DYNAMIC MEMORY ALLOCATION

Dr Frank Guan

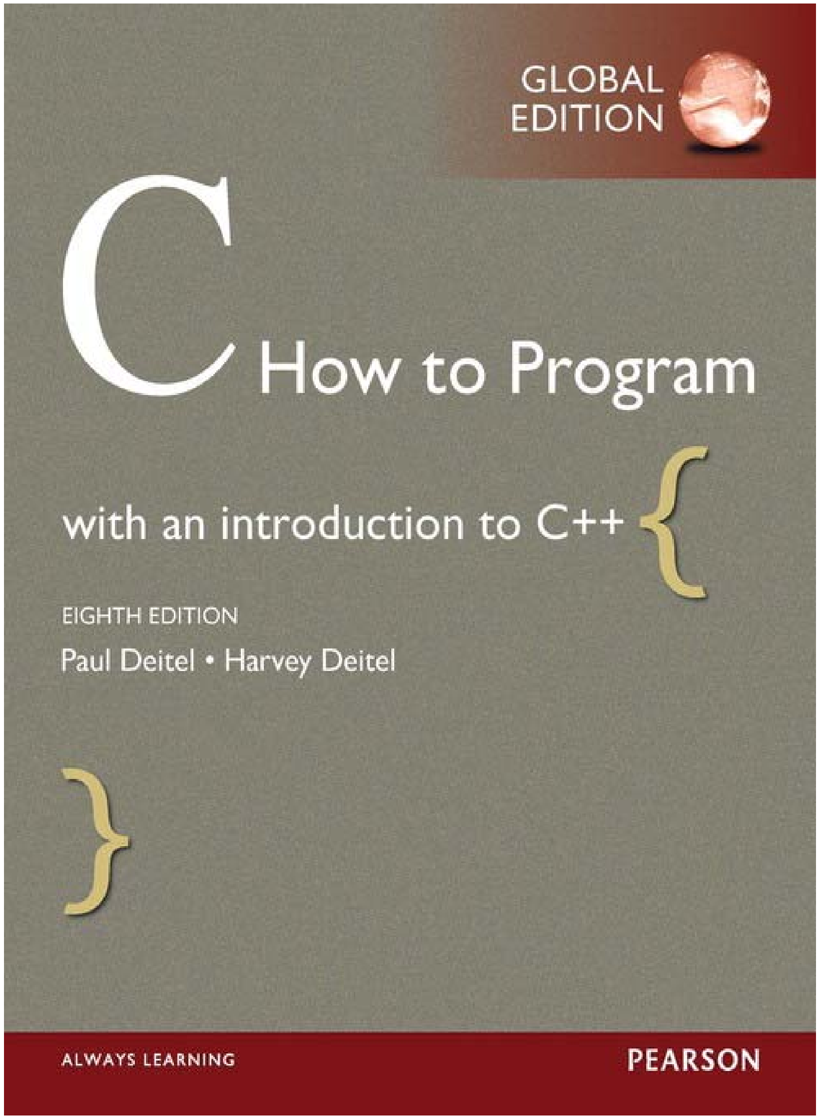
ICT1002 – Programming Fundamentals

Week 11

Agenda

1. Dynamic memory allocation
2. Linked lists

RECOMMENDED READING



Paul Deitel and Harvey Deitel, *C: How to Program*, 8th Edition, Prentice Hall, 2016

