Some Search Structures

• **Sorted Arrays**
  – Advantages
    • Search in $O(\log n)$ time (binary search)
  – Disadvantages
    • Need to know size in advance
    • Insertion, deletion $O(n)$ – need to shift elements

• **Lists**
  – Advantages
    • No need to know size in advance
    • Insertion, deletion $O(1)$ (not counting search time)
  – Disadvantages
    • Search is $O(n)$, even if list is sorted
Balanced Search Trees

• Best of both!
  – Search, insert, delete in $O(\log n)$ time
  – No need to know size in advance

• Several flavors
  – AVL trees, 2-3 trees, red-black trees, skip lists, random treaps, ...
Review – Binary Search Trees

• Every node has a left child, a right child, both, or neither
• Data elements are drawn from a totally ordered set
• Every node contains one data element
• Data elements are ordered in inorder
A Binary Search Tree
Binary Search Trees

In any subtree:

- all elements smaller than the element at the root are in the left subtree

- all elements larger than the element at the root are in the right subtree
Search

To search for an element $x$:

- if tree is empty, return false
- if $x = \text{object at root}$, return true
- If $x < \text{object at root}$, search left subtree
- If $x > \text{object at root}$, search right subtree
Search

Example: search for 13
Search

25

6

1

13

20

29

47

80

54

91

48

13?
Search

```
   6
  / \   /
 20 25 47
 /   /   /
1 13 29 80
   /     /
  13 54 91
   /   /   
  13 48   
```

13 is not found in the tree.
Search
Search

25

6

1

20

25

47

29

80

54

48

91
type 'a tree =
    Node of 'a * 'a tree * 'a tree
  | Leaf

let rec contains (t :'a tree) (x :'a) : bool =
    match t with
    Leaf -> false
  | Node (y, l, r) ->
      if x = y then true
      else if x < y then contains l x
      else contains r x
Insertion

To insert an element $x$:

- search for $x$ – if there, just return
- when arrive at a leaf $y$, make $x$ a child of $y$
  - left child if $x < y$
  - right child if $x > y$
Insertion

Example: insert 15
Insertion
Insertion
Insertion
Insertion
Insertion

let rec insert (x : 'a) (t : 'a tree) : 'a tree =
  match t with
  Leaf -> Node (x, Leaf, Leaf)
  (* if at a leaf, put new node there *)
  | Node (y, l, r) as t ->
  (* recursively search for insert point *)
    if x = y then t
    else if x > y then Node (y, l, insert x r)
    else (* x < y *) Node (y, insert x l, r)
Deletion

To delete an element $x$:
  • remove $x$ from its node – this creates a hole
  • if the node was a leaf, just delete it
  • find greatest $y$ less than $x$ in the left subtree
    (or least $y$ greater than $x$ in the right subtree), move it to $x$'s node
  • this creates a hole where $y$ was – repeat
Deletion

To find least $y$ greater than $x$:
- follow left children as far as possible in right subtree
Deletion

To find greatest y less than x:
- follow right children as far as possible in left subtree
Deletion

Example: delete 25
Deletion
Deletion
Deletion
Deletion
Deletion
Deletion

```
  20
 /  \  
 6   47
 / \   /  
1   29 80
 /  /    /   
13 29 54 91
```

13
Deletion
Deletion

```
Deletion

20
  /  
  6   47
 /    /
1  13  29
     /  
     80
   /    
   54   91
   /    
  48
```
Deletion

Example: delete 47
Deletion

```
    20
   /  \
  6    47
 / \
1  13
   /    \
  29    80
   /     /
  54  91
   /  \
  48
```
Deletion
Deletion
Deletion
Deletion
Deletion

Example: delete 29
Deletion
Deletion

```
   20
  /  \\
 6    80
  \
   13
```

- 20 (root)
- 6 (left of 20)
- 13 (right of 6)
- 80 (right of 20)
- 54 (left of 80)
- 91 (right of 80)
- 48 (left of 54)
Deletion

```
20
  /   \
 6  80
 /  |
1 13
```

- 20 has a right child 80.
- 80 has a left child 54 and a right child 91.
- 54 has a right child 48.
Deletion

```
                  20
                 /   \
                6     80
               /     /  \
              1     13   91
```

Node 54 is marked as deleted.
Deletion
Deletion
Deletion

Tree:

- Root: 20
  - Left: 6
    - Left: 1
    - Right: 13
  - Right: 48
    - Right: 80
      - Left: 54
      - Right: 91
Observation

- These operations take time proportional to the height of the tree (length of the longest path)
- $O(n)$ if tree is not sufficiently balanced

Bad case for search, insertion, and deletion – essentially like searching a list
Solution

Try to keep the tree *balanced* (all paths roughly the same length)
Balanced Trees

• Size is exponential in height
• Height = \log_2(\text{size})
• Search, insert, delete will be O(log n)
Creating a Balanced Tree

Creating one from a sorted array:
- Find the median, place that at the root
- Recursively form the left subtree from the left half of the array and the right subtree from the right half of the array

\[
\begin{array}{ccccccc}
1 & 6 & 13 & 20 & 48 & 54 & 80
\end{array}
\]

\[
\begin{array}{ccccccc}
1 & 6 & 13 & 20 & 48 & 54 & 80
\end{array}
\]
Keeping the Tree Balanced

• Insertions and deletions can put tree out of balance – we may have to rebalance it
• Can we do this efficiently?
AVL Trees

Adelson-Velsky and Landis, 1962

AVL Invariant:
The difference in height between the left and right subtrees of any node is never more than one
An AVL Tree

- Nonexistent children are considered to have height $-1$.

