



# Midterm Exam Notes



Can use AI just don't rely on it, double check outputs and understand what is being outputted

# Memory and OOP

## Abstract Data Types (ADT)

- Model of a data structure that specifies:

(i) what operations can be performed on the data

(ii) not how these operations are implemented

### Example: A Bag ADT

just a container for a group of data items

(i) Positions of data items don't matter

$$4 \in \{3, 2, 6\} \Leftrightarrow \{2, 6, 3\}$$

(ii) Data items do not need to be unique

$$\{2, 2, 10, 2, 5\} \text{ is a bag, but not set}$$

### Encapsulation

Suppose a client has a class 'MyClass' and wants to use IntBag.

```
class MyClass {
    ...
    void myMethod() {
        IntBag b = new IntBag();
        b.items[0] = 17; // not allowed!
        ...
    }
}
```

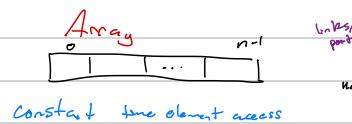
Cannot modify field directly due to being private but add method allows controlled modifications

## Generic Types

Generics allow you to store any data type, so the bag can use any type specified

Can use Object  
expands, just matches  
for type parameters

## Arrays vs. Linked Lists



Linked List  
only know ref to header  
 $O(n)$  direct access time

## Phonebook Example

- Implement a phonebook

(1) Operations: add(element)

remove(element)

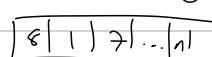
Search(Phone) return number

(2) Data Structures: Array (sorted/unsorted)

Linked List (sorted/unsorted)

add/remove should not disturb sorting

## Unsorted Array



Best case: 1 check

Worst case:  $n$  comparisons

Average case:  $\frac{n}{2}$  comparisons

### Add

Just Add to end

### Remove

## Implementation of IntBag in Java

public class IntBag implements Bag {

### // instance variables

Access Modifiers  
private int[] items;

private int numItems;

### // methods

public boolean add(int item) {

if (numItems == items.length)

return false;

items[numItems++] = item;

return true;

} ...

}

Container using private fields  
and public methods. Occasionally  
private helper methods

## Using a Superclass to Implement Generics

public class Bag {

### // instance variables

private Object[] numObj;

private int numItems;

### // methods

...

public class Test {

### push

Bag m = new Bag();

m.add("37G");

most specific to prevent runtime error

String bodyTemp = (String) m.grabItem();

m.add(new Integer(96));

Assume grabs at C0

Integer temp = (Integer) m.grabItem();

// runtime error bc ["37G", 96]

} ...

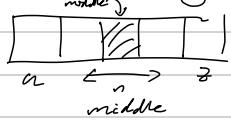
## Example of Sorted Array Addition

A	G	E	K	O	...	n
---	---	---	---	---	-----	---

- 1) Logn Binary Search add  $\rightarrow O(n + logn)$
- 2)  $n$  Shifting Elements logn  $\ll n$  so we say  $O(n)$

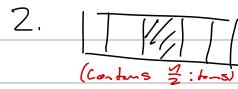
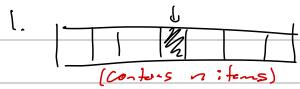
## Binary Search

Sorted Array



You need a  
sorted array  
for this to work

$O(\log(n))$  Complexity



Target lies left of target →  $\frac{n}{4}$  items  
now only need  $\frac{n}{2}$  of array

Find Number (Person)

Pseudocode

Low = 0

High = phonebook.size

while low <= high  $\geq$

P = floor((low + high) / 2)

Compare P<sup>th</sup> person

if f then same

return number

else if contain in book

high = P - 1

else

low = P + 1

3 return Not Found

3

# Recursion Mathematical Background

## Non-recursive Programming

- More familiar
- Calculate  $\sum_{i=1}^n i$  can be done using 'for' loop  
 $\rightarrow \text{sum}(n) = \sum_{i=1}^n i = 1+2+3+\dots+n$
- Can calculate recursives, but less intuitive

```
int sum(int n) {
    int i, sum;
    sum = 0;
    for (i=1; i<=n; i++) sum += i;
    return sum;
}
```

## Recursive Programming

- $\sum_{i=1}^n i = 1+2+3+\dots+(n-1)+n$   
 $\text{sum}(n) = \text{sum}(n-1) + n$
- ↑ recursive call
- Reduces size of problem!

Recursive call must

be shorter to allow  
code to terminate

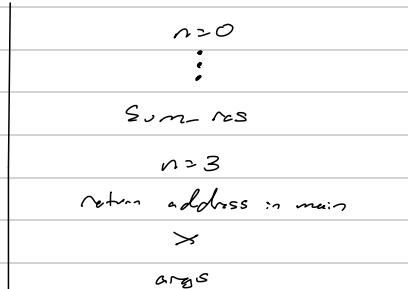
- A recursive method is a method that invokes itself
- Must specify a base/stopping condition!
- Can get farcely to look at n stack  
 $\hookrightarrow$  Simple to implement but hard on computer memory as no computations done until 'n=0'
- Code will run forever without a base case  $\rightarrow$  stack overflow usually
- Stack works in a First In - Last Out manner

```
int sum(int n) {
    if (n <= 0) {
        return 0;
    } else {
        int sum_result = n + sum(n-1);
        return sum_result;
    }
}
```

↑ This does not work; tail recursion  
We need to make a base condition of 0!

```
int sum(int n) {
    ...
    if (n == 0) {
        return 0;
    } else {
        int x = sum(n-1);
        ...
        return x;
    }
}
```

main() calls sum(8)  
 $\text{sum}(8)$  calls  $\text{sum}(7)$   
 $\vdots$   
 $\text{sum}(8)$  returns 3+3 or 6  
main assigns x=6



uses n grows space  
 complexity grows a lot  
 more  $\rightarrow$  inefficient

## Example: Counting Occurrences of a character in a String

### Thinking Recursively

How can we break this problem down into smaller sub-problems?

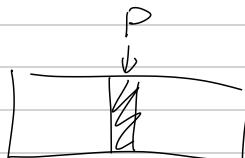
What is the base case? : if (s.length() == 0) return 0;

Do we need to combine the solutions to the sub-problems? If so, how?

```
int occurrences (String s, char c) {
    if (s.length() == 0) return 0;
    if (s.charAt(0) == c)
        return 1 + occurrences (s.substring(1), c);
    else
        return 0 + occurrences (s.substring(1), c);
}
```

## Example: Reversing an Array $\rightarrow$ code11.java

```
myFindNumber (person, low, high)
    if (low > high) return Not_Found
    p = floorAvg (low, high)
    if the same
        return number
    else if even
        high = p-1
    else
        low = p+1
```



## Fibonacci

- Call Tree Created
- Iteration better



# Runtime Analysis

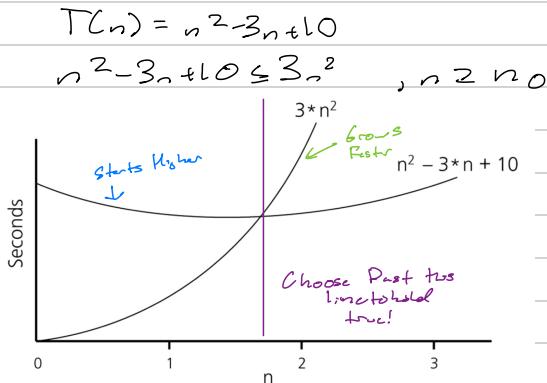
```
int sum(float[] array) {
    for (int i = 0; i <= array.length; i++)
        array[i] = array[i]/0.2;
}
```

```
int sum(float[] array) {
    for (int i = 0; i <= array.length; i++)
        array[i] = array[i]*5;
}
```

- Size of the Array Determines runtime  
↳ These shown as constant time operations
- Both methods need  $2N$  operations

Show that  $T(n) = n^2 - 3 \cdot n + 10$  is order of  $O(n^2)$

- Show that there exist constants  $c$  and  $n_0$  that satisfy the condition



**Remember**

We use

almost always

interested in

the tightest

boundary!