- Chapter 12: *C Data Structures*

DYNAMIC MEMORY ALLOCATION

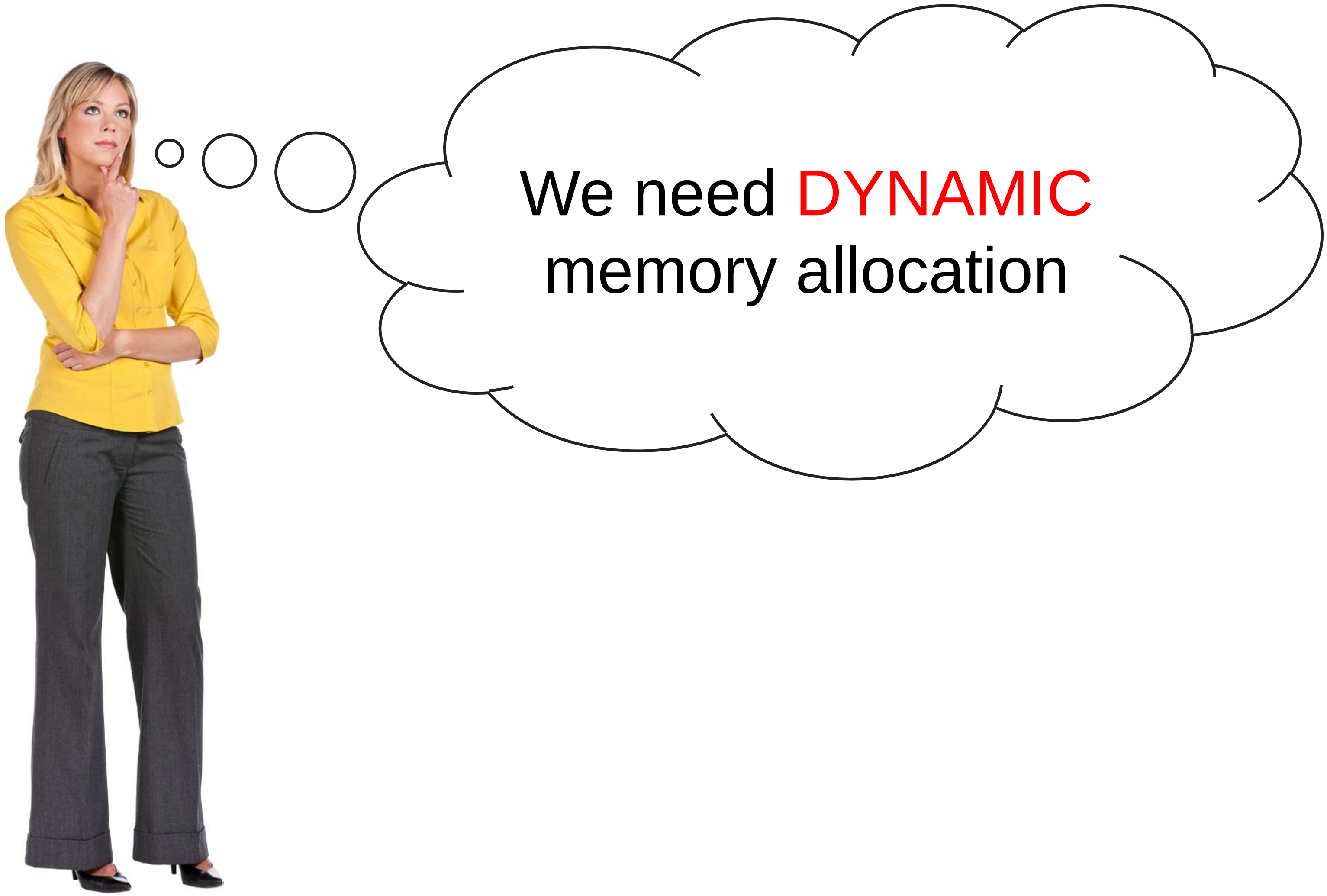
WHY DYNAMIC MEMORY ALLOCATION?

```c
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

MEMORY ALLOCATION

Compile time (STATIC) Allocation

Memory for named variables is allocated by the compiler.

The exact size and type of storage must be known at compile time.

Run time (DYNAMIC) Allocation

Memory allocated during run time is allocated space in a program segment known as the *heap* or the *free store*.

The exact amount of space or number of items does not have to be known by the compiler in advance.

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
```

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));
```

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

