Prep: bring ID, water, lots of paper, pen, watch and this set of notes
You got this!

Continued from Midterm 1 Sheet

Asymptotic Analysis

- Common Bounds

Equation	Bound	Equation	Bound
$T(N) = T\left(\frac{N}{2}\right) + \Theta(N)$	$\Theta(N)$	$\log 1 + \log 2 + \cdots + \log N = \log N!$	$\Theta(N \log N)$
$T(N) = 2T\left(\frac{N}{2}\right) + \Theta(N)$	$\Theta(N \log N)$	Stirling's Approximation: $\log N! = N \log N - N + O(\log N)$	

- Master's Theorem

$$T(N) = a\,T\left(\frac{N}{b}\right) + f(N)$$
$$c_{\text{crit}} = \log_b a$$

#	Description	Master Theorem Bound
1	$f(N) = O(N^c)$ where $c < c_{\text{crit}}$	$T(N) = \Theta(N^{c_{\text{crit}}})$
2	$f(N) = \Theta(N^{c_{\text{crit}}} \log^k N)$ where $k \geq 0$	$T(N) = \Theta(N^{c_{\text{crit}}} \log^{k+1} N)$
3	$f(N) = \Omega(N^c)$ where $c > c_{\text{crit}}$	No information; but if $af\left(\frac{N}{b}\right) < kf(N)$ for constant $k < 1$ and sufficiently large N, then $T(N) = \Theta\big(f(N)\big)$.

Bit Operations

Representations

Hexadecimal		Binary	
`0xdeadbeef`	`-559038737`	`0b1100100`	`100`
`0xdebe`	`57022`	`0b100`	`4`
`0xd`	`13`	`0b110`	`6`
`0xda`	`218 = 13*16 + 10*1 = 0xd*16 + 0xa*1`	`0b0`	`0`
`0x88888888`	`10001000100010001000100010001000`		
Trivia			
`-1`	`0b11111111111111111111111111111111`	`0xffffffff`	`-1`
`2147483647`	`0b01111111111111111111111111111111`	`0`	`(k >>> 31)`
`-2147483648`	`0b10000000000000000000000000000000`	`1`	`((-k) >>> 31)`
`-1`	`((-1) >> k)`	`-k`	`((-k) >> 32)`
`-1`	`((-k) >> 31)`		

Conclusion: `(x >>> 1)` right shifts `x` by inserting the sign bit on the left; `(x >> 1)` right shifts `x` by inserting `0` on the left.

Classics

Function	Examples	Code
Get least significant bit of x	$f(0b1100100) = 0b100$ $f(0b100) = 0b100 = 4$	`return ((x ^ (x-1)) + 1) >> 1;`
	$f(15) = 1$ $f(12) = 4$ $f(16) = 16$ $f(-10) = 2$	`return (x & (-x));` `// works for negative x`

Check if x is a (positive) power of 2	$f(2) = \text{true}$ $f(32) = \text{true}$	`(x > 0) && (x & (x - 1)) == 0`
Obtains $\lvert x \rvert$	$f(1) = 1$ $f(-1) = 1$	`int sign = x >> 31;` `return ((sign + x) ^ sign);`

Check if kth item is taken	$f(0b100, 2) = \text{true}$
	$f(0b100, 1) = \text{false}$
		`if ((mask & (1 << k)) > 0){`
		`    // logic`
		`}`

Set first k items as taken	$f(3) = 0b111$ $f(5) = 0b11111$	`int mask = (1 << k) – 1;` `return (x | mask);`
Set first k items as not taken	$f(5,2) = 0b100$ $f(15,2) = 0b1100$	`int mask = (1 << k) – 1;` `return (x & (~mask));`
Toggle first k items	$f(5,2) = 0b110$ $f(15,2) = 0b1100$	`int mask = (1 << k) – 1;` `return (x ^ mask);`
Set **only** kth item to be taken	$f(0b100, 2) = 0b100$ $f(0b100, 1) = 0b110$	`mask = mask | (1 << k);`
Set **only** kth item as not taken	$f(0b100, 2) = 0b000$ $f(0b100, 1) = 0b100$	`mask = mask & (~(1 << k));`
Modulo 2^k	$f(127,6) = 63$ $f(67,6) = 3$	`int mask = (1 << k) – 1;` `return (x & mask);`
Floor division by 2^k; $\left\lfloor \frac{x}{2^k} \right\rfloor$	$f(127,3) = 63$	`return (x >> k);`

Addition with only bit operations	`int add(int x, int y){`
	`    if (y == 0) return x;`
	`    return add(x ^ y, (x & y) << 1);`
	`}`
		`int add(int x, int y){`
		`    while (y != 0){`
		`        int carry = x & y;`
		`        x = x ^ y; y = carry << 1;`
		`    }`
		`}`

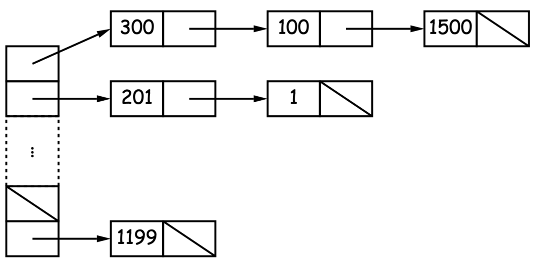
Remember to put brackets everywhere! And note that each hexadecimal digit occupies **four** binary digits.

