Data Structure and Algorithm
Homework #2

=== Homework Reference Solution ===

***Problem* 1.** Transfer a letter

We can use the following structure to represent our list.

```
struct Node{
    char word[50];
    struct Node *next;
};
```

Then execute the following steps to implement the list.

Step 1: Use

```
while (scanf("%c", &get)==1){
    scanf("%c", &get)
    tmp[pos++] = get;          // tmp is a char array to store a string
}
```

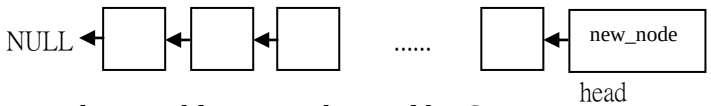
to read the input characters until we encounter a character which is not an alphabet. This method can automatically ignore any character which is not an alphabet.

Step 2: Use

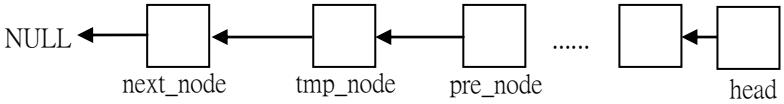
```
struct Node* list=head;  //head is the pointer which points to the head of the list
int pos = 0;
while (list != NULL && strcmp(list->word, tmp) != 0) {
    list = list->next;
    ++pos;
}
```

to determine whether the obtained word (stored in the *tmp* array) has already existed in the list or not. If at the end *list* points to NULL, it means that the obtained word is a new word and we should go to Step 3 to insert it into the list. Otherwise, the word has been inserted to the list before (and the pointer *list* is pointing at it) and we should go to Step 4.

Step 3: To insert the new word into the head of the list, we have to construct a *new_node* and *new_node->next* must point to the current head of the list. Then we store the value of the word to *new_node* and make *new_node* the head of the list.
Hence, the list will be as follows.



Step 4: The variable "*pos*" obtained by Step 2 is the position of the word in the list.
We use *target_node* to store the given word, *pre_node* to be the previous node of *target_node* and *next_node* to be the node pointed by *target_node->next*. The next step is to move *target_node* to the head in the list.
First, use a *tmp_node* to record the content of *target_node*, and let *pre_node->next* point to the *next_node*. Finally, let *tmp_node->next* point to current head of the list and *tmp_node* will become the head of the list.



Step 5: Repeat the Step 1 to Step 4 until the file is completely processed.

***Problem* 2.** Back to stacks & queues

2.1

Initialization:

```
Node* head = (Node*)malloc(sizeof(Node));
head = NULL;

void push(Node **head, SomeType data){
	Node* temp = (Node*)malloc(sizeof(Node));
	temp->data = data;
	temp->next = *head;
	*head = temp;
}

SomeType pop(Node **head){
	SomeType data;
	Node* temp;

	if((*head) == NULL){
		printf("Error. Stack is empty.\n");
		return 0;
	}
	data = (*head)->data;
	temp = *head;
	*head = (*head)->next;
	free(temp);
	return data;
}
```

2.2

Initialization:

```
Node* head = (Node*)malloc(sizeof(Node));
Node* tail = (Node*)malloc(sizeof(Node));
head = NULL;
tail = NULL;

void enqueue(Node **head, SomeType data,Node **tail){
	Node* temp = (Node*)malloc(sizeof(Node));
	temp->data = data;
	temp->next = NULL;
	if((*head) == NULL){
		(*head) = temp;
		(*tail) = temp;
	}
	else{
		(*tail)->next = temp;
		*tail = temp;
	}
```

```
}

SomeType dequeue(Node **head, Node **tail){
	SomeType data;
	Node* temp;
	if((*head) == NULL){
		printf("Error. Queue is empty.\n");
		return 0;
	}
	data = (*head)->data;
	temp = *head;
	*head = (*head)->next;
	if((*head) == NULL) *tail = NULL;
		free(temp);
		return data;
}
```

***Problem* 3.** Double-ended queues

3.1

```
void pushBack(Node **head, SomeType data){
      Node *newTail = (Node*) malloc(sizeof(Node));
      newTail -> data = data;
    if(*head == NULL){
      *head = newTail;
      (*head) -> next = *head;
      (*head) -> prev = *head;
    }
    else{
      newTail -> prev = (*head) -> prev;
      newTail -> next = *head;
      (*head) -> prev -> next = newTail;
      (*head) -> prev = newTail;
    }
}
void pushFront(Node **head, SomeType data){
      pushBack(head,data);
      *head = (*head) -> prev;
}
SomeType popBack(Node **head){
      SomeType returnData;
      if( (*head)->prev == *head ){
      returnData = (*head) -> data;
      free(*head);
      *head = NULL;
    }
    else{
      Node *newTail = (*head)->prev->prev;
      returnData = (*head) -> prev -> data;
      newTail -> next = *head;
      free((*head)->prev);
      (*head) -> prev = newTail;
    }
    return returnData;
}
SomeType popFront(Node **head){
      SomeType returnData;
      *head = (*head)->next;
      returnData = popBack(head);
      return returnData;
}
```