**Remember you can choose  $c$  and  $n_0$  values!**

## Functions Growth Rates: 'Big O' Notation

Consider possible functions  $T(N)$  and  $f(N)$

$\hookrightarrow f(N)$  is runtime complexity of  $T(N)$

"Big O":  $T(N) = O(f(N))$  : if  $\exists c, n_0 > 0$  s.t.

$T(N) \leq c f(N) \quad \forall N \geq n_0$

↳ we focus on very Large Numbers to determine

Example:  $10 * N^2 + 10000 = O(N^2)$ ?

$10 * N^2 + 10000 \leq c N^2$   
constants don't matter for small  $N$   $\hookrightarrow N^2 \leq N^3 \hookrightarrow$  grows much faster

Interested in tightest bound  $10 * N^2 + 10000 = O(N^2)$ ?

↳ Can be true for large  $c$ , so choose  $O(N^2)$

## Functions Growth Rates: Other Definitions

$T(N) = \Omega(f(N))$  if there are positive constants  $c$  and  $n_0$  such that

$T(N) \geq c f(N)$  for all  $N \geq n_0$  **Opposite to Big O**

Example:  $0.0001 * N^3 = \Omega(N^2)$

$T(N) = \Theta(f(N))$  iff  $T(N) = O(f(N))$  and  $T(N) = \Omega(f(N))$

Example:  $0.001 * N^2 + 10000 * N = \Theta(N^2)$  **Must be equal!**

$T(N) = o(f(N))$  if for all constants  $c$  there exists an  $n_0$  such that

$T(N) < c f(N)$  for all  $N > n_0$ .  $\checkmark$  instead of  $\exists$

or

$T(N) = \omega(f(N))$  iff  $T(N) = O(f(N))$  and  $T(N) \neq \Omega(f(N))$

Example:  $10 * N^2 + 10000 = \omega(N^3)$

## Big-O Toolbox (for positive monotonic Functions)

• Constants do not matter:  $T(N) = O(f(N) + c) \rightarrow T(N) = O(f(N))$

$T(N) = O(c * f(N)) \rightarrow T(N) = O(f(N))$

$T(N) + c = O(f(N)) \rightarrow T(N) = O(f(N))$

$T(N) * c = O(f(N)) \rightarrow T(N) = O(f(N))$

• Algebraic Properties:

- If  $T_1(N) = O(f(N))$  and  $T_2(N) = O(g(N))$  then  $T_1(N) + T_2(N) = O(f(N) + g(N))$

If  $T_1(N) = O(f(N))$  and  $T_2(N) = O(g(N))$  then  $T_1(N) * T_2(N) = O(f(N) * g(N))$

If  $T_1(N) = O(f(N))$  and  $g(x)$  is monotonic, then  $g(T_1(N)) = O(g(f(N)))$

• Dominated Terms don't matter:  $T(N) = O(f(N) + g(N))$   $\& g(N) = o(f(N))$   
Then  $T(N) = O(f(N))$  **Little-o, smaller!**

## Logarithmic Properties

$$\log(cd) = \log(c) + \log(d)$$

$$\log(c/d) = \log(c) - \log(d)$$

$$\log(c^d) = d \log(c)$$

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$$

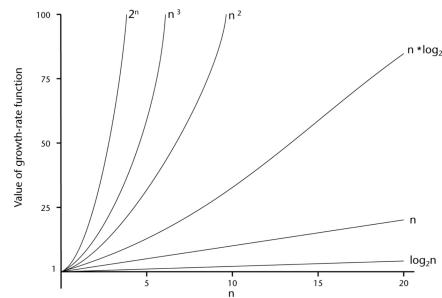
Corollary: Logarithmic grows

slower than any polynomial

$$\log N < N^c \quad \text{Dominant}$$

$$\rightarrow T(N) = N^c + \log(N)$$

## Comparison of Growth-Rate Functions



## Useful Mathematical Equations

$$\sum_{i=1}^n i = 1+2+\dots+n = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=1}^n i^2 = 1^2 + 4 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3}$$

$$\sum_{i=0}^{n-1} 2^i = 0 + 1 + 2 + \dots + 2^{n-1} = 2^{n-1}$$

## Applying Concepts & Relative Rates

- > 1000000 versus 0.01 \* sqrt(N)
- > log(N) versus sqrt(N)
- > log(N) versus N<sup>0.001</sup>  $N^{0.001} = N^{\log_2 1.001}$
- > N<sup>3</sup> versus 10000 \* N<sup>2</sup>
- > log<sup>2</sup>(N) versus 10<sup>4</sup> \* log(N<sup>2</sup>)  $10^4 \log(N^2) = 50 \log(N)$
- > 2 \* log(N) versus log<sub>2</sub>(N)  $\log_2(N) = \frac{\log(N)}{\log(2)}$
- > N<sup>0.25</sup> versus 3<sup>N</sup>  $3^N = 1.5^N \cdot 2^N$

## Algorithm Analysis

### Models and Assumptions

- > Consider rather abstract algorithm (a procedure or method)
- > Ignore the details/specifics of a computer
- > Assume sequential process (a sequence of instructions)
- > Ignore small constant factors (e.g., differences among "primitive" instructions)
- > Ignore language differences (e.g., C++ versus Java)

### Average-case versus worst-case performance

- > Example: finding a person in phonebook using sequential search
  - Best-case?
  - Worst-case?
  - Average-case?

## How to Calculate Running Time

$$(\cancel{N} \cancel{N}) + \cancel{b} \rightarrow 0$$

Time Complexity:  $O(N)$

```
public static int sum(int n)
{
    int partialSum = 0;
    for (int i = 1; i <= n; i++)
        partialSum += i*i*i;
    return partialSum;
}
```

## General Rules

- Simple Statement: Constant
  - $\cancel{O(1)}$ :  $i++, i < n, \text{etc.}$
- Simple Loops: # iterations
- Nested Loops: Product of # iterations of outer and inner loops cost of inner body
  - $\cancel{O(1)}$ :  $\text{for}(n)$
  - $\cancel{O(n)}$ :  $\text{for}(m)$
  - $\cancel{O(n^2)}$ :  $K++$

- Consecutive Statements: Count most expensive

$\cancel{O(1)}$ :  $i = 0;$   
 $\cancel{O(n)}$ :  $\text{while}();$   
 $\cancel{O(n^2)}$ :  $\text{for}(n)$   
 $\cancel{O(n^2)}$ :  $\text{for}(m)$   
 $\cancel{O(n^2)}$ :  $K++$

- Conditions: Count most expensive branch

$\hookrightarrow$  Focus on worst-case.

## Examples

$i = 0;$   
 $\cancel{O(1)}$ :  $\text{if}(\times \times \times) \cancel{O(1)}$

$\cancel{O(1)}$ :  $\cancel{O(1)}$   
 $\cancel{O(1)}$ :  $\cancel{O(1)}$

$K = 1;$

$\text{for}(i=1; i < n; i++)$

$K++;$   $\cancel{O(n)}$

$\cancel{O(1)}$

$\cancel{O(1)}$

$\cancel{O(n)}$

$i = 0;$

$\cancel{O(n)}$ :  $\text{while}(i < n) \{ \dots i++; \dots \}$   $m$   
 $\cancel{O(n^2)}$ :  $\text{for}(i=0; i < n; i++)$   $n^2$   
 $\cancel{O(n^2)}$ :  $\text{for}(j=0; j < n; j++)$   $n^2 \gg m$ , but  $n^2 \neq m$

$K = 1;$

$\cancel{O(n)}$ :  $\text{for}(i < n)$

$\cancel{O(n)}$ :  $\text{for}(j < m)$

$K++;$

$\cancel{O(n)}$

# Linked lists

## Array Representation

- ✓ Easy efficient access to any access
- ✓ Constant time : for access
- ✗ Need to have size specified
- ✗ Difficult to add/remove in arbitrary indices

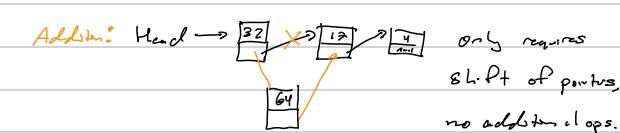
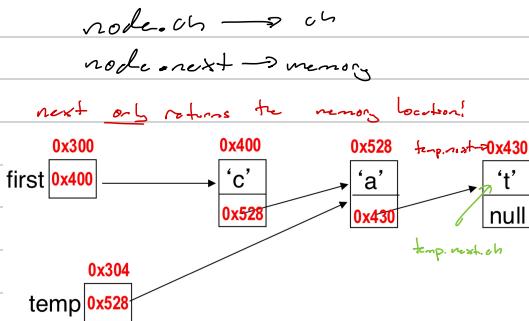
## LinkedList Representation

