EE2008
Data Structures and Algorithms

Tay Wee Peng

wptay@ntu.edu.sg

COPYRIGHT STATEMENT

All course materials, including but not limited to, lecture slides, handout and recordings, are for your own educational purposes only. All the contents of the materials are protected by copyright, trademark or other forms of proprietary rights.

All rights, title and interest in the materials are owned by, licensed to or controlled by the University, unless otherwise expressly stated. The materials shall not be uploaded, reproduced, distributed, republished or transmitted in any form or by any means, in whole or in part, without written approval from the University.

You are also not allowed to take any photograph, film, audio record or other means of capturing images or voice of any contents during lecture(s) and/or tutorial(s) and reproduce, distribute and/or transmit any form or by any means, in whole or in part, without the written permission from the University.

Appropriate action(s) will be taken against you including but not limited to disciplinary proceeding and/or legal action if you are found to have committed any of the above or infringed the University's copyright.

Trees

What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
    - How to efficiently search a file with specific names/types?

Trees: Terminology

Tree Terminology

- **Root:** node without parent (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, grand-grandparent, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (e.g, 3)
- **Degree**: The number of children of a node
- **Subtree**: tree consisting of a node and its descendants

✓ This is a general rooted tree

Binary Trees

- A binary tree is a rooted tree in which each node has either no children, one child, or two children. --- Binary
  - If a node has one child, that child is designated as either a left child or a right child (but not both).
  - If a node has two children, one child is designated a left child and the other a right child.

✓ When a binary tree is drawn, a left child is drawn to the left and a right child is drawn to the right.

Binary Tree: Examples

Height = 2

Height = 4

- ✓ **Height:** Max number of ancestors

Recursion and recursive

```
factorial( n )=n*(n-1)*…*2*1
```

n=5

```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

n=4

```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

n=3

```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

n=2

```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

n=1

```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

1 2 3 4 5 6 7 8 9

return 24

return 6

return 2

return 120

Binary Tree: Definition

- A **binary tree T** is a structure defined on a finite set of nodes that

Binary Tree:

1. Either contains no nodes, <u>or</u>
2. Is composed of **three** disjoint sets of nodes: a root node, its left subtree and its right subtree.
3. Left and right **subtrees are binary trees** (recursive definition)

✓ The binary tree is defined **recursively**.

Understanding the recursive nature of trees

✓ Subtree is a tree.

✓ All parts of a tree are considered as trees

Structure of a node

T is a pointer pointing to a node in a binary tree, i.e., T is a variable containing the memory address of this node.

Depending on the application, storing T.parent is optional.

T.left.parent == T

T.left.data

T.left.left

T.left.right

Binary Tree Traversals

- **To traverse a binary tree is to visit each node in the tree in some *prescribed order*.**
  - **Depending on the application, "visit" may be interpreted in various ways.**
  - **For example, if we want to print the data in each node in a binary tree, we would interpret "visit" as "print the data."**
- **The three most common traversal orders are *preorder, inorder*, and *postorder*.**
  - **Each is most easily defined recursively.**

Preorder traversal: Definition

- ***Preorder traversal*** **of a binary tree rooted at *root* is defined by the rules:**

1. **If *root* is empty, stop.**
2. **Visit *root.***
3. **Execute *preorder* on the binary tree rooted at the left child of the root.**
4. **Execute *preorder* on the binary tree rooted at the right child of the root.**

✓ **In a preorder traversal, a node is visited *before* its descendants**

✓ **In short, root—left— right**

Preorder traversal: Example

Beginning with *root* = *1*, we visit 1

We then execute preorder on the subtree rooted at 2 (i.e., the subtree rooted at 1's left child)

The order of visitation of this subtree is 2 (root), (no left subtree), 4 (right subtree). The overall order of visitation so far is 1, 2, 4.

Since we have finished executing preorder on 1's left subtree, we next execute preorder on 1's right subtree

The order of visitation for this subtree is 3, 5, 6, 7.

Therefore, the order of visitation for the tree
1, 2,4, 3, 5, 6, 7.

Algorithm Preorder

- **This algorithm performs a preorder traversal of the binary tree with root *root*.**

1. If *root* is empty, stop.
2. Visit *root.*
3. Execute *preorder* on the binary tree rooted at the left child of the root.
4. Execute *preorder* on the binary tree rooted at the right child of the root.

pseudo code

```
preorder(root)     {
    if (root != null) {
            visit root;
            preorder(root.left);
            preorder(root.right);
    }
}
```

Counting Nodes in a Binary Tree

- **"Visit node" means different operations for different problems.**

- **Example: as an application of the preorder algorithm, sum up all the values stored at the nodes.**
  - **visit node: sum + node.data**

Inorder Traversal: Definition

- **Inorder traversal of a binary tree rooted at *root* is defined by the rules:**

1. **If *root* is empty, stop.**
2. **Execute *inorder* on the binary tree rooted at the left child of the root.**
3. **Visit *root.***
4. **Execute *inorder* on the binary tree rooted at the right child of the root.**

✓ **In short, left—root— right**

Inorder Traversal: Example

- Inorder visits the nodes of the binary tree in the order

```
inorder(root)     {
    if (root != null) {
        inorder(root.left);
        visit root;
        inorder(root.right);
    }
}
```

✓ In order of left—root— right

Inorder Traversal By Projection (Squishing)

- An easy way …

Inorder traversal: 2 4 1 6 5 7 3

Postorder traversal: Definition

- ***Postorder traversal*** **of a binary tree rooted at *root* is defined by the rules**

1. **If *root* is empty, stop.**
2. **Execute *postorder* on the binary tree rooted at the left child of the root.**
3. **Execute *postorder* on the binary tree rooted at the right child of the root.**
4. **Visit *root*.**

✓ **In short, left—right—root**

Postorder Traversal: Example

- **Postorder visits the nodes of the binary tree in the order**

```
postorder(root)     {
    if (root != null) {
        postorder(root.left);
        postorder(root.right);
        visit root;
    }
}
```

✓ In short, left—right—root

Binary Search Trees

What is a Binary Search Tree

Suppose that you have to sort a group of people by height so that you can easily search for someone by their height later on.

How would you go about doing this efficiently?

What is a Binary Search Tree

Pick a midpoint (say, 165cm) and look at the first person in line. If s/he is below that height, you move him/her to the left, otherwise to the right.

This partition method is employed by the binary search tree data structure

What is a Binary Search Tree

Binary search trees: Example

4, 2, 6, 5, 1, 3, 7

- ✓ **Building a BST, starting from value 4**

- ✓ **at most 4**
- ✓ **greater than 4**

- ✓ **"Binary search tree property": a node's left child must have a data less than or equal to its parent, and a node's right child must have a data greater than its parent**

Insertion and deletion

- **The operations of insertion and deletion cause a binary search tree to change**
- **This change is done in such a way that the binary-search-tree property continues to hold.**

BST Insertion

Node Representation

Set up a node to be inserted to tree

```
// set up node that contains value val to be added to tree

temp = new node;
temp.data = val
temp.left = temp.right = null
```

BST Insertion

❑ *BSTinsert:*

1. If *root* is empty, return *temp* as the root of the BST, stop.
2. Execute *BSTinsert_recurs* procedure on tree root
   - 2.1 if *temp.data<=root.data* and left child of root is empty, add *temp* as the left child of root
   - 2.2 if *temp.data>root.data* and right child of root is empty, add temp as the right child of root
   - 2.3 if *temp.data<=root.data* and left child of root is not empty, apply *BSTinsert_recurs* in its left subtree.
   - 2.4 if *temp.data>root.data* and right child of root is not empty, apply *BSTinsert_recurs* in its right subtree.

5 Insertion Cases

Case 1: ***Insert a node to an empty tree***

Case 2: ***Insert to a non-empty tree, left subtree empty***

Case 3: ***Insert to a non-empty tree, right subtree empty***

Case 4: ***Insert to a non-empty tree, left subtree non-empty***

Case 5: ***Insert to a non-empty tree, right subtree non-empty***

Case 1: Insert a node to an empty tree

✓ Since the tree is empty, this newly inserted node becomes the root

