Lists
(Section 5)

Lists, linked lists
Implementation of lists in C
Other list structures
List implementation of stacks, queues, priority queues

By:
Pramod Parajuli,
Department of Computer Science,
St. Xavier's College,
Nepal.

Lists, linked lists

List definition

List is a data structure that holds elements in a continuous order. The insertion/deletion is possible at / from any point / location but not far away from the existing elements.

Contiguous lists

7	19	67	99	8		
0	1	2	3	4		

If a new element 10 has to be inserted in position 3, then the elements in position 3 and 4 are shifted towards right and the new element is inserted.

7	19	67		99	8	
0	1	2	3	4	5	

7	19	67	10	99	8	
0	1	2	3	4	5	

And if, an element at position ‘2’ has to be removed, the value is read and then all of the elements on the right hand side are shifted towards left to make it contiguous.

7	19		10	99	8	
0	1	2	3	4	5	

7	19	10	99	8		
0	1	2	3	4		

Insertion/deletion far away from current elements is invalid.
For example;
Insertion at position 6 in the given list is not allowed.

7	19	67	99	8		
0	1	2	3	4		

It is not always the case that the insertion is not allowed for given situation, if inserted, then we will have burden of moving the elements down to the last position.

Disadvantage:

1. Inefficient use of memory space. For example, if an array for 1,000 elements is declared but only 500 elements will be there in the list then memory space for 500 elements will just be wasted. And in another case, if the user wants to store 2,000 elements at run time, then he have no way at all to store extra 1,000 elements.
2. To achieve the contiguous property, every time an element is inserted or deleted, the elements on the right side of the current insertion/deletion position have to be shifted right or left. This also requires lots of processing time.

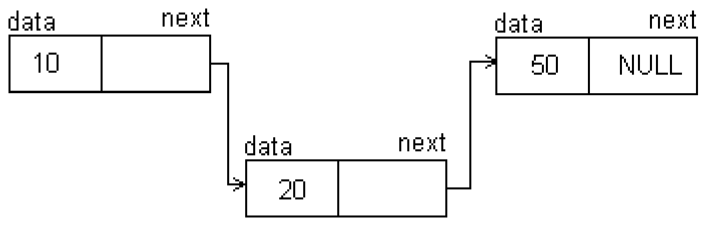
Linked lists

The contiguous list has drawbacks of inefficient use of memory and overhead of shifting the elements each time an element is inserted or deleted. To overcome the drawbacks, linked lists are introduced. This problem is due to the fixed/static memory allocation for the elements during compile and linking time.

What if we can make some kind of mechanism so that we create home for each and every element whenever needed and clear the home whenever the element is removed??

The general idea behind the linked list is that, to hold each and every element, we create home for the element during run time. Then we make some link between those homes and make a logical chain of the elements. If an element is no longer needed, then the memory allocated for the element is freed. For these purpose the C language provides memory allocation and de-allocation functions.

Let's see how actually the elements are organized.



Implementation of lists in C

Array implementation (contiguous list) - Look at (program – 19)

Data structure

```
struct List{
      int item[LISTMAXSIZE];
      int lastPos;
};
```

Initialization

```
      lastPos = -1
```

Insert operation

```
1.    get list structure, position to insert and value to insert.
2.    if lastPos == maxSize-1 OR position > lastPos + 1
            return insert-Invalid.

      else if position <= lastPos
            shift elements right
            item[position] = x
            lastPos++;

      else
            item[++lastPos] = x
```

Remove operation

```
1.    get list structure, position of element to remove
2.    if lastPos == -1 OR position > lastPos
            return remove-Invalid

      else if position < lastPos
            *x = item[position];
            shift elements left
            lastPos--;

      else
            *x = item[lastPos--];
```

Find operation

```
1.    get list structure, element to search
2.    start from the beginning
3.    compare the value with each and every element until lastPos is
      not reached.
4.    if found, return
5.    if exceeds lastPos, then return invalid.
```

Dynamic implementation (linked list)

Data structure

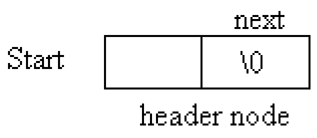
We want to store information about the data, and the address of next same type of node that contains data and link to other nodes.

```
struct linkedList{
      int data;
      struct linkedList *next;
};
```

Insert operation

Initialization:

At the first time, we don't have any memory location. Therefore, we create one node to hold the starting address of the list. The starting address (i.e. variable to hold the starting address) do not hold any data at all.



