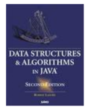
Data Structures & Algorithms 2

Topic 1 – Binary Trees

Lectures

- Same again
  - Mon 9-10am JH1
  - Mon 2-3pm Theatre 2
  - Fri 12-2pm Eolas lab
  - Fri 2-4pm Eolas lab



- Text-book
  - Data Structures and Algorithms by Robert Lafore
- Website
  - Moodle CS211

Content

- 70% end of year exam
- 30% continuous assessment
  - 10 labs worth 2.4% each
  - 1 project worth 6%
- The course involves elaborate abstract algorithms which would take a long time to code from scratch
- Labs will have two components
  - A pen and paper exercise (1%)
  - A programming exercise (1.4%)

Topics

- Bit Manipulation
- Binary Trees
- Quicksort
- Balanced Binary Trees (Red/Black)
- Data Compression (Huffman Encoding)
- Cryptography
- Hashing
- Text Searching Algorithms
- Graph Searching Algorithms (Travelling Salesman Problem)

Why Trees?

- We have seen that linked lists are far more efficient than arrays
- However, they have one big disadvantage
- In order to access a link in the middle, you need to walk through all the links from the beginning or end
- A tree combines the advantages of an ordered array and a linked list

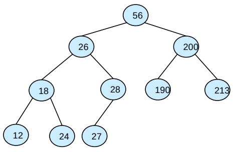
- You can search a tree quickly and you can insert and delete quickly

Evaluation

- Ordered arrays:
  - Advantages: binary search possible
  - Disadvantages: slow insertion, slow deletion
- Linked lists:
  - Advantages: quick insertion, quick deletion (no copying required, just update references)
  - Disadvantages: slow search (average N/2)
- Trees
  - Quick insertion and deletion just like a linked list
  - Quick binary search possible

Tree Terminology

- Trees are made up of nodes just like the links in a linked list – these are objects
- The difference is that nodes contain references to children instead of just the next link
- Each node has exactly one parent

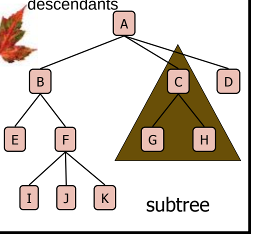


- Each node can have many children, although binary trees have exactly two children
- As well as references to children, each node holds a piece of data, just like a link in a linked list

Tree Terminology

- Root: node without parent (eg. A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, great-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, great-grandchild, etc.

- Subtree: tree consisting of a node and its descendants



More Terms

- **Visiting**
  A node is visited when program control arrives at the node for the purpose of carrying out some operation on the node
- **Traversing**
  To traverse a tree means to visit all the nodes in some specified order
- **Levels**
  The level of a node refers to how many generations the node is from the root (e.g. root is level 0)
- **Key Value**
  The key value is the data field used to search for a node or perform operations on it (this is the value displayed in the circle)

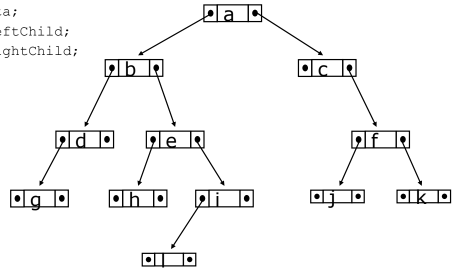
Parts of a binary tree

- A binary tree is composed of zero or more nodes
- Each node contains:
  - A value (some sort of data item)
  - A reference or pointer to a left child (may be null), and
  - A reference or pointer to a right child (may be null)

- A binary tree may be *empty* (contain no nodes)
- If not empty, a binary tree has a root node
  - Every node in the binary tree is reachable from the root node by a *unique* path
- A node with neither a left child nor a right child will be a leaf

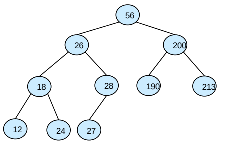
Node code

```
class Node{
  int data;
  Node leftChild;
  Node rightChild;
}
```

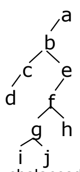


Node arrangement

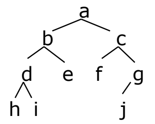
- The nodes in a tree are arranged so as to facilitate quick searching
- The left child of a node always has a lower value
- The right child always has a higher value
- If we want to find an item, we start at the root, and go left or right depending on whether the value is bigger or smaller
- This results in a binary search
  if tree is balanced



