COMP20003 Workshop Week 3

Linked Lists, Assignment 1

1. Linked Lists: what, why, how?
2. Dictionary ADT

LAB: Team Work

- linked lists
- `gdb`
- Assignment 1

A Common Task: Maintaining A Dynamic Collection of Data

Example: Collection HP of characters in the HP book, maintained as one reads through the story.

- contains data records
- desired actions:

insert(record)

create an empty collection

search(key)

delete(key)

delete the collection

a key (could be any column)

ID	Name	other attributes .				
7	Lord Voldermort	...	...	...		...
10	Harry Potter	...	...	...		...
1	Albus Dumbledore	...	...	...		...
	• • •					
772	Draco Malfoy	...	...	...		...

a record

A Common Task: Maintaining A Dynamic Collection of Data

Example: Collection HP of characters in the HP book, maintained as one reads through the story.

- contains data records
- desired actions:

create an empty collection

insert(record)

search(key)

delete(key)

delete the collection

a key (could be any column)

ID	Name	other attributes ...				
7	Lord Voldermort	...	...	...		...
10	Harry Potter	...	...	...		...
1	Albus Dumbledore	...	...	...		...
	• • •					
772	Draco Malfoy	...	...	...		...

a record

- Such a collection is called a dictionary
- Dictionary is an Abstract Data Type (ADT)
- An ADT represents an interface

- Notes: AT are built-in in Python but not C

In C:

- ADT implemented using a concrete Data Structure (DS)

- Example: dynamic arrays

A Common Task: Maintaining A Dynamic Collection of Data

Example: Collection HP of characters in the HP book, maintained as one reads through the story.

- contains data records
- desired actions:

create an empty collection

insert(record)

search(key)

delete(key)

delete the collection

a key (could be any column)

ID	Name	other attributes ...				
7	Lord Voldermort	...	...	...		...
10	Harry Potter	...	...	...		...
1	Albus Dumbledore	...	...	...		...
	• • •					
772	Draco Malfoy	...	...	...		...

a record

- Such a collection is called a dictionary
- Dictionary is an Abstract Data Type (ADT)
- An ADT represents an interface:
  - what data
  - what operations
- Notes: AT are built-in in Python but not C

In C:

- ADT implemented using a concrete DS
- A DS specifies:
  - the representation of data in memory
  - algorithms for performing operations
- Example: dynamic arrays

Using Dynamic Arrays to Implement Dictionary

Frequently used operations:
(need to be efficient)

insert(record) search(key) delete(key)

array size = 10
length = 5

Index	0	1	2	3	4	5	6	7	8	9
Data	7 Lord...	3 James	4 Lily	1 Alb...	10 Ha...					

How to:

```
search(1)
delete(3)
insert({772, Draco Malfoy, ...})
```

Complexity of:

- search:
- insert at the start:
- insert at the end:
- delete:
    - search for the key
    - remove the record

Using Dynamic Arrays to Implement Dictionary

Frequently used operations:
(need to be efficient)

insert(record) search(key) delete(key)

array size = 10
length = 5

Index	0	1	2	3	4	5	6	7	8	9
Data	7 Lord...	3 James	4 Lily	1 Alb...	10 Ha...					

How to:

```
search(1)
delete(3)
insert({772, Draco Malfoy, ...})
```

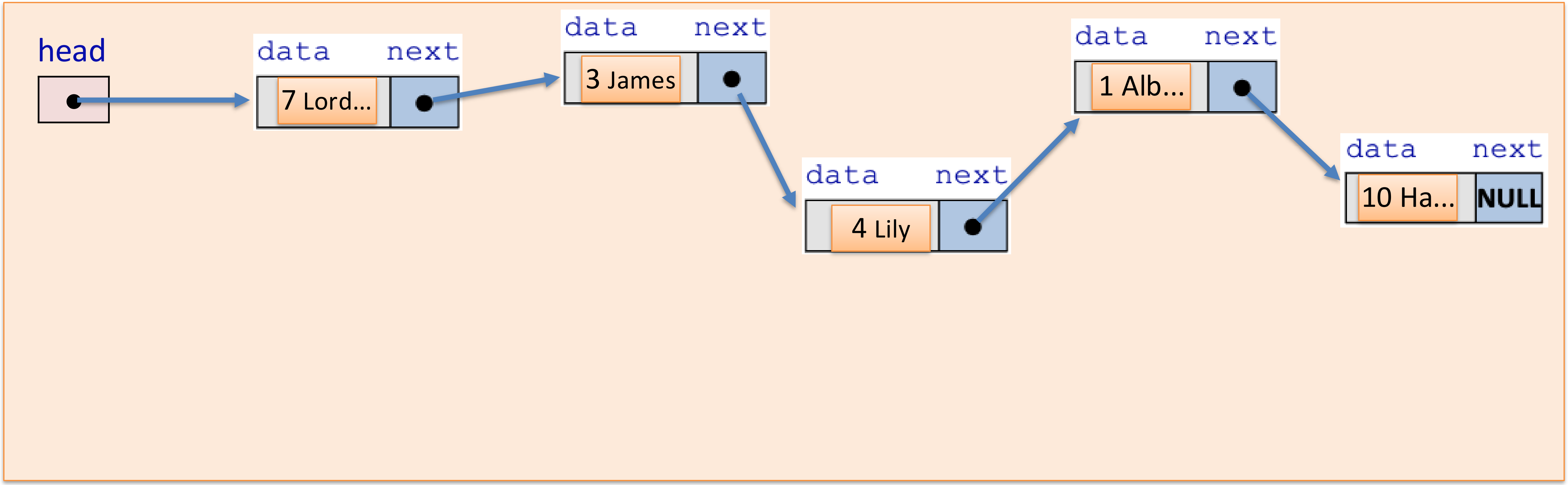
Complexity of:

- search: O(n)
- insert at the start: O(n)
- insert at the end: O(n) due to possible resizing
- delete: O(n) :
  - search for the key O(n)
  - remove the record O(n) due to shifting

Insert/Delete are inefficient due to:

- elements are adjacent
- the “dynamic” feature is not flexible

new DS: Linked Lists – Relax from the Adjacency Requirement

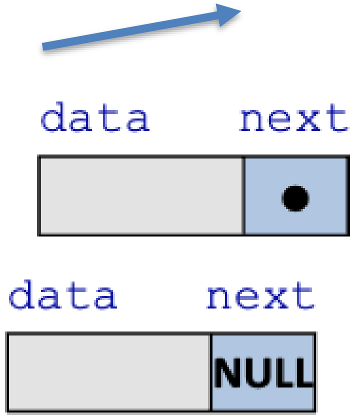


Linked Lists: A Chain of Clues
Data is stored in individual "nodes" that can be anywhere in memory.
Each node contains:

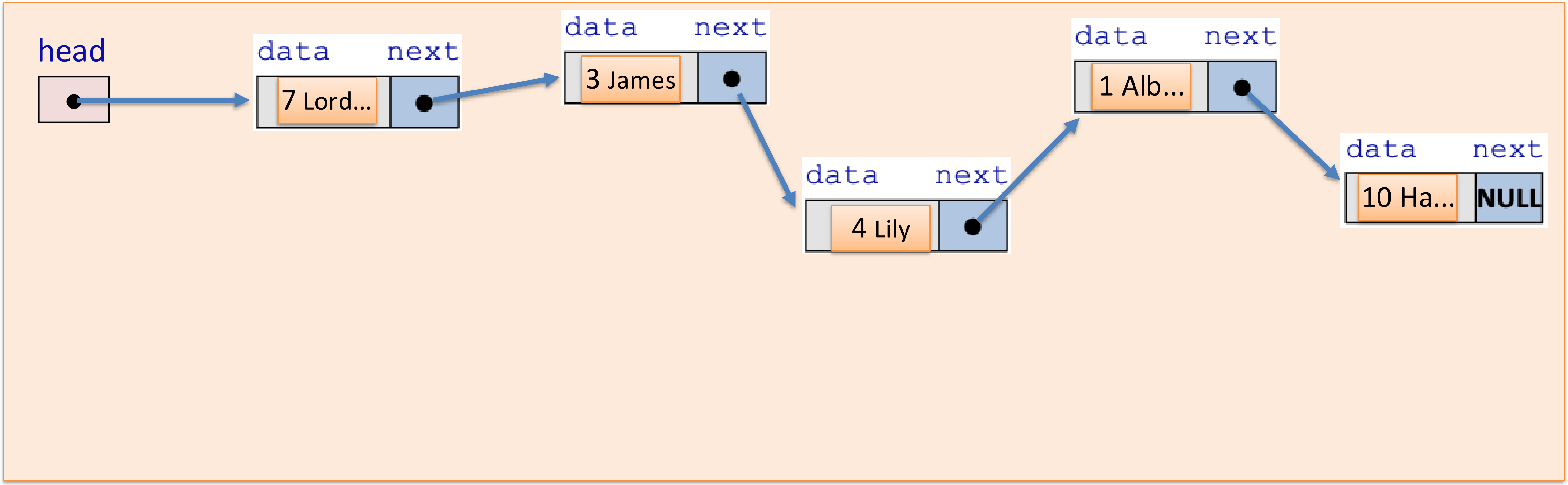
The **data** itself.

A **pointer** to the next node.

The list is managed by a **head pointer** to the first node.