Binary Search Trees

- **ALL** nodes in the left tree have smaller keys; **ALL** nodes in the right tree have larger keys
- Use `.compareTo()`
- Height of tree may **NOT** be $\log N$ (only for bushy BST); may reach max of $N - 1$ with ***ANY*** item at the root (e.g. the median)

Hashing

External Chaining



- M buckets
- $L = N/M$ load factor
- Keep M within constant factor of N
- If $L > threshold$, resize hash table and rehash all items.

Open Addressing

- Various ways to do this:
  - Linear probes: If there is a collision at $h(K)$, try $h(K)+m$, $h(K)+2m$, etc. (wrap around at end), until you find an empty bucket.
  - Quadratic probes: $h(K) + 1 \cdot m$, $h(K) + 2^2 \cdot m$, $h(K) + 3^2 \cdot m, \ldots$
  - Double hashing: $h(K) + h'(K)$, $h(K) + 2h'(K)$, etc.
- Example: $h(K) = K\%M$, with $M = 10$, linear probes with $m = 1$.
  - Add 1, 2, 11, 3, 102, 9, 18, 108, 309 to empty table.

108	1	2	11	3	102	309		18	9

- Put **only** one data item in each bucket
- If collision happens, just use another bucket (depending on implementation)

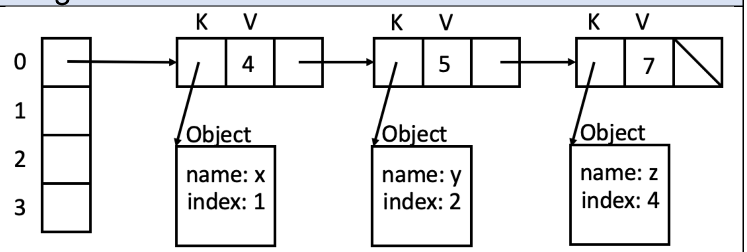
- Perfect Hashing: Tailor-made hash function that maps every key to a different value.

Line of Attack

- A good `.hashCode()` satisfies two properties:
  - Items that are equal according to `.equals()` **MUST** return the same hash so as to map to the same bucket
    - Corollary: value returned by `.hashCode()` must be deterministic
    - Corollary: if invoked on the same item more than once, must consistently return the same value
  - Good distribution of values (*)
    - Corollary: adding two "good" `.hashCode()` makes a good hash function
    - Corollary: xor-ing two "good" `.hashCode()` does not necessarily make a good hash function
- (*) Different items need not return distinct .hashCode() value, but a poor distribution increases search time
- By default, returns the identity hash function

- General algorithm for `put(K, V)` (credits to TA Sohum)
  - Compute `hashCode()` of `K`
  - Go to bucket with index `hashCode() % M` where `M` is number of buckets
  - Iterate through the linked list (or whatever data structure) of bucket
    - If `.equals(K)`, override value with `V` and `return;`
    - Else add new `(K, V)` pair at the end of the linked list (or whatever data structure)

HashSet (credits to TA Sohum)

`put(K, V)`	Diagram
• Compute `.hashCode()` of `K` • Go to bucket with index `hashCode() % M` where `M` is # of buckets • Iterate over the `LinkedList` (or whatever data structure) of the bucket • If `.equals(K)`, override value with `V` and `return;` • Else, add new `(K, V)` pair at the end of `LinkedList` (or whatever data struct)	

Heap (Max-Heap)

- Heap property: labels of both children of each node are less than node's label
- Completeness: No "gaps" in the tree.
- *Heapify* takes $\Theta(N)$

Trivia

1. `removeMin()` has best runtime of $\Theta(1)$ and worst runtime of $\Theta(\log N)$
2. `insert(obj)` has best runtime of $\Theta(1)$ and worst runtime of $\Theta(\log N)$
3. A complete binary tree with N levels has $2^N - 1$ elements
4. Given a min-heap of $2^N - 1$ distinct elements
   i. An element on second level is less than $2^{N-1} - 2$ elements and greater than 1 element
   ii. An element on bottommost level is less than 0 elements and greater than $N - 1$ elements

Generic Types

Type Bounds

Bound 1: T Must be a SubType of N (Type Bound)

```
class NSet<T extends N> extends HashSet<T>{
    T min(){ // logic }
}
```

Bound 2: Q Must be a supertype of T

```
static <T> void copy(List<? super T> dest, List<T> src){ // logic }
```

Bound 3: Subtype and Super type

```
static <T> int binarySearch(List<? extends Comparable<? super T> > L, T key){ // logic }
```

Note that L might not be able to contain the value key, but as long as can compare, can already.