```
BSTinsert (root, val) {
   // set up node to be added to tree
   temp = new node;
   temp.data = val
   temp.left = temp.right = null

   // special case: empty tree
   if (root = = null)
      return temp;
   else  // for all other cases
     BSTinsert_recurs (root, temp);
   return root;
}
```

Case 1: *Insert a node to an empty tree*
Case 2: *Insert to a non-empty tree, left subtree empty*
Case 3: *Insert to a non-empty tree, righ subtree empty*
Case 4: *Insert to a non-empty tree, left subtree non-empty*
Case 5: *Insert to a non-empty tree, right subtree non-empty*

Case 2: Insert to a non-empty tree (into left subtree)

1) Inserted data is less than root and
2) left subtree is empty

13 < 15

```
BSTinsert_recurs (root, temp) {

  if (temp.data ≤ root.data) {
    if (root.left = = null)
      root.left = temp;
  }

}
```

Case 1: *Insert a node to an empty tree*
Case 2: *Insert to a non-empty tree, left subtree empty*
Case 3: *Insert to a non-empty tree, righ subtree empty*
Case 4: *Insert to a non-empty tree, left subtree non-empty*
Case 5: *Insert to a non-empty tree, right subtree non-empty*

Case 3: Insert to a non-empty tree (into right subtree)

1) Inserted data is larger than root and
2) right subtree is empty

✓ Similar to the case for left subtree

```
BSTinsert_recurs (root, temp) {
  if (temp.data ≤ root.data) {  1
    // insert at left
  }
  else {     // goes to right

    if (root.right = = null)   2
      root.right = temp;   3

  }
}
```

Case 1: *Insert a node to an empty tree*
Case 2: *Insert to a non-empty tree, left subtree empty*
Case 3: *Insert to a non-empty tree, righ subtree empty*
Case 4: *Insert to a non-empty tree, left subtree non-empty*
Case 5: *Insert to a non-empty tree, right subtree non-empty*

Case 4: Insert to a non-empty tree, left subtree non-empty

1) Inserted data is less than root and
2) left subtree is non-empty

```
BSTinsert_recurs (root, temp) {

  if (temp.data ≤ root.data) {        1
    if (root.left = = null)
      root.left = temp

    else   // non-empty              2
      BSTinsert_recurs (root.left, temp);
  }
  else {     // goes to right
  }
}
```

Case 1: *Insert a node to an empty tree*
Case 2: *Insert to a non-empty tree, left subtree empty*
Case 3: *Insert to a non-empty tree, righ subtree empty*
Case 4: *Insert to a non-empty tree, left subtree non-empty*
Case 5: *Insert to a non-empty tree, right subtree non-empty*

Case 5: Insert to a non-empty tree, right subtree non-empty

1) Inserted data is larger than root and
2) right subtree is non-empty

```
BSTinsert_recurs (root, temp) {
  if (temp.data ≤ root.data) {
     // insert at left
  }
  else {  // goes to right
    if (root.right = = null)
      root.right = temp
    else
      BSTinsert_recurs (root.right, temp)
  }
}
```

Case 1: *Insert a node to an empty tree*
Case 2: *Insert to a non-empty tree, left subtree empty*
Case 3: *Insert to a non-empty tree, righ subtree empty*
Case 4: *Insert to a non-empty tree, left subtree non-empty*
Case 5: *Insert to a non-empty tree, right subtree non-empty*

An exercise

Write a general algorithm that inserts a node into a binary search tree (BST).

Hint:

Put all the code for all five cases previously discussed together.

BST Insertion Algorithm: Put all together

```
BSTinsert (root, val) {
   // set up node to be added to tree
   temp = new node;
   temp.data = val
   temp.left = temp.right = null

   // special case: empty tree
   if (root = = null)
     return temp;
   else
     // for all other cases
     BSTinsert_recurs (root, temp);
   return root;
}
```

```
BSTinsert_recurs (root, temp) {
  if (temp.data ≤ root.data) {
    if (root.left = = null)
      root.left = temp
    else
      BSTinsert_recurs (root.left, temp);
  }
  else {  // goes to right
    if (root.right = = null)
      root.right = temp
    else
      BSTinsert_recurs (root.right, temp)
  }
}
```

BST Deletion

Node Representation

BST Deletion

- ***BSTreplace(root, ref)*: (for the case *ref* has one or no child)**

1. **If *ref* is *root*, set the child of *ref* as the root, stop.**
2. **If *ref* is neither *root* nor *leaf*, set the child of *ref* as *ref.parent*'s child, stop.**
3. **If *ref* is *leaf*, set *ref.parent*'s child NULL, stop.**

BST Deletion

- ***BSTreplace(root, ref)*: (for the case *ref* has one or no child)**

1. **If *ref* is *root*, set the child of *ref* as the root, stop.**
2. **If *ref* is neither *root* nor *leaf*, set the child of *ref* as *ref.parent*'s child, stop.**
3. **If *ref* is *leaf*, set *ref.parent*'s child NULL, stop.**

BST Deletion

- ***BSTreplace(root, ref)*: (for the case *ref* has one or no child)**

1. **If *ref* is *root*, set the child of *ref* as the root, stop.**
2. **If *ref* is neither *root* nor *leaf*, set the child of *ref* as *ref.parent*'s child, stop.**
3. **If *ref* is *leaf*, set *ref.parent*'s child NULL, stop.**

Deletion Example: only one child (Delete root)

- ✓ First for the case of right child

```
if (ref.left = = null)                1
        child = ref.right
else child = ref.left

if (ref == root) {
    if (child ≠ null)                 2
        child.parent = null,          3
    return child;
}
```

- ✓ Keep track its child 1
- ✓ modify its parent child.parent with null 3

Deletion Example: only one child (Delete root)

- ✓ Similarly for the case of left child

```
if (ref.left = = null)
        child = ref.right;
else child = ref.left;   ❶

if (ref == root) {
    if (child ≠ null) ❷
        child.parent = null; ❸
    return child
}
```

- ✓ Keep track its child
- ✓ modify its parent child.parent with null

Deletion Example: only one child (not root)

```
if (ref.left == null)
   child = ref.right;  1
else child = ref.left;

// is ref left child
if (ref.parent.left == ref )
   ref.parent.left = child;
else  2
   ref.parent.right = child;

if (child ≠ null)   3
   child.parent = ref.parent;
return root;
```

- ✓ Keep track its child
- ✓ modify its parent child.parent

Deletion Example: only one child

- ✓ Tide up

- ✓ That is, if ref has only one child, we replace it by its children

Deletion Example: only one child (not root)

```
child.parent = ref.parent;
```

```
if (ref.left == null)
   child = ref.right;
else child = ref.left;  ❶

// is ref left child
if (ref.parent.left == ref )
   ref.parent.left = child;  ❷
else
   ref.parent.right = child;

if (child ≠ null)      ❸
   child.parent = ref.parent;
return root;
```

- ✓ Keep track its child
- ✓ modify its parent child.parent

Deletion Example: only one child

- ✓ Tide up

- ✓ That is, if ref has only one child, we replace it by its children

Deletion Example: no children (Delete *leaf*)

- ✓ If *ref* has no children, modify its parent ref.parent with null as its child (which means, *ref* has a null child)

- ✓ *ref* is the right child of its parent

```
if (ref.left == null)
    child = ref.right;  1
else child = ref.left;

// is ref left child
if (ref.parent.left == ref )
    ref.parent.left = child;
else
    ref.parent.right = child;  2

if (child ≠ null)
    child.parent = ref.parent;
return root;
```

Deletion Example: no children (Delete *leaf*)

- ✓ If *ref* has no children, modify its parent ref.parent with null as its child (which means, *ref* has a null child)

- ✓ *ref* is the left child of its parent

```
if (ref.left == null)
   child = ref.right;
else child = ref.left;

// is ref left child
if (ref.parent.left == ref )
   ref.parent.left = child;
else
   ref.parent.right = child;

if (child ≠ null)
   child.parent = ref.parent;
return root;
```

Put all together-- Deletion: No child or only one child

