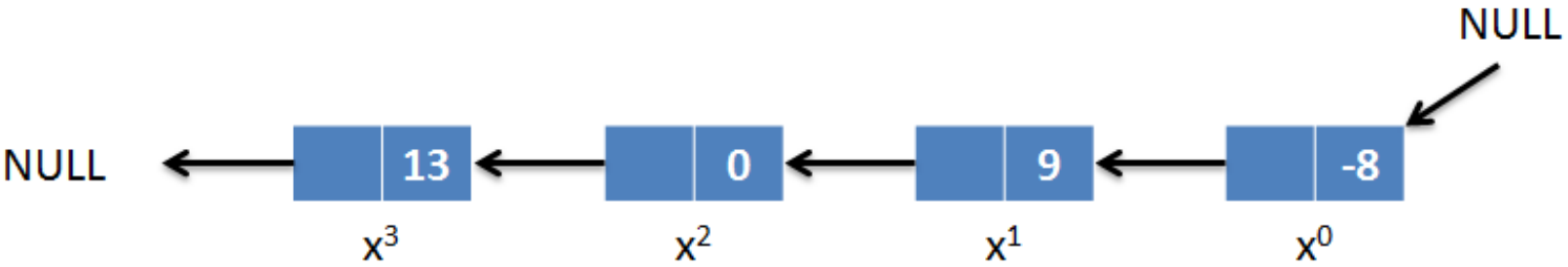
```
1  typedef struct _Node {
2      int data;
3      struct _Node *pnext;
4  } Node;
5  typedef struct _List {
6      int size;
7      Node  *pHead;
8  } List;
```



- -874



- $13x^3 + 9x^1 - 8$



Linked Lists

- A linked list must be initialized with the head node which is a NULL pointer.

```
1  void init(List *l){
2     l->size=0;
3     l->pHead=NULL;
4  }
5  Node* initNode(int val){
6     Node *node = (Node *) malloc(sizeof *node);
7     node->data = val;
8     return node;
9  }
```

- A list is empty if there is no element or the head node is NULL.

```
1  int isEmpty(List *l){
2   return (l->size == 0);
3  }
```

Linked Lists

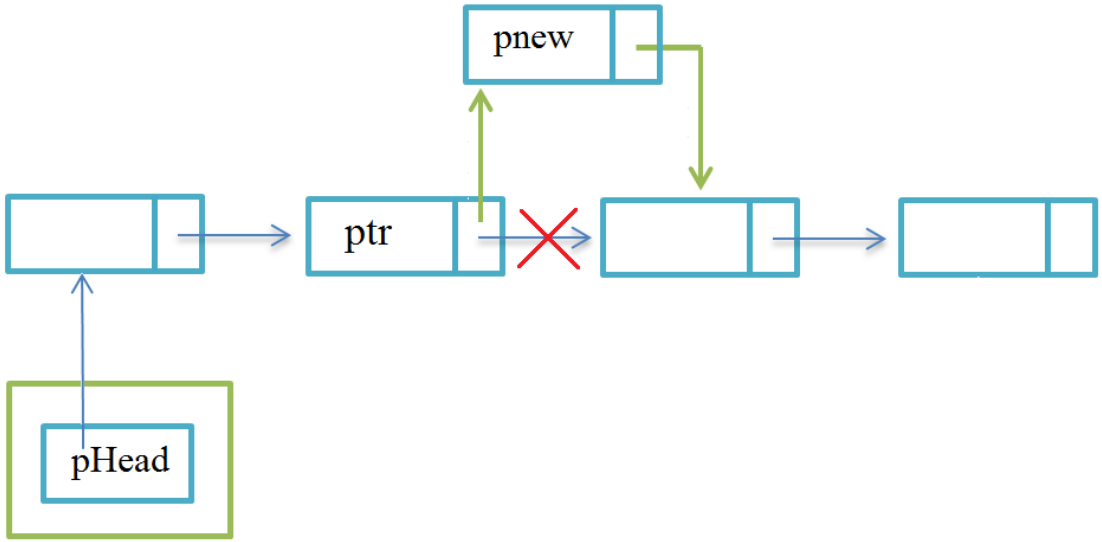
There are two cases while inserting new elements into a list:

1. If the list is empty, its first element will be added at the list head.

```
1 void insertFirst(Node *pnew, List *l){
2     if (isEmpty(l)){
3         l->size ++;
4         l->pHead = pnew;
5     }
6 }
```

