ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

Module 9 (Part I)

Binary Search Trees, Huffman Trees

Dayu Wang

- **Binary Search Trees (BST)**

  → Binary search tree does not allow ~~duplicated values~~ stored in the same tree.

  Therefore, we call the data stored in a node the key of that node.

  → In a binary search tree:

  1) The key of each node is greater than all the keys in the left subtree.
  2) The key of each node is less than all the keys in the right subtree.

  → [Search Example] Search for 4.

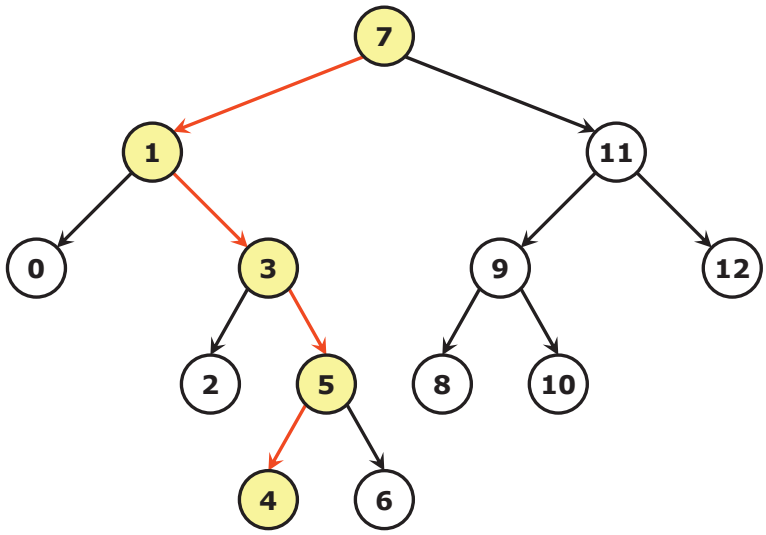
  - Compare 4 with the root's key, 7.

    4 < 7, so 4 resides in the left subtree.

    Go to the left subtree.
  - Compare 4 with the root's key, 1.

    4 > 1, so 4 resides in the right subtree.

    Go to the right subtree.
  - 4 > 3, so go to the right subtree.
  - 4 < 5, so go to the left subtree.
  - 4 is found.

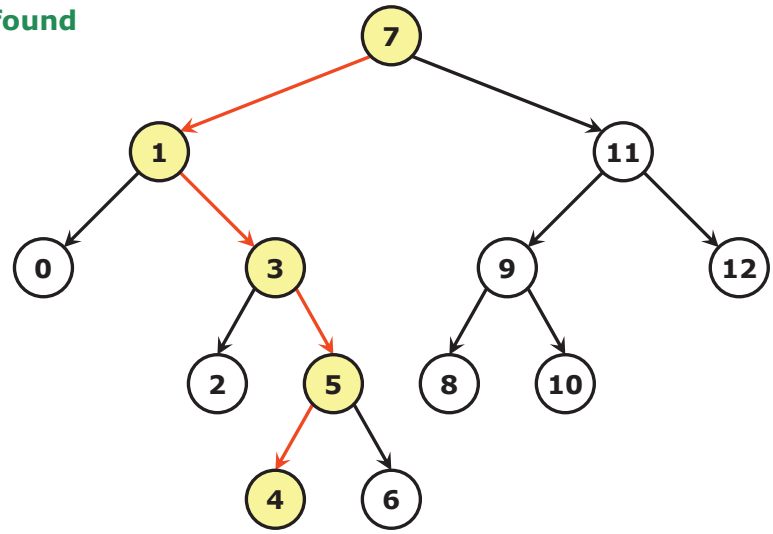


• **Search Algorithm in Binary Search Trees**

→ **Algorithm Search(target)**

```
If the tree is empty
    Return  // Target not found
If target equals to root's key
    Return a pointer to that data  // Target found
If target less than root's key
    Return Search(left subtree)
Else
    Return Search(right subtree)
```

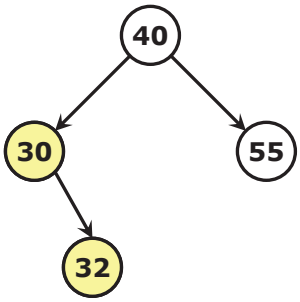
→ **[Example] Search(4)**



• **[Exercise]**

→ **For each of the following binary trees, answer the questions below:**

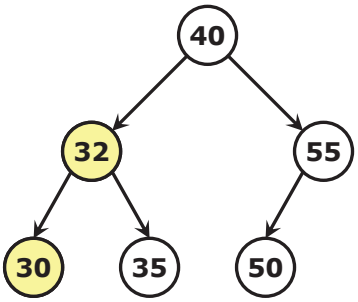
1) **Is it a full tree? Explain your answer.**
2) **Is it a complete tree? Explain your answer.**
3) **What is the height of the tree?**
4) **Is it a binary search tree? If not, how to make it a binary search tree?**



