Singly-Linked Lists
Singly-Linked Lists

• So far in the course, we’ve only had one kind of data structure for representing collections of like values.
  • structs, recall, give us “containers” for holding variables of different data types, typically.

• Arrays are great for element lookup, but unless we want to insert at the very end of the array, inserting elements is quite costly – remember insertion sort?
Singly-Linked Lists

• Arrays also suffer from a great inflexibility – what happens if we need a larger array than we thought?

• Through clever use of pointers, dynamic memory allocation, and structs, we can put those two pieces together to develop a new kind of data structure that gives us the ability to grow and shrink a collection of like values to fit our needs.
Singly-Linked Lists

• We call this combination of elements, when used in this way, a **linked list**.

• A linked list **node** is a special kind of struct with two members:
  • Data of some data type (int, char, float...)
  • A pointer to another node of the same type

• In this way, a set of nodes together can be thought of as forming a chain of elements that we can follow from beginning to end.
Singly-Linked Lists

typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
typedef struct sllist {
    VALUE val;
    struct sllist* next;
} sllnode;
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
Singly-Linked Lists

typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
Singly-Linked Lists

• In order to work with linked lists effectively, there are a number of operations that we need to understand:

1. Create a linked list when it doesn’t already exist.
2. Search through a linked list to find an element.
3. Insert a new node into the linked list.
4. Delete a single element from a linked list.
5. Delete an entire linked list.
Singly-Linked Lists

• Create a linked list.

```c
sllnode* create(VALUE val);
```
Singly-Linked Lists

• Create a linked list.

\[
\text{sllnode}^* \ \text{create}(\text{VALUE } \text{val});
\]

• Steps involved:
  a. Dynamically allocate space for a new \text{sllnode}.
  b. Check to make sure we didn’t run out of memory.
  c. Initialize the node’s \text{val} field.
  d. Initialize the node’s \text{next} field.
  e. Return a pointer to the newly created \text{sllnode}.
Singly-Linked Lists

sllnode* new = create(6);

a. Dynamically allocate space for a new sllnode.
b. Check to make sure we didn’t run out of memory.
c. Initialize the node’s val field.
d. Initialize the node’s next field.
e. Return a pointer to the newly created sllnode.
Singly-Linked Lists

sllnode* new = create(6);

a. Dynamically allocate space for a new sllnode.
b. Check to make sure we didn’t run out of memory.
c. Initialize the node’s val field.
d. Initialize the node’s next field.
e. Return a pointer to the newly created sllnode.
Singly-Linked Lists

sllnode* new = create(6);

a. Dynamically allocate space for a new sllnode.
b. Check to make sure we didn’t run out of memory.
c. Initialize the node’s val field.
d. Initialize the node’s next field.
e. Return a pointer to the newly created sllnode.
Singly-Linked Lists

sllnode* new = create(6);

a. Dynamically allocate space for a new sllnode.
b. Check to make sure we didn’t run out of memory.
c. Initialize the node’s val field.
d. Initialize the node’s next field.
e. Return a pointer to the newly created sllnode.
Singly-Linked Lists

sllnode* new = create(6);

a. Dynamically allocate space for a new sllnode.
b. Check to make sure we didn’t run out of memory.
c. Initialize the node’s val field.
d. Initialize the node’s next field.
e. Return a pointer to the newly created sllnode.
Singly-Linked Lists

- Search through a linked list to find an element.

```cpp
bool find(sllnode* head, VALUE val);
```
Singly-Linked Lists

• Search through a linked list to find an element.

bool find(sllnode* head, VALUE val);

• Steps involved:
  a. Create a traversal pointer pointing to the list’s head.
  b. If the current node’s val field is what we’re looking for, report success.
  c. If not, set the traversal pointer to the next pointer in the list and go back to step b.
  d. If you’ve reached the end of the list, report failure.
Singly-Linked Lists

bool exists = find(list, 6);
bool exists = find(list, 6);
Singly-Linked Lists

```c
bool exists = find(list, 6);
```

```
list

2 → 3 → 5 → 6
```

```c
trav

8
```
Singly-Linked Lists

bool exists = find(list, 6);
Singly-Linked Lists

bool exists = find(list, 6);
Singly-Linked Lists

```cpp
bool exists = find(list, 6);
```

![Singly-Linked List Diagram]

bool exists = find(list, 6);
Singly-Linked Lists

bool exists = find(list, 6);

list

2 → 7 → 5 → 3 → 8

trav
Singly-Linked Lists

bool exists = find(list, 6);
bool exists = find(list, 6);
Singly-Linked Lists

bool exists = find(list, 6);
Singly-Linked Lists

• Insert a new node into the linked list.

```c
sllnode* insert(sllnode* head, VALUE val);
```
Singly-Linked Lists

• Insert a new node into the linked list.

        sllnode* insert(sllnode* head, VALUE val);

• Steps involved:
  a. Dynamically allocate space for a new sllnode.
  b. Check to make sure we didn’t run out of memory.
  c. Populate and insert the node at the beginning of the linked list.
  d. Return a pointer to the new head of the linked list.
Singly-Linked Lists

- Insert a new node into the linked list.

```c
sllnode* insert(sllnode* head, VALUE val);
```

- Steps involved:
  a. Dynamically allocate space for a new sllnode.
  b. Check to make sure we didn’t run out of memory.
  c. Populate and insert the node at the beginning of the linked list.
  d. Return a pointer to the new head of the linked list.
Singly-Linked Lists

list = insert(list, 12);
Singly-Linked Lists

```plaintext
list = insert(list, 12);
```
Singly-Linked Lists

```
list = insert(list, 12);
```
Singly-Linked Lists

• Decision time!

