ECE 264 Spring 2023
Advanced C Programming

Aravind Machiry
Purdue University

Midterm 2

- When: Thursday (9th March)
- How: Online via Brightspace for 24 hours from 7:30 am (9th) to 7:29 am (10th)
- Time : 3 hours (Expected to be done in 1 hour).
- Questions similar to quiz but expect some code to be understood or written.

Topics for Midterm 2

- Compilation and Makefile
- Heap
- GDB
- Structures

Pointers

- How addresses are calculated for array accesses?
  - arr[i]?

- Size of pointers to different types?
  - int arr[10];
    - ❑ sizeof(arr)?
  - struct vector x
    - ❑ sizeof(x)?

Files

- When can fopen return NULL?
- what does fread return?

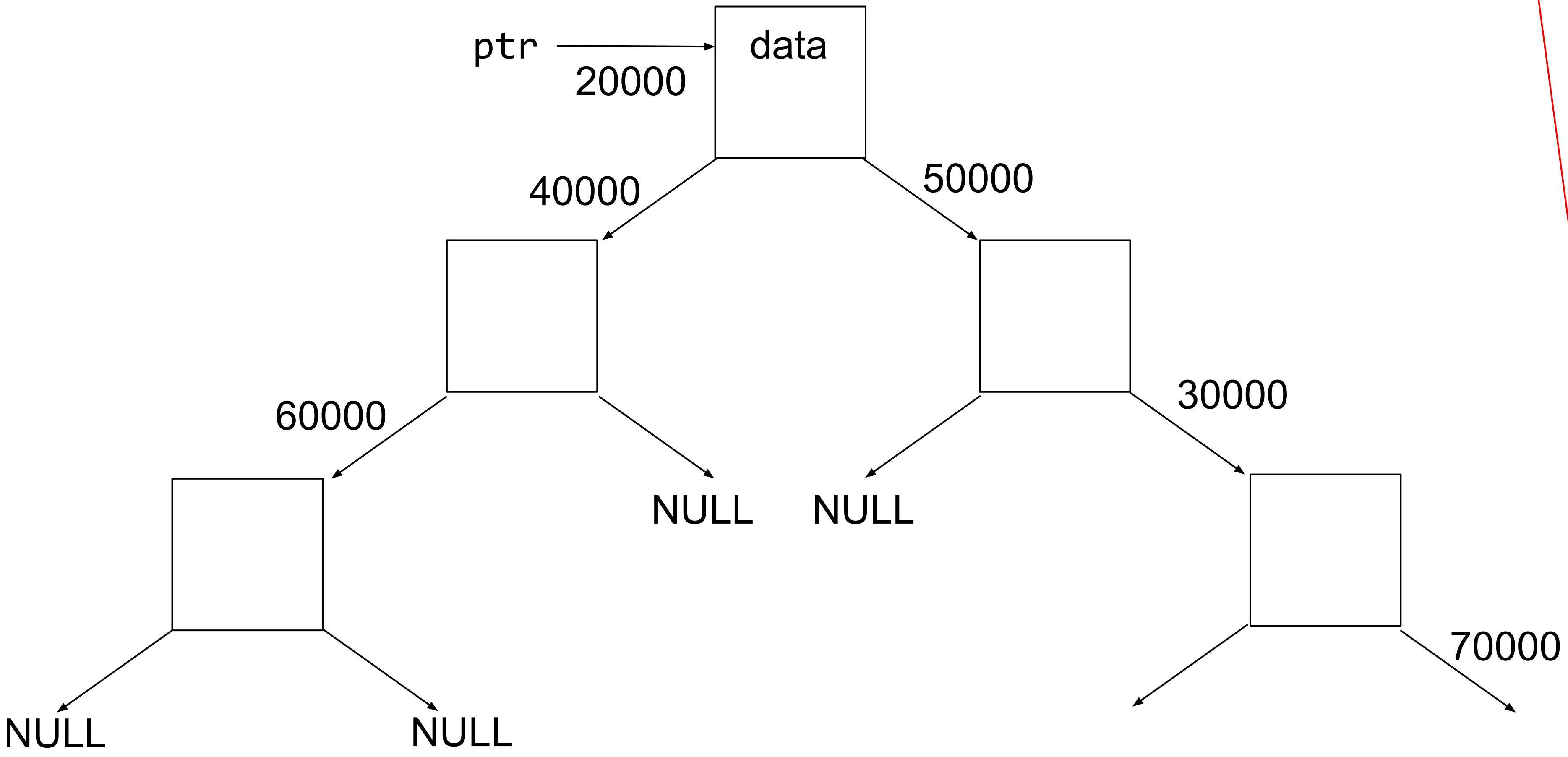
GDB

- Command to view all breakpoints
- How to put a breakpoint?
- How to print values?
- How to view call-stack?
- GDB Cheat sheet: https://purs3lab.github.io/ece264/static_files/read/reference_sheet.pdf

Binary Tree

Binary tree (Review)

Each piece of memory has two pointers



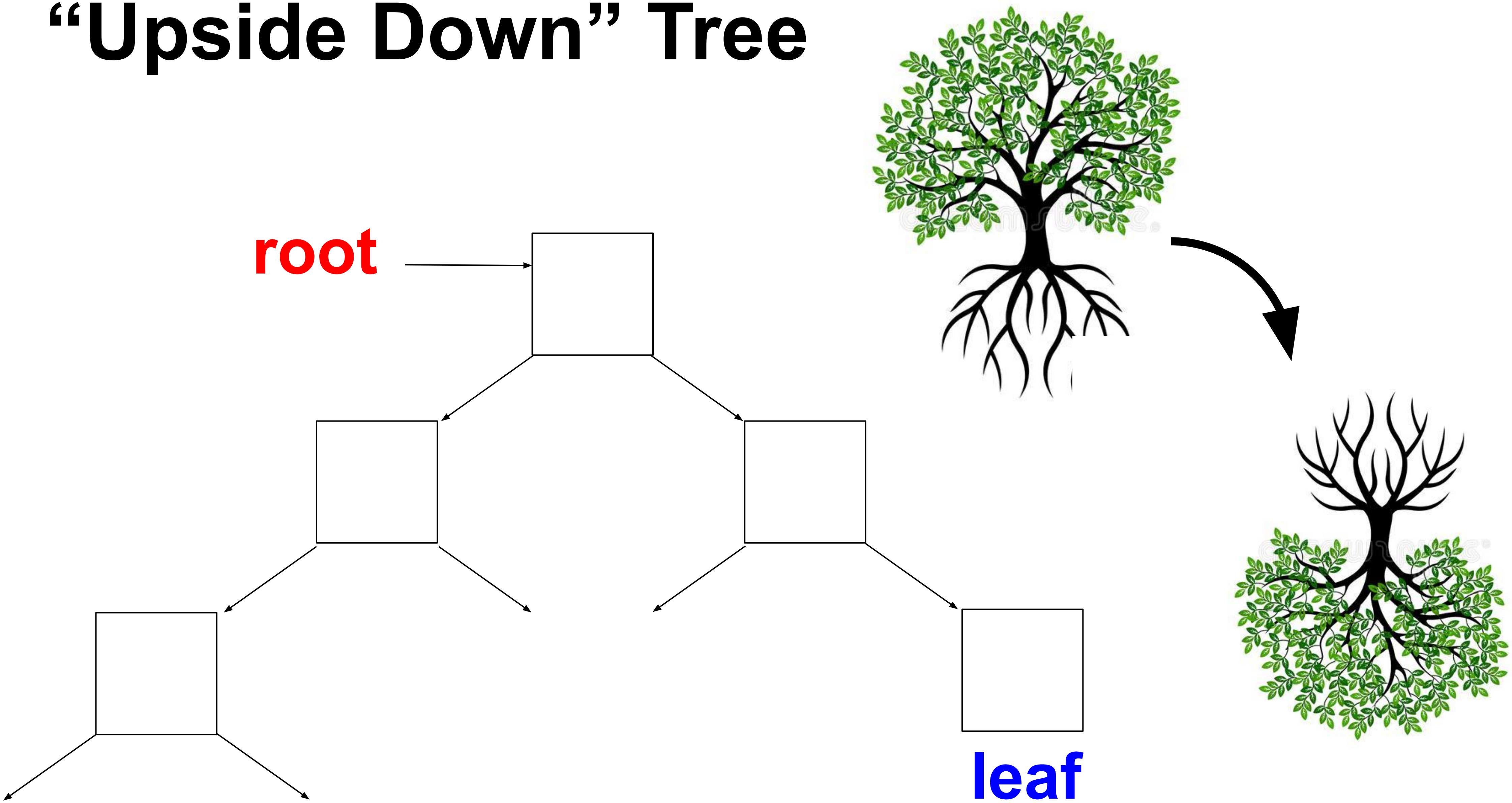
Stack Memory

	Address	Value
ptr	100	20000

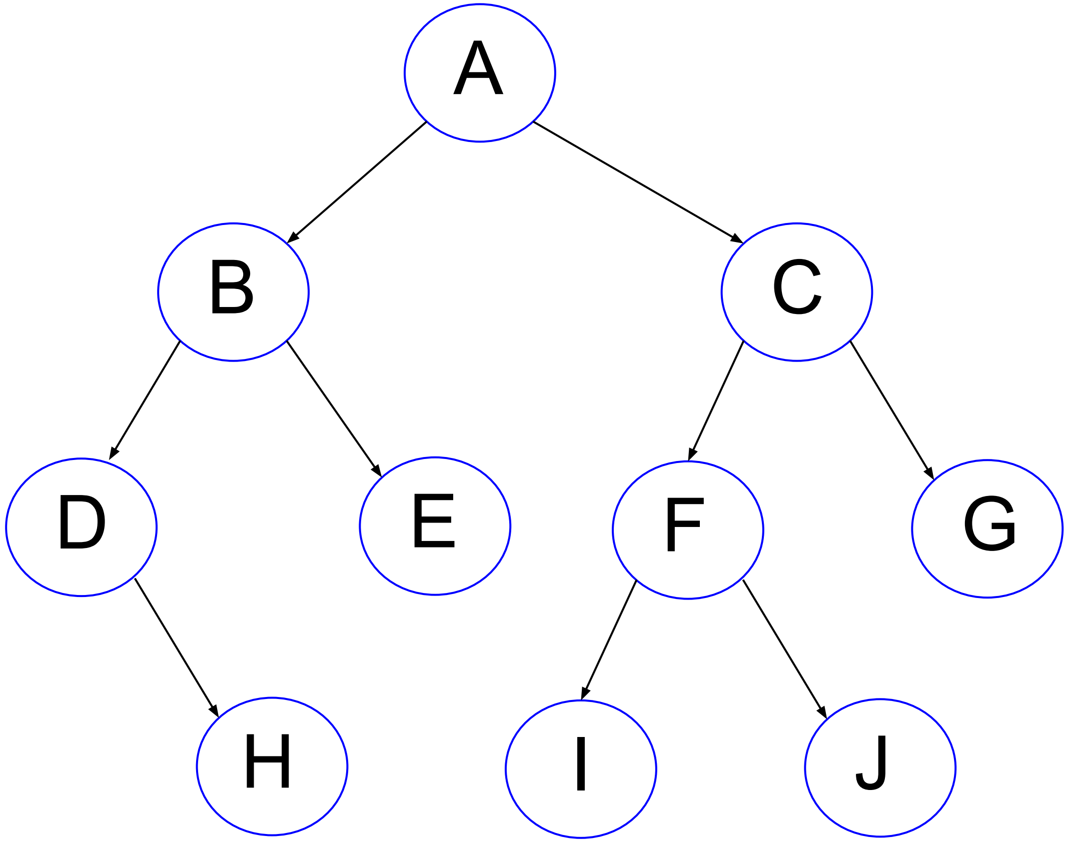
Heap Memory

Address	Value
	data
	NULL
70000	NULL
	data
	NULL
60000	NULL
	data
	30000
50000	NULL
	data
	NULL
40000	60000
	data
	70000
30000	NULL
	data
	50000
20000	40000

“Upside Down” Tree

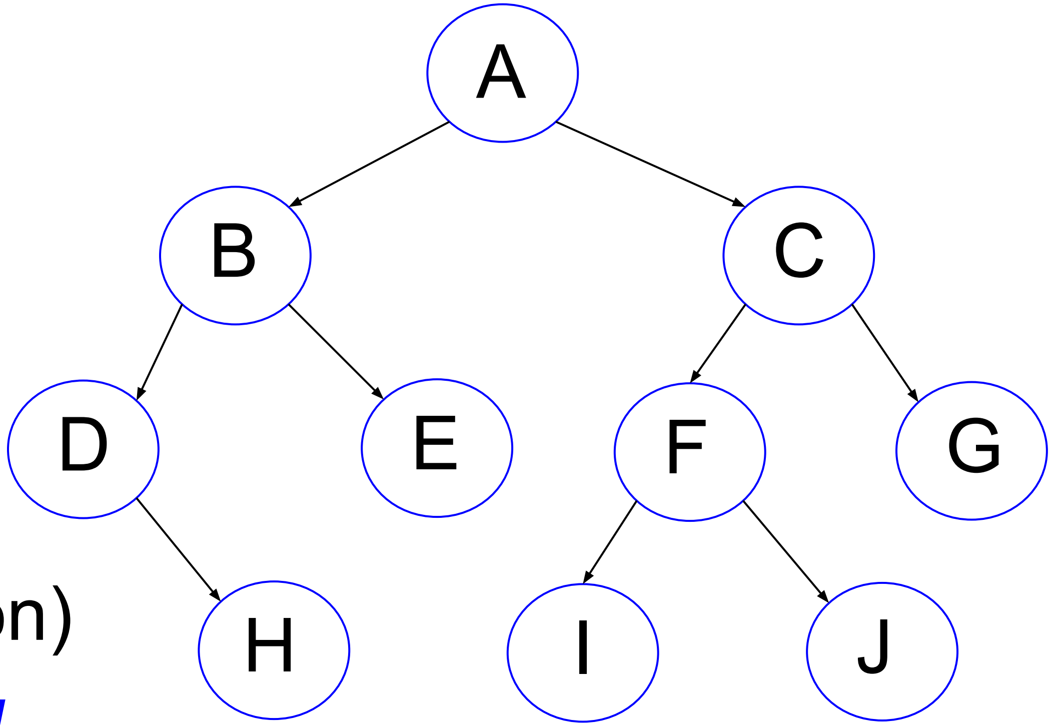


Terminology

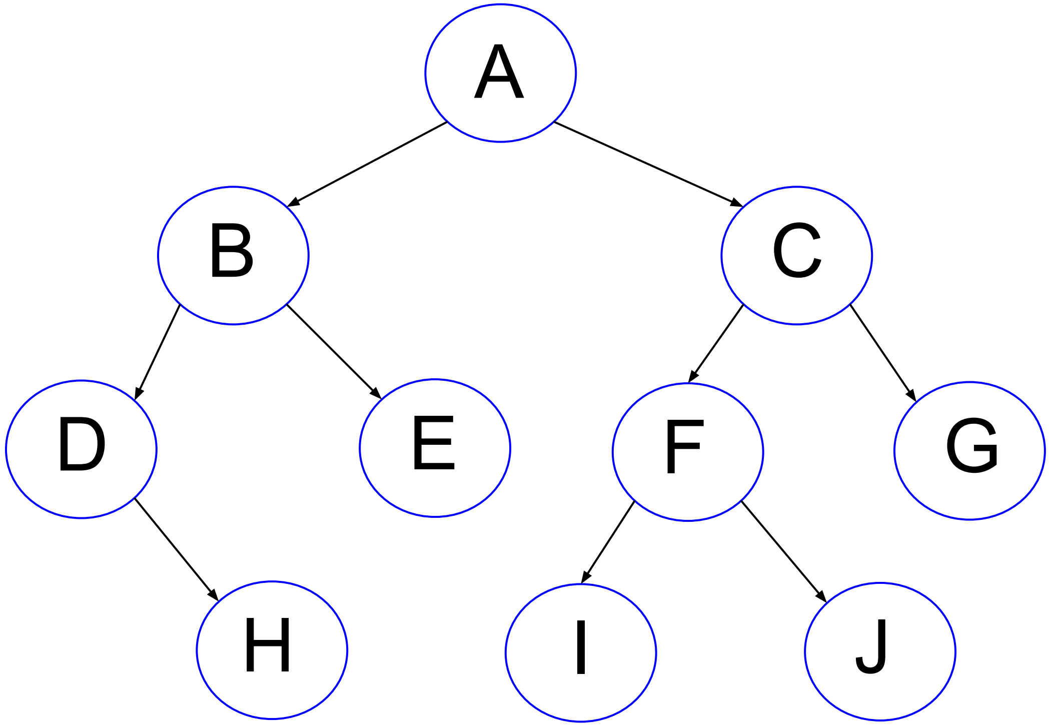


- A, B, C … : each is a ***node***
- An ***edge*** connects from A to B
- Do not draw edges point to NULL
- A is the ***parent*** node of B and C
- B and C are A's ***child*** nodes
- B and C are ***siblings***
- ***Binary tree***: each node has at most two children
- If a node has no parent node, this node is the tree's ***root***
- If a node has no child node, this is a ***leaf*** node

Terminology



- If A is B’s parent, A is B’s ***ancestor***.
- If A is B’s parent, B is D’s ancestor,
  A is D’s ancestor (recursive definition)
- If A is B’s ancestor, B is A’s ***offspring***.
- A ***path*** is the sequence of edges from an ancestor to an offspring.
- The ***height*** of a node is the length of the longest path to a leaf.
  The heights of E, D, B are 0, 1, 2 respectively.
- The ***height*** of a tree is the height of the root.
  The height of the tree is 3.

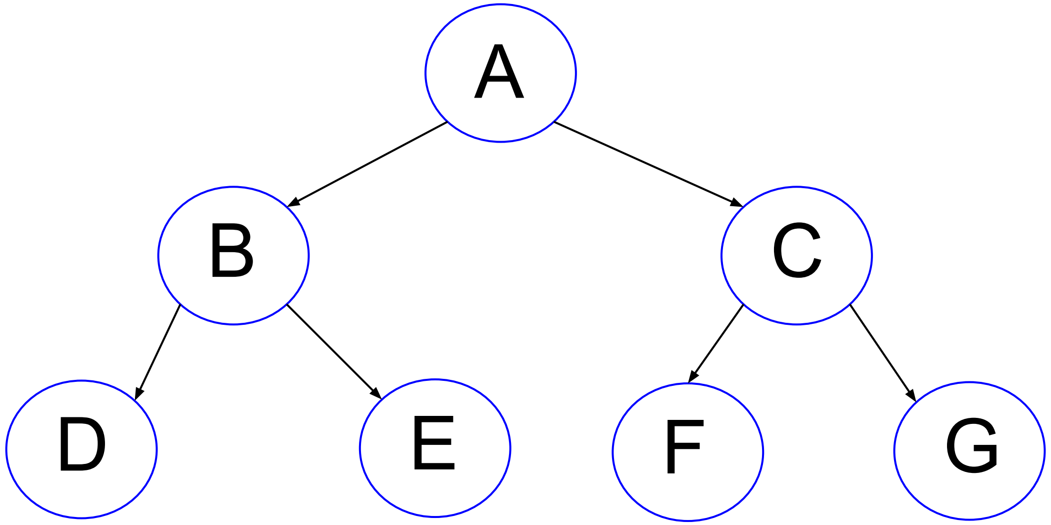


- The ***depth*** of a node is the distance to the root
- ***Full*** binary tree: nodes have 2 children or 0 child
- ***Perfect*** binary tree: full + leaf nodes of the same distance to root
- B and B’s offsprings are A’s left ***subtree***.
- C and C’s offsprings are A’s right subtree.

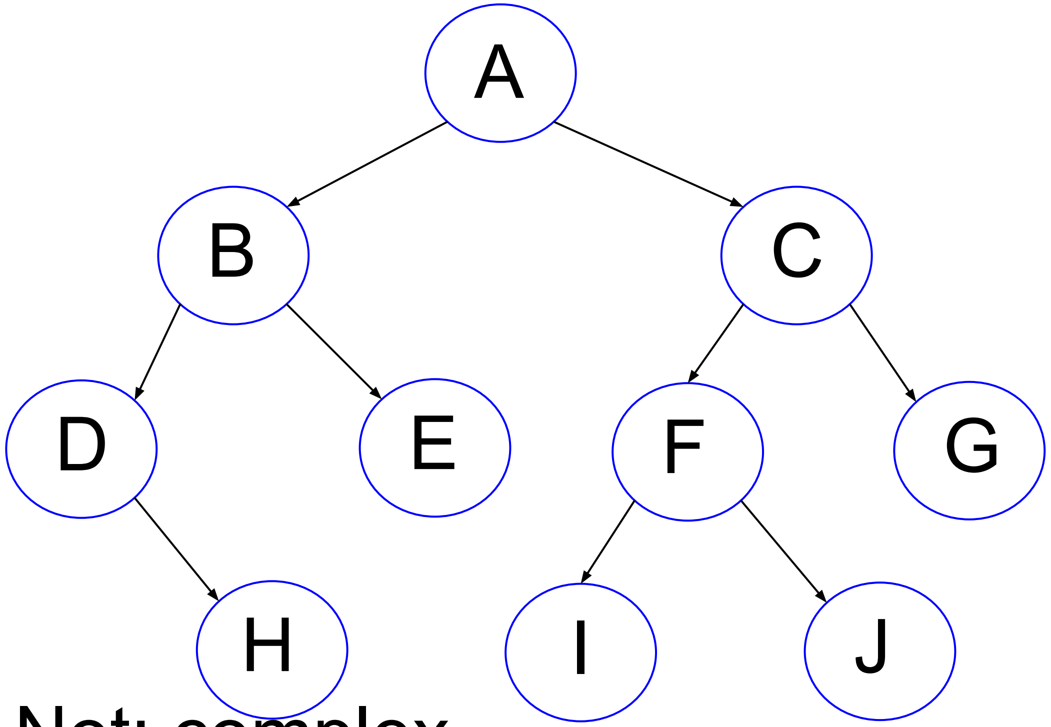
Why Are Binary Tree Important

- Power of logarithm: log(n) grows very slowly. 2^n grows very fast
- A perfect binary tree of height n has $2^{n+1} – 1$ nodes
- In a single step, a program decides to go left or right:

```
if (condition)
{
   go left
}
else
{
   go right
}
```

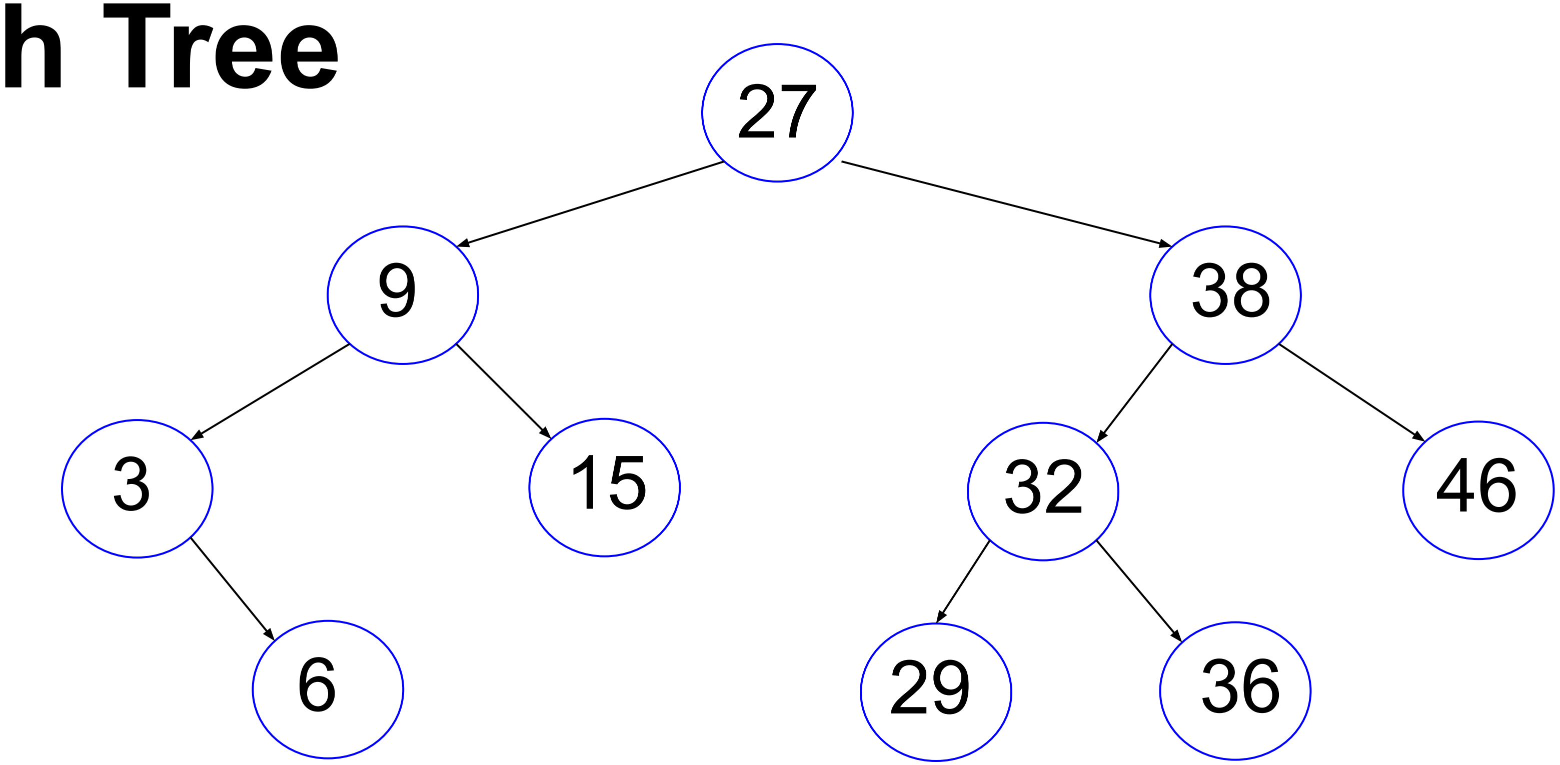


Binary Search Tree

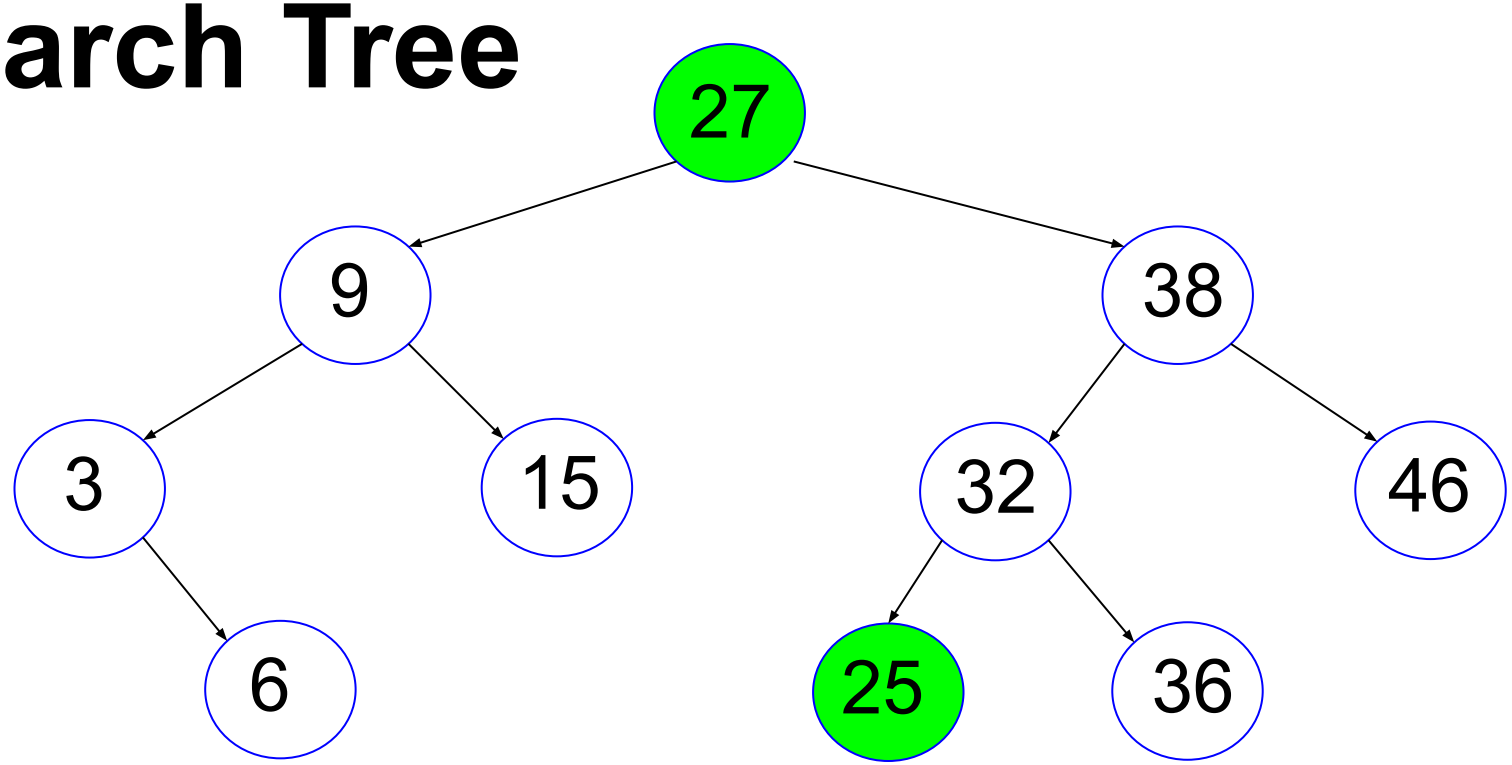


- Every node stores a value as the ***key***.
- The keys must be ***totally ordered***:
  - if a ≤ b and b ≤ a then a = b
  - if a ≤ b and b ≤ c then a ≤ c
  - either a ≤ b or b ≤ a
- Totally ordered: integer, real numbers. Not: complex.
- For ***every node***, the following is tree:
- Keys of all nodes of the left subtree of a node < this node's key
- Keys of all nodes of the right subtree of a node > this node's key

