Outline of Today's Class
Binary Search Trees
Balanced Search Trees
Binomial Trees
Binomial Heaps

Lecture: Analysis of Algorithms (CS583 - 002)

Amarda Shehu

Fall 2017
1. **Outline of Today’s Class**

2. **Binary Search Trees**
   - Traversals
   - Querying
   - Insertion and Deletion
   - Sorting with BSTs

3. **Balanced Search Trees**
   - Red-black Trees
   - Height of a Red-black Tree
   - Operations on Red-black Trees
     - Rotations
     - Insertion

4. **Binomial Trees**
   - Properties of Binomial Trees
   - Binomial Trees and Binomial Theorem
   - Merge Operation on Binomial Trees

5. **Binomial Heaps**

---

Lecture: Analysis of Algorithms (CS583 - 002)
Definition of a Binary Search Tree

A Binary Search Tree (BST) is a binary tree where:

- if \( y \) is a node in the left subtree of \( x \), then \( \text{key}[y] \leq \text{key}[x] \)
- if \( y \) is a node in the right subtree of \( x \), then \( \text{key}[y] \geq \text{key}[x] \)
### BST Traversals or Walks

#### BST Walks
- **Preorder**: Visit node, then left subtree, then right subtree.
- **Inorder**: Visit left subtree, then node, then right subtree.
- **Postorder**: Visit left subtree, then right subtree, then node.

#### Inorder Tree Walk

**Inorder-Tree-Walk**(x)

1. **if** x ≠ NULL **then**
2. Inorder-Tree-Walk(left[x])
3. print key[x]
4. InorderTree-Walk(right[x])

- What changes in the other algorithms?
- What is the running time?
- What is the recurrence relationship?
- How would you solve the recurrence?
Querying a BST: Searching for a Key

Querying using Recursion:

**Tree-Search**($x, k$)

1: if $x$ = NULL or $k$ = key[$x$] then
2: return $x$
3: if $k$ < key[$x$] then
4: return Tree-Search(left[$x$], $k$)
5: else return
   Tree-Search(right[$x$], $k$)

Querying using No Recursion:

**Iterative-Tree-Search**($x, k$)

1: while $x$ $\neq$ NULL and $k$ $\neq$ key[$x$]
   do
2: if $k$ < key[$x$] then
3: $x$ $\leftarrow$ left[$x$]
4: else $x$ $\leftarrow$ right[$x$]
5: return $x$

- What do the algorithms return when $k$ is not in the BST?
- What is the running time of these algorithms?
- What is the recurrence relationship of the recursive version?
- How can you solve the recurrence?
- Which implementation would you choose?
Querying a BST: More Operations

Searching for Minimum and Maximum Keys
- How would you search for the minimum key in a BST?
- Similarly, how would you search for the maximum key?
- What time cost do you incur on each?

Searching for Successors and Predecessors
- Define successor\( (x) \) as the node with smallest key \( \geq \text{key}[x] \)
- Define predecessor\( (x) \) as the node with largest key \( \leq \text{key}[x] \)
- Design an algorithm to find the successor of a node \( x \)
- Design an algorithm to find the predecessor of a node \( x \)
- What time cost do you incur on each?
Finding the Successor of a Node: Case 1

- Successor when right subtree not empty:
  - is leftmost node of right subtree
  - has minimum key in right subtree

\[ \text{successor}(x) \]
Finding the Successor of a Node: Case 1

- **Successor when right subtree is empty:**
  - lowest ancestor of $x$
  - whose left child is also an ancestor of $x$
Finding the Successor of a Node

\textbf{Successor}(x)

1. \textbf{if} right[x] \neq \text{NULL} \textbf{ then}
2. \hspace{1cm} \textbf{return} \ Tree-Minimum(right[x])
3. \hspace{1cm} y \leftarrow p[x]
4. \hspace{1cm} \textbf{while} y \neq \text{NULL} \text{ and } x = \text{right}[y] \textbf{ do}
5. \hspace{1.5cm} x \leftarrow y
6. \hspace{1.5cm} y \leftarrow p[y]
7. \hspace{1cm} \textbf{return} y

- What is the running time of this algorithm?
- How is finding the predecessor different?
**Question:** How would you insert a key in a BST?

