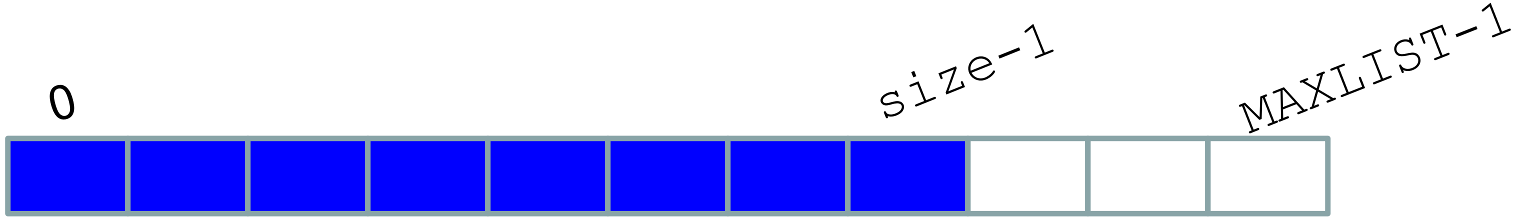
Data Structure

Lecture 7

Dr. Ahmed Fathalla

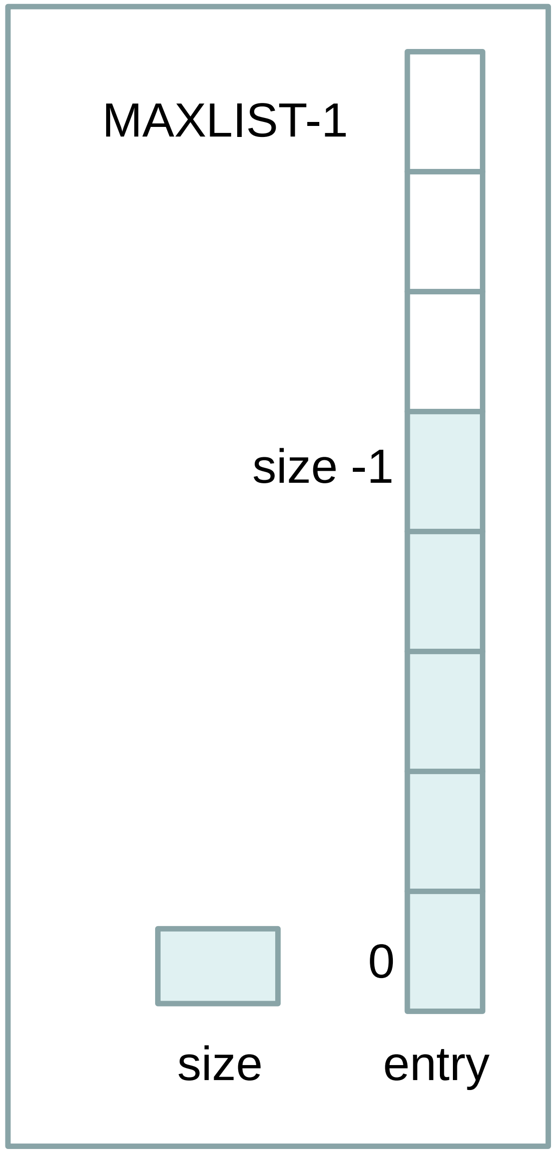
Array-Based Implementation



- This is a list (just a logical view, no implementation yet) with number of entries equals to `size`
- We can **add a new element** in position `0 ≤ p ≤ size`.
- We **delete from** `0 ≤ p ≤ size-1`.

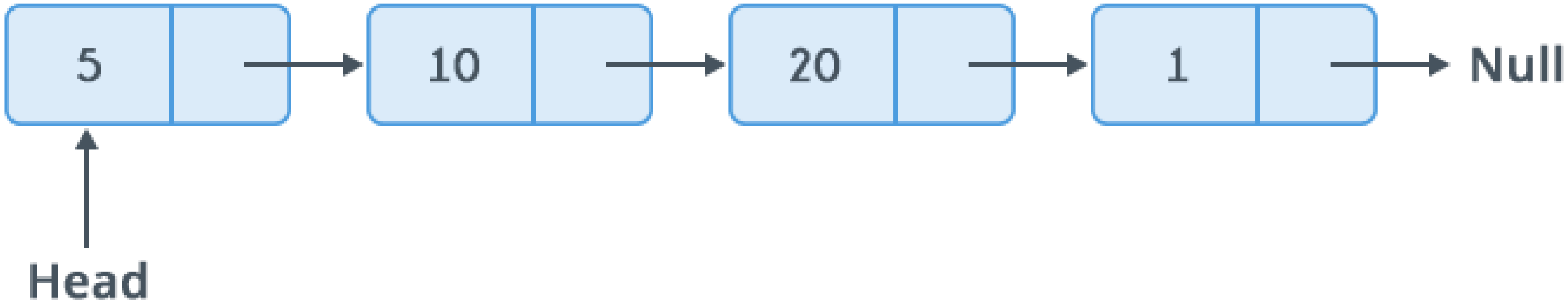
Array-Based Implementation

```
struct List{
      ListEntry entry[MAXLIST];
      int size;
};

void  CreateList  (List *);
int   ListEmpty   (List *);
int   ListFull    (List *);
int   ListSize    (List *);
void  DestroyList (List *);
void  InsertList  (int, ListEntry, List *);
void  DeleteList  (int, ListEntry *, List *);
void  TraverseList(List *);
void  RetrieveItem(int, ListEntry *, List *);
void  ReplaceItem (int, ListEntry, List *);
```

