DYNAMIC MEMORY ALLOCATION AND LINKED LIST

Dr Frank Guan

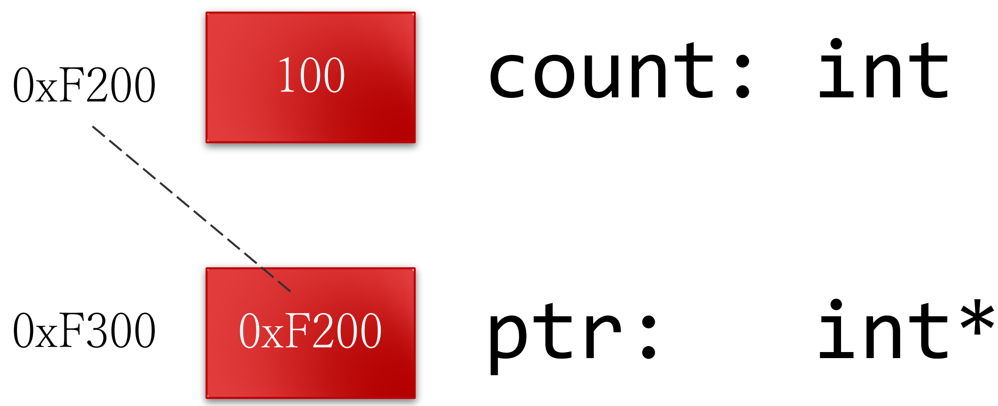
INF1002 – Programming Fundamentals

Week 11

RECAP OF LAST WEEK

Pointers are variables whose values are memory addresses.

```
 int count = 100;

int* ptr = &count;
```



HOW TO USE POINTER

- D: Declaration/definition
  - int variable;
  - int *ptr;

- I: Initialization (value assignment)
  - int variable = 10;
  - ptr = &variable;

- D: Dereference
  - *ptr = 20; (update the value stored in the pointed memory space)
  - int a = *ptr; (retrieve the value stored in the pointed memory space)

POINTERS AND ARRAY

Pointers and arrays are intimately related in C.

- An array name can be thought of as a `constant pointer` to the start of the array.
- Array subscripts can be applied to pointers.
- Pointer arithmetic can be used to navigate arrays.

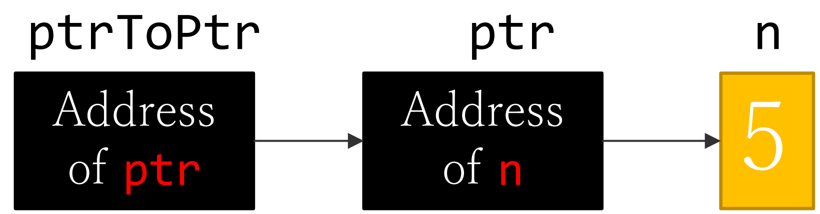
```
int main() {

    char b[] = {'a', 'b', 'c', 'd', 'e' };
    char *bPtr = b;

    printf("*(bPtr + 3): \t%c\n", *(bPtr + 3));
    printf("*(b + 3): \t%c\n", *(b + 3));
    printf("bPtr[3]: \t%c\n", bPtr[3]);

        return 0;
}
```

Output

```
*(bPtr + 3): d
*(b + 3):    d
bPtr[3]:     d
```

POINTER TO POINTER

```
int n = 5;
int *ptr = &n;
int **ptrToPtr = &ptr;
```



`(int *)`: pointer to an integer

`(int *)*`: pointer to a pointer which points to an integer

Many uses in C:

- Arrays of pointers
- Arrays of strings

Agenda

1. Void pointer
2. Dynamic memory allocation
3. User-defined data types
4. Linked lists

VOID POINTER

SINGAPORE INSTITUTE OF TECHNOLOGY

VOID POINTERS

```
int x;
void *xPtr = &x;
printf("xPtr: %p\n", xPtr);

float f;
void *fPtr = &f;
printf("fPtr: %p\n", fPtr);
```

All pointers can be assigned to a pointer to `void`.

A pointer to `void` can point to a variable of any type.

VOID POINTERS

```
float f = 123.45;

/* incorrect */
void *fPtr = &f;
printf("*fPtr: %f\n", *fPtr);

/* correct */
float *fPtr2 = (float *)fPtr;
printf("*fPtr2: %0.2f\n", *fPtr2);
```

The compiler says:

```
void_pointers.c
void_pointers.c(16): error C2100:
illegal indirection
```

A pointer to void **cannot be dereferenced**. Void pointers should always be **cast** before dereferencing.

DYNAMIC MEMORY ALLOCATION

SINGAPORE INSTITUTE OF TECHNOLOGY

WHY DYNAMIC MEMORY ALLOCATION?

```
include <stdio.h>

#define NUM_STUDENTS 10

int main() {

        int grades[NUM_STUDENTS];
        int i;

        for (i = 0; i < NUM_STUDENTS; i++) {
                printf("Grade for student %d: ", i + 1);
                scanf("%d", &grades[i]);
        }

        return 0;

}
```

NUM_STUDENTS has to be defined at compile time

This loop reads the grades for students

What happens if we do not know how many students we have in advance?