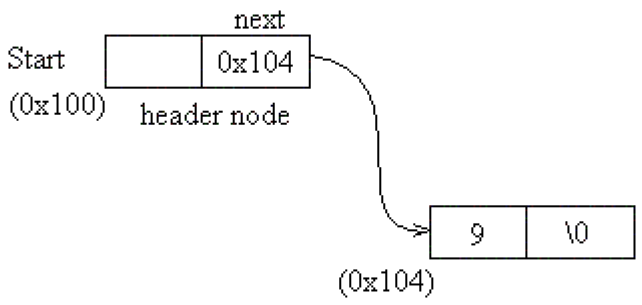
To make such a node, we just make an instance of the given structure.

```
struct linkedList *start;
start -> next = NULL;
```

Insertion for first time:

Up to now, we have one node that contains nothing and that point to nothing. To add new node, we must first allocate the required space, store the data and then make a link.

While creating space for new element, we are not sure where the new memory space will be allocated. The address of new memory location is returned by the 'malloc' function. This address is copied to the 'next' field of previous node so that while traversing, we know where the new node lies. The newly added node is the last node in this linked list. Therefore to indicate that this is the last one, we must put 'NULL' in the 'next' field of the node.

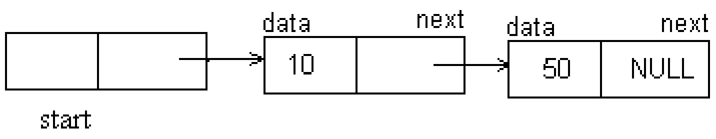


Insertion for second time:

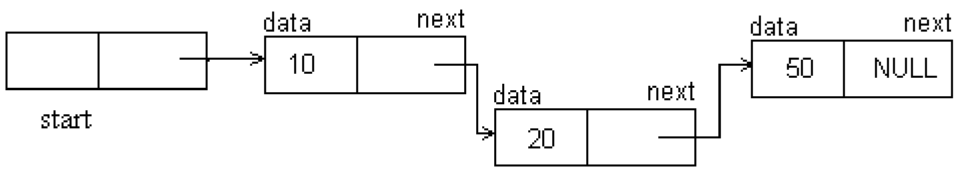
At this point, we know the starting address (0x100) of the linked list. But we don't know what the address of the last node is? So, we start from the 'start' node and look at the 'next' field, then go to the node in the 'next' field. Again the same checking is done until 'NULL' is not found in the 'next' field. If 'NULL' is found, then new memory space is created as in previous example and then new data is added and linked.

Insertion in the middle:

If a new node is to be inserted in the middle, then the location at which the insertion is to be done must be known. For example, if we have a linked list as shown;

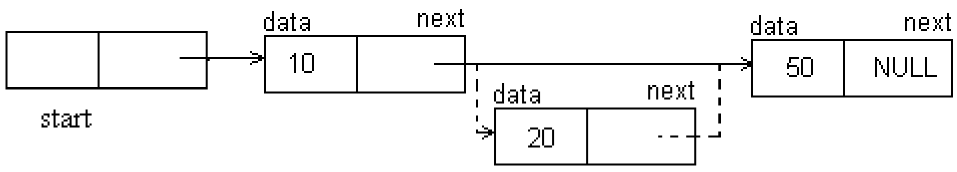


and we need to add new element after data '10', then we start from the 'start' node and traverse until we don't find node with data '10'. Then we save the address of the node pointed by node containing '10' i.e. address of node with data '50'. Now, a new node is created and then the data is copied in the data section. Then, the address of new node is saved in node '10' and address of node '50' is saved in the 'next' field of new node.

