Linked lists

Lecture 03.05

Linked lists

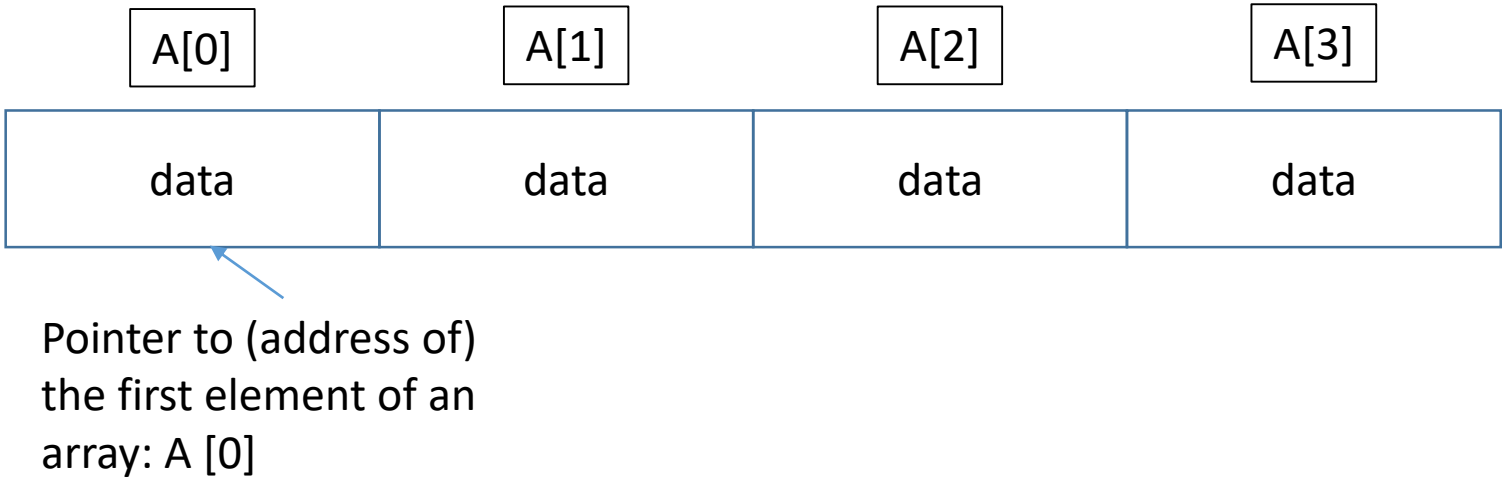
- Linked lists - recursive structs linked through pointers
- Motivation: flexible storage