```
#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

```
/* 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.

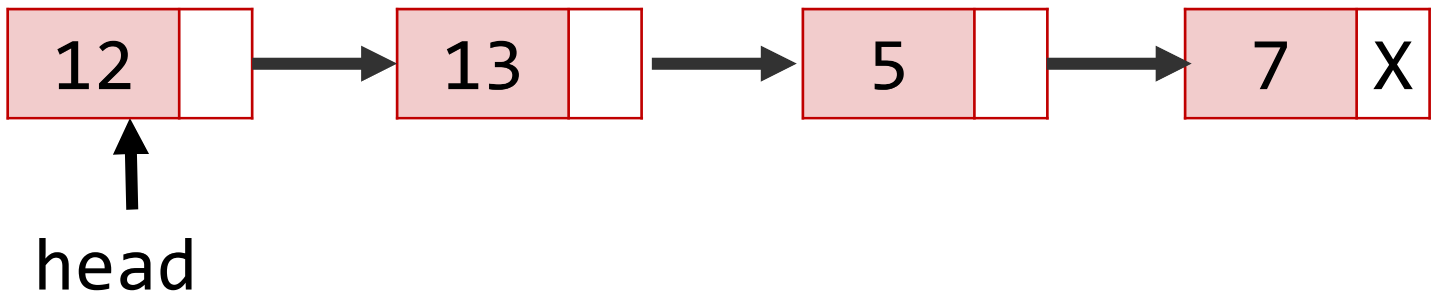
Allocate space for each string according to its length.

Invoke `free` for each allocated block of memory.

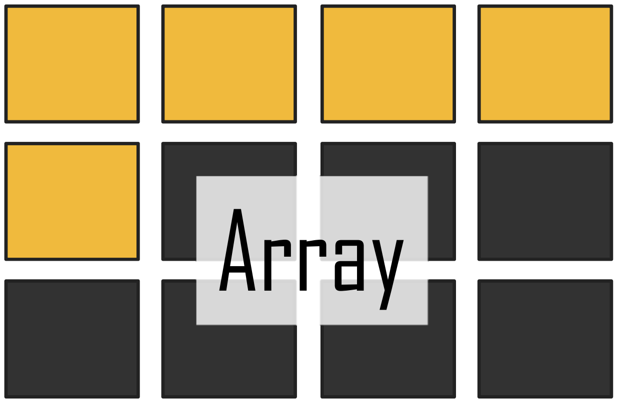
LINKED LISTS

SiT SINGAPORE INSTITUTE OF TECHNOLOGY

LINKED LISTS

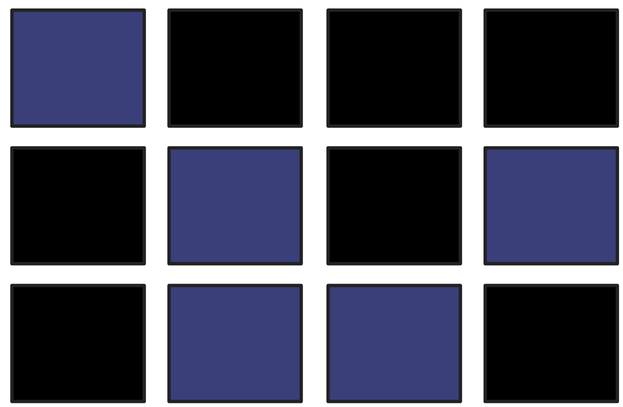


A linked list is a linear collection of **self-referential** structures, called **nodes**, connected by pointers, called **links**.



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

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



Linked list

SELF-REFERENTIAL STRUCTURES

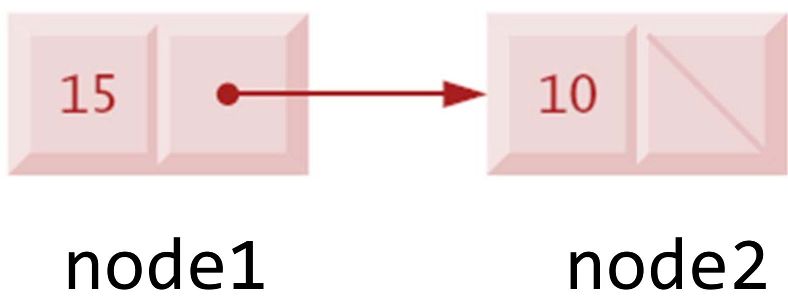
```
typedef struct node_struct {
    int data;
    struct node_struct *next;
} Node;
```

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

SELF-REFERENTIAL STRUCTURES

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?



CREATING A LINKED LIST

