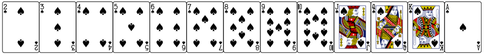
COMP 2011: Data Structures

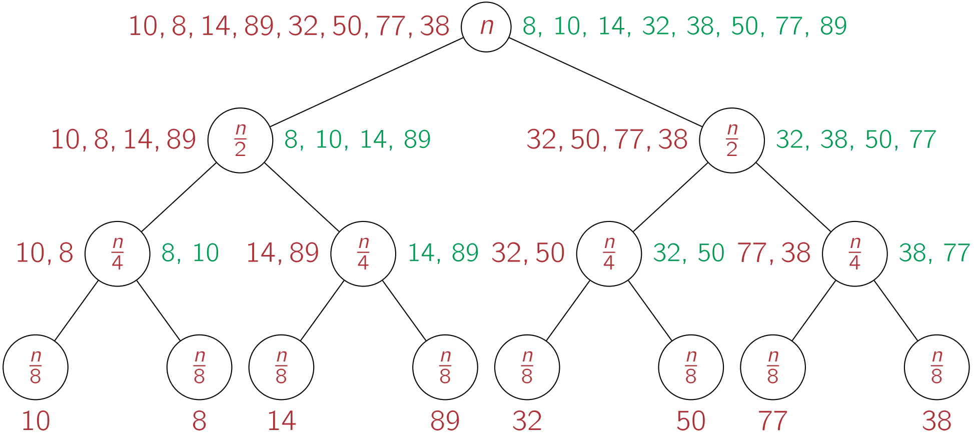
Lecture 8. Binary (Search) Trees

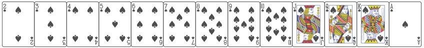
Dr. CAO Yixin

October, 2021

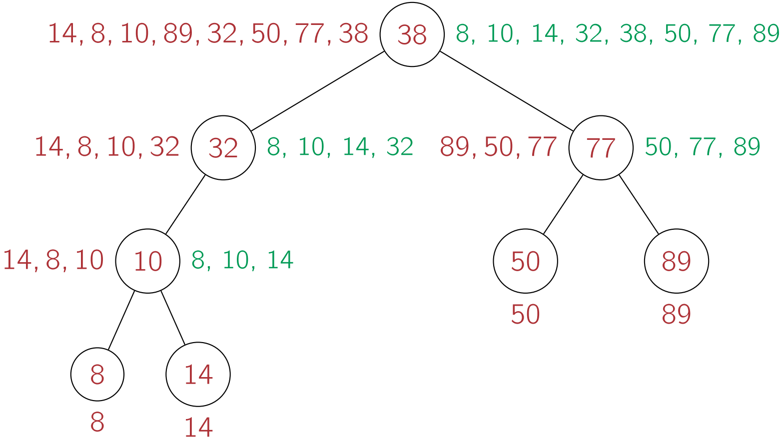


The execution of mergesort



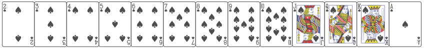


The execution of quicksort



Review of sorting algorithms

- The idea of mergesort and quicksort is still very easy,
- The challenge is how to implement them, especially in-place quicksort.
- They cannot compare to insertion sort for almost sorted arrays.
- So what are used in practice are timsort (merge+insertion) and quick+insertion.
- "quick+insertion" is mostly faster than timsort, why?
- It seems that bubble and selection are too slow to be useful?
- We don't use bubble sort, but the idea of checking through to make sure it's sorted is very useful.
- We can go through the array to see whether how far it's from sorted.
- Selection is always bad. But can it be salvaged?
- It's slow because of repetitive comparisons in finding a largest element.
- What if there is a data structure that can keep track the comparisons, then ...

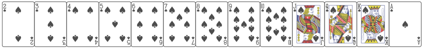


Overview

- Trees and binary trees.
- Traversals: different ways of visiting a binary tree.

- Binary search trees.
- insertions;
- search;
- deletions.


ROUTE
66



Decidophobia: The fear of making the wrong decision

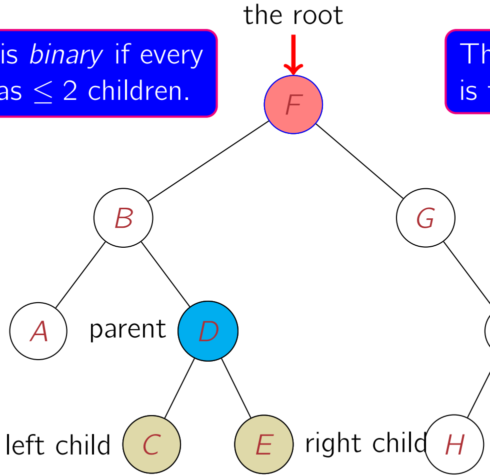
Walter Kaufmann (), *Without Guilt and Justice*

Binary Trees

A binary tree

A tree is *binary* if every node has ≤ 2 children.

The only node with no parent is the *root* of the tree

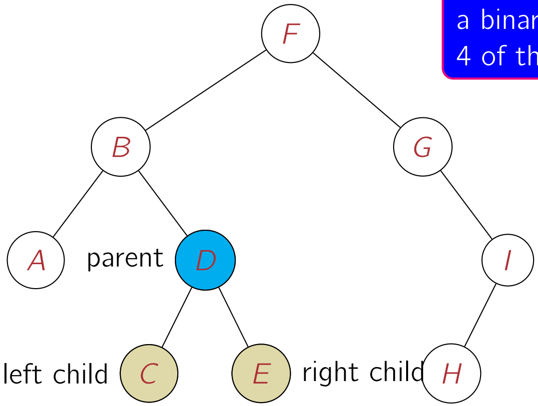


I is the right child of G.

Nodes with no child are called *leaves*

D is the *parent* of C and E;
C and E are *children* of node D;
C and E are *siblings* of each other.

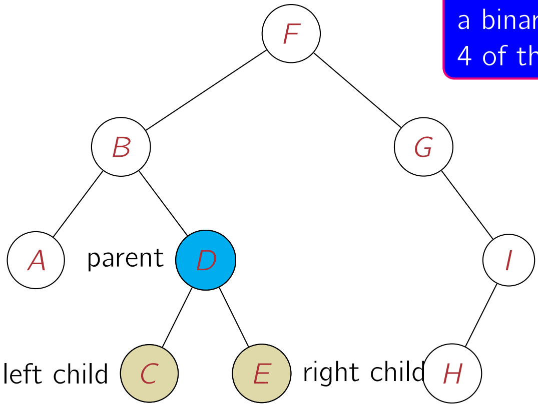
A binary tree



Summary:
a binary tree of height 3;
4 of the 9 nodes are leaves

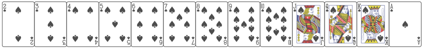
How many more nodes we can add to this tree and it remains a binary tree of height 3?

A binary tree

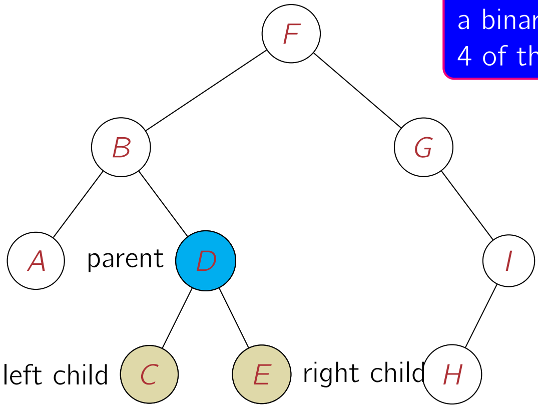


Summary:
a binary tree of height 3;
4 of the 9 nodes are leaves

How many more nodes we can add to this tree and it remains a binary tree of height 3?
How many leaves then?