- ✓ Dynamic sequence: easy addition/removal
- Links of nodes both ends have a reference and data value
- ✗ Access is linear O(n) to get the  $n^{th}$  element: Head  $\rightarrow$  desired node
- Last Data Node: is a null reference
- ✓ No capacity limit provided enough memory
- ✗ Memory overhead for the links

## Example LL

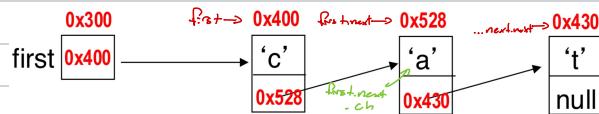
```
public class StringNode {
    private char ch;
    private StringNode next;
    ...
}
```

```
public class LLString {
    private StringNode head;
    private int theSize;
    ...
}
```



## A reference is also a variable

$\rightarrow$  Has its location in memory and whose value is the address (location) of the data



Two nodes can have two different pointers.

Different pointers can point to same memory address

## Recursion on Linked Lists

### Recursive definition of a LL

A LL is either :

(Empty Case)

(i) a single node followed by

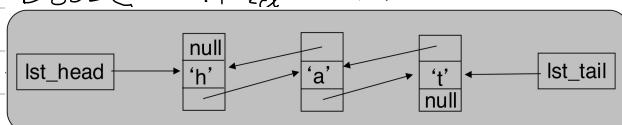
a LL (Recursive case)

### Example: length of a string

- length of "cat" = 1 + length of "at"
- length of "at" = 1 + length of "t"
- length of "t" = 1 + length of the empty string (which is 0)

```
private static int length(StringNode str) {
    if (str == null)
        return 0;
    else
        return 1 + length(str.next);
}
```

## Double Linked List



## Example DLL

```
public class LLString {
    private StringNode lst_head;
    private StringNode lst_tail;
    private int theSize;
    ...
}
```

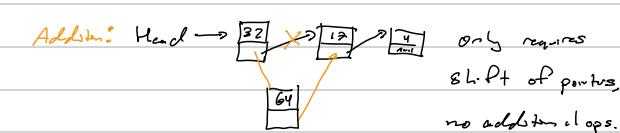
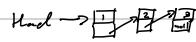
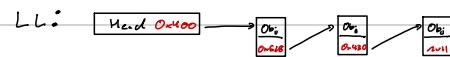
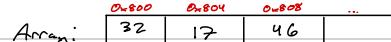
```
public class StringNode {
    private char ch;
    private StringNode next;
    private StringNode prev;
    ...
}
```

Allows for traversal in either direction

## Memory Diagrams

Array: Elements occupy consecutive memory locations in heap  
LinkedList: Node is a distinct object in heap so locations

don't have to be next to each other in memory



## In Class DLL example

Assume you have LLString and StringNode

$\rightarrow$  Access node at position i in a doubly linked list

public StringNode getNode(int i) {

if ( $i < 0$  ||  $i >$  theSize) return null;

// Use tail - i : is closest too

: if ( $i < \text{theSize}/2$ ) {

ptr = this.lst\_head;

for ( $j=0$ ;  $j < i$ ;  $j++$ ) ptr = ptr.next;

else {

ptr = this.lst\_tail;

for ( $j=\text{theSize}-1$ ;  $j > i$ ;  $j--$ ) ptr = ptr.prev;

} return ptr;

### Example Removing a Node from a DLL

↳ Assume Access to previous & next

```
public char remove(ElemNode p) {
    if (p == lstHead || p == lstTail)
        Need diff calc!
```

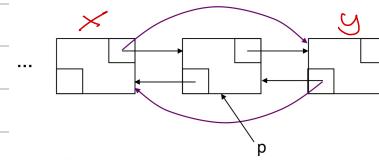
3

```
p.next.prev = p.prev;  
p.prev.next = p.next;  
theSize--;
```

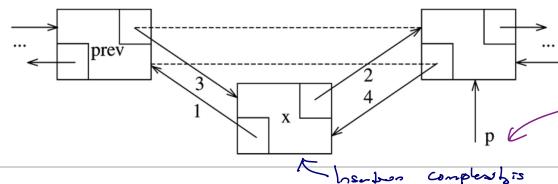
return p.ch;

3

Number of removal references depends on if nodes head or tail or not



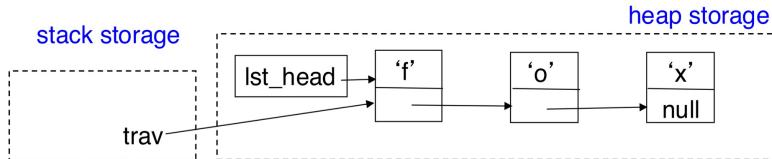
### Same idea for inserting a node



$$O(n+4) = O(n)$$

Always need to find 'p', so shifts at  $O(n)$  complexity  
however complexity is a constant time operation

# More LinkedList Operations



Rest of the class will use singly linked lists usually

## General Traversal Support

→ Inefficiencies: use public LLString methods

```
public class MyClass {
    public static int numOccur(LLString str, char ch) {
        int numOccur = 0;
        for (int i = 0; i < str.length(); i++) {
            if (ch == str.get(i))
                numOccur++;
        }
        return numOccur;
    }
}
```

*Complexity: O(n)*

$O(n^2)$  as each iteration requires a full search to find position

*Bypass private fields*

## Duplicating a Singly LL

- Helper method copy(str)
- Copies all elements through head
- Returns first element of new list

→ Recursive Implementation

(i) Base Case: If str is empty, return null

(ii) Recursive: Copy to call on next

Assume StringNode is well defined

```
private static StringNode copy(StringNode str) {
    if (str == null) // base case
        return null;
    // create the first node, copying the first character into it
    StringNode copyFirst = new StringNode(str.ch);
    // make a recursive call to get a copy of the rest and
    // store the result in the first node's next field
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```

## How can we improve efficiency and follow good OOP practices

→ (i) Or you can use a traversal node

```
public class LLString {
    public int numOccur(char ch) {
        int numOccur = 0;
        StringNode trav = lst_head;
        while (trav != null) {
            if (trav.ch == ch)
                numOccur++;
            trav = trav.next;
        }
        return numOccur;
    }
}
```

*Not  
Good  
Because*

$O(n)$

*Starts from  
beginning, goes  
to end &*

Only allows you to use numOccur, doesn't let to have to run to methods for any other use cases.

→ (ii) Get Access to list internals

```
public class MyClass {
    public static int numOccur(LLString str, char ch) {
        int numOccur = 0;
        StringNode trav = str.getNode(0); // Constant Time Operations
        while (trav != null) {
            char c = trav.getChar();
            if (c == ch)
                numOccur++;
            trav = trav.getNext(); // Get Next Node
        }
        return numOccur;
    }
}
```

This is  $O(n)$  as well

Makes public fields, which you can do too though getters and setters methods.

→ (iii) Double as Iterator

```
public class MyClass {
    public static int numOccur(LLString str, char ch) {
        int numOccur = 0;
        LLString.Iterator iter = str.iterator();
        while (iter.hasNext()) {
            char c = iter.next();
            if (c == ch)
                numOccur++;
        }
        return numOccur;
    }
}
```

Also  $O(n)$

- No use of StringNode Objects
- Does not depend on ListString internals

## Iterator ← This is an Object, it needs to be created

- Provides Iteration ability w/o violating encapsulation
- hasNext() - Checks if current node points to a non-null node
- next() - Returns next internal : increments the iterator

It is implemented below as ↪ inner class

```
public class LLString {
    private StringNode head;
    private StringNode tail;
    ...
    public Iterator iterator(){
        Iterator iter = new Iterator();
        return iter;
    }
}

public class Iterator {
    private StringNode nextNode;
    private Iterator (){
        nextNode = head;
    }
    public boolean hasNext() {...}
    public char next() {...}
    ...
}
```

Instances:

```
LLString.Iterator myIter1 = string.iterator();
LLString.Iterator myIter2 = string.iterator();
```

*Point class & Nested class*

*None of object to create is for*

## Internal workings of Iterators

```
public boolean hasNext() {
    return (nextNode != null);
}

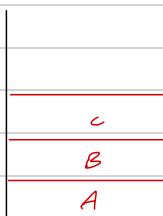
public char next() {
    if (nextNode == null)
        throw new Exception("Falling off the list end");
    char ch = nextNode.ch;
    nextNode = nextNode.next;
    return ch;
}
```

F "O" "X" ...  
F "I" "O" "X" ...  
F "O" "I" "X" ...

# Stacks & Queues

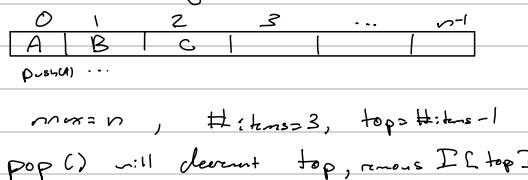
- Linear Data Structures just like arrays and linked lists

## Stacks



- First In, Last Out! (FILO)
- push(...)= add
- pop() = remove
- peek() = get item @ top, but don't remove
- is Empty() and is Full()

### Build with an array



### Build with a Linked List



- push(...) should add to Front, changes ptn
- pop() removes element, Head pts to next

### Generic Array Stack Class

```
public class ArrayStack<T> {
```

```
    ...
    public ArrayStack (int max) {
        items = (T[]) new Object[max]
        top = -1;
        maxsize = max;
    }
```

```
    ...
    ...
}
```

`top++`: use current var, then increment  
`++top`: increment and use that var

### Queue Array Implementation

```
// Add an element to the circular array
public void add(int value) {
    if (size == array.length) {
        System.out.println("Array is full");
        return;
    }
    array[tail] = value;
    tail = (tail + 1) % array.length;
    size++;
}

// Remove and return an element from the
// circular array
public int remove() {
    if (size == 0) {
        throw new IllegalStateException("Array is empty");
    }
    int value = array[head];
    head = (head + 1) % array.length;
    size--;
    return value;
}
```

## Queue

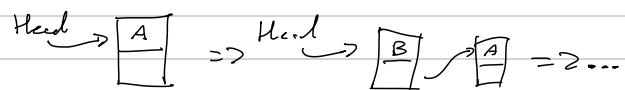


- First In First Out (FIFO)

Everything inserted from tail and comes out of queue from front

- enqueue(...)= add
- dequeue() = remove
- peek(), remove
- insert
- is Empty(), is Full

### Linked List Implementation



- Adding => Constant time O(1)
- This makes Removal linearly O(n)

Making this a DLL, it has constant enqueue and dequeue operations

### No Capacity Issue

Need to use a DLL with two pointers so that the nodes are as efficient as possible

### Summary Q: Emulating Queue using Stacks

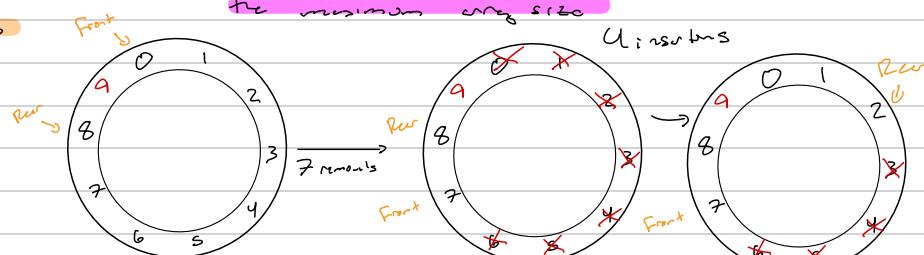
#### Array Implementation

Consider a circular array

- T[] items
  - int front, rear, numItems, maxSize
- Instead of shifting indices, increment the front

Get to end of array, add to start of array

All index operations are modulo array, and modulo is the maximum array size



The queue is empty when Front "overcomes" rear:

$$\rightarrow ((rear + 1) \% maxSize) == front$$

You also have to look at number of items as this increments for both full & empty

# Basics of Trees

↳ First non-linear data structure

## What is a Tree

- A set of nodes, top one is called the 'root'
- A set of edges connecting pairs of nodes
- Cannot have cycles - only unique path to any given node!
- Nodes have data ("payload") consists of one or more fields
- Recursive Data Structure: Each Node in the tree is the root of a smaller tree
  - ↳ Refers to these types of trees as subtrees

## Recall: Phone Book

Data Structure	Search	Insert
Sorted Array	$O(\log n)$ w/ binary	$O(n)$ due to Shifts
Linked List	$O(n)$ using linear	$O(1)$ as LL

## Terminology

- **Key Field:** Field used when searching for a data item
- If a Node N is connected to other nodes below it, it is called the **parent** and its nodes below are **children**
  - ↳ A node can only have one parent, but can be one of many 'siblings'
- **Leaf Nodes:** A node w/o children & doesn't have to be the deepest node!
- **Interior Nodes:** A node which is neither Leaf nor Root
- **Depth of a Node:** # of edges on the path from it to the root
- **Level of a tree:** Made up of Nodes with the same depth
- **Height of a tree:** Maximum depth of its nodes
- **Binary Trees:** Each Node has at most Two children

## Binary Trees

- **Recursive Definition:** A binary tree is either an collection of nodes that is either

(i) empty

(ii) contains a node R (the root tree) that has

- a binary left subtree → choose root (if  $\neq$ ) connects to it
- a binary right subtree

```
public class LinkedTree {
```

```
private class Node {
```

```
private int key;
```

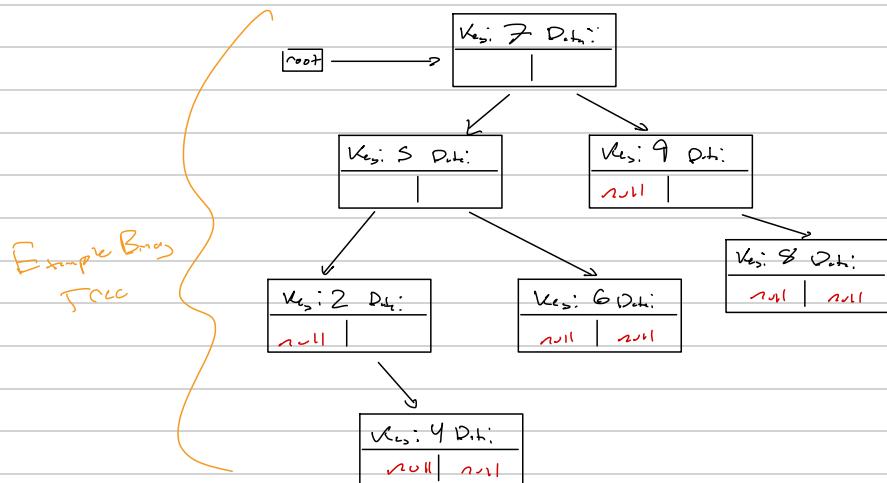
```
private String data;
```

```
private Node left;
```

```
private Node right;
```

```
...  
}
```

```
private Node root;
```



## Preorder Traversal

- 1) Visit Root, N
- 2) Recursively go to L
- 3) Recursively go to R

~~N - L - R~~

Result from Bini: 7 5 2 4 6 9 8

