Algorithms: Elementary Data Structures and Binary Search Trees

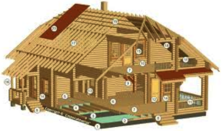
Ola Svensson

EPFL School of Computer and Communication Sciences

Lecture 8, 12.03.2025

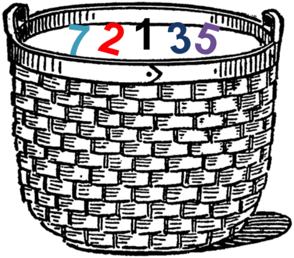
Elementary Data Structures

Algorithm



Algorithm

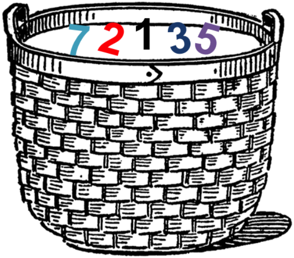
Data structures = dynamic sets of items



Data structure containing numbers

Data structures = dynamic sets of items

What kind of operations do we want to do?

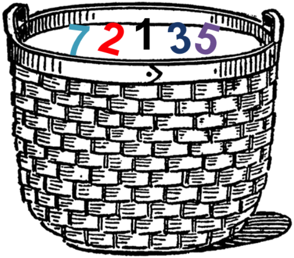


Data structure containing numbers

Data structures = dynamic sets of items

What kind of operations do we want to do?

- Modifying operations: insertion, deletion, ...
- Query operations: search, maximum, minimum, ...

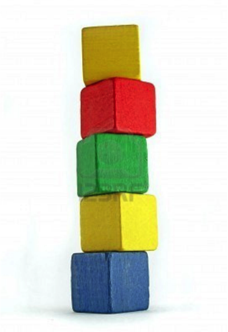


Data structure containing numbers

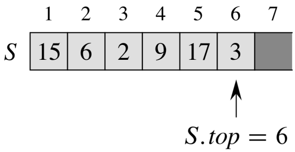
Stacks (last-in, first-out)

- Insert operation called $\textsc{push}(S,x)$

- Delete operation called $\textsc{pop}(S)$



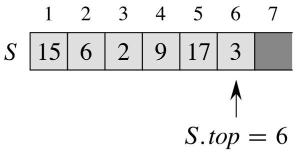
Stacks Implementation



Implementation using arrays: S consists of elements $S[1, \dots, S.top]$

- $S[1]$ element at the bottom
- $S[S.top]$ element at the top

Stacks Implementation



What is the running time of these operations? $O(1)$

STACK-EMPTY(S)

1. **if** $S.top = 0$
2. **return** TRUE
3. **else return** FALSE

PUSH(S,x)

1. $S.top \leftarrow S.top + 1$
2. $S[S.top] \leftarrow x$

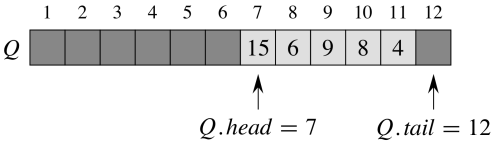
POP(S)

1. **if** STACK-EMPTY(S)
2. **error** "underflow"
3. **else**
4. $S.top \leftarrow S.top - 1$
5. **return** $S[S.top + 1]$

Queues (first-in, first-out)

- Insert operation called $\textsc{Enqueue}(Q,x)$
- Delete operation called $\textsc{Dequeue}(Q)$

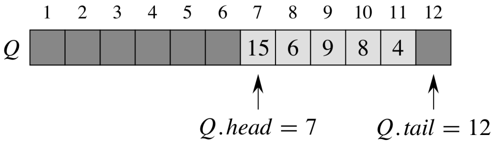
Queue Implementation



Implementation using arrays: Q consists of elements $S[Q.head, \ldots, Q.tail - 1]$

- $Q.head$ points at the first element
- $Q.tail$ points at the next location where a newly arrived element will be placed

Queue Implementation



What is the running time of these operations? $O(1)$

```
ENQUEUE(Q,x)
Q[Q.tail] = x
if Q.tail = Q.length
    Q.tail ← 1
else Q.tail ← Q.tail + 1
```

```
DEQUEUE(Q)
x = Q[Q.head]
if Q.head = Q.length
    Q.head ← 1
else Q.head ← Q.head + 1
return x
```

Stacks and Queues

Positives

- Very efficient
- Natural operations

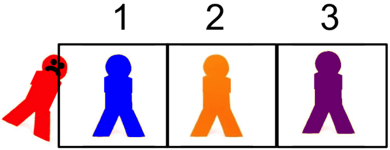
Negatives

- Limited support: for example, no search
- Implementations using arrays have a *fixed* capacity

Linked List

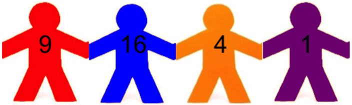
Objects are arranged in a linear order

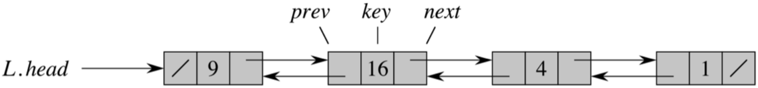
Not indexes in array



But pointers in each object

Linked List

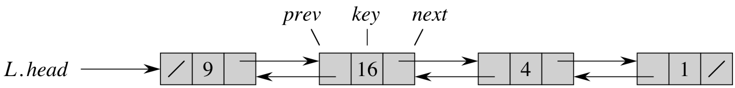




A list can be

- Single linked or double linked
- Sorted or unsorted
- etc.

Searching a Linked List



Task: Given k return pointer to first element with key k

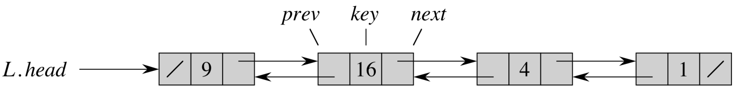
List-Search(L,k)

1. $x \leftarrow L.head$
2. **while** $x \neq nil$ and $x.key \neq k$
3. $\quad x \leftarrow x.next$
4. **return** x

Running time? $O(n)$

What if no element with key k exists? returns nil

Inserting into a Linked List



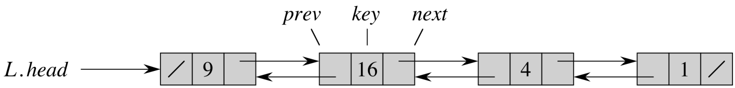
Task: Insert a new element x

LIST-INSERT(L,x)

1. $x.next \leftarrow L.head$
2. **if** $L.head \neq nil$
3. $\quad L.head.prev \leftarrow x$
4. $L.head \leftarrow x$
5. $x.prev = NIL$

Running time?