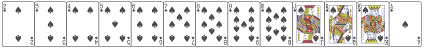
A binary tree



Summary:
a binary tree of height 3;
4 of the 9 nodes are leaves

How many more nodes we can add to this tree and it remains a binary tree of height 3?
How many leaves then?

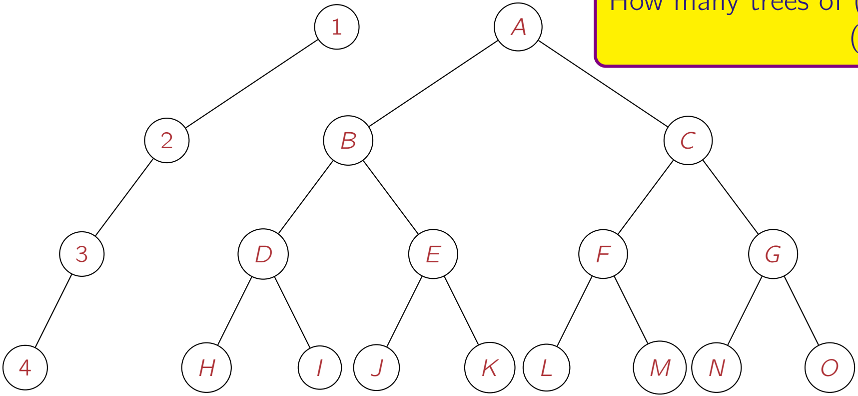
How many nodes we can delete from this tree and it remains a binary tree of height 3?
How many leaves then?



Glossary

- a tree is *binary* if every node has at most two children
  a child is either left child or right child; they are *siblings* of each other;
- the only node with no parent is the *root* of the tree;
- nodes with no child are *leaves*;
- a *path* is a sequence of edges, each starting with the node the previous edge ends;
- the *length* of path is the number of edges in it;
- any node may be considered to be the root of a *subtree*, this subtree consists of its children, and its children's children, and so on;
- the *depth* (or *level*) of a node is the length of path from root to the node;
- the *height* of a node is the length of the longest path from it down to a leaf;
- the *height* of a tree is the height of its root, which equals its height;
- all leaves have height 0.

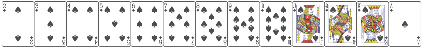
Skew and perfect binary trees



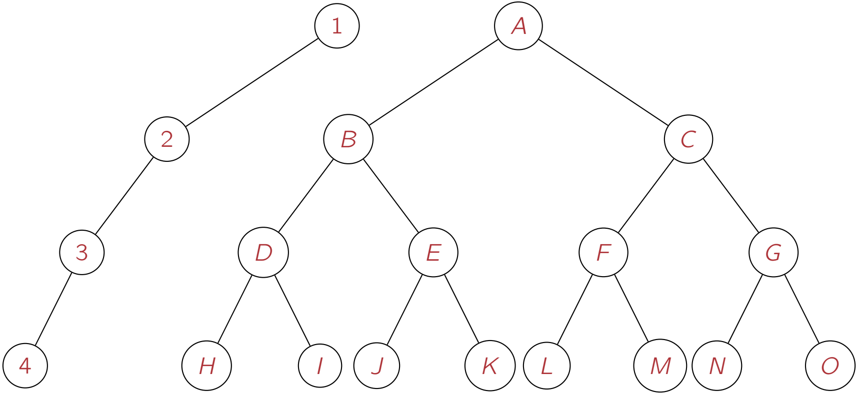
How many trees of (k levels)?
(n nodes)?

#nodes in a tree of depth k is between $k+1$ and $2^{k+1}-1$

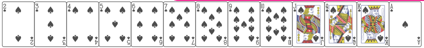
Depth of a tree of n nodes is between $\lfloor \log n \rfloor$ and $n-1$



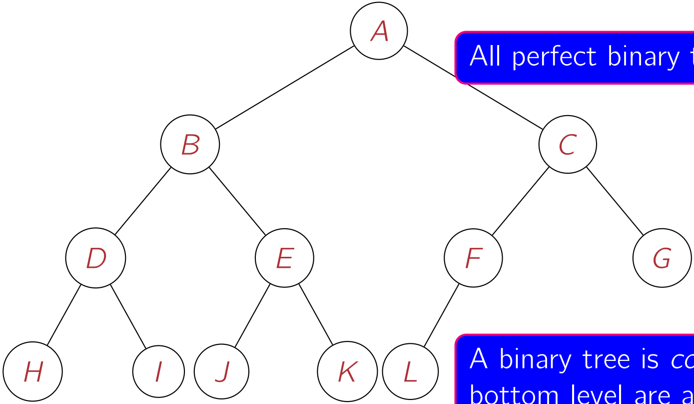
Skew and perfect binary trees



A binary tree is *perfect* if all leaves on the same level and each non-leaf node has exactly two children.

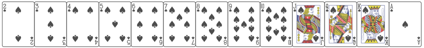


Complete (leftmost) binary trees



All perfect binary trees are complete.

A binary tree is *complete* if all leaves at the bottom level are as far to the left as possible and every other level is completely filled.

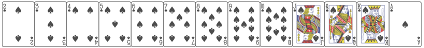


The codes

```
public class BinaryTree<T> {
    private class Node<T> {
        T data;
        public Node<T> leftChild, rightChild;
    }

    Node<T> root;
}
```

Very similar as a linked list, but with two references.

How many `null` references in a tree of n nodes?



Trees are difficult in general

Both power and difficulty of trees are from their great flexibility.

A general tree may have an arbitrary number of children. How to implement it?

- For each node, use a linked list to store references to its children.
  If you're interested, see Figure 8.12 in Section 8.3.3 of the textbook by Goodrich et al.
- Each node stores the reference to its first child and its next sibling.

For some applications, we may want to make the links bidirectional.
how to augment the binary tree data structure? `parent` or `sibling`

Examples of Binary Trees

World Cup 2018

30 June – Sochi

Uruguay	**2**
Portugal	1

30 June – Kazan

France	**4**
Argentina	3

2 July – Samara

Brazil	**2**
Mexico	0

2 July – Rostov-on-Don

Belgium	**3**
Japan	2

1 July – Moscow (Luzhniki)

Spain	1 (3)
Russia (p)	**1 (4)**

1 July – Nizhny Novgorod

Croatia (p)	**1 (3)**
Denmark	1 (2)

3 July – Saint Petersburg

Sweden	**1**
Switzerland	0

3 July – Moscow (Otkritie)

Colombia	1 (3)
England (p)	**1 (4)**

6 July – Nizhny Novgorod

Uruguay	0
France	**2**

6 July – Kazan

Brazil	1
Belgium	**2**

7 July – Sochi

Russia	2 (3)
Croatia (p)	**2 (4)**

7 July – Samara

Sweden	0
England	**2**

10 July – Saint Petersburg

France	**1**
Belgium	0

11 July – Moscow (Luzhniki)

Croatia (a.e.t.)	**2**
England	1

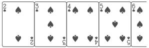
15 July – Moscow (Luzhniki)

