Binary Search Trees

Department of Electrical and Computer Engineering
McMaster University

January 2020

Textbook Reading

- Chapter 19, Sections: 19.1, 19.3

Searching

- One of the main operations of computer systems is to store and process vast quantities of information or **data**
  - → The world-wide web (WWW) is a vast storage network of billions of web pages.
  - → Bank accounts
  - → Social Insurance Numbers
  - → the simple telephone book!
- Simply storing items is not enough, need to have a way to **efficiently** find, access and modify these records.

Searching

- One of the main operations of computer systems is to store and process vast quantities of information or **data**
  - → The world-wide web (WWW) is a vast storage network of billions of web pages.
  - → Bank accounts
  - → Social Insurance Numbers
  - → the simple telephone book!
- Simply storing items is not enough, need to have a way to **efficiently** find, access and modify these records.

Question: What data structures and algorithms can we develop to:

- have efficient representation of the data
- to permit quick insertions and deletions
- to permit quick searches for specific elements in the data set.

Answer: Some techniques we will consider are ...

- Binary Search
- Binary Search Trees
- Hashing

Note: Each technique has strengths and weaknesses which we will discuss.

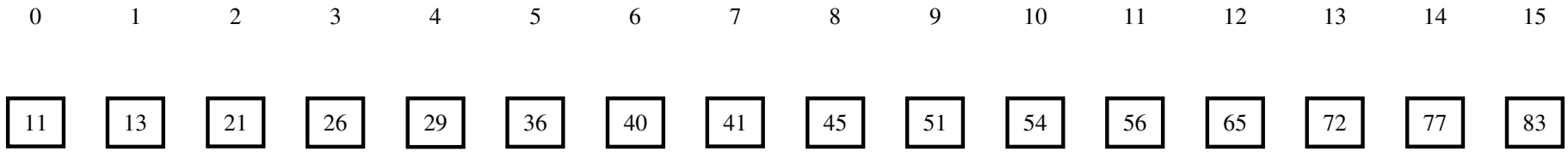
Answer: Some techniques we will consider are ...

- Binary Search
  - → Search: $\Theta(\log n)$, Insertions/Deletions: $\Theta(n)$ (average and worst case)
- Binary Search Trees
  - → Search: $\Theta(\log n)$, Insertions/Deletions: $\Theta(\log n)$ (average case).
- Hashing
  - → Search: $\Theta(1)$, Insertions/Deletions: $\Theta(1)$ (average case).

Note: Each technique has strengths and weaknesses which we will discuss.

Binary Search

Assume we have an array of sorted integers in memory.

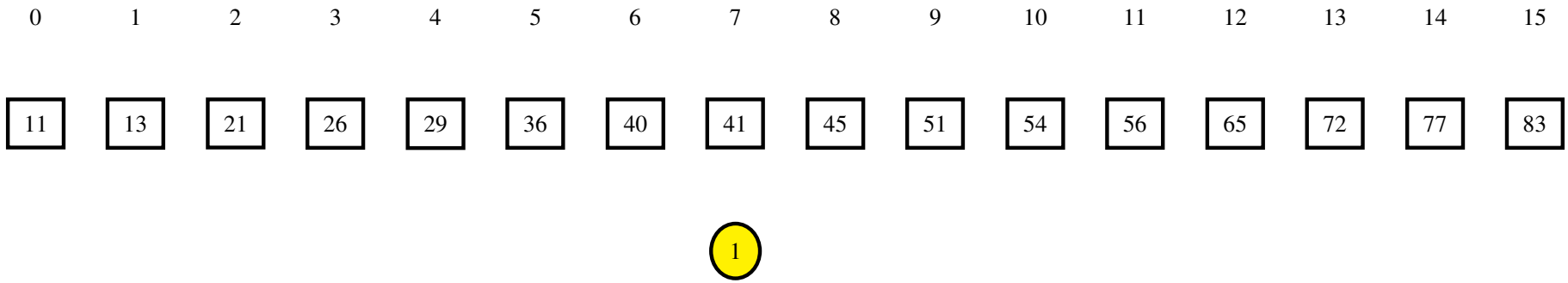


Search for `K` in a pre-sorted list of length `n`.

```
1 int l = left-1; int r = right+1;   /* l and r are beyond array bounds */
2 while (l+1 != r) {                 /* loop until l and r meet */
3     int i = (l+r)/2;               /* find middle of sub-list */
4     if (K < array[i]) r = i;       /* Value in left half */
5     if (K == array[i]) return i;   /* Found it! */
6     if (K > array[i]) l = i;       /* Value in right half */
7 } return UNSUCCESSFUL;             /* Not in array */
```

Binary Search

Assume we have an array of sorted integers in memory.

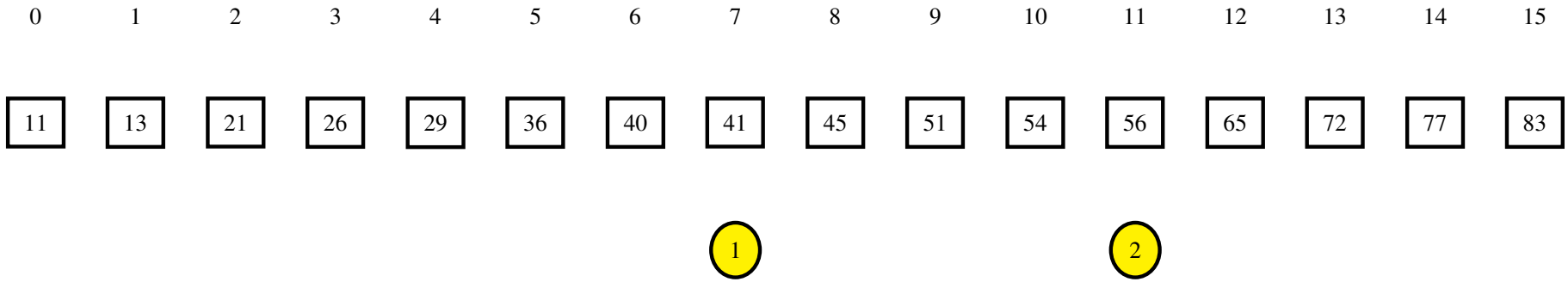


Search for K in a pre-sorted list of length n. **Example:** Search for K=45.

```
1  int l = left-1; int r = right+1;   /* l and r are beyond array bounds */
2  while (l+1 != r) {                 /* loop until l and r meet */
3      int i = (l+r)/2;               /* find middle of sub-list */
4      if (K < array[i]) r = i;       /* Value in left half */
5      if (K == array[i]) return i;   /* Found it! */
6      if (K > array[i]) l = i;       /* Value in right half */
7  } return UNSUCCESSFUL;             /* Not in array */
```

Binary Search

Assume we have an array of sorted integers in memory.

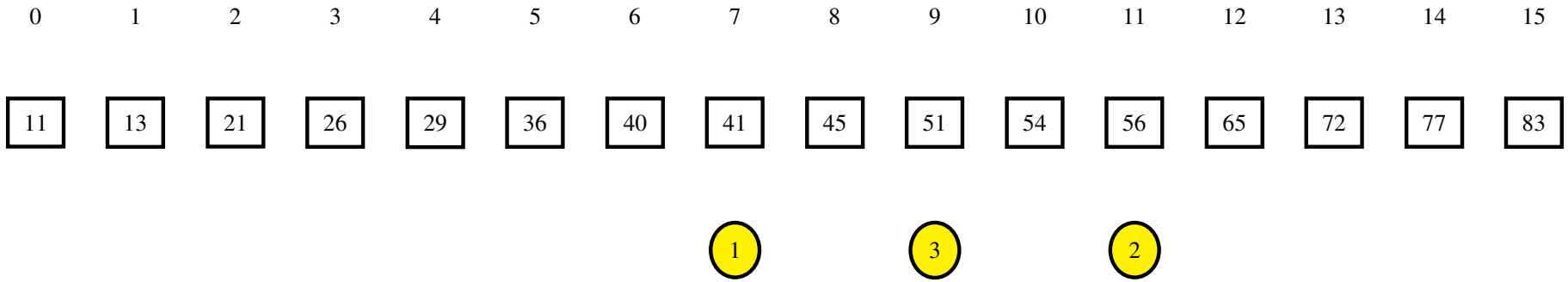


Search for K in a pre-sorted list of length n. **Example:** Search for K=45.

```
1  int l = left-1; int r = right+1;   /* l and r are beyond array bounds */
2  while (l+1 != r) {                 /* loop until l and r meet */
3      int i = (l+r)/2;               /* find middle of sub-list */
4      if (K < array[i]) r = i;       /* Value in left half */
5      if (K == array[i]) return i;   /* Found it! */
6      if (K > array[i]) l = i;       /* Value in right half */
7  } return UNSUCCESSFUL;             /* Not in array */
```

Binary Search

Assume we have an array of sorted integers in memory.

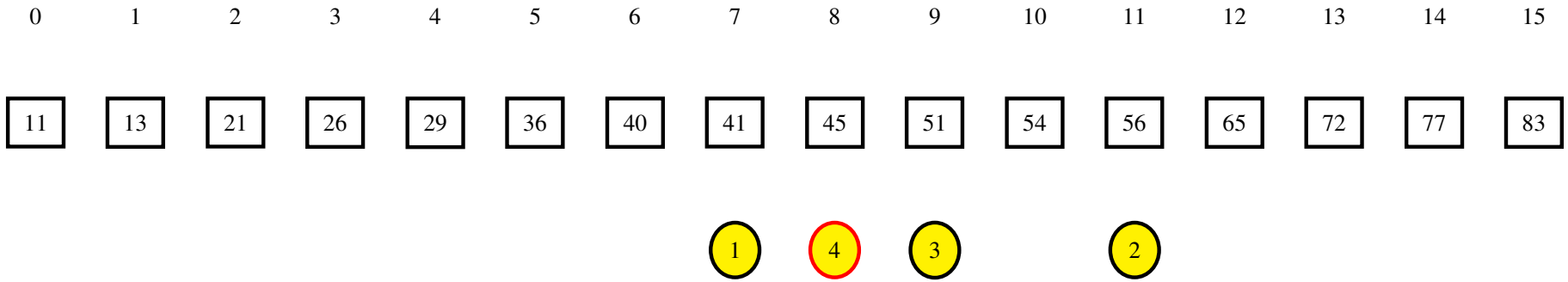


Search for K in a pre-sorted list of length n. **Example:** Search for K=45.

```
1  int l = left-1; int r = right+1;   /* l and r are beyond array bounds */
2  while (l+1 != r) {                 /* loop until l and r meet */
3      int i = (l+r)/2;               /* find middle of sub-list */
4      if (K < array[i]) r = i;       /* Value in left half */
5      if (K == array[i]) return i;   /* Found it! */
6      if (K > array[i]) l = i;       /* Value in right half */
7  } return UNSUCCESSFUL;             /* Not in array */
```

Binary Search

Assume we have an array of sorted integers in memory.



Search for K in a pre-sorted list of length n. **Example:** Search for K=45.

```
1 int l = left-1; int r = right+1;   /* l and r are beyond array bounds */
2 while (l+1 != r) {                 /* loop until l and r meet */
3     int i = (l+r)/2;               /* find middle of sub-list */
4     if (K < array[i]) r = i;       /* Value in left half */
5     if (K == array[i]) return i;   /* Found it! */
6     if (K > array[i]) l = i;       /* Value in right half */
7 } return UNSUCCESSFUL;             /* Not in array */
```

Binary Search

- **Searching** At each iteration of the `while` loop, **half** of the remaining elements in the list are discarded. Run time (average and worst case):

$$\Theta(\log n)$$

- **Insertion/Deletion** Inserting into a sorted array list has runtime $\Theta(n)$ (average and worst case).
  - → $\Theta(\log n)$ to find the spot
  - → $\Theta(n)$ to "shuffle" elements.

Binary Search

- **Searching** At each iteration of the `while` loop, **half** of the remaining elements in the list are discarded. Run time (average and worst case):

$$\Theta(\log n)$$

- **Insertion/Deletion** Inserting into a sorted array list has runtime $\Theta(n)$ (average and worst case).
  - → $\Theta(\log n)$ to find the spot
  - → $\Theta(n)$ to "shuffle" elements.
- ⇨ **Advantage** Searching is very fast
- ⇨ **Disadvantage** Insertion and deletion are slow.
- ⇨ This technique is good for applications which need many searches, but few insertions/deletions:
  - → Bank accounts, Telephone book, Dictionary

Binary Search

- **Searching** At each iteration of the `while` loop, **half** of the remaining elements in the list are discarded. Run time (average and worst case):

$$\Theta(\log n)$$

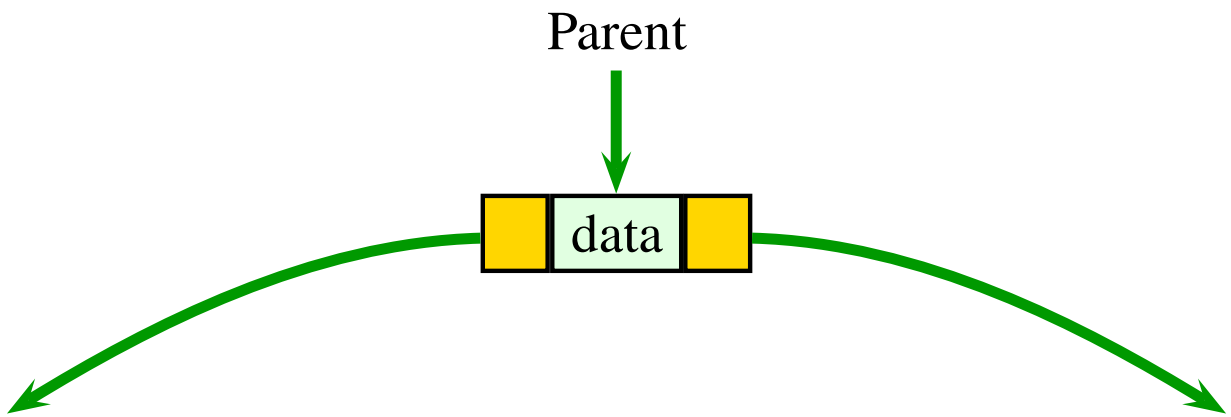
- **Insertion/Deletion** Inserting into a sorted array list has runtime $\Theta(n)$ (average and worst case).
  - → $\Theta(\log n)$ to find the spot
  - → $\Theta(n)$ to "shuffle" elements.
- ⇨ **Advantage** Searching is very fast
- ⇨ **Disadvantage** Insertion and deletion are slow.
- ⇨ This technique is good for applications which need many searches, but few insertions/deletions:
  - → Bank accounts, Telephone book, Dictionary
- ☞ Note that this is much better than the $\Theta(n)$ performance of linear search on the same list.
- ☞ **Tradeoff** between ease of inserting new elements and searching!

Assumptions for average run time:

- When computing the average case running time it is assumed that:
  - for successful search - all values in the array are equally likely to match the input key;
  - for insertion - all positions in the array are equally likely to be the position where the new item has to be inserted.

Definition: A **binary search tree** (BST) is a binary tree in which data values stored at each node have been ordered such that for any node in the tree:

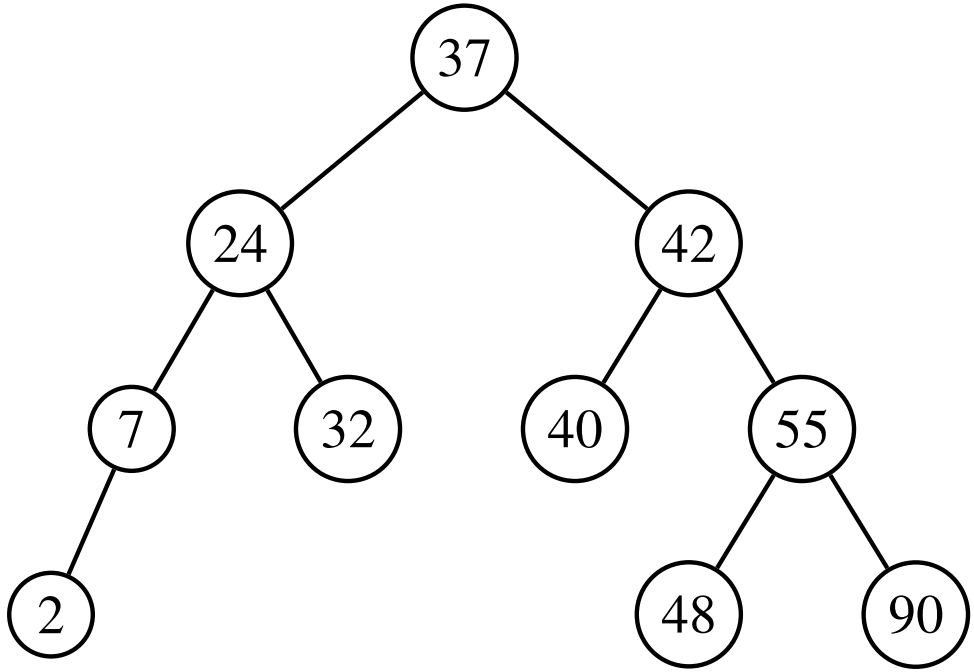
- → elements in the left sub-tree have value less than the value in the parent
- → elements in the right sub-tree have value greater than the value in the parent



BST with elements < `data`

BST with elements > `data`

Example:



- Searching for a value in this tree is like the game **20-questions** where at each node you ask the question

  Is *search value* < *node value* **?**

  → if the answer is YES, follow the left branch

  → if the answer is NO, follow the right branch

  Continue until the value is found or you fall off the tree.
- The number of questions you need to ask is at **most** the height of the tree plus 1.

Binary Search Tree Operations

- `createTree` - creates a new empty tree.
- `isEmptyTree` - returns `TRUE` if the tree is empty, otherwise returns `FALSE`.
- `isLeaf` - returns `TRUE` if the node is a leaf node, otherwise `FALSE`.
- `destroyTree` - deallocates the memory of the tree.
- `isInTree` - searches a tree for a given value and returns `true` if found, `false` otherwise.
- `findTree` - searches a tree for a given value. If found, returns a reference to the object stored, otherwise `null`.
- `findTreeMin` - returns the minimum value in the tree.
- `findTreeMax` - returns the maximum value in the tree.
- `insertTree` - inserts a value into a tree.
- `deleteTree` - deletes the node with a given value if in the tree, otherwise do nothing.

