FoA Revision Workshop

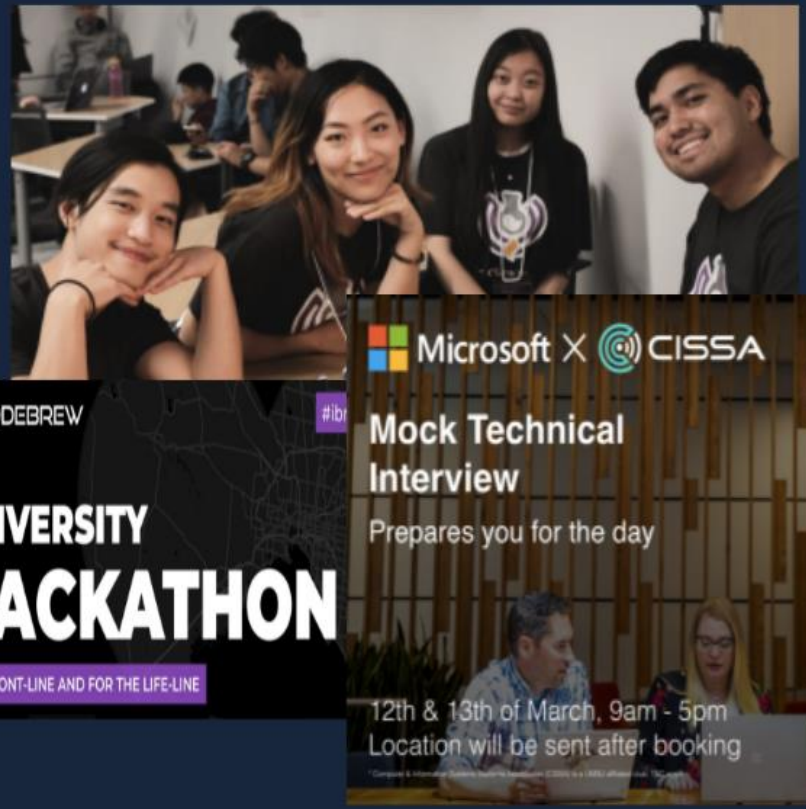
Stella Li

Sign up for CISSA!

cissa.org.au/signup

CISSA

- Largest Tech Club in UoM.
- Representative of Computer Science / Data Science students in both undergrad and postgrad.
- Joint events and collaborations with…
  - Tech Companies (Google, Microsoft, Amazon, Canva etc.)
  - Consulting Firms (Deloitte)
  - Trading Companies (Optiver, Jane Street, IMC etc.)
  - Researchers in UoM
- Social events for you to make more friends in the tech community!



Join Our Club (it's FREE)!

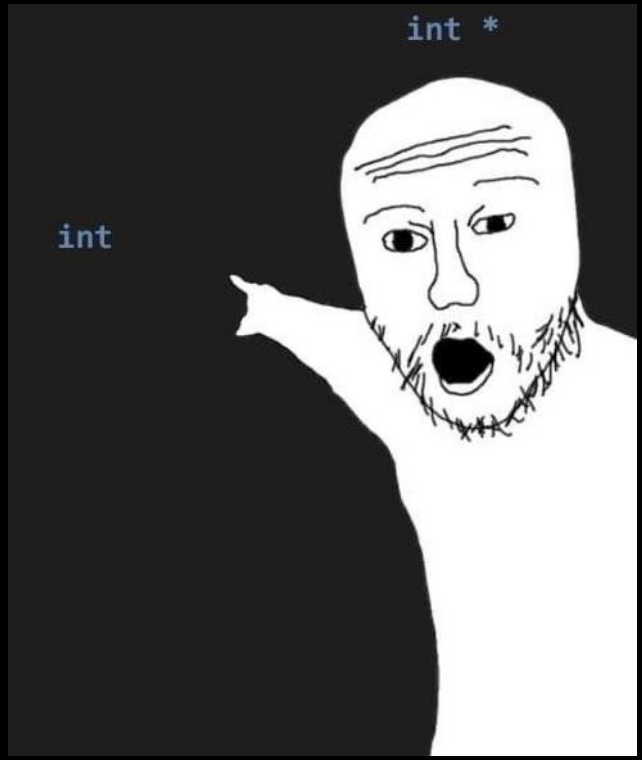
- Access our Discussion Space on Facebook / Discord to connect with over 4000 people in the UoM tech community.
- Ask questions re. Subjects, Careers, Interviews etc.
- Events:
  - Industry Connect & Tech talks
  - Mock Interviews + Office Tours
  - Hackathons - Codebrew + Catalyst
  - Start-Up & Entrepreneurship Competitions
  - Subject Revision + Interview Workshops