Not full

Not complete

Not a binary search tree

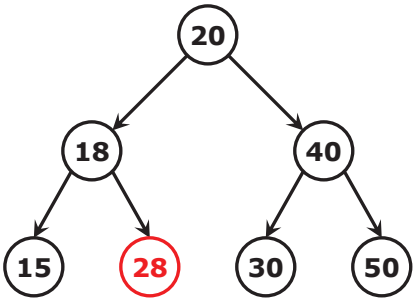


Not full

Complete

Not a binary search tree

• **How to test whether a binary tree is a binary search tree?**



To test whether a binary tree is a binary search tree, you need:

1) **Test whether the root's key is greater than the maximum key in the left subtree.**
2) **Test whether the root's key is less than the minimum key in the right subtree.**
3) **Test whether the left and right subtree are also binary search trees.**

→ **This algorithm tests whether "each node's key is greater than its left child and less than its right child".**

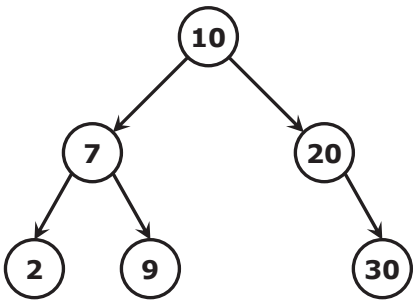
→ **Is this algorithm correct?**

The algorithm is incorrect.

→ **Study resources:** https://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not
https://leetcode.com/problems/validate-binary-search-tree

• **[Exercise] Traversal of Binary Search Tree**

→ **Given a binary search tree below, write the sequence of keys using preorder traversal, inorder traversal, and postorder traversal.**



→ **Preorder traversal:**

10, 7, 2, 9, 20, 30

→ **Inorder traversal:**

2, 7, 9, 10, 20, 30

→ **Postorder traversal:**

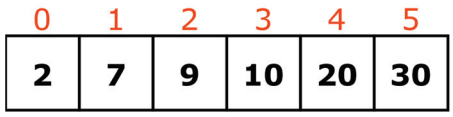
2, 9, 7, 30, 20, 10

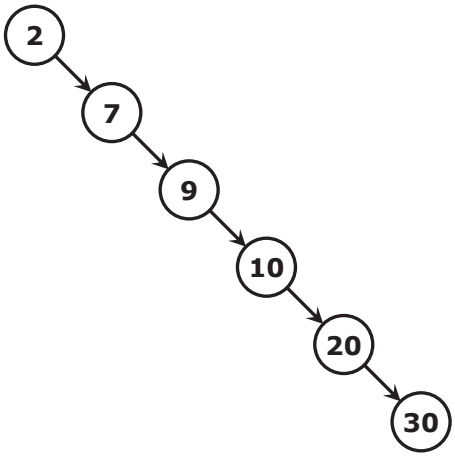
→ **[Conclusion] Inorder traversal of a binary search tree generates a sorted list of keys.**

• **[Exercise] Generating a sorted vector from a binary search tree**

```
1  /** Generates a sorted vector from a binary search tree.
2      @param root: a pointer to the root node of the binary search tree
3      @return: a sorted vector generated from the binary search tree
4  */
5  template<class T>
6  vector<T> to_vector(const BTNode<T>* root) {
7      // Base case
8      if (!root) { return vector<T>(); }
9      // Generate a sorted vector representing the left subtree.
10     vector<T> left;
11     if (root->left) { left = to_vector(root->left); }
12     // Generate a sorted vector representing the right subtree.
13     vector<T> right;
14     if (root->right) { right = to_vector(root->right); }
15     // Generate the result vector.
16     vector<T> result;
17     result.insert(result.end(), left.begin(), left.end());
18     result.push_back(root->data);
19     result.insert(result.end(), right.begin(), right.end());
20     return result;
21 }  // Time complexity: O(n)
```


• **[Exercise] Generating a balanced binary search tree from a sorted vector**





This is a binary search tree, but not ~~balanced binary search tree~~.

→ **Algorithm**

If no value in the segment, return NULL.

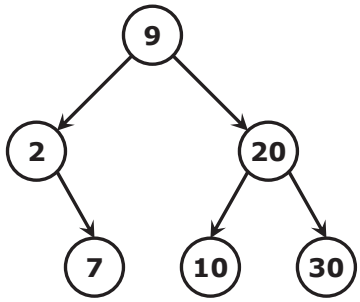
Pick the middle value as the root.

