

# List ADT

CS2263 – Systems Software Development

1

## Learning Outcomes

At the conclusion of this lecture students should be able to:

- List the abstract required functionality of a list
- List and evaluate two ways of implementing a list as an ADT
- Create a linked-list as a typedef/struct in C
- Write the CRUD functions to use a linked-list data structure as a list.

2

## References

- Lu, Yung-Hsiang. 2015. Intermediate C Programming. CRC Press. New York. (Chapter 18)

3

## Note: Copy at Your Own Risk

- The code fragments in this presentation are meant to outline the processes that need to happen
- No guarantee it compiles
- Error checking has been left in the interests of clarity

4

## What is a List?

A collection of items

- What functionality is required?
  - Create the list
  - Read ("get") items in the list
  - Update
    - Add items
    - Remove items
  - Delete the list
- Does order matter? ► Ordered List
  - List with a modified Add functionality

5

What data structures can we use to implement a list?

- Array
- File
- Linked-list

Evaluation for each choice (implications for CRUD):

- Array
- File
- Linked-list

## List: Implementation Choices

6

## Why Linked Lists



- Linked Lists
- Modified at runtime Arrays
  - Overhead?
  - Complexity
- Defined at runtime Arrays
- Compile-time Arrays

7

## Linked Lists in Java

- What do you remember about Linked Lists from CS1083?
  - What are the parts?
  - How do they work?

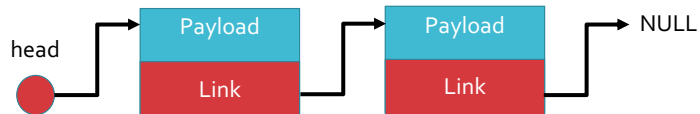
8

## Declaring a Linked List in C

- Yes!

```
typedef struct link{
    Payload* payload;
    struct link* next;
} <Name>
```

- What's the payload Kenneth? (REM shout out)
- Link
- Head
- Tail?
- Iterator?



9

## Operations on Linked List Elements

- **C**reate
  - A pointer to the list (head)
  - or the creation of a List typedef/struct to hold (ADT e.g. head, tail)
- **R**ead
  - Process the list
- **U**pdate
  - Add items
  - Remove items
- **D**elete
  - Remove the items
  - Remove the List

10

## Create a Linked-List

### As a linked-list

```
Link* pHead = (Link* NULL);
```

### As an ADT

```
List* pList = mallocList();
-----
List* mallocList(){
    allocate the structure
    set the head to NULL
    (set the tail to NULL)
}
```

11

## Read a Linked-List

### As a linked-list

```
Link* pWorking = pHead;
while(pWorking != (Link*) NULL){
    /* your processing here (e.g. printf() )*/
    pWorking = pWorking->next;
}
```

### As an ADT

- same as above, except within a function that takes `pList` as an argument, e.g.

```
int printPointsList(List* pList, FILE* pFOut);
```

12

## Update a Linked-List: Add

### As a linked-list

```
Payload* pPayload = createPayload(/* arguments */);
Link* p = (Link*)malloc(sizeof(Link) );
p->next = (Link*) NULL;
-----
p->next = pHead;
pHead = p;
```

### As an ADT

- same as above, except within a function that takes pList as an argument, e.g.

```
int addPointList(List* pList, Point* pPtThis);
```

13

## Update a Linked-List: Remove

### As a linked-list

```
Is it the first in the list?
    set pTemp to pHead
    set pHead to pHead->next;
    free pTemp // implementation specific
else
    traverse the list until pWorking->next meets the criteria
        set pTemp to pWorking->next
        set pWorking->next to pWorking->next->next
        freePayload(pTemp) // implementation specific
```

### As an ADT

- same as above, except within a function that takes pList as an argument, e.g.

```
int removePointList(List* pList, <match condition>);
```

- Of course the Payload's free() should be called

14

## Delete a Linked-List

### As a linked-list

```
Link* pWorking = pHead;
Link* pTemp;
while(pWorking != (Link*) NULL){
    pTemp = pWorking->next;
    //free link/payload
    pWorking = pTemp;
}
```

### As an ADT

- same as above, except within a function that takes `pList` as an argument, e.g.  

```
int freeList(List* pList);
```
- Of course the Payload's `free()` should be called