Binary Search Trees

Implementing **`Set`** and **`Map`**

- So far we've seen how to implement the **`HashMap`**
- Let's now turn our attention to the **`Set`** and **`Map`**.
- Major operations:
  - Insert
  - Remove
  - Contains

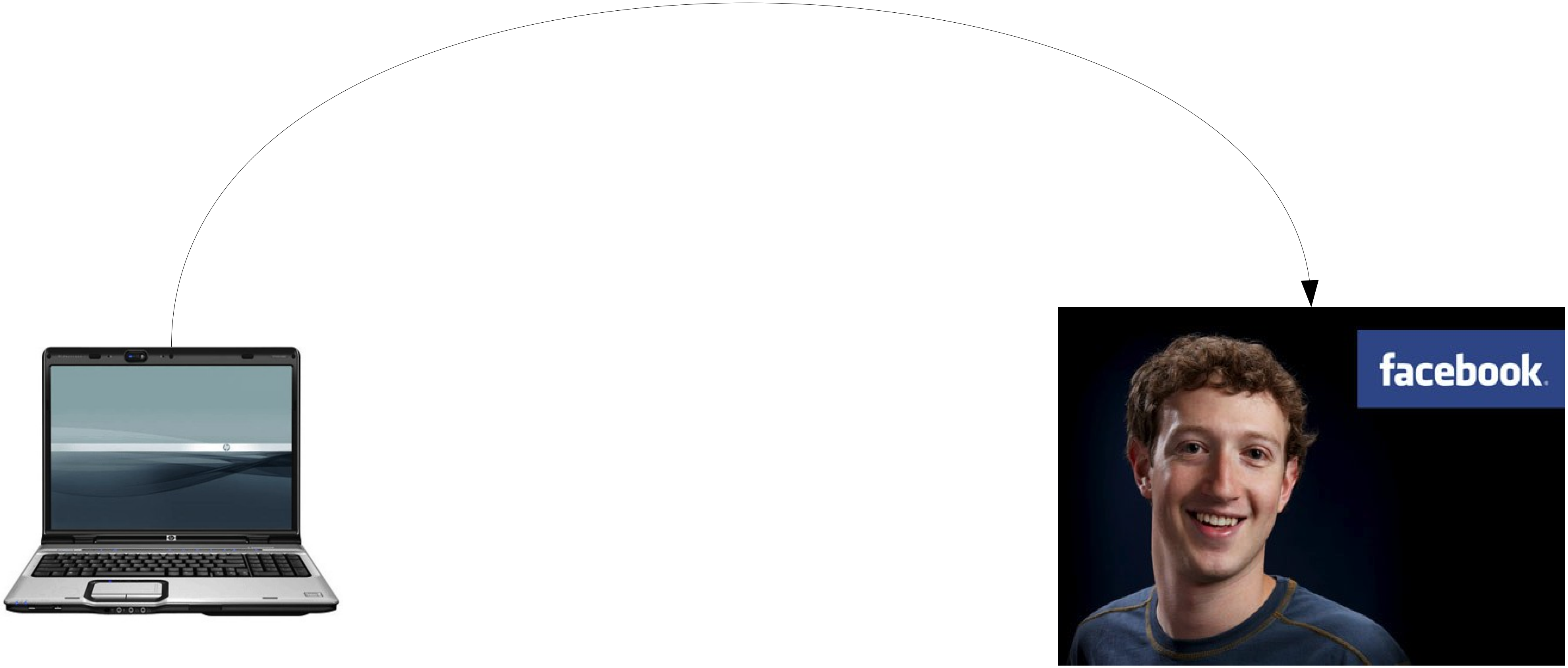
Goals for Set

- Fast insert, contains, remove
  - “Fast” = better than O(n)

Goals for Set

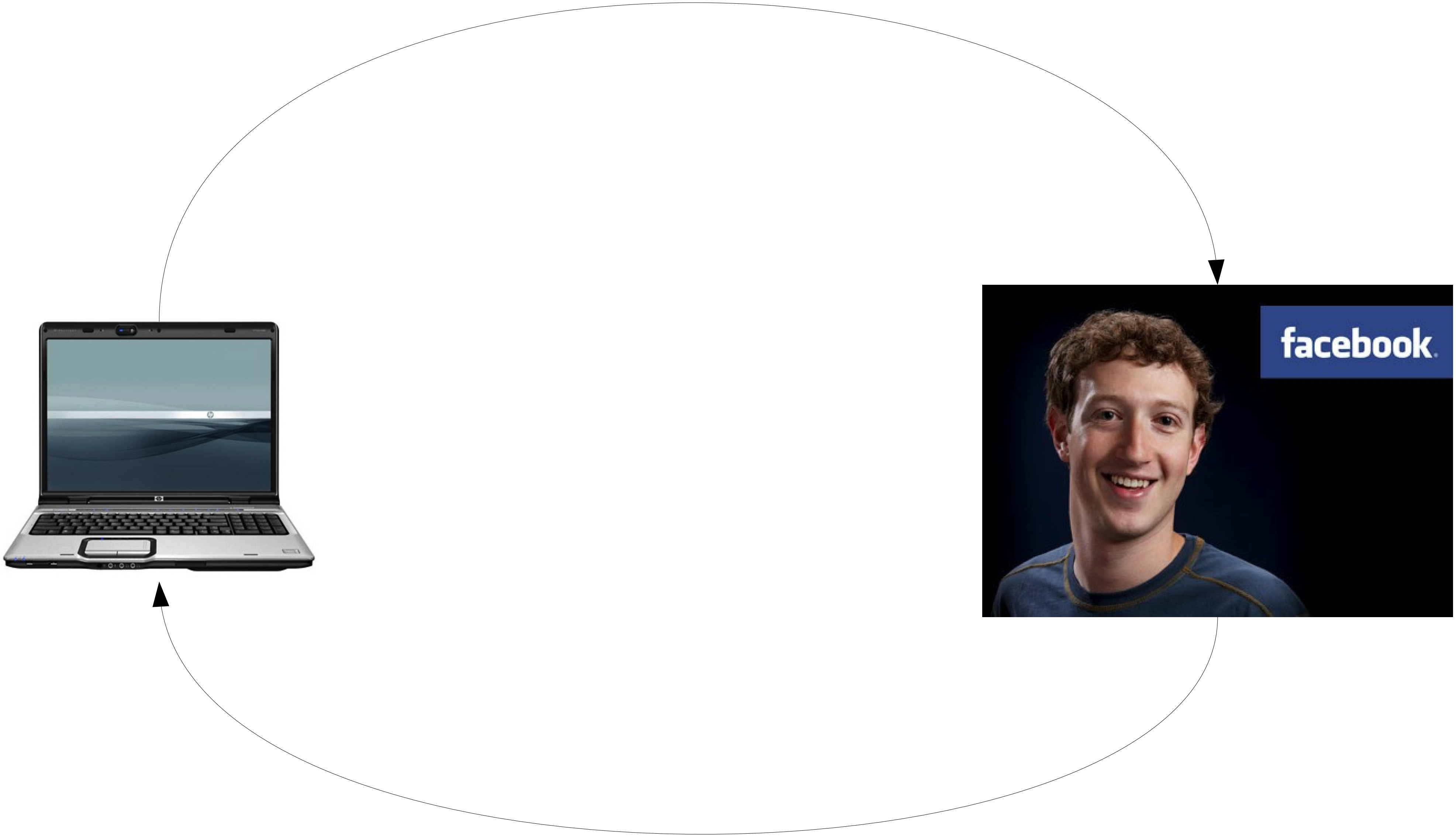
- Fast insert, contains, remove
  - “Fast” = better than O(n)
- To have our data be stored in sorted order.
  - Why would we want this?


facebook



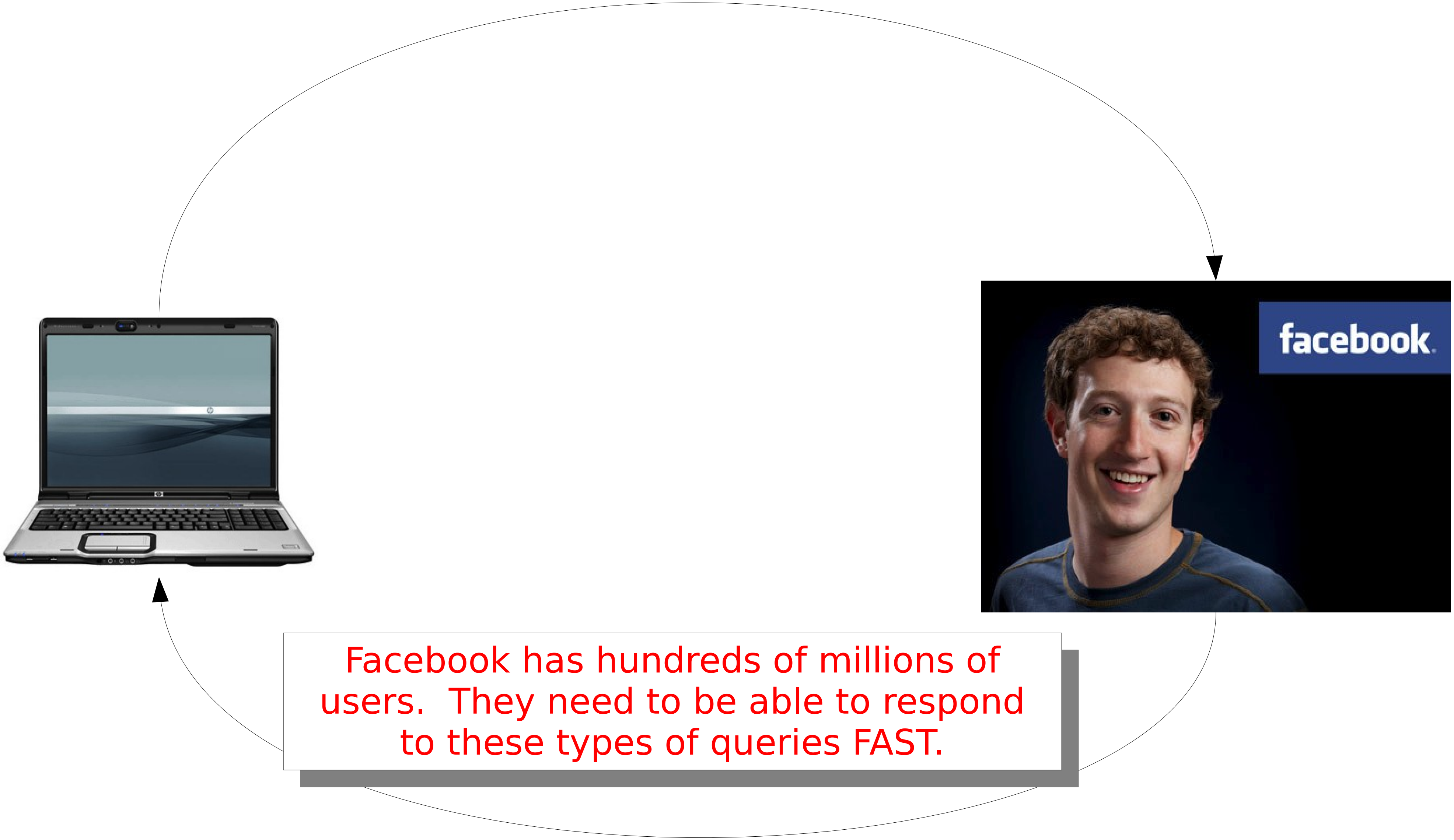
Yo Mark, give me all my facebook
friends whose names start with 'K'

Yo Mark, give me all my facebook friends whose names start with 'K'



“Karen, Kara, Kaylee, Keith, Kevin, Kyle”

Yo Mark, give me all my facebook
friends whose names start with 'K'



Facebook has hundreds of millions of users. They need to be able to respond to these types of queries FAST.

“Karen, Kara, Kaylee, Keith,
Kevin, Kyle”

If Names in a Sorted Array

...	Jack	Karen	Kara	Kaylee	Keith	Kevin	Kyle	...

If Names in a Sorted Array

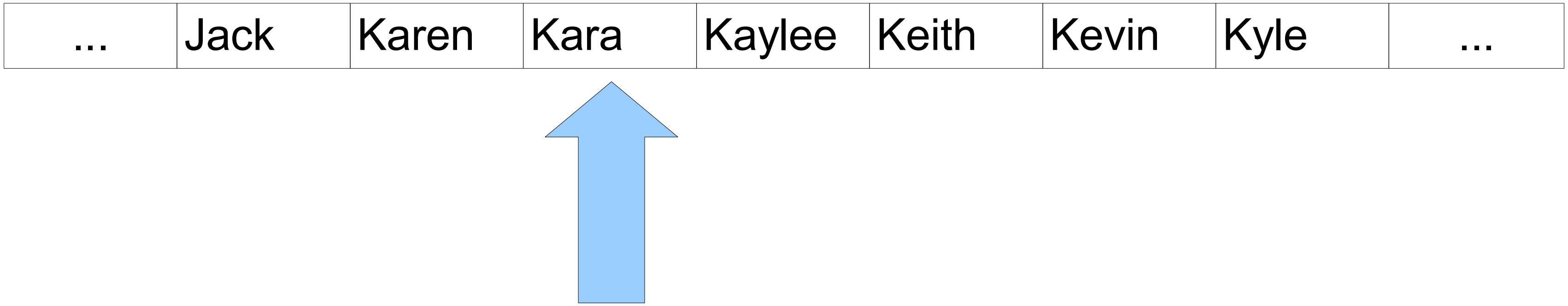
...	Jack	Karen	Kara	Kaylee	Keith	Kevin	Kyle	...

Binary Search finds first friend whose
name starts with 'K' in O(log(n)) time

If Names in a Sorted Array

...	Jack	Karen	Kara	Kaylee	Keith	Kevin	Kyle	...

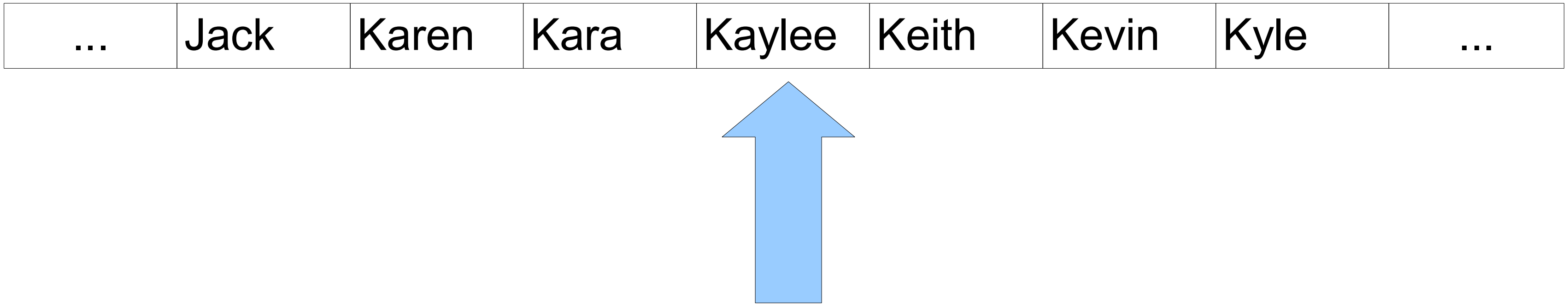
If Names in a Sorted Array



...	Jack	Karen	Kara	Kaylee	Keith	Kevin	Kyle	...

Friends: Karen,

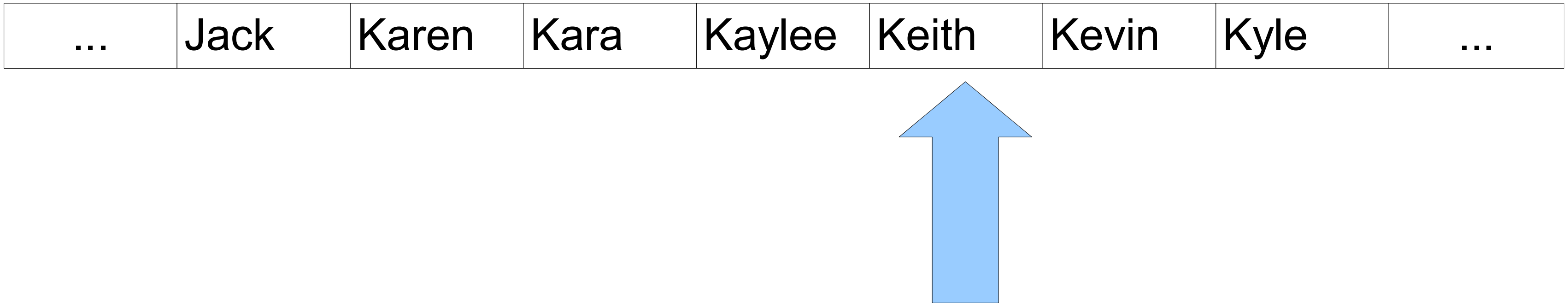
If Names in a Sorted Array



...	Jack	Karen	Kara	Kaylee	Keith	Kevin	Kyle	...

Friends: Karen, Kara,

If Names in a Sorted Array



...	Jack	Karen	Kara	Kaylee	Keith	Kevin	Kyle	...

Friends: Karen, Kara, Kaylee,

If Names in a Sorted Array

...	Jack	Karen	Kara	Kaylee	Keith	Kevin	Kyle	...

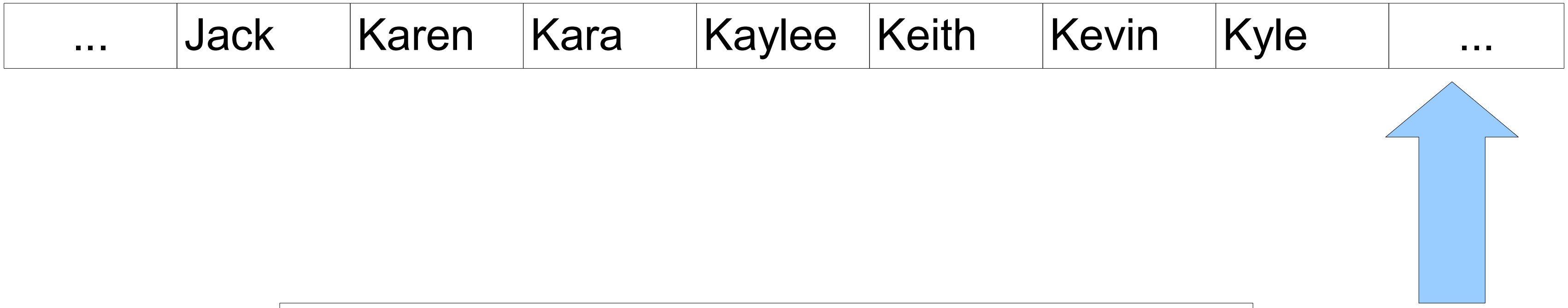
Friends: Karen, Kara, Kaylee, Keith

If Names in a Sorted Array

...	Jack	Karen	Kara	Kaylee	Keith	Kevin	Kyle	...

Friends: Karen, Kara, Kaylee, Keith, Kevin

If Names in a Sorted Array



...	Jack	Karen	Kara	Kaylee	Keith	Kevin	Kyle	...

If names were not sorted, then we would need to perform an O(n) linear search for all friends whose names start with 'K'

Friends: Karen, Kara, Kaylee, Keith, Kevin, Kyle

Range Query

- A **range query** is a request for all values within a range.
  - Databases: “Give me all Facebook friends who have posted a status update in the past week”
  - Data Mining/Machine Learning: “Give me all training instances 'close to' this test instance”
  - Computer Graphics: “Give me all the geometry within this neighborhood”
- If your data is sorted, then range queries can be executed very quickly.

Array Implementation

- We could implement the **`Set`** as a list of all the values it contains.
- To add an element: **O(*n*)**
  - Check if the element already exists.
  - If not, append it.
- To remove an element: **O(*n*)**
  - Find and remove it from the list.
- To see if an element exists: **O(log *n*)**
  - Search the list for the element.

Using Hashing

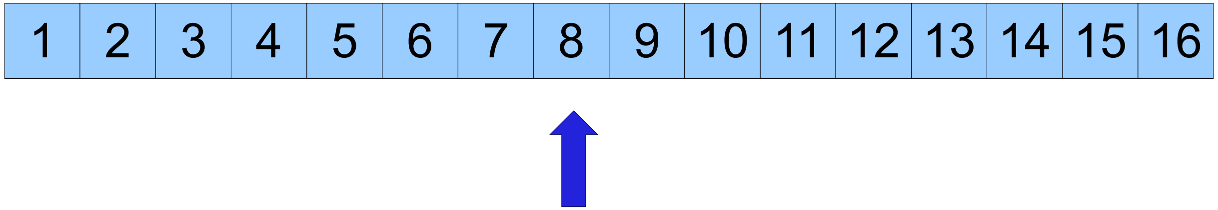
- If we have a hash function for the elements being stored, we can implement a **`Set`** using a hash table.
- What is the expected time to insert a value?
  - Answer: **O(1)**.
- What is the expected time to remove a value?
  - Answer: **O(1)**.
- What is the expected time to check if a value exists?
  - Answer: **O(1)**.
- When a **`Set`** is implemented using hashing it is called a **`HashSet`**
  - Effective implementation, elements are not sorted.

An Entirely Different Approach

`x = 6`

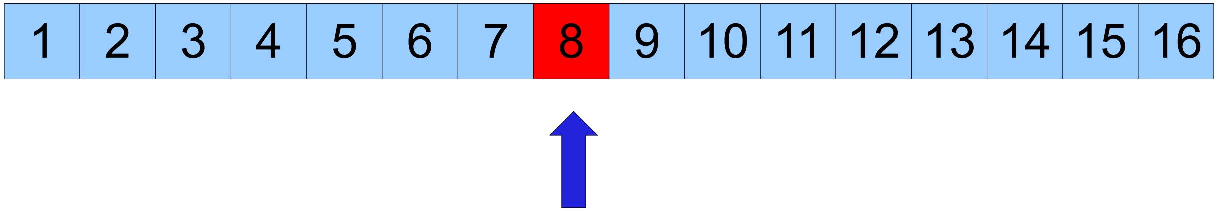
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

`x = 6`



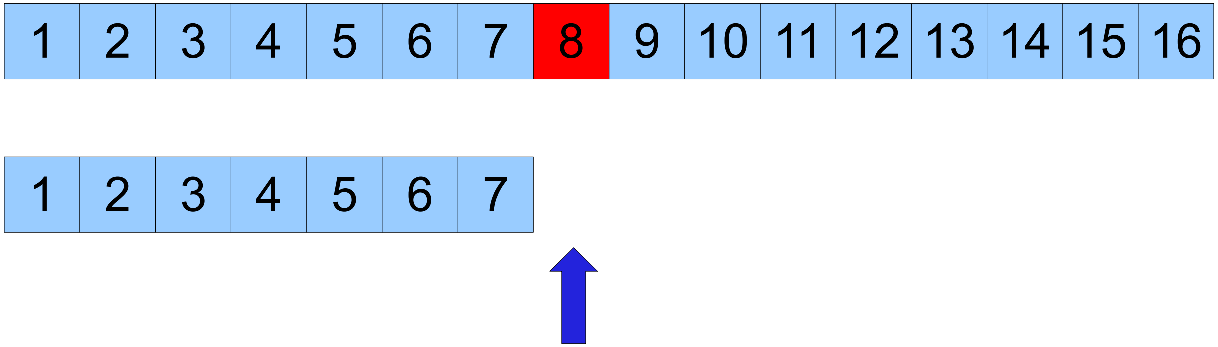
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

`x = 6`



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

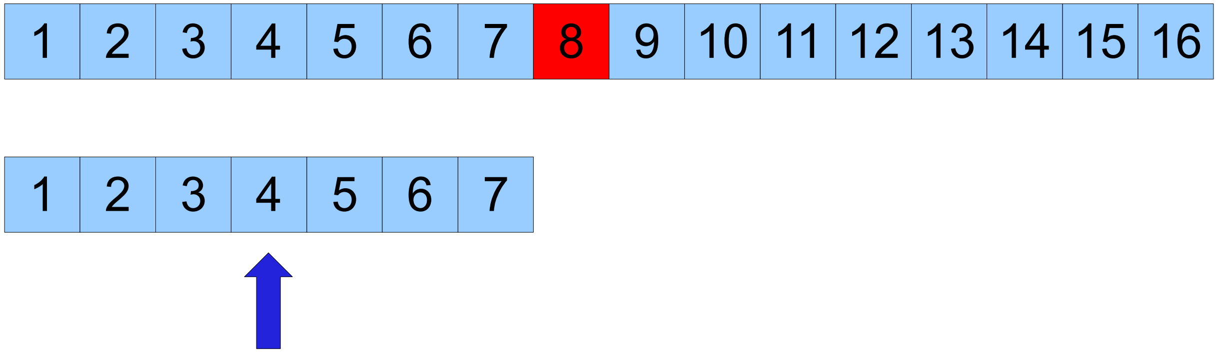
`x = 6`



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1	2	3	4	5	6	7

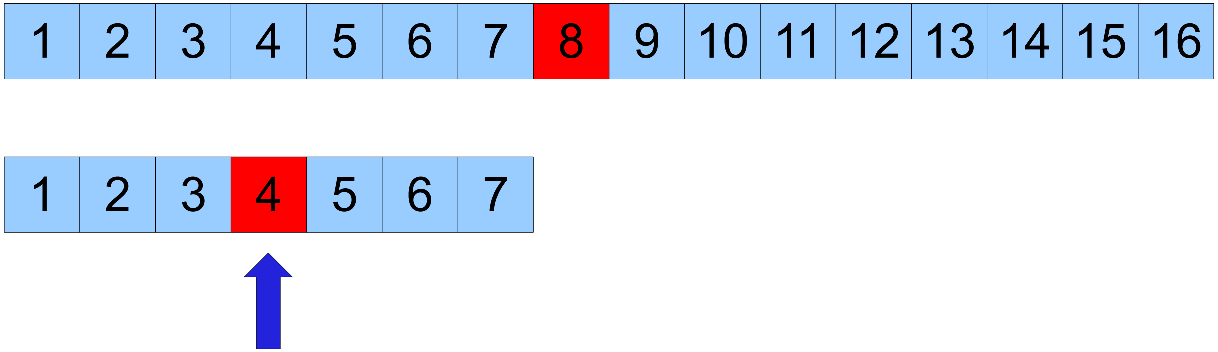
`x = 6`



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1	2	3	4	5	6	7

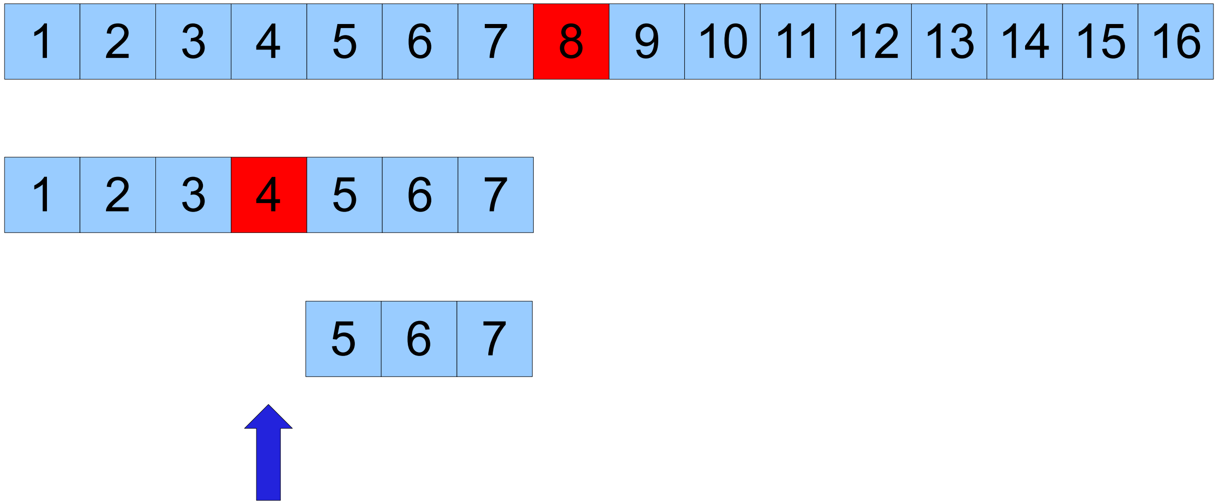
`x = 6`



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1	2	3	4	5	6	7

`x = 6`

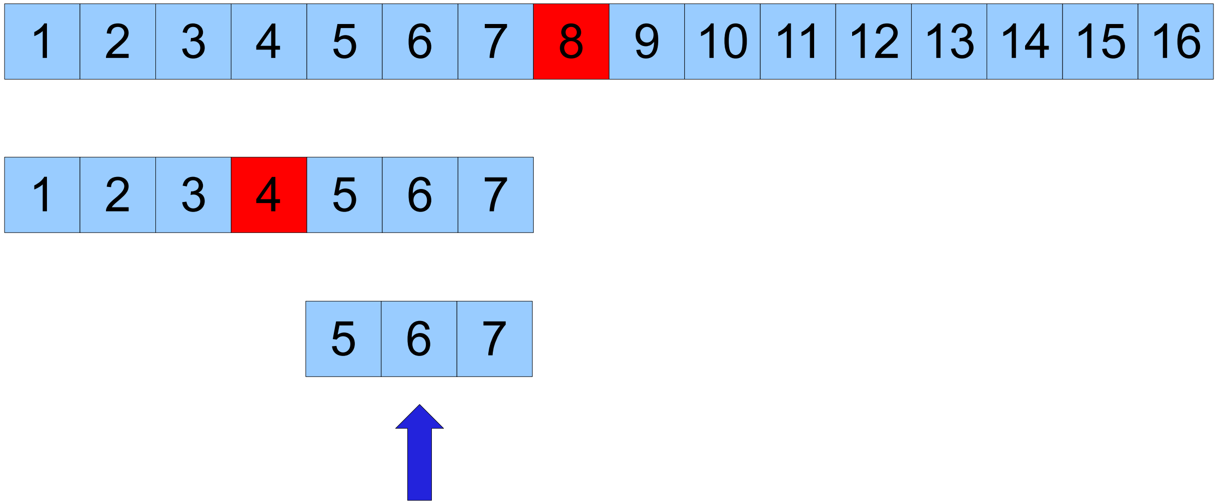


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1	2	3	4	5	6	7

5	6	7

`x = 6`

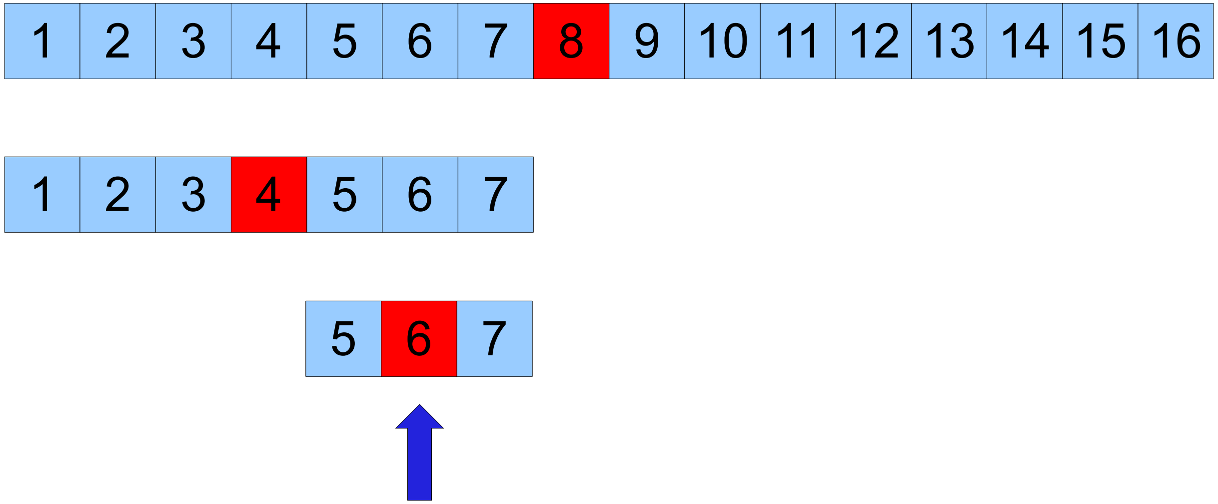


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

1 2 3 4 5 6 7

5 6 7

`x = 6`

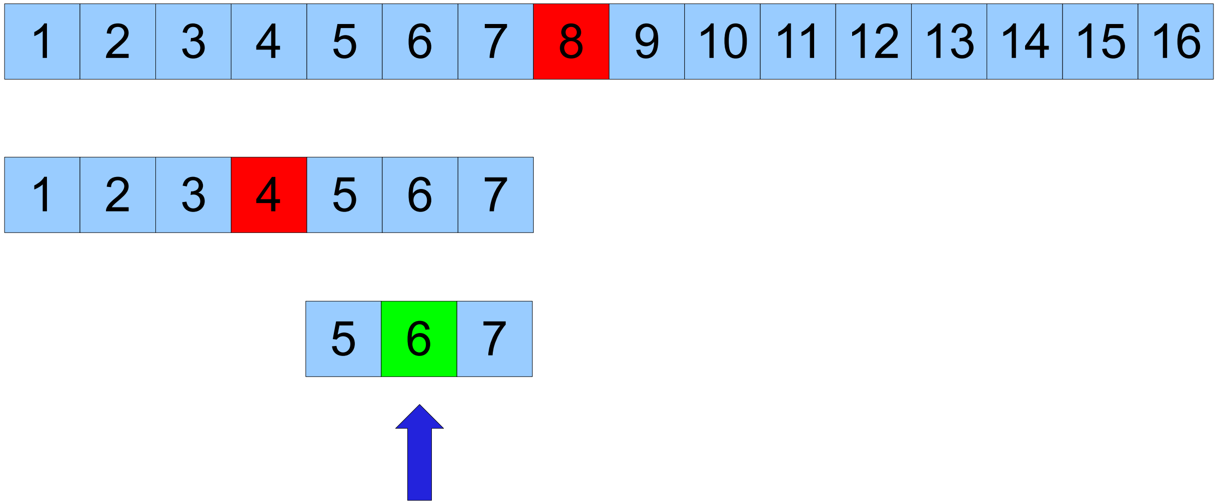


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1	2	3	4	5	6	7

5	6	7

`x = 6`



1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

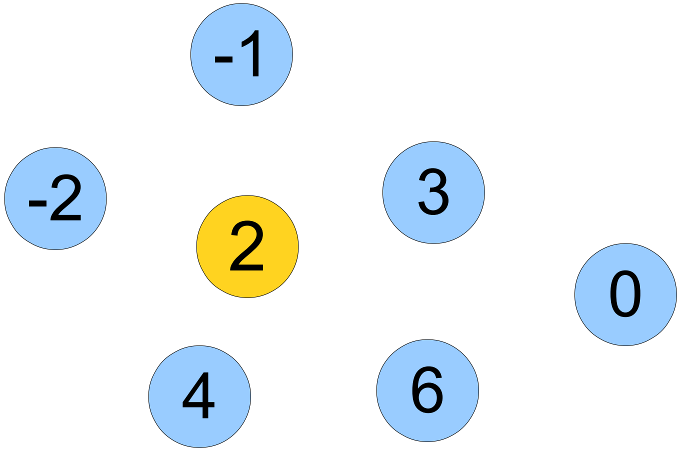
1 2 3 4 5 6 7

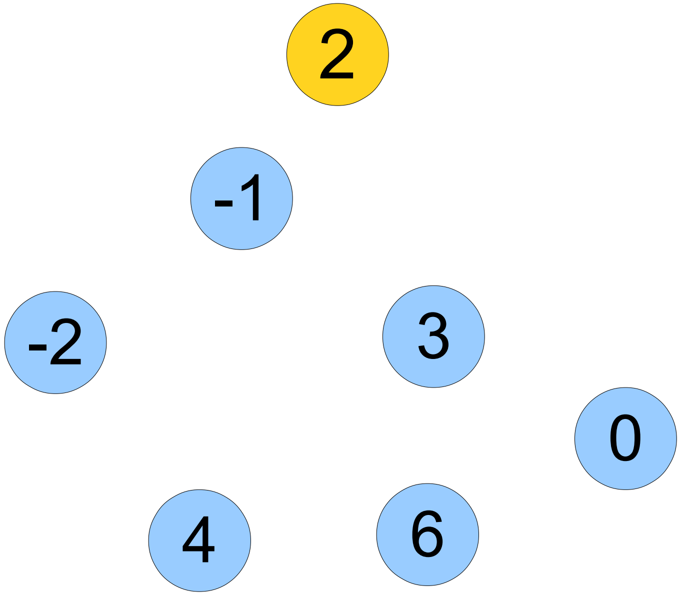
5 6 7

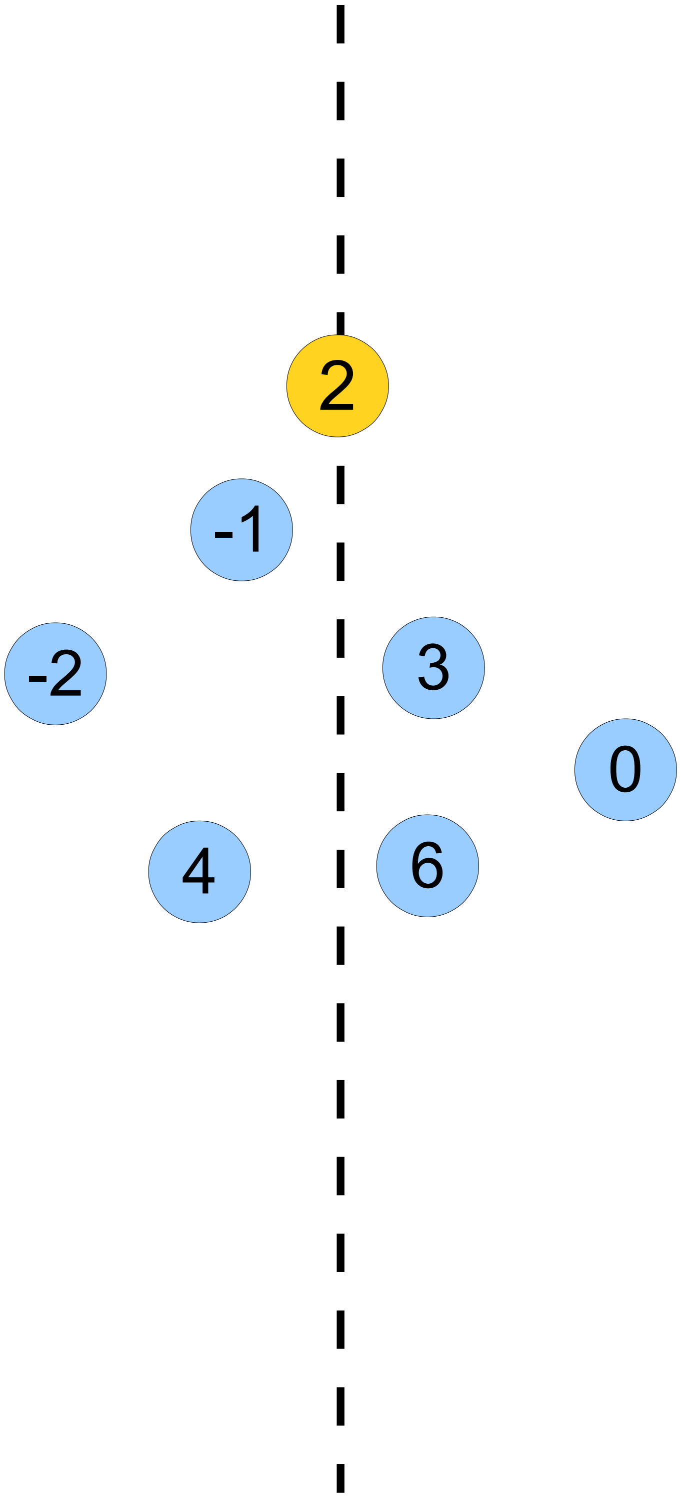
Inspiration: Binary Search

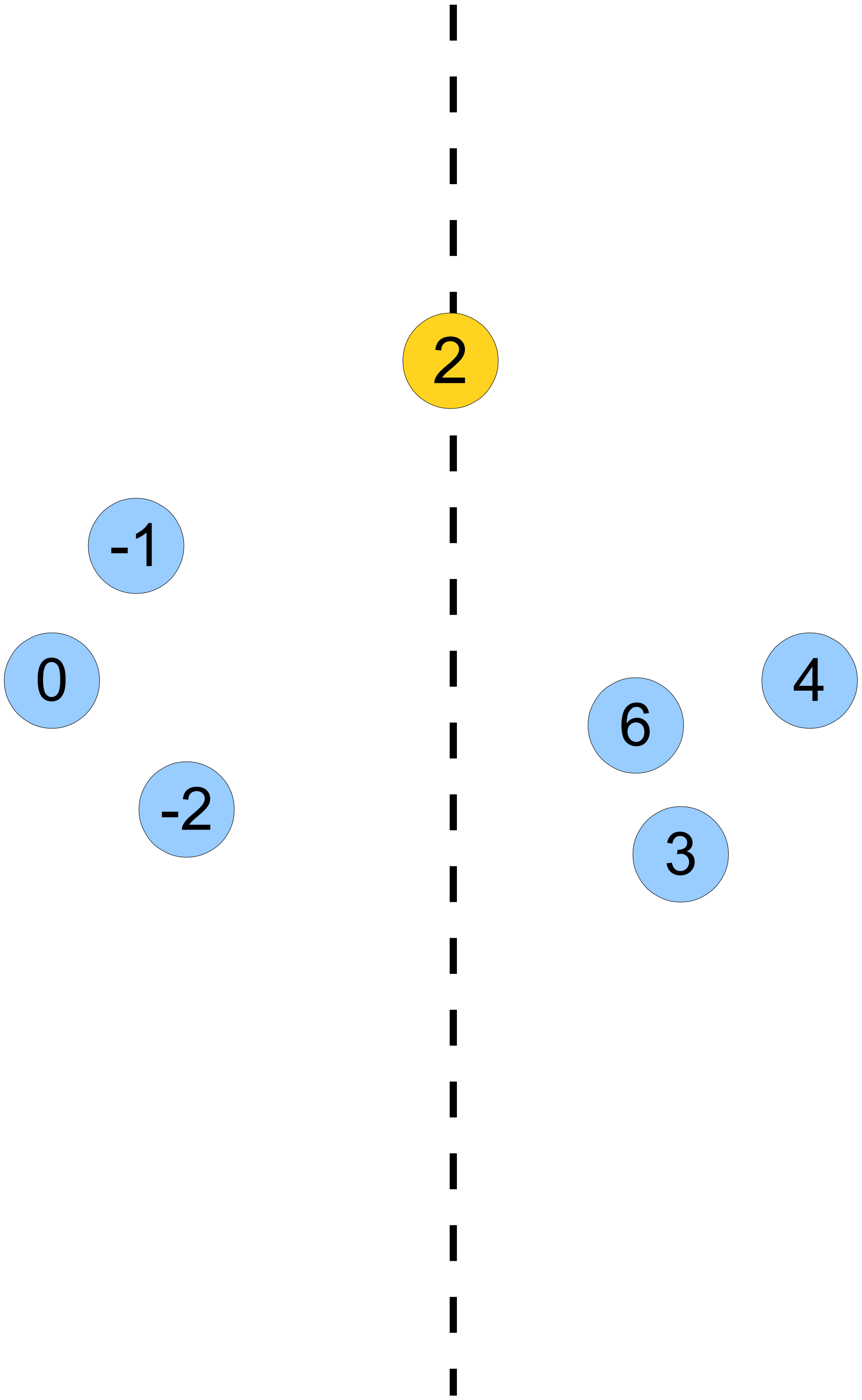
- Binary search is so fast because at every step we are able to discard half of the remaining elements.
- Let's try to do something similar!
- Note: There are 2 ways to “derive” the structure I'm about to show you. I'll show you both.

