Linked List

Prepared by: RNC Recario
rncrecario@gmail.com
Institute of Computer Science UPLB
Feb 2012

OBJECTIVES
At the end of the laboratory session, the student is expected to
- Define what a linked list is
- Identify the different linked lists
- Implement a linked list

Static variable
- Variables that exist throughout the program.

Dynamic variable
- Variables that are created and disposed during the program execution.
- Used to conserve main memory space and use it efficiently.
- Used to store data that vary in size.

Creating Dynamic Variables
malloc – allocates memory to a pointer.

Syntax:
void ***malloc**(size_t size);

where:
size_t type used for memory object sizes and repeat counts special data type that can store a very large integer value.

Example1:

```
int *p;
p = (int *)malloc(4);
//creates a dynamic variable of size 4
      bytes (integer)

*p=17;
```

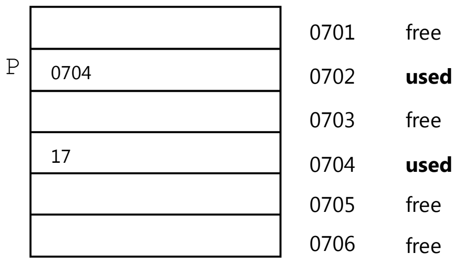


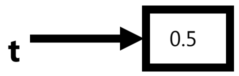
Figure 1. A conceptual view of the memory
Shows how a pointer variable works in the memory

Note: Size of data types vary in different computer systems

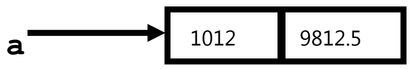
sizeof – returns the size, in bytes, of the given data type (as an unsigned integer).

Syntax:
sizeof(<data_type>);

Example2:
```
float *t;
int *p;

p = (int *)malloc(sizeof(int));
t = (float *)malloc(sizeof(float));
*t = 0.5;
printf("%f", *t);
```



Example3:
```
typedef struct account{
      int number;
      float balance;
}acct;;

acct *a;
a = (acct *)malloc(sizeof(acct));
a->number = 1012;
a->balance=9812.5
```



free – disposes the dynamic variable pointed to by the pointer

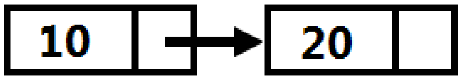
Syntax:
free(<pointer_variable>);

Example4:
```
free(p); //frees memory allocated to p
free(a); //frees memory allocated to a
free(t); //frees memory allocated to t
```

Self-Referential Structures

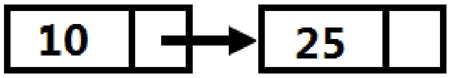
Self-Referential Structures – structure that contains a field that is a pointer to a structure similar to itself.

Example5:
```
struct node{
      int x;
      struct node * ptr;
};

struct node A, B; //the variables are
      A and B

A.x=10;
B.x=20;
A.ptr = &B;
```

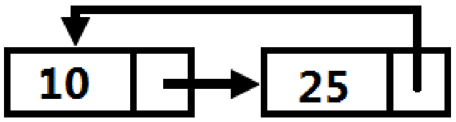


Having pointer **A.ptr** point to B, B can now be referred to as **A.ptr**. Having this in mind, **B.x**, ***(A.ptr)**, **A.ptr->** refer to the integer value 20.
Example6 & 7:

```
a.ptr->x = 25;
```



```
B.ptr = &A;
```



We can also make the variable point to itself. An example would be A.ptr = &A;

In this case, A.x, A.ptr->, A.ptr->ptr->x refer to the same variable.
Question: Using this example can you think of other statements equivalent to 3 given examples for referring the integer portion of A???

Linked List

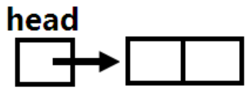
Linked List – a collection of self-referential structures dynamic interconnected to form a chain-like structure.

Node – dynamic variables (which are basically structures) that make up the linked list
- consists of one (or more) field that stores data and another field which is a pointer to the next node.
- head is a pointer variable that points to the first node. The pointer field of the last node is (and SHOULD be) **NULL**, denoting the end of the list.

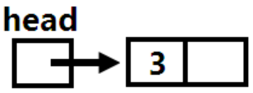
Notes:
A dynamic variable doesn't have a name, thus a pointer is needed to refer to them.
A pointer field having a NULL value does not point to any memory location.

Example8:
Build a linked list consisting of integers 3, 20, 8, 5.

