Abstract Data Types

Objectives

- Abstract data types
- Linked lists
- Linked list functions

Abstract Data Types

Stacks Review

- Stacks
  - A stack is an ordered collection of items
- LIFO
  - Items are inserted at the top
    - Pushed
  - And removed from the top
    - Popped

Stack Description

- The definition of a stack was independent from its implementation
  - An example of an *abstract data type* (ADT)
- An abstract data type is a collection of data and operations on that data
  - An ADT describes **what** operations and data are allowed
  - But **not how** they are implemented

Abstract Data Types

- An ADT is a collection of data and a set of allowed operations on that data
  - Does not specify how the data are stored or how the operations are performed
  - The definition focuses on the use of the ADT
- A data structure specifies the implemenation
  - And particular data structures are often used to implement an ADT
  - The two interact via an *interface*

Queue

- A queue is another example of an ADT
  - Behaves like a line-up
    - Or, in Britain, a queue
- FIFO
  - Items are inserted at the back of the queue
    - Enqueued
  - And removed from the front of the queue
    - Dequeued

Interfaces

- An interface refers to a collection of data and expected behaviours
  - Specifies inputs and outputs
  - Serves as a contract
- Interfaces we have seen in CMPT 125 or 127
  - Functions pre and post conditions and invariants
  - Collections of functions
  - Header files

Why Use Interfaces?

- There are advantages in using an interface
  - Code re-use
  - Code independence
  - Modularity
- Interfaces specify what data and operations are required
  - Without implementation details
  - The same interface could be implemented in widely different ways

Software Engineering Principles

- Encapsulation
  - Bundle related data and operations together
- Modularity
  - Break up problems into smaller, more manageable, programming tasks
- Information hiding
  - Keep implementation details private
  - Keep the interface stable
- Finding a good selection of interfaces is one of the foundations of writing large scale software

Interface Examples

- Stack
  - Sequence of data
  - LIFO
  - insert (push)
  - remove (pop)
  - isEmpty
  - peek
  - size

  not required

- Appendable array
  - Sequence of data
  - append (add to end)
  - size
  - access (get)
  - change (set)

Appendable Array ADT

- Array implementation – variables
  - Keep track of current size
  - Keep a pointer to the array
- Array implementation – operations
  - Access – index look up and bounds check
  - Change – index look up, set value and bounds check
  - Append – need to malloc a new array and copy the contents of the old array to it
    - Running time?

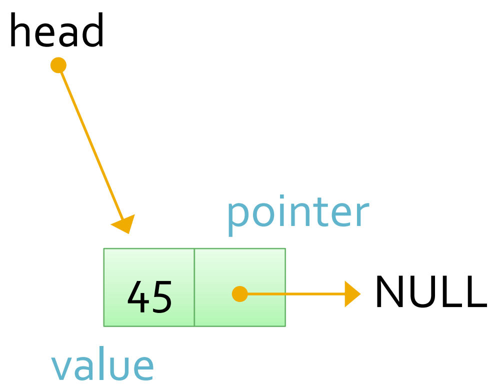
Appendable Array ADT

- Linked list implementation
  - Items consist of a value and a pointer to the next item
  - Keep track of the head (front) and tail (back) of the list
  - When an element is appended add it to the list's tail
  - The tail's next pointer is set to NULL to indicate that it does not point to anything
  - Running time of operations?

Linked Lists

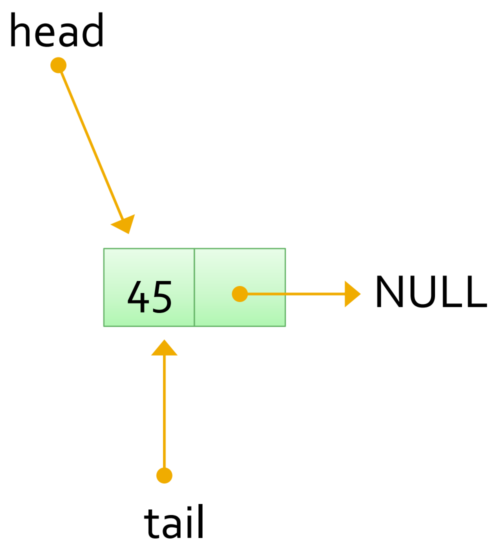
Linked Lists

- Linked list items
  - Consist of pairs
    - A value (the list data)
    - A pointer to the next item in the list
    - Together they form a *node*
  - The pointer is initially set to null
  - The list requires a pointer to the first node
    - The head of the list



