Dynamic Sets

DSA, Fall 2022

Dynamic Set ADT

The Dynamic Set ADT implements sets of objects that can grow or shrink.

These objects can be base values (ints, strings) or structured objects (with fields).

Typical operations:

```
NewSet()

Search(s, key)
Insert(s, x)
Delete(s, x)
```

```
Minimum(s)
Maximum(s)

Successor(s, x)
Predecessor(s, x)
```

Variants

Search by value vs by key (field of an object)

Keys are totally ordered vs not ordered

Keys are distinct vs allowed to be the same (set vs multiset)

Mutable structure vs immutable structure

Variants

Search by value vs by key (field of an object)

Keys are totally ordered vs not ordered

Keys are distinct vs allowed to be the same (set vs multiset)

Mutable structure vs immutable structure

Our signature

```
type Set
type Cell

NewSet:   ()            → *Set
Search:   (*Set, int)   → *Cell
Insert:   (*Set, *Cell) → ()
Delete:   (*Set, *Cell) → ()
Minimum:  *Set          → *Cell
Maximum:  *Set          → *Cell
```

Linked list implementation

```
type Set struct {
   head *Cell
}


type Cell struct {
   value int
   next *Cell
}
```

Linked list implementation

```
func Search(s *Set, k int) *Cell {
   Scan from s.head
   Follow next ptrs until
      current cell value == k
}
```

Linked list implementation

```
func Insert(s *Set, c *Cell) {
    Add c to the front of the list
    Update s.head pointer
}
```

Linked list implementation

```
func Delete(s *Set, c *Cell) {
   Scan from s.head
   Follow next ptrs p until
      cell p.next == c
   Set p.next = c.next
}
```

Linked list implementation

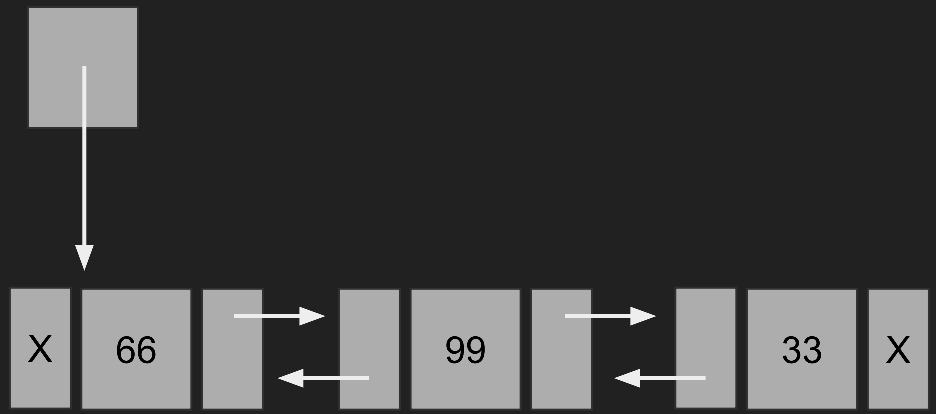
```
func Minimum(s *Set) *Cell {
   Scan from s.head
   Follow next ptrs remembering
      cell with min value
   Return cell with min value
}

func Maximum(s *Set) *Cell {
   Scan from s.head
   Follow next ptrs remembering
      cell with max value
   Return cell with max value
}
```

Asymptotic running times

	Linked list
Search	$\Theta(n)$
Insert	$\Theta(1)$
Delete	$\Theta(n)$
Minimum	$\Theta(n)$
Maximum	$\Theta(n)$

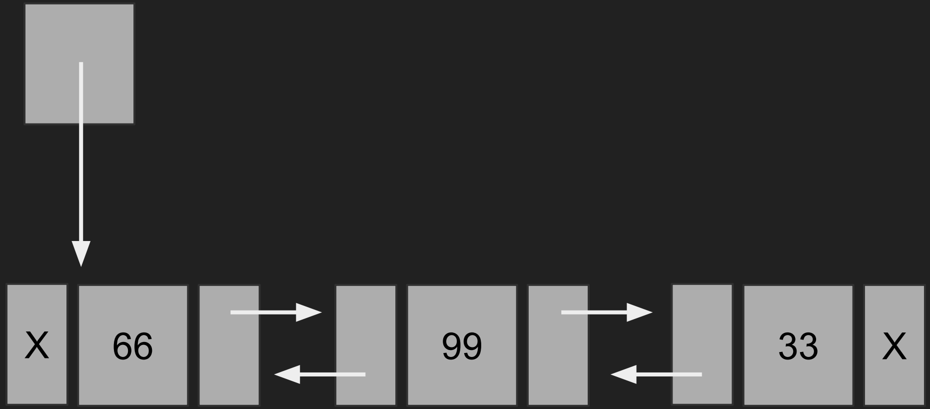
Doubly-linked list implementation

```
type Set struct {
    head *Cell
}


type Cell struct {
    value int
    prev *Cell
    next *Cell
}
```



