CS F111: Computer Programming

(Second Semester 2020-21)

Lect 29: Pointers contd., Linked List

Nikumani Choudhury

Asst. Professor, Dept. of Computer Sc. & Information System

BITS Pilani

Hyderabad Campus

Dynamically Allocated Strings

- Write a function (named "concat") that takes two strings s1 and s2 and returns a third string obtained by concatenating s1 and s2

Allocate enough memory to hold s1 and s2:
    size of s1 + size of s2 + 1 (for '\0')
Write s1 to s3
Write s2 to s3 right after writing s1
Add the termination character \0

Continued…

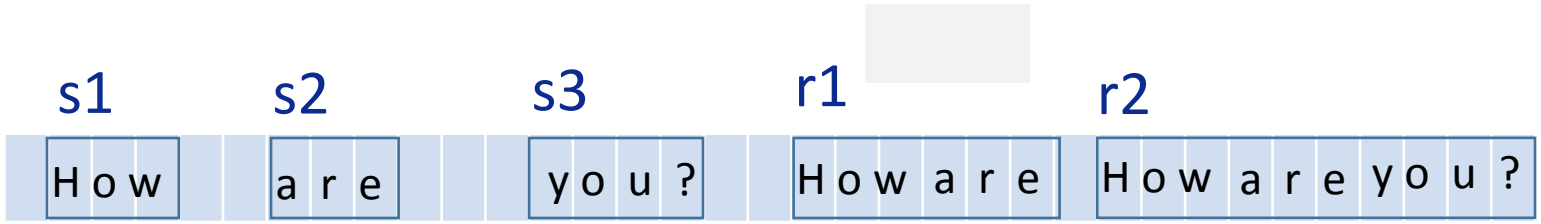
```
char * concat(char *s1, char *s2) {
    char * result;
    result = malloc(strlen(s1) + strlen(s2) + 1);
    if (result == NULL)
        printf("Error: could not allocate memory\n");
    strcpy(result, s1);
    strcat(result, s2);
    return result;
    }
```

Deallocating memory: free

- Memory allocated with malloc lives for the lifetime of the program
- If no longer needed, make sure to free the memory used otherwise program might run out of memory.

```
char *s1 = “How”, *s2 = “are”;
char *s3 = “you?”;
char *r1 = concat(s1, s2);
char *r2 = concat(r1, s3);
```

Continued…



Make sure to free memory that is not needed:

```
free(r1);
```

Another Example: free

• In the following example, the memory pointed to by p becomes garbage when p is assigned a new value:

```
int *p, *q;
p = malloc (sizeof (int) );
q = malloc (sizeof (int) );
p = q;
```

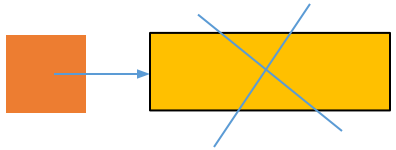
Garbage should be avoided; we need to “recycle” memory instead.

Continued…

• When a block of memory is no longer needed, it can be released by calling the free function:

```
int *p, *q;
p = malloc(sizeof(int));
q = malloc(sizeof(int));
free(p);
p = q;
```

```
int *p;
p = malloc(sizeof(int));
…
free(p);
p = NULL;
```



• *Warning*: Watch for “dangling pointers” left behind by a call of free:

(pointer pointing to non-existing memory location)

Dangling pointer continued…

```
#include<stdio.h>
#include<stdlib.h>
int main() {
  char **strPtr;
  char *str = "Hello!";
  strPtr = &str;
  free(str);

  printf("%s", *strPtr);
}
```

str = NULL;

```
*** Error in `./a.out': munmap_chunk(): invalid pointer: 0x0000000000400654 ***
Aborted (core dumped)
```

