13 A: External Algorithms; Disjoint Sets; Java API Support

CS1102S: Data Structures and Algorithms

Martin Henz

April 14, 2010

Internal Storage

Assumption so far: random-access memory

Memory can be read and written at a speed $O(1)$

Internal Storage

Assumption so far: random-access memory

Memory can be read and written at a speed $O(1)$

Abstraction

Even for main memory accessible to the CPU through a fast data bus, this is a coarse simplification

Internal Storage

Assumption so far: random-access memory

Memory can be read and written at a speed $O(1)$

Abstraction

Even for main memory accessible to the CPU through a fast data bus, this is a coarse simplification

Complicating factors

- very fast memory on the CPU: registers
- cache hierarchy: layers of larger and slower memory units between CPU and main memory chips
- virtual memory: disk memory used when required memory exceeds physically available main memory

External Storage

Internal vs external storage

Internal storage is governed by electricity;

External Storage

Internal vs external storage

Internal storage is governed by electricity;
external storage is governed by mechanics

External Storage

Internal vs external storage

Internal storage is governed by electricity;
external storage is governed by mechanics

Disk storage

Access time depends on speed of disk, e.g. 7,200 RPM

External Storage

Internal vs external storage

Internal storage is governed by electricity;
external storage is governed by mechanics

Disk storage

Access time depends on speed of disk, e.g. 7,200 RPM

How many disk accesses possible per second?

External Storage

Internal vs external storage

Internal storage is governed by electricity;
external storage is governed by mechanics

Disk storage

Access time depends on speed of disk, e.g. 7,200 RPM

How many disk accesses possible per second?

120 accesses per second

External Storage

Internal vs external storage

Internal storage is governed by electricity;
external storage is governed by mechanics

Disk storage

Access time depends on speed of disk, e.g. 7,200 RPM

How many disk accesses possible per second?

120 accesses per second

How many instructions are executed per minute?

External Storage

Internal vs external storage

Internal storage is governed by electricity;
external storage is governed by mechanics

Disk storage

Access time depends on speed of disk, e.g. 7,200 RPM

How many disk accesses possible per second?

120 accesses per second

How many instructions are executed per minute?

With 1.2-GIPS processor, we have 1,2 billion instructions per second,

External Storage

Internal vs external storage

Internal storage is governed by electricity;
external storage is governed by mechanics

Disk storage

Access time depends on speed of disk, e.g. 7,200 RPM

How many disk accesses possible per second?

120 accesses per second

How many instructions are executed per minute?

With 1.2-GIPS processor, we have 1,2 billion instructions per second, a factor of 10,000,000 faster than disks!

Another Characteristics of Disk Storage

We know already

Access *time* is limited by mechanics

Another Characteristics of Disk Storage

We know already

Access *time* is limited by mechanics

How about access *size*?

Another Characteristics of Disk Storage

We know already

Access *time* is limited by mechanics

How about access *size*?

defined by operating system/hardware design;

Another Characteristics of Disk Storage

We know already

Access *time* is limited by mechanics

How about access *size*?

defined by operating system/hardware design;
typically large "chunks", called *blocks*, of data can be read very fast, once the disk head has reached the correct location

Another Characteristics of Disk Storage

We know already

Access *time* is limited by mechanics

How about access *size*?

defined by operating system/hardware design;
typically large "chunks", called *blocks*, of data can be read very fast, once the disk head has reached the correct location

Reading and writing in blocks

Reading and writing is typically done in large blocks to take advantage of this feature of disk storage

Search Trees—Revisited

Setup

We would like to quickly find out if a given data item is included in a collection.

Search Trees—Revisited

Setup

We would like to quickly find out if a given data item is included in a collection.

Example

In an underground carpark, a system captures the licence plate numbers of incoming and outgoing cars.

Search Trees—Revisited

Setup

We would like to quickly find out if a given data item is included in a collection.

Example

In an underground carpark, a system captures the licence plate numbers of incoming and outgoing cars.
Problem: Find out if a particular car is in the carpark.

Binary Search

Setup

Keep items in a tree. Each node holds one data item.

Binary Search

Setup

Keep items in a tree. Each node holds one data item.

Idea

The left subtree of a node V only contains items smaller than V and the right subtree only contains items larger than V.

Binary Search

Setup

Keep items in a tree. Each node holds one data item.

