15 Binary Search Trees

Instructor : Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

March 18, 2019

Outline

1. **Associative Arrays**
2. **Binary Search Trees**
   - Searching
   - Insertion
   - In-order Traversals
   - Deletion
3. **Balance of Binary Search Trees**

Associative Arrays

- Given an index i of an array-based list a, $a[i]$ is the value stored at location i.
- If we abstract the location away, an array associates a value ($a[i]$) with each integer i in the range from 0 to `len`(a). For example,

  a = ['John', 'Mary', 'Mary', 'Susan']

  can be regarded as a set of associations:

  $$\{0 \mapsto \texttt{'John'}, 1 \mapsto \texttt{'Mary'}, 2 \mapsto \texttt{'Mary'}, 3 \mapsto \texttt{'Susan'}\}.$$

- The indices of a list are the keys, each of which is unique. If we generalize the integer keys to any type of data, we have an associative array. For example,

  $$\{\texttt{'North'} \mapsto \texttt{'John'}, \texttt{'South'} \mapsto \texttt{'Mary'}, \texttt{'West'} \mapsto \texttt{'Mary'}, \texttt{'East'} \mapsto \texttt{'Susan'}\}.$$

- An associative array is a map from keys to values, where each of the keys is unique. An associative array is also called a *map*, a *table*, or a *dictionary*.

Associative Array Operations

The operations on associative arrays varies from system to system, depending on applications. The following are a few common operations.

- *__getitem__*(*self*, *key*) — returns the value associated with the *key*. If the *key* is not in the associative array, it raises *KeyError*.

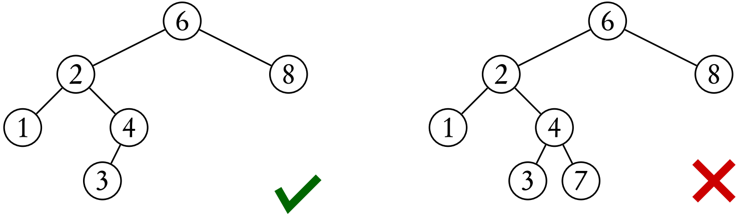
  For an associative array *m*, *__getitem__*(*self*, *key*) is called by `m[key]`.
- *__setitem__*(*self*, *key*, *value*) — inserts the *key* $\mapsto$ *value* association into the associative array if the *key* is new. If the *key* exists in the associative array, it associates the new *value* with the *key*.

  For an associative array *m*, *__setitem__*(*self*, *key*, *value*) is called by `m[key] = value`.
- *__delitem__*(*self*, *key*) — removes the *key* and its associated value from the the associative array. If the *key* is not in the associative array, it raises *KeyError*.

  For an associative array *m*, *__delitem__*(*self*, *key*) is called by `del m[key]`.
- *__iter__*(*self*) — iterates over all the keys in the associative array.

Binary Search Trees

- A binary tree can hold a collection of $\langle key, value \rangle$ pairs, each of them is stored in a node. Each *key* in the tree is *unique*, and it is also the *key* of the node.
- There must be an order $\_\_lt\_\_(self,\ other)$ ($<$) defined between the keys.
- Such a binary tree is a binary search tree if for every node in the tree,
  1. all the keys in its left subtree are less than the key of the node, and
  2. all the keys in its right subtree are greater than the key of the node.



Searching

- The main application of binary search trees is to search for a node with a given *key*.
- According to the properties of binary search trees, we can find a node very quickly:
  1. If the tree is empty, we return `None`, indicating that the key is not found.
  2. If the root node has the *key*, we simply return the root.
  3. If the *key* is less than the root key, the node to find must be in the left subtree. We can recursively search for it.
  4. If the *key* is greater than the root key, we recursively search for it in the right subtree.
- The searching advances a level in each step, so the number of steps cannot be more than the depth of the tree.

Finding the Node with a Given Key

The tree node is defined as usual, except that we split *itm* into *key* and *value*.

```
class Node:
    def __init__(self, key, value):
        self.key, self.value = key, value
        self.left = self.right = None
```

We search for a given key by the *find* find function.

```
def find(root, key):
    if root is None or key == root.key:
        return root
    elif key < root.key:
        return find(root.left, key)
    else:
        return find(root.right, key)
```

Finding the Node with a Given Key (2)

The tail-recursion on the previous slide can be transformed to a loop.