We need DYNAMIC memory allocation

DYNAMIC MEMORY ALLOCATION

Three steps to dynamic memory allocation:

1. include

```
#include <stdlib.h>
```

2. malloc

Use `malloc` or `calloc` to request memory.

```
int *ptr = (int *)malloc(sizeof(int)*N);
```

3. free

Free up the memory when no longer needed.

```
free(ptr);
```

MALLOC AND CALLOC

`malloc` allocates a block of memory with a given number of bytes

```
int *ptr = (int *)malloc(sizeof(int) * N);
```

`calloc` allocates a block of memory with space for a given number of elements, and sets them to zero

```
int *ptr = (int *)calloc(N, sizeof(int));
```

malloc VS calloc

https://stackoverflow.com/questions/1538420/difference-between-malloc-and-calloc

MALLOC AND CALLOC

`malloc` and `calloc` return a void pointer to the start of the allocated memory

- it MUST be explicitly cast to the appropriate type before use
- it is often used like an array
- if not enough memory is available, the pointer has the special value `NULL`

```
int *grades = (int *)malloc(num_students * sizeof(int));
for (i = 0; i < num_students; i++)
        grades[i] = …;
```

THE SIZEOF OPERATOR

To be able to allocate memory dynamically, we need to tell the compiler the size of memory we need at runtime.

The `sizeof` operator returns the number of bytes required to hold a type.

- E.g.
  - `sizeof(char)` evaluates to 1
  - `sizeof(int)` evaluates to 2, 4 or 8 depending on the word size of the compiler

THE SIZEOF OPERATOR

This gives the size of an integer.

```
int size = sizeof(int) * 4;
printf("size of 4 integers is:
    %d bytes\n", size);
```

THE SIZEOF OPERATOR

This gives the size of 4 integers.

```
int size = sizeof(int) * 4;
printf("size of 4 integers is:
    %d bytes\n", size);
```

```
Output:
size of 4 integers is: 16 bytes
```

FREE

`free` de-allocates memory previously allocated by `malloc` or `calloc`

- all memory allocated by `malloc` or `calloc` should eventually be `free`'d
- this allows the memory to be re-used
- failure to de-allocate memory is called a memory leak
- a leaking program will use up more and more memory over time, and eventually crash

EXAMPLE

```c
#include <stdio.h>
#include <stdlib.h>

int main() {

    int *grades;
    int num_students;

    /* ask how many grades need to be stored */
    printf("How many students are in your class? ");
    scanf("%d", &num_students);

    /* allocate enough space to hold num_students integers */
    grades = (int *)malloc(num_students * sizeof(int));
    if (grades == NULL) {
        printf("Out of memory.");
        return 1;
    }

    /* read the grades */
    for (int i = 0; i < num_students; i++) {
        printf("Grade for student %d: ", i + 1);
        scanf("%d", &grades[i]);
    }

    /* de-allocate memory */
    free(grades);

    return 0;

}
```

With dynamic memory allocation, we can set the number of students at run time.

If there is not enough memory, `malloc` returns NULL.

Use `free` to de-allocate memory when it is no longer required.

DYNAMIC MEMORY ALLOCATION

STRINGS

Self-exercise: names_dynamic.c

```c
/* allocate enough space to hold num_students strings */
char **names = (char **)malloc(num_students * sizeof(char *));
if (names == NULL) {
    printf("Out of memory.");
    return 1;
}

for (i = 0; i < num_students; i++) {
    /* read the name */
    printf("Name of student %d: ", i + 1);
    fgets(buf, MAX_NAME, stdin);

    /* copy the name into the array */
    int length = strchr(buf, '\n') - buf;
    names[i] = (char *)calloc(length + 1, sizeof(char));
    if (names[i] == NULL) {
        printf("Out of memory.");
        return 1;
    }
    strncpy(names[i], buf, length);
}

/* de-allocate memory */
for (i = 0; i < num_students; i++)
    free(names[i]);
free(names);
```

`names` is an array of pointers to characters.

Allocate space for each string according to its length.

Invoke **free** for each allocated block of memory.

USER-DEFINED DATA TYPES

SINGAPORE INSTITUTE OF TECHNOLOGY

STRUCTURES

Suppose you want to represent this information about a `student`.