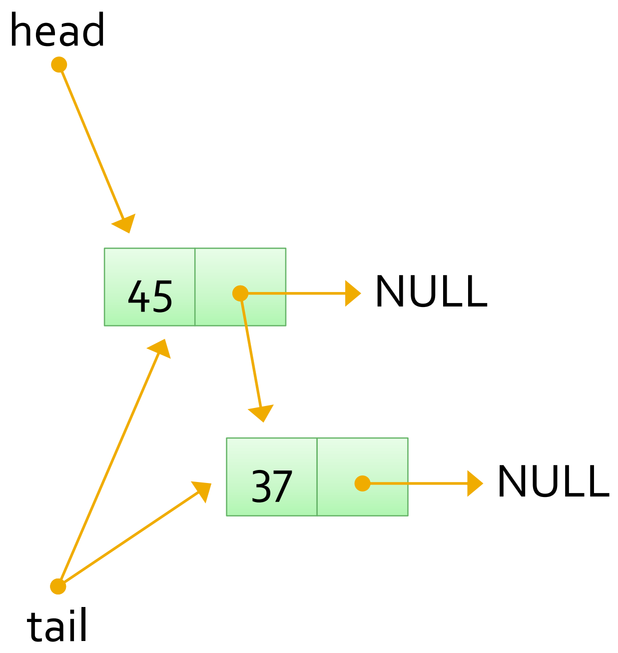
Appendable Lists

- Appendable list structure
  - An appendable list adds new values to the end
- We could find the end by traversing the list
  - It is much more efficient to keep track of the end
    - With another pointer to the tail of the list
    - Initially the same as the head



Appending Values

- To append a value
  - Create a new value : pointer pair in dynamic memory
    - In C, use malloc
  - Assign its address to the pointer of the tail node
  - Then make the tail pointer point to the new node
    - Because it is the new value at the end of the list



Packaging Nodes

- It is convenient to create a datatype to represent a node
  - So that nodes can be passed to functions
  - And created in dynamic memory as a single unit
- In C this is achieved by defining a node as a structure
  - Using the keyword *struct*
- Structure declarations must be preceded by the keyword struct
  - `struct node x1;` and not `node x1;`
  - So use *typedef* to name the structure and avoid this

Building A Linked List

- A node is not the same as a linked list
  - It is just a single link
- To create a linked list
  - Write a struct for the linked list
  - And functions to insert, remove and query the list
- The linked list struct contains
  - A node for the head of the list
  - And, for an appendable list, a node for the tail

Interface and Implementation

- It is good practice to separate the interface and the implementation
  - By putting each in its own file
    - The interface consists of a .h file
    - The implementation consists of a .c file
- The .h file contains
  - The struct definitions
  - Function prototypes
- The .c file contains
  - Function definitions

