60-141-02 LECTURE 8: C DATA STRUCTURES

Edited by Dr. Mina Maleki

Outline

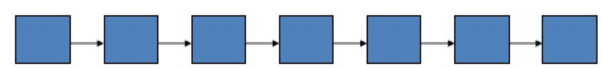
- Introduction
- Self-referential data structures
- Memory Allocation
- Concept of linked structures
- Linked lists
- Stacks and Queues
- Trees and advanced data structures

Introduction

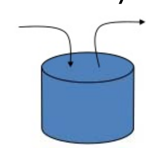
- Fixed-size data structures
  - Arrays
  - Structs
- Dynamic data structures
  - Can grow and shrink at execution time
  - Linked lists
  - Stacks
  - Queues
  - Binary trees

Dynamic Data Structures

- Linked lists
  - Collections of data items "lined up in a row"
  - Insertions and deletions are made anywhere in a linked list.

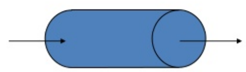


- Stacks
  - Important in compilers and operating systems
  - Insertions and deletions are made only at one end of a stack—its top

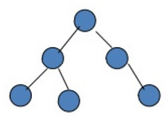


Dynamic Data Structures

- Queues
  - Represent waiting lines
  - Insertions are made only at the back (tail) of a queue and deletions are made only from the front (head) of a queue.

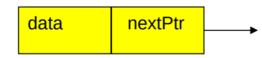


- Binary trees
  - Facilitate high-speed searching and sorting of data



Self-referential data structures

- Contains **a pointer** member that points to a structure of the same structure type.
  - For example




```
struct node {
    int data;
    struct node *nextPtr;
};
```

- Each structure is linked to a succeeding structure by way of the field nextPtr (called a **link**)
  - `nextPtr` contains an address of either the **location in memory** of the successor struct list element or the special value **NULL**.
- Self-referential data structures can be linked together to form useful data structures such as lists, queues, stacks and trees.

Example


```
struct NodeStruct {
    int ID ;
    char Name[50] ;
    double Score ;
    struct NodeStruct * NextPtr ;
} ;
typedef struct NodeStruct Node_t ;

Node_t Node = {0, "", 0.0, NULL}, Node2 ;

Node_t *NodePtr = &Node ;

Node.NextPtr = &Node2 ;        // This way …
NodePtr->NextPtr = &Node2 ;    // … or that way!
```

Memory Allocation

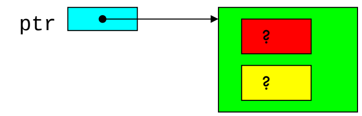
- Static Memory Allocation
  - When programmers write programs
  - We utilize names to declare variables and data structures so that they can refer to those memory locations within their code
  - Once compiled, names and logical locations do not change
- Dynamic Memory Allocation
  - Creating and maintaining dynamic data structures at **execution time**
  - The ability for a program to **obtain more memory space** at execution time to hold new values, and to release space no longer needed.
  - We cannot create variable names to refer to locations – we must use **pointers** instead
  - We focus on **`malloc()`**, **`free()`** and **`sizeof`** – all three are important!

Dynamic Memory Operators

- `sizeof`
  - Unary operator to determine the size in bytes of any data type.
  - Example: `sizeof(double),sizeof(int)`
- `malloc`
  - Takes as an argument the number of bytes to be allocated and return a pointer of type `void *` to the allocated memory. (A `void *` pointer may be assigned to a variable of any pointer type.)
  - It is normally used with the **`sizeof`** operator.
- `free`
  - To de-allocate memory
  - The memory is returned to the system so that the memory can be reallocated in the future.

Example 1


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

ptr = (struct node *) malloc(sizeof(struct node));

free(ptr);
```



Type Casting

Example 2

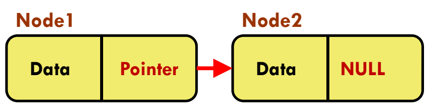

```
struct NodeStruct {
    int ID ;
    char Name[50] ;
    double Score ;
    struct NodeStruct * NextPtr ;
} ;
typedef struct NodeStruct Node_t ;