```
BSTreplace(root, ref) {// ref has only one child
   // set child to ref's child, or null if no child
   if (ref.left == null)
       child = ref.right
   else child = ref.left

   if (ref == root) {// delete root
       if (child ≠ null)
          child.parent = null
       return child
   }

   if (ref.parent.left == ref ) // is ref left child
       ref.parent.left = child
   else
       ref.parent.right = child

   if (child ≠ null)
       child.parent = ref.parent
   return root
}
```

Deletion Example: Two children

Deletion Example: Two children

Code for step 1:

```
succ = ref.right
while (succ.left ≠ null)
    succ = succ.left
```

1

The smaller data is on left subtree, we search for mini Until *succ* has no left child

1. We locate the node *succ* containing the mini data, 6, in its right subtree, (*succ* must have no left child)

Deletion Example: Two children

Code for step 2:

2 `ref.data = succ.data`

2. **Replace the value of the node to be deleted (5) by the minimum value *succ* (6).**

Deletion Example: Two children

Code for step 3:

```
// delete succ
   return BSTreplace (root, succ)
```

✓ As we discussed early, **BSTreplace** can delete a node in various cases

3. Delete *succ* (*succ* must have no left child) *(we have already discussed on how to delete)*

Put all together: Deletion Algorithm

```
BSTdelete(root, ref) {
   // if zero or one children, use Algorithm BSTreplace
   if (ref.left == null || ref.right == null)
      return BSTreplace (root, ref)

   //find node succ containing a minimum data item
   //   in ref's right subtree
   succ = ref.right                                  ❶
   while (succ.left ≠ null)
      succ = succ.left

   // "move" succ to ref, thus deleting ref          ❷
   ref.data = succ.data

   // delete succ                ❸
   return BSTreplace (root, succ)
}
```

AVL Tree

Unbalanced Tree

- Suppose we use BSTInsert to insert the data values 3, 2, 1 into an empty BST:

- Same as a linked list!
- Worst case height: O(n) – inefficient for searching

Example

- We can easily fix this

- Tree is now balanced!

AVL Trees

- Named after Adelson-Velskii and Landis
- Define height of a tree as the number of edges on the longest path from the root to a leaf.
  - An empty tree has height –1
  - A tree with a single node has height 0
- A BST is said to be AVL balanced if the difference in the heights between the left and right sub-trees of any node is at most 1.

AVL trees with 1, 2, 3, and 4 nodes:

Here is a larger AVL tree (42 nodes):

AVL Trees

The root node is AVL-balanced:

- **Both sub-trees are of height 4:**

AVL Trees

All other nodes (*e.g.*, AF and BL) are AVL balanced

- The sub-trees differ in height by at most one

Maintaining Balance

If a tree is AVL balanced, for an insertion to cause an imbalance:

- **The heights of the sub-trees must differ by 1**
- **The insertion must increase the height of the deeper sub-tree by 1**

Maintaining Balance

Suppose we insert 23 into our initial tree

Maintaining Balance

The heights of each of the sub-trees from here to the root are increased by one

Maintaining Balance

However, only two of the nodes are unbalanced: 17 and 36

Maintaining Balance

However, only two of the nodes are unbalanced: 17 and 36

- **We only have to fix the imbalance at the lowest node**

Maintaining Balance

We can promote 23 to where 17 is, and make 17 the left child of 23

Maintaining Balance: Case Left-Left

Consider the following setup

- Each blue triangle represents a tree of height h

Maintaining Balance: Case Left-Left

Insert *a* into this tree: it falls into the left subtree B_L of *b*

- **Assume B_L remains balanced**
- **Thus, the tree rooted at *b* is also balanced**

Maintaining Balance: Case Left-Left

The tree rooted at node *f* is now unbalanced

- We will correct the imbalance at this node

Maintaining Balance: Case Left-Left

Here are examples of when the insertion of 7 may cause this situation when $h = -1$, 0, and 1

Maintaining Balance: Case Left-Left

Node *f is the lowest unbalanced node*. Suppose its address is given in `ref_f`.

We will modify these three pointers:

Maintaining Balance: Case Left-Left

```
ref_b = ref_f.left
ref_BR = ref_b.right
```

```
ref_b.right = ref_f
```

Maintaining Balance: Case Left-Left

```
ref_p = ref_f.parent
ref_p.left = ref_b
ref_f.left = ref_BR
ref_f.parent = ref_b
```

Maintaining Balance: Case Left-Left

The nodes *b* and *f* are now balanced and all remaining nodes of the subtrees are in their correct positions

Maintaining Balance: Case Left-Left

Additionally, height of the tree rooted at *b* equals the original height of the tree rooted at *f*

- **Thus, this insertion will no longer affect the balance of any ancestors all the way back to the root**

Maintaining Balance: Case Left-Left

In our example case, the correction

Maintaining Balance: Case Left-Left

Maintaining Balance: Case Left-Right

Alternatively, consider the insertion of c where $b < c < f$ into our original tree

Maintaining Balance: Case Left-Right

Assume that the insertion of *c* increases the height of B_R

- **Once again, *f* becomes unbalanced**

Maintaining Balance: Case Left-Right

Here are examples of when the insertion of 14 may cause this situation when $h = -1$, 0, and 1

Maintaining Balance: Case Left-Right

Unfortunately, the previous correction does not fix the imbalance at the root of this sub-tree: the new root, *b*, remains unbalanced

Maintaining Balance: Case Left-Right

In our three sample cases with $h = -1$, 0, and 1, doing the same thing as before results in a tree that is still unbalanced…

- **The imbalance is just shifted to the other side**

Maintaining Balance: Case Left-Right

Consider tree rooted at *d* with two subtrees of height $h - 1$

Maintaining Balance: Case Left-Right

Now an insertion causes an imbalance at *f*

- The addition of either *c* or *e* will cause this

Maintaining Balance: Case Left-Right

Given ref_f, we will reassign the following pointers

```
ref_p = ref_f.parent
ref_b = ref_f.left
ref_d = ref_b.right
ref_DL = ref_d.left
ref_DR = ref_d.right
```

Maintaining Balance: Case Left-Right

```
ref_d.left = ref_b
ref_d.right = ref_f
```

Maintaining Balance: Case Left-Right