• Which pointer should we move first? Should the “12” node be the new head of the linked list, since it now exists, or should we connect it to the list first?

• This is one of the trickiest things with linked lists. Order matters!
Singly-Linked Lists

```python
list = insert(list, 12);
```

![Diagram of a singly-linked list with nodes 12, 15, 9, 13, and 10, with a new node at the end inserted by the function `insert(list, 12)`.]
Singly-Linked Lists

```java
list = insert(list, 12);
```
Singly-Linked Lists

list = insert(list, 12);
Singly-Linked Lists

```
list = insert(list, 12);
```

![Diagram of a list with nodes 12, 15, 9, 13, and 10. The new node with value 12 is inserted after node 15.]
Singly-Linked Lists

```c
list = insert(list, 12);
```

![Diagram of singly-linked list with nodes 12, 15, 9, 13, and 10 with an arrow pointing to the new node.](image-url)
Singly-Linked Lists

```java
list = insert(list, 12);
```
Singly-Linked Lists

- Delete an entire linked list.

```c
void destroy(sllnode* head);
```
Singly-Linked Lists

• Delete an entire linked list.

\[
\text{void destroy(sllnode* head);} 
\]

• Steps involved:
  a. If you’ve reached a null pointer, stop.
  b. Delete the rest of the list.
  c. Free the current node.
Singly-Linked Lists

• Delete an entire linked list.

```c
void destroy(sllnode* head);
```

• Steps involved:
  a. If you’ve reached a null pointer, stop.
  b. Delete the rest of the list.
  c. Free the current node.
Singly-Linked Lists

```plaintext
destroy(list);
```

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.
Singly-Linked Lists

destroy(list);

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.
Singly-Linked Lists

\[ \text{destroy(list)}; \]

a. If you’ve reached a null pointer, stop.

b. Delete the rest of the list.

c. Free the current node.
Singly-Linked Lists

```plaintext
destroy(list);
```

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

![Diagram of a singly-linked list with nodes labeled 12, 15, 9, 13, and 10. The list is being destroyed from the head node to the tail node.](image)
Singly-Linked Lists

```c
destroy(list);
```

a. If you’ve reached a null pointer, stop.

b. Delete the rest of the list.

c. Free the current node.

```
list
  12
  ↓
  15
  ↓
  9
  ↓
  13
  ↓
  10
```

STACK FRAMES

```c
destroy()
```

```c
destroy()
```

```c
destroy()
```
Singly-Linked Lists

```python
def destroy(list):
    a. If you’ve reached a null pointer, stop.
    b. Delete the rest of the list.
    c. Free the current node.
```

```
list
12
```

```
list
15
```

```
list
9
```

```
list
13
```

```
list
10
```

STACK FRAMES

```
Singly-Linked Lists

```c
destroy(list);
```

a. If you’ve reached a null pointer, stop.

b. Delete the rest of the list.

c. Free the current node.

```
destroy(list);
```

- list
  - 12
  - 15
  - 9
  - 13
  - 10

STACK FRAMES

- destroy()
- destroy()
Singly-Linked Lists

destroy(list);

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.
Singly-Linked Lists

```cpp
destroy(list);
```

a. If you’ve reached a null pointer, stop.
b. Delete **the rest of the list**.
c. Free the current node.
Singly-Linked Lists

destroy(list);

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.
Singly-Linked Lists

destroy(list);

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.
Singly-Linked Lists

```cpp
destroy(list);
```

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.
Singly-Linked Lists

```c
destroy(list);
```

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

```
12

15

9

13

10
```

STACK FRAMES

```
destroy()
destroy()
destroy()
destroy()
destroy()
```
Singly-Linked Lists

```cpp
destroy(list);
```

a. If you’ve reached a null pointer, stop.

b. Delete the rest of the list.

c. Free the current node.
destroy(list);

a. If you’ve reached a null pointer, stop.

b. Delete the rest of the list.

c. Free the current node.
Singly-Linked Lists

destroy(list);

a. If you’ve reached a null pointer, stop.

b. Delete the rest of the list.

c. Free the current node.
Singly-Linked Lists

defroy(list);

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

stack frames
Singly-Linked Lists

destroy(list);

- a. If you’ve reached a null pointer, stop.
- b. Delete the rest of the list.
- c. Free the current node.
Singly-Linked Lists

destroy(list);

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.

12

15
Singly-Linked Lists

```
destroy(list);
```

1. If you’ve reached a null pointer, stop.
2. Delete the rest of the list.
3. Free the current node.
Singly-Linked Lists

```c
destroy(list);
```

- a. If you’ve reached a null pointer, stop.
- b. Delete the rest of the list.
- c. Free the current node.

![Diagram of list with node labeled 12]
Singly-Linked Lists

defroy(list);

a. If you’ve reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.
Singly-Linked Lists

```c
destroy(list);
```

a. If you’ve reached a null pointer, stop.
b. Delete **the rest of the list**.
c. Free the current node.
Singly-Linked Lists

• In order to work with linked lists effectively, there are a number of operations that we need to understand:

1. Create a linked list when it doesn’t already exist.
2. Search through a linked list to find an element.
3. Insert a new node into the linked list.
4. Delete a single element from a linked list.
5. Delete an entire linked list.