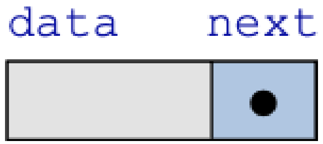


Node Anatomy



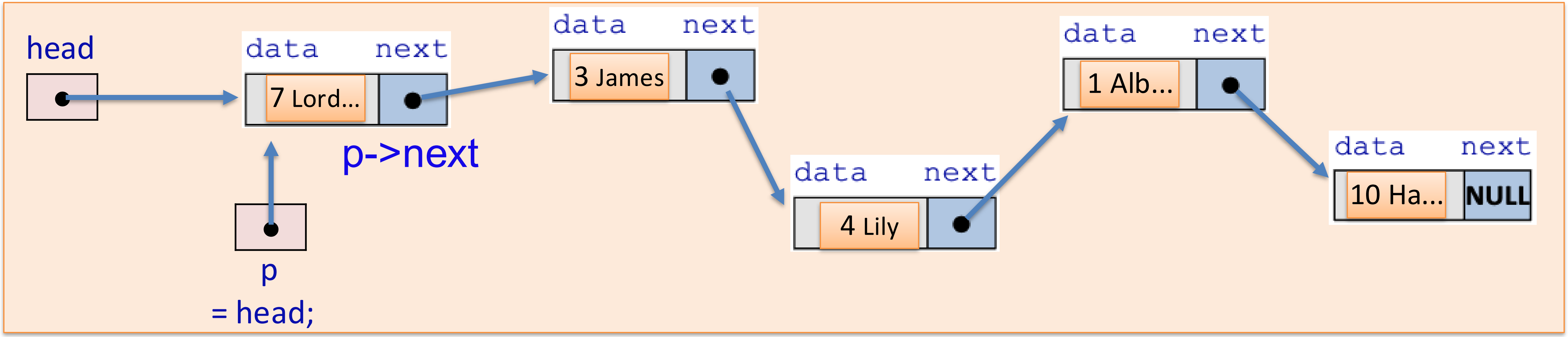
The Building Blocks:
The Node

We use a self-referential struct in C.
The next pointer is the "link" in the chain.




```
typedef struct node {
  data_t data;       // The data the node holds
                     // better: data_t *dataPtr
  struct node *next; // Pointer to the next node
} node_t;
```

How It Works



Connecting the Nodes

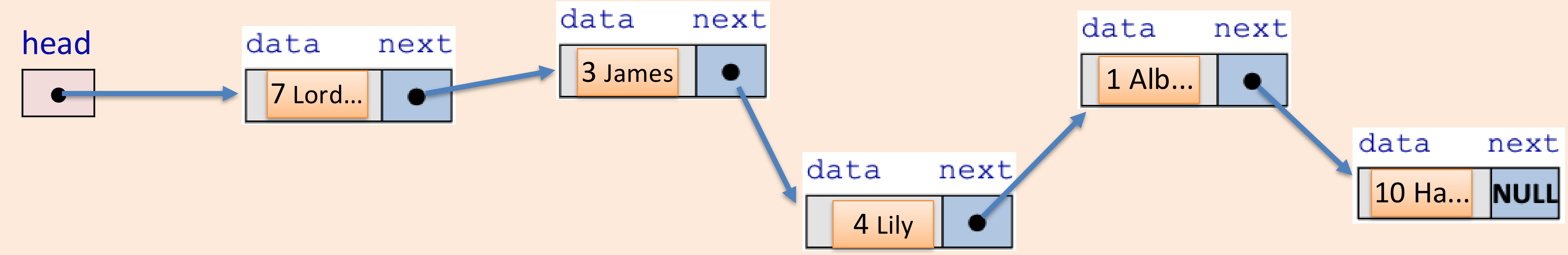
The head pointer is our entry point to the list.

We traverse the list by following the next pointer from one node to the next.

The next pointer of the final node is always NULL to mark the end of the list.

```
node_t *p = head;
while ( p != NULL) {
   // process p->data
   p= p->next;
}
```

Examples of Operations



How to:

`search(1)`

`insert({772, Draco Malfoy, ...})` 772 Dra

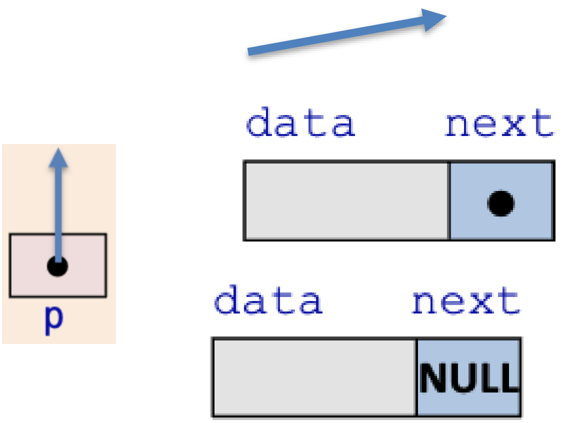
`delete(1)`

Complexity:

- search: O(n)
- insertAtStart : O(1)
- deleteAtStart: O(1)
- deleteAtEnd: O(n)

If we keep track of the last element:

- insertAtEnd: O(1)

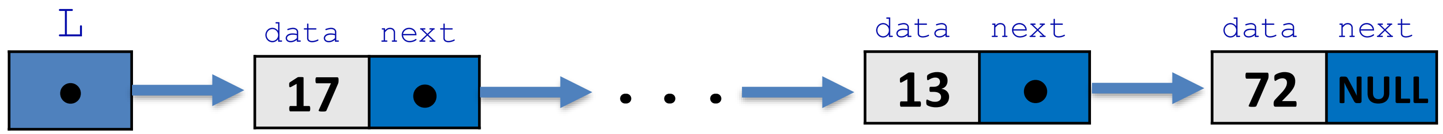


Define Linked Lists: Method 1 (simple, not always usable)

Method 1: Linked list == pointer to the first node of the chain.

```
typedef struct node {
   data_t data;
   struct node *next;
} node_t;
...
node_t *L= NULL; // L is an empty list
...
```

- ✓ Simple and well abstracted!
- ✓ Efficient insert/delete at the start
- ✗ Inefficient insert/delete at the end



Do W3.3 Now

Notes:

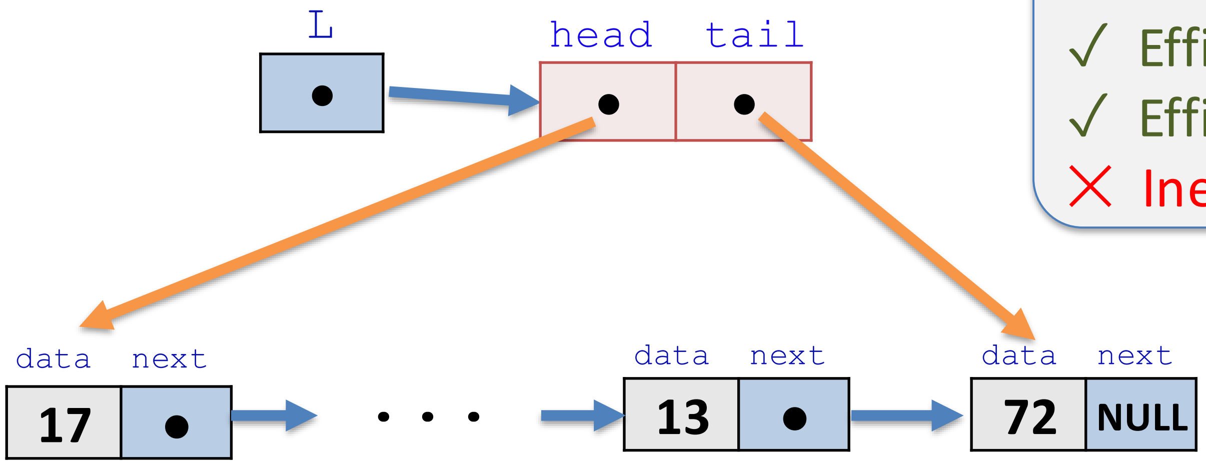
- Using malloc to create nodes. For example: `node_t *n1= malloc(sizeof(*n1))`
- If having nodes n1 and n2, we can link them together with: `n1->next = n2`

Define Linked Lists: Method 2 (popular, used from now on)

Method 2: Linked list is a pair of pointers.

```
typedef struct node {
    data_t data;
    struct node *next;
} node_t;
```

```
struct list {
    node_t *head;
    node_t *tail;
} list_t;
list_t *L= createList();
...
```