Functions parameterized by type

```
static <T> List<T> singleton(T item){ // logic }

static <T> List<T> emptyList(){ // logic }
```

Wildcard

Don't care what a type parameter is (i.e. it can be any subtype of `Object`)

```
static int frequency(Collection<?> c, Object x){
    // logic
}
```

Subtyping

```
T1<X> ⊂ T2<Y> iff X = Y
T1<X> ⊂ T2<X> iff T1 ⊂ T2
```

Note the following exception:
`String[] ⊂ Object[]`

May result in `ArrayStoreException`;

Sorting Algorithms

Quicksort

- Select a pivot; partition everything $>$ pivot on the right side and $\leq$ pivot on the left side
- Recursively sort the two ends
- Stop when segments are small enough and proceed with insertion sort
- Good constant factor w.r.t. Merge Sort

16	10	13	18	-4	-7	12	-5	19	15	0	22	29	34	-1*
-4	-5	-7	-1	18	13	12	10	19	15	0	22	29	34	16*
-4	-5	-7	-1	15	13	12*	10	0	16	19*	22	29	34	18
-4	-5	-7	-1	10	0	12	15	13	16	18	19	29	34	22
-7	-5	-4	-1	0	10	12	13	15	16	18	19	22	29	34

- Quick selection: probabilistically $\Theta(N)$

Initial contents:

51	60	21	-4	37	4	49	10	40*	59	0	13	2	39	11	46	31

0

Looking for #10 to left of pivot 40:

13	31	21	-4	37	4*	11	10	39	2	0	40	59	51	49	46	60

0

Looking for #6 to right of pivot 4:

-4	0	2	4	37	13	11	10	39	21	31*	40	59	51	49	46	60

4

Looking for #1 to right of pivot 31:

-4	0	2	4	21	13	11	10	31	39	37	40	59	51	49	46	60

9

Just two elements; just sort and return #1:

-4	0	2	4	21	13	11	10	31	37	39	40	59	51	49	46	60

9

Merge Sort

- Split into two parts `L` and `R`, sort & merge

```
void merge(int arr[], int l, int m, int r){
  int n1 = m - l + 1, n2 = r - m;
  int[] L = new int[n1], R = new int[n2];

  for (int i = 0; i < n1; i++) L[i] = arr[l + i];
  for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

  int i = 0, j = 0, k = l;
  while (i < n1 && j < n2){
      if (L[i] <= R[j]){
          arr[k] = L[i]; i++;
      } else {
          arr[k] = R[j]; j++;
      }
      k++;
  }

  while (i < n1){
      arr[k] = L[i];
      i++; k++;
  }
  while (j < n2){
      arr[k] = R[j];
      j++; k++;
  }
}

void sort(int[] arr, int l, int r){
  if (l < r){
      int m = l + (r - l)/2;
      sort(arr, l, m);
      sort(arr, m + 1, l);
      merge(arr, l, m, r);
  }
}
```

Insertion Sort

- Each execution of `/* Step */` reduces the number of inversions by 1.

```
for (int i = 0; i < A.length; i++){
    int j;
    Object x = A[i];
    for (j = i - 1; j >= 0; j--){
```

Heapsort

- Bottom-up heapify $\Theta(N)$ (i.e. bubble down in reverse level order)
- Repeat `removeMax()`
- Sorts in-place (constant memory)

```
        if (A[j].compareTo(x) <= 0) break;
        A[j + 1] = A[j]; /* Step */
    }
    A[j + 1] = x;
}
```

Inbuilt Java Sort

```
import java.util.Arrays;
// P: primitive type, C: reference type implements Comparable, R: normal reference type

static void sort(P[] arr){ /*…*/ }
static void sort(P[] arr, int first, int end){ /*…*/ }

static <C extends Comparable<? super C> > sort(C[] arr){ /*…*/ }

static <R> void sort(R[] arr, Comparator<? super R> cmp){/* … */}
static <C extends Comparable<? super C> > sort(List<C> lst) {/*…*/}
static <R> void sort<List<R>, Comparator<? super R> comp) {/*…*/}

sort(X, (String x, String y) -> { return y.compareTo(x); });
sort(X, Collections.reverseOrder());
X.sort(Collections.reverseOrder()); // for X a List
```

```
void heapify(int[] arr){
    int N = arr.length;
    for (int k = N/2; k >= 0; k--){
        for (int p = k, c = 0; 2*p + 1 < N; p = c){
            c = 2*p + 1;
            if (arr[c] > arr[p]){
                int x = arr[p];
                arr[p] = arr[c];
                arr[c] = x;
            } else break;
        }
    }
}

for (int i = N – 1; i >= 0; i--){
    int x = removeMax();
    arr[i] = x;
}
```