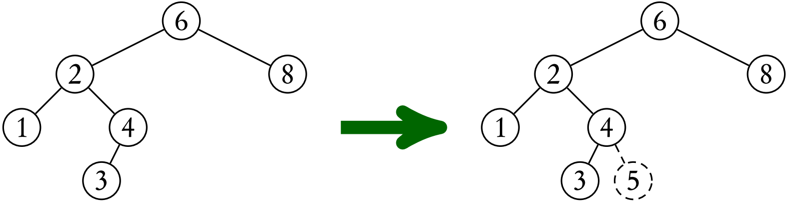
```
def find(root, key):
    while root is not None and key != root.key:
        root = root.left if key < root.key else root.right
    return root
```

The node with the least key is the leftmost node.

```
def find_min(root):
    if root is not None:
        while root.left:
            root = root.left
    return root
```

Inserting a Pair

- Insertion of a pair *key* and *value* can also be recursively performed.
  1. If the tree is empty, we make a tree of a single node with the *key* and the *value*.
  2. If the root has the *key*, we associate it with the new *value*.
  3. If the *key* is less than the root key, we insert the pair to the left subtree.
  4. If the *key* is greater than the root key, we insert the pair to the right subtree.
- The new node is always a leaf.

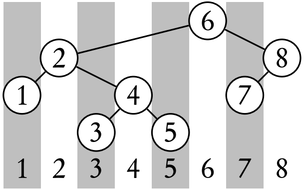


Inserting a Pair — Code

```
def insert(root, key, value):
    if root is None:
        return Node(key, value)
    else:
        if key == root.key:
            root.value = value
        elif key < root.key:
            root.left = insert(root.left, key, value)
        else:
            root.right = insert(root.right, key, value)
        return root
```

Traversals of Binary Search Trees

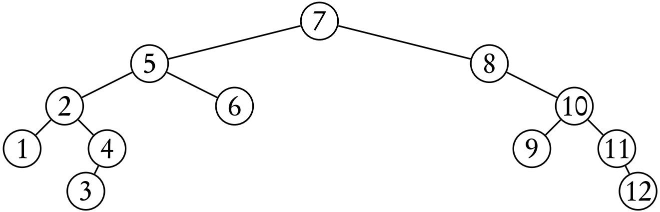
- For a binary tree, a parent node may be considered between its left and right children. Thus, we can define the *in-order* traversal as follows:
  1. recursively traverse the left subtree;
  2. visit the root node;
  3. recursively traverse the right subtree.



- The in-order traversal of a binary search tree results an ordered sequence.
- We can rebuild a binary search tree from the pre-order traversal sequence.
  - Just keep fetching items from the sequence and inserting them to a tree (initially empty), using the previously described insertion method.
- For a general binary tree, we can rebuild it from the sequences of pre-order and in-order traversals.

Deleting a Node by a Key

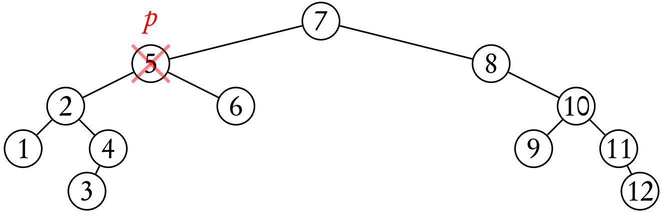
- Deletion must keep the ordering property of a binary search tree.



- We must move the nearest left or nearest right node in place of the deleted one.
    1. First, find the node p to delete.
    2. If one of p's subtrees is empty, move the other one in place of it.
    3. Find the right-most node q of its left subtree (or the other way).
    4. Since q has no right subtree, move q's left subtree in place of q.
    5. Move q in place of p.

Deleting a Node by a Key

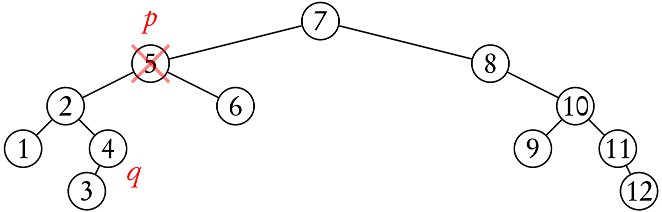
- Deletion must keep the ordering property of a binary search tree.



- We must move the nearest left or nearest right node in place of the deleted one.
  1. First, find the node p to delete.
  2. If one of p's subtrees is empty, move the other one in place of it.
  3. Find the right-most node q of its left subtree (or the other way).
  4. Since q has no right subtree, move q's left subtree in place of q.
  5. Move q in place of p.