Recursively generate the left subtree.

Recursively generate the right subtree.

Connect the root with left and right subtree.

Return the root.



- **[Exercise] Generating a balanced binary search tree from a sorted vector**

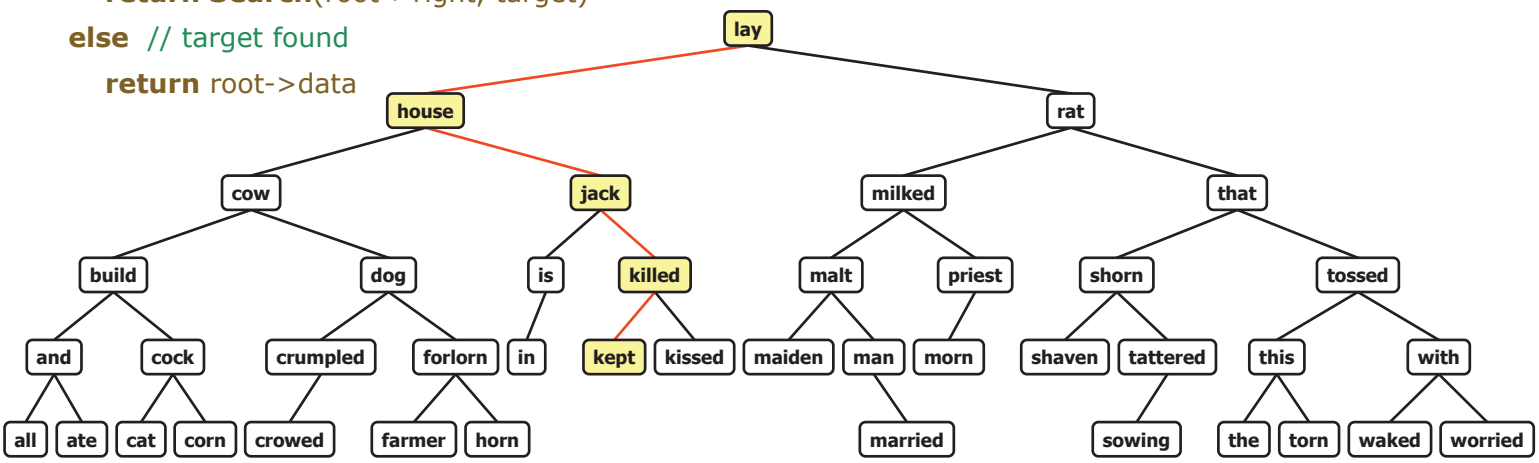
```cpp
/** Generates a balanced binary search tree from a segment of a sorted vector.
    @param vec: a sorted vector
    @param i: index of the beginning of the segment
    @param j: index of the end of the segment
    @return: pointer to the root of a BST generated from the vector
*/
template<class T>
BTNode<T>* to_bst(const vector<T>& vec, int i, int j) {
    if (i > j) { return NULL; }  // Base case
    int mid = (i + j) / 2;  // Get the middle value.
    BTNode<T>* left = to_bst(vec, i, mid - 1);  // Recursively generate the left subtree.
    BTNode<T>* right = to_bst(vec, mid + 1, j);  // Recursively generate the right subtree.
    return new BTNode<T>(vec.at(mid), left, right);
}  // Time complexity: O(n)

// Wrapper function
template<class T>
BTNode<T>* to_bst(const vector<T>& vec) { return to_bst(vec, 0, vec.size() - 1); }
```


- **Analysis of Binary Search Algorithm**

→ **Algorithm Search**(root, target)

```
if root == NULL
    return NULL  // target not in tree
else if target < root->data
    return Search(root->left, target)
else if target > root->data
    return Search(root->right, target)
else  // target found
    return root->data
```

[Example] Searching for "kept"