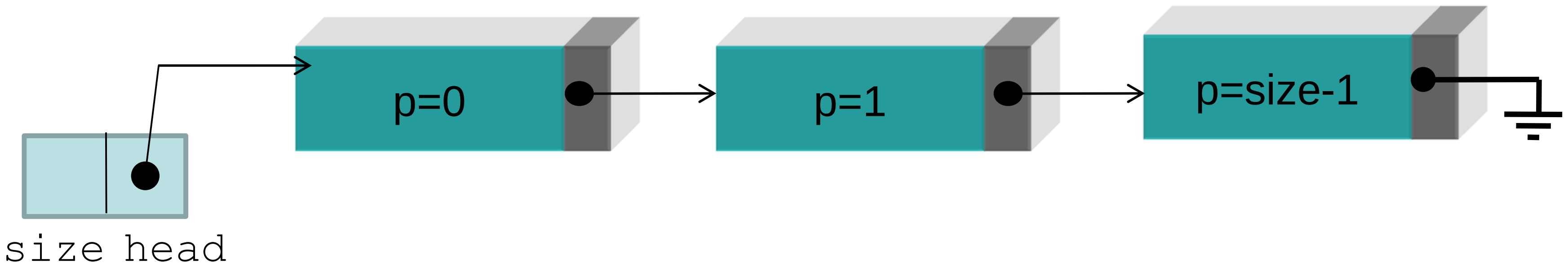


LINKED-BASED IMPLEMENTATION

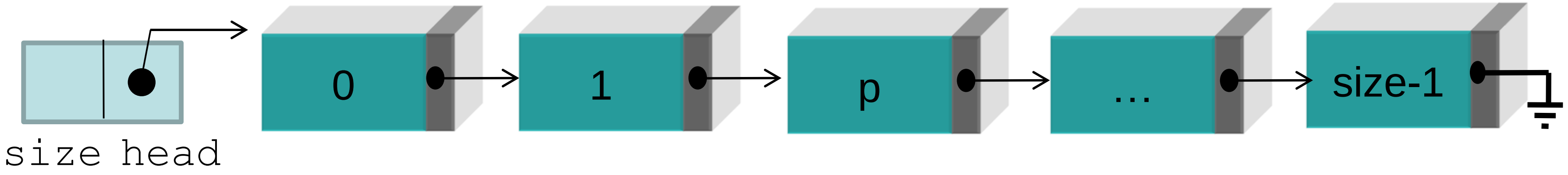
Singly Linked List



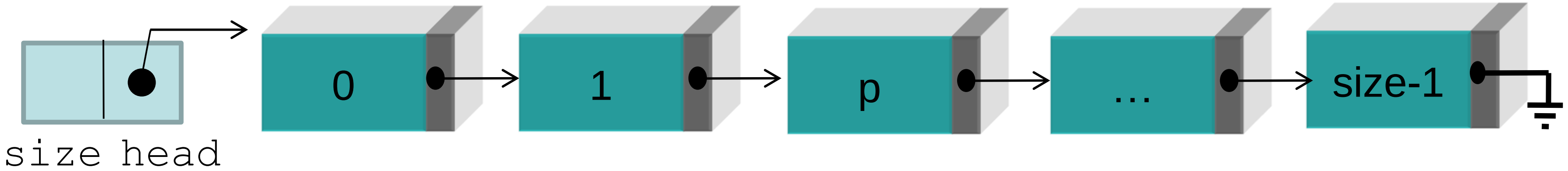
```
struct ListNode{
      ListEntry entry;
      struct Listnode *next;
};

struct List{
      ListNode   *head;
      int        size;
};
```