```
ref_p.left = ref_d
ref_b.right = ref_DL
ref_f.left = ref_DR
```

Maintaining Balance: Case Left-Right

Now the tree rooted at d is balanced

Maintaining Balance: Case Left-Right

Again, the height of the root did not change

Maintaining Balance: Case Left-Right

Before **After**

Maintaining balance: Summary

Case Left-Left: promotes first intermediate node b to root

Maintaining balance: Summary

Case Left-Right: promotes second intermediate node d to root

Maintaining balance: Summary

There are two symmetric cases to those we have examined:

- **Insertions into the right-right sub-tree: similar to left-left**

- **Insertions into either the right-left sub-tree: similar to left-right**

Insertion Example

Insertion Example I

Insert 73

Insertion Example I

The node 81 is unbalanced

- A left-left imbalance

Insertion Example I

The node 81 is unbalanced

- A left-left imbalance
- Promote the intermediate node 75 to the imbalanced node

Insertion Example I

The node 81 is unbalanced

- A left-left imbalance
- Promote the intermediate node 75 to the imbalanced node

Insertion Example I

The tree is AVL balanced

Insertion Example II

Insert 77

Insertion Example II

The node 87 is unbalanced

- A left-right imbalance

Insertion Example II

The node 87 is unbalanced

- A left-right imbalance
- Promote the intermediate node 81 to the imbalanced node

Insertion Example II

The node 87 is unbalanced

- A left-right imbalance
- Promote the intermediate node 81 to the imbalanced node

Insertion Example II

The tree is balanced

Insertion Example III

Insert 76

Insertion Example III

The node 78 is unbalanced

- A left-left imbalance

Insertion Example III

The node 78 is unbalanced

- **Promote 77**

Insertion Example III

Again, balanced

Insertion Example IV

Insert 80

Insertion Example IV

The node 69 is unbalanced

- A right-left imbalance
- Promote the intermediate node 75 to the imbalanced node

Insertion Example IV

The node 69 is unbalanced

- A left-right imbalance
- Promote the intermediate node 75 to the imbalanced node

Insertion Example IV

Again, balanced

Insertion Example V

Insert 74

Insertion Example V

The node 72 is unbalanced

- A right-right imbalance
- Promote the intermediate node 73 to the imbalanced node

Insertion Example V

The node 72 is unbalanced

- A right-right imbalance
- Promote the intermediate node to the imbalanced node

Insertion Example V

Again, balanced

Insertion Example VI

Insert 64

Insertion Example VI

This causes no imbalances

Insertion Example VII

Insert 55

Insertion Example VII

The node 69 is imbalanced

- A left-left imbalance
- Promote the intermediate node 63 to the imbalanced node

Insertion Example VII

The node 69 is imbalanced

- A left-left imbalance
- Promote the intermediate node 63 to the imbalanced node

Insertion Example VII

The tree is now balanced

Insertion Example VIII

Insert 70

Insertion Example VIII

The root node is now imbalanced

- A right-left imbalance
- Promote the intermediate node 63 to the root

Insertion Example VIII

The root node is imbalanced

- A right-left imbalance
- Promote the intermediate node 63 to the root

Insertion Example VIII

The result is AVL balanced

Sorting

- Merge and mergesort
- Heaps and heapsort
- Partition and quicksort
- Order statistics
- Sorting in linear time

Sorting

- Input: sequence of elements $a_1, a_2, \ldots, a_n$ (e.g. numbers, strings, dates) that can be sorted
- Output: Permutation $a'_1, a'_2, \ldots, a'_n$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- Why study sorting?
  - Applications: program renders graphical objects that are layered on top of each other needs to sort the objects according to an "above" relationship so that it can draw these objects in order.
  - Many other applications: sorting values in a database, sorting dates etc.
  - Wide variety of sorting algorithms – teach us many important techniques in algorithm design.

Sorting

Algorithm	Worst case complexity	Average case complexity
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$

Although quicksort has worst case complexity more than the other two algorithms, it is often the best practical choice for sorting because of its remarkable efficiency *on average* (the constant factor hidden in O-notation is actually very small).

Merge and Mergesort

- Merging two arrays
- Application in mergesort

Problem Solving Idea

- Want to sort a given array.
- Intuition: it should be easier to sort a smaller array compared to a larger one.
- Divide and conquer:
  - divide the array into two smaller arrays
  - sort each of the smaller arrays (assuming this can be done!)
  - find a way to combine the two sorted arrays together to form one single sorted array
- Recursion to the rescue!
  - to sort the smaller arrays, repeat the same procedure
  - till you reach a single element array – easy to sort!

Merge Problem

- **Input:**
  - **Two sorted arrays, of *nearly equal lengths***
- **Output**
  - **One sorted array.**

Merging Two Sorted Arrays

We begin by examining the first element in each array

✓ Since 10 < 12 and each array is sorted, 10 is smallest, so copied to output

✓ copy smaller value to A

```
if (L[i] ≤ R[j]) {
    A[r] = L[i];
    i = i + 1;
}
else {
    A[r] = R[j];
    j = j + 1;
}
r = r +1;
```

Merging Two Sorted Arrays

We move to the next item in the first array

```
if (L[i] ≤ R[j]) {
    A[r] = L[i];
    i = i + 1;
}
else {
    A[r] = R[j];
    j = j + 1;
}
r = r +1;
```

✓ copy smaller value to A

Merging Two Sorted Arrays

We move to the next item in the second array

✓ copy smaller value to A

```
if (L[i] ≤ R[j]) {
    A[r] = L[i];
    i = i + 1;
}
else {
    A[r] = R[j];
    j = j + 1;
}
r = r +1;
```

Merging Two Sorted Arrays

We move to the next item in the second array

✓ copy smaller value to A

```
if (L[i] ≤ R[j]) {
    A[r] = L[i];
    i = i + 1;
}
else {
    A[r] = R[j];
    j = j + 1;
}
r = r +1;
```

Merging Two Sorted Arrays

```
while (i ≤ a and j ≤ b) {


}
```

✓ We continue as long as $i \le a$ and $j \le b$

```
if (L[i] ≤ R[j]) {
    A[r] = L[i];
    i = i + 1;
}
else {
    A[r] = R[j];
    j = j + 1;
}
r = r +1;
```

✓ copy smaller value to *A*

Merging Two Sorted Arrays

```
while (i ≤ a and j ≤ b) {

}
```

✓ We continue as long as $i \le a$ and $j \le b$

```
if (L[i] ≤ R[j]) {
    A[r] = L[i];
    i = i + 1;
}
else {
    A[r] = R[j];
    j = j + 1;
}
r = r +1;
```

✓ copy smaller value to *A*

Merging Two Sorted Arrays

```
while (i ≤ a and j ≤ b) {


}
```

✓ We continue as long as $i \le a$ and $j \le b$

```
if (L[i] ≤ R[j]) {
    A[r] = L[i];
    i = i + 1;
}
else {
    A[r] = R[j];
    j = j + 1;
}
r = r +1;
```

✓ copy smaller value to *A*

Merging Two Sorted Arrays

Copy the remainder

- ✓ all of the data from the first array have been copied to the output array; we conclude by copying the remainder of the second array to the output array
- ✓ What is the condition: $i > a$ or $j > b$

✓ copy the remainder to A

```
while (j ≤ b) {
    A[r] = R[j];
    j = j + 1;
    r = r + 1;
}
```

Summary: Algorithm Merge

- **Given an array A[p..n].**
- **Divide and copied into two subarrays L[1..a] = A[p..q] and R[1..b] = A[q+1…n].**
- **Each sorted in non-decreasing order.**
- **These two non-decreasing subarrays are merged into a single sorted array that replaces A[p..n].**

Put all together: Algorithm Merge

```
merge (A,p,q,n) {
   a = q - p + 1 // number of elements in L subarray
   b = n - q     // number of elements in R subarray


   // Copy subarray A[p..q] to L and A[q+1..n] to R
   for (k = 1 to a)
      L[k] = A[p + k - 1]
   for (k = 1 to b)
      R[k] = A[q + k]

   i = 1
   j = 1
   r = p
```

… Algorithm Merge

```
while (i ≤ a and j ≤ b) {
   // copy smaller value to A
   if (L[i] ≤ R[j]) {
      A[r] = L[i];
      i = i + 1;
   }
   else {
      A[r] = R[j];
      j = j + 1;
   }
   r = r +1;
}

// copy the remainder
```

... Algorithm Merge

```
    ...
    // copy remainder, if any, of first sub-array to A
    while (i ≤ a) {
        A[r] = L[i]
        i = i + 1
        r = r + 1
    }
    // copy remainder, if any, of second sub-array to A
    while (j ≤ b) {
        A[r] = R[j]
        j = j + 1
        r = r + 1
    }

}
```

Mergesort

- **Mergesort : Divide and Conquer paradigm**
  - **If the array has single element, stop**
  - **Divide the array to be sorted into two *nearly* equal parts.**
  - **Conquer: Each part is then sorted using mergesort.**
  - **The two sorted subarrays are then merged into one sorted array**

Mergesort

(first + last) /2
first
last
Divide
Sort recursively
by Mergesort
Sort recursively
by Mergesort
Conquer
sorted
sorted
Merge
Combine
sorted

Mergesort : Example

Sort each part recursively by Mergesort

Mergesort

1) If the array has single element, stop
2) Divide the array to be sorted into two *nearly* equal parts.
3) Each part is then sorted using mergesort.
4) The two sorted halves are then merged into one sorted array

Mergesort : Example

Mergesort

1) If the array has single element, stop
2) Divide the array to be sorted into two *nearly* equal parts.
3) Each part is then sorted using mergesort.
4) The two sorted halves are then merged into one sorted array

Algorithm Mergesort

This algorithm sorts the array *A[i], ... , A[j]* in nondecreasing order. It uses the merge algorithm

