COMP2521 23T3
Binary Search Trees

Kevin Luxa
cs2521@cse.unsw.edu.au

trees
binary search trees

Trees

Trees

A tree is a branched data structure
consisting of a set of connected nodes where:

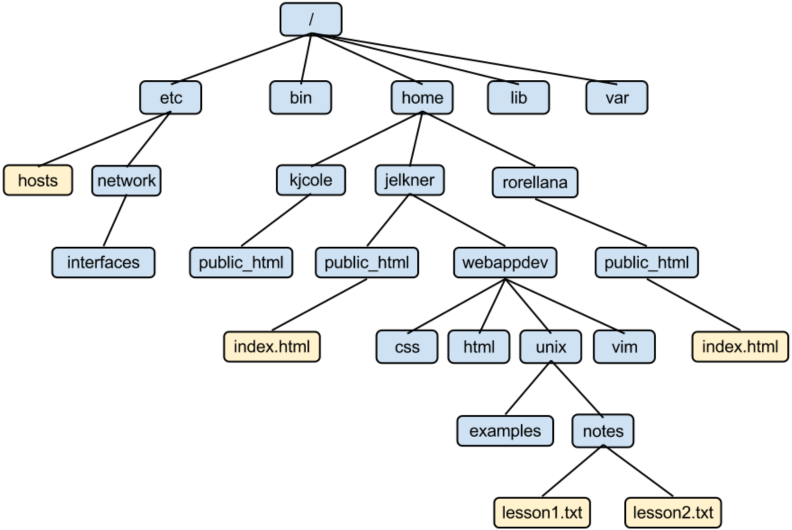
each node may have multiple other nodes as children
(depending on the type of tree)

each node is connected to one parent *except* the root node

trees do not contain cycles

Trees

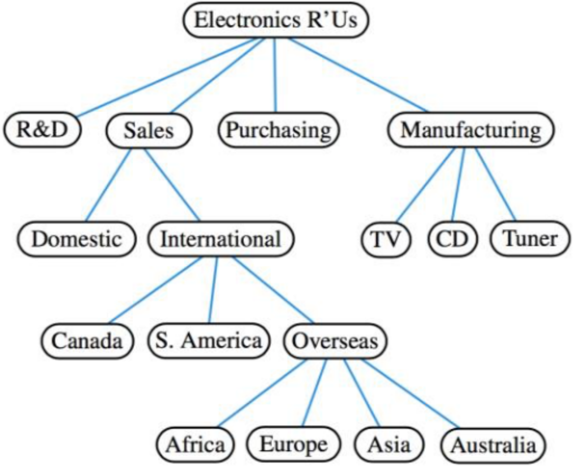
Example - File System Tree



Source: https://www.openbookproject.net/tutorials/getdown/unix/lesson2.html

Trees

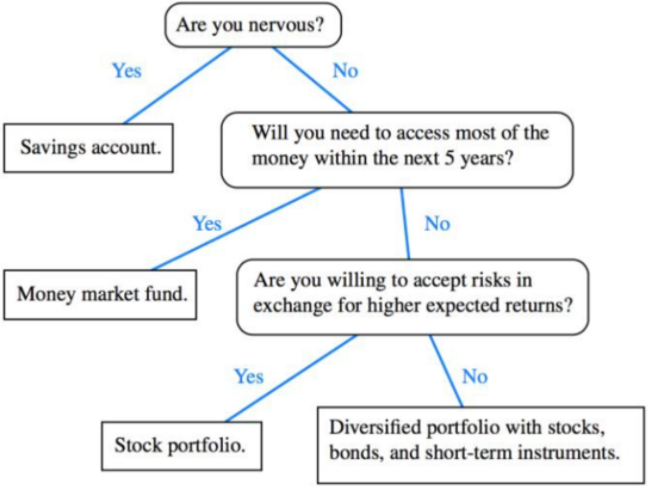
Example - Organisational Structure



Source: ”Data Structures and Algorithms in Java” (6th ed) by Goodrich et al.

Trees

Example - Decision Tree



Are you nervous?

Savings account.

Will you need to access most of the money within the next 5 years?

Money market fund.

Are you willing to accept risks in exchange for higher expected returns?