Inserting into a Linked List



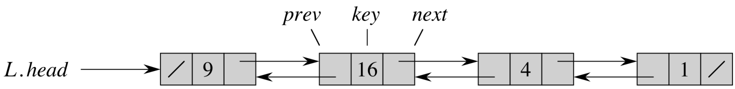
Task: Insert a new element x

LIST-INSERT(L,X)

1. $x.next \leftarrow L.head$
2. **if** $L.head \neq nil$
3. $\quad L.head.prev \leftarrow x$
4. $L.head \leftarrow x$
5. $x.prev = NIL$

Running time? $O(1)$

Deleting From a Linked List



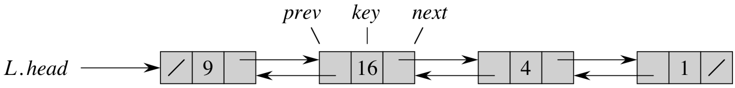
Task: Given a pointer to an element x remove it from L

LIST-DELETE(L,x)

1. **if** $x.prev \neq nil$
2. $\quad x.prev.next \leftarrow x.next$
3. **else** $L.head \leftarrow x.next$
4. **if** $x.next \neq nil$
5. $\quad x.next.prev \leftarrow x.prev$

Running time? $O(1)$

Sentinels



Note: If x is in the middle of the list then

LIST-DELETE(L,x)

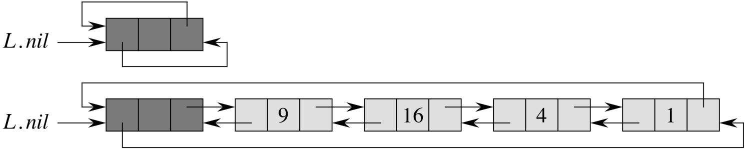
1. **if** $x.prev \neq nil$
2. $\quad x.prev.next \leftarrow x.next$
3. **else** $L.head \leftarrow x.next$
4. **if** $x.next \neq nil$
5. $\quad x.next.prev \leftarrow x.prev$

simplified

LIST-DELETE'(L,x)

1. $x.prev.next \leftarrow x.next$
2. $x.next.prev \leftarrow x.prev$

Sentinels



simplified

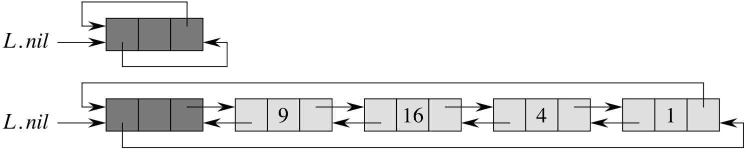
List-Delete(L,x)

if $x.prev \neq nil$
    $x.prev.next \leftarrow x.next$
else $L.head \leftarrow x.next$
if $x.next \neq nil$
    $x.next.prev \leftarrow x.prev$

List-Delete'(L,x)

$x.prev.next \leftarrow x.next$
$x.next.prev \leftarrow x.prev$

Sentinels



simplified

List-Insert(L,x)

```
x.next ← L.head
if L.head ≠ nil
    L.head.prev ← x
L.head ← x
x.prev = NIL
```

List-Insert'(L,x)

```
x.next ← L.nil.next
L.nil.next.prev ← x
L.nil.next ← x
x.prev ← L.nil
```

Summary Linked List

- Dynamic data structure without predefined capacity
- Insertion: $O(1)$
- Deletion: $O(1)$ (if double linked)
  - Question in book: can you do it for single linked?
- Search: $O(n)$

Summary Linked List

- Dynamic data structure without predefined capacity
- Insertion: $O(1)$
- Deletion: $O(1)$ (if double linked)
  - Question in book: can you do it for single linked?
- Search: $O(n)$

Search $O(n) =$ no fun!

Search $O(n)$ = no fun!

Search $O(n)$ = no fun!

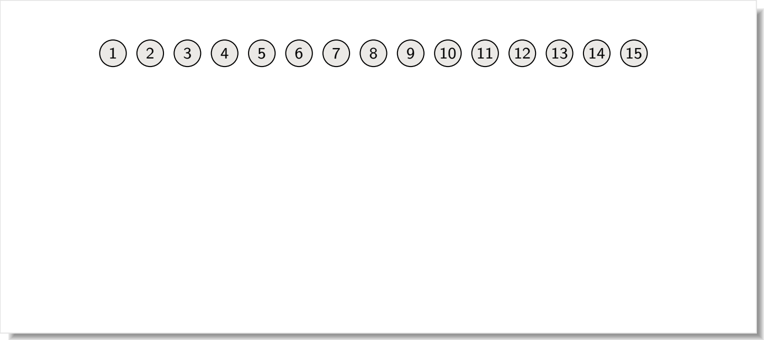
We will have fun: **Binary Search Trees**

BINARY SEARCH TREES

Idea

Guessing Game:

- Ola thinks of an integer between 1 and 15
- When you guess a number, answer either *correct*, *smaller*, or *larger*
  - For example: is it 5? Ola: *larger*
- What is your best strategy to minimize number of guesses?



Idea

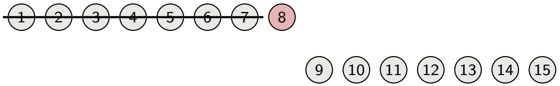
Guessing Game:

- Ola thinks of an integer between 1 and 15
- When you guess a number, answer either *correct*, *smaller*, or *larger*
  - For example: is it 5? Ola: *larger*
- What is your best strategy to minimize number of guesses?

Idea

Guessing Game:

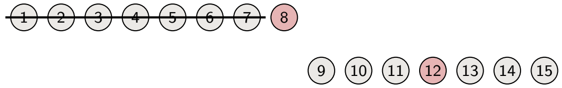
- Ola thinks of an integer between 1 and 15
- When you guess a number, answer either *correct*, *smaller*, or *larger*
  - For example: is it 5? Ola: *larger*
- What is your best strategy to minimize number of guesses?



Idea

Guessing Game:

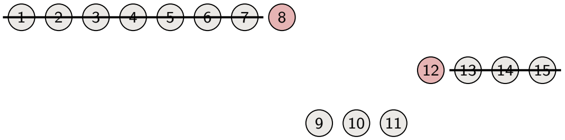
- Ola thinks of an integer between 1 and 15
- When you guess a number, answer either *correct*, *smaller*, or *larger*
  - For example: is it 5? Ola: *larger*
- What is your best strategy to minimize number of guesses?



Idea

Guessing Game:

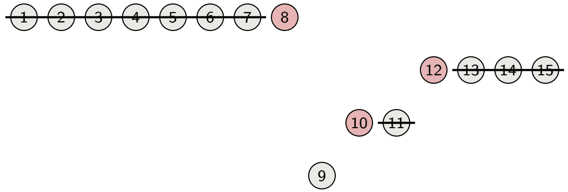
- Ola thinks of an integer between 1 and 15
- When you guess a number, answer either *correct*, *smaller*, or *larger*
  - For example: is it 5? Ola: *larger*
- What is your best strategy to minimize number of guesses?



Idea

Guessing Game:

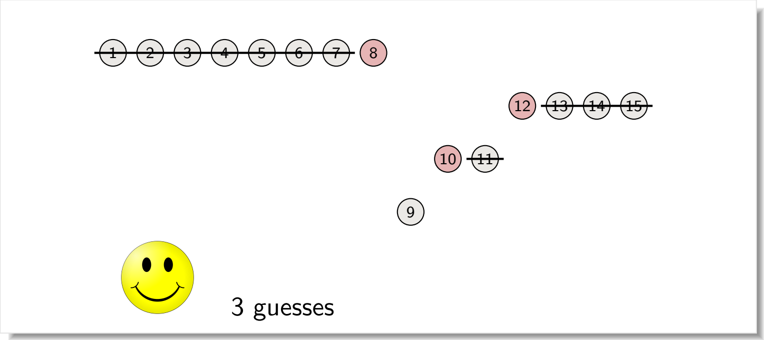
- Ola thinks of an integer between 1 and 15
- When you guess a number, answer either *correct*, *smaller*, or *larger*
  - For example: is it 5? Ola: *larger*
- What is your best strategy to minimize number of guesses?



Idea

Guessing Game:

- Ola thinks of an integer between 1 and 15
- When you guess a number, answer either *correct*, *smaller*, or *larger*
  - For example: is it 5? Ola: *larger*
- What is your best strategy to minimize number of guesses?