Node_t * NodePtr = malloc(sizeof(Node_t)) ;

if( NodePtr != NULL )
   printf( “Memory allocation successful!\n” ) ;
else
   printf( “Memory not allocated!\n” ) ;

free( NodePtr ) ;
```

Linked List

Concept of linked structures

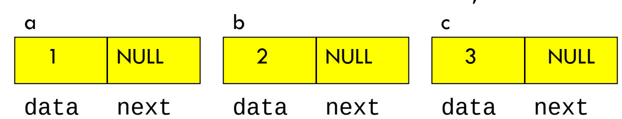
- We will be creating dynamic (ie. runtime) data structures that will contain pointers to other structures
  - These are called **linked structures**.
  - Example:

```
Node_t  Node1, Node2 ;
Node1.NextPtr = &Node2 ;   // points at Node2
Node2.NextPtr = NULL ;    // points nowhere!
```



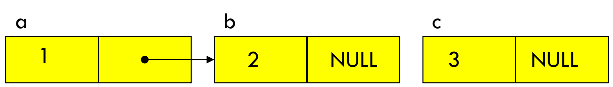
Example 1


```
struct list{
     int data;
     struct node *next;
};
struct list *a, *b, *c;

a->data = 1;
b->data = 2;
c->data = 3;
a->next = b->next = c->next = NULL;
```

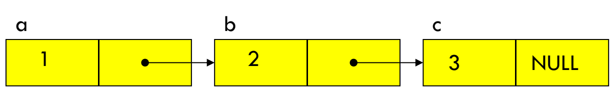


Example 2

- `a->next = &b;`



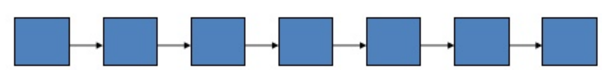
- `b->next = &c;`



- `a->next->data` has value 2
- `a->next->next->data` has value 3
- `b->next->next->data` Error!

Linked lists

- A dynamic data structure whose length can be increased or decreased at run time.
- Contain pointer sub-fields, so that each pointer *points at* another allocated structure
- How Linked lists are different from arrays?
  - An array is a static data structure (the length of array cannot be altered at run time) While, 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 still connected to each other.



Linked lists vs. Arrays

- Advantages of Linked lists over arrays:
  - Items can be added or removed from the middle of the list
  - There is no need to define an initial size
- Disadvantages of linked lists
  - There is no "random" access → Start from the beginning to the desired item.
  - Dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
  - Linked lists have a much larger overhead over arrays, since linked list items are dynamically allocated (which is less efficient in memory usage)

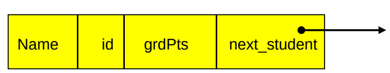
Linked lists

- All linked lists must have an associated **<u>root pointer (head)</u>** that is a <u>named pointer variable</u>.
  - This provides the known address location to enter the list.
- There will be a **last**, or final, element and that one must have a **NULL** value in its link pointer to indicate the logical end-of-list.
- There are several kinds of linked list structures
  - Singly linked list
  - Doubly linked list

Example

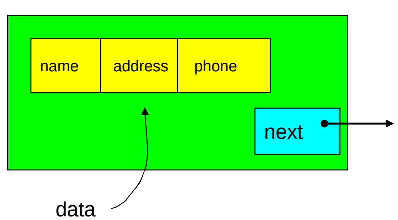
- A node with three data fields:

```
struct student{
  char name[20];
  int id;
  double grdPts;
  struct student *next_student;
};
```



- A structure in a node:

```
struct person{
  char name[20];
  char address[30];
  char phone[10];
};
struct person_node{
  struct person data;
  struct person_node *next;
};
```



Basic Operations on a Linked List

1. Add a node.
2. Delete a node.
3. Search for a node.
4. Traverse (walk) the list.

Adding Nodes to a Linked List

- There are four steps to add a node to a linked list:
  1. Allocate memory for the new node.
  2. Determine the insertion point
  3. Point the new node to its successor.
  4. Point the predecessor to the new node.

- Given the head pointer (**`pHead`**), the predecessor (**`pPre`**) and the data to be inserted (**`item`**).

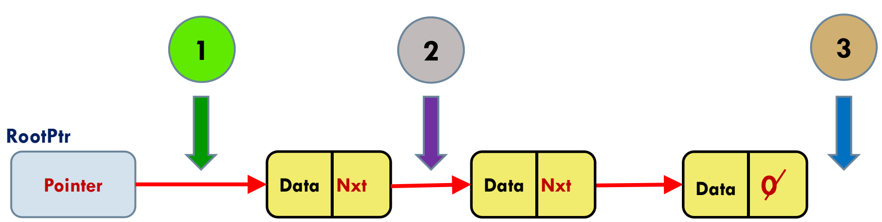
Adding Nodes

- Step1: Allocate memory for the new node and initialize its data

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

struct node *pNew;  //Pointer to a struct
pNew = (struct node *) malloc(sizeof(struct node));
pNew -> data = item;
```