Idea

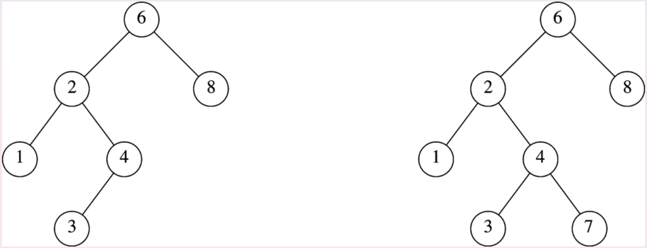
The left subtree of a node V only contains items smaller than V and the right subtree only contains items larger than V.

Search

can then proceed top-down, starting at the root. If the search item is smaller than the item at the root, go down to the left, and if it is larger, go right.

Example

Both trees are binary trees, but only the left tree is a search tree.

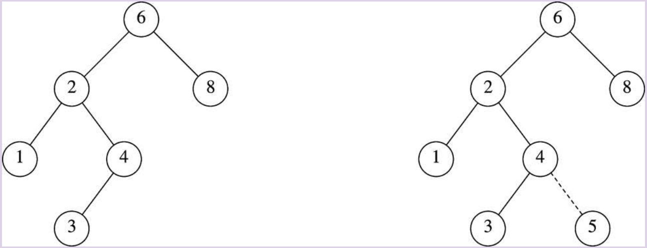


Insertion

Idea

Proceed like in search. If item is found, do nothing. If not, insert it in the last visited position.

Example



Deletion

Idea

Proceed like in search. If item is not found, do nothing. If item is found, take action depending on node.

Deletion

Idea

Proceed like in search. If item is not found, do nothing. If item is found, take action depending on node.

Leaf

If the node is leaf, delete it from parent.

Deletion

Idea

Proceed like in search. If item is not found, do nothing. If item is found, take action depending on node.

Leaf

If the node is leaf, delete it from parent.

One child

If the node has one child, move the child to parent.

Average-case Analysis

Average Depth

If all insertion sequences are equally likely, the average depth of any node is $O(\log N)$

Average-case Analysis

Average Depth

If all insertion sequences are equally likely, the average depth of any node is $O(\log N)$

Deletion introduces imbalance

Deletion favours right subtree, and therefore trees become "left-heavy" on the long run.

A Cure: AVL Trees

Worst-case depth

We want to restrict all operations to $O(\log N)$ in the *worst* case.

A Cure: AVL Trees

Worst-case depth

We want to restrict all operations to $O(\log N)$ in the *worst* case.

AVL Trees

Make sure that the height of the subtrees of any node differ by at most one (Adelson-Velskii and Landis), using rebalancing if necessary

A Cure: AVL Trees

Worst-case depth

We want to restrict all operations to $O(\log N)$ in the *worst* case.

AVL Trees

Make sure that the height of the subtrees of any node differ by at most one (Adelson-Velskii and Landis), using rebalancing if necessary

Bound

The height of an AVL tree is at most $1.44\log(N+2) - 1.328$, thus $O(\log N)$. In practice, the height is only slightly more than $\log N$.

Search Trees with External Storage

Main issue

When data does not fit in main memory, the number of block accesses needs to be minimized

Search Trees with External Storage

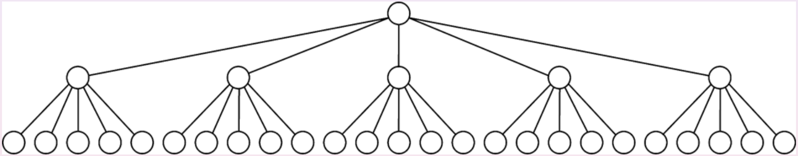
Main issue

When data does not fit in main memory, the number of block accesses needs to be minimized