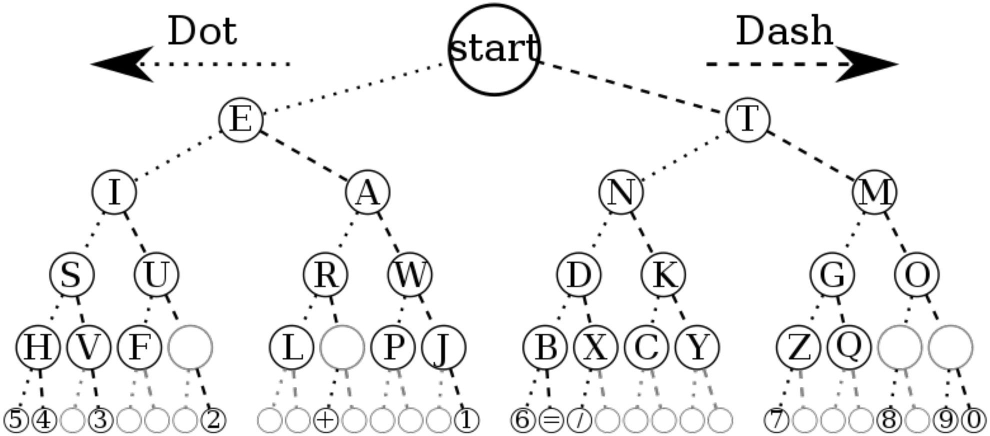
Stock portfolio.

Diversified portfolio with stocks, bonds, and short-term instruments.

Source: ”Data Structures and Algorithms in Java” (6th ed) by Goodrich et al.

Trees

Example - Decoding Morse Code



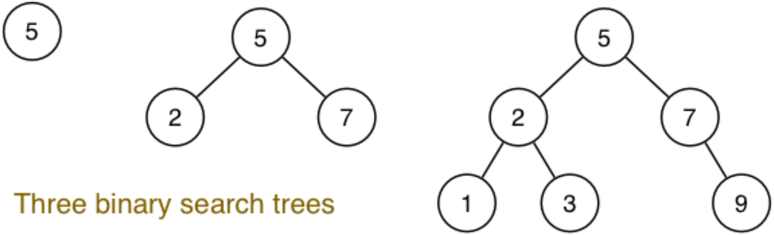
Binary Trees

Binary trees are trees where
each node can have up to two child nodes,
typically called the left child and right child

Binary Search Trees

A binary search tree is an ordered binary tree, where for each node:

- All values in the left subtree are less than the value in the node
- All values in the right subtree are greater than the value in the node

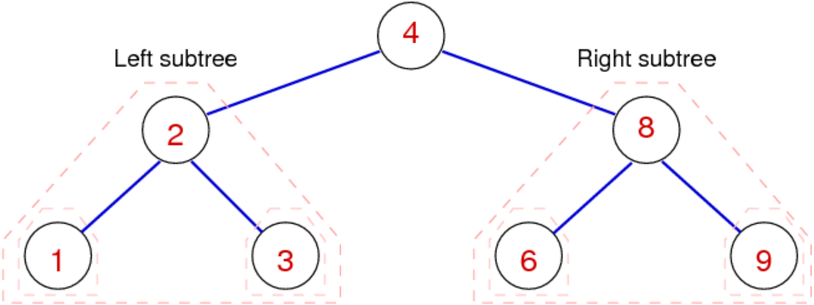


Three binary search trees

Binary Search Trees

A binary search tree is either:

- empty; or
- consists of a node with two subtrees
    - node contains a value
    - left and right subtrees are also BSTs (recursive)



Binary Search Trees

Why?

Why use binary search trees?

Search is an extremely common operation in computing:

- selecting records in databases
- searching for pages on the web

Typically, there is a very large amount of data (very many items)

Binary Search Trees

Why?