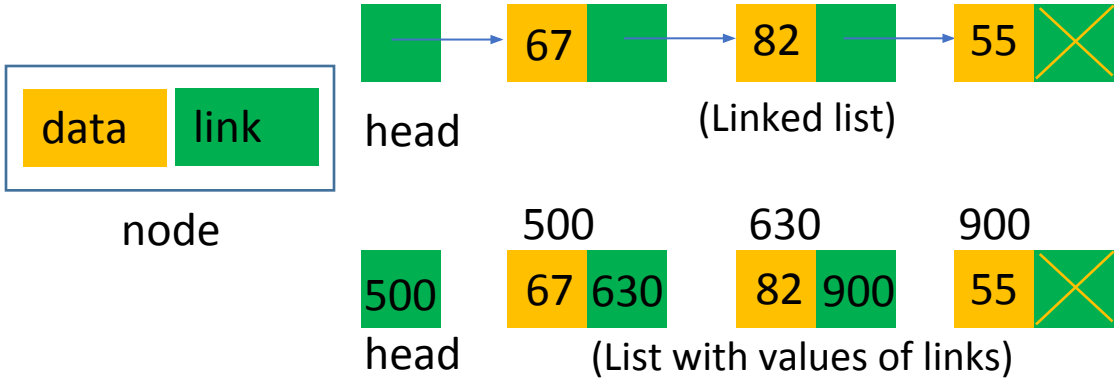
Linked Lists: Why?

- Data can be organized and processed sequentially using an array, called a sequential list
- Problems with an array
  - Array size is fixed
  - Unsorted array: searching for an item is slow
  - Sorted array: insertion and deletion is slow because it requires data movement

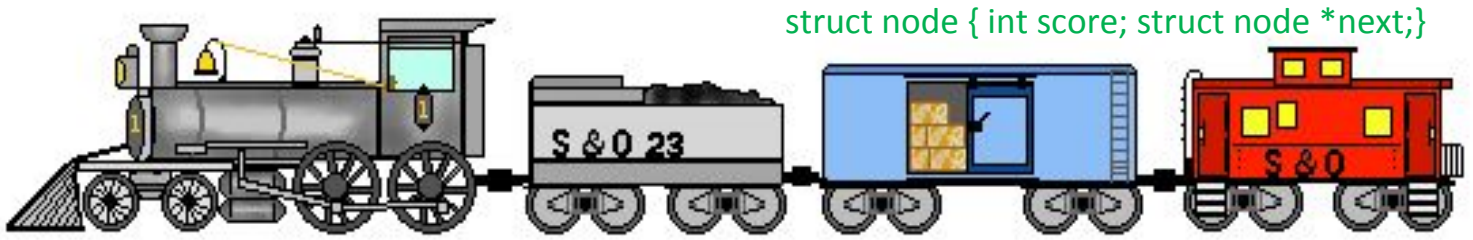
Linked Lists

- Dynamic storage allocation is useful for building lists, trees, graphs, and other linked structures.
- A linked structure consists of a collection of nodes. Each node contains one or more pointers to other nodes. In C, a node is represented by a structure.

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



Linked list: dynamic data structure



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

struct node { int score; char grade, struct node *next; };

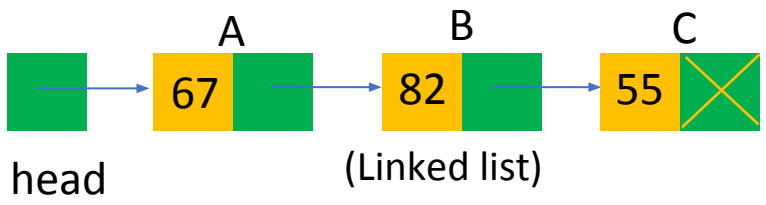
(A real life example, Image Source: https://www.sitesbay.com)

Advantages:

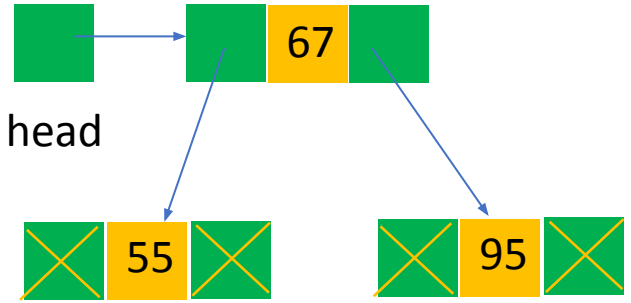
- Dynamic data structure, and hence no memory wastage.
- Insertion and deletion operations are faster.
- Linear data structures like Stacks and Queues can be implemented using Linked lists.

