Binary Search Trees

EECS 214, Fall 2018

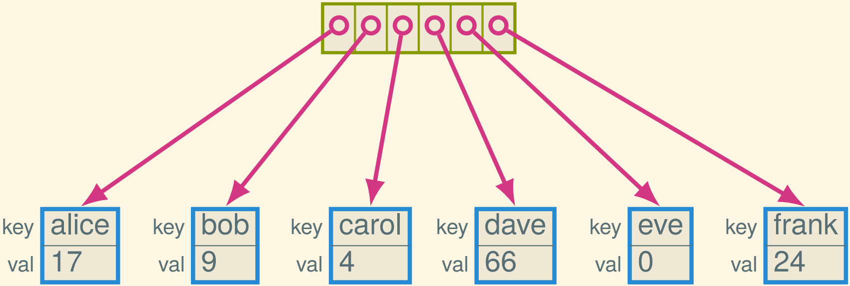
A data structure for dictionaries

There are several data structures that we can use to represent dictionaries:

- A list of keys-value pairs
- A hash table
- An array of key-value pairs
- A sorted array of key-value pairs

Let's consider the last one

A sorted array dictionary



Easy to lookup

Input: a dictionary array *array* and a key *key*
Output: a value, or nothing

```
start ← 0;
limit ← the length of array;

while start < limit do
    mid ← the average of start and limit;
    if key < array[mid].key then
        limit ← mid
    else if key > array[mid].key then
        start ← mid + 1
    else
        return array[mid].val
    end
end

return null
```

Complexity of lookup

2	7	17	19	56	75	77	90

- Initially, the element could be anywhere in the array

Complexity of lookup

2	7	17	19	56	75	77	90

- Initially, the element could be anywhere in the array
- At each iteration, $limit - start$ is cut in half

Complexity of lookup

2	7	17	19	56	75	77	90

- Initially, the element could be anywhere in the array
- At each iteration, $\mathit{limit} - \mathit{start}$ is cut in half
- This can happen at most $\log_2 n$ times

Complexity of lookup

2	7	17	19	56	75	77	90

- Initially, the element could be anywhere in the array
- At each iteration, *limit* $-$ *start* is cut in half
- This can happen at most $\log_2 n$ times
- Hence, $\mathcal{O}(\log n)$

Difficult to insert

- Inserting into an array requires shifting elements out of the way

Difficult to insert

- Inserting into an array requires shifting elements out of the way
- There may be as many as n elements to move

Difficult to insert

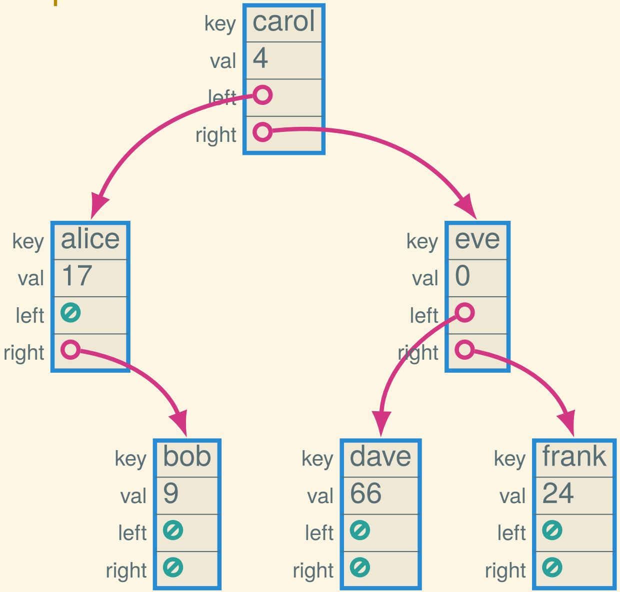
- Inserting into an array requires shifting elements out of the way
- There may be as many as n elements to move
- Hence insertion is $\mathcal{O}(n)$

Enter the BST

A *binary search tree* stores elements in order in a linked data structure

- In order means we can binary search
- Linked means we can easily insert new elements

