Data Structure and Algorithm
handout 3
Linked List

Apurba Sarkar

August 23, 2017

1 Motivation

So far, we have studied the representation of two simple data structures stacks and queues using an array and a sequential mapping. These representations had the property that successive elements of the data object are stored a fixed distance apart. Thus, (i) if the element $a(i, j)$ of a table is stored at location L_{ij}, then $a(i, j + 1)$ is at the location $L(i, j) + c$ for some constant c; (ii) if the i^{th} element in a queue is at location $L(i)$, then the $i + 1$st node is at location $(L(i) + c)$ **mod** n for the circular representation; (iii) if the topmost element of a stack was at location $L(T)$, then the node beneath it was at location $L(T) - c$, etc. These sequential storage schemes proved adequate for the operations such as access to an arbitrary element in a table, insertion or deletion of element within a stack or queue. However when a sequential mapping is used for ordered lists, operations such as insertion and deletion of arbitrary elements become very expensive. For example, consider the following ordered list of integers:

$$(10, 15, 20, 25, 30, 35, 40, 50, 55, 60, 65, 70, 75, 80)$$

Suppose, we want to insert 45 in the list. If we are using an array to keep this list, then the insertion of 45 will require us to move elements already in the list either one location higher or lower. We must either move $50, 55, 60, \ldots, 80$ or else move $10, 15, 20, \ldots, 40$ to make room for 45. If we have to do many such insertions into the middle, then neither alternative is attractive because of the amount of data movement required. On the other hand, suppose we decide to remove 50. Then again, we have to move many elements so as to maintain the sequential representation of the list.

Another situation where sequential representation is proved to be inadequate is maintaining several ordered list of varying sizes. This is because if we store each list in a different array of maximum size, storage may be wasted. On the other hand if we try to maintain the lists in a single array a potentially large amount of data movement is required. This was explicitly observed when we talked about representing several stacks and queues in a single array. We shall present an alternative way of representing ordered lists which will significantly reduce the time needed for arbitrary insertion and deletion. An elegant solution to this problem of data movement in sequential representations is achieved by using linked representations. The main problem of sequential representation is

that successive elements of a list are located a fixed distance apart and that causes the movement of data in case of addition or deletion to maintain the sequential representation costly. Instead of moving the data can we somehow link the data to maintain the sequential ordering? The answer is yes. A linked representation exactly does it. It allows the element to be scattered anywhere in the memory. Another way of saying this is that in a sequential representation the order of elements is the same as in the ordered list, while in a linked representation these two sequences need not be the same. To access elements in the linked list in the correct order, with each element the address or location of the next element in that list is stored. Thus, associated with each data item in a linked representation is a pointer to the next item. This pointer is often referred to as a link. In general, a node is a collection of data, $DATA_1, \ldots, DATA_n$ and links $LINK_1, \ldots, LINK_m$. Each item in a node is called a field. A field contains either a data item or a link.

Figure 1 shows how the data we have considered may be represented in memory by pointers. We can think that data part of the nodes are stored in a one dimensional array called *data* and the corresponding pointers are stored in a one dimensional array *link*. It is to be observed that the data no longer appears in sequential order. Linked representation relax this restriction and allow them to be scattered anywhere in the memory. To maintain the real order, the second array *link* is used. It contains the pointers to the elements in the *data* array.

The usual way of drawing the linked list is an ordered sequence of nodes with links being represented by arrows as in Fig. 2. The general convention is that the name of the pointer variable that points to the first node of the list is used as the name of the entire list. Thus the list in Fig. 2 is the list f. In this case we do not explicitly put in the values of the pointers but simply draw arrows to indicate that they are there. This is convince ourselves the facts that (i) the nodes do not actually reside in sequential locations, and that (ii) the locations of nodes may change on different runs. Therefore, when we write a program which works with lists, we almost never look for a specific address except when we test for zero.

	data	link
1	10	11
2	35	4
3		
4	40	7
5		
6	30	2
7	50	20
8		
9		
10		
11	15	15
12	25	6
13		
14		
15	20	12
	⋮	⋮

