Recursive Data Representations

We can define structures with pointer fields that refer to the structure type containing them.

```c
struct list {  
    int data;  
    struct list *next;  
}
```

The pointer variable `next` is called a link. Each structure is linked to a succeeding structure by way of the field `next`. The pointer variable `next` contains an address of either the location in memory of the successor `struct list` element or the special value `NULL`.

Example:

```c
struct list a, b, c;
```

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

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NULL</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```c
a.next = &b;
b.next = &c;
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>——</td>
<td>2</td>
</tr>
</tbody>
</table>
```

- `a.next->data` has value 2
- `a.next->next->data` has value 3
- `b.next->next->data` error!!
Static and Dynamic Variables

- Static Variables:
  - They are created during compilation. (Fixed memory is reserved for them.)
  - They cannot be allocated/deallocated during the execution of the program.
  - Names are associated with them.
    
    ```c
    int x;
    char y[10];
    int z[100];
    ```

- Dynamic Variables:
  - They are created (allocated) and deallocated during the execution of the program.
  - No names are associated with them. The only way to access them is to use pointers.
  - They don't exist during compilation. Once they are created they contain data and must have a type like any other variable. Thus we can talk about creating a new dynamic variable of type \( x \) and setting a pointer to point to it, or returning a dynamic variable of type \( x \) to the system (deallocation).

A Conceptual View of Memory
Dynamic Data

For example:

• We must maintain a list of data
• At some moments, the list is small, so we want to use only a little memory

• At other moments, the list is larger, so we need to use more memory

• Declaring variables in the standard way won’t work here because we don’t know how many variables to declare
• We need a way to allocate and deallocate data dynamically (i.e., on the fly)

Dynamic Memory Allocation

Creating and maintaining dynamic data structures requires dynamic memory allocation – the ability for a program to obtain more memory space at execution time to hold new values, and to release space no longer needed.

In C, functions malloc and free, and operator sizeof are essential to dynamic memory allocation.

• Unary operator sizeof is used to determine the size in bytes of any data type.
  
  e.g.
  
  ```
  sizeof(double)
  sizeof(int)
  ```

• Function 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.
Example:

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

struct node *ptr;

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

• Function `free` deallocates memory- i.e. the memory is returned to the system so that the memory can be reallocated in the future.

  e.g.

  ```c
  free(ptr);
  ```

Linked Lists

• It is an important data structure.
• An abstraction of a list: i.e. a sequence of nodes in which each node is linked to the node following it.
• Lists of data can be stored in arrays, but linked lists provide several advantages:

  Arrays
  – In an array each node (element) follows the previous one physically (i.e. contiguous spaces in the memory)
  – Arrays are fixed size: either too big (unused space or not big enough (overflow problem)
  – Maximum size of the array must be predicted which is sometimes impossible.
  – Inserting and deleting elements into an array is difficult.

Linked Lists
  – Linked lists are appropriate when the number of data elements to be represented in the data structure at once is unpredictable.
  – Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
  – Each node does not necessarily follow the previous one physically in the memory.
  – Linked lists can be maintained in sorted order by inserting or deleting an element at the proper point in the list.
Simple Linked List

- The head pointer addresses the first node of the list, and each node points at a successor node. The last node has a link value NULL.

Empty List

Empty Linked list is a single pointer having the value of NULL.

head = NULL;

Nodes

A node in a linked list is a structure that has at least two fields. One of the fields is a data field; the other is a pointer that contains the address of the next node in the sequence.

A node with one data field:

```
struct node{
    int number;
    struct node * link;
};
```

A node with 3 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 Linked List Operations

1. Add a node
2. Delete a node
3. Looking up a node
4. List Traversal (e.g. Counting nodes)

Add a Node

There are four steps to add a node to a linked list:

1. Allocate memory for the new node.
2. Determine the insertion point (you need to know only the new node’s predecessor (pPre))
3. Point the new node to its successor.
4. Point the predecessor to the new node.

Pointer to the predecessor (pPre) can be in one of two states:
- it can contain the address of a node (i.e. you are adding somewhere after the first node – in the middle or at the end)
- it can be NULL (i.e. you are adding either to an empty list or at the beginning of the list)

Adding to Empty List

BEFORE

\[ p_{\text{New}} \rightarrow 39 \]

\[ p_{\text{Head}} \]

\[ p_{\text{Pre}} \]

\[ p_{\text{New}}\rightarrow\text{next} = p_{\text{Head}}; \quad // \text{set link to NULL} \]
\[ p_{\text{Head}} = p_{\text{New}}; \quad // \text{point list to first node} \]

AFTER

\[ p_{\text{New}} \rightarrow 39 \]

\[ p_{\text{Head}} \]

\[ p_{\text{Pre}} \]
Add Node at Beginning

BEFORE

```
// Same code
pNew->next = pHead;
pHead = pNew;
```

AFTER

```
pNew  39
   ^
   |   75   124
   |       ^
   |       |   ---
pHead   pPre
```

Add Node in Middle

BEFORE

```
pNew   96
   ^
   |   75   124
   |       ^
   |       |   ---
pPre
```

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

AFTER

```
pNew   96
   ^
   |   75   124
   |       ^
   |       |   ---
pPre
```
Add Node at End

BEFORE

```
pNew = 134
```

```
    75
    ---
    124

pPre
```

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

OR:
```
// Same as the code for adding a node in the
middle
pNew->next = pPre->next;
pPre->next = pNew;
```

AFTER

```
pNew = 134
```

```
    75
    ---
    124

pPre
```

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

Inserting a Node to a Linked List

Given the head pointer (`pHead`), the predecessor (`pPre`) and the data to be inserted (`item`), we must allocate memory for the new node (`pNew`) and adjust the link pointers.
```
// Insert a node into a linked list

struct node *pNew;
pNew = (struct node *) malloc(sizeof(struct node));
pNew->data = item;
```

```
if (pPre == NULL){
    // Adding before first node or to empty list
    pNew->next = pHead;
pHead = pNew;
}
else {
    // Adding in middle or at end
    pNew->next = pPre->next;
pPre->next = pNew;
}
Delete a Node

Deleting a node requires that we logically remove the node from the list by changing various link pointers and then physically deleting the node from the heap.

We can delete
- the first node
- any node in the middle
- the end node

To logically delete a node:
1. first locate the node itself (pCur) and its predecessor (pPre)
2. change its predecessor’s link field to point to the deleted node’s successor.
3. recycle the node using free.

Note: We may be deleting the only node in a list. This will result in an empty list in which case the head pointer is set to NULL.

Delete First Node

BEFORE

```
pNext = pCur -> next;
free (pCur);
```

AFTER

```
pNext = pCur -> next;
free (pCur);
```
Deleting a node

Given a pointer to the head of the list, the node to be deleted, and the delete node’s predecessor, we delete the node and recycle its memory.

```c
// Delete a node from a linked list
if (pPre == NULL)  // Deleting first node
    pHead = pCur->next;
else  // Deleting other nodes
    pPre->next = pCur->next;

free(pCur);
```
Search Linked List
Both insert and delete operations have to search the linked list.
• To add a node, we must identify the logical predecessor
  of the new node.
• To delete a node, we must identify the location of the
  node to be deleted and its logical predecessor.

Basic Search Concept
Given a target value, the search attempts to locate the requested
node in the linked list. If a node in the list matches the target
value, the search return true; otherwise it returns false.

Traversing Linked Lists
List traversal requires that all of the data in the list be processed.

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

```c
// Search nodes in a linked list
pPre = NULL;
pCur = pHead;

// Search until target is found or we reach
  // the end of list
while (pCur != NULL && pCur->data != target){
    pPre = pCur;
pCur = pCur->next;
}

// Determine if target is found
if (pCur)
    found = 1;
else
    found = 0;
```