- **Analysis of Binary Search Algorithm**

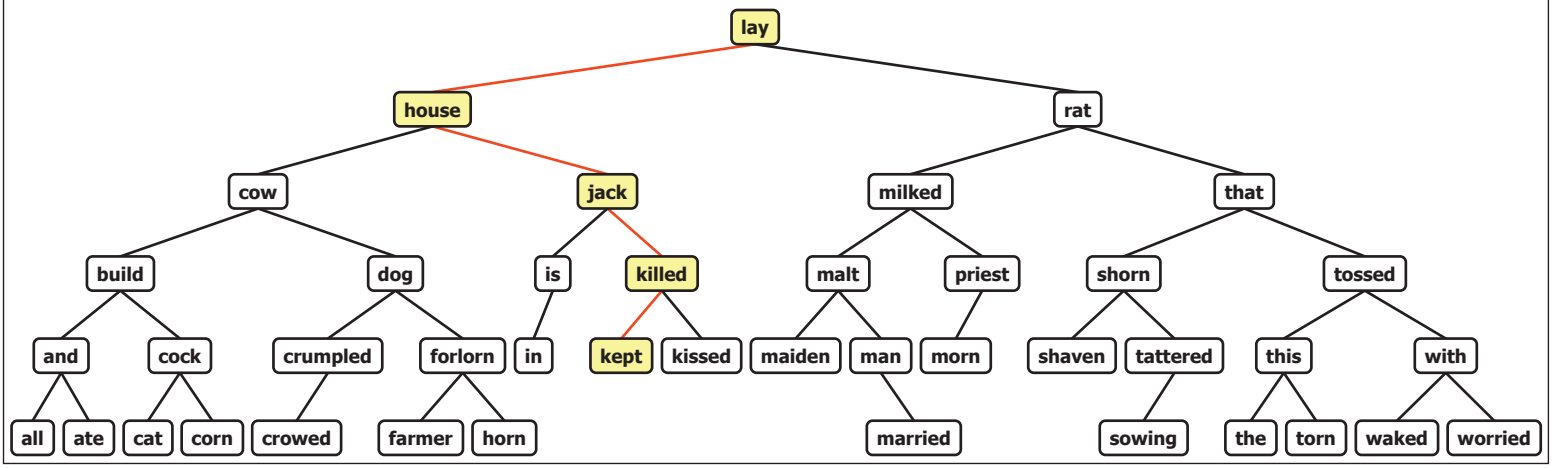
→ In this example, the binary search tree has 42 keys.

Searching for "kept" took 5 steps.

$\log 42 \approx 5.3$

→ Time complexity of binary search is $O(\log n)$, under one condition.

The tree is (almost) balanced.



- **Analysis of Binary Search Algorithm**

→ Let's search for "fox".

n = 4

Search for "fox" takes 4 steps.

→ Worst case time complexity of search:

$O(n)$

→ In general, search in binary search tree has time complexity of:

$O(h)$

where h is the height of the tree.

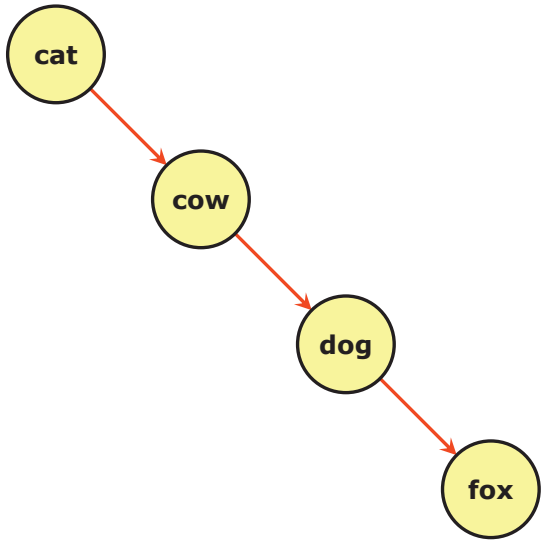
→ For full-filled binary search trees:

$n = 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = 2^h - 1$

$h = \log(n+1) = O(\log n)$

→ If the binary search tree is (almost) balanced, then

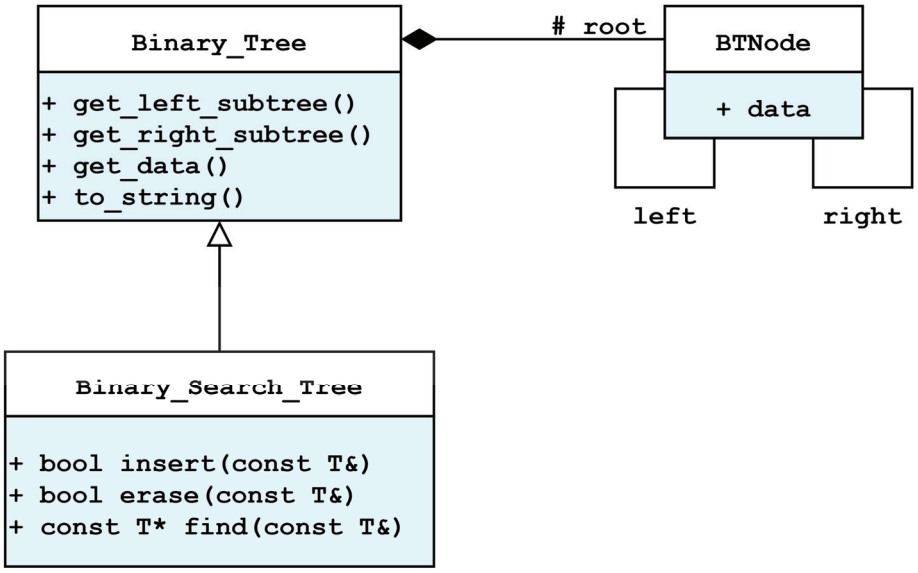
$O(h) = O(\log n)$



• **Implementing Binary Search Trees**

→ **Binary search tree is a derived class of binary tree.**

It inherits all the attributes from a binary tree.