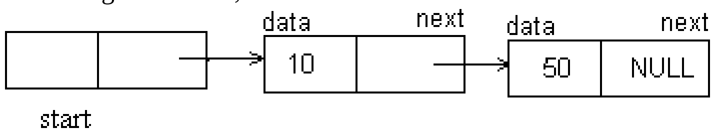


Delete operation

To delete a given node, first, the address of the node is found, and then the remaining elements are re-linked so that they also form a linked list, then the memory associated with the node is released.



And the resulting list will be;



To delete a given node, we must know either where the node lies in terms of position or what the value in the node is?

If the position of the node is given;

Then, a counter is used and the nodes are traversed for the given number of count. Then the node is deleted.

If the value in the node is given;

Then, the nodes are traversed and for each and every node, the value inside the node is compared. If the right value is found, then the node is deleted and remaining nodes are re-linked.

Display elements

To display elements, we start from the starting/head node and then traverse the nodes and display the values until we do not encounter with NULL in the next field of the node.

Find operation

To find a given node, we also start from the head node and then compare the values. To each and every successive traverse, count up by 1. After finding the node with appropriate value, we return the position of the node.

If we could not find the node with given value but encountered end of the list i.e. NULL, then we say, the find operation was unsuccessful.

Revision in C for memory management function

Two primary functions for memory management in C are;

i. malloc
ii. free

malloc

Let's see the manual in Borland C++ 3.1 for the malloc function.

- Allocates memory

Declaration: void *malloc(size_t size);

Remarks:

malloc allocates a block of size bytes from the memory heap. It allows a program to allocate memory explicitly as it's needed, and in the exact amounts needed.

The heap is used for dynamic allocation of variable-sized blocks of memory. Many data structures, such as trees and lists, naturally employ heap memory allocation.

Return Value:

- On success, malloc returns a pointer to the newly allocated block of memory.
- On error (if not enough space exists for the new block), malloc returns NULL. The contents of the block are left unchanged.
- If the argument size == 0, malloc returns NULL

free

frees allocated blocks

Declaration: void free(void *block);

Remarks:

free deallocates a memory block allocated by a previous call to calloc, malloc, or realloc.

Return Value: None

Creating a new node

```
linkedList * createNode(void){
      linkedList *newnode;
      newnode = (linkedList *) malloc(sizeof(struct llist));
      return newnode;
}
```

Insert function

i. Get starting point of the linked list, value to put in the new node
ii. traverse up to the end in the linked list
iii. create a new node
iv. put the value to the data field
v. put the address of new node in the ‘next’ field of the last node and make the new node last node of the linked list.

```
int insert(linkedList **previousNode, int value){
      linkedList *tempNode;
      tempNode = previousNode;

      while(tempNode -> next != NULL){
            tempNode = tempNode -> next;
      }
      previousNode = tempNode;

      // let's create a new node
      tempNode = createNode();
      tempNode -> data = value;
      tempNode -> next = previousNode -> next;
      previousNode -> next = tempNode;
      return SUCCESS;
}
```

Delete function

```
int del(linkedList *previousNode, int *value){
      linkedList *tempNode;
      tempNode = previousNode;

      if(previousNode -> next == NULL)
            return DEL_INVALID;
      while(tempNode -> next -> next != NULL){
            tempNode = tempNode -> next;
      }

      *value = tempNode -> next -> data;
      tempNode -> next = tempNode -> next -> next;
      freeNode(tempNode -> next);
      return SUCCESS;
}
```

Display elements function

```
void printLList(linkedList *start){
      linkedList *nextNode;
      nextNode = start;
      while(nextNode -> next != NULL){
          printf("\nData read at node 0x%-8X is %5d,
                  next node is at : 0x%-8X",
            nextNode -> next,
            nextNode -> next -> data,
            nextNode -> next -> next);

            nextNode = nextNode -> next;
      }
      printf("\n");
}
```

Find function

```
int findNode (linkedList *start, int value){
      linkedList *nextNode;
      int counter = 0;
      nextNode = start;

      while(nextNode != NULL && nextNode -> data != value){
                  nextNode = nextNode -> next;
                  counter++;
      }

      if(nextNode == NULL){
            return SEARCH_UNSUCCESSFUL;
      }

      return counter;
}
```

Look at program 20.