Most Significant Digit (MSD) Sort

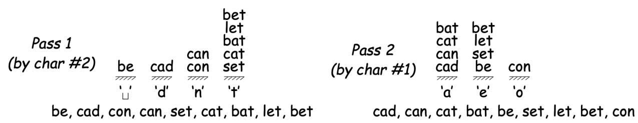
- Sort keys one character at a time from the most significant digit.
- Must keep lists from each step separate
- Takes $\Theta(B)$ where B is total key data
- High constant factor

A	posn
⋆ set, cat, cad, con, bat, can, be, let, bet	0
⋆ bat, be, bet / cat, cad, con, can / let / set	1
bat / ⋆ be, bet / cat, cad, con, can / let / set	2
bat / be / bet / ⋆ cat, cad, con, can / let / set	1
bat / be / bet / ⋆ cat, cad, can / con / let / set	2
bat / be / bet / cad / can / cat / con / let / set	

Least Significant Digit (LSD) Sort

- Sort keys one character at a time from the most significant digit.
- Must keep lists from each step separate
- Takes $\Theta(B)$ where B is total key data
- High constant factor

Initial: set, cat, cad, con, bat, can, be, let, bet



Counting Sort (Discussion) $\Theta(N)$

- Set of keys range from 0 to kN for some small constant k
- Initialize a `count` array of size kN that tracks the tally of each key.
- Iterate through the `count` array to position the keys correctly

Bubble Sort (Lab 8) $\Theta(N^2)$

- Repeatedly loop through the list
  - Swap adjacent out-of-order elements until the whole array is sorted

Runtime Analysis

Algorithm	Selection	Insertion [**]	Merge	Quick	Heap
Worst Case	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N \log N)$	$\Theta(N^2)$	$\Theta(N \log N)$
Best Case	$\Theta(N^2)$	$\Theta(N)$	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$ [*]
Stable?	No	Yes	Yes	No	No

[*] When all items are the same [**] More precisely, $\Theta(N + \#\text{ of inversions})$

Algorithm	Distribution Counting	MSD Radix	LSD Radix
Worst Case	$\Theta(N)$	$\Theta(NL)$	$\Theta(NL)$
Best Case	$\Theta(N)$	$\Theta(NL)$	$\Theta(NL)$
Stable?	Yes	Yes	Yes

Last Resorts

- `Math.max(int a, int b); Math.min(int a, int b);`
- Ternary operators: `return (x == null) ? 1 : 0;`
- Initialize values with assignments: `private int _size = 1;`
- Double assignments: `int x = 0, y = 0; x = y = 1;`
- Whatever works, rely on your instincts / brute force!

Analysis of Data Structures (for quick referencing)

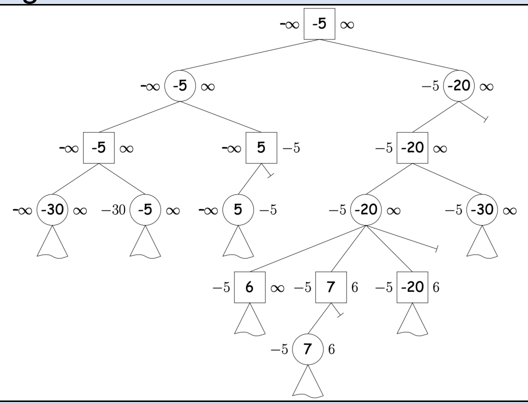
Function	Unordered List	Sorted Array	Bushy Search Tree	"Good" Hash Table	Heap
find	$\Theta(N)$	$\Theta(\log N)$	$\Theta(\log N)$	$\Theta(1)$	$\Theta(N)$
add	$\Theta(1)$	$\Theta(N)$	$\Theta(\log N)$	$\Theta(1)$	$\Theta(\log N)$
range query ($L \leq x \leq U$)	$\Theta(N)$	$\Theta(k + \log N)$	$\Theta(k + \log N)$	$\Theta(N)$	$\Theta(N)$
find largest	$\Theta(N)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(1)$
remove largest	$\Theta(N)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(\log N)$

