Recall: Dynamic Sets

• data structures rather than straight algorithms
• In particular, structures for dynamic sets
  – Elements have a key and satellite data
  – Dynamic sets support queries such as:
    \[ \textbf{Search}(S, k), \textbf{Minimum}(S), \textbf{Maximum}(S), \]
    \[ \textbf{Successor}(S, x), \textbf{Predecessor}(S, x) \]
  – They may also support modifying operations like:
    \[ \textbf{Insert}(S, x), \textbf{Delete}(S, x) \]
Motivation

• Given a sequence of values:
  – How to get the max, min value efficiently?
  – How to find the location of a given value?
  – …

• Trivial solution
  – Linearly check elements one by one

• Searching Tree data structure supports better:
  – SEARCH, MINIMUM, MAXIMUM,
  – PREDECESSOR, SUCCESSOR,
  – INSERT, and DELETE operations of dynamic sets
Binary Search Trees

- **Binary Search Trees (BSTs)**
  - Each node has at most two children
  - An important data structure for dynamic sets

- Each node contains:
  - key and data
  - left: points to the left child
  - right: points to the right child
  - p(parent): point to parent

- Binary-search-tree property:
  - y is a node in the left subtree of x: \( y.key \leq x.key \)
  - y is a node in the right subtree of x: \( y.key \geq x.key \)
  - The value stored at a node is greater than the value stored at its left child and less than the value stored at its right child

- Height: \( h \)
Binary Search Trees

• The height (h) is important for Binary Search Trees
  – Tree operations (e.g., insert, delete, retrieve etc.) are typically expressed in terms of h.
  – So, h determines running time!

• What is the max height of a tree with N nodes?
  – N (same as a linked list)

• What is the min height of a tree with N nodes?
  – \( \log(N+1) \)
How to search: Binary Search Trees

- BST property:
  \[ \text{key}[\text{leftSubtree}(x)] \leq \text{key}[x] \leq \text{key}[\text{rightSubtree}(x)] \]

- Example:
Examples
Print out Keys

• Preorder tree walk
  – Print key of node before printing keys in subtrees (node left right)

• Inorder tree walk
  – Print key of node after printing keys in its left subtree and before printing keys in its right subtree (left node right)

• Postorder tree walk
  – Print key of node after printing keys in subtrees (left right node)
Example

- Preorder tree walk
  - F, B, A, D, C, E, G, I, H

- Inorder tree walk
  - A, B, C, D, E, F, G, H, I
    - Sorted (why?)

- Postorder tree walk
  - A, C, E, D, B, H, I, G, F
Inorder Tree Walk

\textbf{Inorder-Tree-Walk}(x)

1 \hspace{1em} \textbf{if} \ x \neq \text{NIL} \\
2 \hspace{1em} \text{Inorder-Tree-Walk}(x.\text{left}) \\
3 \hspace{1em} \text{print} \ x.\text{key} \\
4 \hspace{1em} \text{Inorder-Tree-Walk}(x.\text{right})

- Inorder tree walk
  - \textit{Visit and print each node once}
  - \textit{Time: } \Theta(n)
Inorder Tree Walk

• Example:

How long will a tree walk take?

Prove that inorder walk prints in monotonically increasing order
Operations

• Querying operations
  – Search: get node of given key
  – Minimum: get node having minimum key
  – Maximum: get node having maximum key
  – Successor: get node right after current node
  – Predecessor: get node right before current node

• Updating operations
  – Insertion: insert a new node
  – Deletion: delete a node with given key
Operations on BSTs: Search

• Given a key and a pointer to a node, returns an element with that key or NULL:

\[
\text{TreeSearch}(x, k) \\
\text{if } (x = \text{NULL} \text{ or } k = \text{key}[x]) \\
\text{\hspace{1cm} return } x; \\
\text{if } (k < \text{key}[x]) \\
\text{\hspace{1cm} return TreeSearch(left}[x], k); \\
\text{else} \\
\text{\hspace{1cm} return TreeSearch(right}[x], k); \\
\]

Time = the length of path from root to found node

Time: \(O(h)\)
BST Search: Example

• Search for $D$ and $C$:
Operations on BSTs: Search

• Here’s another function that does the same:

```
TreeSearch(x, k)
    while (x != NULL and k != key[x])
        if (k < key[x])
            x = left[x];
        else
            x = right[x];
    return x;
```

• Which of these two functions is more efficient?
Operations: Minimum and Maximum

- Minimum: left most node
- Maximum: right most node
- Time: $O(h)$
Operations of BSTs: Insert

- Adds an element $x$ to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert $x$ in place of NULL
  - Use a “trailing pointer” to keep track of where you came from (like inserting into singly linked list)
  - Time: $O(h)$
Operations of BSTs: Insert

TREE-INSERT \((T, z)\)

1 \(y = \text{NIL}\)
2 \(x = T.\text{root}\)
3 \textbf{while} \(x \neq \text{NIL}\)
4 \hspace{1em} \(y = x\)
5 \hspace{1em} \textbf{if} \(z.\text{key} < x.\text{key}\)
6 \hspace{2em} \(x = x.\text{left}\)
7 \hspace{1em} \textbf{else} \(x = x.\text{right}\)
8 \(z.p = y\)
9 \textbf{if} \(y == \text{NIL}\)
10 \hspace{1em} \(T.\text{root} = z\) \hspace{1em} // \text{tree } T \text{ was empty}
11 \hspace{1em} \textbf{elseif} \(z.\text{key} < y.\text{key}\)
12 \hspace{2em} \(y.\text{left} = z\)
13 \hspace{1em} \textbf{else} \(y.\text{right} = z\)
BST Insert: Example

• Example: Insert C
BST Search/Insert: Running Time

• What is the running time of `TreeSearch()` or `TreeInsert()`?
  