Overall idea

Put more data into each node; use $n-ary$ trees instead of binary trees

Example of 5-ary Tree

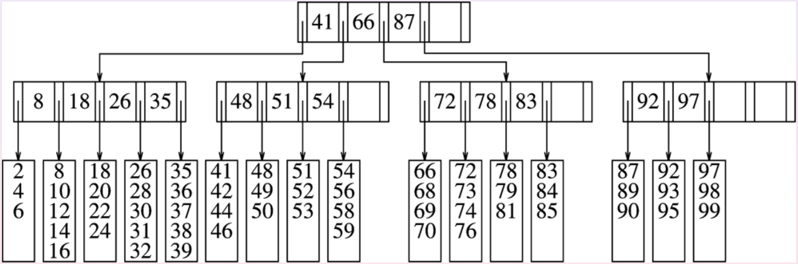


B-tree Definition

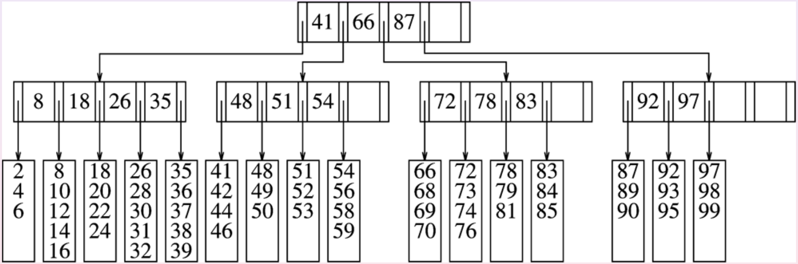
A B-tree of order M is an M-ary tree with the following properties:

1. Data items are stored at leaves
2. Nonleaf nodes store up to $M - 1$ keys to guide search; key i represents smallest key in subtree $i + 1$
3. Root is either a leaf or has between two and M children
4. Non-leaf non-root nodes have between $\lceil M/2 \rceil$ and M children
5. Leaves are at same depth, have between $\lceil L/2 \rceil$ and L children

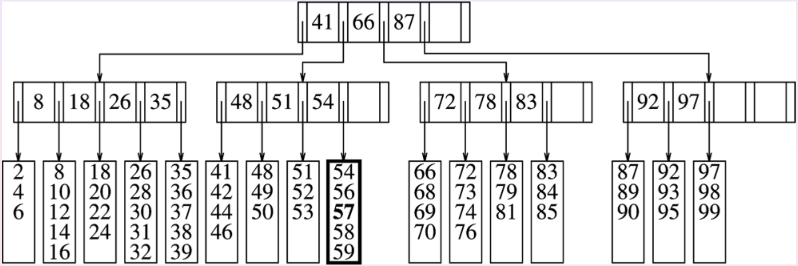
Example of B-Tree of Order 5



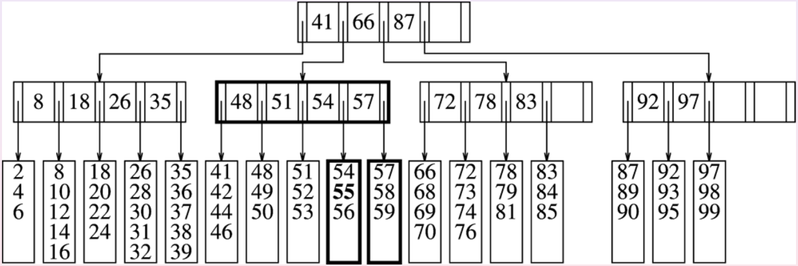
B-Tree Before Insertion of 57



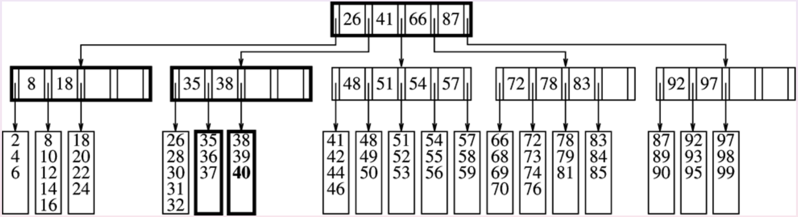
B-Tree After Insertion of 57



Insertion of 55 Causes Split



Insertion of 40 Causes Two Splits



What if a Split Reaches the Root?

Splitting root is allowed

Create a new root, and have the two halves as children

What if a Split Reaches the Root?

Splitting root is allowed

Create a new root, and have the two halves as children

Exception in definition makes sense

Root can have between 2 and M children

What if a Split Reaches the Root?