Name	Sachin Kumar
Roll	101
Age	16
Class	INF1002

STRUCTURES

C allows structured collections of information to be defined using the struct keyword.

```
struct <name> {
  member 1;
  member 2;
  :
  member n;
};
```

```
struct student {
  char name[20];
  int roll;
  int age;
  char class[12];
};
```

DECLARE STRUCTURES VARIABLE

```
Option #1:

struct student {
    char name[20];
    int roll;
    int age;
    char class[12];
};

struct student student_1;
```

```
Option #2:

struct student {
    char name[20];
    int roll;
    int age;
    char class[12];
} student_2, student_3;
```

The code above declares 3 variables of type `struct student`, called `student_1`, `student_2`, and `student_3`

USE STRUCTURE VARIABLE

```
/*
 * struct example from Sharma
 */
#include <stdio.h>

struct student {
         char name[20];
         int roll;
         int age;
         char class[12];
};

int main() {

         /* initialise a variable of type student */
         struct student stud1 = { "Sachin Kumar", 101, 16, "INF1002" };

         /* display contents of stud1 */
         printf("\n Name : %s", stud1.name);
         printf("\n Roll : %d", stud1.roll);
         printf("\n Age  : %d", stud1.age);
         printf("\n Class: %s", stud1.class);

         return 0;

}
```

Output

```
Name : Sachin Kumar
Roll : 101
Age  : 16
Class: INF1002
```

Structures can be initialised similar to arrays.

Use the dot operator to refer to members of a structure.

PASSING STRUCTURES TO FUNCTIONS BY REFERENCE

```
/*
 * struct example with functions
 */
#include <stdio.h>

void print_student(Student *s);

int main() {

        /* initialise a variable of type Student */
        Student stud1 = { "Sachin Kumar", 101, 16, "INF1002" };

        /* display contents of stud1 */
        print_student(&stud1);

        return 0;
}

void print_student(Student *s) {

        printf("\n Name : %s", s->name);
        printf("\n Roll : %d", s->roll);
        printf("\n Age  : %d", s->age);
        printf("\n Class: %s", s->class);

}
```

Structures can be passed by reference.

Use the "->" arrow operator to de-reference a pointer to a structure.

HOW TO MAKE IT SHORT?

– struct student stud1 = { “Bill Gates", 101, 16, "INF1002" };
– struct student stud2 = { “Elon Musk", 102, 16, “INF1003" };
– …

LIFE EXAMPLE

- Singapore Institute of Technology is Singapore's University of Applied Learning. Singapore Institute of Technology's vision is to be a leader in innovative learning by integrating learning, industry and community. Singapore Institute of Technology's mission is to nurture and develop individuals who build on their interests and talents to impact society in meaningful ways.

- Singapore Institute of Technology (from now on referred as "SIT") is Singapore's University of Applied Learning. SIT's vision is to be a leader in innovative learning by integrating learning, industry and community. SIT's mission is to nurture and develop individuals who build on their interests and talents to impact society in meaningful ways.

USER-DEFINED DATA TYPES - TYPEDEF

```
typedef <type> <new_type>
```

User-defined data types can be declared using `typedef`:

`<type>` can be a basic data type or struct

`<new_type>` is the user-defined data type

USER-DEFINED DATA TYPES - TYPEDEF

```
struct student {
   char name[20];
   int roll;
   int age;
   char class[12];
};

typedef (struct student) Student;

/* initialise a variable of type Student */
Student student_1 = { "Sachin Kumar", 101, 16, "INF1002" };
```

USER-DEFINED DATA TYPES - TYPEDEF

We can make it even more concise!!!

```
typedef struct {
    char name[20];
    int roll;
    int age;
    char class[12];
} Student;

/* initialise a variable of type student */
Student student_1 = { "Sachin Kumar", 101, 16, "INF1002" };
```

USER-DEFINED DATA TYPES - TYPEDEF

```
typedef float salary;
salary wages_of_month;
```

In this example `wages_of_month` is of type `salary` which is a `float` by itself.
This enhances the readability of the program.

THE SIZEOF OPERATOR

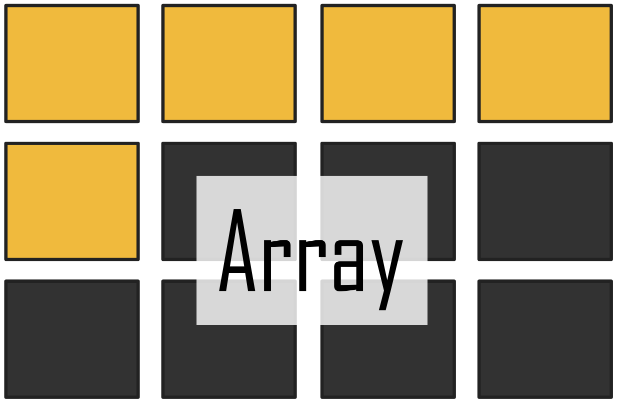
The **sizeof** operator can also be used with user-defined types:

```
typedef struct {
	int id;
	char name[25];
} Student;
```

```
printf("The size of a Student record is:
	%d bytes\n", sizeof(Student));
```