We've explored multliple approaches for searching:

- Ordered array
  - Searching/finding insertion point is $O(\log n)$ due to binary search
  - Inserting is $O(n)$ due to the need to shift items to preserve sortedness
- Ordered linked list
  - Searching/finding insertion point is $O(n)$ due to the nature of linked lists
  - Inserting *once we have found the insertion point* is $O(1)$ as there is no need to shift

Binary Search Trees

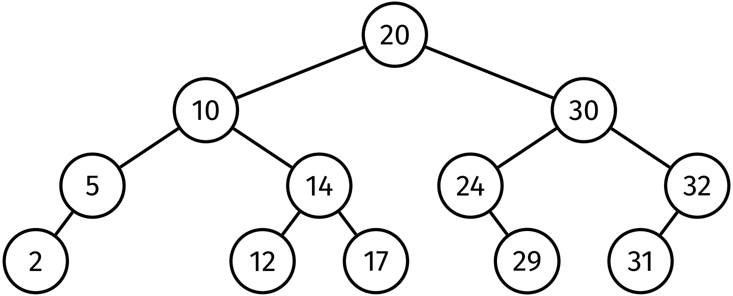
Why?

Binary search trees are efficient to search *and* maintain:

- Searching in a binary search tree is similar to how binary search works
  - Explained below
- A binary search tree is a linked data structure (like a linked list), so there is no need to shift elements when inserting/deleting

Binary Search Trees

2	5	10	12	14	17	20	24	29	30	31	32



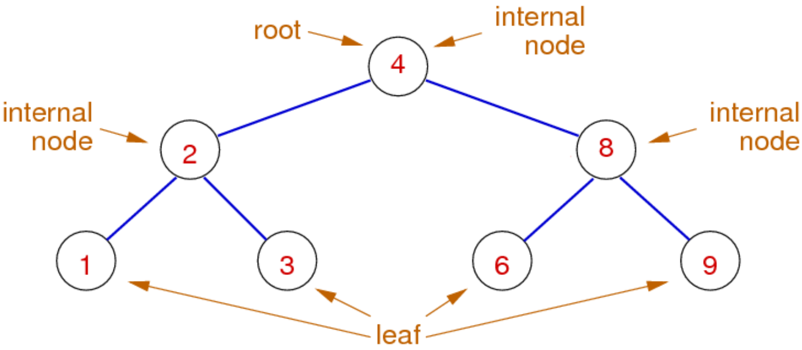
Binary Search Trees

Terminology

The root node is the node with no parent node.

A leaf node is a node that has no child nodes.

An internal node is a node that has at least one child node.

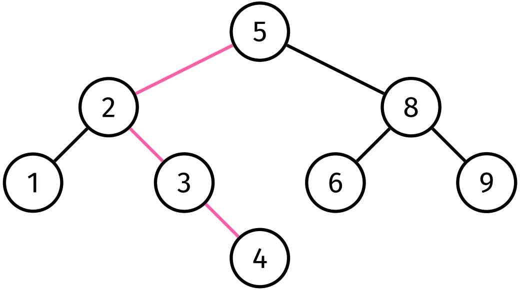


Binary Search Trees

Terminology

Height of a tree: Maximum path length from the root node to a leaf

- The height of an empty tree is considered to be -1
- The height of the following tree is 3

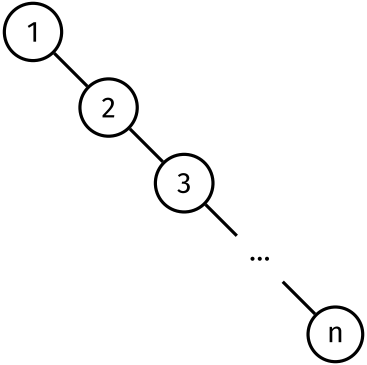


Binary Search Trees

Terminology

For a tree with n nodes:

The maximum possible height is $n - 1$

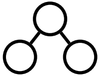
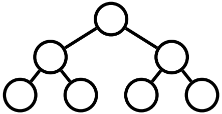


Binary Search Trees

Terminology

For a tree with n nodes:

The minimum possible height is $\lfloor \log_2 n \rfloor$

n	minimum height	tree
1	0	
2-3	1	
4-7	2	
...	...	...

Binary Search Trees

Terminology

For a given number of nodes, a tree is said to be balanced if it has (close to) minimal height, and degenerate if it has (close to) maximal height.

Binary Search Trees

Terminology

The height h of a binary search tree
determines the efficiency of many operations,
so we will use both n and h when expressing time complexities.