Game Tree

Backtracking Algorithm

```
boolean findPath(Map visited, Game g){
    if (all_visited()) return check_state();
    for (ng){ // ng is a next game state from g
        if (!visited[ng]){
            visited[ng] = true;
            g.makeMove(x);
            if (findPath(visited, ng)) return true;
            g.undoMove(x);
            visited[ng] = false;
    }
    return false;
}
```

Diagram



$\alpha - \beta$ pruning (Maximizing Player)

```
int maxPlayerValue(Position posn, int depth, int
alpha, int beta){
     if (depth == 0 || <final_state>) return
staticGuess(posn);
    int bestSoFar = -inf;
    for (legal_move in posn){
        Position next = makeMove(posn, M);
        int response = minPlayerValue(next, depth –
1, alpha, beta);
        if (response > bestSoFar){
            bestSoFar = response;
            alpha = max(alpha, bestSoFar);
            if (alpha >= beta) return bestSoFar;
        }
    }
    return bestSoFar;
}
```

$\alpha - \beta$ pruning (Minimizing Player)

```
int maxPlayerValue(Position posn, int depth, int
alpha, int beta){
     if (depth == 0 || <final_state>) return
staticGuess(posn);
    int bestSoFar = inf;
    for (legal_move in posn){
        Position next = makeMove(posn, M);
        int response = maxPlayerValue(next, depth –
1, alpha, beta);
        if (response < bestSoFar){
            bestSoFar = response;
            beta = min(beta, bestSoFar);
            if (alpha >= beta) return bestSoFar;
        }
    }
    return bestSoFar;
}
```

Final Checks

- Read all examples and problem statement properly!
- What is the purpose of a method? Exploit previous methods.
- Check edge cases: empty lists, base cases, `null` values. **NEVER** call method on `null`!
- Check return value of functions; sometimes hints at assignment: `_left = _left.add(item);`
- Common problem setter tricks: variable not reset; continuous variable; exhausted variable
- Asymptotic analysis tricks:
  - With data structures: `list.get(x)` takes $\Theta(N)$
  - Drop constants but not N; in particular $\Theta(N^{N-1}) \neq \Theta(N^N)$
- Check arguments when calling recursive functions! Did you miss out any arguments?

Appendix IV: Data Structures II

Tree (Preorder, Inorder, Postorder DFS, BFS)

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.function.Consumer;
import java.util.Stack;
import java.util.ArrayDeque;
import java.util.Iterator;

public class AdvTree<T> implements Iterable<T> {
    @SuppressWarnings("unchecked")
    public AdvTree(T label, AdvTree<T>... children){
        _label = label;
        if (children == null){
            _children = null;
        } else {
            _children = new
ArrayList<>(Arrays.asList(children));
        }
    }

    public int arity() {
        if (_children == null) return 0;
        return _children.size();
    }

    public T label() { return _label; }

    public AdvTree<T> child(int k) {
        if (_children == null) return null;
        if (k < 0 || k >= _children.size()) return
null;
        return _children.get(k); }

    public static <T> void preorder(AdvTree<T> t,
Consumer<AdvTree<T> > x){
        x.accept(t);
        for (int i = 0; i < t.arity(); i++){
            preorder(t.child(i), x);
        }
    }

    public static <T> void inorder(AdvTree<T> t,
Consumer<AdvTree<T> > x){
        if (t == null) return;
        inorder(t.child(0), x);
        x.accept(t);
        inorder(t.child(1), x);
    }

    public static <T> void postorder(AdvTree<T> t,
Consumer<AdvTree<T> > x){
        for (int i = 0; i < t.arity(); i++){
            postorder(t.child(i), x);
        }
        x.accept(t);
    }

    public static<T> void idfs(AdvTree<T> t,
Consumer<AdvTree<T> > visit){
        Stack<AdvTree<T> > work = new Stack<>();
        work.push(t);
        while (!work.isEmpty()){
            AdvTree<T> node = work.pop();
            visit.accept(node);
            for (int i = node.arity() - 1; i >= 0;
i--){
                work.push(node.child(i));
            }
        }
    }

    public static<T> void bfs(AdvTree<T> t,
Consumer<AdvTree<T> > visit){
        ArrayDeque<AdvTree<T> > work = new
ArrayDeque<>();
        work.push(t);
        while (!work.isEmpty()){
            AdvTree<T> node = work.removeFirst();
            if (node != null){
```

Stack

```
import java.util.Stack;

Stack<String> s = new Stack<String>();

s.push("rn");
s.push("jz");

System.out.println(s.peek()); // jz
System.out.println(s.size()); // 2
System.out.println(s.empty()); // false

System.out.println(s.pop()); // jz
// throws EmptyStackException if s is empty

s.clear();

System.out.println(s.contains("rn")); // true
```

Deque (Interface)

Note: `LinkedList` and `ArrayDeque` implements `Deque`.