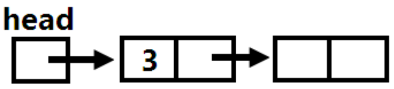
```
struct node{
        int x;
        struct node *next;
};

typedef struct node * nd;
nd *head, *p;
//head and p are both pointers to a structure

//create the first node
head = (nd *)malloc(sizeof(nd));
```



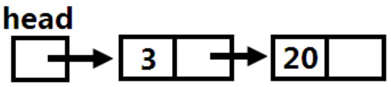
```
//store the value 3 on the first node
head->x = 3;
```



```
//create the second node and store 20
in it
head->next = (nd *)malloc(sizeof(nd));
```



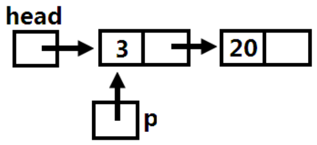
```
head->next->x = 20;
```



Alternatively, we can use the other pointer variable p to effectively create the second node.

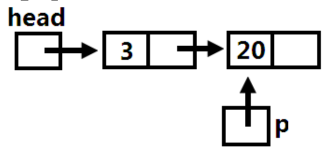
```
p = head;
```

```
//p points to the address pointed to
by head

p->next = (nd *)malloc(sizeof(nd));
p->next->x = 20;
```

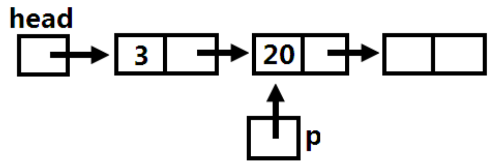


Using the auxiliary pointer (p for example) leads to a more compact and efficient code.

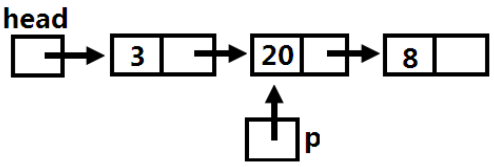
```
p = p->next;
//p points to the address pointed to
by p->next
```



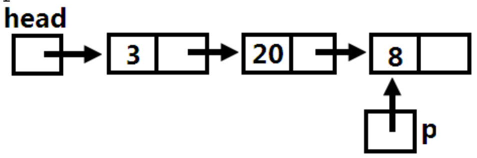
```
p->next = (nd *)malloc(sizeof(nd));
//create a new node
```



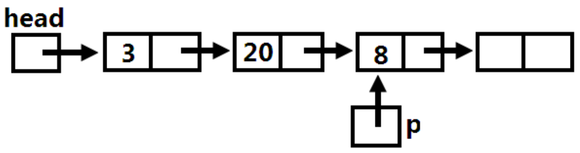
```
p->next->x = 8;
//initialize with 8
```



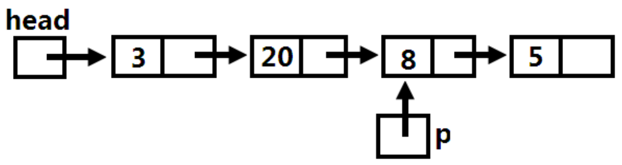
```
p = p->next;
//p points to the address pointed by
p->next
```



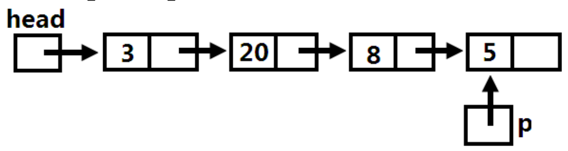
```
p->next   =   (nd   *)malloc(sizeof(nd));
//create a new node
```



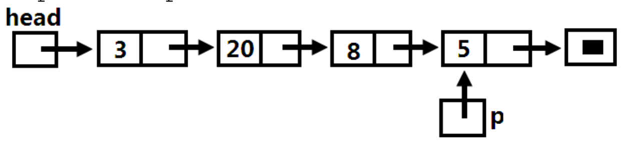
```
p->next->x = 5;
//you   already   know   what   happens,
right???
```



```
p = p->next;
//very easy to understand…
```



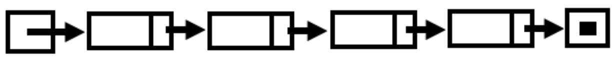
```
p->next = NULL;
//p->next points to NULL
```



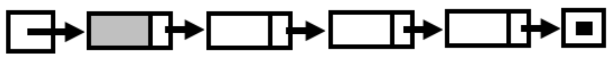
Kinds of Linked List

Here are some of the linked lists that we will be discussing through the semester…

- **Singly Linked List**
  -a linked list with just one pointer variable for the node usually named as next. Next pointer is always NULL for the last node.