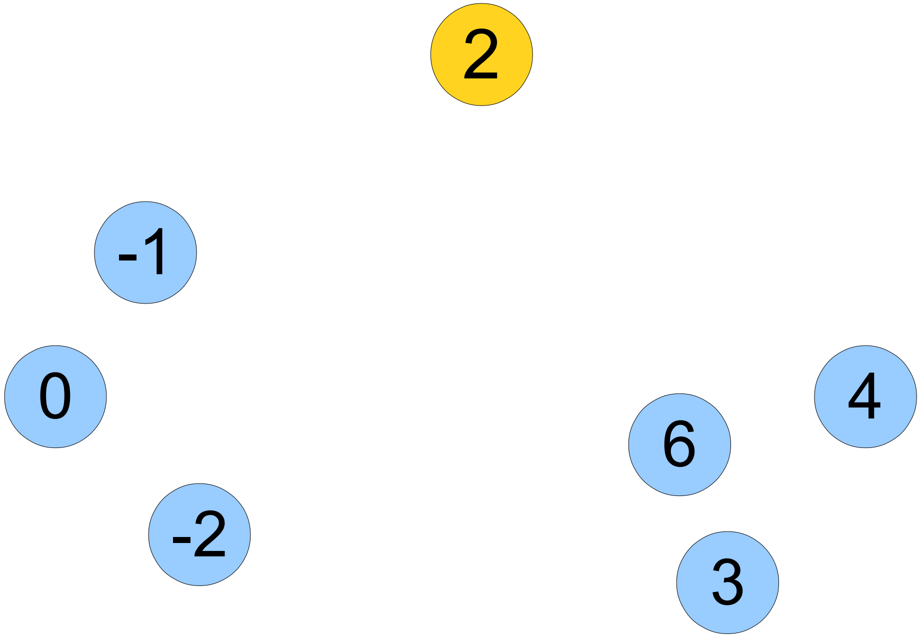
Derivation 1

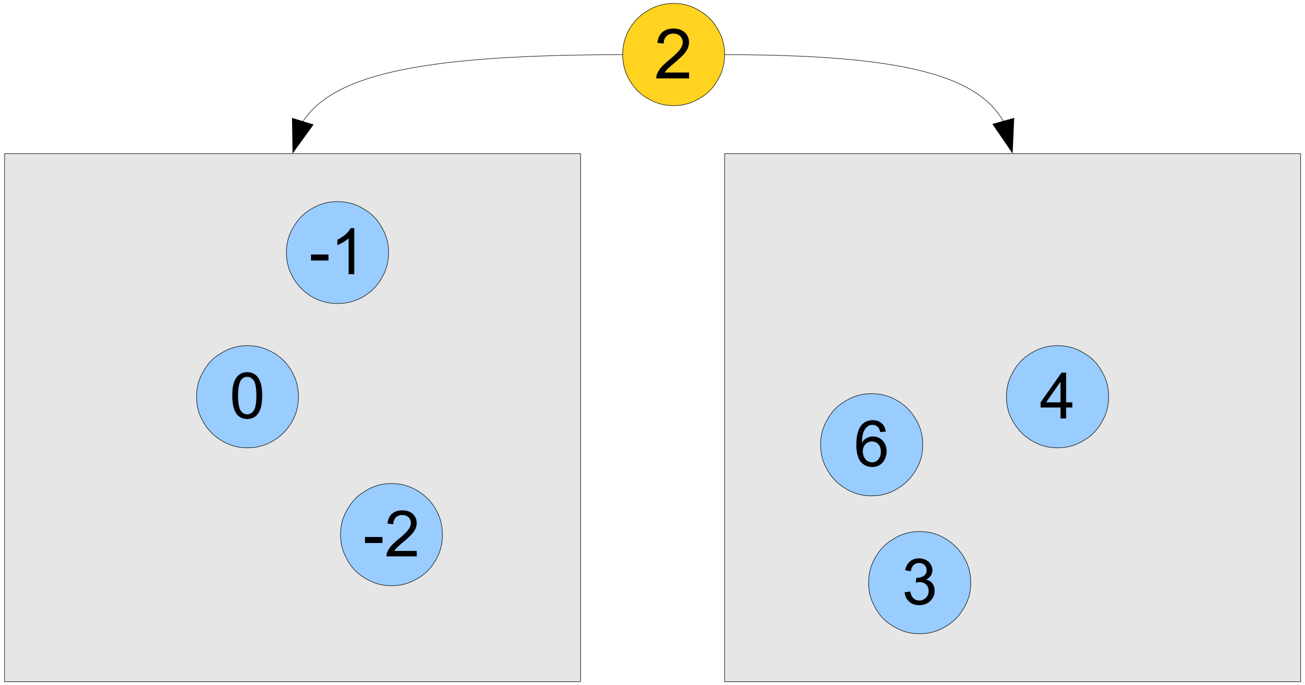

-1
-2
2
3
0
4
6

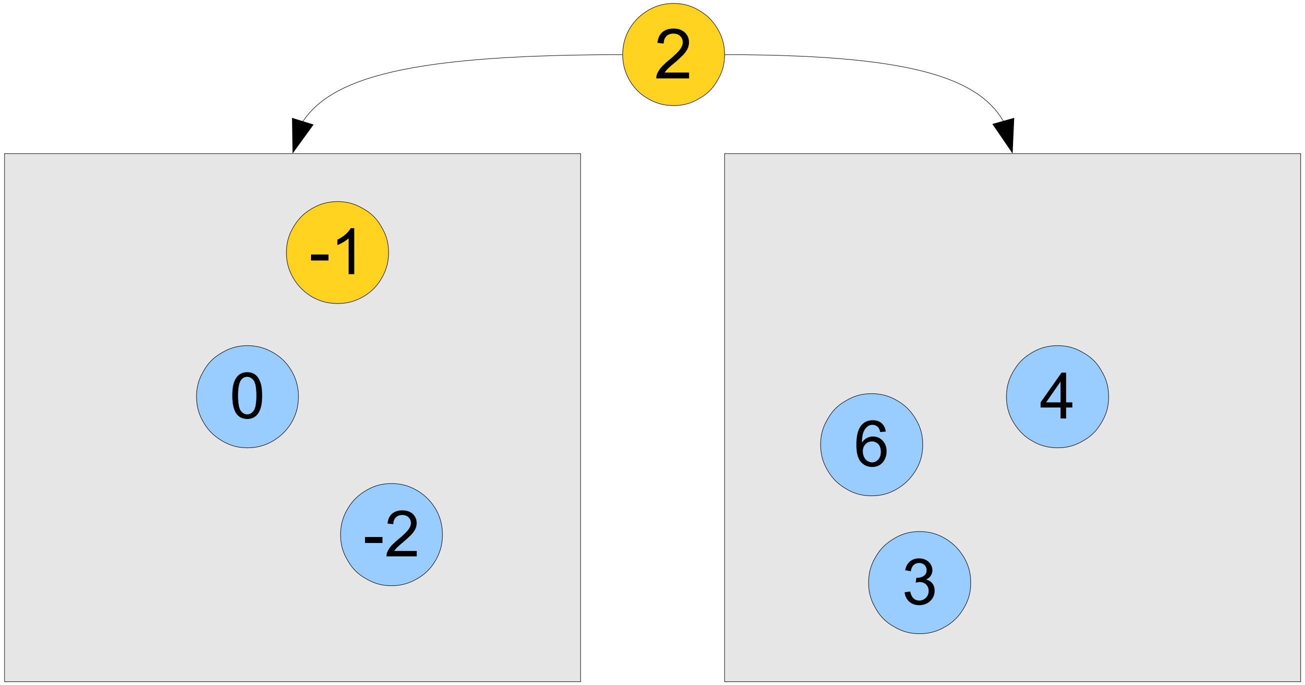

2
-1
-2
3
0
4
6

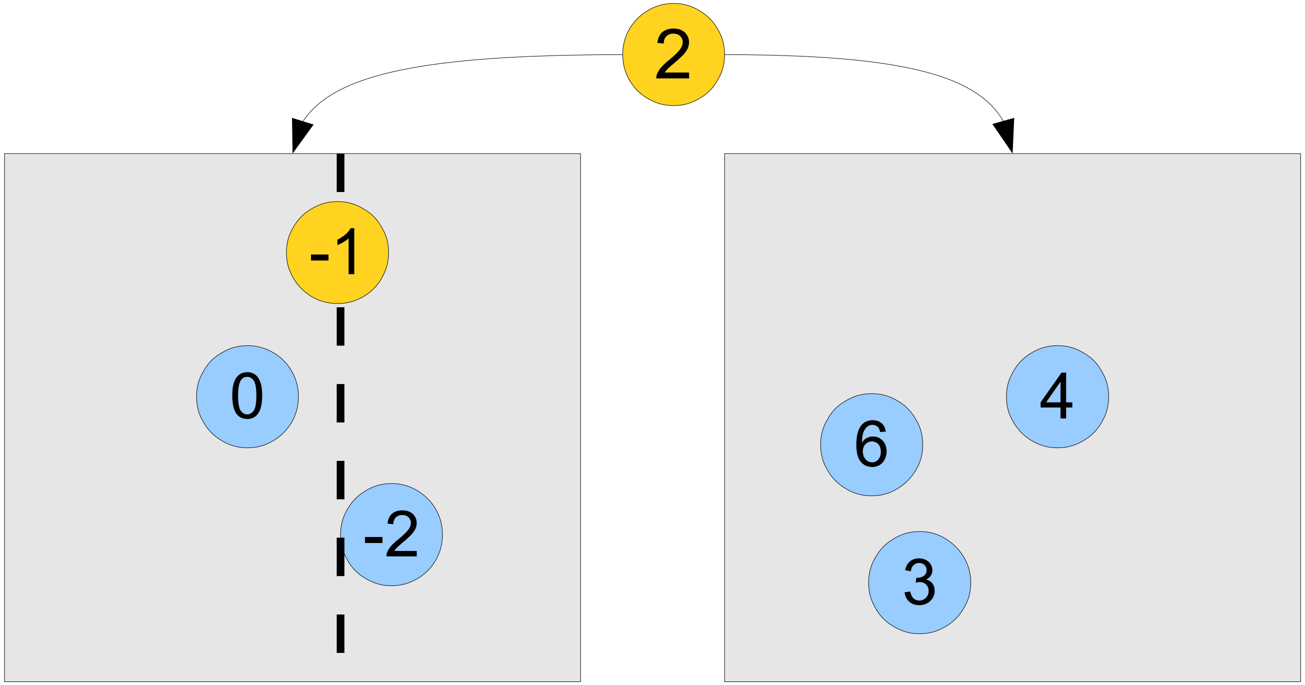

2
-1
-2
3
0
4
6

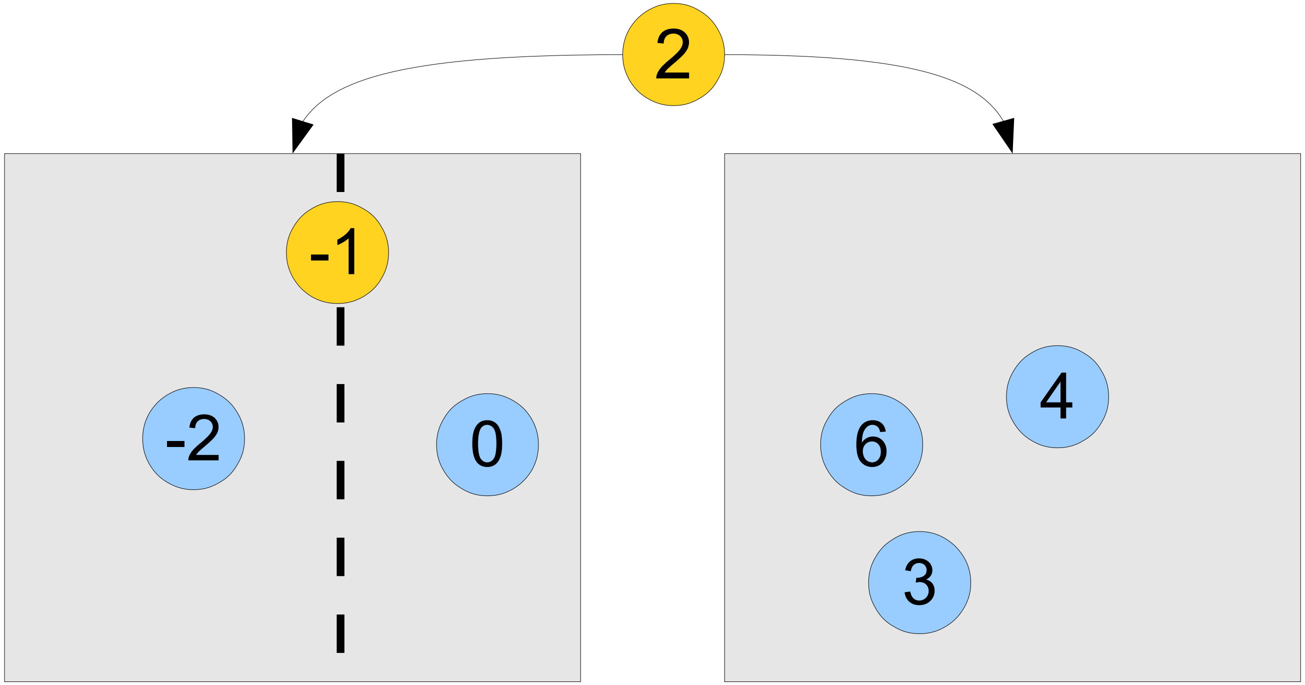

2
-1
0
-2
6
4
3

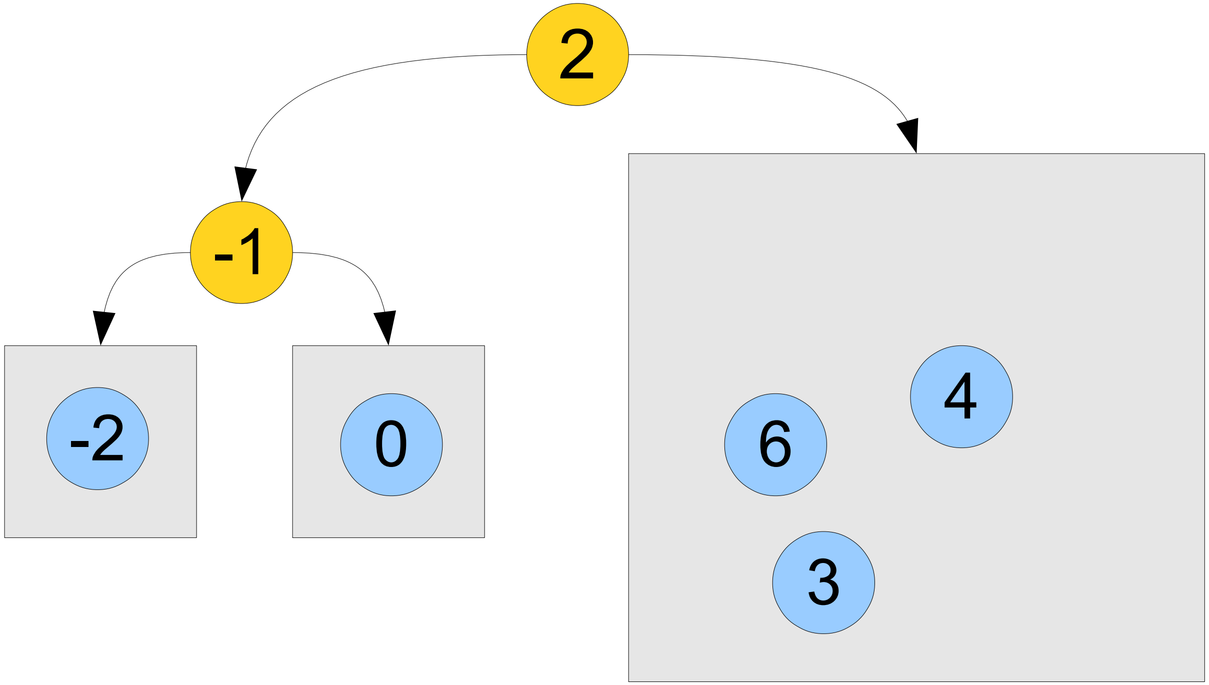

2
-1
0
4
6
-2
3

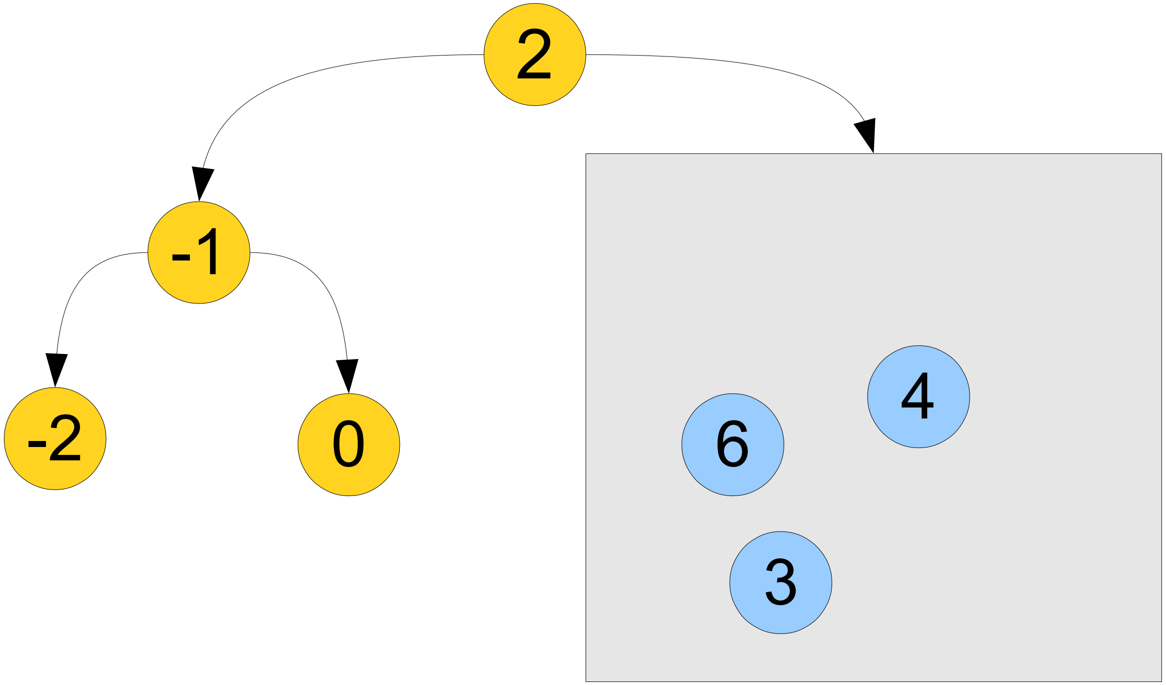

2
-1
0
-2
4
6
3

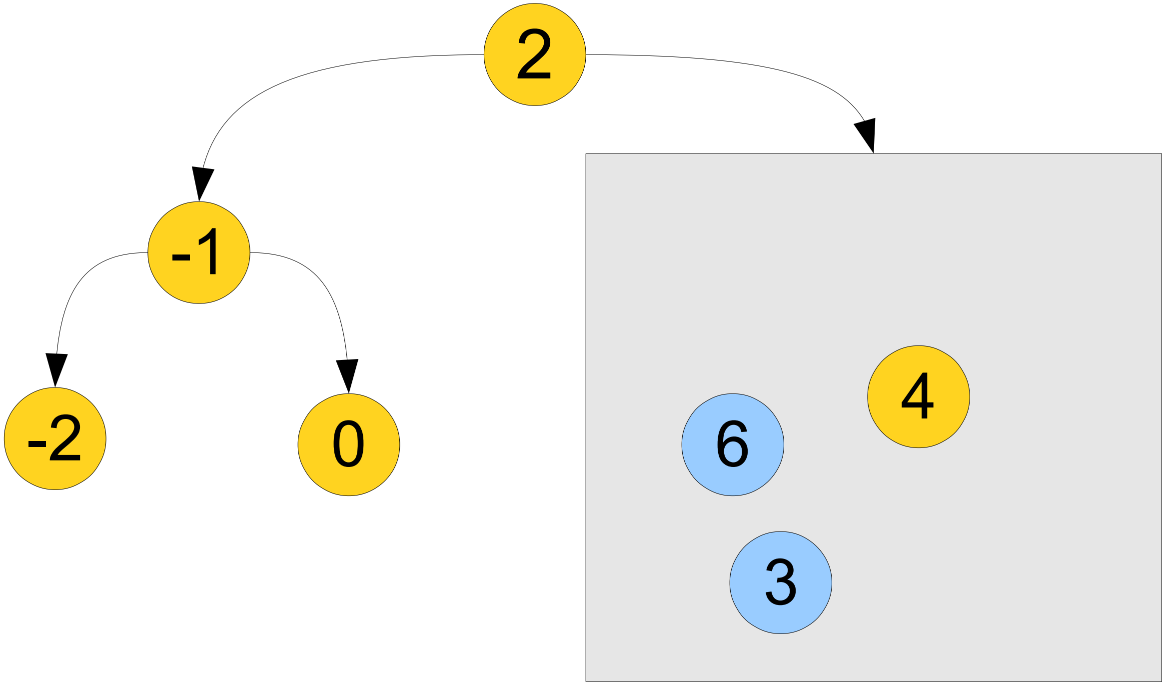

2
-1
0
-2
6
4
3

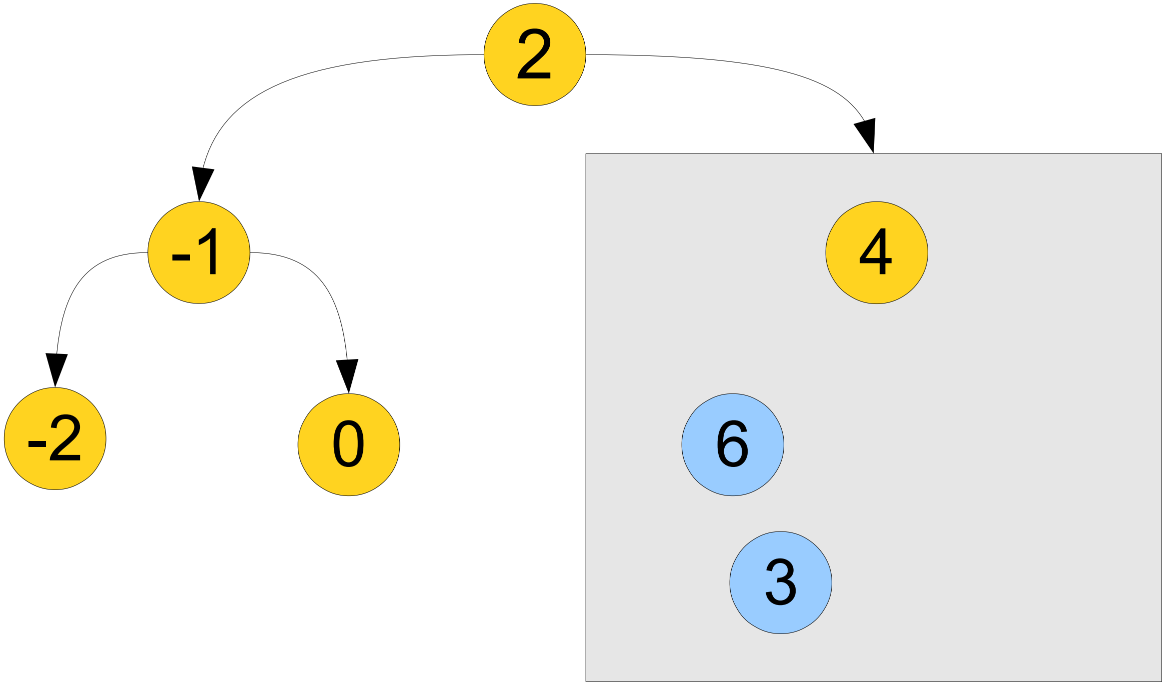

2
-1
0
-2
6
4
3

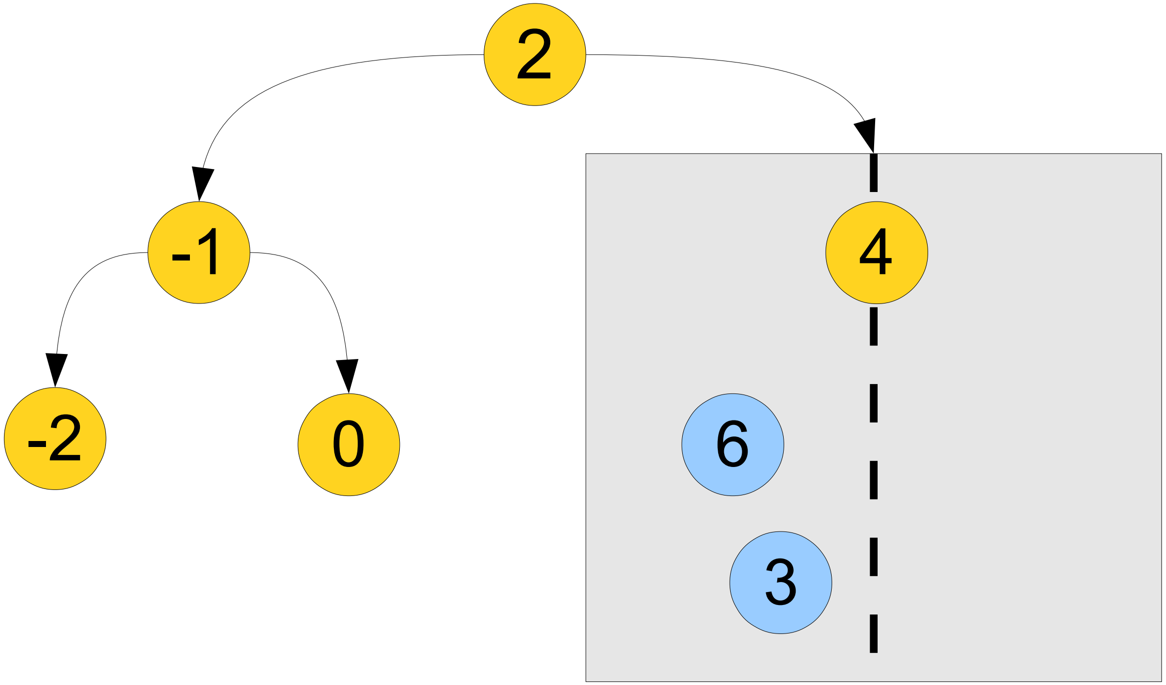

2
-1
-2
0
4
6
3

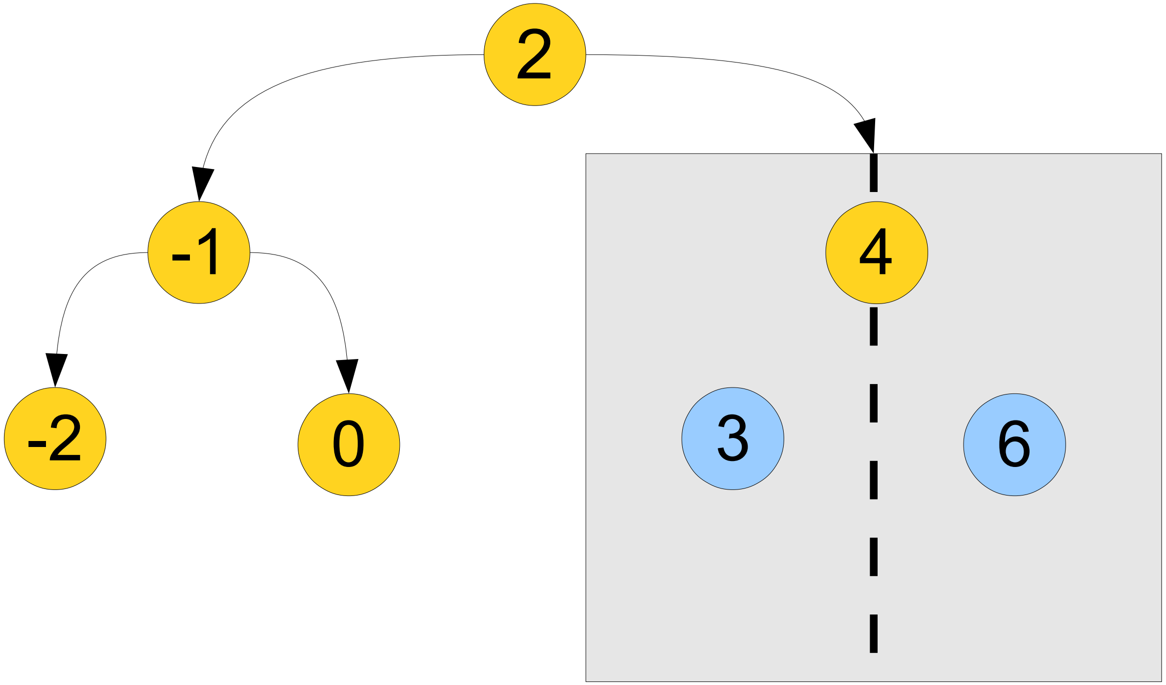

2
-1
-2
0
6
4
3

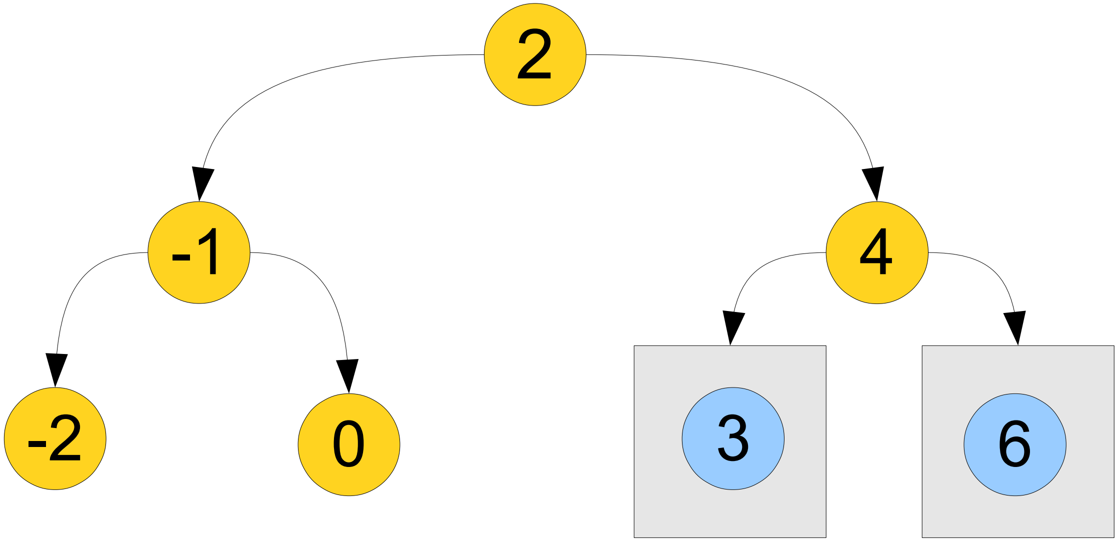

2
-1
-2
0
6
4
3

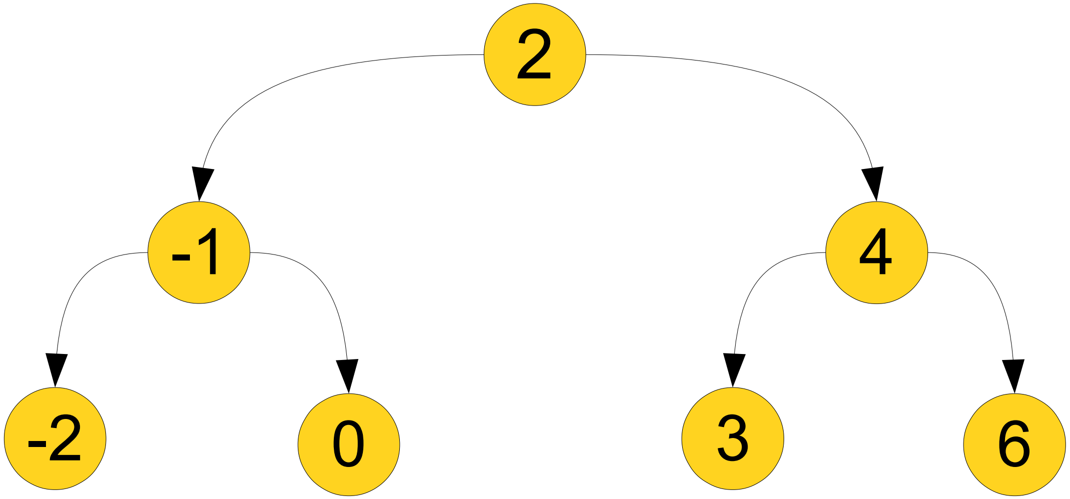

2
-1
-2
0
6
4
3

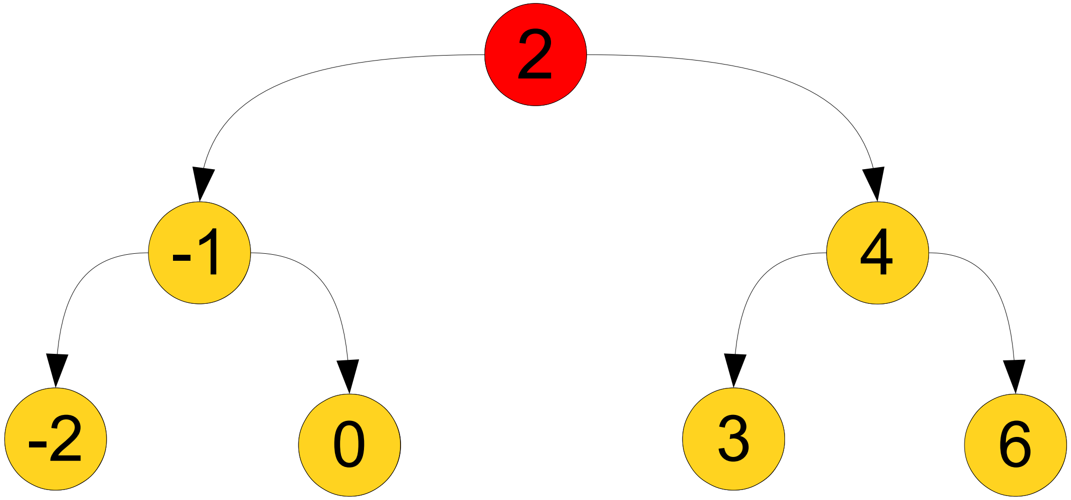

2
-1
-2
0
4
6
3

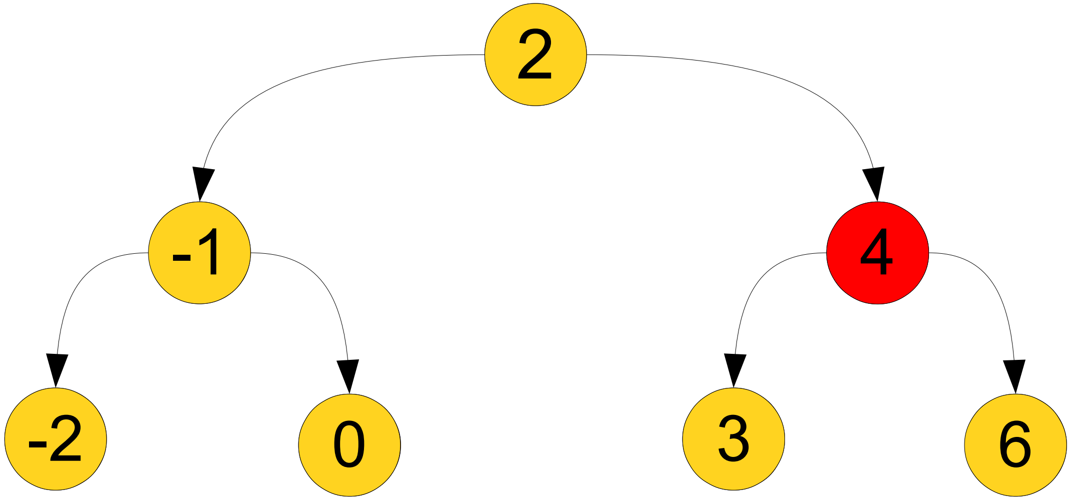

2
-1
-2
0
4
6
3

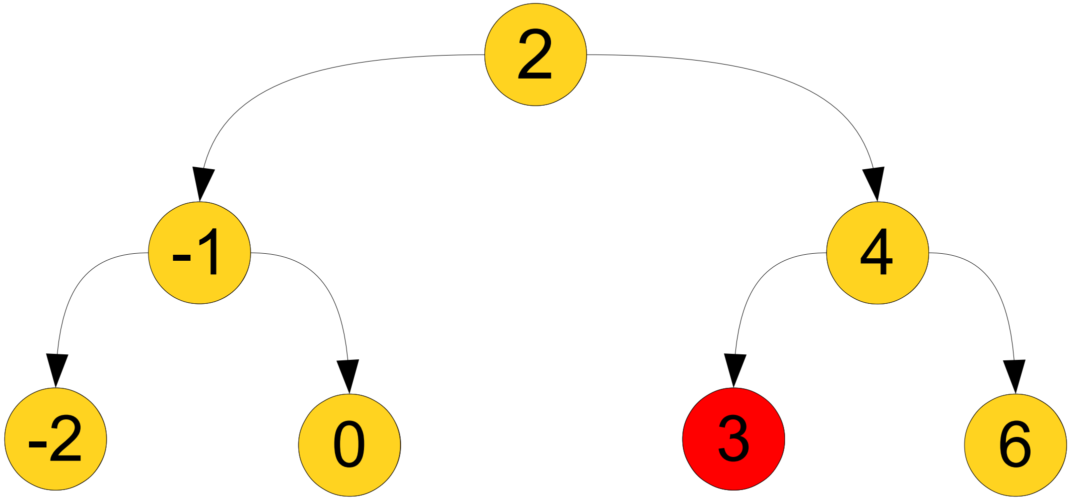

2
-1
4
-2
0
3
6

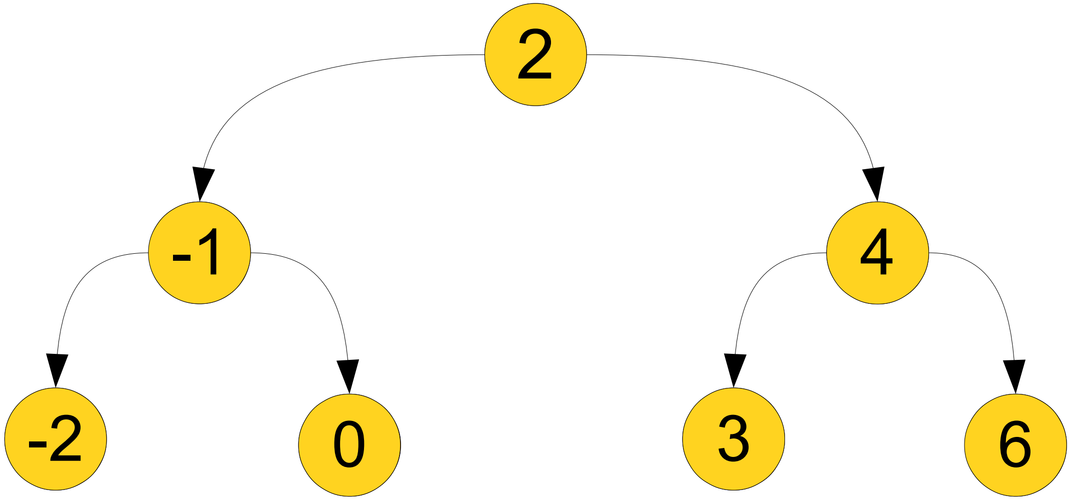

2
-1
4
-2
0
3
6

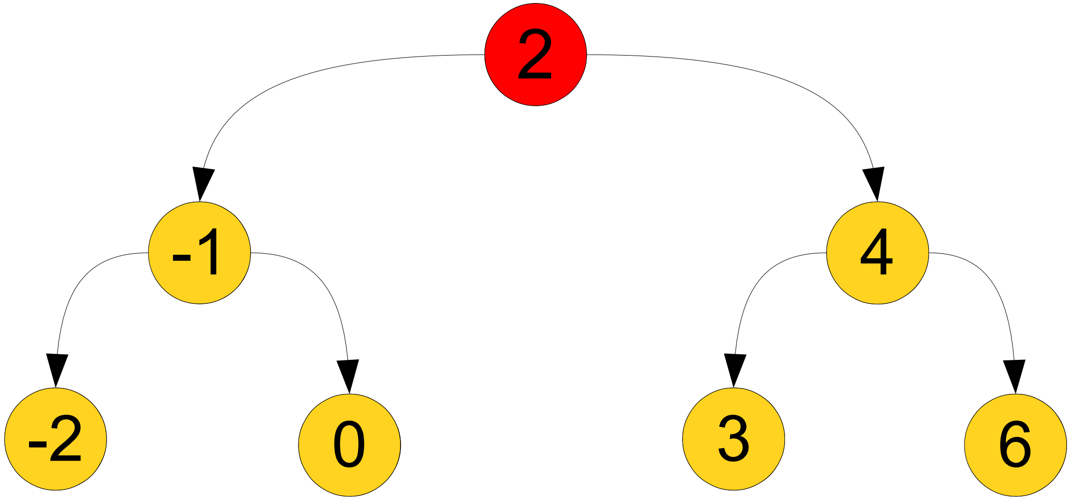

2
-1
4
-2
0
3
6

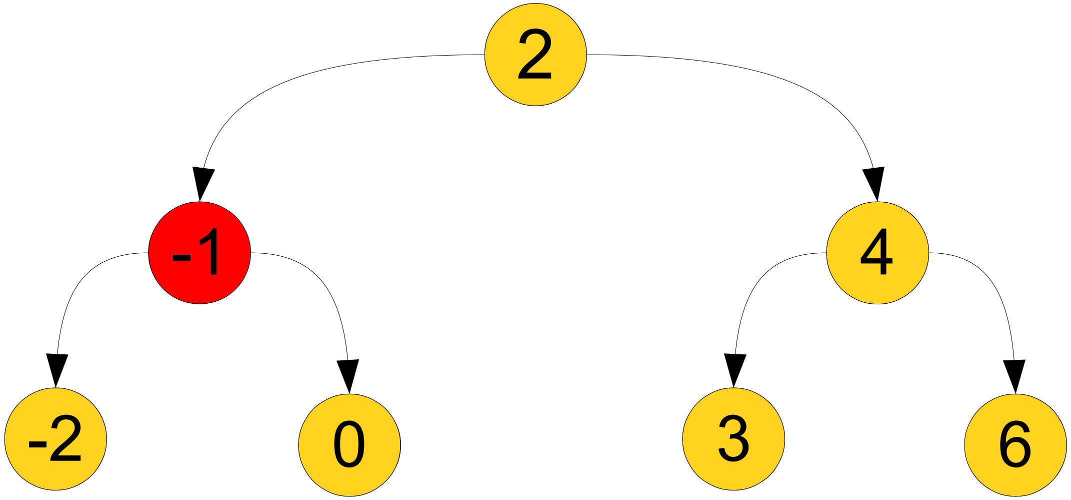

2
-1
4
-2
0
3
6

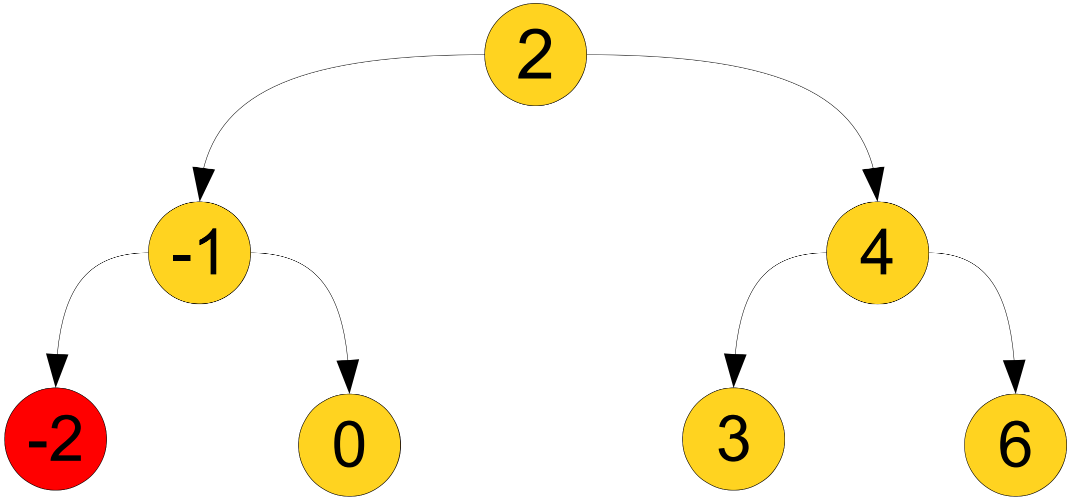

2
-1
4
-2
0
3
6

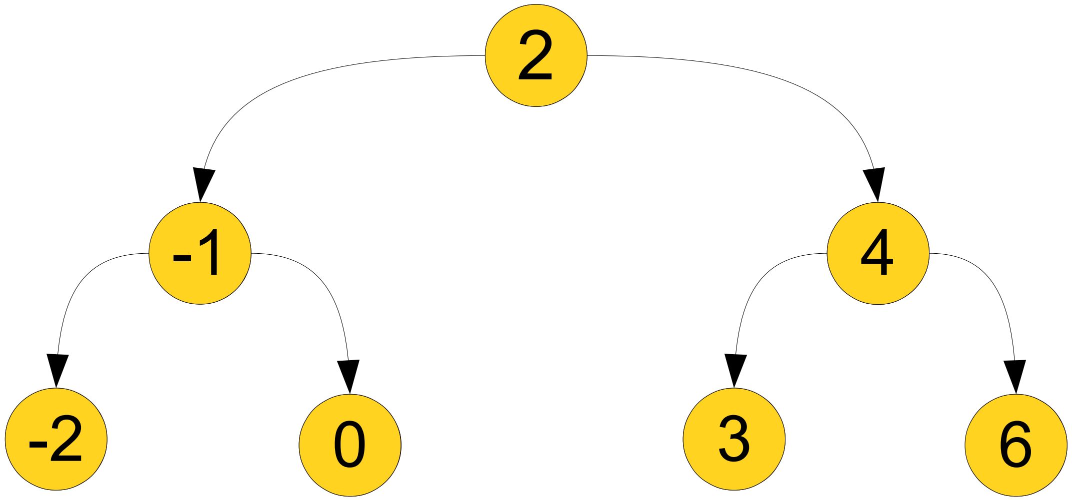

2
-1
4
-2
0
3
6


2
-1
4
-2
0
3
6


2
-1
4
-2
0
3
6


2
-1
4
-2
0
3
6

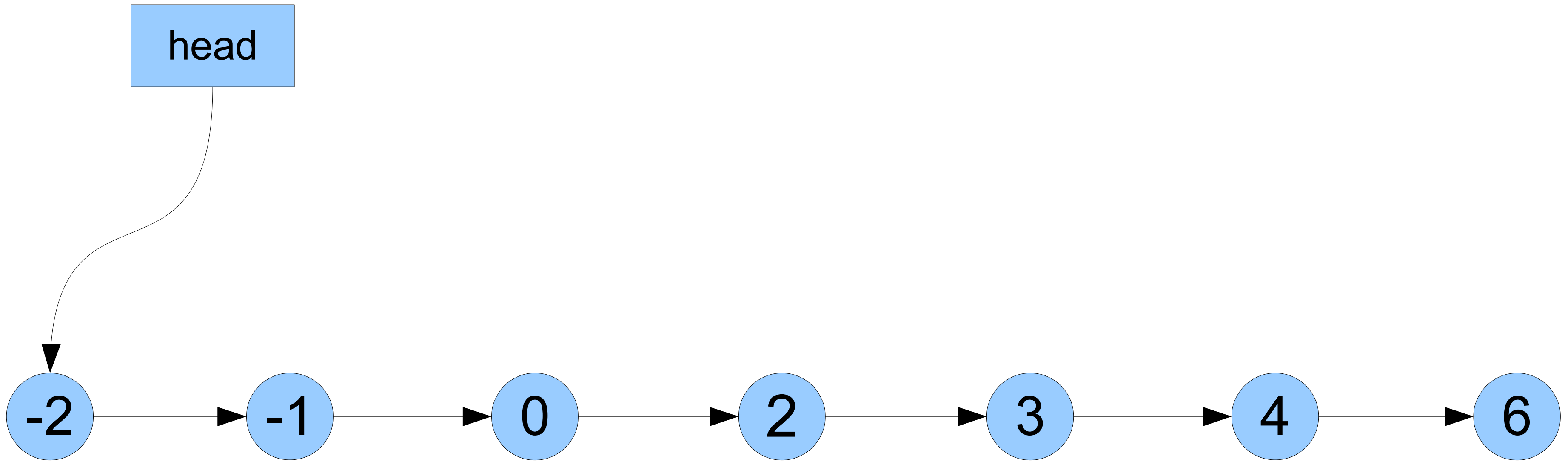

2
-1
4
-2
0
3
6

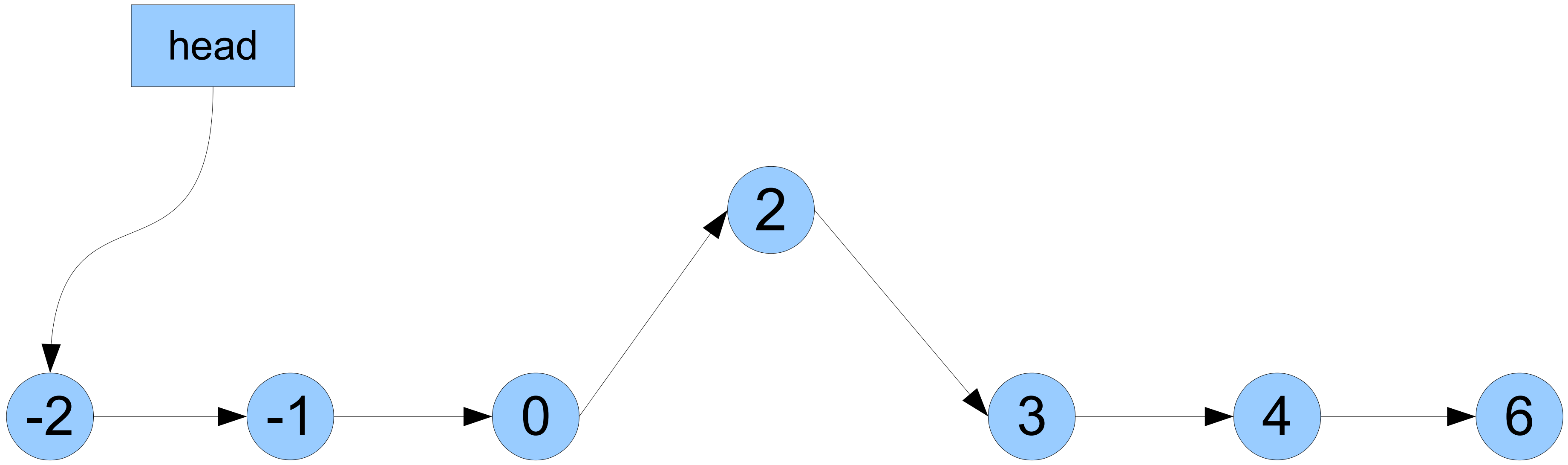