Binary Search Trees

Concrete Representation

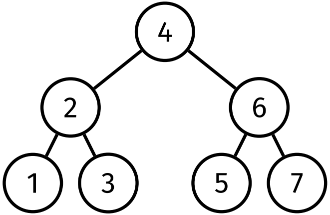
Binary trees are typically represented by node structures

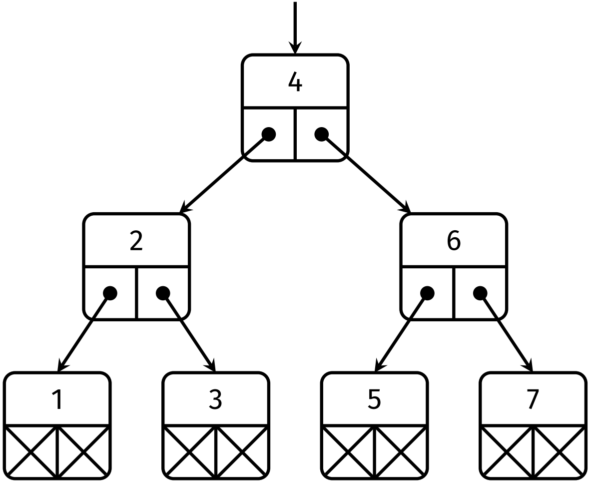
- Where each node contains a value and pointers to child nodes

```
struct node {
    int item;
    struct node *left;
    struct node *right;
};
```

Binary Search Trees

Concrete Representation





Binary Search Trees

Operations

Key operations on binary search trees:

- Insert
- Search
- Traversal
- Join
- Delete

BST Insertion

Insertion

`bstInsert(t, v)`

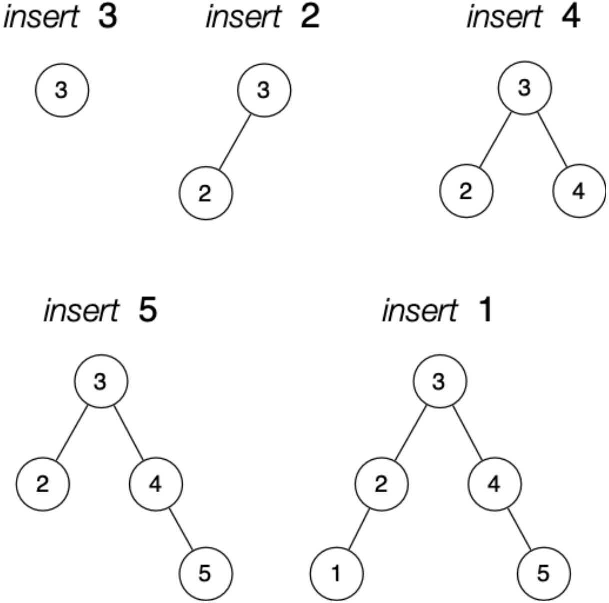
Given a BST t and a value v,
insert v into the BST
and return the root of the updated BST

BST Insertion

Insertion is straightforward:

- Start at the root
- Compare value to be inserted with value in the node
  - If value being inserted is less, descend to left child
  - If value being inserted is greater, descend to right child
- Repeat until...
  you have to go left/right but current node has no left/right child
  - Create new node and attach to current node

BST Insertion



BST Insertion

Example 1

Insert the following into an empty tree:

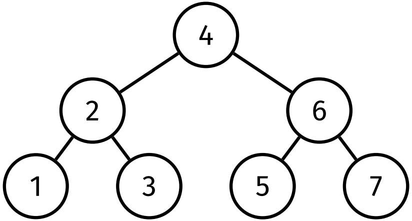
4 2 6 5 1 7 3

BST Insertion

Example 1

Insert the following into an empty tree:

4 2 6 5 1 7 3



BST Insertion

Example 2

Insert the following into an empty tree:

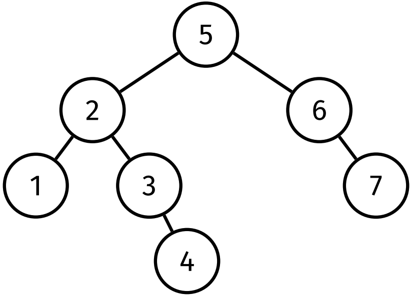
5 6 2 3 4 7 1

BST Insertion

Example 2

Insert the following into an empty tree:

5 6 2 3 4 7 1



BST Insertion

Example 3

Insert the following into an empty tree:

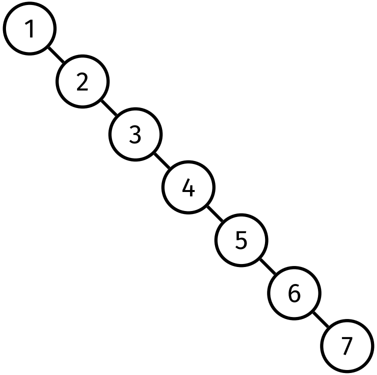
1 2 3 4 5 6 7

BST Insertion

Example 3

Insert the following into an empty tree:

1 2 3 4 5 6 7



BST Insertion

BST insertion can be implemented recursively.

Cases:

- t is empty
  $\Rightarrow$ make a new node with v as the root of the new tree
- $v <$ t`->item`
  $\Rightarrow$ insert v into t's left subtree
- $v >$ t`->item`
  $\Rightarrow$ insert v into t's right subtree
- $v =$ t`->item`
  $\Rightarrow$ tree unchanged (assuming no duplicates)

EXERCISE Try writing an iterative version.

BST Insertion

Pseudocode

```
bstInsert($t$, $v$):
    Inputs: tree $t$, value $v$
    Output: $t$ with $v$ inserted

    if $t$ is empty:
        return new node containing $v$
    else if $v$ < $t$->item:
        $t$->left = bstInsert($t$->left, $v$)
    else if $v$ > $t$->item:
        $t$->right = bstInsert($t$->right, $v$)

    return $t$
```

BST Insertion

Analysis

Analysis:

- At most one node is examined on each level
- Number of operations performed per node is constant
- Therefore, the worst-case time complexity of insertion is $O(h)$ where h is the height of the BST

BST Search

Search

`bstSearch(t, v)`

Given a BST t and a value v,
return true if v is in the BST and false otherwise

BST Search

BST search can be implemented recursively.

Cases:

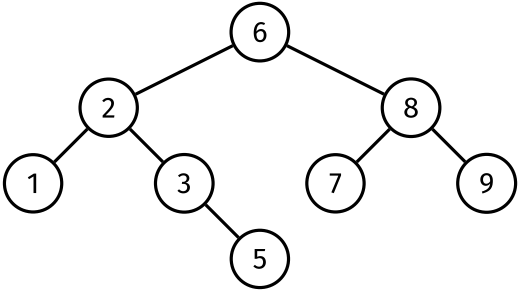
- t is empty:
  $\Rightarrow$ return false
- $v <$ t`->item`
  $\Rightarrow$ search for v in t's left subtree
- $v >$ t`->item`
  $\Rightarrow$ search for v in t's right subtree
- $v =$ t`->item`
  $\Rightarrow$ return true

EXERCISE Try writing an iterative version.

BST Search

Example

Search for 4 and 7 in the following BST:



BST Search

Pseudocode

```
bstSearch($t$, $v$):
    Inputs: tree $t$, value $v$
    Output: true if $v$ is in $t$
            false otherwise

    if $t$ is empty:
        return false
    else if $v$ < $t$->item:
        return bstSearch($t$->left, $v$)
    else if $v$ > $t$->item:
        return bstSearch($t$->right, $v$)
    else:
        return true
```

BST Search

Analysis

Analysis:

- At most one node is examined on each level
- Number of operations performed per node is constant
- Therefore, the worst-case time complexity of search is $O(h)$ where h is the height of the BST

Tree Traversal

To traverse a linked list, we simply traverse from start to end.

There are 4 common ways to traverse a binary tree:

1. Pre-order (NLR):
   visit root, then traverse left subtree, then traverse right subtree
2. In-order (LNR):
   traverse left subtree, then visit root, then traverse right subtree
3. Post-order (LRN):
   traverse left subtree, then traverse right subtree, then visit root
4. Level-order:
   visit root, then its children, then their children, and so on

Tree Traversal

Pseudocode

Pseudocode:

```
preorder($t$):
    **Inputs:** tree $t$

    **if** $t$ is empty:
        **return**

    visit($t$)
    preorder($t$->left)
    preorder($t$->right)
```