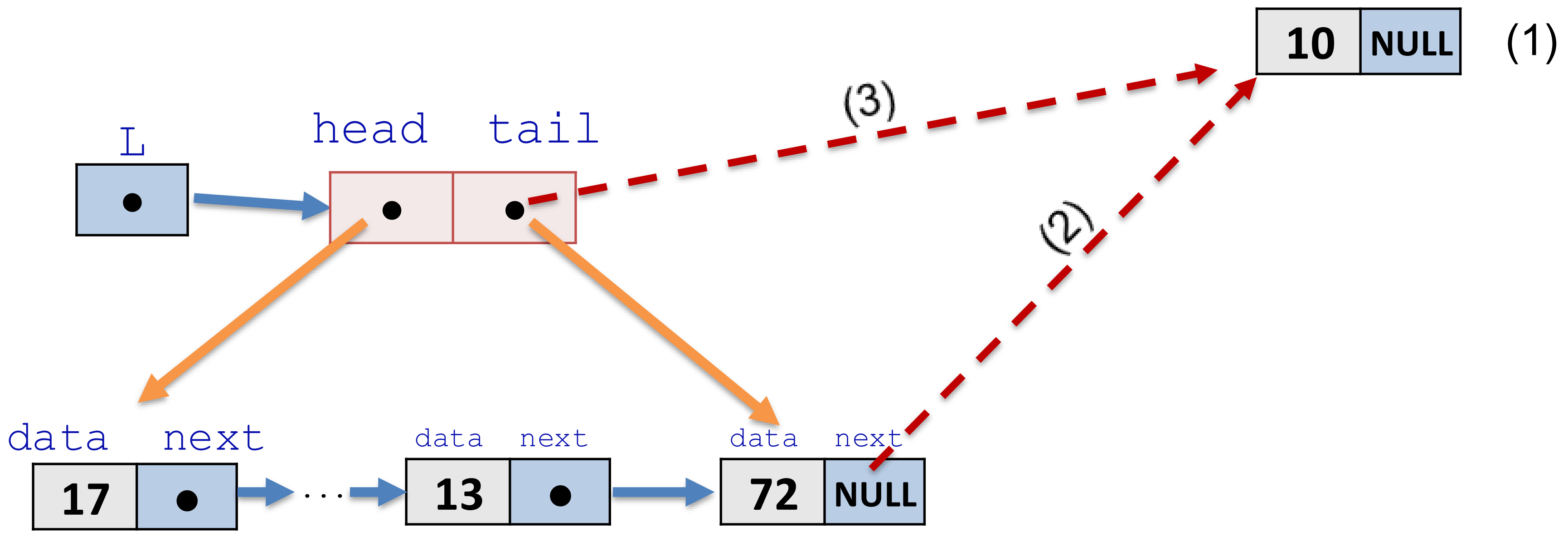


- ✗ more complicated (a bit)
- ✓ Efficient insert/delete at the start
- ✓ Efficient insert at the end
- ✗ Inefficient delete at the end

Example: `listAppend` : append node with `data_t 10` to `list_t *L` – *is this correct?*

1. create new node and set data	`node_t *new= malloc(sizeof(*new));`
	`new->data= 10;    //` here, `data_t` is `int`
	`new->next= NULL;`

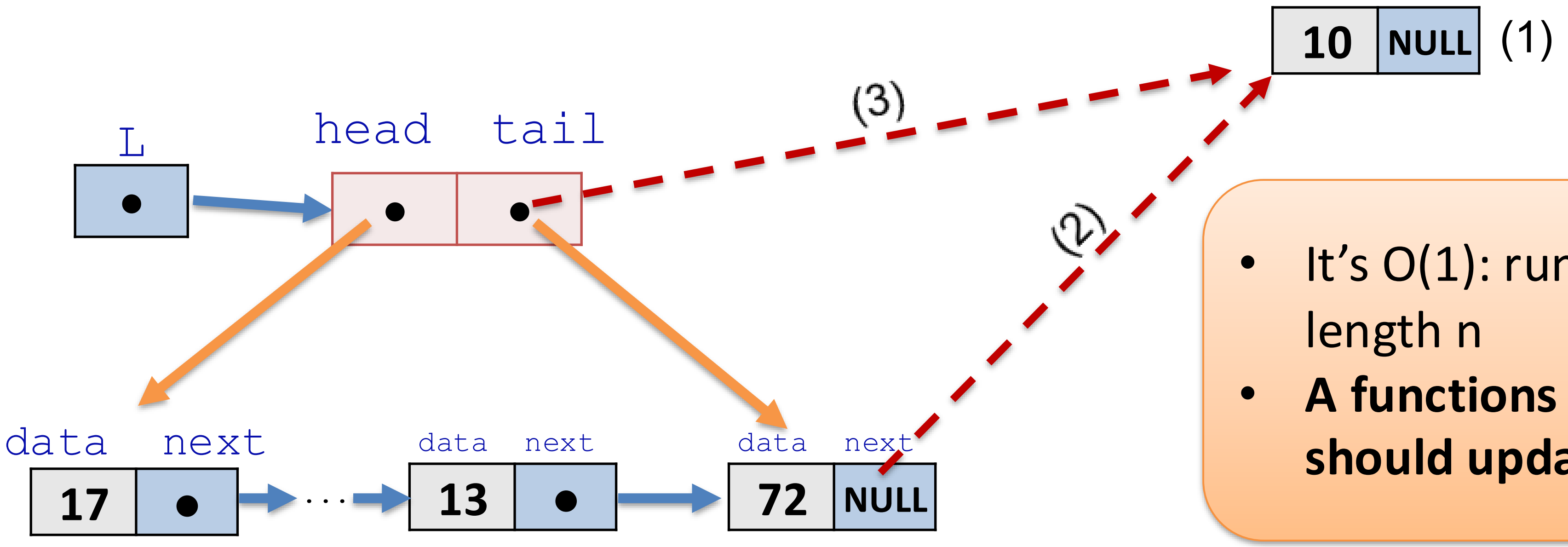
2. Link the node to the chain	`L->tail->next= new;`
3. Repair `tail`	`L->tail= new;`