Figure 1: Non sequential representation

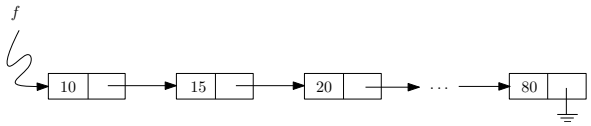


Figure 2: Usual way to draw a linked list

Let us now see why it is easier to make arbitrary insertions and deletions using a linked representation (i.e. linked list) rather than a sequential representation (i.e. array). To insert the data item 45 between 40 and 50 the following steps are adequate:
(i) get a new node which is currently unused; let its address be x;
(ii) set the *data* field of this node to 45;
(iii) set the *link* field of x to point to the node after 40 which contains 50;
(iv) set the *link* field of the node containing 40 to x.
Figure 3 shows the modification of the arrays *data* and *link* after insertion of 45 and Fig. 4 shows the same thing using the usual arrow representation of the linked list. The important thing to notice here is that to insert a new element in the list we did not have to move any data elements which were already in the list. We have overcome the data movement but it is not for free we have done it at the expense of the storage needed for the second field, *link* which is not severe a penalty. Now, suppose we are interested in deleting 45 from the list.

	data	link
1	10	11
2	35	4
3		
4	40	10
5		
6	30	2
7	50	20
8		
9		
10	45	7
11	15	15
12	25	6
13		
14		
15	20	12
	⋮	⋮

Figure 3: Insertion of 45

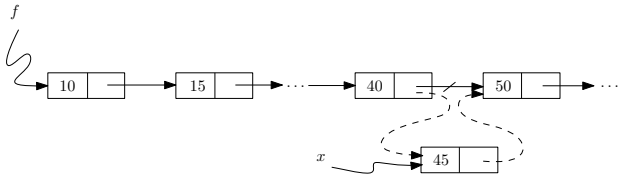


Figure 4: Insertion of 45

All we need to do is to find the element which immediately precedes 45 and set the *link* field of that node to point to the node which is immediately after 45. This is shown in figure 5

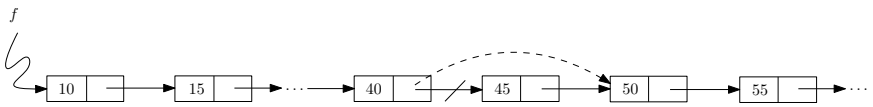


Figure 5: Deletion of 45

From our brief discussion of linked lists we see that the following capabilities are needed to make linked representations possible:

(i) A mechanism to define the structure of a node.

(ii) A means to create nodes ad needed.

(iii) A mechanism to free nodes that are no longer needed.

These capabilities are provided in almost all programming languages. We will restrict ourselves in C language only. To define a node structure in C, we need to know the type of each fields and definition is as follows:

```
typedef struct nodetype {
    int data1;
    int data2;
    int data3;
    struct nodetype * link1;
    struct nodetype * link2;
    } node;
```

The structure of the node is diagrammatically shown below in Fig. 6: For the

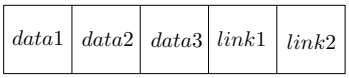


Figure 6: Structure of a node

time being we will deal with nodes that have a *data* field of *type* integer and a *link* (renamed as *next*) field and for that following definition will be used.

```
typedef struct nodetype {
    int data;
    struct nodetype * next;
    } node;
```

We will be using a procedure *new* to create a node of predefined type. If f is a pointer of type node defined above then the call *new(&f)* will create a node pointed to by f. Similarly if x, y, and z are three pointers of type node, then the call *new(&x)*, *new(&y)*, and *new(&z)* will create three nodes pointed to by x, y, and z respectively. The procedure *new()*, if written in C will look as follows:

```
void new(node ** f) {
```

```
*f = (node *) malloc (sizeof(node));
}
```

The node created by *new(&z)* can be freed by a call to a function *free()* when the node is no longer in use. Most of the programming languages also provides a special constant **NULL** that can be assigned any pointer variable. The use of **NULL** is to denote a pointer field that points to nothing or to denote an empty list. Arithmetic operations are generally not permitted on pointer variables. However two pointer variables can be compared to see if they both point to the same node. For example, if x and y are pointer variables of same type then the expressions:
$x = y$, $x \neq y$, $x = NULL$, and $x \neq NULL$
are valid while the expressions:
$x + 1$ and $y * 2$ are invalid.
Let us now see effects of some pointer assignments. Let x and y be two pointers which are pointing to two nodes with their data fields set to a and b respectively as shown in Fig. 7(a). The effect of the assignment $x = y$ will result in the situation as shown in Fig. 7(b) and finally if we do the assignment $x \rightarrow next = y \rightarrow next$ then it will have the effect as shown in Fig. 7(c).