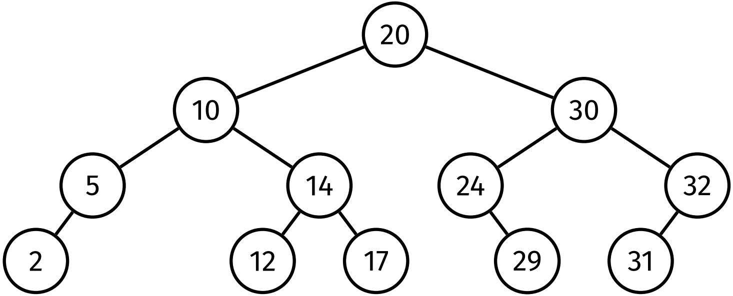
```
inorder($t$):
    **Inputs:** tree $t$

    **if** $t$ is empty:
        **return**

    inorder($t$->left)
    visit($t$)
    inorder($t$->right)
```

```
postorder($t$):
    **Inputs:** tree $t$

    **if** $t$ is empty:
        **return**

    postorder($t$->left)
    postorder($t$->right)
    visit($t$)
```

Note:
Level-order traversal is difficult to implement recursively.
It is typically implemented using a queue.

Tree Traversal

Example: Binary Search Tree

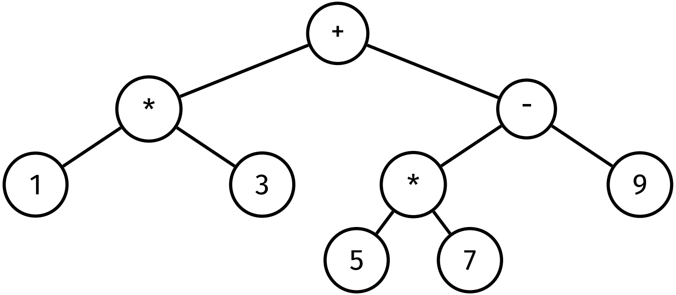


Pre-order	20 10 5 2 14 12 17 30 24 29 32 31
In-order	2 5 10 12 14 17 20 24 29 30 31 32
Post-order	2 5 12 17 14 10 29 24 31 32 30 20
Level-order	20 10 30 5 14 24 32 2 12 17 29 31

Tree Traversal

Example: Expression Tree

Expression tree for 1 * 3 + (5 * 7 - 9)



Pre-order + * 1 3 - * 5 7 9

In-order 1 * 3 + 5 * 7 - 9

Post-order 1 3 * 5 7 * 9 - +

Tree Traversal

Applications

Pre-order traversal:

- Useful for reconstructing a tree

In-order traversal:

- Useful for traversing a BST in ascending order

Post-order traversal:

- Useful for evaluating an expression tree
- Useful for freeing a tree

Level-order traversal:

- Useful for printing a tree

Tree Traversal

Analysis

Analysis:

- Each node is visited once
- Hence, time complexity of tree traversal is $O(n)$, where n is the number of nodes

BST Join

Join

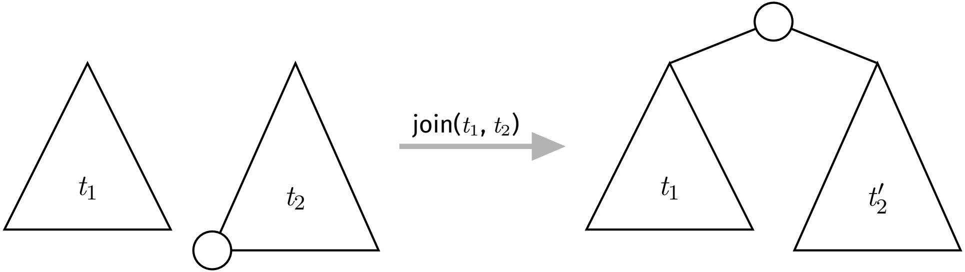
`bstJoin(t1, t2)`

Given two BSTs t_1 and t_2
where $\max(t_1) < \min(t_2)$
return a BST containing all items from t_1 and t_2