Binary Search Trees

- There are numerous ways to implement the data structures and operations of the binary search ADT
  - Array based implementation
  - Linked implementation
- Here we present the algorithms for the linked implementation only.[1]

[1] See text Chapter 19

☞ Use the following class to define each Tree Node:

```
1  class TNode<E>{
2      E element;
3      TNode<E> left;
4      TNode<E> right;
5
6      public TNode(E e, TNode<E> l, TNode<E> r){
7          element=e;
8          left=l;
9          right=r;
10     }
11 }
```

Binary Search Trees

McMaster University

☞ Use the following class to define the Tree ADT:

```
1  public class LLTree<E> {
2      protected TNode<E> root;
3      public LLTree(){ ... }
4      public boolean isEmptyTree(){ ... }
5      private boolean isLeaf(TNode<E> n){ ... }
6      public boolean printTree(){ return printTree(root); }
7      private boolean printTree(TNode n){ ... }
8      public boolean isInTree(E e){ return findTree(e,root)!=null;
9      public E findTree(E e){ return findTree(e,root); }
10     private E findTree(E e, TNode t){ ... }
11     public E findTreeMin(){ ... }
12     public E findTreeMax(){ ... }
13     public void insertTree(E e){ root = insertTree(e,root); }
14     private TNode<E> insertTree(E e, TNode<E> t){ ... }
15     public void deleteTree(E e){ deleteTree(e,root); }
16     private TNode<E> deleteTree(E e, TNode<E> t){ ... }
17 }
```

```
1 public LLTree(){
2     root = null;
3 }
```

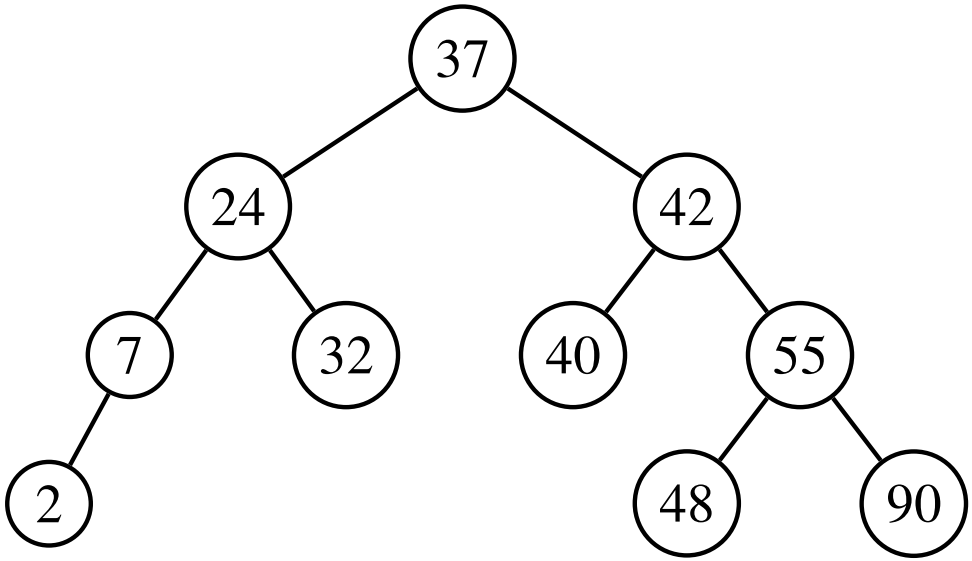
```
1 public boolean isEmptyTree(){
2         return(root == null);
```

```
1 private boolean isLeaf(TNode<E> n){
2     return(n!=null && n.left==null && n.right==null);
3 }
```

- → A tree is represented by a reference to the root node.
- → A tree is initialized by setting the root reference to null.
- → Leaf nodes have both left and right pointers set to null.
- → All operations run in $\Theta(1)$ time.

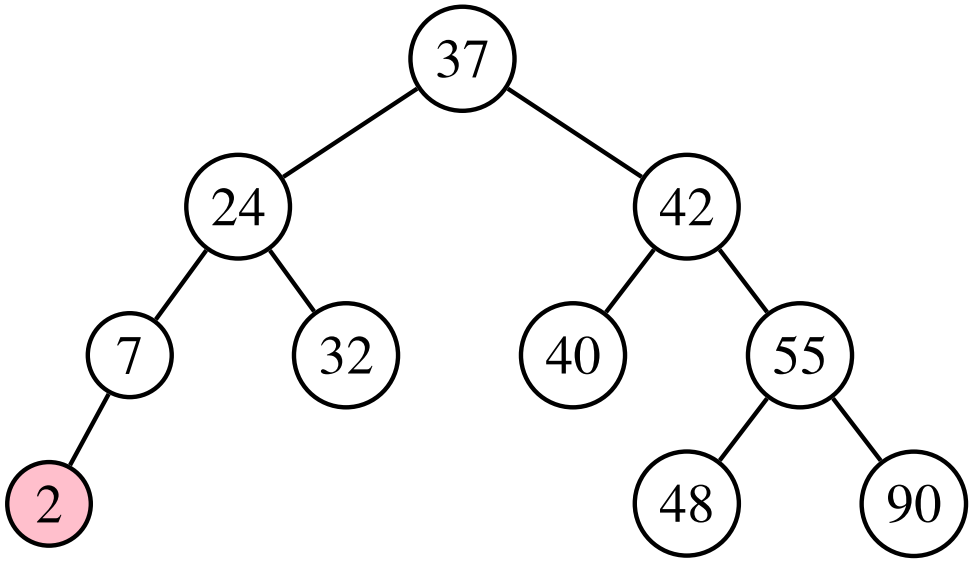
Binary Search Trees - printTree

```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```



Binary Search Trees - printTree

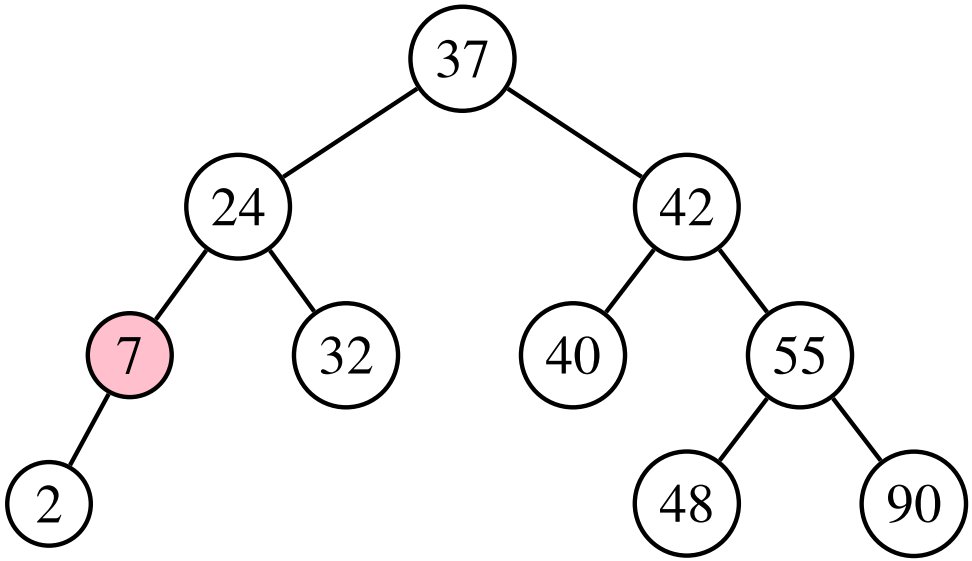
```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```



Output: 2,

Binary Search Trees - printTree

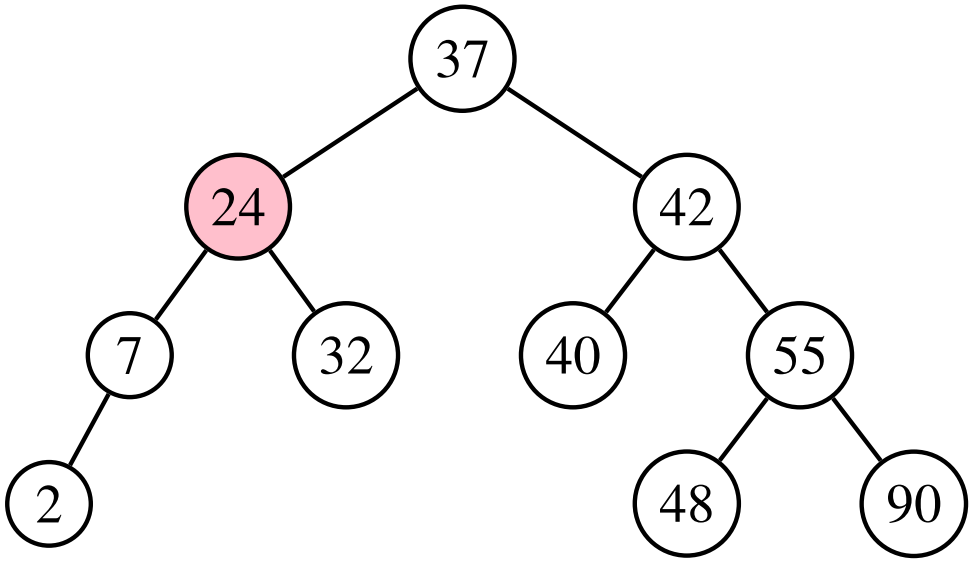
```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```



Output: 2, 7

Binary Search Trees - printTree

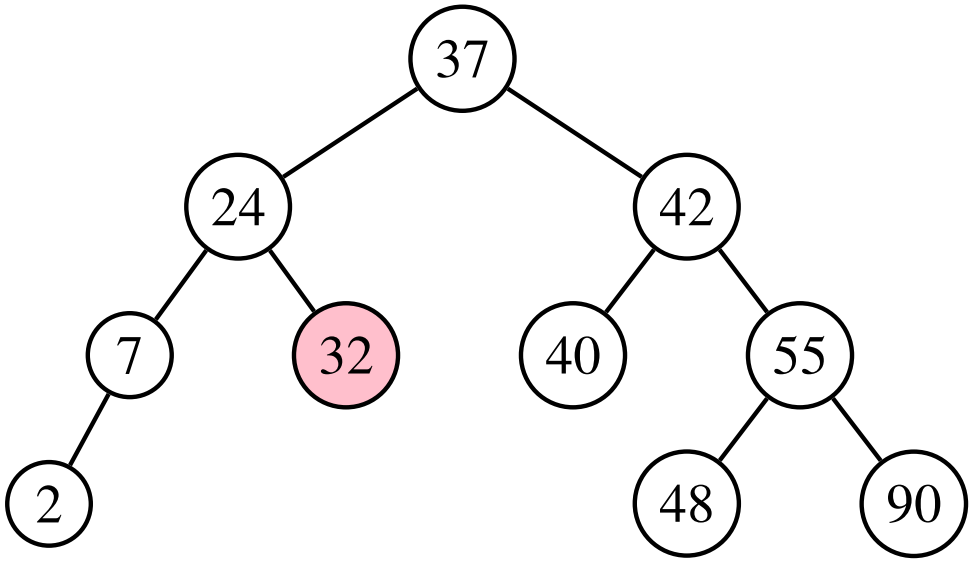
```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```



Output: 2, 7, 24,

Binary Search Trees - printTree

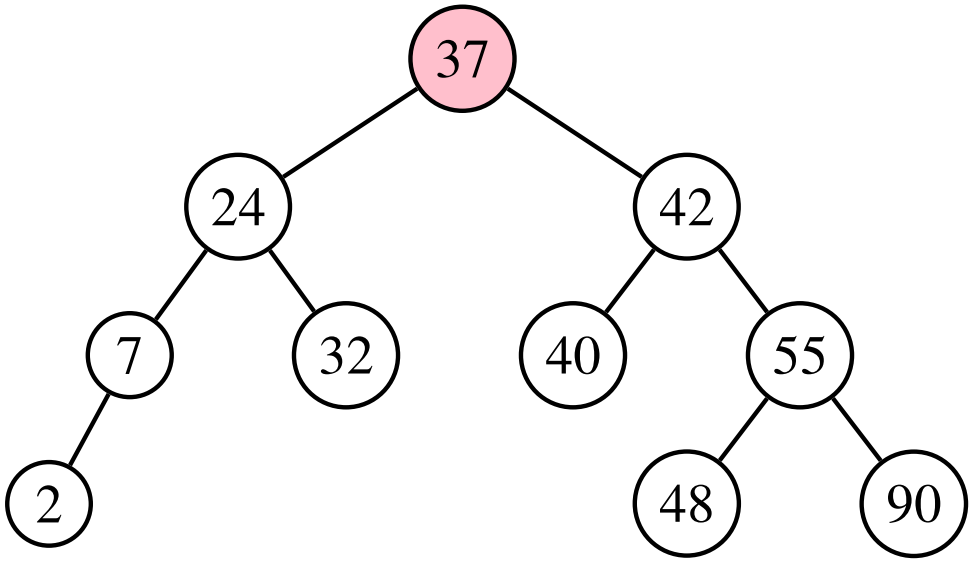
```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```



Output: 2, 7, 24, 32,

Binary Search Trees - printTree

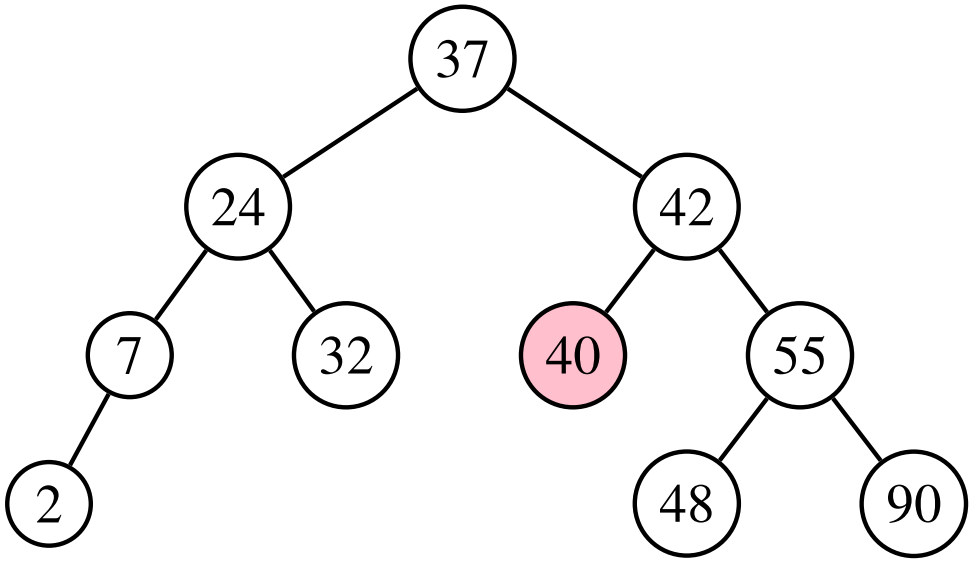
```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```



Output: 2, 7, 24, 32, 37,

Binary Search Trees - printTree

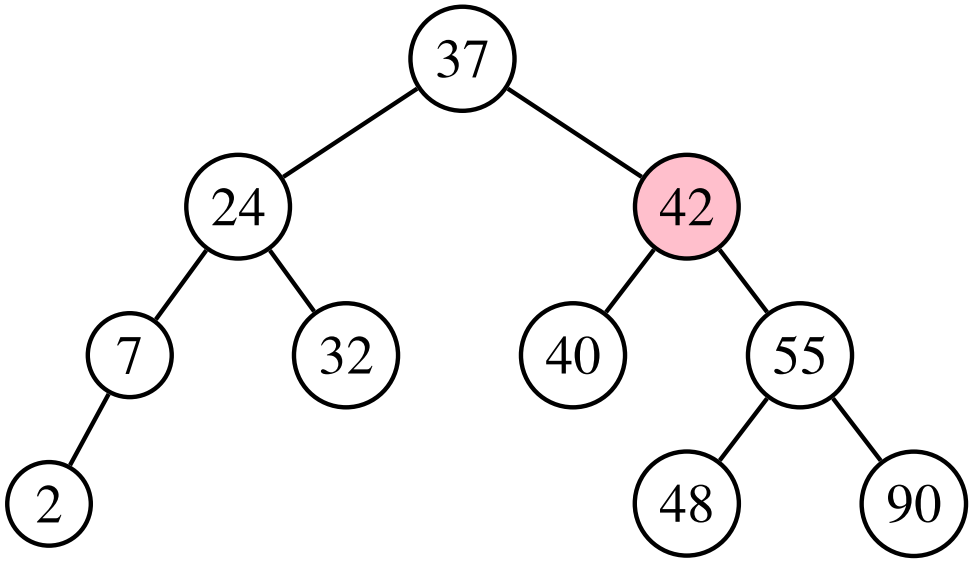
```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```



Output: 2, 7, 24, 32, 37, 40,

Binary Search Trees - printTree

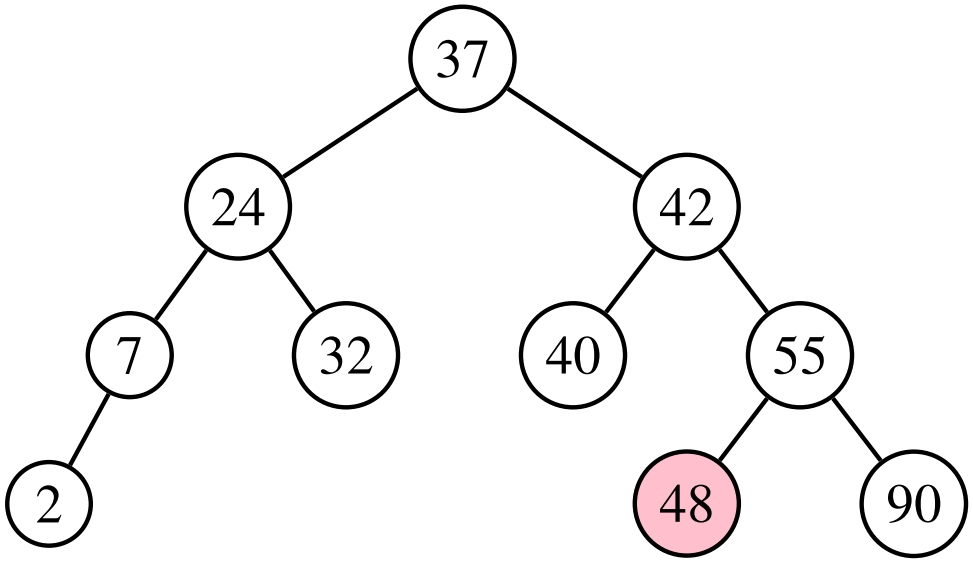
```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```



Output: 2, 7, 24, 32, 37, 40, 42,

Binary Search Trees - printTree

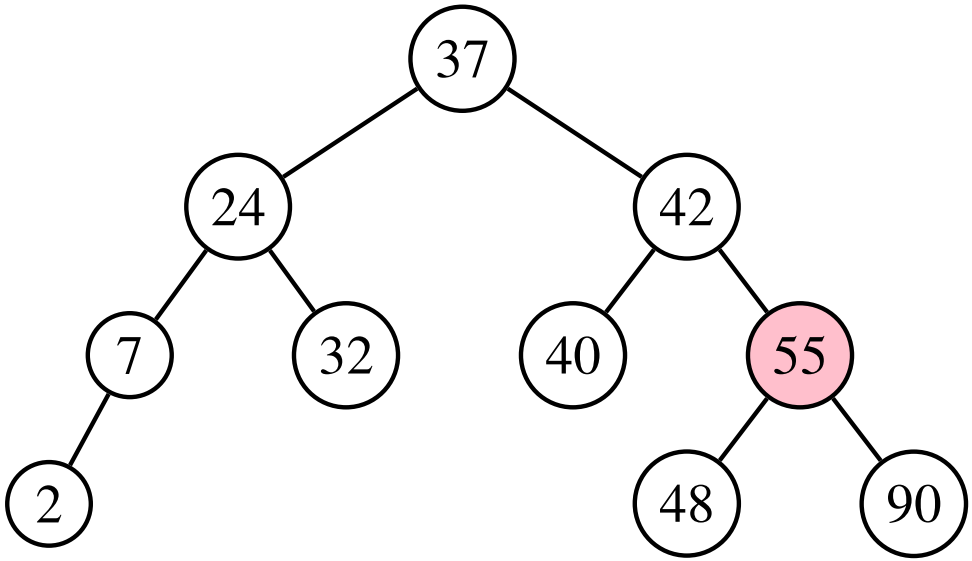
```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```



Output: 2, 7, 24, 32, 37, 40, 42, 48,

Binary Search Trees - printTree

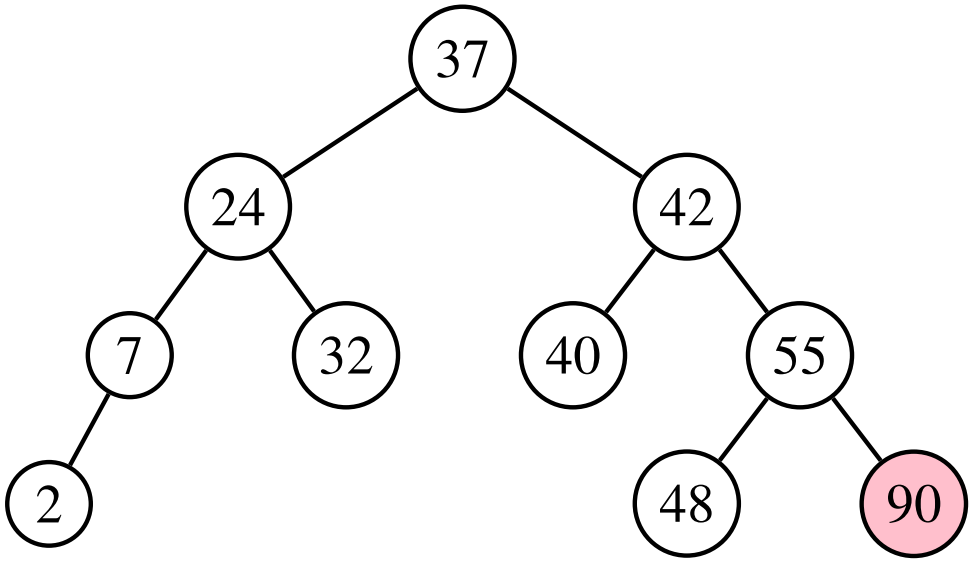
```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```