Splitting root is allowed

Create a new root, and have the two halves as children

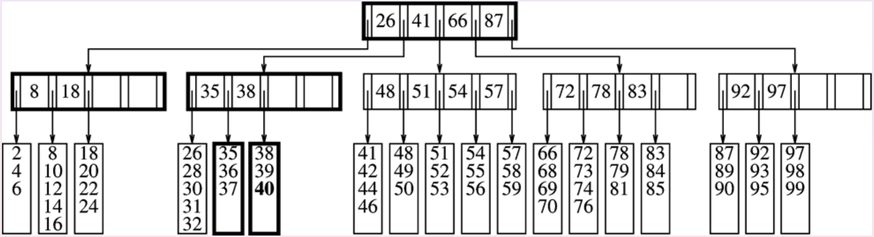
Exception in definition makes sense

Root can have between 2 and M children

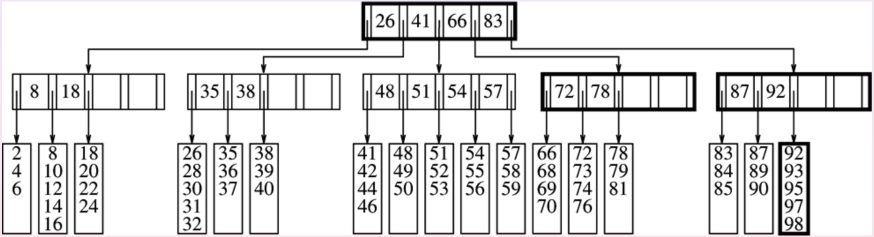
Growing B-trees

Splitting root as result of insertion is the *only* way that a B-tree can gain height

Before Deletion of 99



After Deletion of 99



Tapes as Storage

Similar to disks

Access time many orders of magnitude slower than main memory

Tapes as Storage

Similar to disks

Access time many orders of magnitude slower than main memory

Additional characteristics

Large amounts of data can be read *sequentially* quite efficiently

Tapes as Storage

Similar to disks

Access time many orders of magnitude slower than main memory

Additional characteristics

Large amounts of data can be read *sequentially* quite efficiently

Access of previous locations

Tapes as Storage

Similar to disks

Access time many orders of magnitude slower than main memory

Additional characteristics

Large amounts of data can be read *sequentially* quite efficiently

Access of previous locations

is *extremely* slow, as it requires re-winding the tape!

External Sorting

Main idea

Use tapes sequentially, and read one block from each input tape tape

External Sorting

Main idea

Use tapes sequentially, and read one block from each input tape tape

Merge blocks

Sort the blocks

Use merge procedure from mergesort to merge

The Simple Algorithm: Overview

Four tapes

Two input tapes; two output tapes

The Simple Algorithm: Overview

Four tapes

Two input tapes; two output tapes

Read and write runs

Read runs from input tape, sort them and write alternatively to output tapes

The Simple Algorithm: Overview

Four tapes

Two input tapes; two output tapes

Read and write runs

Read runs from input tape, sort them and write alternatively to output tapes

Continue, writing larger runs

Read two runs from each "output" tape, and merge them on the fly, writing alternatively to "input" tapes

The Simple Algorithm: Overview

Four tapes

Two input tapes; two output tapes

Read and write runs

Read runs from input tape, sort them and write alternatively to output tapes

Continue, writing larger runs

Read two runs from each “output” tape, and merge them on the fly, writing alternatively to “input” tapes

Continue

until one tape has all sorted data

Multiway Merge

Why only four tapes?

If we have more than four tapes, we can take advantage of them by using *multiway merge*

Multiway Merge

Why only four tapes?

If we have more than four tapes, we can take advantage of them by using *multiway merge*

How finding the smallest element during merge?

Multiway Merge

Why only four tapes?

If we have more than four tapes, we can take advantage of them by using *multiway merge*

How finding the smallest element during merge?

Priority queue!

Multiway Merge

Why only four tapes?

If we have more than four tapes, we can take advantage of them by using *multiway merge*

How finding the smallest element during merge?

Priority queue!

Each iteration of inner loop

deleteMin to find smallest element

Multiway Merge

Why only four tapes?

If we have more than four tapes, we can take advantage of them by using *multiway merge*

How finding the smallest element during merge?

Priority queue!