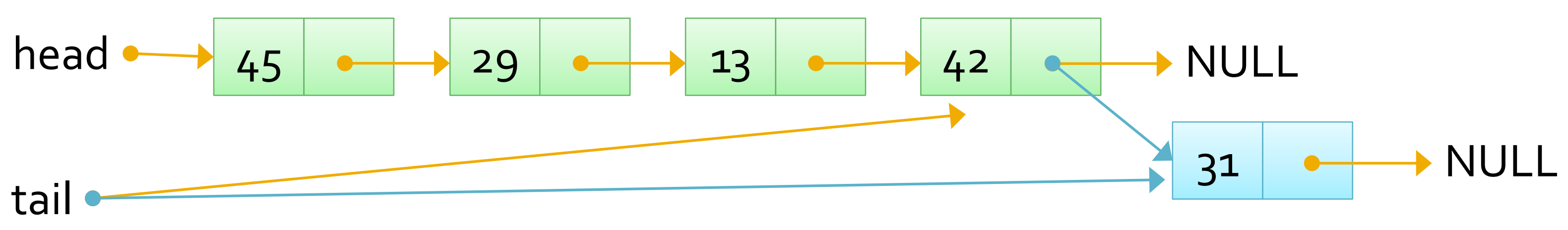
Compiling Multiple Files

- Assume that a project consists of
  - `node.h` – contains the definition of a node struct
  - `LL.h` – contains the definition of a linked list struct and function prototypes
  - `LL.c` – contains the definition of the linked list functions listed in `LL.h`
  - `lists.c` – contains a main function that tests linked lists
- To compile the project
  - `gcc -o lists lists.c LL.c`

Linked List Functions

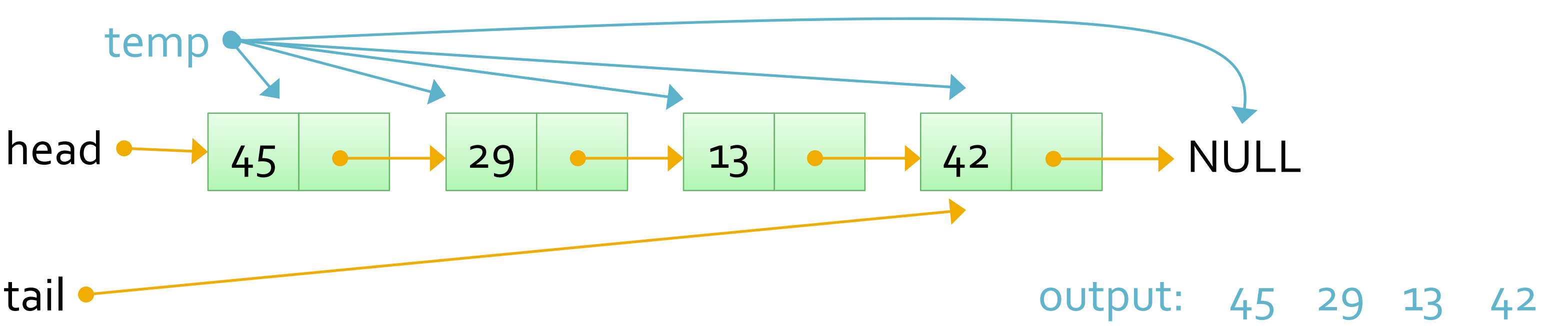
Append

- There are two major steps
  - Allocate space for the new node using malloc
    - Assign its address to the tail node's next pointer
  - Correctly maintain head and tail
    - Head doesn't change and tail points to the new node
  - But don't just consider the typical case
    - What happens when the list is empty?