Output: 2, 7, 24, 32, 37, 40, 42, 48, 55,

Binary Search Trees - printTree

```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```



Output: 2, 7, 24, 32, 37, 40, 42, 48, 55, 90.

Binary Search Trees - printTree

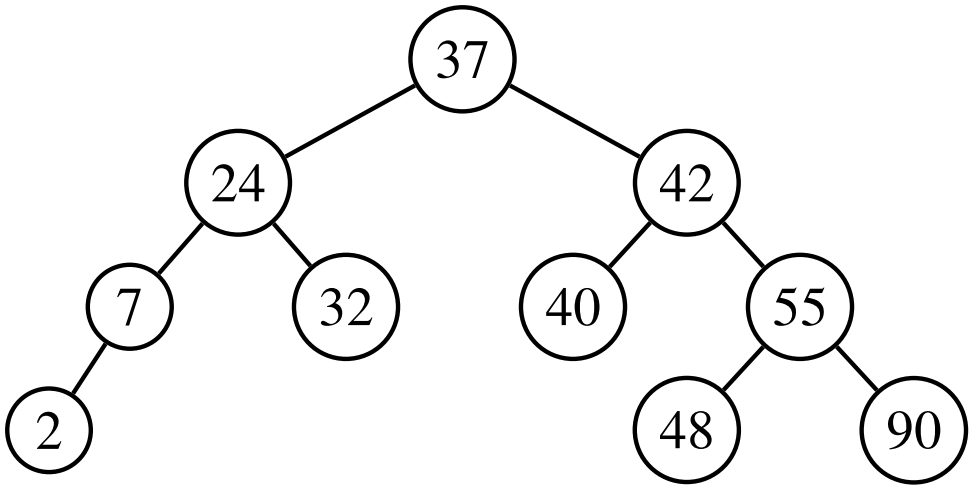
```
1 private void printTree(TNode<E> n){
2     if(n!=null){
3         printTree(n.left);
4         System.out.println(n.element);
5         printTree(n.right);
6     }
7 }
```

- In-Order traversal.
- Outputs the data in increasing order.
- Running time: $\Theta(n)$, where n is the number of tree nodes
  - → Visits every node and every `null` link in the tree and performs a constant number of operations at each node

Binary Search Trees - findTree

```
1  private E findTree(E e, TNode<E> t){
2      if(t==null)
3          return(null);
4      if(e.compareTo(t.element)<0)
5          return(findTree(e,t.left));
6      if(e.compareTo(t.element)>0)
7          return(findTree(e,t.right));
8      else
9          return(t.element);
10 }
```

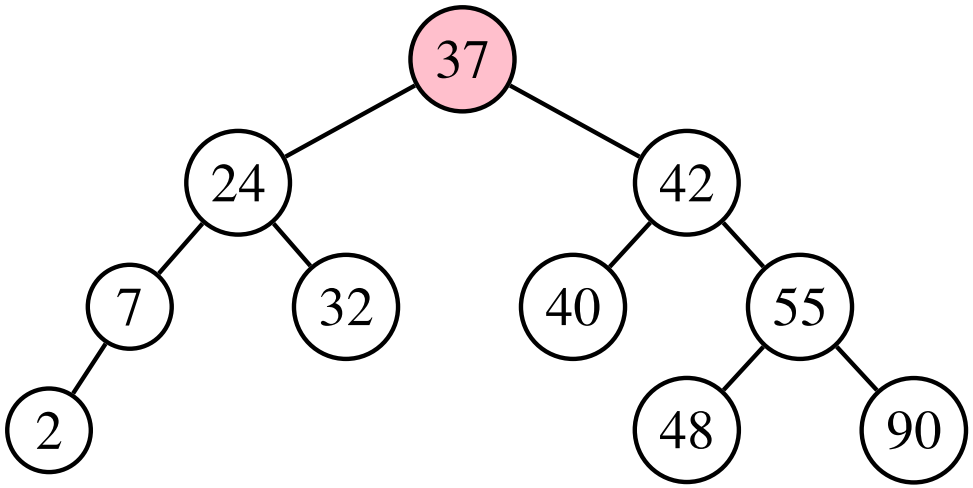
Example: findTree(new Integer(48),root)



Binary Search Trees - findTree

```
1  private E findTree(E e, TNode<E> t){
2      if(t==null)
3          return(null);
4      if(e.compareTo(t.element)<0)
5          return(findTree(e,t.left));
6      if(e.compareTo(t.element)>0)
7          return(findTree(e,t.right));
8      else
9          return(t.element);
10 }
```

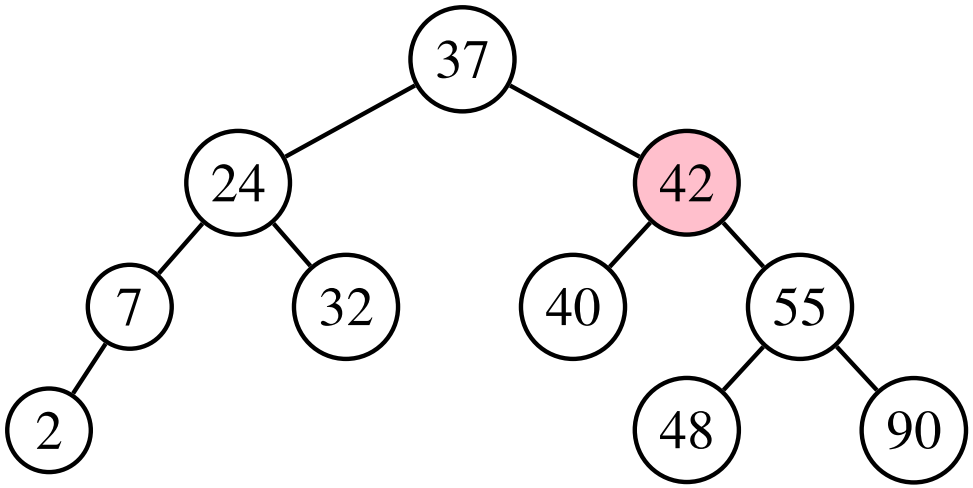
Example: findTree(48,t)



Binary Search Trees - findTree

```
1  private E findTree(E e, TNode<E> t){
2      if(t==null)
3          return(null);
4      if(e.compareTo(t.element)<0)
5          return(findTree(e,t.left));
6      if(e.compareTo(t.element)>0)
7          return(findTree(e,t.right));
8      else
9          return(t.element);
10 }
```

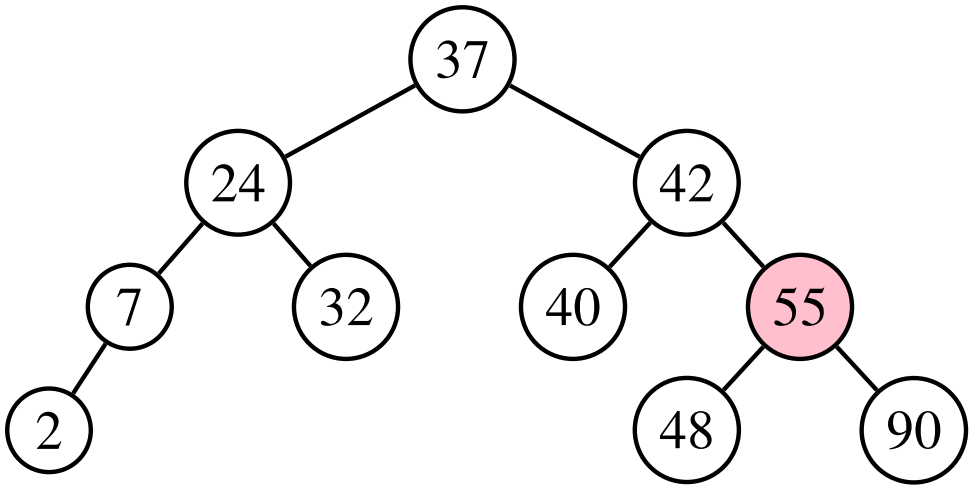
Example: findTree(48,t)



Binary Search Trees - findTree

```
1  private E findTree(E e, TNode<E> t){
2      if(t==null)
3          return(null);
4      if(e.compareTo(t.element)<0)
5          return(findTree(e,t.left));
6      if(e.compareTo(t.element)>0)
7          return(findTree(e,t.right));
8      else
9          return(t.element);
10 }
```

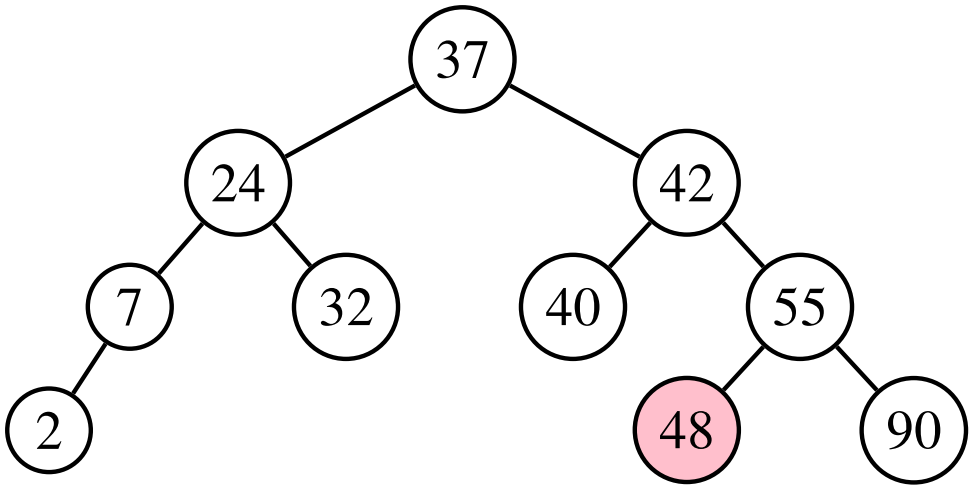
Example: findTree(48,t)



Binary Search Trees - findTree

```
1  private E findTree(E e, TNode<E> t){
2      if(t==null)
3          return(null);
4      if(e.compareTo(t.element)<0)
5          return(findTree(e,t.left));
6      if(e.compareTo(t.element)>0)
7          return(findTree(e,t.right));
8      else
9          return(t.element);
10 }
```

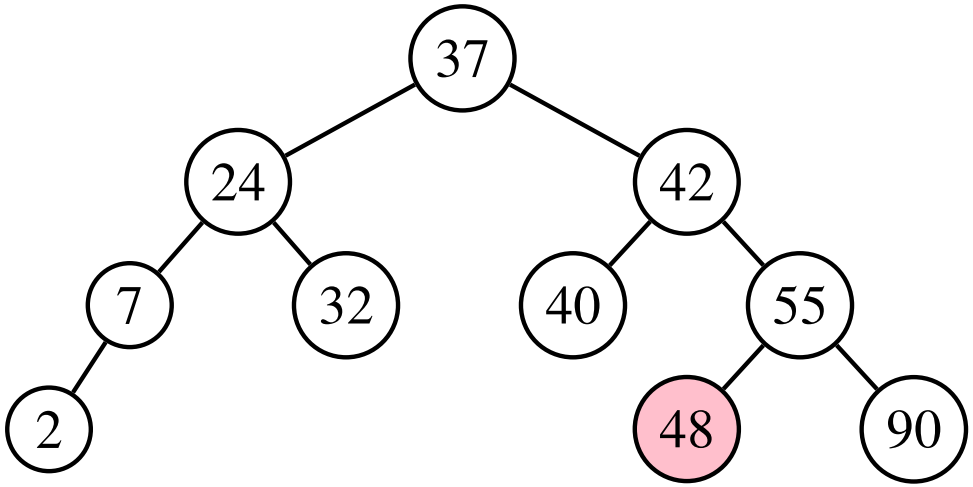
Example: findTree(48,t)



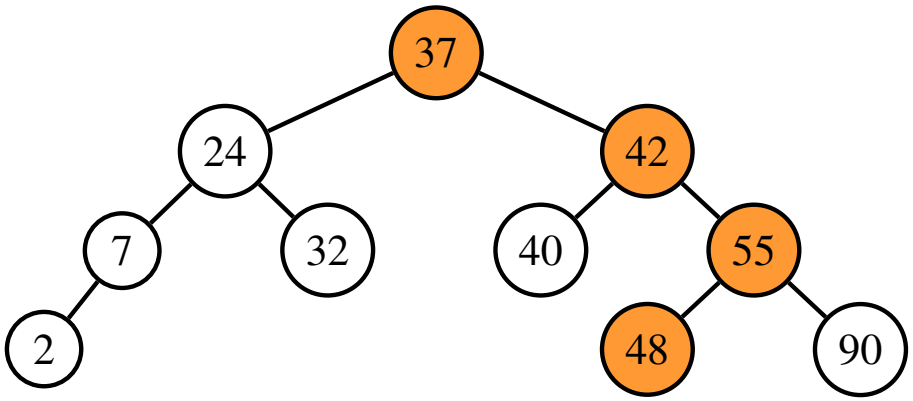
Binary Search Trees - findTree

```
1  private E findTree(E e, TNode<E> t){
2      if(t==null)
3          return(null);
4      if(e.compareTo(t.element)<0)
5          return(findTree(e,t.left));
6      if(e.compareTo(t.element)>0)
7          return(findTree(e,t.right));
8      else
9          return(t.element);
10 }
```

Example: findTree(48,t) – **found** in 4 calls.

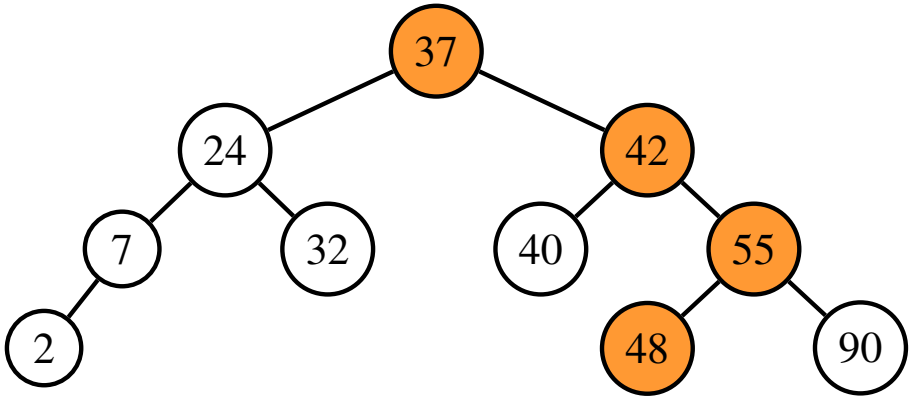


Run Time of `findTree`



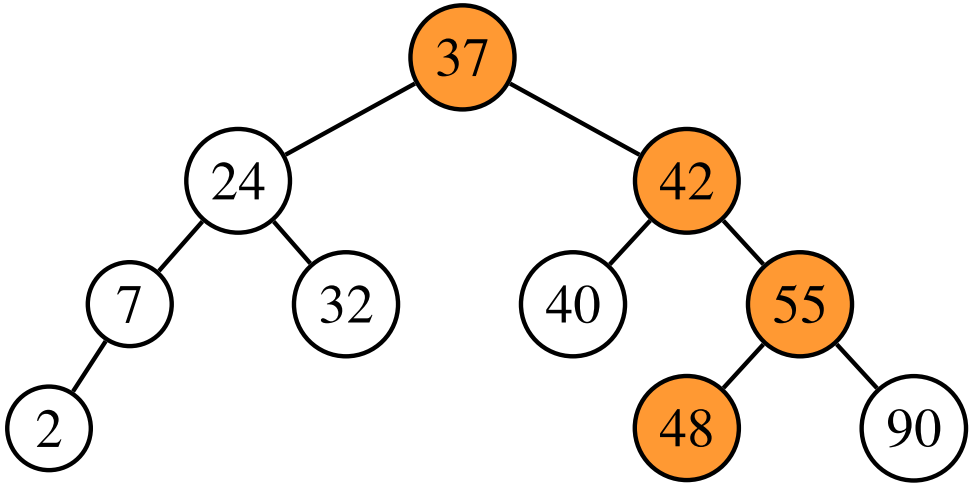
- The run time is proportional to the number of nodes on the path starting at the root and ending at some tree node.
- Average run time:
  - For successful search - proportional to the average node depth + 1
  - For unsuccessful search - proportional to the average depth of a `null` link + 1
- Assumptions for average-case analysis
  - For successful search - all nodes are equally likely to match the input key K.
  - For unsuccessful search -all `null` references are equally likely to be the link where the search stops.

Run Time of `findTree`



- The worst case run time is linear in the **height** of the tree, i.e., $\Theta(h)$.
  - $\Theta(n)$ (worst case) where n is the number tree nodes.
  - The tree is *degenerate* (i. e., a sequence of nodes) and the key is either leaf (success) or longest path in the tree is followed (key not in tree)
- Average run time is $\Theta(\log n)$
  - Consider binary search trees generated by using random insertion sequences
  - Assume that all $n!$ possible orderings of the n keys are equally likely to be the sequence of insertions for building the tree.
  - The expected value of h in a randomly built BST is $\Theta(\log n)$

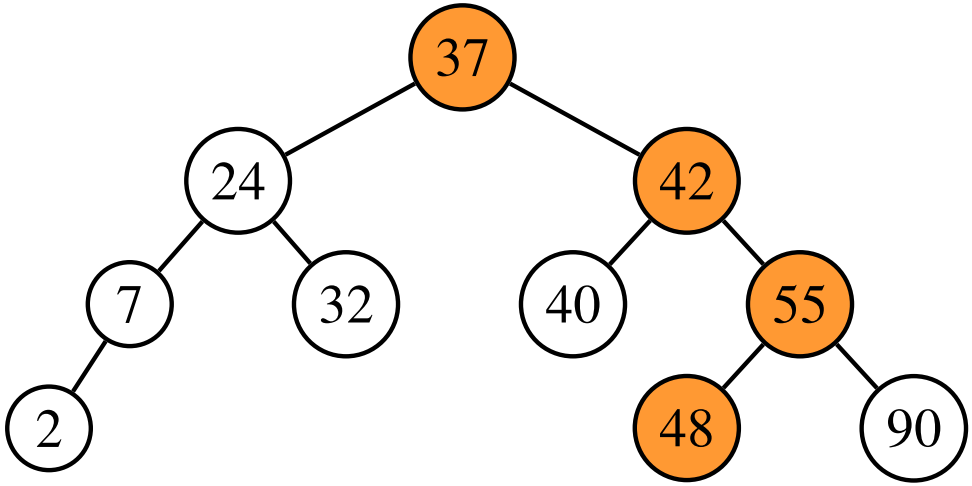
Run Time of `findTree`



Runtime : depends on the **height** of the tree.

- $\Theta(n)$ worst case, if the tree is degenerate
- $\Theta(\log n)$ **on average**. Randomly generated binary search trees are reasonably well balanced and have height near that of a complete tree (i.e., $\approx \log_2 n$).

Run Time of `findTree`



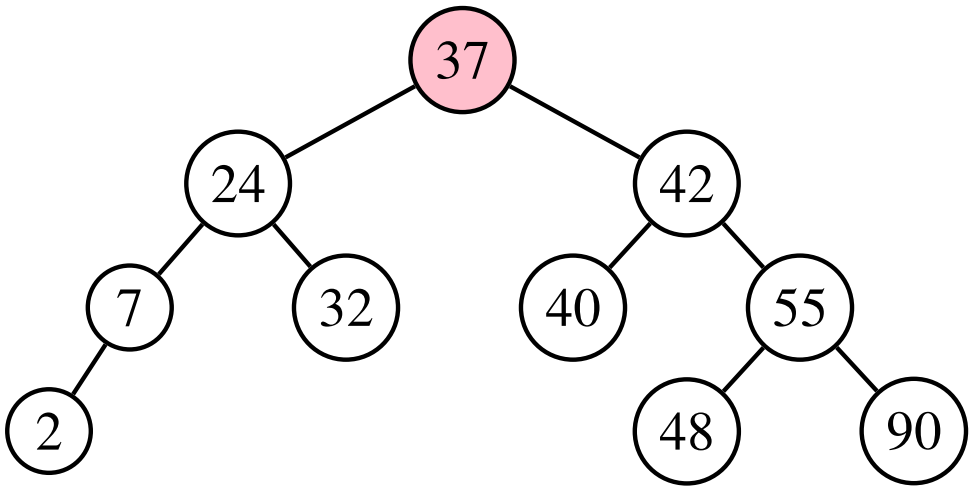
- Can `findTree()` be implemented without recursion?

Binary Search Trees - findTreeMin

- Minimum element in binary search tree is the **left-most** element.