Outline

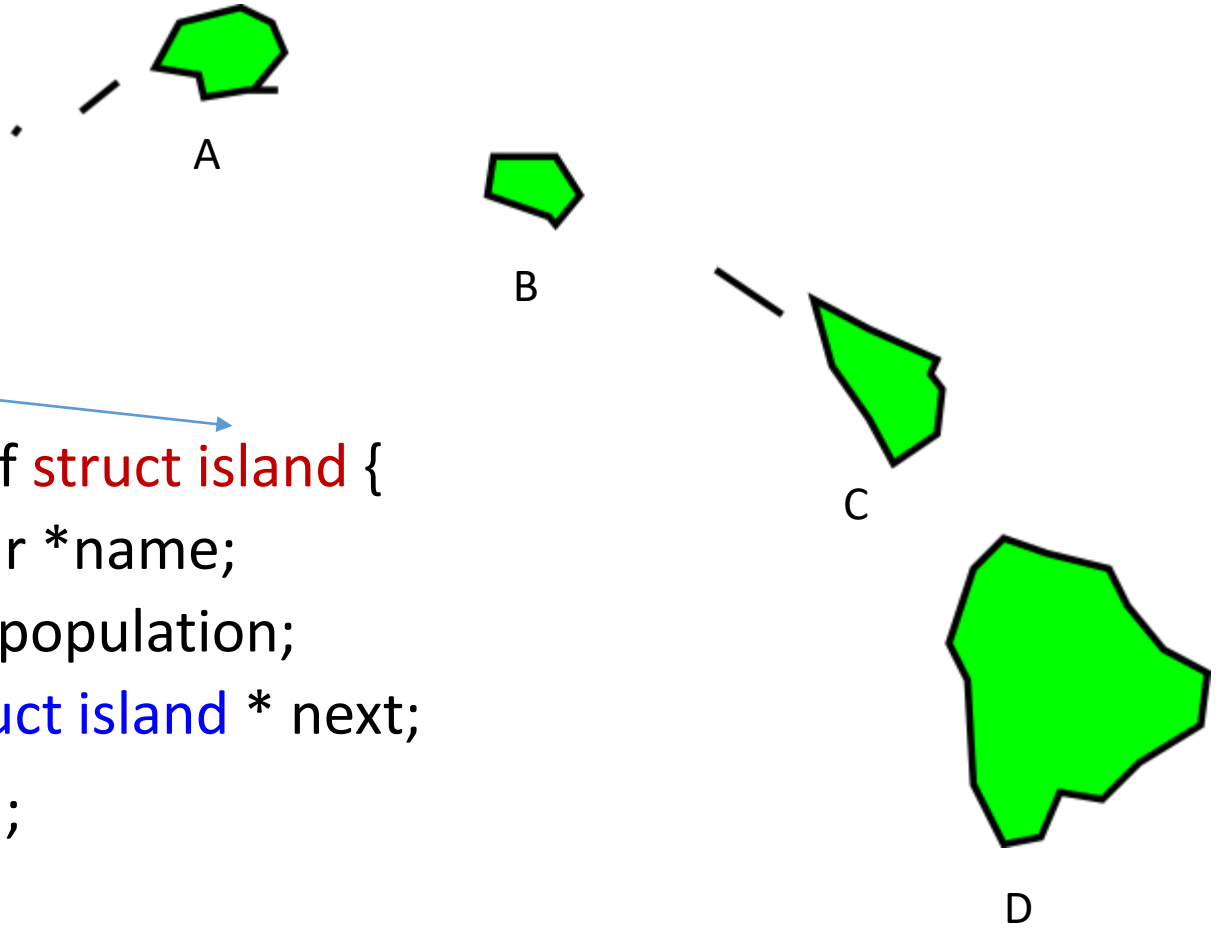
- Navigating the list
- Dynamic allocation
- Inserting new nodes
- Detecting memory leaks

Storing sequence in order: array

```
typedef struct island {
    char *name;
    int population;
} Island;
```