```
import java.util.Deque;


Deque<String> dq = new LinkedList<String>();
dq.add("jz"); // add at back
dq.addFirst("jj"); // add at front
dq.addLast("rn"); // add at back
dq.push("ag"); // add at front
dq.offer("jw"); // add at back
dq.offerFirst("zn"); // add at front
dq.offerLast("br"); // add at back

dq.removeFirst();
dq.removeLast();
dq.pop();
dq.poll();
dq.pollFirst();
dq.pollLast(); // null

for (Iterator<String> it = dq.iterator();
it.hasNext();){
    System.out.println(it.next());
}

dq.getFirst();
dq.getLast();
dq.peekFirst();
dq.peekLast();

dq.size();
```

Class

```
Class c = Dog.class;
A a = new A();
Class c = a.getClass();

Class<T> ?!
```

Misc

```
String h = "22"
Integer.parseInt(h);

Integer x = 9;
String.valueOf(x);
```

import static java.util.Collections.max;
import static java.util.Collections.min;

Map / HashMap (interface in java.util)

```
import java.util.*;

Map<String, Integer> hm = new HashMap<String,
Integer>();

hm.put("a", new Integer(100));
hm.put("b", new Integer(100));
```

```java
                visit.accept(node);
                for (int i = 0; i < node.arity();
i++){
                    work.addLast(node.child(i));
                }
            }
        }
    }

    public static<T> void iddfs(AdvTree<T> t, int
level, Consumer<AdvTree<T> > visit){
        if (level == 0){
            visit.accept(t);
            return;
        }
        for (int i = 0; i < t.arity(); i++){
            iddfs(t.child(i), level - 1, visit);
        }
    }

    static class PreorderIterator<T> implements
Iterator<T> {
        private Stack<AdvTree<T> > s = new
Stack<AdvTree<T> >();
        public PreorderIterator(AdvTree<T> t) {
s.push(t); }
        public boolean hasNext() { return
!s.isEmpty(); }
        public T next(){
            AdvTree<T> result = s.pop();
            for (int i = result.arity() - 1; i >= 0;
i--){
                s.push(result.child(i));
            }
            return result.label();
        }
    }

    @Override
    public Iterator<T> iterator(){
        return new PreorderIterator(this);
    }

    private T _label;
    private ArrayList<AdvTree<T> > _children;

    public static void main(String[] args){
        AdvTree<String> t11 = new
AdvTree<String>("ji", (AdvTree<String>[]) null);
        AdvTree<String> t12 = new
AdvTree<String>("an", (AdvTree<String>[]) null);
        AdvTree<String> t1 = new
AdvTree<String>("jian", new AdvTree[]{t11, t12});

        AdvTree<String> t21 = new
AdvTree<String>("zh", (AdvTree<String>[]) null);
        AdvTree<String> t22 = new
AdvTree<String>("i", (AdvTree<String>[]) null);
        AdvTree<String> t2 = new
AdvTree<String>("zhi", new AdvTree[]{t21, t22});
        AdvTree<String> t = new
AdvTree<String>("jianzhi", new AdvTree[]{t1, t2});

        System.out.println("preorder");
        preorder(t, a ->
System.out.println(a.label()));
        System.out.println();

        System.out.println("inorder");
        inorder(t, a ->
System.out.println(a.label()));
        System.out.println();

        System.out.println("postorder");
        postorder(t, a ->
System.out.println(a.label()));
        System.out.println();

        System.out.println("iterative dfs");
        idfs(t, a -> System.out.println(a.label()));
        System.out.println();

        System.out.println("bfs");
```

```java
for (Map.Entry<String, Integer> me: hm.entrySet()){
    System.out.print(me.getKey() + " : ");
    System.out.println(me.getValue());
}

hm.remove("a");

hm.clear();

hm.containsKey("b");
hm.containsValue(100);
hm.get("b");
hm.isEmpty();
hm.size();
hm.values();


hm.keySet();
```

Binary Search Tree (BST)

```java
class BST<Key extends Comparable<Key>, Value>{

    Key _key;
    Value _value;
    BST<Key, Value> _left, _right;

    BST(Key key0, Value value0, BST<Key, Value>
left0, BST<Key, Value> right0){
        _key = key0; _value = value0;
        _left = left0; _right = right0;
    }