Presentation 1: Array implementation of linked lists.
Reference: Langsam, Augenstein, Tanenbaum, *Data Structures using C and C++*, page 203.

Other list structures

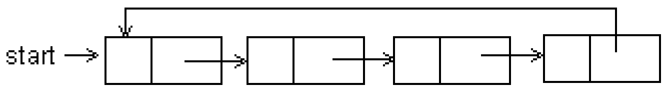
Header Nodes

For the implementation of linked list, we must hold the address of the first node. To hold the address, a new pointer is required to the same kind of node. While creating the pointer, we have access to the information part also.

Some operations like, finding the number of nodes in the linked list are time consuming. The information part in the **header node** is used to store total number of nodes in the list or in case of circular linked list implementation; the header node contains information saying that this is the header node and etc.

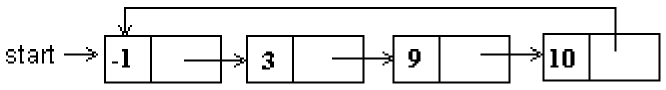
Circular linked list (CLL)

A general linked list is linear i.e. it have one starting end (start/header node) and another terminating end (last node). In the 'next' field of last node, a value of NULL is kept indicating that this is the end of the linked list. Now, what we do here is; the address of the first/header node is kept in the 'next' field of the last node.



In circular list, if we start from the first node and then start traversing the list, we won't be able to know where the end of the list is. The traversal might to infinite times. Here, we use the information field in the header node to indicate that this is the starting point of the linked list.

One possible implementation could be putting '-1' in the information field of the header node for a linked list of the integers. Similar kind of tricks can be implemented for other scenario.



The insertion and deletion process are same as that of linear linked list but for the find operation, we search a given value until we do not encounter a node with value '-1'.

The circular list facilitates entry at any point in the list and still being able to traverse the whole linked list.

Uses

Circular lists are useful in lots of problem solving techniques. One classic problem that can be solved using circular list is '**Josephus problem**.'

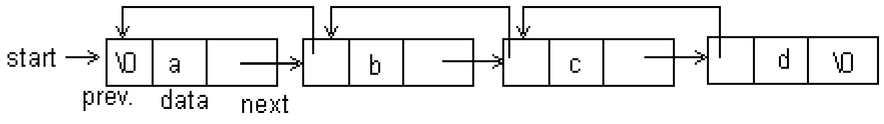
Another real world implementation of circular list is for token management in ring topology. In this topology, the computer that has the token can send the data. The token is generated by the computers themselves in the ring. This indicates the header node. After finishing the data send, the token is passed to the next computer and so on. In this topology, the header computer might change during the course of time.

Look at program 21.

Doubly linked list (DLL)

In linear and circular linked list, there is no way so that we can back trace the nodes. In these linked lists, to delete a given node, we stay in the node before the deleting node; we read the next value of the node, delete the data and then re-link the remaining nodes. It means, if the address of a node is given that needs to be deleted, then the deletion operation is impossible. It's because, after deletion we must re-link the remaining nodes and in linear and circular linked lists we can not traverse back to the previous node. And without going to previous node, the re-linking process is impossible.

To overcome the problem, doubly linked list is introduced. In the nodes of DLL, the addresses of previous as well as next nodes are stored.



To represent a node in DLL in C, we use;

```
struct DLL{
      struct DLL *previous;
      int data;
      struct DLL *next;
};
```

Insertion operation

Let’s say, we want to insert a node after a given node ‘previousNode’.

To do that, we have the address of ‘previousNode’ or we know after how many nodes it lies. After getting the address, let’s create a new node called ‘new’.

```
new = createNode();

newNode -> info = value;

newNode -> previous = previousNode;
newNode -> next = previousNode -> next;
previousNode -> next = newNode;
```

Deletion operation

The major benefit of DLL is during deletion operation. If the address of deleting node is already known, then no traversal is required.