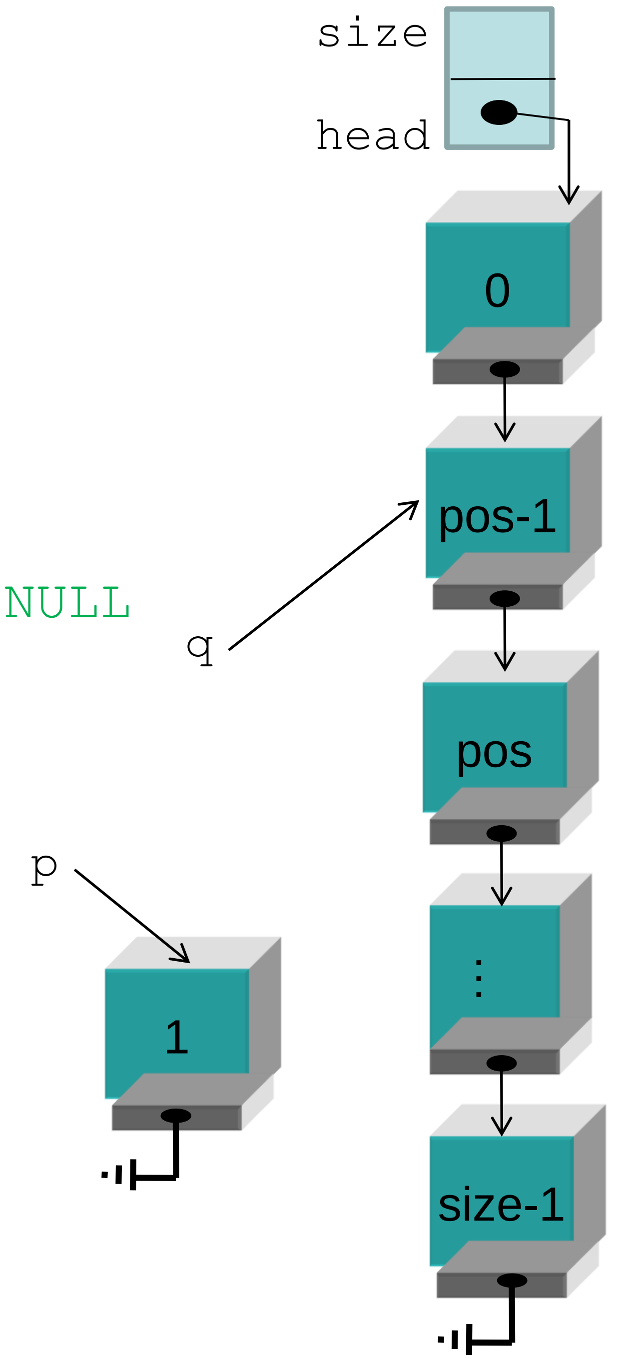
```c
void CreateList(List *pl){
      pl->head=NULL;
      pl->size=0;
} // θ(1)

int ListEmpty(List *pl){
      return (pl->size==0);
      //or return !pl->head
} // θ(1)

int ListFull(List *pl){
      return 0;
} // θ(1)

int ListSize(List *pl){
      return pl->size;
} // θ(1)
```



```
void DestroyList(List *pl){
      ListNode *q;
      while(pl->head){
            q=pl->head->next;
            delete pl->head;
            pl->head=q;
      }
      pl->size=0;
}//we took it before many times: // θ(n)

void TraverseList(List* pl){
      ListNode *p=pl->head;
      while(p){
            cout<<p->entry;
            p=p->next;
      }
} // θ(n)
```

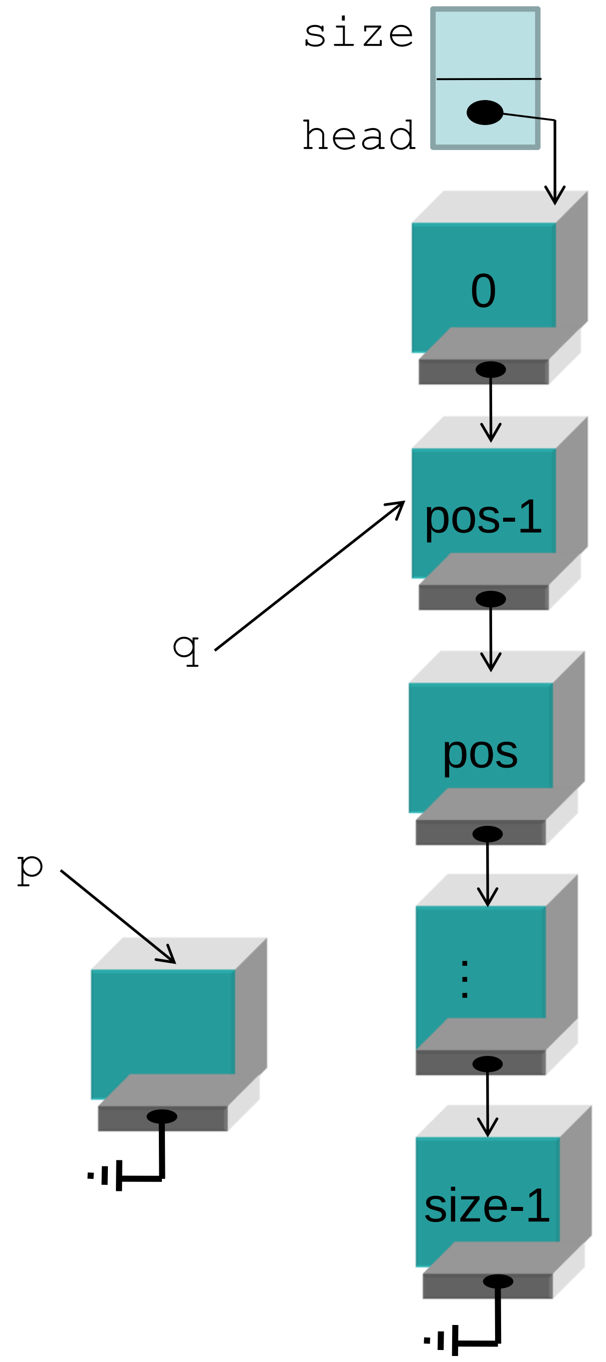


```
void InsertList(int pos, ListEntry e, List *pl)
{
    ListNode *p, *q;
    int i;
    p= new ListNode;
    p->entry=e;
    p->next=NULL;

    if (pos==0){//will work also for head equals NULL
        p->next=pl->head;
        pl->head=p;
    }
    else{
        for(q=pl->head, i=0; i<pos-1; i++)
              q=q->next;
        p->next=q->next;
        q->next=p;
    }
    pl->size++;
} // θ(n) but without shifting elements.
```

