5.7

Binary Search Trees

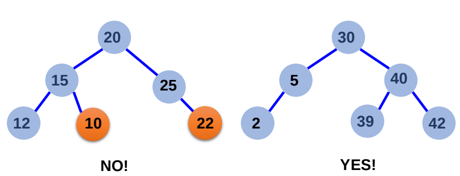
5.7

Binary Search Tree Definition

A ***binary search tree (BST)*** is a binary tree which satisfies the following properties:

- Every element has a ***key*** and no two elements have the same key.
- The keys (if any) in the **left subtree** are **smaller** than the key in the root
- The keys (if any) in the **right subtree** are **larger** than the key in the root
- The left and right subtrees are also BST

BST: Examples



Inorder traversal?

Inorder traversal of a BST will result in a sorted list.

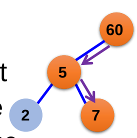
BST: Operations

- Search an element in a BST
- Search for the r^{th} smallest element in a BST
- Insert an element into a BST
- Delete max/min from a BST
- Delete an arbitrary element from a BST

64

BST: Search an Element

1. Search for key 7
2. Start from root
3. Compare the key with root
   - If '<' search the left subtree
   - If '>' search the right subtree
4. Repeat step 3 until the key is found or a leaf is visited



65

BST: Recursive Search Codes

```
template < class K, class E >
pair<K,E>* BST<K,E>::Get(const K& k)
{ // Search the BST for a pair with key k
  // If this pair is found, return a pointer to this
  // pair, otherwise return 0
  return Get(root, k);
}

template < class K, class E >
pair<K,E>* BST<K,E>::Get(TreeNode<pair<K,E>>* p, const K& k)
{
  if(!p) return 0;
  if(k < p->data.key) return Get(p->leftChild, k);
  if(k > p->data.key) return Get(p->rightChild, k);
  return &p->data;
}
```

p->data.key = key
p->data.element = element

66

TreeNode Review

```
template <class T > class Tree; // Forward declaration

template < class T >
Class TreeNode {
friend class Tree <T>;
private:
        T data;
        TreeNode<T>* leftChild;
        TreeNode<T>* rightChild;
};

Template < class K, class E >
Class pair {
Private:
        K key;
        E element;
}
```

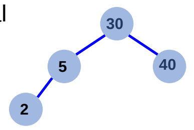
BST: Iterative Search Codes

```
template < class K, class E >
pair<K,E>* BST<K,E>::Get(const K& k)
{
  TreeNode < pair<K, E> > *currentNode = root;
  while (currentNode) {
     if (k < currentNode->data.key)
        currentNode = currentNode->leftChild;
     else if (k > currentNode->data.key)
        currentNode = currentNode->rightChild;
     else return & currentNode->data;
  }
  return NULL; // no match found
}
```

68

BST: Search an Element by Rank

- Definition of **rank**:
  - A ***rank*** of a node is its position in inorder traversal



Inorder traversal : 2 →5 → 30 → 40

Rank : 1 2 3 4

Therefore, the r^{th} smallest element is the node with rank r

BST: Search by Rank, Codes

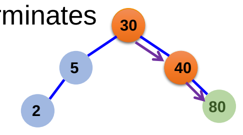
- For each node, we store an additional information "leftSize" which is 1 + (# of nodes in the left subtree)

```
template < class K, class E >
pair<K,E>* BST<K,E>::RankGet(int r)
{ // Search BST for the rth smallest pair
  TreeNode<pair<K,E>>* currentNode = root;
  while(currentNode){
    if(r < currentNode->leftSize)
      currentNode = currentNode->leftChild;
    else if(r > currentNode->leftSize) {
      r -= currentNode->leftSize;
      currentNode = currentNode->rigthChild;
    }
    else return &currentNode->data;
  }
  return 0;
}
```

70

BST: Insert

1. To insert an element with key 80
2. First we search for the existence of the element
3. If the search is unsuccessful, then the element is inserted at the point the search terminates



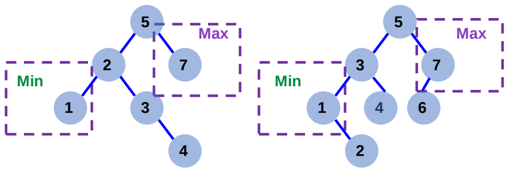
71

BST: Insert Codes

```
template < class K, class E >
void BST<K,E>::Insert(const pair<K,E>& thePair)
{ // Search for key "thePair.key", pp is the parent of p
  TreeNode<pair<K,E>>* p = root, *pp=0;
  while(p){
    pp = p;
    if(thePair.key < p->data.key)
      p = p->leftChild;
    else if(thePair.key > p->data.key)
      p = p->rightChild;
    else // Duplicate, update the value of element
    { p->data.element = thePair.element; return; }
  }
  // Perform the insertion
  p = new pair<K,E>(thePair);
  if(root) // tree is not empty
    if(thePair.key < pp->data.key) pp->leftChild = p;
    else pp->rightChild = p;
  else root = p;
}
```