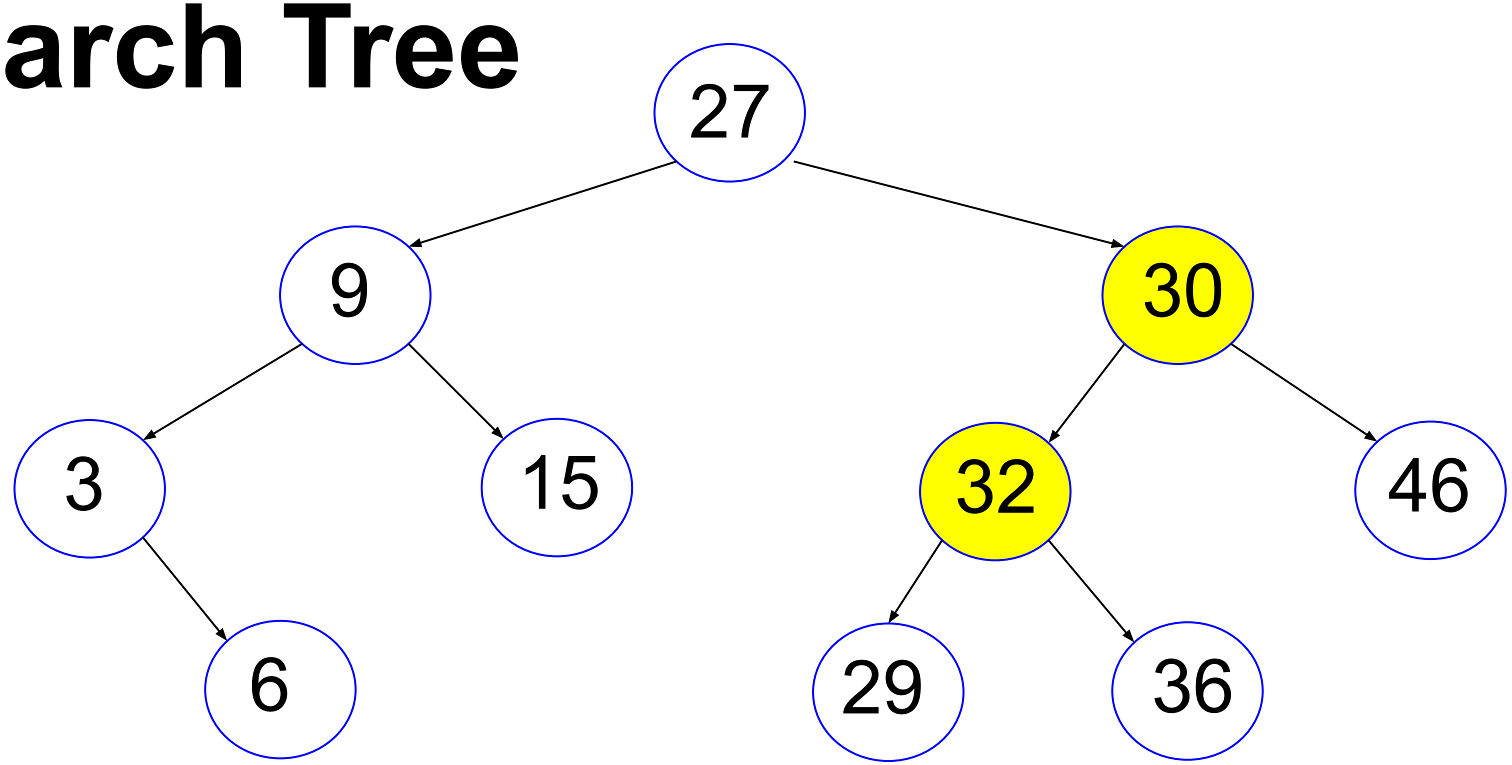
Binary Search Tree



Not Binary Search Tree



Not Binary Search Tree

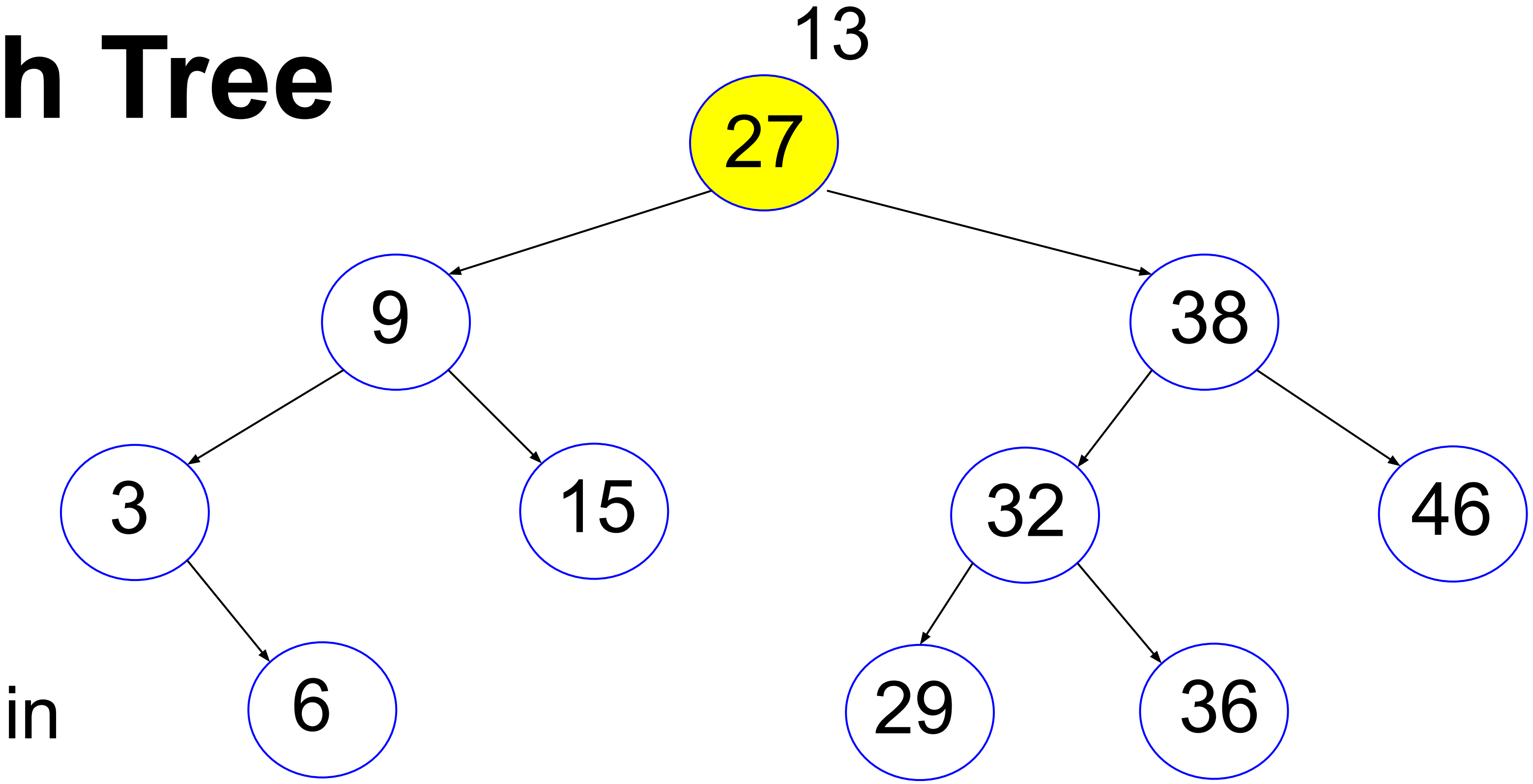


Binary Tree is a Container Structure

- insert: insert data
- delete: delete (a single piece of) data
- search: is a piece of data stored
- destroy: delete all data

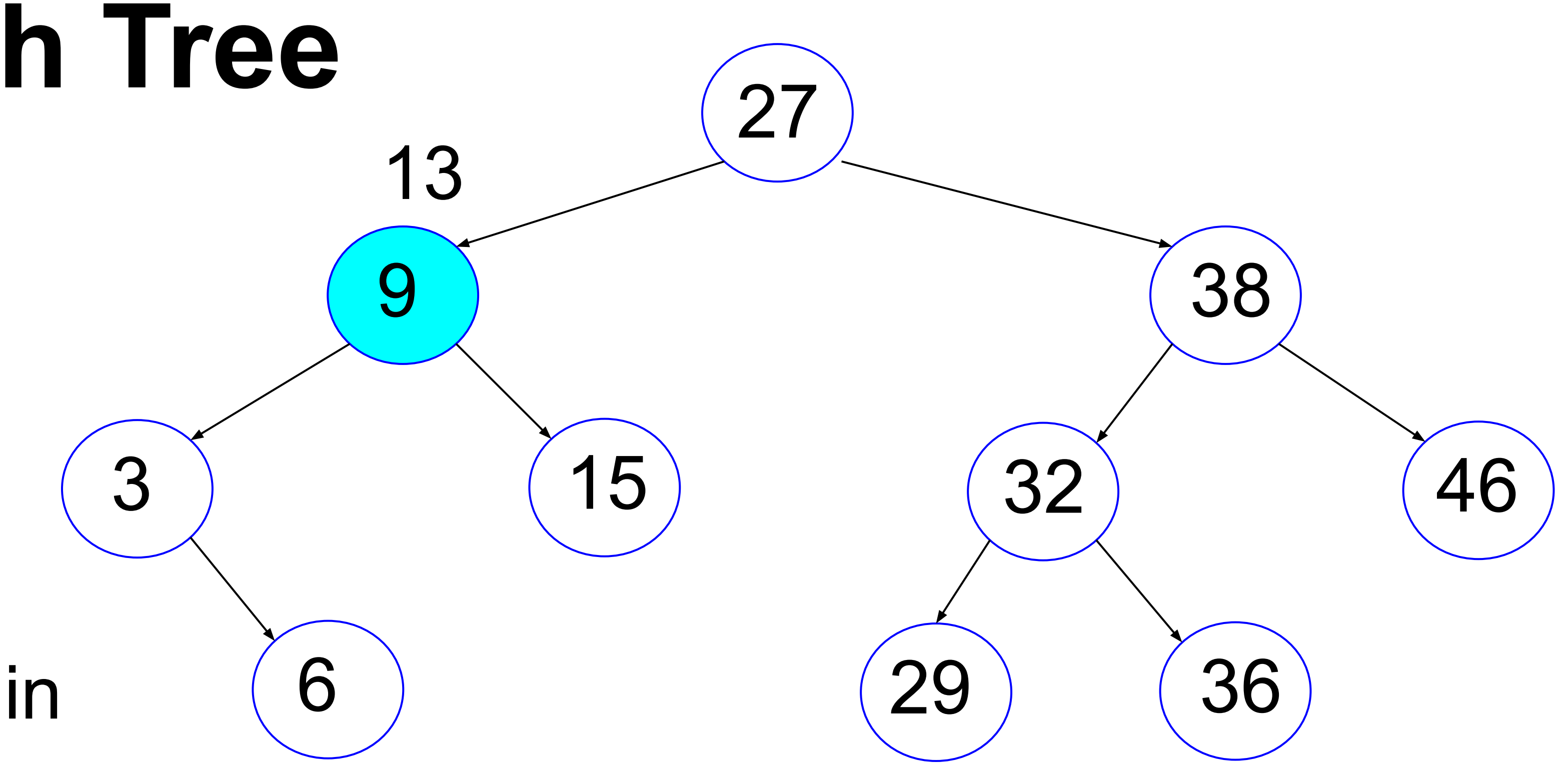
Binary Search Tree

- Is 13 stored?
- 13 < 27 ⇒ go left
- 13 > 9 ⇒ go right
- 13 > 15 ⇒ go left
- Nothing ⇒ 13 is not in



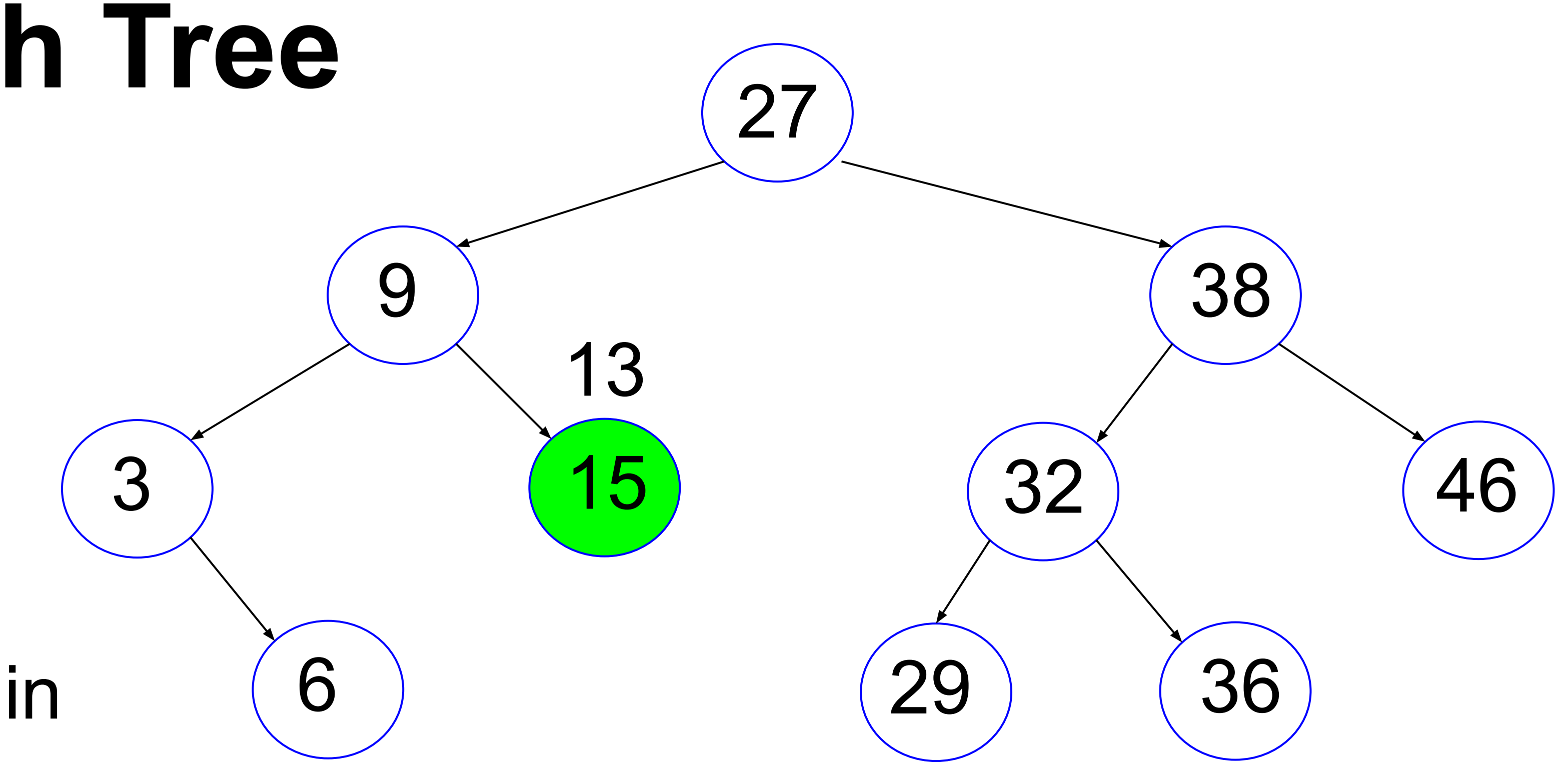
Binary Search Tree

- Is 13 stored?
- 13 < 27 ⇒ go left
- 13 > 9 ⇒ go right
- 13 > 15 ⇒ go left
- Nothing ⇒ 13 is not in



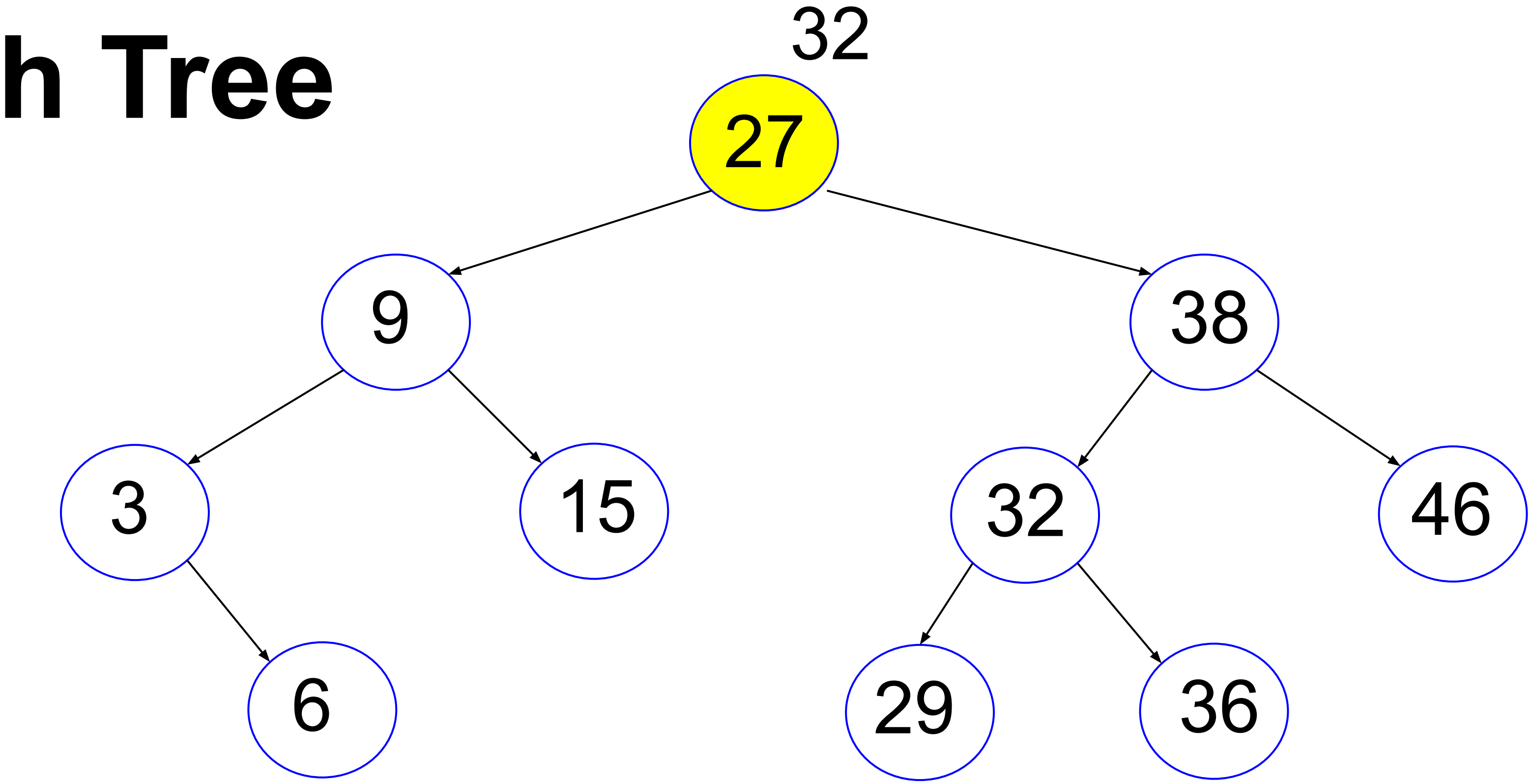
Binary Search Tree

- Is 13 stored?
- 13 < 27 ⇒ go left
- 13 > 9 ⇒ go right
- 13 > 15 ⇒ go left
- Nothing ⇒ 13 is not in



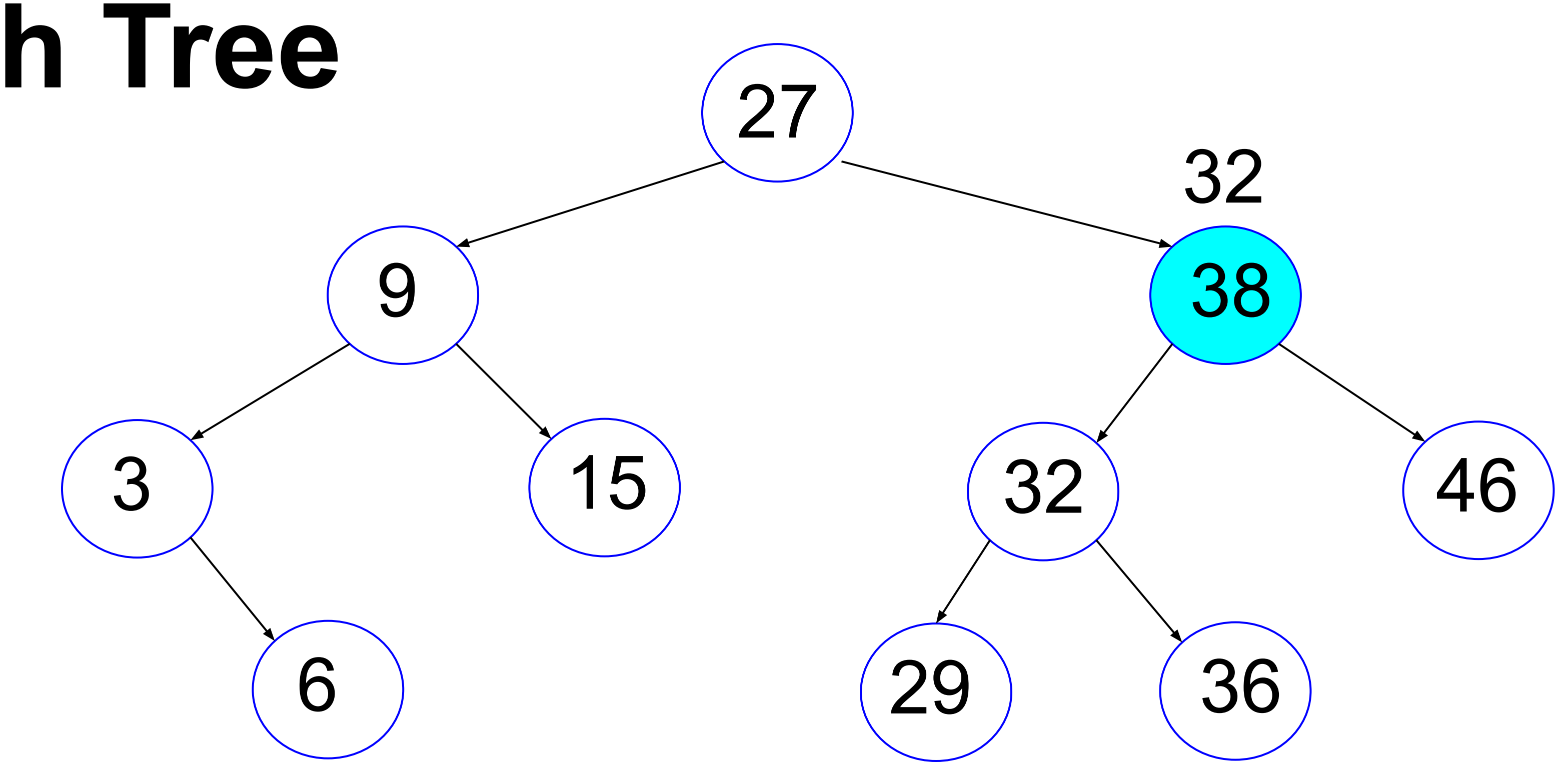
Binary Search Tree

- Is 32 stored?
- 32 > 27 ⇒ go right
- 32 < 38 ⇒ go left
- 32 = 32 ⇒ found



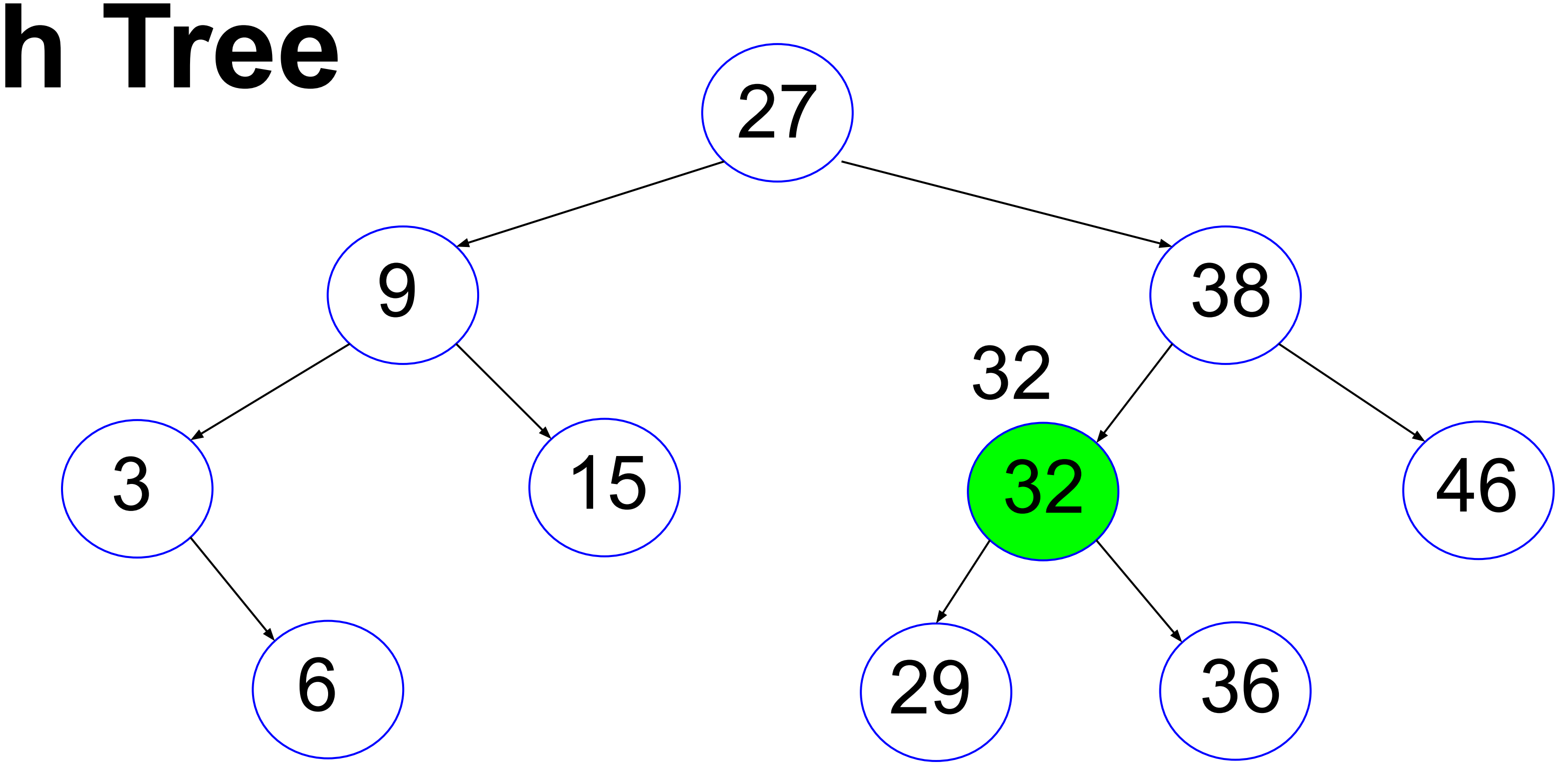
Binary Search Tree

- Is 32 stored?
- 32 > 27 ⇒ go right
- 32 < 38 ⇒ go left
- 32 = 32 ⇒ found



Binary Search Tree

- Is 32 stored?
- 32 > 27 ⇒ go right
- 32 < 38 ⇒ go left
- 32 = 32 ⇒ found



```
typedef struct tnode
{
  struct tnode * left;
  struct tnode * right;
  // data, must have a way to compare keys
  // may be a structure
  int value; // use int for simplicity
} TreeNode;
// search a value in a binary search tree starting
// with r, return the node whose value is v,
// or NULL if no such node exists
TreeNode * Tree_search(TreeNode * tn, int v);
TreeNode * Tree_search(TreeNode * tn, int val)
{
  if (tn == NULL) { return NULL; } // cannot find
  if (val == (tn -> value)) // found
    { return tn;}
  if (val < (tn -> value))
    {
      // search the left side
      return Tree_search(tn -> left, val);
    }
  return Tree_search(tn -> right, val);
}
```

three components of recursion:

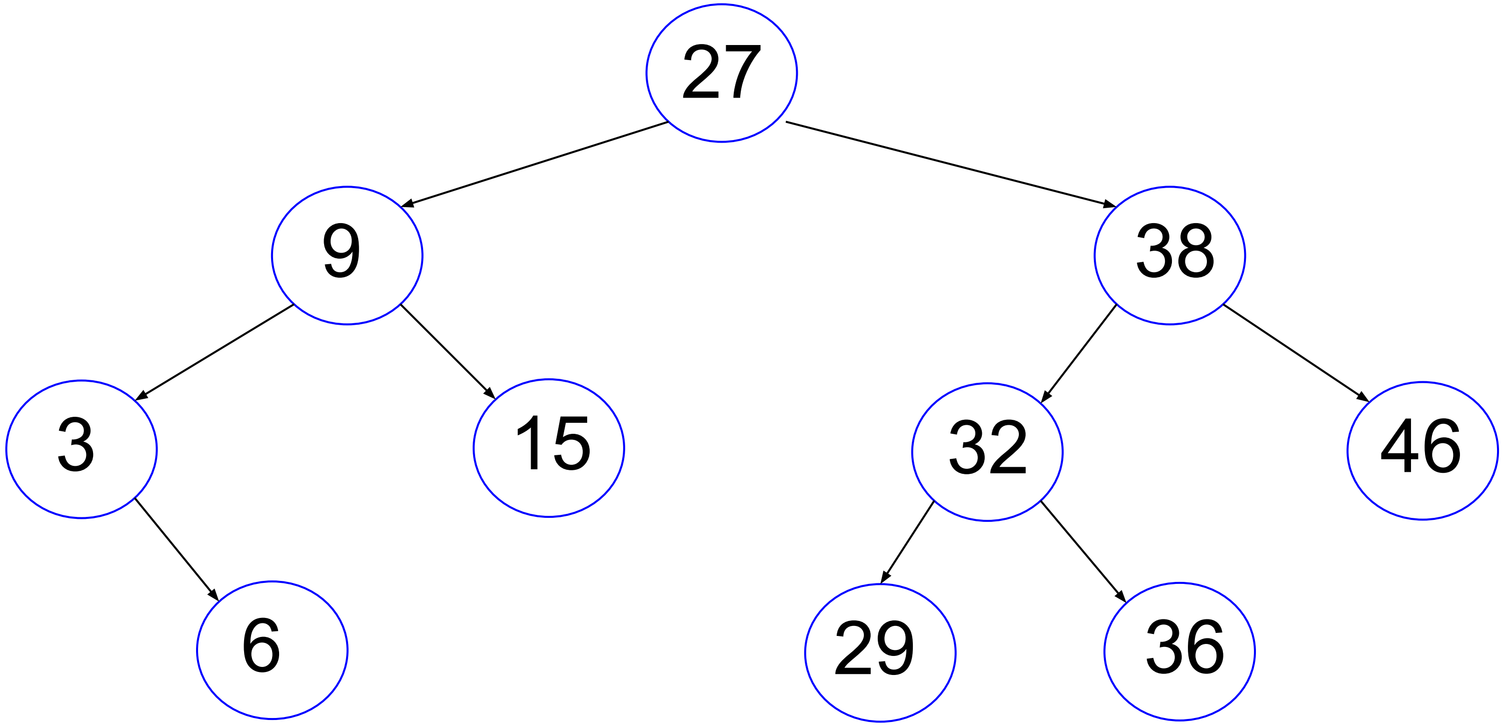
1. stop condition: NULL
2. change: go to child
3. recurring pattern: same method to search

Binary Tree Insert

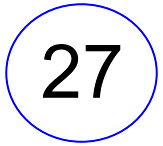
Binary Search Tree

How to create a tree like this?

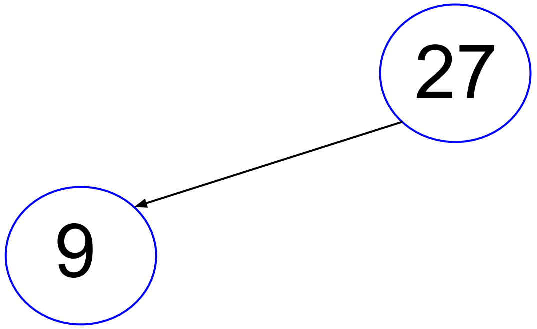
The insert function



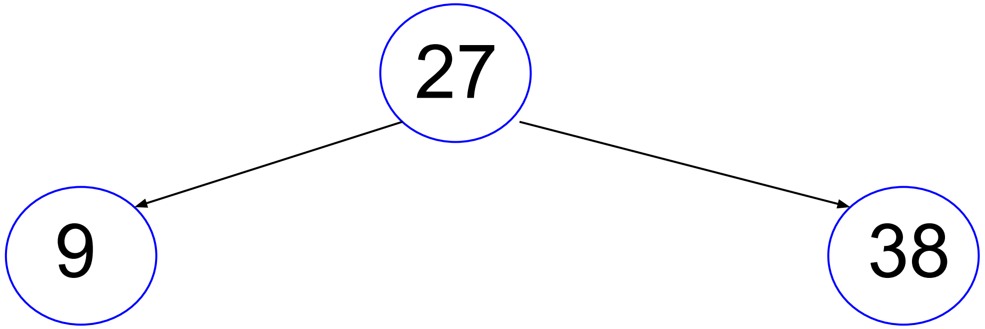
insert 27



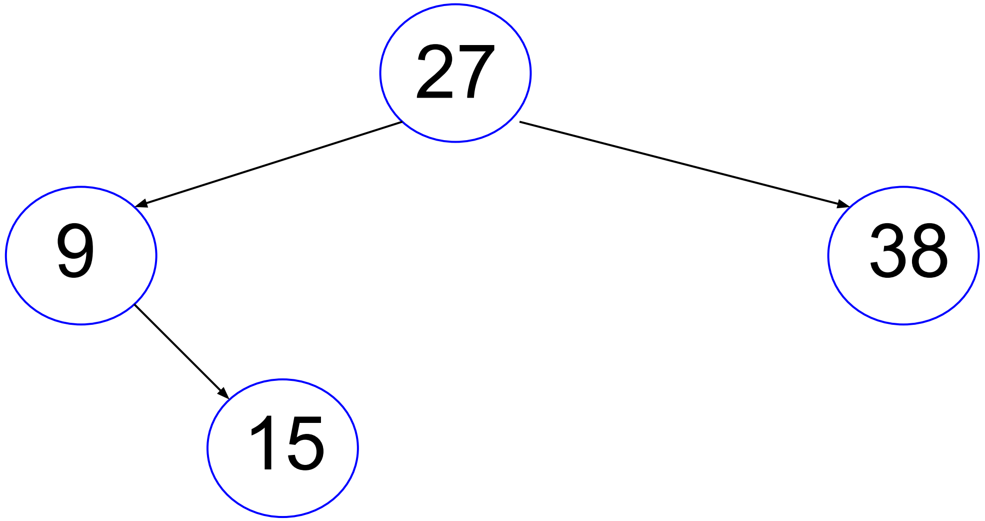
insert 27, 9



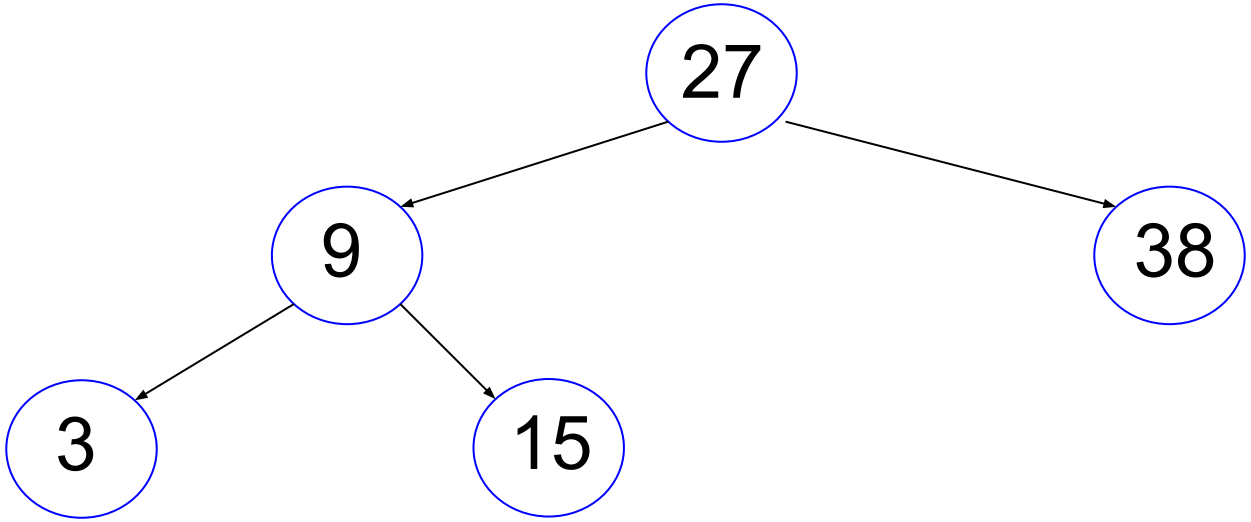
insert 27, 9, 38



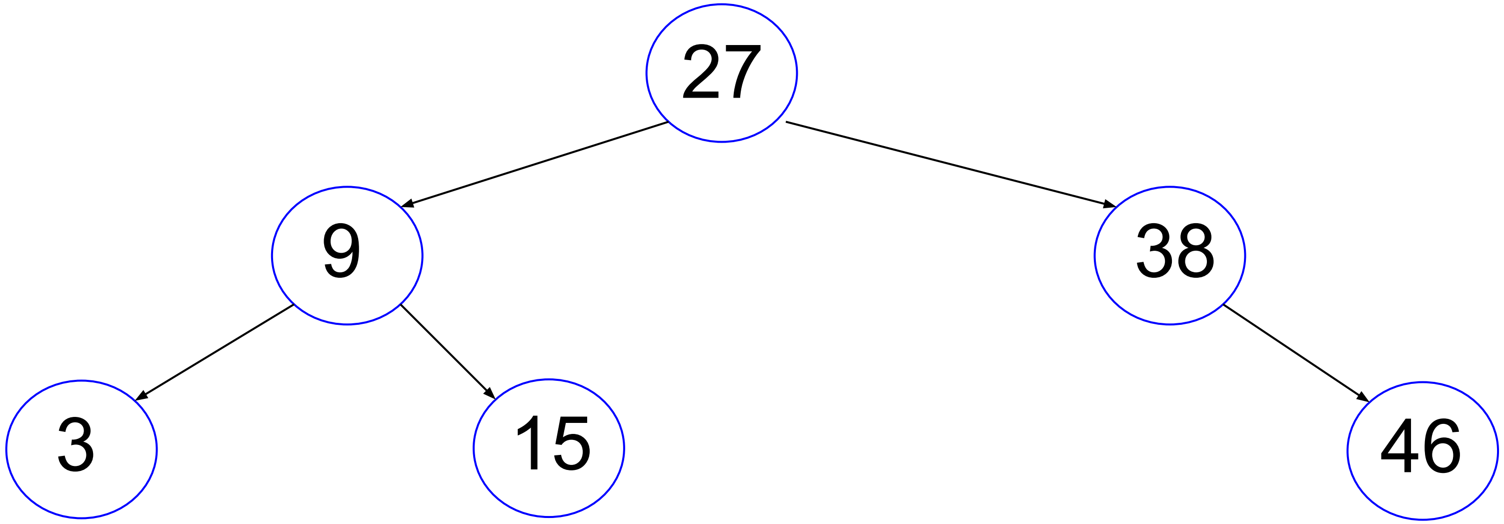
insert 27, 9, 38, 15



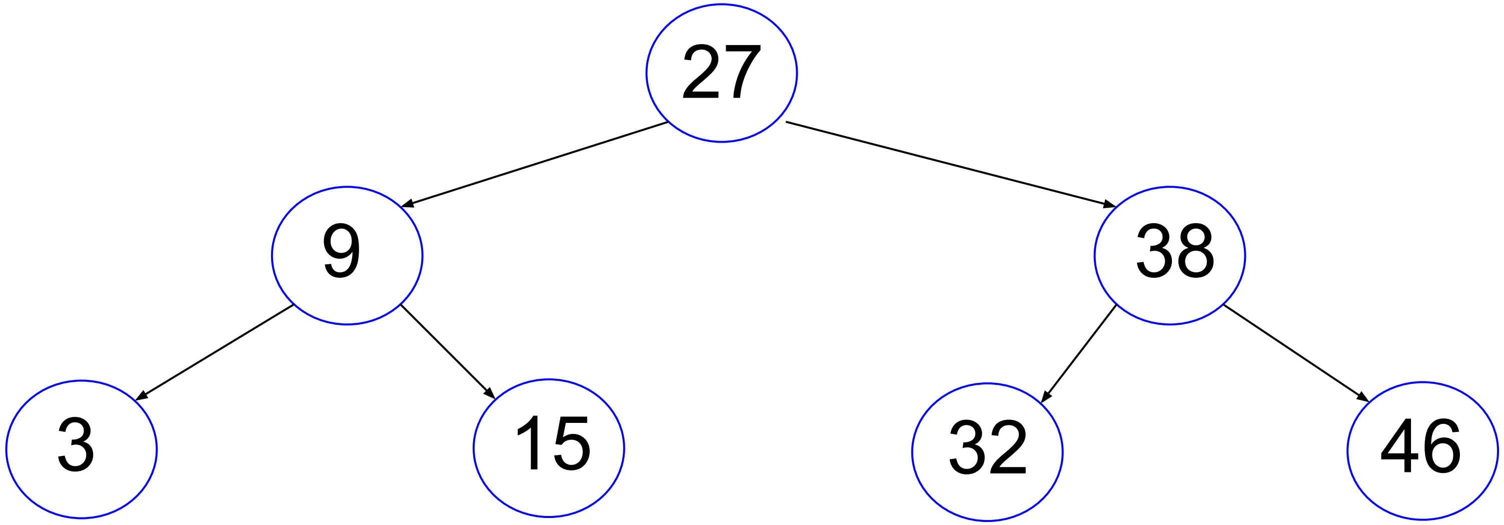
insert 27, 9, 38, 15, 3



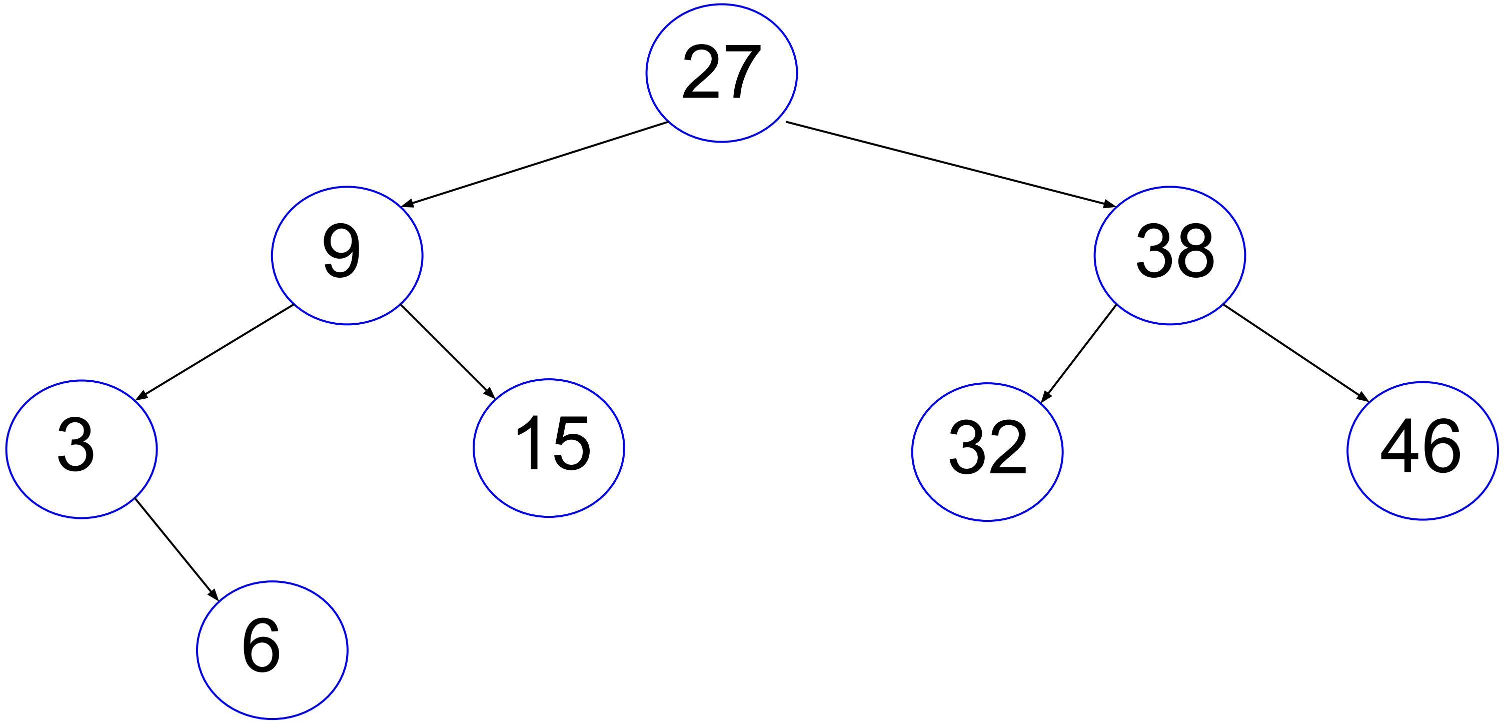
insert 27, 9, 38, 15, 3, 46



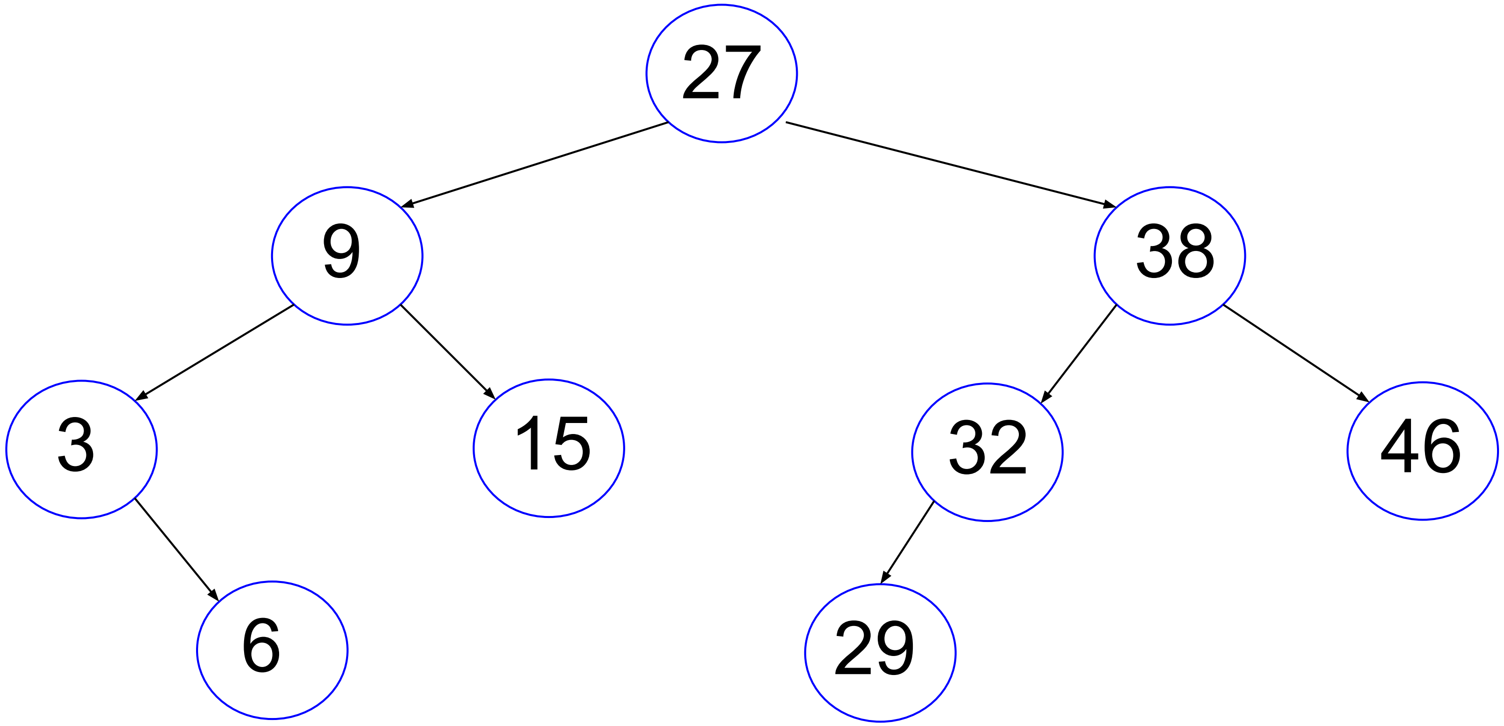
insert 27, 9, 38, 15, 3, 46, 32



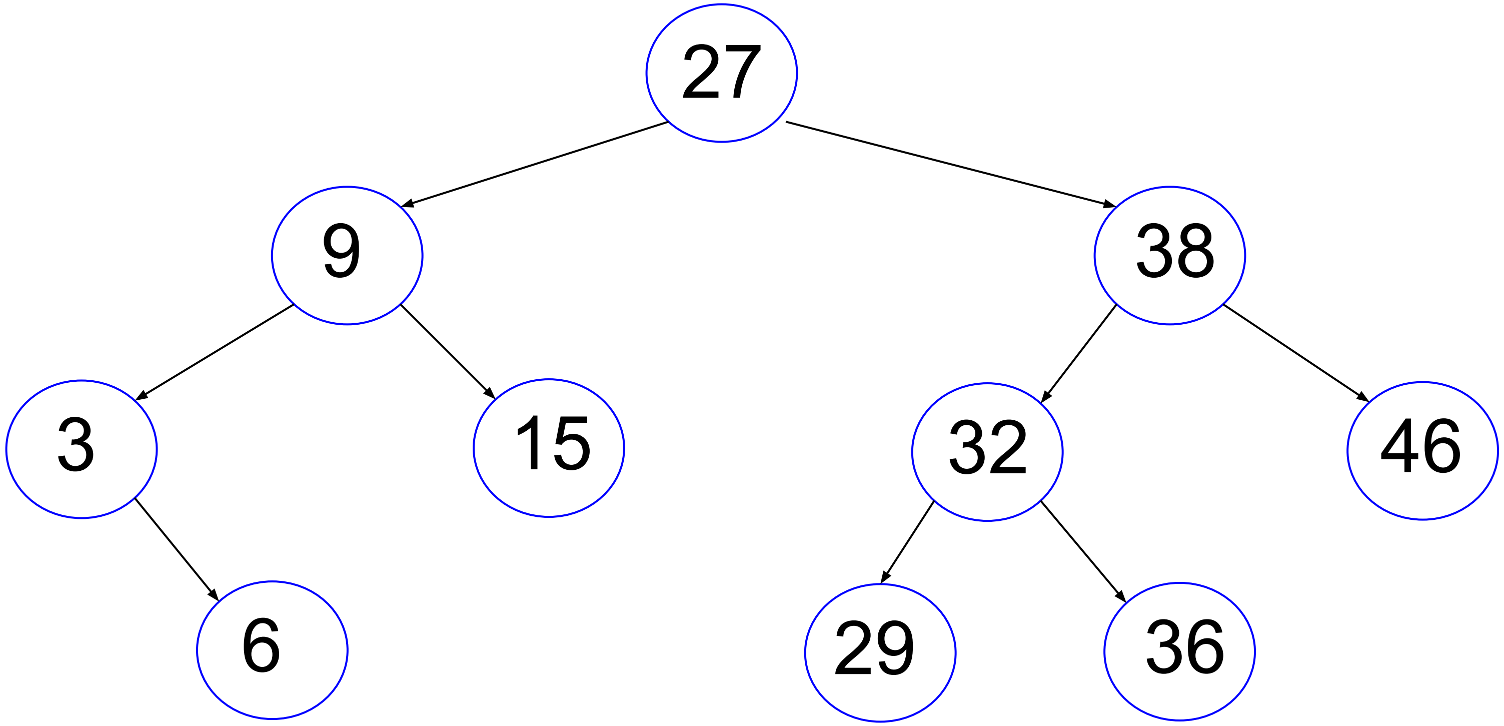
insert 27, 9, 38, 15, 3, 46, 32, 6



insert 27, 9, 38, 15, 3, 46, 32, 6, 29

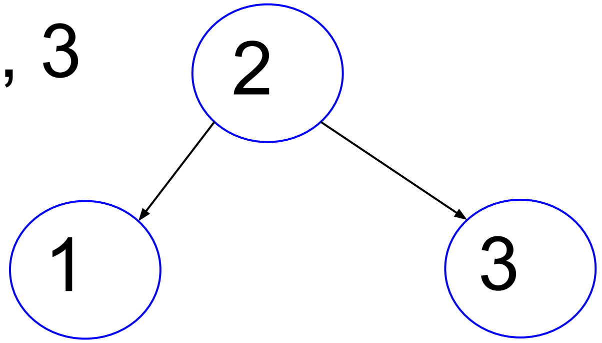


insert 27, 9, 38, 15, 3, 46, 32, 6, 29, 36

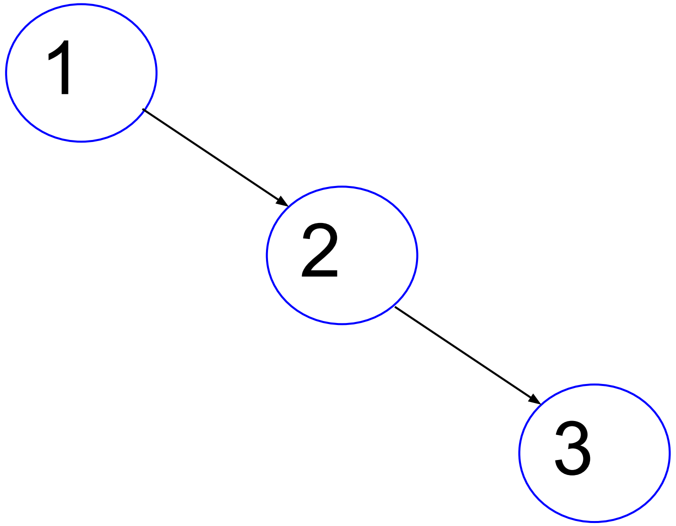


Order of insertion may change tree

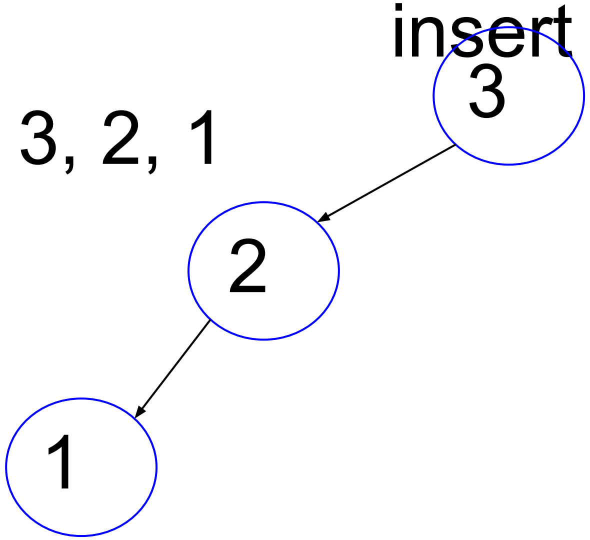
insert 2, 3, 1 or 2, 1, 3



insert 1, 2, 3



insert 3, 2, 1



```
static TreeNode * TreeNode_construct(int val)
{
        TreeNode * tn;
        tn = malloc(sizeof(TreeNode));
        tn -> left = NULL;   // remember to initialize
        tn -> right = NULL; // remember to initialize
        tn -> value = val;
        return tn;
}
```

