Dictionary ADT

Binary Search Trees

Lecture 20

Dictionary ADT

- A dictionary (a.k.a. map) stores a set of pairs (key, value)
  - (word, definition)
  - (studentID, studentRecord)
  - (flightNumber, flightInformation)

- Data is accessed only through key:
  - Object find(key k)
  - void insert(key k, Object v)
  - Object remove(key k)

- If the keys can be ordered
  - Object previous(key k)
  - Object next(key k)

Dictionary vehicle = {
  'car':'a road vehicle, typically with four wheels, powered by an internal combustion engine and able to carry a small number of people.';
  'bicyle':' a vehicle composed of two wheels held in a frame one behind the other, propelled by pedals and steered with handlebars attached to the front wheel.'
}

Array implementation

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key 1</td>
<td>Content 1</td>
</tr>
<tr>
<td>Key 2</td>
<td>Content 2</td>
</tr>
<tr>
<td>Key 3</td>
<td>Content 3</td>
</tr>
<tr>
<td>Key 4</td>
<td>Content 4</td>
</tr>
<tr>
<td>Ø</td>
<td>Ø</td>
</tr>
</tbody>
</table>

Size = 4

Array implementation

Array of pairs (key, value)
- find(key k) : scan array to find key O(n)
- insert(key k, Object v):
  - Add the pair (k, v) at the end of the array O(1)
  - Increase size by one
- remove(key k)
  - Scan array to find k O(n)
  - Shift left remaining elements
Sorted array implementation

Array of pairs (key, value), sorted by key

- **find**(key k) : binary search to find key
  - Binary search to find where to insert, O(log n)
  - Shift element right to insert new element, O(n)
- **insert**(key k, Object v):
  - Binary search to find key, O(log n)
  - Shift left remaining elements, O(n)
- **remove**(key k)
  - Binary search to find key, O(log n)
  - Shift left remaining elements, O(n)

Array implementation

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>O(log n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Linked-list implementation

Linked-list where each node contain a pair (key, value)

- **find**(key k) : scan list to find key
  - O(n)
- **insert**(key k, Object v):
  - Add the pair (k, v) at the end of the list
  - O(1)
- **remove**(key k)
  - Scan list to find k, O(n)
  - Remove node, O(1)

- Note: Keeping the linked-list sorted does not help, as binary search can't be done in time O(log n) in linked lists. (why?)

Linked-list implementation

<table>
<thead>
<tr>
<th>Key1</th>
<th>Value1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key2</td>
<td>Value2</td>
</tr>
<tr>
<td>Key3</td>
<td>Value3</td>
</tr>
</tbody>
</table>

Implementations of dictionary

<table>
<thead>
<tr>
<th>Method</th>
<th>find</th>
<th>insert</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Linked-list</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted array</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

BST - Definition

- A binary search tree is a binary tree such that for any node n,
  - The elements of the left subtree of n have values smaller or equal to n
  - The elements of the right subtree of n have values larger of equal to n
- In the figure, we show only the keys
**BST - Find**

Idea: 1) Start from the root of the tree
2) Choose if you should go to the left or right child.
3) Repeat until you find the key sought or get to a leaf.

Algorithm: find(node n, key k)
Input: The node n at the root of the tree to explore.
The key k to find
Output: Returns one node with key equal to k
if (n = null) then return null
if (n.key = k) then return n
if (n.key > k) then return find(n.leftChild, k)
if (n.key < k) then return find(n.rightChild, k)

* Can you write a non-recursive version of this algorithm?

**BST - insert**

Idea: 1) Find the leaf where the insertion will take place, by going down the tree as for the "find" algo.
2) Add a new left or right child to that leaf

Algorithm: insert(node n, key k, object v)
Input: The key k and information i to be added to the subtree rooted at n. Assumes n!= null
Output: Inserts a new node (k,i) in the subtree rooted at n
if (k ≤ n.key) then
  if (n.leftChild != null) then
    insert(n.leftChild, k, v)
  else
    n.setLeftChild(new node(k,v));
else
  if (n.rightChild != null) then
    insert(n.rightChild, k, v)
  else
    n.setRightChild(new node(k,v));

**BST - remove**

Idea: 1) Find the node N to be removed using the “find” algo
2) - If N is a leaf, simply remove it
       - If N is an internal node with only one child, replace N by its child
       - If N is an internal node with two children, N will be replaced by the node N’ that has the next key largest key after N.
   
To find N’:
   i) Follow the right child of N and then go down left children until no left child is found.
   The node found is N’
   Overwrite N by N’.

// A small utility function
Algorithm: replace(node x, node y)
Input: Two nodes x and y
Output: Copies node y onto node x, overwriting x.
if (x.parent != null) then
  if (x.parent.leftChild = x) then
    x.parent.setLeftChild(y)
  else
    x.parent.setRightChild(y)
if (y != null) then y.parent ← x.parent

Algorithm: remove(node root, key k)
Input: The key k of the node to be removed from the subtree rooted at n
Output: Removes node with key k and returns it.
node x ← find(root, k)
if (x=null) then return null // key k was not found
if (x.isALeaf()) then replace(x, null), return
if (x.leftChild = null or x.rightChild = null) then // x has only one child
  if (x.leftChild = null) then replace(x, x.rightChild) // x was right child
  else if (x.rightChild = null) then replace(x, x.leftChild) // x was left child
else // x has two children
  (see next page)
// x has two children. First find successor of x
suc ← x.rightChild
while (suc.leftChild != null) do suc ← suc.leftChild
x.value = suc.value
x.key = suc.key
replace(suc, suc.rightChild)