```
int main() {

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

}
```

node1 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);

}
```

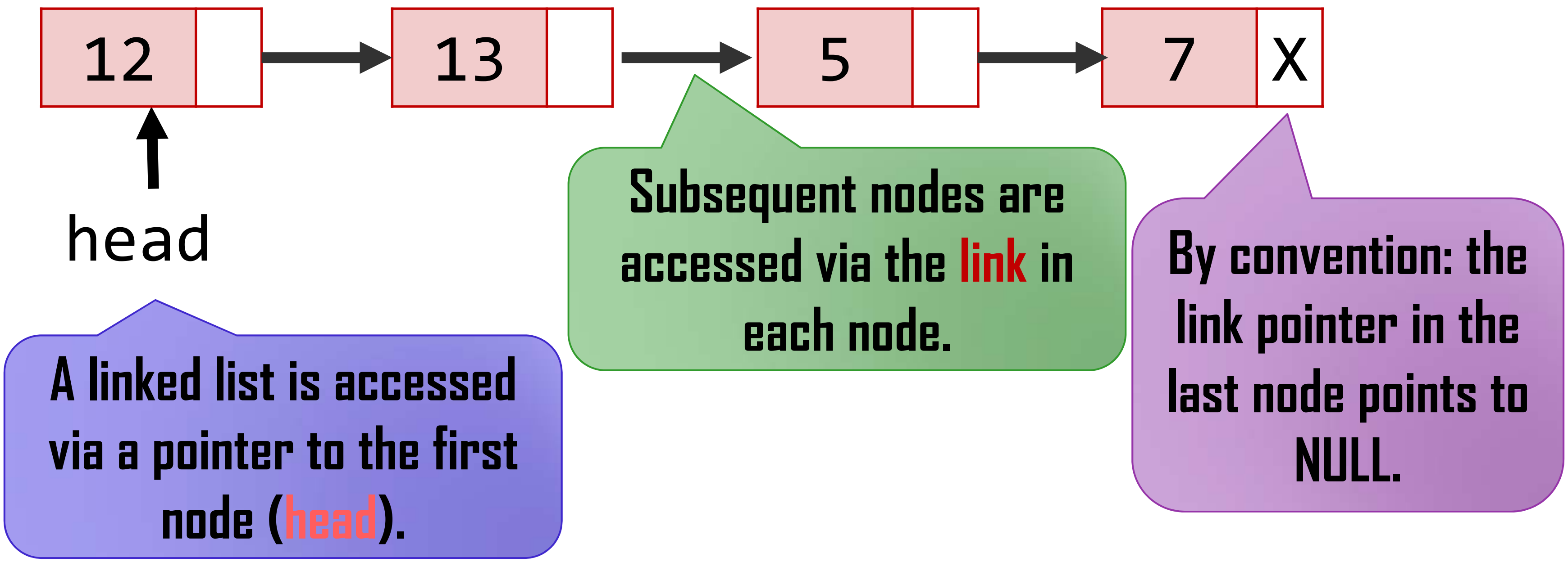
Use the dot "." notation for **non-pointer variables**