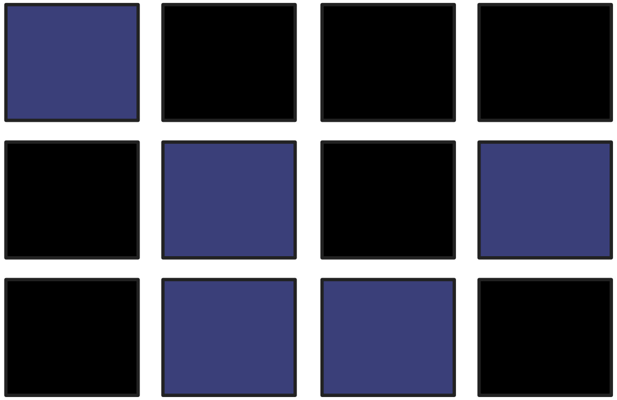
LINKED LISTS

SINGAPORE INSTITUTE OF TECHNOLOGY



Arrays need contiguous memory slots. With a linked list, you can optimise memory by **linking data** at **different** memory **locations**.

Why are you teaching us **Linked Lists**? We are happy with arrays!

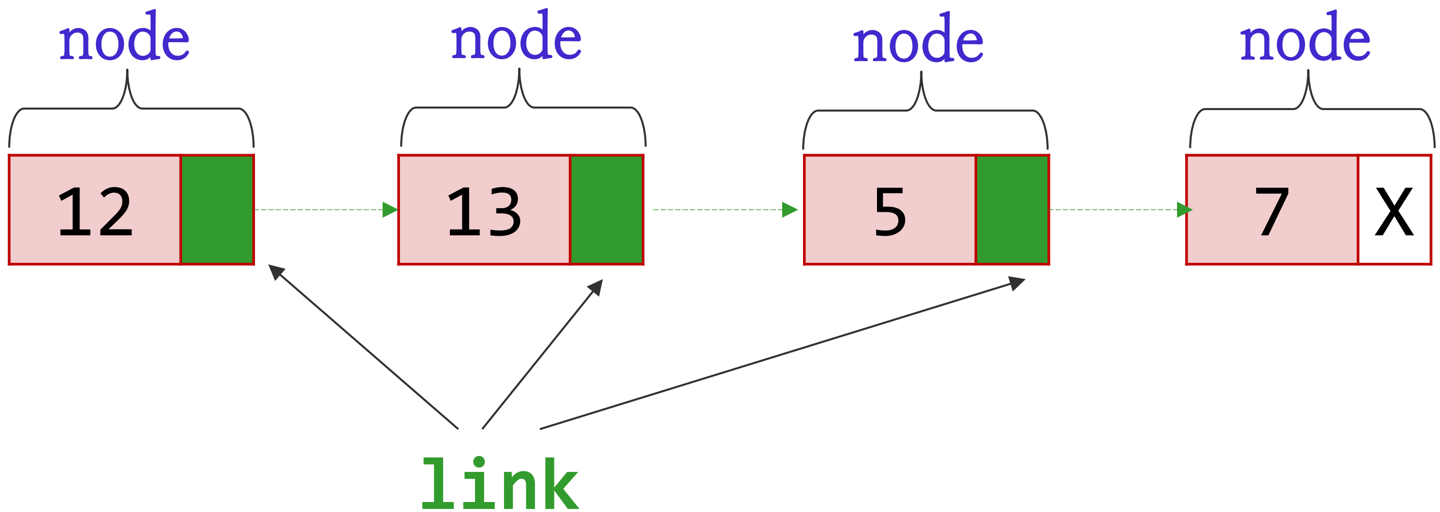


Linked list

LIFE EXAMPLE

- The professor (me) left his water bottle in the classroom and one student found it and kept it for the professor
- There are 5 students: A, B, C, D, E
- A knows the address of B, B knows the address of C, and so forth
- I only know the address of A.
- Q: how can I get my bottle back?

LINKED LISTS



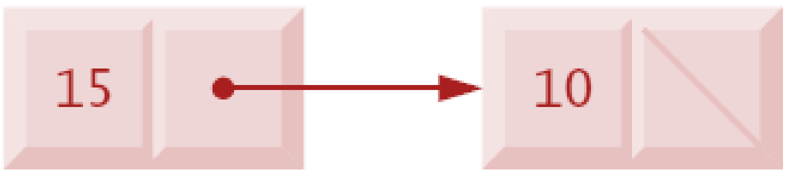
A linked list is a linear collection of self-referential structures, called **nodes**, connected by pointers, called links.