Balance



An unbalanced binary tree

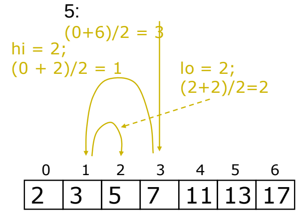


A balanced binary tree

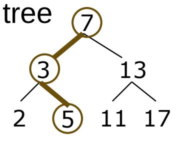
- A binary tree is balanced if every level above the lowest is “full” (contains 2^n nodes)
- In most applications, a reasonably balanced binary tree is desirable
- A linked list is a completely unbalanced tree – each node only has one child
- Trees can become unbalanced because of the order to which nodes are added

Binary search in an array

Searching for 5:

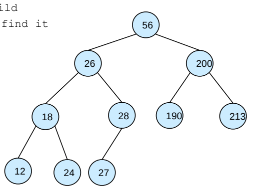


Using a binary search tree



Code to find a node

```
public Node find (int key){
   Node current = root;  //start at the root
   while(current.data != key){ //while no match
         if(key < current.data)  //go left?
                  current=current.leftChild;
         else
                  current=current.rightChild //or go right?
         if(current == null) // if no child
                  return null; // didn’t find it
         }
   }
   return current;
}
```

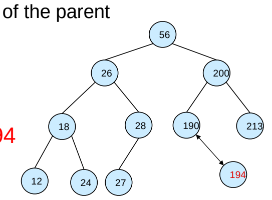


If tree is balanced, efficiency is O(logn) as there are $\log_2 n$ levels

Inserting a node

- Same idea – pretend to search for the node you’re trying to insert by going left and right
- When you find where it should exist you will reach a null
- Create a new node, set its parent reference
- Update the child reference of the parent

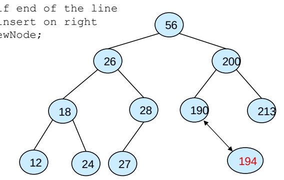
inserting 194



Code for inserting nodes Part I

```
public void insert(int id)
      {
      Node newNode = new Node();    // make new node
      newNode.data = id;           // insert data
      if(root==null)                // no node in root
         root = newNode;
      else{                          // root occupied

         Node current = root;       // start at root
         Node parent;
         while(true)                // (exits internally)

                    .........
```

```
   while(true) {                 // (exits internally)

      parent = current;
      if(id < current.data)  // go left?
         {
         current = current.leftChild;
         if(current == null)  // if end of the line,
            {                 // insert on left
            parent.leftChild = newNode;
            return;
            }
         }  // end if go left
      else                    // or go right?
         {
         current = current.rightChild;
         if(current == null)  // if end of the line
            {                 // insert on right
            parent.rightChild = newNode;
            return;
            }
         }  // end else go right
      }  // end while
   }  // end else not root
}  // end insert()
```

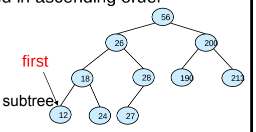


Tree traversal

- Traversing a tree means to visit each node in a specified order (i.e. printing out the entire contents of the tree)
- Since a binary tree has three “parts,” there are six possible ways to traverse the binary tree:
  - root, left, right-**PREORDER** — root, right, left
  - left, root, right-**INORDER** — right, root, left → these are reverse order
  - left, right, root-**POSTORDER** — right, left, root
- The simplest way to carry out a traversal is to use a recursive method
- We will look at three orders – preorder, inorder and postorder

InOrder Traversal

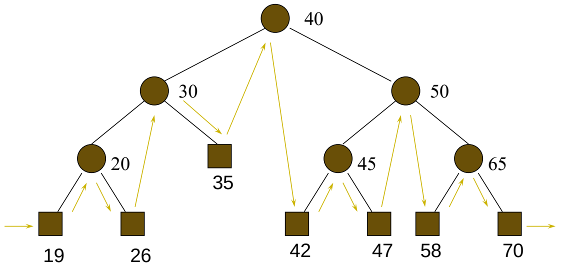
- Inorder traversal is the most sensible traversal of the tree
- It causes all the nodes to be visited in ascending order based on their key values
- Recursive method is as follows



1. Call itself to traverse the node’s left subtree
2. Visit the node
3. Call itself to traverse the node’s right subtree

- Visiting a node can mean displaying the value, writing it to a file or whatever