```
1  public E findTreeMin(){
2      LLTree<E> t = root;
3      if(t==null)
4          throw new NoSuchElementException();
5      while(t.left != null)
6          t=t.left;
7      return(t.element);
8  }
```

Example:findTreeMin()



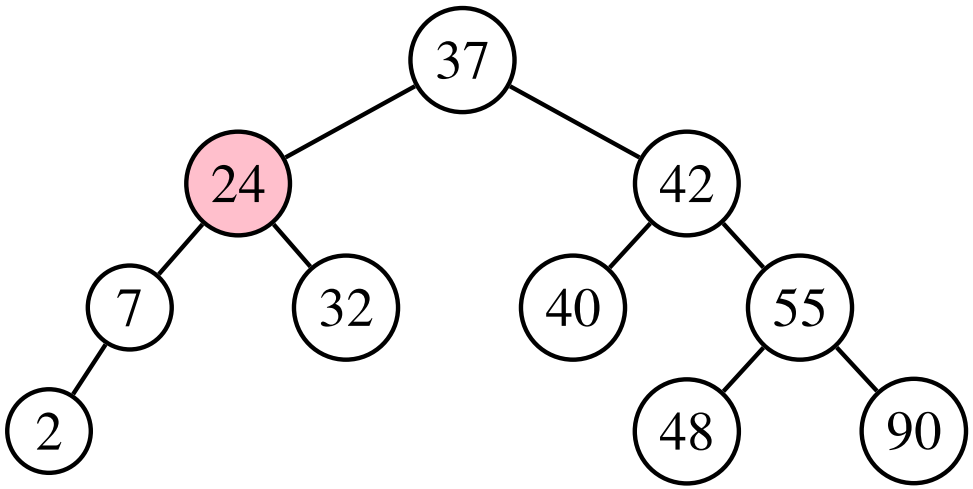
Binary Search Trees - findTreeMin

McMaster University

- Minimum element in binary search tree is the **left-most** element.

```
1  public E findTreeMin(){
2      LLTree<E> t = root;
3      if(t==null)
4          throw new NoSuchElementException();
5      while(t.left != null)
6          t=t.left;
7      return(t.element);
8  }
```

Example:findTreeMin()



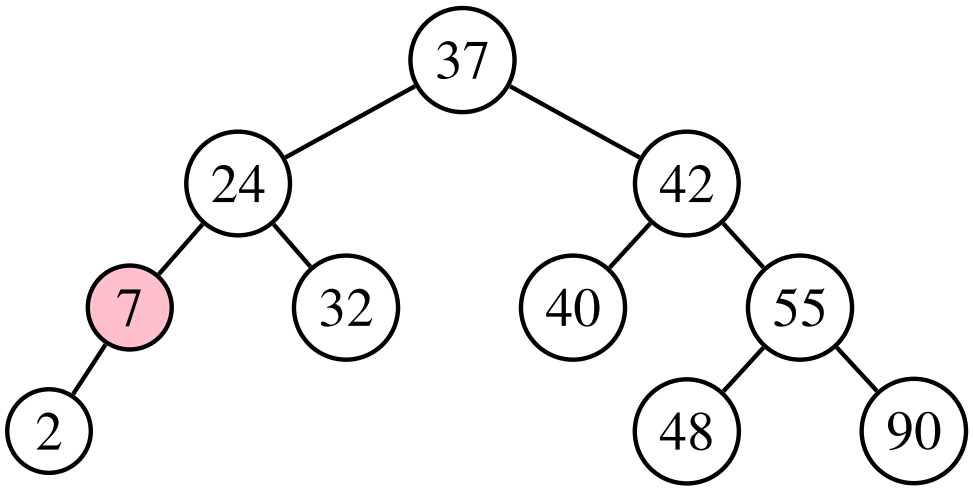
Binary Search Trees - findTreeMin

McMaster University

- Minimum element in binary search tree is the **left-most** element.

```
1  public E findTreeMin(){
2      LLTree<E> t = root;
3      if(t==null)
4          throw new NoSuchElementException();
5      while(t.left != null)
6          t=t.left;
7      return(t.element);
8  }
```

Example:findTreeMin()



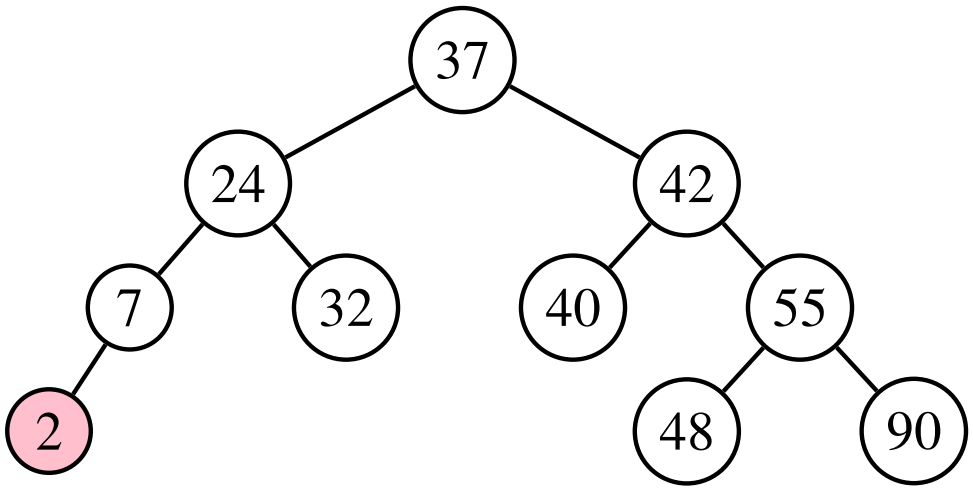
Binary Search Trees - findTreeMin

McMaster University

- Minimum element in binary search tree is the **left-most** element.

```
1  public E findTreeMin(){
2      LLTree<E> t = root;
3      if(t==null)
4          throw new NoSuchElementException();
5      while(t.left != null)
6          t=t.left;
7      return(t.element);
8  }
```

Example:findTreeMin()



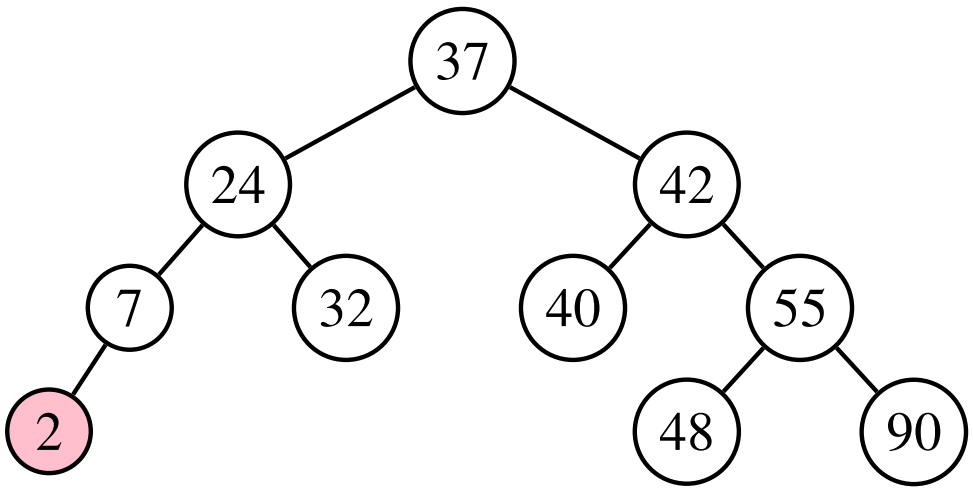
Binary Search Trees - findTreeMin

McMaster University

- Minimum element in binary search tree is the **left-most** element.

```
1  public E findTreeMin(){
2      LLTree<E> t = root;
3      if(t==null)
4          throw new NoSuchElementException();
5      while(t.left != null)
6          t=t.left;
7      return(t.element);
8  }
```

Example:findTreeMin() – **Found** – $\Theta(n)$ worst case, but $\Theta(\log n)$ on average.

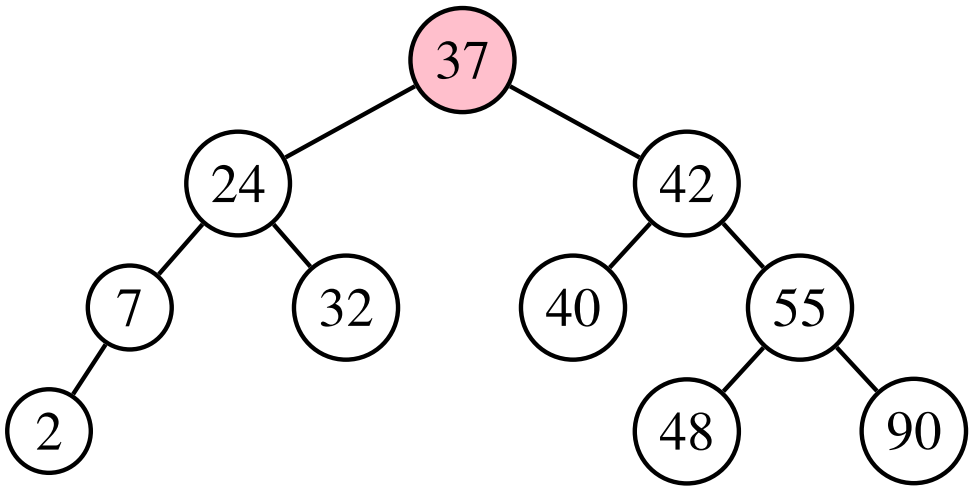


Binary Search Trees - findTreeMax

- Maximum element in binary search tree is the **right-most** element.

```
1  public E findTreeMax(){
2      LLTree<E> t = root;
3      if(t==null)
4          throw new NoSuchElementException();
5      while(t.right != null)
6          t=t.right;
7      return(t.element);
8  }
```

Example:findTreeMax()



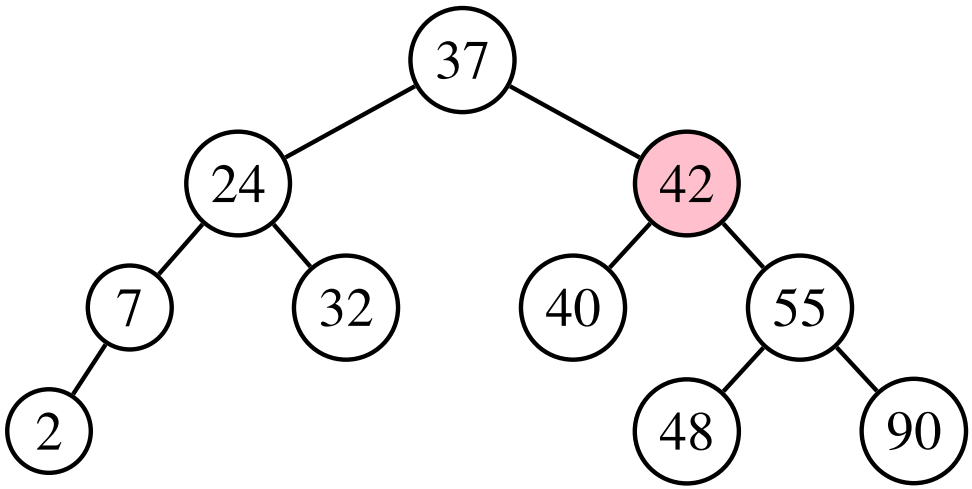
Binary Search Trees - findTreeMax

McMaster University

- Maximum element in binary search tree is the **right-most** element.

```
1  public E findTreeMax(){
2      LLTree<E> t = root;
3      if(t==null)
4          throw new NoSuchElementException();
5      while(t.right != null)
6          t=t.right;
7      return(t.element);
8  }
```

Example:findTreeMax()



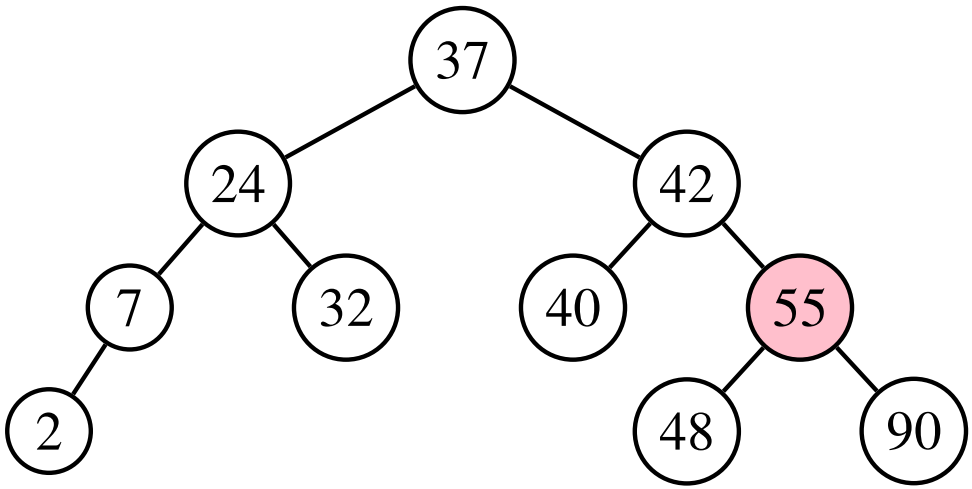
Binary Search Trees - findTreeMax

McMaster University

- Maximum element in binary search tree is the **right-most** element.

```
1  public E findTreeMax(){
2      LLTree<E> t = root;
3      if(t==null)
4          throw new NoSuchElementException();
5      while(t.right != null)
6          t=t.right;
7      return(t.element);
8  }
```

Example:findTreeMax()

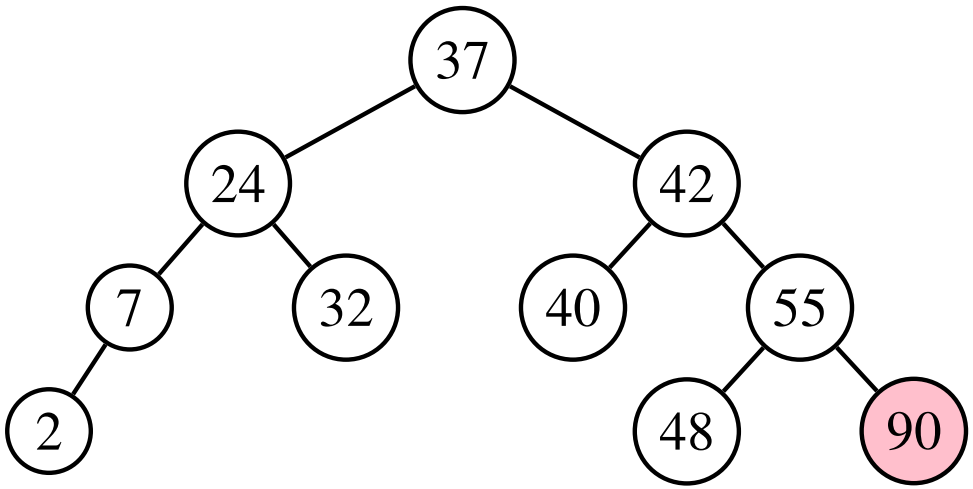


Binary Search Trees - findTreeMax

- Maximum element in binary search tree is the **right-most** element.

```
1  public E findTreeMax(){
2      LLTree<E> t = root;
3      if(t==null)
4          throw new NoSuchElementException();
5      while(t.right != null)
6          t=t.right;
7      return(t.element);
8  }
```

Example:findTreeMax()



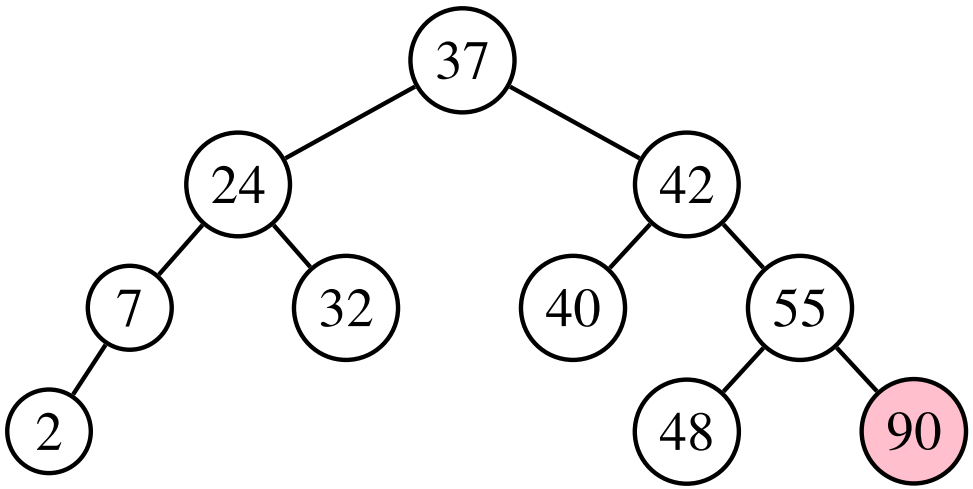
Binary Search Trees - findTreeMax

McMaster University

- Maximum element in binary search tree is the **right-most** element.