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        {  return TreeNode_construct(val);   }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn;   }
    if (val < (tn -> value))
        {   tn -> left = Tree_insert(tn -> left, val);    }
    else
        {   tn -> right = Tree_insert(tn -> right, val);     }
    return tn;
}
TreeNode * root = NULL; // must be initialized to NULL
root = Tree_insert(root, 27)
```

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        {  return TreeNode_construct(val);   }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn;   }
    if (val < (tn -> value))
        {   tn -> left = Tree_insert(tn -> left, val);    }
    else
        {   tn -> right = Tree_insert(tn -> right, val);     }
    return tn;
}
⇨ TreeNode * root = NULL;
root = Tree_insert(root, 27)
```

Stack Memory

Frame	Symbol	Address	Value
main	root	100	NULL

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                  {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                   { return tn;   }
        if (val < (tn -> value))
                   {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                   {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Stack Memory

Frame	Symbol	Address	Value
insert	val	208	27
	tn	200	NULL
	value address 100		
main	root	100	NULL

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
            {  return TreeNode_construct(val);   }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
            { return tn;   }
    if (val < (tn -> value))
            {   tn -> left = Tree_insert(tn -> left, val);    }
    else
            {   tn -> right = Tree_insert(tn -> right, val);     }
    return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Stack Memory

Frame	Symbol	Address	Value
insert	val	208	27
	tn	200	NULL
	value address 100		
main	root	100	NULL

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                { return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                { return tn;   }
        if (val < (tn -> value))
                {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	27
	tn	200	NULL
	value address 100		
main	root	100	~~NULL~~

A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                 { return tn;   }
        if (val < (tn -> value))
                 {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                 {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
main	root	100	A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
            {  return TreeNode_construct(val);   }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
             { return tn;   }
    if (val < (tn -> value))
             {   tn -> left = Tree_insert(tn -> left, val);    }
    else
             {   tn -> right = Tree_insert(tn -> right, val);     }
    return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory

symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory

Frame	Symbol	Address	Value
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
      if (tn == NULL) // empty, create a node
              {  return TreeNode_construct(val);   }
      // not empty
      if (val == (tn -> value)) // do not insert the same value
               { return tn;   }
      if (val < (tn -> value))
               {   tn -> left = Tree_insert(tn -> left, val);    }
      else
               {   tn -> right = Tree_insert(tn -> right, val);     }
      return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                { return tn;   }
        if (val < (tn -> value))
                {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                { return tn;   }
        if (val < (tn -> value))
                {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory

symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory

Frame	Symbol	Address	Value
insert	val	308	9
	tn	300	NULL
	value address 70000		
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                  {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                  { return tn;   }
        if (val < (tn -> value))
                  {  tn -> left = Tree_insert(tn -> left, val);    }
        else
                  {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
insert	val	308	9
	tn	300	NULL
	value address 70000		
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

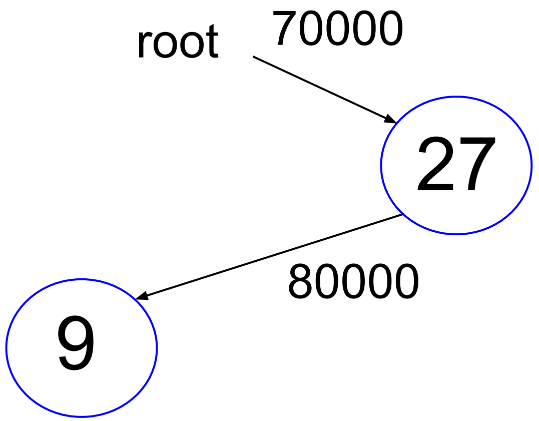
```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                    {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                    { return tn;   }
        if (val < (tn -> value))
                    {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                    {   tn -> right = Tree_insert(tn -> right, val);     }
   ⇨    return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                 {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                  { return tn;   }
        if (val < (tn -> value))
                  {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                  {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

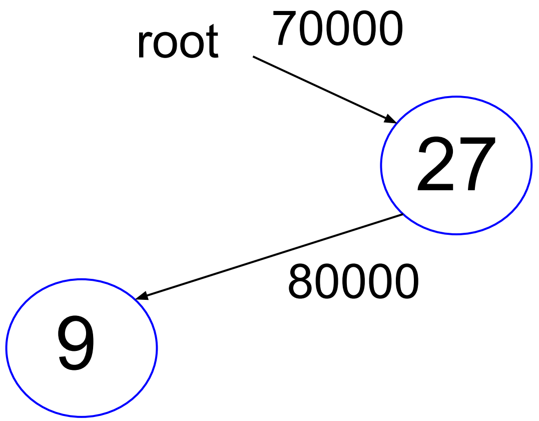


Stack Memory			
Frame	Symbol	Address	Value
main	root	100	A70000

yunglu@purdue.edu

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                  {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                   { return tn;   }
        if (val < (tn -> value))
                   {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                   {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

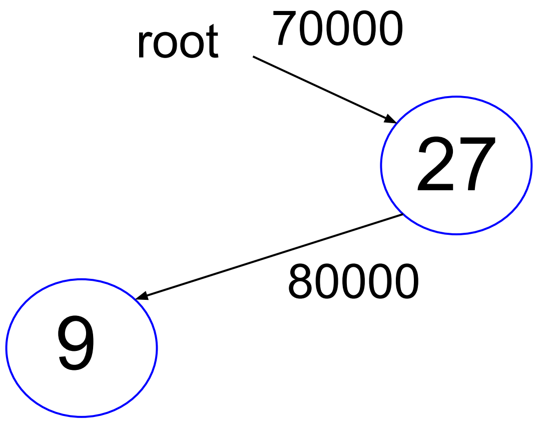
Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000



Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
            {  return TreeNode_construct(val);   }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
            { return tn;   }
    if (val < (tn -> value))
            {   tn -> left = Tree_insert(tn -> left, val);    }
    else
            {   tn -> right = Tree_insert(tn -> right, val);     }
    return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

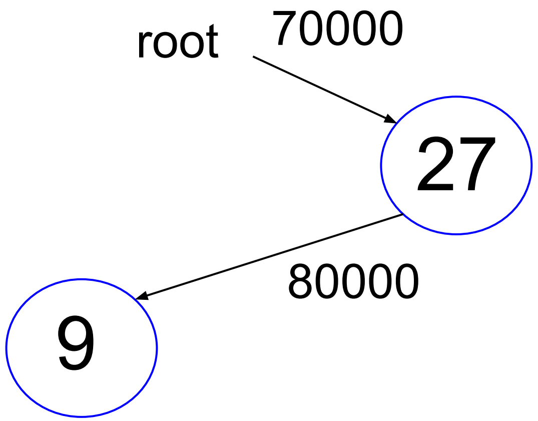
Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000



Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
      if (tn == NULL) // empty, create a node
                { return TreeNode_construct(val);   }
      // not empty
      if (val == (tn -> value)) // do not insert the same value
                { return tn;   }
      if (val < (tn -> value))
                {   tn -> left = Tree_insert(tn -> left, val);    }
      else
                {   tn -> right = Tree_insert(tn -> right, val);     }
      return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

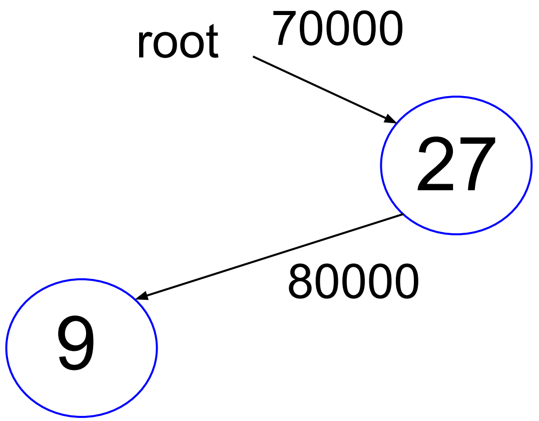


Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                  {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                   { return tn;   }
        if (val < (tn -> value))
                   {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                   {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

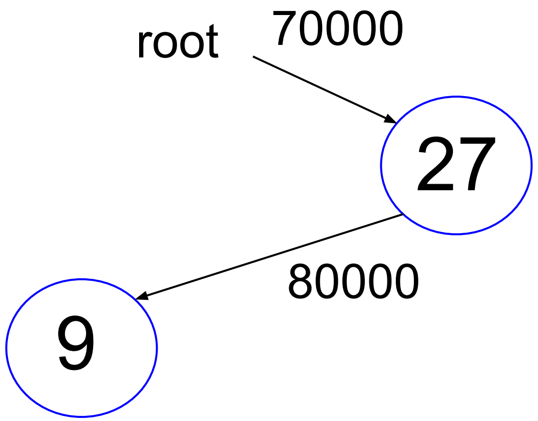
Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000



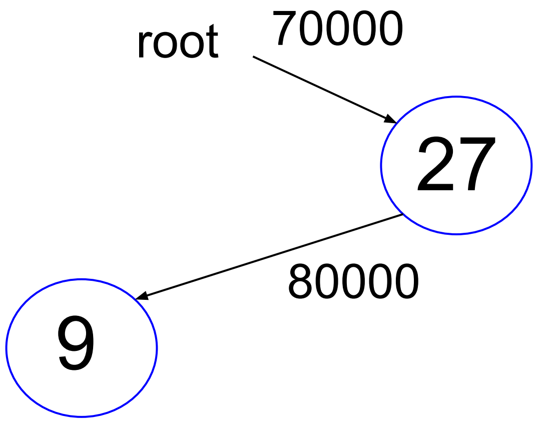
```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                  {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                   { return tn;   }
        if (val < (tn -> value))
                   {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                   {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000



Stack Memory			
Frame	Symbol	Address	Value
insert	val	308	38
	tn	300	NULL
	value address 70008		
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000

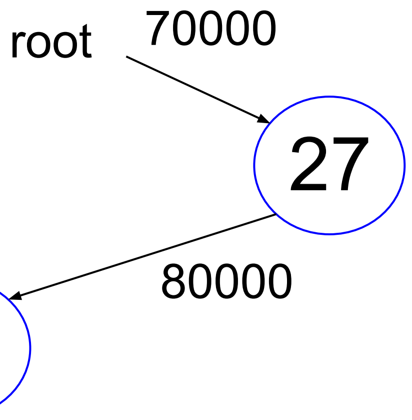
```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                  {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                  { return tn;   }
        if (val < (tn -> value))
                  {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                  {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```



Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
insert	val	308	38
	tn	300	NULL
	value address 70008		
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                { return tn;   }
        if (val < (tn -> value))
                {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                {   tn -> right = Tree_insert(tn -> right, val);    }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```



Heap Memory

symbol	Address	Value
value	90016	38
right	90008	NULL
left	90000	NULL
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	A90000
left	70000	A80000

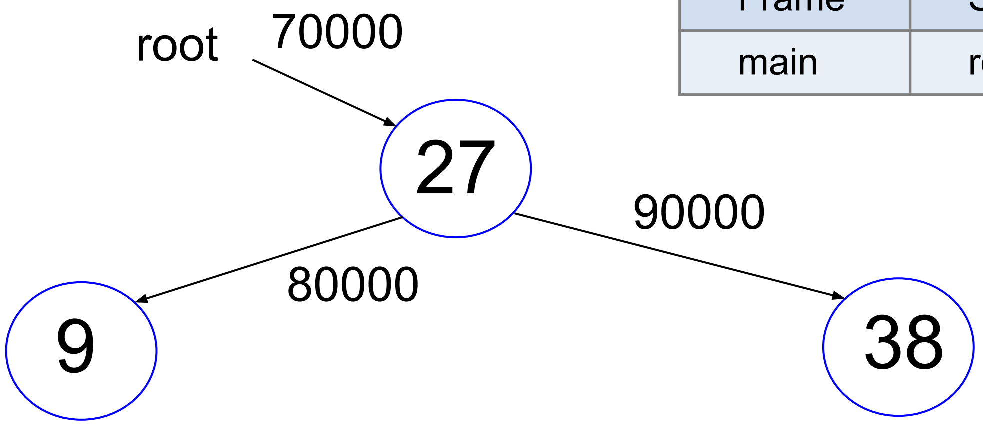
Stack Memory

Frame	Symbol	Address	Value
insert	val	308	38
	tn	300	NULL
	value address 70008		
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000