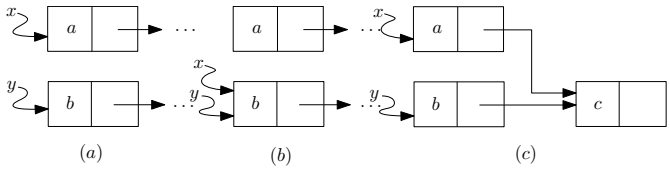


Figure 7: Effect of pointer assignments: (a) $x = y$ and (b) $x \rightarrow next = y \rightarrow next$.

In general, it does not make sense to perform arithmetic on pointers. However there are situation where it can be helpful. For example, consider an array of integers $a[10]$ (in C) and two integer pointer p and q pointing to $a[1]$ and $a[7]$ respectively as shown in Fig. 8. The result of the operation $p - q$ will be 6. This is because between p and q, 6 integers can be accommodated. However addition, multiplication, and division of pointers do not make any sense. Let us

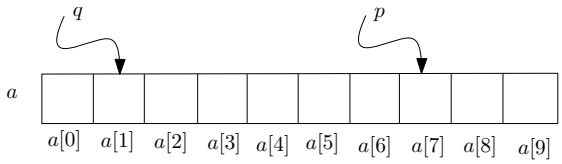


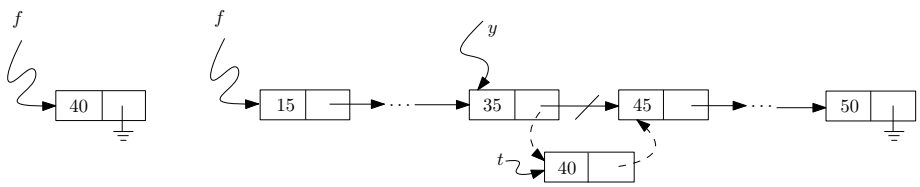
Figure 8: Structure of a node

now see few examples that perform different operations on linked list.

Example 1. *Let there be a linked list f. f = NULL, if f is empty. Let x be a pointer to some arbitrary node in the list. The following procedure inserts a node with data, say,* 40 *after the node pointed at by x. If f is NULL then if will be the first node in the list. The procedure will be called from the main program as* ***insert(&f,x);***

The procedure is pictorially depicted in Fig. 9

```
void insert(node **p, node *y)
{
   node * t ;
   new(&t);
   t->data = 40;
   if(*p == NULL)
   {
      *p = NULL;
      t->next = NULL;
   }
   else
   {
      t->next = y->next;
      y->next = t;
   }
}
```



(a) f = NULL (b) $f \neq$ NULL

Figure 9: Deletion of a node

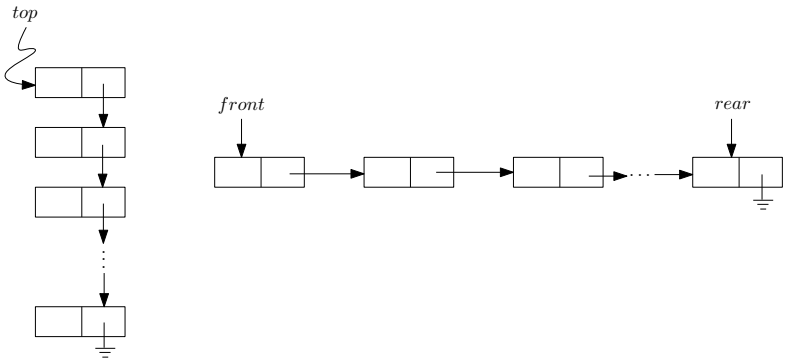
Example 2. *Let x be a pointer to some node in a linked list f as in Example. 1 and let y be the node preceding x. y = NULL if x is the first node in f (i.e., if $x = f$). The following procedure deletes node x from f. Note that the procedure will be called from the main program as* `delete(&f,x,y);`

```
void delete(node **p, node *x, node *y)
{
   if(*p == NULL)
      *p = (*p)->next;
   else
      y->next = x->next;
   free(x);
}
```