```
mergesort (A,i,j) {
   // if only one element, just return
   if (i == j)
      return
   // divide A into two nearly equal parts
   m = (i + j)/2
   // mergesort each half - recursive calls
   mergesort(A,i,m)
   mergesort(A,m + 1,j)
   // merge the two sorted halves
   merge(A,i,m,j)
}
```

Mergesort : Example

Mergesort: Complexity

- Let *T(n)* be the number of operations needed for an input array of size n.
- Divide into 2 subarrays of about size n/2. Recursively apply mergesort on each. Merge the two sorted subarrays.
- Each subarray takes *T(n/2)* operations.
- Merge: $\Theta(n)$ operations
- $T(n) = 2\,T(n/2) + \Theta(n)$
- Exercise: show that $T(n) = \Theta(n \log n)$

Learning Takeaway

- **Divide and Conquer makes problems easier!**
  - **keep dividing till you reach a very simple problem**
  - **need to find a way to combine the solutions of the sub-problems together**

- **Can often be used with recursion**
  - **allows code reuse**
  - **makes coding very simple**

Heaps and Heapsort

Binary Heap: Example

Level *i* has 2^i *nodes*

- ✓ All levels, except the last level, i.e. level 3, have as many nodes as possible.
- ✓ On the last level, there is at most one node with one child, which must be a left child
- ✓ No vacancies at levels 0 to h-1

- ✓ Levels 0 (the root), 1, and 2 (the next-to-last level) have as many nodes as possible.
- ✓ On level 3 (the last level), there is at most one node with no sibling, which must be a left child --- also called left-complete binary tree

Defining a heap structure

- **A *heap structure* is a binary tree in which all levels, except possibly the last (bottom) level, have as many nodes as possible. On the last level, all of the leaves are at the left.**
  - ***This is a Heap Structural Property***

- ✓ **A heap must be a *nearly complete* binary tree.**
- ✓ **levels *0, 1, 2, …, h-1* (except the last level) have the maximum number of the nodes possible (i.e., level *i* has 2^i node for $0 \le i \le h-1$**
- ✓ **The lowest level is filled from the left up to a point**

A binary minheap: Definition

- **A binary *minheap* is a heap structure in which values are assigned to the nodes so that the value of each node is less than or equal to the values of its children (if any).**
  - ***This is a Heap Order Property***

Minheap property:
value of each node ≤ value of its children

✓ **A minheap must maintain two properties: structural and order property**

A binary maxheap: Definition

- A binary *maxheap* is a heap structure in which values are assigned to the nodes so that the value of each node is greater than or equal to the values of its children (if any)

Maxheap property:

value of each node ≥ value of its children

Representing a heap using an array

- To represent a heap, we store the value in node number *i* (of heap) in cell *i* of an array

A	104	71	24	66	27	23	8	5	32	25	18	22
i	1	2	3	4	5	6	7	8	9	10	11	12

```
parent(i){
   return ⌊i/2⌋
}

left(i){
   return 2*i
}

right(i){
   return 2*i + 1
}
```

A.last : number of elements in the array

A.heapsize : number of elements in heap stored in array A

Maxheap is "weakly sorted"

- ✓ In a maxheap, the **maximum value is at the root.**

- ✓ For example, the values 104, 71, 66, 5, along the path from the root to the terminal node with value 5, are in **non-increasing order**.

- ✓ For example, the values 5, 32, 25, 18, 22 on level 3 are in neither non-increasing nor non-decreasing order.

"Weakly sorted"

- **A maxheap is "weakly sorted" in the sense that the values along a path from the root to a terminal node are in non-increasing order.**

- **At the same time, the values along a *level* are, in general, in no particular order**

Maintaining a heap: siftdown

- **Assume:**
  - **A is an array [*1* ..*n*] representing a heap structure.**
  - **But the order property does not hold at node *i*, i.e., A[i] might be smaller than its children.**
  - **For node *i*, the left subtree and right subtree are maxheaps.**

Algorithm siftdown: Example

- **Idea: repeatedly swap a value with its larger child.**

Repeatedly Swapping (Siftdown)

1. Find larger child
2. Swap

Repeatedly Swapping (Siftdown)

Repeatedly Swapping (Siftdown)

When swapping repeat ends?

1) Node has no child (2*i>n)
2) Node is already larger than children

Siftdown : Recursive Implementation

```
siftdown(A,i) {
    n = A.heapsize
    largest = i
    left = 2 * i
    // 2 * i ≤ n tests for a left child
    if (left ≤ n && A[left] > A[i]) {
        largest = left
    }
    right = 2 * i + 1
    // tests for a right child
    // if right child has larger value, update largest
    if (right ≤ n && A[right] > A[largest]) {
        largest = right
    }
    if (largest ≠ i) {
        swap(A, i, largest)
        siftdown(A,largest) // recursively perform siftdown
    }
}
```

Siftdown : Another Implementation

```
siftdown(A,i) {
    n = A.heapsize
    // 2 * i ≤ n tests for a left child
    while (2 * i ≤ n) {
        child = 2 * i
        // if there is a right child and it is
        // bigger than the left child, update child
        if (child < n && A[child + 1] > A[child]) {
            child = child + 1
        }
        // need to swap?
        if (A[child] > A[i]) {
            swap(A, i, child);
        } else {  // done, exit
            break // if node is already larger, exit while loop
        }
        i = child
    }
}
```

✓ As long as a node has the child, repeat …

✓ *child* always refers to the larger child

✓ Swapping

✓ Consider next

Siftdown : Complexity

- Suppose the subtree at node *i* has n nodes.
- Takes $\Theta(1)$ time to determine which of *i*, *left(i)* and *right(i)* is largest, and perform swapping.
- Call siftdown on one of the children's subtree.
- Worst case: bottom level of tree is exactly half full, i.e., the two children's subtrees differ most in number of nodes.
- Size of child subtree in worst case: $\lfloor 2n/3 \rfloor$
- $T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$
- Exercise: show that $T(n) = O(\log n)$

Siftdown : Complexity

```
siftdown(A,i) {
    n = A.heapsize
    largest = i
    left = 2 * i
    // 2 * i ≤ n tests for a left child
    if (left ≤ n && A[left] > A[i]) {
        largest = left
    }
    right = 2 * i + 1
    // tests for a right child
    // if right child has larger value, update largest
    if (right ≤ n && A[right] > A[largest]) {
        largest = right
    }
    if (largest ≠ i) {
        swap (A, i, largest)
        siftdown(A,largest) // recursively perform siftdown
    }
}
```

$\Theta(1)$

$\leq T(2n/3)$

Summary: Algorithm Siftdown

- The array *A* represents a heap structure indexed from *1* to *n*. The left and right subtrees of node *i* are maxheaps. After *siftdown(A, i)* is called, the tree rooted at *i* is a maxheap.

- ✓ Initially, the left and right subtrees of node *i* are maxheaps (a).
- ✓ After siftdown is called, the tree rooted at *i* is a heap (b).

Making a maxheap: idea

- Given an array, how to turn it into a maxheap?
- Idea: why not make use of siftdown?
- But siftdown only works for array that looks like

Making a maxheap: idea

- Idea: apply siftdown starting near the leaves:

Making a maxheap: idea

- Idea: apply siftdown starting near the leaves:

Can apply
siftdown again!

Making a maxheap: Heapify

- The problem of organizing the data into a maxheap or minheap is called *heapify*

A	3	6	10	18	8	7	25	400
i	1	2	3	4	5	6	7	8

n

```
for i = n/2 downto 1
      siftdown(A,i)
```

Making a maxheap: Heapify

```
for i = n/2 downto ... .
      siftdown(A,i)
```

Making a maxheap: Heapify

```
for i = n/2 downto ... .
        siftdown(A,i)
```

When index 1, it terminates

Summary Algorithm Heapify

- We state the algorithm to make a maxheap/minheap as **heapify( )**.
  - This algorithm rearranges the data in the array *A*, indexed from *1* to *n*, so that it represents a heap.