```
TreeNode * Tree_insert(TreeNode * tn, int val)
{
        if (tn == NULL) // empty, create a node
                  {  return TreeNode_construct(val);   }
        // not empty
        if (val == (tn -> value)) // do not insert the same value
                  { return tn;   }
        if (val < (tn -> value))
                  {   tn -> left = Tree_insert(tn -> left, val);    }
        else
                  {   tn -> right = Tree_insert(tn -> right, val);     }
        return tn;
}
TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);
```

Heap Memory		
symbol	Address	Value
value	90016	38
right	90008	NULL
left	90000	NULL
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	A90000
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
main	root	100	A70000

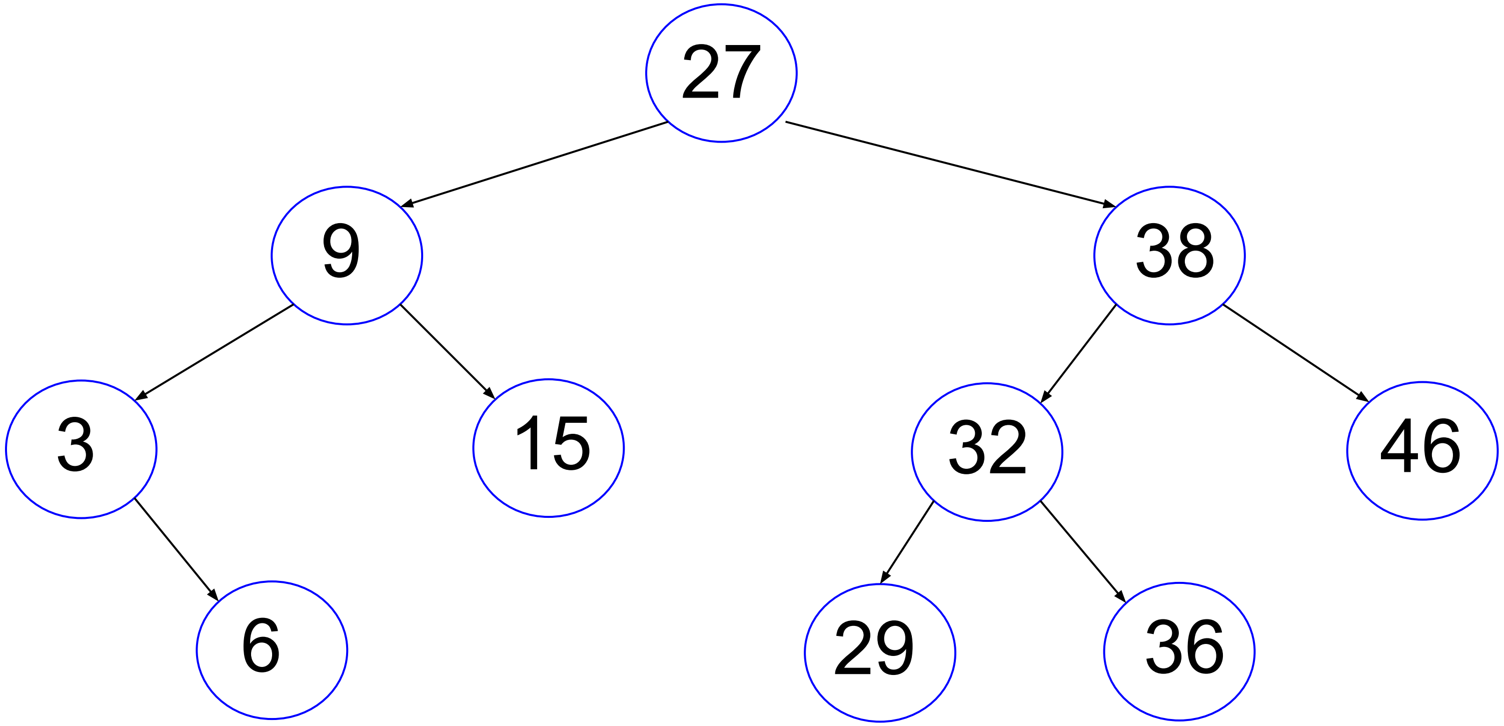


Print and Destroy

Traverse Binary Tree

How to visit every node in a binary tree?

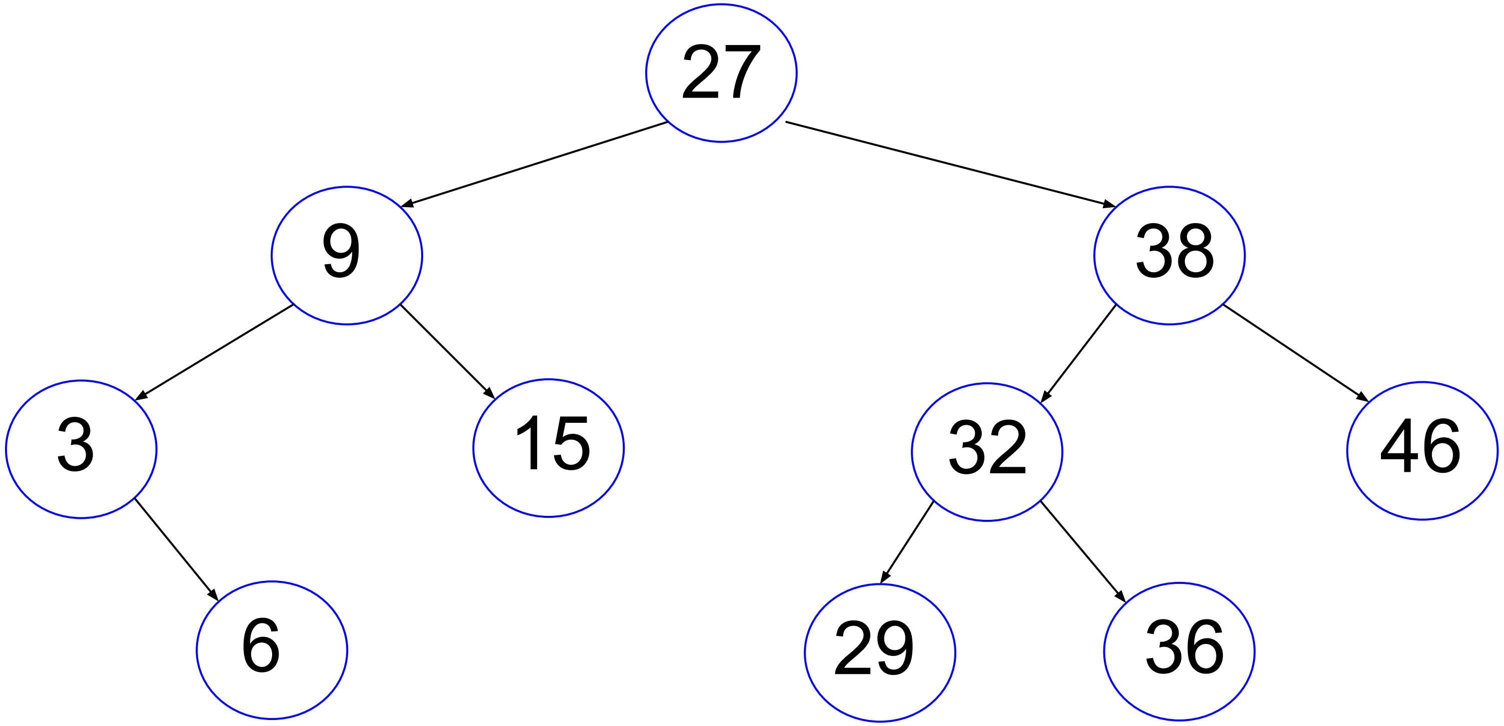
(may not be search tree)



Traverse Binary Tree

A. visit the node
B. visit the left subtree
C. visit the right subtree

A – B – C: pre-order
B – A – C: in-order
B – C – A: post-order



Pre-Order

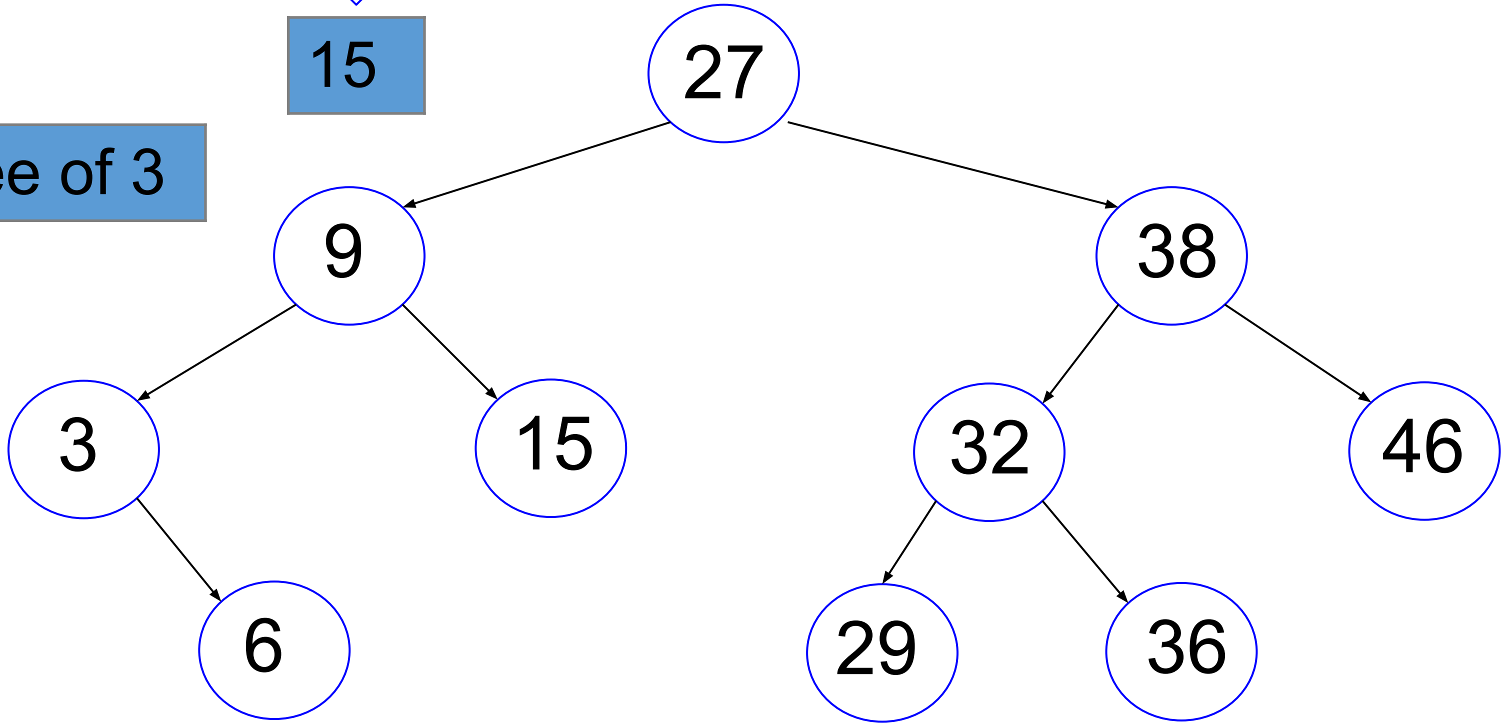
27	left subtree of 27	Right subtree of 27

27	9	left subtree of 9	right subtree of 9	Right subtree of 27

15

3	left subtree of 3	right subtree of 3

3		6



left subtree of 27

9	3	6	15

Pre-Order

27	9	3	6	15	Right subtree of 27

Right subtree of 27

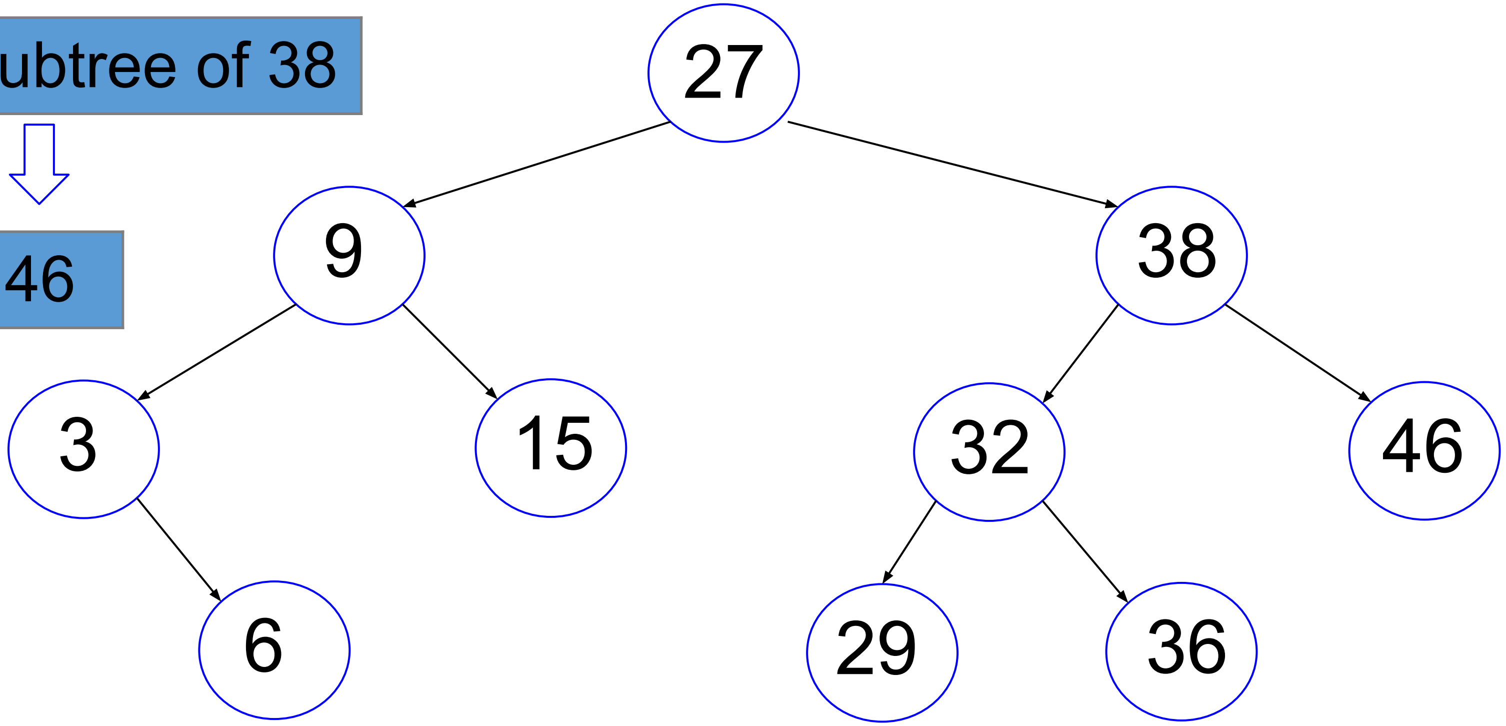
⇩

38	left subtree of 38	right subtree of 38

⇩ ⇩

32	29	36

46



right subtree of 27

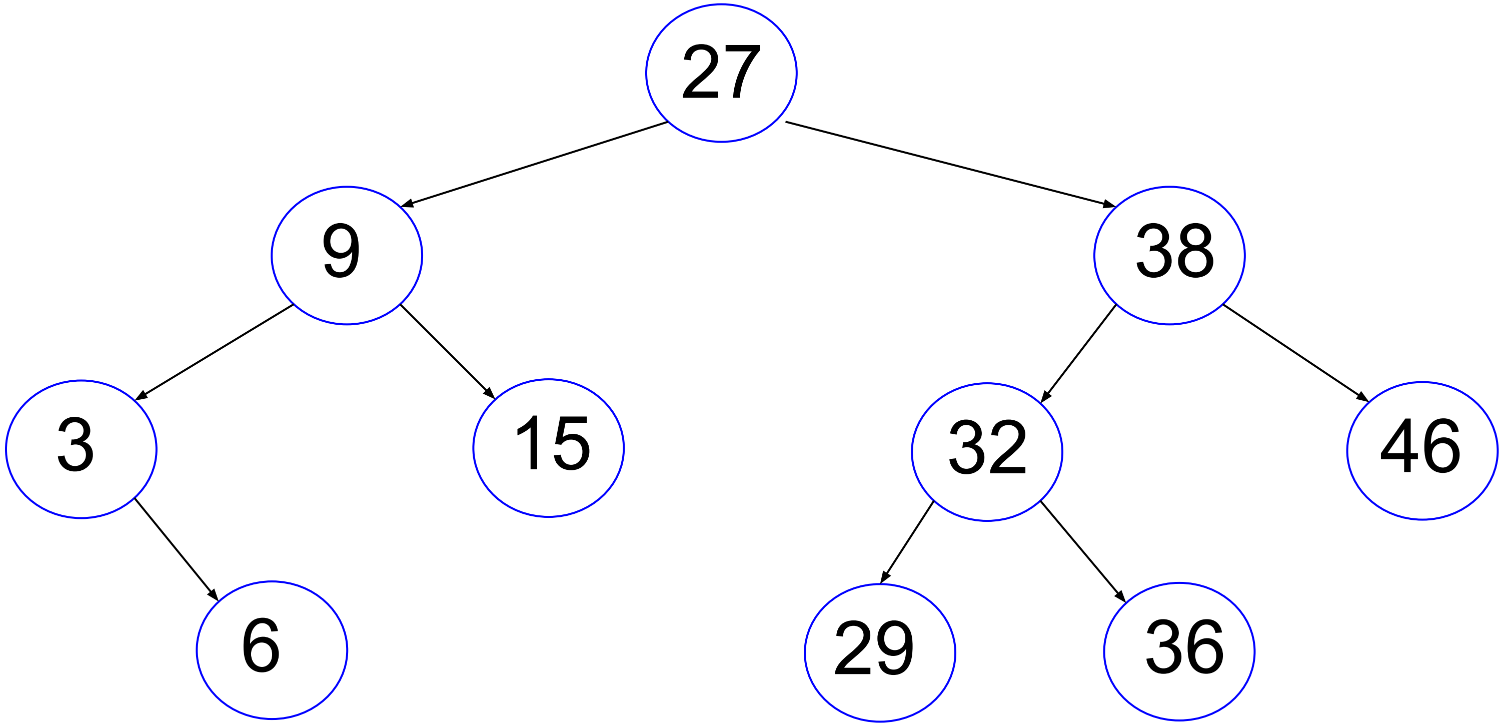
⇩

38	32	29	36	46

yunglu@purdue.edu

Pre-Order

27	9	3	6	15	38	32	29	36	46



In-Order

left subtree of 27	27	Right subtree of 27

left subtree of 9	9	right subtree of 9

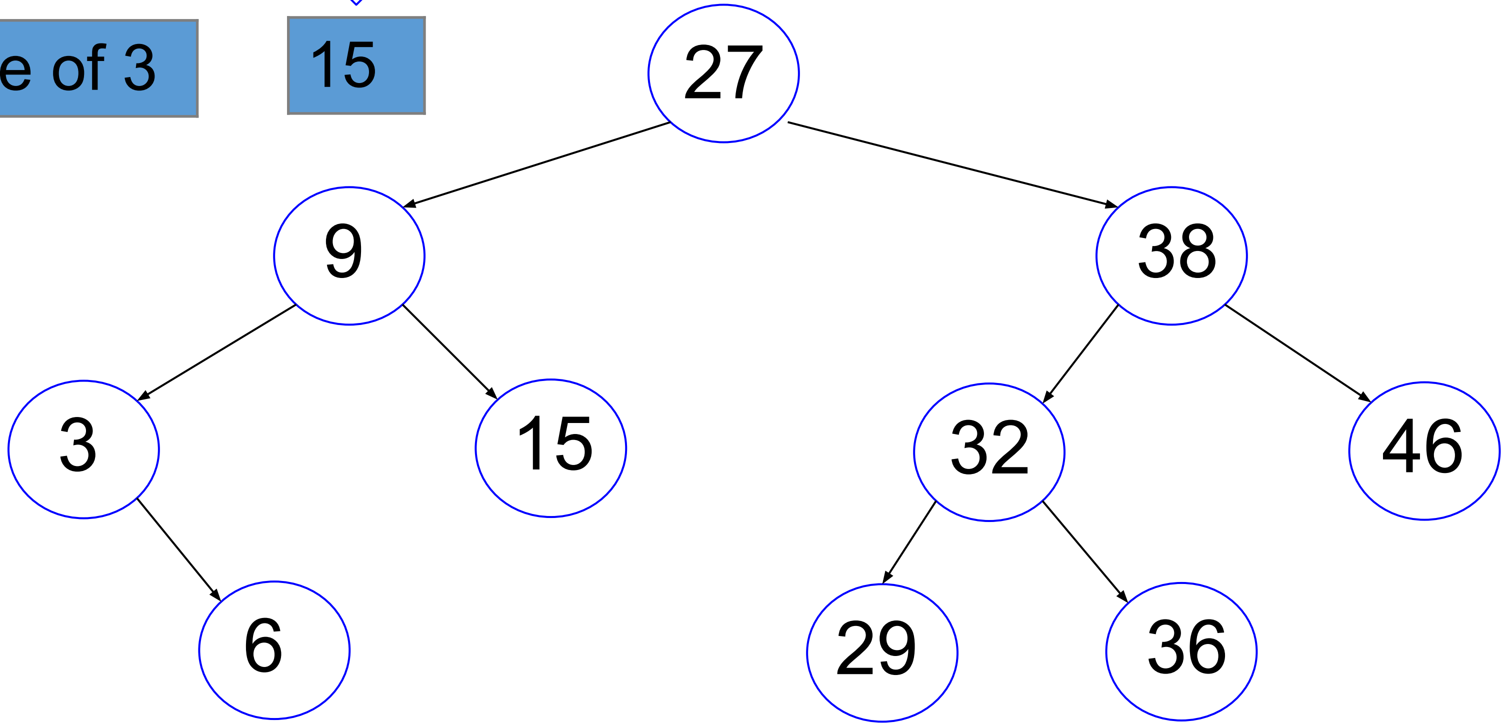
left subtree of 3	3	right subtree of 3

15

	3	6

left subtree of 27

3	6	9	15



In-Order

3	6	9	15	27	Right subtree of 27

Right subtree of 27

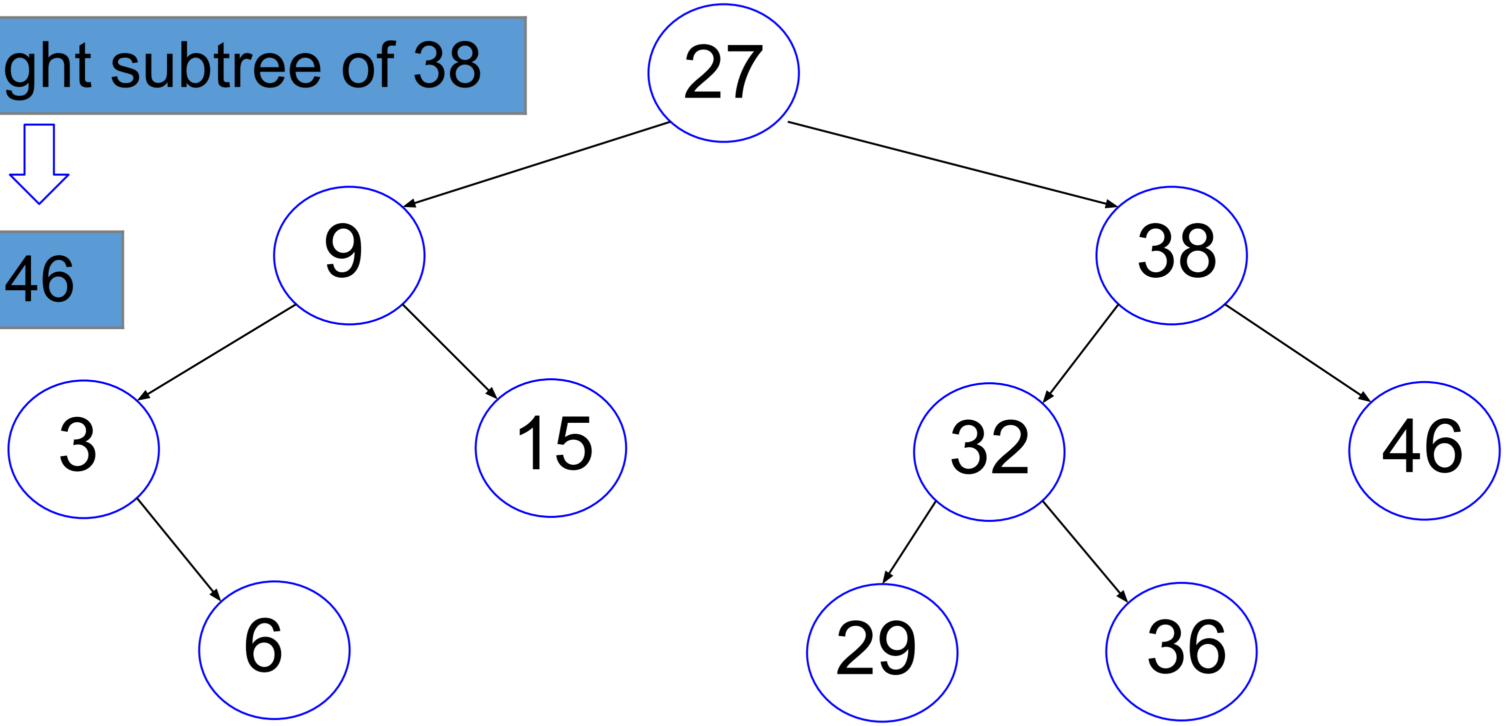
⇩

left subtree of 38	38	right subtree of 38

⇩ ⇩

29	32	36

46



right subtree of 27

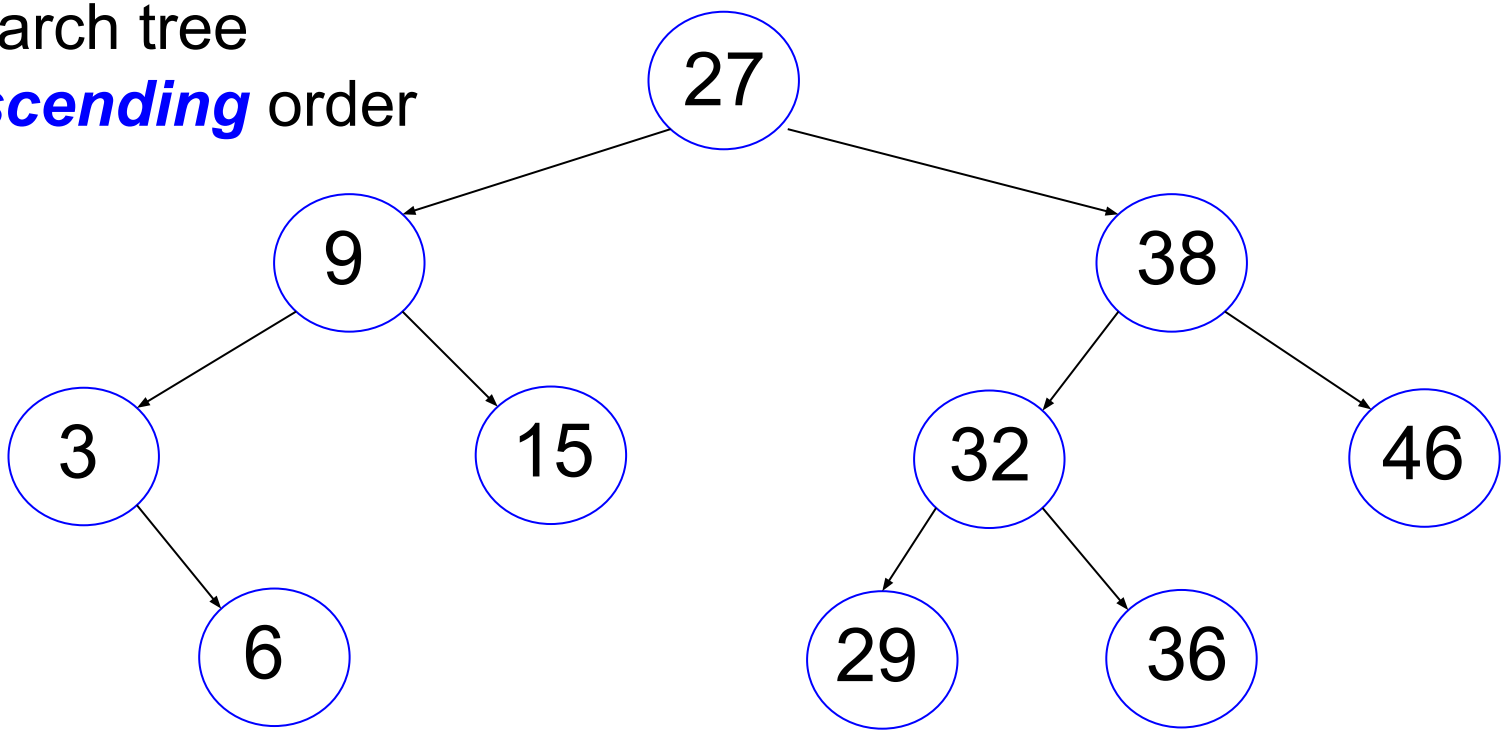
⇩

29	32	36	38	46

In-Order

3	6	9	15	27	29	32	36	38	46

In-Order traversal of a binary search tree
create output the keys by the ***ascending*** order



Post-Order

left subtree of 27	Right subtree of 27	27

left subtree of 9	right subtree of 9	9

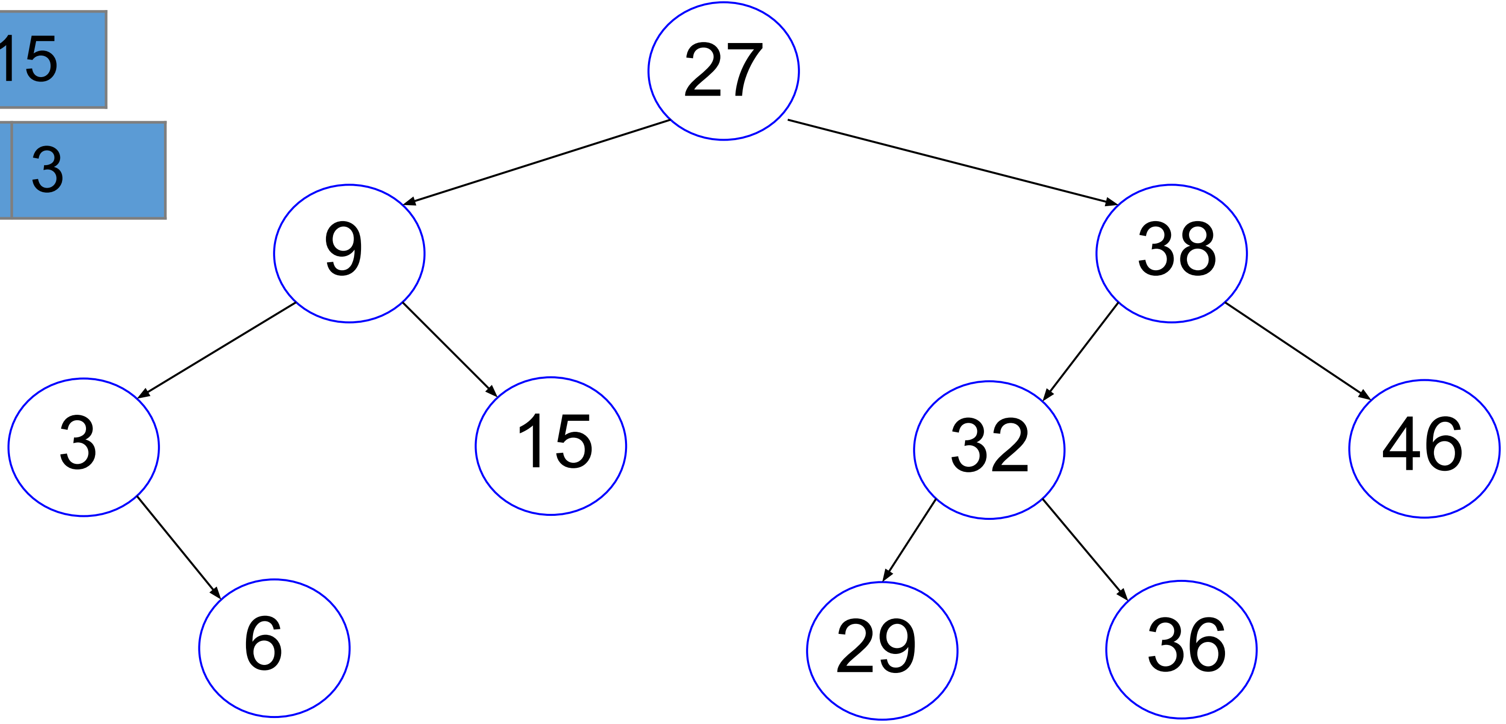
15

left subtree of 3	right subtree of 3	3

	6	3

left subtree of 27

6	3	15	9



Post-Order

6	3	15	9	Right subtree of 27	27

Right subtree of 27

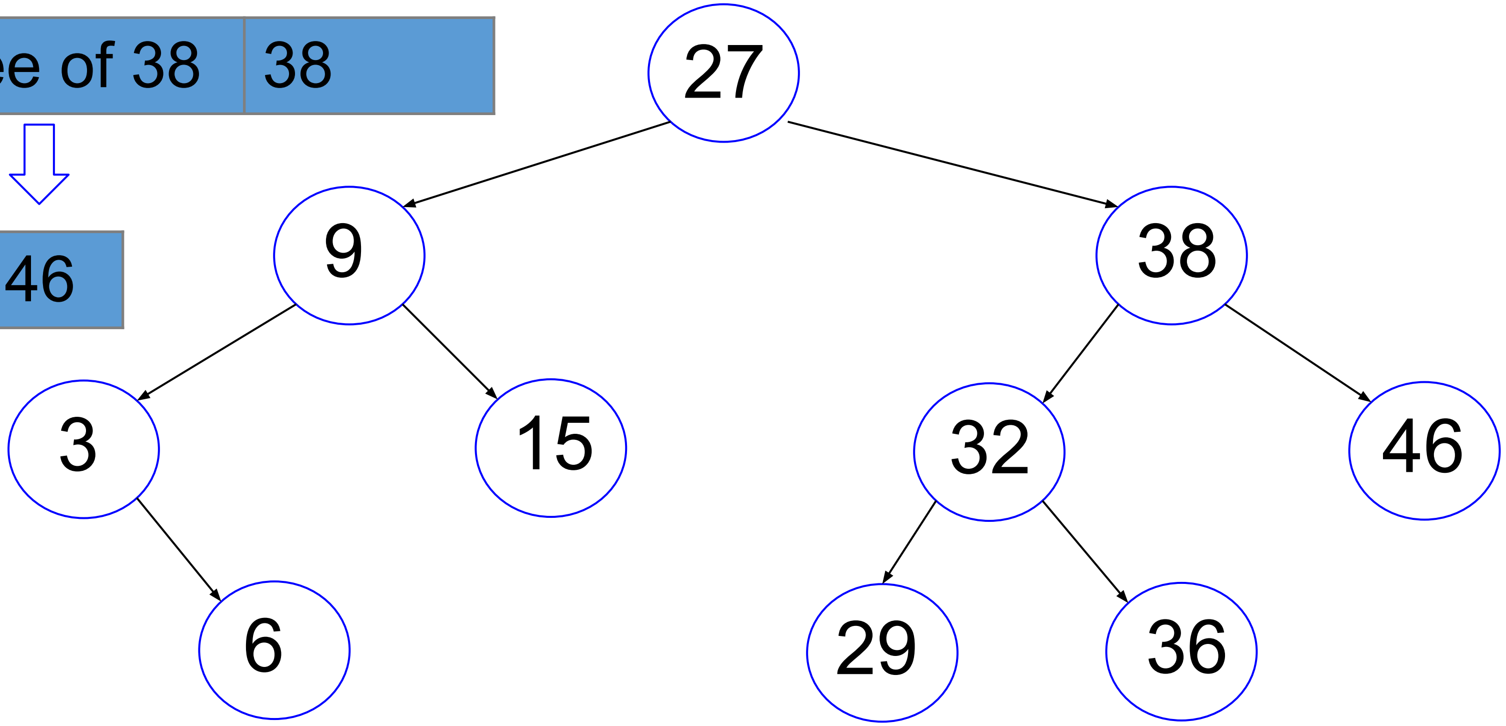
⇩

left subtree of 38	right subtree of 38	38

⇩ ⇩

29	36	32

46



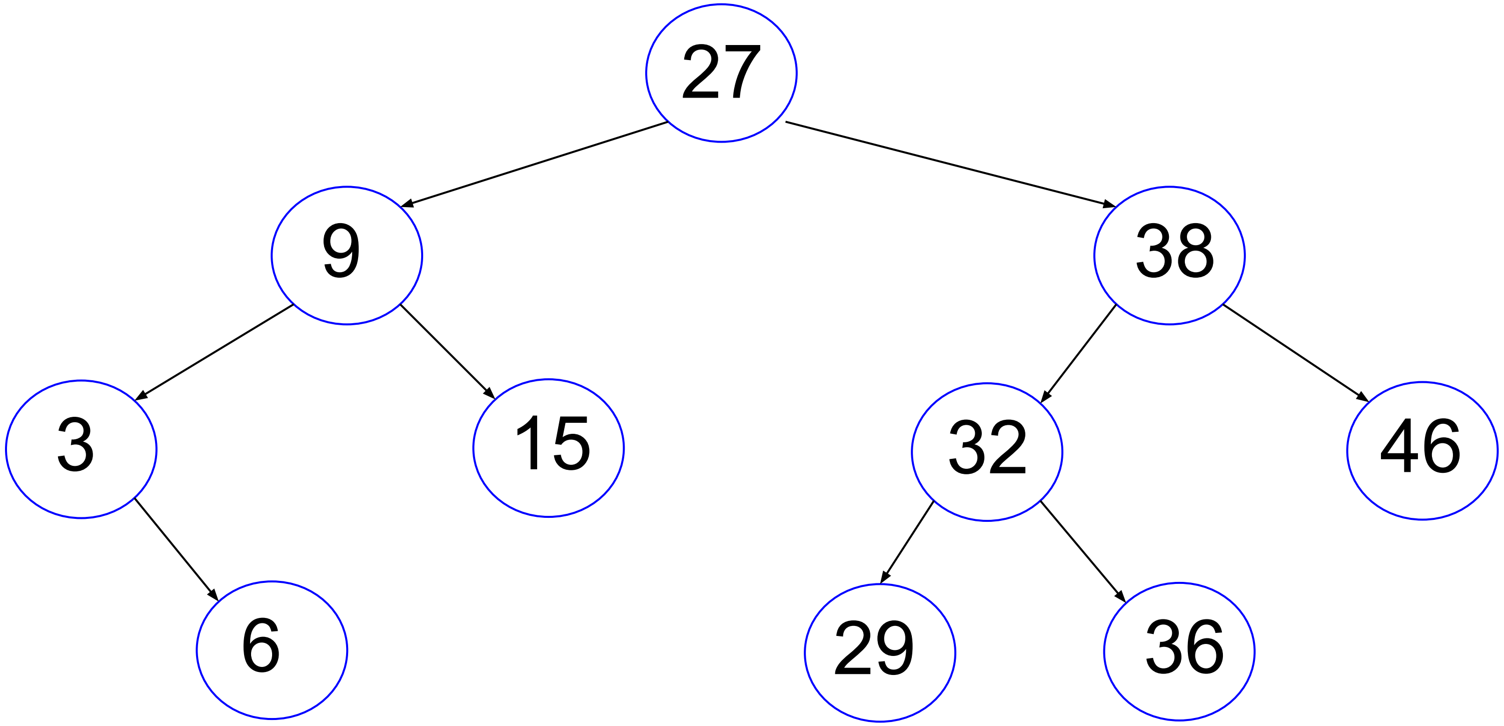
right subtree of 27

⇩

29	36	32	46	38

Post-Order

6	3	15	9	29	36	32	46	38	27



```
void Tree_printPreorder(TreeNode *tn)
{
  if (tn == NULL)
    {
      return;
    }
  printf("%d ",tn -> value);
  Tree_printPreorder(tn -> left);
  Tree_printPreorder(tn -> right);
}
```

