Q6.

```
void dynArrayAddAt (struct DynArr * da, int index, TYPE val) {
        assert(da!=0);
        assert(index >= da->size); //because it does not overwrite a value.
          if(index == da->capacity)
          {
                int tempSize = da->size;
                int I;
                struct DynArr* newArray = (struct DynArr*)malloc(sizeof(struct DynArrr)*2*da-
>capacity));
                for(i=0; i < da->size; i++)
                 newArray[i] = da->data[i];
                free(da->data);
                da->data = newArray;
          }
          da->data[index] = val;
          da->size = tempSize;
          da->size++;
 }
```

Q7.

1. [ ][ ] size = 0, capacity = 2

2. [5][ ] size = 1, capacity = 2

3. [ ][ ] size = 0, capacity = 2

4. [4][ ] size = 1, capacity = 2

5. [4][5] size = 2, capacity = 2

6. [4][5][9][ ] size = 3, capacity = 4

7. [4][5][ ][ ] size = 2, capacity = 4

8. [4][5][3][ ] size = 3, capacity = 4

9. [4][5][3][9] size = 4, capacity = 4

10. [4][5][3][9][11][ ][ ][ ] size = 5, capacity = 8

Q8.

The worst-computational complexity for removing from the back of a deque using a linked list with only next pointers is O(n). Assuming we only have the next pointer towards the back, then we have to traverse all the way to back of the list to remove the link from the back. So the worst-computational complexity should be O(n).

Q9.

	Dynamic Array Bag	Linked List Bag	Ordered Array Bag
Add	O(1)	O(1)	O(n)
Remove	O(n)	O(n)	O(n)
Contain	O(n)	O(n)	O(logn)

Q10.

```
void reverseAndPrint(linkedList *lst)

{

        int i;

        struct DLink *current = lst->backSentinel;

        do

        {

                struct DLink *tmp = lst->current->prev;

                lst->current->prev = lst->current->next;

                lst->current->next = tmp;

                current = current->next;

        }while(current->next != NULL);

        /*switching the sentinels before printing the list from front to back*/

        struct DLink *tmpSentinel = lst->frontSentinel;

        lst->frontSentinel = lst->backSentinel;

        lst->backSentinel = lst->tmpSentinel;

        current = lst->frontSentinel->next;

        /*printing the list from the front to back; the result will be reversed*/

        while(current != lst->backSentinel)

        {

                pritntf("%d ", current->value);

                current = current->next;

        }

}
```

Q11.

The answer is **15.**

In this picture, the current pointer is pointing 20 now.

so after first hasNext(itr) the current will be pointing to 15 because the implementation of iterator increments the current pointer during hasNext. and the next hasNext(itr) will make the current pointer to point to 10. and the first pintf function will print out 10. and first remove function will remove 10 but will make the current pointer point to the element whose value is 15. So the result will be **15.**

Q12.

```
int cirListDequeIteratorHasNext (struct cirListDequeIterator*itr) {

        if(itr>currentLink->next != lit->lst->Sentinel)

        {        itr->currentLink = itr->currentLink->next;

                 return 1;

        }

        else{

        return 0; //there is nothing next. there is only a sentinel, which means there is nothing next.

        }

}


TYPE cirListDequeIteratorNext (struct cirListDequeIterator *itr)

{

        return  itr->currentLink->value;  //we  already  incremented  the  currentLink  pointer  in
cirListDequeIteratorHasNext()

}
```

Q13.

To perform binary search,

1. the collection should be implemented in array.
2. the collection should be already sorted.