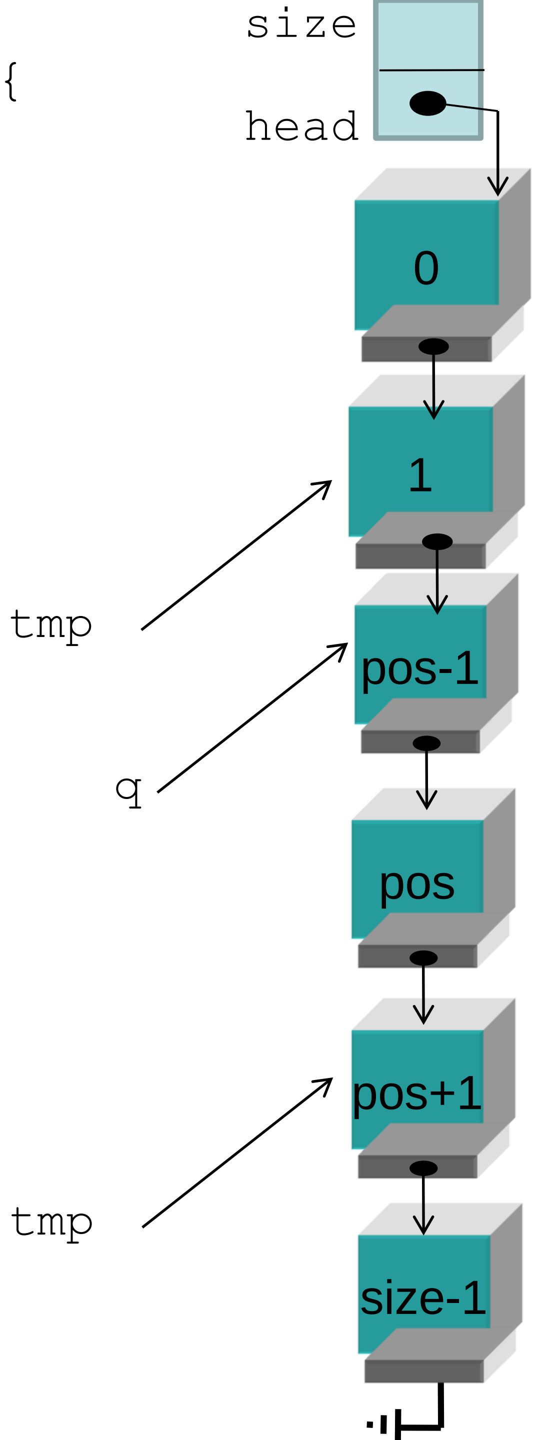


- We should make sure that the memory is not full when we call `new` (as we did previously).
- We have to design the function to return `int` not void and we modify the contiguous implementation accordingly to `return 1` always.