2 Linked Stack and Queues

We have seen how to implement stacks and queues using an array i.e sequential allocation. Stacks and Queues can also be implemented using linked representation. Figure 10 shows pictorial representation of linked stack and queue. It is to be noticed that the direction of links for both the stack and queue are taken

as shown so as to facilitate easy insertion and deletion of nodes. In the case of Figure 10(a), one can easily add a node at the top or delete one from the top. In Figure 10(b), one can easily add a node at the rear and deletion can be performed at the front. The procedure for `push()`, `pop()` are shown below



(a) Linked Stack (b) Linked Queue

Figure 10: Linked Stack and Queue

```
void push(int item)
{
   node *temp;
   new(&temp);
   temp->data = item;
   temp->next = top;
   top = temp;
}
```

```
int pop()
{
   node *temp;
   if(top == NULL)
   {
      printf("Stack is empty");
      return NULL;
   }
   temp = top;
   item = top->data;
   top = top->next;
   free(temp);
   return item;
}
```

Similarly, the insert an delete procedures for linked queue are shown bellow. Initially, the pointers `front` and `rear` are set to `NULL`.

```
void insertQ(int item)
{
   node *temp;
```

```
   new(&temp);
   temp->data = item;
   temp->next = NULL;
   if(front == NULL)
      front = temp;    //empty queue
   else
      rear->next = temp;
   rear = temp;
}
```

```
int deleteQ()
{
   int item;
   node *temp;
   if(front == NULL)
   {
      printf("Queue is empty");
      return NULL;
   }
   temp = front;
   item = front->data;
   front = front->next;
   free(temp);
   return item;
}
```

It is to be noted that when several stacks and queues coexists, their implementation using array is not efficient. However, we have an efficient solution using linked representation. Suppose, we wish to maintain n stacks and m queues simultaneously, then our implementation will be as follows. We use the following global arrays of pointers.
$top[i]$ = `top` of the ith stak, $1 \leq i \leq n$
$front[i]$ = `front` of the ith queue, $1 \leq i \leq m$
$rear[i]$ = `rear` of the ith queue, $1 \leq i \leq m$

The initial conditions are:
$top[i]$ = NULL $1 \leq i \leq n$
$front[i]$ = NULL $1 \leq i \leq m$

and the bondary conditions are:
$top[i]$ = NULL iff the ith stack is empty
$front[i]$ = NULL iff the ith queue is empty

with these initial and boundary conditions the implementation of multiple stacks queues are as follows:

```
void pushM(int i, int item)
// pushes item at the top of the ith stak.
{
   node *temp;
```

```
   new(&temp);
   temp->data = item;
   temp->next = top[i];
   top[i] = temp;
}
```

```
int popM(int i)
{
   node *temp;
   if(top[i] == NULL)
   {
      printf("Stack is empty");
      return NULL;
   }
   temp = top[i];
   item = top->data;
   top[i] = top[i]->next;
   free(temp);
   return item;
}
```

```
void addMQ(int i, int item)
// add item at the rear of the ith queue.
{
   node *temp;
   new(&temp);
   temp->data = item;
   temp->next = NULL;
   if(front[i] = NULL)
      front[i] = temp;
   else
      rear[i]->next = temp;
   rear[i] = temp;
}
```

```
int deleteMQ(int i)
{
   node *temp;
   if(front[i] == NULL)
   {
      printf("Queue is empty");
      return NULL;
   }
   temp = front[i];
   item = front[i]->data;
   front[i] = front[i]->next;
   if(front[i] = NULL)
      rear[i] = NULL;
   free(temp);
   return item;
}
```

The important thing to notice about the the implementation of multiple queues and stacks is that both the implementation are computationally inexpensive. Unlike the implementation of multiple stacks and queues in a an array, here there is no need to shift stacks and queues around to make room. One can keep on pushing new elements on to the stack and keep on adding new elements in the queue as long as free nodes are available. This simplicity and improvement of multiple stacks and queues did not come for free, we need extra space for the link fields. But this extra space is no more than a factor of 2 and this can be easily if we consider following two advantages of using linked list i.e. (1) the virtue of being able to represent complex list in a simple way, and (ii) the computing time for manipulating the list is less than for a sequential allocation.