BST example


key
carol
val
4
left
right
key
alice
val
17
left
right
key
eve
val
0
left
right
key
bob
val
9
left
right
key
dave
val
66
left
right
key
frank
val
24
left
right

BST lookup algorithm

Input: a BST root *root* and a key *key*
Output: a value, or nothing

```
curr ← root;
while curr is not null do
    if key < curr.key then
        curr ← curr.left
    else if key > curr.key then
        curr ← curr.right
    else
        return curr.val
    end
end
return null
```

Complexity of BST lookup

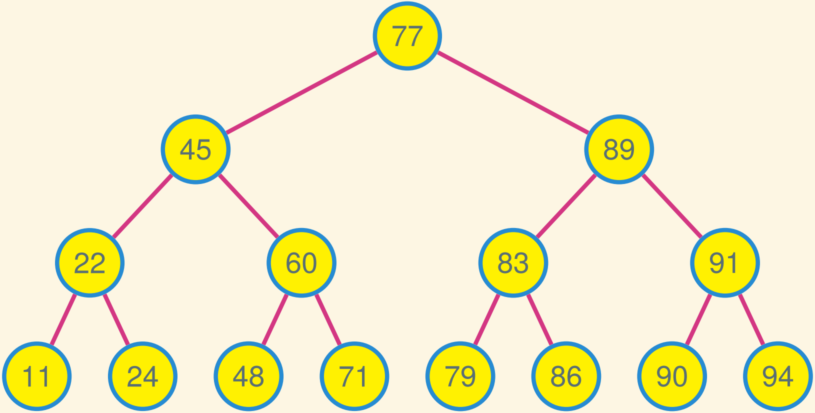
It’s binary search, right?

Binary search, again

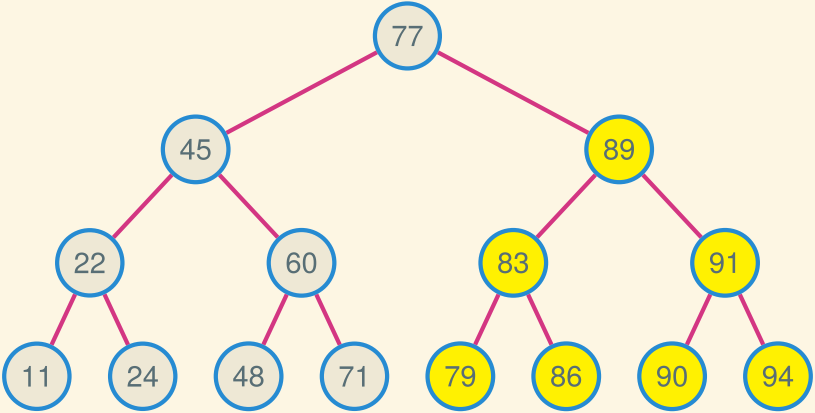
Input: a dictionary array *array* and a key *key*
Output: a value, or nothing

```
start ← 0;
limit ← the length of array;

while start < limit do
    mid ← the average of start and limit;
    if key < array[mid].key then
        limit ← mid
    else if key > array[mid].key then
        start ← mid + 1
    else
        return array[mid].val
    end
end

return null
```

