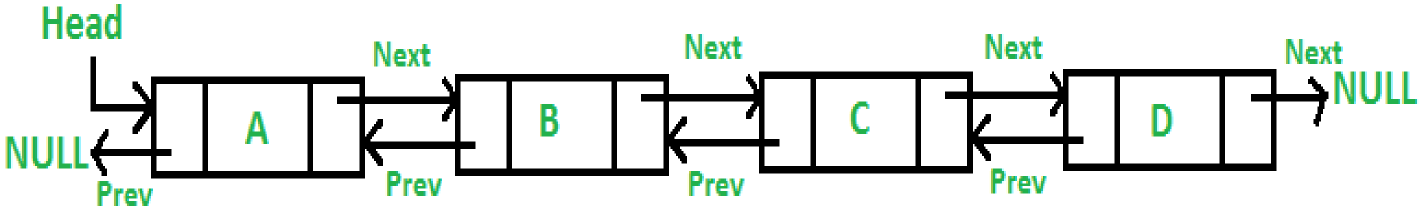
Lecture_9

Double Linked List

Doubly Linked List

If we want to add a previous operation to the list by adding a predecessor link to every node. A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.

- Double linked lists simplify the insertion and removal operations
- Doubling the links will double the amount of storage required than the single linked list



Node of a doubly linked list

```
struct Node;
typedef  Node * link;
Struct  Node
{    elemtype elem;
   link next; // Pointer to next node in DLL
   link prev; // Pointer to previous node in DLL
};
```

Doubly linked list class

```
Class double_list{ public:  double_list ( )
  { head=0; current=0;}
 void insert( ); bool first( ); bool next( );
                        bool previous( elemtype & e);
private:
struct Node;
typedef  Node * link;
Struct  Node
{    elemtype elem;
    link next; // Pointer to next node in DLL
    link prev; // Pointer to previous node in DLL
          };
Link head, current;  };
```

Function insert( ) will be changed as follows, insert at beginning of list:

insert front

```
void double_list:: insert(const elemtype &e)
{ link add= new  Node;
assert (add);                          new node --------------------> first(head)
add->elem=e;  add->next = head;
If( head) // test to see if list is not empty
head->prev= add;                       new node <---------------------first(head)
add->prev= 0;                          0<-------------new node
head=add; }                            new node (head)
```

Previous Function( )

```
bool double_list:: previous(elemtype & e)
{ assert (current);
  if( current->prev== 0) return false;
    else { current= current-> prev;
                e= current-> elem;
                return true; }}
```

Ex_1: Function insert( ) to insert a node at the end of double linked list:

```
void double_list:: insert_end(const elemtype &e)
{link add= new Node; assert (add);add->elem =e;
if (head ==0){add->next=0; add->prev=0;
head=add;}
else {current= head; while( current->next!= 0)
    current= current->next;
    current->next=add; add->next=0;
    add->prev=current;
```

very very importnat
curr->next not curr

هاااام

instead
of tail

Ex_2: For the double linked list, add the member function insert_after(), to insert a node after the **<u>nth node</u>**

```
void double_list:: insert_after(elemtype &e, int n)
link add= new node;
assert (add);   add->elem=e;
if (head ==0){add->next=0; add->prev=0; head=add;}
else{ current= head; for(int i=1;i<n; i++) current= current->next;
add->prev=current; add->next=current->next;
if(current->next!=0)
current->next->prev=add;
current->next=add;}   }
```

هاااه

node <-> added <-> node/NULL

cases :
1- empty linked list (normal insertion)
2- insert at middle
3 - insert at end   (normal insertion)

Output

insert elements in the list

50 40 30 20 10

print of list

10 20 30 40 50

enter number of nodes to insert after 1

insert element to insert 22

list after insert after node

print of list

10 22 20 30 40 50

enter number of nodes to insert after 3

insert element to insert 55

list after insert after node

print of list

10 22 20 55 30 40 50

enter number of nodes to insert after 7

insert element to insert 77

list after insert after node

print of list

10 22 20 55 30 40 50 77

Ex_3: member function append( ) to append list L2 at the end of double linked list:

```
void double_list:: append_end( double_list & L2)
{current= head; while( current->next!= 0)
      current= current->next;
      current->next=L2.head;
     L2.head->prev=current;  }
```

curr = tail

هاااام

Ex_4: Another solution of append list L2 at the end of double linked list as external function:

cant use head tail curr pointers
must use functions like
first and next

```
void  append_end( double_list & L, double_list &
L2)
{ elemtype x; bool found = L.first (x);
   while (found){ found= L.next(x);}
   found = L2. first(x);
   while( found){ L.insert(x);// use insert() at end of
list
  found= L2. next(x);}     }
In main(){ double_list L, L2;
append_end(L, L2);
```