```

private void myPreorderPrint(Node node)
{
    System.out.print(node.key + " ");
    if (node.left != null)
        myPreorderPrint(node.left);
    if (node.left == null)
        myPreorderPrint(node.right);
}
  
```

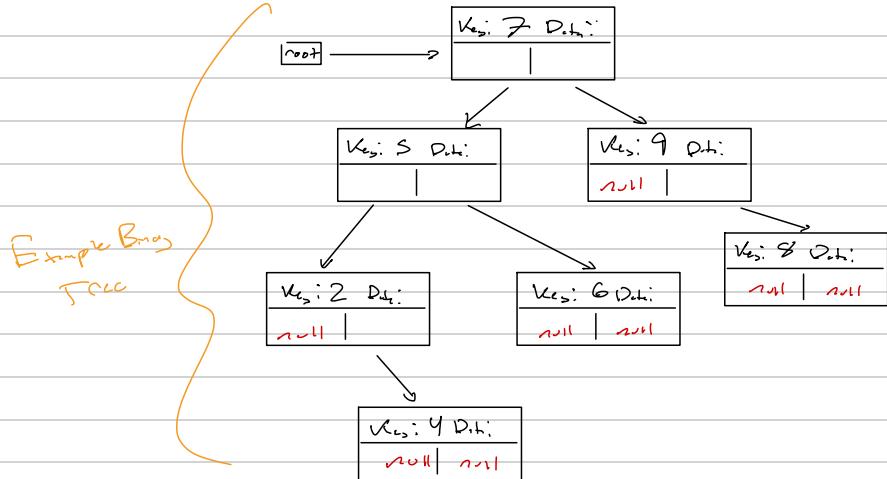
### Postorder Traversal

- 1) Recursively go to L
- 2) Recursively go to R
- 3) Visit N

L R N

Results: 4 2 6 5 8 9 7

Some code almost, just move  
Sout to end of method



### Inorder Traversal

L N R

Results: 2 4 5 6 7 8 9

### Level-Order Traversal

- Also "Breadth-First Traversal"
- Visit nodes by level, from top to bottom and left to right

Results: 7 5 9 2 6 8 4

### Tree Traversal Summary

- preorder: root, left subtree, right subtree
- postorder: left subtree, right subtree, root
- inorder: left subtree, root, right subtree
- level-order: top to bottom, left to right

### High-Level Implementation

Queue  $\xrightarrow{\text{root}}$



- (i) Remove  $\top$
- (ii) Insert left then right node

$\begin{matrix} 5 \\ 9 \end{matrix}$



- (iii) Remove  $\top$
- (iv) Insert  $\top$ 's children

$\begin{matrix} 9 \\ 2 \\ 6 \end{matrix}$



- (v) Remove  $\top$
- (vi) Insert  $\top$ 's children

$\begin{matrix} 2 \\ 6 \\ 8 \end{matrix}$

Repeat until Queue is empty

# Tree Species (Binary & Binary Search Trees)

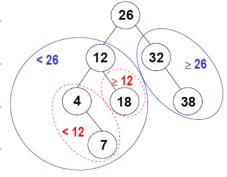
## Binary Search Trees

→ For Each node K

(i) All nodes in K's left subtree are less than K

(ii) All nodes in K's right subtree are greater or equal to K

→ Performing an Inorder Traversal of a Binary Search Tree returns nodes in ascending order.



## Searching An Item in a BST

if  $K == \text{root}$  nodes  $\leftarrow$  do!

else if  $K < \text{root.key}$  nodes  $\leftarrow$  search left

else search right subtree

## Recursive Search Implementation

```
public class LinkedTree {
    ...
    private Node root;
    public String search(int key) {
        Node n = searchTree(root, key);
        return (n == null ? null : n.data);
    }
    private Node searchTree(Node root, int key) {
```

Node trav = root; // Data pointer

while (trav != null) {

; if ( $\text{trav.key} == K$ )

return trav

else if ( $\text{trav.key} < K$ )

trav = trav.left

else

trav = trav.right

}

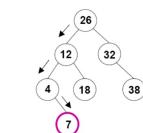
return null;

}

## Recursive Search Implementation

```
public class LinkedTree {
    ...
    private Node root;
    public String search(int key) {
        Node n = searchTree(root, key);
        return (n == null ? null : n.data);
    }
    private Node searchTree(Node root, int key) {
        ...
        if (root == null) Basic case
            return null;
        else if (key == root.key)
            return root;
        else if (key < root.key)
            return searchTree(root.left, key);
        else
            return searchTree(root.right, key);
    }
}
```

```
private class Node {
    private int key;
    private String data;
    private Node left;
    private Node right;
}
```



## Iterative Code Solution

```
public class LinkedTree {
    ...
    private Node root;
    public String search(int key) {
        Node n = searchTree(root, key);
        return (n == null ? null : n.data);
    }
    private Node searchTree(Node root, int key) {
        Node trav = root;
        while (trav != null) {
            if (key == trav.key)
                return trav;
            else if (key < root.key)
                trav = trav.left;
            else
                trav = trav.right;
        }
        return null;
    }
}
```

## Binary Tree Item Insertion

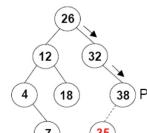
We want to insert an item whose key is k.

First, we find the node P that will be the parent of the new node:

➤ we traverse the tree as if we were searching for k, but we don't stop if we find it - we continue until we can't go any further

Next, we add the new node to the tree:

if  $k < P$ 's key, make the node P's left child  
else make the node P's right child



Special case: if the tree is empty, make the new node the root of the tree

## Iterative Insertion

### Two Phases

(i)  $\text{trav} = \text{parent}$  performs traversal

down to point of insertion

(ii) parent's  $\text{ptr}$  stays one above

$\text{trav}$

## Iterative Insertion Code

```
public void insert(int key, String data) {
    // Find the parent of the new node.
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        parent = trav;
        if (key < trav.key)
            trav = trav.left;
        else
            trav = trav.right;
    }
    // Insert the new node.

    if (parent == null) // the tree was empty
        root = new Node(key,data);
    else if (key < parent.key)
        parent.left = new Node(key,data);
    else
        parent.right = new Node(key,data);
}
```

## Node Deletion

Three Cases for deleting a Node

- Case 1: X has No Children

(i) Remove X from Tree by updating its parents reference to null

- Case 2: X has One Child equivalent to saying pointer

(i) Take the parent's reference to X and set it to X's child

- Case 3: X has Two Children

(i) Find leftmost node in X's right subtree (Smallestnode) - call it Y

(ii) Copy Y's key to X, but not its potential child

(iii) Delete Y using Case 1 or Case 2.

## Insertion / Deletion

don't Guarantee Balanced Tree

## Deletion Implementation

```
public String delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key)
            trav = trav.left;
        else
            trav = trav.right;
    }
    // Delete the node (if any) and return the removed item.
    if (trav == null) // no such key
        return null;
    else {
        String removedData = trav.data;
        deleteNode(trav, parent); // Helper Method
        return removedData;
    }
}
```



```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left == null || toDelete.right == null) {
        // Cases 1 and 2
        Node toDeleteChild = null;
        if (toDelete.left == null)
            toDeleteChild = toDelete.right;
        else
            toDeleteChild = toDelete.left;
        if (toDelete == root)
            root = toDeleteChild;
        else if (toDelete.key < parent.key)
            parent.left = toDeleteChild;
        else
            parent.right = toDeleteChild;
    } else { // case 3
        ...
    }
}
```

## Case 3

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left == null || toDelete.right == null) // case 1 and 2
        ...
    else { // case 3
        // Get the smallest item in the right subtree.
        Node replacementParent = toDelete;
        Node replacement = toDelete.right;
        while (replacement.left != null)
            replacementParent = replacement;
            replacement = replacement.left;
        ...
        // Replace toDelete's key and data
        "toDelete.key = replacement.key;" // toDelete.data = replacement.data;
        ...
        // Recursively delete the replacement item's old node.
        deleteNode(replacement, replacementParent);
    }
}
```

3 Finding In-Order Successor

Height of Tree  
Containing n items  
depends on balancing

## Efficiency of Binary Search Tree

Search, Insert, Delete all have same time complexity

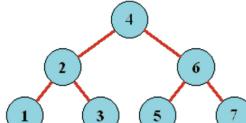
Time Complexity for searching a binary tree

→ Best Case:  $O(1)$

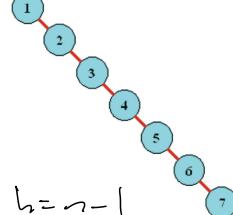
→ Worst Case:  $O(h)$  } This is the height of tree

→ Average Case:  $O(h)$

• Insert and delete both involve path traversal from root  $\rightarrow$  node  $\rightarrow$  less than one child



$$h = \log n$$



$$h = n-1$$

# AVL Trees

## Height and Balance

- Height of a Tree:** The length of the longest path from the root node to a leaf node

- Balance of a Tree:** Given node N as the root:

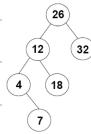
$$\text{Balance}(N) = |\text{height}(N\text{'s right subtree}) - \text{height}(N\text{'s left subtree})|$$