2
-1
4
-2
0
3
6

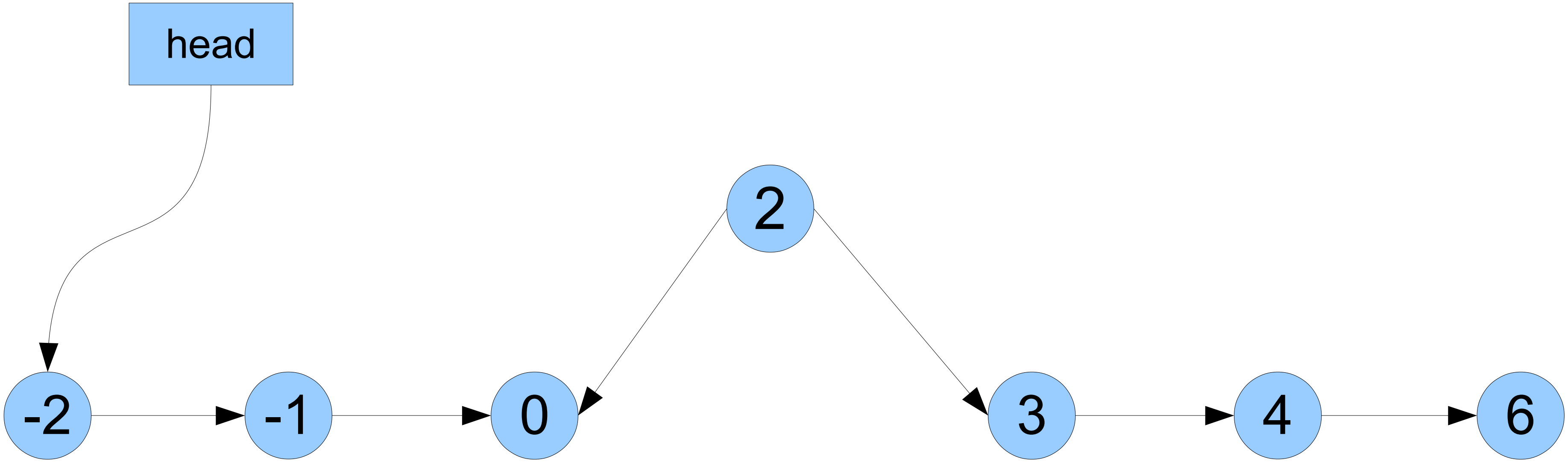
Derivation 2

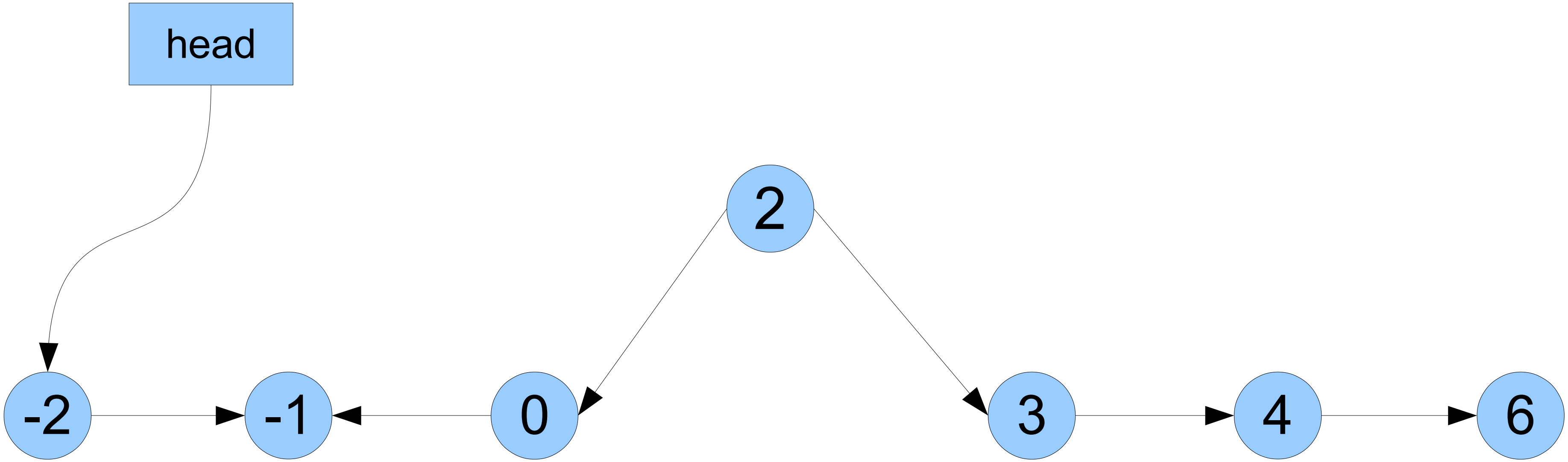
Sorted Vectors and Linked Lists

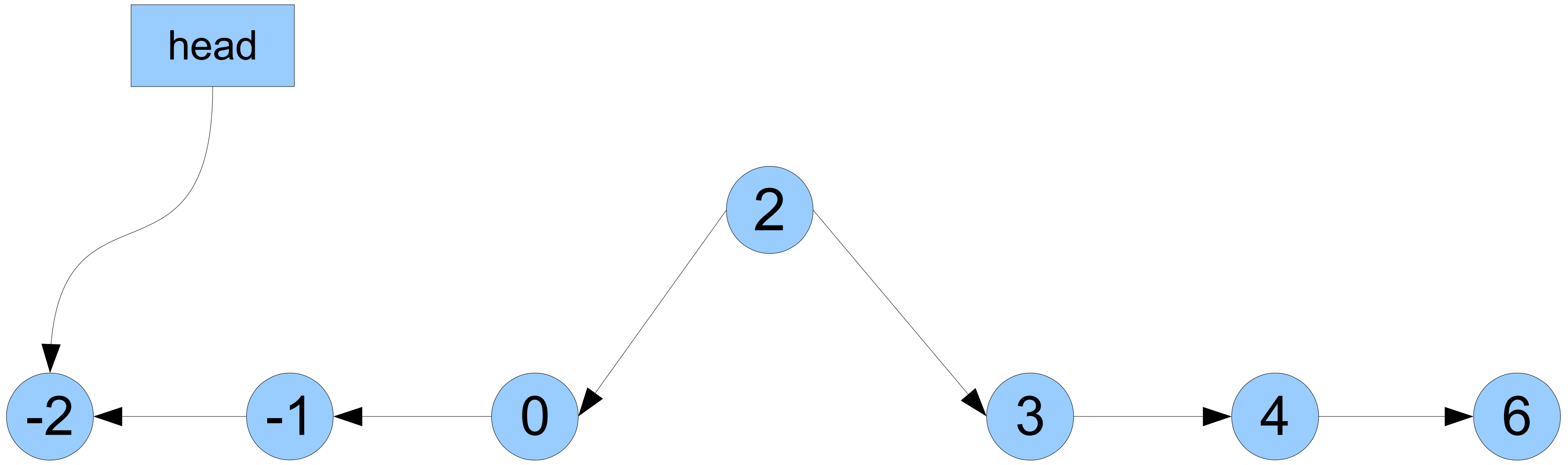
- If we want to store a sorted sequence of elements, we have two choices:
  - Sorted array
    - Pro: Can run binary search
    - Con: Insertion takes **O(*n*)** time
  - Sorted linked list
    - Pro: Insertion takes **O(1)** time ***if you know where you are inserting***
    - Con: Cannot run binary search
- Is there a way we can have the best of both worlds?
  - Can we run binary search on a linked list?

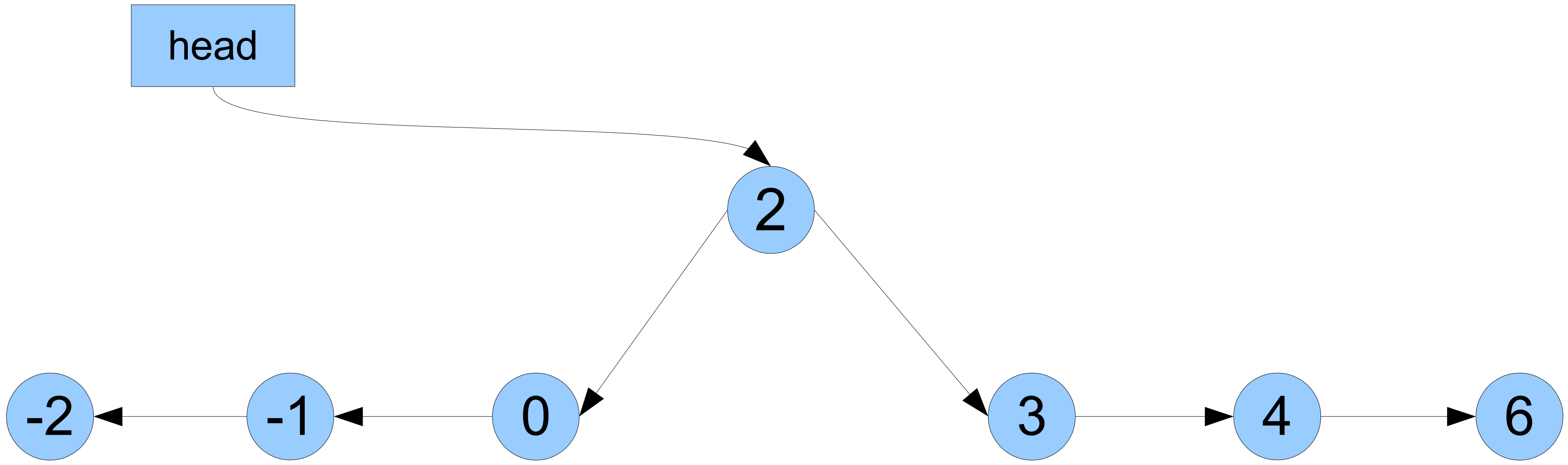

head
-2
-1
0
2
3
4
6

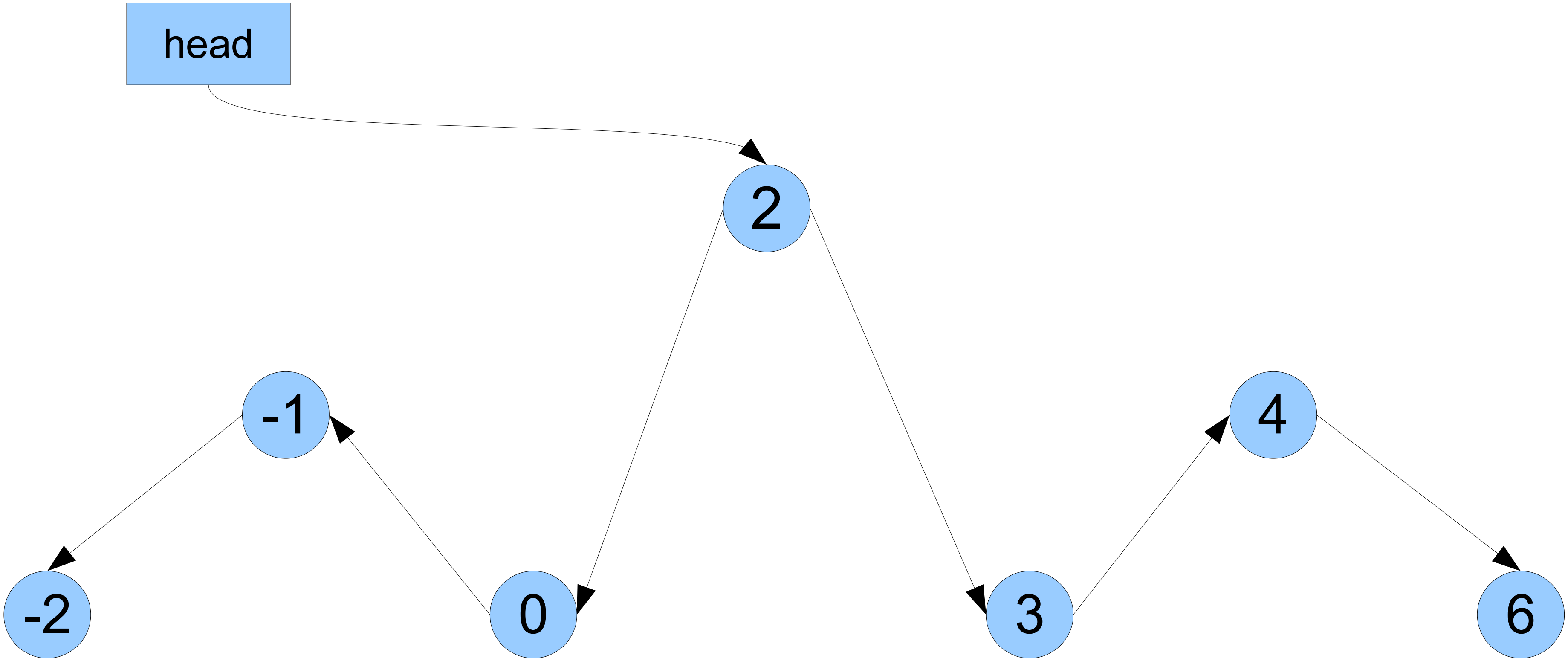

head
-2
-1
0
2
3
4
6

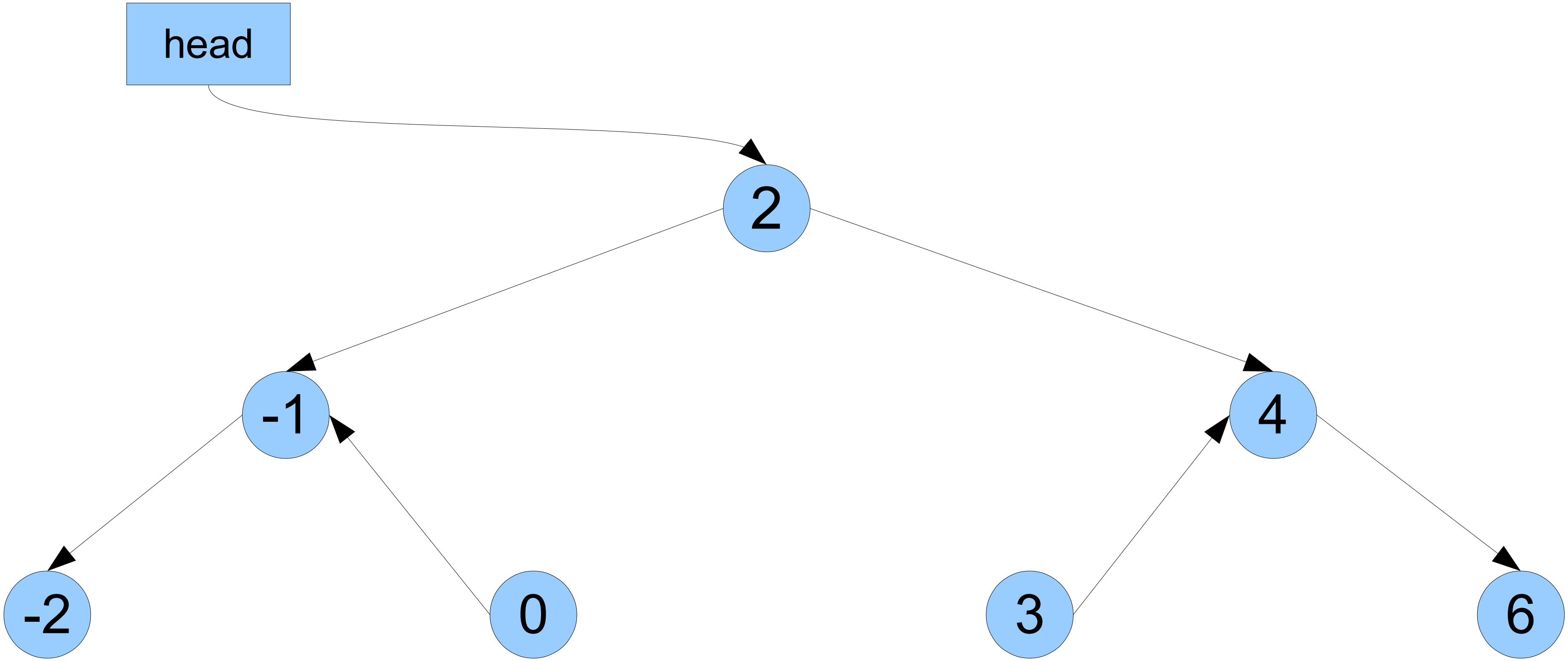

head
-2
-1
0
2
3
4
6


head
2
-2
-1
0
3
4
6

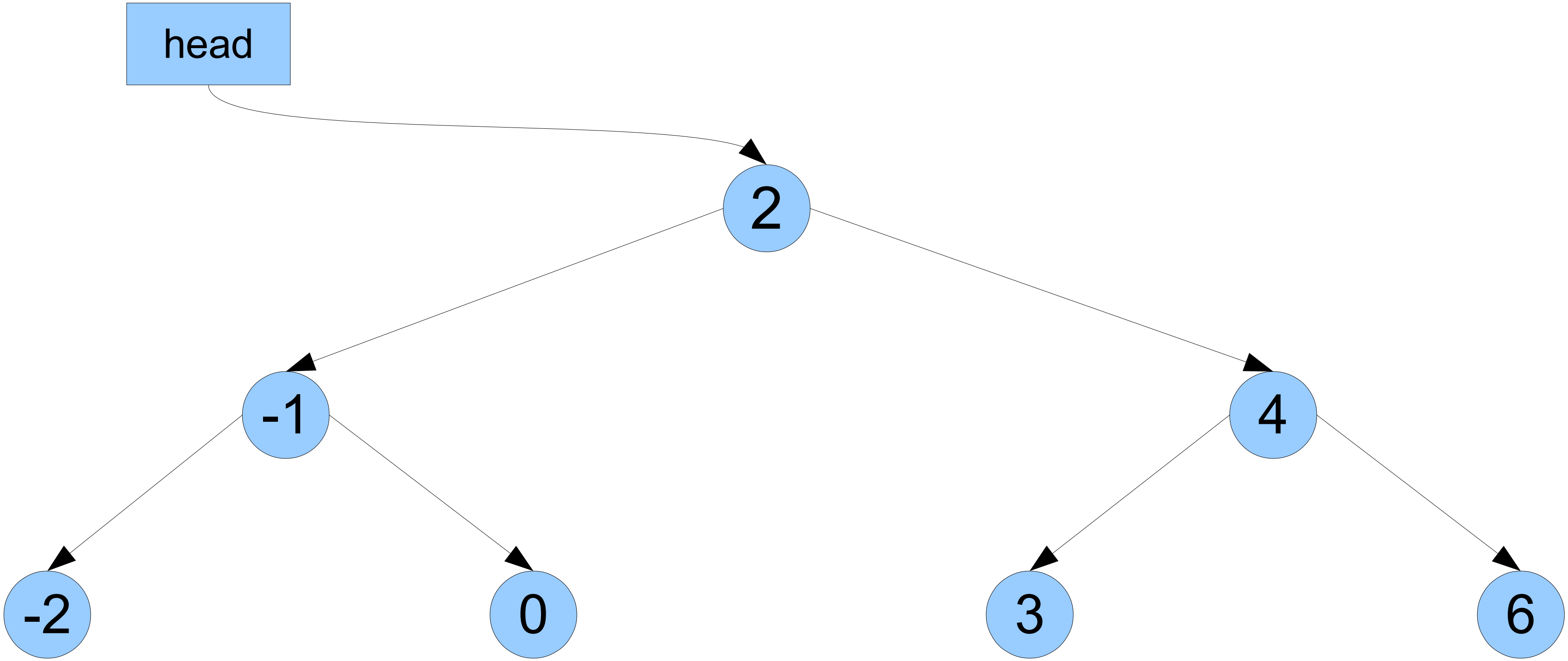

head
2
-2
-1
0
3
4
6


head
2
-2
-1
0
3
4
6


head
2
-1
4
-2
0
3
6



head
2
-1
4
-2
0
3
6

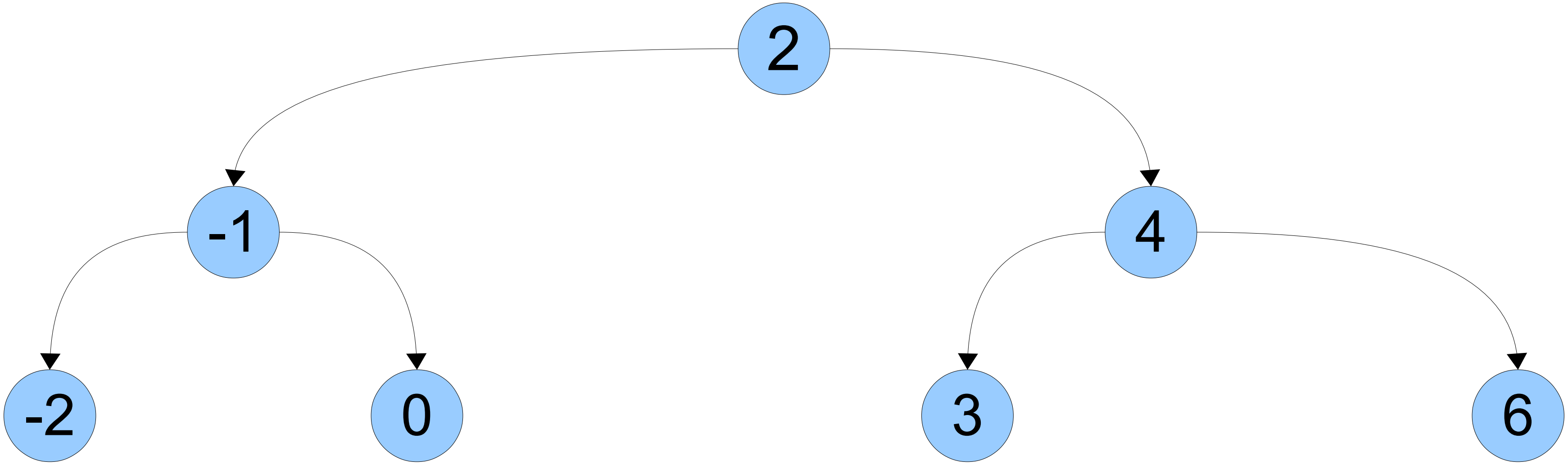

head
2
-1
4
-2
0
3
6

Binary Search Trees

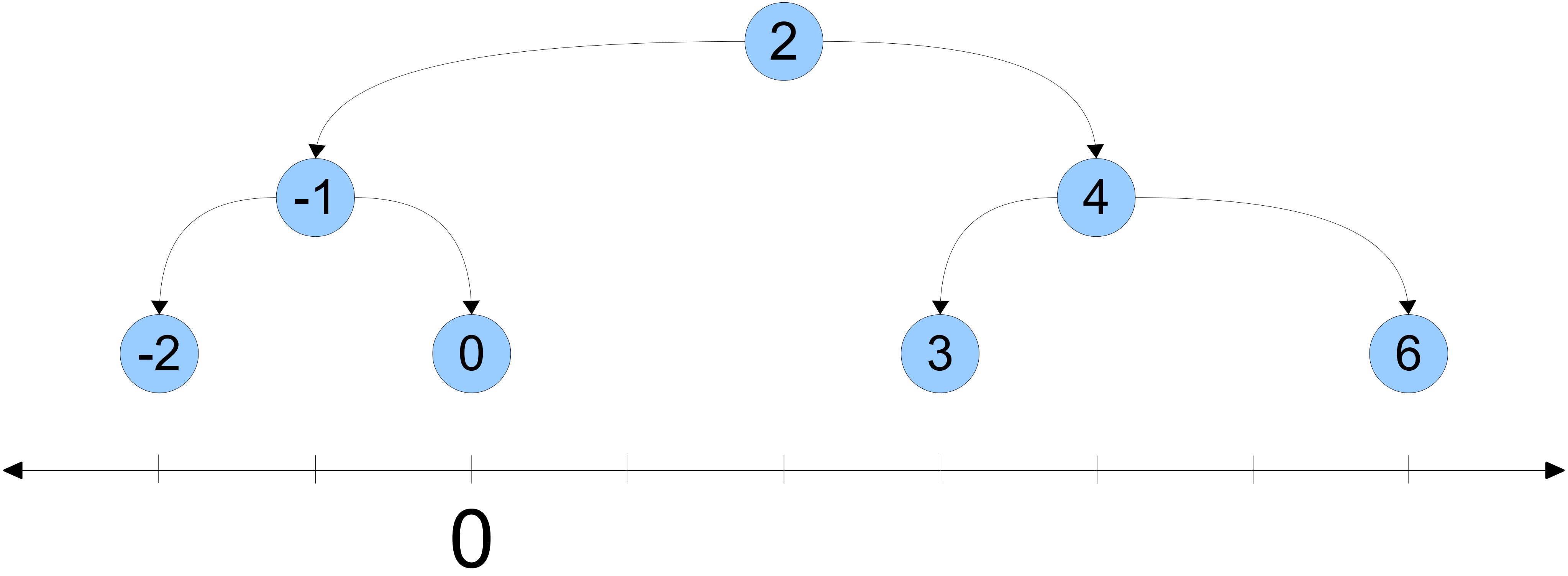
- The data structure we have just seen is called a **binary search tree** (or **BST**).
- Uses comparisons between elements to store elements efficiently.
- What our **`Set`** and **`Map`** use
  - This is why a **`Set`** can only store elements for which the < operator is defined!