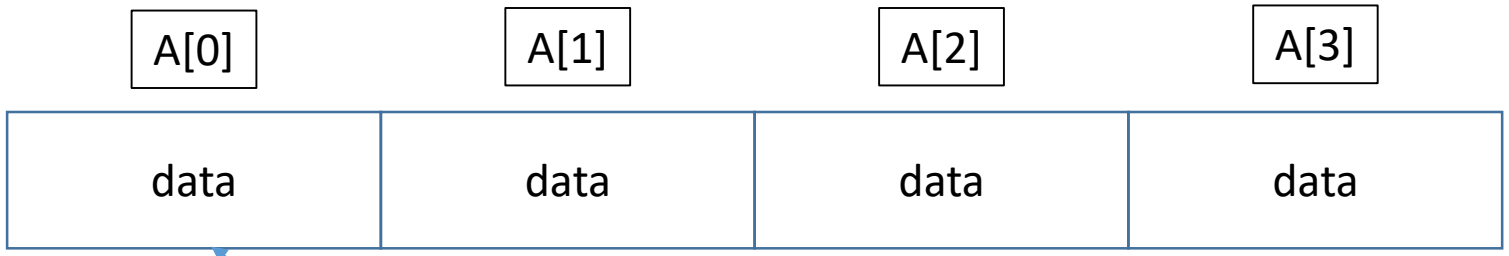
Recursive struct



Have to use name, not alias

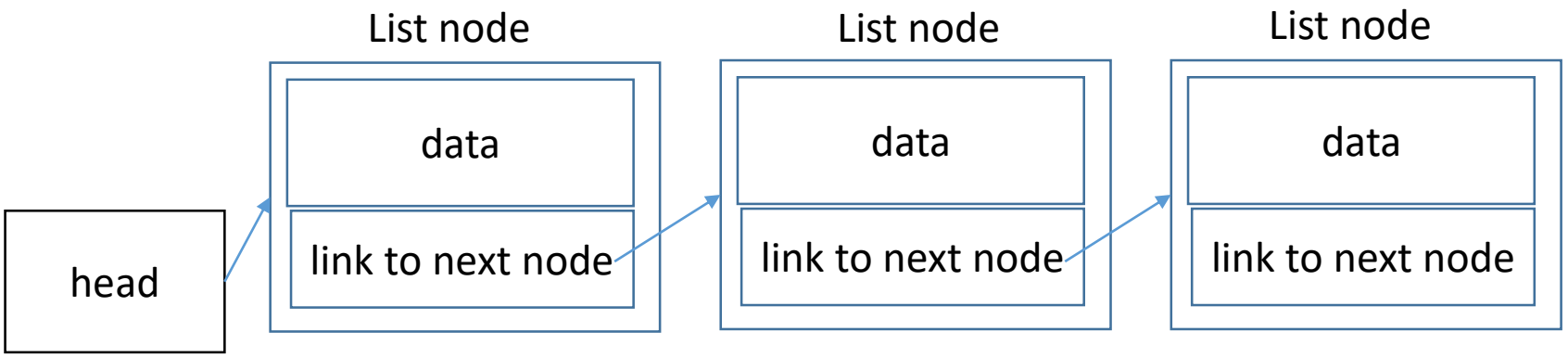
```
typedef struct island {
        char *name;
        int population;
        struct island * next;
} Island;
```

Two ways for storing a sequence of values



Pointer to (address of) the first element of an array: A [0]

Array



Pointer to the first node

Linked list

Array of structs: non-flexible storage

```
typedef struct island {
   char * name;
   int population;
}Island;


Island one = {"Happy",1000};
Island two = {"Empty",0};
Island three = {"Dense",1000000};

Island * tour [3];
tour[0] = &one;
tour[1] = &two;
tour[2] = &three;
```

Island four = {"Sad", 1, NULL};

Insertion in the middle is very inefficient

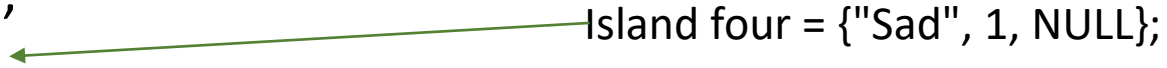
Linked list of structs: more flexible

```
typedef struct island {
   char * name;
   int population;
   struct island * next;
}Island;

Island * head = &one;
one.next = &two;
two.next = &three;

four.next = two.next;
two.next = &four;
```

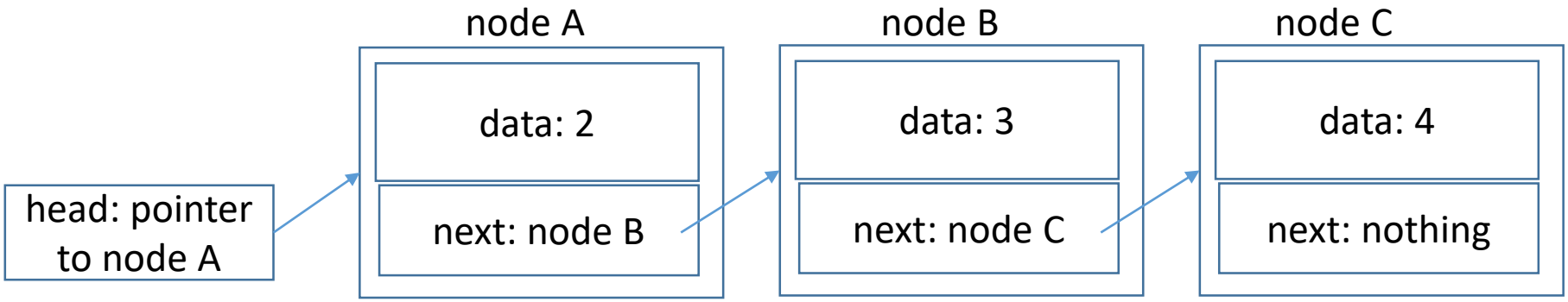
Insertion in the middle in 2 operations, without shifting other values



