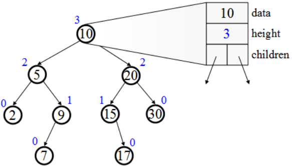
An AVL Tree



CSE 326 Data Structures

CSE 326 : Dave Bacon

Priority Queues : Binomial Queues, Binary Search Trees, AVL Trees

Logistics

- Updated Due Dates
  - Project 2, Phase A, due Friday, January 26
  - Homework 3, due Monday, January 29 in class
- Project 2A
  - Generic Arrays in Java

Amortized Analysis

- Grow an Array when full…
  - Option 1: increase array by constant c
  - Option 2: increase array by doubling size

- Analyze n push operations…Amortized time?

Option 1: $k=n/c$ replaces

$O(n+c+2c+3c+\ldots+kc)=O(n+ck(k-1)/2)=O(n^2)$

Option 2: $k=\log n$ replaces

$O(n+c+2c+4c+\ldots+2^k c)=O(n+c(2^{k+1}-1))=O(n)$

Merging Two Binomial Queues

Essentially like adding two binary numbers!

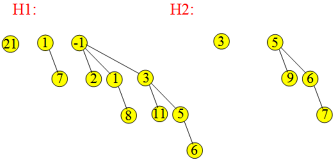
1. Combine the two forests
2. For k from 0 to maxheight {
   a. $m \leftarrow$ total number of B_k's in the two BQs
   b. if m=0: continue;
   c. if m=1: continue;
   d. if m=2: combine the two B_k's to form a B_{k+1}
   e. if m=3: retain one B_k and combine the other two to form a B_{k+1}

}

# of 1's
0+0 = 0
1+0 = 1
1+1 = 0+c
1+1+c = 1+c

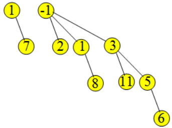
Claim: When this process ends, the forest has at most one tree of any height

Example: Binomial Queue Merge

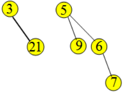


Example: Binomial Queue Merge

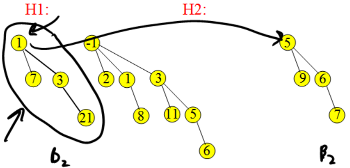
H1:



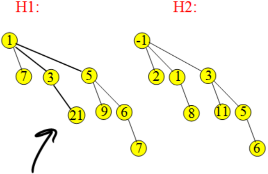
H2:



Example: Binomial Queue Merge



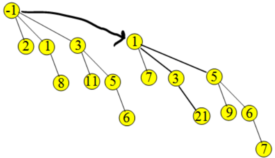
Example: Binomial Queue Merge



Example: Binomial Queue Merge

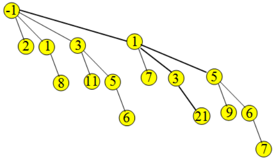
H1:

H2:



Example: Binomial Queue Merge

H1: H2:



Complexity of Merge

Constant time for each height

Max height is: log *n*

$\Rightarrow$ worst case running time = $\Theta($ log n $)$

Insert in a Binomial Queue

Insert(x): Similar to leftist or skew heap

merge

runtime

Worst case complexity: same as merge

O($\log n$)

Average case complexity: O(1)

Why?? *Hint: Think of adding 1 to 1101*

$$
\begin{array}{r}
10111011 \\
+ \qquad\quad 1 \\
\hline
10111100
\end{array}
$$

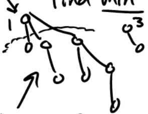
111111

most not

deleteMin in Binomial Queue

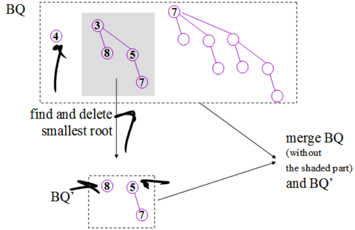
Similar to leftist and skew heaps….

find min   log n



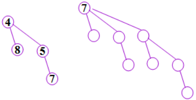
binomial forest

deleteMin: Example



deleteMin: Example

Result:

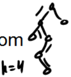


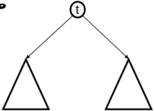
runtime: O(log n)

Bye Bye
Chapter 6.

Tree Calculations

Recall: height is max number of edges from root to a leaf





Find the height of the tree...

root left right

height(t) = 1 + max{height(t.left), height(t.right)}

O(n)

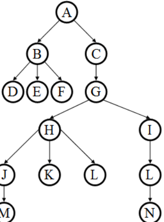
runtime:

n nodes $\underline{O(n)}$

n

Tree Calculations Example

How high is this tree?