The Intuition

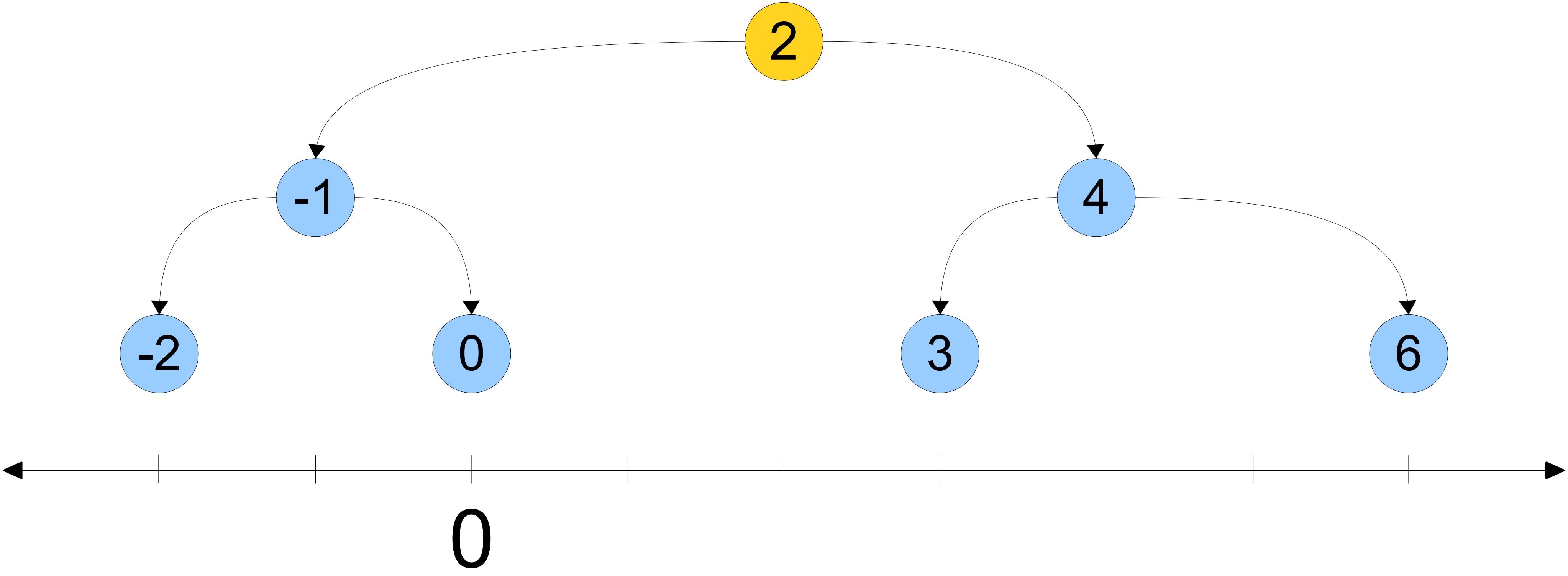
The Intuition



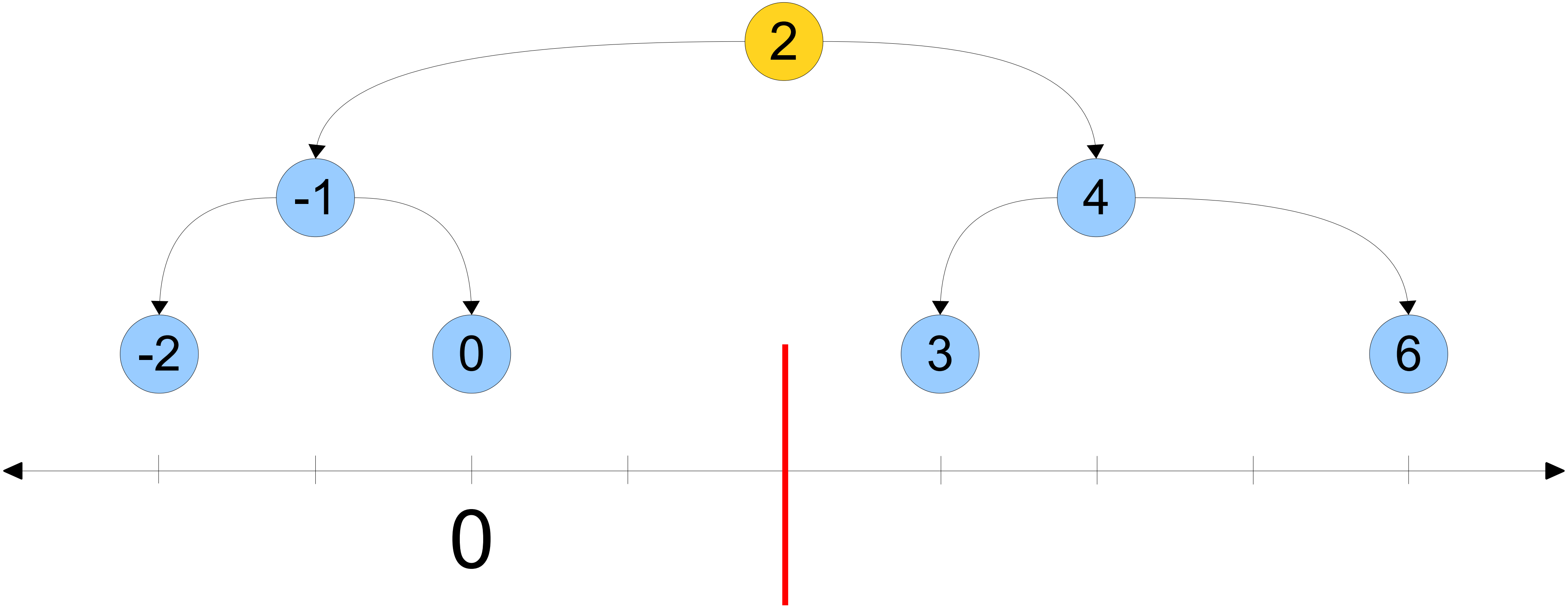
The Intuition



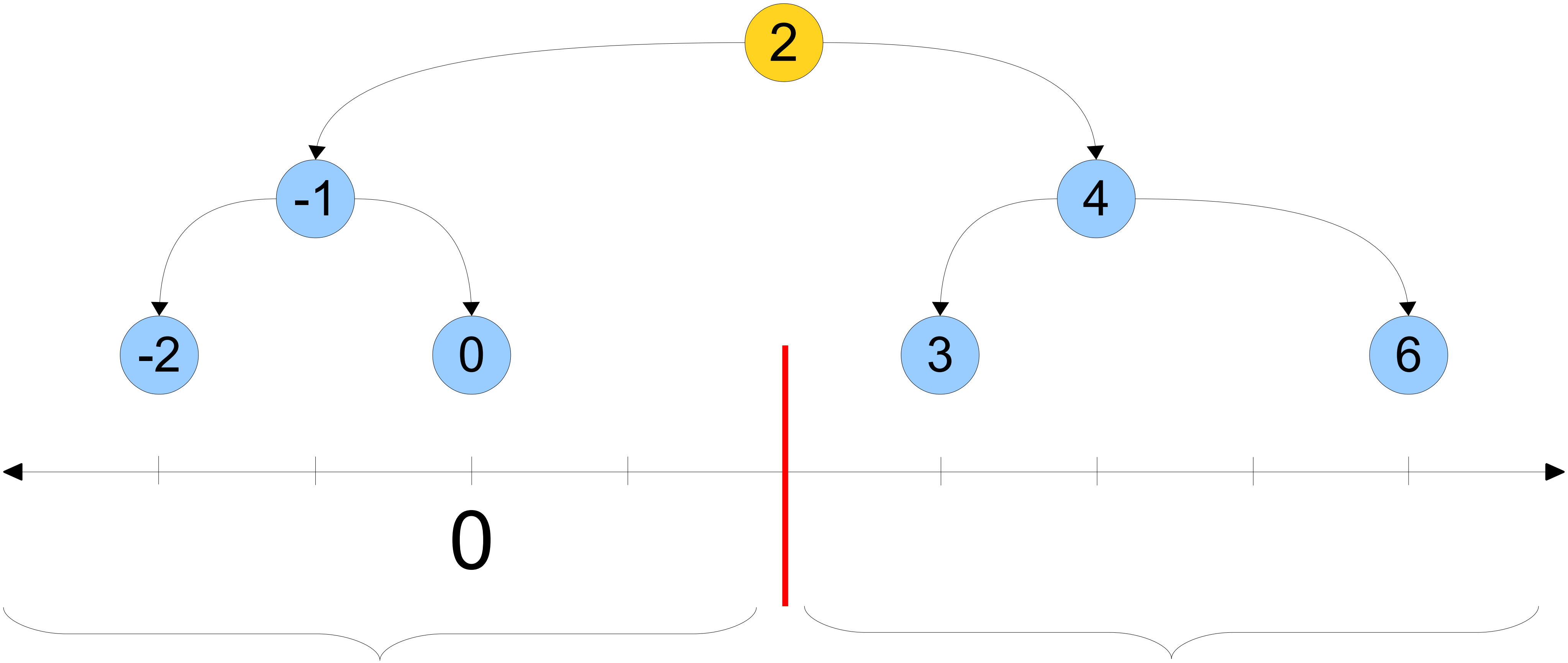
The Intuition



The Intuition



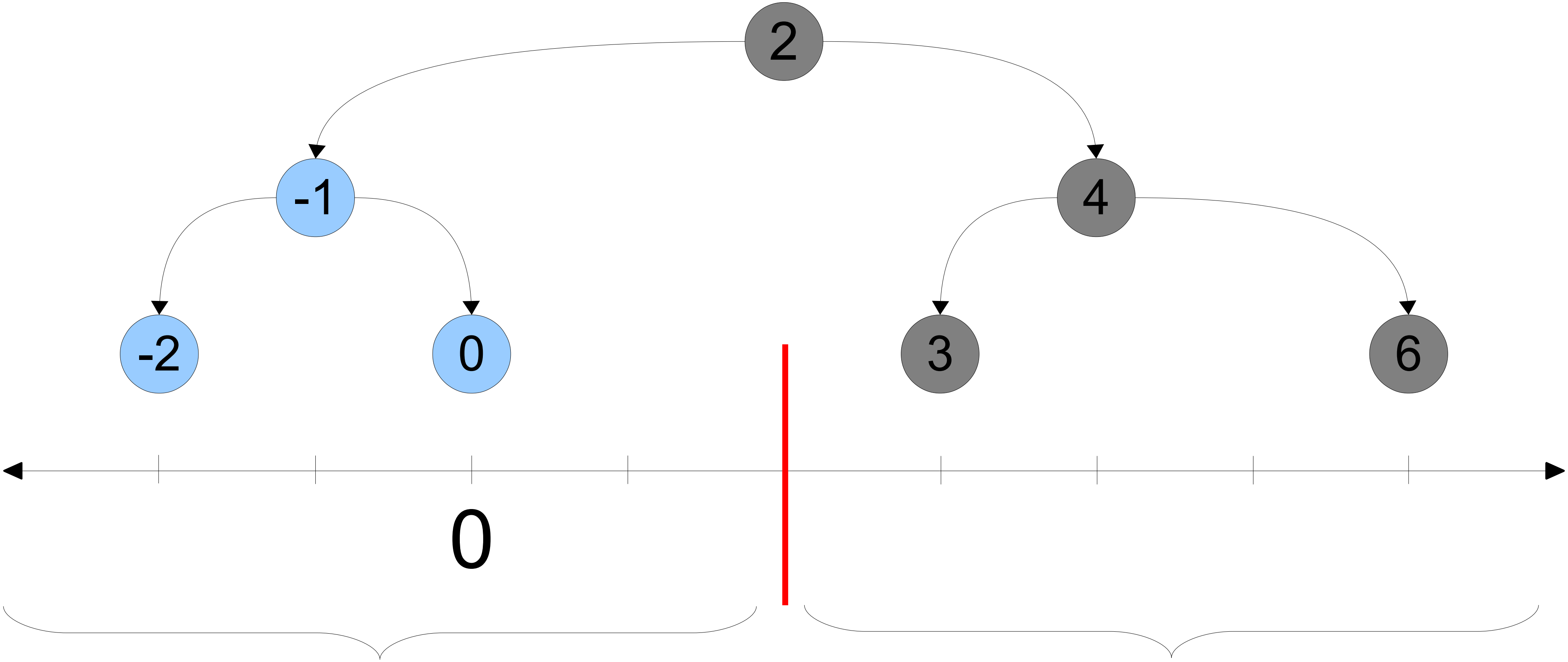
The Intuition



Values less than two

Values greater than two

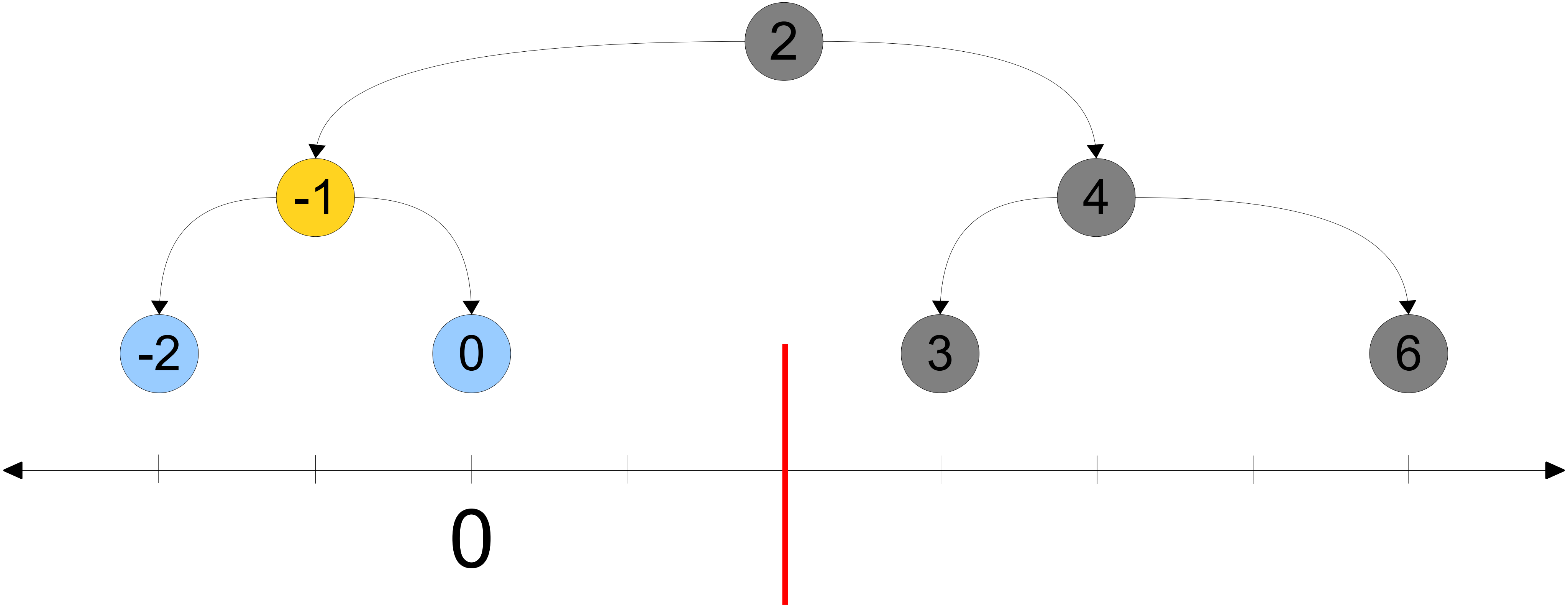
The Intuition



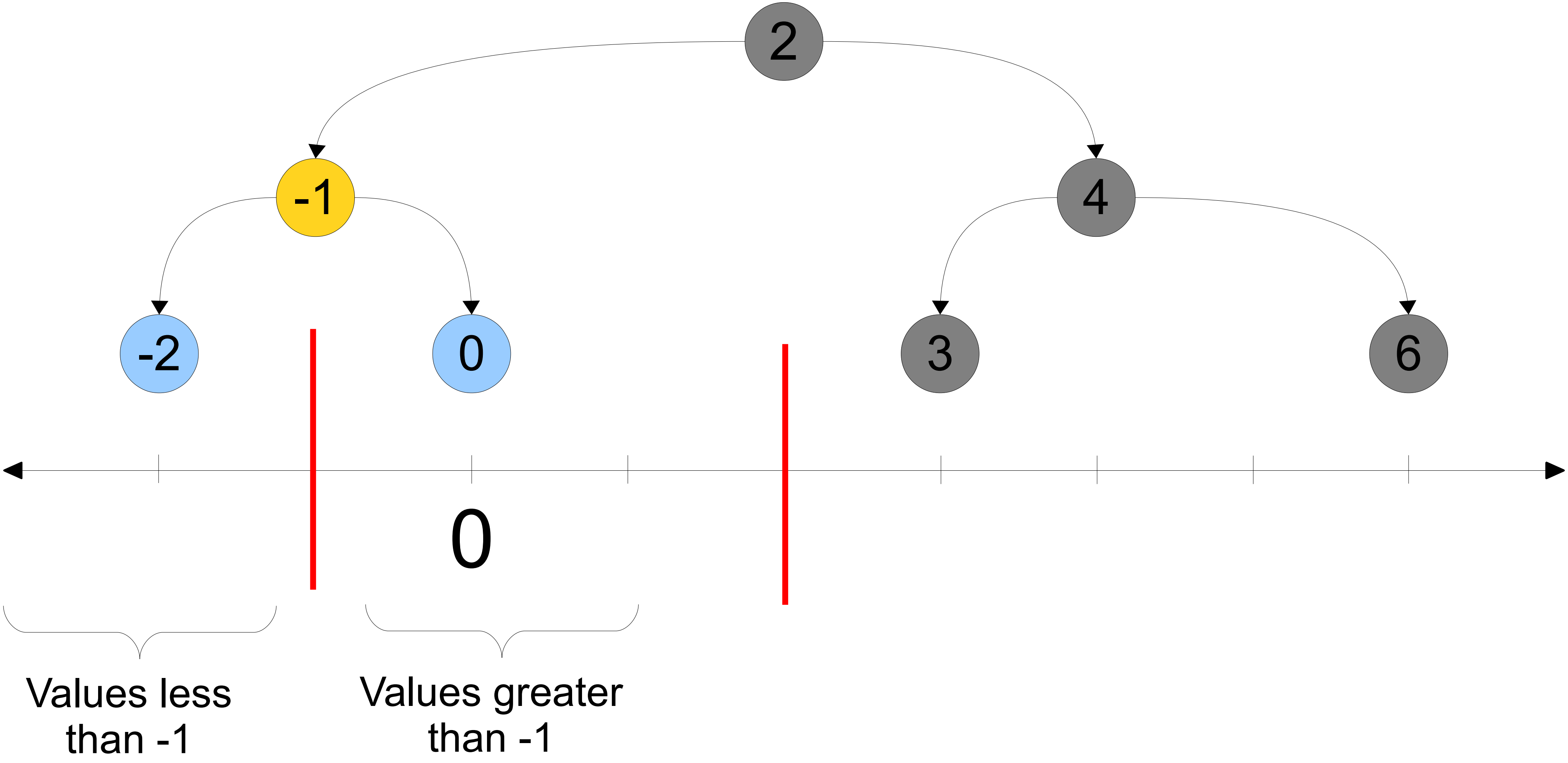
Values less than two

Values greater than two

The Intuition



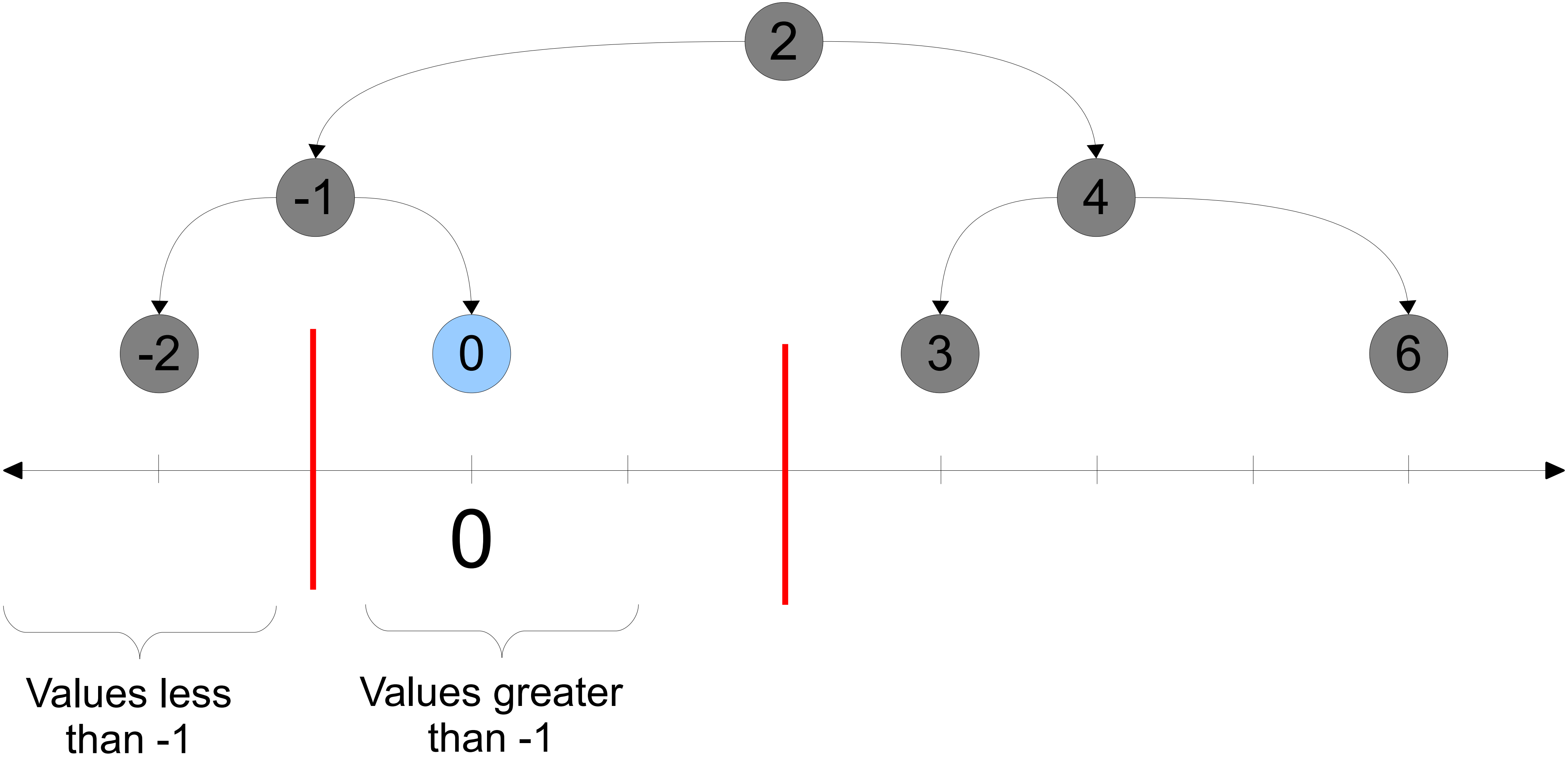
The Intuition



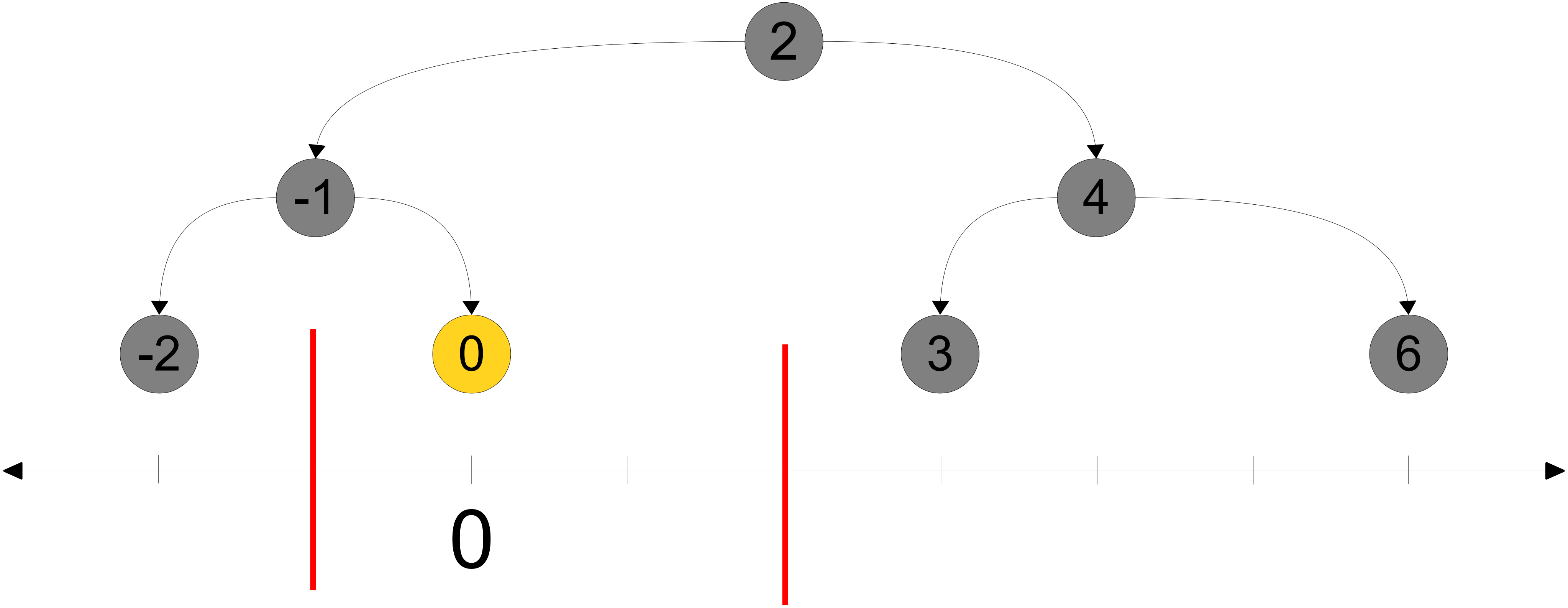
Values less than -1

Values greater than -1

The Intuition

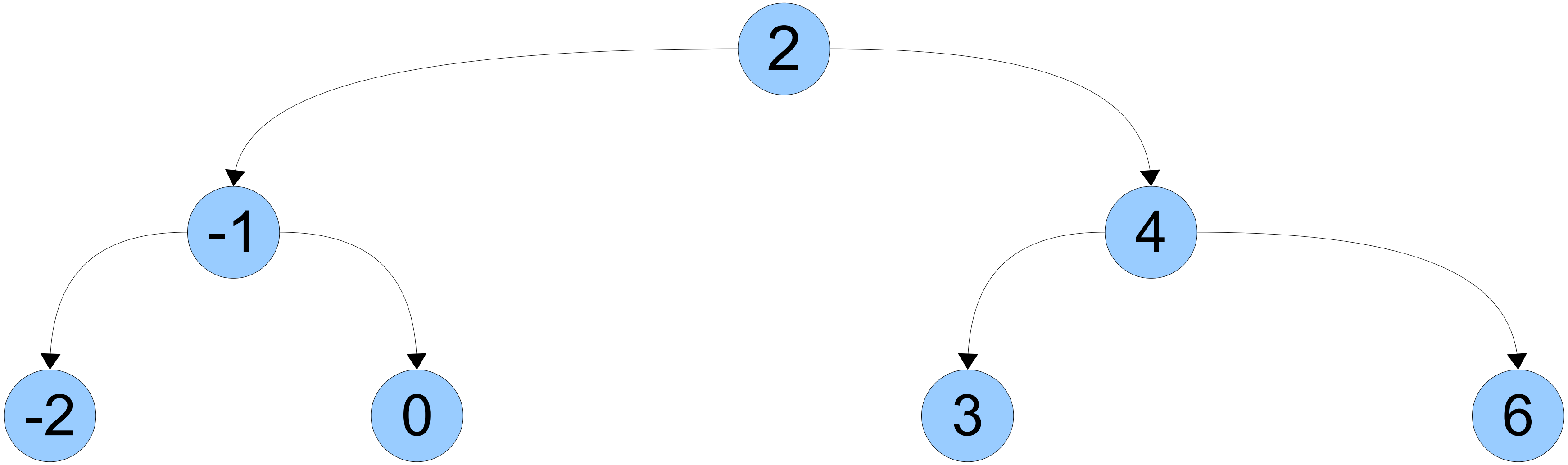


The Intuition

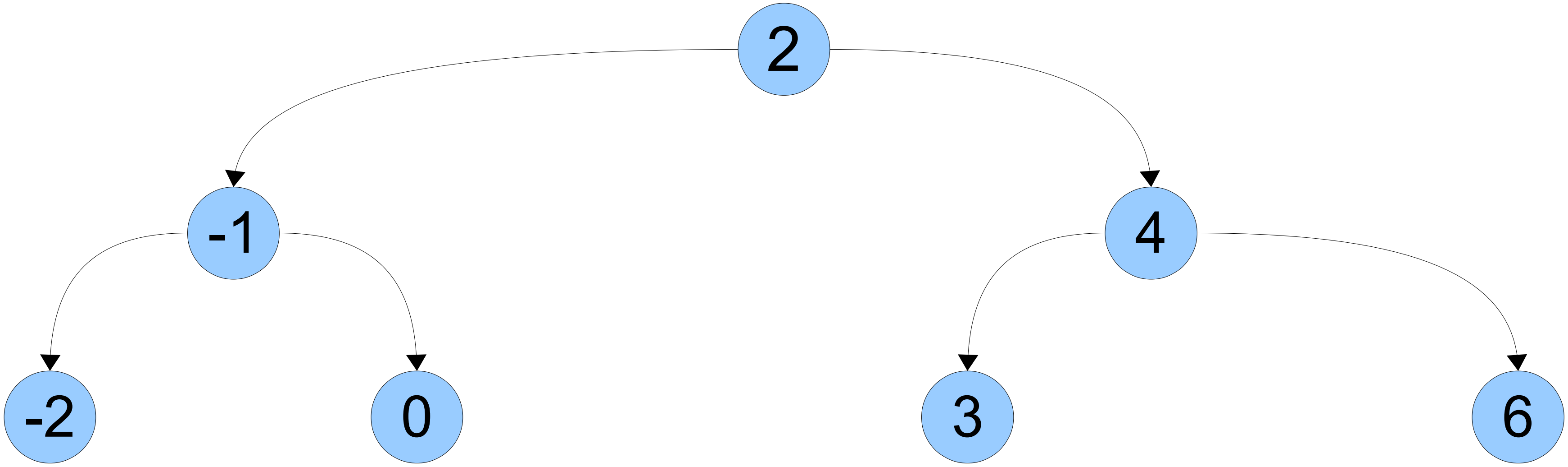


Tree Terminology

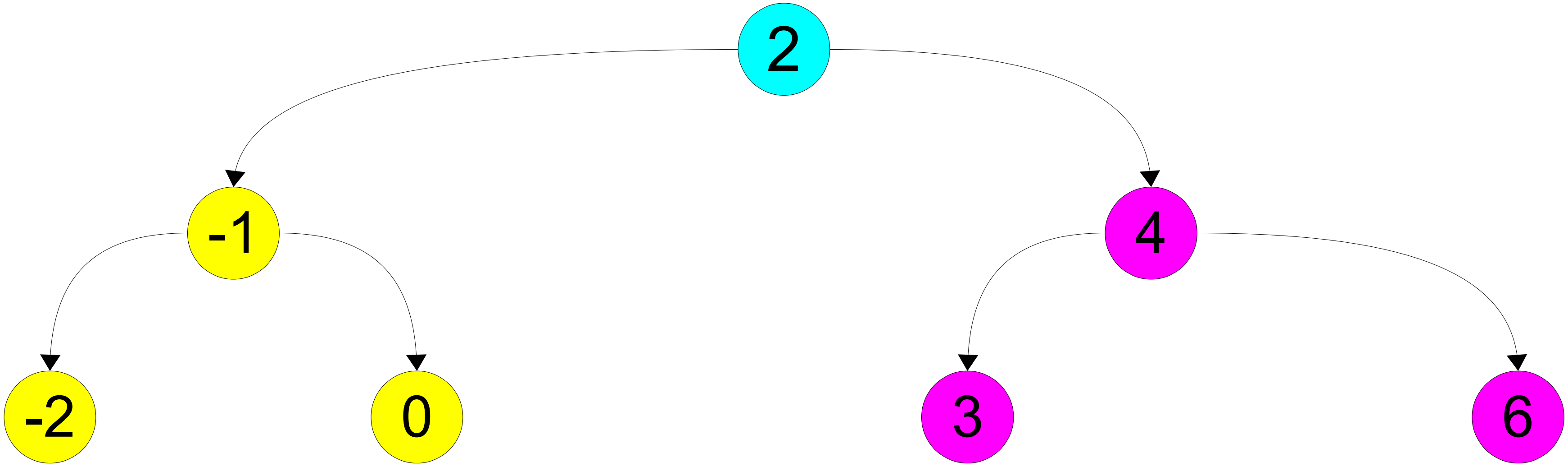
- A BST is a collection of **nodes**.
- The top node is called the **root node**.
- Nodes with no children are called **leaves**.



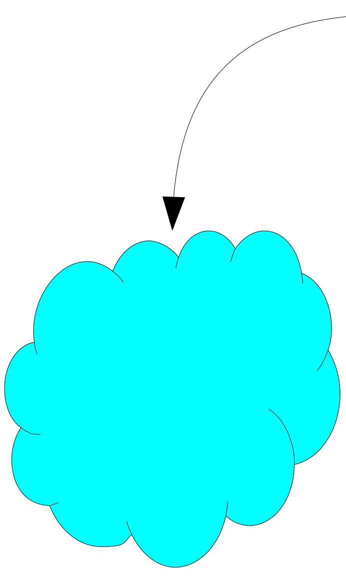
A Recursive View of BSTs



A Recursive View of BSTs



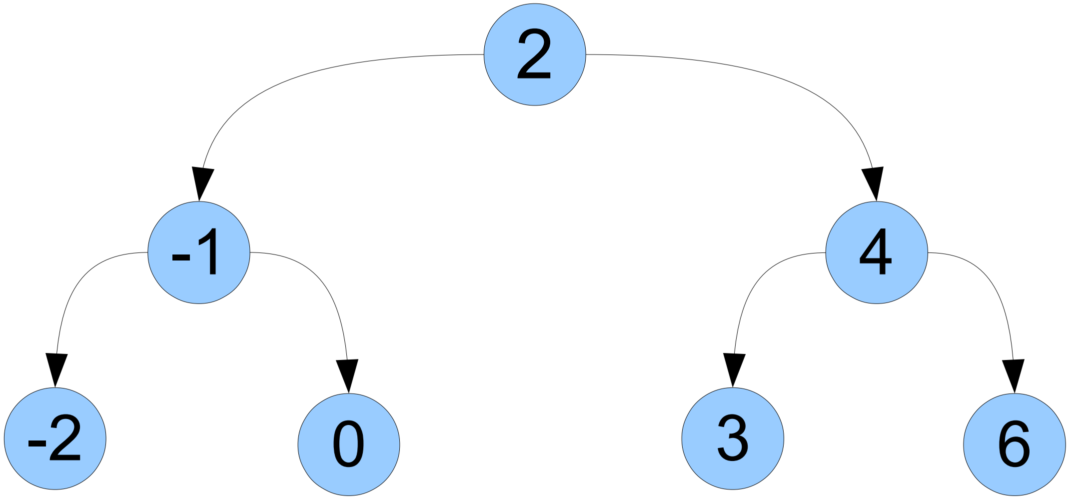
A Recursive View of BSTs



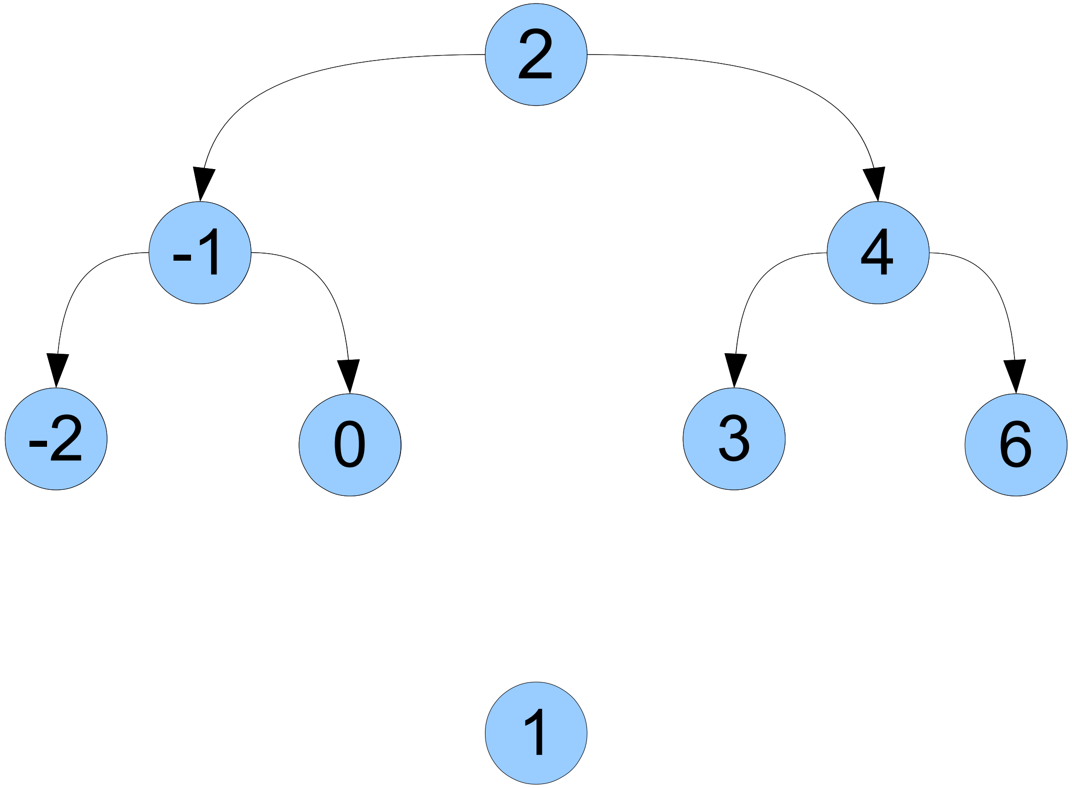
Lookup
(Pseudocode)

Lookup
(`bst.cpp`)

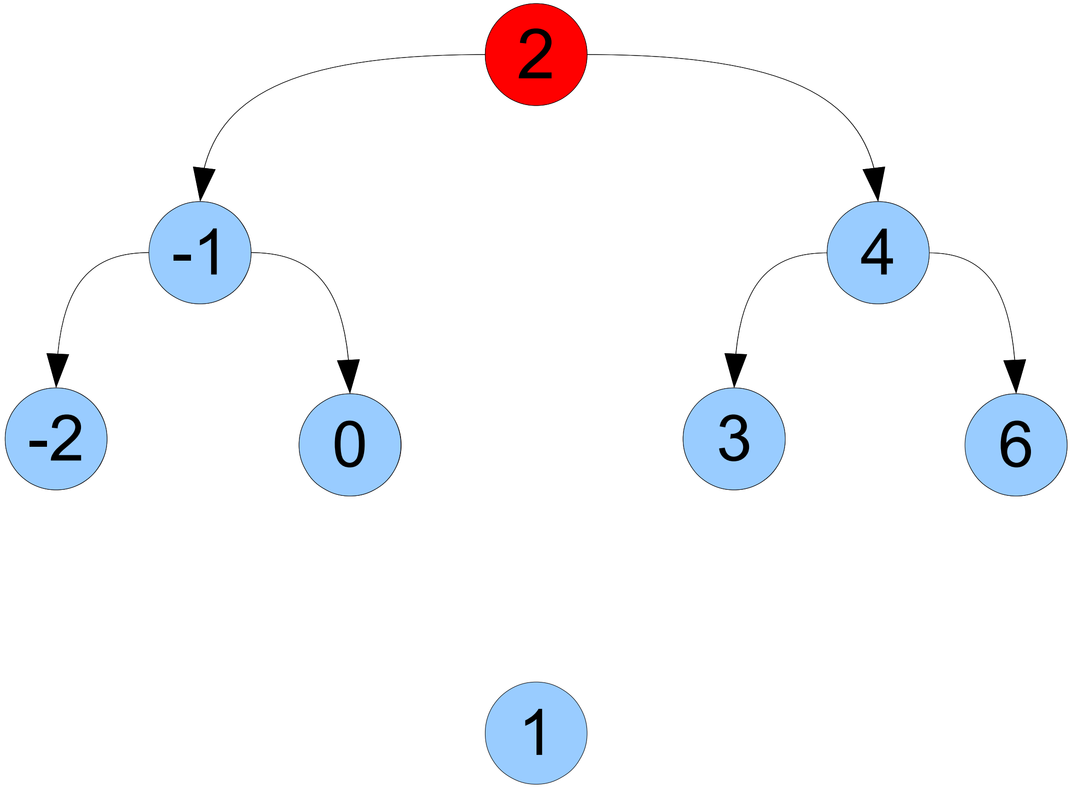
Inserting into a BST



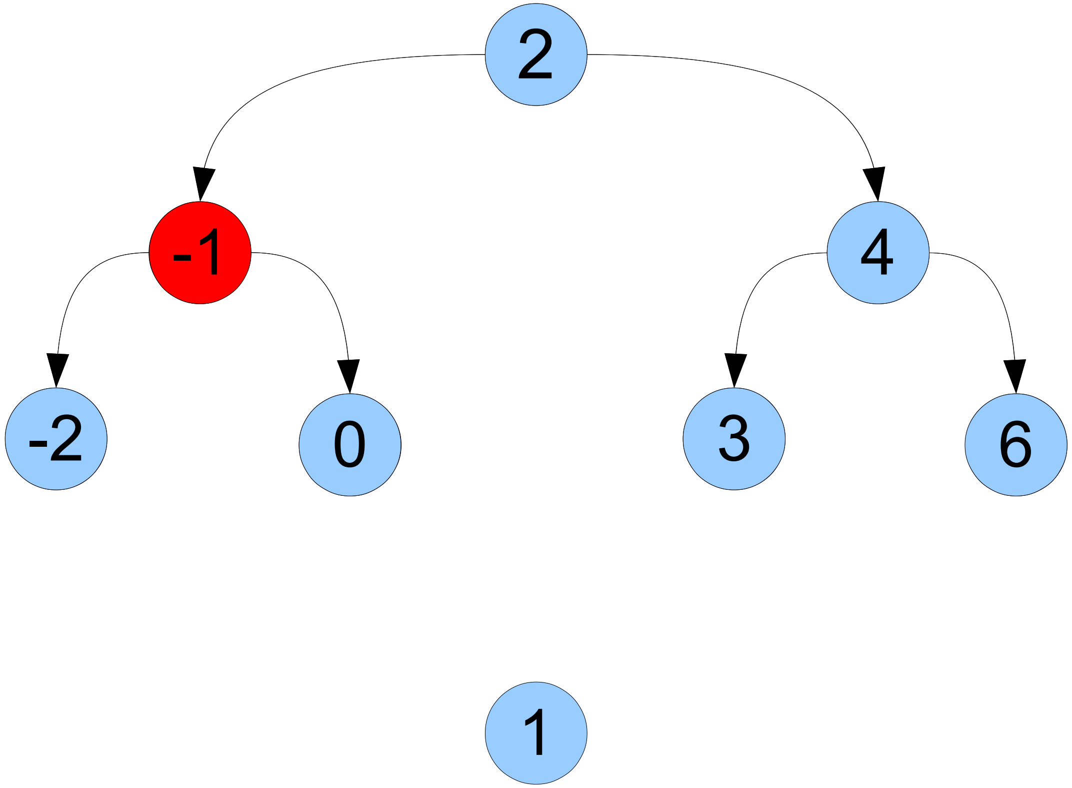
Inserting into a BST



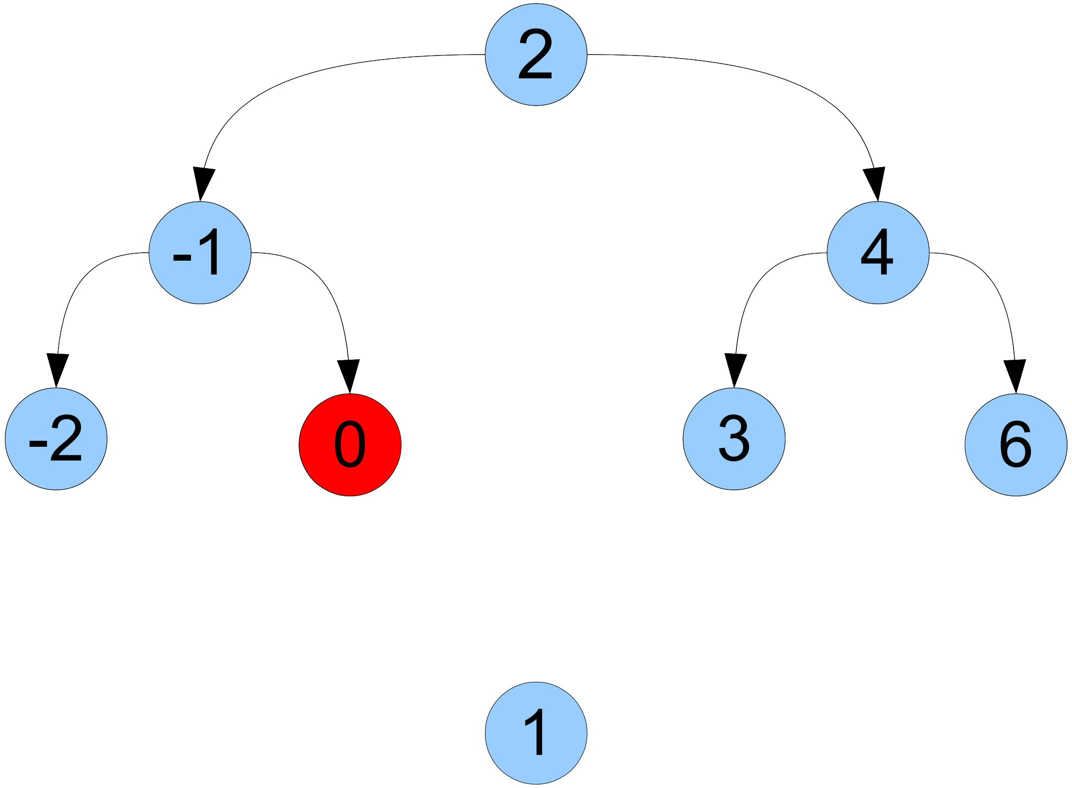
Inserting into a BST



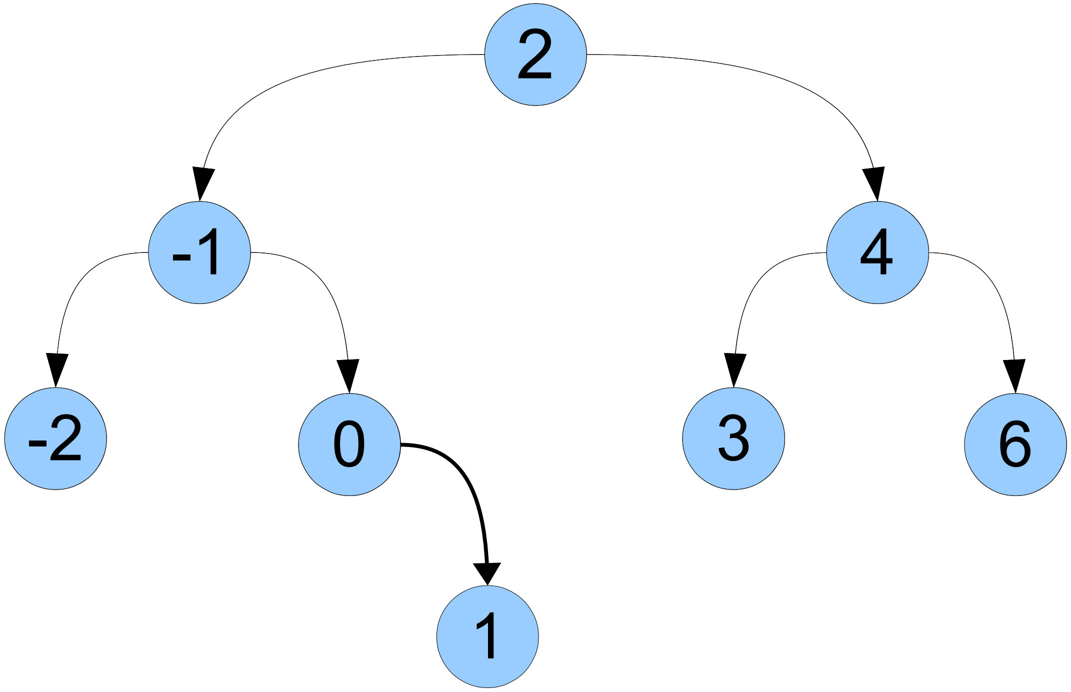
Inserting into a BST



Inserting into a BST



Inserting into a BST

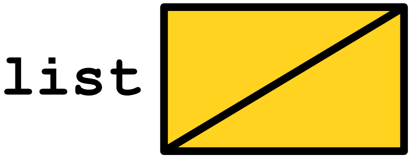


Insertion
(Pseudocode)