• **Class-Member Functions**

Function	Behavior
`bool insert(const T&);`	Inserts a key into the tree. Returns `true` if the key was inserted; `false` if the key was already in the tree.
`bool erase(const T&);`	Deletes a key from the tree. Returns `true` if the key was deleted; `false` if the key was **not** already in the tree.
`const T* find(const T&) const;`	Returns a `const` pointer to the found key in the tree; or `NULL` if the key is **not** found in the tree.

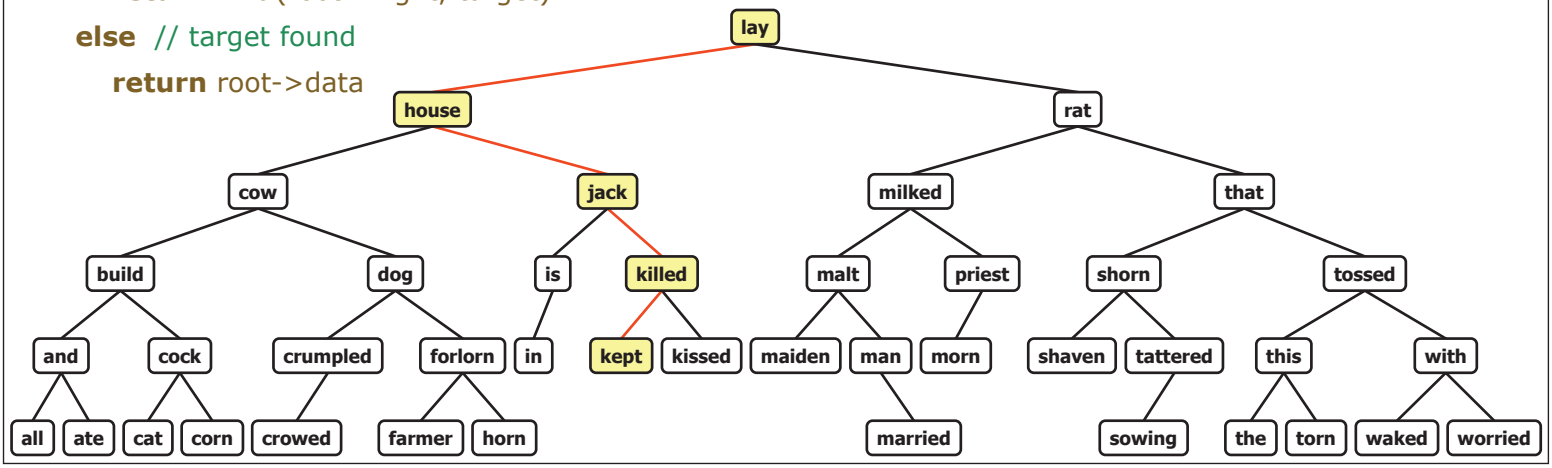
→ **[Do Not Forget] Binary search trees do not allow ~~duplicate keys~~.**

- **The "find" Algorithm**

→ **Algorithm find**(root, target)

```
if root == NULL
    return NULL  // target not in tree
else if target < root->data
    return find(root->left, target)
else if target > root->data
    return find(root->right, target)
else  // target found
    return root->data
```

[Example] Searching for "kept"