SELF-REFERENTIAL STRUCTURES

```
typedef struct node_struct {
    int data;
    struct node_struct *next;
} Node;
```

A self-referential structure contains a `pointer member` that points to a structure of the same type

SELF-REFERENTIAL STRUCTURES

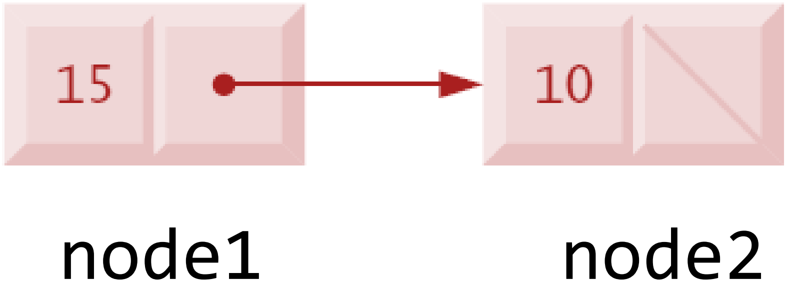



```
typedef struct node_struct {
    int data;
    struct node_struct *next;
} Node;
```

Self-referential structures can be linked together to form useful data structures such as linked lists, queues, stacks and trees.

How do you create two nodes and link **node1** to **node2**?

```
int main() {

       Node node1 = { 15, NULL };
       Node node2 = { 10, NULL };
       node1.next = &node2;

}
```



ACCESSING DATA IN A LINKED LIST

```
int main() {

        Node node1 = { 15, NULL };
        Node node2 = { 10, NULL };
        node1.next = &node2;

        printf("node1.data = %d\n", node1.data);
        printf("node1.next = %p\n", node1.next);

}
```

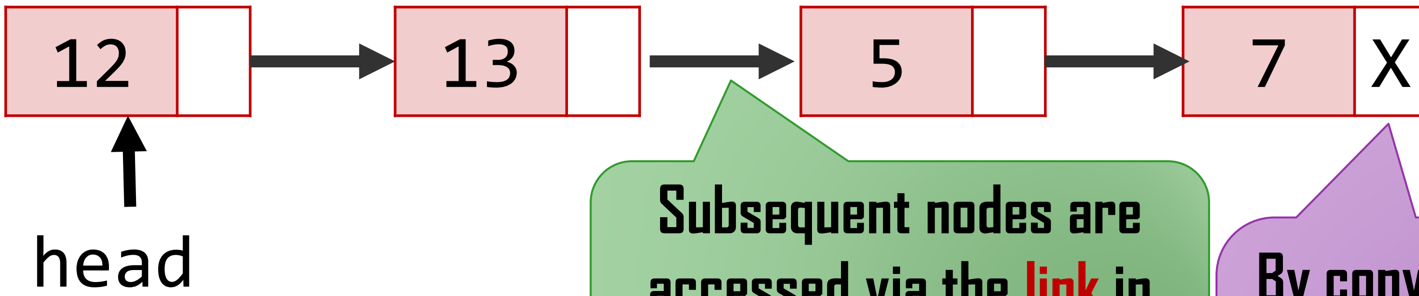
Use the dot "." notation for **non-pointer variables**

ACCESSING DATA IN A LINKED LIST

```
int main() {

        Node node1 = { 15, NULL };
        Node node2 = { 10, NULL };
        node1.next = &node2;

        Node *node_ptr = &node1;
        printf("node1.data = %d\n", node_ptr->data);
        node_ptr = node_ptr->next;
        printf("node2.data = %d\n", node_ptr->data);

}
```

Use the arrow "->" operator for pointer variables

ACCESSING DATA IN A LINKED LIST



A linked list is accessed via a pointer to the first node (head).

Subsequent nodes are accessed via the link in each node.

By convention: the link pointer in the last node points to NULL.