Each iteration of inner loop

deleteMin to find smallest element
insert new element from tape from which element was deleted

Polyphase Merge and Replacement Selection

Polyphase merge: main idea

Make use of fewer tapes, by re-using tapes for reading and writing

Polyphase Merge and Replacement Selection

Polyphase merge: main idea

Make use of fewer tapes, by re-using tapes for reading and writing

Leading to tape organization using kth order Fibonacci numbers

Polyphase Merge and Replacement Selection

Polyphase merge: main idea

Make use of fewer tapes, by re-using tapes for reading and writing

Leading to tape organization using kth order Fibonacci numbers

Replacement selection: main idea

Make use of input tape as output tape, reusing the tapes “on the fly”

1. External Search Trees: B-Trees
2. External Sorting
3. Disjoint Sets
   - Equivalence Relations
   - The Dynamic Equivalence Problem
   - Basic Data Structure
   - Variants
4. Java API Support for Data Structures
5. Another Puzzler

Equivalence Relations

Definition

An *equivalence relation* is a relation R that satisfies three properties:

1. (Reflexive) aRa, for all $a \in S$.
2. (Symmetric) aRb if and only if bRa.
3. (Transitive) aRb and bRc implies aRc.

Equivalence Relations

Definition

An *equivalence relation* is a relation R that satisfies three properties:

1. (Reflexive) aRa, for all $a \in S$.
2. (Symmetric) aRb if and only if bRa.
3. (Transitive) aRb and bRc implies aRc.

Examples

- Electrical connectivity (metal wires between points)
- Cities belonging to same country

The Dynamic Equivalence Problem

Initial setup

Collection of N disjoint sets, each with one element

The Dynamic Equivalence Problem

Initial setup

Collection of N disjoint sets, each with one element

Operations

- $find(a)$: return the set of which x is element
- $union(a, b)$: merge the sets to which a and b belong, so that $find(a) = find(b)$

Strategies

Fast Find, Slow Union

Use array *repres* to store equivalence class for each element

Strategies

Fast Find, Slow Union

Use array *repres* to store equivalence class for each element

- $find(a)$: return $repres[a]$
- $union(a, b)$: if $repres[x] = repres[b]$ then set $repres[x]$ to $repres[a]$

Strategies

Fast Find, Slow Union

Use array *repres* to store equivalence class for each element

- $find(a)$: return $repres[a]$
- $union(a, b)$: if $repres[x] = repres[b]$ then set $repres[x]$ to $repres[a]$

Fast Union, Reasonable Find

Union/find data structure

Basic Data Structure

Idea

Maintain *forest* corresponding to equivalence relation

Basic Data Structure

Idea

Maintain *forest* corresponding to equivalence relation

Union

Merge trees

Basic Data Structure

Idea

Maintain *forest* corresponding to equivalence relation

Union

Merge trees

Find

Return root of tree

Basic Data Structure

Idea

Maintain *forest* corresponding to equivalence relation

Union

Merge trees

Find

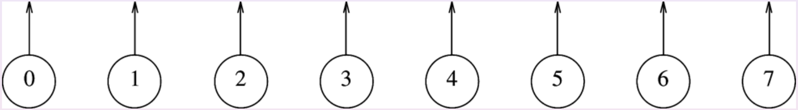
Return root of tree

Observe

Only upward direction needed!

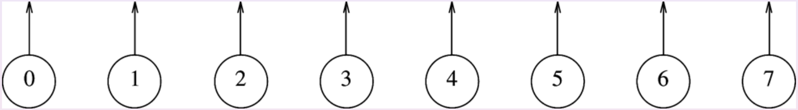
Example

Initial setup:

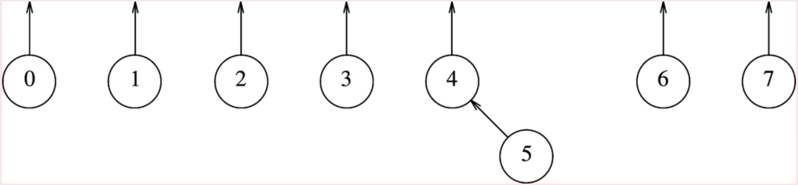


Example

Initial setup:

