CME 112- Programming Languages II
Lecture 5: Linked Lists (Part-1)

Assist. Prof. Dr. Caner ÖZCAN

Understanding is the beginning of love.

B. Spinoza
LINKED LISTS

Linked lists are useful to study for some reasons.

- Most obviously, linked lists are a data structure for real programming.
- Knowing the strengths and weaknesses of linked lists will help you thinking about complexity of processing time and memory space of algorithms.
- Linked lists are a good way of understanding the pointers.
- Linked list problems are a nice combination of algorithms and pointer manipulation.
LINKED LISTS

A linked list is a data structure that can be changed during execution.

- Consecutive elements are connected with the pointer.
- Last element points to NULL.
- Size can grow or shrink during the execution of the program (It can be made just as long as required)
- It doesn’t made waste memory
ARRAYS vs. LINKED LISTS

- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the array for a particular value.

- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements can not be predicted beforehand.
TYPES of LISTS

- Depending on the construction of connection style, several different types of linked lists are possible.
  - 1. Linear singly linked list (Linear List)
Liste Tipleri

2. Circular linked list
   - The pointer of the last element in the list points back to the first element.
3. Doubly linked list

- The connection is established between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
LINKED LISTS

- List is an abstract data type
- This data type is defined by the user.
- Typically more complex than simple data types like int, float, etc.
- Main aim is:
Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item to the list
- Deleting an item from the list
- Concatenating two lists into one
Working with Linked Lists

Consider the structure of a node in the list as follows:

```c
#include <stdio.h>
#include <stdlib.h>

struct student{
    int no;
    char name[40];
    int age;
    struct student *next;
};
typedef struct student node;
node *head,*newNode;
```
Creating a Linear List

- First, a node must be created and the head must be provided to point that node.

```
head = (node *) malloc(sizeof(node));
```
Creating a Linear List

- If number of nodes is n in the initial linked list:
  - Allocate n records, one by one.
  - Read in the fields of the records.
  - Modify the links of the records so that the chain is formed.
Creating a Linear List

```c
node* createList(){
    int n,k;
    node *head,*p;
    printf("How many students in the list? ");
    scanf("%d",&n);
    for(k=0;k<n;k++){
        if(k==0){
            head = (node *)malloc(sizeof(node));
            p = head;
        }
        else{
            p->next = (node *) malloc(sizeof(node));
            p = p->next;
        }
        printf("Enter %d. student number: ",k+1); scanf("%d",&p->no);
        printf("Enter %d. student name: ",k+1); scanf("%s",p->name);
        printf("Enter %d. student age: ",k+1); scanf("%d",&p->age);
    }
    p->next = NULL;
    return head;
}
```
Traversing the List

- Once the linked list has been constructed and head points to the first node of the list:
  - Follow the pointers.
  - Display the contents of the nodes as they are traversed.
  - Stop when the next pointer points to NULL.
Traversing the List

```c
void traverseList(node* head) {
    int counter = 1;
    node *p;
    p = head;
    while (p != NULL) {
        printf("%d %d %s %d \n", counter, p->no, p->name, p->age);
        p = p->next;
        counter++;
    }
}
```
Inserting a Node in a List

- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link it to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- The problem is to insert a node before a specified node.
  - Specified means some value is given for the node (called key).
  - In this example, we consider it to be number 16.
Inserting a Node in a List

![Diagram showing the process of inserting a node in a list.](image)
Inserting a Node in a List

- When a node is added to the beginning,
  - Only one «next» pointer needs to be modified.
    - Head is made to point to the new node.
    - New node points to the previously first element.

- When a node is added to the end,
  - Two «next» pointers need to be modified.
    - Last node now points to the new node.
    - New node points to NULL

- When a node is added to the middle,
  - Two «next» pointers need to be modified.
    - Previous node now points to the new node.
    - New node points to the next node.
Inserting a Node in a List

```c
node* addNode(node* head){
    int stdNo;
    node *p, *q;
    node *newNode = (node *) malloc(sizeof(node));
    printf("Enter new student number: "); scanf("%d",&newNode->no);
    printf("Enter new student name: "); scanf("%s",newNode->name);
    printf("Enter new student age: "); scanf("%d",&newNode->age);

    printf("Enter std number that new record will be added before: \n");
    printf("Press 0 to add to the end of list\n");
    scanf("%d",&stdNo);

    p = head;
    if(p->no == stdNo){ //add to beginning
        newNode->next = p;
        head = newNode;
    }
```
Inserting a Node in a List

```c
p = head;
if(p->no == stdNo){ //add to beginning
    newNode->next = p;
    head = newNode;
}
else{
    while(p->next != NULL && p->no != stdNo){
        q=p;
        p= p->next;
    }
    if(p->no == stdNo){ //Add nor beginning neither end
        q->next = newNode;
        newNode->next = p;
    }
    else if(p->next == NULL){ //Add to end
        p->next = newNode;
        newNode->next = NULL;
    }
}
return head;
```
Deleting a Node from the List

For deletion:

- The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.
Deleting a Node from the List

- To delete a specified node (give the node whose number field is given)
- Three conditions arise:
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.
Deleting a Node from the List

```c
node* deleteNode(node *head){
    int stdNo;
    node *p, *q;

    printf("Enter student number that will be deleted?");
    scanf("%d",&stdNo);

    p = head;
    if(p->no == stdNo){    //delete node at the beginning
        head = p->next;
        free(p);
    }

```
Deleting a Node from the List

```c
p = head;
if(p->no == stdNo){  //delete node at the beginning
    head = p->next;
    free(p);
}
else{
    while(p->next != NULL && p->no != stdNo){
        q=p;
        p= p->next;
    }
    if(p->no == stdNo){  //Delete from nor beginning neither end
        q->next = p->next;
        free(p);
    }
    else if(p->next == NULL){  //No node found to delete
        printf("No node found to delete");
    }
}
return head;
```
Main Function

```c
int main(void)
{
    node *head;
    int selection=0;
    while(1)
    {
        printf("Selection [1-5]?\n");
        scanf("%d", &selection);
        switch (selection)
        {
            case 1: head = createList(); printf("Adres: %x\n", head);
                    traverseList(head); break;
            case 2: traverseList(head); break;
            case 3: head = addNode(head);
                    traverseList(head); break;
            case 4: head = deleteNode(head);
                    traverseList(head); break;
            case 5: exit(0);
        }
    }
}
```