LINKED LIST OPERATIONS

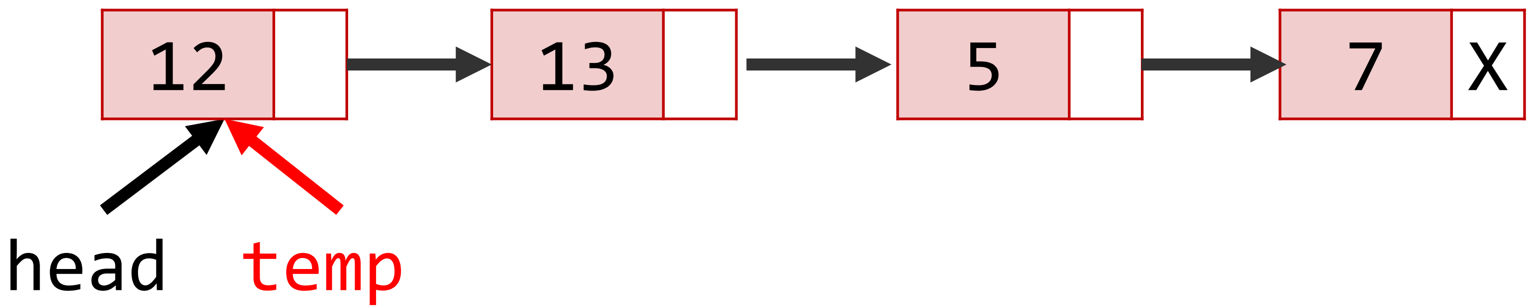
Search/update

Insert

Delete

LINKED LIST OPERATIONS

Search

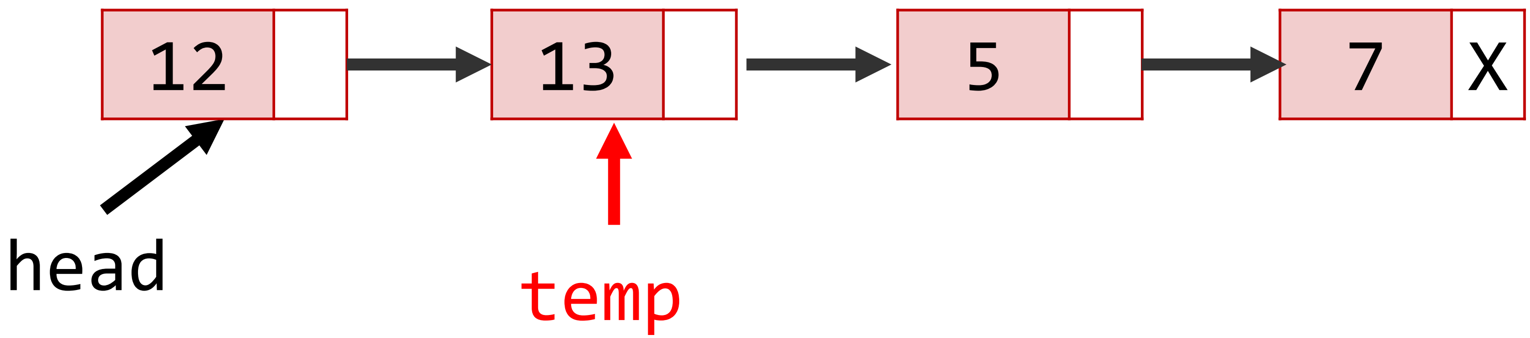


`temp` is a pointer to the first node of the list.

How do we move **`temp`** to the node containing **`5`**?

LINKED LIST OPERATIONS

Search

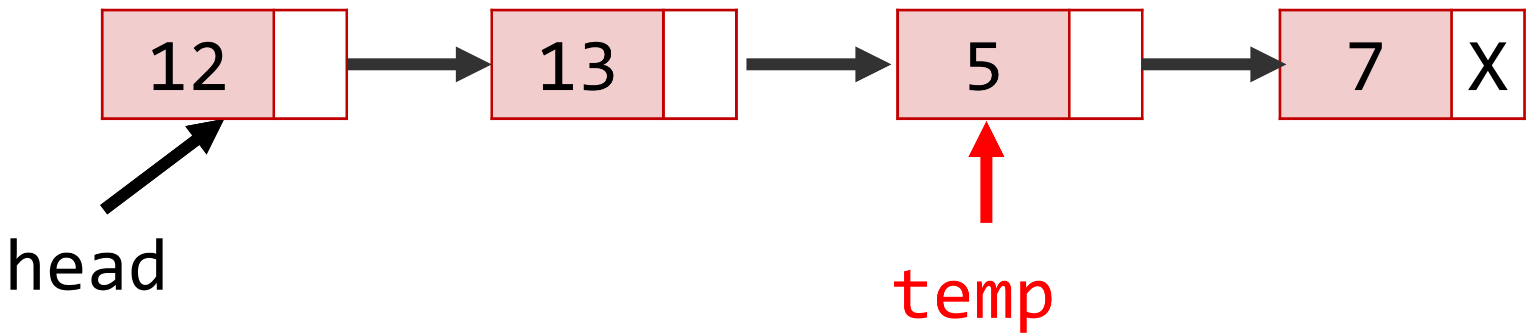



```
temp = temp->next;
```

Using the **next** pointer

LINKED LIST OPERATIONS

Search

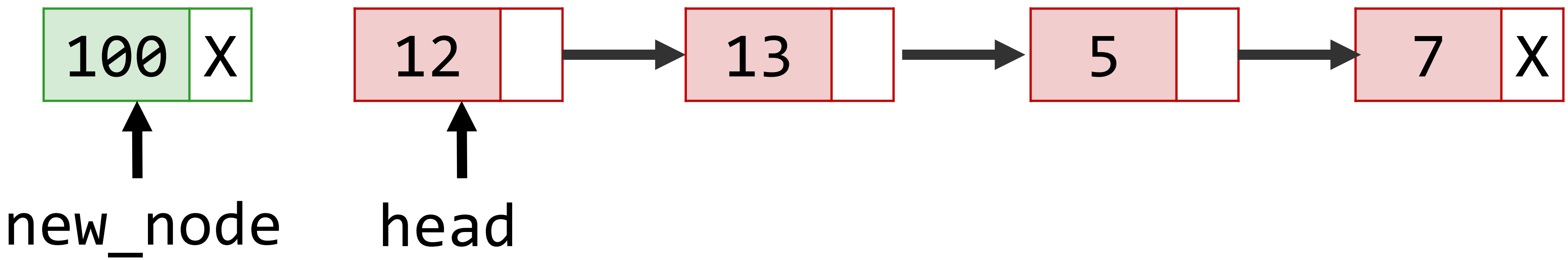



```
temp = temp->next;
temp = temp->next;
```