Binary Search Trees

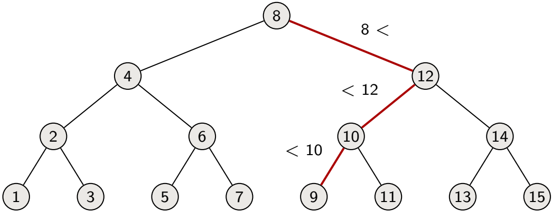
Encodes a strategy whatever number we look for

Binary Search Trees

Encodes a strategy whatever number we look for

Key property:

- If y is in the left subtree of x then $y.key < x.key$
- If y is in the right subtree of x then $y.key \geq x.key$

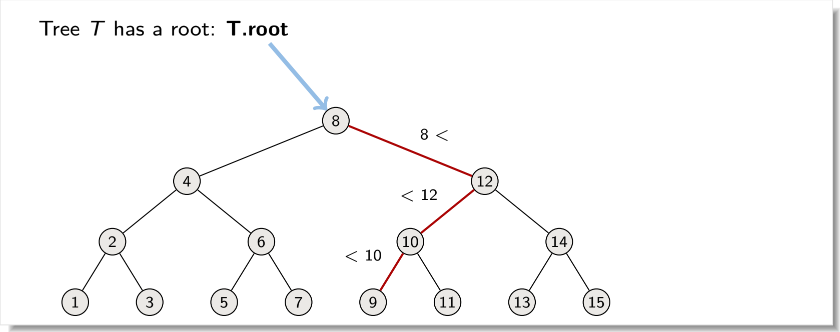


Binary Search Trees

Encodes a strategy whatever number we look for

Key property:

- If y is in the left subtree of x then $y.key < x.key$
- If y is in the right subtree of x then $y.key \geq x.key$



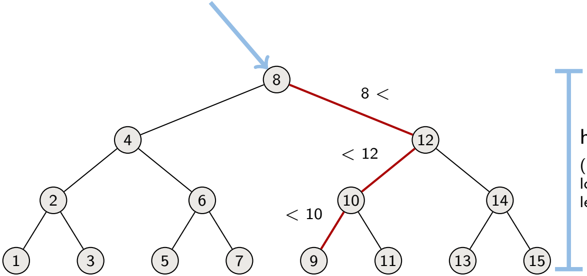
Binary Search Trees

Encodes a strategy whatever number we look for

Key property:

- If y is in the left subtree of x then $y.key < x.key$
- If y is in the right subtree of x then $y.key \geq x.key$

Tree T has a root: **T.root**



height $h = 3$

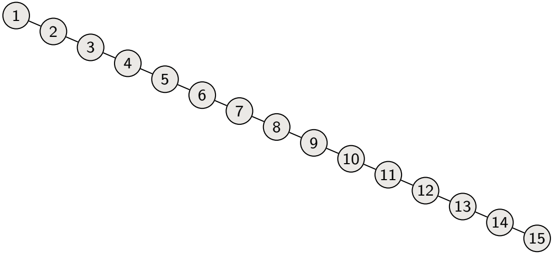
(number of edges in longest path from root to leaf)

Binary Search Trees

Encodes a strategy whatever number we look for

Key property:

- If y is in the left subtree of x then $y.key < x.key$
- If y is in the right subtree of x then $y.key \geq x.key$

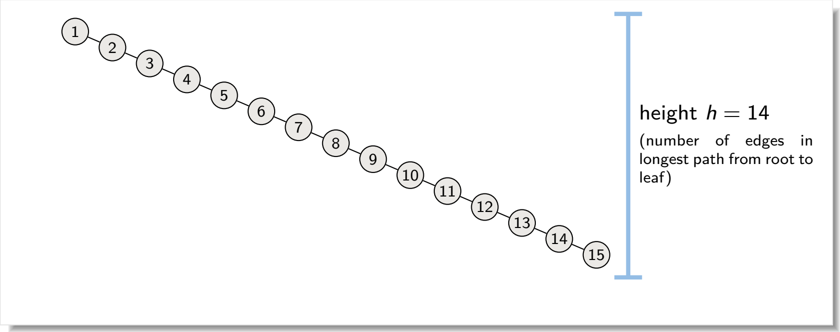


Binary Search Trees

Encodes a strategy whatever number we look for

Key property:

- If y is in the left subtree of x then $y.key < x.key$
- If y is in the right subtree of x then $y.key \geq x.key$

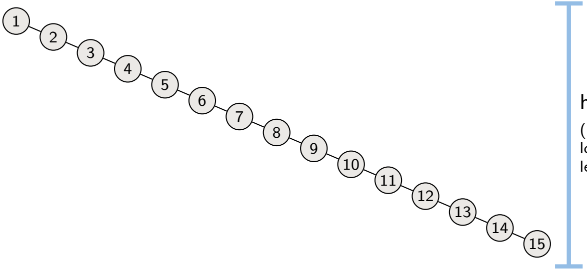


Binary Search Trees

Encodes a strategy whatever number we look for

Key property:

- If y is in the left subtree of x then $y.key < x.key$
- If y is in the right subtree of x then $y.key \geq x.key$



height $h = 14$

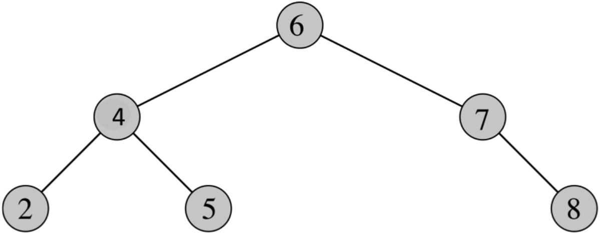
(number of edges in longest path from root to leaf)

Basic operations take time proportional to height: $O(h)$

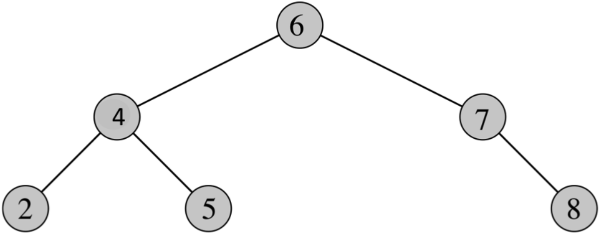
QUERYING A BINARY SEARCH TREE

(Searching, Minimum, Maximum, Successor, Predecessor)

Searching

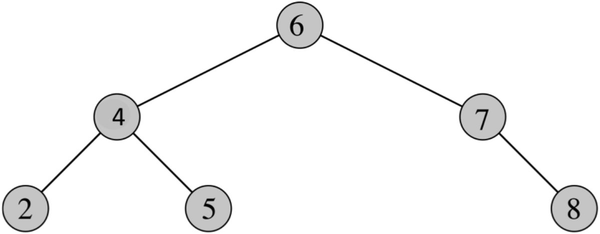


Searching




```
TREE-SEARCH(x, k)
  if x == NIL or k == key[x]
      return x
  if k < x.key
      return TREE-SEARCH(x.left, k)
  else return TREE-SEARCH(x.right, k)
```

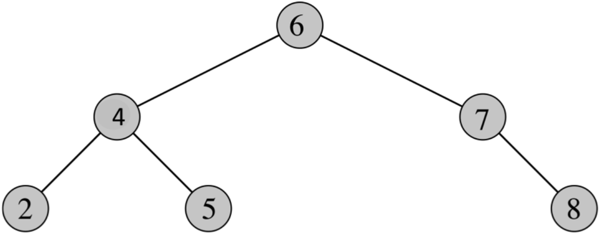
Searching



What is the running time?

```
TREE-SEARCH(x, k)
  if x == NIL or k == key[x]
      return x
  if k < x.key
      return TREE-SEARCH(x.left, k)
  else return TREE-SEARCH(x.right, k)
```