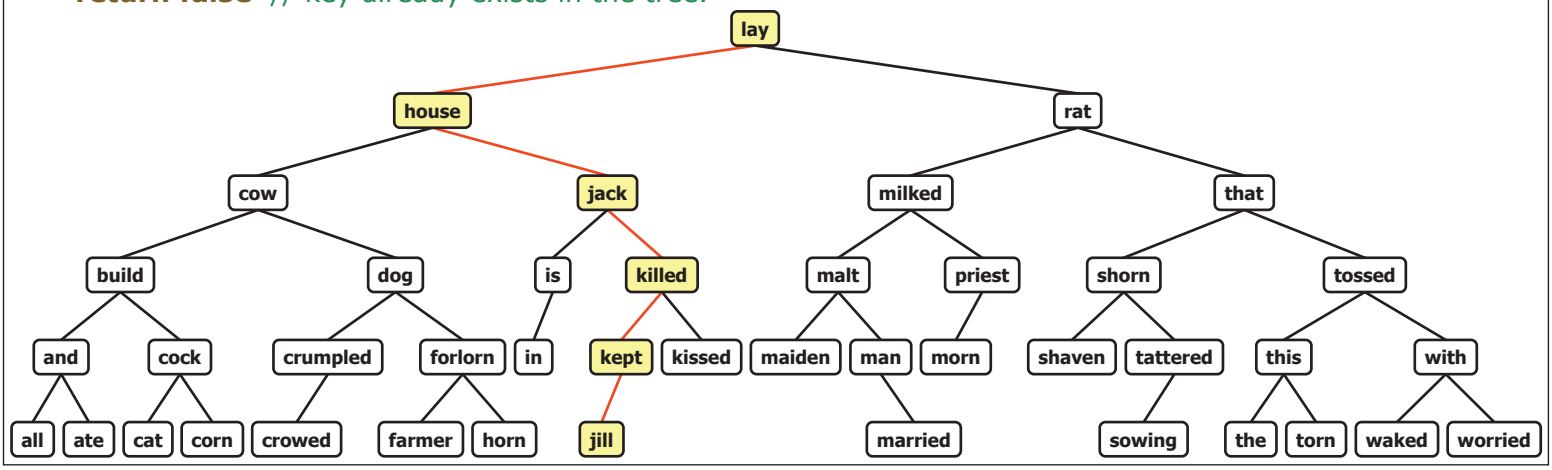


- **The "insert" Algorithm**

→ **Algorithm insert**(root, key)

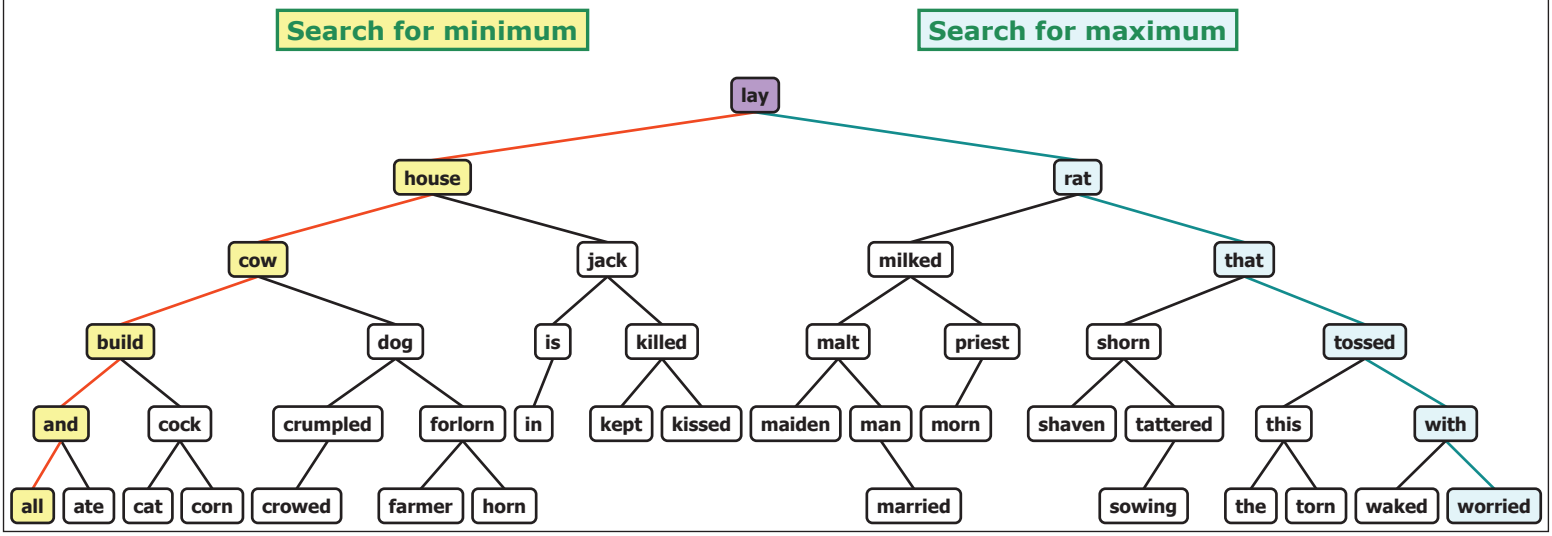
```
if root == NULL
    replace empty tree with new data leaf
    return true
if key < root->data return insert(root->left, key)
if key > root->data return insert(root->right, key)
return false  // key already exists in the tree.
```