Using the **next** pointer

LINKED LIST OPERATIONS

Insert

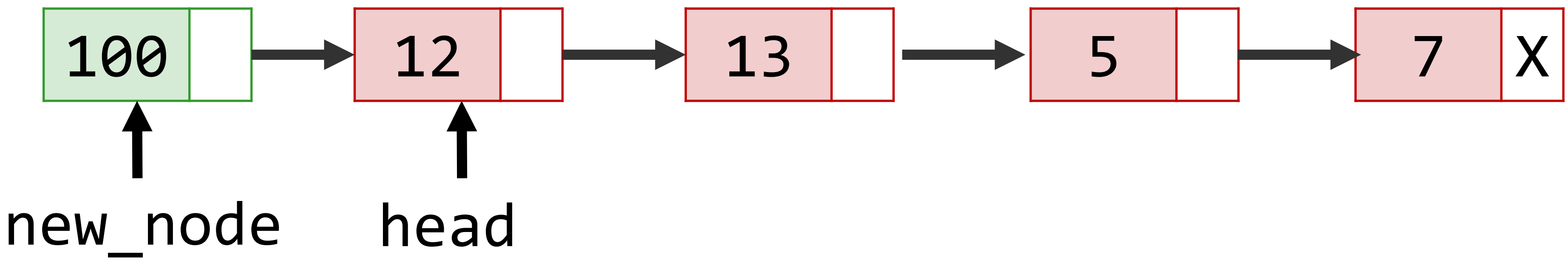



```
Node *new_node = (Node *)malloc(sizeof(Node));
new_node->data = 100;
new_node->next = NULL;
```

How do we insert **new_node** at the beginning of the list?

LINKED LIST OPERATIONS

Insert

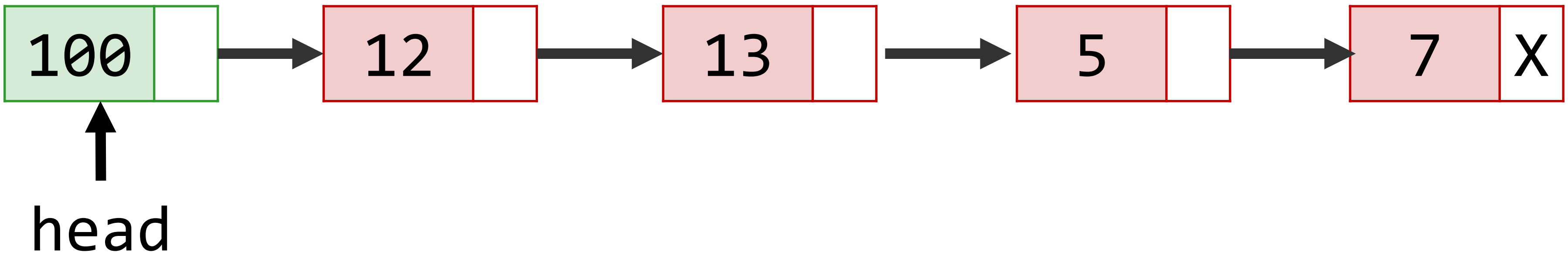



```
new_node->next = head;
```

1. Link the new node to the old head.

LINKED LIST OPERATIONS

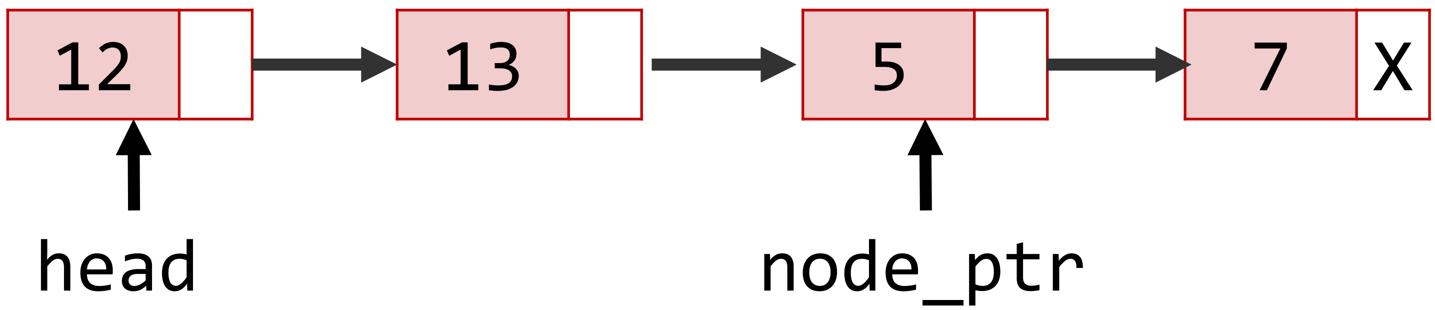
Insert




```
new_node->next = head;
head = new_node;
```