3.2

(a) 1,5,4,3,2

pushFront(1)
popFront() -> got 1
pushFront(2)
pushFront(3)
pushFront(4)
pushFront(5)
popFront() -> got 5
popFront() -> got 4
popFront() -> got 3
popFront() -> got 2

(b) 5,3,2,4,1

pushFront(1)
pushFront(2)
pushFront(3)
pushback(4)
pushback(5)
popBack() -> got 5
popFront() -> got 3
popFront() -> got 2
popBack() -> got 4
popFront() -> got 1

(c) 1,4,2,3,5

pushFront(1)
popFront() -> got 1
pushFront(2)
pushFront(3)
pushFront (4)
popFront() -> got 4
popBack() -> got 2
popFront() -> got 3
pushFront(5)
popFront() -> got 5

***Problem* 4.** More operations on lists

4.1

```
Node* find_n_kth(Node* head, int k){
        Node* ptrN_K, *ptr;
        int count=0;

    if(k<=0){
      return NULL;
    }

    For (ptr = head; ptr!= NULL; ptr = ptr -> next){
      if(count%k == 0){
        ptrN_K = ptr;
      }
        count++;
    }
```

```
    return ptrN_K;
}
```

4.2

```
function(x, y)
      z = new Node
      z->next = NULL
      tmp = z
      while x!=NULL and y!=NULL
            tmp->next = x
            tmp = tmp->next
            x = x->next
            tmp->next = y
            y = y->next
            tmp = tmp->next
      if x!=NULL
            tmp->next = x
      else
            tmp->next = y
      tmp = z->next
      delete z
      return tmp
```

Time Complexity: The while loop takes O(min(n,m)) time as it will run for min(n,m) times. The other steps run in O(1). Therefore the total time complexity is O(min(n,m)).

4.3

```
function(x, y)
      z = new Node
      z->next = NULL
      tmp = z
      while x!=NULL and y!=NULL
            if x->data<=y->data
                        tmp->next = x
                        tmp = tmp->next
                        x = x->next
            else
                        tmp->next = y
                  tmp = tmp->next
                  y = y->next
      if x!=NULL
            tmp->next = x
      else
            tmp->next = y
      tmp = z->next
      delete z
      return tmp
```

Time Complexity: O(m+n), as the algorithm goes through all nodes in x and y for exactly once.

***Problem* 5.** Skip lists

5.1

A skip list is a data structure for storing a sorted list of items using a hierarchy of linked lists that connects increasingly sparse subsequences of the items.

A skip list is built in layers. The bottom layer is an ordinary ordered linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer i appears in layer (i+1) with some fixed probability p.

On average, each element appears in 1/(1-p) lists.

Hence, the space complexity of this representation is O(n*(1/(1-p))) = O(n) due to the fact that 1/(1-p) is a constant.

Reference: http://en.wikipedia.org/wiki/Skip_list

5.2

For inserting a given value into the skip list, we should first search the appropriate position (after inserting the list in every level must still be sorted) to insert.

After searching, we insert the given value into the bottom level, and use the probability p to determine whether this node have to grow to the higher level or not. The maximal number of level we have to insert is $\log_{1-p} n$.

According to problem 5.3, the time complexity of searching is O(log n), hence the total time complexity of insertion is O(log n) + O(log n) = O(log n).

5.3

For searching a given value in the skip list,

1) start from the list in top level of the skip list.

2) If the value of the node in current list is smaller than the given value, scan to the next node in current list.

If the value of the node in current list is larger than the given value or the next node we want to scan is NULL, drop one level.

3) Repeat step 2) until the value of the node in the current list equals to the given value.

4) If we drop down to the bottom level, and the next node in current list is NULL, we could say that the given value does not exist in our skip list.

Time complexity: O(log(n)). We can treat the searching opeartion in the skip list as the searching operation in an ordinary binary search. The details of proof can be found in page 4 of the paper, William Pugh, "Skip lists: A probabilistic alternative to balanced trees," ACM Commun., June, 1990.