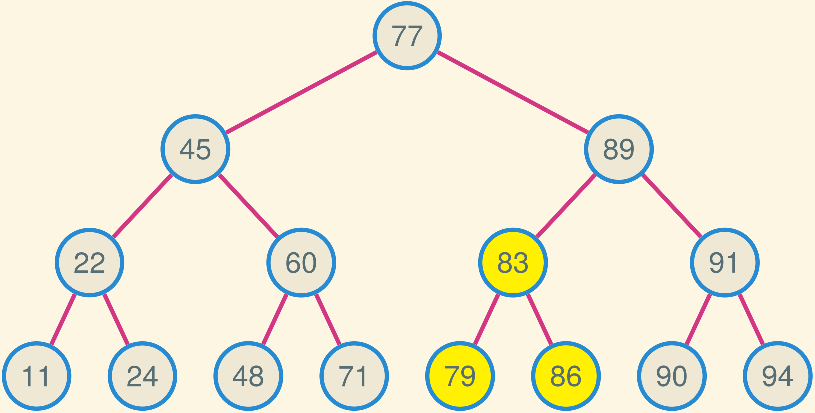
Complexity of BST lookup



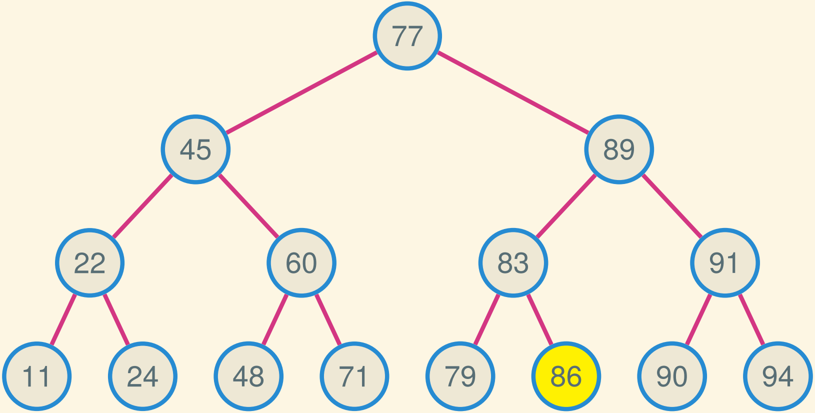
Complexity of BST lookup



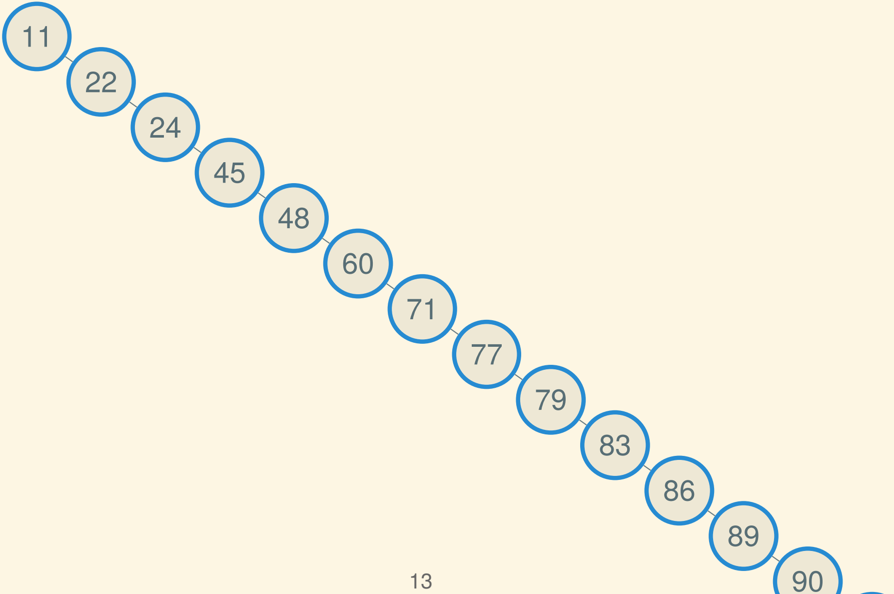
Complexity of BST lookup



Complexity of BST lookup



Complexity of BST lookup, take 2



BST insert algorithm (recursive)

```
Function BstInsert(node, key, value) is
    Output: the updated BST

    if node is null then
        return a new node with key, value, and null for both
          children
    else if key < node.key then
        node.left ← BstInsert(node.left, key, value);
        return node
    else if key > node.key then
        node.right ← BstInsert(node.right, key, value);
        return node
    else
        node.val = value;
        return node
    end
end
```

BST insert algorithm (with pointers)