**Answer:** In its proper place.

- Insert 10, 21, 24, 30 into BST

- What is the worst-case running time?
Deleting a Node From a BST

There are three possible cases when removing a node $z$:

1. $z$ has no children: delete it
2. $z$ has only one child: splice it out
3. $z$ has two children:
   - find the successor $y$ of $z$
   - splice out $y$
   - replace the key of $z$ with that of $y$
Consider the array \( A = [3, 1, 8, 2, 6, 7, 5] \)

Sorting it with a BST requires only two stages:
1. First insert all the elements in a BST
2. Then perform an inorder traversal
Time Complexity of Sorting an Array with BST

- BST sort performs the same comparisons as quicksort, but in a different order.
- So, the expected time to build the tree is asymptotically the same as the running time of quicksort.
The depth of a node is the number of comparisons made during its insertion into the BST.

A total of $O(n \cdot \log n)$ comparisons are made (expected running time of quicksort).

So, average depth of a node is $\frac{1}{n} \cdot O(n \cdot \log n)$, or $O(\log n)$. 
Definition of a Balanced Search Tree

Balanced Search Tree:
A search-tree data structures for which a height of $O(lgn)$ is guaranteed when inserting a dynamic set of $n$ items.

Examples
- AA Trees
- Splay Trees
- Scapegoat Trees
- Treaps
- AVL Trees
- 2-3 and 2-3-4 trees
- B-trees
- Red-black trees
Outline of Today’s Class
Binary Search Trees
Balanced Search Trees
Binomial Trees
Binomial Heaps

Red-black Trees
Height of a Red-black Tree
Operations on Red-black Trees

Example of a Red-black tree

13
8
1
6
NIL
NIL

11
NIL
NIL

17
15
22
25
27
NIL
NIL
NIL
NIL
NIL
NIL
NIL
Red-black trees are self-balancing binary search trees: they keep their height small through transformations known as rotations.

Properties

- Each node is either red or black
  - A one-bit color field is needed per node
- The root is black
- The leaves are the NIL’s and are colored black
- If a node is red, then both its children are black
  - This also means that a red node has a black parent
- All paths from a node to descendant leaves have the same number of black nodes (black-height)
  - Property ensures that the tree is balanced
For convenience, NILs sometimes collected in a NIL sentinel

Nr. of black nodes on any path from a node $x$ (not including $x$) down to a leaf is black-height of $x$, or $bh(x)$
Theorem: A red-black tree with $n$ internal nodes has height

$$h \leq 2 \cdot \lg(n + 1)$$

Proof: Makes use of two corollaries

1. The height $h$ of a tree is at most twice the black-height $bh$
2. Subtree rooted at a node $x$ has $\geq 2^{bh(x)} - 1$ internal nodes

1. Since a red parent can only have black children, at least half the nodes on a path from root to leaf are black: $bh \geq h/2$. 
2. A subtree rooted at \( x \) has \( n_x \) internal nodes, which are \( x \) and the number of internal nodes in its left and right subtrees. Let them be \( n_l \) and \( n_r \), respectively. Hence, \( n_x = 1 + n_l + n_r \).

The left and right subtrees each have \( \geq bh(x) - 1 \) black heights (property of red-black tree). Assuming that the property holds for the left and right subtrees (we are using induction), then
\[
  n_x \geq 1 + 2 \cdot (2^{bh(x)-1} - 1) = 2^{bh(x)} - 1. \quad \text{[what is the base case?]}
\]

Putting it all together: When \( x \) is root, \( n_x = n \). So, \( n \geq 2^{bh(x)} - 1 \).

This means that \( \log(n + 1) \geq bh \geq h/2 \). Hence, \( h \geq 2 \cdot \log(n + 1) \).
Query Operations: SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR run in $O(lgn)$ time on a red-black tree with $n$ internal nodes.

Modifying Operations: INSERT and DELETE, also running in $O(lgn)$ time on the number of internal nodes, cause modifications to the red-black tree.

- color changes
- red-black property may be violated
  - restored by restructuring the links of the tree via rotations
  - rotations restore the $O(lgn)$ bound of the height
Rotations maintain the inorder ordering of keys (BST property):

\[ a \in \alpha, \ b \in \beta, \ c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c \]

A rotation can be performed in \( O(1) \) time.
**Insertion into a Red-black Tree**

**Idea:** Insert $x$ in tree and color it red. Only red-black property may be violated. Move the violation up the tree recoloring it until can be fixed with rotations and recoloring.

