VE281

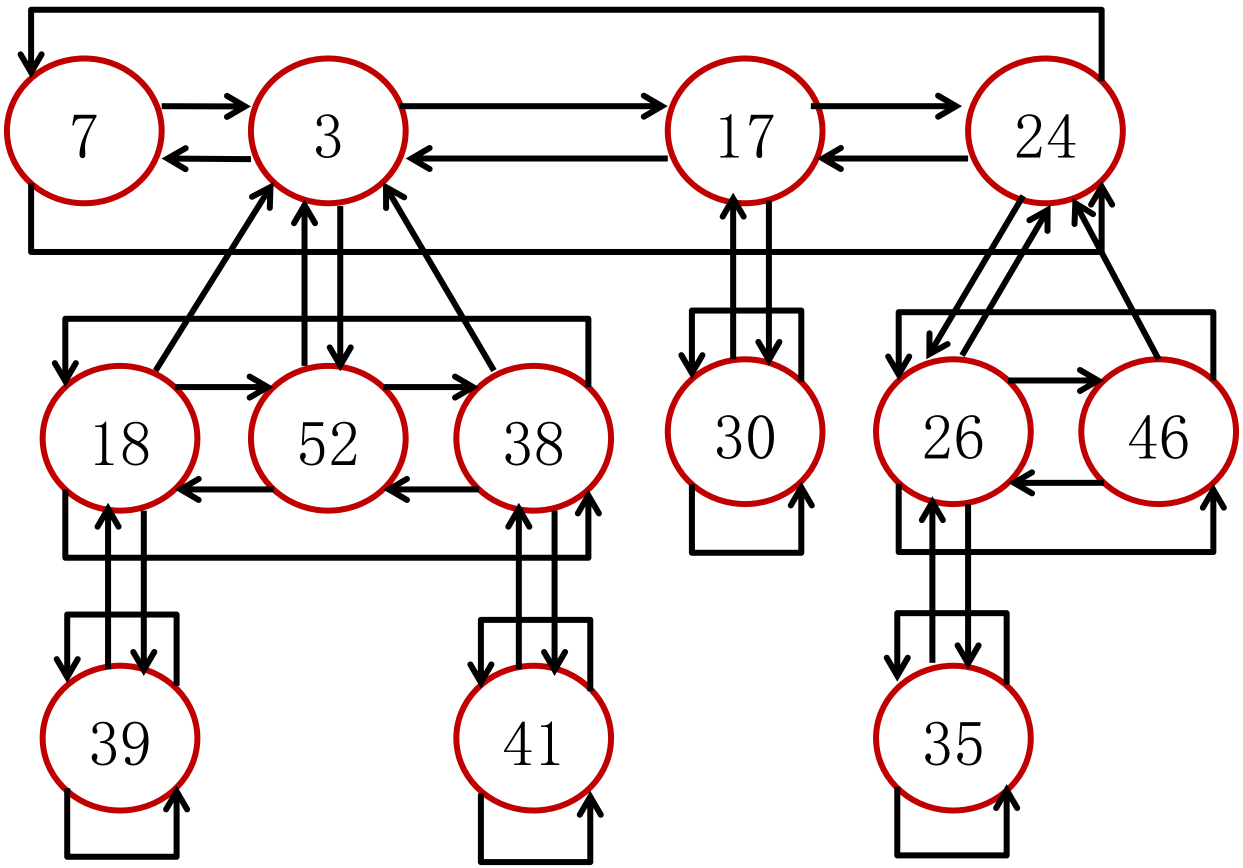
Data Structures and Algorithms

Recitation Class

Nov. 12 2018

VE281 TA Group

Fibonacci Heap



Fibonacci Heap

Operation	Binary Heap (worst case)	Fibonacci Heap (amortized analysis)
insert	$\Theta(\log n)$	$\Theta(1)$
extractMin	$\Theta(\log n)$	$O(\log n)$
getMin	$\Theta(1)$	$\Theta(1)$
makeHeap	$\Theta(1)$	$\Theta(1)$
union	$\Theta(n)$	$\Theta(1)$
decreaseKey	$\Theta(\log n)$	$\Theta(1)$

Fibonacci Heap

- `Insert:` Put into the root list
- `getMin:` Return H.min
- `makeHeap:` H.min = NULL and H.n = 0
- `extractMin:` Remove min and concatenate its children into root list
- `Union:` Connect the two root lists and determine the minimum
- `decreaseKey`:
  - min heap property violated
    - Cut between the node and its parent.
    - Move the subtree to the root list.
    - if a node n not in the root list has lost a child for the second time, cut again

Binary Search Tree

- Each node is associated with a **key**.
- The key of **any** node is greater than the keys of all nodes in its left subtree and smaller than the keys of all nodes in its right tree.