BST: Min or Max Element

- **Min (Max)** element is at the **leftmost (rightmost)** of the tree



- Min or max are not always terminal nodes
- Min or max has ***at most one child***

73

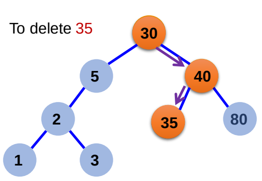
BST: Delete

- To delete an element with key k
- Search for the key k
- If the search is unsuccessful, no need to do anything.
- If the search is successful, we have to deal three scenarios
  1) The element is a **leaf** node
  2) The element is a **non-leaf** node with **one child**
  3) The element is a **non-leaf** node with **two children**

74

BST: Delete

- Scenario 1: the element is a leaf node

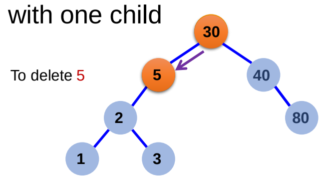


- The child field of parent node is set to NULL
- Dispose the node

75

BST: Delete

- Scenario 2: the element is a non-leaf node with one child

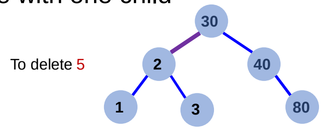


- Simply change the pointer from the parent node (i.e. node with key 30) to the single-child node (i.e. node with key 2)
- Dispose the node

76

BST: Delete

- Scenario 2: the element is a non-leaf node with one child

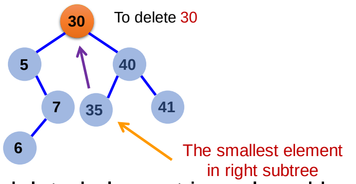


- Simply change the pointer from the parent node (i.e. node with key 30) to the single-child node (i.e. node with key 2)
- Dispose the node

77

BST: Delete

- Scenario 3: the element is a non-leaf node with two children

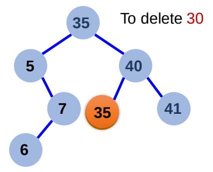


- The deleted element is replaced by either
  - the **smallest** element in **right** subtree or
  - the **largest** element in **left** subtree

78

BST: Delete

- Scenario 3: the element is a non-leaf node with two children

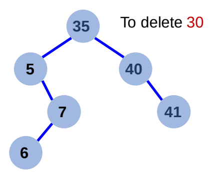


- Delete the node
  - It is a leaf node → apply scenario 1!

79

BST: Delete

- Scenario 3: the element is a non-leaf node with two children

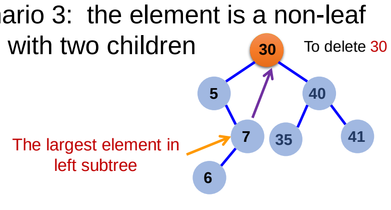


- Delete the node
  - It is a leaf node → apply scenario 1!

80

BST: Delete

- Scenario 3: the element is a non-leaf node with two children

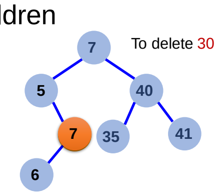


- The deleted element is replaced by either
  - the **smallest** element in **right** subtree or
  - the **largest** element in **left** subtree

81

BST: Delete

- Scenario 3: the element is a non-leaf node with two children

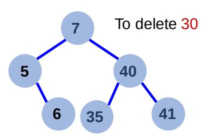


- Delete the node
  - It is a non-leaf node with one child → apply scenario 2!

82

BST: Delete

- Scenario 3: the element is a non-leaf node with two children



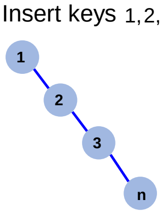
- Delete the node
  - It is a non-leaf node with one child → apply scenario 2!

83

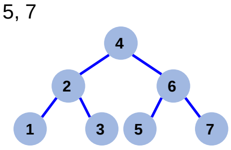
BST: Time Complexity

- Search, insertion, or deletion takes O(h)
- h = Height of a BST

- Worst case $h = n$
  - Insert keys 1, 2, 3, …



- Best case $h = \log n$
  - Insert keys : 4, 2, 6, 1, 3, 5, 7



84

Self-Study Topics

- **Write the pseudo code of BST deletion**
- Selection trees
- AVL trees (Ch. 10)
  - Worst case height : $\boldsymbol{O}(\log \boldsymbol{n})$

85