BST Join

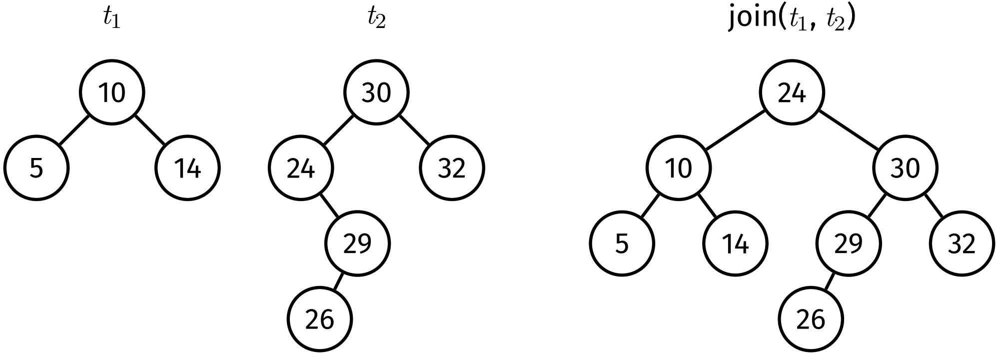
Method:

1. Find the minimum node min in t_2
2. Replace min by its right subtree (if it exists)
3. Elevate min to be the new root of t_1 and t_2



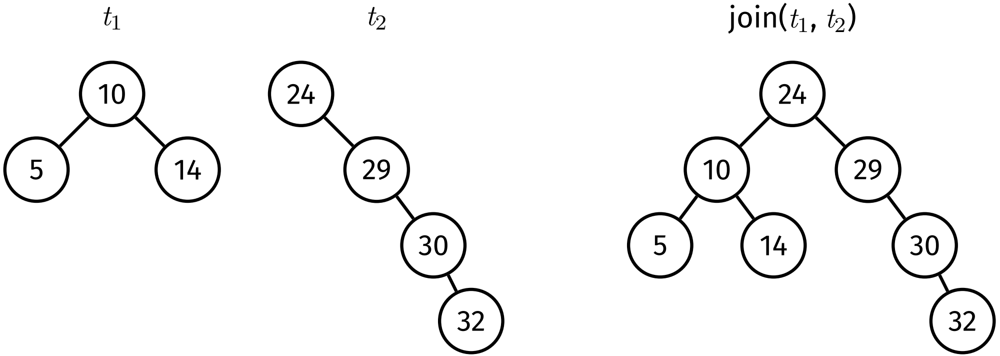
BST Join

Example 1



BST Join

Example 2



BST Join

Pseudocode

```
bstJoin($t_1$, $t_2$):
    Inputs: trees $t_1$, $t_2$
    Output: $t_1$ and $t_2$ joined together

    if $t_1$ is empty:
        return $t_2$
    else if $t_2$ is empty:
        return $t_1$
    else:
        curr = $t_2$
        parent = NULL
        while curr->left $\neq$ NULL:
            parent = curr
            curr = curr->left

        if parent $\neq$ NULL:
            parent->left = curr->right
            curr->right = $t_2$

        curr->left = $t_1$
        return curr
```

BST Join

Analysis

Analysis:

- The join algorithm simply finds the minimum node in t_2
- Thus, at most one node is visited per level of t_2
- Therefore, the worst-case time complexity of join is $O(h_2)$ where h_2 is the height of t_2

BST Deletion

Deletion

`bstDelete(t, v)`

Given a BST t and a value v
delete v from the BST
and return the root of the updated BST

BST Deletion

BST deletion can be implemented recursively.

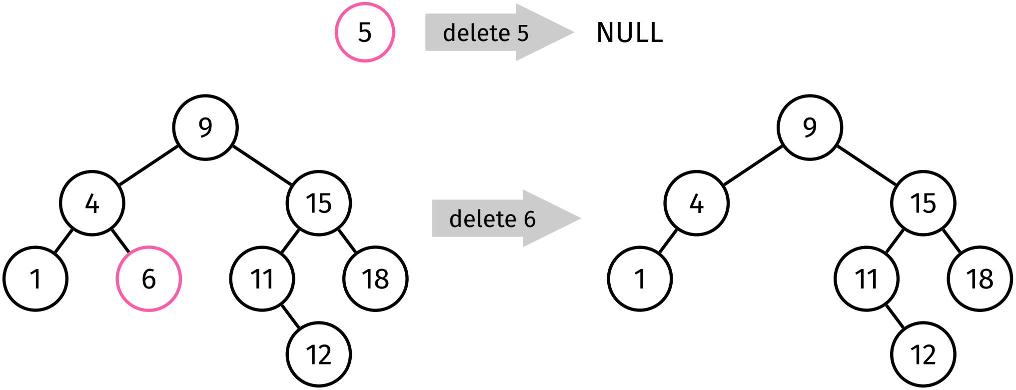
Cases:

- t is empty:
  $\Rightarrow$ result is empty
- $v <$ t`->item`
  $\Rightarrow$ delete v from t's left subtree
- $v >$ t`->item`
  $\Rightarrow$ delete v from t's right subtree
- $v =$ t`->item`
  $\Rightarrow$ three sub-cases:
  - t is a leaf
    $\Rightarrow$ result is empty tree
  - t has one subtree
    $\Rightarrow$ replace with subtree
  - t has two subtrees
    $\Rightarrow$ join the two subtrees

BST Deletion

Zero subtrees

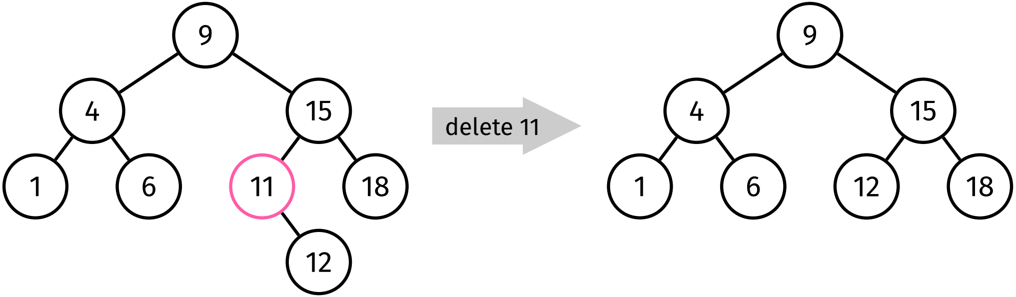
If the node being deleted is a leaf, then the result is an empty tree



BST Deletion

One subtree

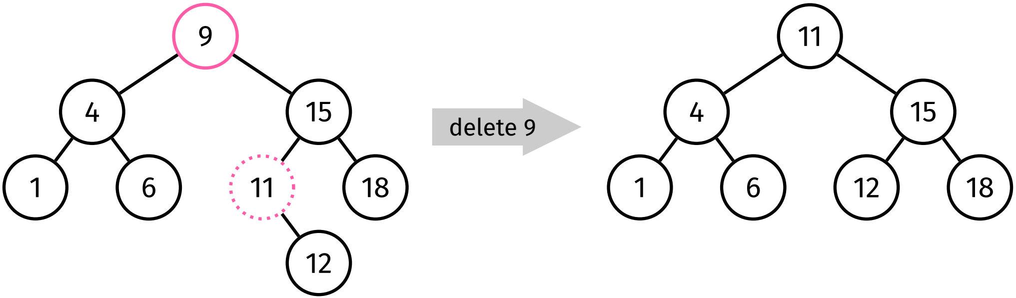
Node to be deleted has one subtree



BST Deletion

Two subtrees

Node to be deleted has two subtrees



BST Deletion

Pseudocode

```
bstDelete($t$, $v$):
    **Inputs:** tree $t$, value $v$
    **Output:** $t$ with $v$ deleted

    if $t$ is empty:
        return empty tree
    else if $v < t$->item:
        $t$->left = bstDelete($t$->left, $v$)
    else if $v > t$->item:
        $t$->right = bstDelete($t$->right, $v$)
    else:
        if $t$->left is empty:
            new = $t$->right
        else if $t$->right is empty:
            new = $t$->left
        else:
            new = bstJoin($t$->left, $t$->right)

        free($t$)
        $t$ = new

    return $t$
```

BST Deletion

Analysis

Analysis:

- The deletion algorithm traverses down just one branch
    - First, the item being deleted is found
    - If the item exists and has two subtrees, its successor is found
- Thus, at most one node is visited per level
- Therefore, the worst-case time complexity of deletion is $O(h)$ where h is the height of the BST

BST Exercises

- `bstFree`
  free a tree
- `bstSize`
  return the size of a tree
- `bstHeight`
  return the height of a tree
- `bstPrune`
  given values lo and hi, remove all values outside the range $[lo, hi]$

Feedback

`https://forms.office.com/r/aPF09YHZ3X`