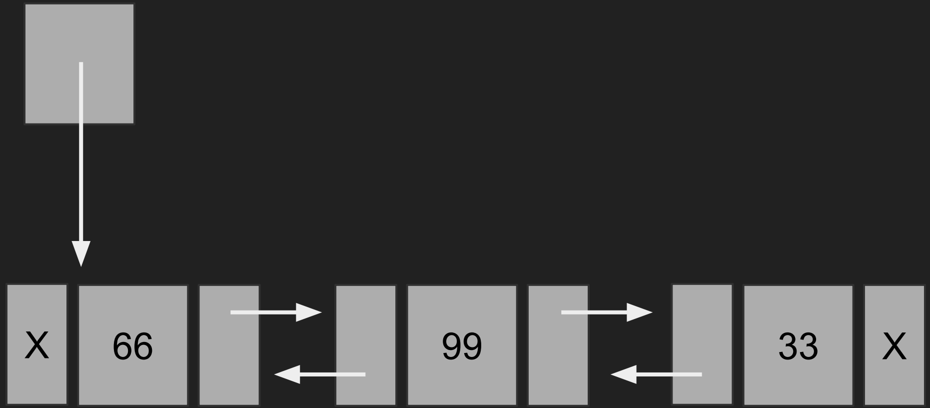
Doubly-linked list implementation

```
func Search(s *Set, k int) *Cell {
   Scan from s.head
   Follow next ptrs until
      current cell value == k
}
```



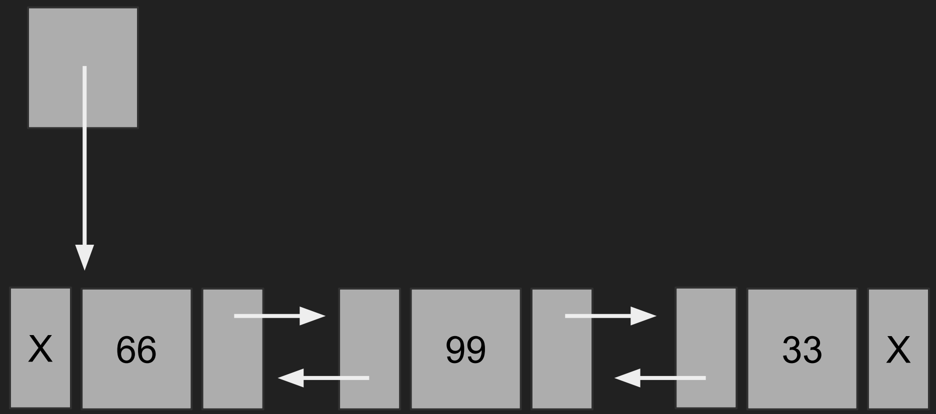
Doubly-linked list implementation

```
func Insert(s *Set, c *Cell) {
    Add c to the front of the list
    Update s.head pointer
    Update prev pointers
}
```



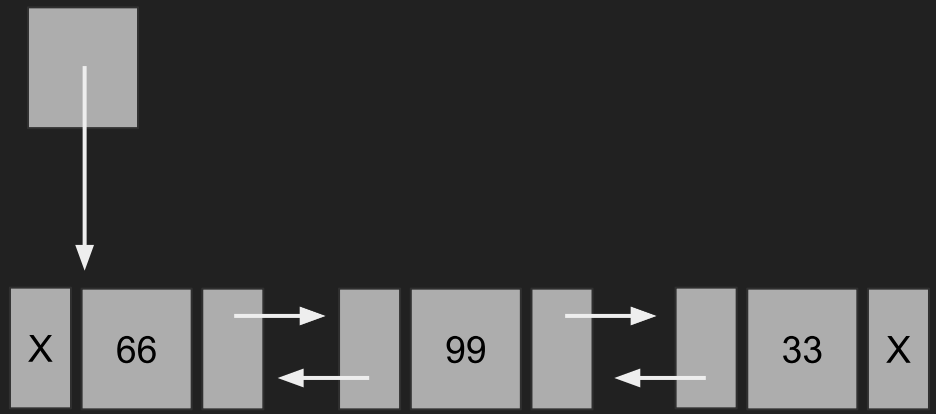
Doubly-linked list implementation

```
func Delete(s *Set, c *Cell) {
  Update prev pointer of next
  Update next pointer of prev
}
```