```
1  public E findTreeMax(){
2      LLTree<E> t = root;
3      if(t==null)
4          throw new NoSuchElementException();
5      while(t.right != null)
6          t=t.right;
7      return(t.element);
8  }
```

Example:findTreeMax() – Found – $\Theta(n)$ worst case, but $\Theta(\log n)$ on average.



Binary Search Trees - insertTree

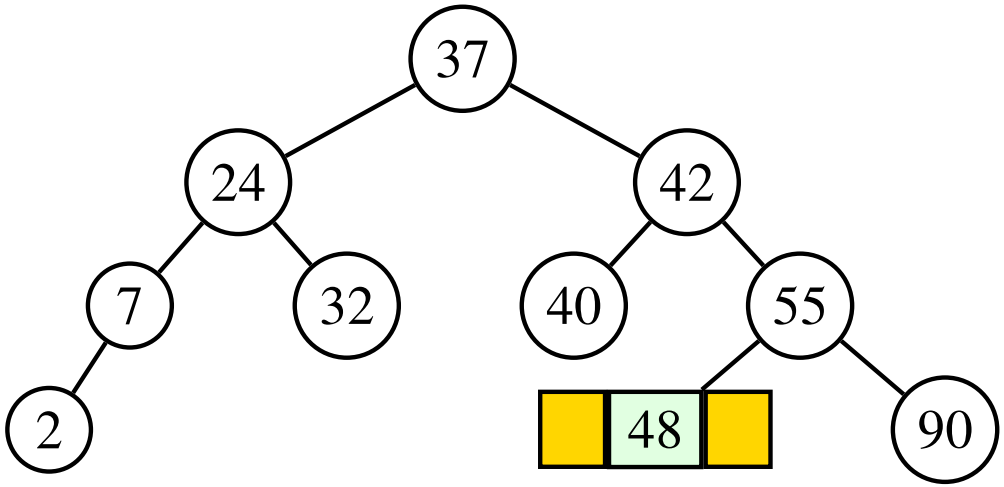
- Always insert a new node as a **leaf** on a tree.
  - → Find place where to insert element, then insert element

```
1  private TNode<E> insertTree(E val, TNode<E> t){
2      if(t==null){
3          t=new TNode<E>(val,null,null);
4      }else if(val.compareTo(t.element)<0)
5          t.left = insertTree(val,t.left);
6      else if(val.compareTo(t.element)>0)
7          t.right = insertTree(val,t.right);
8      else
9          throw new DuplicateItemException();
10     return t;
11 }
```

Binary Search Trees - insertTree

Example:insertTree(43)

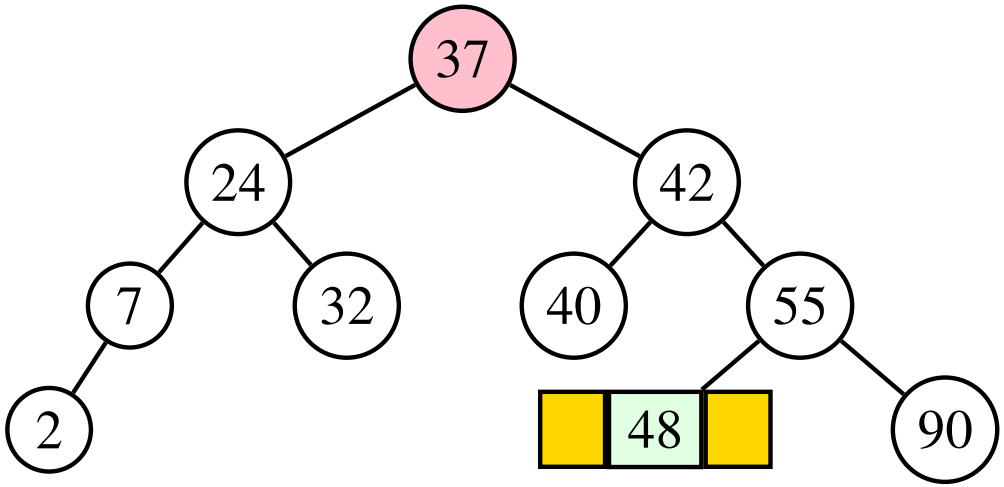
```
1  private TNode<E> insertTree(E val, TNode<E> t){
2      ...
3      }else if(val.compareTo(t.element)<0)
4          t.left = insertTree(val,t.left);
5      else if(val.compareTo(t.element)>0)
6          t.right = insertTree(val,t.right);
7      else
8          throw new DuplicateItemException();
9      return t;
10 }
```



Binary Search Trees - insertTree

Example:insertTree(43,root)

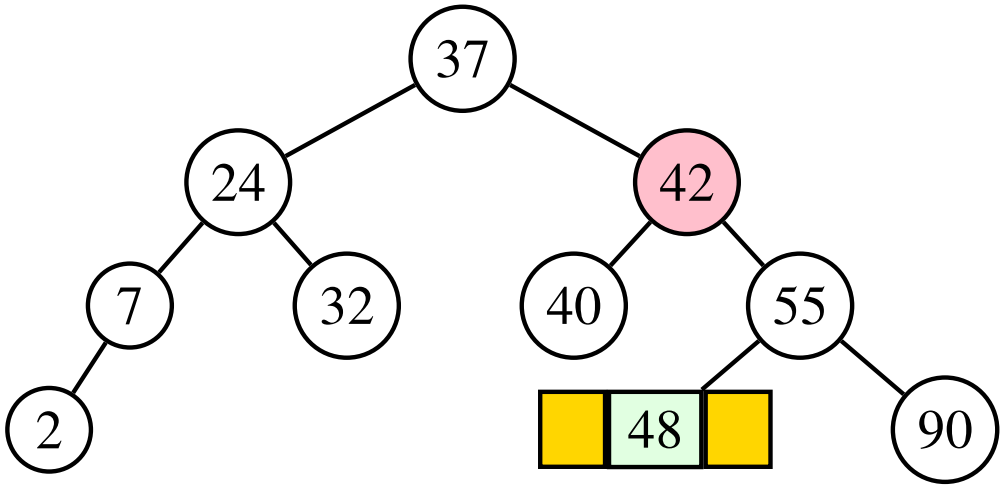
```
1  private TNode<E> insertTree(E val, TNode<E> t){
2      ...
3      }else if(val.compareTo(t.element)<0)
4          t.left = insertTree(val,t.left);
5      else if(val.compareTo(t.element)>0)
6          t.right = insertTree(val,t.right);
7      else
8          throw new DuplicateItemException();
9      return t;
10 }
```



Binary Search Trees - insertTree

Example:insertTree(43,t)

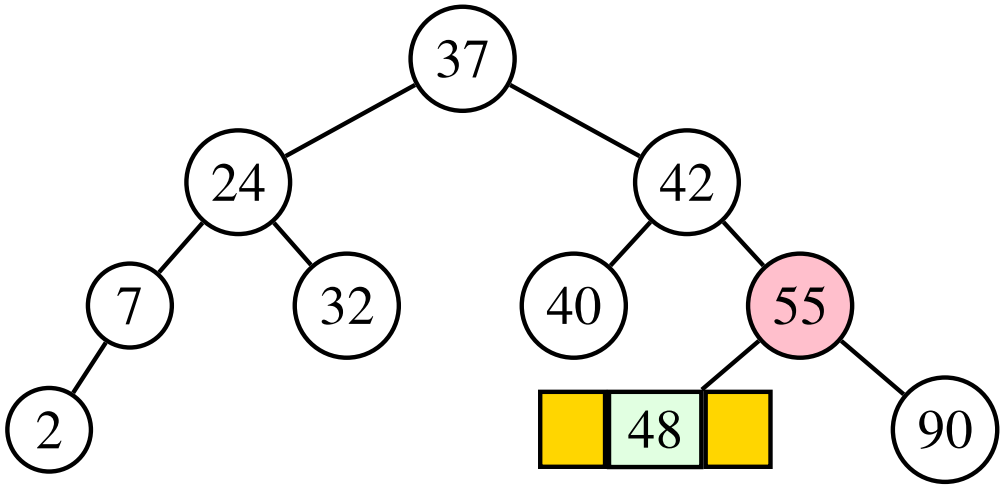
```
1  private TNode<E> insertTree(E val, TNode<E> t){
2      ...
3      }else if(val.compareTo(t.element)<0)
4          t.left = insertTree(val,t.left);
5      else if(val.compareTo(t.element)>0)
6          t.right = insertTree(val,t.right);
7      else
8          throw new DuplicateItemException();
9      return t;
10 }
```



Binary Search Trees - insertTree

Example:insertTree(43,t)

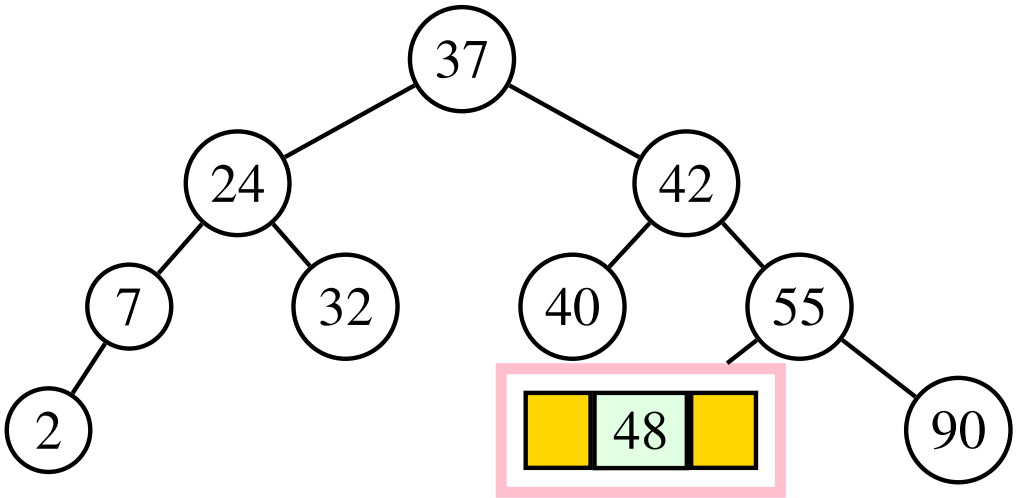
```
1  private TNode<E> insertTree(E val, TNode<E> t){
2      ...
3      }else if(val.compareTo(t.element)<0)
4          t.left = insertTree(val,t.left);
5      else if(val.compareTo(t.element)>0)
6          t.right = insertTree(val,t.right);
7      else
8          throw new DuplicateItemException();
9      return t;
10 }
```



Binary Search Trees - insertTree

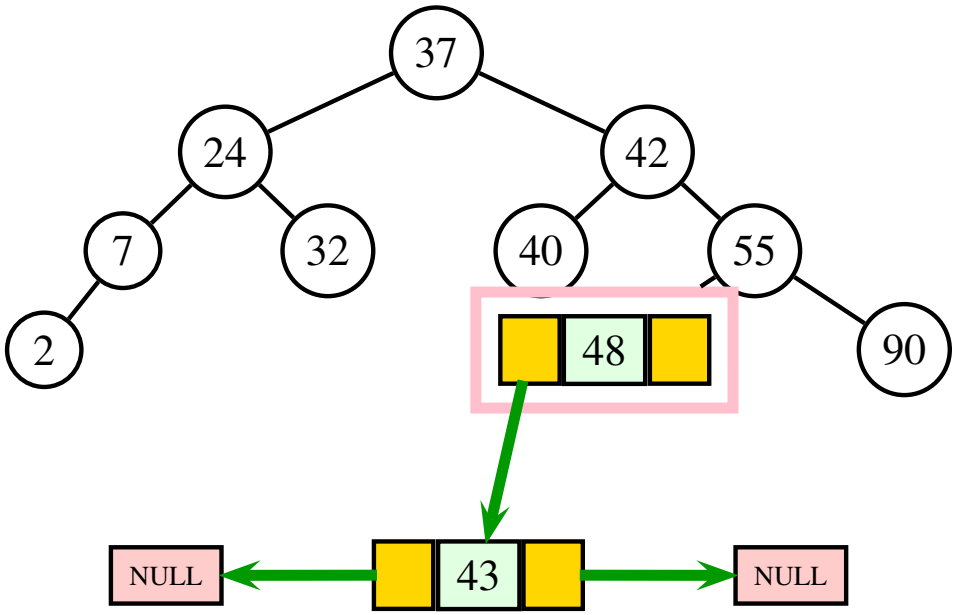
Example:insertTree(43,t)

```
1  private TNode<E> insertTree(E val, TNode<E> t){
2      ...
3      }else if(val.compareTo(t.element)<0)
4          t.left = insertTree(val,t.left);
5      else if(val.compareTo(t.element)>0)
6          t.right = insertTree(val,t.right);
7      else
8          throw new DuplicateItemException();
9      return t;
10 }
```



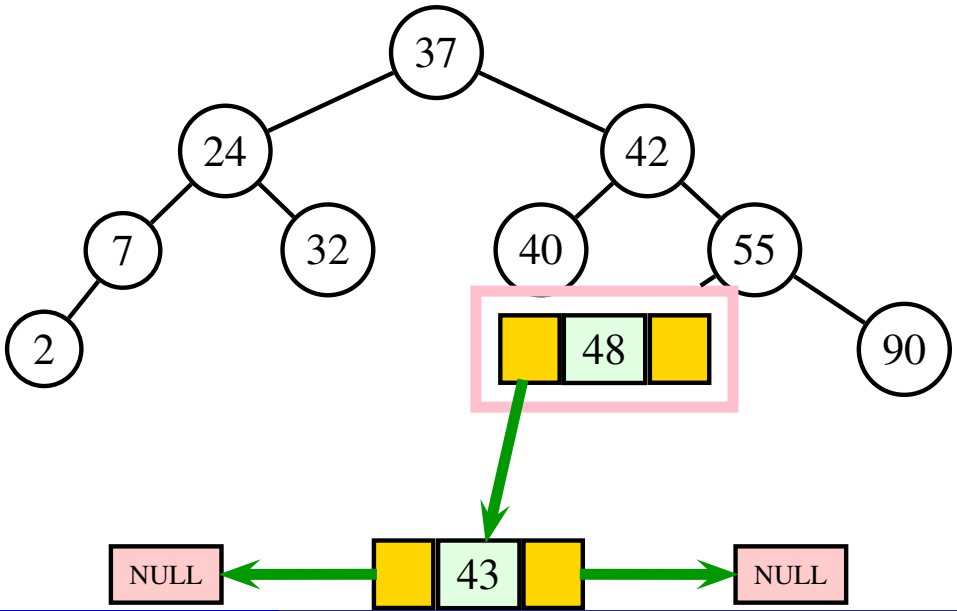
Example:insertTree(43,t)

```
1  private TNode<E> insertTree(E val, TNode<E> t){
2      if(t==null){
3          t=new TNode<E>(val,null,null);
4      }else if(val.compareTo(t.element)<0)
5          t.left = insertTree(val,t.left);
6      ...
7      return t;
8  }
```



Example:`insertTree(43,t)` – Finished – $\Theta(h)$ worst case – $\Theta(n)$ worst case, but $\Theta(\log n)$ on average.

