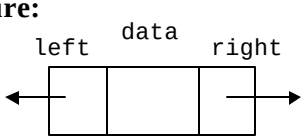
Doubly Linked List

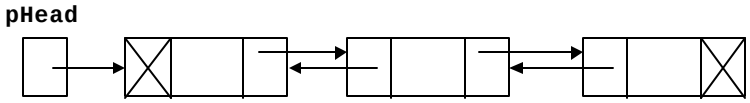
- Simple linked lists only allow making search from the beginning to end.
- Doubly linked lists allow searches in both directions (while keeping a single pointer)

Each node contains two pointers, one to the next node, one to the preceding node.

```
struct dllNode {
     int data;
     struct dllNode *left;
     struct dllNode *right;
}
```

Node Structure:

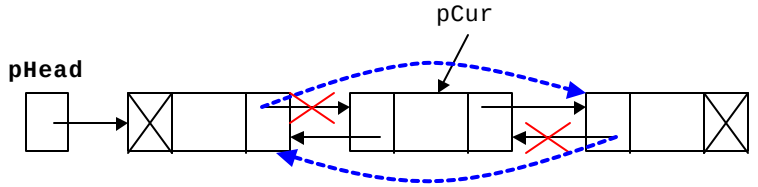




Advantage:

- insertion and deletion can be easily done with a single pointer.

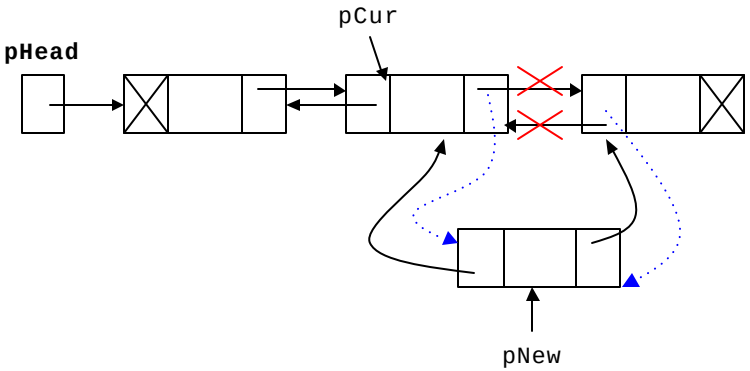
Deletion




```
pCur->left->right = pCur->right;
pCur->right->left = pCur->left;
```

(Assuming `pCur->left` and `pCur->right` are not NULL)

Insertion




```
pNew->left = pCur;
pNew->right = pCur->right;
pCur->right->left = pNew;
pCur->right = pNew;
```

Disadvantage of Doubly Linked Lists:

- extra space for extra link fields
- maintaining extra link during insertion and deletion

Array Implementation of Linked Lists

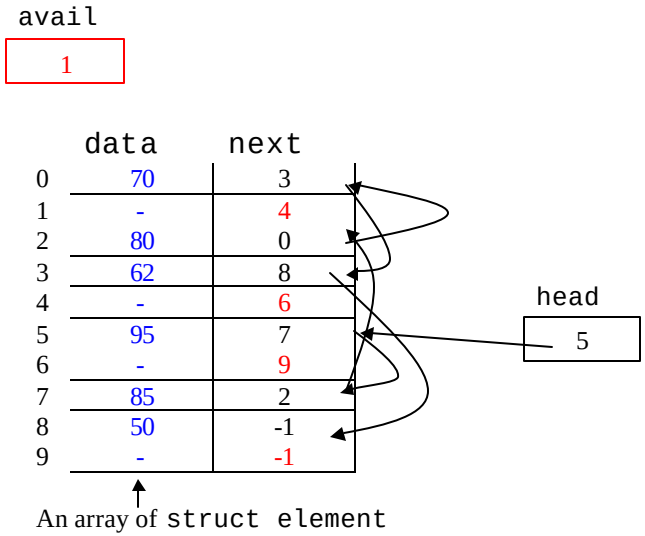
The idea is to begin with a large array and regard the array as our allocation of unused space. We then set up our own procedures to keep track of which parts of the array are unused and to link entries of the array together in the desired order.

Dynamic memory:

- By using an array we'll lose the flexibility of dynamic allocation of storage. But all the remaining advantages of linked lists such as insertions and deletions anywhere in the list will still apply.
- We can easily rearrange items without moving them.

Example

```
struct element{
   int data;
   int next;
};
struct element grades[10];
```



Sorted Linked List: in ascending order of grades

Add 65 at position 1.

Implementation:

```
#define NIL  -1

/* array nodes */
struct element grades[10];

int head;    // keeps head of list
int avail;   // gives the index of the first
             // available element in array.

/* Set up a list of available space and write
 * functions to obtain a new node and return a
 *  node to available space.
 */

/* Initially */
avail = 0;
head = NIL;

/* Function to obtain a new node location.
 * Returns the index of the next available entry
 * if there is one
 */
int newNode(struct element List[], int * avail)
{
   int loc;
   if (*avail == -1)
      return -1;
   else{
      loc = *avail;
      *avail = List[*avail].next;
      return loc;
   }
}
```

```
int releaseNode(struct element List[], int loc,
                int *avail)
{
    List[loc].next = *avail;
    *avail = loc;
}


void traverse(struct element List[], int head)
{
   int c;

   c = head;
   while (c != NIL){
      printf("%d\n", List[c].data);
      c = List[c].next;
   }

}
```

```
// Given the head, the predecessor (pre) and the
// data to be inserted (item), we must allocate
// space in the array for the new node (new)
// and adjust the indices.


void insertNode(struct element List[], int *head,
                int pre, int item, int* avail)
{
   int new;

   new = newNode(List, avail);
   if (new == -1)
     printf("No more space, item cannot be inserted\n");
   else {
      List[new].data = item;
      if (pre == NIL){
         List[new].next = pre;
         *head = new;
      }
      else {
         List[new].next = List[pre].next;
         List[pre].next = new;
      }
    }
}
```