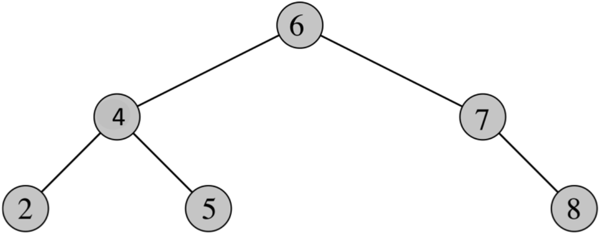
Searching



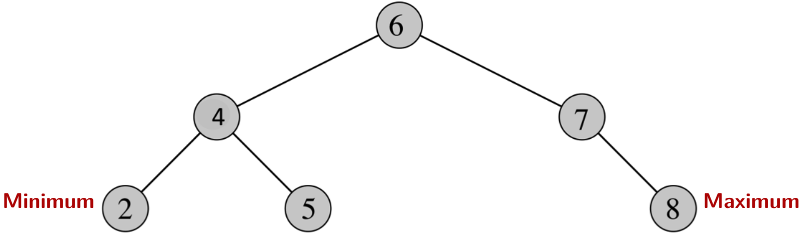
What is the running time? $O(h)$

```
TREE-SEARCH(x, k)
  if x == NIL or k == key[x]
      return x
  if k < x.key
      return TREE-SEARCH(x.left, k)
  else return TREE-SEARCH(x.right, k)
```

Minimum and Maximum



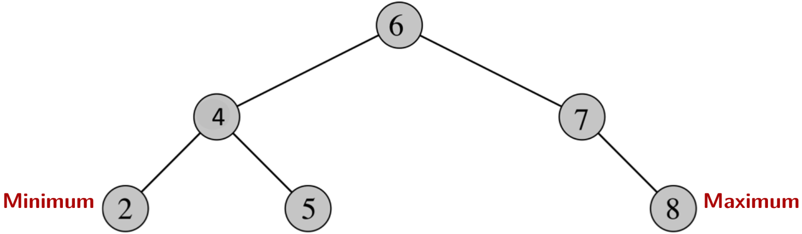
Minimum and Maximum



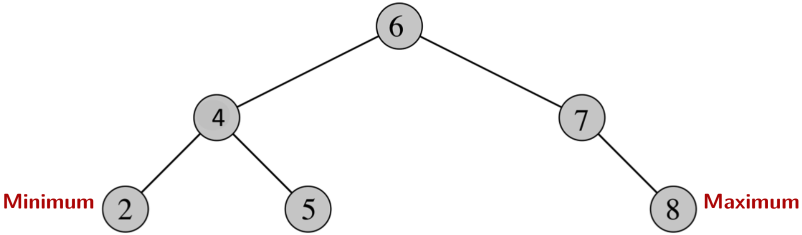
By key property:

- Minimum is located in leftmost node
- Maximum is located in rightmost node

Minimum and Maximum

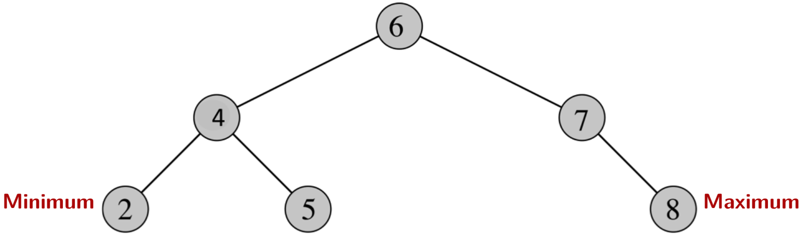


Minimum and Maximum



TREE-MINIMUM(x)
  while $x.left \neq$ NIL
    $x = x.left$
  return x

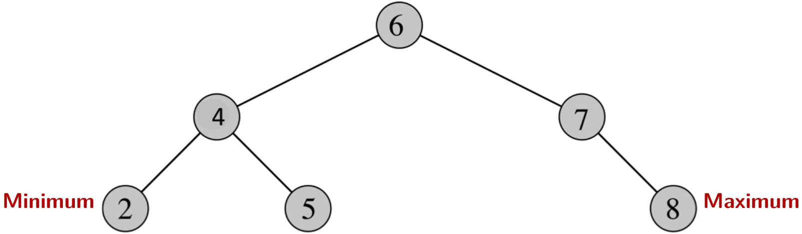
Minimum and Maximum



TREE-MINIMUM(x)
  while $x.left \neq$ NIL
    $x = x.left$
  return x

TREE-MAXIMUM(x)
  while $x.right \neq$ NIL
    $x = x.right$
  return x

Minimum and Maximum

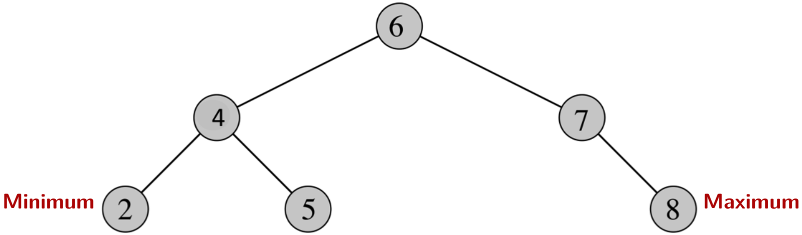


What is the running time?

TREE-MINIMUM(x)
while $x.left \neq$ NIL
$x = x.left$
return x

TREE-MAXIMUM(x)
while $x.right \neq$ NIL
$x = x.right$
return x

Minimum and Maximum

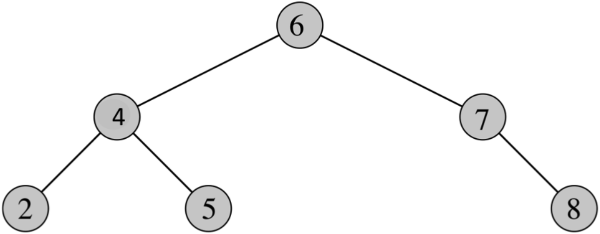


What is the running time? $O(h)$

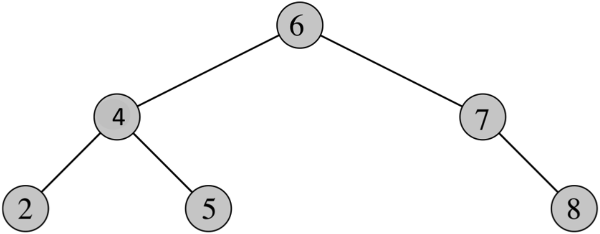
```
TREE-MINIMUM(x)
  while x.left ≠ NIL
      x = x.left
  return x
```

```
TREE-MAXIMUM(x)
  while x.right ≠ NIL
      x = x.right
  return x
```

Successor



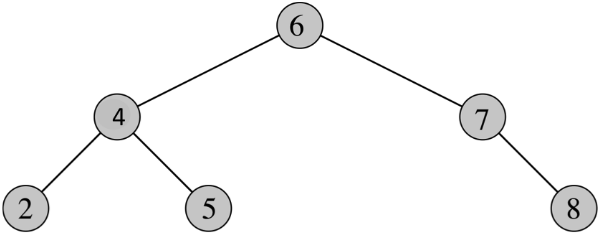
Successor



Successor of a noce x is the node y such that $y.key$ is the

$$\text{"smallest key"} > x.key$$

Successor

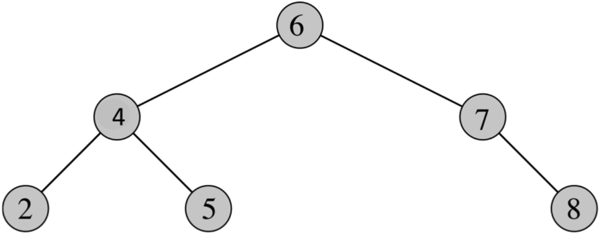


Successor of a noce x is the node y such that $y.key$ is the

$$\text{“smallest key”} > x.key$$

- What is the successor of 6?