ACCESSING DATA IN A LINKED LIST

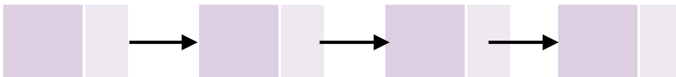
```
int main() {

        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



LINKED LIST



Dynamic

The length of the list can increase or decrease as necessary.

Cannot be full easily

A linked list becomes full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

GOOD when size is unpredictable

ARRAY

STATIC

The size of an array cannot be altered once memory is allocated.

Arrays can become **full**.

Arrays can be declared to contain **more** elements than the number of data items expected, but this can **waste memory**.

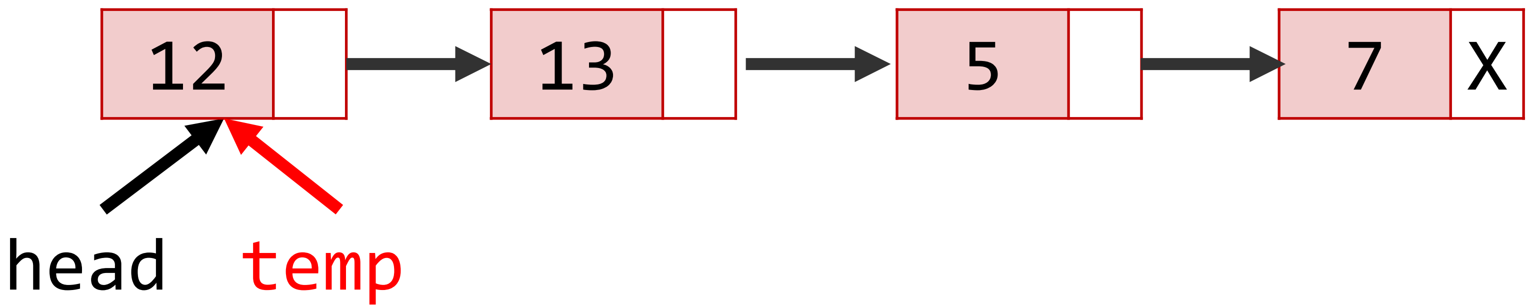