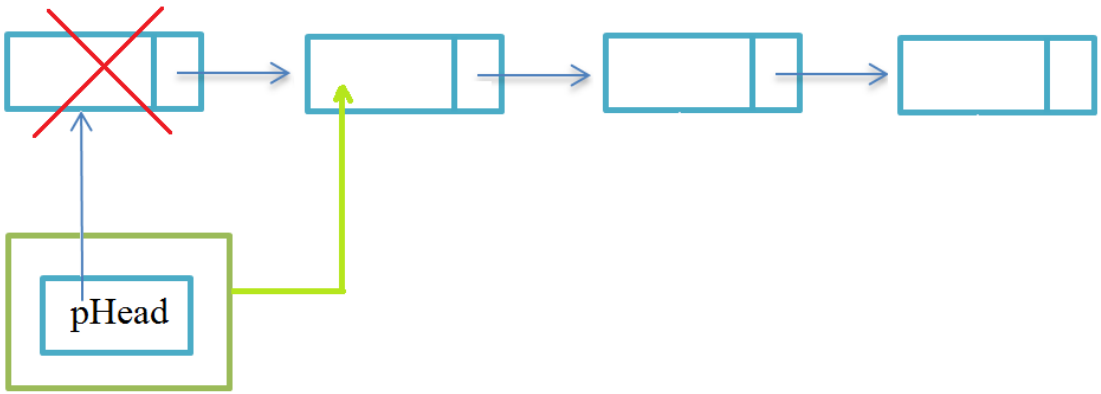
2. A new node is linked after a specific node.

```
1  void insert(Node *pnew, Node *ptr, List *l){
2    pnew->pnext = ptr->pnext;
3    ptr->pnext=pnew;
4    l->size++;
5  }
```

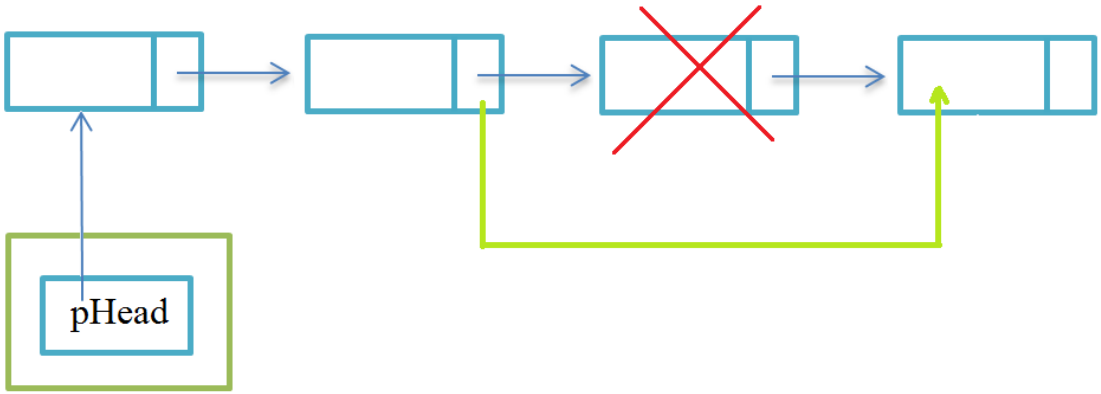


There are two cases while removing an element from a list:

1. If the first element is removed, the head should point to the next node of the first element.



2. In the another case, any node is removed, a new link is created between the previous node and the next node.



free() is used to free the allocated memory.

```
1  void remove(List *l, int val){
2       Node *p = l->pHead;
3       if (p->data == val){
4            l->pHead = p->pnext;
5            free(p);
6            l->size--;
7            return;
8       }
9       Node *q = p->pnext;
10      int cnt = 1;
11      while ((p->data != val) && (cnt <length(l))){
12           q = p;
13           p = p->pnext;
14      }
15      if (p != NULL){
16         q->pnext = p->pnext;
17         l->size--;
18         free(p);
19      }
20 }
```

Comparisons

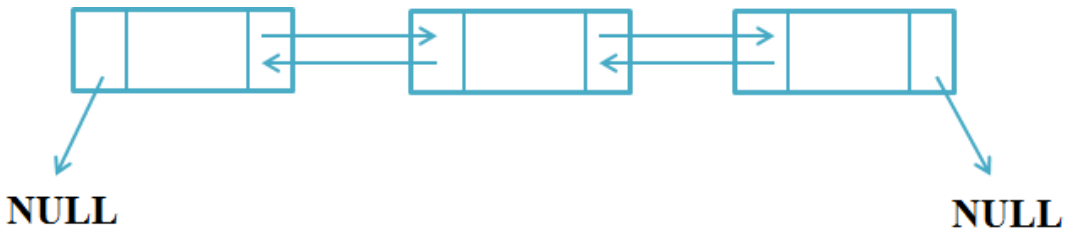
Comparisons of complexity for different list implementations

	Linked list	Array
Indexing	O(n)	O(1)
Insert/delete at the begining	O(1)	O(n)
Insert/delete at the end	O(n)	O(1)
Insert/detele at the middle	searching time	shifting time