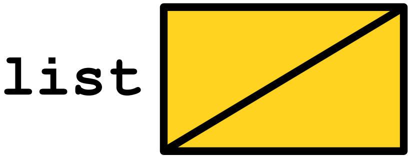
```c
int main() {
    Cell* list = NULL;
    listInsert(list, 137);
    listInsert(list, 42);
    listInsert(list, 271);
}
```

```
int main() {
    Cell* list = NULL;
    listInsert(list, 137);
    listInsert(list, 42);
    listInsert(list, 271);
}
```

```
int main() {
    Cell* list = NULL;
    listInsert(list, 137);
    listInsert(list, 42);
    listInsert(list, 271);
}
```



```
int main() {
    Cell* list = NULL;
    listInsert(list, 137);
    listInsert(list, 42);
    listInsert(list, 271);
}
```



```
int main() {

}
```

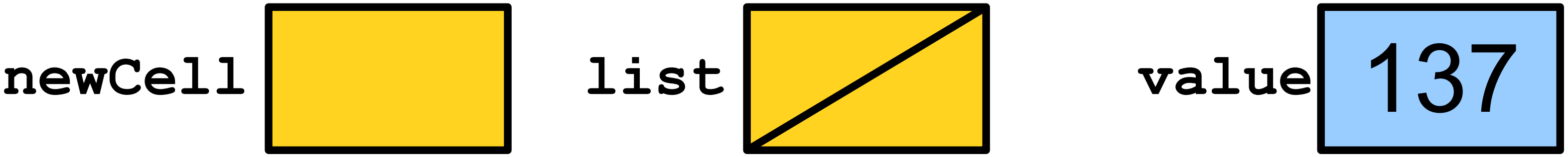
```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

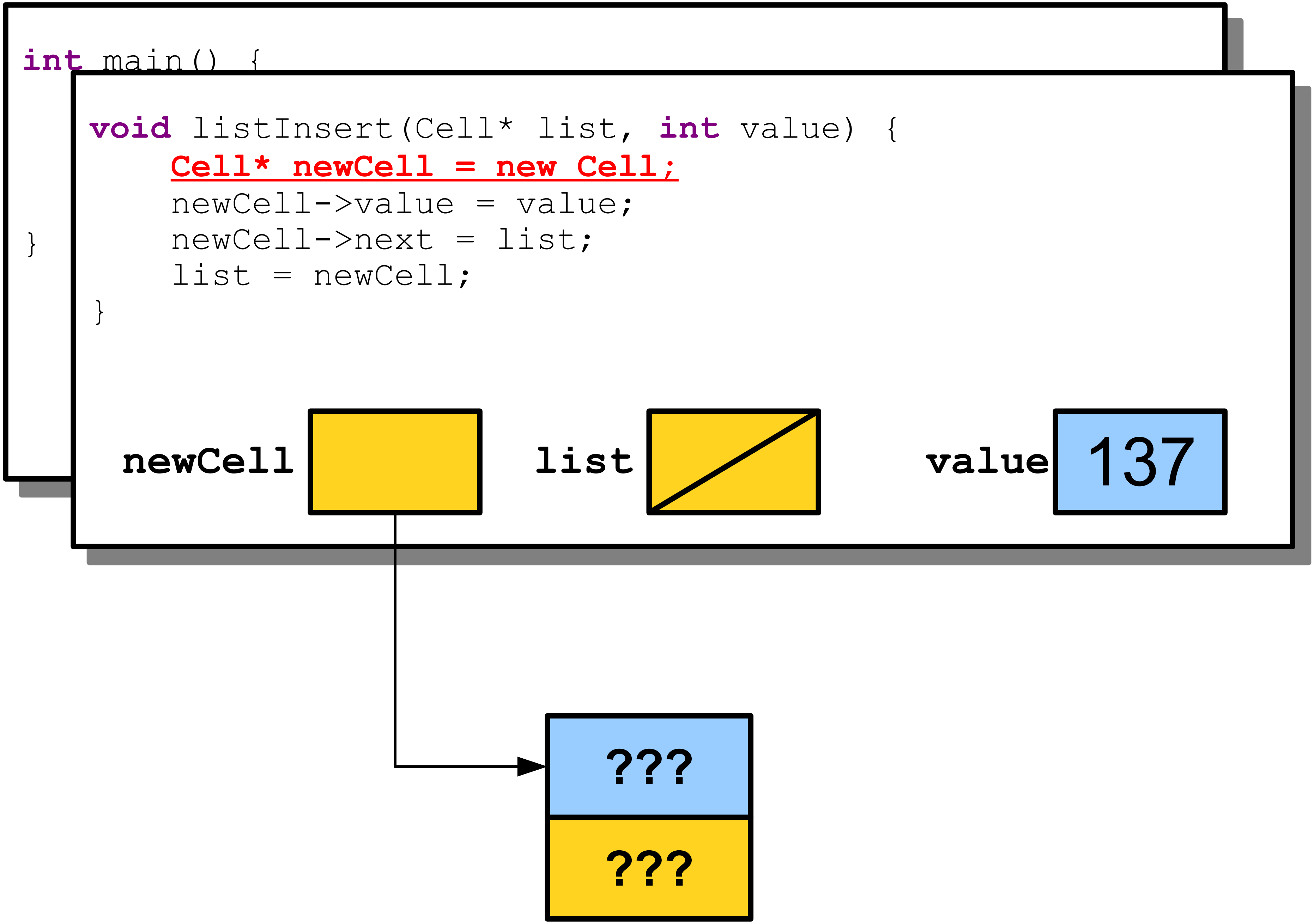
```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



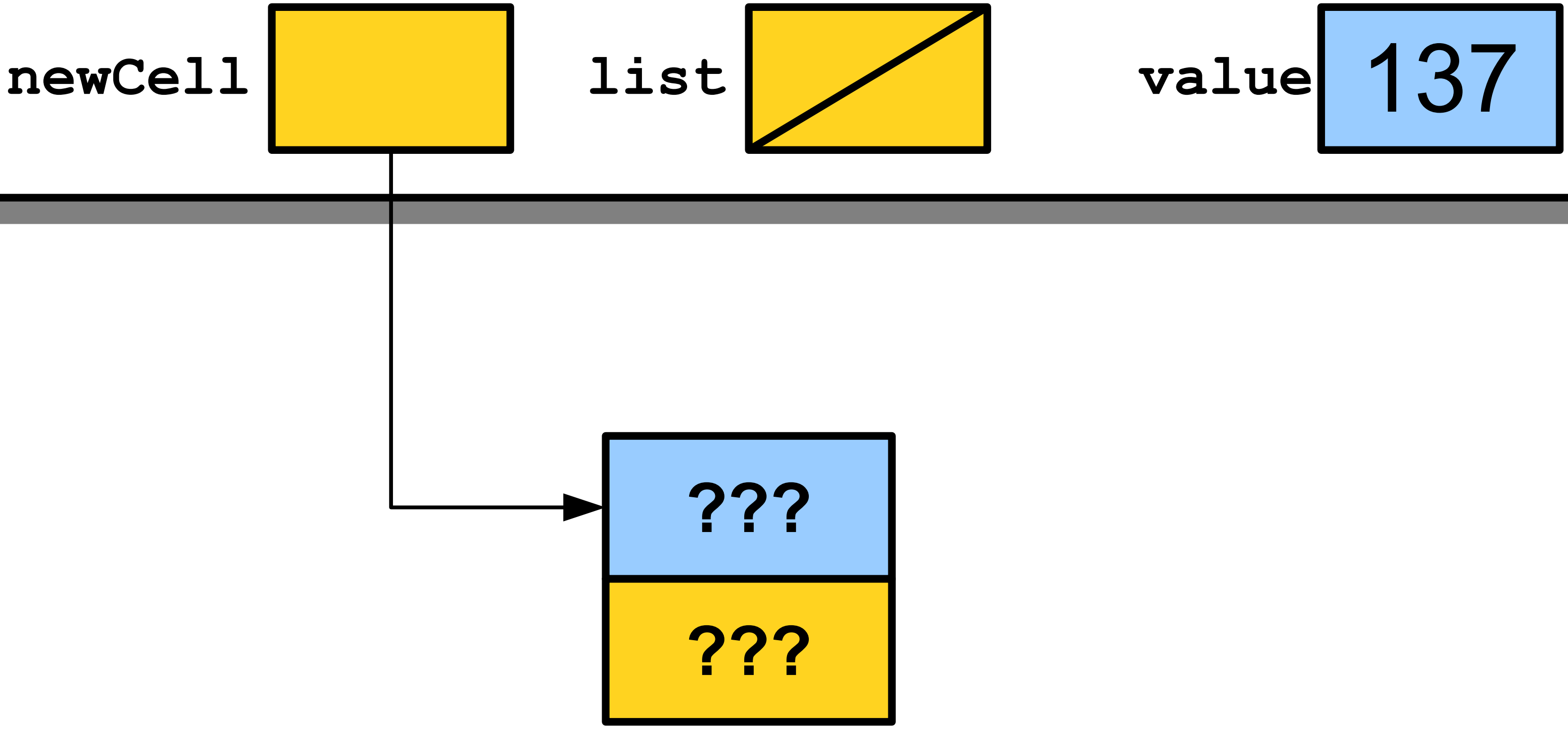


```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

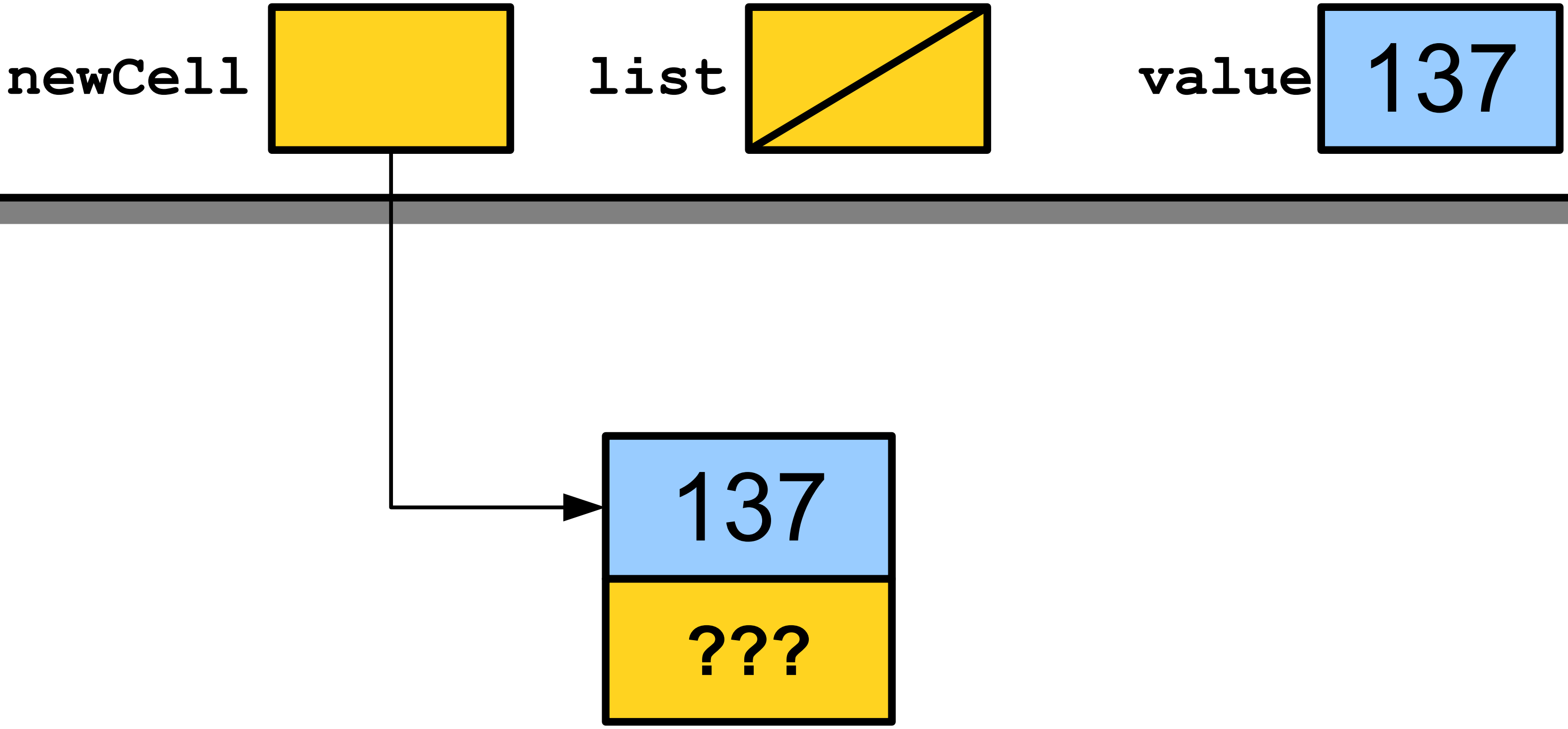
```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



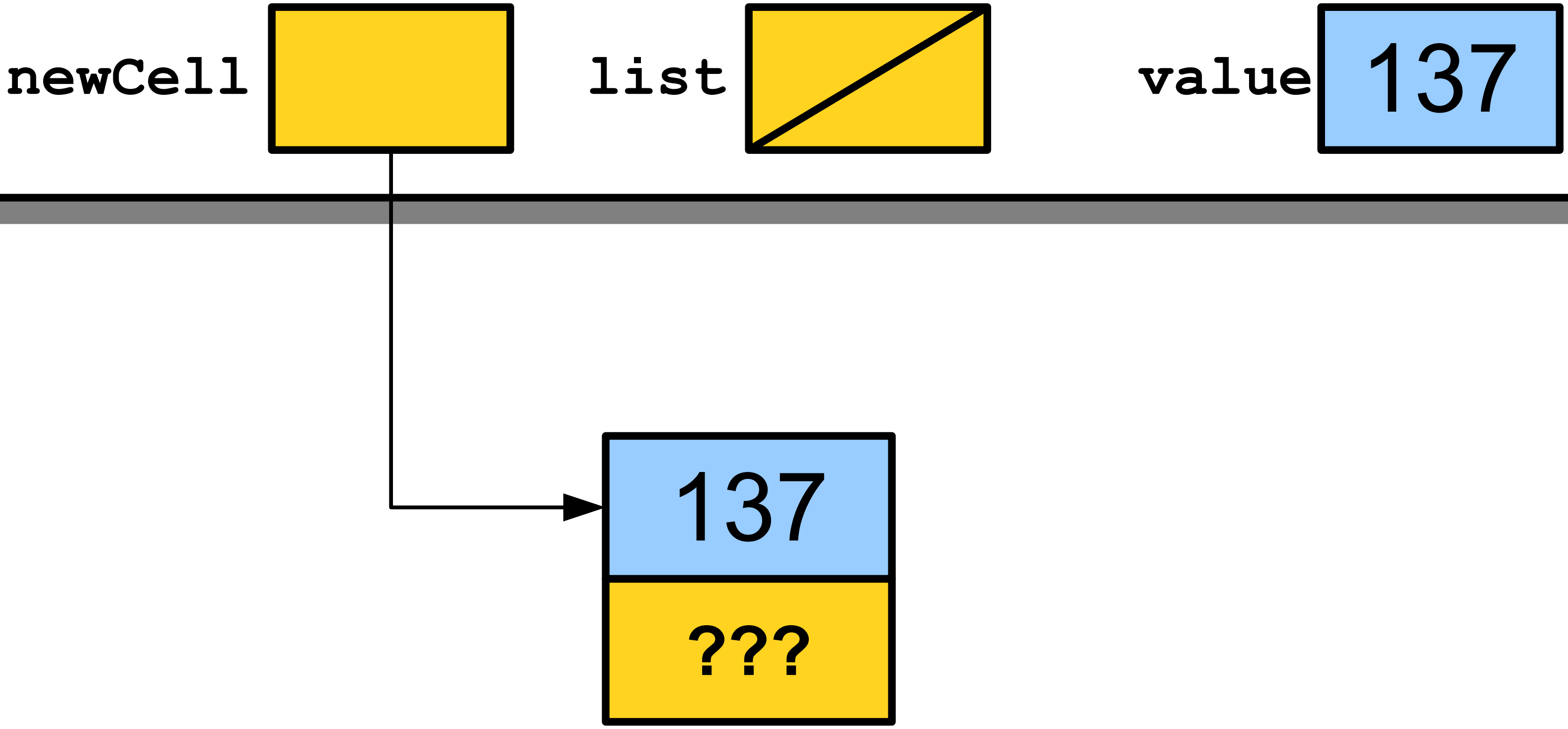
```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



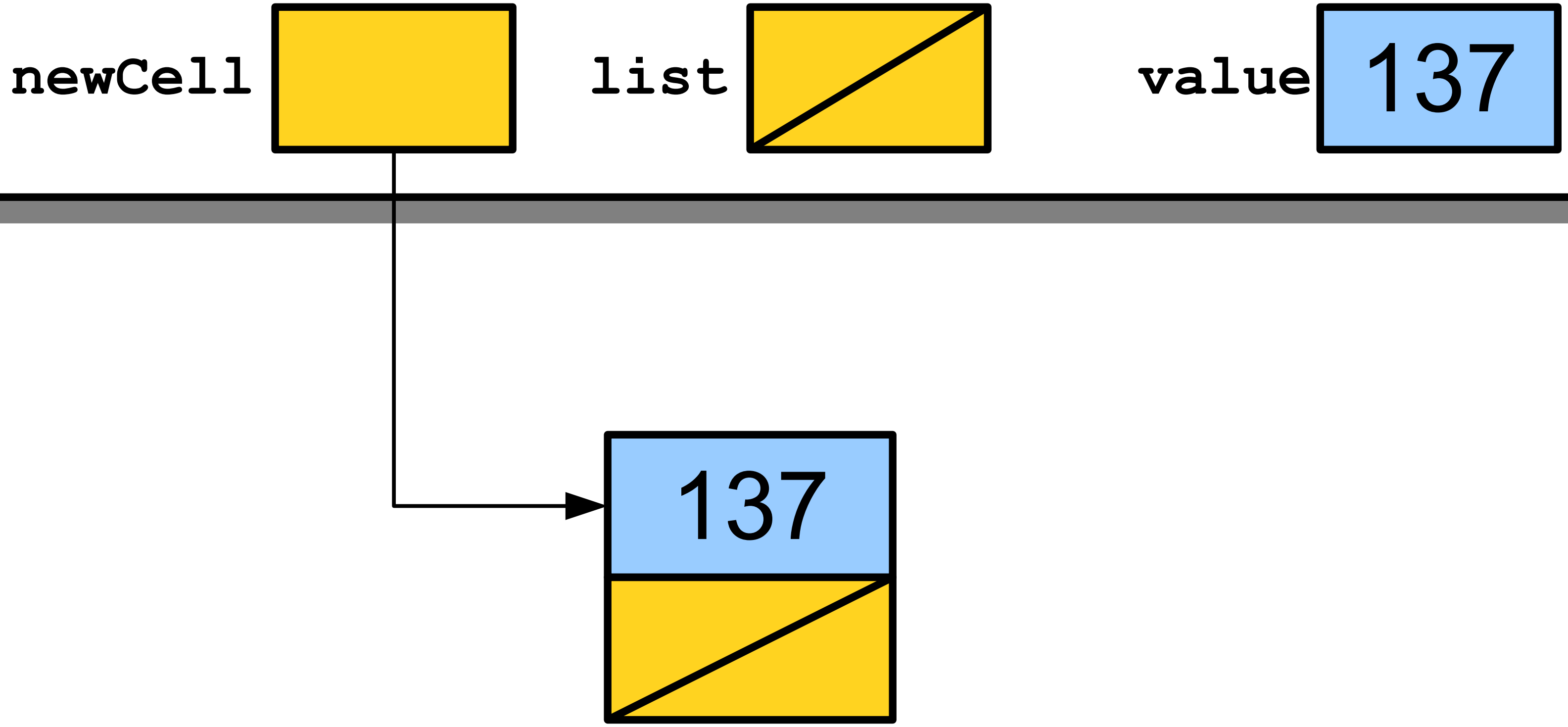
```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



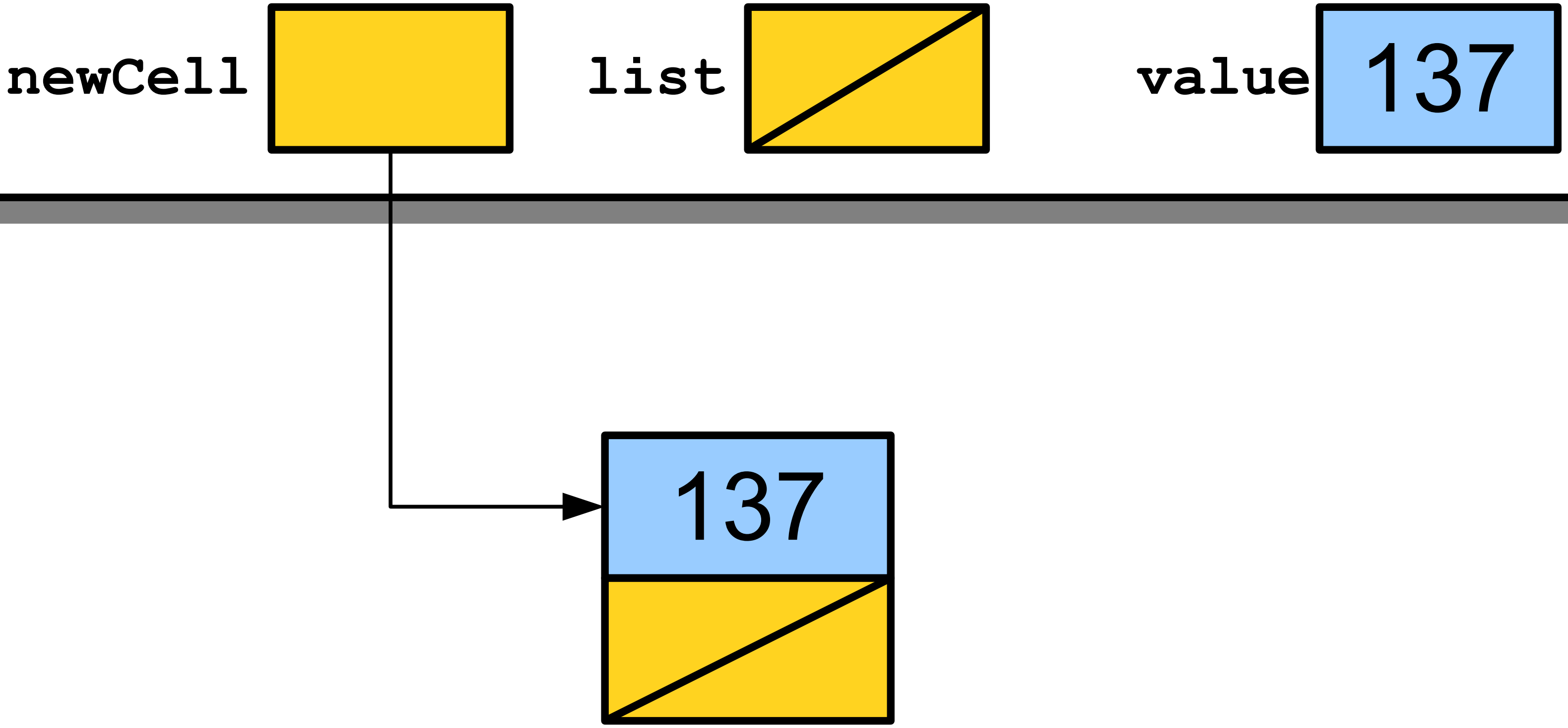
```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```



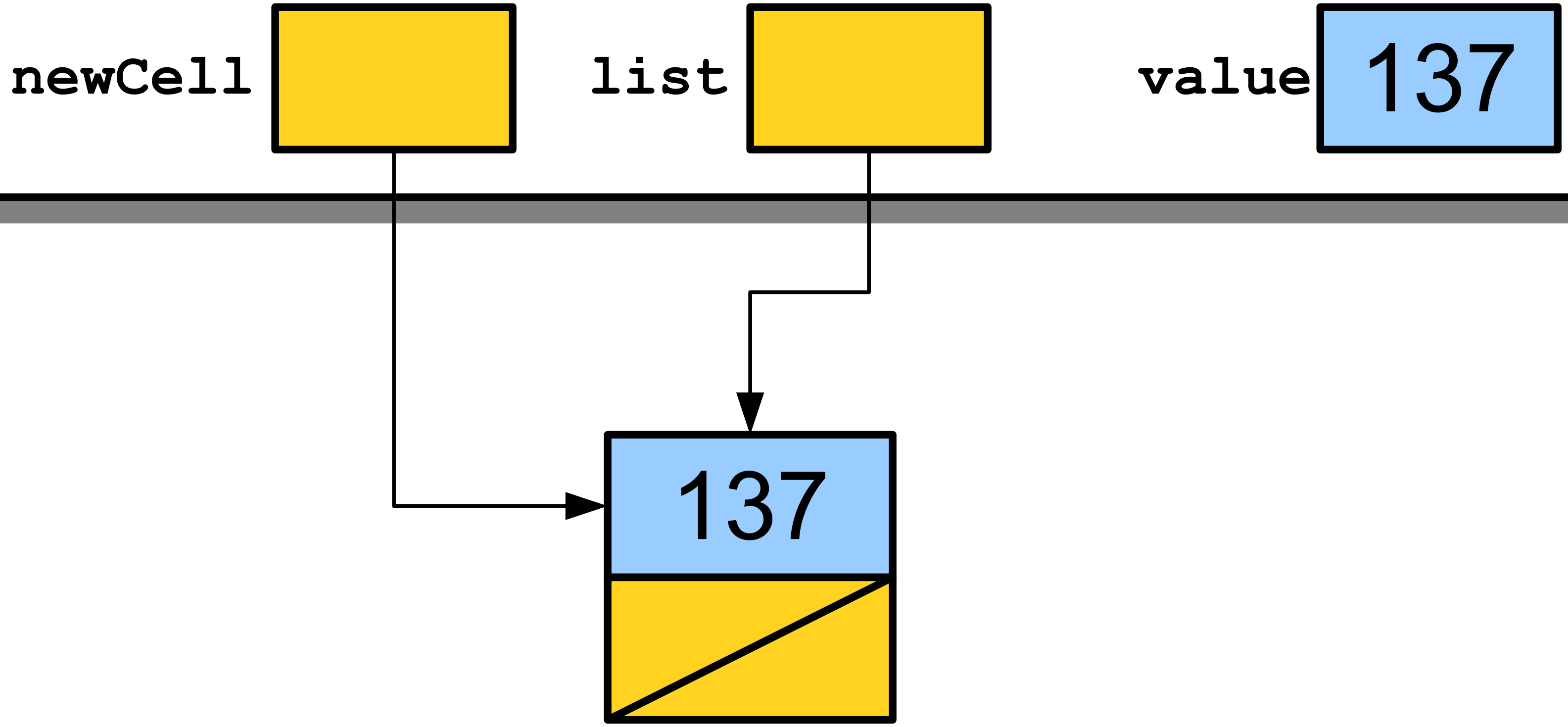
```
int main() {

}
```

```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

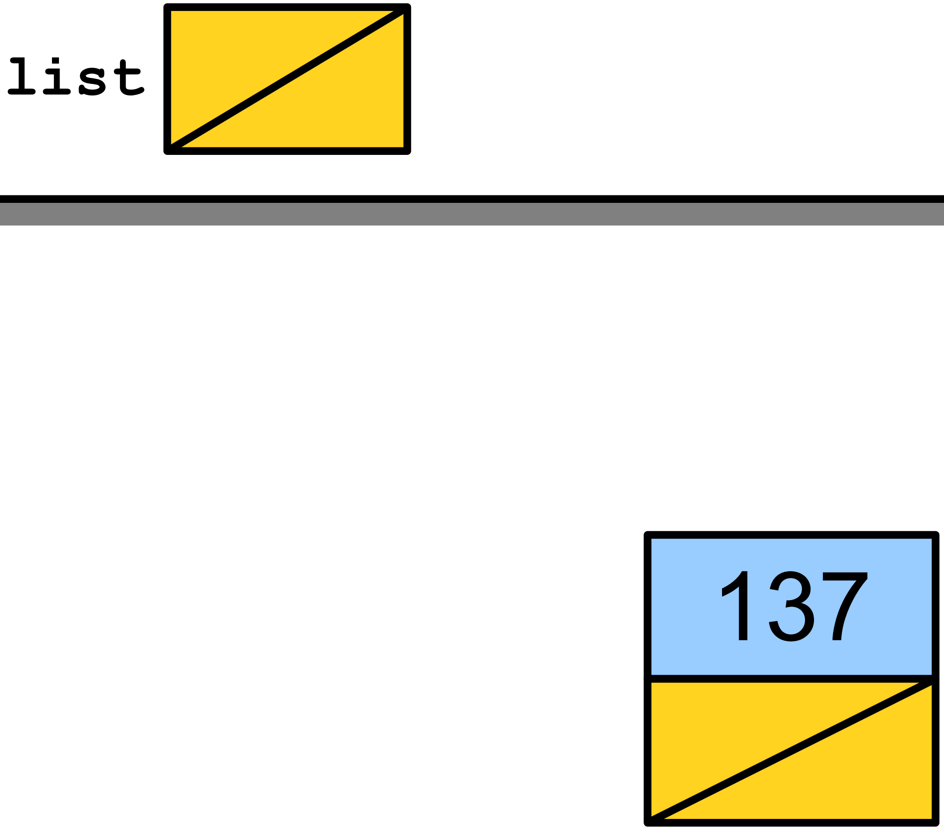


```
int main() {

}
```

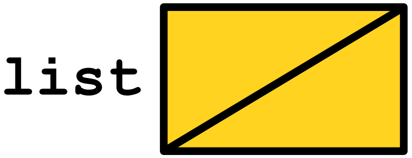
```
void listInsert(Cell* list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

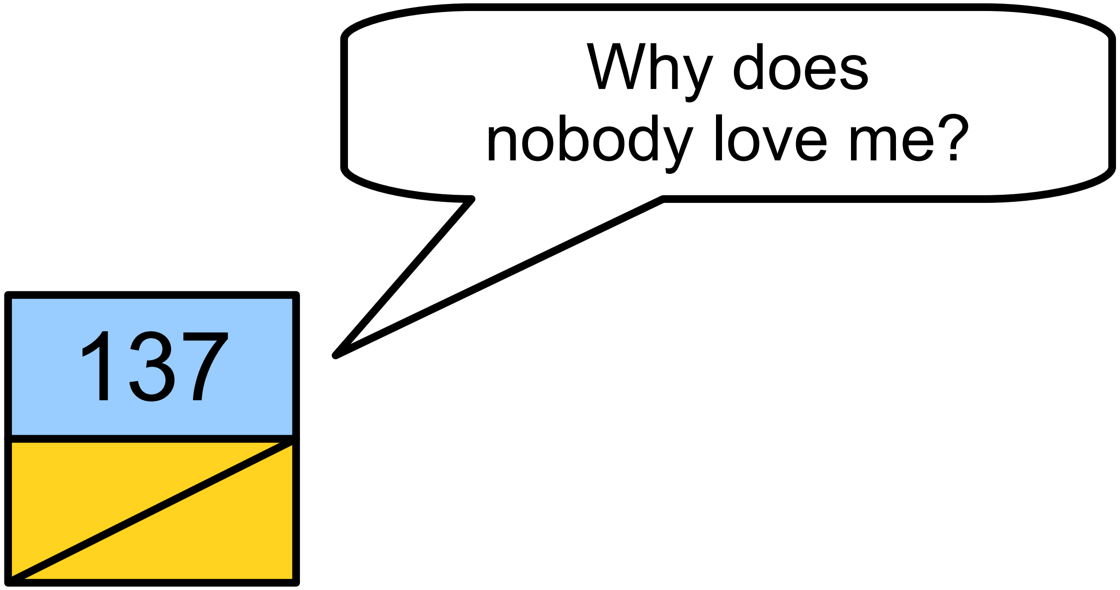


```
int main() {
    Cell* list = NULL;
    listInsert(list, 137);
    listInsert(list, 42);
    listInsert(list, 271);
}
```



```
int main() {
    Cell* list = NULL;
    listInsert(list, 137);
    listInsert(list, 42);
    listInsert(list, 271);
}
```



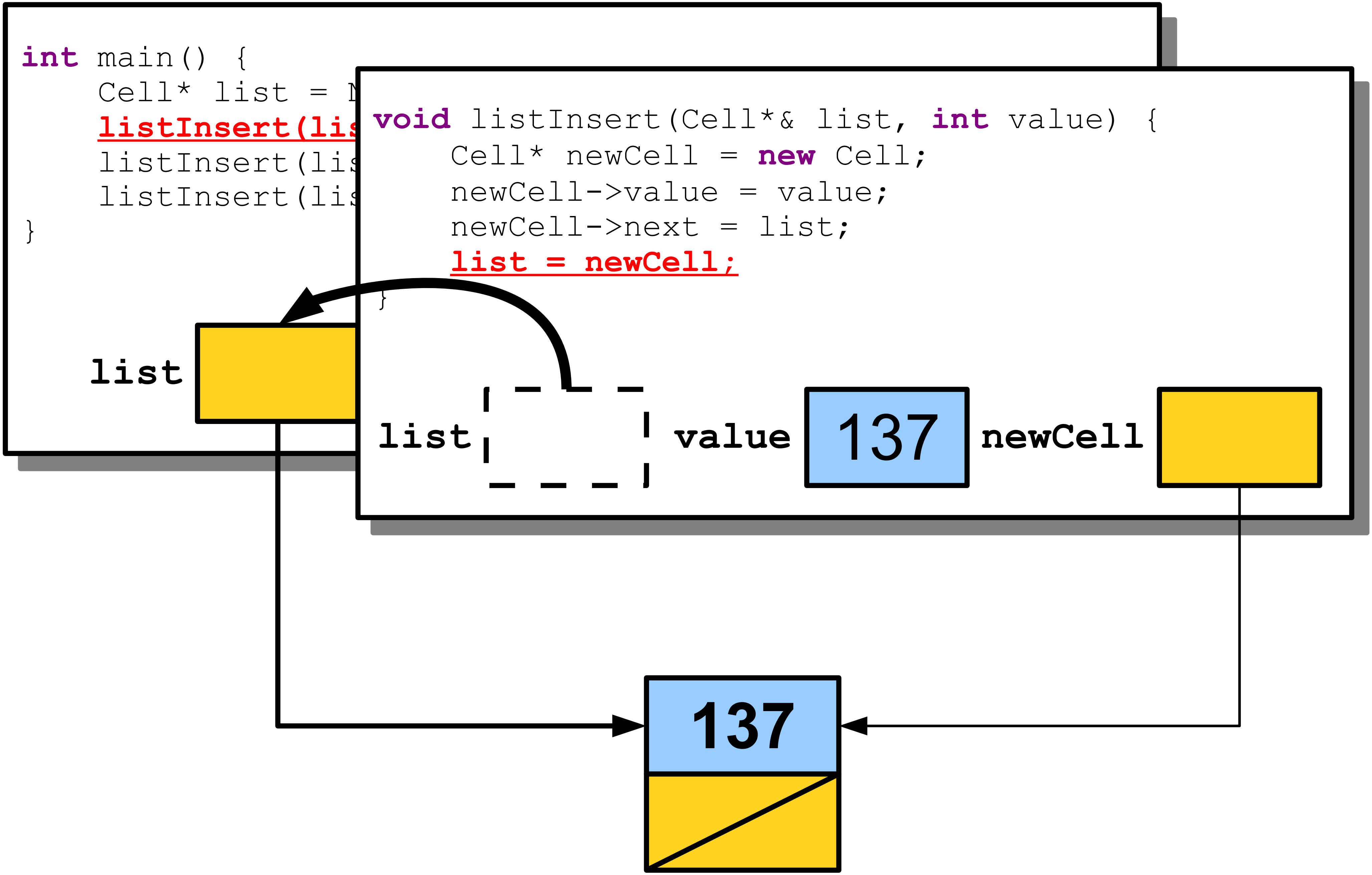


Pointers by Reference

- In order to resolve this problem, we must pass the linked list pointer by reference.
- Our new function:

```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    cell->value = value
    cell->next = list;
    list = cell;
}
```

This is a **reference to a pointer to a Cell**. It's often useful to read this from the right to the left.



