CS2351
Data Structures

Lecture 13:
Binary Search Tree
About this lecture

• A binary search tree (BST) is a binary tree that stores a set of items, and each item has a distinct key chosen from an ordered set
  • allows various queries and updates

• In this lecture, we discuss how the BST supports the queries and the updates
Binary Search Tree (BST)

- Each node in a BST has a distinct key
- The keys in the nodes satisfies the following BST property:

  Let $x$ be a node in a BST. Let $y$ and $z$ be nodes in the left and right subtrees of $x$, respectively. Then we have $y.key < x.key < z.key$
Example of BST

Is this a BST?
What happens if we perform inorder traversal in a BST?
Queries in a BST
Queries in a BST

- A BST supports the following queries:
  1. Finding nodes with min or max keys
  2. Given a value $k$, search for a node that contains $k$ as the key
  3. Given a node $x$, return the successor or the predecessor of $x$

  successor: node with key just larger than $x$.key
  predecessor: node with key just smaller than $x$.key
Successor and Predecessor

Successor(x) = node with key 8
Predecessor(y) = node with key 31
Finding Min or Max

• Where is the node with min key?
  ➔ The leftmost node in BST

• Where is the node with max key?
  ➔ The rightmost node in BST

• In general, let \( x \) be a node in the BST
  Q: Where is the node with min/max key in the subtree rooted at \( x \)?
Implementation in C

• We define a function \texttt{Min}, which returns a pointer to the min key node in subtree of \( x \)

\[
\text{Node} \ast \text{Min}( \text{Node} \ast x ) \{
  \text{while} ( x \rightarrow \text{left} \neq \text{NULL} ) \\
  x = x \rightarrow \text{left} ; \\
  \text{return} x ;
\}
\]

• Then desired min is equal to \texttt{Min}(r), where \( r \) = a pointer to the root of BST
Implementation in C

• We define a function \textbf{Max}, which returns a pointer to the max key node in subtree of x

```c
Node * Max( Node *x ) {
    while ( x->right != NULL )
        x = x->right ;
    return x ;
}
```

• Then desired max is equal to \textbf{Max(r)}
Searching a Key

- Let \( k \) be the key to be searched. Suppose \( k < \text{root.key} \). What can we conclude?
- In fact, searching a BST is very similar to doing binary search in a sorted array:

1. If \( k \) is equal to root.key, done!
2. Else if \( k < \text{root.key} \), recursively search left subtree of root
3. Else, recursively search right subtree
Example of Searching a BST

Search for key 30

not found
We define a function `Search`:

```c
Node * Search( Node *x, int k ) {
    if ( x == NULL ) return NULL;
    if ( x->key == k ) return x;
    if ( x->key > k )
        return Search( x->left, k );
    return Search( x->right, k );
}
```

Then, desired node = `Search(r, k)` , where
r = pointer to root of BST
Finding Successor

- Let $x$ be a node in the BST
- The successor of $x$ is the next node in the inorder traversal

1. What if $x$ has a right child?
   - $\min$ in the subtree of right child
2. What if not?
   - First ancestor “on the right” of $x$
Finding Successor

Successor of x

Successor of y
To help finding successor, we assume that each node has a parent pointer

Then we can define Successor as follows:

```c
Node * Successor( Node *x ) {
    if ( x->right != NULL )
        return Min( x->right ) ;
    y = x->parent ;
    while ( y != NULL && x == y->right )
    {
        x = y ;
        y = y->parent ;
    }
    return y ;
}
```
• Similarly, we can define **Predecessor**:

```c
Node * Predecessor( Node *x ) {
    if ( x->left != NULL )
        return Max( x->left ) ;
    y = x->parent ;
    while ( y != NULL && x == y->left )
    {
        x = y ;
        y = y->parent ;
    }
    return y ;
}
```
Query Performance

• Let $h$ denote the node-height of the BST

**Theorem:**
The queries minimum, maximum, search, predecessor, and successor can each be performed in $O(h)$ time

• What is the value of $h$ in the best case? How about the worst case?
Updates in a BST
Updates in a BST

- A BST supports the following updates:
  1. Inserting a node $z$ with key $k$
  2. Deleting a node $x$

- Note: When we perform updates, we have to maintain the BST property
Inserting a Node

• Let $z$ be a new node to be inserted, and $k$ be its key.