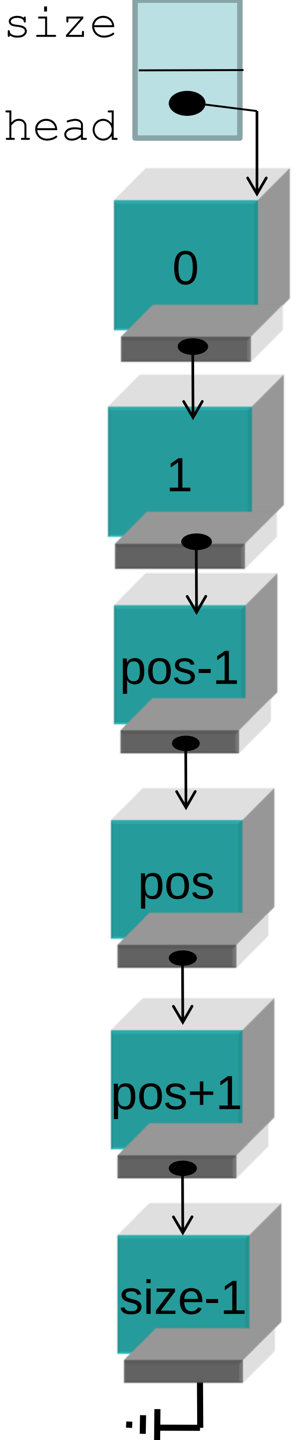
```
int InsertList(int pos, ListEntry e, List *pl)
{
   ListNode *p, *q;
   int i;
   if (p = new ListNode){
       p->entry=e;
       p->next=NULL;

       if (pos==0){//works also for head = NULL
          p->next=pl->head;
          pl->head=p;
       }
       else{
          for(q=pl->head, i=0; i<pos-1; i++)
              q=q->next;
          p->next=q->next;
         q->next=p;
      }
      pl->size++;
      return 1;
   }
   else return 0;
}
```



```
void DeleteList(int pos, ListEntry *pe, List *pl){
   int i;
   ListNode *q, *tmp;

   if (pos==0){
      *pe=pl->head->entry;
      tmp=pl->head->next;
      delete pl->head;
      pl->head=tmp;
   }// it works also for one node
   else
   {
      for(q=pl->head, i=0; i<pos-1; i++)
            q=q->next;
      *pe=q->next->entry;
      tmp=q->next->next;
      delete q->next;
      q->next=tmp;
   }// check for pos=size-1 (tmp will be NULL)
   pl->size--;
} //O(n) but without shifting elements.
```



```
void RetrieveItem(int pos, ListEntry *pe, List *pl){
      int i;
      ListNode *q;
      for(q=pl->head, i=0; i<pos; i++)
            q=q->next;
      *pe=q->entry;
}


void ReplaceItem(int pos, ListEntry e, List *pl){
      int i;
      ListNode *q;
      for(q=pl->head, i=0; i<pos; i++)
            q=q->next;
      q->entry=e;
}
```



Comparison between the array-based and the linked implementation: “Which is **always** better?” is a wrong question!

	Array-based	**Linked**
`CreateList`	θ(1)	θ(1)
`InsertList`	θ(n)	θ(n)
`DeleteList`	θ(n)	θ(n)
`RetrieveList`	θ(1)	θ(n)
`ReplaceItem`	θ(1)	θ(n)

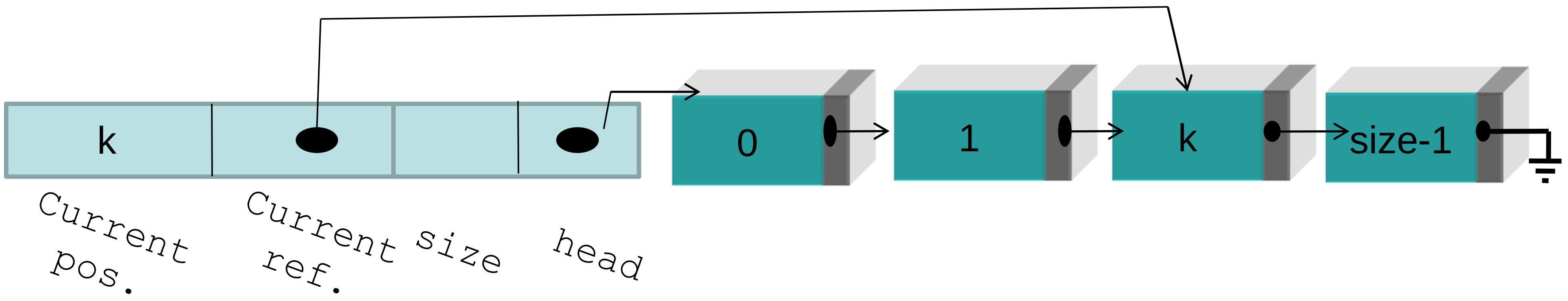
InsertList is very time consuming for Array-based because of copying elements, especially if the elements are large records.

RetrieveList and ReplaceList are always better for contiguous implementation.

Read the book very well.

Design Enhancement_1: Learn how you modify your design to enhance the performance

- Many applications process the entries in order, i.e., moving from one entry to the next.
- Many other applications refer to the same entry many times.
- Then, our current linked implementation is very inefficient, since it moves from the head to the element every time!
- Then, we need to *remember* the last position `currentpos` and start navigating from it, and we use `currentref` to start walking from `currentpos`.