```
void Tree_printInorder(TreeNode *tn)
{
  if (tn == NULL)
    {
      return;
    }
  Tree_printInorder(tn -> left);
  printf("%d ",tn -> value);
  Tree_printInorder(tn -> right);
}
```

```
void Tree_printPostorder(TreeNode *tn)
{
  if (tn == NULL)
    {
      return;
    }
  Tree_printPostorder(tn -> left);
  Tree_printPostorder(tn -> right);
  printf("%d ",tn -> value);
}
```

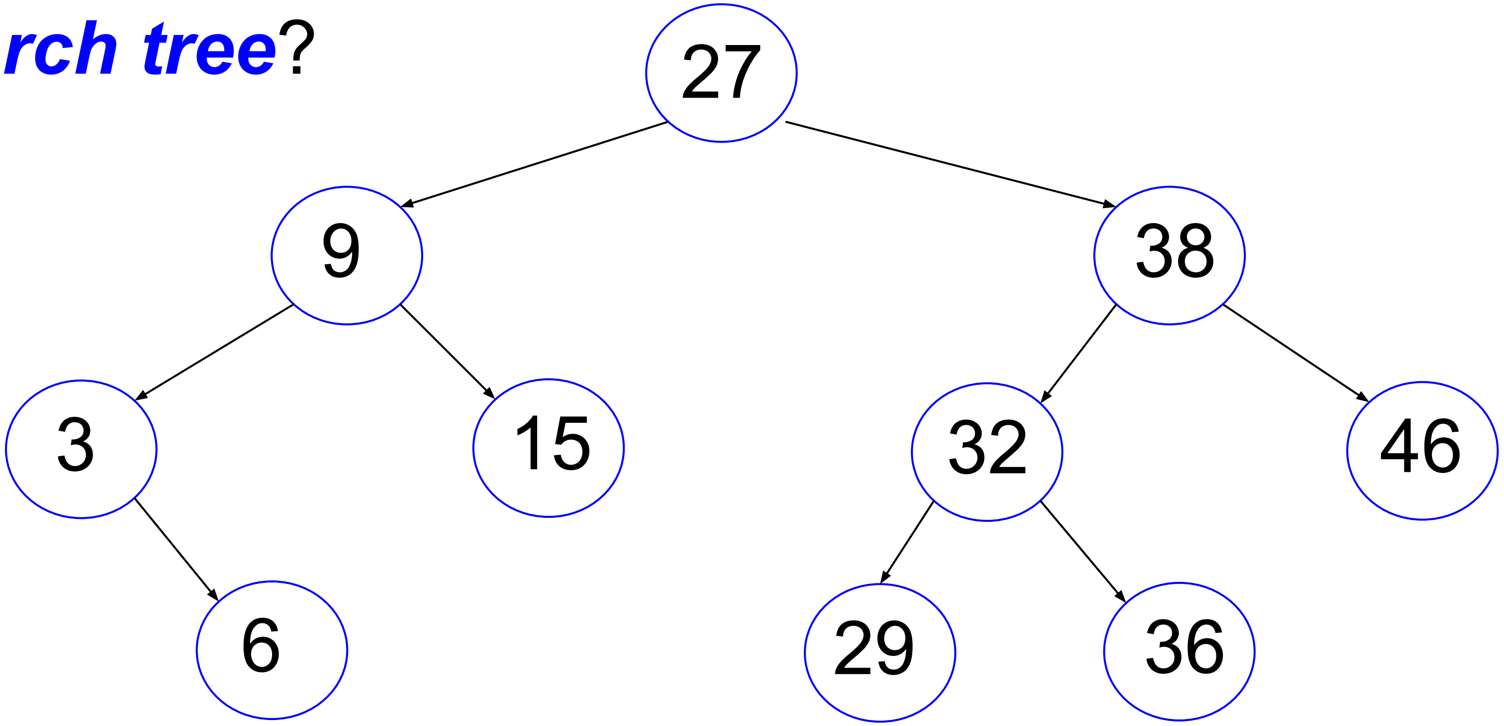
```
void Tree_destroy(TreeNode *tn)
// delete every node
{
  if (tn == NULL)
    {
      return;
    }
  Tree_destroy (tn -> left);
  Tree_destroy (tn -> right);
  free (tn); // must be post-order
  // here tn -> left and tn-> right undefined
}
```

Delete a Node in Binary ***Search*** Tree

yunglu@purdue.edu

Delete a Node in Binary Search Tree

How to a node in a binary search tree
and ***keep it a binary search tree***?

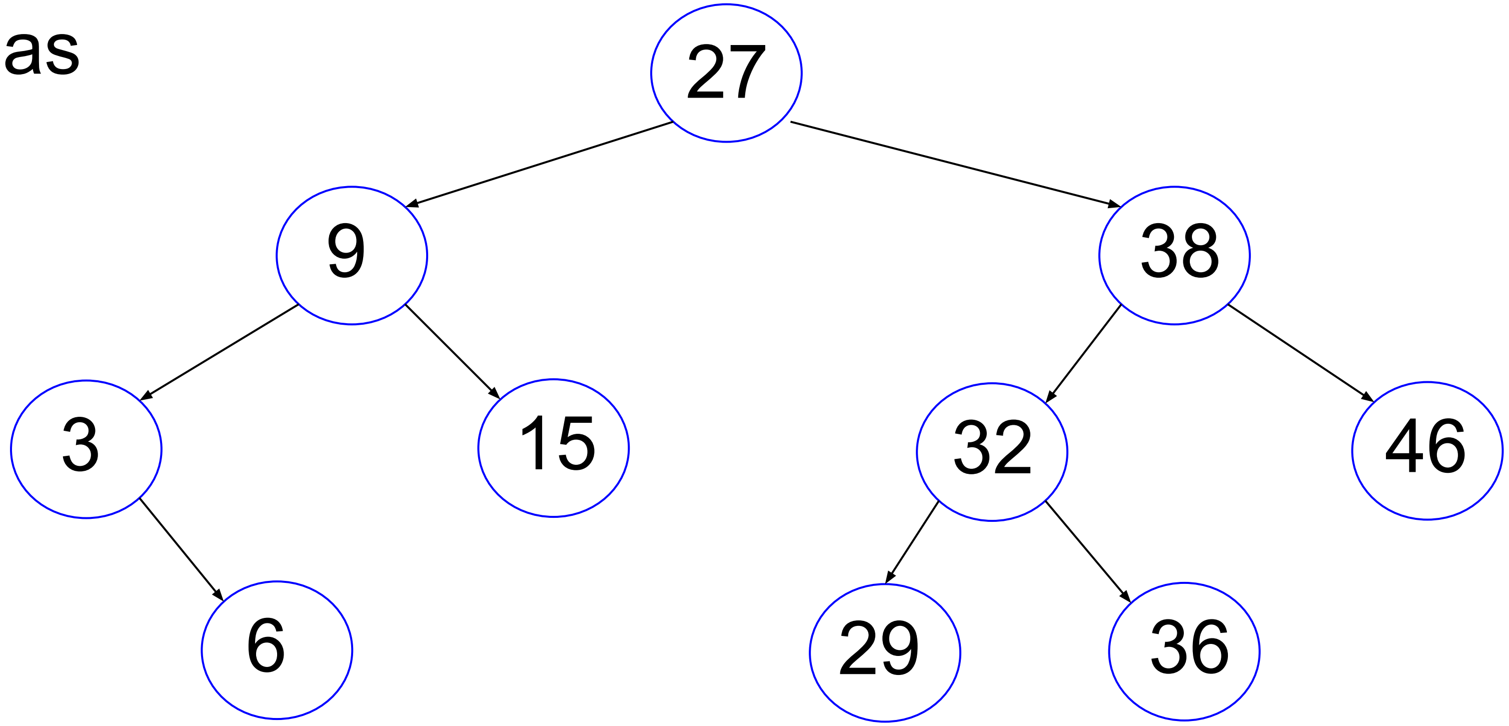


Delete a Node in Binary Search Tree

Three different cases.

The node to be deleted has

1. no child
2. one child
3. two children



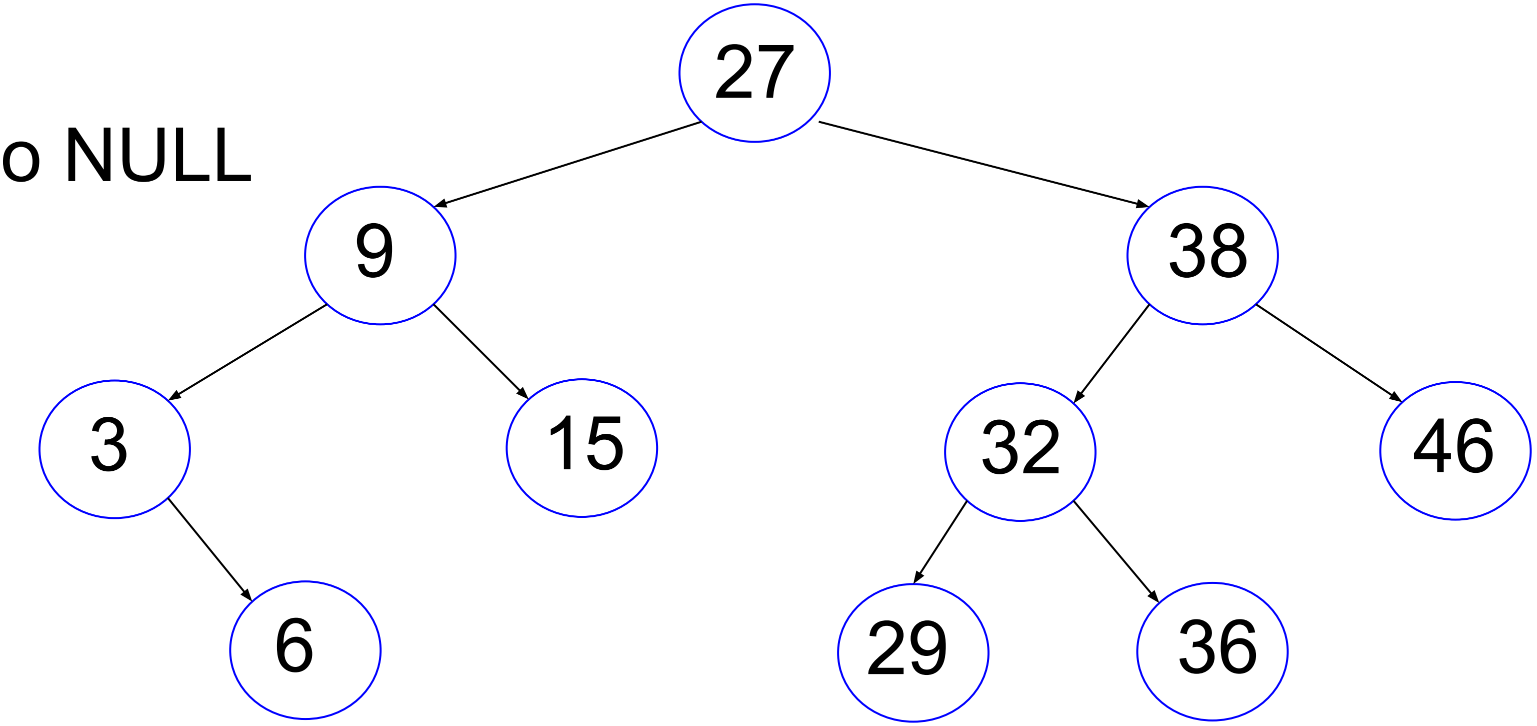
Delete a Node in Binary Search Tree

no child (delete a leaf node) ⇒

6, 15, 29, 36, or 46

set the parent’s pointer to NULL

free memory



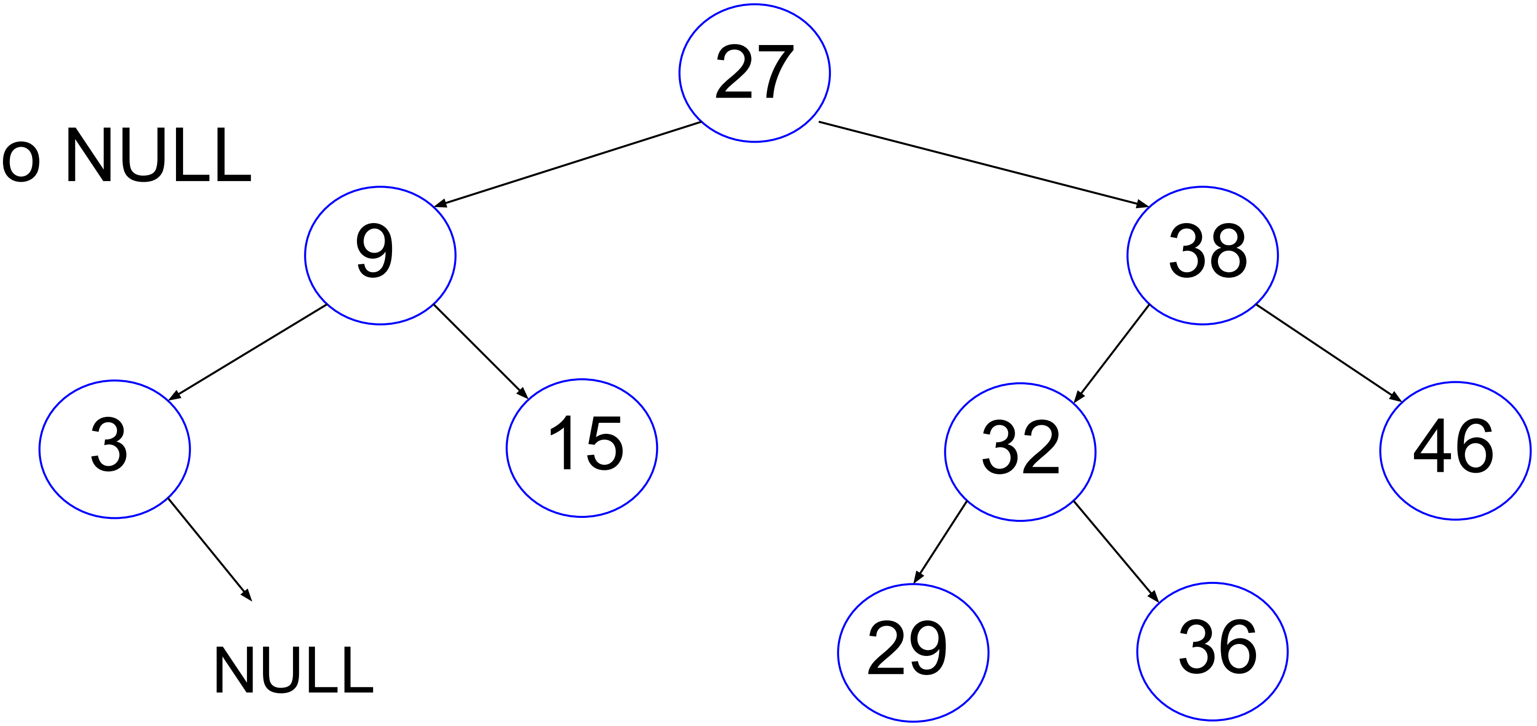
Delete a Node in Binary Search Tree

no child (delete a leaf node) ⇒

6, 15, 29, 36, or 46

set the parent’s pointer to NULL

free memory



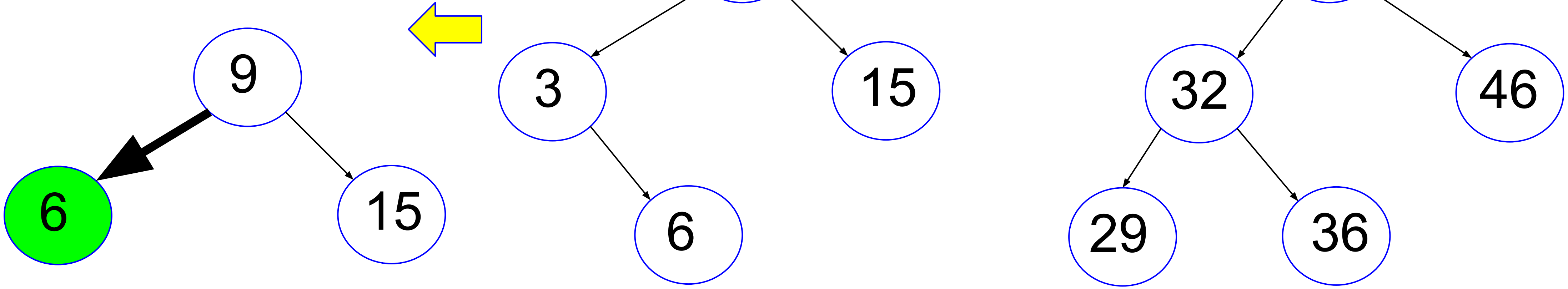
Delete a Node in Binary Search Tree

one child ⇒

For example, 3

make the parent’s pointer to child

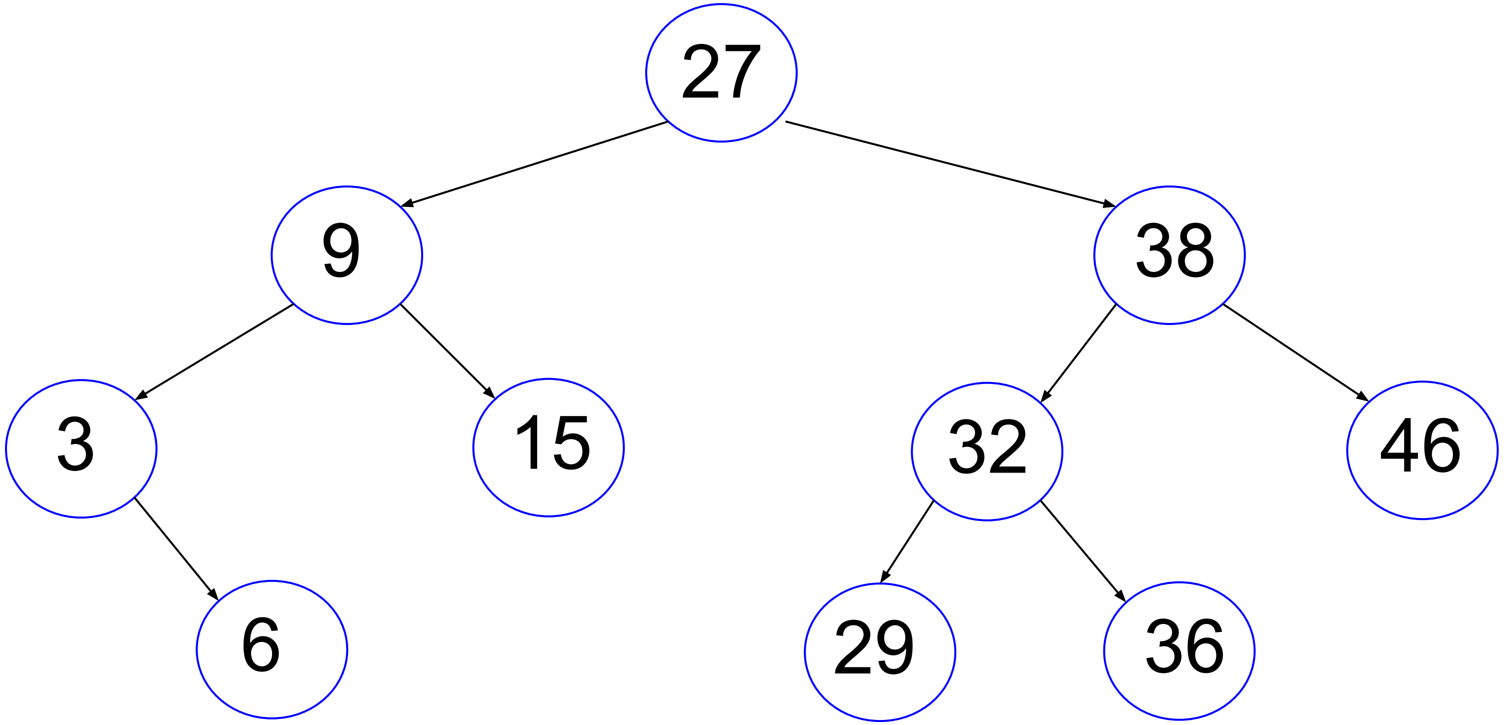
free memory



Delete a Node in Binary Search Tree

two children ⇒

For example, 27



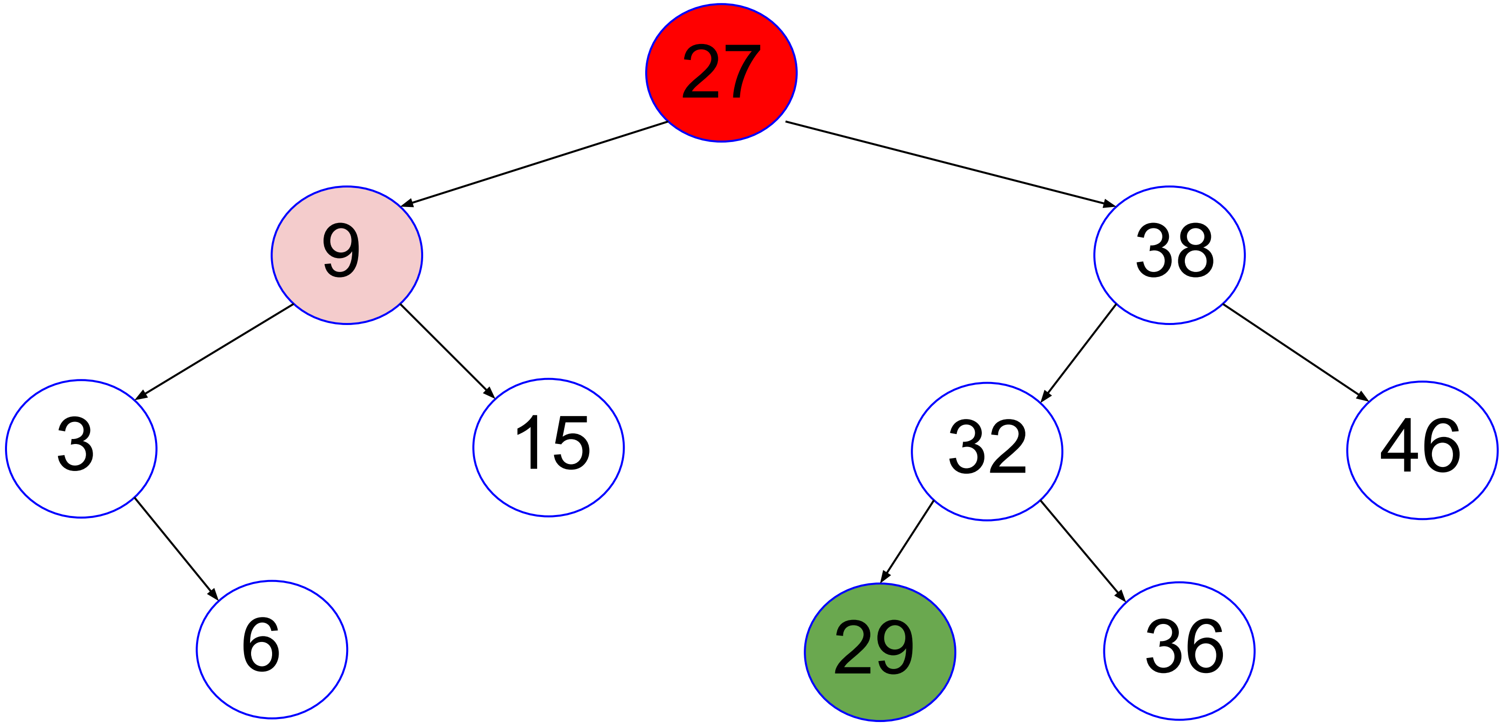
Delete a Node in Binary Search Tree

two children ⇒

For example, 27

Choice 1:

Put **left-sub tree** as the **left most child**



Delete a Node in Binary Search Tree

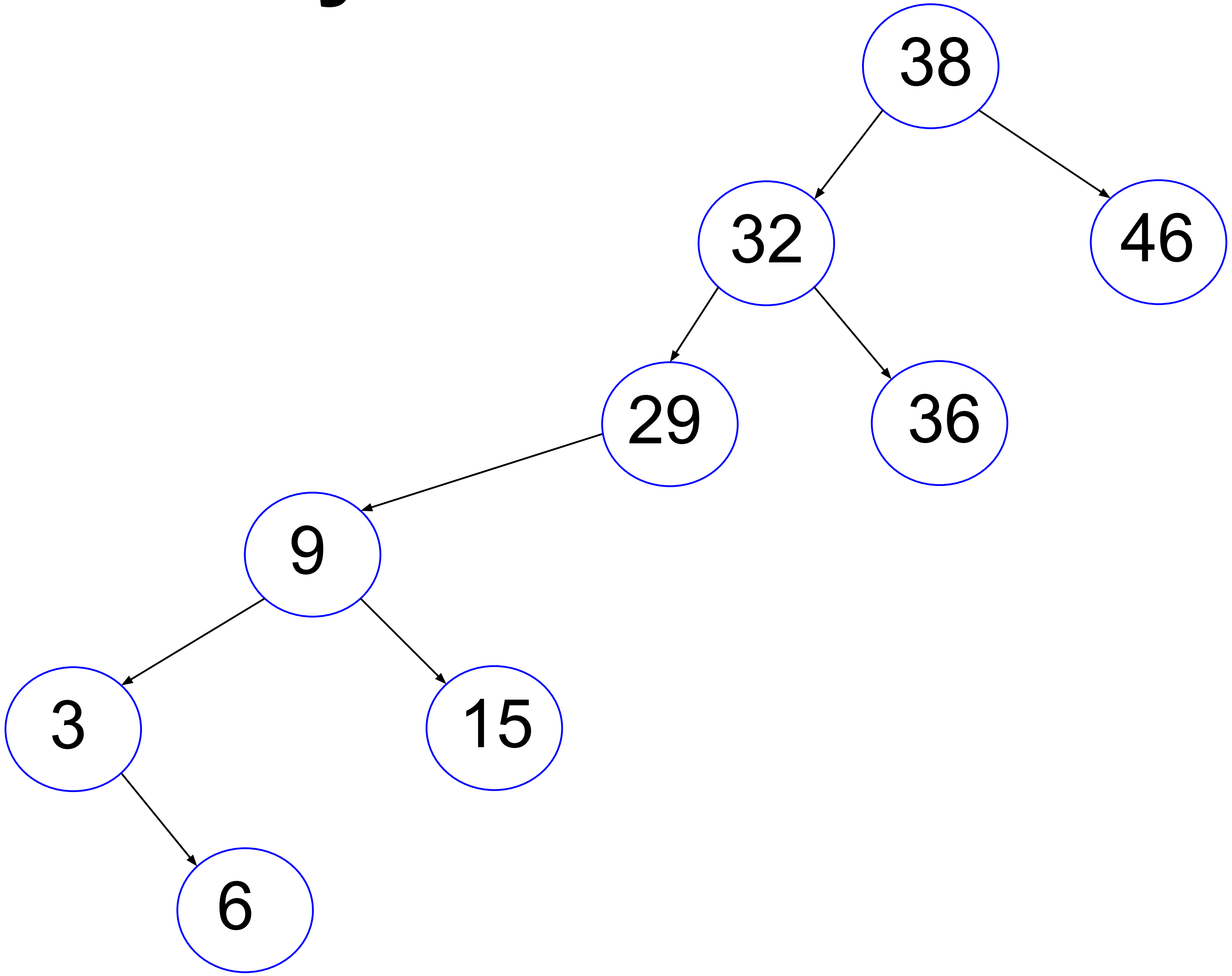
two children ⇒

For example, 27

Choice 1:

Put left-sub tree as the

left most child



Delete a Node in Binary Search Tree

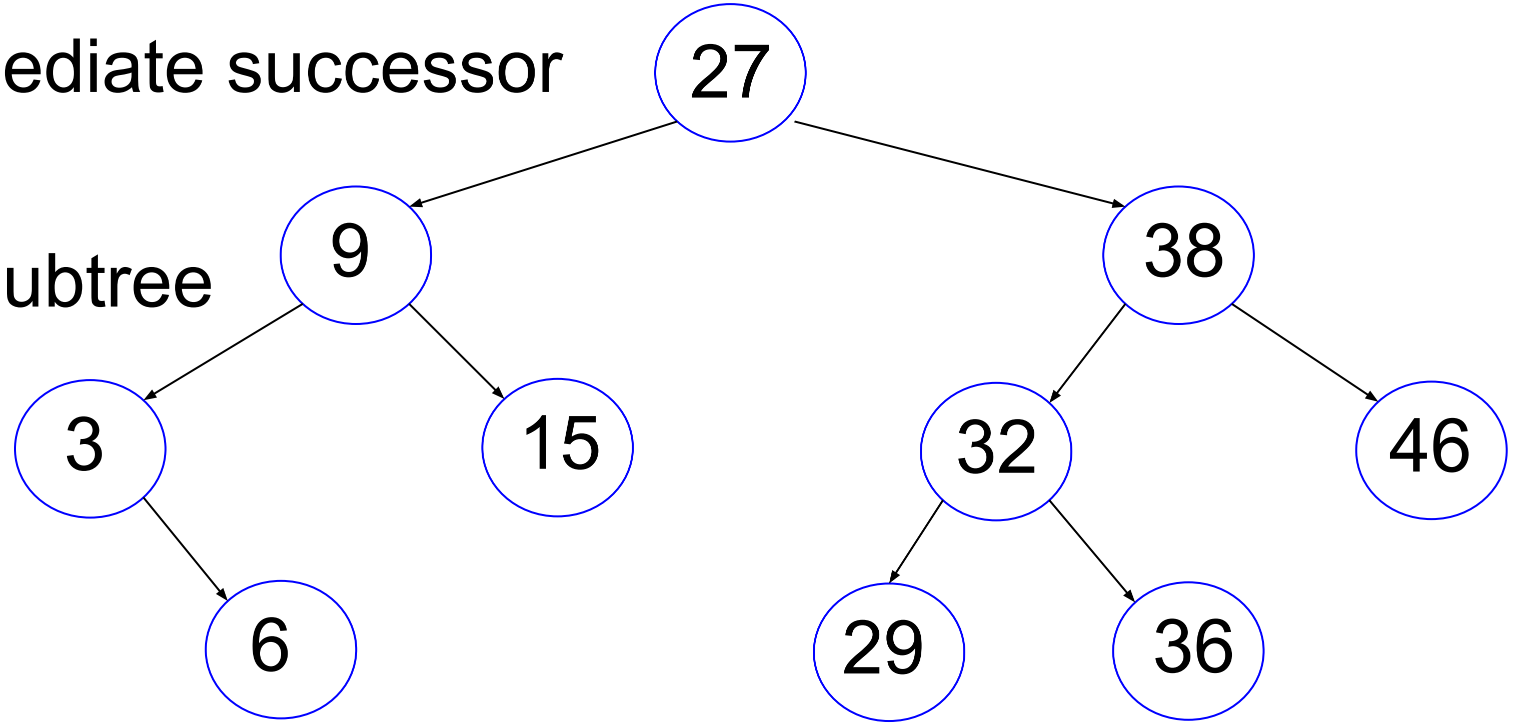
two children ⇒

For example, 27

Choice 2: swap with immediate successor

in in-order traversal

delete 27 from the right subtree



Delete a Node in Binary Search Tree

two children ⇒

For example, 27

swap with immediate successor

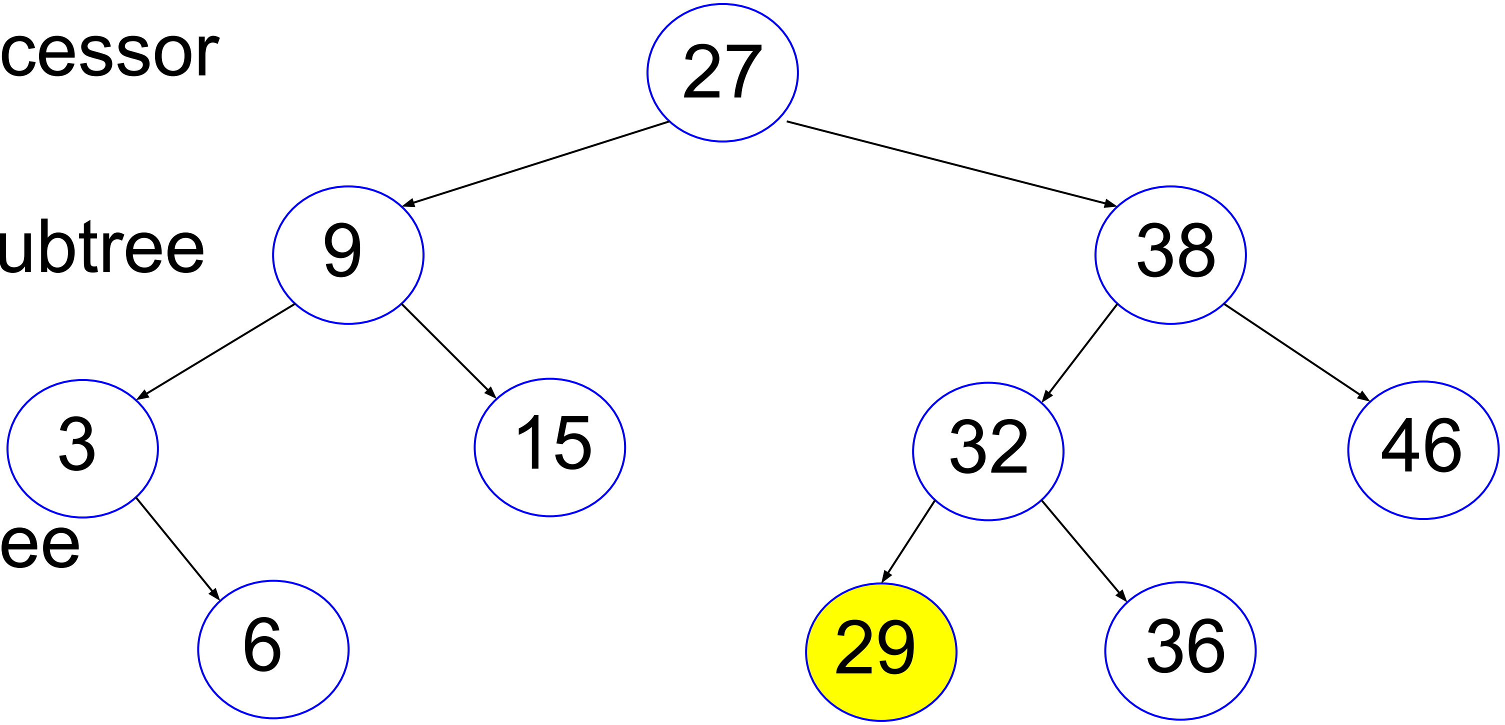
in in-order traversal

delete 27 from the right subtree

immediate successor:

smallest in the right subtree

leftmost node



Delete a Node in Binary Search Tree

two children ⇒

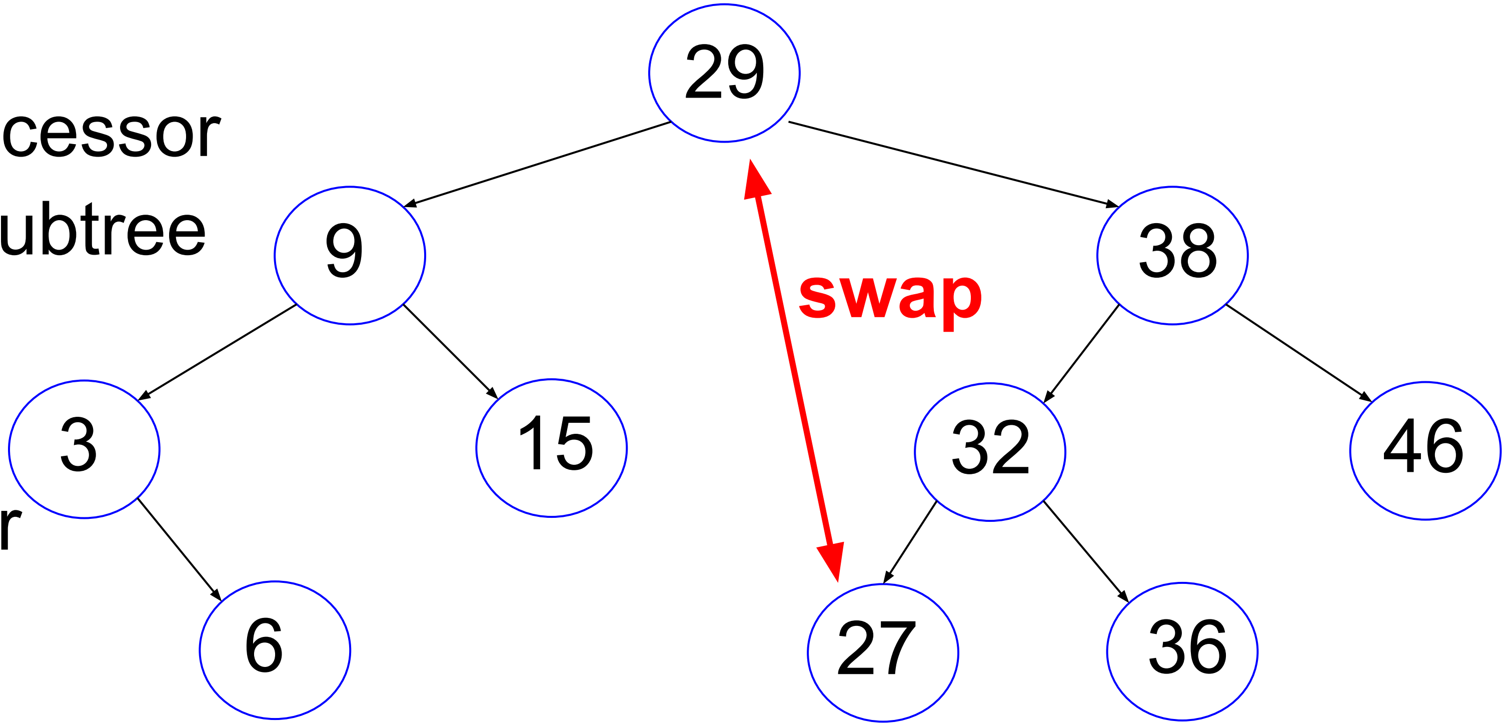
For example, 27

swap with immediate successor

delete 27 from the right subtree

The immediate successor

must not have left child



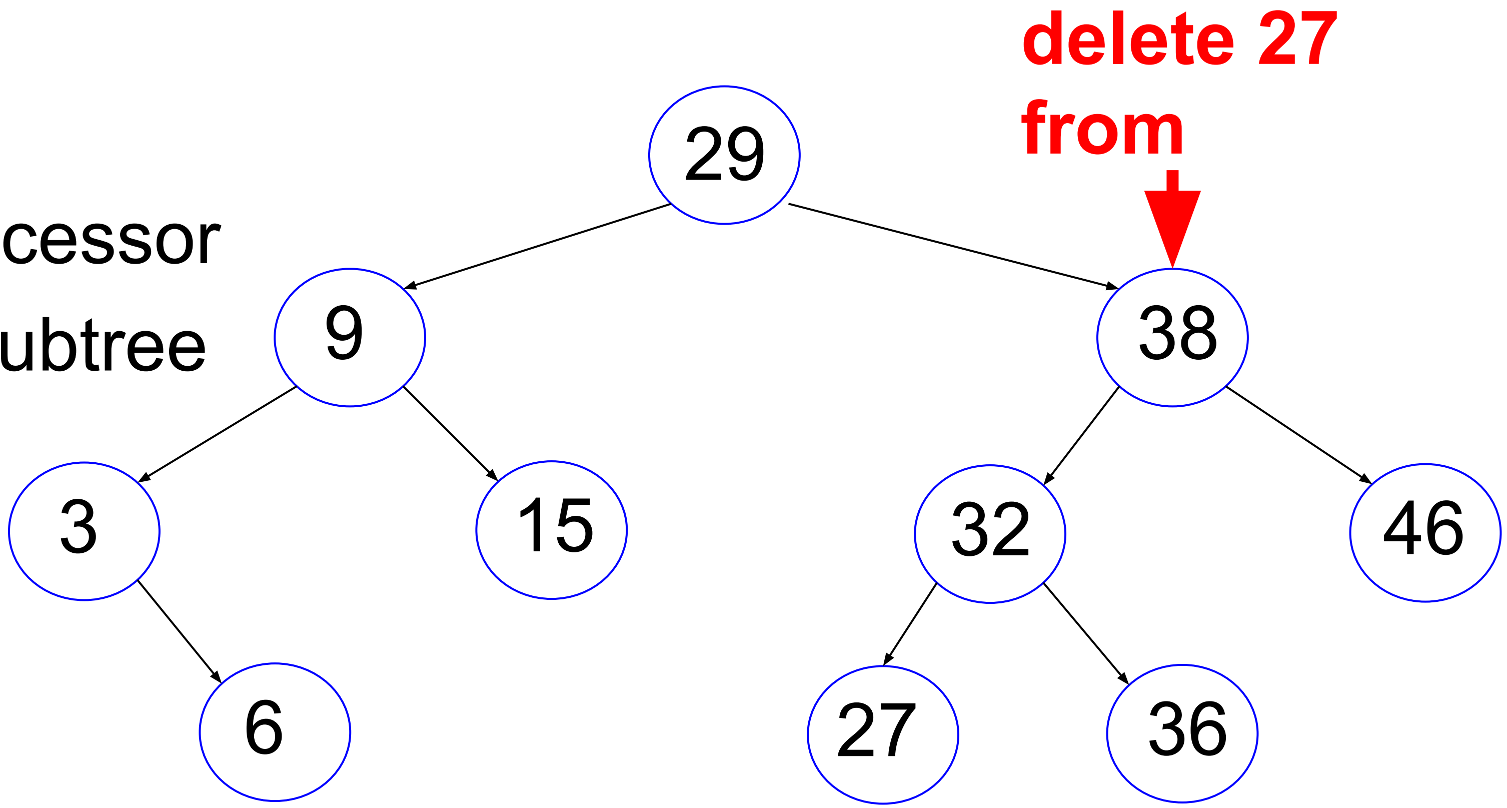
Delete a Node in Binary Search Tree

two children ⇒

For example, 27

swap with immediate successor

delete 27 from the right subtree



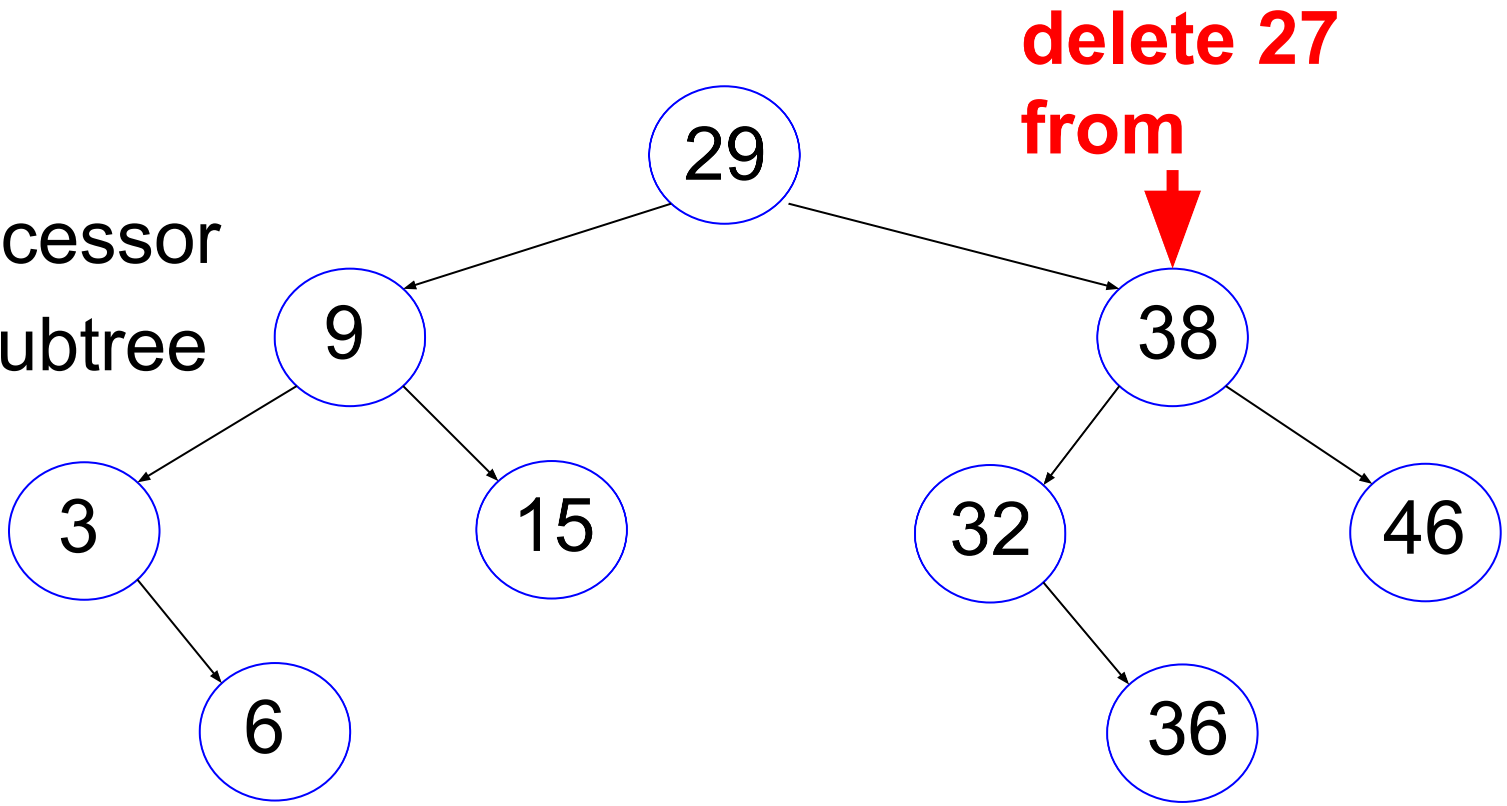
Delete a Node in Binary Search Tree

two children ⇒

For example, 27

swap with immediate successor

delete 27 from the right subtree