Disadvantage: Need more memory to store addresses.

Linear and Non-linear Lists



Data elements are arranged sequentially and all the elements can be traversed in a single run. Ex.s: Array, Stack, Queue, Linked list.

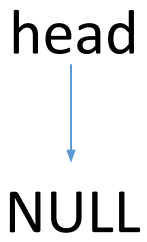


Data elements are not arranged sequentially and not all the elements can be traversed in a single run. Ex.s: Trees and Graphs.

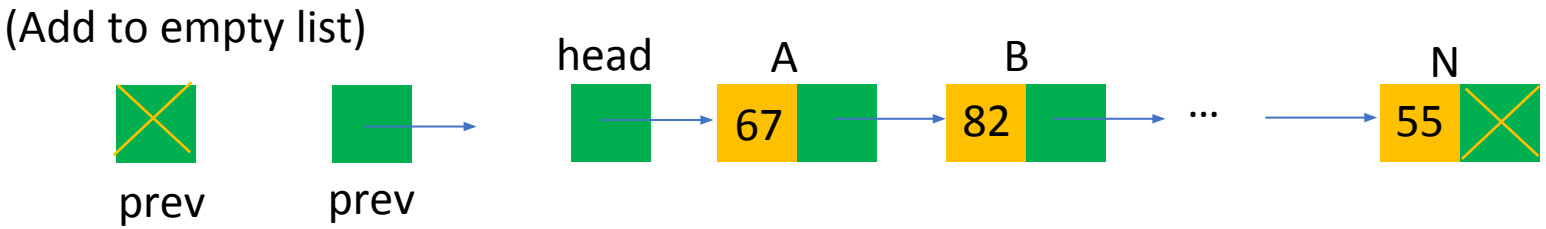
Empty List

• An ordinary pointer variable points to the first node in the list; to indicate that the list is empty, the variable can be assigned NULL:

```
struct node *head = NULL;
```



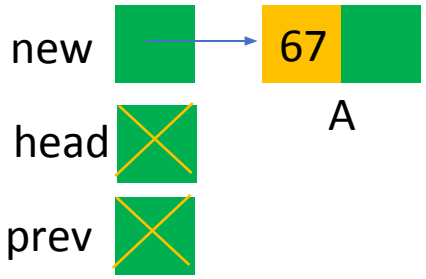
Inserting into Empty list



(Add to middle or end)

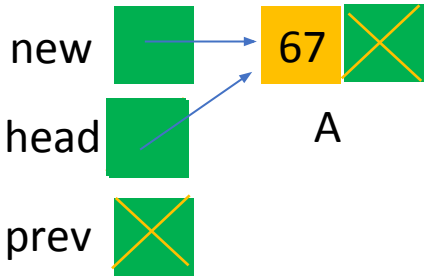
(Pointer combinations for Insert)

(Before adding)