  A: $O(h)$, where $h =$ height of tree

• What is the height of a binary search tree?
  
  A: worst case: $h = O(n)$ when tree is just a linear string of left or right children

  – We’ll keep all analysis in terms of $h$ for now
  – Later we’ll see how to maintain $h = O(lg n)$
Sorting With Binary Search Trees

• Informal code for sorting array $A$ of length $n$:
  
  $\text{BSTSort}(A)$
  
  for $i=1$ to $n$
    $\text{TreeInsert}(A[i])$;
  
  $\text{InorderTreeWalk}(\text{root})$;

• Argue that this is $\Omega(n \lg n)$

• What will be the running time in the
  – Worst case?
  – Average case? (hint: remind you of anything?)
Sorting With BSTs

• Average case analysis
  – It’s a form of quicksort!

```plaintext
for i=1 to n
  TreeInsert(A[i]);
  InorderTreeWalk(root);
```
Sorting with BSTs

• Same partitions are done as with quicksort, but in a different order
  – In previous example
    ➢ Everything was compared to 3 once
    ➢ Then those items < 3 were compared to 1 once
    ➢ Etc.
  – Same comparisons as quicksort, different order!
    ➢ Example: consider inserting 5
Sorting with BSTs

• Since run time is proportional to the number of comparisons, same time as quicksort: $O(n \lg n)$

• *Which do you think is better, quicksort or BSTsort? Why?*
Sorting with BSTs

• Since run time is proportional to the number of comparisons, same time as quicksort: $O(n \lg n)$

• Which do you think is better, quicksort or BSTSort? Why?

• A: quicksort
  – Better constants
  – Sorts in place
  – Doesn’t need to build data structure
More BST Operations

- BSTs are good for more than sorting. For example, can implement a priority queue

- *What operations must a priority queue have?*
  - Insert
  - Minimum
  - Extract-Min
BST Operations: Successor

```
TREE-SUCCESSOR(x)
1     if x.right ≠ NIL
2         return TREE-MINIMUM(x.right)
3     y = x.p
4     while y ≠ NIL and x == y.right
5         x = y
6     y = y.p
7     return y
```

- Time: O(h)
Example

• Successor of 15 is 17
• Successor of 13 is 15
BST Operations: Successor

• Two cases:
  – x has a right subtree: successor is minimum node in right subtree
  – x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x

  ➢ Intuition: As long as you move to the left up the tree, you’re visiting smaller nodes.

• Predecessor: similar algorithm
BST Operations: Delete

- Deletion is a bit tricky
  - Key point: choose a node in subtree rooted at x to replace the deleted node x
  - Node to replace x: predecessor or successor of x

- 3 cases:
  - x has no children:  
    - Remove x
  - x has one child: 
    - Splice out x
  - x has two children:
    - Swap x with successor
    - Perform case 1 or 2 to delete it

Example: delete K or H or B
BST Operations: Delete

- *Why will case 2 always go to case 0 or case 1?*
  
  A: because when x has 2 children, its successor is the minimum in its right subtree

- *Could we swap x with predecessor instead of successor?*
  
  A: yes. *Would it be a good idea?*
  
  A: might be good to alternate

- Up next: guaranteeing a O(lg n) height tree
Has one child

Replace $z$ by its child
Right child has no left subtree

Replace \( z \) by its successor \( y \)
Right child has left subtree

1. Find successor $y$ of $z$
2. Replace $y$ by its child
3. Replace $z$ by $y$
Replace a mode by its Child

- Replace the subtree rooted at node $u$ with the subtree rooted at node $v$
- Running time: $O(1)$

```plaintext
TRANSPLANT($T$, $u$, $v$)
1    if $u.p = \text{NIL}$
2        $T.root = v$
3    elseif $u = u.p.left$
4        $u.p.left = v$
5    else $u.p.right = v$
6    if $v \neq \text{NIL}$
7        $v.p = u.p$
```
Deletion Algorithm

- Main running time: find z’s successor
- Time: $O(h)$

TREE-DELETE($T$, $z$)
1. if $z.left == \text{NIL}$
2. TRANSPLANT($T$, $z$, $z.right$)
3. elseif $z.right == \text{NIL}$
4. TRANSPLANT($T$, $z$, $z.left$)
5. else $y = \text{TREE-MINIMUM}(z.right)$
6. if $y.p \neq z$
7. TRANSPLANT($T$, $y$, $y.right$)
8. $y.right = z.right$
9. $y.right.p = y$
10. TRANSPLANT($T$, $z$, $y$)
11. $y.left = z.left$
12. $y.left.p = y$
Binary Search Tree

• View today as data structures that can support **dynamic set operations**.
  – Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.

• Can be used to build
  – *Dictionaries*.
  – *Priority Queues*.

• Basic operations take time proportional to the height of the tree – $O(h)$. 
## Binary Search Tree vs Linear List

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary Search Tree</th>
<th>Array-based List</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Destructor</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>IsFull</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>IsEmpty</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>RetrieveItem</td>
<td>O(logN)(^*)</td>
<td>O(logN)</td>
<td>O(N)</td>
</tr>
<tr>
<td>InsertItem</td>
<td>O(logN)(^*)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>DeleteItem</td>
<td>O(logN)(^*)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

*Assuming h = O(logN)
Summary

• Binary search tree stores data hierarchically
• Support SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE operations
• Running time of all operation is $O(h)$
• Question: What is the lower bound of $h$? How to achieve it?