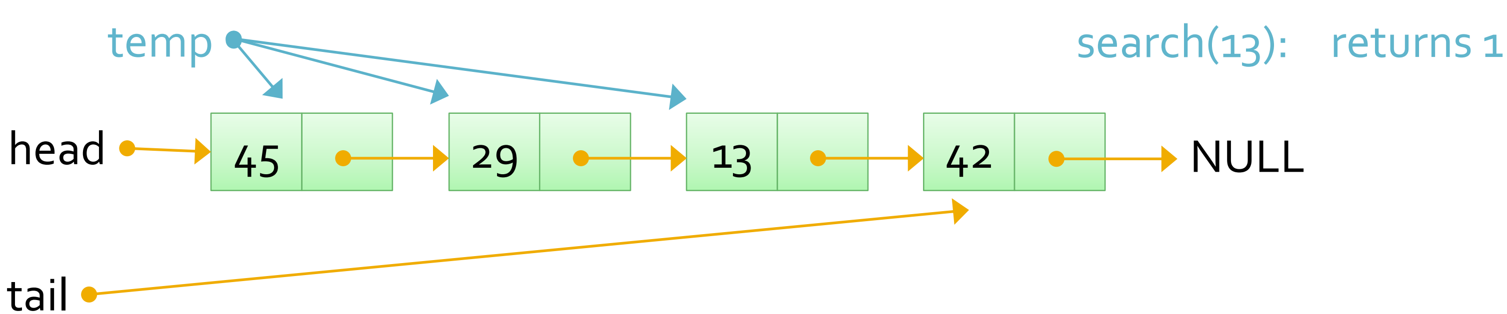
Print

- Print should output the values in the list
  - In order from *head* to *tail*
  - Dereference all the pointers in the list
    - head, then *head->next*, then *head->next->next*, …
  - Stop when *next* is NULL
    - Use a while loop



Search

- Very similar to print
  - Traverses the list from head to tail
  - Except that it returns 1 if a node's value is the same as the target
  - Linear search
    - On a linked list rather than an array
    - Running time?



Other Linked List Functions

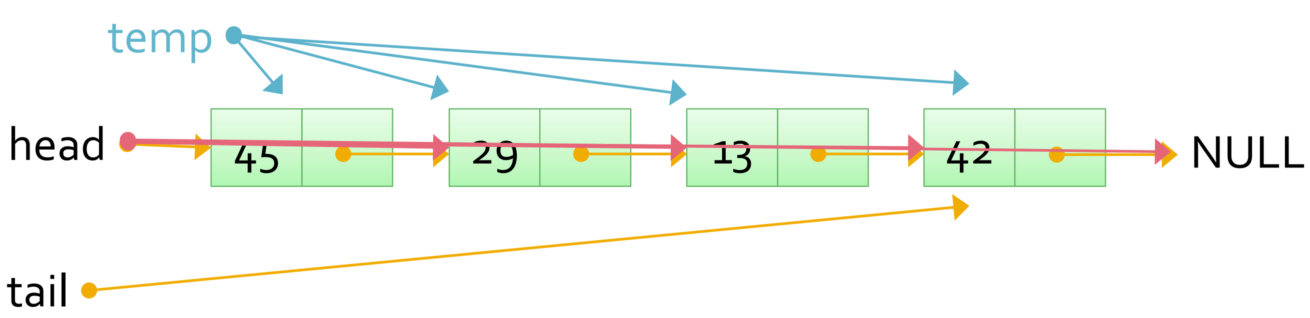
- There are many functions we could imagine writing for linked lists
  - Destroying the list
  - Concatenating two lists
  - Inserting elements at the front of the list
  - Inserting elements next to some other element
  - Sorting the list
  - …
- An important design issue for an ADT is that it should not have more operations than are specified in its description
  - A stack that allows insertions anywhere is not a stack

Linked List Edge Cases

- If a function modifies a linked list then design for the typical case and consider
  - What happens if the list is empty?
  - What happens if the list consists of a single item?
  - When should the head change?
  - When should the tail change?
    - If there is a tail

Destroying a List

- Once a list is no longer needed its memory should be deallocated
  - Using *free*
  - There is a timing issue here
    - You can't look at the next pointer once you've destroyed its node
  - So set *temp* to *head* and set *head* to point to the next node
    - And stop when *head* is NULL



Linked Lists and Recursion

Recursive Definition

- A linked list can be defined recursively
  - It is composed of its first node
  - And another, slightly smaller, list
- LISP (**Lis**t **P**rocessing) is a programming language based around lists
  - The *car* operator refers to the first node
  - The *cdr* operator refers to the rest of the list

Recursive List Functions

- Many linked list functions can be written recursively
  - Base case – stop if node is NULL
  - Do something with the current node
  - Call the function recursively on the rest of the list
- Write a function to print the contents of a list in reverse order
  - Try writing this iteratively
  - And then recursively