Adding Nodes

- Step 2: Determine the insertion point
  - only the new node's predecessor (`pPre`)
- We must consider four different special cases
  - **1. Insert to an empty list / at head of list**
  - **2. Insert between two list nodes**
  - **3. Insert at End of List**



Adding Nodes

- Step 3: Point the new node to its successor.
- Step 4: Point the predecessor to the new node.
- Pointer to the predecessor (pPre) can be in one of two states:
  - it can be NULL (i.e. Adding either to an empty list or at the beginning of the list)

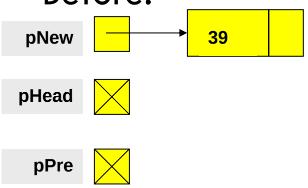
```
pNew -> next = pHead;
pHead = pNew;
```

  - it can contain the address of a node (i.e. Adding either in the middle or at the end the list)

```
pNew -> next = pPre -> next;
pPre -> next = pNew;
```

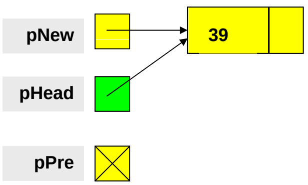
Adding Nodes (Empty List)

Before:



Code:
```
pNew->next = pHead;  // set link to NULL
pHead = pNew ;       // point list to first node
```

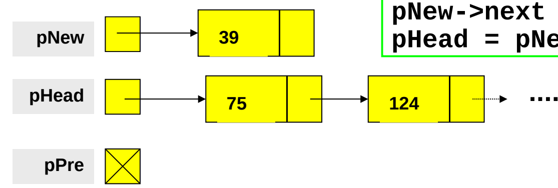
After:



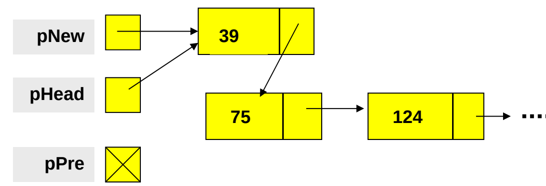
Adding Nodes (Beginning of a List)

Before

Code: (the same)
```
pNew->next = pHead;
pHead = pNew;
```



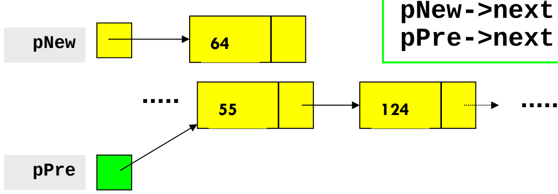
After:



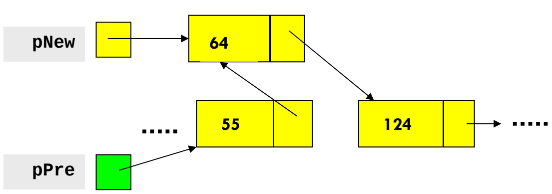
Adding Nodes (Middle of a List)

Before:

Code:
```
pNew->next = pPre->next;
pPre->next = pNew;
```



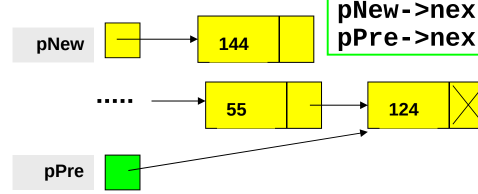
After:



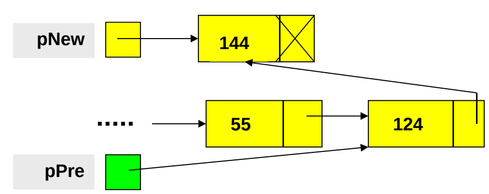
Adding Nodes (End of a List)

Before

Code:
```
pNew->next = NULL; //pPre->next;
pPre->next = pNew;
```



After:



Inserting a Node into a Linked List

- Given the head pointer (`pHead`), the predecessor (`pPre`) and the data to be inserted (`item`).