Doubly-linked list implementation

```
func Minimum(s *Set) *Cell {
   Scan from s.head
   Follow next ptrs remembering
      cell with min value
   Return cell with min value
}

func Maximum(s *Set) *Cell {
   Scan from s.head
   Follow next ptrs remembering
      cell with max value
   Return cell with max value
}
```

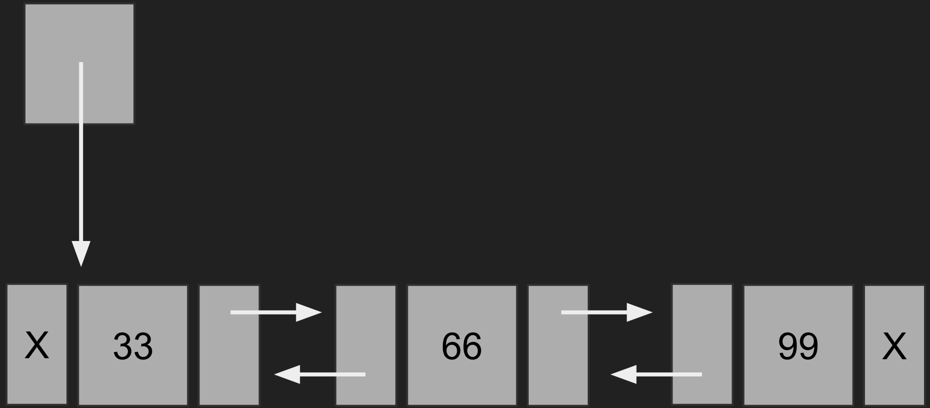


Asymptotic running times

	Linked list	Doubly-linked list
Search	$\Theta(n)$	$\Theta(n)$
Insert	$\Theta(1)$	$\Theta(1)$
Delete	$\Theta(n)$	$\Theta(1)$
Minimum	$\Theta(n)$	$\Theta(n)$
Maximum	$\Theta(n)$	$\Theta(n)$

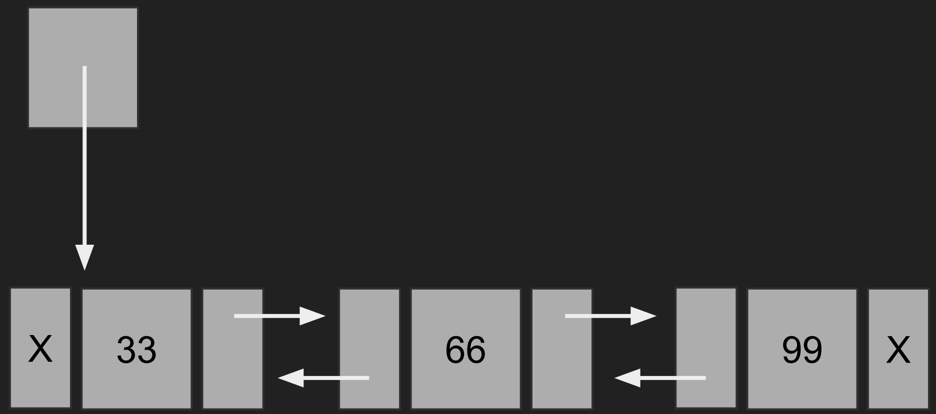
Sorted doubly-linked list implementation

```
type Set struct {
   head *Cell
}


type Cell struct {
   value int
   prev *Cell
   next *Cell
}
```