Sign-Up!

cissa.org.au/signup

- Memory & Pointers
- Dynamic Memory Allocation
- Linked Lists, Stacks & Queues
- Binary Search & BST
- Practice Questions

Memory & Pointers



Variables & Functions

```c
int main(int argc,  char* argv[]) {
    int n = 5;
    my_function(n);
    n += 2;      // <-- point 2
    printf("Point 2: %d\n", n);
    return 0;
}

void my_function(int n) {
    n += 3;      // <-- point 1
    printf("Point 1: %d\n", n);
}
```

← Assigns space in memory for n and set it to 5

← Passing the value of n into a function

← Makes a copy of n

← Operates on a copy of n

```c
int main(int argc,  char* argv[]) {
    int n = 5;
    my_function(n);
    n += 2;      // <-- point 2
    printf("Point 2: %d\n", n);
    return 0;
}

void my_function(int n) {
    n += 3;      // <-- point 1
    printf("Point 1: %d\n", n);
}
```

Changing the Original n

- Use pointers!
- Pointers are just memory addresses, they “point” to a piece of memory

```c
int main(int argc,  char* argv[]) {
    int n = 5;
    my_function(&n);
    n += 2;      // <-- point 2
    printf("Point 2: %d\n", n);
    return 0;
}

void my_function(int *n) {
    *n += 3;      // <-- point 1
    printf("Point 1: %d\n", *n);
}
```

← Passing address of n rather than the value

← Modified to use pointers

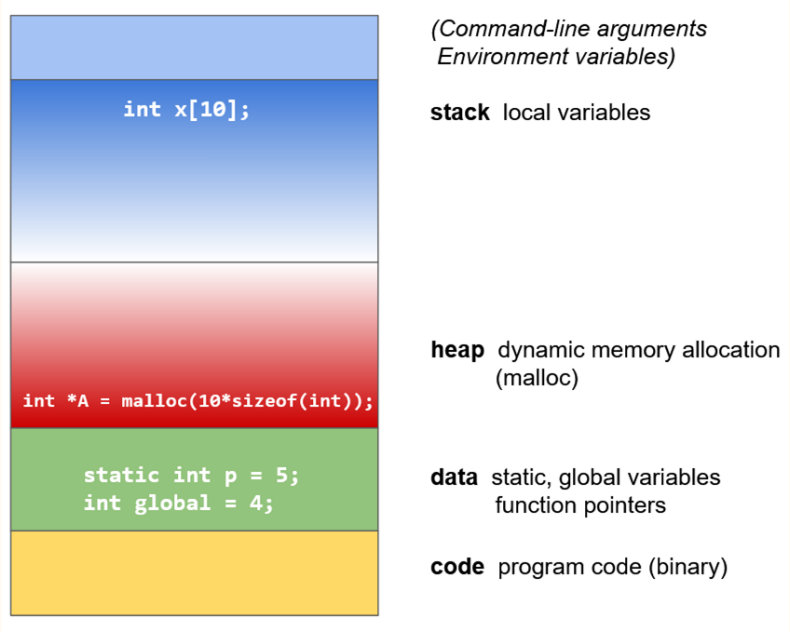
```c
int main(int argc,  char* argv[]) {
    int n = 5;
    my_function(&n);
    n += 2;      // <-- point 2
    printf("Point 2: %d\n", n);
    return 0;
}

void my_function(int *n) {
    *n += 3;      // <-- point 1
    printf("Point 1: %d\n", *n);
}
```

Memory Addresses

- Memory addresses are usually represented in hexadecimal and are all the same size (8 bytes usually)
  - `void *p;`
  - `int *p;`
  - `restaurant_t *p;`
- All data is stored in binary in the memory and your code determines how to read these bits
  - C won't stop you from interpreting an `int*` as a `restaurant_t*`
  - Segfaults, or no warnings at all...