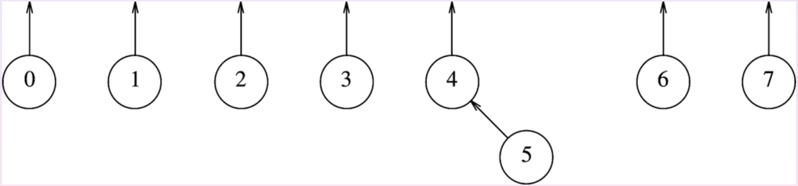


After *union*$(4, 5)$



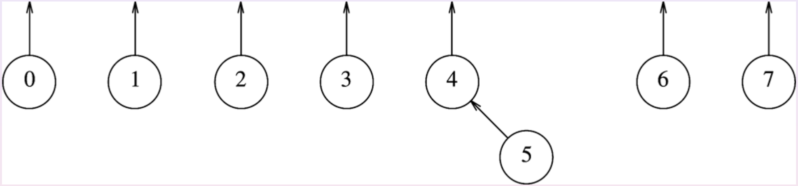
Example

After $union(4, 5)$

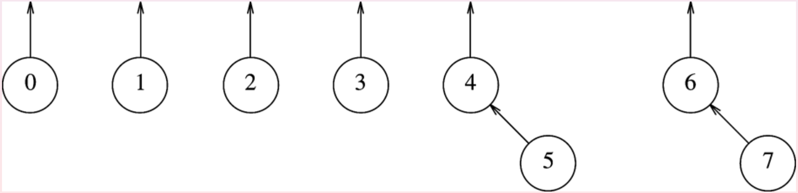


Example

After *union*$(4, 5)$

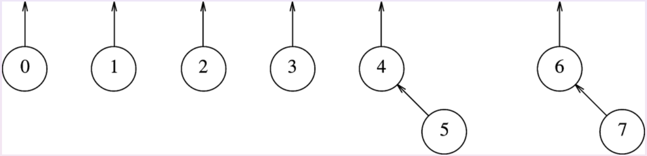


After *union*$(6, 7)$



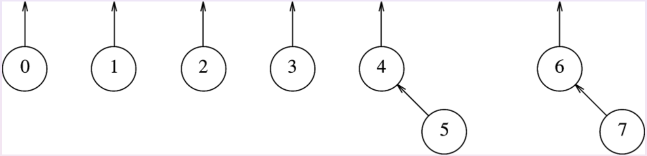
Example

After $union(6, 7)$

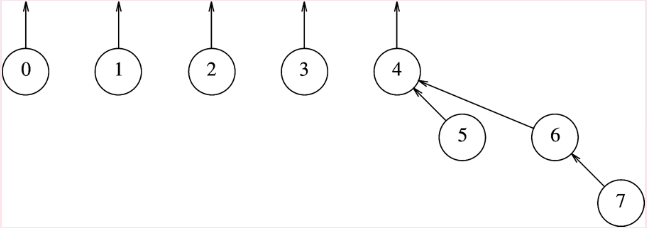


Example

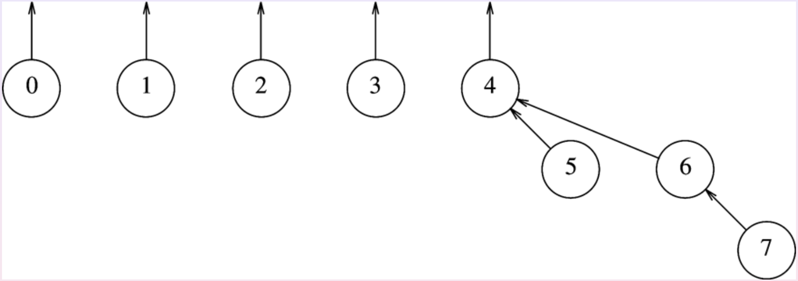
After $union(6,7)$



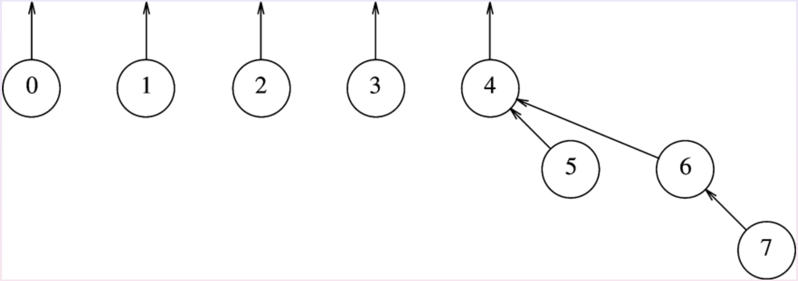
After $union(4,6)$



Representation



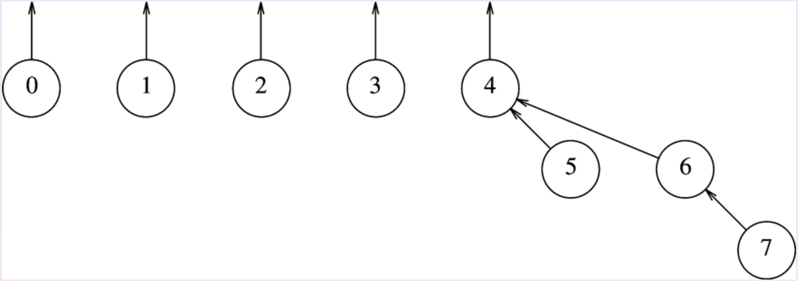
Representation



Idea

Remember parent node only; mark root with -1

Representation



Idea

Remember parent node only; mark root with -1

–1	–1	–1	–1	–1	4	4	6
0	1	2	3	4	5	6	7

Variants

Problem

How to choose root for union? Bad choice can lead to long paths

Variants

Problem

How to choose root for union? Bad choice can lead to long paths

Union-by-size

Always make the smaller tree a subtree of the larger tree

Variants

Problem

How to choose root for union? Bad choice can lead to long paths

Union-by-size

Always make the smaller tree a subtree of the larger tree

Analysis

When depth increases, the tree is smaller than the other side. Thus, after union, it is at least twice as large.

Variants