**Example:**
- Insert $x = 15$ as in BST
- Recolor and move violation up the tree
**Idea:** Insert $x$ in tree and color it red. Only red-black property may be violated. Move the violation up the tree recoloring it until can be fixed with rotations and recoloring.

**Example:**
- Insert $x = 15$ as in BST
- Recolor and move violation up the tree
- Right-rotate(18)
**Idea:** Insert $x$ in tree and color it red. Only red-black property may be violated. Move the violation up the tree recoloring it until can be fixed with rotations and recoloring.

**Example:**
- Insert $x = 15$ as in BST
- Recolor and move violation up the tree
- Right-rotate(18)
- Left-rotate(7) and recolor
Insertion into a Red-black Tree

**Idea:** Insert $x$ in tree and color it red. Only red-black property may be violated. Move the violation up the tree recoloring it until can be fixed with rotations and recoloring.

**Example:**

- Insert $x = 15$ as in BST
- Recolor and move violation up the tree
- Right-rotate(18)
- Left-rotate(7) and recolor
Final Words on Red-black Trees

Insertion running time: $O(lgn)$ with $O(1)$ rotations

Delete has same asymptotic running time and number of rotations

**Conclusion**: Red-black trees are very useful self-balancing binary search trees to implement associative arrays (e.g., STL map uses such trees). Implementing Insert and Delete for them is complex (and almost always requires looking up material) but pays off.
Binomial Trees

Definition of a Binomial Tree $B_k$

The binomial tree of order/degree $k$ is defined as follows:

- The binomial tree of order $k = 0$ consists of a single node, $r$.
- The binomial tree of order $k > 0$ consists of root $r$ and $k$ binomial subtrees $\{B_0, \ldots, B_{k-1}\}$.

Applications and Examples of Binomial Trees

- Price options, dividends, interest rates
- Construct binomial heaps
Properties of Binomial Trees

1. \( \text{degree}(\text{root}(B_k)) = k \)
2. \( |B_k| = 2^k \)
3. \( h(B_k) = k \)

These properties mean that the degree and depth of a binomial tree with \( n \) nodes is \( \log(n) \).
Number of Nodes in a Binomial Tree

**Theorem:** The binomial tree $B_k$ has $2^k$ nodes

**Proof:** Let $n_k = |B_k|$. We proceed by induction.

**Base Case:** $|B_0| = 1 = 2^0$ So, the formula holds.

**Inductive Step:** Assume that the formula holds for all binomial trees of order 0 to $k - 1$. By definition, $B_k = \{r, B_0, B_1, \ldots, B_{k-1}\}$. Hence, the number of nodes in $B_k$ is:

$$
\begin{align*}
    n_k &= 1 + \sum_{i=0}^{k-1} B_i \\
    &= 1 + \sum_{i=0}^{k-1} 2^i \\
    &= 1 + \frac{2^k - 1}{2 - 1} \\
    &= 2^k
\end{align*}
$$

We have shown by induction that $n_k = 2^k$, $\forall k \geq 0$. It follows that binomial trees only come in sizes that are powers of 2. Moreover, for a given power of 2, there is a unique binomial tree.
Height of a Binomial Tree

**Theorem:** The height of $B_k$ is $k$.

**Proof:** Let $h_k$ be the height of $B_k$. We proceed by induction.

**Base Case:** By definition, $B_0$ consists of a single node. So, its height is 0, same as its degree. So, the formula holds.

**Inductive Step:** Assume that the formula holds for all binomial trees of order 0 to $k - 1$. By definition, $B_k = \{r, B_0, B_1, \ldots, B_{k-1}\}$. Hence, the height of $B_k$ is:

\[
 h_k = 1 + \max_{0 \leq i \leq k-1} h_i \\
 = 1 + \max_{0 \leq i \leq k-1} i \\
 = 1 + k - 1 \\
 = k
\]

Hence, we have shown by induction that $h_k = k$, $\forall k \geq 0$. Since $|B_k| = 2^k$ and $h_k = k$, then $h_k = \log(|B_k|)$. 
Alternative Ways of Constructing Binomial Trees

Figure: $B_k = \{ r, B_0, \ldots, B_{k-1} \}$

Figure: $B_k = \{ B_{k-1}, B_{k-1} \}$
What is the origin of the name *binomial tree*?

- Number of nodes at a given depth in a binomial tree is determined by the binomial coefficient.
- The binomial coefficient is in the context of the binomial theorem.
- The binomial theorem computes the power of a binomial.
- A binomial is an expression that consists of two terms, $x + y$. That is why $B_k = \{ B_{k-1}, B_{k-1} \}$ is called a binomial tree.