Dynamic Memory Allocation

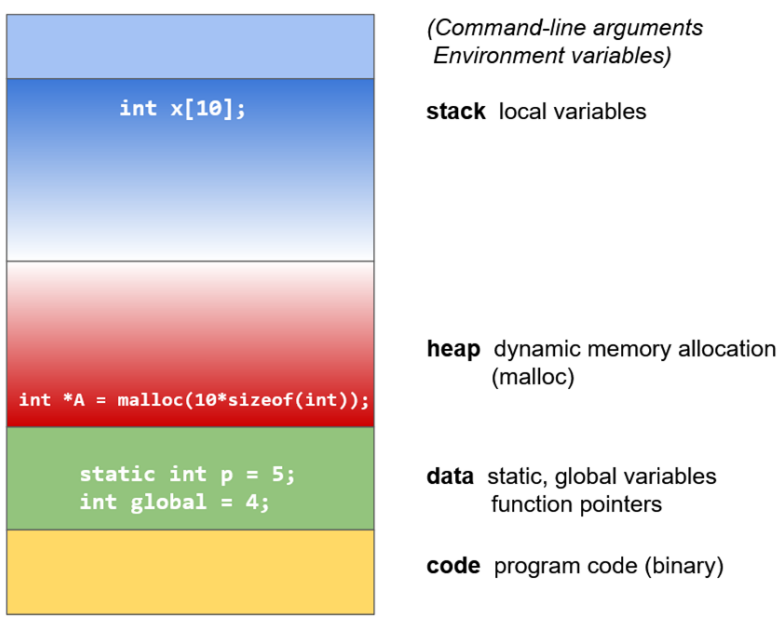
Memory for a program




```c
int *function() {
    int x[10];
    return x;
}
```

- Variables declared this way (static memory allocation) is stored in the stack
- Each function has their own slice of the stack
- When a function finishes running, its stack is freed
- Access *x after the function finished running = Accessing memory that no longer belong to you, Seg fault

Memory for a program




```c
int *function() {
    int *x = malloc(10 * sizeof(int));
    return x;
}
```

- Dynamically allocated memory is stored in the heap
- The heap is shared between all functions
- When a function finishes running, the memory it allocated in the heap is not freed
- Can still access *x after the function finishes running

Functions for Dynamically Allocating Memory

```
void *malloc(size_t size);                                          //
important

void free(void *ptr);
        // important

void *realloc(void* ptr, size_t new_size);          // not as often used

void *calloc(size_t nitem, size_t size);            // not as often used
```

- Every malloc must have a corresponding free.
  - Too few: memory leak
  - Too many: seg fault
  - Set pointer to `NULL` after freeing to prevent double freeing

Don’t know the size of input?

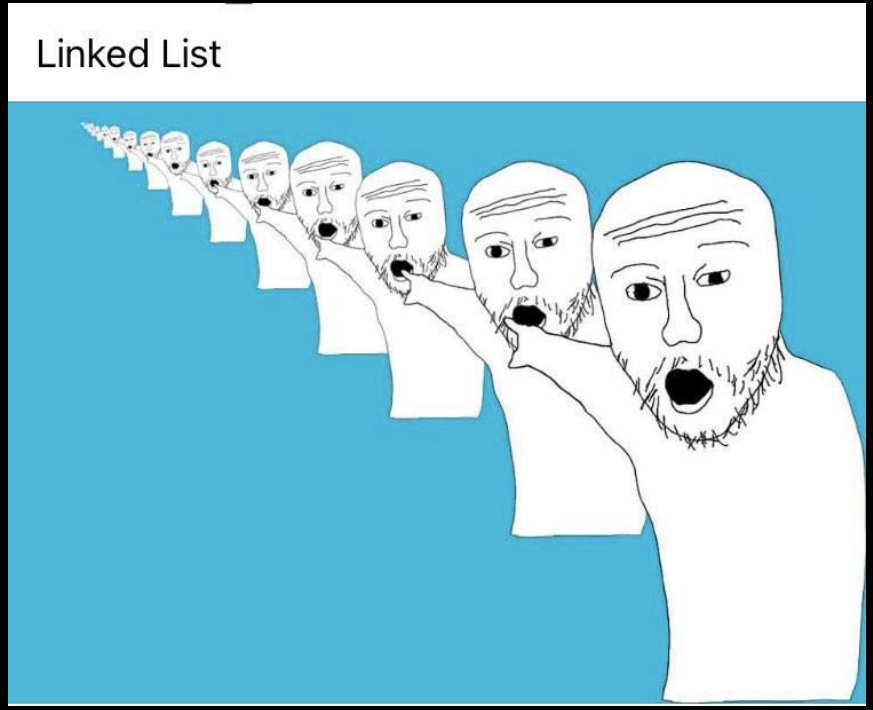
- Use realloc to move data to a larger piece of memory
- Eg. a program that takes an unknown number of integer inputs and stores them in an array

```c
int size = 5;
int *arr = malloc(size * sizeof(int));

int buffer, nitems=0;

while (scanf("%d", buffer) == 1) {
    if (nitems > size) {
        size *= 2;
        arr = realloc(arr, size);
    }

    arr[nitems] = buffer;
    nitems++;
}
```