Successor



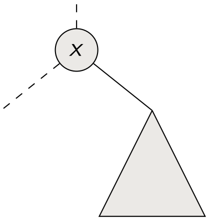
Successor of a noce x is the node y such that $y.key$ is the

$$\text{“smallest key”} > x.key$$

- What is the successor of 6?
- What is the successor of 5?

Two cases when finding successor of x:

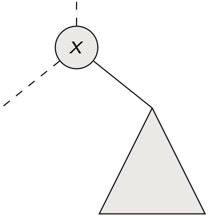
Case 1: x has a non-empty right subtree



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

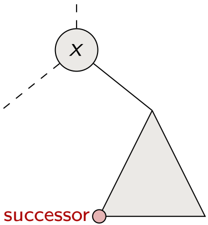
x's successor is the minimum in the right subtree



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

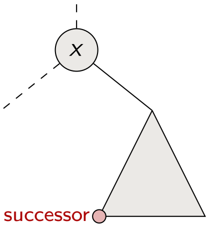
x's successor is the minimum in the right subtree



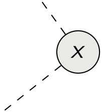
Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

x's successor is the minimum in the right subtree



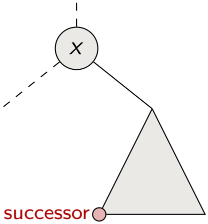
Case 2: x has an empty right subtree



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

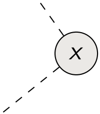
x's successor is the minimum in the right subtree



Case 2: x has an empty right subtree

As long as we to the left up the tree we're visiting smaller keys

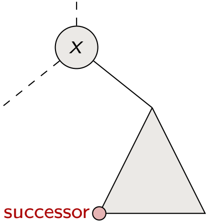
x's successor is y is the node that x is the predecessor of (x is the maximum in y's left subtree)



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

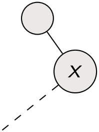
x's successor is the minimum in the right subtree



Case 2: x has an empty right subtree

As long as we to the left up the tree we're visiting smaller keys

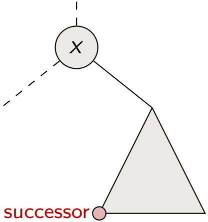
x's successor is y is the node that x is the predecessor of (x is the maximum in y's left subtree)



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

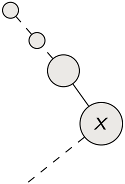
x's successor is the minimum in the right subtree



Case 2: x has an empty right subtree

As long as we to the left up the tree we're visiting smaller keys

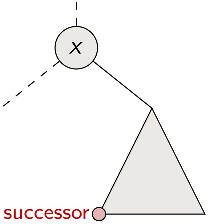
x's successor is y is the node that x is the predecessor of (x is the maximum in y's left subtree)



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

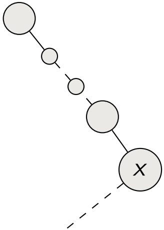
x's successor is the minimum in the right subtree



Case 2: x has an empty right subtree

As long as we to the left up the tree we're visiting smaller keys

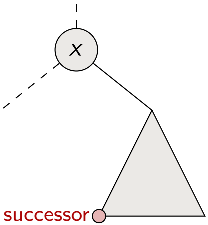
x's successor is y is the node that x is the predecessor of (x is the maximum in y's left subtree)



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

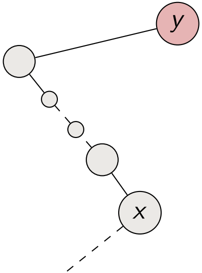
x's successor is the minimum in the right subtree



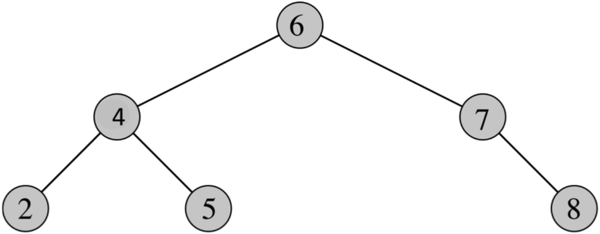
Case 2: x has an empty right subtree

As long as we to the left up the tree we're visiting smaller keys

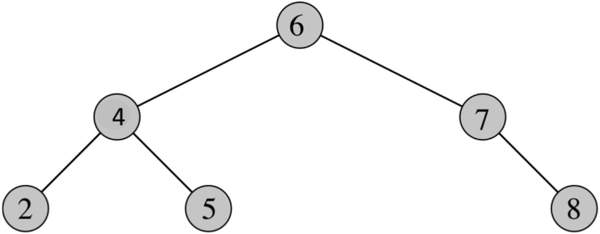
x's successor is y is the node that x is the predecessor of (x is the maximum in y's left subtree)



Successor (Predecessor is symmetric)

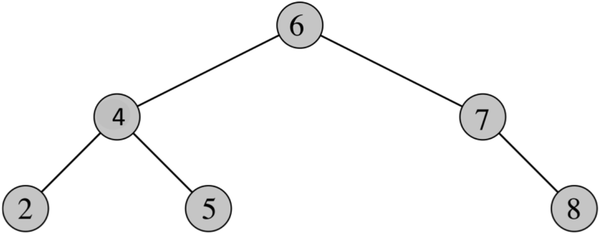


Successor (Predecessor is symmetric)




```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

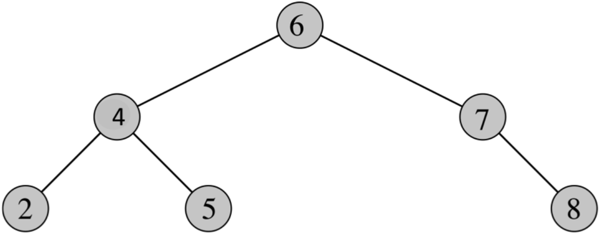
Successor (Predecessor is symmetric)



What is the running time?

```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

Successor (Predecessor is symmetric)