`listAppend`: append node with `data_t 10` to `list_t *L` – a correct version

1. create new node and set data	`node_t *new= malloc(sizeof(*new));`
	`assert(new);`
	`new->data= 10;    // here, data_t is int`
	`new->next= NULL;`

	`if (L->tail) {`
2. Link the node to the chain	`    L->tail->next= new;`
3. Repair `tail`	`    L->tail= new;`
	`} else`
4. Repair `head` and `tail` if needed	`    L->head= L->tail= new;`

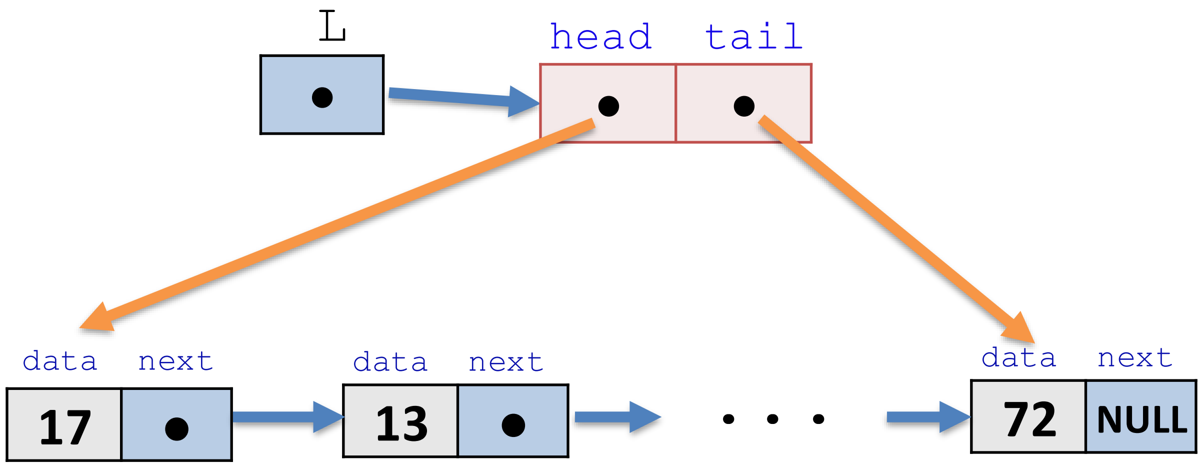


- It's O(1): running time not depending on length n
- **A functions that changes a linked list should update both `head` and `tail`**

Linked lists: Basic Operations

Basic operations:

- `list_t *create()` : create an empty list
- `void prepend(list_t *L, void *data)`: insert a data to the start of a list
- `void  append(list_t *L, void *data)`: append a data to the end of a list
- `void *search(list_t *L, int key, int (*dataGetKey)(void *) )`: search for a data in a list
- `void *deleteHead(list_t *L)` : remove & return (the pointer to) the first data
- `void *deleteTail(list_t *L)` : remove & return the last element
- `void freeList(list_t *L)` : delete a list, free the memory



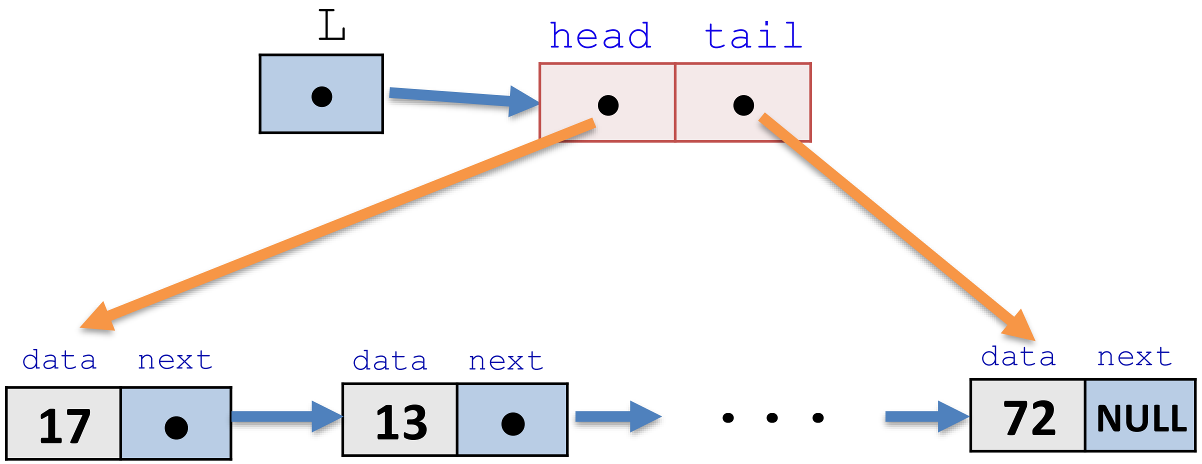
Discussions:

- In function header `prepend`:
  can list_t *L
  be replaced with list_t L ?
- In `search`, explain (what & why):
  int (*dataGetKey)(void *)

Linked lists: Example of Applications

Basic operations:

- `create`: create an empty list
- `prepend/insertHead`: insert a data to the start of a list
- `append/insertTail`: append a data to the end of a list
- `search`: search for a data in a list
- `deleteHead`: remove & return the first element
- `deleteTail`: remove & return the last element
- `free`: delete a list, free the memory



Discussion: We have a sequence of integers such as 17, 13, ..., 72. How to:

- *build a linked list for these integers in the input order?*
- *build a linked list for these integers in the reversed input order?*

Peer Activity on `deleteHead`

What is the correct ordering for these code snippets to implement a function

`void delete_head(struct list *lst)`

that deletes a `struct list`**'s head node?**

a. 2-5-3-4-1

b. 5-2-3-4-1

c. 2-5-4-1-3

d. 5-2-4-1-3

```c
/* Snippet 1 */
free(old);
(lst->size)--;

/* Snippet 2 */
if ((lst->head == NULL) && (lst->tail == NULL))
    return;

/* Snippet 3 */
if (new == NULL) lst->tail = NULL;

/* Snippet 4 */
lst->head = new;

/* Snippet 5 */
struct lnode *old = lst->head,
             *new = old->next;
```

Arrays vs. Linked Lists

Features/Operations	Arrays	Linked Lists
Memory Representation	contiguous memory chunk	chain of scattered nodes
Size	Fixed (or slow to resize)	Dynamic (easy to grow/shrink)
Access by index	**Fast (O(1))**	Slow (O(n), must traverse)
Search	O(n)	O(n)
Delete by Key: • search by Key • remove the record	 O(n) O(1)	 O(n) O(1)
Insert at the start	O(n)	O(1)
Insert at the end	O(n)	O(1)
Delete at the start	O(n)	O(1)
Delete at the end	O(1)	O(n)

Summary: Dictionary ADT and some corresponding concrete DS

ADT
= interface
= what?

ADT: Dictionary

- data records
- operations:

create

insert

search

delete

free

ID	Name
1007	Lord Voldermort
1010	Harry Potter
1001	Albus Dumbledore
• • •	

a datum

a key

DS
= implementation
= how?

Static Array

Dynamic Array

Sorted Dynamic Array

Linked List

• • •

Discussion: In these DS, how efficient can insert/delete/search be? Using notations:

- O(1) if the running time does not depend on the array length n
- O(n) if the running time is at most linear to n
- O(log n) if the running time is at most linear to $\log_k n$ (where k is a constant > 1)

Summary: Dictionary as an ADT and some concrete DS

ADT: Dictionary

- contains data records, each record has a key
- operations:

create

insert(datum)

search(key)

free

delete(key)

ID	Name
1007	Lord Voldermort
1010	Harry Potter
1001	Albus Dumbledore
	• • •
1772	Draco Malfoy

Unsorted Array (supposing size is OK)

insert(datum):
O(1): just add to the end

delete (key):
O(n): do sequential search O(n) then shifting O(n) → O(n) in total

search(key):
O(n): do sequential search

Sorted Array

insert(datum):
O(n): do binary search + shifting

delete (key):
O(n): do binary search O(log n) + shifting O(n) O(n) → O(n) in total

search(key):
O(log n): do binary search

Linked List

insert(datum):
O(1): insert at start or end

delete (key):
O(n): do sequential search O(n) then removing O(1) → O(n) in total

search(key):
O(n): do sequential search

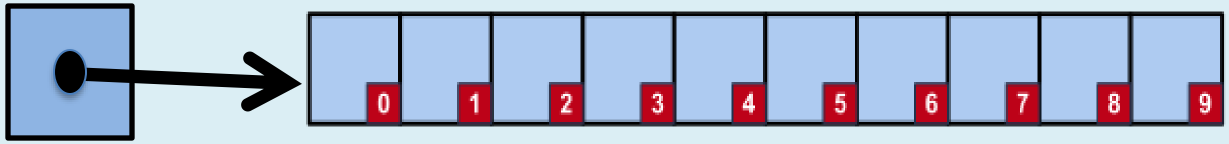
Notes on size problem for arrays: insert is different for static & dynamic arrays

- static arrays: insert is impossible when arrays are full
- dynamic arrays: complexity is O(n) even for the unsorted case due to occasional resizing