France	**4**
Croatia	2

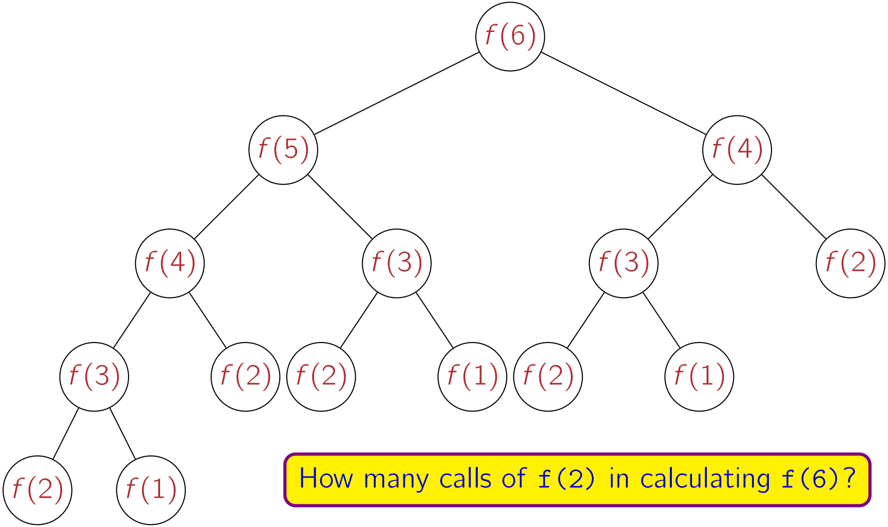
Third place play-off

14 July – Saint Petersburg

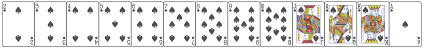
Belgium	**2**
England	0



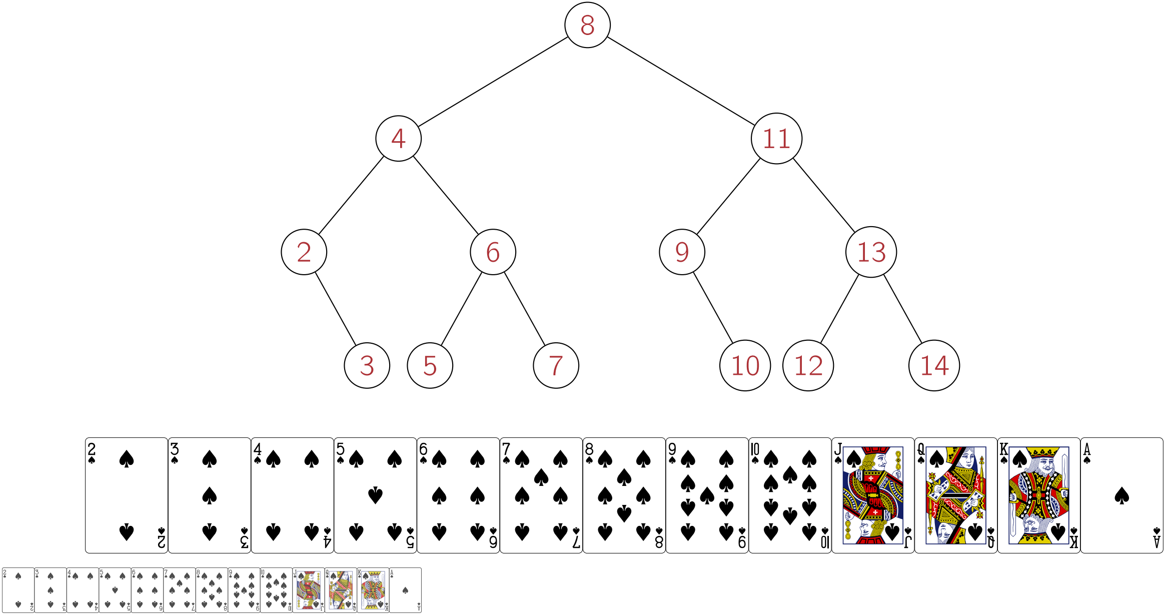
Fibonacci numbers



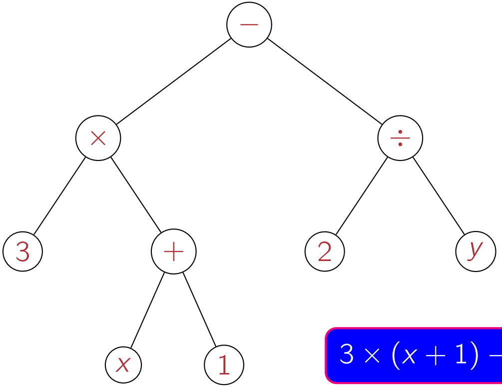
How many calls of `f(2)` in calculating `f(6)`?



Binary search trees



Expressions



$$3 \times (x + 1) - 2 \div y$$

Binary Tree Traversals

Binary tree traversals

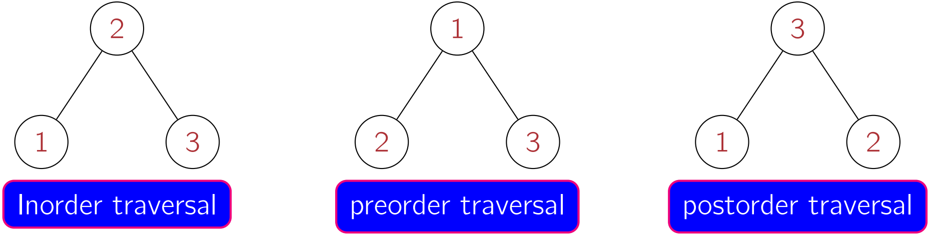
Tree traversal ( ) is the process of visiting (examining and/or updating) each node in a tree data structure, exactly once, in a systematic way.

- To traverse a tree means to visit all the nodes in some specified order.
- Usually, we visit a left subtree before the other (always assumed in this course).

Binary tree traversals

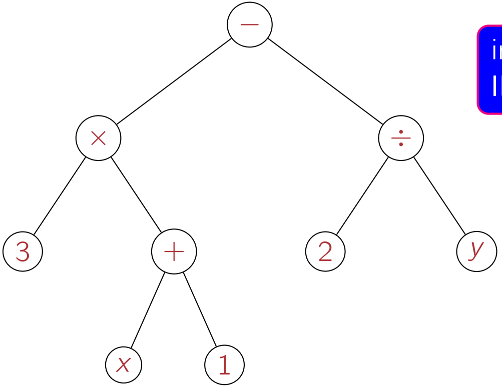
Tree traversal ( ) is the process of visiting (examining and/or updating) each node in a tree data structure, exactly once, in a systematic way.

- To traverse a tree means to visit all the nodes in some specified order.
- Usually, we visit a left subtree before the other (always assumed in this course).



visit a node IN BETWEEN/BEFORE/AFTER visiting its children

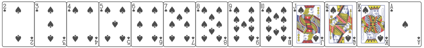
Demonstration



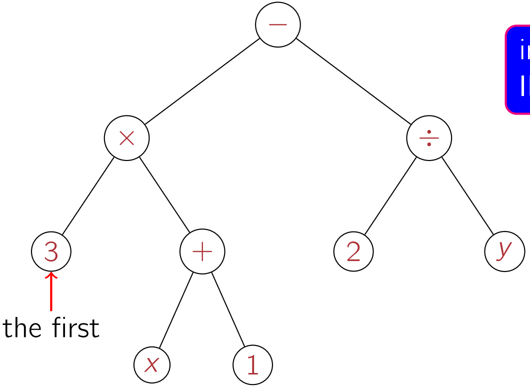
inorder traversal: visit a node
IN between visiting its two children

Which is the first node?

By *traversing* a tree, we “visit” (do some computation on) each node exactly once.