3 Polynomial Addition

```
node *c padd(node *a, node *b )
{  node *p,*q,*c,*d,*t;
   int x; p = a; q = b;
   new(&c); d = c;
   while(p != NULL && q != NULL)
   {  if(p->exp == q->exp)
      {  x = p->coef + q->coef;
         if(x != 0)
         {  new(&t);
            t->coef = x; t->exp = p->exp;
            d->next = t; d = t;
         }
         p = p->next; q = q->next;
      }
      else if(p->exp < q->exp)
      {  new(&t);
         t->coef = q->coef; t->exp = q->exp;
         d->next = t; d = t;
         q = q->next;
      }
      else if(p->exp > q->exp)
      {  new(&t);
         t->coef = p->coef;  t->exp = p->exp;
         d->next = t; d = t;
         p = p->next;
      }
   }
   while(p != NULL)
   {  new(&t);
      t->coef = p->coef;  t->exp = p->exp;
      d->next = t; d = t;
      p = p->next
   }
   while(q != NULL)
   {  new(&t);
      t->coef = q->coef;  t->exp = q->exp;
      d->next = t; d = t;
```

```
      q = q->next
   }
   d->next = NULL;
   t = c; c = c->next; free(t);
   return c;
}
```

Finally, some comments about the computing time of this algorithm. In order to carry out a computing time analysis it is first necessary to determine which operations contribute to the cost. For this algorithm there are several cost measures:

(i) coefficient additions;

(ii) coefficient comparisons;

(iii) creation of new nodes.

Let us assume that each of these three operations, if done once, takes a single unit of time. The total time taken by algorithm (padd) is then determined by the number of times these operations are performed. This number clearly depends on how many terms are present in the polynomials A and B. Assume that A and B have m and n terms respectively.
$a(x) = a_m x^{e_m} + \ldots + a_1 x^{e_1}$ and $b(x) = b_n x^{f_n} + \ldots + b_1 x^{f_1}$
where $a_i, b_i \neq 0$ and $e_m > \ldots > e_1 \geq 0$, $f_n > \ldots > f_1 \geq 0$
Then clearly the number of coefficient additions can vary as
$0 \leq$ coefficient additions $\leq min(m, n)$

The lower bound is achieved when none of the exponents are equal, while the upper bound is achieved when the exponents of one polynomial are a subset of the exponents of the other.

Now let us see the number of times exponents are compared.We see that one comparison is made on each iteration of the while loop of lines. On each iteration either p or q or both move to the next term in their respective polynomials. Since the total number of terms is $m + n$, the number of iterations and hence the number of exponent comparisons is bounded by $m + n$. One can easily construct a case when $m + n - 1$ comparisons will be necessary.

The maximum number of terms in c is $m + n$, and so no more than $m + n$ new nodes are created. In summary then, the maximum number of executions of any of the statements in (padd) is bounded above by $m + n$. Therefore, the computing time is $O(m + n)$.

4 Circular Linked List

Consider the procedure `erase()` that frees up the nodes in a linked list `t` one by one. The method is not that efficient as all the nodes in the list needs to be visited. It is possible to free all the nodes in `t` in more efficient way by modifying the list structure in such a way that the link field of the last node points to the first node in `t` and instead of freeing the node that are no longer in use if we `dispose`(explained later) them. A list in which the `last` node points back to the `first` node is called a `circular` linked list. The structure of a circular

linked list is shown in Fig. 11. The kind of linked list we have discussed so far has a `NULL` in the last node and it is called a `chain`.

```
void erase(node *t)
{
   node *temp;
   while(temp != NULL)
   {
      temp = t;
      t = t->next;
      free(temp);
   }
}
```

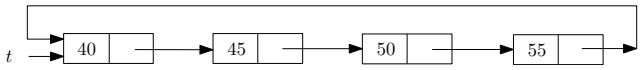


Figure 11: Circular linked list

The reason we `dispose` of nodes that are no longer in use is so that these nodes may be reused later. This objective, along with an efficient `erase()` method for circular list may be implemented by maintaining our own list of nodes that have been disposed. Whenever a procedure needs a new node this list of disposed nodes are examined first. If this list is not empty, a node from this list is made available to the procedure. Only when this list is empty, procedure `new()` is called.