Review Questions: Arrays vs Linked Lists

ARRAY

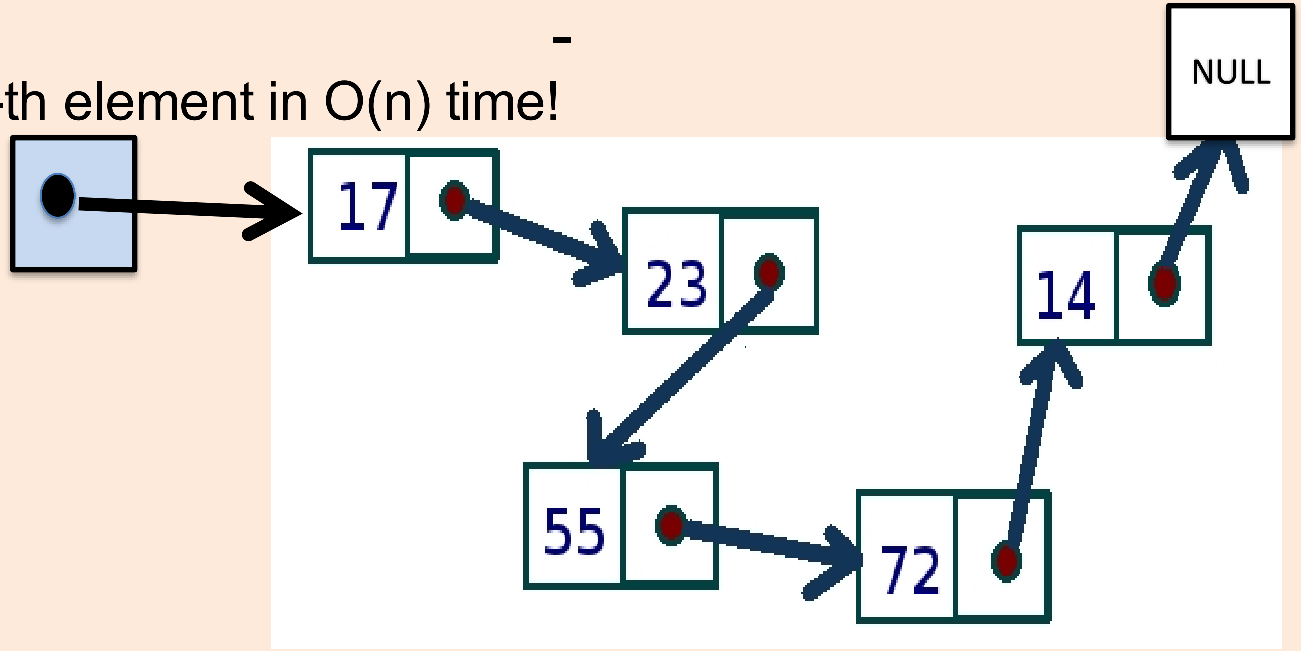
Access A[k] in O(1) time!



A (`A= malloc(10 * sizeof(*A);`)

LINKED LIST

Access k-th element in O(n) time!



Comparing arrays and linked lists. Which one is more efficient for:

- accessing the k-th element?
- inserting a new element at the start?
- deleting the first element?
- searching for a key?

Lab Time: Work with your A1-Teamate

Together with your A1-teamate:

1. explore & do W3.4
2. learn gdb by following the videos in W3.5, W3.6
3. do assignment 1
4. perhaps complete other W3.x at home ...

A1 Consultations

There will be a few(?) next week. One is:

Tuesday 1PM-4PM
PAR-260-L1-101a-101 & 102 combined (32)
== **200 BERKELEY ST** (BUILDING **260**).
Room 101 & 102 in Level 1

Assignment 1

- Effective Team Collaboration is important.
- Use the Discussion Forum actively & wisely!
- Ideally, finish your assignment a day before the deadline.

Notes:

- `valgrind` is excellent for finding potential problems
- `gdb` is great for tracing your code
- but don't rely too much on debugging: *having a clear algorithm saves days of debugging*

Debugging: GDB: Cheat Sheet

Synopsis:

```
gdb [prog]
```

Essential commands (to be executed from within GDB's shell):

```
(h)elp                              # lists help topics
(b)reakpoint file.c:line_num        # sets a breakpoint in file.c on line_num
(r)un arg1 arg2 ... < in.file       # runs the program with extra args and file redirections
(n)ext                              # runs the next *line*
(s)tep                              # runs the next *instruction*
set var var_name=val                # sets the value of var_name
bt                                  # back trace (to see all function call at the stop point)
(p)rint var_name                    # prints var_name's current value
(p)rint func(arg1, arg2, ...)       # prints the return value of func when called with args
(q)uit                              # quits GDB
Ctrl+X Ctrl+A                       # displays the loaded program's source code alongside GDB's shell
```

Note: This is an excerpt of GDB's man page.