↳ Balance of a Leaf Node

is always 0

Ex: (i)  $\text{balance}(\text{node } 26) = |0 - 2| = 2$

(ii)  $\text{balance}(\text{node } 4) = |0 - (-1)| = 1$



One Leaf: Height = 0

Empty Tree: Height = -1

## AVL Trees (Adelson-Velsky & Landis' G2)

- The balance of all Nodes are never exceeding  $|1|$ ,  $\text{balance}(N) \leq |1|$

↳ can either be  $-1, 0$ , or  $1$

- This rule must be fulfilled after every insertion & deletion

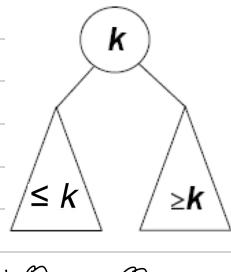
- Steps taken to restore balance must

(i) Maintain the search-tree inequalities

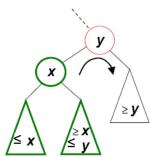
(ii) Have a worst-case time complexity of  $O(\log n)$

Ex: Empty Tree  $\xrightarrow{\text{Add 1,2}}$  ① ②  $\xrightarrow{\text{Add 3}}$  ② ~~③~~  $\xrightarrow{\text{Not Balanced}}$  ③  $\xrightarrow{\text{Invert AVL}}$  ②  $\xrightarrow{\text{Rebalance}}$  ① violates balance  $\xrightarrow{\text{Analysis}}$  ②, ③ are ok  $\xrightarrow{\text{Rotate ① To Left}}$  ① ② ③

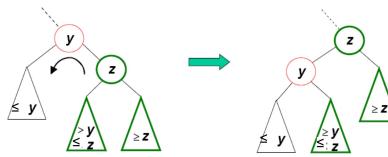
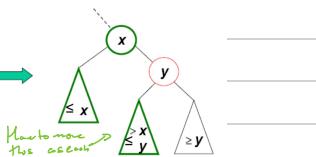
Result: Tree Invariants



## Rotation Operations

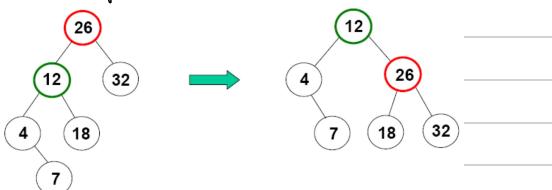


Right Rotation on (around) y

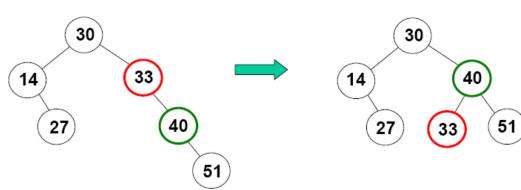


Left Rotation on y

## Example Rotations



Right Rotation on Node 26



Left Rotation on node 33

## Implementation of AVL Trees

- Node Class is similar, but you need to keep track of the balance of each node!

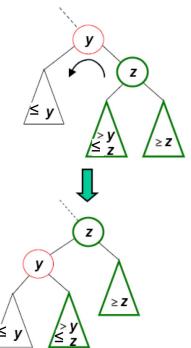
```
public class AVLTree {
    private class Node {
        private int key;
        private String data;
        private Node left; // reference to left child
        private Node right; // reference to right child
        private Node parent; // reference to parent node
        private int balance; // balance value of the node
        ...
    }
    private Node root;
    ...
}
```

## Rotations Implementation

- Just involves changing pointers

- # ptr upds
- 1 (i) Right child of parent of y
  - 2 (ii) Right Child and parent of y
  - 2 (iii) Left child and parent of z
  - 1 (iv) Parent of left child of z

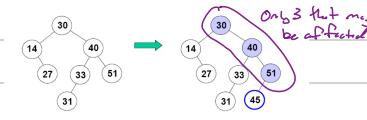
6 Total Ptrs  $\rightarrow$  Constant Time Complexity



## Insertion into AVL Tree

Remember 2 new fields: Ref to nodes parent & its balance

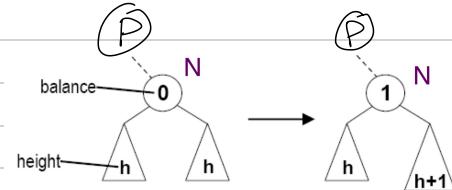
- An insertion can only affect the height values and balance values in the new nodes' ancestors
- First step is inserting node as in a binary search tree.



**Change in Balance** Assume node N to be affected by insertion

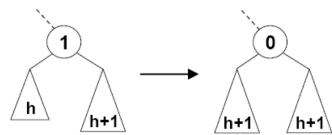
**Case 1** N's balance goes from 0 to 1

- N's subtree still within AVL rules
- Height of N has increased
- $\hookrightarrow$  Balance of N's parent  $\leftarrow h+1$  Change
- Need to check if N's parent violates the AVL rule
- $\hookrightarrow$  Balance of other ancestors may also be affected



**Case 2** N's balance goes from 1 to 0

- Height of the subtree of which N is the root has not changed
- $\hookrightarrow$  Don't need to look at ancestors!



**Case 3** N's balance goes from 1 to -2

- Need to rebalance tree using rotations
- Rotations will restore the height of the rotated subtree
- Lead prior to rotation
- $\hookrightarrow$  N's ancestors' balances won't change

$\rightarrow$  **Case 3a:** Inside vs. Outside Insertions on left subtree *Left Subtree of Left Child*

$\nearrow$  Inside: Closer to center line  
 $\searrow$  Outside: Further from center line

- Perform single right rotation about y

$\rightarrow$  **Case 3b:** y's balance to +2,  $\rightarrow$  mirror to 3b *Right subtree of Right Child*

- Perform single left rotation about y

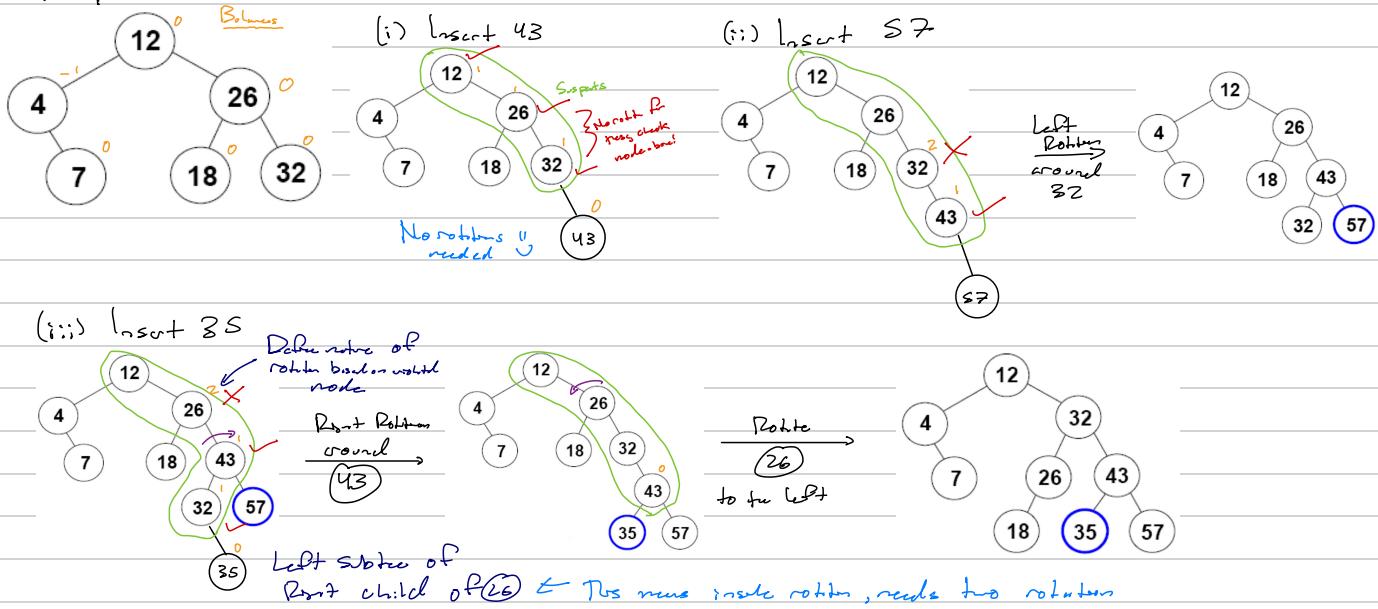
$\rightarrow$  **Case 3c:** we've added a node to the right subtree of y's left child, x, bringing y's balance to -2.