Input: a BST root *root*, a key *key*, and a value *value*
Output: the updated BST

```
curr ← the address of root;
while the value addressed by curr is not null do
    if key < curr.key then
        curr ← the address of curr.left
    else if key > curr.key then
        curr ← the address of curr.right
    else
        curr.val ← value;
        return
    end
end
newNode ←
  a new node with key, value, and null for both children;
the value addressed by curr ← newNode
```

Complexity of BST insert

- First do a search — $\mathcal{O}(\log n)$

Complexity of BST insert

- First do a search — $\mathcal{O}(\log n)$
- If we find the key, replace the value — $\mathcal{O}(1)$

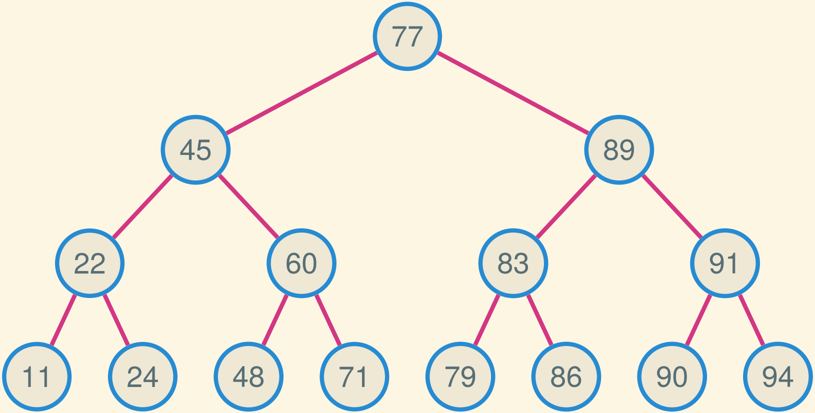
Complexity of BST insert

- First do a search — $\mathcal{O}(\log n)$
- If we find the key, replace the value — $\mathcal{O}(1)$
- If not, add a new leaf where we hit bottom — $\mathcal{O}(1)$

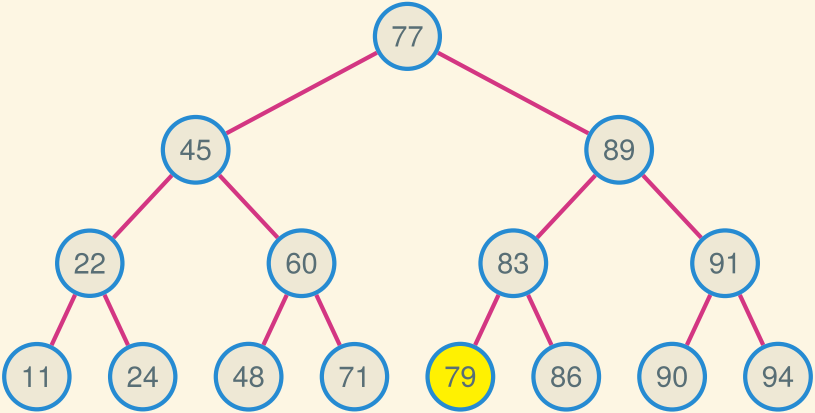
Complexity of BST insert

- First do a search — $\mathcal{O}(\log n)$
- If we find the key, replace the value — $\mathcal{O}(1)$
- If not, add a new leaf where we hit bottom — $\mathcal{O}(1)$
- Hence, $\mathcal{O}(\log n)$