```
TreeNode * Tree_delete(TreeNode * tn, int val)
{
  if (tn == NULL) { return NULL; }
  if (val < (tn -> value))
    {
      tn -> left = Tree_delete(tn -> left, val);
      return tn;
    }
  if (val > (tn -> value))
    {
      tn -> right = Tree_delete(tn -> right, val);
      return tn;
    }
  // val is the same as tn ->  value, delete this node
```

```
if (((tn ->  left) == NULL) && ((tn ->  right) == NULL))
  {
    // tn has no child
    free (tn);
    return NULL;
  }
if ((tn ->  left) == NULL)
  {
    // tn ->  right must not be NULL
    TreeNode * rc = tn ->  right;
    free (tn);
    return rc;
  }
```

```
if ((tn ->  right) == NULL)
    {
      // tn ->  left must not be NULL
      TreeNode * lc = tn ->  left;
      free (tn);
      return lc;
    }
```

```
// tn have two children
```

```
  // tn have two children
  // find the immediate successor
  TreeNode * su = tn ->  right; // su must not be NULL
  while ((su -> left) != NULL)
    {
      su = su -> left;
    }
  // su is tn's immediate successor
  // swap their values
  tn ->  value = su -> value;
  su -> value = val;
  // delete su
  tn ->  right = Tree_delete(tn ->  right, val);
  return tn;
}
```

Common Mistakes

```
TreeNode * Tree_delete(TreeNode * tn, int val)
{
  if (tn == NULL) { return NULL; } // must check first
  if (val < (tn -> value))
    {
      tn -> left = Tree_delete(tn -> left, val);
      // wrong if using tn in either place
      // using tn in the argument: recursion will not end
      // using tn = loses this node
      return tn; // remember to return tn
    }
```

```
if (((tn ->  left) == NULL) && ((tn ->  right) == NULL))
  {
    // tn has no child
    free (tn);
    return NULL;
  }
if ((tn ->  left) == NULL)
  {
    // tn ->  right must not be NULL
    TreeNode * rc = tn ->  right;
    free (tn); // careful order
    return rc;
  }