Linked list vs. array: summary

Linked list	Array
✔ Not limited in size	⛔ Limited in size. Need to re-allocate memory to grow
✔ Insertion or deletion of a node is performed by updating links	⛔ Insertion or deletion of an element may require to move multiple elements
⛔ Access to an indexed position requires sequential scan from the head of the list	✔ Access to an indexed position is performed by adding an index to an address of the first element of an array: constant-time random access
⛔ Memory overhead to store links	

Traversing the list

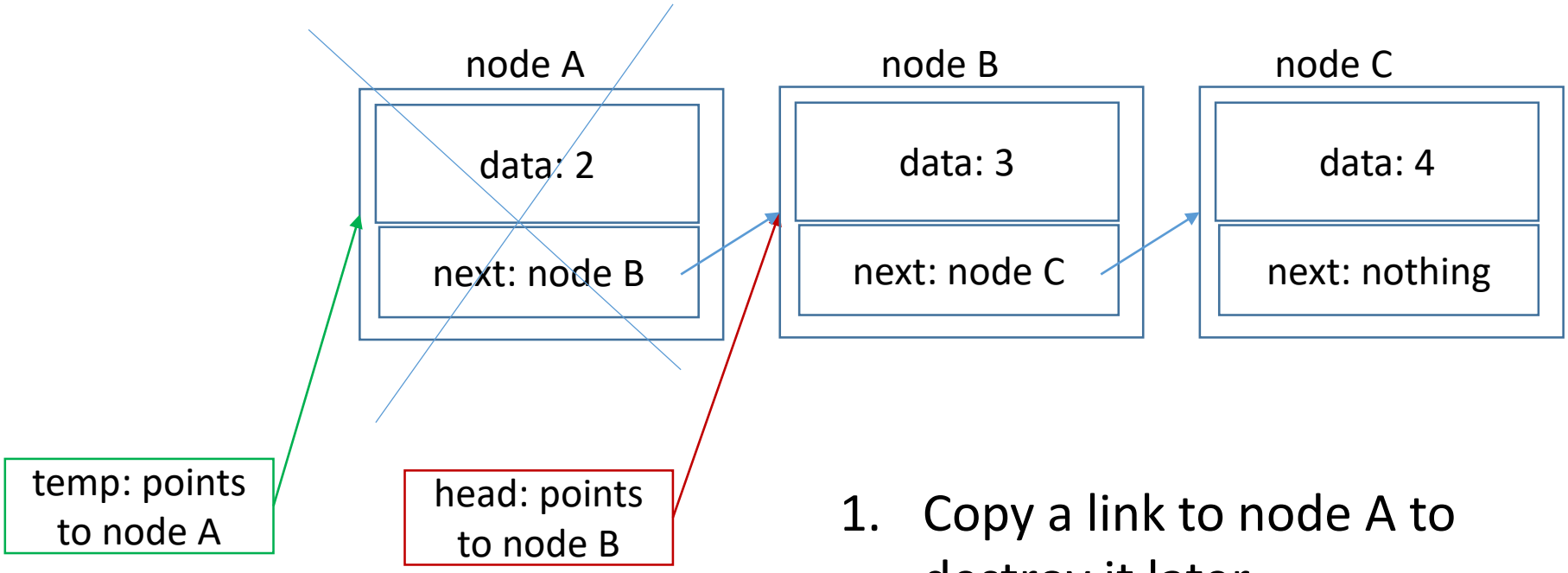


1. Head is all we need to know
2. We follow the sequence by following the links
3. We stop when there is no link to the next node