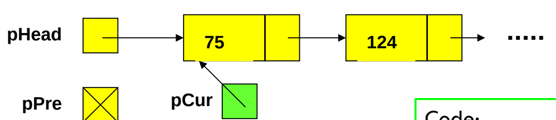
```
//insert a node into a linked list
    struct node *pNew;
    pNew = (struct node *)  malloc(sizeof(struct node));
    pNew -> data = item;
    if (pPre == NULL){
       //add before first logical node or to an empty list
      pNew -> next = pHead;
       pHead = pNew;
    }
    else {
       //add in the middle or at the end
       pNew -> next = pPre -> next;
       pPre -> next = pNew;
   }
```

Deleting a Node from a Linked List

- Deleting a node requires that we logically remove the node from the list by changing various links and then physically deleting the node from the list (i.e., return it to the heap).
- Any node in the list can be deleted.
  - Note that if the only node in the list is to be deleted, an empty list will result. In this case the head pointer will be set to NULL.
- To logically delete a node:
  - First locate the node itself (`pCur`) and its logical predecessor (`pPre`).
  - Change the predecessor's link field to point to the deleted node's successor (located at `pCur -> next`).
  - Recycle the node using the `free()` function.

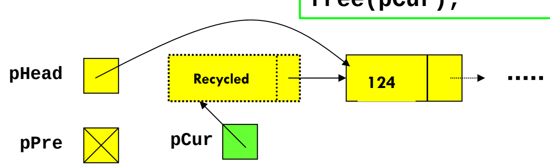
Deleting the First Node from a List

Before:



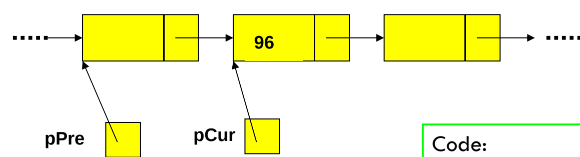
Code:
```
pHead = pCur -> next;
free(pCur);
```

After:



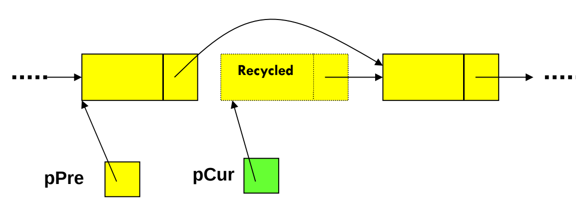
Deleting a Node from a Linked List

Before:



Code:
```
pPre -> next = pCur -> next;
free(pCur);
```

After:



Deleting a Node from a Linked List

33

- Given the head pointer (pHead), the node to be deleted (pCur), and its predecessor (pPre), delete pCur and free the memory allocated to it.

```
//delete a node from a linked list
if (pPre == NULL)
     //deletion is on the first node of the list
      pHead = pCur -> next;
else
    //deleting a node other than the first node of the list
      pPre -> next = pCur -> next;
free(pCur);
```

Searching a Linked List

34

- Notice that both the **insert** and **delete** operations on a linked list must search the list for either the proper insertion point or to locate the node corresponding to the logical data value that is to be deleted.

```
//search the nodes in a linked list to find target
pPre = NULL;
pCur = pHead;
//search until the target value is found or the end of the
   list is reached
while (pCur != NULL && pCur->data != target) {
    pPre = pCur;
    pCur = pCur -> next;
}
//determine if the target is found or ran off the end of the
   list
if (pCur != NULL)
   found = 1;
else
   found = 0;
```

Traversing a Linked List

35

- List traversal requires that all of the data in the list be processed. Thus, each node must be visited and the data value examined.

```
//traverse a linked list
Struct node *pWalker;
pWalker = pHead;
printf("List contains:\n");
while (pWalker != NULL){
     printf("%d ", pWalker -> data);
     pWalker = pWalker -> next;
}
```

Lecture 8: Summary

36

- Self-referential data structures
- Dynamic memory allocation
- Linked lists
- Reading – Chapter 12: Data Structures
  - Abstract data structures, dynamic memory allocation, using pointers and self-referential data structures, linked lists.
- Assignment
  - Deadline of the fourth assignment is **March 25**.
  - Deadline of the fifth assignment is **March 29**.