Containers: Set and Dictionary

Jordi Cortadella and Jordi Petit
Department of Computer Science

Sets and Dictionaries

• A set: a collection of items. The typical operations are:
  – Add/remove one element
  – Does it contain an element?
  – Size?, Is it empty?
  – Visit all items

• A dictionary (map): a collection of key-value pairs. The typical operations are:
  – Put a new key-value pair
  – Remove a key-value pair with a specific key
  – Get the value associated to a key
  – Does it contain a key?
  – Visit all key-value pairs

Possible implementations of a set

<table>
<thead>
<tr>
<th></th>
<th>Unsorted list or vector</th>
<th>Sorted vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>O(n), if checking for duplicate keys, O(1) otherwise.</td>
<td>O(n) in the worst case (similar to insertion sort)</td>
</tr>
<tr>
<td>Deletion</td>
<td>O(n) since it has to find the item along the list.</td>
<td>O(n) since it has to sift the elements after deletion.</td>
</tr>
<tr>
<td>Lookup</td>
<td>O(n) since the list must be scanned.</td>
<td>O(log n) with binary search.</td>
</tr>
<tr>
<td>Good for</td>
<td>Small sets.</td>
<td>Read-only collections (only lookups) or very few updates.</td>
</tr>
</tbody>
</table>

Note: n is the number of items in the set.

Source: Natural Language Processing with Python, by Steven Bird, Ewan Klein and Edward Loper
Binary Search Trees

**BST property**: for every node in the tree with value V:
- All values in the left subtree are smaller than V.
- All values in the right subtree are larger than V.

This is a binary search tree

This is **not** a binary search tree

**BST: public methods**

```cpp
template<typename T>
class Set {
public:
    // Constructors, assignment and destructor
    Set();
    Set(const Set& S);
    Set& operator=(const Set& S);
    ~Set();

    // Finding elements
    const T& findMin() const;
    const T& findMax() const;
    bool contains(const T& x) const;
    int size() const;
    bool isEmpty() const;

    // Insert/remove methods
    void insert(const T& x);
    void remove(const T& x);

private:
    Node* root; // Pointer to the root of the tree
    int n; // Number of elements
```

**BST: private methods**

```cpp
private:
    // Public methods require a private pointer-based
    // version to traverse the tree.
    // Finding elements
    Node* findMin(Node* t) const;
    Node* findMax(Node* t) const;
    bool contains(const T& x, Node* t) const;

    // Insert/remove methods
    void insert(const T& x, Node*& t);
    void remove(const T& x, Node*& t);
    void makeEmpty(Node*& t);
```
/** Find the smallest item in the (non-empty) subtree t.
* Returns a ptr to the node with the smallest item. */
Node* findMin(Node* t) const {
    if (t->left == nullptr) return t;
    return findMin(t->left);
}

/** Find the largest item in the (non-empty) subtree t.
* Returns a ptr to the node with the largest item. */
Node* findMax(Node* t) const {
    while (t->right != nullptr) t = t->right;
    return t;
}

/** Find the smallest item in the (non-empty) Set. */
const T& findMin() const {
    assert(not isEmpty());
    return findMin(root)->elem;
}

/** Inserts item x into the subtree t.
* It may modify the value of t. */
void insert(const T& x, Node*& t) {
    if (t == nullptr) {
        t = new Node {x, nullptr, nullptr};
        ++n;
    } else if (x < t->elem) insert(x, t->left);
    else if (x > t->elem) insert(x, t->right);
    // else: duplicated item, do nothing
}

/** Inserts item x into the set. */
void insert(const T& x) {
    insert(x, root);
}

/** Find an item in the subtree represented by t.
* Returns true if found, and false otherwise. */
bool contains(const T& x, Node* t) const {
    if (t == nullptr) return false;
    if (x < t->elem) return contains(x, t->left);
    if (x > t->elem) return contains(x, t->right);
    return true;
}

/** Checks whether the tree is empty. */
bool isEmpty() const {
    return size() == 0; // or also root == nullptr
}

/** Find an item in the set. */
bool contains(const T& x) const {
    return contains(x, root);
}

/** Inserts item x into the set. */
void insert(const T& x) {
    insert(x, root);
}

/** Removes item x from the subtree t. */
void remove(const T& x) {
    remove(x, root);
}

/** Removes item x from the set. */
void remove(const T& x) {
    remove(x, root);
}

/** Removes item x from the subtree t. */
void remove(Node* t) {
    if (t == nullptr) return;
    if (t->left != nullptr) remove(t->left);
    if (t->right != nullptr) remove(t->right);
    if (t->left == nullptr && t->right == nullptr) {
        delete t;
        --n;
    }
}