Problem

How to choose root for union? Bad choice can lead to long paths

Union-by-size

Always make the smaller tree a subtree of the larger tree

Analysis

When depth increases, the tree is smaller than the other side. Thus, after union, it is at least twice as large.

Height

less than or equal to $\log N$

Variants

Union-by-height

Always make the shorter tree a subtree of the higher tree

Variants

Union-by-height

Always make the shorter tree a subtree of the higher tree

Height

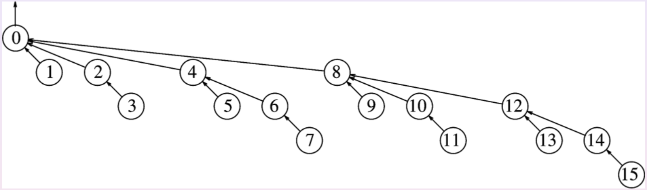
As with union-by-size: $O(\log N)$

Path Compression

During *find* make every node point to root

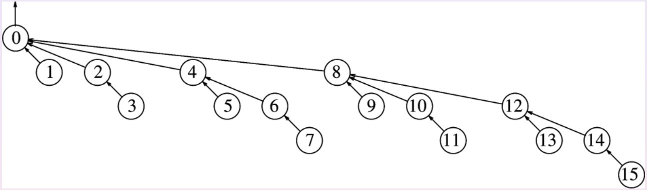
Path Compression

During *find* make every node point to root

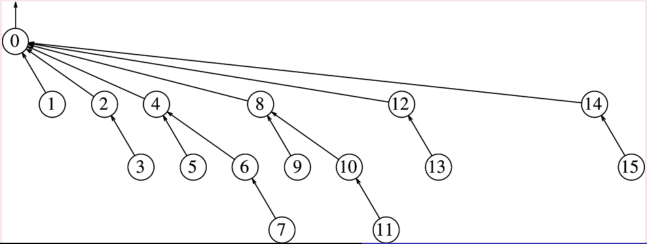


Path Compression

During *find* make every node point to root



after *find*(14)



A Very Slowly Growing Function

Definition

$\log^* N$ is the number of times log needs to be applied to N until $N \leq 1$.

Examples

- $\log^* 2 = 1$
- $\log^* 4 = 2$
- $\log^* 16 = 3$
- $\log^* 65536 = 4$
- ...

Runtime

Consider variant

Union-by-height combined with path compression

Runtime

Consider variant

Union-by-height combined with path compression

Theorem

The running time of M *unions* and *finds* is $O(M \log^* N)$.