Demonstration

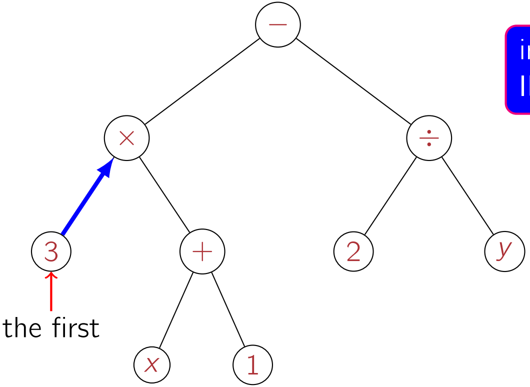


inorder traversal: visit a node
IN between visiting its two children

and next?

By *traversing* a tree, we "visit" (do some computation on) each node exactly once.

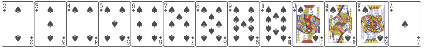
Demonstration



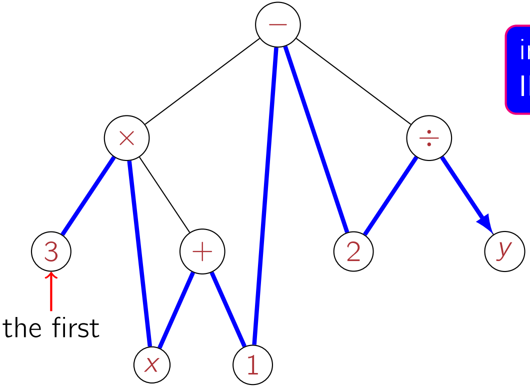
inorder traversal: visit a node
IN between visiting its two children

and next?

By *traversing* a tree, we "visit" (do some computation on) each node exactly once.



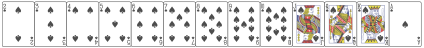
Demonstration



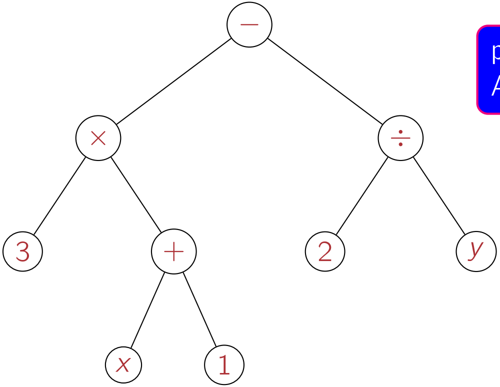
inorder traversal: visit a node
IN between visiting its two children

inorder: $3 \times x + 1 - 2 \div y$

By *traversing* a tree, we “visit” (do some computation on) each node exactly once.



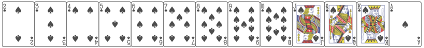
Demonstration



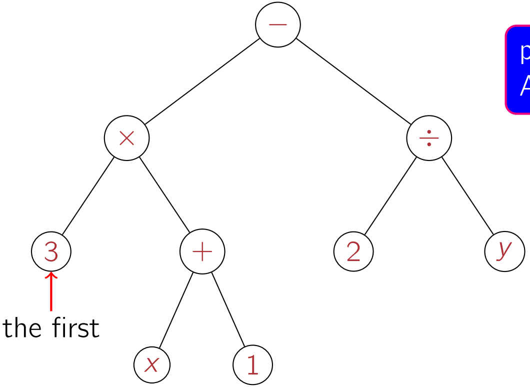
postorder traversal: visit a node AFTER visiting its children

Which is the first node?

By *traversing* a tree, we "visit" (do some computation on) each node exactly once.



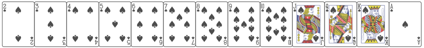
Demonstration



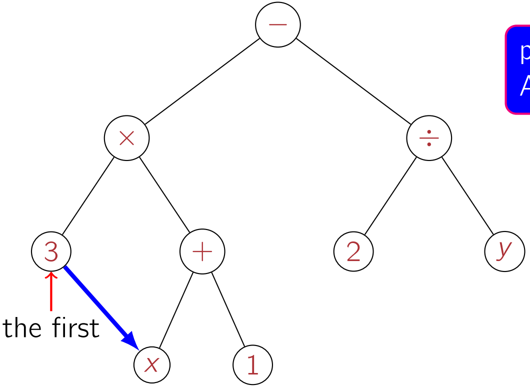
postorder traversal: visit a node AFTER visiting its children

and next?

By *traversing* a tree, we “visit” (do some computation on) each node exactly once.



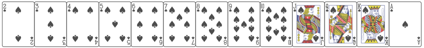
Demonstration



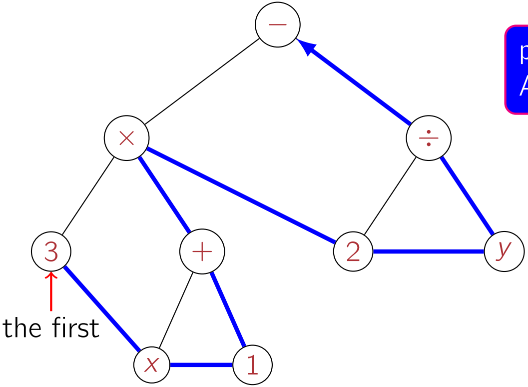
postorder traversal: visit a node AFTER visiting its children

and next?

By *traversing* a tree, we "visit" (do some computation on) each node exactly once.



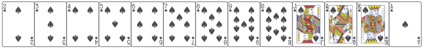
Demonstration



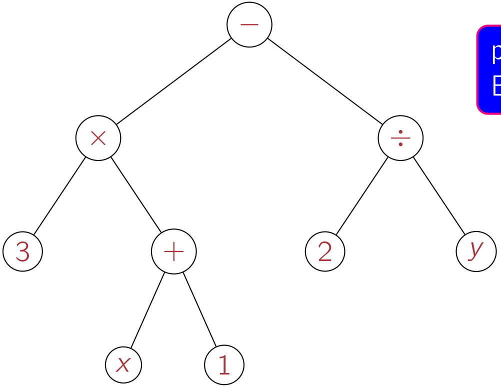
postorder traversal: visit a node AFTER visiting its children

postorder:

By *traversing* a tree, we “visit” (do some computation on) each node exactly once.



Demonstration



preorder traversal: visit a node BEFORE visiting its children

Try preorder traversal!

By *traversing* a tree, we “visit” (do some computation on) each node exactly once.

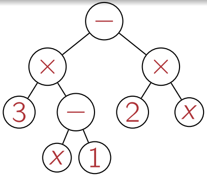
- Is there any other traversal?
- Please write the codes.

The codes

```
preorder(Node<T> curRoot) {
    if (curRoot == null) return;
    System.out.println(curRoot.data);
    preorder(curRoot.leftChild);
    preorder(curRoot.rightChild);
}
```

```
inorder(Node<T> curRoot) {



}
```

```
postorder(Node<T> curRoot) {



}
```



Running time?

Summary of traversals

- An inorder traversal visits nodes in order of ascending keys.
- Preorder and postorder traversals are useful for parsing algebraic expressions.

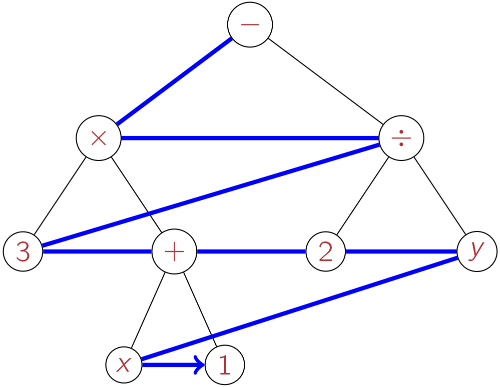
A question: Succession to the British throne ()

What kind of traversal on the royal family tree (not a binary tree)?