Let `av` be the list of all disposed nodes. Initially `av = NULL`. With this setup, we no longer use `new` and `free` instead we use following `getnode` and `ret`.

```
node *getnode()
{
   node *temp;
   if(av == NULL)
      new(&temp);
   else
   {
      temp = av;
      av = av->next;
   }
   return temp;
}
```

```
void ret(node *temp)
{
   temp->next = av;
   av = temp;
}
```

With this procedures a circular list can now be erased in constant amount of time irrespective of the number of nodes in the list. The procedure `cerase()` does this.

```
void cerase(node *t)
{
   node *temp;
   if(t != NULL)
   {
      temp = t->next;
      t->next = av;
      av = x;
   }
}
```

The procedure `cerase` is diagrammatically shown in Fig. 12

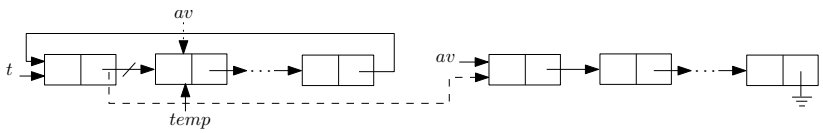


Figure 12: Erasing circular linked list

5 More on Linked List

```
node* reverse(node *t)
{  node *p, *q, *r;
   p = t; q = NULL;
   while(p != NULL)
   {  r = q; q = p;
      p = p->next; q->next = r;
   }
   t = q;
   return t;
}
```

```
node* concatenate(node *p, node * q)
{  node *z; *t;
   if(p = NULL)
      z = q;
   else
   {  if(q != NULL)
      {  t = p;
         while(t->next != NULL)
            t = t->next;
         t->next = q;
      }
   }
   return z;
}
```

Inserting at the front of circular linked list: Suppose we have a list as follows

Figure 13: circular linked list

Suppose we want to insert a new node at the front of this list. We have to change the LINK field of the node containing x_3 so that it points to the newly inserted node. This requires that we move down the entire length of a until we find the last node. It is more convenient if the name of a circular list points to the last node rather than the first as follows:

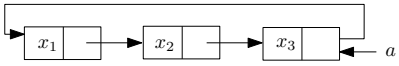


Figure 14: circular linked list with name pointing to last node

```
node* insertfront (node *a, node *t)
{
   if(a = NULL)
      a = t;
      t->next = t;
   else
   {
      t->next = a->next;
      a->next = t;
   }
   return a;
}
```

To insert x at the rear, one only needs to add the additional statement $a = x$ to the else clause.

Finding length of circular linked list:

```
int lengtht (node *a)
{  int l=0;
   node *t;
   if(a != NULL)
      t  = a;
   repeat
   {  l = l + 1;
      t = t->next;
   }until(t = a );
   return l;
}
```

Improved polynomial addition.

```
node *c CpAdd(node *a, node *b )
{  node *p,*q,*c,*d,*t;
   int x; p = a->next; q = b->next;
   new(&c); c->exp = -1; d = c;
   while(1)
   {  if(p == a && q == b)
         break;
      if(p->exp == q->exp)
      {  x = p->coef + q->coef;
         if(x != 0)
         {  new(&t);
            t->coef = x; t->exp = p->exp;
            d->next = t; d = t;
         }
         p = p->next; q = q->next;
      }
      else if(p->exp < q->exp)
      {  new(&t);
         t->coef = q->coef; t->exp = q->exp;
         d->next = t; d = t;
         q = q->next;
      }
      else if(p->exp > q->exp)
      {  new(&t);
         t->coef = p->coef;  t->exp = p->exp;
         d->next = t; d = t;
         p = p->next;
      }
   }
   d->next = c;
   return c;
}
```