More Recursive Tree Calculations: Tree Traversals

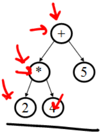
A *traversal* is an order for visiting all the nodes of a tree

Three types:

- Pre-order: Root, left subtree, right subtree
- In-order: Left subtree, root, right subtree

  2*4 + 5

- Post-order: Left subtree, right subtree, root



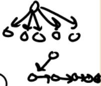
(an expression tree)

Traversals

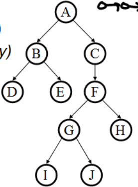
```
void traverse(BNode t){
  if (t != NULL){
    traverse (t.left);
    print t.element;
    traverse (t.right);
  }
}
```

Pre
In — Order
Post

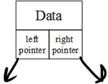
Binary Trees



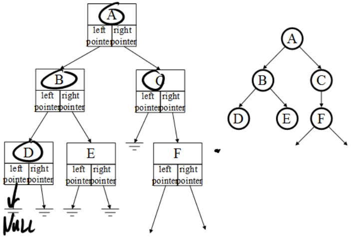
- Binary tree is
  - a root
  - left subtree *(maybe empty)*
  - right subtree *(maybe empty)*



- Representation:



Binary Tree: Representation



ADTs Seen So Far

- Stack
  - Push
  - Pop
- Queue
  - Enqueue
  - Dequeue
- Priority Queue
  - Insert
  - DeleteMin

Remember decreaseKey?

The Dictionary ADT

- Data:
  - a set of (key, value) pairs
- Operations:
  - Insert (key, value)
  - Find (key)
  - Remove (key)

insert(rea, ….)

find(dabacon)

- dabacon
  Dave Bacon, …

- dabacon
  Dave Bacon
  OH: Tu 12:30-1:30
  CSE 460
- ethanpg
  Ethan Phelps-Goodman,
  OH: Th 10:30-11:30
  CSE 218
- rea
  Ruth Anderson
  OH: M 3:30-4:30
  CSE 460

*The Dictionary ADT is sometimes called the "**Map ADT**"*

A Modest Few Uses

- Sets
- Dictionaries
- Networks : Router tables
- Operating systems : Page tables
- Compilers : Symbol tables

Probably the most widely used ADT!

Implementations

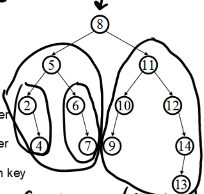
	insert	find	delete
• Unsorted Linked-list	$O(1)$	$O(n)$	find + delete $O(n)$
• Unsorted array	$O(1)$	$O(n)$	$O(n)$
• Sorted array	$\log n + n$ $O(n)$	$O(\log n)$	$\log n + n$ $O(n)$

O→O→O→O

moving nicely

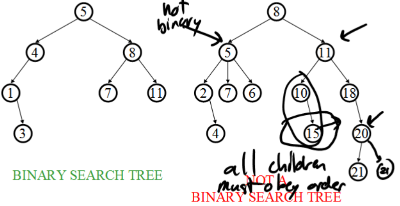
Binary Search Tree Data Structure

- Structural property
  - each node has ≤ 2 children
  - result:
    - storage is small
    - operations are simple
    - average depth is small
- Order property
  - all keys in left subtree smaller than root's key
  - all keys in right subtree larger than root's key
  - result: easy to find any given key
- What must I know about what I store?

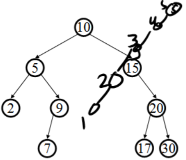


Comparison (equality)

Example and Counter-Example



Find in BST, Recursive

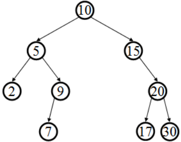



```
Node Find(Object key,
          Node root) {
  if (root == NULL)
    return NULL;

  if (key < root.key)
    return Find(key,

  root.left);
  else if (key > root.key)
    return Find(key,

  root.right);
  else
    return root;
}
```

Runtime: O(n)

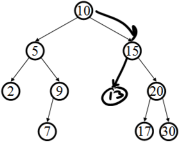
Find in BST, Iterative

```
Node Find(Object key,
          Node root) {

  while (root != NULL &&
         root.key !=
  key) {
    if (key < root.key)
      root = root.left;
    else
      root = root.right;
  }

  return root;
}
```



Runtime: Same as before O(n)

Insert in BST



Insert(13)
Insert(8)
Insert(31)

Find 13

Insertions happen only at the leaves – easy!

Runtime:

O(n) worst case

BuildTree for BST

- Suppose keys 1, 2, 3, 4, 5, 6, 7, 8, 9 are inserted into an initially empty BST.