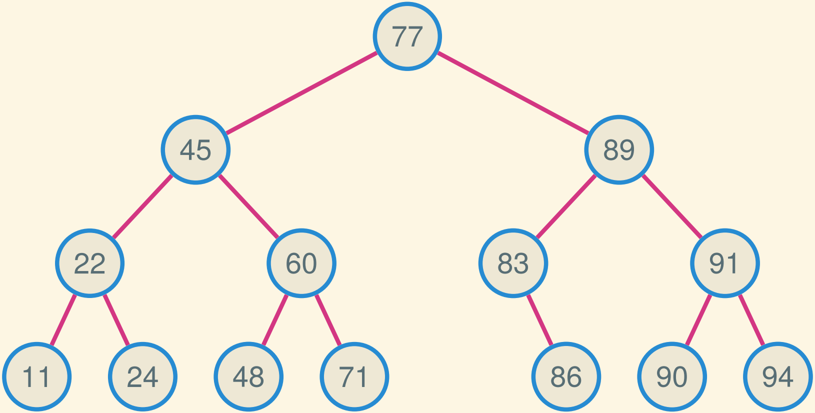
BST delete



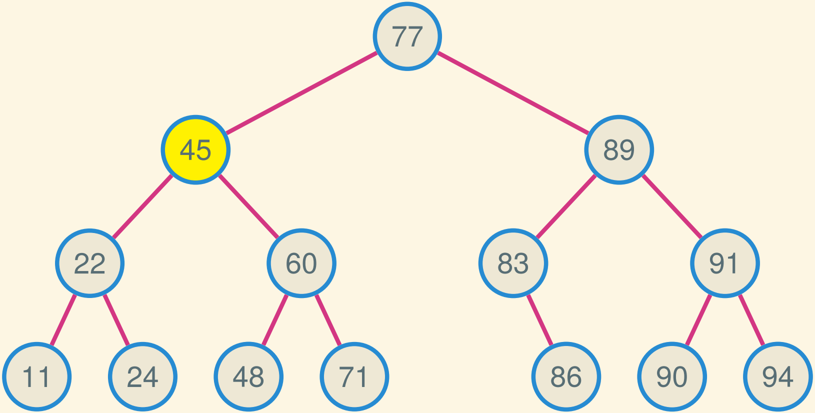
BST delete



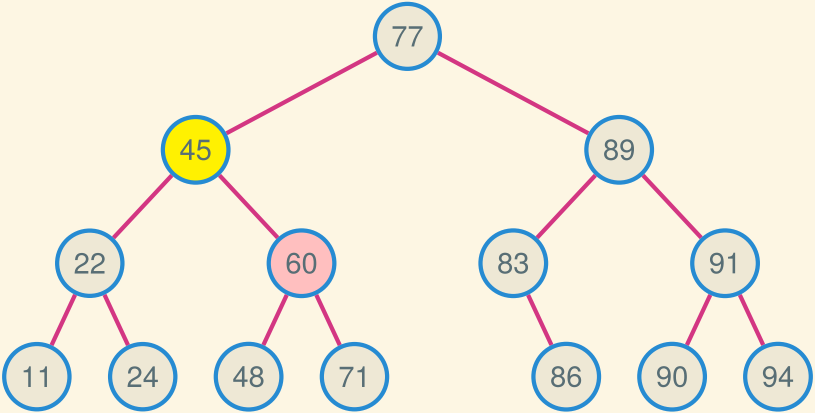
BST delete



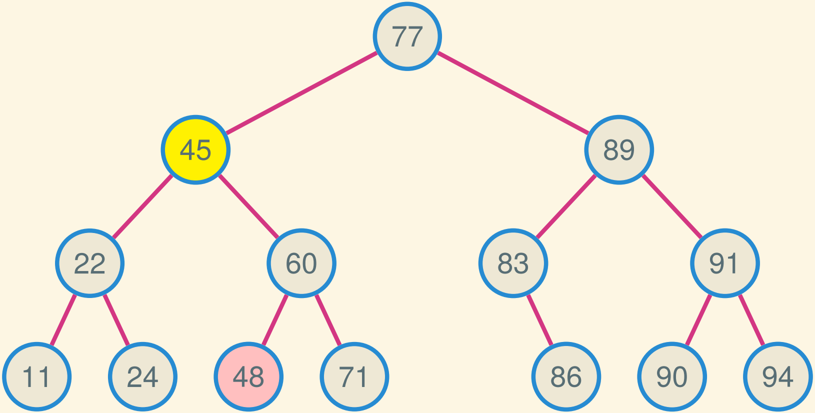
BST delete