/** Removes item x from the set. */
void remove(Node* t) {
    if (t == nullptr) return;
    if (t->left != nullptr) remove(t->left);
    if (t->right != nullptr) remove(t->right);
    if (t->left == nullptr && t->right == nullptr) {
        delete t;
        --n;
    }
}

/** Removes item x from the subtree t. */
void remove(const T& x) {
    remove(x, root);
}

/** Removes item x from the set. */
void remove(const T& x) {
    remove(x, root);
}

/** Finds the smallest item in the (non-empty) subtree t. */
const T& findMin(Node* t) const {
    if (t->left == nullptr) return t->elem;
    return findMin(t->left)->elem;
}

/** Finds the largest item in the (non-empty) subtree t. */
const T& findMax(Node* t) const {
    while (t->right != nullptr) t = t->right;
    return t->elem;
}

/** Finds the smallest item in the (non-empty) Set. */
const T& findMin() const {
    assert(not isEmpty());
    return findMin(root)->elem;
}

/** Finds the largest item in the (non-empty) Set. */
const T& findMax() const {
    assert(not isEmpty());
    return findMax(root)->elem;
}

/** Finds the smallest item in the (non-empty) subtree t. */
const T& findMin(Node* t) const {
    if (t->left == nullptr) return t->elem;
    return findMin(t->left)->elem;
}

/** Finds the largest item in the (non-empty) subtree t. */
const T& findMax(Node* t) const {
    while (t->right != nullptr) t = t->right;
    return t->elem;
}

/** Finds the smallest item in the (non-empty) Set. */
const T& findMin() const {
    assert(not isEmpty());
    return findMin(root)->elem;
}

/** Finds the largest item in the (non-empty) Set. */
const T& findMax() const {
    assert(not isEmpty());
    return findMax(root)->elem;
}
/** Removes item x from the subtree t. */
void remove(const T& x, Node*& t) {
    if (t == nullptr) return; // Not found
    if (x < t->elem) return remove(x, t->left);
    if (x > t->elem) return remove(x, t->right);

    // We have found the item
    if (t->left == nullptr || t->right == nullptr) {
        Node* old = t;
        t = t->left ? t->left : t->right;
        delete t;
        --n;
    } else {
        ...
        ...
        // Case with two children
        ...
    }
}
remove: complex case (two children)

1. Find the element.
2. Find the min value of the left subtree.
3. Copy the min value onto the element to be removed.
4. Remove the min value in the left subtree (simple case).

remove(2)

Constructors and destructor

/** Default constructor (empty set). */
Set() : root(nullptr), n(0) {}
/** Copy constructor. */
Set(const Set& S) {
    root = copy(S.root);
    n = S.n;
}
/** Assignment operator. */
Set& operator=(const Set& S) {
    if (&S != this) {
        makeEmpty(root);
        root = copy(S.root);
        n = S.n;
    }
    return *this;
}
/** Destructor. */
~Set() {
    makeEmpty(root);
}

remove: all cases

/** Removes item x from the subtree t. */
void remove(const T& x, Node*& t) {
    if (t == nullptr) return; // Not found
    if (x < t->elem) return remove(x, t->left);
    if (x > t->elem) return remove(x, t->right);

    // We have found the item
    if (t->left == nullptr || t->right == nullptr) {
        Node* old = t;
        t = t->left ? t->left : t->right;
        delete t;
        --n;
    } else { // Case with two children (simple version with copy)
        // A version manipulating only pointers is also possible
        t->elem = findMin(t->right)->elem; // Copy the min element
        remove(t->elem, t->right); // Remove the min elem
    }
}
/** Public method for remove. */
void remove(const T& x) {
    remove(x, root);
}

/** Recursive method to clone a subtree. */
Node* copy(Node* t) const {
    if (t == nullptr) return nullptr;
    return new Node{t->elem, copy(t->left), copy(t->right)};
}
/** Recursive method to clean a subtree. */
void makeEmpty(Node*& t) {
    if (t != nullptr) {
        makeEmpty(t->left);
        makeEmpty(t->right);
        delete t;
    }
    t = nullptr;
}

/** Recursive method to clone a subtree. */
Node* copy(Node* t) const {
    if (t == nullptr) return nullptr;
    return new Node{t->elem, copy(t->left), copy(t->right)};
}
/** Recursive method to clean a subtree. */
void makeEmpty(Node*& t) {
    if (t != nullptr) {
        makeEmpty(t->left);
        makeEmpty(t->right);
        delete t;
    }
    t = nullptr;
Visiting the items in ascending order

Question:
How can we visit the items of a BST in ascending order?

Answer:
Using an in-order traversal

BST: runtime analysis

• Let us assume that the set has \( n \) elements. The operations copy and makeEmpty take \( O(n) \).