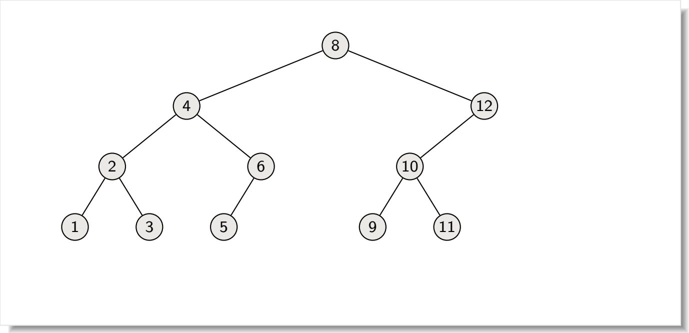
What is the running time? $O(h)$

```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

PRINTING A BINARY SEARCH TREE

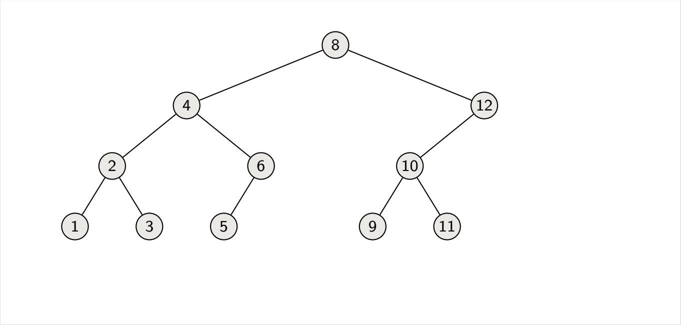
(Inorder, Preorder, Postorder)

Printing Inorder (Idea)



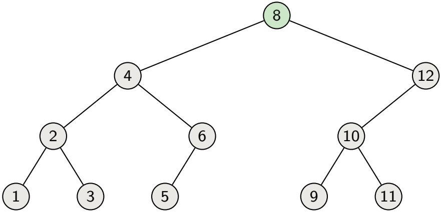
Printing Inorder (Idea)

- Print left subtree recursively
- Print root
- Print right subtree recursively



Printing Inorder (Idea)

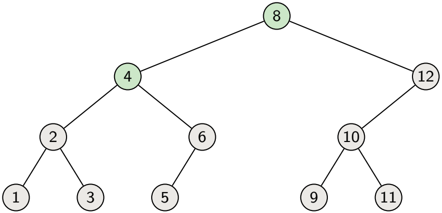
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output:

Printing Inorder (Idea)

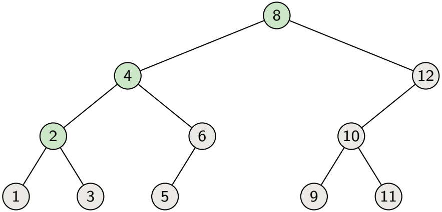
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output:

Printing Inorder (Idea)

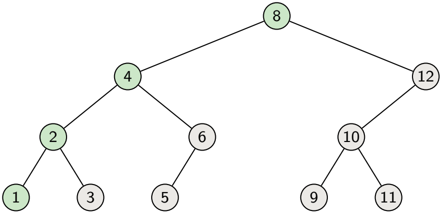
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output:

Printing Inorder (Idea)

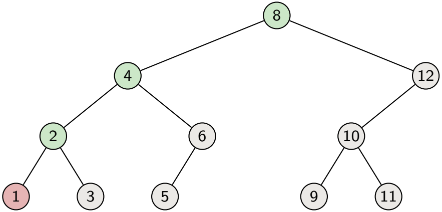
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output:

Printing Inorder (Idea)

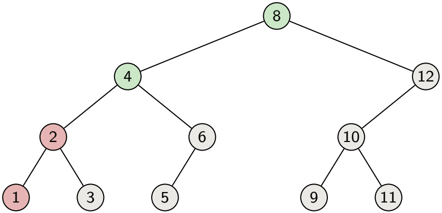
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,

Printing Inorder (Idea)

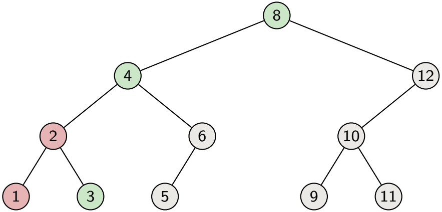
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,

Printing Inorder (Idea)

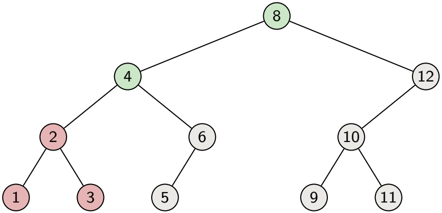
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,

Printing Inorder (Idea)

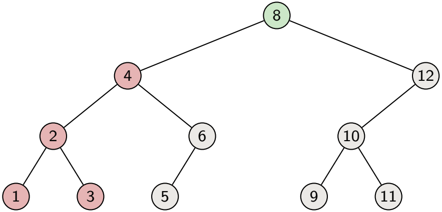
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,

Printing Inorder (Idea)

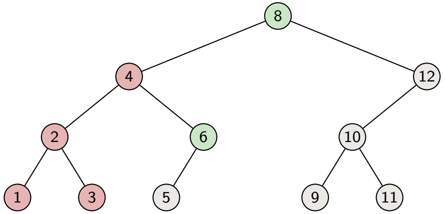
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,

Printing Inorder (Idea)

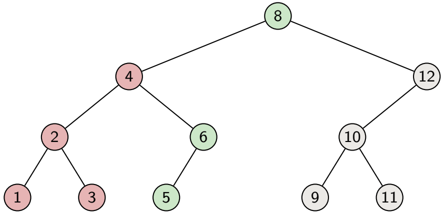
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,

Printing Inorder (Idea)

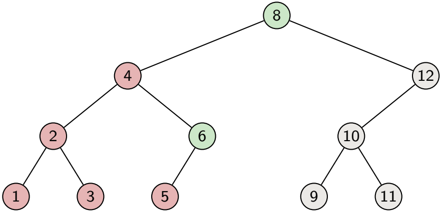
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,

Printing Inorder (Idea)

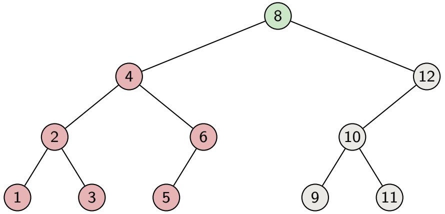
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,

Printing Inorder (Idea)

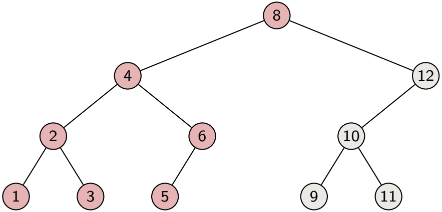
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,

Printing Inorder (Idea)

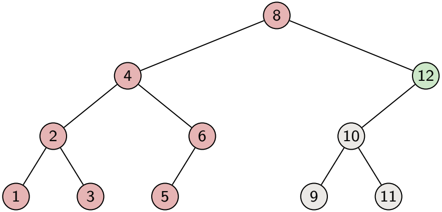
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,

Printing Inorder (Idea)

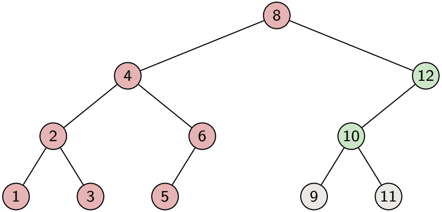
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,

Printing Inorder (Idea)

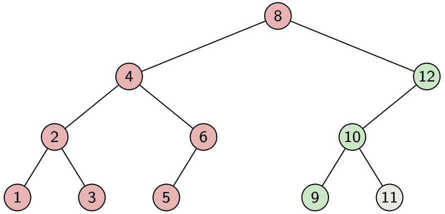
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,

Printing Inorder (Idea)

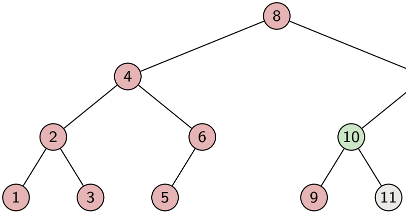
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,

Printing Inorder (Idea)

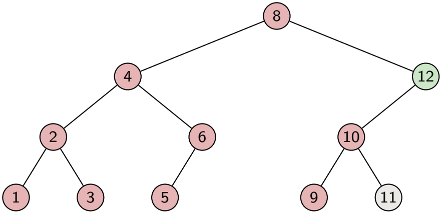
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,9,