- Search, Insert, Remove by key $O(\log n)$

Binary Search Tree

- Search

```
node *search(node *root, Key k) {
  if(root == NULL) return NULL;
  if(k == root->item.key) return root;
  if(k < root->item.key)
    return search(root->left, k);
  else return search(root->right, k);
}
```

- Insert

```
void insert(node *&root, Item item) {
  if(root == NULL) {
    root = new node(item);
    return;
  }
  if(item.key < root->item.key)
    insert(root->left, item);
  else if(item.key > root->item.key)
    insert(root->right, item);
}
```

Binary Search Tree

Remove:

- Remove a leaf:

```
delete root;
root = NULL;
```

- Remove a degree-one node
  - With left child
  - With right child

```
node *tmp = root;
root = root->left;
delete tmp;
```

- Remove a degree-two node

```
node *&replace = findMax(root->left);
root->item = replace->item;
node *tmp = replace;
replace = replace->left;
delete tmp;
```

reference to pointer

```
node *&findMax(node *&root) {
   if(root->right == NULL) return root;
   return findMax(root->right);
}
```

Average-Case Time Complexity of BST

- If the successful search reaches a node at level d, the number of nodes visited is $d+1$.
- depth of the i-th node d_i
- Internal path length $\sum_{i=1}^{n} d_i$
- Average internal path length of a tree containing n nodes $I(n)$
  - $I(1) = 0$.
- For a tree of n nodes, suppose it has l nodes in its left subtree.
  - The number of nodes in its right subtree is $n-1-l$.
  - The average internal path length for such a tree is

$$T(n;l) = I(l) + I(n-1-l) + n - 1$$

- $I(n)$ is average of $T(n;l)$ over $l = 0,1,\ldots,n-1$.

Average-Case Time Complexity of BST

$$I(n) = \frac{1}{n}\sum_{l=0}^{n-1} T(n;l) \;= \frac{1}{n}\sum_{l=0}^{n-1}[I(l) + I(n-1-l) + n - 1]$$

$$\boxed{= \frac{2}{n}\sum_{l=0}^{n-1} I(l) + (n-1)}$$

$$I(n-1) = \frac{2}{n-1}\sum_{l=0}^{n-2} I(l) + (n-2)$$

$$\boxed{\sum_{l=0}^{n-2} I(l) = \frac{(n-1)[I(n-1) - (n-2)]}{2}}$$

$$\boxed{I(n) = \frac{n+1}{n} I(n-1) + \frac{2(n-1)}{n}}$$

$$\frac{I(n)}{n+1} = \frac{I(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

$$\le \frac{I(n-1)}{n} + \frac{2}{n}$$

$$\le 2\sum_{k=2}^{n}\frac{1}{k} < 2\ln n$$

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

Average-Case Time Complexity of BST

Thus, the average complexity for a search is

$$\Theta\left(\frac{1}{n}I(n)\right) = O(\log n)$$

	Search	Insert	Remove
Linked List	$O(n)$	$O(n)$	$O(n)$
Sorted Array	$O(\log n)$	$O(n)$	$O(n)$
Hash Table	$O(1)$	$O(1)$	$O(1)$
BST	$O(\log n)$	$O(\log n)$	$O(\log n)$

BST Additional Operations

- Other Operations Supported by BST
  - Output in Sorted Order $O(n)$
  - Get Min/Max $O(\log n)$
  - Get Predecessor/Successor $O(\log n)$
  - Rank Search $O(\log n)$
  - Range Search $O(n)$

Note: Hash table does not support efficient implementation of the above methods.