Ex_4 Solution is the same of single linked list, as follows:

```
void linked_list:: append_end( linked_list & L2)
{current= head; while( current->next!= 0)
     current= current->next;
     current->next=L2.head;
      }
In main(){ linked_list L, L2;
L. append_end ( L2);
```

Ex: 5 member function insert_before() to insert a node befor the nth node n in double linked list:

هاالم

insert as first node

insertion in linked list (->) need 2 pointer adjustments

```
void double_list::insert_before(int n, elemtype &e)
  { link add= new node; add->elem=e;
    current=head; int i=1;
     if(head==0){add->next=0; add->prev=0; head=add;return;}
      if( n<= 1) { head->prev=add; add->prev= 0; add->next=head; head=add;
return;}
    while(i<n&& current->next!=0)
    {i++; current=current->next; }
           add->prev= current->prev;
       current->prev->next=add;
       current->prev= add;          add->next=current;          }
```

in any insertion either in begin end or middle 4 pointer must be adjusted (incase tail in found)

1- borders

1.1 head/tail

1.2 null

3. and 4. next and prevesus

2- middle

2.1 and 2.2 next and prev to nex node

2.3 and 2.4 next and prev to pre. node

insert elements in the list
50 40 30 20 10

10 20 30 40 50
enter number of nodes to insert before 1
insert element to insert 22
new list
22 10 20 30 40 50
list after insert after node
enter number of nodes to insert before 4
insert element to insert 55
new list
22 10 20 55 30 40 50
list after insert after node
enter number of nodes to insert before 7
insert element to insert 77
new list
22 10 20 55 30 40 77 50
list after insert after node

If the elements of the list:
10
20
30
40
50
<u>Output</u>
insert element to insert
33
enter number of node
1
new list after insert
print of list
33
10
20
30
40
50

هاالم

Function to search for certain element and delete its node

```
bool double_list :: remove (elemtype &e)
 {  link p; cout<<"element to search  "<<e<<endl;
 if (head->elem ==e) {current= head->next; current->prev=0;
   delete head; head=current;return true; }
 current=head;
 while (current->next->elem!= e&& current-> next!=0)
    { current=current->next;
   if(current->next==0)break; {// it is used to not ask the
loop  condition of current->next->elem
        if(current->next== 0) {cout<<" element was not found"; return false;}
 if (current->next->elem ==e)
{  p=current->next;
    current->next= p-> next; cout<<current->next<<endl;
         if(p->next!=0)p->next->prev=current;
         delete p; return true;}}
```

Sorting Algorithms

Realistic sorting problems involve files of records containing keys, small parts of the records that are used to control the sort. The objective is to rearrange the records so the keys are ordered according to some well-defined rule, usually alphanumeric, order. We will just look at the sorting of arrays of integers, corresponding to the keys in a more realistic situation, and for simplicity (so all the array elements are distinct) consider only sorting of permutations of the integers 1 … n

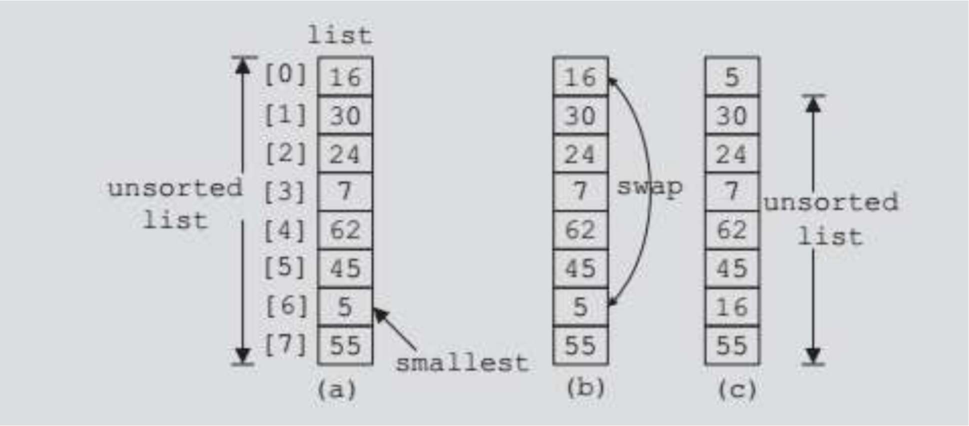
Selection Sort

In selection sort, a list is sorted by selecting elements in the list, one at a time, and moving them to their proper positions. This algorithm finds the location of the smallest element in the unsorted portion of the list and moves it to the top of the unsorted portion (that is, the whole list) of the list. The first time we locate the smallest item in the entire list, the second time we locate the smallest item in the list starting from the second element in the list, and so on. Selection sort described here is designed for array-based lists. Suppose you have the list shown in Figure . List of 8 elements