Inserting "jill"



- **[Exercise] Write functions to returns the minimum and maximum key in a binary search tree.**

  → **In a binary tree, graphically where is the minimum key? Where is the maximum key?**



- **[Exercise] Write functions to returns the minimum and maximum key in a binary search tree.**

```
1  /** Returns the minimum key in a binary search tree.
2      @param root: a non-NULL pointer to the root node of the binary search tree
3      @return: a const reference to the minimum key in the binary search tree
4  */
5  template<class T>
6  const T& min_key(const BTNode<T>* root) {
7      BTNode<T>* p = root;
8      while (p->left) { p = p->left; }
9      return p->data;
10 }  // Time complexity: O(h)
11
12 /** Returns the maximum key in a binary search tree.
13     @param root: a non-NULL pointer to the root node of the binary search tree
14     @return: a const reference to the maximum key in the binary search tree
15 */
16 template<class T>
17 const T& max_key(const BTNode<T>* root) {
18     BTNode<T>* p = root;
19     while (p->right) { p = p->right; }
20     return p->data;
21 }  // Time complexity: O(h)
```

- **The "erase" Algorithm - How to delete a key from a binary search tree?**

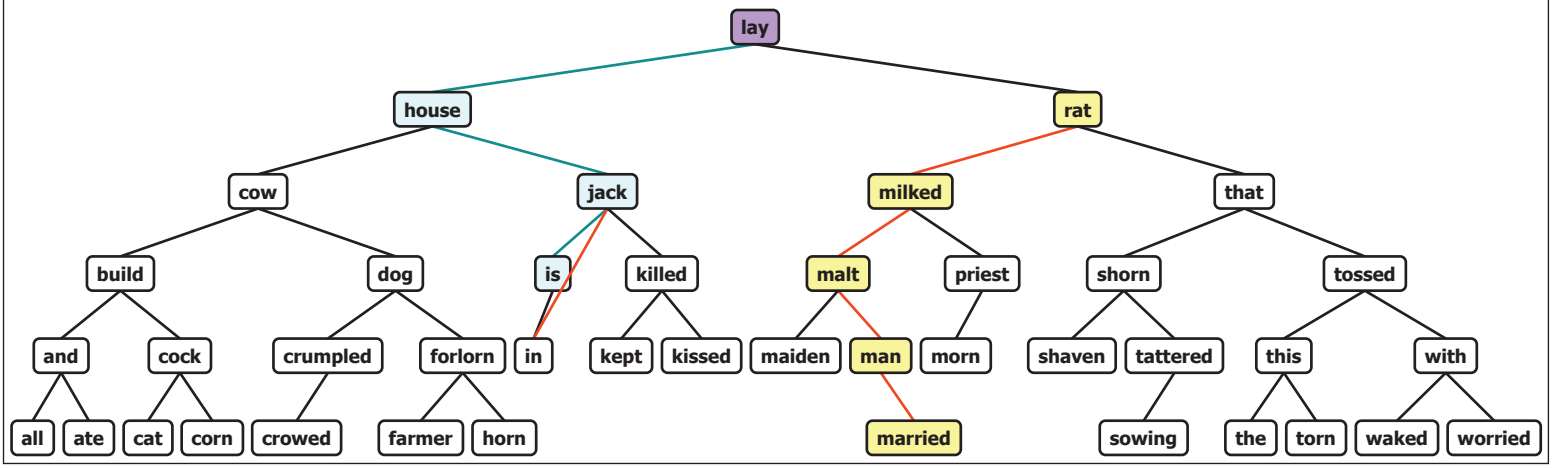
→ **Key not present: do nothing.**

→ **Key present in leaf: remove that leaf (change to NULL).**

[Example] Erasing "married"

→ **Key in non-leaf with one child: replace current node with that child.**

[Example] Erasing "is"



- **How to delete a key that has two children?**

→ **[Example] Erasing "house"**

Which nodes are good candidates to replace "house"?