```
heapify(A) {
   A.heapsize = A.last
   n = A.heapsize
   // n/2 is the index of the parent of the last node
   for i = n/2 downto 1
      siftdown(A,i)
}
```

  - Complexity: $O(n \log n)$ - not tight
  - A better bound can be shown: $O(n)$

Challenge:
Try showing this!

Deleting root from a maxheap: idea

- ✓ The root that contains the largest value, 104, is to be deleted
- ✓ Idea, we move the value at the bottom level, farthest right, 22, to the root

- ✓ After deleting the root, we have to recover the structure

Deleting root from a maxheap: idea

- ✓ **Idea, we move the value at the bottom level, farthest right, 22, to the root**

Deleting root from a maxheap: idea

Code

```
A[1] = A[n]                          ❶
A.heapsize = A.heapsize - 1          ❷
```

✓ **Idea**, we move the value at the bottom level, farthest right, 22, to the root

Deleting root from a maxheap: idea

- ✓ Need to recover to a maxheap
- ✓ siftdown ❸

Summary: Algorithm Delete the root from a maxheap

- This algorithm deletes the root (the item with largest value) from a heap containing n elements.

```
heap_delete_root(A) {
  // move the item with the
largest index to root
   A[1] = A[A.heapsize]
   A.heapsize = A.heapsize - 1
 // maintain a heap
   siftdown(A,1)
}
```

	1	2	3	4	5	6	7	8	9	10	11	12
A	104	71	24	66	27	23	8	5	32	25	18	22

	1	2	3	4	5	6	7	8	9	10	11	12
A	22	71	24	66	27	23	8	5	32	25	18	

Deleting any node from a maxheap

```
If A[i/2] ≥ A[i]
   siftdown(A,i)
else
   siftup(A,i)
```

Insert a value into the maxheap

- ✓ **Where and how to Insert the value 50 into the heap**

Insert a value into the maxheap

❑ Idea:

- **When inserting a value into the maxheap, we first insert it in the bottom level, farthest left, then repeatedly move the parent down, maintaining the maxheap structure and property,**

✓ **Where and how to Insert the value 50 into the heap**

Inserts the value val into a maxheap

- ✓ The value, 50, is inserted in the bottom level, farthest left. (If the bottom level were full, we would begin another level at the left.)

Inserts the value val into a maxheap

- ✓ The value, 50, is inserted in the bottom level, farthest left. (If the bottom level were full, we would begin another level at the left.)

Inserts the value val into a maxheap

✓ algorithm terminates.

Summary: Algorithm Insert into the maxheap

- The algorithm **repeatedly** move the parent down, maintaining the heap structure and property

```
heap_insert(val,A) {
   // a new space at bottom level, farthest left
   A.heapsize = A.heapsize + 1
   i = A.heapsize
   // i is the child and i/2 is the parent.
   // If i > 1, i is not the root.
   while (i > 1 and val > A[i/2]) {
      A[i] = A[i/2]  // move parent down
      i = i/2        // next level up
   }
   A[i] = val
}
```

"siftup"

Heapsort

- **Recall: mergesort has good worst case complexity $\Theta(n \log n)$, but requires auxiliary arrays to be created.**
- **We now look at another sorting algorithm: heapsort**
  - **same worst case complexity $\Theta(n \log n)$**
  - **can be done in place**

Heapsort: idea

A							
3	6	10	18	8	7	25	40

Make a maxheap:

heapify(A)

A							
40	18	25	6	8	7	10	3

Heapsort: idea

- Exchange A[1] with A[n]
- Treat A[1..n-1] as new heap.
- Repeat procedure.

Similar to deleting root of A

Heapsort: idea

- Exchange A[1] with A[n]
- Treat A[1..n-1] as new heap.
- Repeat procedure.

Similar to deleting root of A

Heapsort

```
heapsort(A) {
  // change A into a maxheap
  heapify(A)
  for (i = A.last downto 2) {
    // shift ith largest element
    // to its correct place
    swap(A,1,i)
    // reduce size of heap
    A.heapsize = A.heapsize - 1
    // maintain maxheap
    siftdown(A,1)
  }
}
```

$O(n)$

$O(1)$

$O(n \log n)$

$O(\log n)$

Total: $O(n \log n)$

Learning Takeaway

- **Heaps are very useful data structures.**
  - **Get "weakly sorted" array.**

- **Main operations have low complexity:**
  - **siftdown** $O(log\ n)$
  - **heapify** $O(n)$
  - **delete and insert** $O(log\ n)$

- **Applications: heapsort, priority queues, …**

Partition and Quicksort

Partition

- Partition divides an array into two parts and sizes of the two parts can range from nearly equal to highly unequal.
  - Unlike mergesort, which divides the array into two nearly equal parts.
- The division depends on a particular element, called the partition element, that is chosen.

✓ The partition element is also called a pivot

What is Algorithm Partition

- ✓ This algorithm partitions the array *A*[i..j] by inserting *val* = *A*[*i*] at the index *h* where it would be if the array was sorted.
- ✓ When the algorithm concludes, values at indexes less than *h* are less than *val*, and values at indexes greater than *h* are greater than or equal to *val*.
- ✓ The algorithm returns the index *h*.

Partition: Main Idea

- **Output:**

- **At each iteration, maintain this structure:**

- **If A[k] ≥ x,**

- **If A[k] < x,**

Partition: Summary

- Continue till end of array

Partition: Idea

✓ Scan for the smaller

✓ Make room for the smaller one

✓ Swap

Idea:

scan from left for element smaller than *val* = *A*[*i*]

Increase *h (make room for smaller)*

exchange

repeat until the last element of the array

```
val=A[i]; h=i;

for k = i + 1 to j

    if (A[k] < val) {

        h = h + 1;

        swap(A,h,k);

    }
```

Partition: Idea

✓ *Scan for the smaller*

Idea:

scan from left for element smaller than *val* = *A*[*i*]

Increase *h (make room for smaller)*

exchange

repeat until the last element of the array

```
val=A[i]; h=i;

for k = i + 1 to j

   if (A[k] < val) {

      h = h + 1;

      swap(A,h,k);

   }
```

Partition: Idea

✓ *Scan for the smaller*

Idea:

scan from left for element smaller than *val* = *A*[*i*]

Increase *h (make room for smaller)*

exchange

repeat until the last element of the array

```
val=A[i]; h=i;

for k = i + 1 to j

    if (A[k] < val) {

        h = h + 1;

        swap(A,h,k);

    }
```

Partition: Idea

✓ *Scan for the smaller*

Idea:

scan from left for element smaller than *val* = *A*[*i*]

Increase *h (make room for smaller)*

exchange

repeat until the last element of the array

```
val=A[i]; h=i;

for k = i + 1 to j

    if (A[k] < val) {

        h = h + 1;

        swap(A,h,k);

    }
```

Partition: Idea

✓ *Scan for the smaller*

Idea:

scan from left for element smaller than *val* = *A*[*i*]

Increase *h (make room for smaller)*

exchange

repeat until the last element of the array

```
val=A[i]; h=i;

for k = i + 1 to j

    if (A[k] < val) {

        h = h + 1;

        swap(A,h,k);

    }
```

Partition: Idea

✓ *Swap*

Idea:

scan from left for element smaller than *val* = *A*[*i*]

Increase *h (make room for smaller)*

exchange

repeat until the last element of the array

```
val=A[i]; h=i;

for k = i + 1 to j

   if (A[k] < val) {

      h = h + 1;

      swap(A,h,k);

   }

swap (A, i, h)

      return h
```

Partition Algorithm

Idea:

scan from left for element smaller than *val* = *A*[*i*]

Increase *h (make room for smaller)*

exchange

repeat until the last element of the array