Doubly Linked Lists

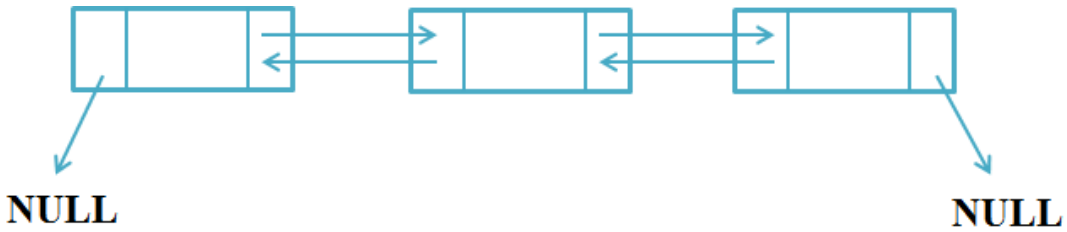
Doubly Linked Lists is a variation of Linked List in which navigation is possible in both ways either forward and backward.

- A data part stores an element value of the list.
- A next part contains a link indicating the next element.
- A previous part contains a link indicating the previous element.

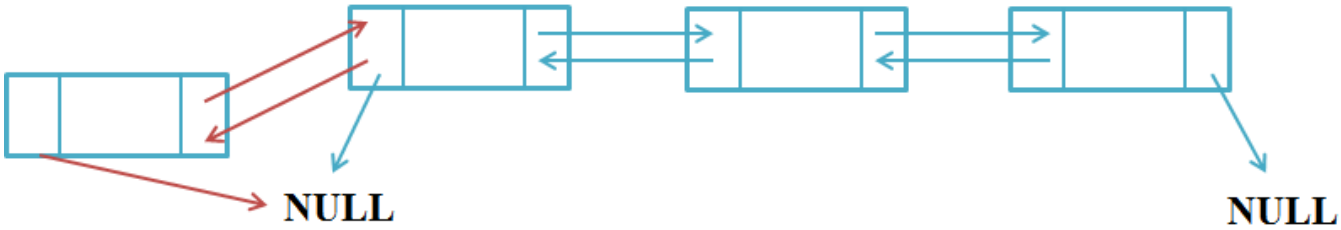


Doubly Linked Lists

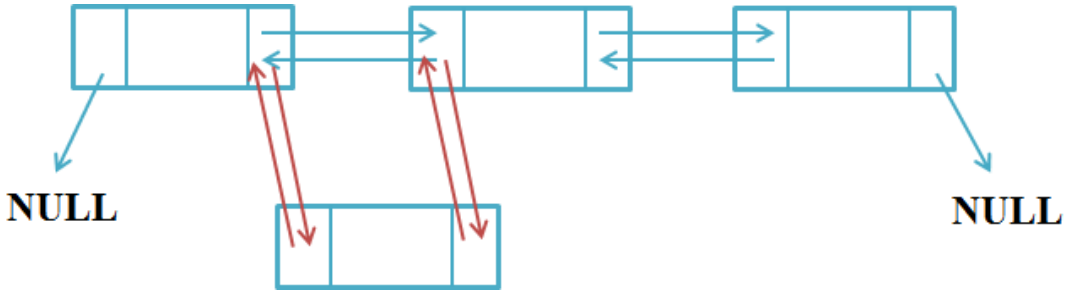
```
1  typedef struct _Node {
2      int data;
3      struct _Node *pnext;
4      struct _Node *pres;
5  } Node;
6  typedef struct _List {
7      int size;
8      Node  *pHead;
9      Node *pTail;
10 } List;
```



Doubly Linked Lists



Insertion at the head of Double Linked Lists



Insertion at the middle of Double Linked Lists

Doubly Linked Lists

Basic operations are often defined

1. init(): create an empty list
2. isEmpty(): check if the list is empty
3. insert(): add new item in a list. It is possible to add new elements to either the head or the tail.
4. remove(): remove an item from a list

Doubly Linked Lists

Basic operations are often defined

1. init(): create an empty list
2. isEmpty(): check if the list is empty
3. insert(): add new item in a list. It is possible to add new elements to either the head or the tail.
4. remove(): remove an item from a list

Some other possible operations can be used:

1. length(): return the length of a list
2. search(): search a specific element in a list
3. display(): display a list from left to right or from right to left.
4. sort(): sort all items in a list

Doubly Linked Lists

Advantages

- Linked lists are a dynamic data structure, which can grow and be pruned, allocating and deallocating memory while the program is running.
- Insertion and deletion node operations are easily implemented in a linked list.
- They can reduce access time and may expand in real time without memory overhead.
- Linear data structures such as stacks and queues are easily executed with a linked list (next chapter).

Doubly Linked Lists

Disadvantages

- Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access.
- Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list.
- Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back-pointer.

Linked List Applications

Applications

- Linked lists can be implemented to represent structured data i.e. graph, tree in the domain of web, social network, etc.
- Multiply linked lists, Circular Linked lists are avaiable to deal with specific issues.