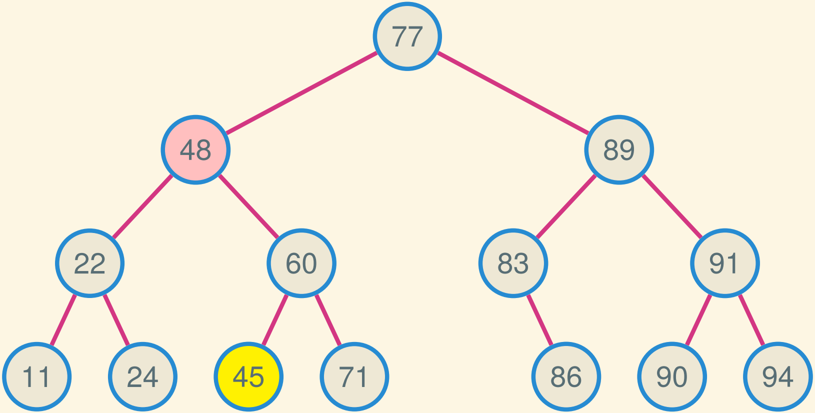
BST delete



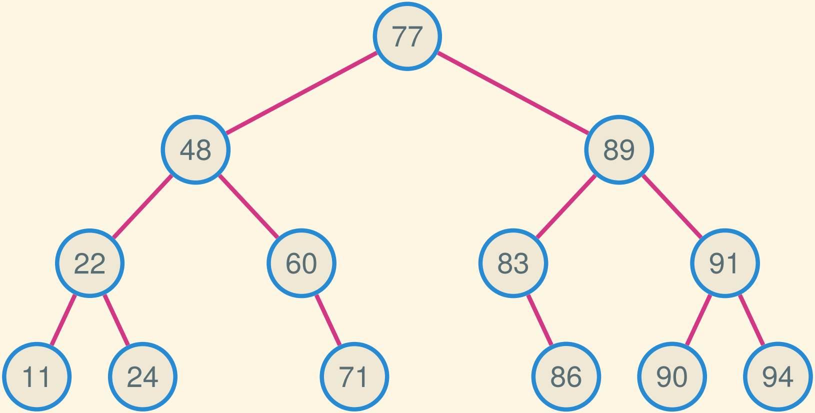
BST delete



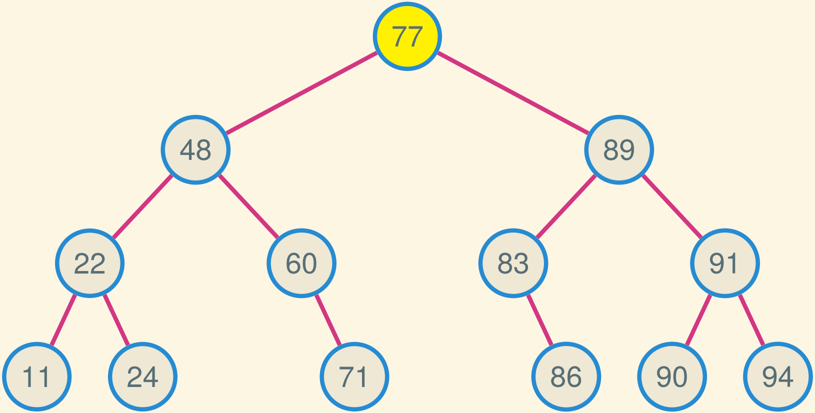
BST delete



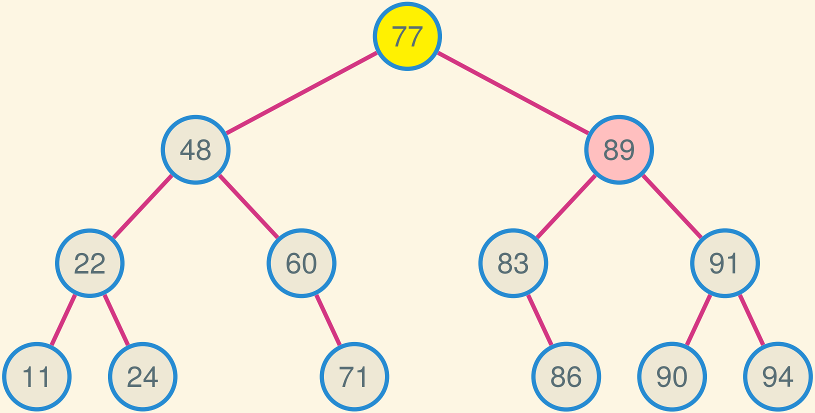
BST delete