Example: printing the list

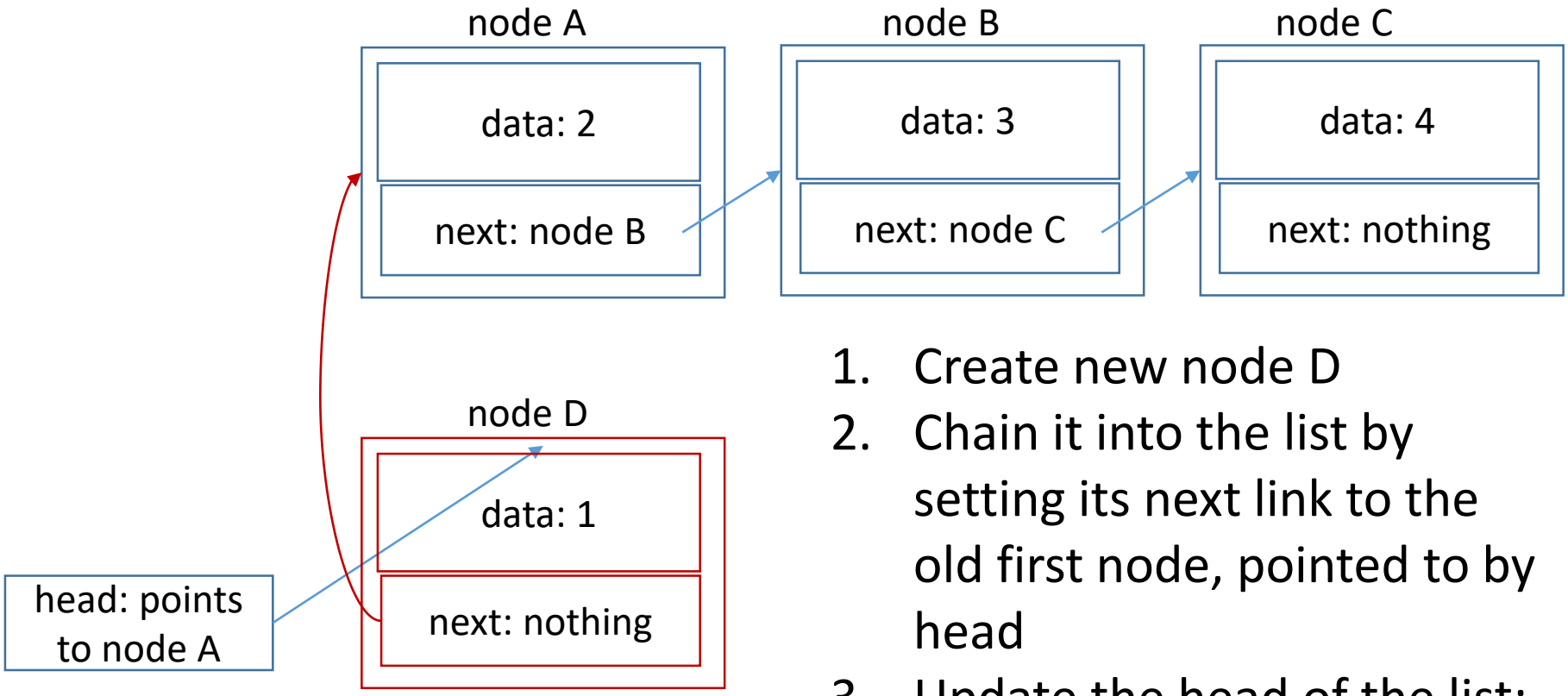
```
void print_tour (Island * head) {
    Island * current = head;
    while (current != NULL) {
        print_island (current);
        current = current->next;
    }
}
```

Removing the first element



1. Copy a link to node A to destroy it later
2. Set head to point to nodeA->next (node B)
3. Destroy node A

Adding a new node at the beginning of the list



1. Create new node D
2. Chain it into the list by setting its next link to the old first node, pointed to by head
3. Update the head of the list: it is now pointing to node D

List nodes dynamically allocated on the heap

```
Island * new_island (char * name) {
    Island * i = malloc (sizeof(Island)*1);
    i->name = name;
    return i;
}
```