Printing Inorder (Idea)

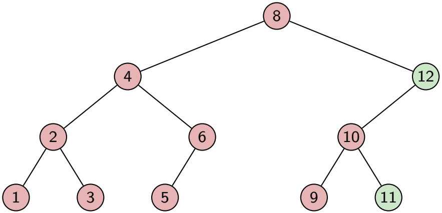
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,9,10,

Printing Inorder (Idea)

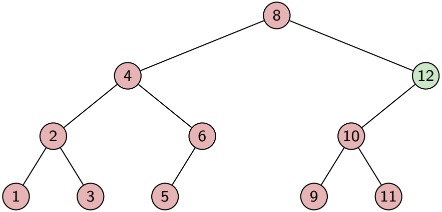
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,9,10,

Printing Inorder (Idea)

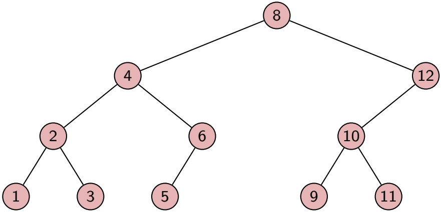
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,9,10,11,

Printing Inorder (Idea)

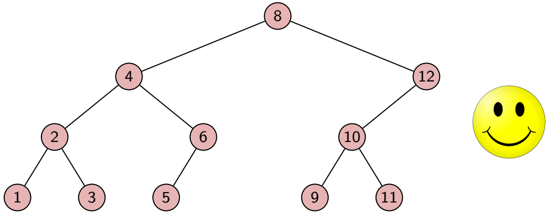
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,9,10,11,12

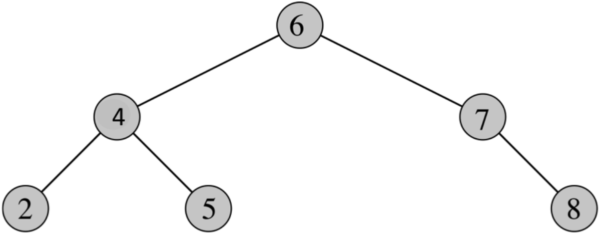
Printing Inorder (Idea)

- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,9,10,11,12

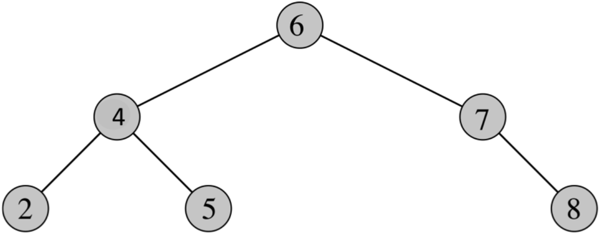
Inorder tree walk



INORDER-TREE-WALK(x)

if $x \neq$ NIL
    INORDER-TREE-WALK$(x.\mathit{left})$
    print $\mathit{key}[x]$
    INORDER-TREE-WALK$(x.\mathit{right})$

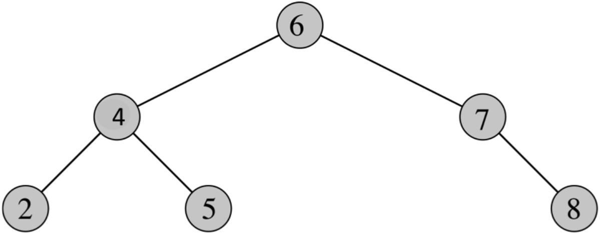
Inorder tree walk



What is the running time?

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

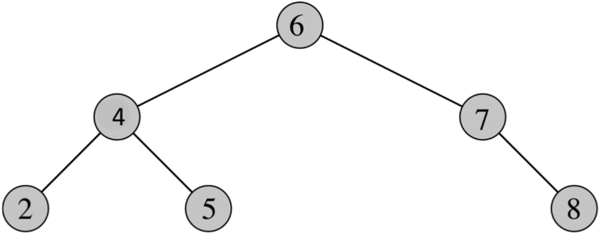
Inorder tree walk



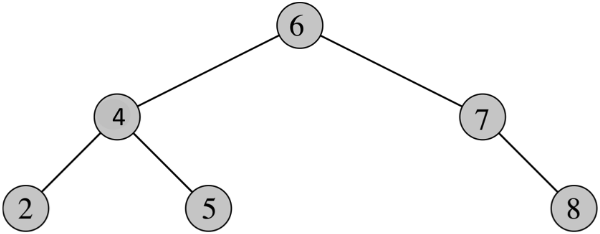
What is the running time? $\Theta(n)$

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

Printing Preorder and Postorder



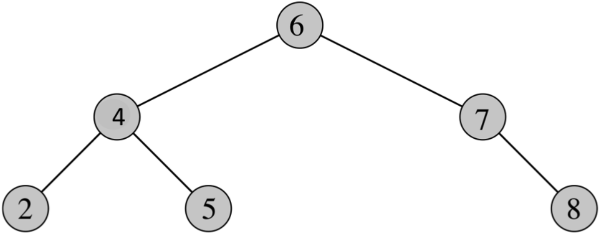
Printing Preorder and Postorder



PREORDER-TREE-WALK(x)

1. **if** $x \neq NIL$
2. print $key[x]$
3. PREORDER-TREE-WALK($x.left$)
4. PREORDER-TREE-WALK($x.right$)

Printing Preorder and Postorder



PREORDER-TREE-WALK(x)

1. **if** $x \neq NIL$
2. print $key[x]$
3. PREORDER-TREE-WALK($x.left$)
4. PREORDER-TREE-WALK($x.right$)

POSTORDER-TREE-WALK(x)

1. **if** $x \neq NIL$
2. POSTORDER-TREE-WALK($x.left$)
3. POSTORDER-TREE-WALK($x.right$)
4. print $key[x]$

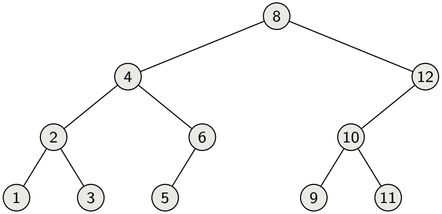
MODIFYING A BINARY SEARCH TREE

(Insertion and Deletion)

Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

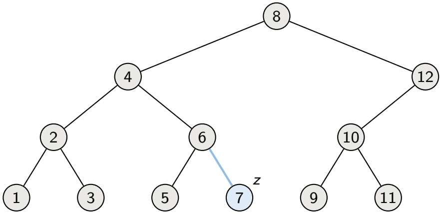
Ex: insert z with key 7



Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

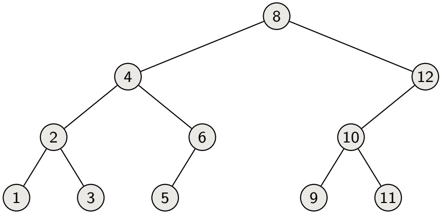
Ex: insert z with key 7



Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

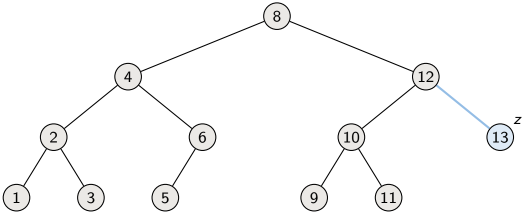
Ex: insert z with key 13



Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

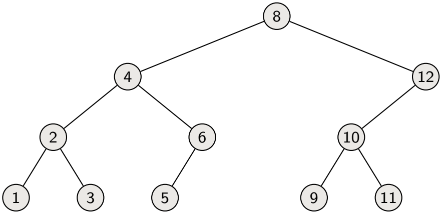
Ex: insert z with key 13



Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

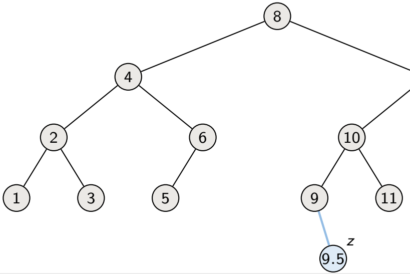
Ex: insert z with key 9.5



Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

Ex: insert z with key 9.5



Insertion

"search" phase

"insert" phase