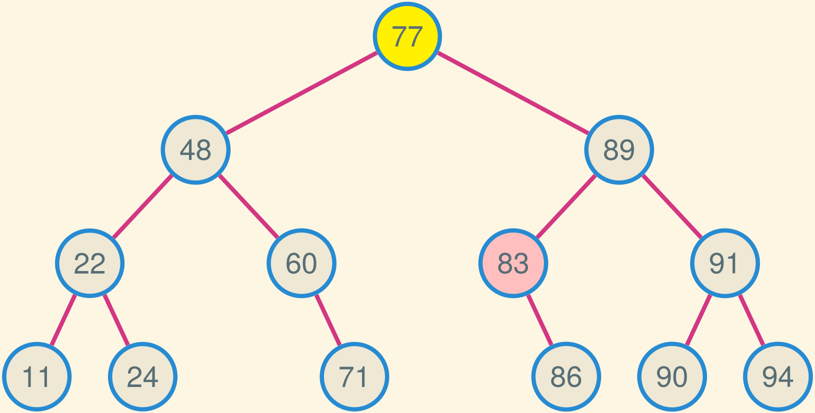
BST delete



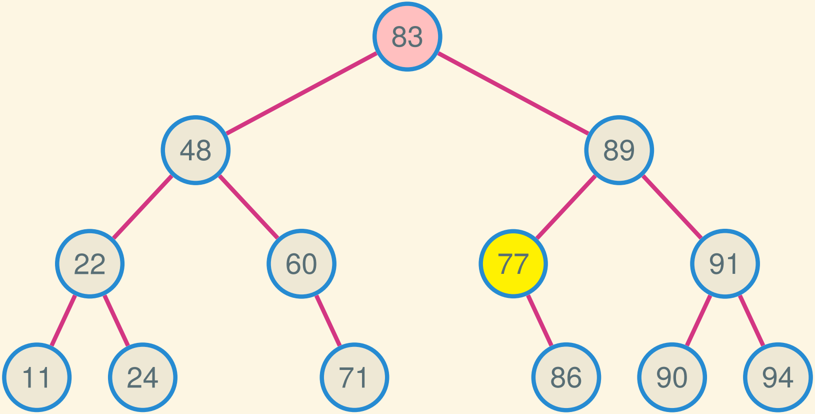
BST delete



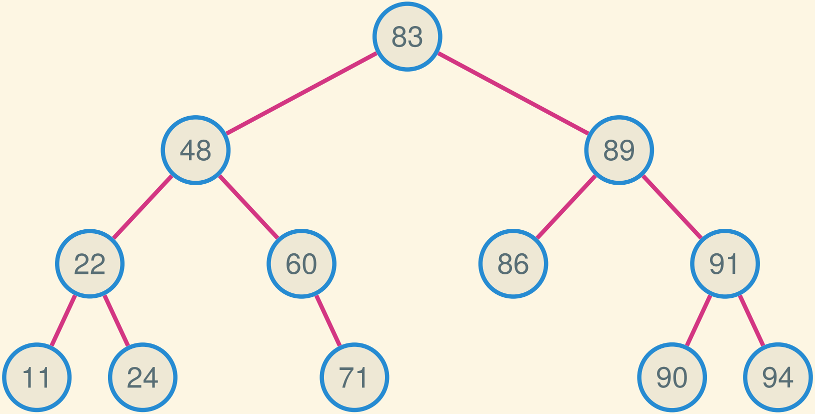
BST delete



BST delete



BST delete



Next time: hashing and hash tables