- Note that paths can differ in length by more than 1 (e.g., paths to 2, 48).
AVL Trees are Balanced

The AVL invariant implies that:

- Size is at least exponential in height
  - \( n \geq \varphi^d \), where \( \varphi = (1 + \sqrt{5})/2 \approx 1.618 \), the golden ratio!

- Height is at most logarithmic in size
  - \( d \leq \log n / \log \varphi \approx 1.44 \log n \)
AVL Trees are Balanced

AVL Invariant:
The difference in height between the left and right subtrees of any node is never more than one

To see that $n \geq \varphi^d$, look at the smallest possible AVL trees of each height

$A_0 \quad A_1 \quad A_2 \quad A_3$
AVL Trees are Balanced

**AVL Invariant:**
The difference in height between the left and right subtrees of any node is never more than one.

To see that $n \geq \varphi^d$, look at the smallest possible AVL trees of each height.
AVL Trees are Balanced

AVL Invariant:
The difference in height between the left and right subtrees of any node is never more than one.

To see that $n \geq \varphi^d$, look at the smallest possible AVL trees of each height:

$A_0$, $A_1$, $A_2$, $A_3$, ..., $A_{d-1}$, $A_d$
AVL Trees are Balanced

\[ A_0 = 1 \]
\[ A_1 = 2 \]
\[ A_d = A_{d-1} + A_{d-2} + 1, \quad d \geq 2 \]
AVL Trees are Balanced

\[ A_0 = 1 \]
\[ A_1 = 2 \]
\[ A_d = A_{d-1} + A_{d-2} + 1, \quad d \geq 2 \]

1 2 4 7 12 20 33 54 88 ...
AVL Trees are Balanced

\[ A_0 = 1 \]
\[ A_1 = 2 \]
\[ A_d = A_{d-1} + A_{d-2} + 1, \quad d \geq 2 \]

1 2 4 7 12 20 33 54 88 ...

1 1 2 3 5 8 13 21 34 55 ...

The Fibonacci sequence
AVL Trees are Balanced

\[ A_0 = 1 \]
\[ A_1 = 2 \]
\[ A_d = A_{d-1} + A_{d-2} + 1, \quad d \geq 2 \]

\[
\begin{align*}
1 & \quad 2 & \quad 4 & \quad 7 & \quad 12 & \quad 20 & \quad 33 & \quad 54 & \quad 88 & \quad \ldots \\
1 & \quad 1 & \quad 2 & \quad 3 & \quad 5 & \quad 8 & \quad 13 & \quad 21 & \quad 34 & \quad 55 & \quad \ldots \\
\end{align*}
\]

\[ A_d = F_{d+2} - 1 = O(\varphi^d) \]
Rebalancing

• Insertion and deletion can invalidate the AVL invariant
• May have to \textit{rebalance}
Rebalancing

Rotation
• A local rebalancing operation
• Preserves inorder ordering of the elements
• The AVL invariant can be reestablished with at most $O(\log n)$ rotations up and down the tree
Rebalancing

Example: delete 27
Rebalancing
Rebalancing
Rebalancing
Rebalancing
Rebalancing
2-3 Trees

Another balanced tree scheme

- Data stored only at the leaves
- Ordered left-to-right
- All paths of the same length
- Every non-leaf has either 2 or 3 children
- Each internal node has smallest, largest element in its subtree (for searching)
2-3 Trees

smallest 2-3 tree of height $d = 3$
$2^d = 8$ data elements

largest 2-3 tree of height $d = 3$
$3^d = 27$ data elements

- number of elements satisfies $2^d \leq n \leq 3^d$
- height satisfies $d \leq \log n$
Insertion in 2-3 Trees
Insertion in 2-3 Trees

want to insert new element here
Insertion in 2-3 Trees
Insertion in 2-3 Trees

want to insert new element here
Insertion in 2-3 Trees
Insertion in 2-3 Trees
Insertion in 2-3 Trees
Insertion in 2-3 Trees
Deletion in 2-3 Trees

want to delete this element
Deletion in 2-3 Trees
Deletion in 2-3 Trees

want to delete this element
Deletion in 2-3 Trees

If neighbor has 3 children, borrow one
Deletion in 2-3 Trees

If neighbor has 3 children, borrow one
Deletion in 2-3 Trees

If neighbor has 2 children, coalesce with neighbor
Deletion in 2-3 Trees

If neighbor has 2 children, coalesce with neighbor
Deletion in 2-3 Trees

This may cascade up the tree!
Deletion in 2-3 Trees

This may cascade up the tree!
Deletion in 2-3 Trees

This may cascade up the tree!
Deletion in 2-3 Trees

This may cascade up the tree!
Conclusion

Balanced search trees are good

- Search, insert, delete in $O(\log n)$ time
- No need to know size in advance
- Several different versions
  - AVL trees, 2-3 trees, red-black trees, skip lists, random treaps, Huffman trees, ...
  - find out more about them in CS4820