InOrder

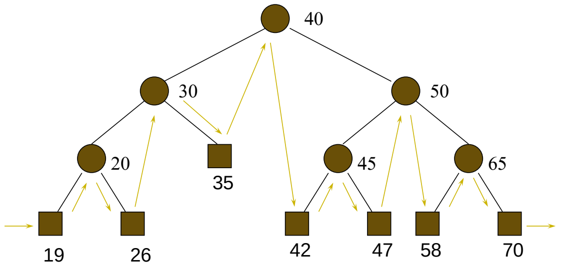


InOrder Traversal

```
......
inOrder(root);
......

private void inOrder(Node localRoot) {
        if(localRoot != null) {
            inOrder(localRoot.leftChild);
            System.out.print(localRoot.iData + " ");
            inOrder(localRoot.rightChild);
        }
}
```

Method Call

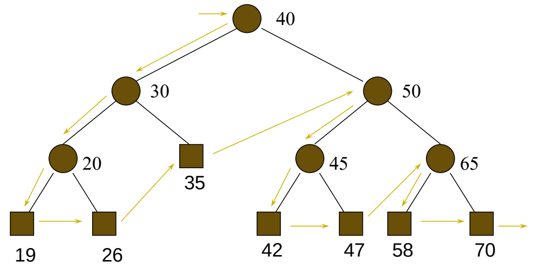
InOrder



PreOrder

- PreOrder traversal is used when you want to store a tree in a file
- Printing out the elements in this order will allow you to recreate the exact same tree at a later date
- Remember, the node is visited first, followed by its left subtree and then its right subtree
- This is effectively the same order in which elements were added to the tree

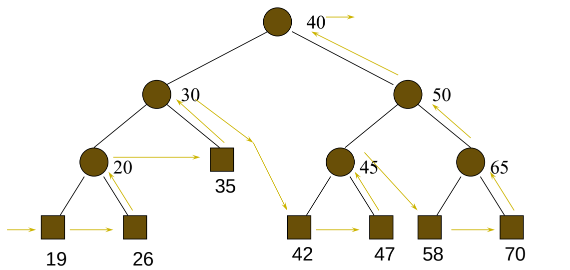
PreOrder



PreOrder Code

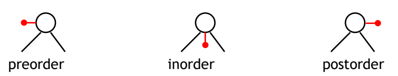
```
private void preOrder(Node localRoot) {

      if(localRoot != null) {

         System.out.print(localRoot.iData + " ");
         preOrder(localRoot.leftChild);
         preOrder(localRoot.rightChild);
      }
}
```

PostOrder

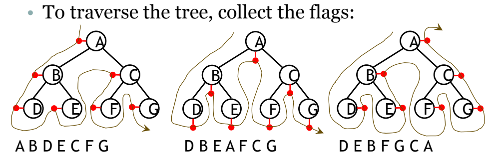


Tree traversals using "flags"

- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a "flag" attached to each node, as follows:

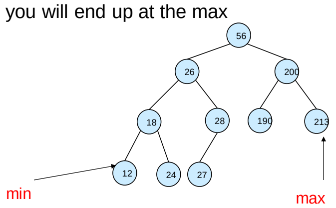


- To traverse the tree, collect the flags:



Max and Min

- This is easy!
- The left child is always smaller than the right child
- Keep going left and you will end up at the min
- Keep going right and you will end up at the max



Code for Min

```
public Node minimum(){
  Node current, last;
  current = root;
  while(current!=null){
      last=current;
      current=current.leftChild;
  }
  return last;
}
```

Deleting a Node

- This is tricky – learn it off carefully!
- Start by finding the node you want to delete
- Check if it is a root – special case
- Then there are three cases to consider:
  1. The node to be deleted is a leaf
  2. The node to be deleted has one child
  3. The node to be deleted has two children

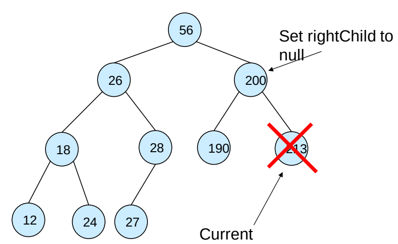
The finding part...

```
public boolean delete(int key){ // delete node with given key
                                // (assumes non-empty list)
     Node current = root;
     Node parent = root;
     boolean isLeftChild = true;

     while(current.iData != key){         // search for node

        parent = current;
        if(key < current.iData){          // go left?

           isLeftChild = true;
           current = current.leftChild;
        }
        else{                             // or go right?

           isLeftChild = false;
           current = current.rightChild;
        }
        if(current == null)              // end of the line,
           return false;                 // didn't find it
     }  // end while
     // found node to delete…continued
```

Case I: Leaf Node

- To delete a leaf node, simply change the appropriate child field in the node's parent to point to *null,* instead of to the node.
- The node still exists, but is no longer a part of the tree.
- Because of Java's garbage collection feature, the node need not be deleted explicitly.