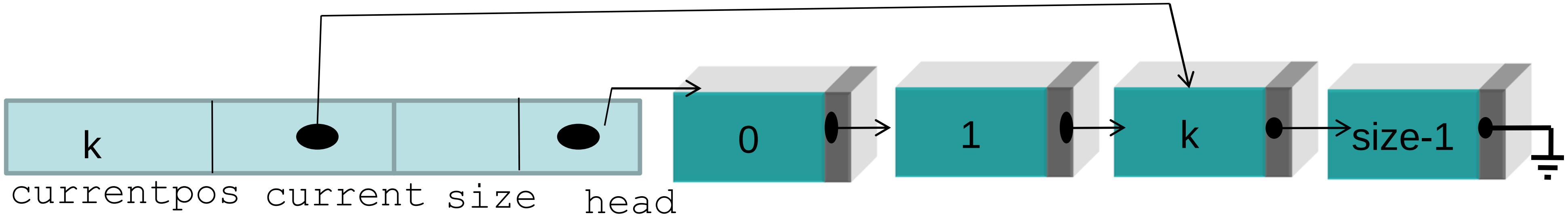


Design Enhancement_1: Learn how you modify your design to enhance the performance

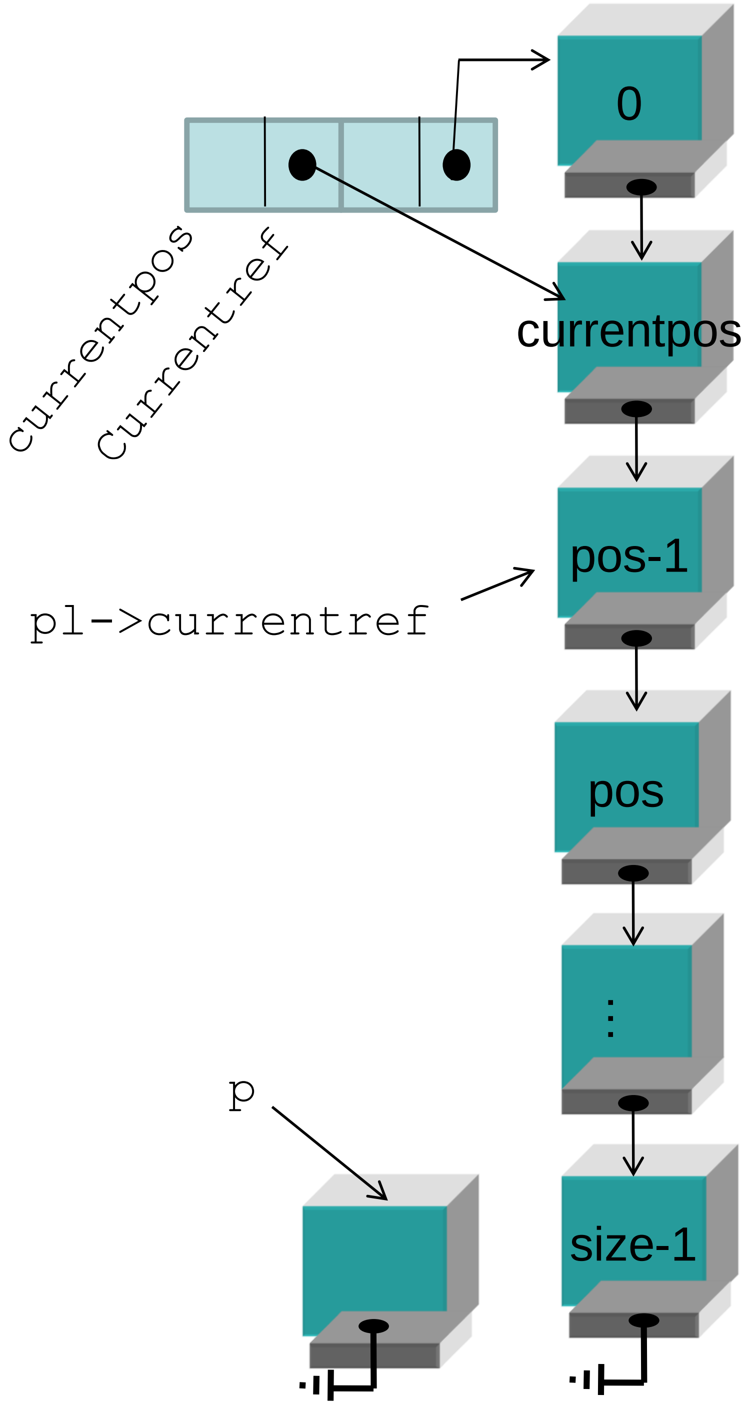
Of course, this **will NOT** help if the new element is preceding the last element visited.

Only the operations/interfaces, `InsertList`, `DeleteList`, `ReplaceItem`, and `RetrieveItem` will be changed.