LINKED LIST OPERATIONS

Search/update

Insert

Delete

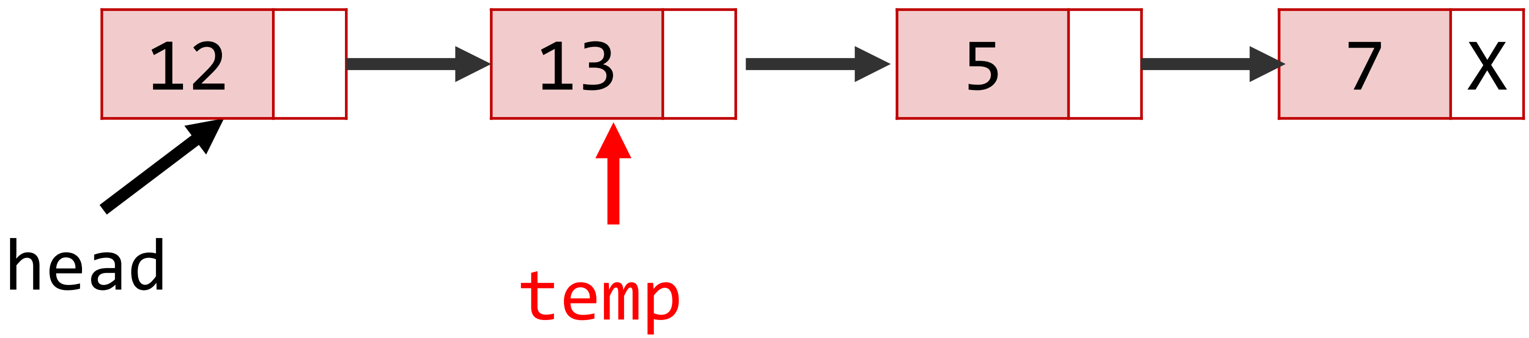
LINKED LIST OPERATIONS Search



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

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

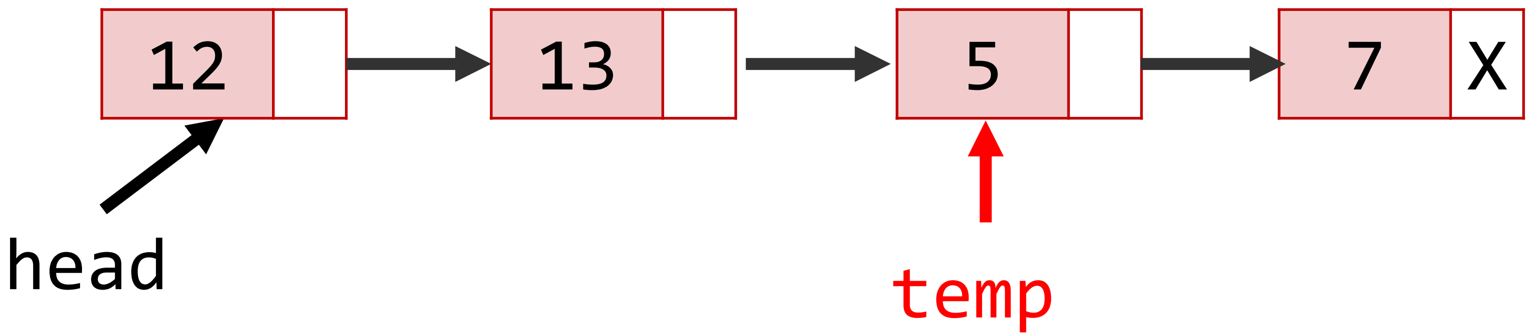
LINKED LIST OPERATIONS Search




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

Using the **next** pointer

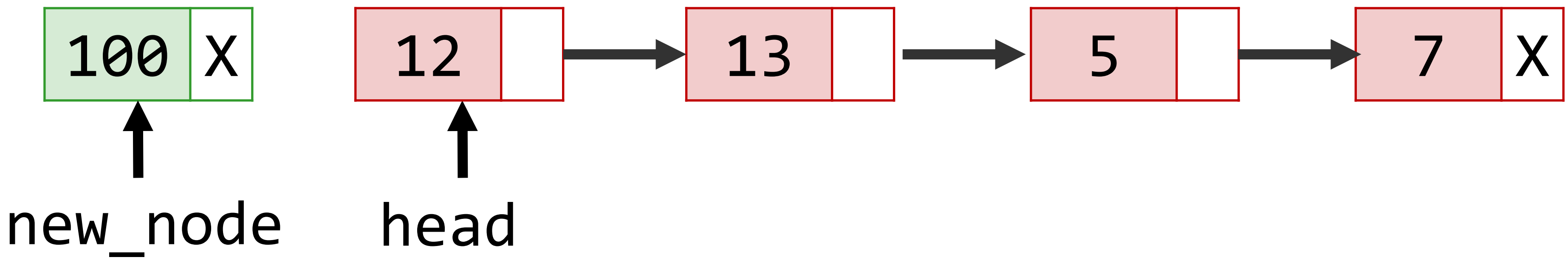
LINKED LIST OPERATIONS Search




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

Using the **next** pointer

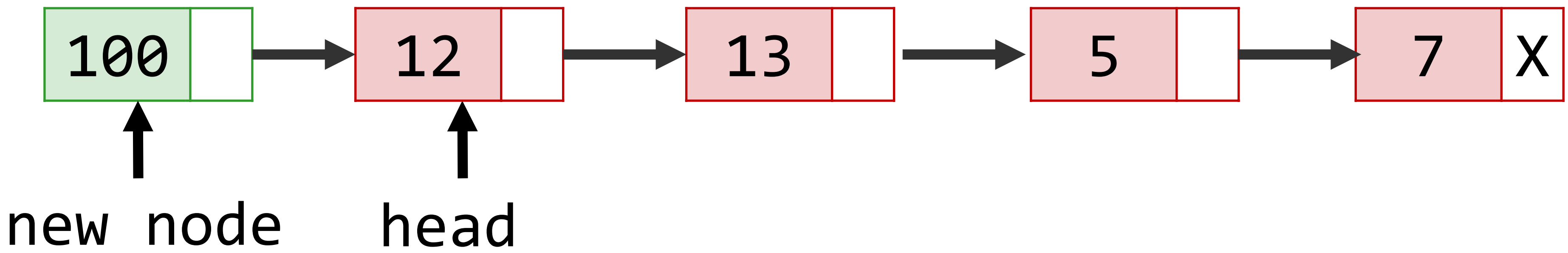
LINKED LIST OPERATIONS Insert