```
1  private TNode<E> insertTree(E val, TNode<E> t){
2      if(t==null){
3          t=new TNode<E>(val,null,null);
4      }else if(val.compareTo(t.element)<0)
5          t.left = insertTree(val,t.left);
6      ...
7      return t;
8  }
```

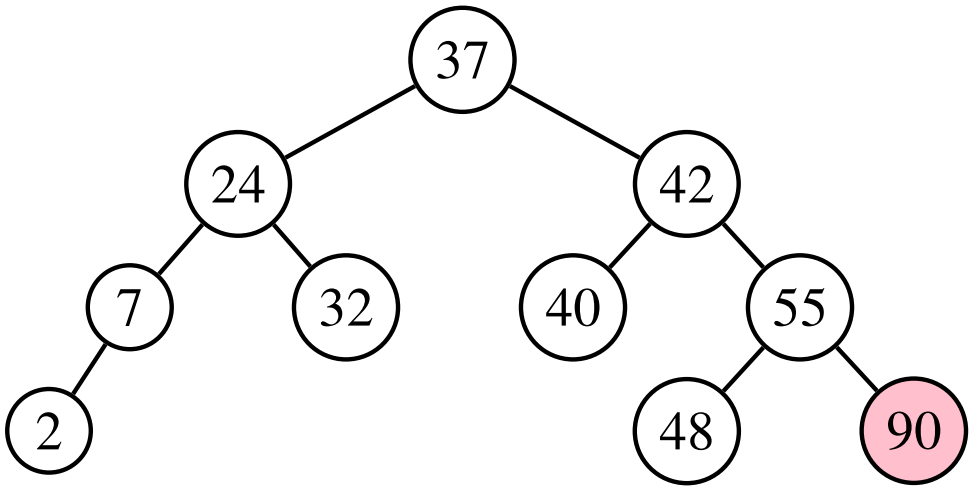


Binary Search Trees - `deleteTree`

- To delete a node, first need to find the node in the tree – takes $\Theta(\log n)$ on average.
- There are three special cases we need to consider:
  - (1) The node is a leaf, in which case you can just remove it

Example: `deleteTree(90)`

Before

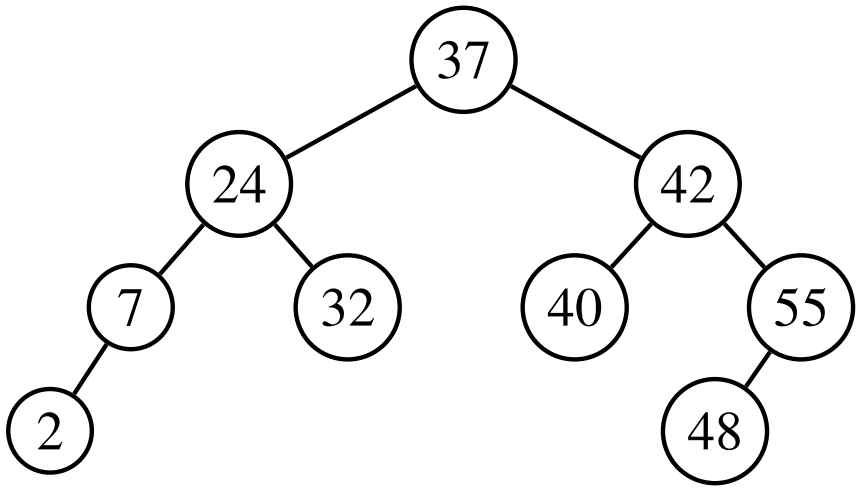


Binary Search Trees - `deleteTree`

- To delete a node, first need to find the node in the tree – takes $\Theta(\log n)$ on average.
- There are three special cases we need to consider:
  - (1) The node is a leaf, in which case you can just remove it

Example: `deleteTree(90)`

After

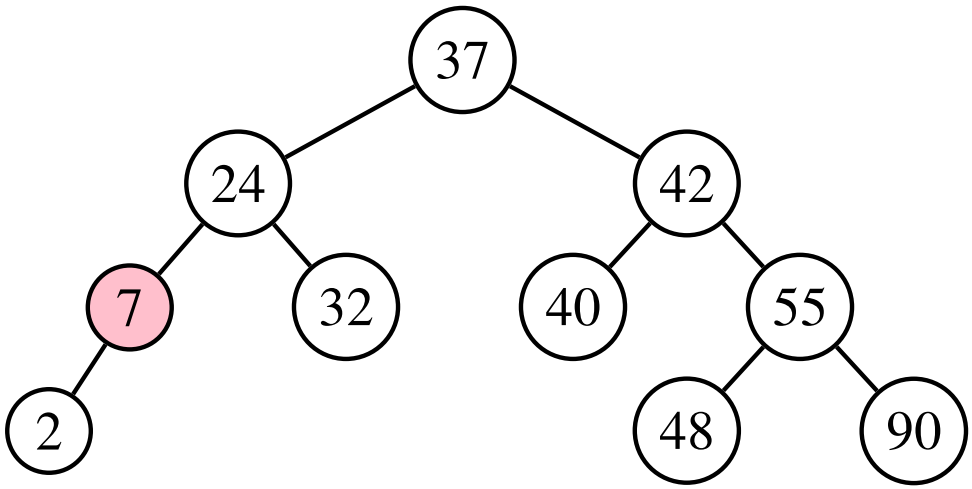


Binary Search Trees - `deleteTree`

- To delete a node, first need to find the node in the tree – takes $\Theta(\log n)$ on average.
- There are three special cases we need to consider:
  - (2) If the node has only one child, delete the node and adjust the parent to point to the only child.

Example: `deleteTree(7)`

Before

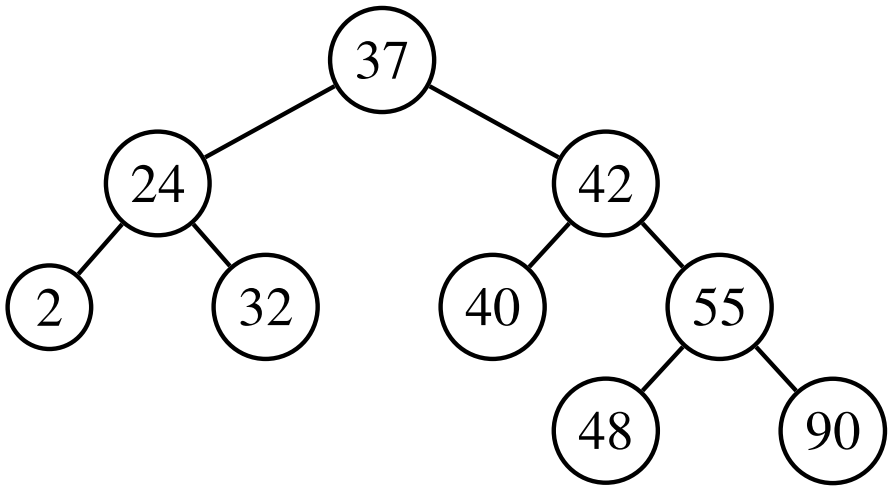


Binary Search Trees - `deleteTree`

- To delete a node, first need to find the node in the tree – takes $\Theta(\log n)$ on average.
- There are three special cases we need to consider:
  - (2) If the node has only one child, delete the node and adjust the parent to point to the only child.

Example: `deleteTree(7)`

After

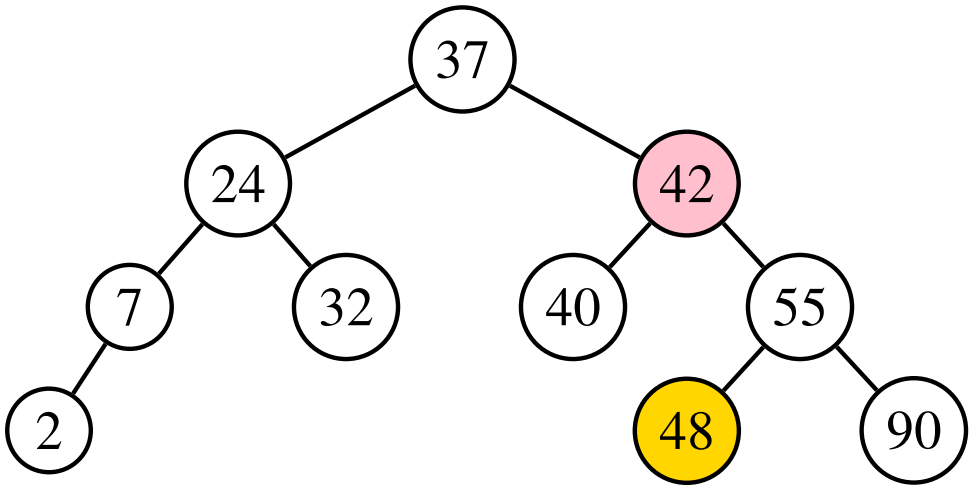


Binary Search Trees - `deleteTree`

- To delete a node, first need to find the node in the tree – takes $\Theta(\log n)$ on average.
- There are three special cases we need to consider:
  - (3) If the node has two children, replace the node with the smallest element in the right sub-tree
  - Then delete the smallest element in right sub-tree (this is easy since it has no left child)

Example: `deleteTree(42)`

Before

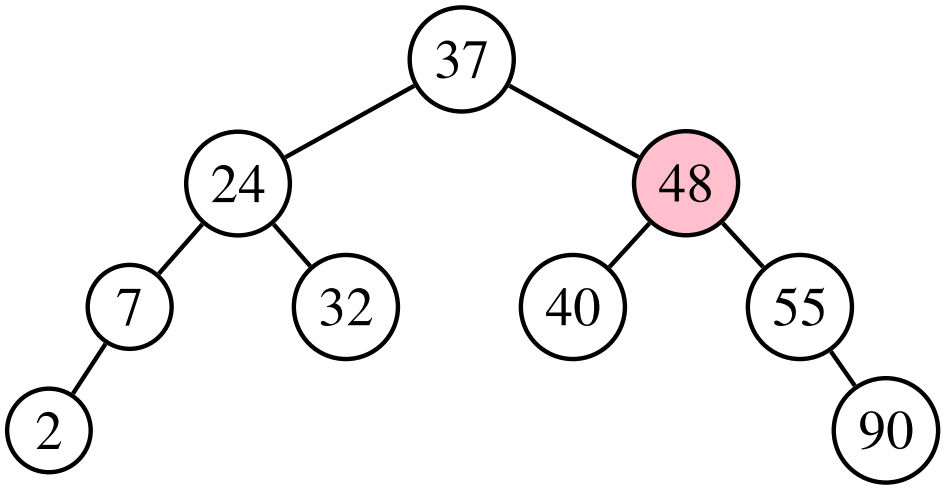


Binary Search Trees - `deleteTree`

- To delete a node, first need to find the node in the tree – takes $\Theta(\log n)$ on average.
- There are three special cases we need to consider:
  - (3) If the node has two children, replace the node with the smallest element in the right sub-tree
  - Then delete the smallest element in right sub-tree (this is easy since it has no left child)

Example: `deleteTree(42)`

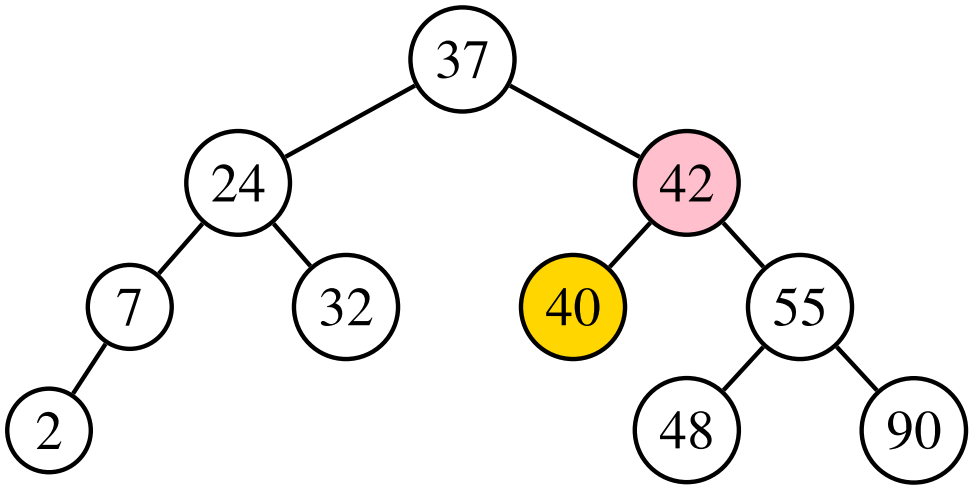
After



- **Alternate solution**
  - (3) If the node has two children, replace the node with the **largest** element in the **left** sub-tree
  - Then delete the largest element in left sub-tree (this is easy since it has no right child)

Example: `deleteTree(42)`

Before

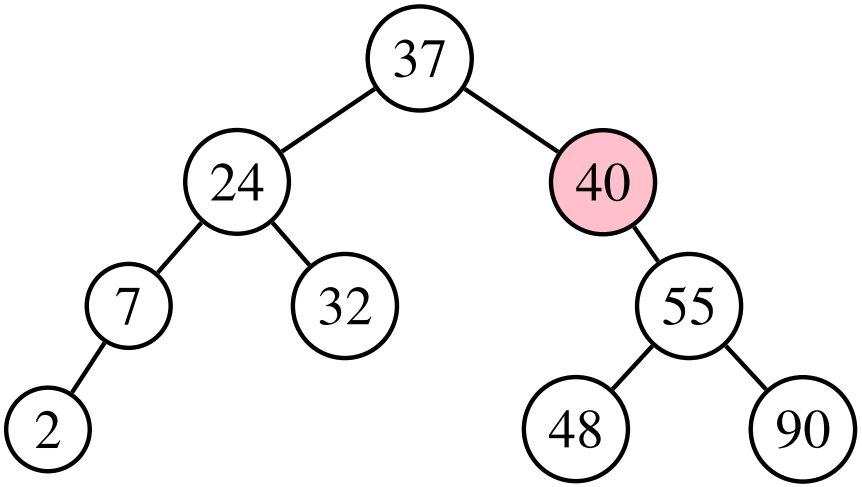


McMaster University

- **Alternate solution**
  - (3) If the node has two children, replace the node with the **largest** element in the **left** sub-tree
  - Then delete the largest element in left sub-tree (this is easy since it has no right child)

Example: `deleteTree(42)`

After



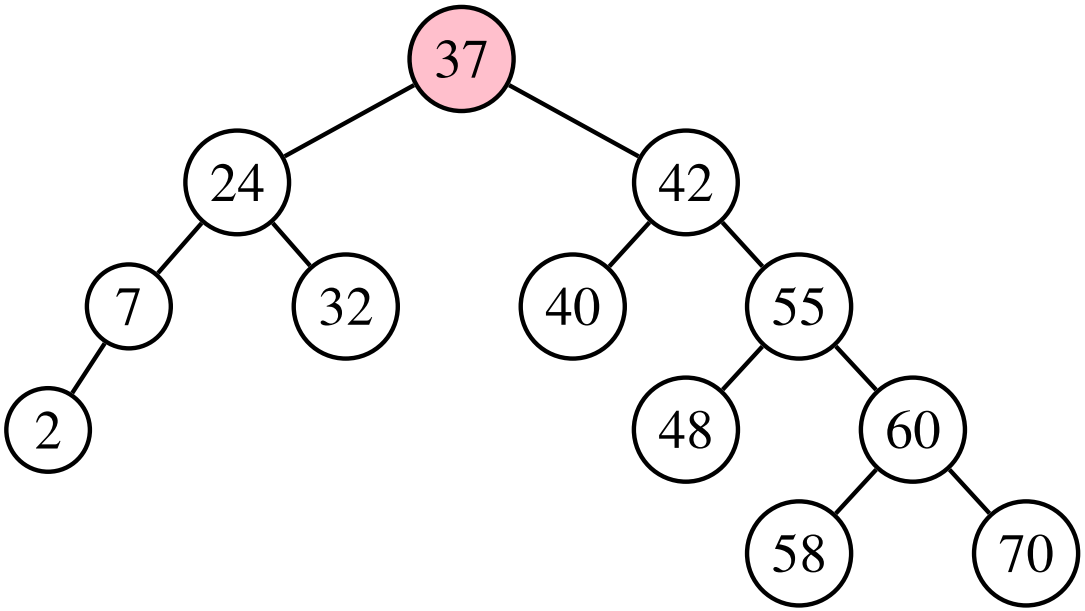
Binary Search Trees - deleteTree

```
1  private TNode<E> deleteTree(E val, TNode<E> t){
2      if (t == null)
3          throw new ItemNotFoundException();
4      if(val.compareTo(t.element)<0)
5          t.left = deleteTree(val,t.left);
6      else if(val.compareTo(t.element)>0)
7          t.right = deleteTree(val,t.right);
8      else if (t.left !=null && t.right != null){
9                      // Delete node with 2 children
10         t.element = (findMin(t.right)).element;
11                     // Find min starting at right child
12         t.right = removeMin(t.right);
13                    // Routine which removes min value
14     }
15     else
16         t = (t.left != null) ? t.left : t.right;     // only child
17     return t;
18 }
```

Binary Search Trees - deleteTree

```
1  private TNode<E> deleteTree(E val, TNode<E> t){
2      ...
3      if(val.compareTo(t.element)<0)
4          t.left = deleteTree(val,t.left);
5      else if(val.compareTo(t.element)>0)
6          t.right = deleteTree(val,t.right);
7      ...
8  }
```

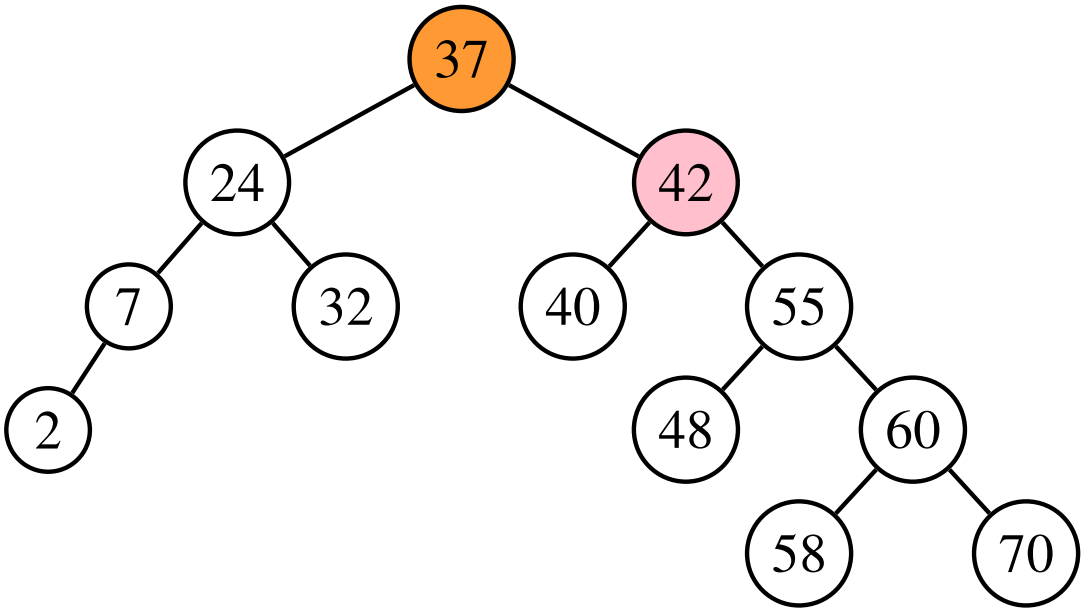
Example: `deleteTree(55,root)`



Binary Search Trees - deleteTree

```
1  private TNode<E> deleteTree(E val, TNode<E> t){
2      ...
3      if(val.compareTo(t.element)<0)
4          t.left = deleteTree(val,t.left);
5      else if(val.compareTo(t.element)>0)
6          t.right = deleteTree(val,t.right);
7      ...
8  }
```

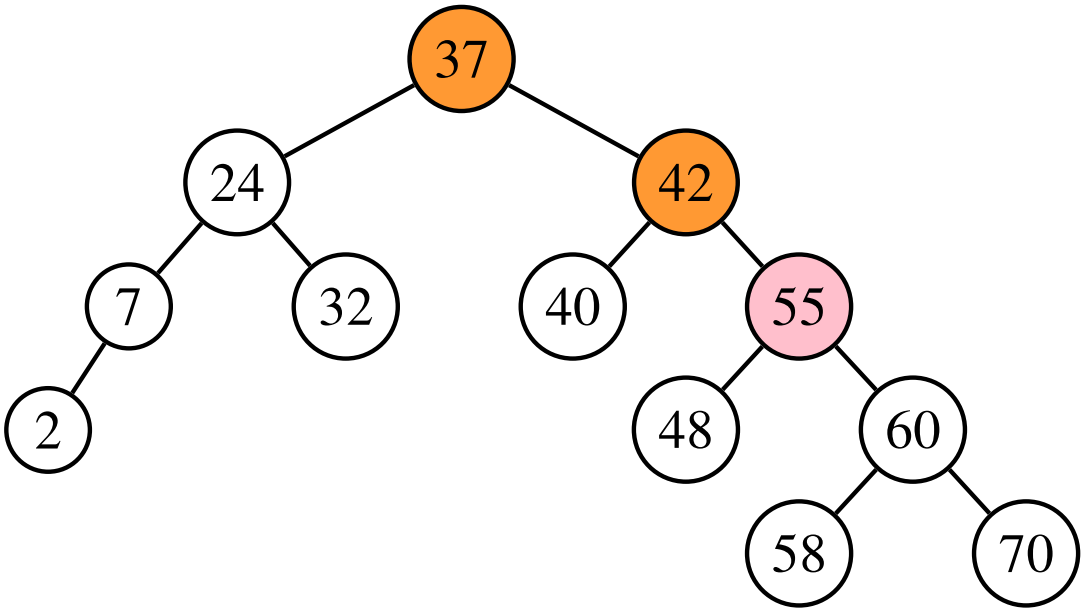
Example: `deleteTree(55,root)`



Binary Search Trees - deleteTree

```
1  private TNode<E> deleteTree(E val, TNode<E> t){
2      ...
3      else if (t.left !=null && t.right != null){
4          t.element = (findMin(t.right)).element;
5          t.right = removeMin(t.right);
6      } else t = (t.left != null) ? t.left : t.right;
7      return t;
8  }
```

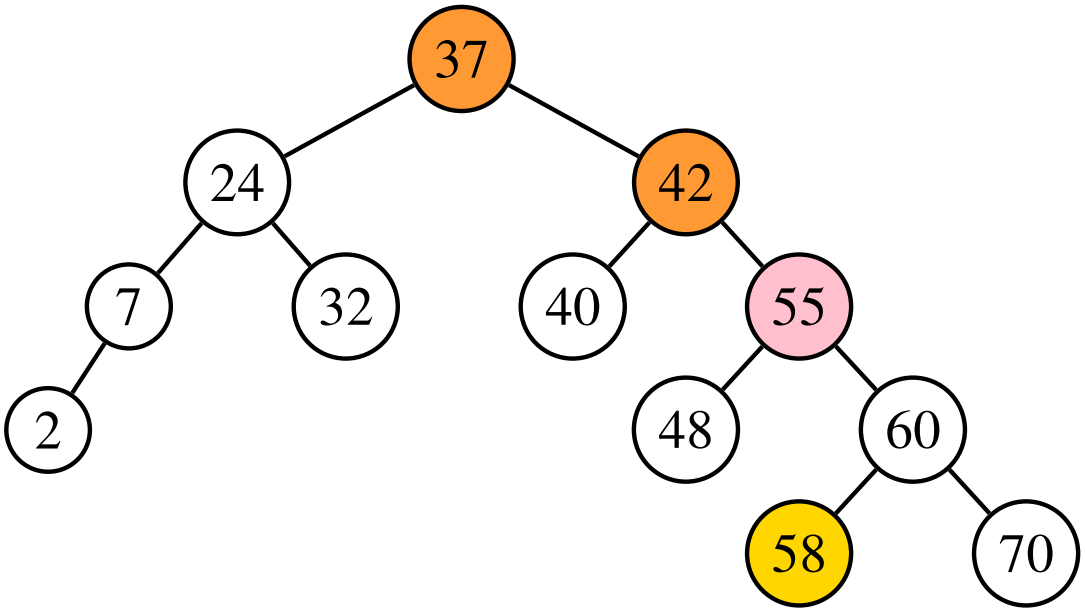
Example: deleteTree(55,root)



Binary Search Trees - deleteTree

```
1  private TNode<E> deleteTree(E val, TNode<E> t){
2      ...
3      else if (t.left !=null && t.right != null){
4          t.element = (findMin(t.right)).element;
5          t.right = removeMin(t.right);
6      } else t = (t.left != null) ? t.left : t.right;
7      return t;
8  }
```

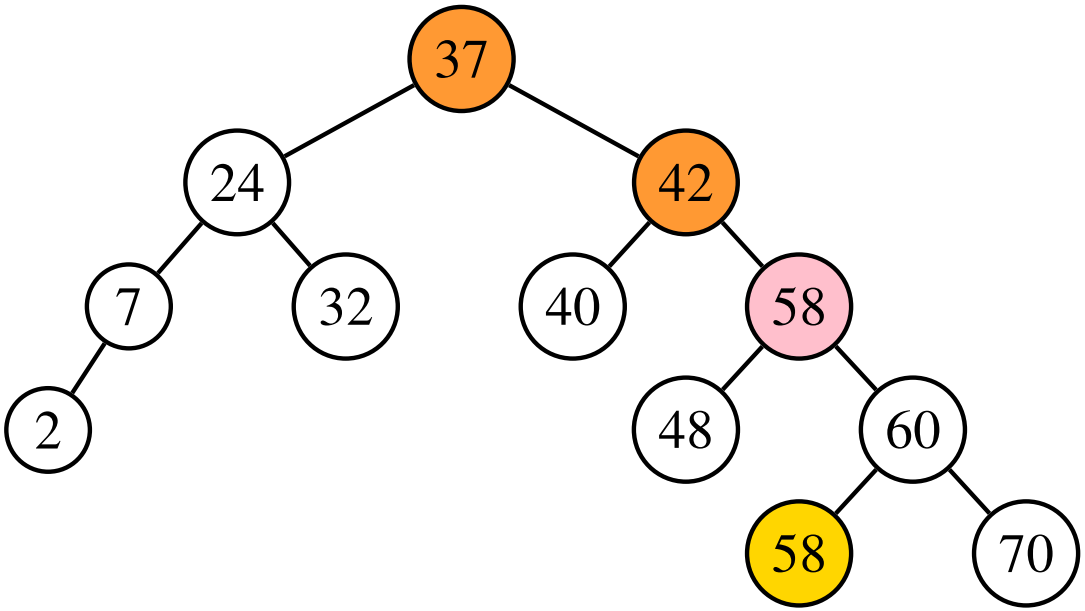
Example: deleteTree(55,root)



Binary Search Trees - deleteTree