2. Move the head to the new node.

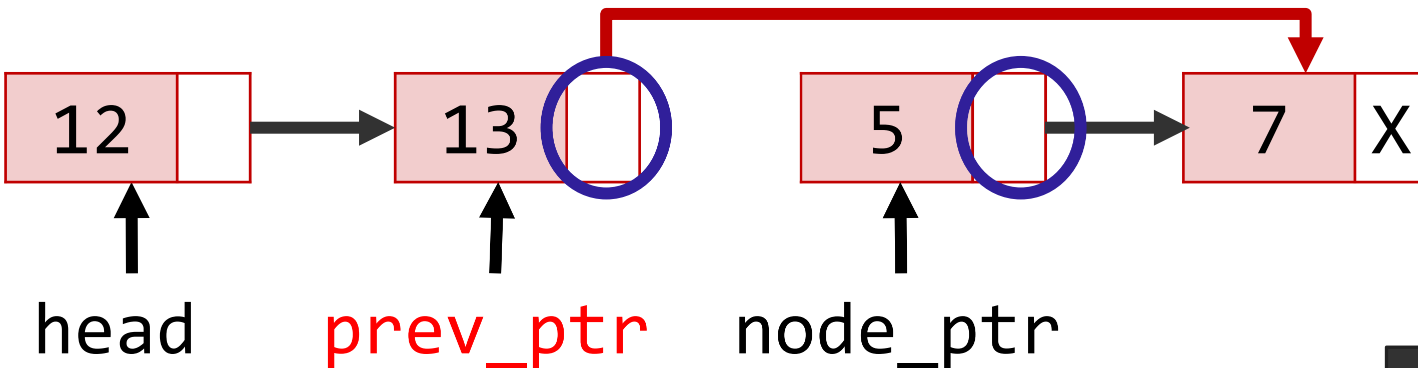
LINKED LIST OPERATIONS



How do we delete the node pointed to by **`node_ptr`**?

LINKED LIST OPERATIONS

Delete



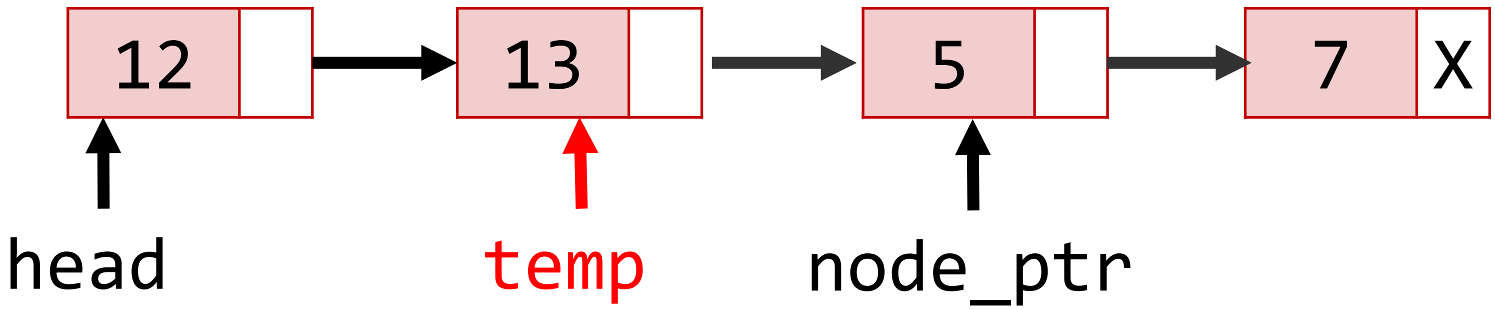

```
prev_ptr->next = node_ptr->next;
free(node_ptr);
```

Don't forget to free the node if it was created by `malloc`.

How do we get prev_ptr?

LINKED LIST OPERATIONS

Delete




```
Node* Find_Pre_Node()
{
    Node *temp = head;
    while (temp->next != NULL)
    {
        if (temp->next == node_ptr){
                return temp;
        }
        temp = temp->next;
    }
}
```

END-OF-WEEK CHECKLIST

- [ ] Dynamic memory allocation
- [ ] The sizeof operator
- [ ] malloc() and free()
- [ ] Self-referential structures
- [ ] User-defined data types
- [ ] Dot operators
- [ ] Linked lists
- [ ] Linked lists vs arrays
- [ ] Searching & updating lists
- [ ] Inserting into linked lists
- [ ] Deleting from linked lists