Advanced Procedural Programming

Lecture 5: Linked Lists

Gemma O'Callaghan

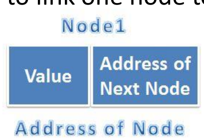
Topics Covered

- Overview
- Arrays Vs. Linked Lists
- Create a linked list
- Add a node
- Search a node
- Display all the nodes in a list
- Find the length of a list
- Delete a node

2

Overview

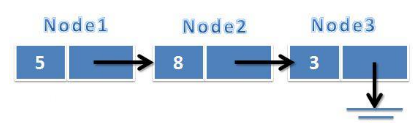
- A Linked list is a dynamic data structure whose length can be increased or decreased at run time.
- A linked list allocates space for each element separately in its own block of memory called an "element" or "node".
- Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field which is a pointer used to link one node to the next node.



3

Overview

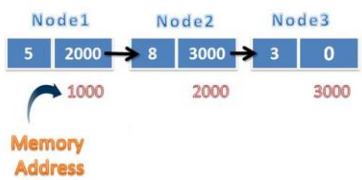
- The front of the list is a pointer to the first node.
- The list gets its overall structure by using pointers to connect all its nodes together like the links in a chain.



4

Overview

- A Linked list consists of memory blocks that are located at random memory locations. They are connected through pointers.
- The pointer stores the address of the next node.



Linked Lists Vs. Arrays

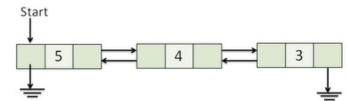
- An array is a static data structure. This means the length of an array cannot be altered at run time.
- A linked list is a dynamic data structure.
- In an array, all the elements are kept at consecutive memory locations while in a linked list the elements (or nodes) may be kept at any location but are still connected to each other.
- Linked lists are preferred mostly when you don’t know the volume of data to be stored.

Singly Linked List

- In this type of linked list, 2 successive nodes are linked together linearly.
- Each node contains the address of the next node.
- Only linear or forward sequential movement is possible.
- Elements are accessed sequentially, no direct access is allowed.
- Each node has a successor except the first node
- Each node has a predecessor except the last node which has a null reference.

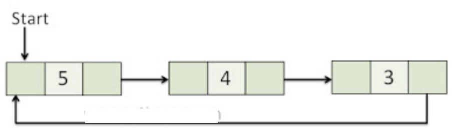
Doubly Linked List

- In a Doubly Linked List, each node contains two address fields.
- One address field for storing the address of the next node and one address field for storing the address of the previous node.
- Two way access is possible. We can access nodes from the beginning or the end but still not directly.



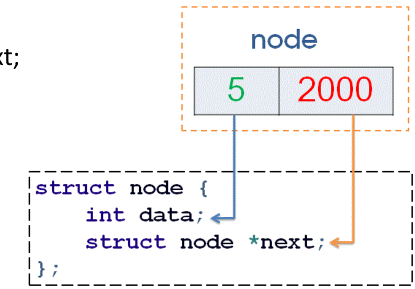
Circular Linked List

- In a Circular Linked List, the address field of the last node contains the address of the first node.
- Sequential Movement is possible but no direct access.



Node Structure

struct node
{
    int data;
    struct node *next;
}



Linked List – Creating a node

- A node is created by allocating memory to a structure in the following way:
- struct node *ptr = (struct node*)malloc(sizeof(struct node));
- Each node is allocated in the heap with a call to malloc(), so the node memory continues to exist until it is explicitly deallocated with a call to free().
- So, the pointer 'ptr' now contains the address of a newly created node.
- If the linked list is empty and the first node is created then it is also known as the head node.

Linked List – Creating a node

- Once a node is created, then it can be assigned the value (that it is created to hold) and its next pointer is assigned the address of the next node.
- If no next node exists (or if its the last node) then a NULL is assigned.
- This can be done as follows:

  ptr->data= data;

  ptr->next = NULL;

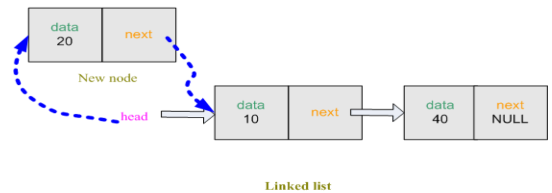
Create a Singly Linked List – Create Head node

```
/* This will be the unchanging first node */
  struct node *head;

  /* Now root points to a node struct */
  head= (struct node *) malloc( sizeof(struct node) );

  /* The node root points to has its next pointer equal to a null
pointer
    set */
  head->next = NULL;
  /* By using the -> operator, you can modify what the node,
    a pointer, (root in this case) points to. */
  head->x = 5;
```

13

Add node to start of list - Explanation

1) Allocate - Allocate the new node in the heap and set its data to whatever needs to be stored.
2) Link Next - Set the next pointer of the new node to point to the current first node of the list.
3) Link Head - Change the head pointer to point to the new node, so it is now the first node in the list.