**Binomial Theorem:** $(x + y)^n = \sum_{i=0}^{n} \binom{n}{i} \cdot x^i \cdot y^{n-i}$, where $\binom{n}{i}$ is the binomial coefficient.
Properties of Binomial Trees

**Theorem:** The number of nodes at depth $l$ in $B_k$, where $0 \leq l \leq k$ is given by the binomial coefficient $\binom{k}{l}$

**Proof:** Let $n_k(l)$ be the number of nodes at depth $l$ in $B_k$. We now proceed by induction.

**Base Case:** Since $|B_0| = 1 = \binom{1}{0} = 1$, the formula holds.

**Inductive Step:** Assume that $n_i(l) = \binom{i}{l}$ for $0 \leq i \leq k - 1$. Since $B_k = \{B_{k-1}, B_k-1\}$, we have:

$$n_k(l) = n_{k-1}(l) + n_{k-1}(l-1)$$

$$= \binom{k-1}{l} + \binom{k-1}{l-1}$$

$$= \frac{(k-1)!}{l!(k-1-l)!} + \frac{(k-1)!}{(l-1)!(k-2-l)!}$$

$$= \frac{(k-1)!((k-1)+1-l)+(k-1)!!}{(k-1+1-l)!!}$$

$$= \binom{k}{l}$$

We showed by induction that $n_k(l) = \binom{k}{l}$
Basic Node Implementation in Binomial Trees

- Maintaining pointer to parent allows up-traversal
- Maintaining degree to know how many children
- Maintaining pointer to sibling allows in-traversal
- Implementation affects Merge operation
Consider binomial trees that satisfy the min-heap property.

Merging two binomial trees of same order is an important operation: \( B_{k+1} \leftarrow B_k + B_k \)

Its running time is \( O(1) \)

**MERGETrees(p, q)**

1. if \( \text{key}[p.\text{root}] \leq \text{key}[q.\text{root}] \) then
2. return \( p.\text{addSubTree}(q) \)
3. else return \( q.\text{addSubTree}(p) \)
A binomial heap is a set of binomial trees with the two properties:

1. Each binomial tree in a heap obeys the minimum-heap property: the key of parent is \( \leq \) than the keys of the children.
   - This property ensures that the root of each binomial tree contains the smallest key in the tree.

2. There are \( \leq 1 \) binomial trees for each order.
   - This property implies that a binomial heap with \( n \) elements has \( \leq \log_2 n + 1 \) binomial trees.
The number and orders of these trees are uniquely determined by the number of elements $n$.

Each binomial tree corresponds to digit one in the binary representation of number $n$.

E.g., 13 is 1101 in binary, which is $2^3 + 2^2 + 2^0$.

A binomial heap with 13 elements consists of three binomial trees of orders 3, 2, and 0.
Operations on Binomial Heaps

- **Insert new element**: $O(lgn)$
  - Create a new heap containing element: $O(1)$
  - Merge new heap with current one: $O(lgn)$
- **Find minimum**: $O(lgn)$
  - Find minimum among roots of binomial trees
  - There are $O(lgn)$ binomial trees
- **Delete minimum**: $O(lgn)$
  - Find minimum element: $O(lgn)$
  - Remove element from its binomial tree: $O(1)$
  - Obtain list of subtrees: $O(1)$
  - Reorder from largest to smallest: $O(lgn)$
  - Merge heap with original heap: $O(lgn)$
- Merge two heaps: $O(lgn)$
**Merging Two Binomial Heaps**

- Lists of roots of heaps are traversed simultaneously.
- If only one heap contains $B_j$, move $B_j$ to merged heap.
- If both heaps $H_1$ and $H_2$ contain $B_j$, create $B_{j+1} \leftarrow B_j + B_j$.
- Alternatively, all roots can be linked in a linked list in sorted order.
- Iterate over list and merge binomial trees of same order.

- Heaps have $\leq \lfloor \lg(n_1) \rfloor + 1$ and $\lfloor \lg(n_2) \rfloor + 1$ roots.
- $T(n)$ is then $O(\lg n)$ calls to MergeTrees $\rightarrow O(\lg n)$. 

---

*Amarda Shehu*  
*Lecture: Analysis of Algorithms (CS583 - 002)*