Deleting a leaf

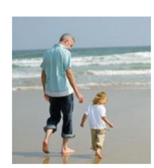


Code to delete leaf

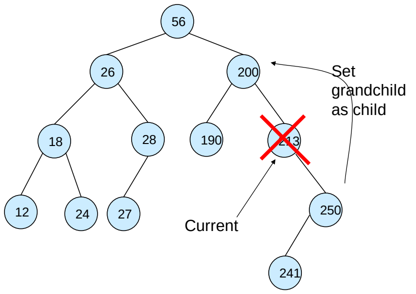
```
// if no children, simply delete it
      if(current.leftChild==null &&
  current.rightChild==null){

         if(current == root)              // if root,
            root = null;                  // tree is empty
         else if(isLeftChild)
            parent.leftChild = null;      // disconnect
         else                             // from parent
            parent.rightChild = null;
      }
```

Case II: One Child

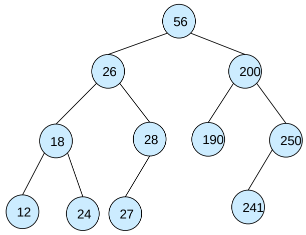
- The node to be deleted in this case has only two connections: to its parent and to its only child
- Connect the child of the node to the node's parent, thus cutting off the connection between the node and its child, and between the node and its parent
- Visualise – parent deleted, grandparent takes on child



Delete node with one child



Delete node with one child



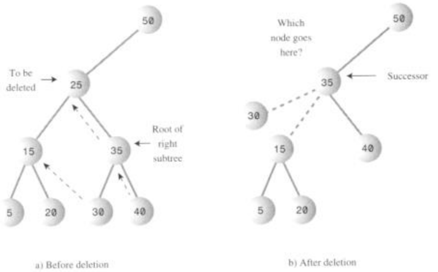
Delete node with One Child

```
// if no right child, replace with left subtree
      else if(current.rightChild==null)
         if(current == root)
            root = current.leftChild;
         else if(isLeftChild)
            parent.leftChild = current.leftChild;
         else
            parent.rightChild = current.leftChild;

// if no left child, replace with right subtree
      else if(current.leftChild==null)
         if(current == root)
            root = current.rightChild;
         else if(isLeftChild)
            parent.leftChild = current.rightChild;
         else
            parent.rightChild = current.rightChild;
```

Case III: Two Children

- This is the trickiest!
- In order to save hassle some people create a whole new tree and put all the remaining elements into it
- To delete a node with two children, replace the node with its inorder successor
- For each node, the node with the next-highest key (to the deleted node) in the subtree is called its inorder successor
- To find the successor,
  - start with the original (deleted) node's right child.
  - Then go to this node's left child and then to its left child and so on, following down the path of left children.
  - The last left child in this path is the successor of the original node.

How *not* to do it!!

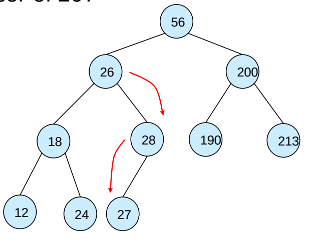


a) Before deletion

b) After deletion

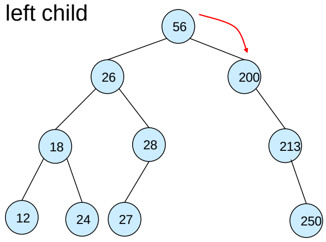
The Successor Node

- The next highest node in the tree
- What is the successor of 26?
- Swap this with 26

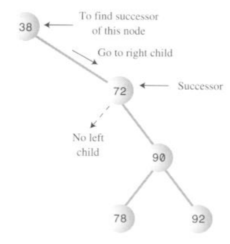


Also

- What is the successor of 56?
- Notice 200 has no left child



Find successor



The right child is the successor.