3. George
4. Charlotte
5. Louis
6. Harry

- Absolute primogeniture (male-preference primogeniture before 2013 ())
- Roman Catholics are disqualified (officially termed as being "naturally dead and deemed to be dead" in terms of succession).

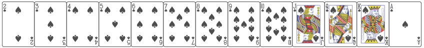
Level-wise traversal



Try to implement this traversal.

Hint: use a queue.

level-wise traversal (not make sense for expressions, but useful for other trees)



Very challenging questions

Using inorder traversal, two different trees can have the same sequence. Example?

Can two different trees have the same inorder sequence and the same preorder sequence?

Very challenging questions

Using inorder traversal, two different trees can have the same sequence. Example?

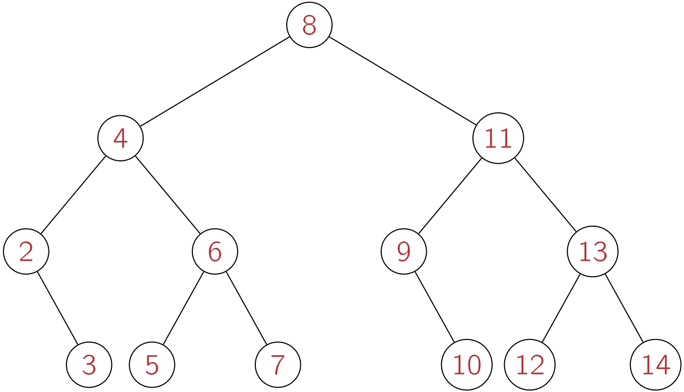
Can two different trees have the same inorder sequence
and the same preorder sequence?

Inorder traversal: □□□X■■■■■
Preorder traversal: X□□□■■■■■

Then □□□ are at the left subtree,
while ■■■■■ are at the right subtree.

Binary Search Trees

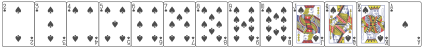
Binary search tree (again)



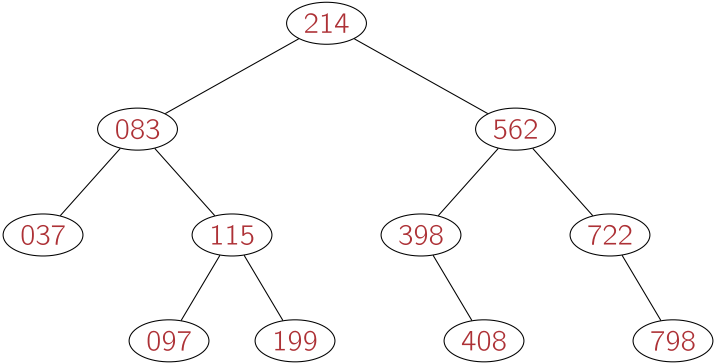
In a binary search tree, every node is

- larger than nodes in its left subtree, and
- smaller than nodes in its right subtree.

Write the output of inorder traversal?



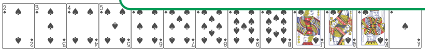
Searching in a binary search tree



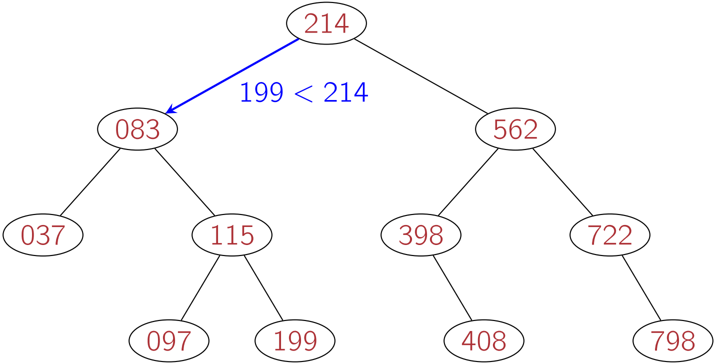
Search for k starting from the root

1. x is `null` — error
2. k `==` x`.id` — found
3. $k < x$`.id` — go left
4. $k > x$`.id` — go right

Let's search for 199 (Tse Kay).



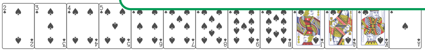
Searching in a binary search tree



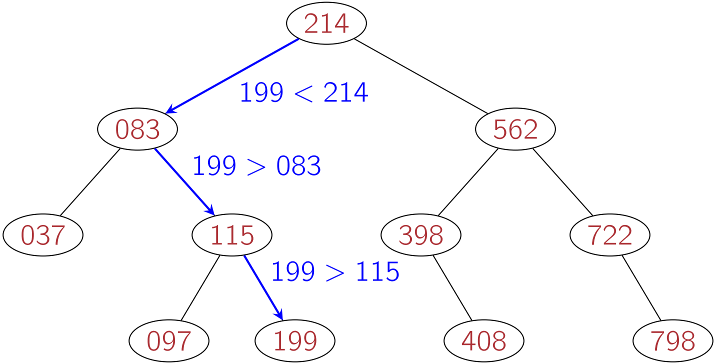
Search for k starting from the root

1. x is `null` — error
2. k `==` x`.id` — found
3. $k < x$`.id` — go left
4. $k > x$`.id` — go right

Let's search for 199 (Tse Kay).



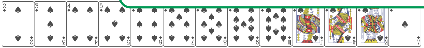
Searching in a binary search tree



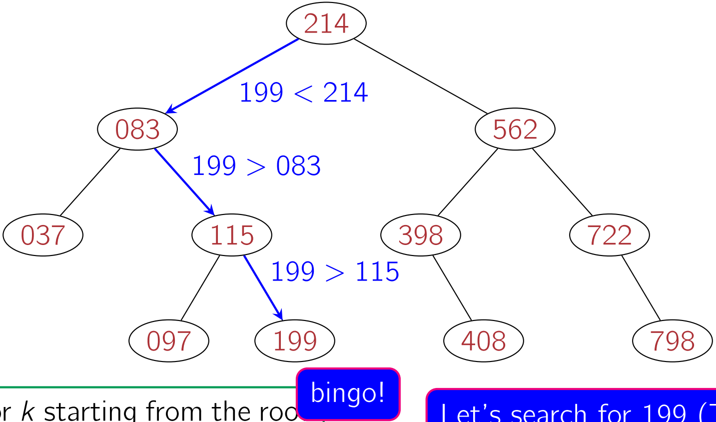
Search for k starting from the root

1. x is `null` — error
2. k `==` x`.id` — found
3. $k < x$`.id` — go left
4. $k > x$`.id` — go right

Let's search for 199 (Tse Kay).



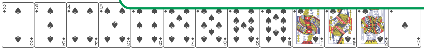
Searching in a binary search tree



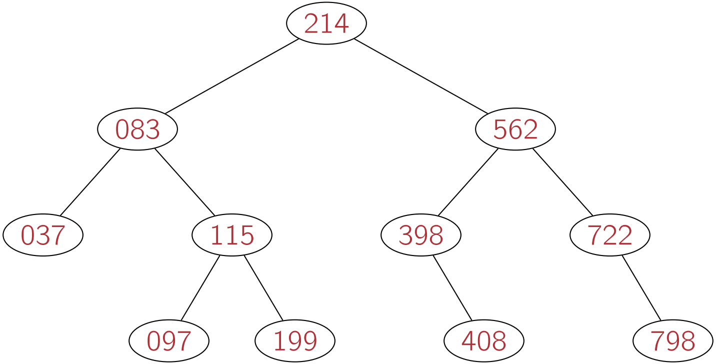
Let's search for 199 (Tse Kay).