```
1  private TNode<E> deleteTree(E val, TNode<E> t){
2      ...
3      else if (t.left !=null && t.right != null){
4          t.element = (findMin(t.right)).element;
5          t.right = removeMin(t.right);
6      } else t = (t.left != null) ? t.left : t.right;
7      return t;
8  }
```

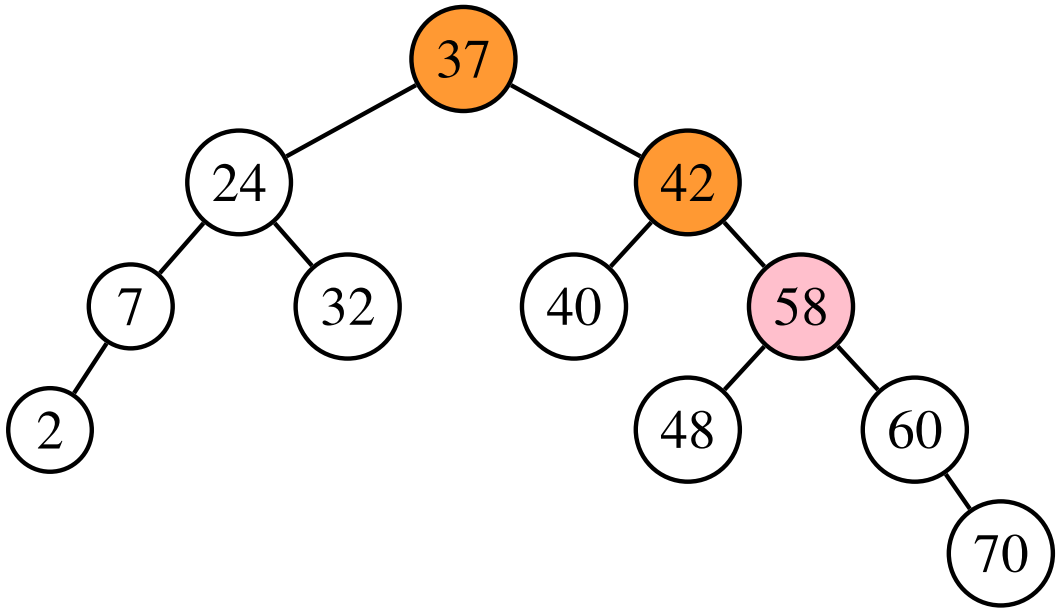
Example: deleteTree(55,root)



Binary Search Trees - deleteTree

```
1  private TNode<E> deleteTree(E val, TNode<E> t){
2      ...
3      else if (t.left !=null && t.right != null){
4          t.element = (findMin(t.right)).element;
5          t.right = removeMin(t.right);
6      } else t = (t.left != null) ? t.left : t.right;
7      return t;
8  }
```

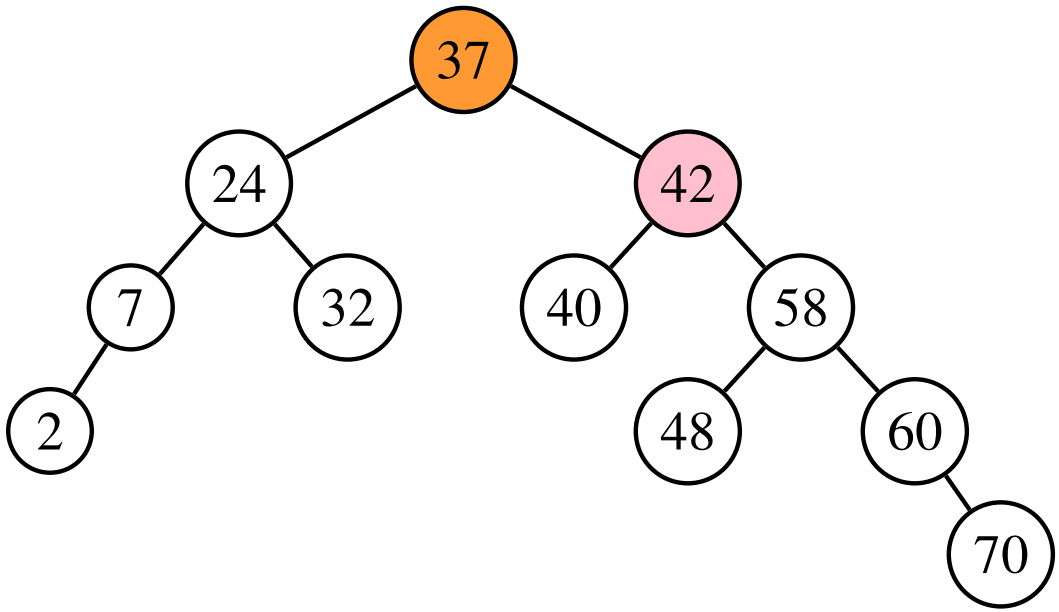
Example: deleteTree(55,root)



Binary Search Trees - deleteTree

```
1  private TNode<E> deleteTree(E val, TNode<E> t){
2  ...  if(val.compareTo(t.element)<0)
3           t.left = deleteTree(val,t.left);
4       else if(val.compareTo(t.element)>0)
5           t.right = deleteTree(val,t.right);
6       ... }
```

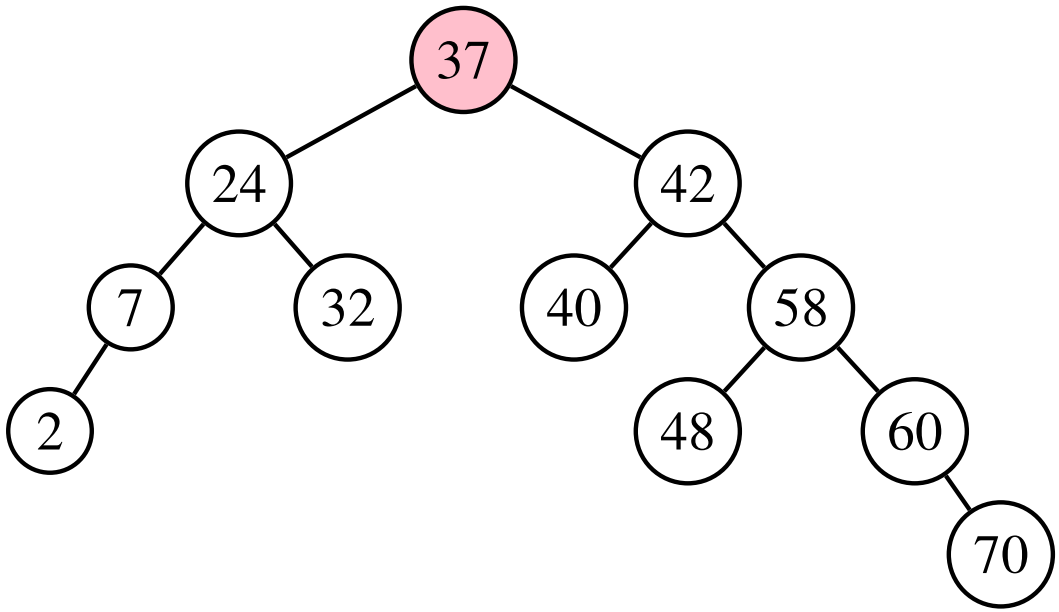
Example: `deleteTree(55,root)`



Binary Search Trees - deleteTree

```
1  private TNode<E> deleteTree(E val, TNode<E> t){
2  ...  if(val.compareTo(t.element)<0)
3           t.left = deleteTree(val,t.left);
4       else if(val.compareTo(t.element)>0)
5           t.right = deleteTree(val,t.right);
6       ... }
```

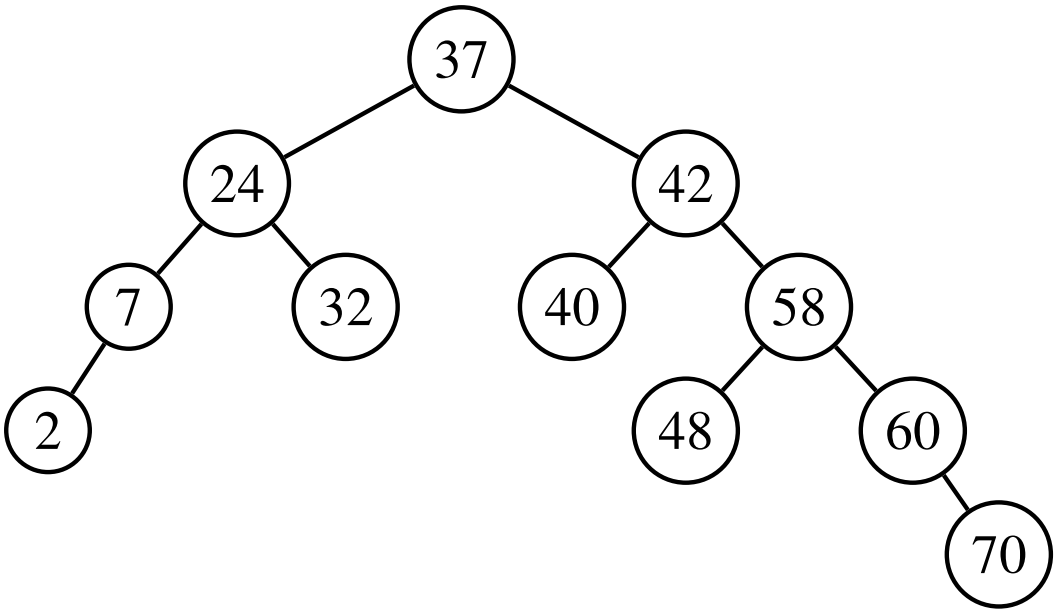
Example: `deleteTree(55,root)`



Binary Search Trees - deleteTree

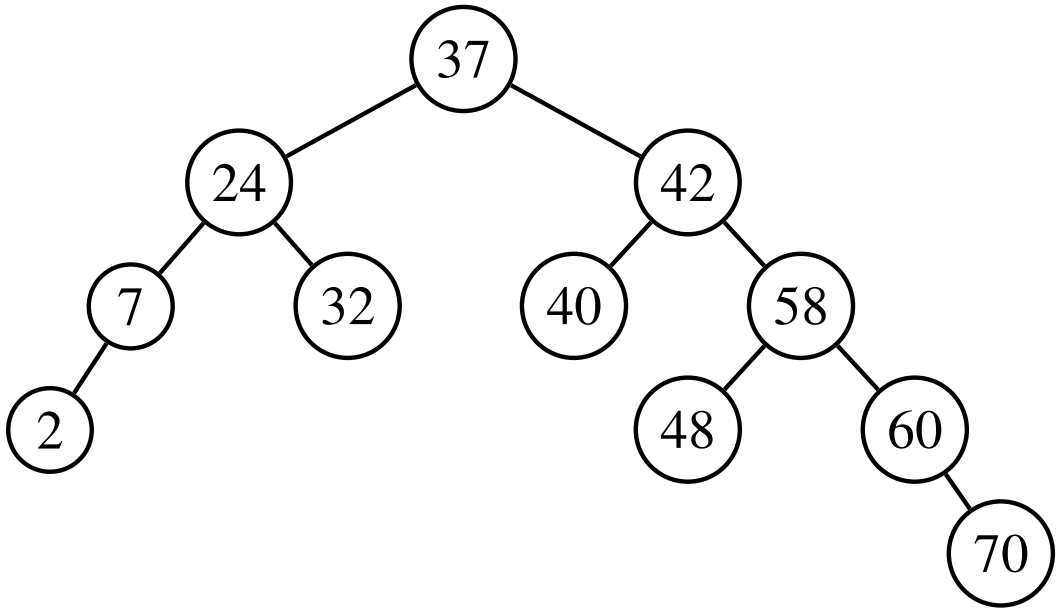
```
1  private TNode<E> deleteTree(E val, TNode<E> t){
2  ... else if (t.left !=null && t.right != null){
3          t.element = (findMin(t.right)).element;
4          t.right = removeMin(t.right);
5      } else t = (t.left != null) ? t.left : t.right;
6      return t;    }
```

Example: deleteTree(55,root) – **Done**



Binary Search Trees - removeMin

```
1  private TNode<E> removeMin(TNode<E> t){
2      if( t == null )
3          throw new ItemNotFoundException( );
4      else if (t.left !=null)
5          t.left = removeMin(t.left);
6      else
7          t = t.right;
8      return t;
9  }
```



Binary Search Trees - Notes

- Worst-case running time for search, insertion and deletion is linear in the height of the tree.
  - → Expected height is $\Theta(\log n)$ for randomly built BST's (using only insertions).
  - → In experiments a reasonable number of random insertions and deletions does not unbalance the tree (no math proof).
  - → Based on the above we can reasonably assume that for random input all operations take $\Theta(\log n)$ on average.

Binary Search Trees - Notes

- Worst-case running time for search, insertion and deletion is linear in the height of the tree.
  - → Expected height is $\Theta(\log n)$ for randomly built BST's (using only insertions).
  - → In experiments a reasonable number of random insertions and deletions does not unbalance the tree (no math proof).
  - → Based on the above we can reasonably assume that for random input all operations take $\Theta(\log n)$ on average.
- On the other hand, the worst-case BST occurs when the input sequence is sorted.

Binary Search Trees - Notes

- Worst-case running time for search, insertion and deletion is linear in the height of the tree.
  - → Expected height is $\Theta(\log n)$ for randomly built BST's (using only insertions).
  - → In experiments a reasonable number of random insertions and deletions does not unbalance the tree (no math proof).
  - → Based on the above we can reasonably assume that for random input all operations take $\Theta(\log n)$ on average.
- On the other hand, the worst-case BST occurs when the input sequence is sorted.
- There are other classes of BST's with a "balance condition"
  - → **AVL trees, red-black trees**
  - → BST operations run in $\Theta(\log n)$ in the worst case.

Binary Search Trees - Notes

- If there are few deletions, can use **lazy deletion**
  - → Just mark a node a being deleted. At a later time you can go back and reclaim this memory – "**garbage collection**"
  - → Since the height of the tree grows logarithmically with the number of nodes, the addition of these deleted nodes does not impact the performance greatly.

Binary Search Trees - `traverseTree`

- Tree traversal is the process of visiting every node in the tree
  - → if there are n nodes in the tree it takes $\Theta(n)$ time.
- There are three ways to visit the nodes:
  - **Pre-Order** - perform operation at node, then visit left and right subtrees.
  - **In-Order** - process left subtree, then node, then right subtree.
  - **Post-Order** - process left subtree, right subtree, then the node.
- In C, **pointers to functions** are useful to write a single function to implement all three traversal methods with any operation over the nodes.

Binary Search Trees - `traverseTree`

- Pass in a pointer to the tree and three pointers functions taking `(TNode *)` to implement operations on the nodes [2]
  - `preorderVisit` is a pointer to a function for operation before going to left subtree
  - `inorderVisit` is a pointer to a function for operation after left subtree before going to right subtree.
  - `postorderVisit` is a pointer to a function for operation after the right subtree.

[2] This traversal code is taken from Dr. D. Jones, *EE2SI4 Course Slides*, January 2004.

Binary Search Trees - traverseTree

```
1  void traverseTree(TNode *t, void (*preorderVisit)(TNode *),
2          void (*inorderVisit)(TNode *), void (*postorderVisit)(TNode *)){
3      if(t){
4          if(preorderVisit)
5              (*preorderVisit)(t);
6          traverseTree(t->left,preorderVisit,inorderVisit,postorderVisit);
7          if(inorderVisit)
8              (*inorderVisit)(t);
9          traverseTree(t->right,preorderVisit,inorderVisit,postorderVisit);
10         if(postorderVisit)
11             (*postorderVisit)(t);
12     }
13 }
```

Binary Search Trees - traverseTree

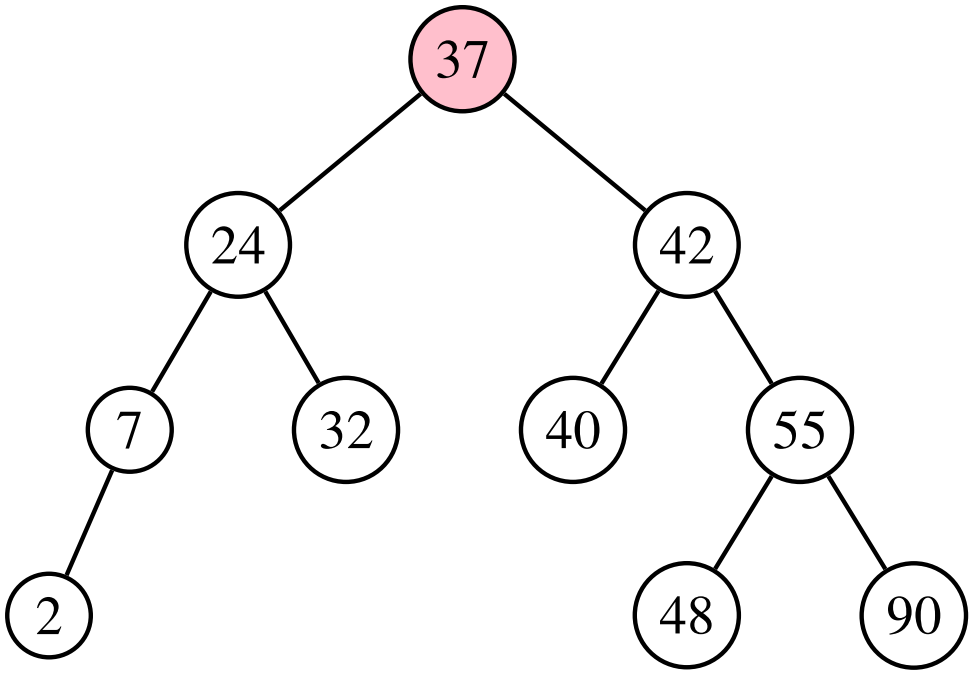
```
1  void traverseTree(TNode *t, void (*preorderVisit)(TNode *),
2          void (*inorderVisit)(TNode *), void (*postorderVisit)(TNode *)){
3      if(t){
4          if(preorderVisit)
5              (*preorderVisit)(t);
6          traverseTree(t->left,preorderVisit,inorderVisit,postorderVisit);
7          if(inorderVisit)
8              (*inorderVisit)(t);
9          traverseTree(t->right,preorderVisit,inorderVisit,postorderVisit);
10         if(postorderVisit)
11             (*postorderVisit)(t);
12     }
13 }
```

Example: Can write `destroyTree()` in terms of this function