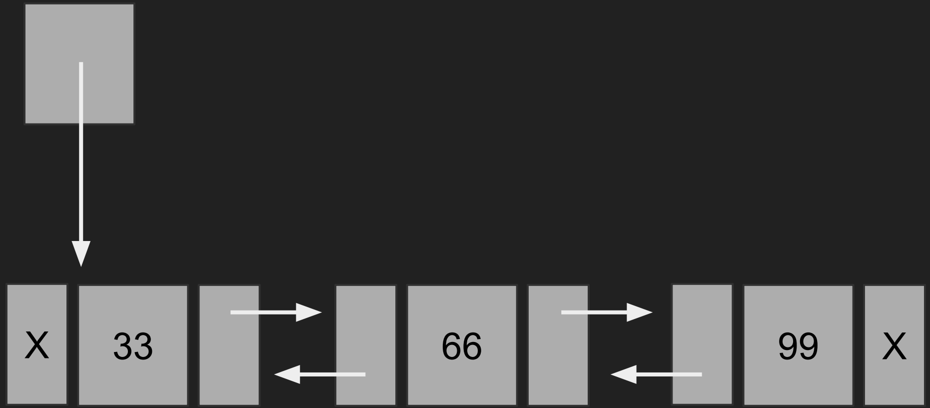
Sorted doubly-linked list implementation

```
func Search(s *Set, k int) *Cell {
   Scan from s.head
   Follow next ptrs until
      current cell value == k
   Can abort when cell value > k
}
```



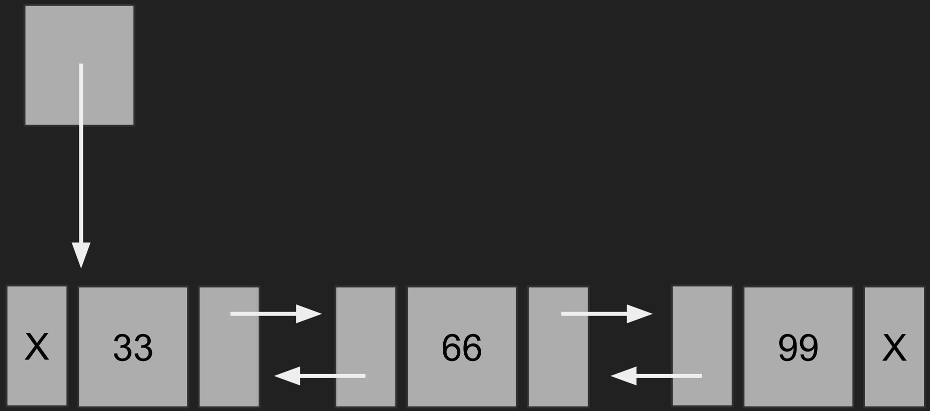
Sorted doubly-linked list implementation

```
func Insert(s *Set, c *Cell) {
   Scan from s.head
   Follow next ptrs until
      current cell value > k
   Update prev and next pointers
      of prev, next, and c
}
```



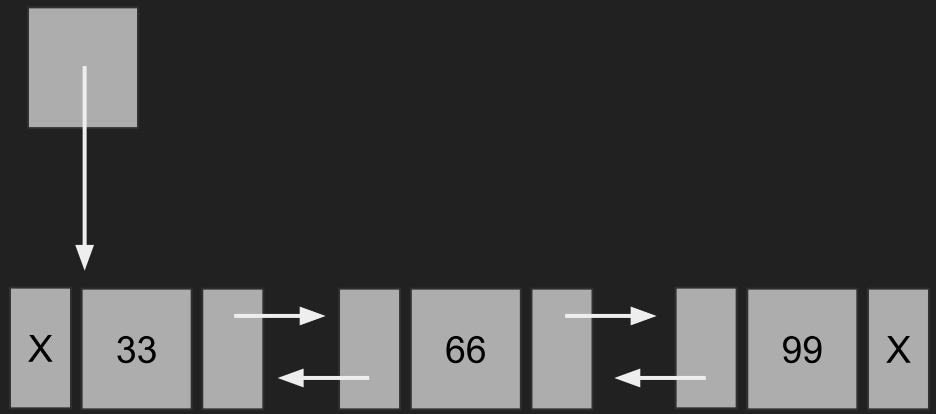
Sorted doubly-linked list implementation

```
func Delete(s *Set, c *Cell) {
  Update prev pointer of next
  Update next pointer of prev
}
```



Sorted doubly-linked list implementation

```
func Minimum(s *Set) *Cell {
    Return first cell
}

func Maximum(s *Set) *Cell {
    Scan from s.head
    Follow next ptrs until
        reaching last cell
    Return last cell
}
```