$\rightarrow$  **Case 3d:** we've added a node to the left subtree of y's right child, x, bringing y's balance to +2. *(symmetric to case 3c; can you draw pictures to show how it works?)*

## Complete Insertion Method

- Insert new node  $N$  as in a binary tree
- Use parent references to follow the path from  $N$  back to the root
  - If an ancestor's balance was 0, it will now be  $\pm 1$  (case 1)  $\rightarrow$  Continue up the path to the root
  - If an ancestor's balance was  $\pm 1$ , there are now two cases:
    - (i) It is now 0  $\rightarrow$  stop
    - (ii) It is now  $\pm 2 \rightarrow$  perform 1 to 2 rotations to rebalance tree (cases 3a-3d)
      - $\hookrightarrow$  number of rotations depends on where  $N$  is inserted
- In either case, stop - there is no need to go any further up the tree

## Examples of Rotations



# AVL Tree Deletion

Deletion of AVL Tree Nodes is completed

- Implementation:
- shorter flag used to indicate if a subtree was shortened
  - each node has a balance factor

- (i) left-high: height of left subtree exceeds right subtree
- (ii) right-high: height of right subtree exceeds left subtree
- (iii) equal: height of left and right subtrees equal

Algorithm:

- (i) shorter initialized as true

↑  
(ii) starting from the deleted node back to the root, take an action based on

After standard  
remove from a  
binary tree

- value of shorter

- balance factor of current node

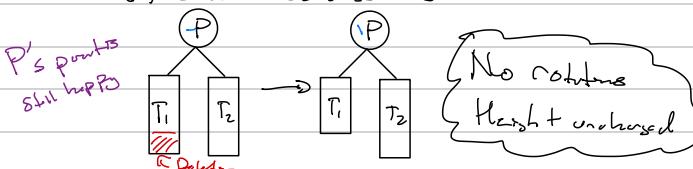
- sometimes balance factor of a child of the current node

- (iii) until shorter flag becomes false or you reach root

Case 1: The balance factor of p is equal

- (i) Change the balance factor of p to right- or left- high

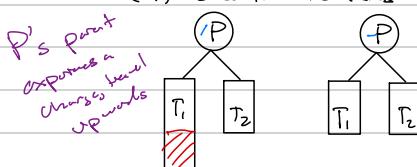
- (ii) shorter becomes false



Case 2: The balance factor of p is not equal & taller subtree is shortened

- (i) Change the balance factor of p to equal

- (ii) shorter removes tree



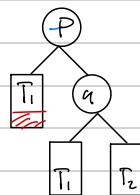
Case 3: The balance factor of p is not equal & shorter subtree is shortened

→ Case 3a: The balance factor of q is equal

- (i) Apply a single rotation

- (ii) Change the balance factor of q to left- or right- high

- (iii) shorter becomes false



→ Case 3b: Balance factor of q is same as p

- (i) Apply single rotation

- (ii) Change the balance factors of p & q to equal

- (iii) shorter removes tree.

→ Case 3c: Balance factor of q is opposite of p

- (i) Apply a double rotation

- (ii) Change balance factor of new root to equal

- (iii) Also change the balance factors of p & q

- (iv) shorter removes tree

Efficiency of AVL tree node deletion

$O(\log_2 n)$  → worst case requires full path up and down with nested constant time operations

## Other Balanced Trees

## AVL Trees Aren't Perfect

- Not memory friendly "Vegs at each node"
    - ↳ Compress file size to a non-binary tree to reduce number of traversals
  - Having more Veggies per node reduces transport looks at unneeded data

Why not store multiple  
keys at each node

## B-Tree

- Reduces the number of l-8L accesses
  - All levels are at the same level
  - Defined by its minimum degree 't' & Input Parameter
  - Every node except root must contain at least t+1 keys
    - ↳ Root can contain minimum 1 key
  - AVL nodes (including root) must contain a minimum of 2t-1 Keys
  - Number of children of a node is equal to number of keys in it plus 1

Example

```

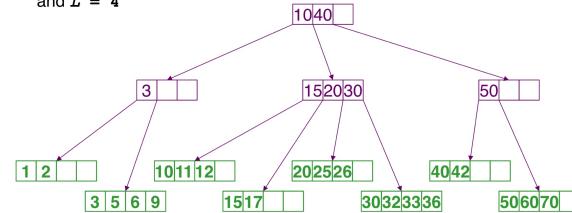
graph TD
    Root[3, 30, 60] --- Node1[1, 2]
    Root --- Node2[4, 5, 6]
    Root --- Node3[40, 50]
  
```

## Alternator Diagrams

- B-Trees are specialized  $M$ -ary search trees
  - Each node has many keys  
  - maximum branching factor of  $M$
  - the root has between 2 and  $M$  children or at most  $\lfloor \frac{M}{2} \rfloor$  keys
  - other internal nodes have between  $\lceil \frac{M}{2} \rceil$  and  $M$  children
  - internal nodes contain only search keys (no data)
  - each (non-root) leaf contains between  $\lceil \frac{M}{2} \rceil$  and  $M$  keys
  - all leaves are at the same depth



B-Tree with  $M = 4$   
and  $L = 4$

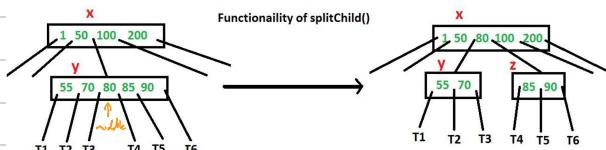


Everything is logarithmic in life

↳ beginning -> to  $\beta$ -trans

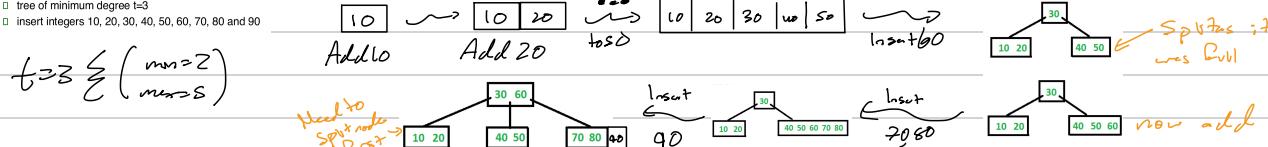
- Nodes w/ Maximum Number of Nodes are a problem with insertion
  - New node is always inserted @ leaf node
  - Before inserting a key to node, make sure the node has extra space
  - If a node is full, use splitChild(i) to split a child of a node

Functionality of s



F - Example

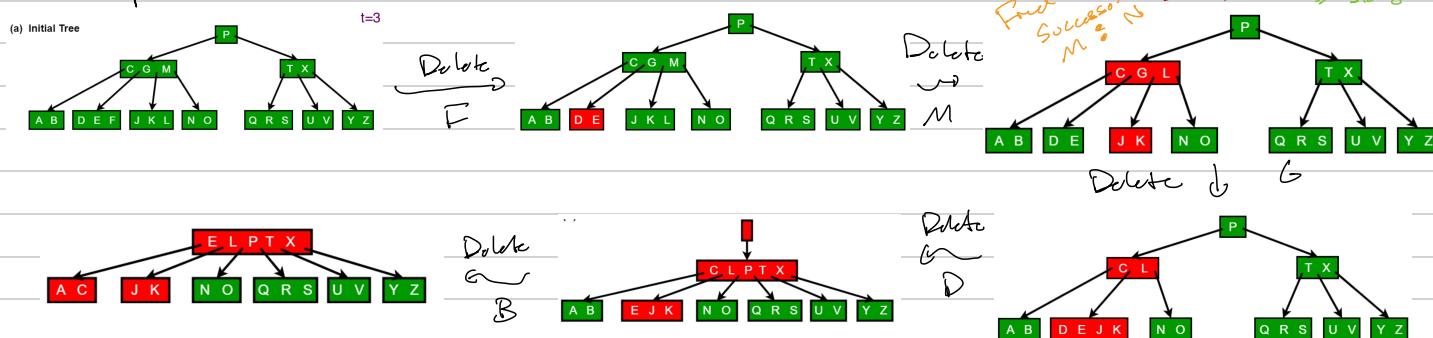
- tree of minimum degree t=3
  - insert integers 10, 20, 30, 40, 50, 60, 70, 80 and 90



## Deletions from B-trees

- Nodes with minimum node is problematic
  - Back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys

## Example



# Final Exam Notes

---



# Heaps & Priority Queues

## Queues

- Items added to rear and removed from front
  - ↳ FIFO (first comes element at head)
- Operators: Insert, remove, peek, isEmpty, size

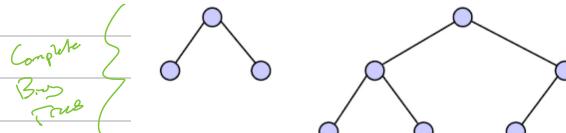
## Priority Queues

- Collection of items with each having an associated priority
- Operations: Insert, remove → item with highest priority
- Uses a Heap!