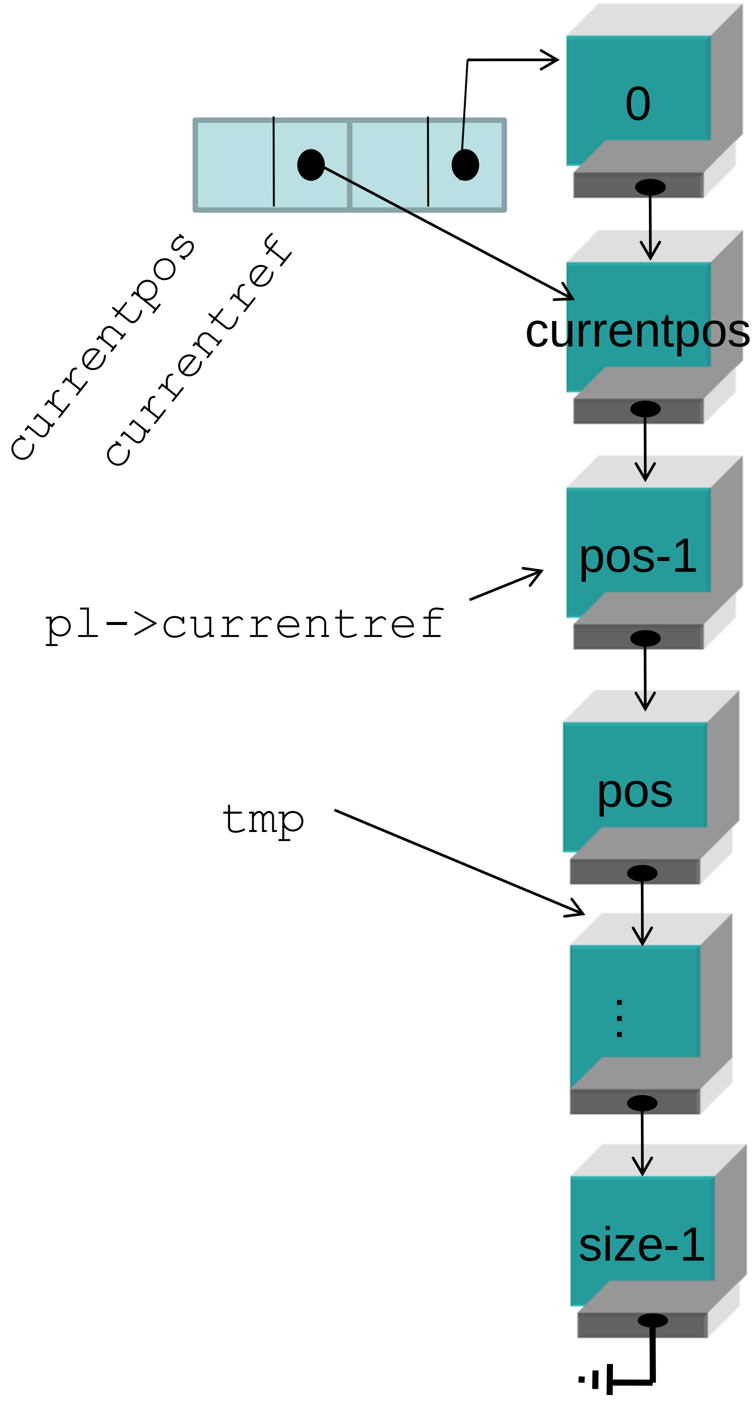
```
struct List{
        ListNode *head, *currentref;
        int        size, currentpos;
};
```



```
void InsertItem(int pos, ListEntry e, List *pl){
   ListNode *p;
   p=new ListNode;
   p->entry=e;
   p->next=NULL;
   if (pos==0){
       p->next=pl->head;
       pl->head=p;
       pl->currentpos=0; //new
        pl->currentref=pl->head; //new
   }
   else{//pl->currentref is used in place of q previously.
       if (pos<=pl->currentpos){
              pl->currentpos=0;//as i=0
              pl->currentref=pl->head;//as q=pl->head
       } //new.
       for(; pl->currentpos!=pos-1; pl->currentpos++)
              pl->currentref=pl->currentref->next;
       p->next=pl->currentref->next;
       pl->currentref->next=p;
   }
   pl->size++;
}
```



```
void DeleteList(int pos, ListEntry *pe, List *pl){
   ListNode *tmp;
   if (pos==0){
       *pe=pl->head->entry;
       pl->current=pl->head->next;
       free(pl->head);
       pl->head=pl->current; //new
       pl->currentpos=0; //new
   }
   else{
       if (pos<=pl->currentpos){
              pl->currentpos=0;
              pl->current=pl->head;
       }
       for(; pl->currentpos!=pos-1; pl->currentpos++)
              pl->current=pl->current->next;
       *pe=pl->current->next->entry;
       tmp=pl->current->next->next;
       free(pl->current->next);
       pl->current->next=tmp;
   }
       pl->size--;
}
```