```
int main() {
    Cell* list = N
    listInsert(lis
    listInsert(lis
    listInsert(lis
}
```

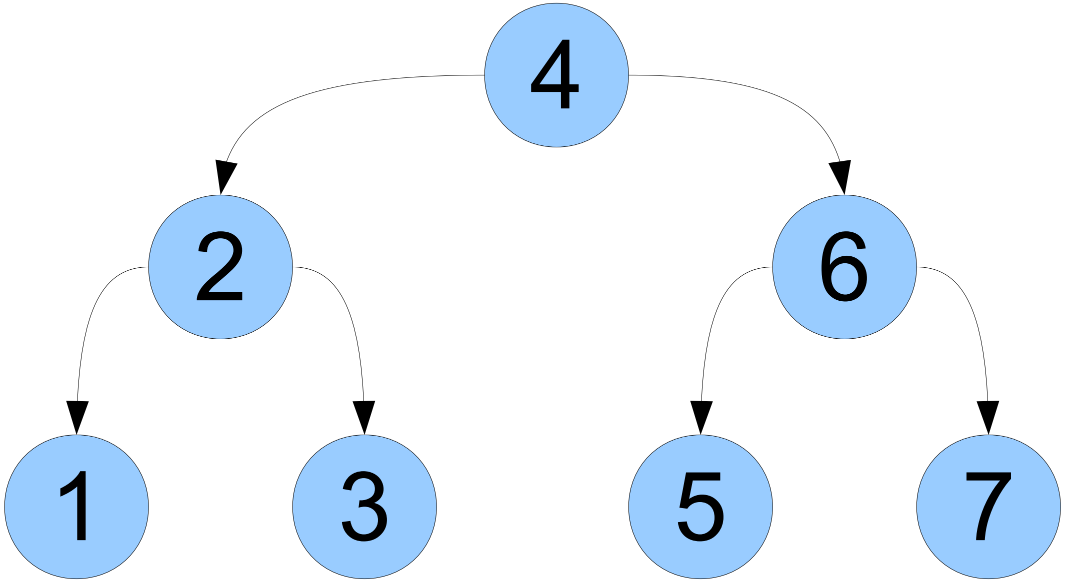
```
void listInsert(Cell*& list, int value) {
    Cell* newCell = new Cell;
    newCell->value = value;
    newCell->next = list;
    list = newCell;
}
```

Insertion
(`bst.cpp`)

Insertion Order Matters

- Suppose we create a BST of numbers in this order:

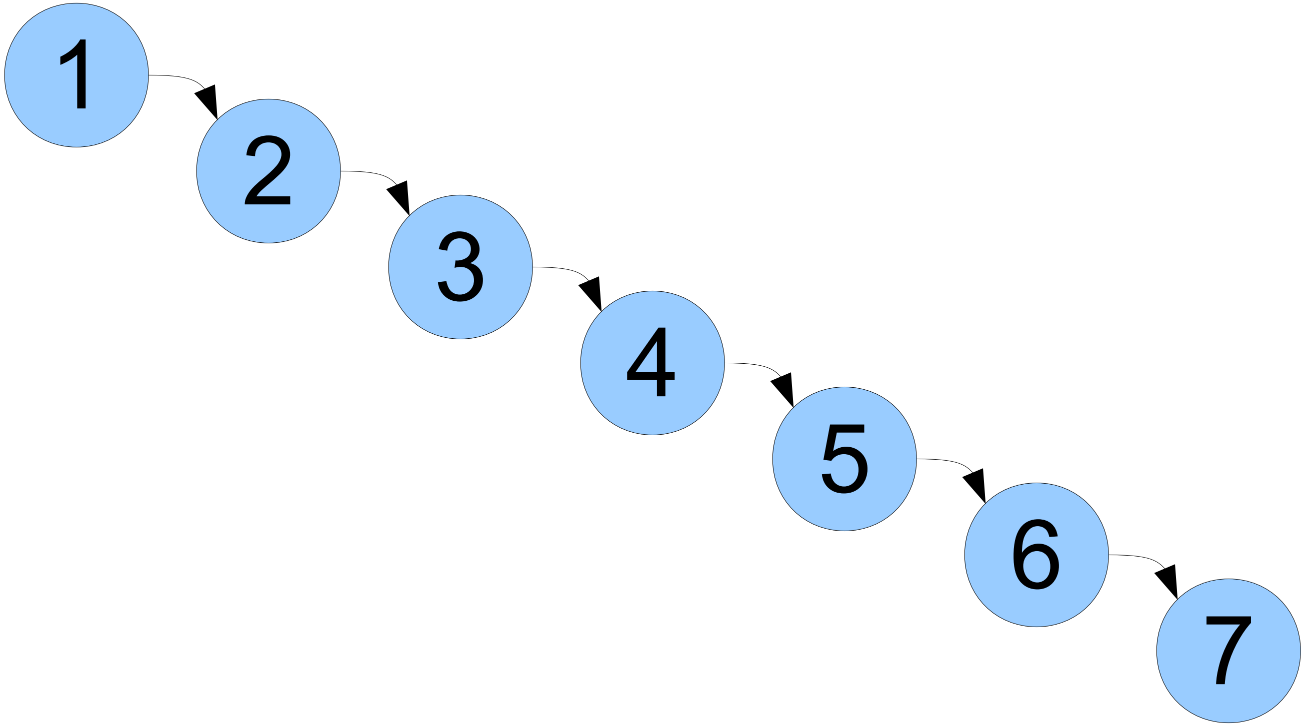
4, 2, 1, 3, 6, 5, 7



Insertion Order Matters

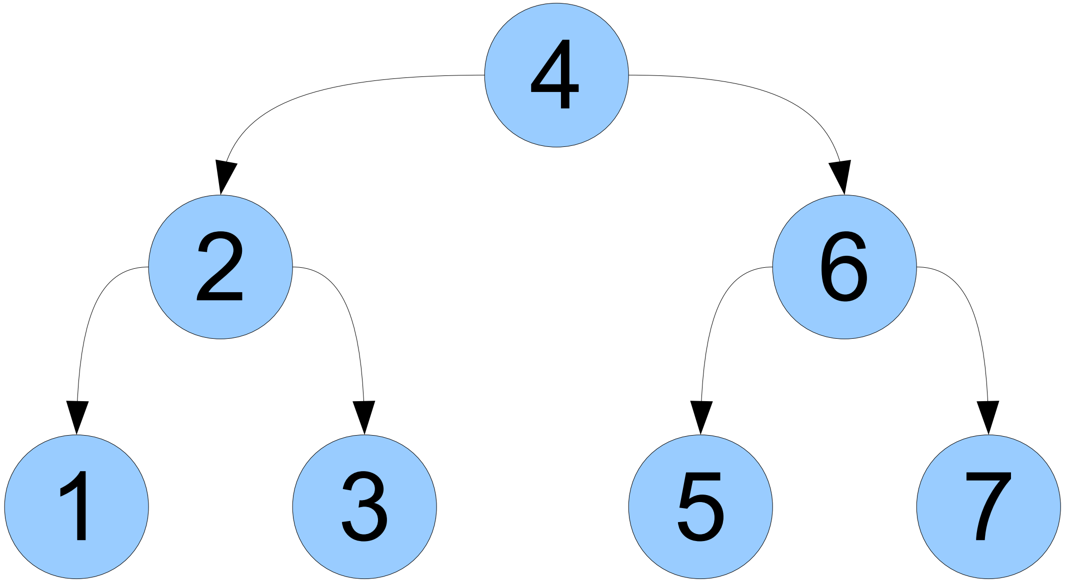
- Suppose we create a BST of numbers in this order:

1, 2, 3, 4, 5, 6, 7



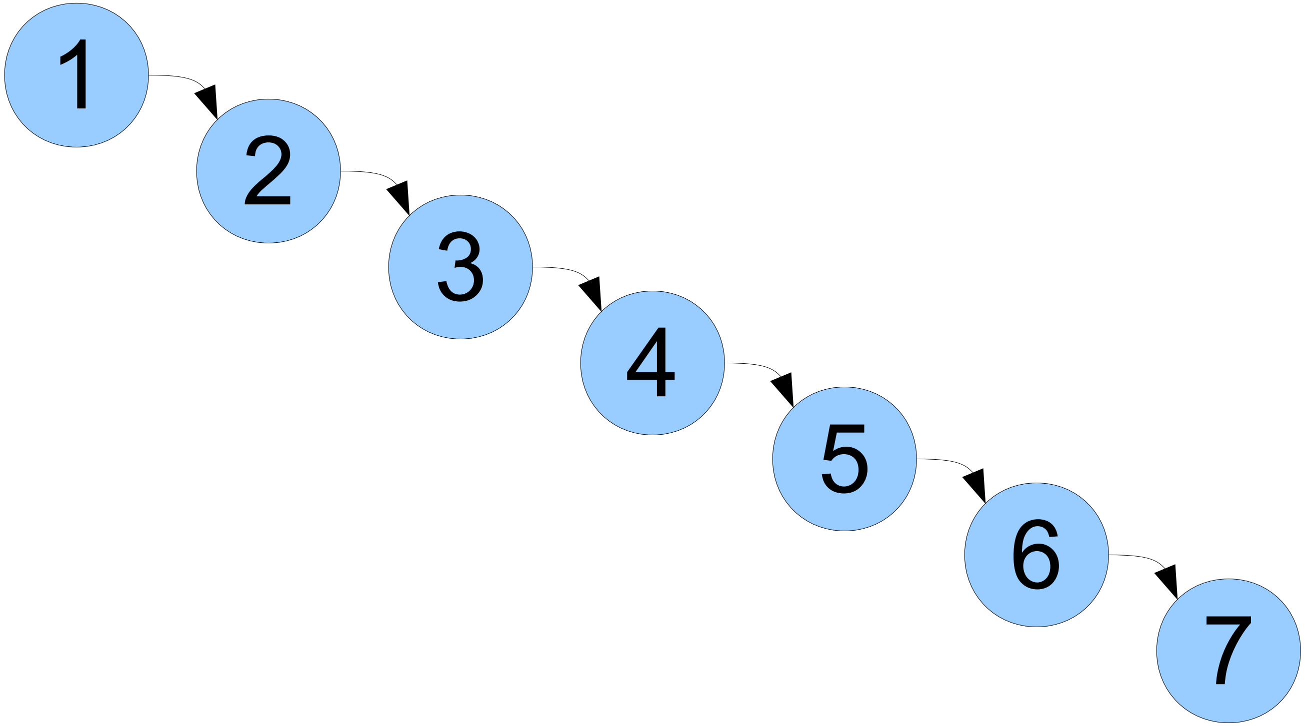
Tree Terminology

- The **height** of a tree is the number of nodes in the longest path from the root to a leaf.



Tree Terminology

- The **height** of a tree is the number of nodes in the longest path from the root to a leaf.



Efficiency of Insertion

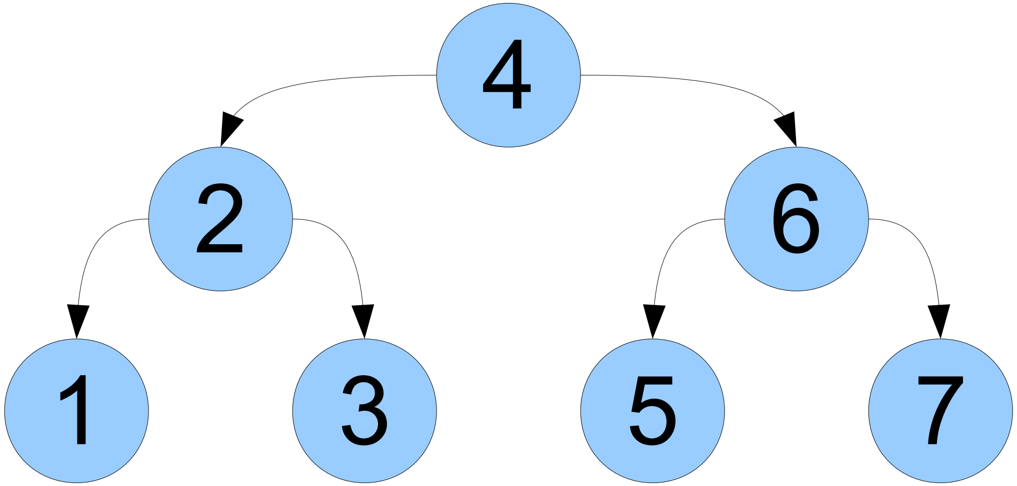
- What is the big-O complexity of adding a node to a tree?
- Depends on the height of a tree!
- Worst-case: have to take the longest path down to find where the node goes.
- Time is **O(*h*)**, where *h* is the height of the tree.

Tree Heights

- What are the maximum and minimum heights of a tree with *n* nodes?
- Maximum height: all nodes in a chain. Height is **O(*n*)**.

Tree Heights

- What are the maximum and minimum heights of a tree with *n* nodes?
- Maximum height: all nodes in a chain. Height is **O(*n*)**.
- Minimum height: Tree is as complete as possible. Height is **O(log *n*)**.

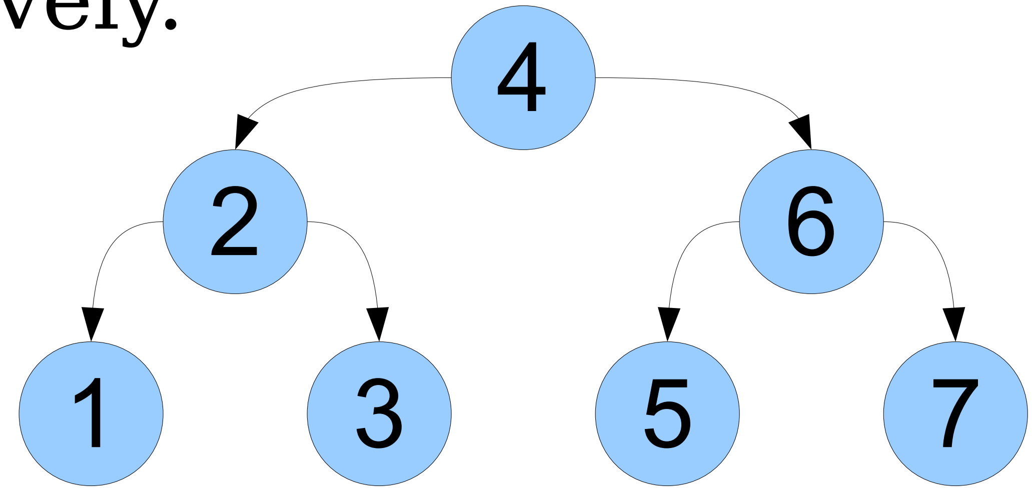


Keeping the Height Low

- There are many modifications of the binary search tree designed to keep the height of the tree low (usually **O(log *n*)**).
- A **self-balancing binary search tree** is a binary search tree that automatically adjusts itself to keep the height low.

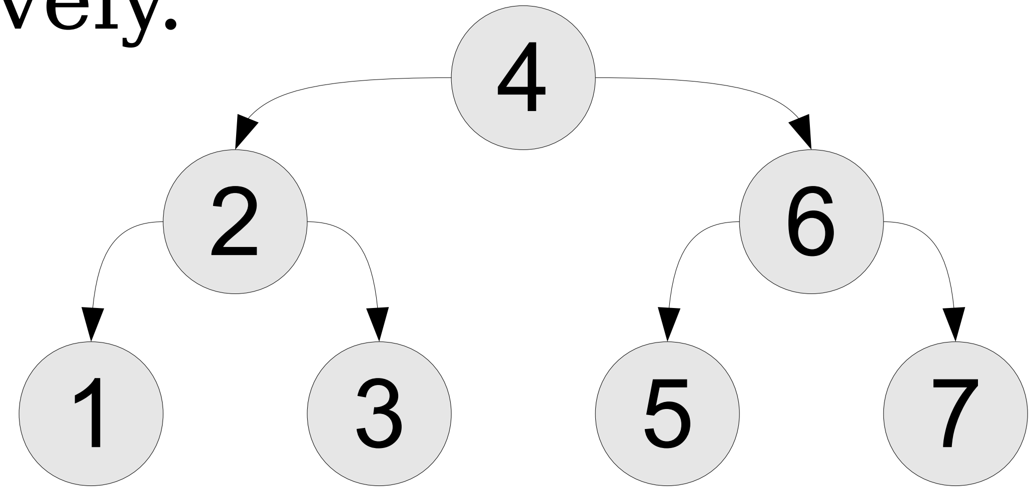
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



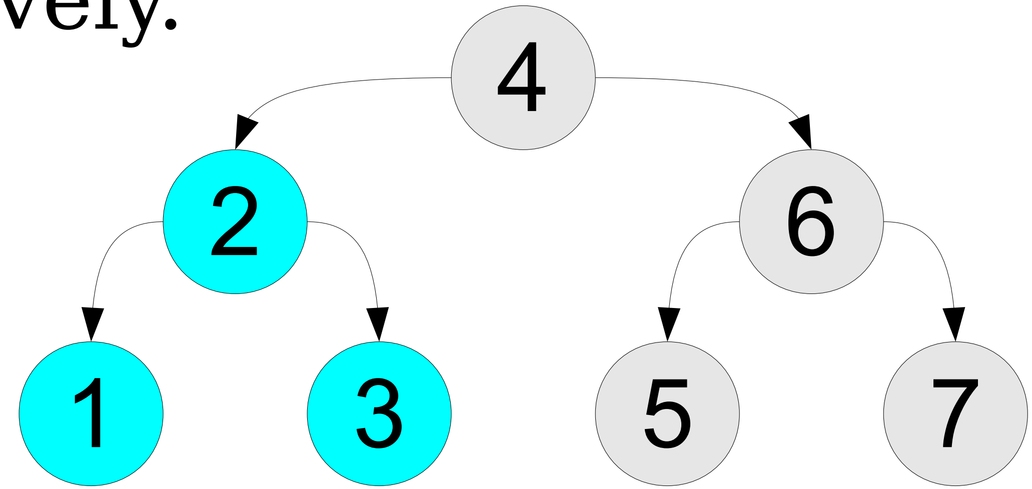
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



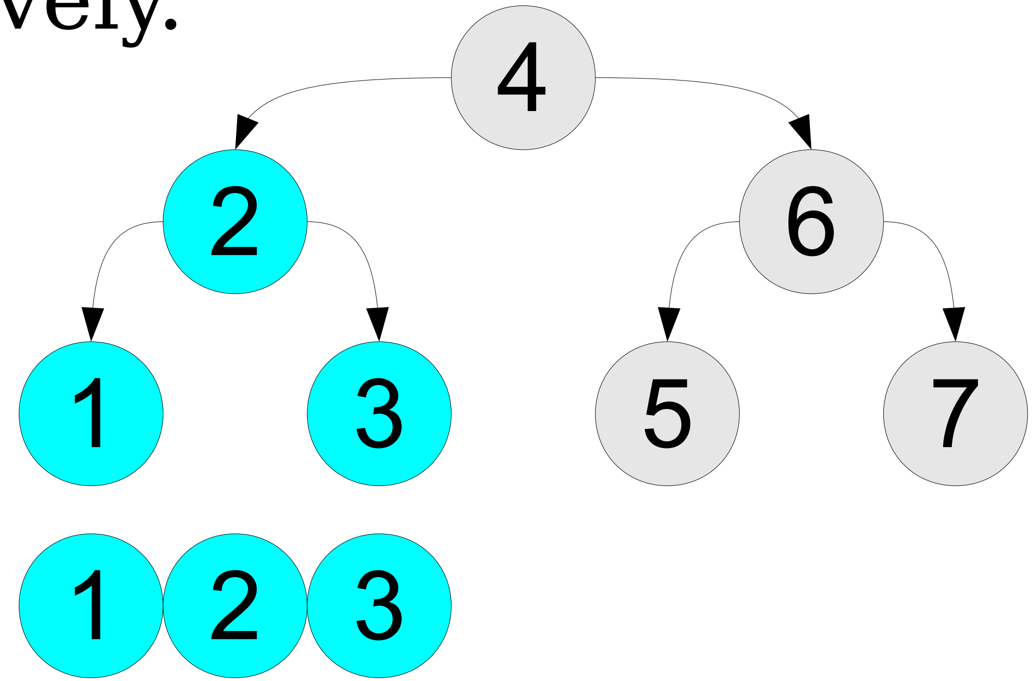
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



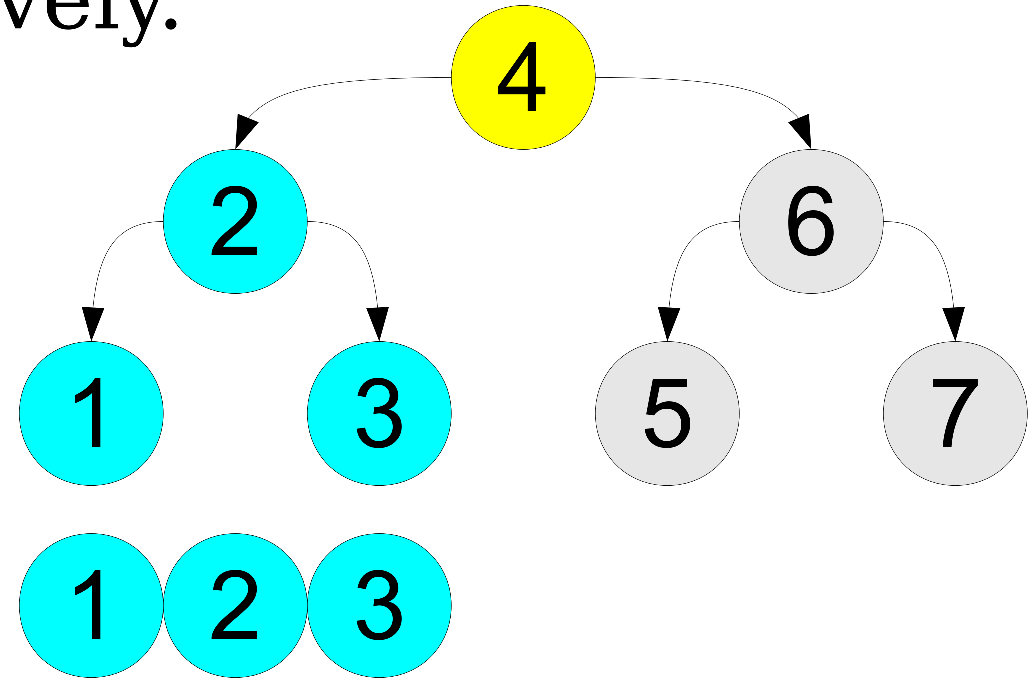
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



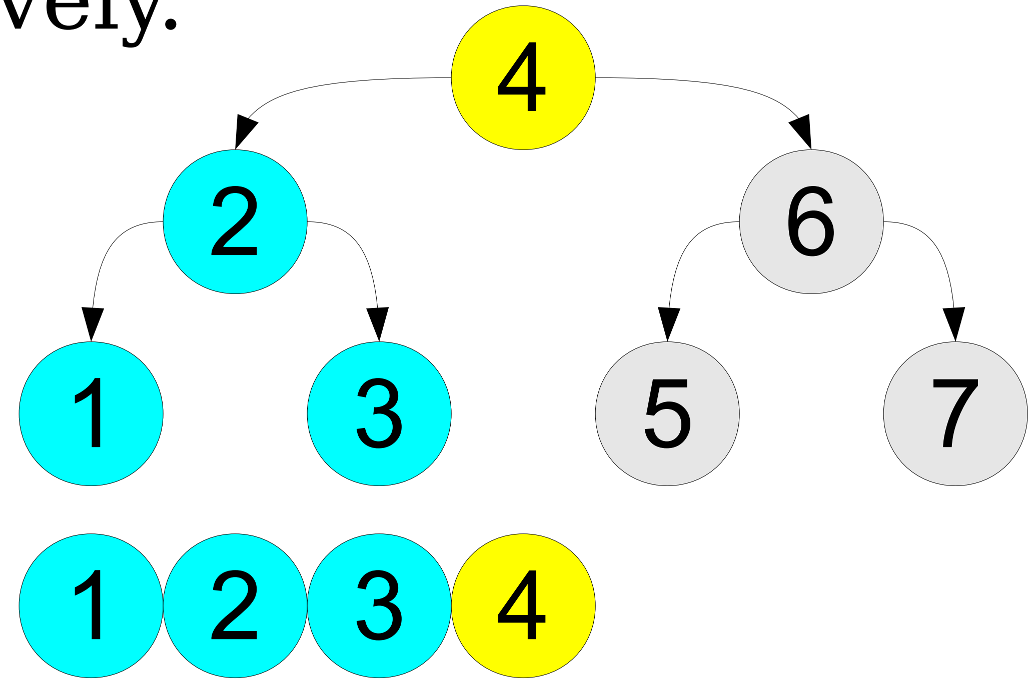
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



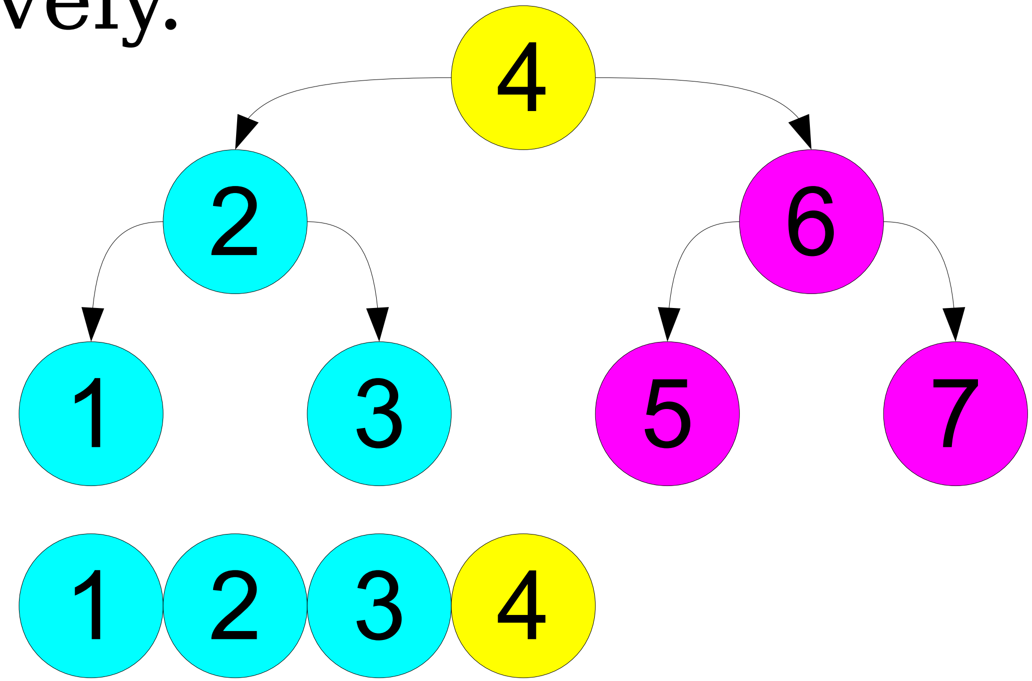
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



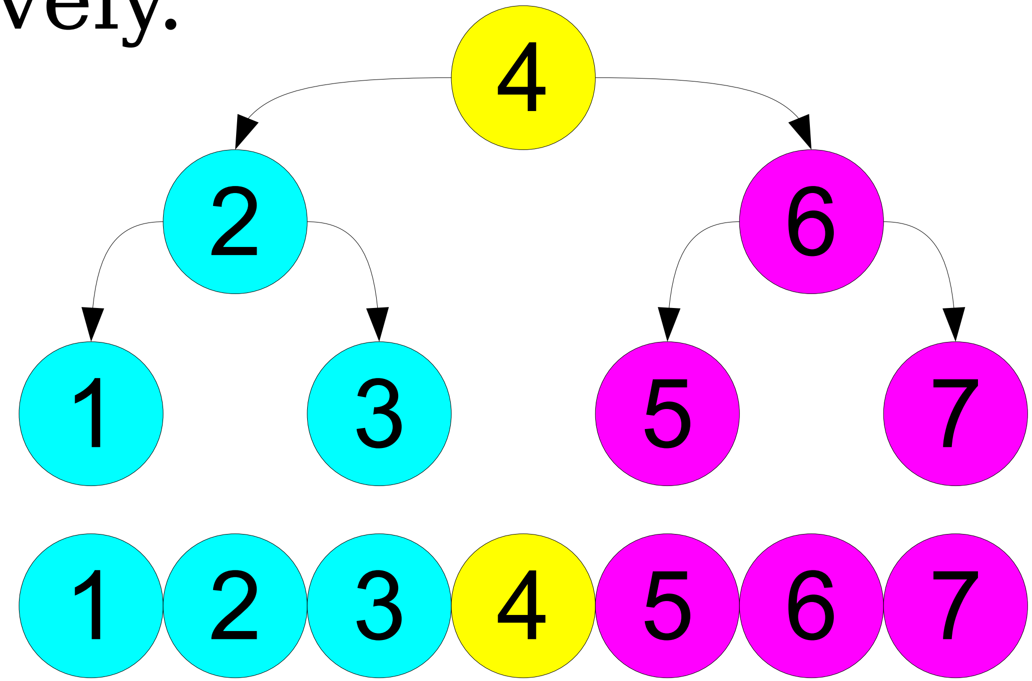
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



Walking a BST

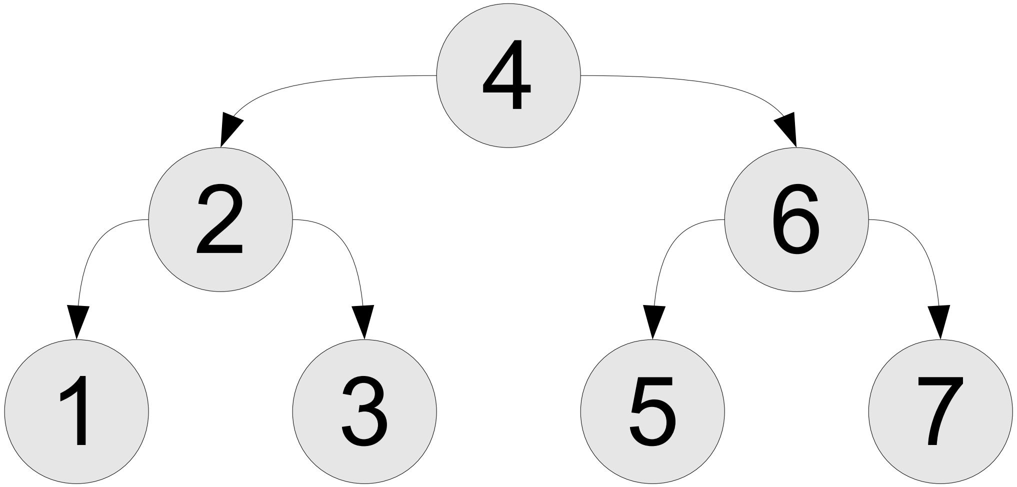
- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