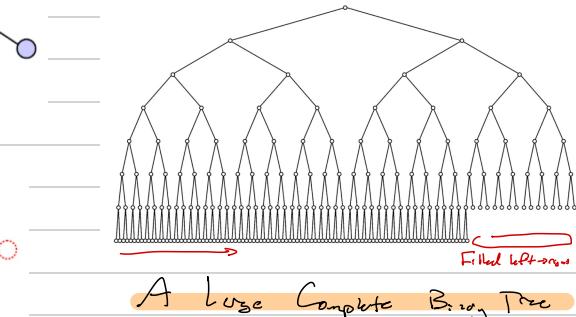
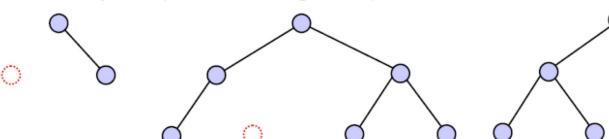
## Structure types of Binary Trees

- Full Binary trees: Every node has exactly two or zero children
- Perfect Binary trees: Full trees with all leafs at the same depth
- Complete Binary trees: Balanced, and at the same level, filled left → right

Complete Binary Tree



Not complete (○ = missing node):



A Large Complete Binary Tree

## Array Representations of a Complete Binary tree

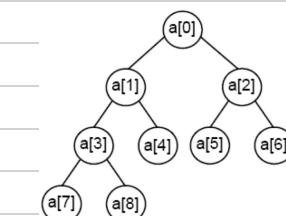
- Nodes of the tree are stored in the order in which they would be visited by a level-order traversal

Can construct array from binary tree from arr.  
Consider level-order and given complete tree

- The root Node is  $a[0]$

↳ Given the node  $a[i]$

- Left child is  $a[2^* i + 1]$
- Right child is  $a[2^* i + 2]$
- Parent is  $a[(i-1)/2]$ , for  $i > 0$



↔  $a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8]$

## Heaps for Priority Queues

- Heaps are a complete binary tree

↳ Constraint: A node has  $\geq 2$  kids of children ( $f = \infty$ )

- Largest value is always @ tree root

- Smallest value can be in any leaf node (no guarantee)

- For min-at-top heaps, any internal node is less than or equal to its children (not a big focus in this course)

- Uses Generics, so ( $item1 < item2$ ) compares memory locations not actual objects! & leads to unexpected errors

↳ Have to use Comparable

class Comparable implements Comparable<Comparable>

- May built-in classes have a compareTo() method?

public int compareTo(Comparable other)

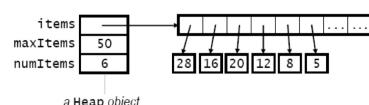
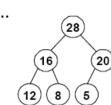
- Example on Point: the object in the heap is

referenced to external Comparable so  
it must have compareTo()

$T$ 's class has implements Comparable< $T$ >

public class Heap<T extends Comparable<T>> {

```
private T[] items;
private int maxItems;
private int numItems;
public Heap(int maxSize) {
    items = (T[])new Comparable[maxSize];
    maxItems = maxSize;
    numItems = 0;
}
```



a Heap object

## 1. Remove From a Heap

• [28, 16, 20, 12, 8, 5] Heap remove 28

[5, 16, 20, 12, 8]

Move 5 to front (now it's complete but not a heap)

↳ Then swap 5 with its children until its greater than all of them

Remove and return the item in the root node.

- In addition, we need to move the largest remaining item to the root, while maintaining a complete tree with each node  $\geq$  children

### 2. Remove an Item (the Greatest)

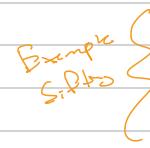
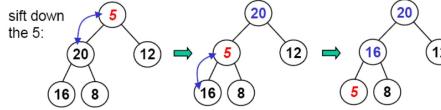
Remove and return the item in the root node.

- In addition, we need to move the largest remaining item to the root, while maintaining a complete tree with each node  $\geq$  children

Method:

- make a copy of the largest item
- move the last item in the heap to the root
- "sift down" the new root item until it is  $\geq$  its children (or it's a leaf)
- return the largest item

"sift": items are filtered such that small ones will fall



Once you have sift down method, remove is trivial

`public T removeMax()`

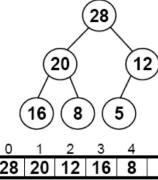
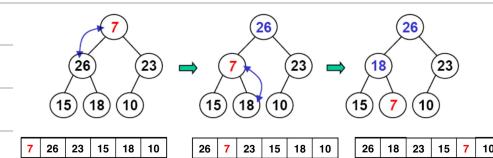
```
T toRemove = items[0];
items[0] = items[numItems-1];
numItems--;
siftDown(0);
return toRemove;
}
```

## Implementation of Sift Down Method

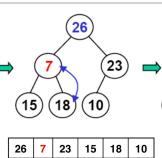
```
private void siftDown(int i) { // Input: the node to sift
    T toSift = items[i];
    int parent = i;
    int child = 2 * parent + 1; // Child to compare with; start with left child
    while (child < numItems) {
        // If the right child is bigger than the left one, use the right child instead.
        if (child + 1 < numItems && items[child + 1] < 0) // if the right child exists
            child = child + 1; // take the right child
        if (toSift.compareTo(items[child]) >= 0)
            break; // we're done
        items[parent] = items[child];
        parent = child;
        child = 2 * parent + 1;
    }
    items[parent] = toSift;
```

This is just a helper method for remove

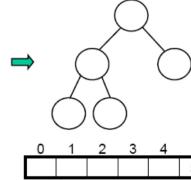
We don't have to put sifted item in place of child. We can wait until the end to put the sifted item in place.



numItems: 6  
toRemove: 28



numItems: 5  
toRemove: 28



numItems: 5  
toRemove: 28

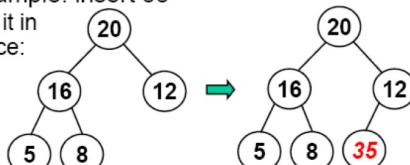
## Insertion into Heaps

- Assume there's additional space in array...
- Insert to end of arr, then sift up

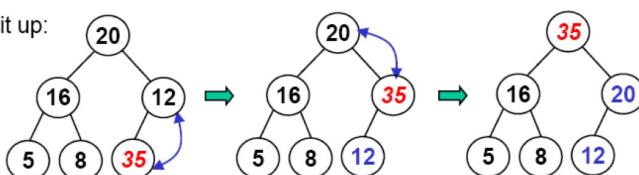
### Algorithm: & Quick Example

- put the item in the next available slot (grow array if needed)
- "sift up" the new item until it is  $\leq$  its parent (or becomes the root)

Example: insert 35  
put it in place:



sift it up:



`public void insert(T item) {`

```
if (numItems == maxItems) {
    // code to grow the array goes here...
}
items[numItems] = item;
numItems++;
siftUp(numItems-1);
```

Need to consider what happens if array is full

## Running Time Analysis

• Sift Up & Sift Down

→  $L = \log N$

→  $O(\log N)$  in worst case

• Insert / Remove are fast  $O(\log N)$

• Search is slow  $O(N \log N)$