~~**A. "cow" B. "jack" C. "all"**~~ ✅ **"horn"** ✅ **"in"** ~~**F. "kissed"**~~ **(more than one correct answer)**

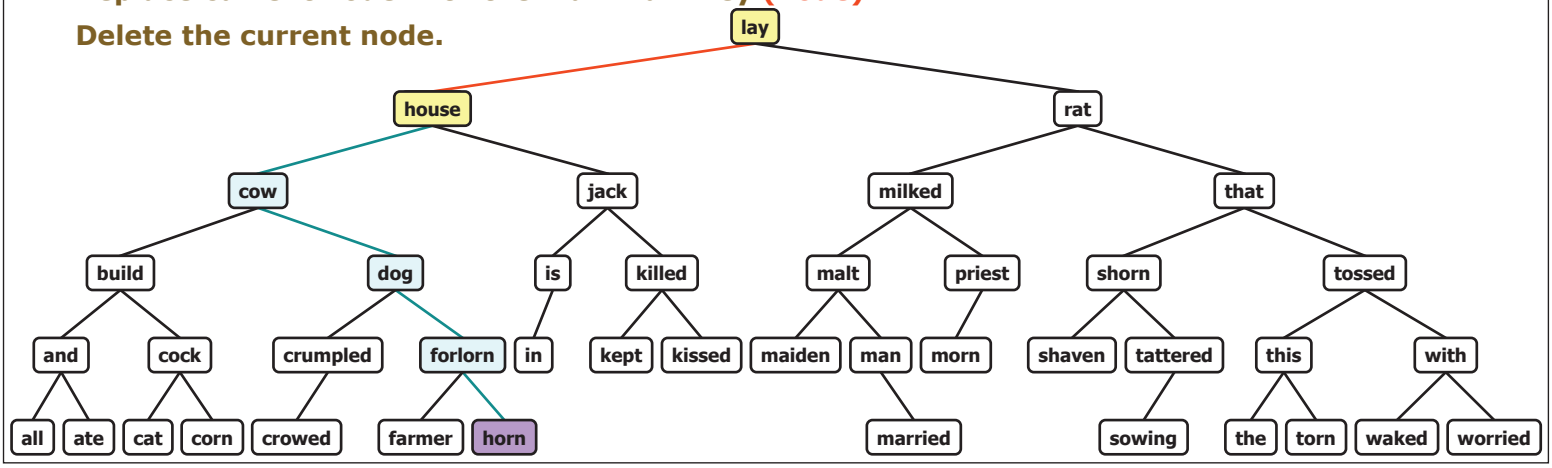
You can use either the max key in the left subtree or the min key in the right subtree.

→ **Delete key in non-leaf with two children:**

Find maximum key in the left subtree.

Replace current node with the maximum key (node)

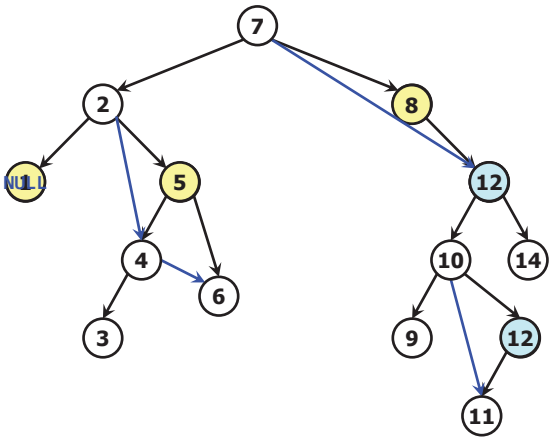
Delete the current node.



- **The algorithm written in your textbook is wrong!**

  → **It misses a step.**

Set the left child's right child to be the local root's right child.



Recursive Algorithm for Removal from a Binary Search Tree

```
1.  if the root is NULL
2.      The item is not in tree – return NULL.
3.  Compare the item to the data at the local root.
4.  if the item is less than the data at the local root
5.      Return the result of deleting from the left subtree.
6.  else if the item is greater than the local root
7.      Return the result of deleting from the right subtree.
8.  else  // The item is in the local root
9.      if the local root has no children
10.         Set the parent of the local root to reference NULL.
11.     else if the local root has one child
12.         Set the parent of the local root to reference that child.
13.     else  // Find the inorder predecessor
14.         if the left child has no right child,
                it is the inorder predecessor
15.             Set the parent of the local root to reference the left child.
16.         else
17.             Find the rightmost node in the right subtree of the left
                child.
18.             Copy its data into the local root's data and remove it by
                setting its parent to reference its left child.
```

Implementing the `erase` Functions

Listing 8.6 shows both the starter and the recursive `erase` functions. As with the `insert` function, the recursive `erase` function returns a `bool` value to indicate whether the item was removed or was not in the tree.

LISTING 8.6
`Binary_Search_Tree` `erase` Functions

```
template<typename Item_Type>
  bool Binary_Search_Tree<Item_Type>::erase(
    const Item_Type& item) {
  return erase(this->root, item);
}

template<typename Item_Type>
  bool Binary_Search_Tree<Item_Type>::erase(
    BTNode<Item_Type>*& local_root,
    const Item_Type& item) {
  if (local_root == NULL) {
    return false;
  } else {
    if (item < local_root->data)
      return erase(local_root->left, item);
```