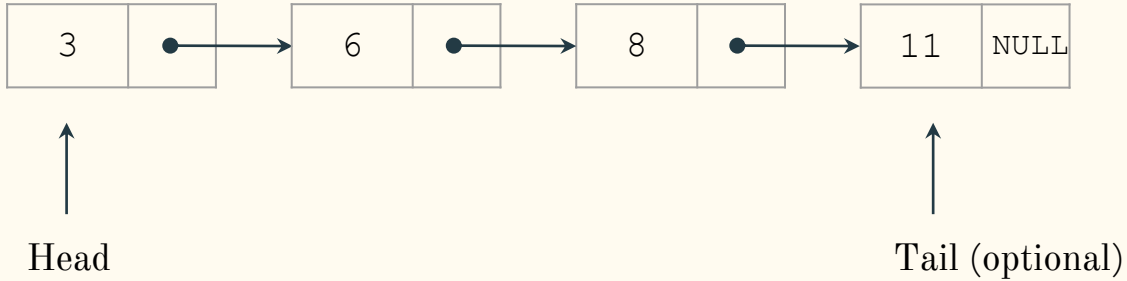
Cons of realloc

- O(n) to reallocate memory
- Could have a lot of unused memory

Linked Lists, Stacks & Queues

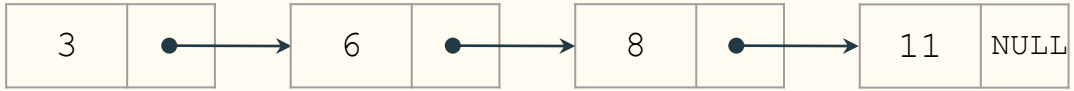


Linked Lists

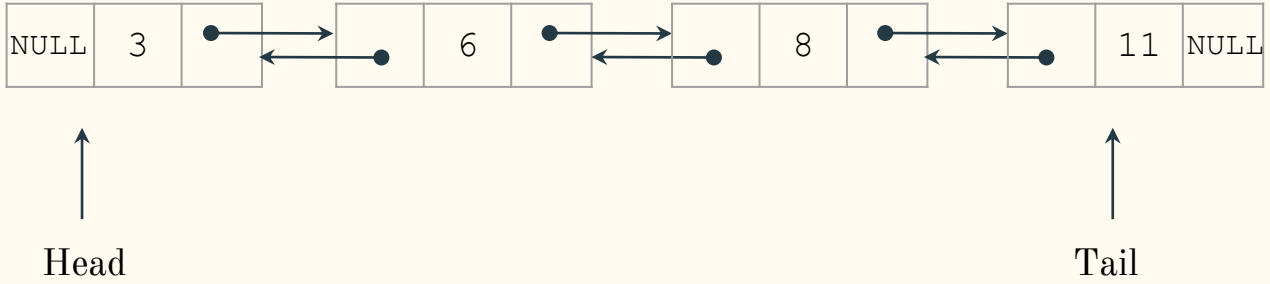


- Collection of nodes containing data
- Each node contains a pointer to the next node

Linked List Operations



Doubly Linked List



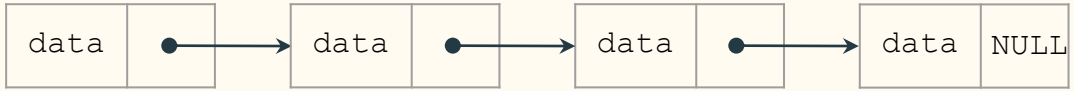
Abstract Data Types

- Concrete data types are actual implementations of data types
  - Eg. int, char, arrays, linked lists
- Abstract data types are models of possible data types, can often be implemented in many ways
  - Eg. Lists, stacks, queues, graphs