Deleting a Node by a Key

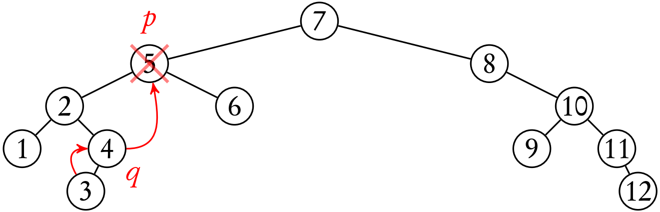
- Deletion must keep the ordering property of a binary search tree.



- We must move the nearest left or nearest right node in place of the deleted one.
  1. First, find the node p to delete.
  2. If one of p's subtrees is empty, move the other one in place of it.
  3. Find the right-most node q of its left subtree (or the other way).
  4. Since q has no right subtree, move q's left subtree in place of q.
  5. Move q in place of p.

Deleting a Node by a Key

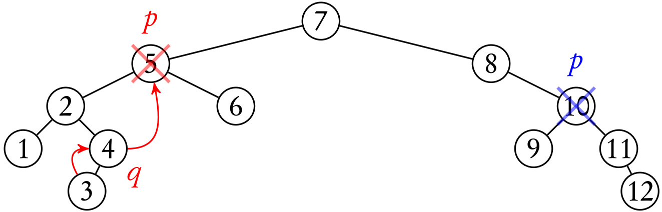
- Deletion must keep the ordering property of a binary search tree.



- We must move the nearest left or nearest right node in place of the deleted one.
  1. First, find the node p to delete.
  2. If one of p's subtrees is empty, move the other one in place of it.
  3. Find the right-most node q of its left subtree (or the other way).
  4. Since q has no right subtree, move q's left subtree in place of q.
  5. Move q in place of p.

Deleting a Node by a Key

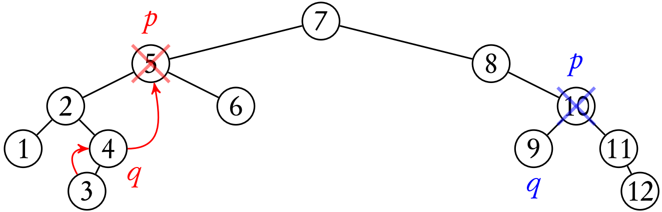
- Deletion must keep the ordering property of a binary search tree.



- We must move the nearest left or nearest right node in place of the deleted one.
    1. First, find the node p to delete.
    2. If one of p's subtrees is empty, move the other one in place of it.
    3. Find the right-most node q of its left subtree (or the other way).
    4. Since q has no right subtree, move q's left subtree in place of q.
    5. Move q in place of p.

Deleting a Node by a Key

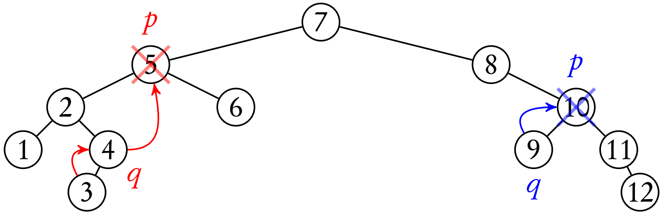
- Deletion must keep the ordering property of a binary search tree.



- We must move the nearest left or nearest right node in place of the deleted one.
  1. First, find the node p to delete.
  2. If one of p's subtrees is empty, move the other one in place of it.
  3. Find the right-most node q of its left subtree (or the other way).
  4. Since q has no right subtree, move q's left subtree in place of q.
  5. Move q in place of p.

Deleting a Node by a Key

- Deletion must keep the ordering property of a binary search tree.



- We must move the nearest left or nearest right node in place of the deleted one.
  1. First, find the node p to delete.
  2. If one of p's subtrees is empty, move the other one in place of it.
  3. Find the right-most node q of its left subtree (or the other way).
  4. Since q has no right subtree, move q's left subtree in place of q.
  5. Move q in place of p.