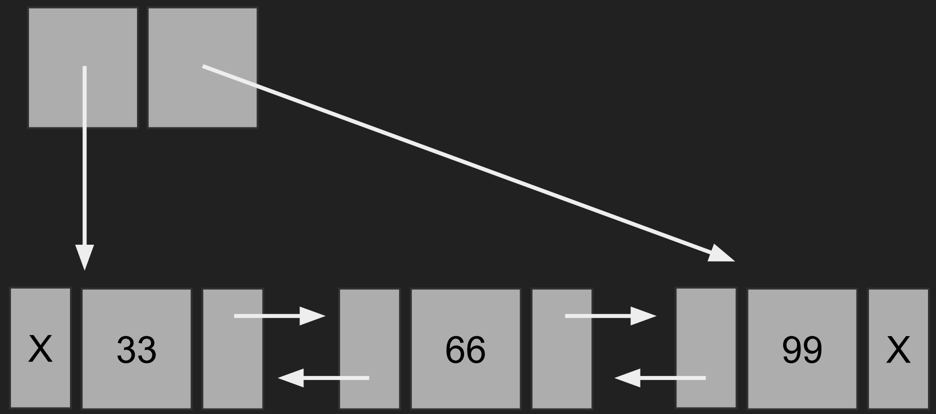


Asymptotic running times

	Linked list	Doubly-linked list	Sorted Doubly-linked list
Search	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Insert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Delete	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Minimum	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Maximum	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

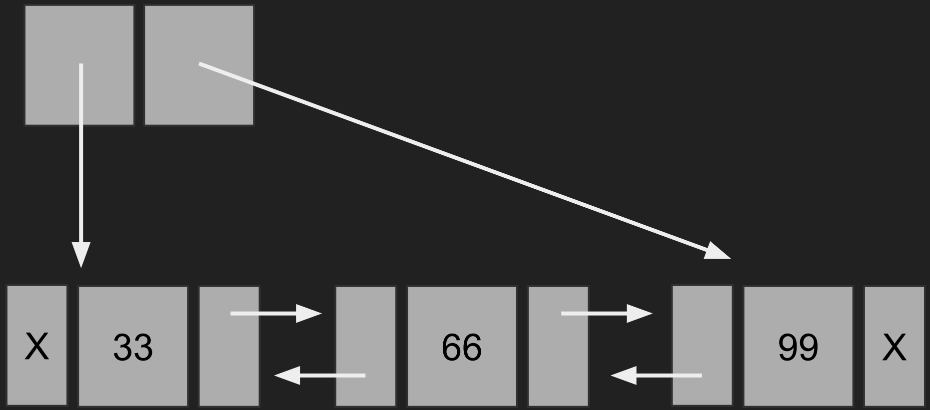
Sorted doubly-linked list implementation with tail pointer

```
type Set struct {
    head *Cell
    tail *Cell
}


type Cell struct {
    value int
    prev *Cell
    next *Cell
}
```



Sorted doubly-linked list implementation with tail pointer

```
func Minimum(s *Set) *Cell {
    Return first cell
}

func Maximum(s *Set) *Cell {
    Return last cell
}
```



Asymptotic running times

	Linked list	Doubly-linked list	Sorted Doubly-linked list
Search	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Insert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Delete	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Minimum	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Maximum	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$ $\Theta(1)$ with tail pointer

Asymptotic running times

	Linked list	Doubly-linked list	Sorted Doubly-linked list
Search	Θ(n)	Θ(n)	Θ(n)
Insert	Θ(1)	Θ(1)	Θ(n)
Delete	Θ(n)	Θ(1)	Θ(1)
Minimum	Θ(n)	Θ(n)	Θ(1)
Maximum	Θ(n)	Θ(n)	Θ(n) Θ(1) with tail pointer

Binary Search

How can we improve the Θ(n) running time for Search and Insert?