    BST(Key key0, Value value0){
        this(key0, value0, null, null);
    }

    static <Key extends Comparable<Key>, Value>
BST<Key, Value> find(BST<Key, Value> T, Key L){
        if (T == null) return T;
        if (L.compareTo(T._key) == 0) return T;
        else if (L.compareTo(T._key) < 0) return
find(T._left, L);
        else return find(T._right, L);
    }


    static <Key extends Comparable<Key>, Value>
BST<Key, Value> insert(BST<Key, Value> T, Key k,
Value v){
        if (T == null) return new BST(k, v);
        if (k.compareTo(T._key) == 0) T._value = v;
        else if (k.compareTo(T._key) < 0) T._left =
insert(T._left, k, v);
        else if (k.compareTo(T._key) > 0) T._right =
insert(T._right, k, v);
        return T;
    }

    static <Key extends Comparable<Key>, Value>
BST<Key, Value> minNode(BST<Key, Value> T){
        if (T._left == null) return T;
        return minNode(T._left);
    }

    static <Key extends Comparable<Key>, Value>
BST<Key, Value> remove(BST<Key, Value> T, Key k){
        if (T == null) return null;
        if (k.compareTo(T._key) == 0){
            if (T._left == null) return T._right;
            else if (T._right == null) return
T._left;
            else {
                BST<Key, Value> smallest =
minNode(T._right);
                T._value = smallest._value;
                T._key = smallest._key;
                T._right = remove(T._right,
smallest._key);
            }
        } else if (k.compareTo(T._key) < 0){
            T._left = remove(T._left, k);
        } else {
```

```
        bfs(t, a -> System.out.println(a.label()));
        System.out.println();

        System.out.println("iterative deepening
dfs");
        System.out.println();
        for (int i = 0; i < 3; i++){
            System.out.println("level " + i);
            iddfs(t, i, a ->
System.out.println(a.label()));
            System.out.println();
        }

        System.out.println("iterator");
        PreorderIterator<String> it = new
PreorderIterator<>(t);
        for (; it.hasNext();){
            System.out.println(it.next());
        }
        System.out.println();

        System.out.println("iterable");
        for (String x : t){
            System.out.println(x);
        }

    }

}
```

```
            T._right = remove(T._right, k);
        }
        return T;
    }

    public static void main(String[] args){
        System.out.println("hello");
        BST<Integer, String> t = new BST<Integer,
String>(2, "jianzhi");
        insert(t, 1, "jz");
        insert(t, 0, "rn");
        insert(t, -1, "jw");
        insert(t, 5, "zn");
        insert(t, 4, "br");
        insert(t, 8, "zy");
        insert(t, 3, "ch");

        System.out.println("finding jz: " + (find(t,
2) != null));
        System.out.println("finding br: " + (find(t,
4) != null));
        remove(t, 4);
        System.out.println("finding br: " + (find(t,
4) != null));

    }
}
```

Appendix V: Regex II

Sample Code:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class Patt {
    public static void main(String[] args){
        String patn = "(jz\\d+)";
        String S = "jz";
        Matcher m = Pattern.compile(patn).matcher(S);
        System.out.println(m.matches());
        if (m.matches()){
            System.out.println(m.group(1));
        }
    }
}
```

<table>
<tr><td colspan="4">Classic</td></tr>
<tr><td>.</td><td>Matches any character</td><td>Matches</td><td>Do Not Match</td></tr>
<tr><td>^str</td><td>str must be the start of the string</td><td></td><td></td></tr>
<tr><td>str$</td><td>str must be the end of the string</td><td></td><td></td></tr>
<tr><td>[abc]</td><td>match any character abc</td><td></td><td></td></tr>
<tr><td>[abc][vz]</td><td>Match a, b, c followed by v, z</td><td>"av"</td><td>"ax"</td></tr>
<tr><td>[abc][^vz]</td><td>Match a, b, c followed by anything except v, z</td><td>"a*", "ax"</td><td>"av", "a**"</td></tr>
<tr><td>[a-d][1-7]</td><td>Matches a to d, followed by 1 to 7 (inclusive)</td><td>"d1"</td><td>"a8", "d0"</td></tr>
<tr><td>[a-d][^1-7]</td><td>Matches a to d, followed by any not from 1 to 7</td><td>"a0"</td><td>"a1"</td></tr>
<tr><td>jz|rn</td><td>Matches 'jz' or 'rn'</td><td>"jz" "rn"</td><td>"jn"</td></tr>
<tr><td>X|Z</td><td>X directly followed by Z</td><td></td><td></td></tr>
<tr><td>XZ</td><td>X directly followed by Z</td><td></td><td></td></tr>
<tr><td colspan="4">Meta</td></tr>
<tr><td>\\d</td><td>Digit, equivalent to [0-9]</td><td>"1"</td><td>"11"</td></tr>
<tr><td>\\D</td><td>Non-digit, equivalent to [^0-9]</td><td></td><td></td></tr>
<tr><td>\\s</td><td>Whitespace char; equivalent to [\t\n\x0b\r\f]</td><td>" "</td><td>"*"</td></tr>
<tr><td>\\S</td><td>Non whitespace; [^ \t\n\x0b\r\f]</td><td>"A"</td><td>" "</td></tr>
<tr><td>\\w</td><td>Alphanumeric char; equivalent to [a-zA-Z0-9]</td><td></td><td></td></tr>
<tr><td>\\W</td><td>Non alphanumeric character; equivalent to [^a-zA-Z0-9] or [^\\w]</td><td>"*"</td><td>"A"</td></tr>
<tr><td colspan="4">Quantifiers</td></tr>
<tr><td>{X}</td><td rowspan="2">Occurs X number of times</td><td rowspan="2">"285"</td><td rowspan="2">"61B"</td></tr>
<tr><td>\\d{3}</td></tr>
<tr><td>{X,Y}</td><td>Occurs between X and Y times</td><td rowspan="2">"285", "3233"</td><td rowspan="2">"CS70", "14641"</td></tr>
<tr><td>\\d{1,4}</td><td>Matches minimum 1 and maximum of 4 digits</td></tr>
<tr><td>?</td><td rowspan="2">Occurs zero or one time; short for {0,1}</td><td rowspan="2">"r", ""</td><td rowspan="2">"rr", "a"</td></tr>
<tr><td>r?</td></tr>
<tr><td>+</td><td rowspan="2">Occurs one or more times; short for {1,}</td><td rowspan="2">"rrr", "rr"</td><td rowspan="2">"", "a", "ra"</td></tr>
<tr><td>r+</td></tr>
<tr><td>*</td><td rowspan="2">Occurs zero or more times; short for {0,}</td><td rowspan="2">"", "rrrr"</td><td rowspan="2">"a", "ra"</td></tr>
<tr><td>*r</td></tr>
<tr><td>*?</td><td colspan="3">? after a quantifier makes it reluctant (non-greedy) matches as little as possible before backtracking. Does not affect result.</td></tr>
<tr><td colspan="4">Grouping</td></tr>
</table>