Deleting the Rightmost Node

- Similar to the deletion of the leftmost leaf in a perfectly balanced tree, except that the rightmost node of a search tree may have a nonempty left subtree.
- The recursion stops when the root is the rightmost node, the root will change to *root.left* after the deletion.
- The *delete_rightmost* function returns a pair — the new root and the deleted rightmost node.

```
def delete_rightmost(root):
    if not root.right:
        return (root.left, root)
    else:
        root.right, rightmost = delete_rightmost(root.right)
        return (root, rightmost)
```

Deleting a Key — Code

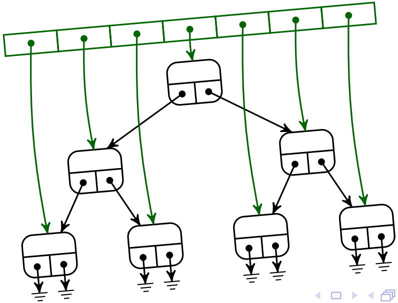
```
def delete(root, key):
    if root is None: return (root, None)
    elif key == root.key:
        if not root.left: return (root.right, root)
        elif not root.right: return (root.left, root)
        else:
            sub, rightmost = delete_rightmost(root.left)
            rightmost.left, rightmost.right = sub, root.right
            return (rightmost, root)
    elif key < root.key:
        root.left, deleted = delete(root.left, key)
        return (root, deleted)
    else:
        root.right, deleted = delete(root.right, key)
        return (root, deleted)
```

Implementing Associative Arrays Based-on Binary Search Trees

```python
class BSTAssocArray:
    def __init__(self):
        self.root = None
    def __getitem__(self, key):
        p = find(self.root, key)
        if p is None: raise KeyError
        return p.value
    def __setitem__(self, key, value):
        self.root = insert(self.root, key, value)
    def __delitem__(self, key):
        self.root, deleted = delete(self.root, key)
        if deleted is None: raise KeyError
    def __iter__(self):
        yield from (p.key for p in inorder_nodes(self.root))
```

Rebalancing

- The best binary search tree of n nodes has the minimum depth of $\log n$. This results the quickest searching.
- The worst has the maximum depth of $n-1$. In this case, the tree degenerates to a linked list.
- Since the in-order traversal results an ordered sequence, we may convert the tree to such a sequence and rebuild a balanced tree from it.
- To build a balanced tree from an ordered sequence:
  1. if the sequence is empty, return an empty tree; otherwise,
  2. take the middle item as the root, and
  3. recursively build the left and right subtrees from the front and rear halves of the sequence.



Rebalancing — Code

This method takes a list of nodes and connects the nodes ranging from index i to index $j-1$ into a balanced binary tree, whose in-order traversal sequence is the same as the list $a[i:j]$. The method returns the root node of the balanced tree.

```
def build_bal(a, i, j):
    if i < j:
        m = (i+j)//2
        root = a[m]
        root.left = build_bal(a, i, m)
        root.right = build_bal(a, m+1, j)
        return root
    else:
        return None
```

Rebalancing — Proof

We prove that when $i \leqslant j$, *build_bal*(a,i,j) returns a perfectly balanced tree Δ of size $j-i$, and with the in-order traversal sequence equals $a[i:j]$, denoted as $\Delta_{j-i}^{a[j:i]}$. We induct on $j-i$.

- Base case: when $j-i=0$, $a[i:j]=[]$ and *build_bal*(a,i,j) returns $\Delta_0^{[]}$.
- Induction step: when $j-i>0$, we have $j>i$. Thus, $m=\lfloor\frac{i+j}{2}\rfloor$, we have 1) $i+j=2m$, or 2) $i+j=2m+1$. In case 1) we have

$$m-i=\frac{2m-2i}{2}=\frac{i+j-2i}{2}=\frac{j-i}{2}\quad\text{and}\quad j-(m+1)=\frac{2j-2m}{2}-1=\frac{2j-i-j}{2}-1=\frac{j-i}{2}-1,$$

and in case 2) we have

$$m-i=\frac{2m-2i}{2}=\frac{i+j-1-2i}{2}=\frac{j-i-1}{2}\quad\text{and}\quad j-(m+1)=\frac{2j-2m-2}{2}=\frac{2j-i-j+1-2}{2}=\frac{j-i-1}{2}.$$

In either case, $0\leqslant$ the sizes $<j-i$ and the size difference is no more than 1. By induction hypothesis, *build_bal*(a,i,m) returns $\Delta_{m-i}^{a[i:m]}$, and *build_bal*$(a,m+1,j)$ returns $\Delta_{j-(m+1)}^{a[m+1:j]}$. Therefore, *build_bal*$(a,i,j)$ returns $\Delta_{m-i+1+j-(m+1)}^{a[i:m]+a[m]+a[m+1:j]}=\Delta_{j-i}^{a[j:i]}$.