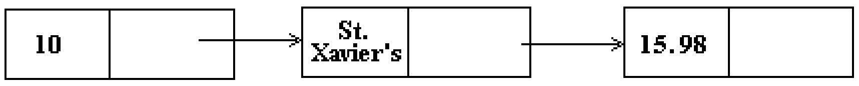
```
deletingNode -> previous -> next = deletingNode -> next;
deletingNode -> next -> previous = deletingNode -> previous;
free(deletingNode);
```

Look at program 22.

Think of how a ‘doubly circular linked list’ is organized.

Generalized linked list (non-homogeneous list)

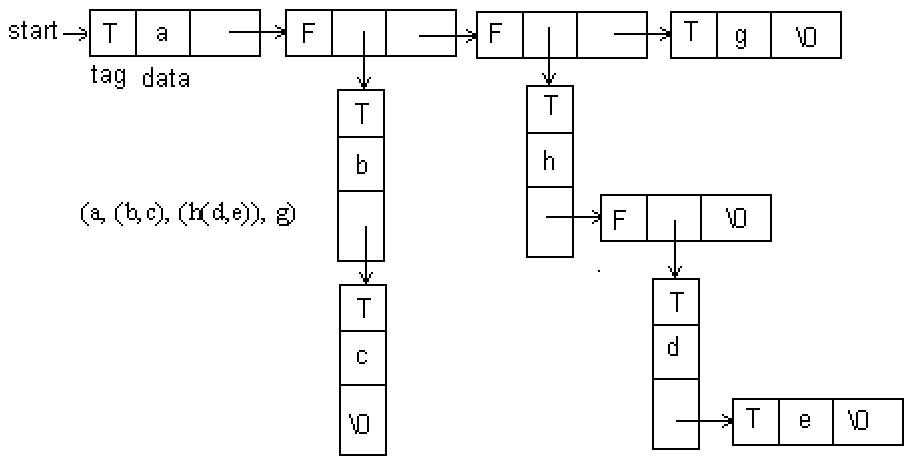
Until now, same data types are being stored in a given linked list. The general idea is to bind nodes with any kind of data type and make a generalized linked list. Such linked list is also known as non-homogeneous linked list.



Not just primitive data types, other different abstract data types can also be the contents of the nodes in a general linked list.

e.g.

a list **(a, (b,c), (h(d,e)), g)** can be represented as;



Note:

Every generalized list can be represented using node structure.

tag = T/F	data / link	link

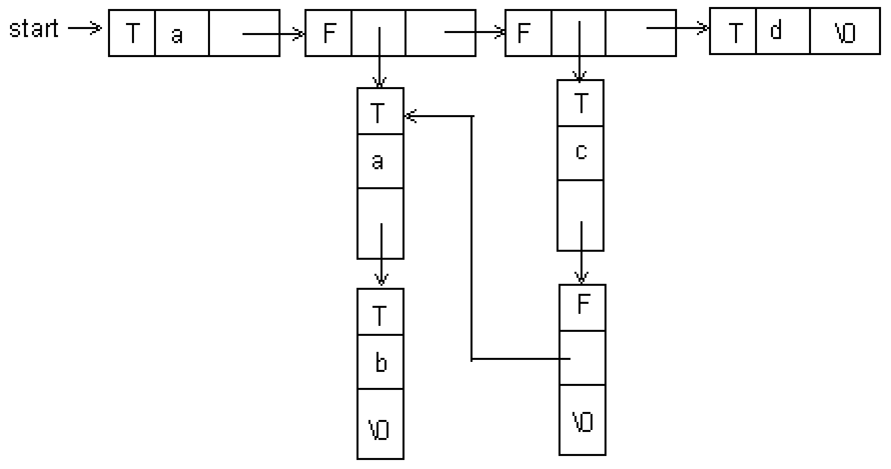
```
#define TRUE 1
#define FALSE 0

enum Boolean {FALSE, TRUE};

struct GenListNode{
      Boolean tag;
      union{
            char data;
            struct GenListNode *dlink;
      };
      struct GenListNode *link;
};
```

A generalized list may contain some of the elements shared.

Ex: Represent a list **`(a, (a, b), (c, (a, b)), d)`**



Look at program 23.