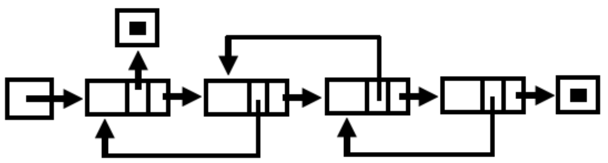


- **Singly Linked List (with dummy)**
  -a singly linked list with a dummy node in the front (or sometimes at the end) of the list. The dummy node is not part of the data but is used to make the coding more efficient and readable.

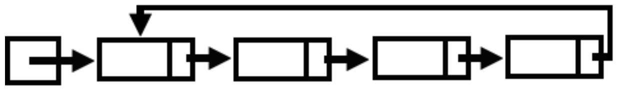


Note: Highlighted node is the dummy node.

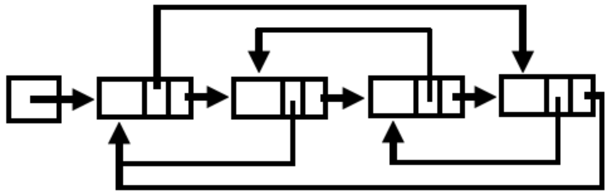
- **Doubly Linked List**
  -a linked list with 2 pointer variables for the nodes succeeding and preceding it. The pointers are usually named as next and prev. The first node's prev pointer is NULL and the last node next pointer is NULL.



- **Circular Singly Linked List**
  -a singly linked list whose last node points to the head (or first node) as opposed to the last node pointing to NULL.



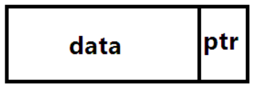
- **Circular Doubly Linked List**
  - a doubly linked list whose last node next pointer points to the head (or first node) as opposed to the last node pointing to NULL. The first node's prev is pointing to the last node.



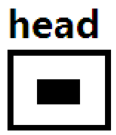
Singly Linked List

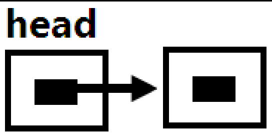
- linked list with one pointer variable usually named NEXT.
- Example8 of this handout shows a sample of a singly linked list.

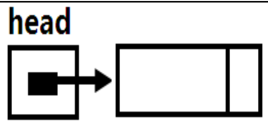
<u>"Parts" of node in a singly linked list</u>



Lab Conventions

Convention	**What it means**
	Uninitialized node pointer named head. Example: MyNode *head;

	A head pointer initially "pointing" to NULL
	
	Example:
	MyNode *head;
	Head = NULL;

	An unintialized node created.
	
	Example:
	MyNode *head;
	
	head = (MyNode *)malloc(sizeof(MyNode));

Operations on Linked List

These operations apply to all the linked lists mention in this handout.

- Building a linked list
- Inserting a node/data
- Deleting a node/data
- Outputting contents of the linked list (Printing)
- Searching data

Building a Linked List

Idea: The values in the list are accessible only by a pointer (usually named head). The important thing to take note of when building a list is that the nodes should be connected together. The head pointer points to the first node; the first node holding the location of the second node, etc. up until the last node. <u>The last node always points to NULL</u>. It is what students always forget.

Consider the example from the data type nd:

```
head = (nd *)malloc(sizeof(nd));

head->x = 3;

p = head;

p->next = (nd *)malloc(sizeof(nd));
//create a new node

p->next->x = 20;

p = p->next;
//p points to the next node

p->next = (nd *)malloc(sizeof(nd));
//create a new node

p->next->x = 8;

p = p->next;
//p points to the next node

p->next = (nd *)malloc(sizeof(nd));
//create a new node

p->next->x = 5;

p = p->next;
//p points to the next node

p->next = NULL;
```

What if there are more than four nodes that we need to add in the list? Obviously, the above approach is inefficient. Below is a pseudocode of a simpler solution.

```
//shorter pseudo code
head = (nd *)malloc(sizeof(nd));
head->x = 3;
p = head;

while there is data{
   p->next = (nd *)malloc(sizeof(nd));
   p->next->x = data;
   p=p->next;
}
p->next = NULL;
```

Of course, if you attempt to use the above "code" it won't work. [It's just a pseudo algorithm anyway! ☺]

A similar approach that you can try coding is the algorithm below:

```
//store integers 1, 2, 3,…10
//in a singly linked list

head = (nd *)malloc(sizeof(nd));
head->x = 1;
p = head;

for(i=2; i<=10; i++){
p->next = (nd*)malloc(sizeof(nd));
p->next->x=i;
p=p->next;
}

p->next=NULL;
```

Inserting a node/data

There are three concerns when inserting data in a linked list (regardless of type/kind).