Search for k starting from the roo

1. x is `null` — error
2. k == x`.id` — found
3. $k < x$`.id` — go left
4. $k > x$`.id` — go right



Searching in a binary search tree

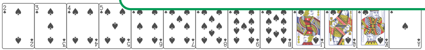


Search for k starting from the ro

1. x is `null` — error
2. k `==` x`.id` — found
3. $k < x$`.id` — go left
4. $k > x$`.id` — go right

How about searching for 336 (Joey)?

Running time of searching?



Summary

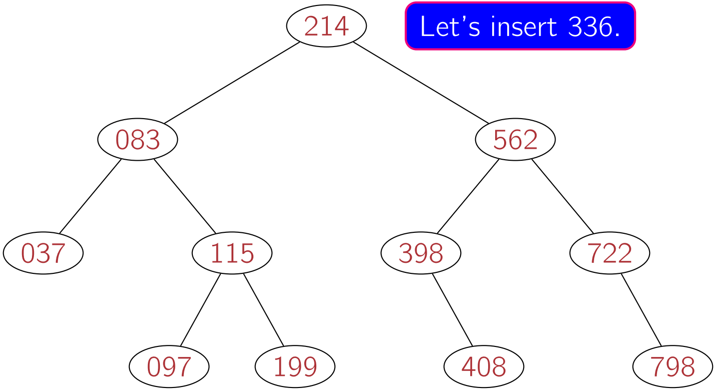
- The running time of `search` is $O(d)$, where d is the depth/height of the tree.
- But the worst case is $d = n - 1$, when the tree is skewed.
- The best case is $d = \lceil \log n \rceil$, when the tree is complete.
- It's hard to maintain a complete binary tree, after inserting and deleting nodes.
- Binary trees with depth $O(\log n)$ are considered balanced: there is balance between the number of nodes in the left subtree and the number of nodes in the right subtree of each node. ( )

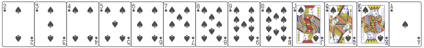
Which one is harder: Insertion or deletion?

Creation and Maintenance

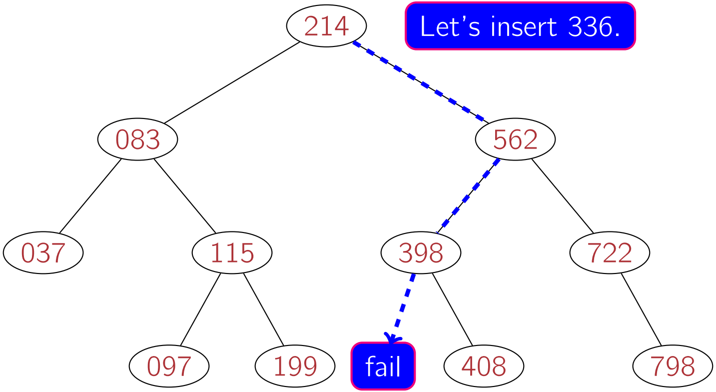
Inserting into a binary search tree

Let's insert 336.



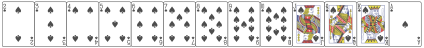


Inserting into a binary search tree

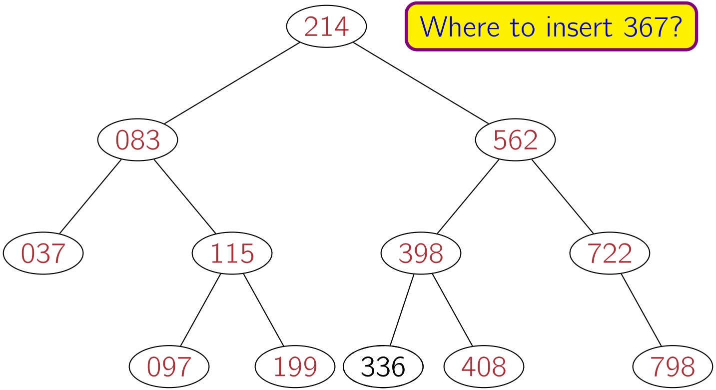


Let's insert 336.

The first step is similar as search.

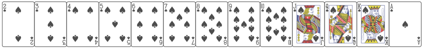


Inserting into a binary search tree

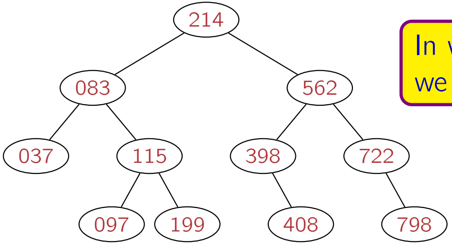


Where to insert 367?

Running time of insertion?

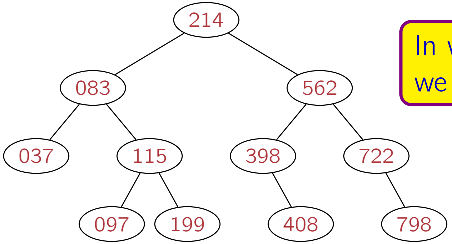


A question



In what order of insertion we obtain this tree?

A question

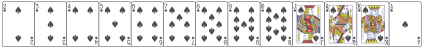


In what order of insertion we obtain this tree?

Is there another order?

```
BinarySearchTree<Student> tree =
    new BinarySearchTree<Student>();
int[] ids = {214,562,83,115,97,722,398,798,408,199,37};
String[] names = {"Chan␣Eason", "Cheung␣Jacky", "Winnie",
    "Ho␣Denise", "Mickey", "Leung␣Gigi", "Joey␣Yung",
    "Teddy", "Peppa", "Tse␣Kay", "Andy␣Lau"};
for (int i = 0; i < 11; i++)
    tree.insert(ids[i], new Student(ids[i], names[i]));
```

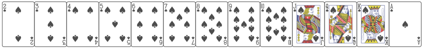
Any order in which no node is earlier than its parent.



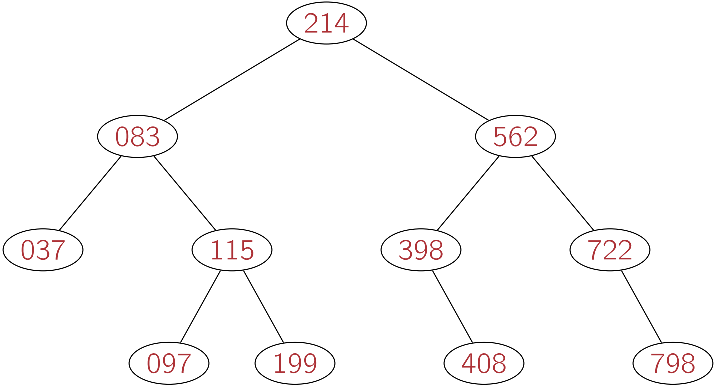
Tree sort

- Insert all students and do inorder traversal.
- What's the output?
- What's the running time?
- The best case: $O(n \log n)$.
- The worst case: $O(n^2)$.
- Again, the worst cases are sorted arrays, ending with skew trees.

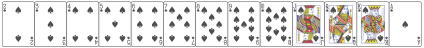
- We've learned a new sorting algorithm!?
- No. It's an worse version of quicksort. The order of insertion is the order of pivots.