<table>
<tr><td><code>()()</code></td><td>creates a back reference. is captured during matching</td></tr>
<tr><td colspan="2"><code>res.replaceFirst(patn, “$1”)</code>
<code>// replace the first match of patn by the first group</code> <code>(1-indexed)</code></td></tr>
</table>

<table>
<tr><td colspan="4">Basic Examples</td></tr>
<tr><td><code>[tT]rue|[yY]es</code></td><td><code>“True”, “yes"</code></td><td><code>[^0-9]*[12]?[0-9]{1,2}[^0-9]</code></td><td>A string that has a number less than <code>300</code></td></tr>
<tr><td><code>.*true.*</code></td><td><code>“altrueistic”, “true”</code></td><td><code>([\\w&&[^b]])*</code></td><td>A string with arbitrary # of characters except <code>b</code></td></tr>
<tr><td><code>[a-zA-Z]{3}</code></td><td><code>“CaT”, “dEl”</code></td><td><code>(jin|joe)</code></td><td><code>“jin”, “joe”</code></td></tr>
<tr><td><code>^[^\\d].*</code></td><td>Any string that does not begin with a number</td><td><code>\\b(\\w+)\\s+\\1\\b</code></td><td>Finds duplicated words</td></tr>
</table>

More Examples

<table>
<tr><td>A valid date of the form MM/DD/YYYY e.g. 9/22/2019</td><td><code>P1 = "([1-9]|0[1-9]|1[0-2])/(0[1-9]|[1-9]|[1-2][0-9]|30|31)/(19[0-9][0-9]|[2-9][0-9][0-9][0-9])";</code></td></tr>
<tr><td>A match for CS61B notation for literal <code>IntLists</code></td><td><code>P2 = "[(]((\\d+, +)*)(\\d+)[)]";</code></td></tr>
<tr><td>A valid domain name, e.g. www.cs61b.com</td><td><code>P3 = "([a-zA-Z0-9]+[\\-\\.])+[a-zA-Z0-9]{2,6}";</code></td></tr>
<tr><td>A valid Java variable name. e.g. <code>_child13$</code></td><td><code>P4 = "[a-zA-Z$_][a-zA-Z0-9$_]*";</code></td></tr>
<tr><td>A valid IPv4 address e.g. <code>127.0.0.1</code></td><td><code>P5 = "([0-9]|[0-9][0-9]|0[0-9][0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\\.([0-9]|[0-9][0-9]|0[0-9][0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\\.([0-9]|[0-9][0-9]|0[0-9][0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\\.([0-9]|[0-9][0-9]|0[0-9][0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])";</code></td></tr>
<tr><td><code>"words 10,7,9,31,22, words 19,5,29,48,30."</code>
<code>m.group(1) = “10,7,9,31,22”</code></td><td><code>P6 = ".*?((\\d+,)*\\d+).*?"</code></td></tr>
</table>

Note: do not indiscriminately insert spaces “ “; otherwise the space will be matched