We can take inspiration from the notion of binary search on an ordered array:

11	22	33	44	55	66	77	88	99	101	104	109

BinarySearch (X) =

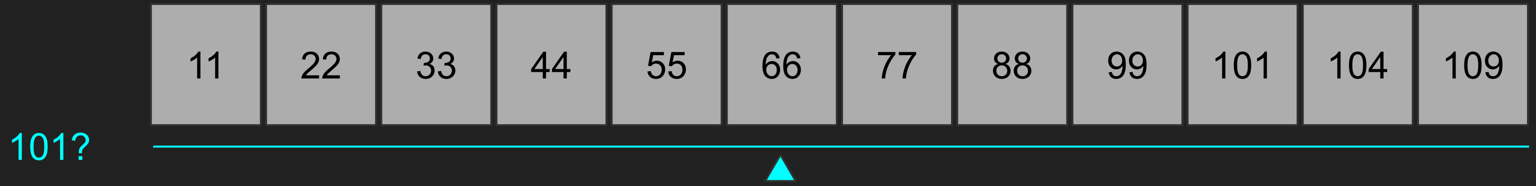
1. Get the value in the middle of the array — if X we're DONE
2. if X < value, recursively look for X in the left part of the array
3. if X > value, recursively look for X in the right part of the array

We divide the "search space" by half every iteration — $\Theta(\log_2 n)$ iterations

Binary Search

How can we improve the Θ(n) running time for Search and Insert?

We can take inspiration from the notion of binary search on an ordered array:



BinarySearch (X) =

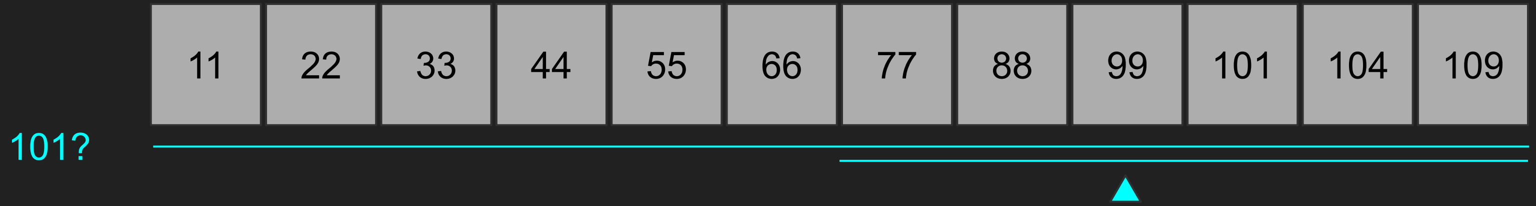
1. Get the value in the middle of the array — if X we're DONE
2. if X < value, recursively look for X in the left part of the array
3. if X > value, recursively look for X in the right part of the array

We divide the "search space" by half every iteration — $\Theta(\log_2 n)$ iterations

Binary Search

How can we improve the Θ(n) running time for Search and Insert?

We can take inspiration from the notion of binary search on an ordered array:



BinarySearch (X) =

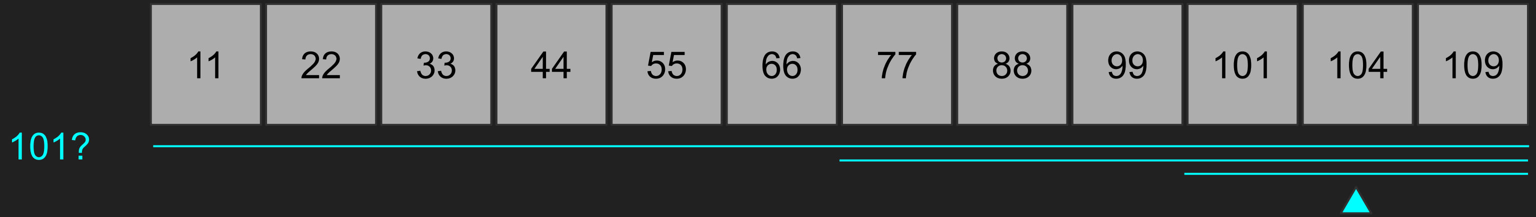
1. Get the value in the middle of the array — if X we're DONE
2. if X < value, recursively look for X in the left part of the array
3. if X > value, recursively look for X in the right part of the array

We divide the "search space" by half every iteration — $\Theta(\log_2 n)$ iterations

Binary Search

How can we improve the $\Theta(n)$ running time for Search and Insert?