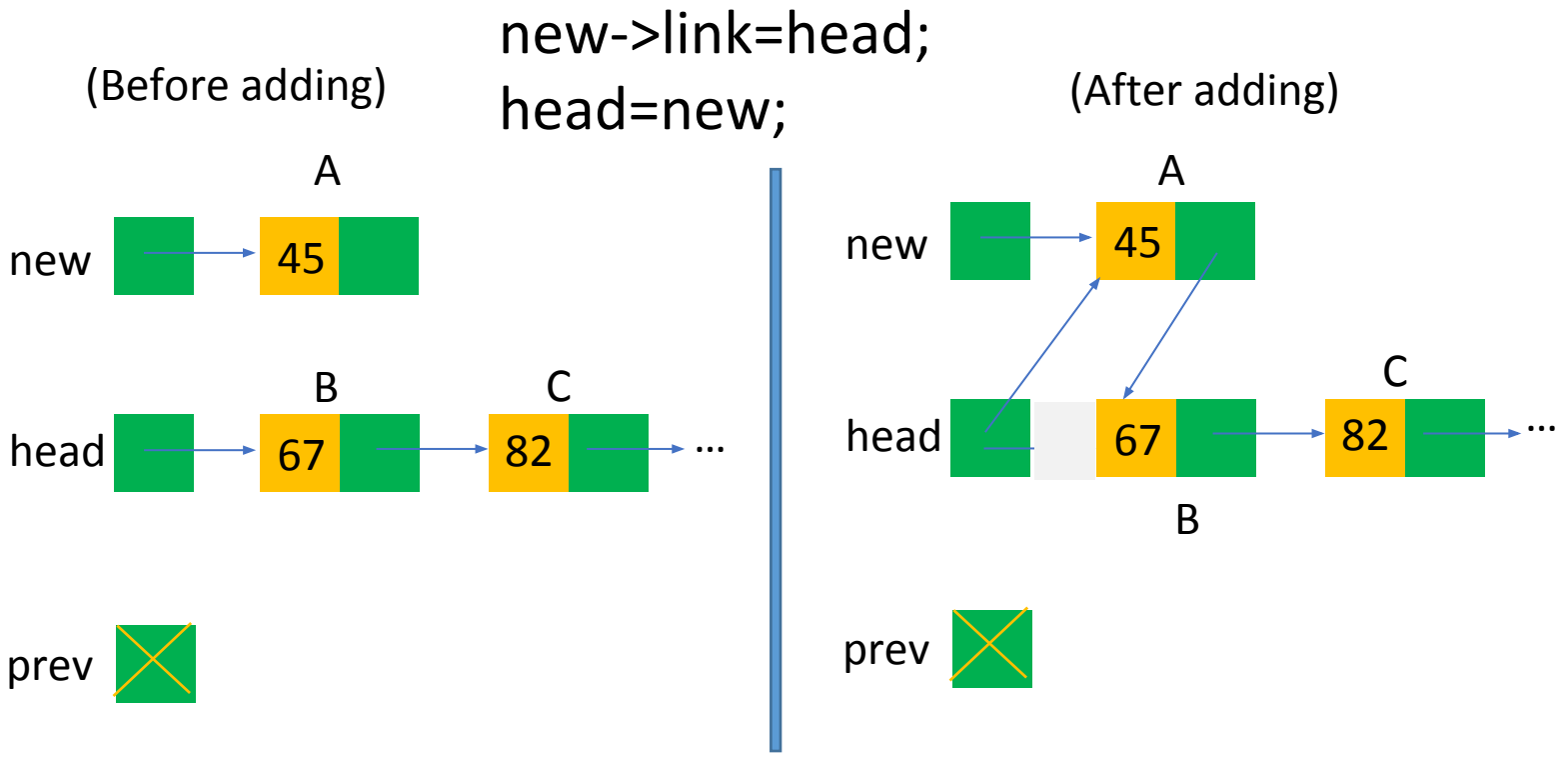


new->link=head;
head=new;

(After adding)