```
1  void destroyTree(TNode *t){
2      traverseTree(t,NULL,NULL,(void (*)(TNode *)) free);
3  }
```

Binary Search Trees - traverseTree

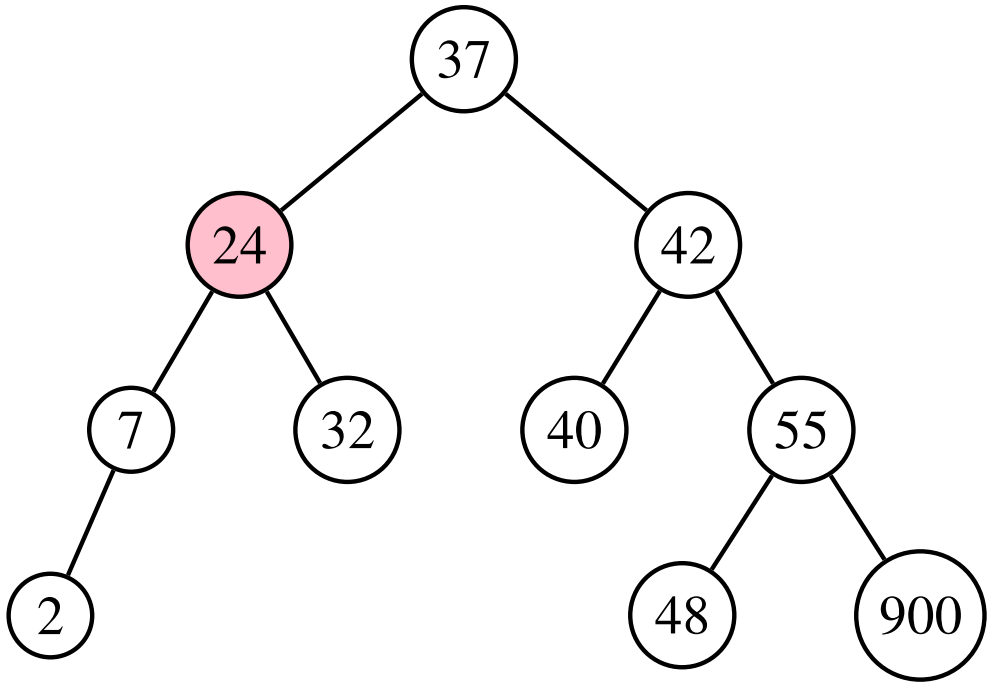
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37,

Binary Search Trees - traverseTree

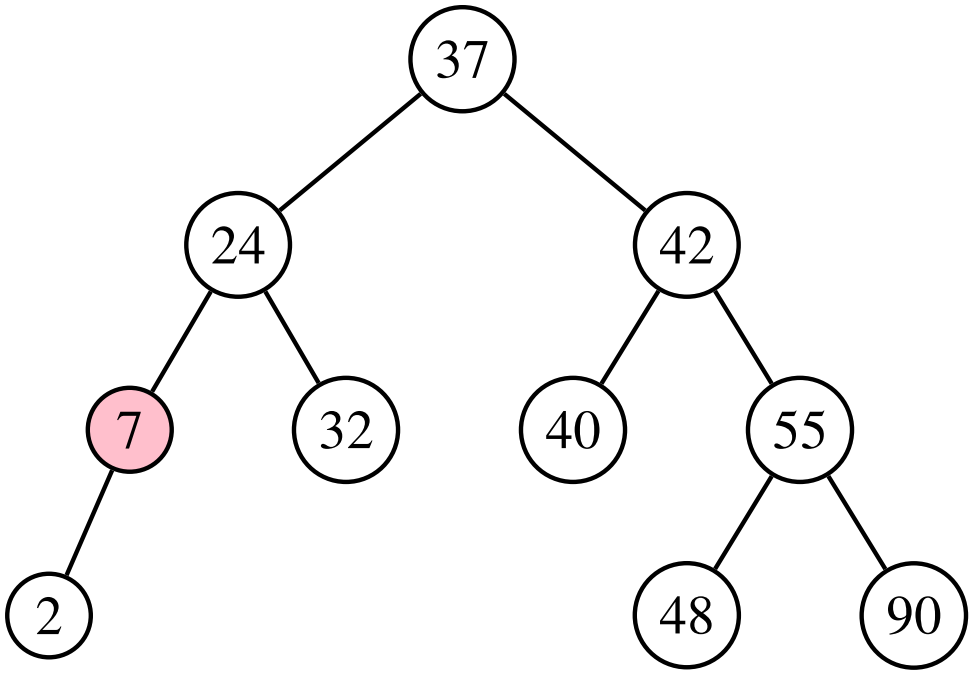
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24,

Binary Search Trees - traverseTree

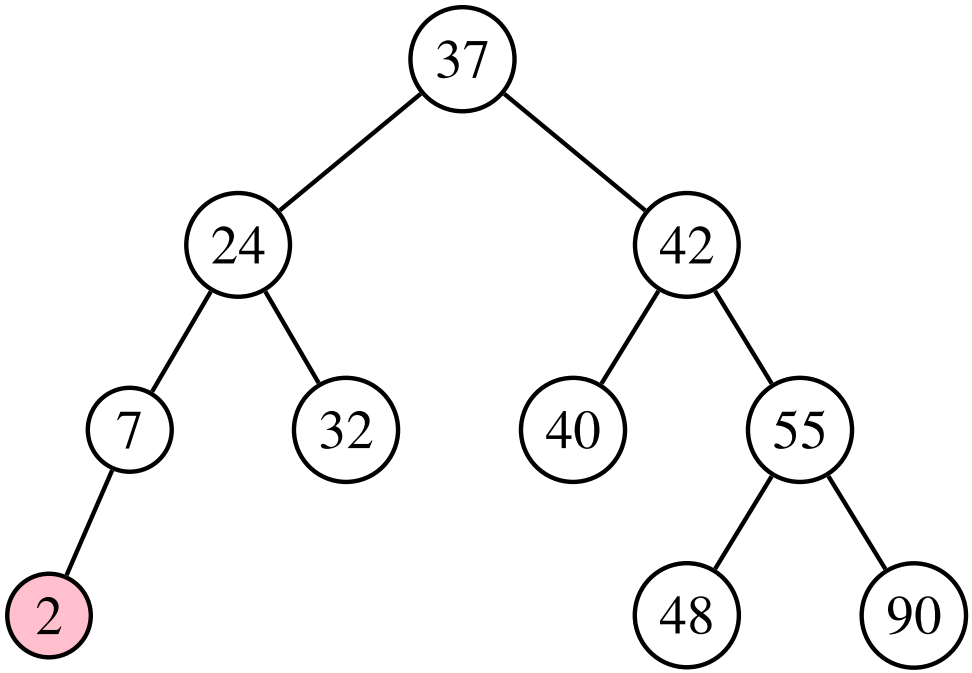
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7,

Binary Search Trees - traverseTree

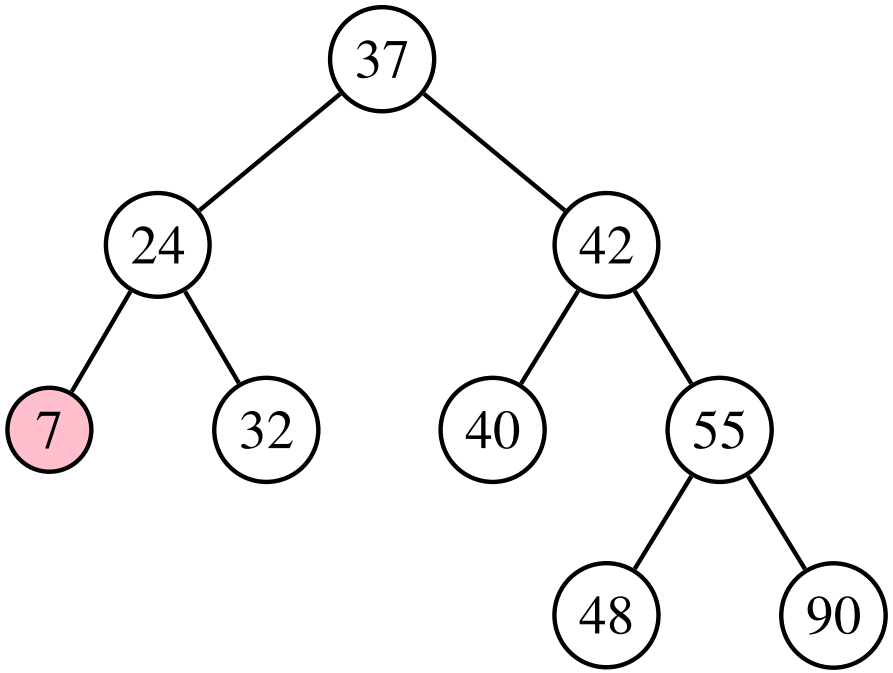
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2,

Binary Search Trees - traverseTree

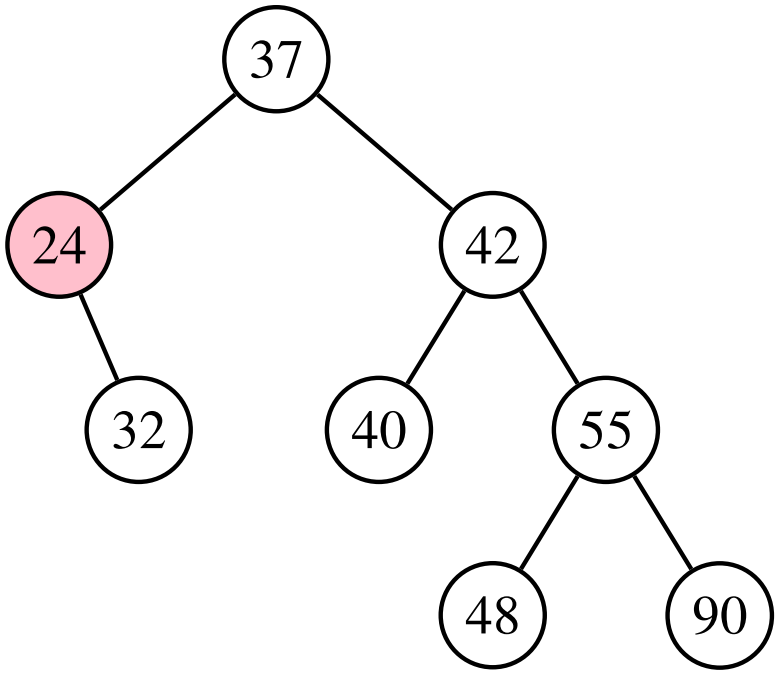
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2,

Binary Search Trees - traverseTree

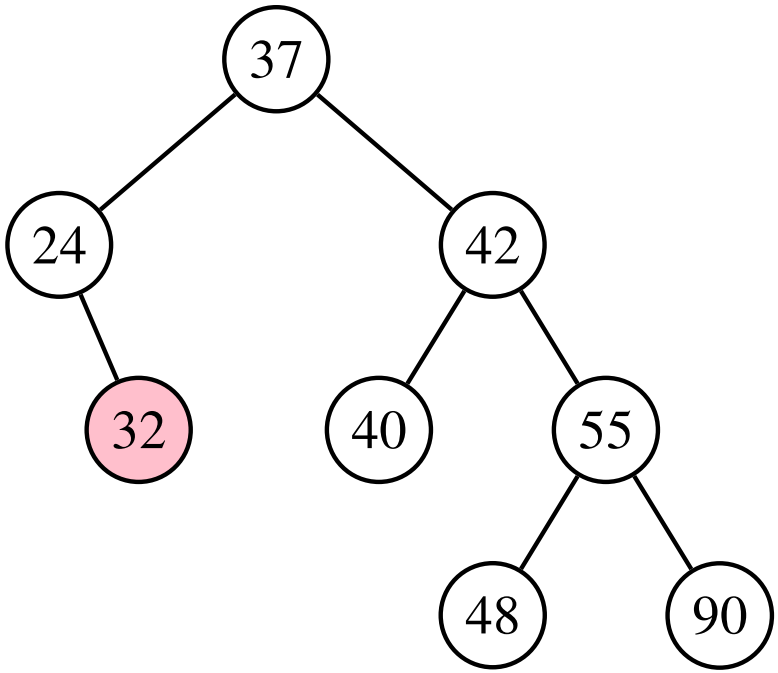
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2,

Binary Search Trees - traverseTree

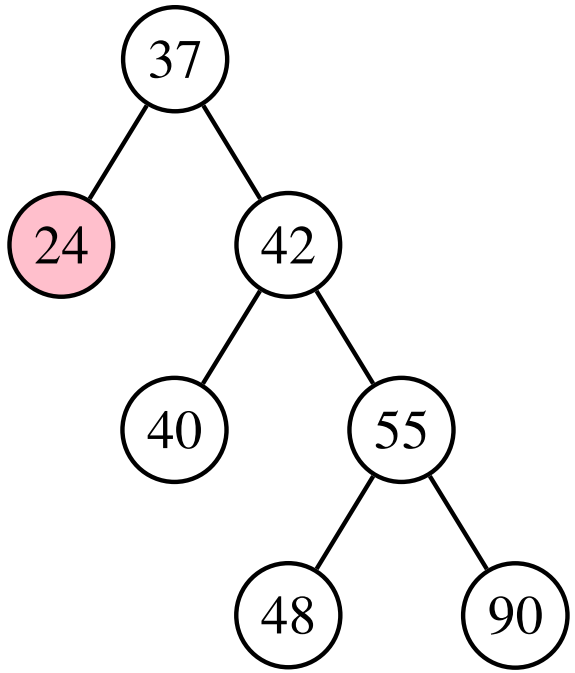
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2, 32,

Binary Search Trees - traverseTree

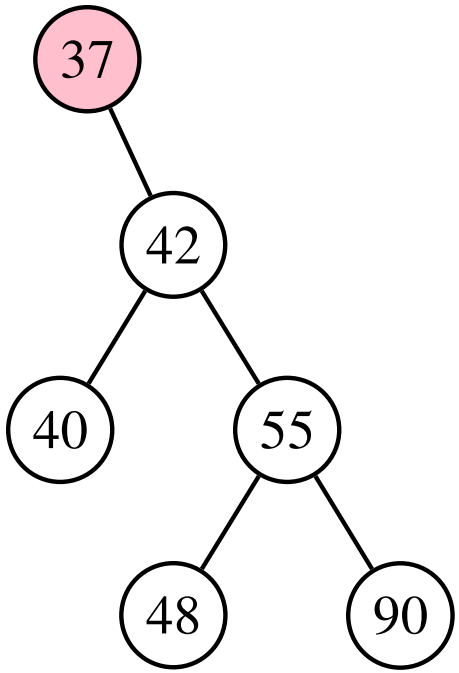
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2, 32,

Binary Search Trees - traverseTree

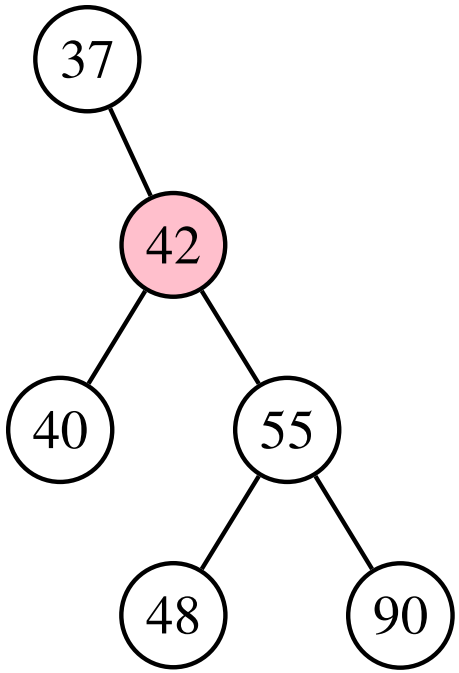
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2, 32,

Binary Search Trees - traverseTree

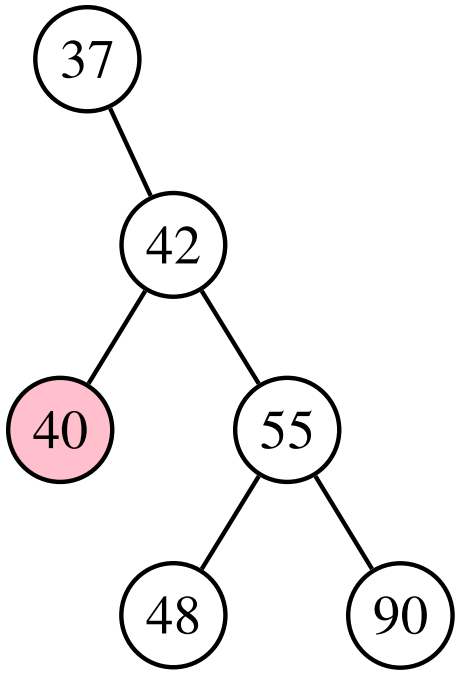
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2, 32, 42,

Binary Search Trees - traverseTree

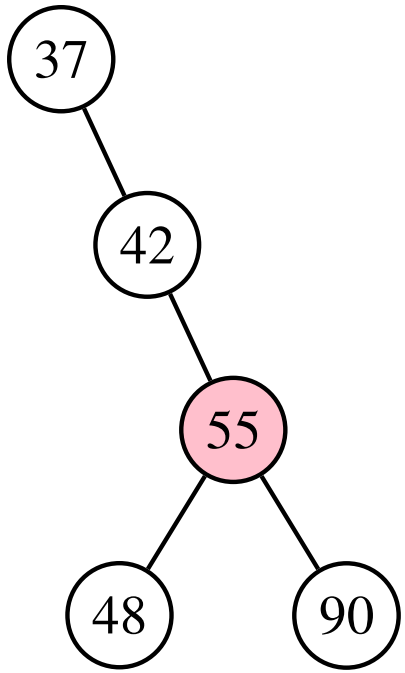
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2, 32, 42, 40,

Binary Search Trees - traverseTree

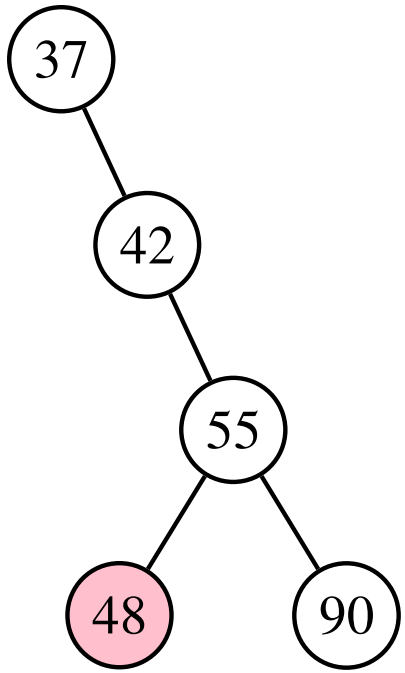
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2, 32, 42, 40, 55,

Binary Search Trees - traverseTree

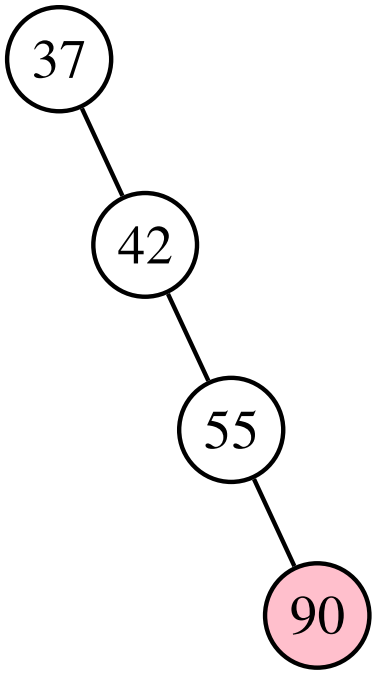
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2, 32, 42, 40, 55, 48,

Binary Search Trees - traverseTree

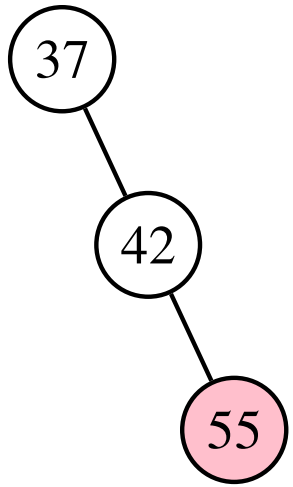
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2, 32, 42, 40, 55, 48, 90,

Binary Search Trees - traverseTree

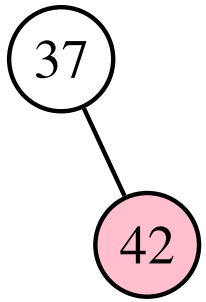
```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2, 32, 42, 40, 55, 48, 90,

Binary Search Trees - traverseTree

```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```



Output: 37, 24, 7, 2, 32, 42, 40, 55, 48, 90,

Binary Search Trees - traverseTree

```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```

Output: 37, 24, 7, 2, 32, 42, 40, 55, 48, 90,

Binary Search Trees - traverseTree

```
1 void printNode(TNode *t){
2     printf("%d, ",t->data);
3 }
4 traverseTree(t,printNode,NULL,(void (*)(TNode *))free);
```

Finish

Output: 37, 24, 7, 2, 32, 42, 40, 55, 48, 90.

Next Topic

- Binary Search
  - → Search: $\Theta(\log n)$, Insertions/Deletions: $\Theta(n)$
- Binary Search Trees
  - → Search: $\Theta(\log n)$, Insertions/Deletions: $\Theta(\log n)$ (average case).
- **Hashing**
  - → Search: $\Theta(1)$, Insertions/Deletions: $\Theta(1)$ (average case).