  Runtime depends on the order!

  – in given order

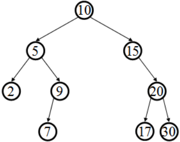
  – in reverse order

  – median first, then left median, right median, etc.

Bonus: FindMin/FindMax

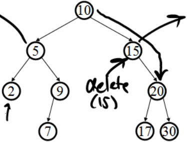
- Find minimum

- Find maximum



Deletion in BST

delete(5)



"hard" part

two children

Why might deletion be harder than insertion?

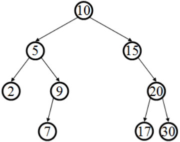
left, then go right

Lazy Deletion

Instead of physically deleting nodes, just mark them as deleted

- + simpler
- + physical deletions done in batches
- + some adds just flip deleted flag

- – extra memory for deleted flag
- – many lazy deletions slow finds
- – some operations may have to be modified (e.g., min and max)

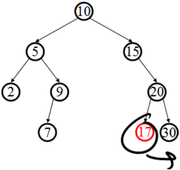


Non-lazy Deletion

- Removing an item disrupts the tree structure.
- Basic idea: find the node that is to be removed. Then “fix” the tree so that it is still a binary search tree.
- Three cases:
  - node has no children (leaf node)
  - node has one child
  - node has two children

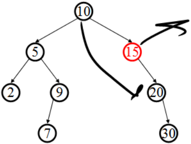
Non-lazy Deletion – The Leaf Case

Delete(17)



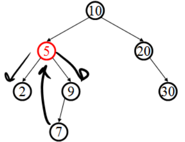
Deletion – The One Child Case

Delete(15)



Deletion – The Two Child Case

Delete(5)



What can we replace 5 with?

Deletion – The Two Child Case

Idea: Replace the deleted node with a value guaranteed to be between the two child subtrees!

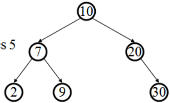
Options:

- *succ* from right subtree: findMin(t.right)
- *pred* from left subtree : findMax(t.left)

Now delete the original node containing *succ* or *pred*

- Leaf or one child case – easy!

Finally…



7 replaces 5

Original node containing
7 gets deleted

Balanced BST

O(n)

O(log n)

goal.

Observation

- BST: the shallower the better!
- For a BST with *n* nodes
  - Average height is O(log *n*)
  - Worst case height is O(*n*)
- Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario

Solution: Require a **Balance Condition** that

1. ensures depth is O(log *n*) – strong enough!
2. is easy to maintain – not too strong!

Potential Balance Conditions

1. Left and right subtrees of the root have equal number of nodes

2. Left and right subtrees of the root have equal *height*

Potential Balance Conditions

3. Left and right subtrees of *every node* have equal number of nodes

4. Left and right subtrees of *every node* have equal *height*

The AVL Balance Condition

Left and right subtrees of *every node*
have equal *heights* **differing by at most 1**

Define: **balance**(x) = height(x.left) – height(x.right)

AVL property: $-1 \leq \textbf{balance}(x) \leq 1$, **for every node** x

- Ensures small depth
  - Will prove this by showing that an AVL tree of height h must have a lot of (i.e. $O(2^h)$) nodes
- Easy to maintain
  - Using single and double rotations

The AVL Tree Data Structure

Structural properties

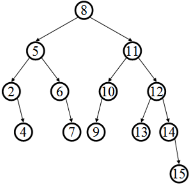
1. Binary tree property
2. Balance property: balance of every node is between -1 and 1

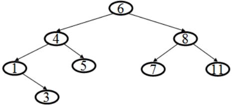
Result:

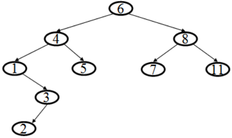
Worst case depth is O(log *n*)

Ordering property

– Same as for BST




6
4
8
1
5
7
11
3


6
4
8
1
5
7
11
3
2

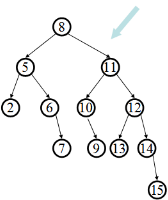
Proving Shallowness Bound

Let **S**(h) be the min # of nodes in an AVL tree of height h

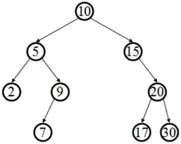
Claim: **S**(h) = **S**(h-1) + **S**(h-2) + 1

Solution of recurrence: **S**(h) = O(2^h)
(like Fibonacci numbers)

AVL tree of height h=4
with the min # of nodes



Testing the Balance Property



We need to be able to:

1.

2.

3.

`NULL`s have
height **`-1`**