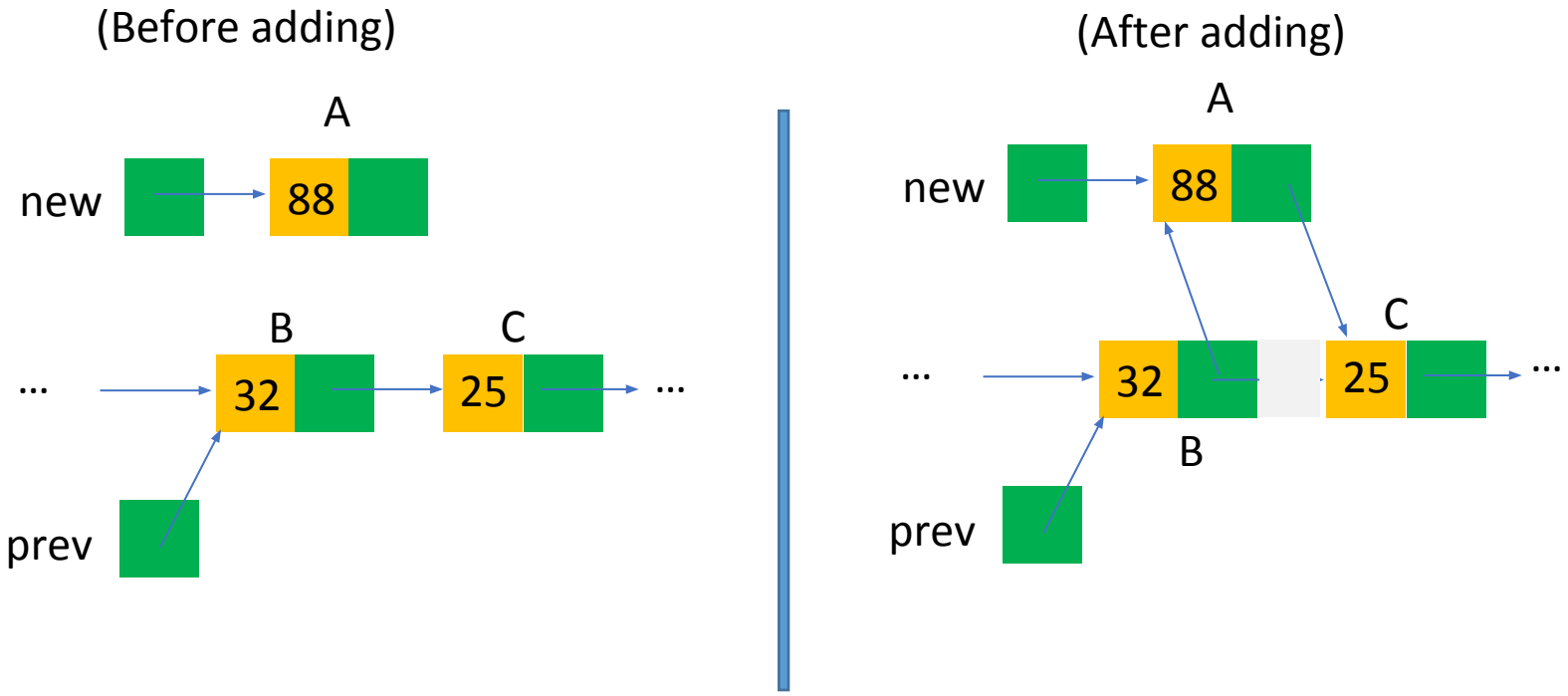


Inserting at the Beginning



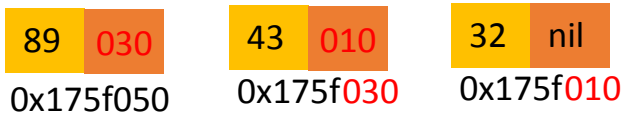
Inserting in the Middle

```
new->link=prev->link;
prev->link=new;
```



Inserting into Linked List: Ex

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  struct node {
     int data;
     struct node *next;
  };
  struct node *first = NULL, *temp;    int n;
  printf("Enter a series of numbers (enter 0 to stop): ");
  scanf("%d", &n);
  while (n != 0) {
     temp = malloc(sizeof(struct node));
     temp->data = n;
     temp->next = first;
     first = temp;
     printf ("%p: %d : %p\n", temp, temp->data, temp->next);
     scanf("%d", &n);}
return 0;}
```




```
Enter a series of numbers (enter 0 to stop): 32
0x175f010: 32 : (nil)
43
0x175f030: 43 : 0x175f010
89
0x175f050: 89 : 0x175f030
```

Insertion at Beginning/ Middle ?