• Observation: After insertion, $k$ becomes searchable in BST.
  
  ➔ the insertion position is the same as the position we expect to find $k$.

• Insertion is done by slightly modifying the searching algorithm.
Example of Insertion in BST

Insert a node with key 30
void Insert( Node *x, Node *z ) {
    if ( x->key > z->key ) {
        if ( x->left ) Insert( x->left, z );
        else x->left = z ;
    }
    else if ( x->key < z->key ) {
        if ( x->right ) Insert( x->right, z );
        else x->right = z ;
    }
}

• Then, insertion is done by Insert(r, z), where r = pointer to root of BST
Deleting a Node

- Let $x$ be a node to be deleted
- **Case 1:**
  
  If $x$ is a leaf, we just remove $x$
Deleting a Node

• **Case 2:**
  
  If \( x \) has one child, we connect \( x \)'s parent to its child

```
Delete 48
```
Deleting a Node

- **Case 3:**
  
  If \( x \) has two children, we swap \( x \) with its successor, and then delete \( x \)
Deleting a Node

• In Case 3, the successor of $x$ does not have a left child. Why?

Before:

```
  8
 /\  \
5 23
 /\  /\  \
3 6 48 31
```

After:

```
  5
 /\  \
3 6
```

Delete 8
Implementation in C

• To ease our discussion, we now define a function \texttt{Transplant}, such that:

\texttt{Transplant}(x, y) links \(x\)'s parent to \(y\) and \(y\)'s parent is changed accordingly.
The function\( \text{Transplant}(x, y) \) can be easily implemented as follows:

```c
void Transplant( Node *x, Node *y ) {
    if ( x->parent == NULL ) // x is root
        r = y ; } // set y as root
    else if ( x->parent->left == x )
        x->parent->left = y ;
    else { x->parent->right = y ; }
    if ( y != NULL )
        y->parent = x->parent ;
}
```
Implementation in C

• Now Case 1 can be implemented as follows:

```c
void Delete( Node *x ) {

    /* Case 1: x is a leaf */
    if ( !x->left && !x->right )
        Transplant( x, NULL );

    /* Case 2 and Case 3 */
    ...
}
```
... and Case 2 can be implemented as follows:

```c
void Delete( Node *x ) {
    /* Case 1 */ ... 
    /* Case 2: x has one child */
    else if ( x->left == NULL )
        Transplant( x, x->right ) ;
    else if ( x->right == NULL )
        Transplant( x, x->left ) ;
    /* Case 3 */ ... 
}
```
Implementation in C

• For Case 3, we have two subcases:

```c
void Delete( Node *x ) {
    /* Case 1 and Case 2 */ ...
    else { /* Case 3 : x has two children */
        y = Min( x->right ); // get successor
        if ( y->parent == x ) { // Subcase 3.1
            Transplant( x, y ); y->left = x->left ;
            x->left->parent = y ;
        }
        else { /* Subcase 3.2 */ ... }
    }
}
```
void Delete( Node *x ) {
    else { /* Case 3: x has two children */

        ...
    }
}

else { // Subcase 3.2
    Transplant( y, y->right ) ;
    Transplant( x, y ) ;
    y->right = x->right; y->left = x->left;
    x->right->parent = x->left->parent = y;
    }
}

}
Update Performance

• Let $h$ denote the node-height of the BST

Theorem:
Inserting or deleting a node in a BST can each be performed in $O(h)$ time
Remarks

• The implementation here discusses the core idea, and does not handle the boundary cases well
  Ex: insertion in an empty BST, or deletion resulting an empty BST

• Also, more than one way to implement
  Ex: deletion can be done by swapping with the predecessor, search can be done with while-loop instead of recursion