List implementation of stacks, queues, priority queues

Linked list as stack

```
struct stack{
      int info;
      struct stack *next;
};

main(){
      struct stack *tos;
}
```

Here, always point the ‘tos’ to the first node. Add node before the node pointed by ‘tos’ for PUSH, and remove node pointed by ‘tos’ for POP. During this course, the address of the first node is changed always, therefore, remember the address of newly created node.

Look at program 24.

Linked list as queue

```
stack queue{
      int info;
      struct queue *next;
};

struct queue *front, *rear;
rear = front = NULL;
```

The front pointer points to the starting node of the linked list and rear pointer points to the last node of the linked list.

For first insertion, rear and front both point to the same node.

While adding, insert a node after the node pointed by rear pointer i.e. at the rear end.

```
rear -> next = new node
rear = newnode
```

While removing, delete a node pointed by front pointer i.e. at the front end.

```
tempNode = front -> next
free(front)
front = tempNode
```

Empty, if rear and front point to the same node.

```
rear = front = NULL
```

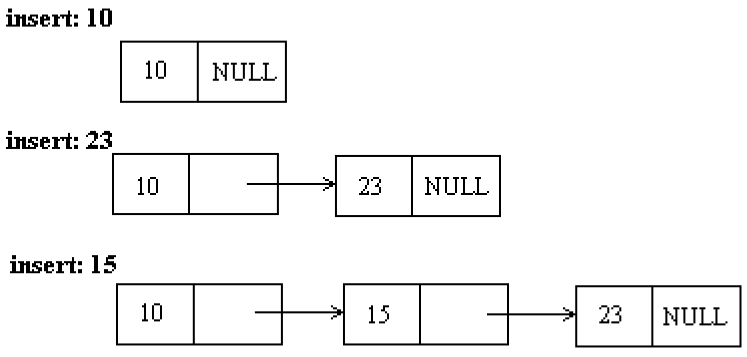
Look at program 25.

Linked list as priority queue

Due to the dynamic nature of liked list, the implementation of priority queue has some critical issues.

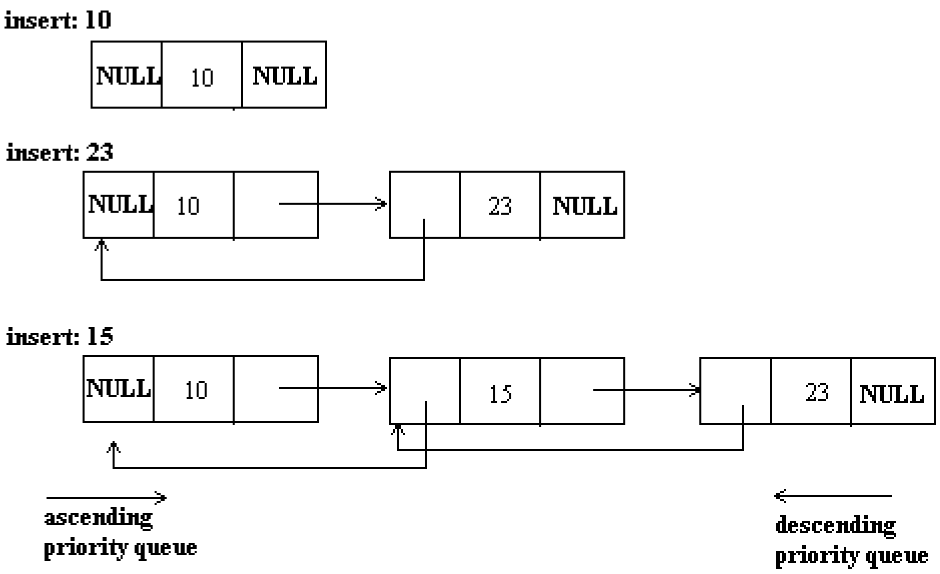
If the priority queue is implemented directly in a normal linked list, then we will have burden of searching the maximum or minimum valued node and remove it each and every time we want to pop the element.

One better solution can be; to write a function 'addNode' that places the node with given value at the appropriate position. This way, a sorted linked list is maintained automatically during the insertion operation.



Linked List representation of priority queue

If implemented using doubly linked list, then we can start from both of the end and so the same linked list can be used for 'ascending' or 'descending' priority queues.