```
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?

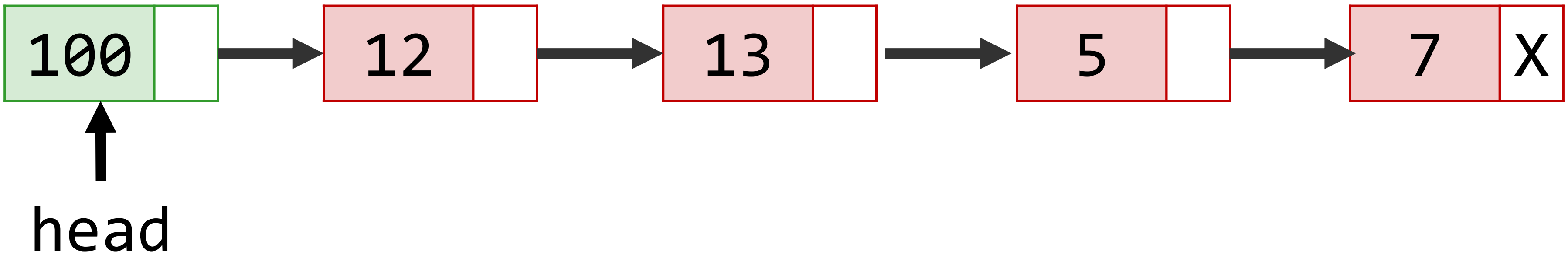
LINKED LIST OPERATIONS Insert




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

1. Link the new node to the old head.

LINKED LIST OPERATIONS Insert

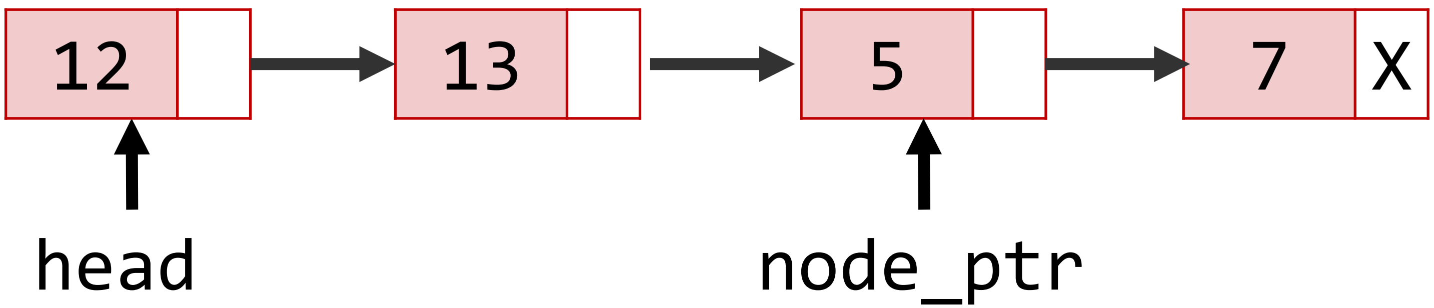



```
new_node->next = head;
head = new_node;
```

2. Move the head to the new node.

LINKED LIST OPERATIONS

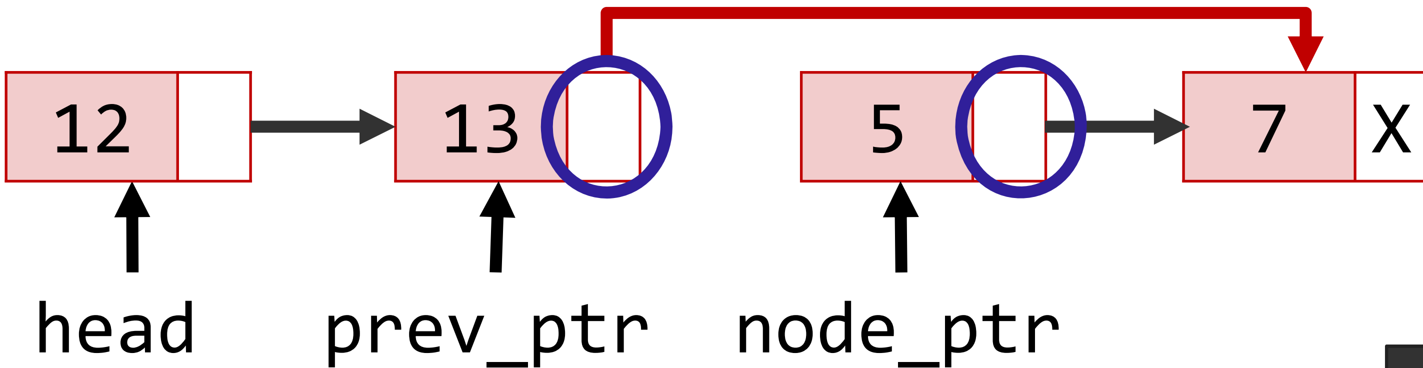
Delete



How do we delete the node pointed to by node_ptr?