```
partition(A,i,j) {
    val = A[i];
    h = i
    for k = i + 1 to j
        if (A[k] < val) {
            h = h + 1
            swap(A, h, k)
        }
    swap (A, i, h)
    return h
}
```

✓ scan

✓ For smaller

✓ swap

Complexity: $\Theta(n)$

Quicksort: idea

- **The quicksort, like mergesort, closely follows the divide-and-conquer paradigm**

1. ***Divide:*** **Partition (rearrange) the array *A*[*p..r*] into two (possibly empty) subarray *A*[*p..h-1*] and *A*[*h+1..r*] such that each element of *A*[*p..h-1*] is less than *A*[*h*], which is in turn less than each element of *A*[*h+1..r*]. This partition procedure returns the index *h***
2. ***Conquer:*** **Sort the two subarrays *A*[*p..h-1*] and *A*[*h+1..r*] by recursive calls to quicksort.**
3. ***Combine:*** **Since the subarrays are sorted in place, no work is needed to combine them: the entire array *A*[*p..r*] is now sorted**

Quicksort

This algorithm sorts the array *A[p .. r]* by using the partition algorithm

```
Input Parameters: A,p,r
Output Parameters: A

quicksort(A,p,r) {
   if (p < r) {
      h = partition(A,p,r)        ❶
      quicksort(A,p,h - 1)        ❷
      quicksort(A,h + 1,r)        ❸
   }
}
```

Quicksort

quicksort
12 30 21 8 6 9 1 7
partition
quicksort
7 8 6 9 1
12
21 30
partition
1 6
7
9 8
21 30
quicksort
1 6 7 8 9
1 6 7 8 9
12
21 30

Quicksort Performance

- **Worst-case: partition produces one subarray with *n-1* elements and one with *0* elements.**
- **This happens when the array is already sorted!**

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \Theta(\mathrm{n}) \\
&= T(n-1) + \Theta(n) \\
&= \Theta(n^2)
\end{aligned}
$$

Backward substitution

- **Worst than heapsort and mergesort.**

Quicksort Performance

- **Best-case: partition produces two subarrays with approximately $n/2$ elements each.**

$$T(n) = 2T(n/2) + \Theta(\mathrm{n})$$
$$= \Theta(n \log n)$$

- **Same as heapsort and mergesort.**
- **It turns out that if given an array whose elements have been randomly permutated, then *on average*, the performance quicksort is $\Theta(n \log n)$.**
- **It has been observed in practice that quicksort is usually faster than heapsort/mergesort.**

Quicksort Performance

- How to avoid the worst-case performance of $\Theta(n^2)$?
- Try to make sure that the input array at every recursive call is not already sorted!

```
rand_partition(A,p,r) {
    i = random(p,r) // chooses a random index between p and r
    swap(A, p, i)  // swap the element at random i into p
    return partition(A,p,r) // run the partition algorithm
}
```

```
rand_quicksort(A,p,r) {
    if (p < r) {
        h = rand_partition(A,p,r)
        rand_quicksort(A,p,h - 1)
        rand_quicksort(A,h + 1,r)
    }
}
```

There is still a small probability you will get $\Theta(n^2)$ performance, but this probability becomes very small when array is long.

Learning Takeaway

- **Partition places a pivot element in the correct place in the array if it is sorted. Worst-case complexity: $\Theta(n)$**

- **Partition is a useful procedure with applications in:**
  - **Quicksort: Divide and Conquer using partition**
  - **Order statistics: we will see this later.**

Comparison Sorting

- Comparisons between two elements to output a sorted sequence. All the sorting methods we have studied so far are comparison sorts.
- The best complexity we have found is $O(n \log n)$.
- Can there exist some method that performs better?

Comparison Sorting

- Run an algorithm on input $a_1, a_2, \ldots, a_n$
- Decision tree: shows how a particular algorithm is executed

1:2 means a com

Algorithm runs till a leaf node is reached – that is the output

- All comparison sort algorithms can be described using a decision tree.

Comparison Sorting

Theorem. Any comparison sort algorithm requires $\Omega(\mathrm{n} \log \mathrm{n})$ comparisons in the worst case.

Proof.

The number of comparisons in the worst case is the height h of the decision tree. Each of the $n!$ permutations of the input sequence must appear at some leaf. A binary tree of height h has at most 2^h leaves, so we have

$$n! \leq 2^h$$

$$h \geq log(n!) = \Omega(n\, log\, n)$$

- Mergesort and heapsort are asymptotically optimal comparison sorts – we cannot do better!
- Quicksort is asymptotically optimal on average.
- This theorem is for comparison sorts. If the input sequence has some special properties (e.g., integers), there are $O(n)$ sorting algorithms like counting sort.

Selection and Order Statistics

Selection in Context

- **There are a number of applications in which we are interested in identifying a single element of its rank relative to the ordering of the entire set**
- **Example**
  - **The minimum and maximum elements**

✓ **In general, queries that ask for an element with a given rank are called "order statistics"**

Definition

- The *k-th* order statistic of a set of *n* elements is its *k-th* smallest element

```
Example:

A={11, 10, 14, 4, 2, 1, 5, 7}
```

✓ If they are sorted {1, 2, 4, 5, 7, 10, 11, 14}

```
1-st order statistic = minimum(A) = 1
```

```
3-rd order statistic = 4
```

```
n-th order statistic = maximum(A) = 14
```

Selecting the minimum and maximum

- Find *1-st* order statistic (minimum)

Idea:

✓ *Examine each element of the array in turn and keep track of the smallest element seen so far.*

Selecting the minimum and maximum

- **Find *1-st* order statistic (minimum)**
  - ***n-1* comparisons**

$n = A.last$

		i				n
A	4	12	30	16	5	15

```
minimum (A) {
   min = A[1]      // start with 1st, assume it smallest
   for i = 2 to A.last
      if (A[i] < min )      // smaller value found
         min = A[i]         // smallest so far
   return min
 }
```

✓ ***Examine each element of the array in turn and keep track of the smallest element seen so far.***

✓ **Finding the maximum is similar with *n-1* comparison.**

Median

- A median, informally, is the "half way point" of the set if the set is sorted
  - When *n* is odd, the median is unique, occurring at

$$i = (n+1) / 2$$

  - When *n* is even, there are two medians, occurring at

$$i = n / 2 \quad \text{and} \quad i = n / 2 + 1$$

✓ Median is the element such that half of the other elements are smaller and the remaining half are larger

Median Example

Consider the annual incomes listed below:

260,750 25,160 72,815 30,570 137,230 55,300 77,800

If sorted …

What is the median?

4

25,160	30,570	55,300	72,815	77,800	137,230	260,750

The median income is the income that would be in the middle if the incomes were sorted --- the median income is $ 72,815.

The Selection problem

- **The problem of selecting the *k-th* order statistic from a set of *n* distinct number**
- **The selection problem is: Given an array *A* and an integer *k*, find the *k-th* smallest element.**

Input: a set *A* of *n* (distinct) numbers and an integer $1 \le k \le n$

Output: the element x in *A* that is larger than $k - 1$ other elements of *A*

Selection Problem

- **Since the selection problem does *not* require that the array be sorted, the goal is to solve the problem faster by doing less work than sorting the entire array.**
  - **Using partition**

✓ **Note that we do not sort the entire array**

Recall: The property of partition algorithm

- The partition algorithm returns the rank of the partition element.

✓Partition element

17	21	5	23	9	37	15	3	11	25	31	13	29	7	19
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

partition

7	5	9	15	3	11	13	17	37	25	31	23	29	21	19
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

✓Partition element 17 is rank 8 if the array is sorted

Selecting by Partition: Example

Suppose that we want to find the *10th* smallest element in the array (i.e., rank 10)

17	21	5	23	9	37	15	3	11	25	31	13	29	7	19
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Using 17 as partition element

Partition

After partition, we got rank 8

7	5	9	15	3	11	13	17	37	25	31	23	29	21	19
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Since 8 < 10, we next call partition on the part of the array to the right of 17

37	25	31	23	29	21	19
9	10	11	12	13	14	15

Partition

Selecting by Partition: Example

✓37 is rank 15

✓19 is rank 9

✓Since rank 9 < 10, we next partition on the part of the array to the right of 19

✓25 is rank 12

Selecting by Partition: Example

Since 10 < 12, we next call partition on the part of the array to the left of 25:

21	23
10	11

The partition element 21 is placed at index 10. The algorithm thus terminates, having found 21 as the *10th* smallest element in the array

Selecting by Partition: Algorithm

- Uses the partition algorithm to find the ith smallest element of an array A[p..r].