We can take inspiration from the notion of binary search on an ordered array:



BinarySearch (X) =

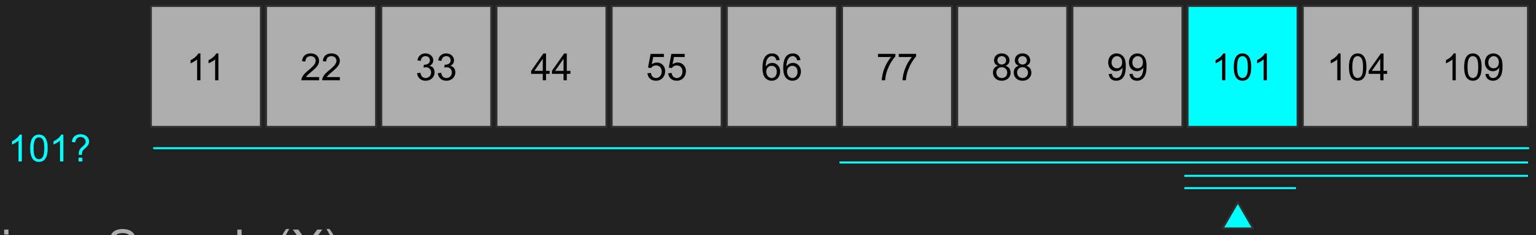
1. Get the value in the middle of the array — if X we're DONE
2. if X < value, recursively look for X in the left part of the array
3. if X > value, recursively look for X in the right part of the array

We divide the "search space" by half every iteration — $\Theta(\log_2 n)$ iterations

Binary Search

How can we improve the Θ(n) running time for Search and Insert?

We can take inspiration from the notion of binary search on an ordered array:



BinarySearch (X) =

1. Get the value in the middle of the array — if X we're DONE
2. if X < value, recursively look for X in the left part of the array
3. if X > value, recursively look for X in the right part of the array

We divide the "search space" by half every iteration — $\Theta(\log_2 n)$ iterations

Trees by way of graphs

An (undirected) graph is a finite set V of vertices and a finite set $E \subseteq V \times V$ of edges between vertices, viewed as unordered pairs.

Vertices v_1 and v_2 are linked, written $v_1 \sim v_2$, when $(v_1, v_2) \in E$

Two vertices v_1 and v_2 are connected if there is a sequence of vertices $u_1, \ldots, u_k$ such that $v_1 \sim u_1 \sim \ldots \sim u_k \sim v_2$

A cycle is a sequences of at least three distinct vertices $v_1, v_2, \ldots, v_k$ such that $v_1 \sim v_2 \sim \ldots \sim v_k \sim v_1$

Trees

A tree is an undirected graph in which:

- there is no cycle
- every pair of vertices are connected

We use the term nodes to refer to vertices in a tree

A rooted tree is a tree with one distinguished node we call the root

A rooted tree has the property:

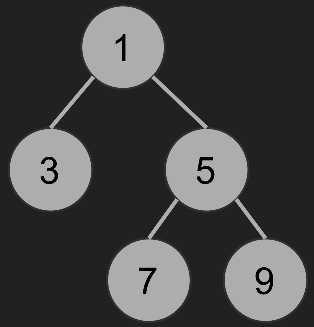
for every node X there is a unique path from X to the root of the tree

The children of node X are the nodes linked to X *not* on the path to the root

(We'll only care about rooted trees, and just refer to them as trees from now on)

Trees

We usually draw trees with the root at the top and children going down:



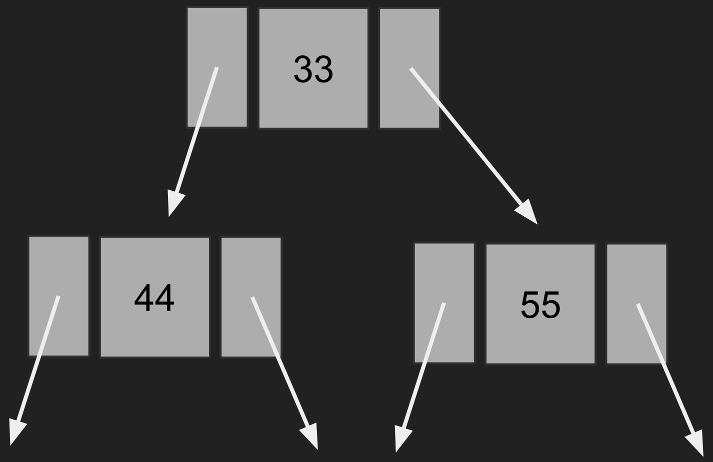
We are going to consider binary trees, in which nodes have at most two children