LINKED LIST OPERATIONS

Delete



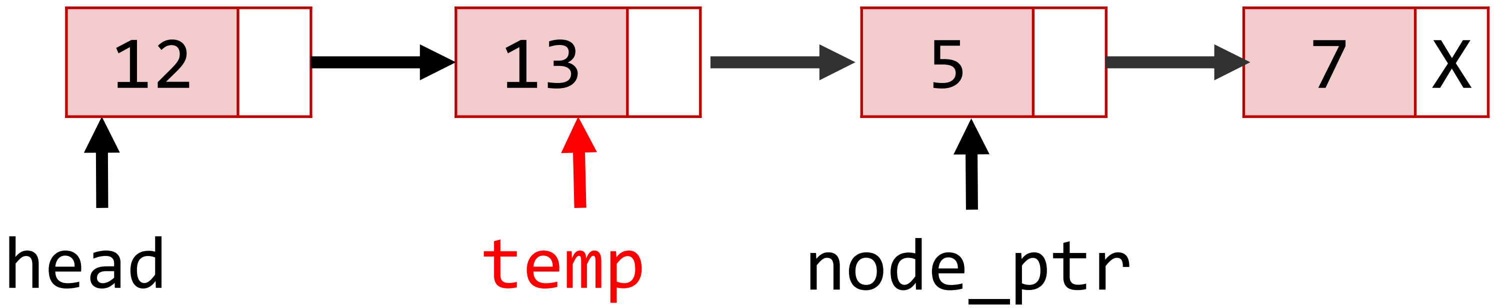

```
prev_ptr->next = node_ptr->next;
free(node_ptr);
```

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

How do we get prev_ptr?

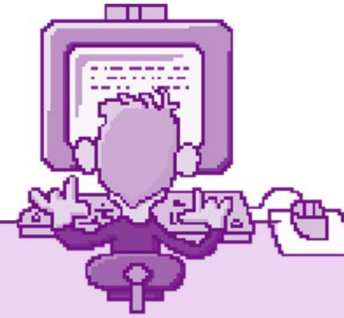
LINKED LIST OPERATIONS

Delete




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

EXERCISE - 1

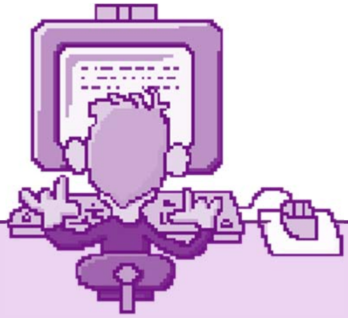


Write a function:

Node *search_list(Node *head, int target);

Given a pointer to the head of a linked list, return a pointer to the first node in the list whose data is equal to `target`.

EXERCISE - 2

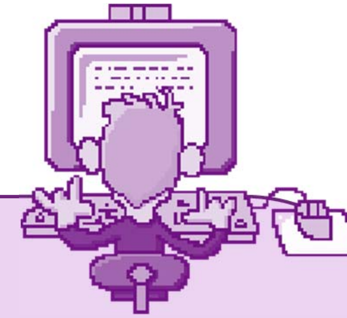


Write a function:

Node *insert_at_head(Node *head,
Node *new_node);

Given a linked list with its head pointer, insert the node pointed to by new_node to linked list and return the head pointer to the list.

EXERCISE - 3



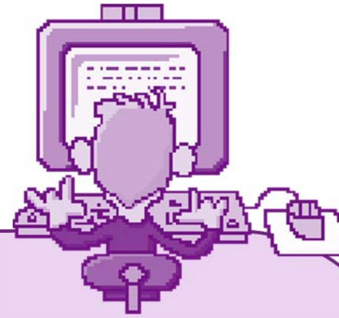
Write a function:

Node *delete_node(Node *head, Node *node_ptr);

Given a linked list with its head pointer, delete the node pointed to by node_ptr and return a pointer to the head node.

Why do we need to return a pointer to the head node?

EXERCISE - 4



Write functions that perform each of the following operations:

print every element in the list

insert an element at the end of list

deallocate the whole list

END-OF-WEEK CHECKLIST

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

Self-assessment (for practice only):
Socrative: https://b.socrative.com/login/student
Room: ICT1002