• We are mostly interested in the runtime of the insert/remove/contains methods.
  – The complexity is \( O(d) \), where \( d \) is the depth of the node containing the required element.

• But, how large is \( d \)?

Random BST

Source: Fig 4.29 of Weiss textbook

BST: runtime analysis

• Internal path length (IPL): The sum of the depths of all nodes in a tree. Let us calculate the average IPL considering all possible insertion sequences.

• \( D(n) \) is the IPL of a tree with \( n \) nodes. \( D(1) = 0 \). The left subtree has \( i \) nodes and the right subtree has \( n - i - 1 \) nodes. Thus,

\[
D(n) = D(i) + D(n - i + 1) + (n - 1)
\]

• If all subtree sizes are equally likely, then the average value for \( D(i) \) and \( D(n - i - 1) \) is

\[
\frac{1}{n} \sum_{j=0}^{n-1} D(j)
\]
BST: runtime analysis

Therefore,

\[ D(n) = \frac{2}{n} \left[ \sum_{j=0}^{n-1} D(j) \right] + n - 1 \]

The previous recurrence gives:

\[ D(n) = O(n \log n) \]

The average height of nodes after \( n \) random insertions is \( O(\log n) \).

However, the \( O(\log n) \) average height is not preserved when doing deletions.

Worst-case runtime: \( O(n) \)

Balanced trees

The worst-case complexity for insert, remove and search operations in a BST is \( O(n) \), where \( n \) is the number of elements.

Various representations have been proposed to keep the height of the tree as \( O(\log n) \):

- AVL trees
- Red-Black trees
- Splay trees
- B-trees
AVL trees

• Named after Adelson-Velsky and Landis (1962).

• Main idea: invest some additional time to balance the tree each time a new element is inserted or deleted.

• Properties:
  – The height of the tree is always $\Theta(\log n)$.
  – The time devoted to balancing is $O(\log n)$.

AVL tree: definition

• An AVL tree is a BST such that, for every node, the difference between the heights of the left and right subtrees is at most 1.

AVL tree in action

https://en.wikipedia.org/wiki/AVL_tree

Smallest AVL tree with $h = 9$. 
AVL trees

Smallest AVL tree with \( h = 6 \).

The important question: what is the size of an AVL tree with height \( h \)?

The height is \( \Omega(\log n) \)

- The size \( n \) of a tree with height \( h \) is:
  \[
  n \leq 1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1.
  \]
  (all levels full of nodes)

- Therefore,
  \[
  \log_2 (n + 1) - 1 \leq h
  \]
  and \( h = \Omega(\log n) \).

The height is \( O(\log n) \)

- Let \( S(h) \) be the min number of nodes of an AVL tree with height \( h \).
- One of the children (e.g., left) must have height \( h - 1 \). The other child must have height \( h - 2 \) (because the AVL has min size).
- Therefore,
  \[
  S(h) = S(h - 1) + S(h - 2) + 1.
  \]
- Thus,
  \[
  S(h) \geq 2 \cdot S(h - 2).
  \]
- Given that \( S(0) = 1 \) and \( S(1) = 2 \), it can be easily proven, by induction, that:
  \[
  S(h) \geq 2^{h/2}
  \]
- Since \( n \geq S(h) \) and \( \log_2 S(h) \geq h/2 \), then \( h \leq 2 \log_2 n \):
  \[
  h = O(\log n).
  \]
• The recurrence

\[ S(h) = S(h-1) + S(h-2) + 1 \]

resembles the one of the Fibonacci numbers. A tighter bound can be obtained.

• Theorem: the height of an AVL tree with \( n \) internal nodes satisfies:

\[ h < 1.44 \log_2 (n + 2) - 1.328 \]

Any newly inserted item may fall into any of the four subtrees (LL, LR, RL or RR). A new insertion may violate the balancing property. Re-balancing might be required.

Single rotation: the left-left case

Single rotation: the right-right case
**Double rotation: the left-right case**

- **Insertion**

**Double rotation: the right-left case**

- **Insertion**

**Implementation details**

- The height must be stored at each node. Only the unbalancing factor (\{-1,0,1\}) is strictly required.

- The insertion/deletion operations are implemented similarly as in BSTs (recursively).

- The re-balancing of the tree is done when the recursive calls return to the ancestors (check heights and rotate if necessary).

**Complexity**

- Single and double rotations only need the manipulation of few pointers and the height of the nodes (\(O(1)\)).

- Insertion: the height of the subtree after a rotation is the same as the height before the insertion. Therefore, at most only one rotation must be applied for each insertion.

- Deletion: more complicated. More than one rotation might be required.

- Worst case for deletion: \(O(\log n)\) rotations (a chain effect from leaves to root).