1. External Search Trees: B-Trees
2. External Sorting
3. Disjoint Sets
4. Java API Support for Data Structures
   - Collections, Lists, Iterators
   - Trees
   - Hashing
   - PriorityQueue
   - Sorting
5. Another Puzzler

The Top-level Collection Interface

```
public interface Collection<Any>
        extends Iterable<Any>
{
    int size();
    boolean isEmpty();
    void clear();
    boolean contains(Any x);
    boolean add(Any x);        // sic
    boolean remove(Any x);     // sic
    java.util.Iterator<Any> iterator();
}
```

The List Interface in Collection API

```
public interface List<Any>
      extends Collection<Any>
{
  Any get(int idx);
  Any set(int idx, Any newVal);
  void add(int idx, Any x);
  void remove(int idx);

  ListIterator<Any> listIterator(int pos);
}
```

ArrayList and LinkedList

```
public class ArrayList<Any>
        implements List<Any> {...}
public class LinkedList<Any>
        implements List<Any> {...}
```

Iterators

```
public interface Iterator<Any> {
  boolean hasNext( );
  Any next( );
  void remove( );
}
```

ListIterators

```
public interface ListIterator<Any>
        extends Iterator<Any>
{
  boolean hasPrevious();
  Any previous();
  void add(Any x);
  void set(Any newVal);
}
```

TreeSet

- Implements Collection
- Guarantees $O(\log N)$ time for add, remove and contains

AbstractMap<K,V>

Basic operations

- V get(K key): Returns the value to which the specified key is mapped.
- V put(K key, V value): Associates the specified value with the specified key in this map.

AbstractMap<K,V>

Basic operations

- V get(K key): Returns the value to which the specified key is mapped.
- V put(K key, V value): Associates the specified value with the specified key in this map.

Other operations

containsKey(key), containsValue(val), remove(key)

TreeMap

- Extends AbstractMap
- Guarantees $O(\log N)$ time for put, get, containsKey, containsValue, remove

HashMap

- Extends AbstractMap
- Uses separate chaining with rehashing
- Rehashing is governed by initial capacity and load factor, set in constructor

HashSet

- Implements Collection using HashMap

PriorityQueue

- Implements Collection
- Efficient implementation of heap data structure
- Operation names:
  - deleteMin is called “poll”
  - insert is called “add”

Sorting

- Generic sorting supported by class Collections

Sorting

- Generic sorting supported by class Collections
- Uses mergesort in order to minimize number of comparisons

Sorting

- Generic sorting supported by class Collections
- Uses mergesort in order to minimize number of comparisons
- Sorting of built-in numerical types supported by class Arrays

Sorting

- Generic sorting supported by class Collections
- Uses mergesort in order to minimize number of comparisons
- Sorting of built-in numerical types supported by class Arrays
- Uses efficient implementation of quicksort, to take advantage of tight inner loop.

Remember Lecture 2 A: Parameter Passing

Java uses pass-by-value parameter passing.

```
public static void tryChanging(int a) {
   a = 1;
   return;
}
...
int b = 2;
tryChanging(b);
System.out.println(b);
```

Remember Lecture 2 A: Parameter Passing with Objects

```
public static void tryChanging(SomeObject obj) {
    obj.someField = 1;
    obj = new SomeObject();
    obj.someField = 2;
    return;
}
...
SomeObject someObj = new SomeObject();
tryChanging(someObj);
System.out.println(someObj.someField);
```

Remember Lecture 7 A: Sorting

Input

Unsorted array of elements

Behavior

Rearrange elements of array such that the smallest appears first, followed by the second smallest etc, finally followed by the largest element

Will This Work?

```
public static <AnyType extends Comparable<? super A
  void mergeSort( AnyType [] a) {
    AnyType[] ret = ....; // declare helper array
    .... // here goes a program that places
    .... // the element of "a" into "ret" so
    .... // that "ret" is sorted
    a = ret;
    return;
}
...
Integer[] myArray = ...;
IterativeMergeSort.mergeSort(myArray);
```

Will This Work?

```
public static <AnyType extends Comparable<? super A
  void mergeSort( AnyType [] a) {
    AnyType[] ret = ....; // declare helper array
    .... // here goes a program that places
    .... // the element of "a" into "ret" so
    .... // that "ret" is sorted
    a = ret;
    return;
}
...
Integer[] myArray = ...;
IterativeMergeSort.mergeSort(myArray);
```

Answer: No! The assignment a = ret; has no effect on myArray!

This Week and Beyond

- Thursday tutorial: outstanding assignments and labs
- Friday lecture: CS1102S summary, outlook; questions?
- Next week: Reading week, consultation by appointment
- 3/5, morning: Final