Look at program 26.

Linked list implementation of long integers

The size of primitive data types is limited in each and every system that is implemented. Similar problem is there with implementation of hardware as well as software. Currently, we are concerned with the compilers. The maximum amount of different data set represented by a given data type is fixed.

For example, if 16 bits are assigned for a data type then it can represent 2^{16} = 65536 number of different data set. If we require representing more than 65536, then we search for a different data type that has more bits. In straight forward manner, if we start from 0, then we can go up to 65535. There are some optimization tricks to use a given data type to represent numbers for larger range.

For example, if we divide the 16 bit data type into slots of 4 bits and 12 bits, and assign 12 bits for mantissa, and 4 bits for exponent. In this way, we can represent numbers up to 4096^{16}, using the same 16 bit data type!!

For more details, read chapter – 1 "Introduction to Data Structures" in *Data Structures using C and C++* by Langsam, Augenstein, Tanenbaum.

Let's say, we have two numbers

534598730598723905029378 5 and
709870767263489712403 78

Can you write a program that asks such numbers as input and adds the numbers in C?

Seems quite difficult?

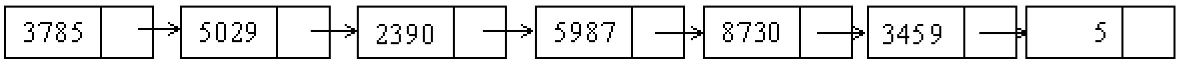
Linked lists can be used to solve such kind of problems.

Linear/circular linked list

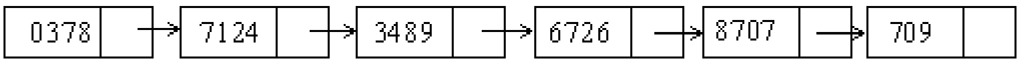
Such kind of arbitrarily large numbers can be represented in linked list and then added. For such task, we break the number into more manageable parts. Each part can be represented by using a single primitive data type. Then all of them are linked using linked list so that the parts still hold the original meaning.

e.g.

To represent 5345987305987239050293785, we use,

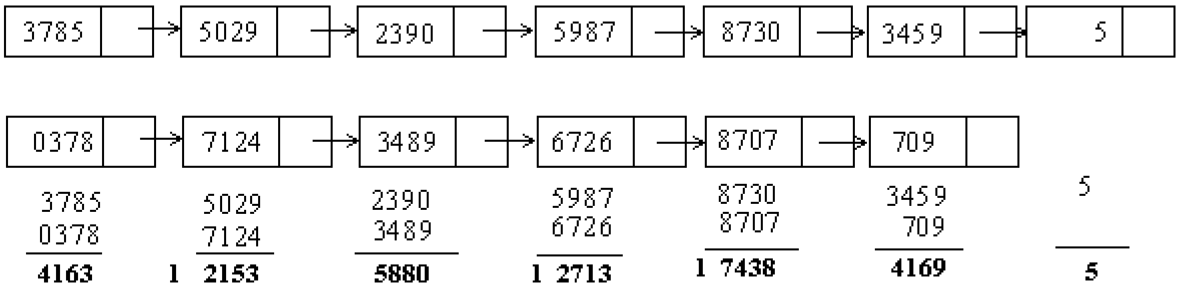


Similarly, the number 709870767263489712403 78 is represented as;

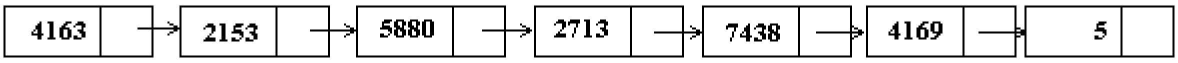


If such organization (4 digits per node) is used, then the data field can be represented by 'int' data type. But if more digits are used (5 digits) then the value might go up to 99999, and this number can not be represented by using 'int'. Therefore be careful while designing the representation.

Now, to add these numbers, each corresponding nodes are added and carry is passed to next step.