We are going to store values at the nodes

A data structure for representing binary trees

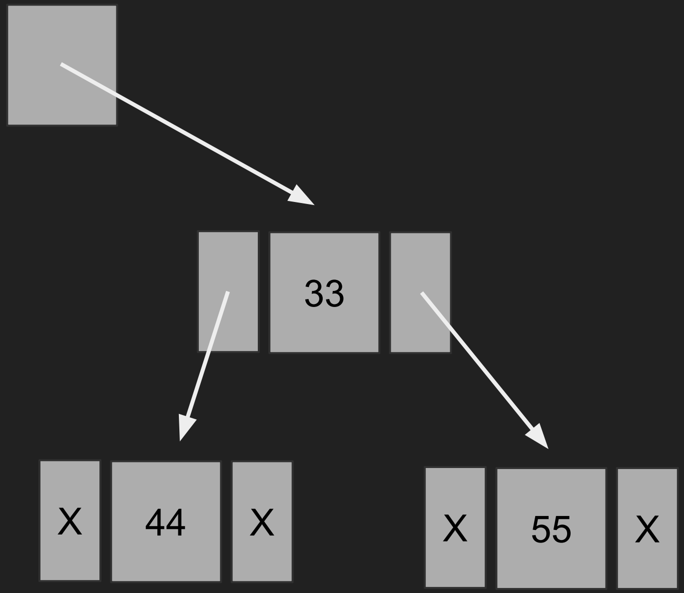
We can use a linked structure to represent a binary tree

Each "cell" has a field for the value, and a field with a pointer to the left and the right children cells



A data structure for representing binary trees

```
type Tree struct {
   root *Cell
}


type Cell struct {
    value int
    left *Cell
    right *Cell
}
```



A data structure for representing binary trees

```
type Tree struct {
   root *Cell
}


type Cell struct {
    value int
    left *Cell
    right *Cell
}
```

What strategy can we use to put values into a binary tree such that we can retrieve them quickly?

Inspiration: binary search

The binary search tree property

A node X in a binary tree has the binary search tree (BST) property if:

- every node in the left subtree of X has a value less than the value at X
- every node in the right subtree of X has a value greater than the value at X

A binary tree is a binary search tree if every node in the tree has the BST property

A binary search tree is a bit like an ordered linked list:

it is structurally a binary tree in which the elements respect some kind of order

Searching in a binary search tree

If you have a binary search tree T, then you can search for key K by navigating down the tree, following left or right child pointers according to the values seen:

- start at the root
- if K is the value in the node you're at, you're DONE — return the node
- if K is less than the value at the node, go to the left child
- if K is more than the value at the node, go to the right child
- repeat until there are no more nodes to follow and report NOT FOUND

This takes time $\Theta(\text{height}(T))$, the height of tree T

Inserting into a binary search tree

To insert object V into a binary search tree, you first search for V (via its key) and when you fail to find it, you insert a new node with value V at the leaf where you failed to find V:

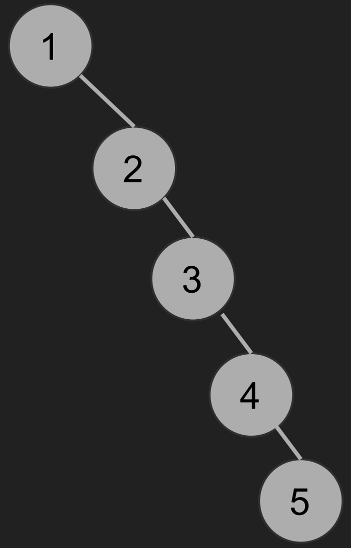
- start at the root
- if V is at the node you're at, you're inserting an existing value — STOP
- if V is less than the value at the node, go to the left child
- if V is more than the value at the node, go to the right child
- repeat until there are no more nodes to follow
- insert a node as the left or right child (depending on V) of the last node visited

Again, this takes time $\Theta(\text{height}(T))$, the height of tree T

Does this help?

Search and Insert are Θ(height(N)) operations, where height(N) is the worst-case height of a binary search tree with N nodes

Consider this perfectly valid binary search tree:



The worst case height of a binary search tree with N nodes is… N

⇒ Search and Insert are Θ(N) operations!

⇒ we haven't gained anything… yet!