SEARCHING

- the dictionary ADT
- binary search
- binary search trees

The Dictionary ADT

- a dictionary is an abstract model of a database
- like a priority queue, a dictionary stores key-element pairs
- the main operation supported by a dictionary is searching by key
- simple container methods:
  - size()
  - isEmpty()
  - elements()
- query methods:
  - findElement(k)
  - findAllElements(k)
- update methods:
  - insertItem(k, e)
  - removeElement(k)
  - removeAllElements(k)
- special element
  - NO_SUCH_KEY, returned by an unsuccessful search

Implementing a Dictionary with a Sequence

- unordered sequence
  - searching and removing takes O(n) time
  - inserting takes O(1) time
  - applications to log files (frequent insertions, rare searches and removals)
- array-based ordered sequence (assumes keys can be ordered)
  - searching takes O(log n) time (binary search)
  - inserting and removing takes O(n) time
  - application to look-up tables (frequent searches, rare insertions and removals)

Binary Search

- narrow down the search range in stages
- “high-low” game
- findElement(22)
Pseudocode for Binary Search

Algorithm BinarySearch(S, k, low, high)
if low > high then
    return NO_SUCH_KEY
else
    mid ← (low+high) / 2
    if k = key(mid) then
        return key(mid)
    else if k < key(mid) then
        return BinarySearch(S, k, low, mid−1)
    else
        return BinarySearch(S, k, mid+1, high)

Running Time of Binary Search

• The range of candidate items to be searched is \textit{halved after each comparison}

<table>
<thead>
<tr>
<th>comparison</th>
<th>search range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(n)</td>
</tr>
<tr>
<td>1</td>
<td>(n/2)</td>
</tr>
<tr>
<td>2</td>
<td>(n/4)</td>
</tr>
<tr>
<td>(2^i)</td>
<td>(n/2^i)</td>
</tr>
<tr>
<td>(\log_2 n)</td>
<td>1</td>
</tr>
</tbody>
</table>

• In the array-based implementation, access by rank takes O(1) time, thus binary search runs in O(\(\log n\)) time

Binary Search Trees

• A binary search tree is a binary tree \(T\) such that
  - each internal node stores an item \((k, e)\) of a dictionary.
  - keys stored at nodes in the left subtree of \(v\) are less than or equal to \(k\).
  - keys stored at nodes in the right subtree of \(v\) are greater than or equal to \(k\).
  - external nodes do not hold elements but serve as place holders.

Search

• A binary search tree \(T\) is a \textit{decision tree}, where the question asked at an internal node \(v\) is whether the search key \(k\) is less than, equal to, or greater than the key stored at \(v\).

• Pseudocode:

  Algorithm TreeSearch(k, v):
  \textbf{Input:} A search key \(k\) and a node \(v\) of a binary search tree \(T\).
  \textbf{Output:} A node \(w\) of the subtree \(T(v)\) of \(T\) rooted at \(v\), such that either \(w\) is an internal node storing key \(k\) or \(w\) is the external node encountered in the inorder traversal of \(T(v)\) after all the internal nodes with keys smaller than \(k\) and before all the internal nodes with keys greater than \(k\).

  if \(v\) is an external node then
    return \(v\)
  if \(k = \text{key}(v)\) then
    return \(v\)
  else if \(k < \text{key}(v)\) then
    return TreeSearch(k, \(T\).leftChild(v))
  else
    \{ \(k > \text{key}(v)\) \}
    return TreeSearch(k, \(T\).rightChild(v))
Search Example I

- Successful `findElement(76)`
- A successful search traverses a path starting at the root and ending at an internal node
- How about `findAllelements(k)`?

Search Example II

- Unsuccessful `findElement(25)`
- An unsuccessful search traverses a path starting at the root and ending at an external node

Insertion

- To perform `insertItem(k, e)`, let `w` be the node returned by `TreeSearch(k, T.root())`
- If `w` is external, we know that `k` is not stored in `T`. We call `expandExternal(w)` on `T` and store `(k, e)` in `w`

Insertion II

- If `w` is internal, we know another item with key `k` is stored at `w`. We call the algorithm recursively starting at `T.rightChild(w)` or `T.leftChild(w)`
Removal I
- We locate the node $w$ where the key is stored with algorithm TreeSearch
- If $w$ has an external child $z$, we remove $w$ and $z$ with removeAboveExternal($z$)

```
44
/   \
/     \
17     88
/   \
/     \
28     65
/   \
/     \
29     54
```

removeElement(32)

```
44
/   \
/     \
17     88
/   \
/     \
28     65
/   \
/     \
29     54
```

Removal II
- If $w$ has no external children:
  - find the internal node $y$ following $w$ in inorder
  - move the item at $y$ into $w$
  - perform removeAboveExternal($x$), where $x$ is the left child of $y$ (guaranteed to be external)

```
54
/   \
/     \
52     88
/   \
/     \
17     65
/   \
/     \
29
```

```
54
/   \
/     \
52     88
/   \
/     \
17     65
/   \
/     \
29
```

Time Complexity
- A search, insertion, or removal, visits the nodes along a root-to-leaf path, plus possibly the siblings of such nodes
- Time $O(1)$ is spent at each node
- The running time of each operation is $O(h)$, where $h$ is the height of the tree
- The height of binary search tree is in $n$ in the worst case, where a binary search tree looks like a sorted sequence

```
10
/   \
/     \
20     50
/   \
/     \
40
```

- To achieve good running time, we need to keep the tree balanced, i.e., with $O(\log n)$ height
- Various balancing schemes will be explored in the next lectures