- Insertion at the head/front of list
- Insertion at the tail/end of list
- Insertion at the middle of the list

General idea when inserting a node in the linked list

To insert a new node into the linked list, four things have to be done…

1. Create a new node using a dynamic variable other than head. Remember: head maintains the linked list. Of course, creation of a new node is only possible using malloc.
2. Initialize values for the elements/components of the newly created node. Do not forget to set your node pointer(s) initially equal to NULL [this is very crucial].
3. Locate the position in the linked list where the newly created node is to be inserted.
4. Manipulate the pointers to cause the insertion. Note that all nodes should still be "linked" together before and after the insertion. Otherwise, the "unlinked" nodes are no longer accessible.
5. Optionally dispose auxiliary/temporary variables by setting them to NULL.

- **Insertion at the head**

Before we start, consider first this declaration:

```
typedef struct Node{
    int num;
    struct Node * next;
} MyNode;

//local or global…
MyNode *head, *temp, *ptr;
Head = NULL;
```

Now, how do we insert st the front of the list. Generally you have to consider two possible states of the list: 1) the list is INITIALLY empty (that is, head points to NULL) and 2) the list contains at least one node.

Here is the algo…

```
temp = (MyNode *)malloc(sizeof(MyNode));
//allocate

if (temp == NULL) //no memory available
    printf("error!");
else{
//there is a memory available
//put some data
temp->num= 4; //let's say 4
temp->next = NULL; //set first to null

if(head == NULL){
     //list is initially empty
      head = temp;
     //just point head to the new node
}
else{
    temp->next = head;
    //temp points to the first node
    head = temp;
    //head points to the same location
    //just like temp

}//else
}//else
```

▪ **Insertion at the end of the list**

```
temp = (MyNode *)malloc(sizeof(MyNode));
//allocate

if (temp == NULL) //no memory available
    printf("error!");
else{
//there is a memory available
//put some data
temp->num= 4; //let's say 4
temp->next = NULL; //set first to null

if(head == NULL){
     //list is initially empty
      head = temp;
     //just point head to the new node
}
else{
      ptr = head;
      while(ptr->next != NULL){
          ptr = ptr->next;
      }//while

      ptr->next = temp;

      //optional
      ptr = NULL;
      temp = NULL;
}//else
}//else
```

▪ **Insertion at the middle of the list**

This will be left as an exercise for you to do

Note: Insertion at the middle is important when dealing with order-oriented linked lists… that it, a linked list whose nodes are ordered in some way. (Example, increasing/decreasing value of the integer component, etc)

Deleting a node/data

Same as the insertion, deletion has three possible cases: deletion at the front, middle and end of the list.

▪ **Deletion at the front of the list**

We need to delete the first node without destroying the link of the remaining nodes with the head pointer. [Note that deletion can be done using the function free().]

```
if (head == NULL){ //linked list is empty
  printf("Nothing to delete!\n");
}
else{
  ptr = head;
  head = head->next;
  free(ptr);

  ptr = NULL; //optional
}//else part
```

▪ **Deletion at the end of the list**

This time we need to traverse the linked list using an extra pointer. We know that we reached the end of the list if the "next" pointer of the current node is already NULL. However, we will modify this approach to make the coding simple.

```
if (head == NULL){ //linked list is empty
  printf("Nothing to delete!\n");
}

else if(head->next == NULL){ //only 1 node
  free(head);
  head = NULL;
}
else{
  while(ptr->next->next == NULL){
       ptr = ptr->next;
  }//while

  free(ptr->next);
  ptr->next = NULL;

  ptr = NULL; //optional
}//else part
```

▪ **Deletion at the middle of the list**

This will be left as an exercise for you to do

Note: Deletion at the middle is important when dealing with order-oriented linked lists… that it, a linked list whose nodes are ordered in some way. (Example, increasing/decreasing value of the integer component, etc)