Building list dynamically

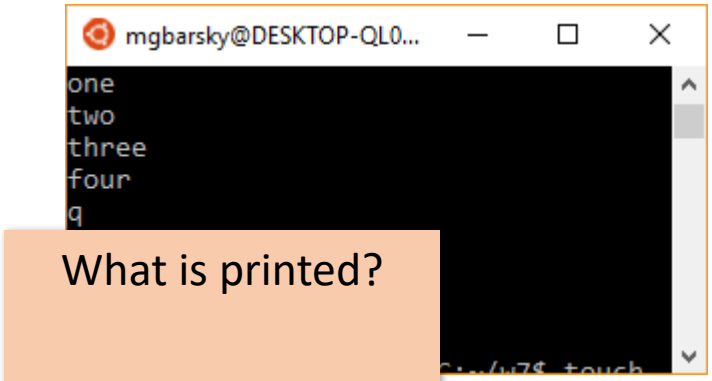
```
while (fgets(buffer, MAX_LINE, stdin)!=NULL){
    buffer [strcspn (buffer, "\r\n")] = '\0';
    Island * i = new_island (buffer);

    if (head != NULL) { //push on top of the list
        i->next = head;
    }
    head = i;
}
```

Is there a problem with this code?

```
while (fgets(buffer, MAX_LINE, stdin)!=NULL){
    buffer [strcspn (buffer, "\r\n")] = '\0';
    Island * i = new_island (buffer);

    if (head != NULL) { //push on top of the list
        i->next = head;
    }
    head = i;
}
```



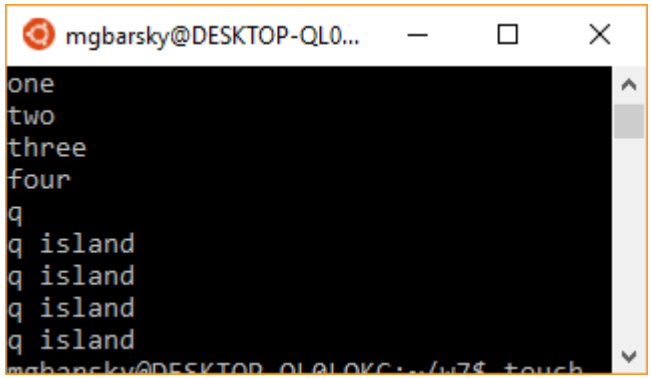
What is printed?

Is there a problem with this code?

```
while (fgets(buffer, MAX_LINE, stdin)!=NULL){
    buffer [strcspn (buffer, "\r\n")] = '\0';
    Island * i = new_island (buffer);

    if (head != NULL) { //push on top of the list
        i->next = head;
    }
    head = i;
}
```

Why do all islands have the same name?



Char pointer needs dynamic allocation too!

```
Island * new_island (char * name) {
    if (name == NULL)
        return NULL;
    Island * i = malloc (sizeof(Island)*1);
    size_t len = strlen (name);
    i->name = malloc (len +1);
    strcpy (i->name, name);
    return i;
}
```

Adding at the end of the list without traversing the list

```
Island * head = NULL;
Island * tail = NULL;

//add at the end of the list
if (head == NULL) {
        head = i;
        tail = i;
}
tail->next = i;
tail=i;
```

Valgrind

- From http://valgrind.org/

"Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools."

- Memcheck is part of valgrind and it checks for the following errors:
  - Use of uninitialized memory
  - Reading/writing memory after it has been freed
  - Reading/writing off the end of malloc'd blocks
  - Memory leaks
  - Doubly freed memory