Code for Successor

```
private Node getSuccessor(Node delNode) {

      Node parent = delNode;
      Node current = delNode.rightChild;    // go to right child
      while(current != null) {               // until no more
                                             // left children,
         parent = current;
         current = current.leftChild;       // go to left child
      }
                                            // if successor not
      return parent;
}
```

Swapping Successor

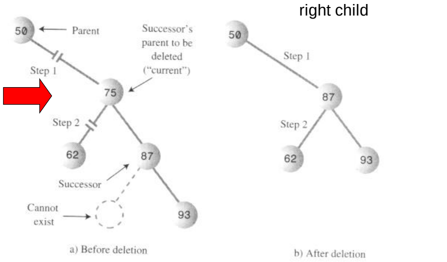
- Again, there are three possibilities, each involve swapping successor with node to be deleted

Successor can either be the right child of node to be deleted or a descendant of that node
Successor can only have one right child, no left child

1. If successor is rightChild of delNode, do swap and successor takes delNode's left child and keeps its right child
2. If successor is not rightChild of delNode, it takes the left and right children of delNode
3. If successor wasn't a leaf one more step – successor's parent takes its right child as its left child

Delete a node with 2 children (successor is right child)

Successor keeps its right child



Code

- There are two main steps:

```
1. parent.(left/right)Child = successor;
2. successor.leftChild = current.leftChild;
```

- We will see that when the successor is a left descendent rather than the right child of the node to be deleted, there are two additional steps
- These are included in the getSuccessor method rather than the delete method

Successor is rightChild

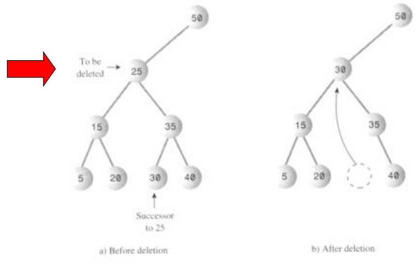
```
    else{  // two children, so replace with inorder
successor
       // get successor of node to delete (current)
       Node successor = getSuccessor(current);

       // connect parent of current to successor instead
       if(current == root)
          root = successor;
       else if(isLeftChild)
          parent.leftChild = successor;
       else
          parent.rightChild = successor;

       // connect successor to current's left child
       successor.leftChild = current.leftChild;
    }  // end else two children
    // (successor cannot have a left child)
```

Successor is left descendant

- If successor is left descendant, then it has to take on both children of the deleted node

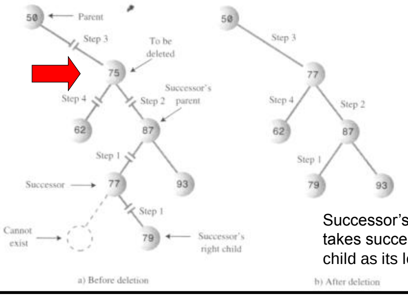
1. Successor takes right child of delNode
2. Successor takes left child of delNode
3. Update delNode's parent to point to successor
4. If successor has a right child, successor's parent takes it as its left child

Delete a node with 2 children (successor is left descendant with no right child)



Successor is leaf – takes on left and right child of delNode

Delete a node with 2 children (successor is left descendent with right child)



Successor's parent takes successor's right child as its left child

Code has four steps

```
1.  successorParent.leftChild =
    succcessor.rightChild;
2.  successor.rightChild = delNode.rightChild;
3.  parent.(left/right)Child = successor;
4.  successor.leftChild = current.leftChild;
```

The first two steps are the additional ones and can be carried out at the end of the getSuccessor method so that we don't need to add anything to the delete method :

```
if(successor != delNode.rightChild){  // right child,
                                      // make connections
       successorParent.leftChild =
    successor.rightChild;
       successor.rightChild = delNode.rightChild;
}
```

See tree.java to see how this is all put together!

Updated Code for Successor

```
private Node getSuccessor(Node delNode) {

    Node successorParent = delNode;
    Node successor = delNode;
    Node current = delNode.rightChild;    // go to right child
    while(current != null) {                // until no more
                                           // left children,
       successorParent = successor;
       successor = current;
       current = current.leftChild;        // go to left child
    }
                                             // if successor not
    if(successor != delNode.rightChild) {  // right child,
                                           // make connections
       successorParent.leftChild = successor.rightChild;
       successor.rightChild = delNode.rightChild;
    }
    return successor;
}
```

Is deletion necessary?

- Deletion is tricky!
- Sometimes programmers avoid it by simply adding a boolean `isDeleted` to node class
- To delete this node simply set the value to true
- Structure of the tree doesn't change but tree can fill up with deleted nodes
- Can be OK if there are few deletions or where you want to keep records forever

Efficiency of Binary Trees

- If we let the number of levels of the tree by L and the number of nodes be N, then in a a balanced tree
  - $N = 2^L – 1$
  - or $L = \log_2(N + 1)$
- The time taken to perform insert, find or delete is proportional to the number of levels
- Therefore all operations on a balanced tree are O(logn)
- Also, no copies required, references just need to be updated

Balanced Tree