Doubly-Linked Lists
Doubly-Linked Lists

- Singly-linked lists really extend our ability to collect and organize data, but they suffer from a crucial limitation.
  - We can only ever move in one direction through the list.

- Consider the implication that would have for trying to delete a node.

- A doubly-linked list, by contrast, allows us to move forward and backward through the list, all by simply adding one extra pointer to our struct definition.
Doubly-Linked Lists

typedef struct dllist
{
    VALUE val;
    struct dllist* prev;
    struct dllist* next;
} dllnode;
Doubly-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.
Doubly-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.
Doubly-Linked Lists

- Insert a new node into the linked list.

```c
dllnode* insert(dllnode* head, VALUE val);
```
Doubly-Linked Lists

• Insert a new node into the linked list.

\[
dllnode* \ insert(dllnode* \ head, \ VALUE \ val)\
\]

• Steps involved:
  a. Dynamically allocate space for a new dllnode.
  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. Fix the prev pointer of the old head of the linked list.
  e. Return a pointer to the new head of the linked list.
Doubly-Linked Lists

• Insert a new node into the linked list.

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

• Steps involved:
  a. Dynamically allocate space for a new dllnode.
  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. Fix the prev pointer of the old head of the linked list.
  e. Return a pointer to the new head of the linked list.
Doubly-Linked Lists

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

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

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

• Remember, we can never break the chain when rearranging the pointers.

• Even if we need to have redundant pointers temporarily, that’s okay.
Doubly-Linked Lists

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

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

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

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

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

- Delete a node from a linked list.

```c
void delete(dllnode* target);
```
Doubly-Linked Lists

• Delete a node from a linked list.

```c
void delete(dllnode* target);
```

• Steps involved:
  a. Fix the pointers of the surrounding nodes to “skip over” target.
  b. Free target.
Doubly-Linked Lists

**delete(x);**
Doubly-Linked Lists

delete(x);
Doubly-Linked Lists

```
delete(x);
```
Doubly-Linked Lists

delete(x);
Doubly-Linked Lists

delete(x);
Doubly-Linked Lists

• Linked lists, of both the singly- and doubly-linked varieties, support extremely efficient insertion and deletion of elements.
  • In fact, these operations can be done in constant time.

• What’s the downside? Remember how we had to find an element? We’ve lost the ability to randomly-access list elements.
  • Accessing a desired element may now take linear time.