Using Valgrind Memcheck

Code should be compiled using gcc with -g option –to generate line numbers in memcheck output

gcc –g myprogram.c

valgrind --tool=memcheck ./a.out

Run valgrind

valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./a.out

```
c
d
q
a island
b island
c island
d island
==9180==
==9180== HEAP SUMMARY:
==9180==     in use at exit: 72 bytes in 8 blocks
==9180==   total heap usage: 8 allocs, 0 frees, 72 bytes allocated
==9180==
==9180== 8 bytes in 4 blocks are indirectly lost in loss record 1 of 3
==9180==    at 0x4C2CC70: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9180==    by 0x4007E0: new_island (dynamic_correct.c:19)
==9180==    by 0x400897: main (dynamic_correct.c:35)
==9180==
==9180== 48 bytes in 3 blocks are indirectly lost in loss record 2 of 3
==9180==    at 0x4C2CC70: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9180==    by 0x4007B7: new_island (dynamic_correct.c:17)
==9180==    by 0x400897: main (dynamic_correct.c:35)
==9180==
==9180== 72 (16 direct, 56 indirect) bytes in 1 blocks are definitely lost in loss record 3 of 3
==9180==    at 0x4C2CC70: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9180==    by 0x4007B7: new_island (dynamic_correct.c:17)
==9180==    by 0x400897: main (dynamic_correct.c:35)
==9180==
==9180== LEAK SUMMARY:
==9180==    definitely lost: 16 bytes in 1 blocks
==9180==    indirectly lost: 56 bytes in 7 blocks
==9180==      possibly lost: 0 bytes in 0 blocks
==9180==    still reachable: 0 bytes in 0 blocks
```

Free dynamically allocated lists

```
void free_islands (Island *head) {
        Island *temp;
        Island *node = head;      //start at the head.
        while (node != NULL) {          //traverse entire list.
                temp = node;            //save node pointer.
                node = node->next;      //advance to next.
                free (temp->name);      //free char *
                free (temp);            // free the current node
        }
        head = NULL;    //finally release the head pointer
}
```

Run valgrind again

```
d island
==4227== Conditional jump or move depends on uninitialised value(s)
==4227==
==4227==    by 0x400959: main (dynamic_free1.c:51)
==4227==  Uninitialised value was created by a heap allocation
==4227==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4227==    by 0x400800: new_island (dynamic_free1.c:20)
==4227==    by 0x4008D9: main (dynamic_free1.c:42)
==4227==
==4227== Conditional jump or move depends on uninitialised value(s)
==4227==    at 0x400A15: free_islands (dynamic_free1.c:71)
==4227==    by 0x400968: main (dynamic_free1.c:52)
==4227==  Uninitialised value was created by a heap allocation
==4227==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4227==    by 0x400800: new_island (dynamic_free1.c:20)
==4227==    by 0x4008D9: main (dynamic_free1.c:42)
```

calloc()

- calloc() is just like malloc(), except that
  - it clears the memory to zero for you
  - it takes two parameters instead of one

```
p = malloc (10 * sizeof(int));

p = calloc (10, sizeof(int));
```

Replace *malloc* with *calloc*

```
Island * new_island (char * name) {
    Island * i = (Island *) calloc (1, sizeof(Island));
    size_t len = strlen (name);
    i->name = (char *) calloc (len +1, sizeof (char));
    strcpy (i->name, name);
    return i;
}
```

Run valgrind again: no errors now

```
c
d
q
a island
b island
c island
d island
==6627==
==6627== HEAP SUMMARY:
==6627==     in use at exit: 0 bytes in 0 blocks
==6627==   total heap usage: 8 allocs, 8 frees, 72 bytes allocated
==6627==
==6627== All heap blocks were freed -- no leaks are possible
==6627==
==6627== For counts of detected and suppressed errors, rerun with: -v
==6627== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Our goal

Rules for avoiding memory leaks

- To **avoid accidental access to uninitialized memory** - always use *memset* along with *malloc*, or always use *calloc*
- When writing values to memory block, make sure you **cross-check the number of bytes available and number of bytes being written**
- Before re-assigning the pointers, make sure **no memory locations will become orphaned**
- When freeing struct (which in turn contains the pointer to dynamically allocated memory location), first traverse to the child memory location and start freeing from there, traversing back to the parent node
- Always properly handle return values of functions returning references to dynamically allocated memory – responsibility to free is on the caller!
- Have a corresponding free to every malloc

See examples at: https://www.ibm.com/developerworks/aix/library/au-toughgame/