```
TREE-INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z          // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
```

Insertion

"search" phase

"insert" phase

```
TREE-INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z          // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
```

What is the running time?

Insertion

"search" phase

"insert" phase

```
TREE-INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z          // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
```

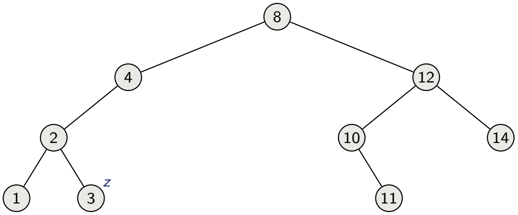
What is the running time? $O(h)$

Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it

Ex: Delete z

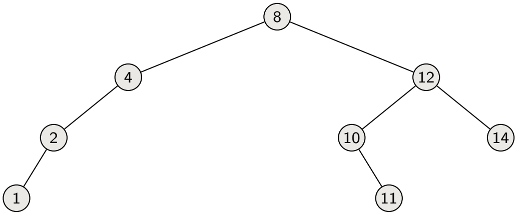


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it

Ex: Delete z

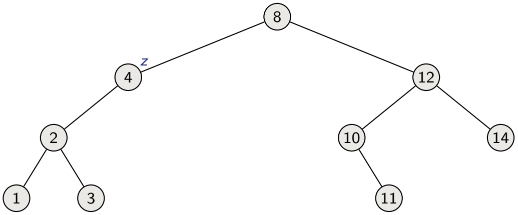


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z'z position in the tree

Ex: Delete z

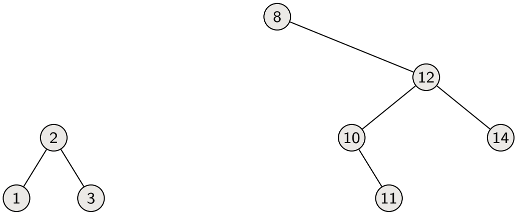


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z'z position in the tree

Ex: Delete z

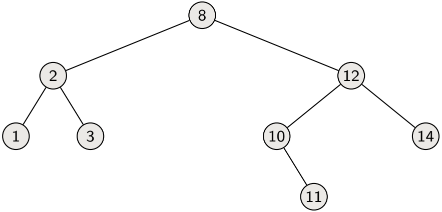


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z'z position in the tree

Ex: Delete z

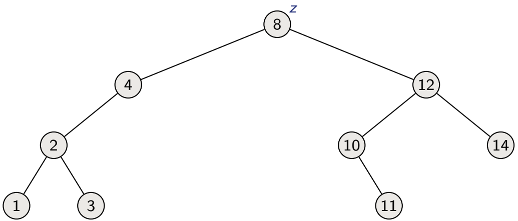


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z'z position in the tree
- If z has two children, then find its successor y and replace z by y

Ex: Delete z

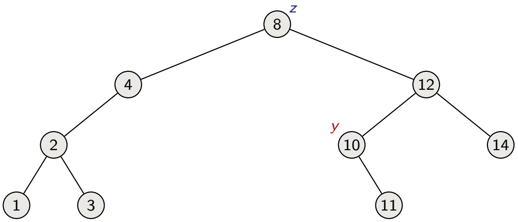


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z'z position in the tree
- If z has two children, then find its successor y and replace z by y

Ex: Delete z

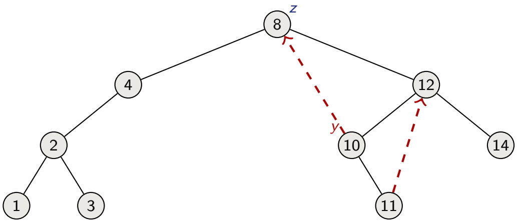


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z'z position in the tree
- If z has two children, then find its successor y and replace z by y

Ex: Delete z

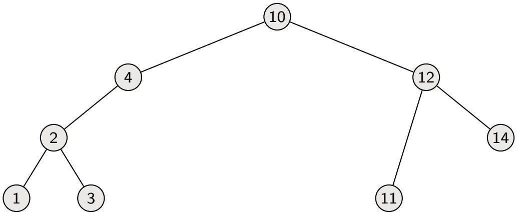


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z'z position in the tree
- If z has two children, then find its successor y and replace z by y

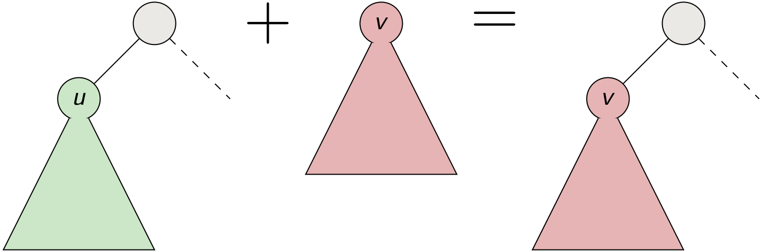
Ex: Delete z



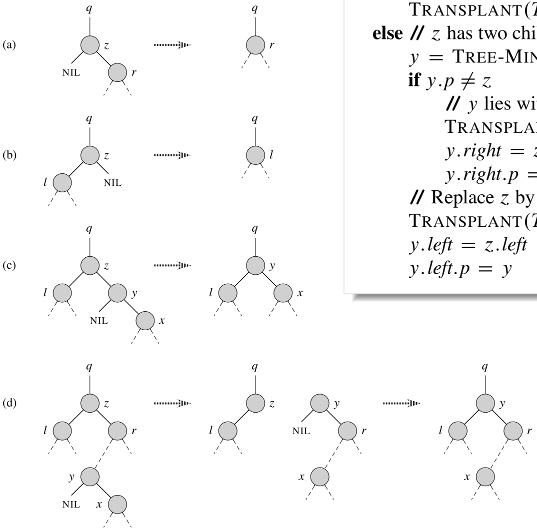
Deletion Implementation: Transplant

TRANSPLANT(T, u, v)
if $u.p$ == NIL
  $T.root = v$
elseif u == $u.p.left$
  $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
  $v.p = u.p$

TRANSPLANT(T, u, v) replaces subtree rooted at u with that rooted at v



Deletion Procedure




```
TREE-DELETE(T, z)
  if z.left == NIL
      TRANSPLANT(T, z, z.right)          // z has no left child
  elseif z.right == NIL
      TRANSPLANT(T, z, z.left)           // z has just a left child
  else // z has two children.
      y = TREE-MINIMUM(z.right)          // y is z's successor
      if y.p ≠ z
          // y lies within z's right subtree but is not the root of this
          TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      // Replace z by y.
      TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
```

Summary

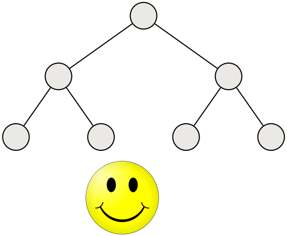
Query operations: Search, Max, Min, Predecessor, Successor: **$O(h)$** time

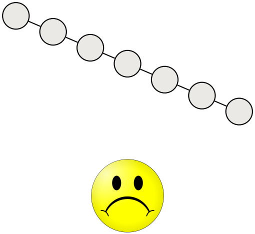
Modifying operations: Insertion, Deletion: **$O(h)$** time

Summary

Query operations: Search, Max, Min, Predecessor, Successor: **O(h)** time

Modifying operations: Insertion, Deletion: **O(h)** time





Summary

Query operations: Search, Max, Min, Predecessor, Successor: **O(h)** time

Modifying operations: Insertion, Deletion: **O(h)** time

Exist efficient procedures to keep tree balanced (AVL trees, red-black trees, etc.)