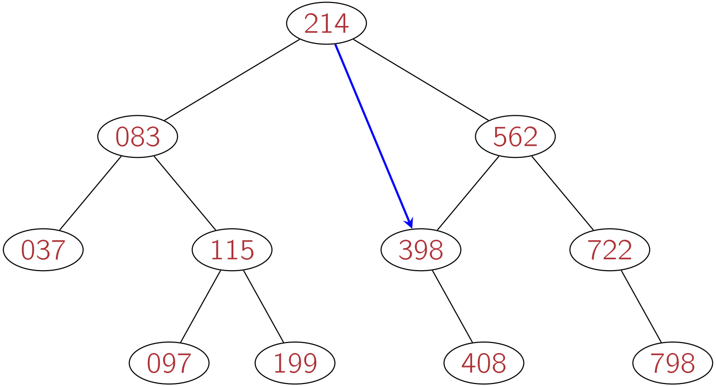
Finding successor in a binary search tree



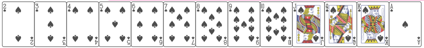
Which node has the min/max key?
How to find them (*recursion* or *iteration*?)



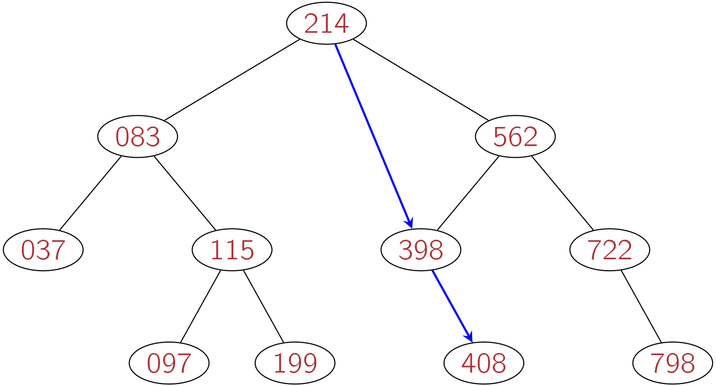
Finding successor in a binary search tree



The successor of a node x is the node whose key immediately follows its.

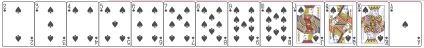


Finding successor in a binary search tree

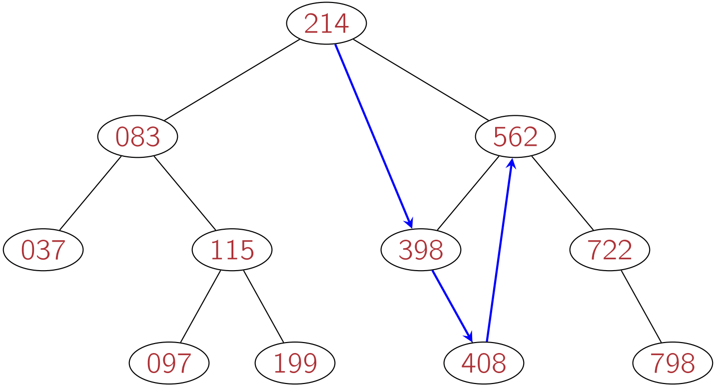


The successor of a node x is the node whose key immediately follows its.

- If x has a right child, the minimum node of its right subtree.
- Otherwise, on the path from the root to x, the first node on which we turn left.

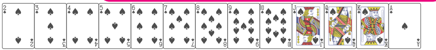


Finding successor in a binary search tree

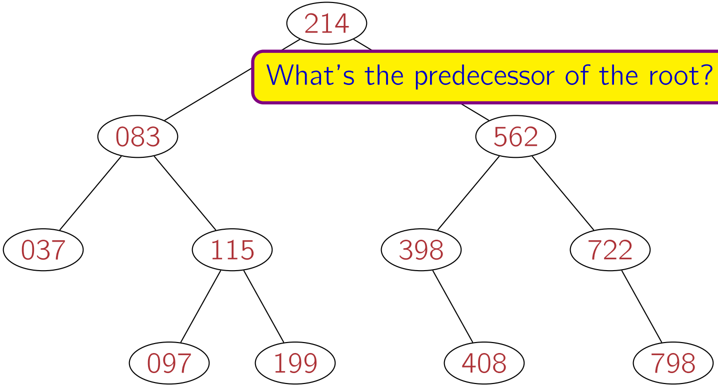


The successor of a node x is the node whose key immediately follows its.

- If x has a right child, the minimum node of its right subtree.
- Otherwise, on the path from the root to x, the first node on which we turn left.



Finding successor in a binary search tree



What's the predecessor of the root?

The predecessor of a node is the node whose key immediately precedes its.

Finding successor/predecessor

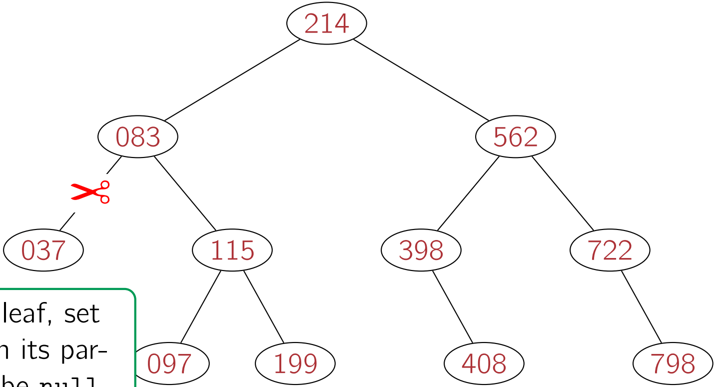
The successor of node x

- If x has a right child, then the successor of x is the *minimum* in the *right* subtree: follow x's *right* pointer, then follow *left* pointers until there are no more.
- If x does not have a right child, then find the lowest ancestor of x whose left child is also an ancestor of x: when searching for x, record the last node whose *left* subtree is used.

The predecessor of node x symmetric to successor

- If x has a left child, then the successor of x is the *maximum* in the *left* subtree: follow x's *left* pointer, then follow *right* pointers until there are no more.
- If x does not have a left child, then find the lowest ancestor of x whose right child is also an ancestor of x: when searching for x, record the last node whose *right* subtree is used.

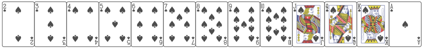
Deleting a node from a binary search tree



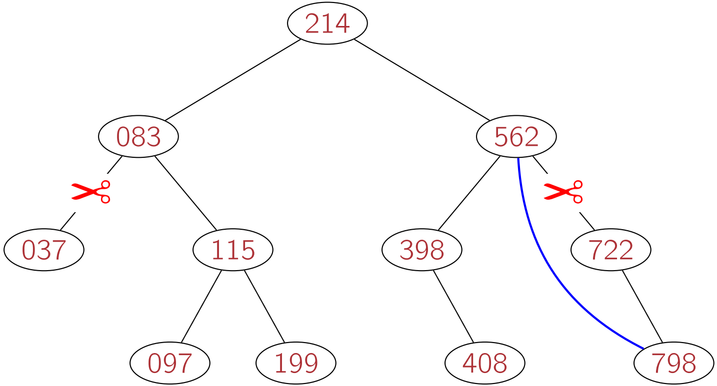
To delete a leaf, set the link from its parent to it to be `null`.

How to delete node 722?

Simply removing a non-leaf node breaks the tree.