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list	16	30	24	7	62	45	5	55

Selection Sort

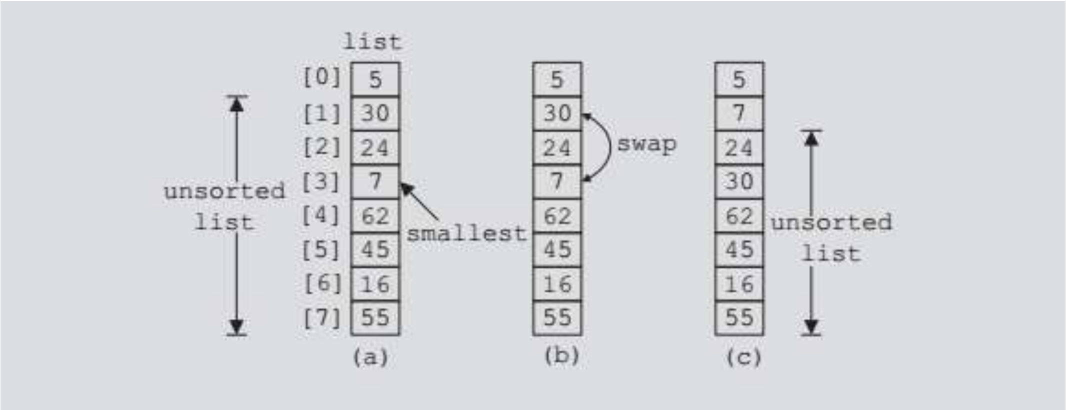
The following figure shows the elements of list in the first iteration.



Selection Sort

Initially, the entire list is unsorted. So we find the smallest item in the list. The smallest item is at position 6, as shown in figure. Because this is the smallest item, it must be moved to position 0. So we swap 16 (that is, list[0]) with 5 (that is, list[6]), as shown in figure.

After swapping these elements, the resulting list is as shown. Now the unsorted list is list[1]...list[7]. So we find the smallest element in the unsorted list. The smallest element is at position 3. Because the smallest element in the unsorted list is at position 3, it must be moved to position 1. So we swap 7 (that is, list[3]) with 30 (that is, list[1]). After swapping list[1] with list[3], the resulting list is as shown in the second iteration.



Selection Sort cont.

Now the unsorted list is list[2]...list[7]. So we repeat the preceding process of finding the (position of the) smallest element in the unsorted portion of the list and moving it to the beginning of the unsorted portion of the list. Selection sort, thus,

involves the following steps.

In the unsorted portion of the list:

1. Find the location of the smallest element.

2. Move the smallest element to the beginning of the unsorted list.

Initially, the entire list, list[0]...list[length-1], is the unsorted list. After executing

Steps 1 and 2 once, the unsorted list is list[1]...list[length-1]. After executing

Steps 1 and 2 a second time, the unsorted list is list[2]...list[length-1], and so on.

Code of Selection Sort

```
// function to get the index of the smallest element

int min_location (int list[ ], int n, int first)

 {  int minIndex=first;
  for(int i=first+1;i<n;i++)
   if ( list[i] < list[minIndex])
     minIndex = i;
     return minIndex; }
//end min_Location
```

Code of Selection Sort (cont.)

We can now complete the definition of the function selection Sort:

```
void selectionSort(int list[ ], int length)
{ int minIndex;  int first= 0;
 while( first<=length-1)
{ minIndex= min_location( list, length, first);
  swap(list, first, minIndex);
 first++; }  }
// swap function
void swap( int list[ ], int first, int second)
{ int temp;
temp = list[first];
list[first] = list[second];
list[second] = temp; }   //end swap
```

Notes on selection sort

1- In the previous code the list is sorted in ascending order.

2- Selection sort can also be implemented to sort the list in descending order, that is can be made by selecting the largest element in the (unsorted portion of the list and moving it to the beginning of the list.

3-You can easily implement this form of selection sort by altering the if statement in the function minLocation, and passing the appropriate parameters to the corresponding function and the function swap, when these functions are called in the function selectionSort.

Analysis of Selection Sort

In the case of search algorithms , our only concern was with the number of key (item) comparisons. A sorting algorithm makes key comparisons and also moves the data. Therefore, in analyzing the sorting algorithm, we look at the number of key comparisons as well as the number of data movements.

Let us look at the performance of selection sort.

Suppose that the length of the list is n. The function swap does three item assignments and is executed n - 1 times. Hence, the number of item assignments is 3(n - 1).

The key comparisons are made by the function minLocation. For a list of length k, the function minLocation makes k- 1 key comparisons. Also, the function minLocation is executed n - 1 times (by the function selectionSort). The first time, the function minLocation finds the index of the smallest key item in the entire list and so makes n – 1 comparisons.

The second time, the function minLocation finds the index of the smallest element in the sublist of length n- 1 and so makes n - 2 comparisons, and so on. Hence the number of key comparisons is as follows:

$(n-1)+(n-2)+\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 2+1= 1/2*n*(n-1) = ½*n^2-1/2*n= O(n^2)$

Thus, it follows that if n = 1000, the number of key comparisons the selection sort makes is 500000 comparisons.