```
select(A, p, r, i) {
    if (p == r) {  // if array has only 1 element
        return A[p]
    }
    q = partition(A, p, r) // get the position of the pivot bet. p and r
    k = q - p + 1 // rank relative to p
    if (i == k) {
        return A[q]
    } else if(i < k) {
        return select(A, p, q-1, i)  // search in first part
    } else {
        return select(A, q+1, r, i-k) // search in second part
    }
}
```

- Can you show that select has $O(n^2)$ complexity?
- See supplementary notes for a linear complexity select, which is more complicated.

Learning Takeaway

- **Partition allows us to perform Divide and Conquer.**
  - **discard the subarray that is not relevant**
  - **use recursion to repeat same procedure on the relevant subarray**

- **Tip: For problems that involves the rank or order of the elements in an array, may consider using partition to split the array, and then recursion to solve.**

Sorting in Linear Time

Counting Sort

- **A[1..n] input elements are integers from 1 to k.**
- **How to sort the array in linear time?**
- **Idea: if j elements are $\leq$ x, then x can be put in the j-th position of the array.**
- **How to find the number of elements $\leq$ A[i] *efficiently*?**

Counting Sort

- Initialize another array count[1..k]
- Go through A[1...n]
  - increase count[A[i]] by 1 ← this can be done because A[i] is a positive integer
- count[i] now contains the number of elements equal to i
- To get the number of elements $\leq$ i : cumulative sum

```
for i = 1 to k
    count[i] = count[i-1] + count[i]
```

- Use count array to sort A[1..n] by placing each A[i] in the right location given by count[A[i]].

Counting Sort Example

Counting Sort Example

A

2	5	3	1	2	3	1	3

B

1	2	3	4	5	6	7	8
1	1	2	2	3	3	3	5

count

1	2	3	4	5
2	4	6	7	8

cumulative sum

This is an important property for many applications. E.g., used in radix sort later.

We start from the back of A so that numbers with the same value appear in B in the same order that they appear in A. This sorting is said to be *stable*.

counting_sort pseudo code

```
A[1..n] - input array of n integers in the range 1 to k
B[1..n] - output array containing the sorted elements
          from A

counting_sort(A,B,k) {
    n = A.last
    for i = 1 to k {
        count[i] = 0
    }
    for j = 1 to n {
        count[A[j]] = count[A[j]] + 1
    } // count[i] now contains the no. of elements = i
    for i = 1 to k {
        count[i] = count[i-1] + count[i]
    } // count[i] now contains no. of elements ≤ i
    for j = n downto 1 {
        B[count[A[j]]] = A[j]
        count[A[j]] = count[A[j]]-1
    }
}
```

$\Theta(k)$

$\Theta(n)$

$\Theta(k)$

$\Theta(n)$

$\Theta(n+k)$

If $k = \Theta(n)$, then overall complexity is $\Theta(n)$.

Radix Sort

- **Aim: sort n words, each with d digits.**

329		329
457		355
657		436
839	$\longrightarrow$	457
436		657
720		720
355		839

$n = 7$ words, each with $d = 3$ digits, each digit is in 0..9

- **Human: sort according to the *most significant* digit first:**

329 355 457 436 657 720 839

sort each pile separately according to second most significant digit and so on

- **Con: need to keep track of many piles of numbers.**

Radix Sort

- Amazingly, radix sort performs sorting on the *least significant* digit first.

- Starting from least significant digit, sort the digit using a stable sort like counting sort.

- Repeat till the most significant digit.

Radix Sort

```
radix_sort(A,d) {
    for i = 1 to d {
        sort A on digit i using a stable sort
    }
}
```

- Suppose we use counting sort to sort each digit in radix sort. Each digit is in range 1..k
  - Each counting sort - $\Theta(n+k)$
  - d counting sorts are used - $\Theta(d(n+k))$

Radix Sort Correctness

- Why does it work? Key: stable sort is used in each iteration.
- Proof by induction on the number of digits:
  - Basis step: $d = 1$, counting sort gives correct sorting.
  - Inductive step: Suppose that radix sort is correct for numbers with $d - 1$ digits.

  After $d - 1$ iterations, the numbers are sorted according to their low-order $d - 1$ digits. The final sort orders the numbers by their d-th digit. Consider 2 numbers a and b with d-th digit a_d and b_d respectively.
  - If $a_d < b_d$, then $a < b$ regardless, so final sort is correct.
  - If $a_d > b_d$, then $a > b$ regardless, so final sort is correct.
  - If $a_d = b_d$, since final sort is stable, the relative order of a and b remains unchanged.
    - Correct ordering is determined by low-order $d - 1$ digits since they have same d-th digit.
    - Induction hypothesis: a and b are in the correct ordering.

Breaking into digits

- How to break given elements into "digits"?
- n words, b bits/word
  - break into r-bit digits $\Longrightarrow$ $d = b/r$ digits/word
  - each digit is then in the range 0.. $k = 2^r - 1$
- Complexity: $\Theta\left(\frac{b}{r}(n + 2^r)\right)$
  - If $r = lg\, n$: $\Theta\left(\frac{b}{lg\, n}(n + n)\right) = \Theta\left(\frac{bn}{lg\, n}\right)$
  - If $r \gg lg\, n$: 2^r term dominates r term in denominator so complexity is at least $\Omega\left(\frac{bn}{lg\, n}\right)$.
  - If $r \ll lg\, n$: $n + 2^r = \Theta(n)$ but $\frac{b}{r}$ increases, so $\Omega\left(\frac{bn}{lg\, n}\right)$.
- In general, choose $r \approx lg\, n$

Comparison with Quicksort

- **Suppose we want to sort 2^{16} 32-bit numbers.**
  - $n = 2^{16}, b = 32$
  - **Radix sort: choose $r = lg\ 2^{16} = 16$, then $d = \frac{b}{r} = 2$**

    $\Longrightarrow$ 2 passes over the n numbers
  - **Quicksort: $lg\ n = 16$ passes over the n numbers**
- **What is the magic?**
  - **Counting sort uses the fact that keys are integers, not just comparison of keys.**
  - **Keys are used as array indices.**
- **However, actual speed depends on:**
  - characteristics of the implementations
  - underlying machine (e.g., quicksort often uses hardware caches more effectively than radix sort)
- **Counting sort also not in-place: storage issues.**

Bucket Sort

- Assumes the input is generated by a random process that distributes values uniformly over [0,1).

- Idea:
  - divide [0,1) into n equal sized buckets represented by B[1..n]
  - distribute the given n inputs into the buckets
    - maintain a linked list for each bucket
  - sort each bucket (or linked list)
  - concatenate B[1], …, B[n] together to form final output

Bucket Sort

0.74
0.15
0.32
0.29
0.71
0.98
0.21
0.14
0.27
0.69
1
2
3
4
5
6
7
8
9
10
0.15
0.14
0.29
0.21
0.27
0.32
0.69
0.74
0.71
0.98

Bucket Sort

0.74
0.15
0.32
0.29
0.79
0.98
0.69
0.74
0.27
0.94
1
2
3
4
5
6
7
8
9
10
0.15
0.15
0.29
0.27
0.29
0.32
0.69
0.74
0.74
0.94
sort
sort
sort
sort
sort
sort
sort
sort
sort
sort

Bucket Sort Exercise

- Values distributed uniformly over [0,1).
- Each link list is on average small, so average complexity per list is $\Theta(1)$ (regardless of the sorting algorithm used, as long as it is reasonable. E.g., use insertion sort for simplicity).
- Therefore, overall average complexity : $\Theta(n)$
- Exercises:
  - write the pseudo-code for insertion sort on a singly linked list
  - write the pseudo-code for bucket sort.

Learning Takeaway

- **Sorting in linear time is possible if we know the values to be sorted have certain properties:**
  - **integers in a given range – counting sort**
  - **can be represented with fixed number of digits – radix sort**
  - **uniformly generated – bucket sort (average complexity)**

- **Depending on your problem, choose the best procedure!**