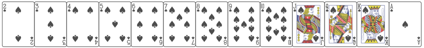


Deleting a node from a binary search tree

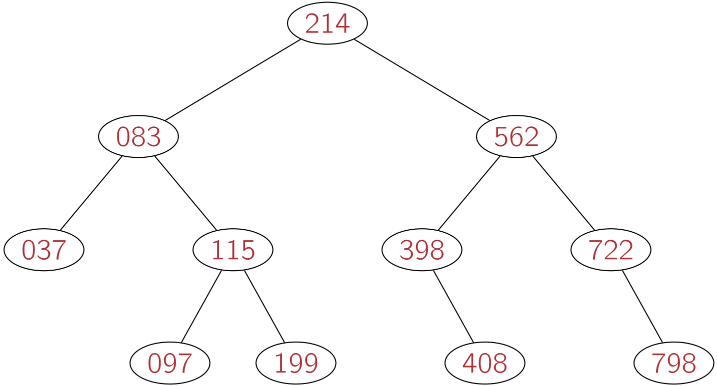


How to delete node 722?

To delete a node x with one child y, set the link from x's parent to x to y.

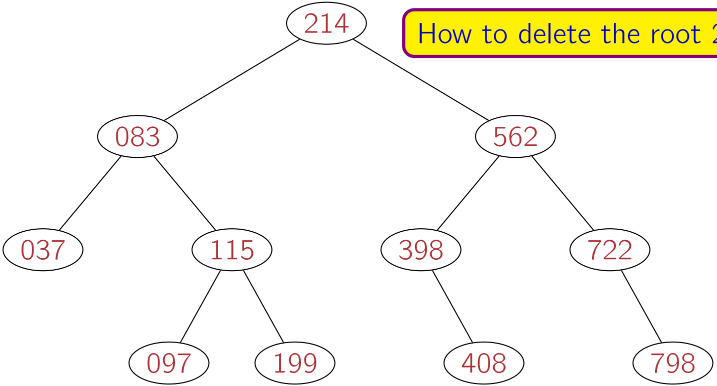


Deleting a node from a binary search tree



How to delete node 083 or node 562?

Deleting a node from a binary search tree

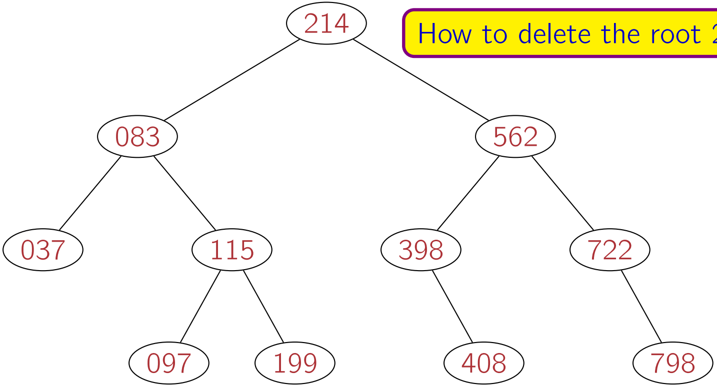


How to delete the root 214?

How to delete node 083 or node 562?

- To delete 083, we can put 037 in its place.
- Which one can be used to replace 562?

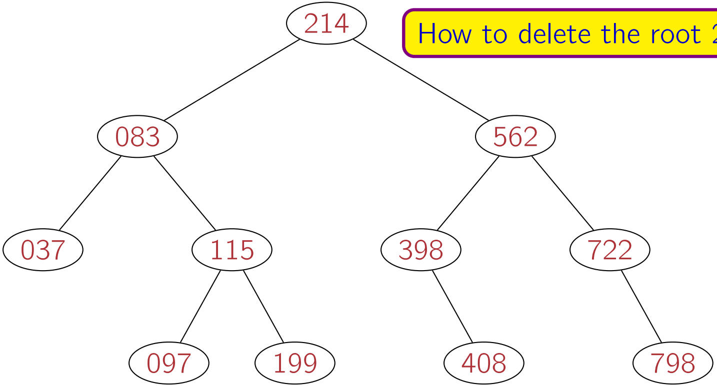
Deleting a node from a binary search tree



How to delete the root 214?

Unless we want to move elements from both sides, the new root has to be either predecessor 199 or successor 398.

Deleting a node from a binary search tree

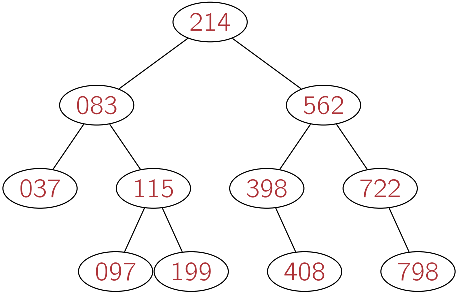


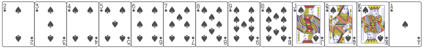
How to delete the root 214?

- If x has a left child, then the predecessor of x is in its left subtree.
- right successor right.
- If x isn't a leaf, find its predecessor or successor y to replace it.
- To find another node to replace y (in a smaller subtree), we do recursion!

Deletion of node x

- x has no children: set the child field in its parent to null.
- x has one child: set the child field in its parent to point to its child.
- x has two children: replace x with its successor y (minimum of x's right subtree).
  - easy if the right child of x has no left child; 115 and 562
  - otherwise, we need to find some node to replace y. 83 and 214
-

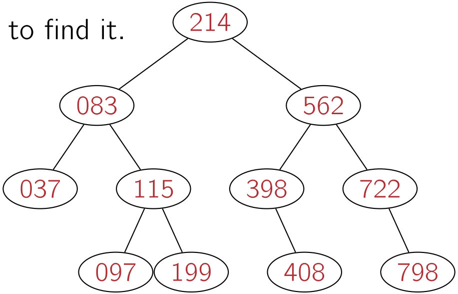


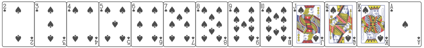


Deletion of node x

- x has no children: set the child field in its parent to null.
- x has one child: set the child field in its parent to point to its child.
- x has two children: replace x with its successor y (minimum of x's right subtree).
  - easy if the right child of x has no left child; 115 and 562
  - otherwise, we need to find some node to replace y. 83 and 214

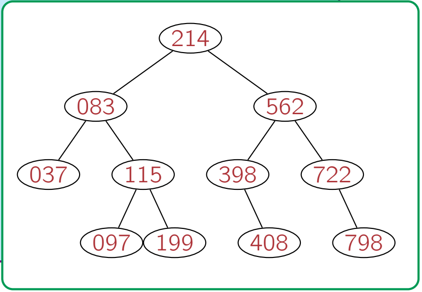
- To delete the node with key k, we need to find it.





The codes for deletion

```
// start by calling root = delete(root, key);
Node<T> delete(Node<T> x, int key) {
    if (x == null) return null;
    if (key < x.key) x.lC = delete(x.lC, key);
    else if (key > x.key) x.rC = delete(x.rC, key);
    else {              // x is the node to be deleted.
      if (x.rC == null) return x.lC;
      if (x.lC  == null) return x.rC;
      Node<T> t = x;
      x = recFindMin(t.rC);
      x.rC = deleteMin(t.rC);
      x.lC = t.lC;
    }
    return x;
}
```



Summary

- the tree has only one variable: the `root`
- each node has a `key`, a `data`, and two references: `leftChild` and `rightChild`.
- traversal: inorder, preorder, postorder, level-wise.
- essential operations: `search`, `insert`, `delete`
- easy operations: `findMin`, `findMax`
- nontrivial operations: `successor`, `predecessor`
- all these operations have running time $O(n)$ in the worst case.
- we can make them $O(\log n)$, by maintaining the tree *balanced*.

- Although it's more common to use references, we can store a tree as an array,

Lecture 9: Heaps