Tree Traversals

- There are three general types of tree traversals:
- **Preorder**: Visit the node, then visit the children.
- **Inorder**: Visit the left child, then the node, then the right child.
- **Postorder**: Visit the children, then visit the node.

Walking a Tree



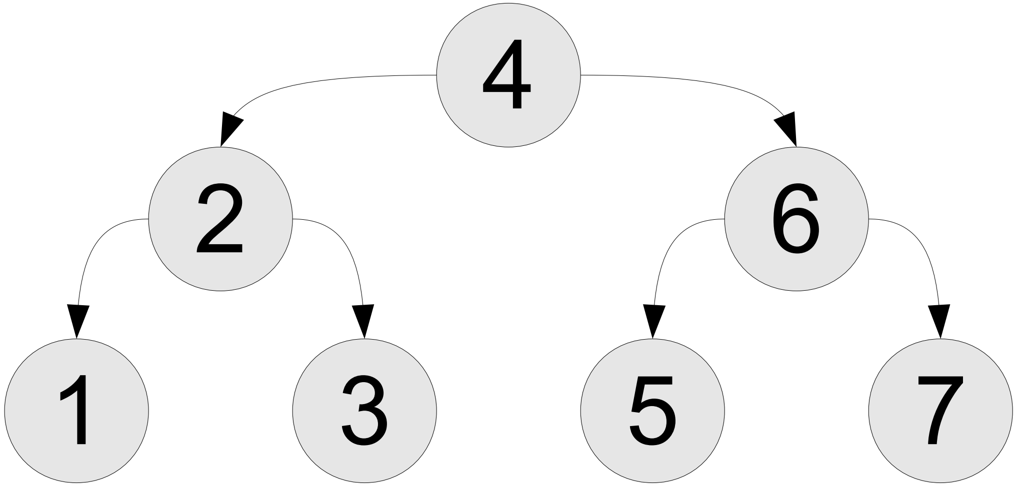
Inorder	1	2	3	4	5	6	7
Preorder	4	2	1	3	6	5	7
Postorder	1	3	2	5	7	6	4

Printing a Tree
(Pseudocode)

Printing a Tree
(`bst.cpp`)

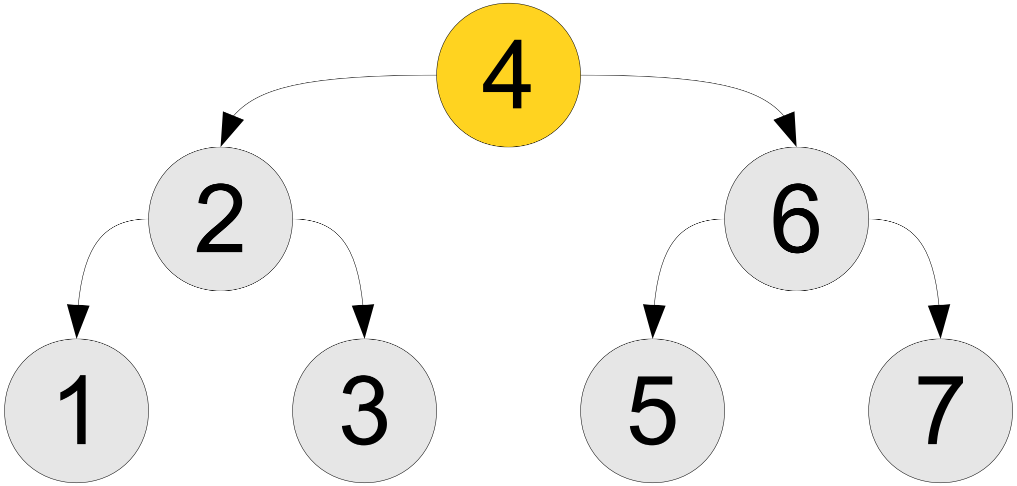
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



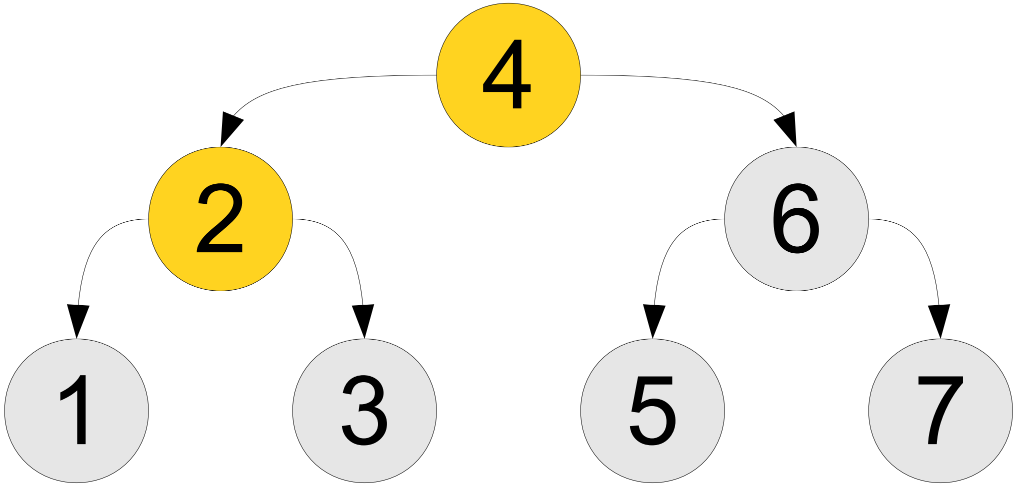
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



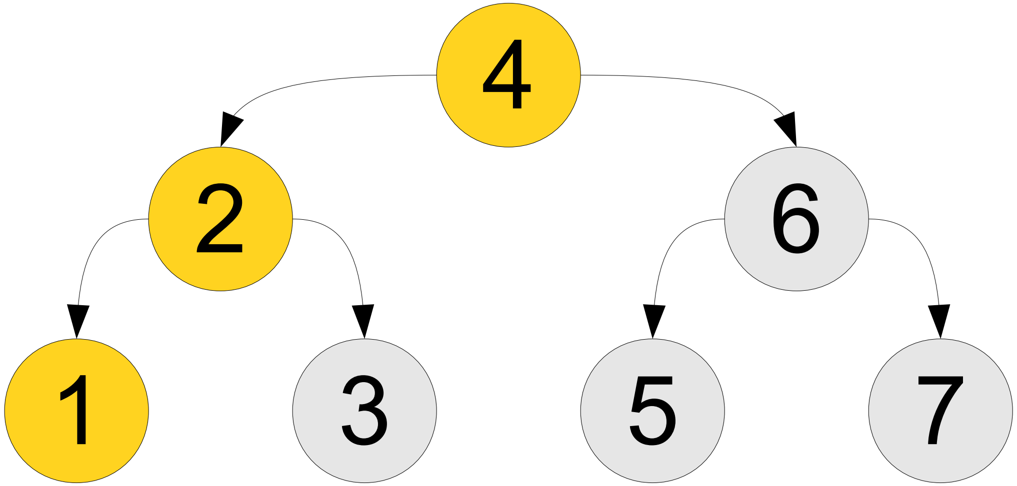
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



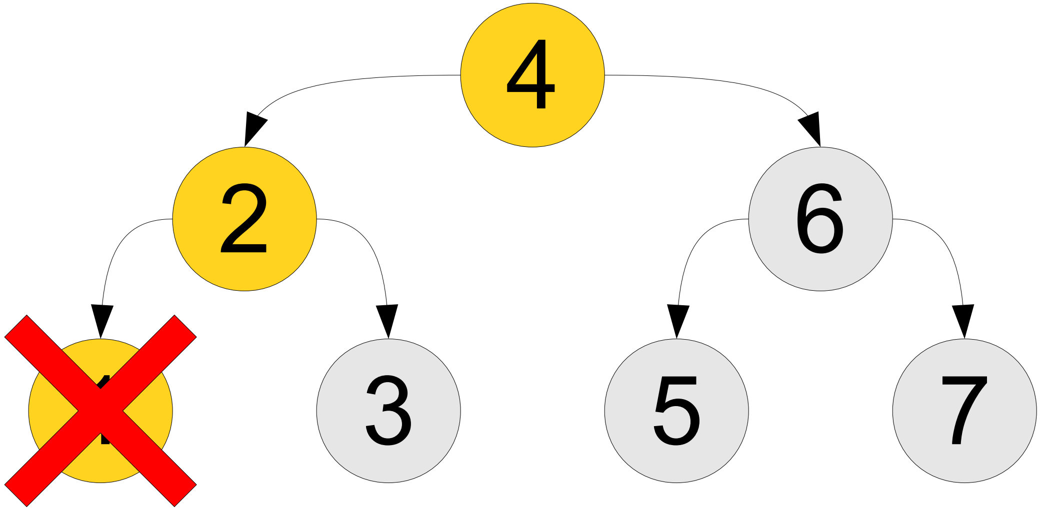
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



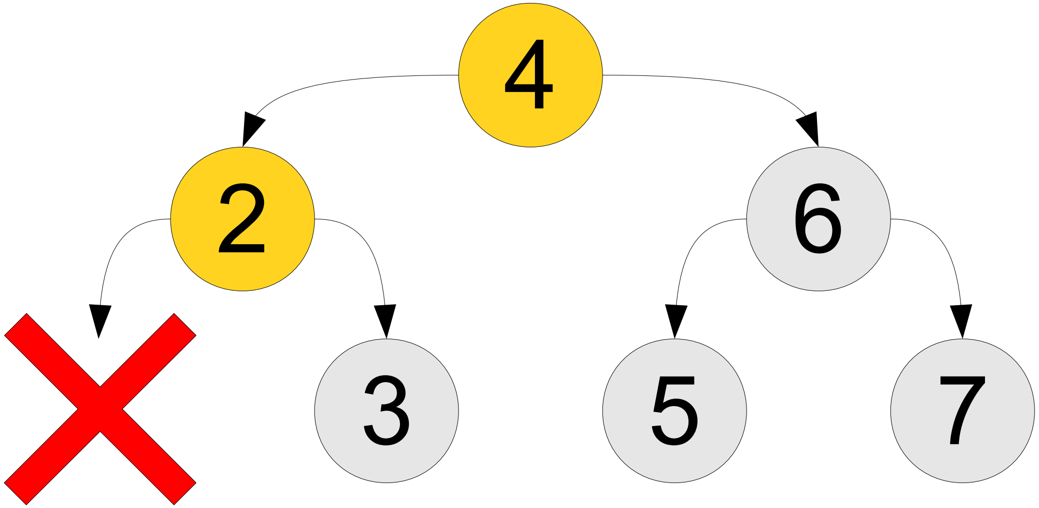
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



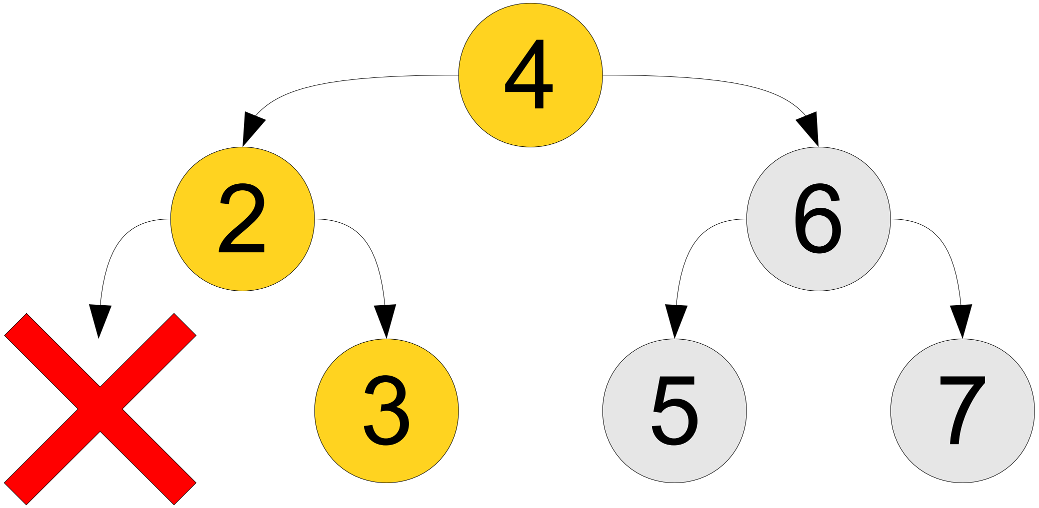
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



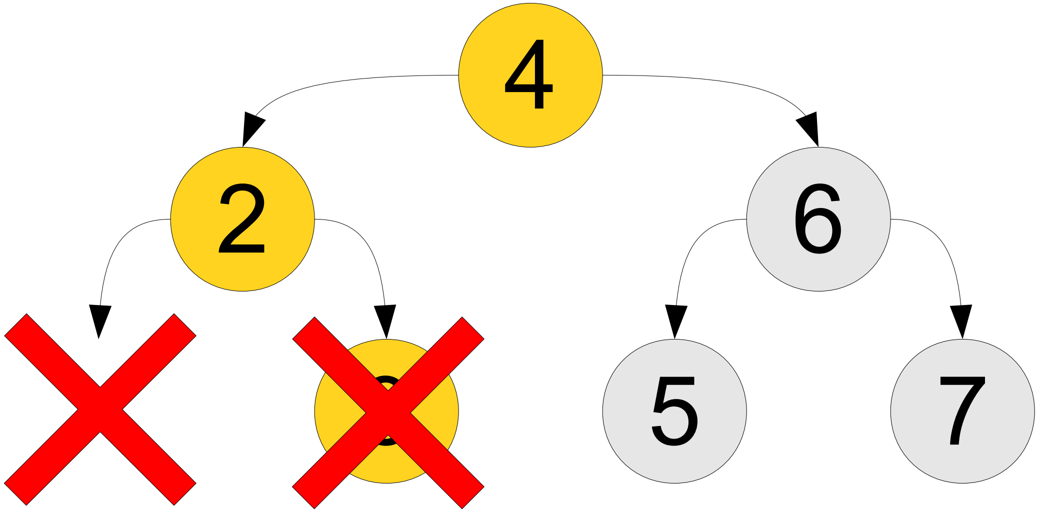
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



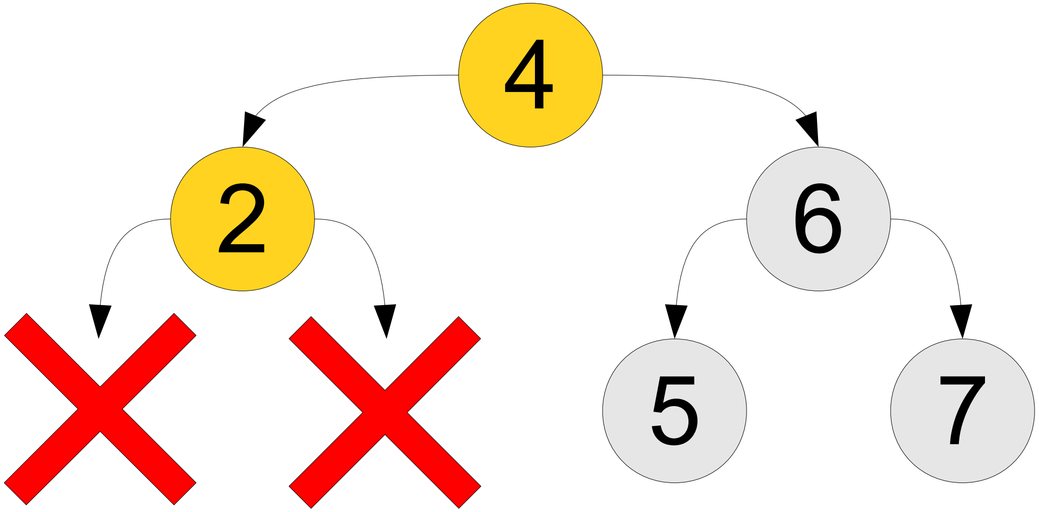
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



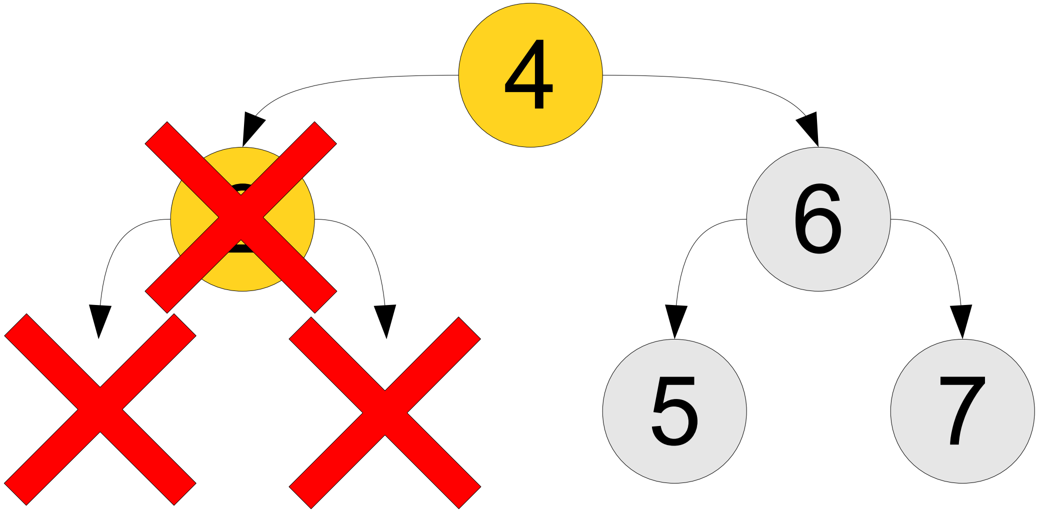
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



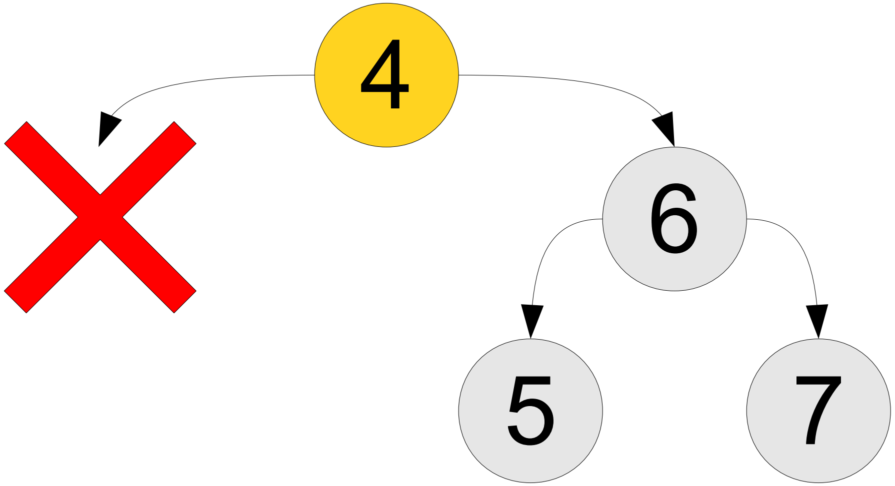
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



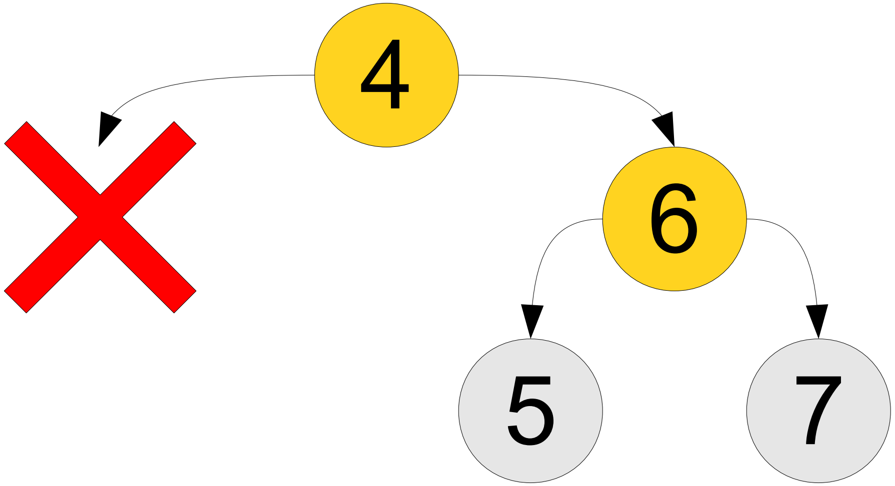
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



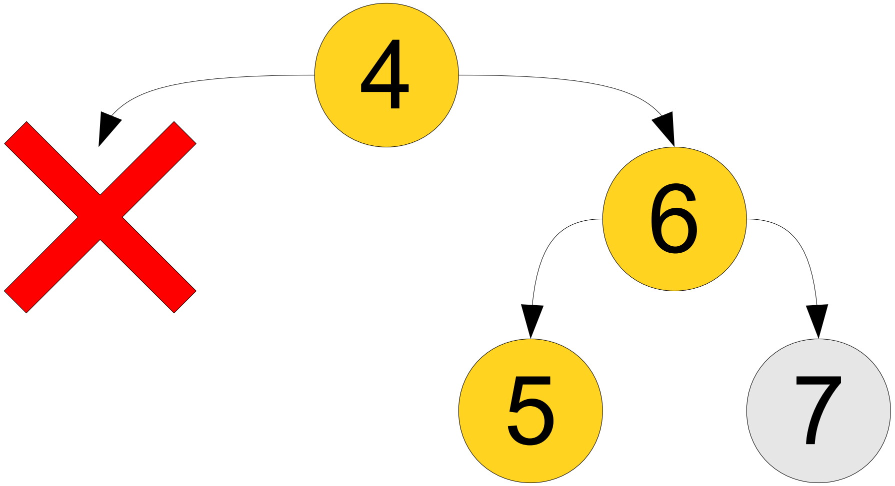
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



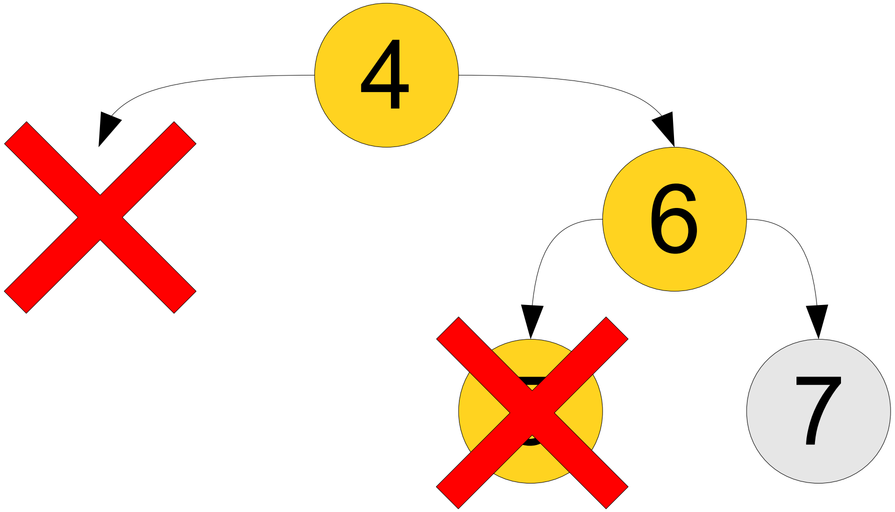
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



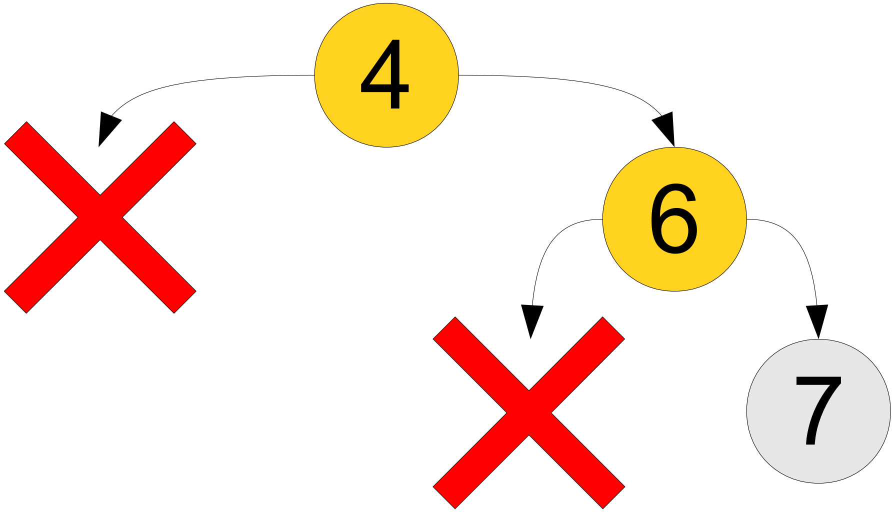
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



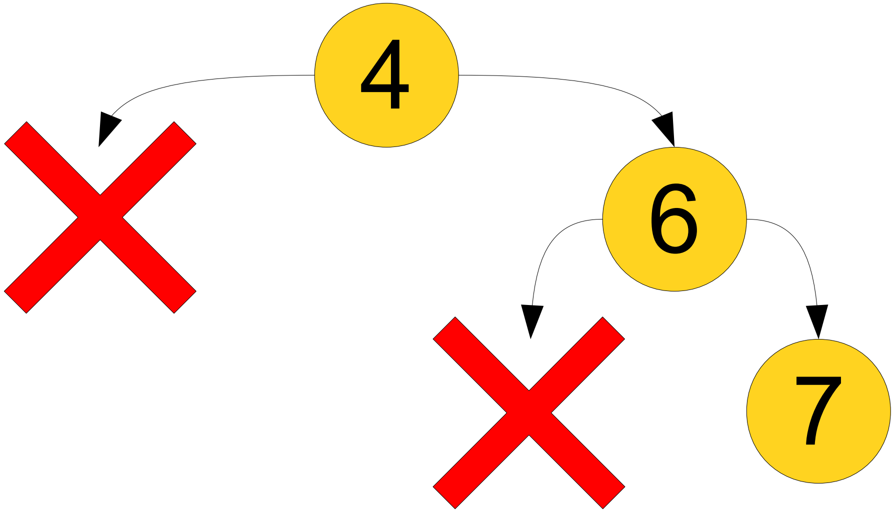
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



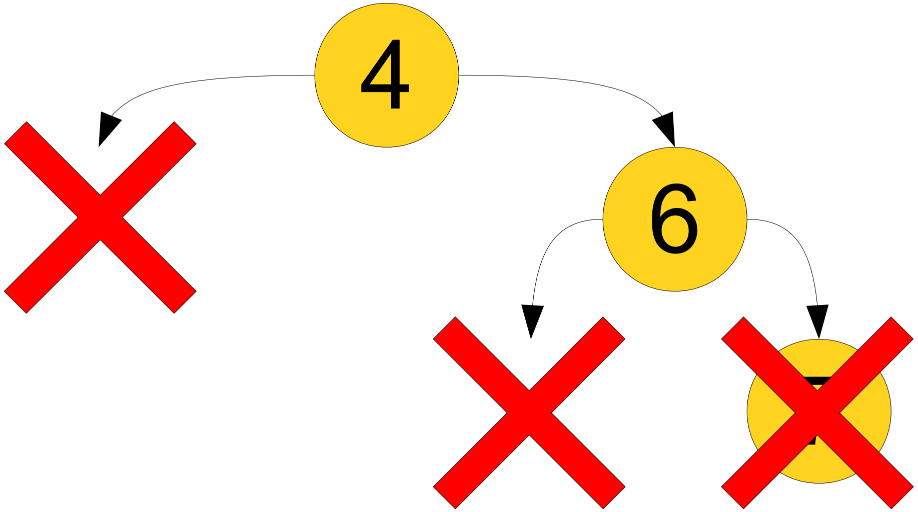
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



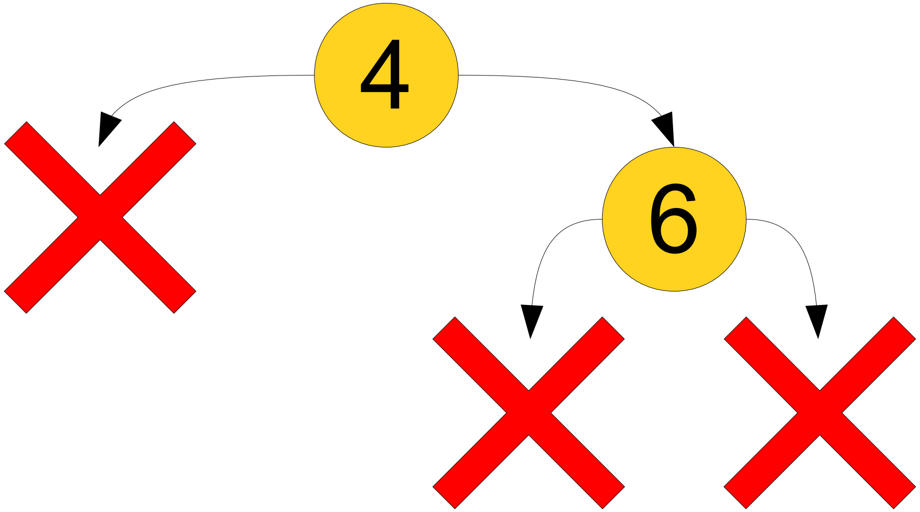
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



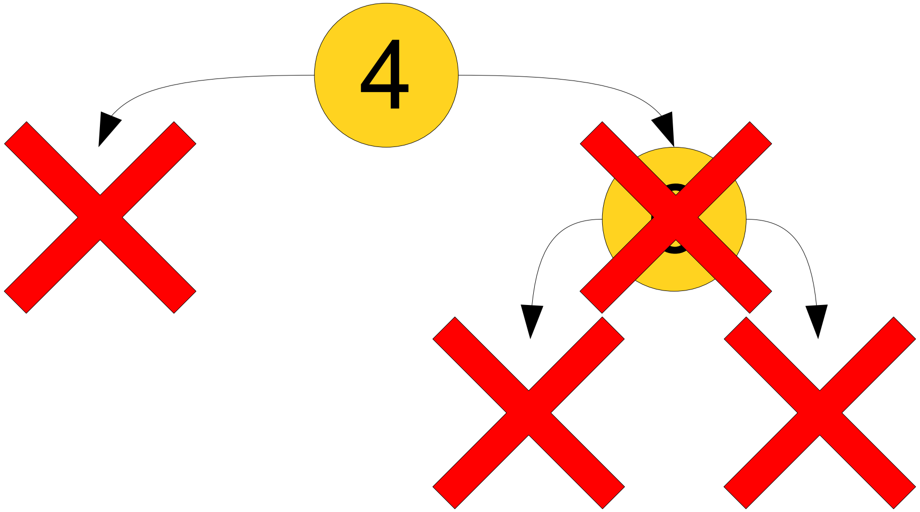
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



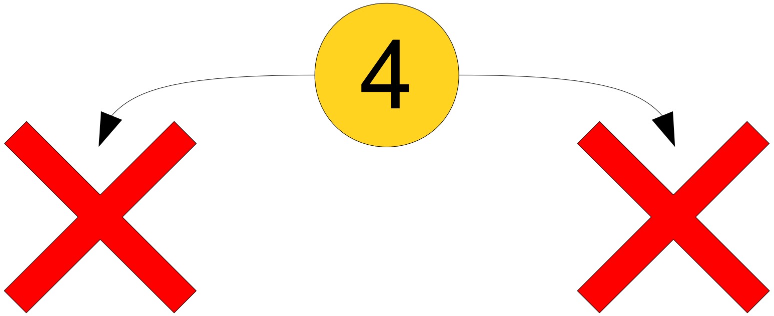
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



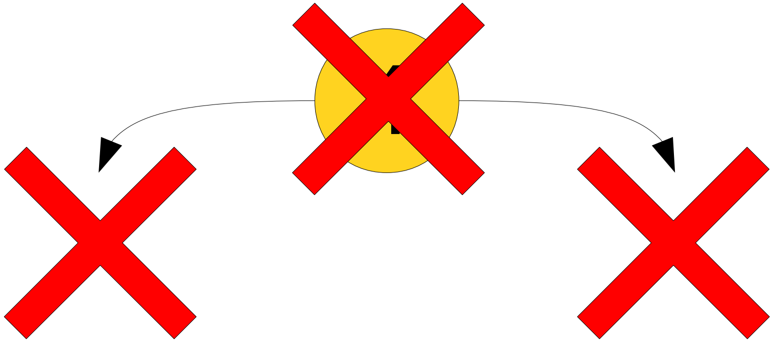
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.

✗

Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.

Freeing a Tree

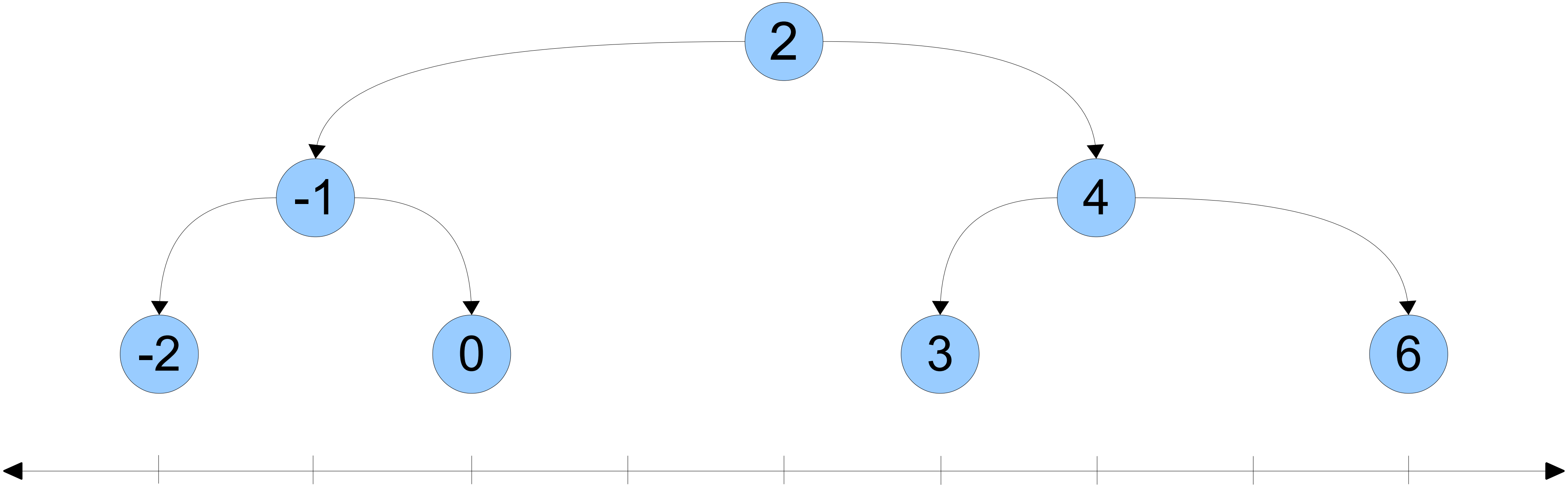
- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.
- This is done as follows:
  - **Base case**: There is nothing to delete in an empty tree.
  - **Recursive step**: Delete both subtrees, then delete the current node.

Freeing a tree
(bst.cpp)

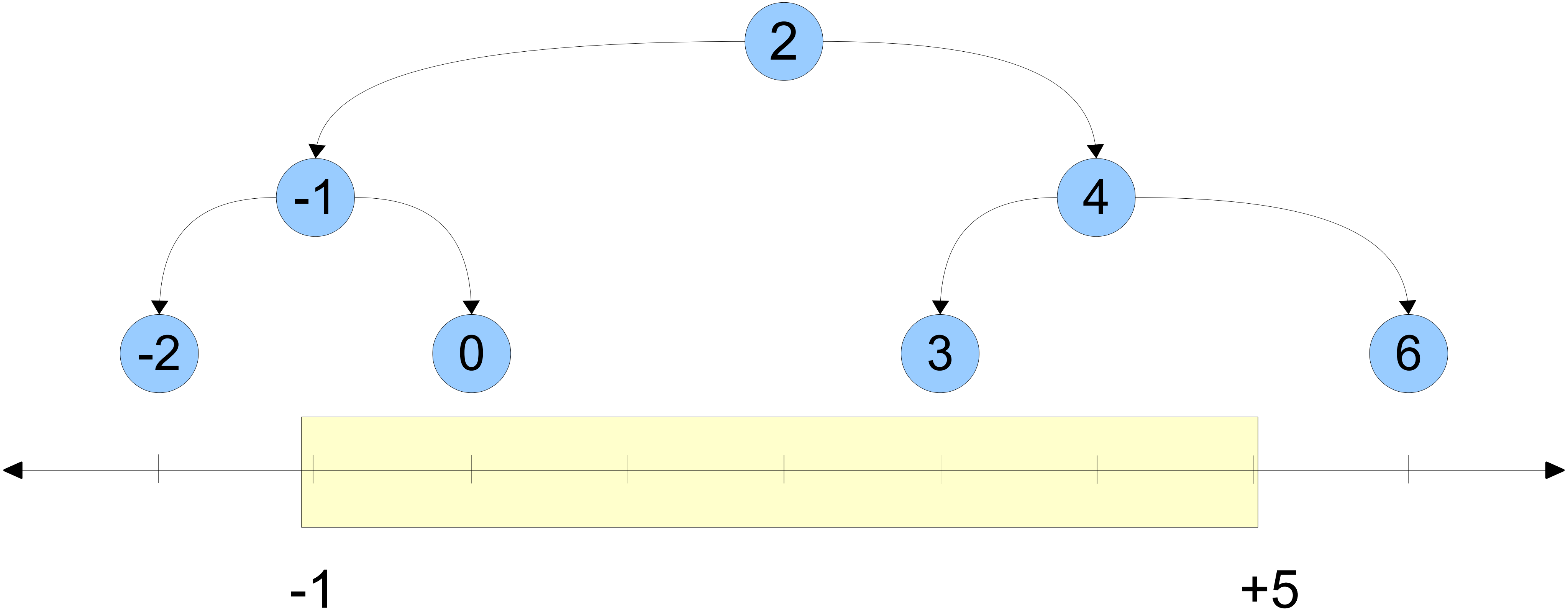
Range Queries

- We can use BSTs to do **range queries**, in which we find all values in the BST within some range.
- For example:
  - If values in a BST are dates, can find all events that occurred within some time window.
  - If values in a BST are samples of a random variable, can find everything within one and two standard deviations above the mean.

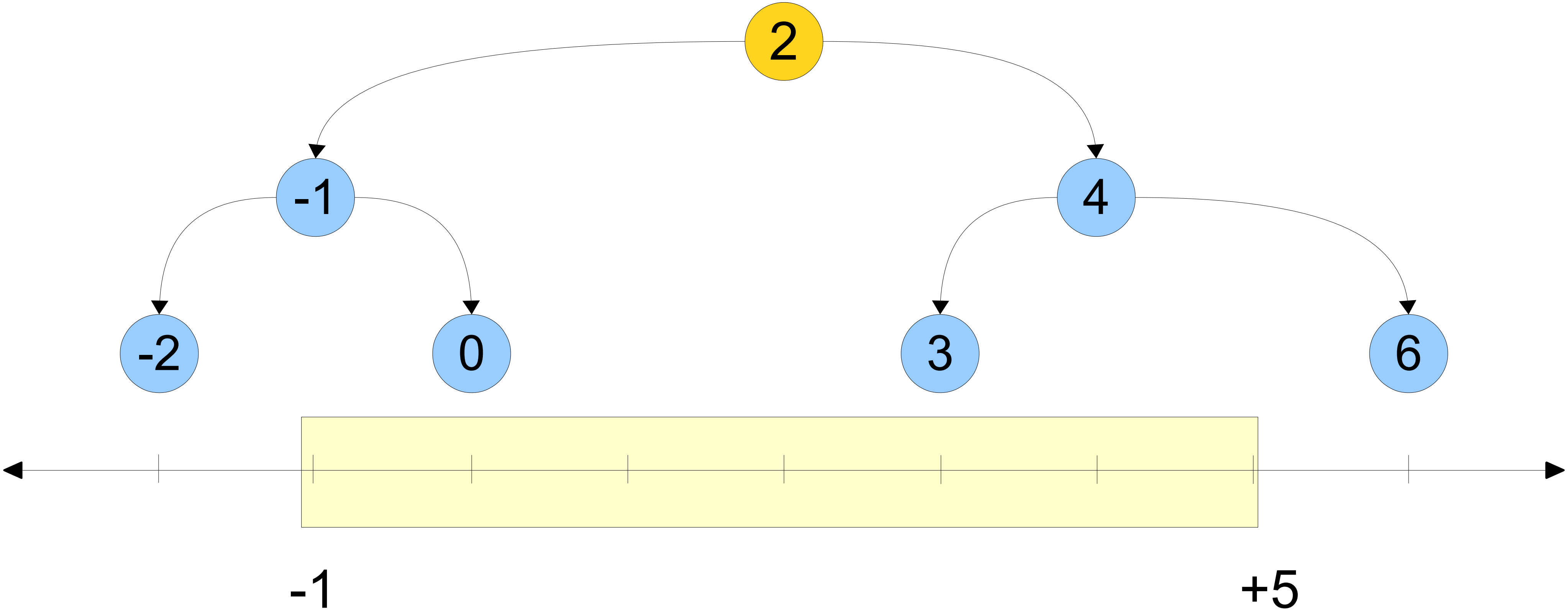
The Intuition



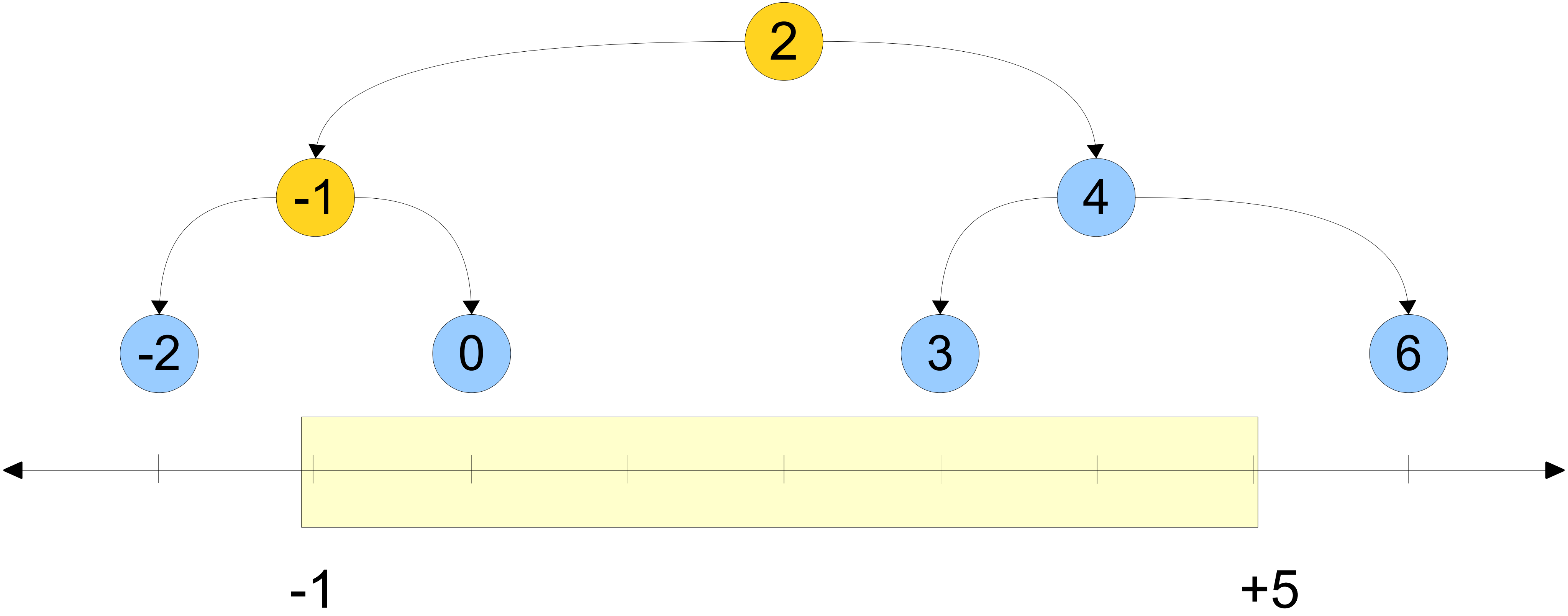
The Intuition



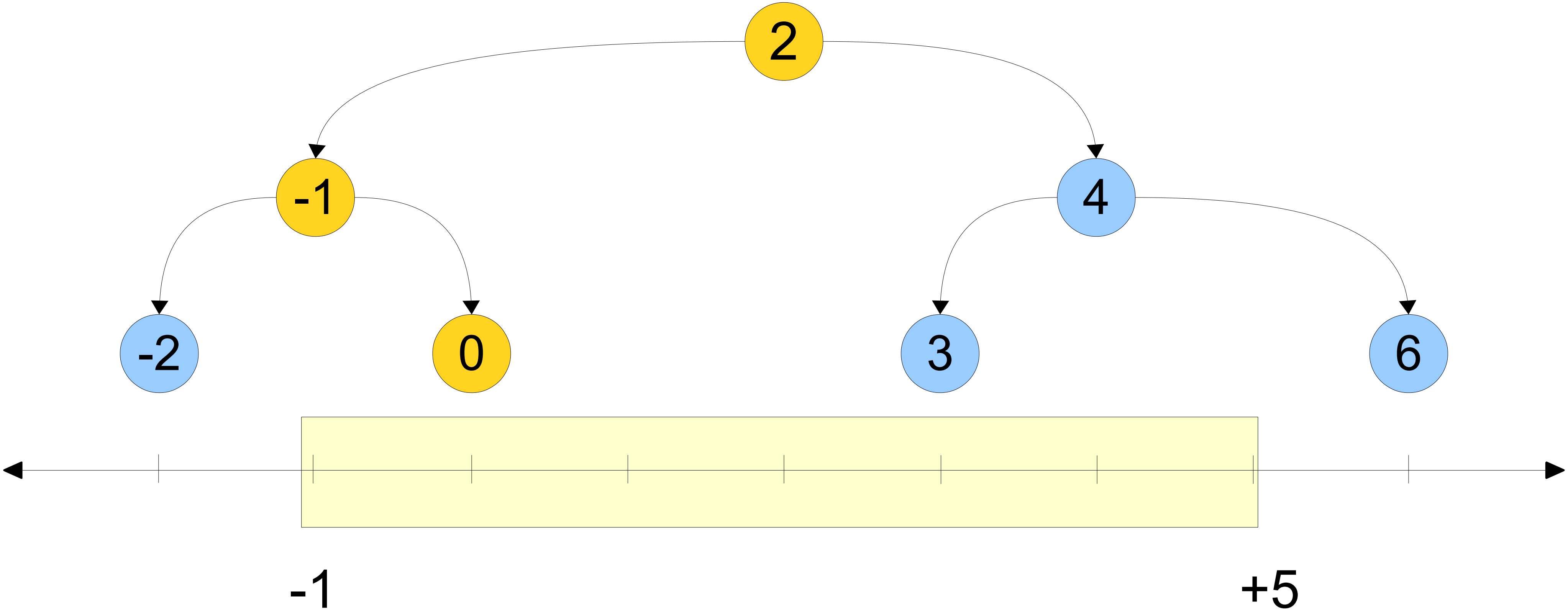
The Intuition



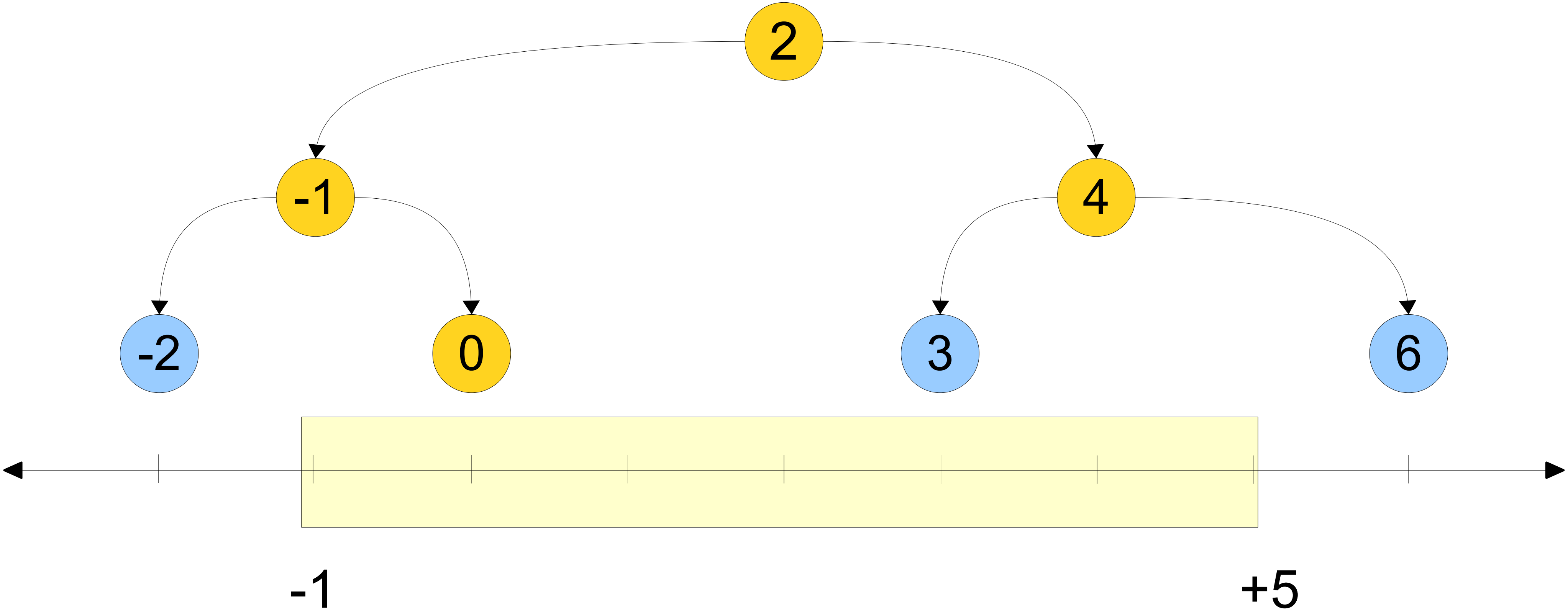
The Intuition



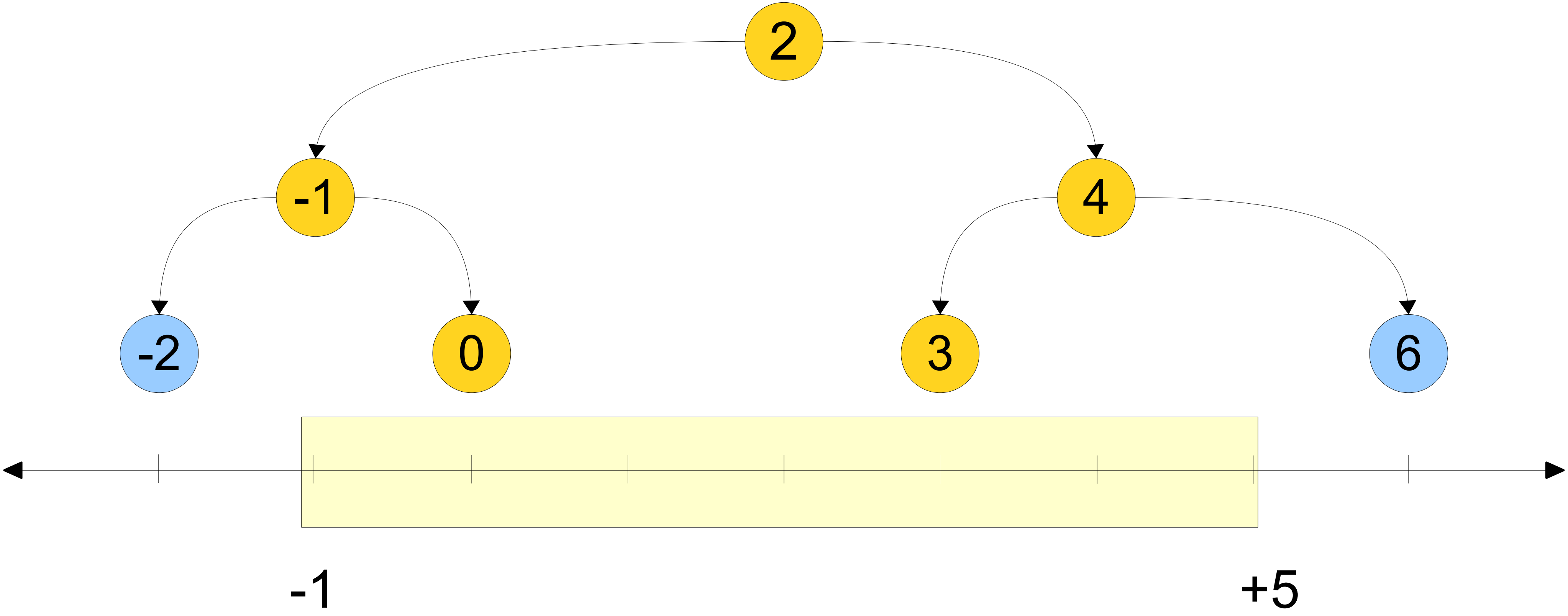
The Intuition



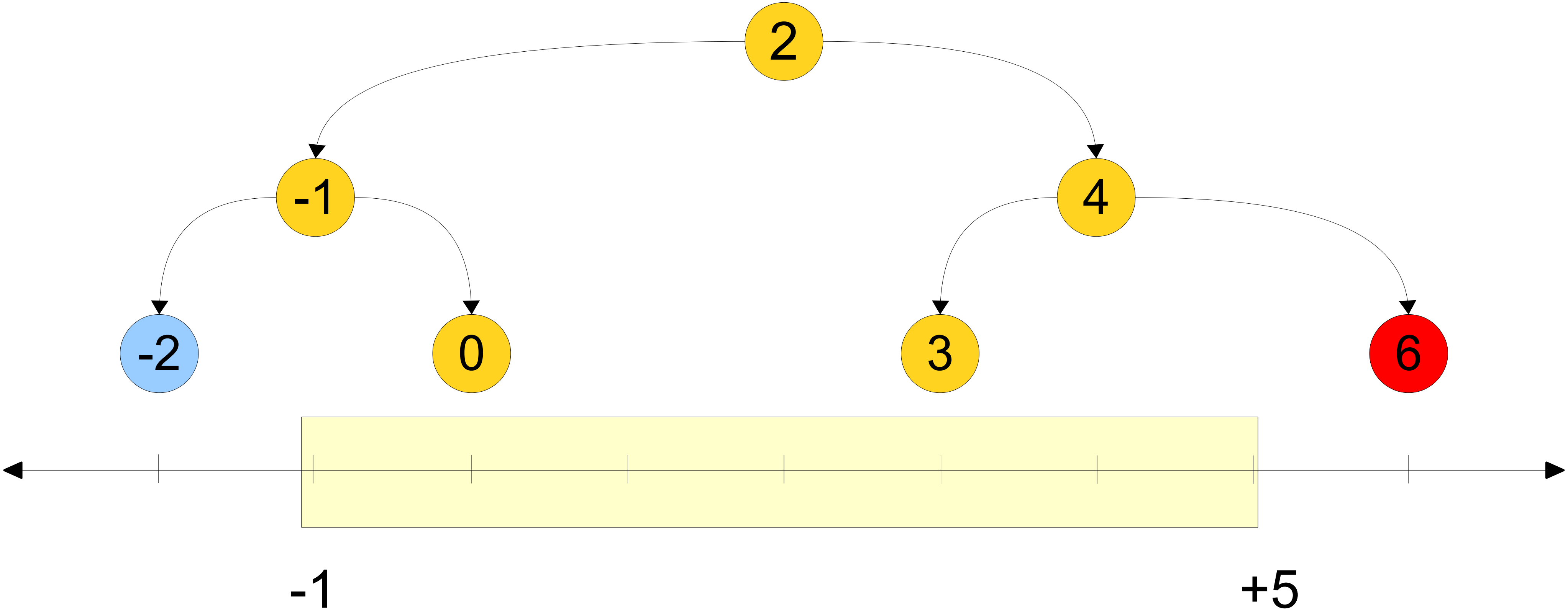
The Intuition



The Intuition



The Intuition

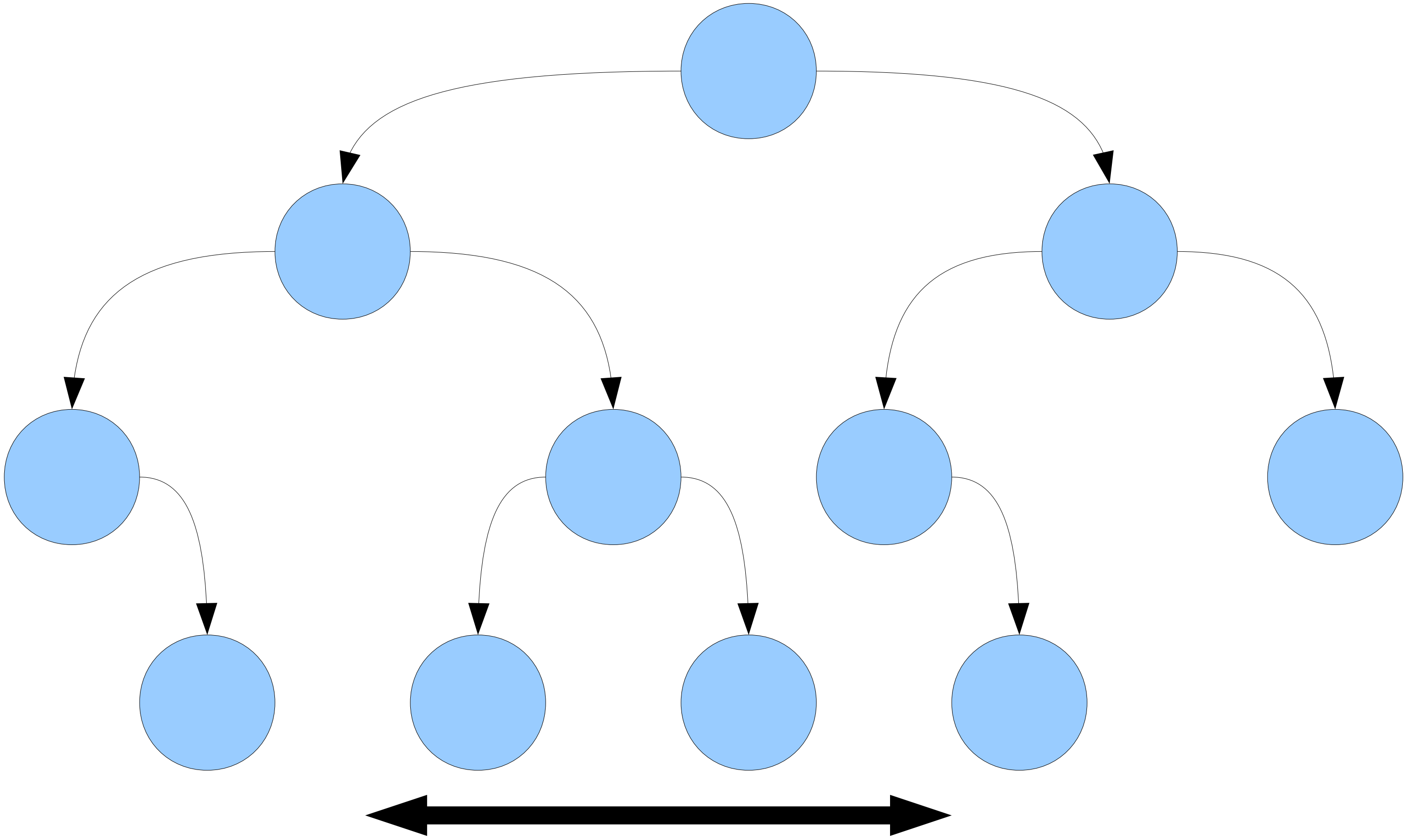


The Logic

- **Base case:**
  - The empty tree has no nodes within any range.
- **Recursive step:**
  - If this node is below the lower bound, recursively search the right subtree.
  - If this node is above the upper bound, recursively search the left subtree.
  - If this node is within bounds:
    - Search the left subtree.
    - Add this node to the output.
    - Search the right subtree.

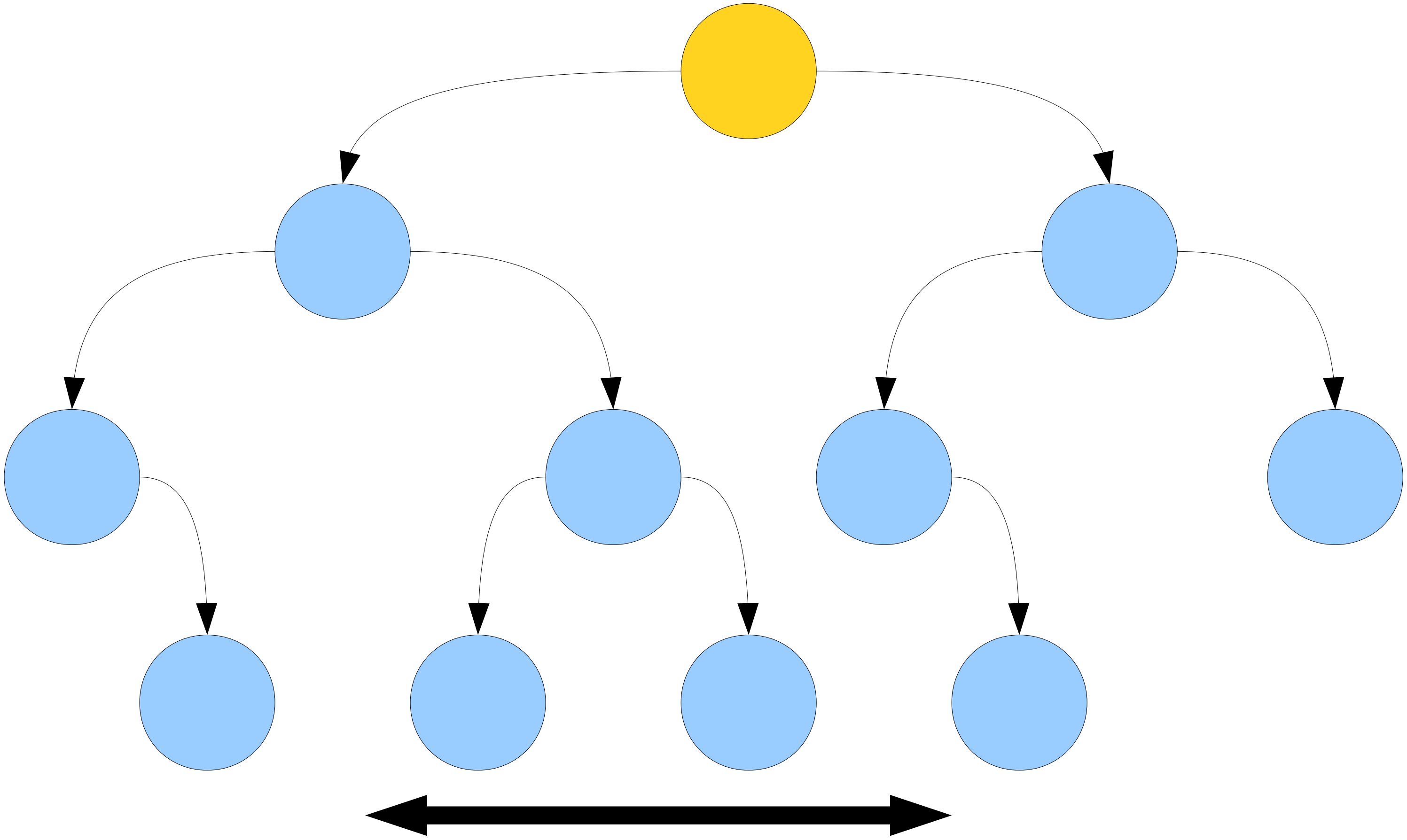
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



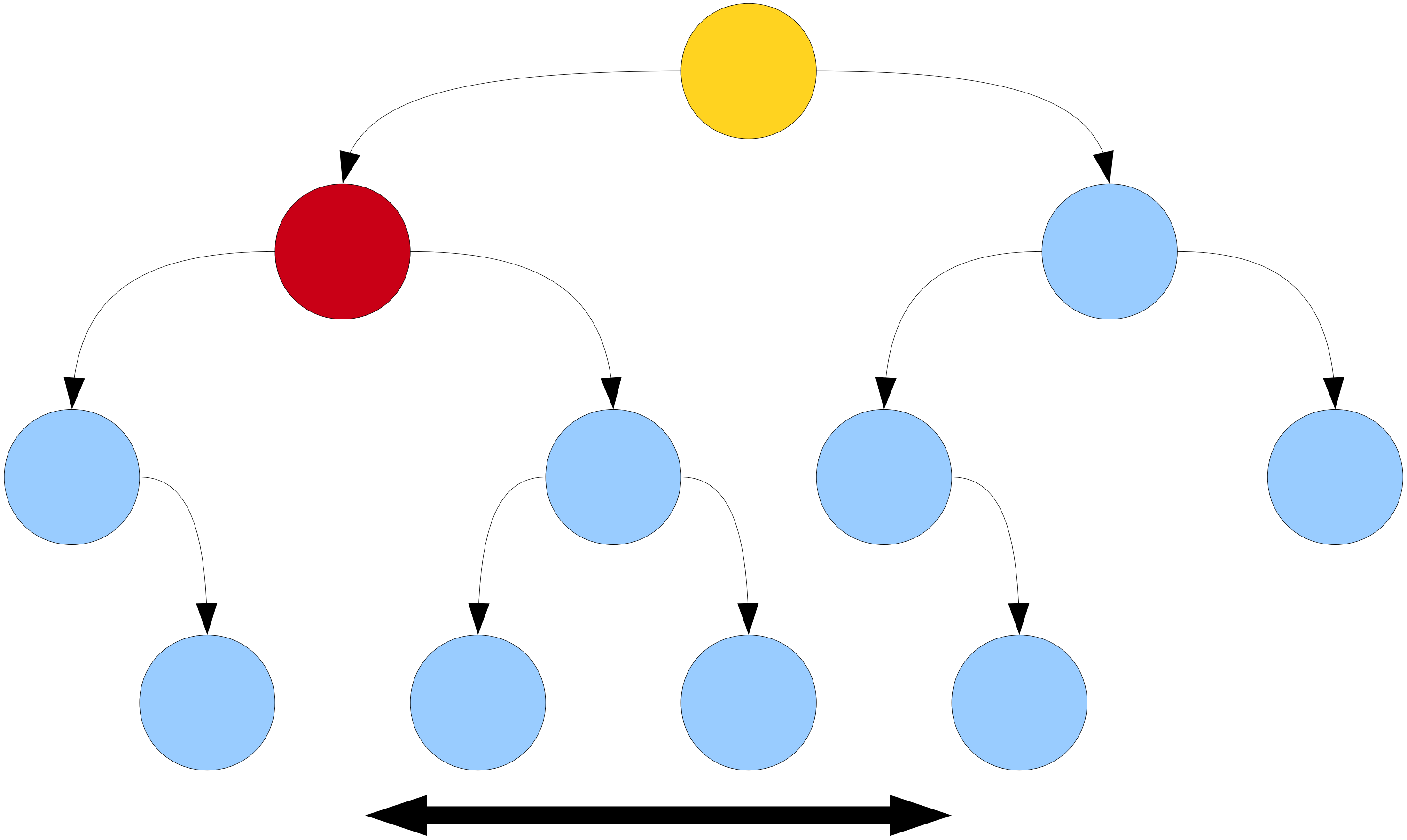
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



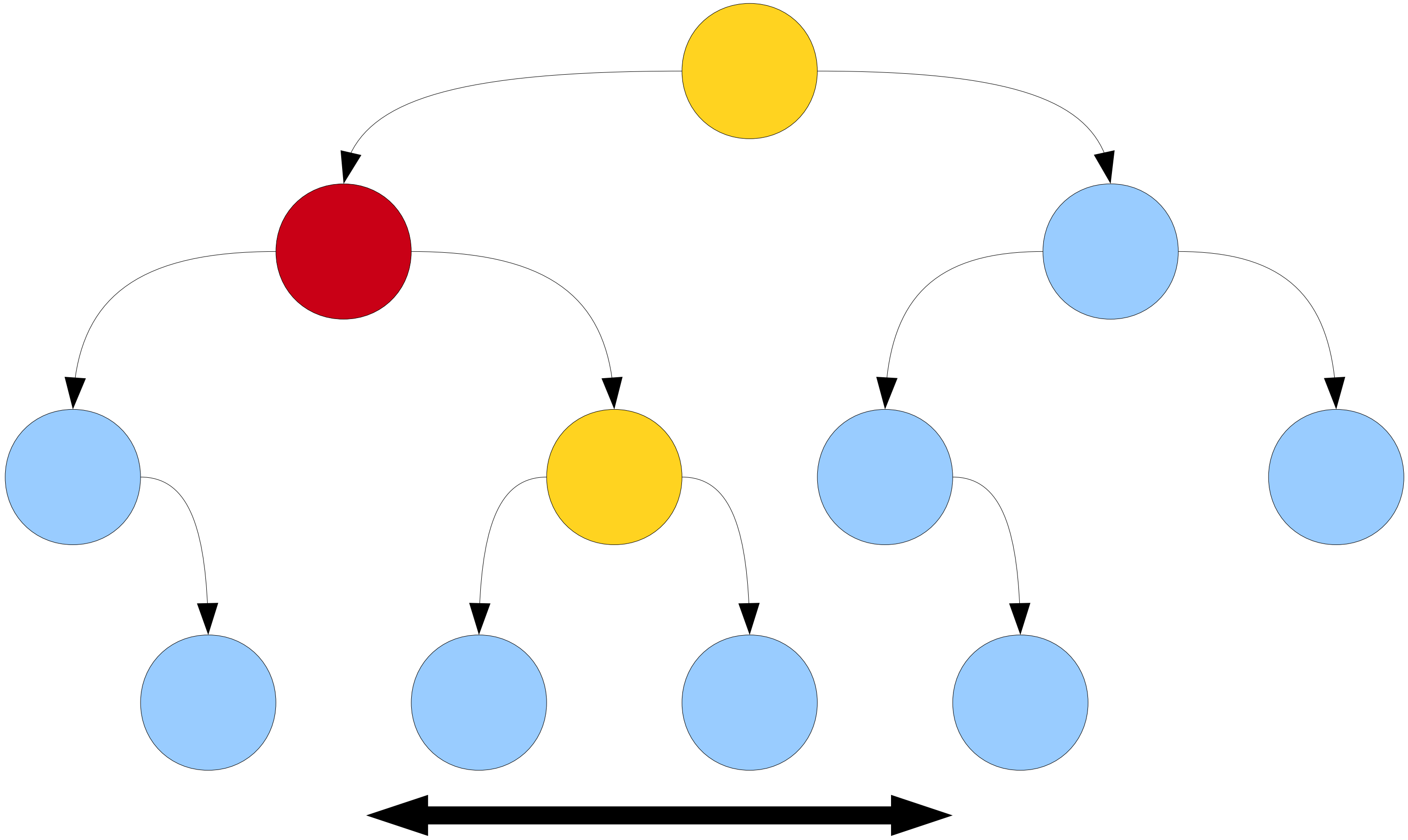
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



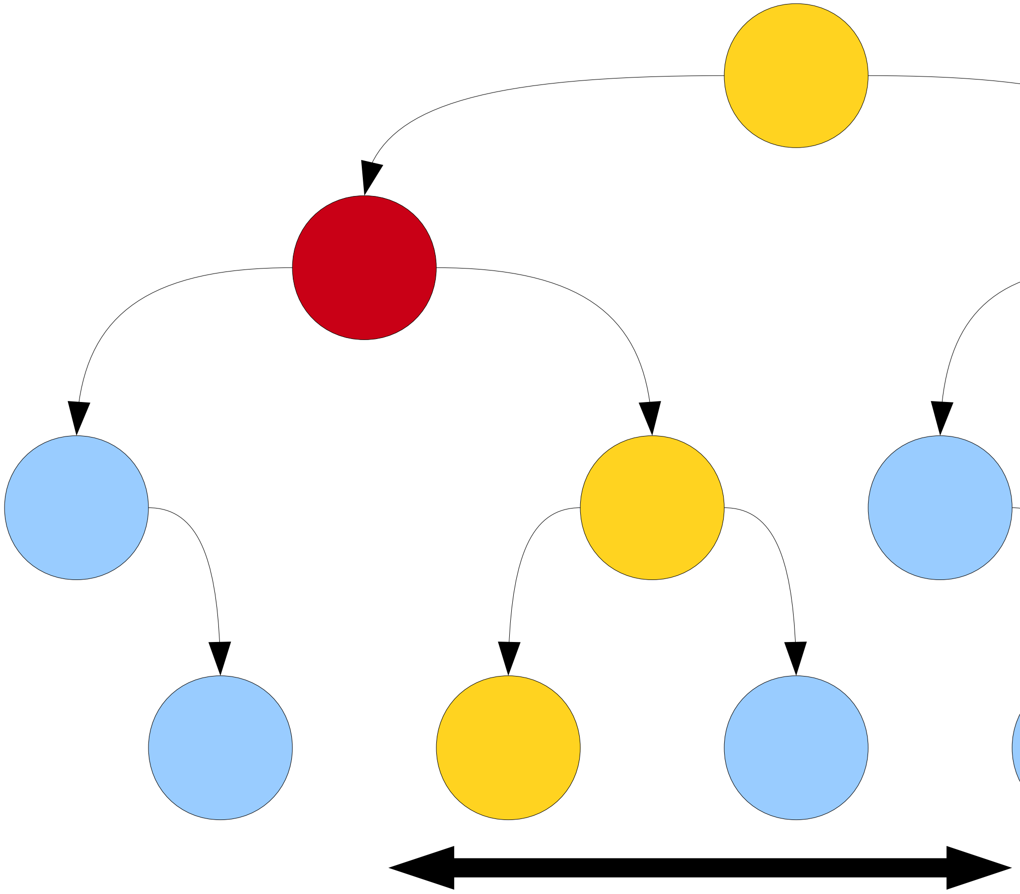
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



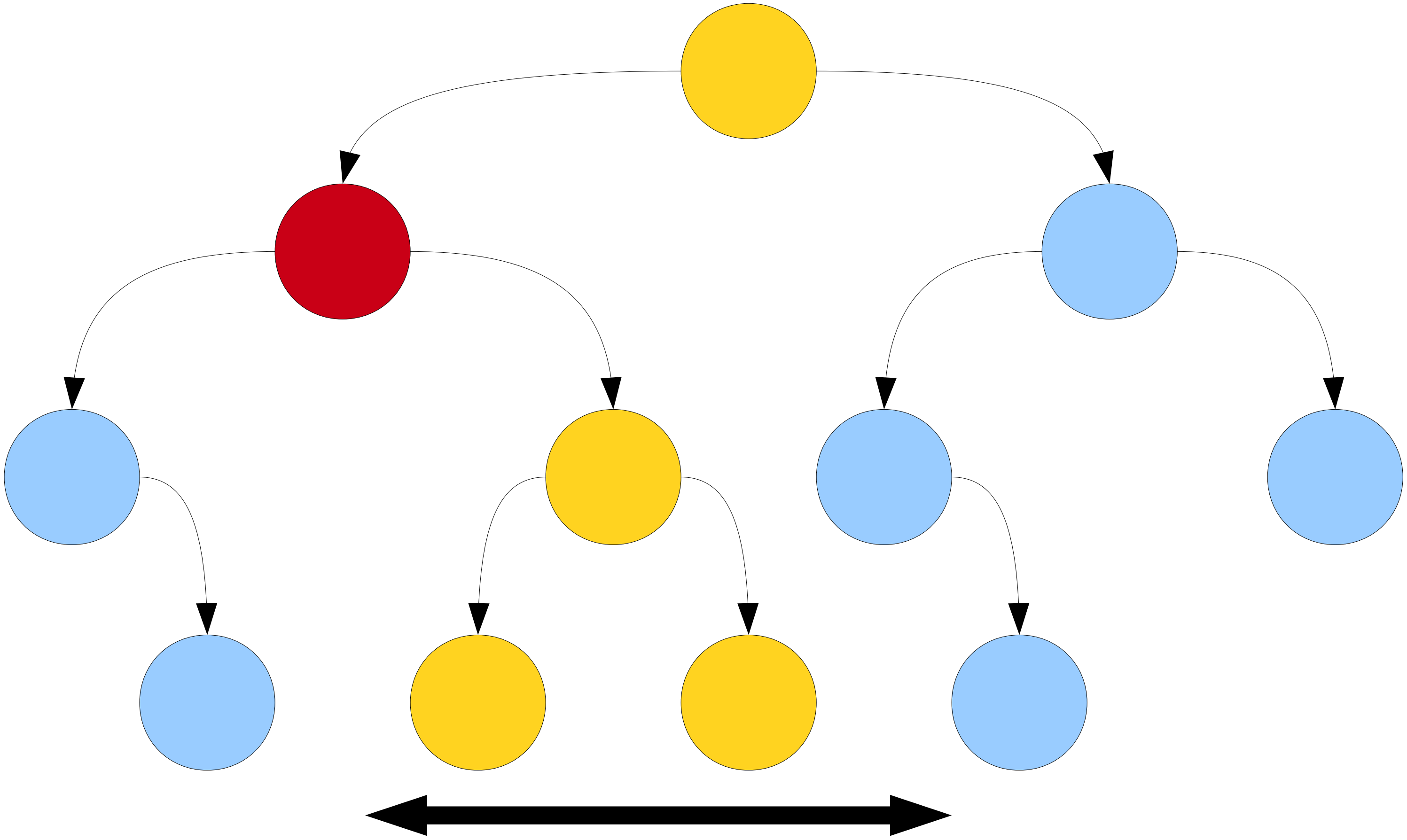
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



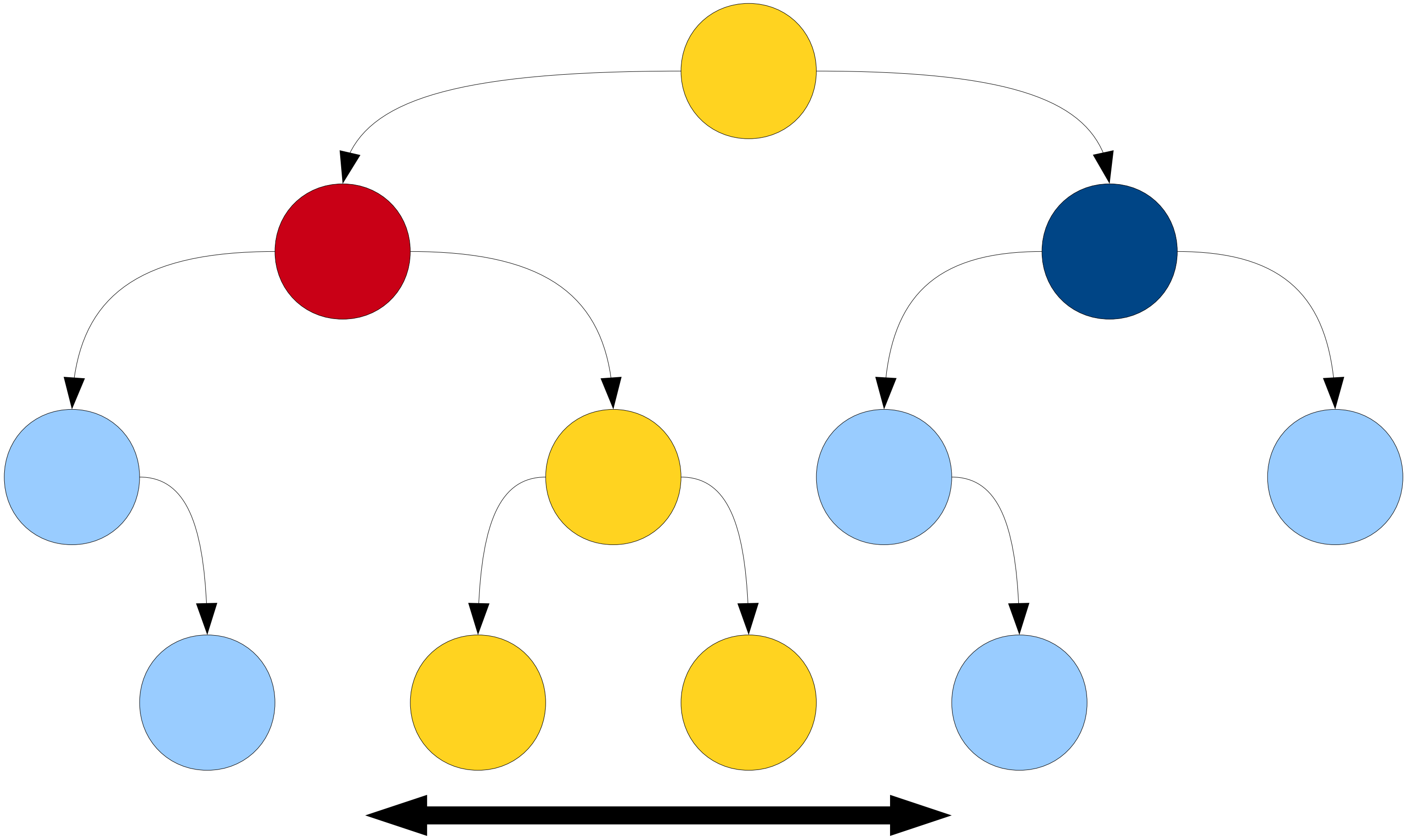
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



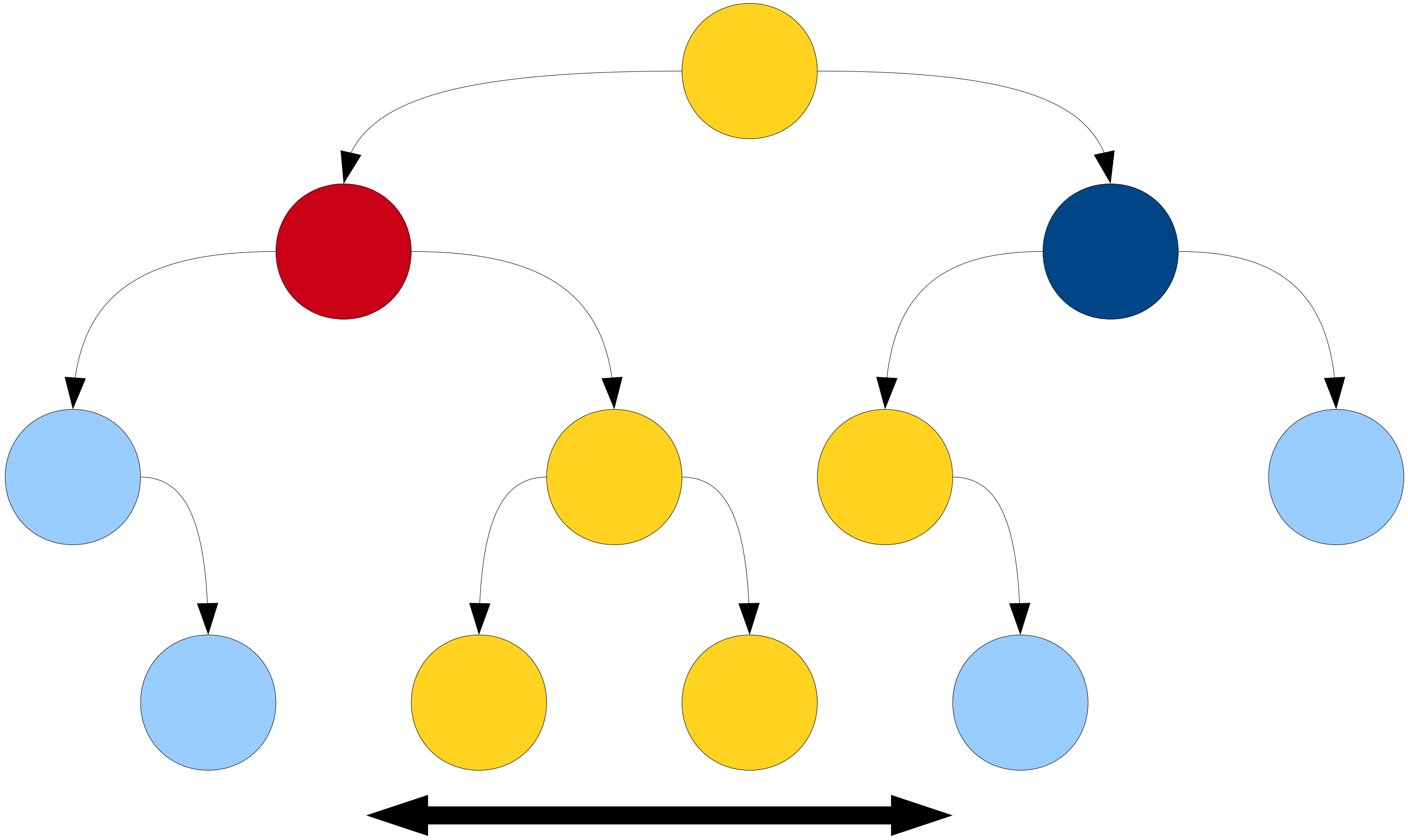
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



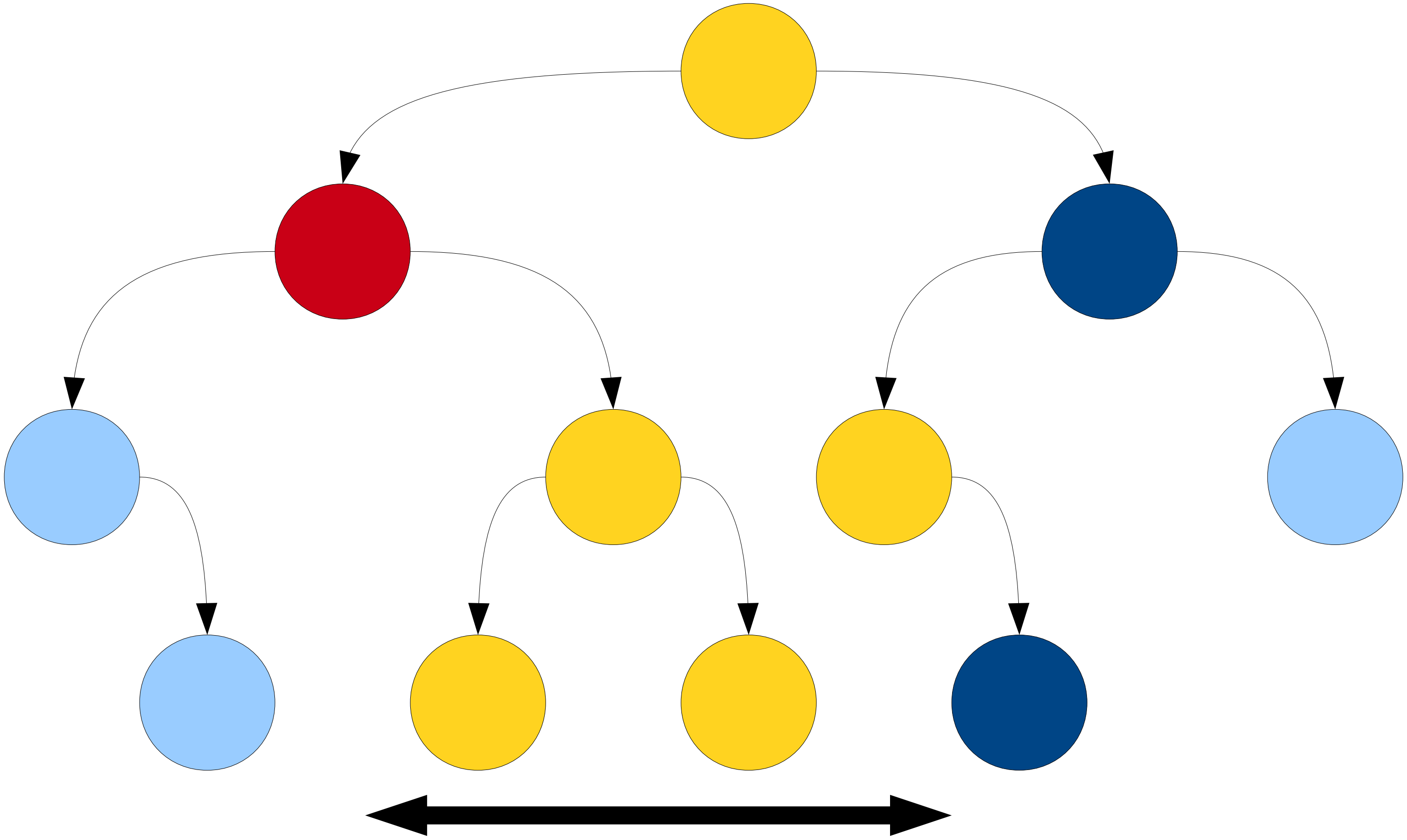
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



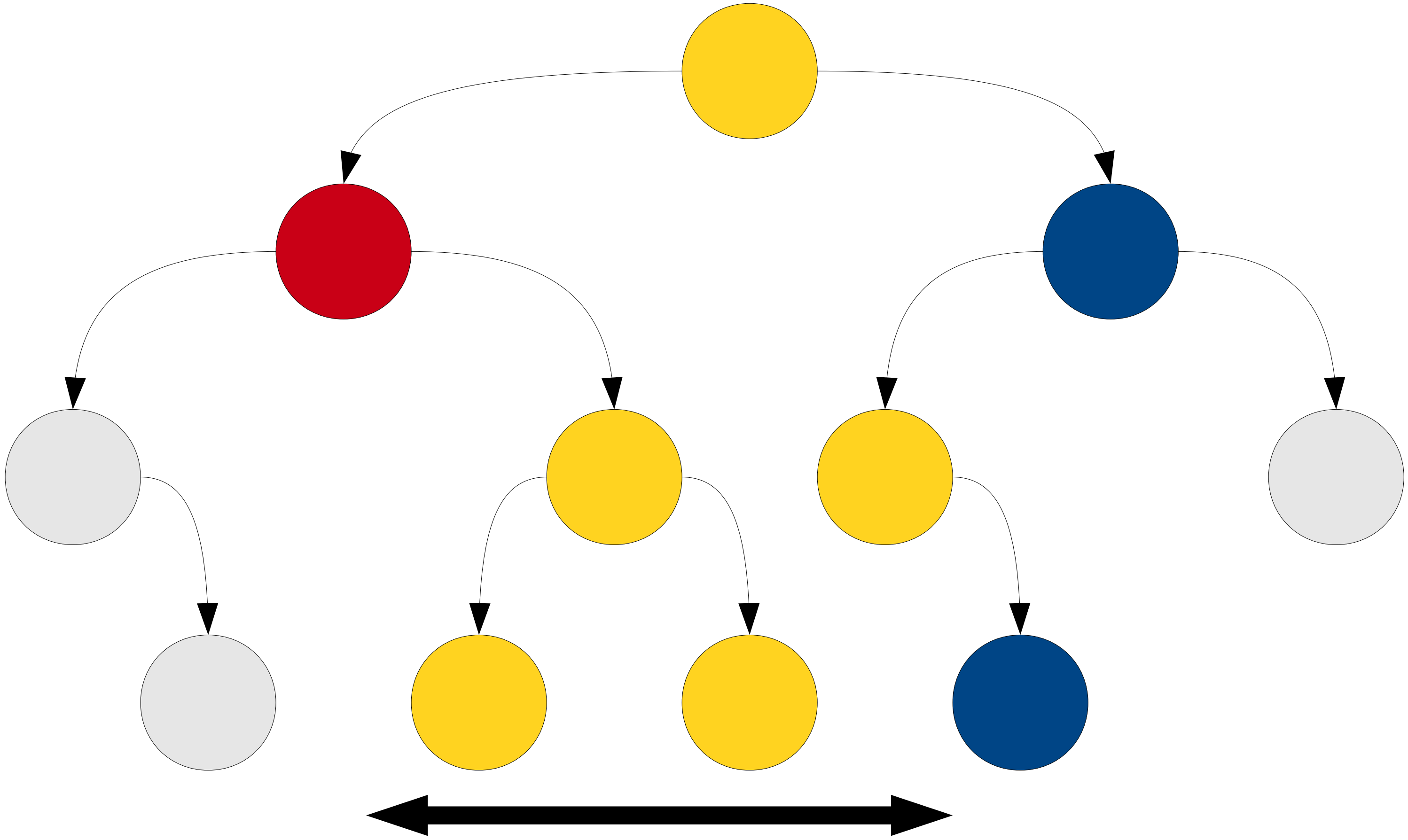
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.
- If there are *k* nodes within the range, we do at least **O(*k*)** work finding them.
- In addition, we have two "border sets" of nodes that are immediately outside that range. Each set has size **O(*h*)**, where *h* is the height of the tree.
- Total work done is **O(*k* + *h*)**.
- This is an **output-sensitive algorithm**.

Next Time

- Tries
    - How our **`Lexicon`** is implemented!