Stacks

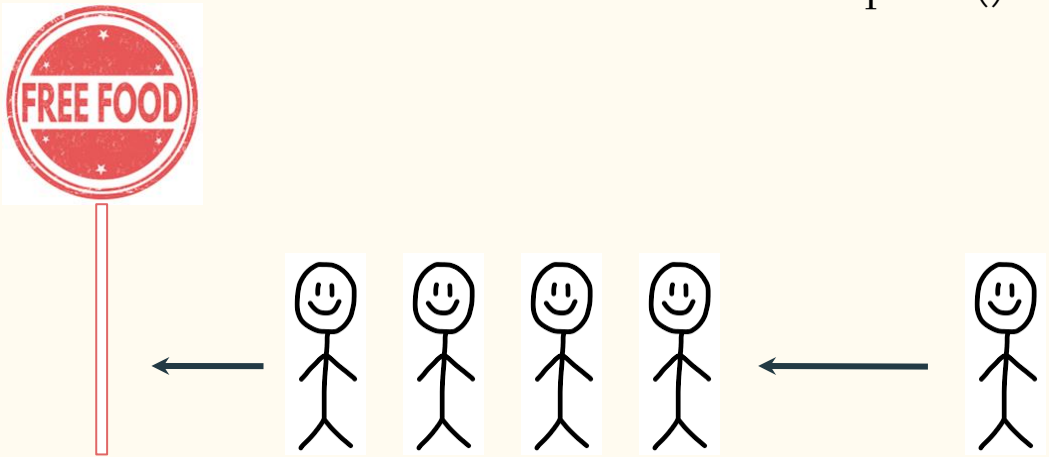
- Last in first out (LIFO)
- Push() to add new item
- Pop() to remove newest item

Linked Lists as Stacks



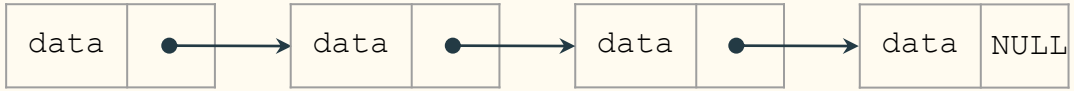
Queue

- First in first out (FIFO)
- Enqueue() to add new item
- Dequeue() to remove oldest item



Uni Students

Linked Lists as Queues



Binary Search & BST

Searching

2	3	5	6	9	15	22	23

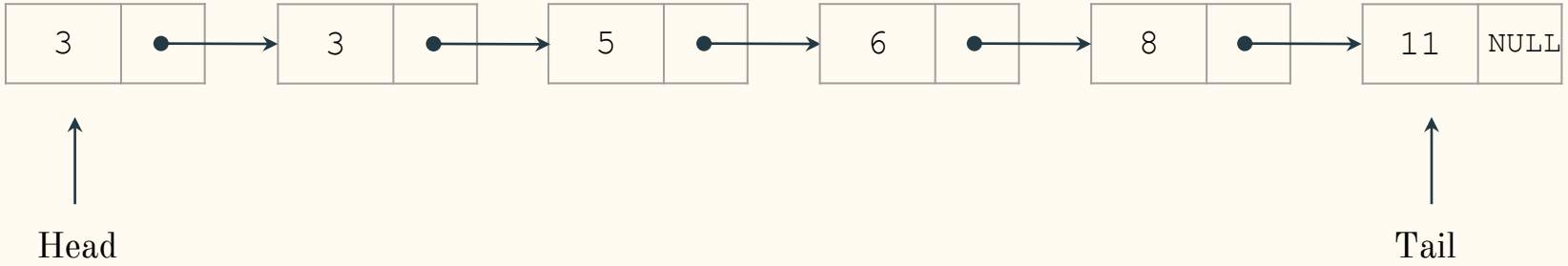
Binary Search

2	3	5	6	9	15	22	23

Array need to be sorted!


Binary Search
Sorted Array

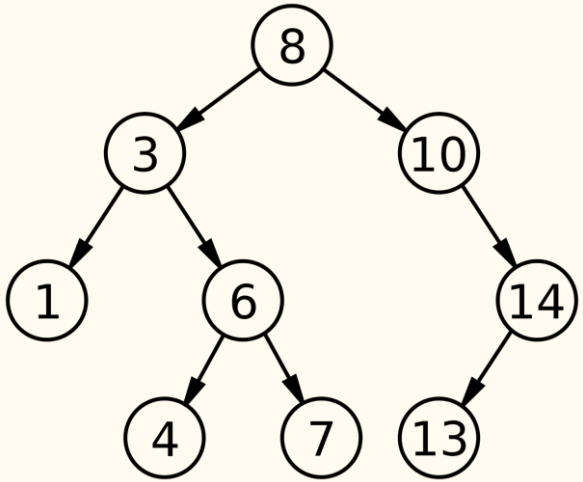
How to Run Binary Search on a Linked List



You can’t.

Binary Search Trees (BST)

- Structure that keeps growing with unknown number of inputs without needing to realloc
- Each node consists of some data and pointers to 2 child nodes
  - Left node is smaller than parent
  - Right node is larger than parent



Inserting into a Binary Tree

```
{8, 3, 6, 10, 9, 1}
```

Inserting into a Binary Tree - Worst Case

{2, 5, 6, 8, 12, 13}

Algorithms are Fun!!!

cissa.org.au/signup