You have to modify `ReplaceItem` and `RetrieveItem` in the same manner. (Do it as a homework). Check also for other functions, e.g., `CreateList` (for initialization, homework)

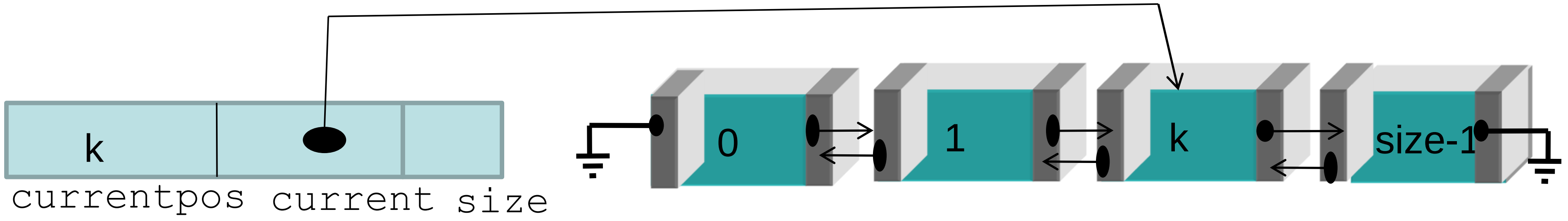
Having this function may simplify the code for the case of having `currentref` and `currentpos`.

Design Enhancement_2: Learn how you modify your design to enhance the performance

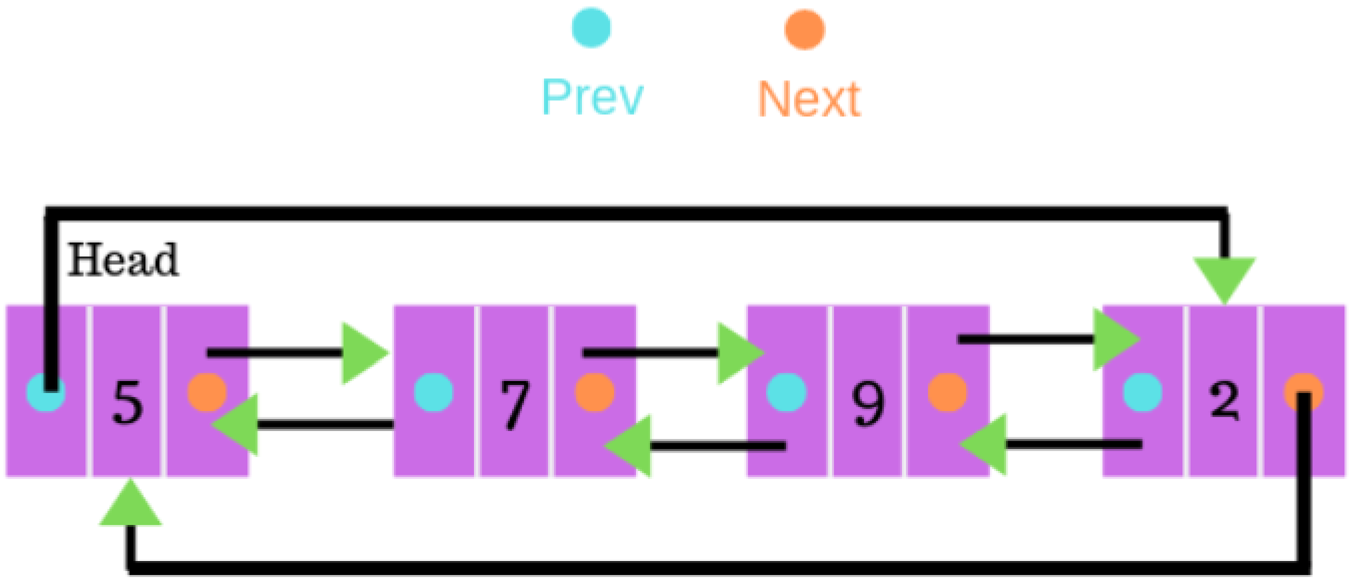
Accessing the list at a position preceding `currentpos` will be slow, since we cannot move back. A possible remedy is using **doubly linked list**.

We need just a pointer, `currentref`, not necessarily point to the first node. `currentpos` will always indicates the order of the current node.

Task: implement the **doubly linked list** and the required operation/interface modification(s) that will be affected according to this new modification.



Design Enhancement_3: Learn how you modify your design to enhance the performance



Circular Doubly linked list

Summary of List data structure

- Array-based
- Linked-base
  - Singly-linked list
  - Singly-linked list. using current- (pos & ref)
  - Doubly-linked list. using current- (pos & ref)
  - Circular Doubly-linked list. using current- (pos & ref)