14

Add node to start of list - Code

```
void addToStart (struct node** head)
{
    struct node *newNode;
    newNode = (struct node*)malloc(sizeof(struct node));
    printf("\nEnter data for this node");
    scanf("%d", &newNode->data);
    newNode->next = *head;
    *head = newNode;          // transfer the address of newNode' to 'head'
}
```

15

Pointer to a Pointer Needed to change Head

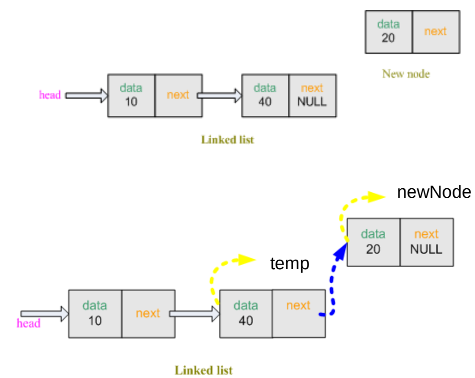
- We need addToStart () to be able to change some of the caller's memory — namely the head variable.
- The traditional method to allow a function to change its caller's memory is to pass a pointer to the caller's memory instead of a copy.
- To change a struct, pass a struct pointer* instead.
- So in this case, the value we want to change is struct node*, so we pass a struct node** instead.
- The value we want to change already has one star (*), so the parameter to change it has two (**).
- The type of the head pointer is "pointer to a struct node."
- In order to change that pointer, we need to pass a pointer to it, which will be a "pointer to a pointer to a struct node".

16

Add node to end of list - Explanation

1) Search for the last node in the heap.
2) Allocate - Allocate the new node in the heap and set its data to whatever needs to be stored.
2) Link Next - Set the next pointer of the new node to NULL as it is now the last node.
3) Link Head - Change the pointer of what was the last node to point to the new node, so it is now the last node in the list.

Add node to end of list - Explanation

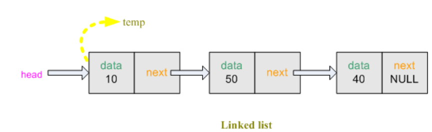


Add node to end of list - Code

```
void addToEnd(struct node* head)
{
     int data;
     struct node *temp;
     temp =(struct node*)malloc(sizeof(struct node));
     temp = head;
     while(temp->next != NULL) // go to the last node
     {
     temp = temp->next;
     }
     struct node *newNode;
     newNode = (struct node*)malloc(sizeof(struct node));
     printf("\nEnter data for this node");
     scanf("%d", &newNode->data);
     newNode->next = NULL;
     temp->next = newNode ;

}
```


Linked List - Display all the nodes in a list

- Begin at first node by creating a temporary node temp.
- Make temp point to the head of the list.
- We can get the data from first node using temp->data.
- To get data from second node, we shift *temp to the second node.
- Now we can get the data from second node.



Linked List - Display all the nodes in a list

```
struct node *temp;
temp =(struct node*)malloc(sizeof(struct node));
temp = head;

while( temp!= NULL )
{
 printf("Data: %d",  temp->data); // show the data
 temp = temp->next;
}
```


Linked List - Search a node (Sequential Search)

- Begin at first node by creating a temporary node temp.
- Make temp point to the head of the list.
- We start at the first element in list, and while the current element is not NULL (meaning we haven't reached the end), we go on to the next element.
- If the current element contains the data we're looking for, then return a pointer to it.
- If after looking at every element in the list, we haven't found the value for which we are searching, then the value doesn't exist in the list. Return a null.

Linked List - Search a node (Output in Function)

```
void searchList(struct node *head, int num)
{
        struct node *temp;
        temp =(struct node*)malloc(sizeof(struct node));       temp = head;
        while( temp!= NULL )
        {
                if(temp->data == num)
                {
                        printf("\nData Found.");
                        return;
                }
                temp = temp->next;
        }
        printf("\nData not found");
}
```


Linked List - Search a node (Return node)

```
Struct node * searchList(struct node *head, int num)
{
    struct node *temp;
    temp =(struct node*)malloc(sizeof(struct node));
    temp = head;

    while( temp!= NULL )
    {
        if(temp ->data == num)
        {
         return(temp );
        }
        temp = temp ->next;
    }
        return head;
}
```

Linked List - Find the length of a list

25

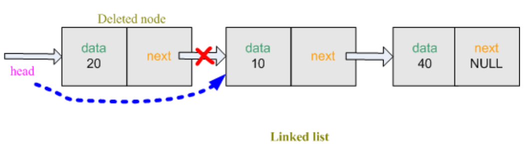
Linked List - Delete a node from start

- First, we create node *temp.
- Transfer the address of *head to *temp so *temp is pointed at the front of the linked list.
- We want to delete the first node.
- So transfer the address of temp->next to head so that it now points to the second node.
- Now free the space allocated for first node.

26

Linked List - Delete a node from start

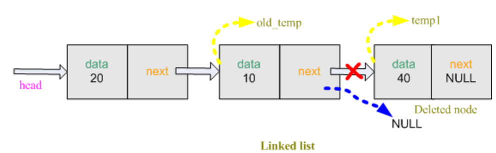
```
void deleteFromStart(struct node** head)
{
    struct node *temp;
    temp = (struct node*)malloc(sizeof(struct node));
    temp = *head;
    *head = temp->next;
    free(temp);
}
```



27

Linked List - Delete a node from end

- The last node`s next (last->next) always points to NULL.
- So when we delete the last node, the previous node of the last node is now pointed at NULL.
- So, we will track last node and previous node of the last node in the linked list.
- Create temporary node * temp1 and *old_temp.



28

Linked List - Delete a node from end

```
void deleteFromEnd(struct node* head)
{	struct node *temp1;
	temp1 = (struct node*)malloc(sizeof(struct node));
	temp1 = head;
	struct node *old_temp;
	old_temp = (struct node*)malloc(sizeof(struct node));

	while(temp1->next!=NULL)
	{
		old_temp = temp1;
		temp1 = temp1->next;
	}
	old_temp->next = NULL;
	free(temp1);
}
```

29

Linked List - Delete a node specified by user

30

Linked List - Delete a node specified by user

31