// tn have two children
```

```
  // tn have two children
  // find the immediate successor
  TreeNode * su = tn ->  right; // su must not be NULL
  while ((su -> left) != NULL)  // not (su != NULL)
    {
      su = su -> left;
    }
  // su is tn's immediate successor
  // swap their values
  tn ->  value = su -> value;
  su -> value = val;
  // delete su
  tn ->  right = Tree_delete(tn ->  right, val); // must not be tn
  return tn;
}
```

Homework 11-12

Shuffle Cards

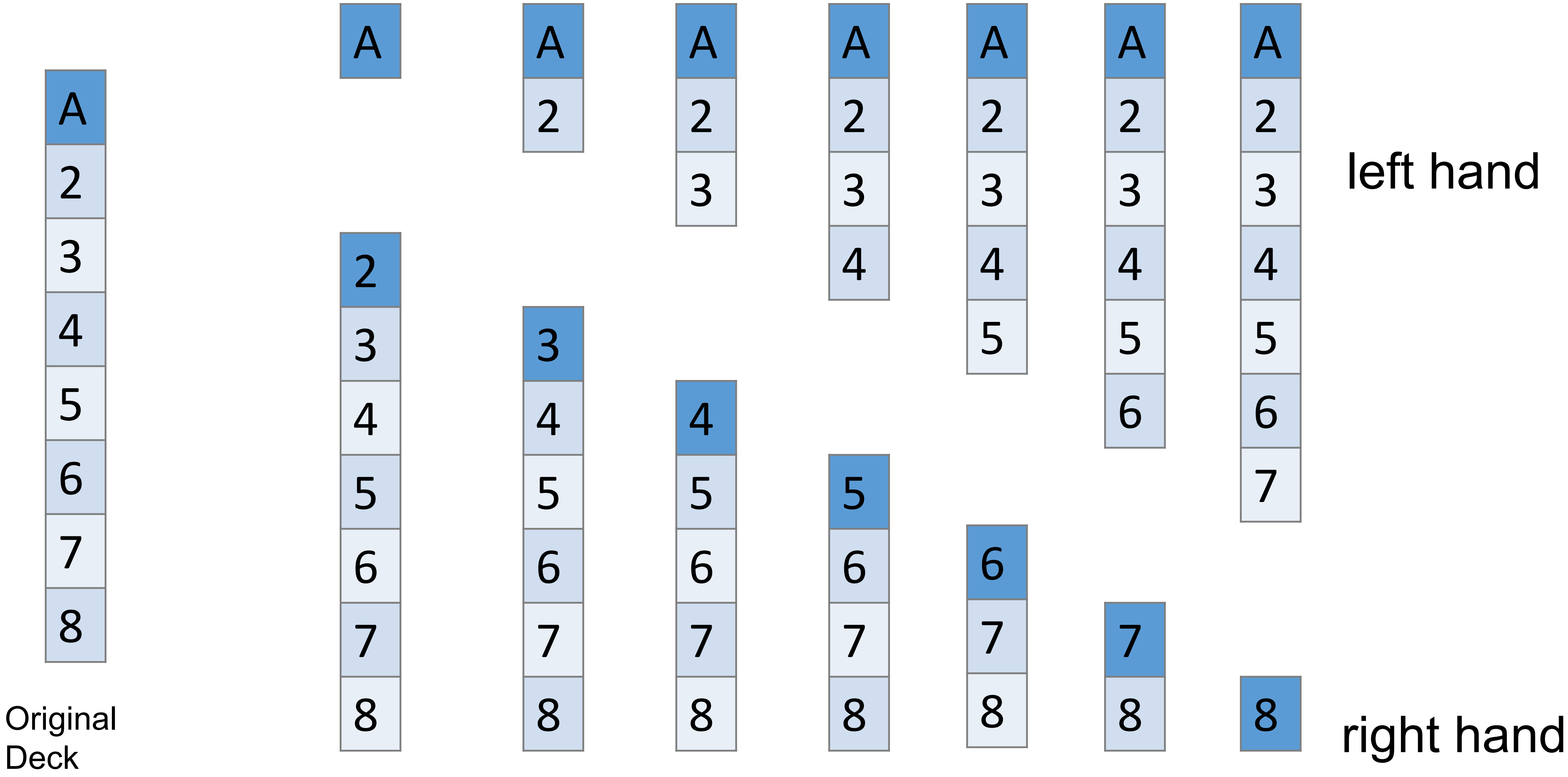
Riffle Shuffling

- Many card games need to shuffle so that players do not know which card may appear next.
- Riffle shuffling is a popular method for shuffling

1. divide a deck of cards into two parts
2. hold the parts by right and left hands
3. interleave the cards

Homework 11-12

- Homework 11: Shuffle Once
- Homework 12: Shuffle Multiple Times
- The homework considers all possible scenarios under these restrictions:
  - Each (of the two) part has at least one card
  - If a card is above another card in one part, the order must be preserved in the interleave result

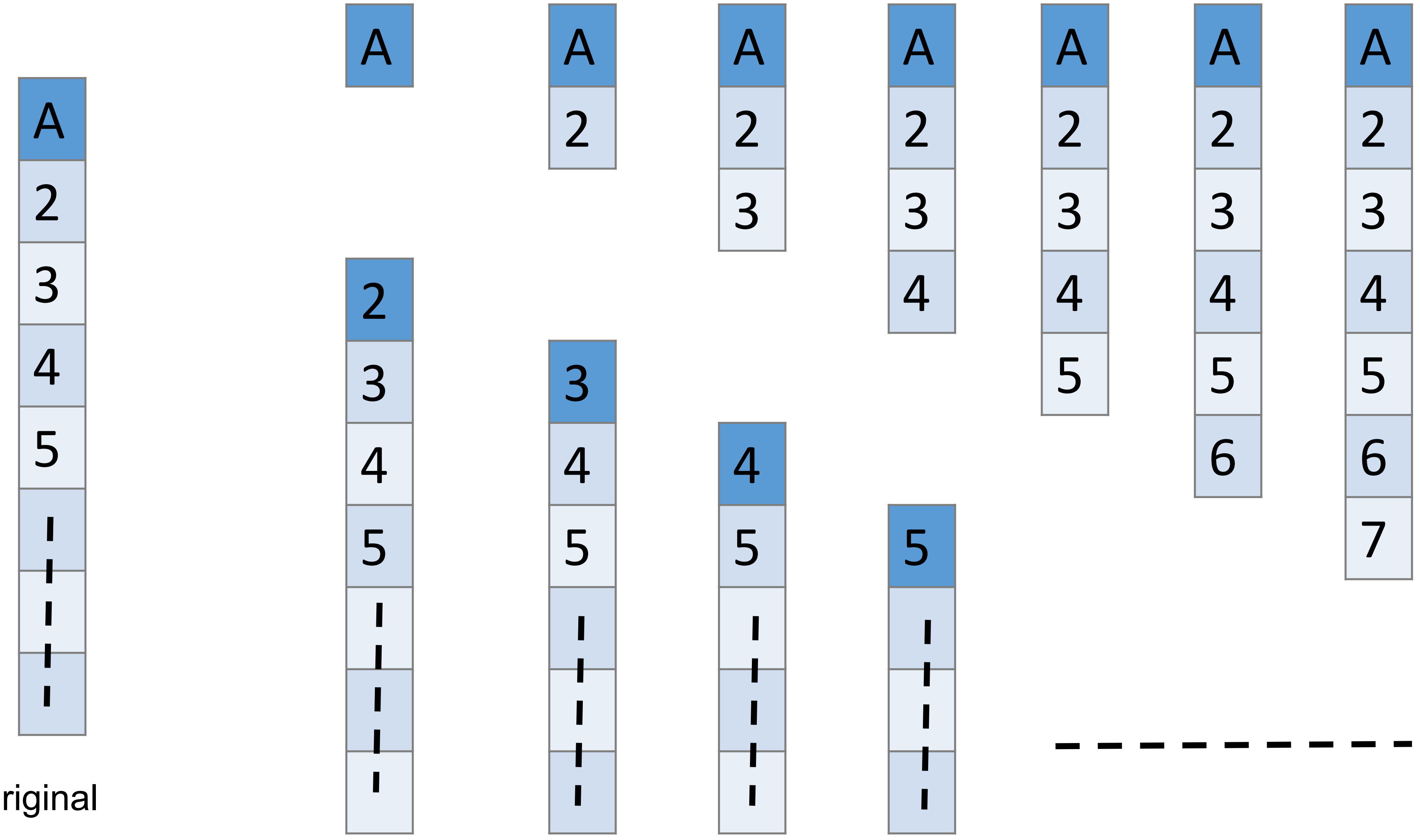


Original Deck

left hand

right hand

Different ways to divide the original deck into two parts



Original

Deck ***n*** cards

Different ways to divide the original deck into two parts

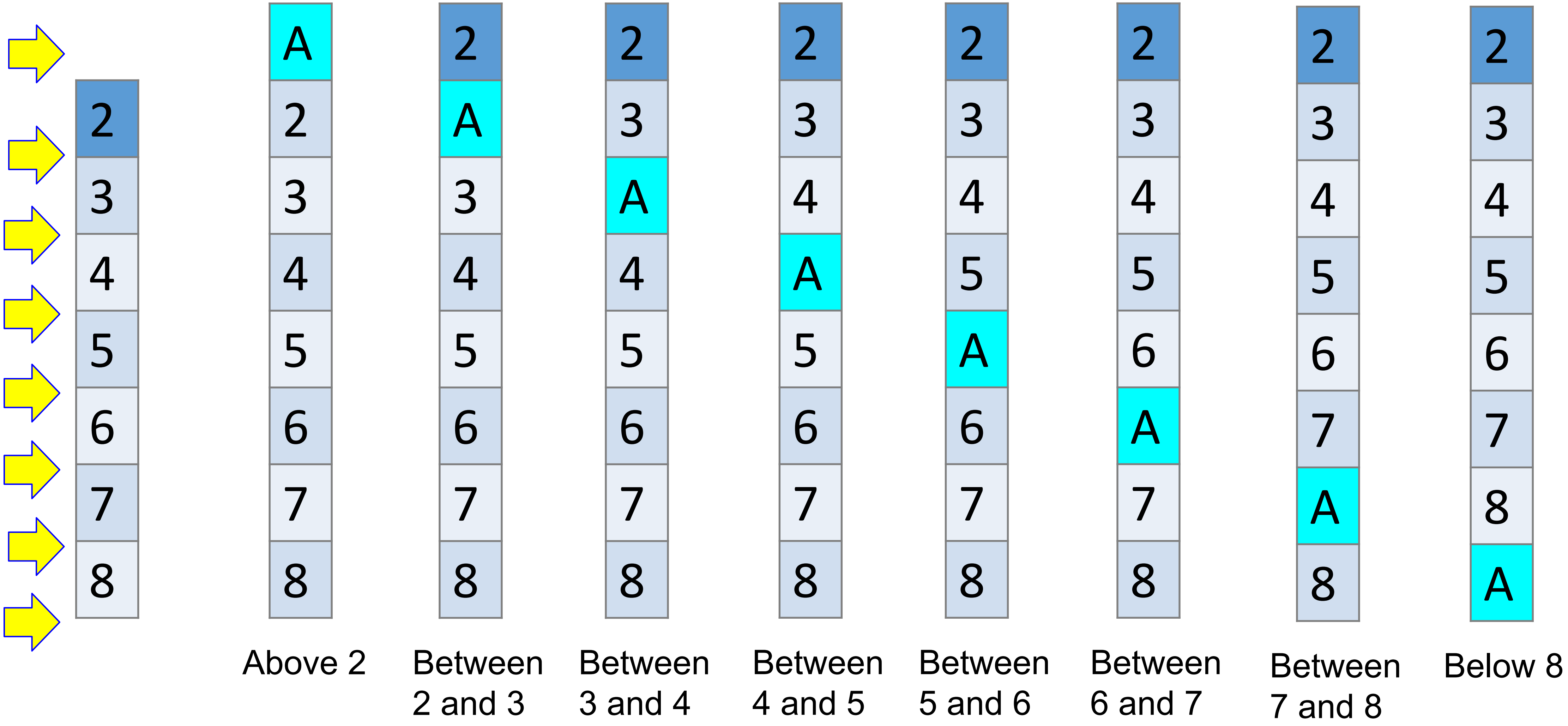
The first part has ***1, 2, 3, …., n – 1*** cards

n – 1 ways to divide the cards

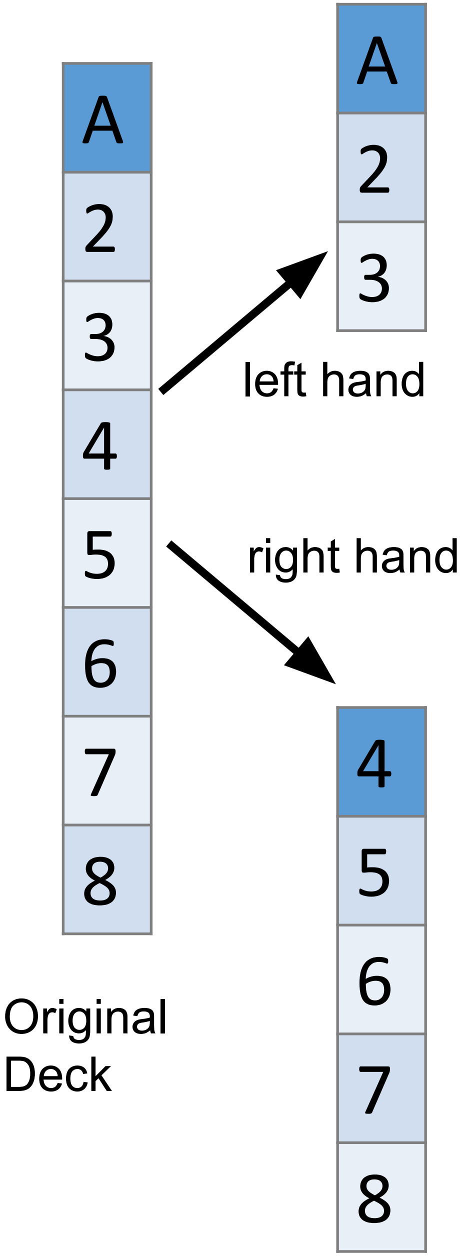
A

Where can A be placed?

The order 2, 3, 4, 5, 6, 7, 8 **will not change**



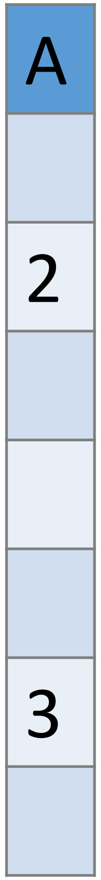
	Above 2	Between 2 and 3	Between 3 and 4	Between 4 and 5	Between 5 and 6	Between 6 and 7	Between 7 and 8	Below 8
	A	2	2	2	2	2	2	2
2	2	A	3	3	3	3	3	3
3	3	3	A	4	4	4	4	4
4	4	4	4	A	5	5	5	5
5	5	5	5	5	A	6	6	6
6	6	6	6	6	6	A	7	7
7	7	7	7	7	7	7	A	8
8	8	8	8	8	8	8	8	A

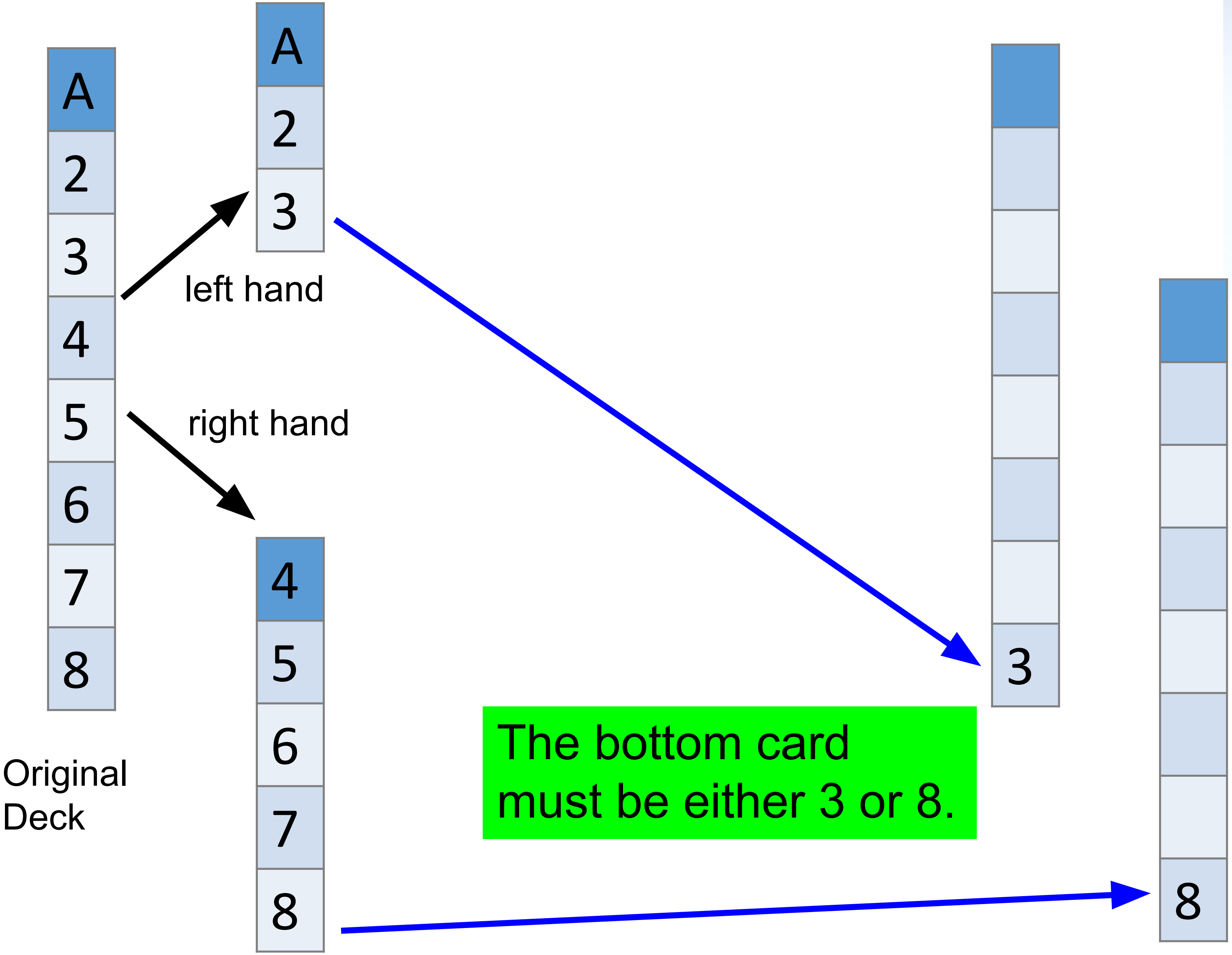


mix the cards

The order A, 2, 3 **will not change**

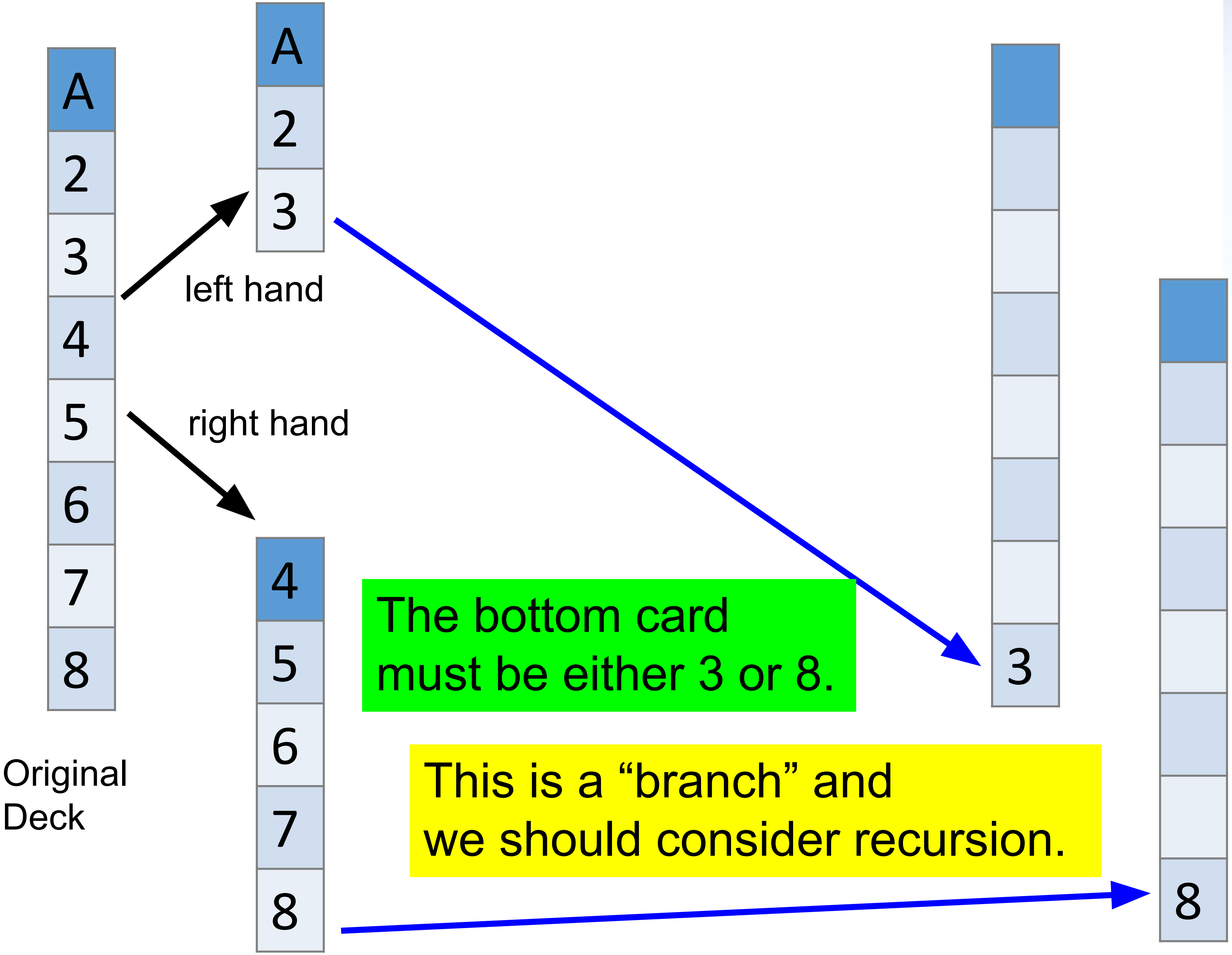
The order 4, 5, 6, 7, 8 **will not change**





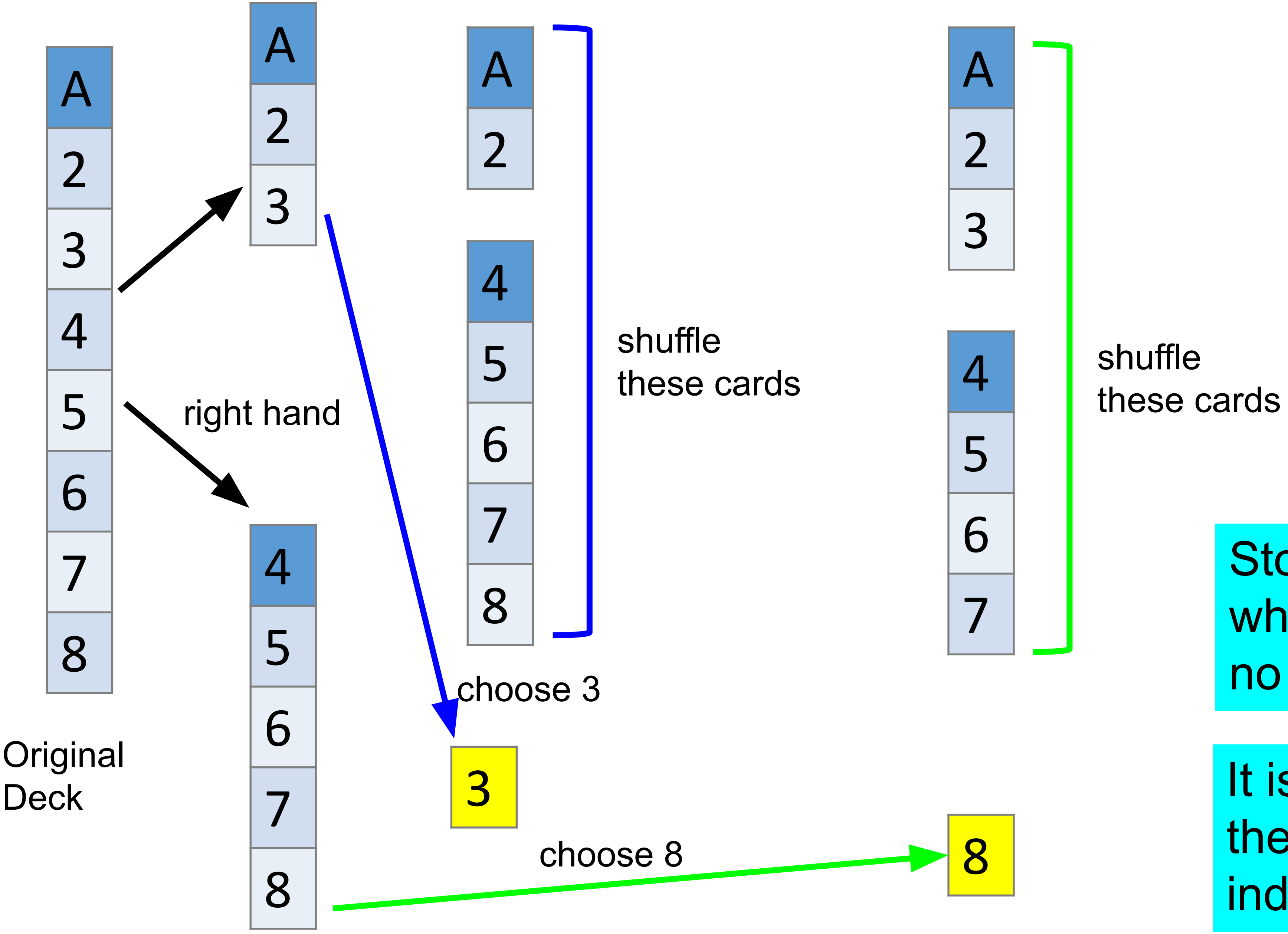
Original Deck

The bottom card must be either 3 or 8.



The bottom card must be either 3 or 8.

This is a “branch” and we should consider recursion.



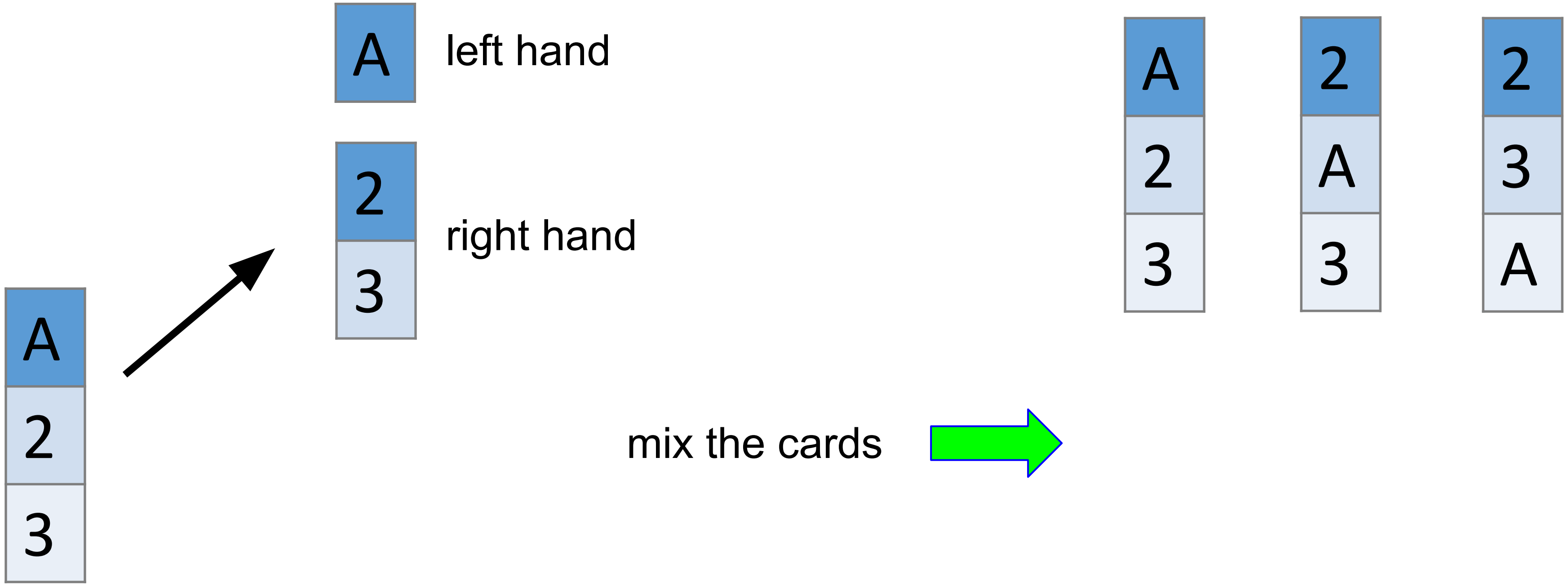
Stop condition: when one hand has no card left

It is easier to start from the bottom. The top's index must be zero

Concept, Not Working Code

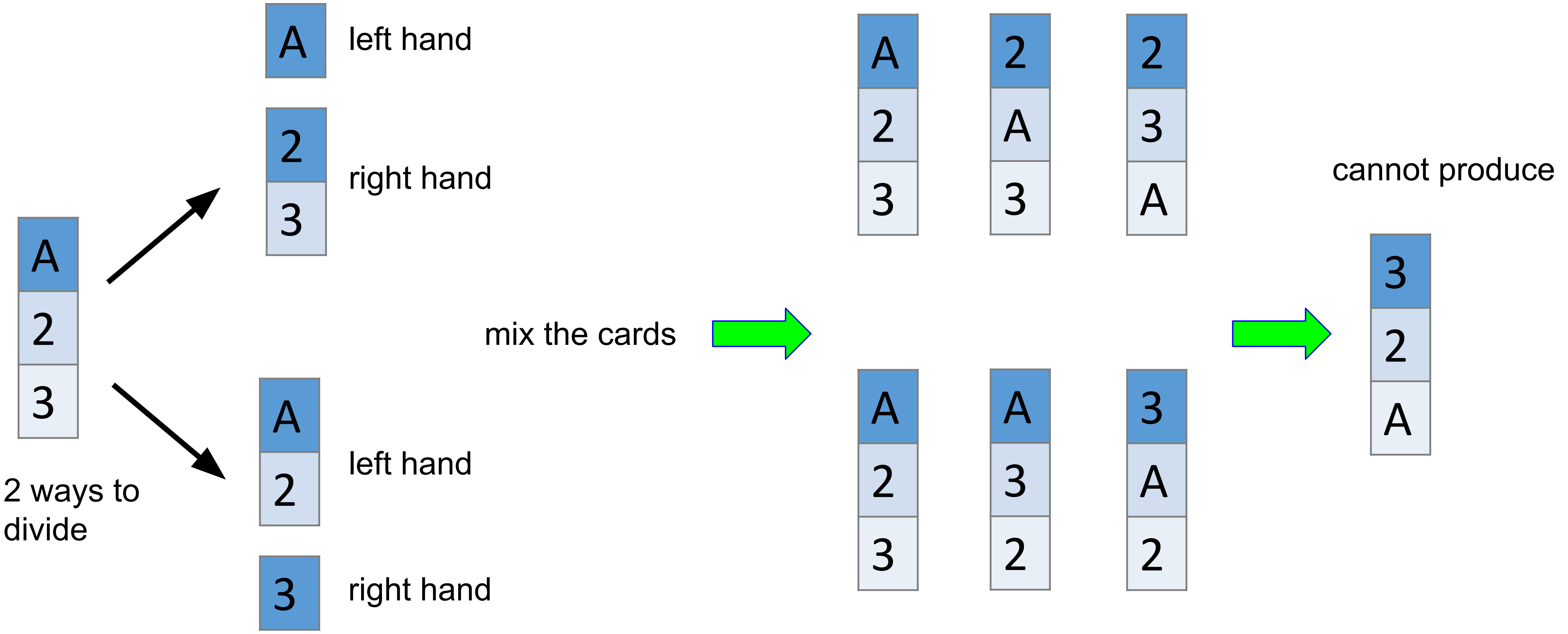
```
interleave(CardDeck leftDeck, CardDeck rightDeck,
     CardDeck destDeck, int leftind, int rightind, int
newind)
{
  // pick one card from left deck
  destDeck.cards[newind] = leftDeck.cards[leftind];
  interleaveHelper(leftDeck, rightDeck, destDeck,
                leftind - 1, rightind, newind - 1);
  // pick one card from right deck
  destDeck.cards[newind] = rightDeck.cards[rightind];
  interleaveHelper(leftDeck, rightDeck, destDeck,
             leftind, rightind - 1, newind - 1);
}
```

Some Orders are Missing



2 ways to divide

Some Orders are Missing



left hand

right hand

left hand

right hand

2 ways to divide

mix the cards

cannot produce

Number of results after shuffling once

- n cards originally: k cards on left hand, n – k cards on right hand
- Interleave n cards and the orders of left hand and right hand must be preserved. $\frac{n!}{k!(n-k)!}$ ways to order these cards.
- k can be 1, 2, …, n - 1$\Rightarrow \sum_{k=1}^{n-1} \frac{n!}{k!(n-k)!}$ ways to shuffle cards
- $(x+y)^n = \sum_{k=0}^{n} \frac{n!}{k!(n-k)!} \ x^k y^{n-k}$
- $x = y = 1 \Rightarrow \sum_{k=0}^{n} \frac{n!}{k!(n-k)!}$ = $2^n \Rightarrow \sum_{k=1}^{n-1} \frac{n!}{k!(n-k)!}$ = $2^n - 2$

Missing Orders

- For n cards, there are n! possible orders.
- Riffle shuffle once can produce only $2^n - 2$ orders (including some repetitions)

n	**2**	**3**	**4**	**5**	**6**
2^n - 2	2	6	14	30	62
n!	2	6	24	120	720

- Please read “Python for Advanced Beginnings” in Brightspace handouts

HW11 and HW12

- When you do HW11, think about HW12.
- **You must design before coding.** If you start coding without design, you will not finish.
- The sample solution for HW12 has 124 lines, including blank lines and comments. If your solution has more than 500 lines, it is probably wrong.
- You must think before writing code.

Recursion Practice

- Find all subsets of length k in a given array of length n:
- Array: 2 3 -1 4; Subsets of length: 2

2 3

2 4

2 -1

3 -1

3 4

-1 4

All permutations of a given array

- Find all permutations of a given array of length n:
- Array: 2 3

    2 3

    3 2

Array: 4 5 -1:

4 5 -1

5 4 -1

-1 4 5

….