Resulting number representation



Presentation 2: Doubly linked list implementation of large integers

Linked list implementation of polynomials and sparse matrix

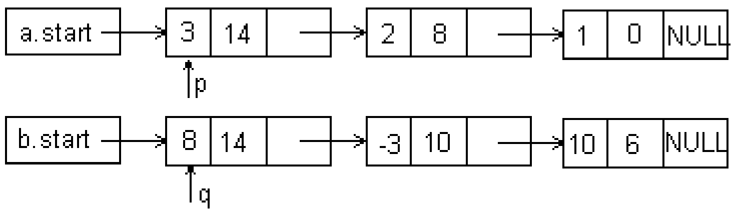
Polynomials

Consider two polynomials:

A(x) $= 3x^{14} + 2x^{8} + 1$
B(x) $= 8x^{14} – 3x^{10} + 10x^{6}$

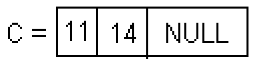
coefficient	exponent	next

```
struct term{
        float coef;
        int exp;
        struct term *next;
};
```

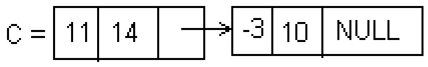


Addition

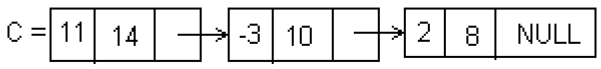
i) p->exp == q->exp



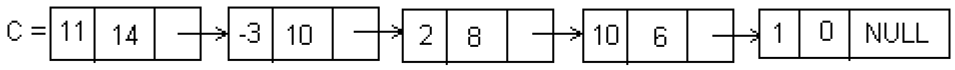
ii) p->exp < q->exp



iii) p->exp > q->exp



iv) .....



Look at program 27.

Sparse Matrix

Matrix having too few non-empty cells.

e.g. a 100 * 100 matrix with only 50 non-zero entries.

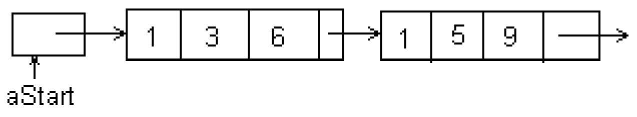
Consider a sparse matrix as;

0	0	6	0	9	0	0
2	0	0	7	8	0	4
10	0	0	0	0	0	0
0	0	12	0	0	0	0
0	0	0	0	0	0	0
0	0	0	3	0	0	5

Row	Col	Value	Next Link

```
struct sm{
      int row, col, value;
      struct term *link;
};

struct sm *aStart;
aStart = NULL;
```

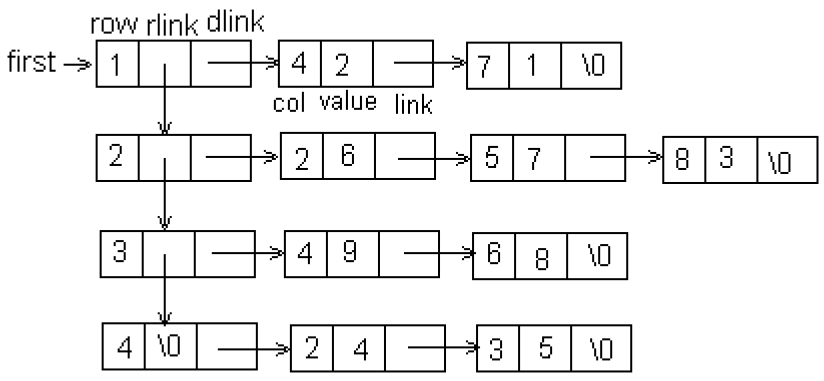


Multi Linked List representation

Generalized list implementation.

Consider the matrix:

0	0	0	2	0	0	1	0
0	0	0	0	7	0	0	3
0	0	0	9	0	8	0	0
0	4	5	0	0	0	0	0



```
// for the data
struct term{
      int col, value;
      struct term *link;
};

// for row indicators
struct spmatrix{
      int row;
      struct spmatrix *rowLink;
      struct term *dlink;
};
```

Look at program 28, 29.

To do:

Write a program that allows users to insert and delete node at any position in a given linked list, circular linked list, and doubly linked list.