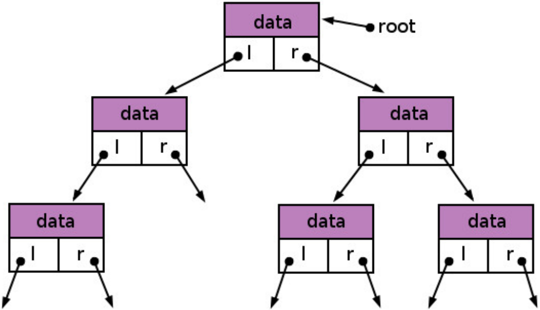
Dynamic Data Structures

Dr Timothy Kimber

January 2016



Dynamic Data Structures

Having efficient data structures is crucial for successful algorithms.

- The problems seen so far involved fixed length lists
- In Java we have a simple way to implement this efficiently — arrays
- Our algorithms assumed some sort of array type was available

Other problems require dynamic data structures such as

- Lists, Stacks and Queues
- Sets and Dictionaries

Java has library versions, but how might they be implemented?

Ordered Data Structures

A *list* is an ordered collection of {nodes, items, elements}.

- The key property of a list is the ordering of the nodes
- A list might support operations such as

push	adds an element to the end of the list
pop	removes the last element of the list
shift	removes the first element of the list
unshift	adds an element to the front of the list
insert	adds an element at a given position
remove	removes the element at a given position
iterate	returns the items in order

- Plus sorting, searching, copying, joining, splitting ...
- The most appropriate implementation depends on which operations are needed.

Stacks

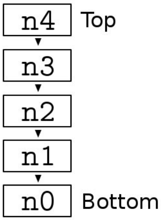
A *stack* is a last-in first-out (LIFO) list.

- Stacks support only

  push for adding elements

  pop for removing elements

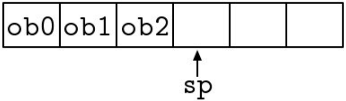
- Stacks are usually pictured as a vertical (stacked!) structure



- Stacks support recursive algorithms including fundamental operations such as calling subprocedures and evaluating arithmetic expressions

Stack Implementation

An array is a natural choice to implement a stack



- Declare an array to hold the values
- Declare an int to point to the top of the stack
- To push a value
  - assign the value to `array[sp]`
  - increment `sp`
- To pop the stack (check `sp > 0`)
  - decrement `sp`
  - return `array[sp]`
- All constant time operations. Good times!..?

Java Stack

```java
public class Stack<T> {

    private static final int INITIAL_CAPACITY = 4;

    private T [] items;
    private int   size;

    public Stack() {
        items = (T []) new Object[INITIAL_CAPACITY];
        size  = 0;
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }
```

- `T` is a Java type variable
- The values are stored in an array of type `T[]`
- We have possible overflow and underflow errors

Java Dynamic Stack

```java
public class Stack<T> {
    ...

    public void push(T item) {
        if (isFull()) { increaseCapacity(); }
        items[size++] = item;
    }

    public T pop() {
        if (isEmpty())  { throw new NoSuchElementException(); }
        if (isTooBig()) { decreaseCapacity(); }
        return items[--size];
    }

    ...
}
```

- push increases the capacity of the stack if it is full
- pop decreases the capacity if it is too big

Java Dynamic Stack

```java
public class Stack<T> {
    ...

    private boolean isFull() {
        return size == items.length;
    }

    private boolean isTooBig() {
        return size < items.length / INITIAL_CAPACITY;
    }

    private void increaseCapacity() {
        items = Arrays.copyOf(items, items.length * 2);
    }

    private void decreaseCapacity() {
        items = Arrays.copyOf(items, items.length / 2);
    }
}
```

- To increase the capacity we copy to a new double size array
- If the array is less than 1/4 full the capacity is halved
- This implementation uses a dynamic array

Performance of Push

What will be the (worst case) time $T(N)$ to push N objects?

- Assume: time to insert (copy, add) one object to array is c

Then the time taken for the each push is:

- c, c, c, c, $c + 4c$, c, ...

So:

- The worst case for a push is Nc
- $T(N) = O(N^2)$

However, this is a big overestimate and not a tight bound

- Most pushes are still $O(1)$
- Want to know what an average push costs

Amortised Analysis

Amortised analysis considers the performance of a sequence of operations.

- Allows average performance of an operation to be determined
- One technique is to spread the cost of expensive operations

The ‘actual’ sequence of costs per push:

- c, c, c, c, $c + 4c$, c, ...

becomes

- $2c$, $2c$, $2c$, $2c$, c, c, ...

if the first four pushes ‘pay in advance’ for the later copying.

- So, is the average cost $2c$?

Amortised Analysis

Represent the cost c by one coin

- Immediately after a copy the array is half full
- Assume all costs have been met so far

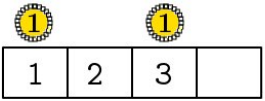
All costs will be covered if each push pays three coins:

- One for the initial insert
- One to copy itself, one to copy an existing item

Amortised Analysis

Represent the cost c by one coin

- Immediately after a copy the array is half full
- Assume all costs have been met so far



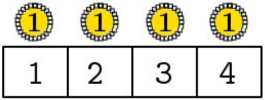
All costs will be covered if each push pays three coins:

- One for the initial insert
- One to copy itself, one to copy an existing item

Amortised Analysis

Represent the cost c by one coin

- Immediately after a copy the array is half full
- Assume all costs have been met so far



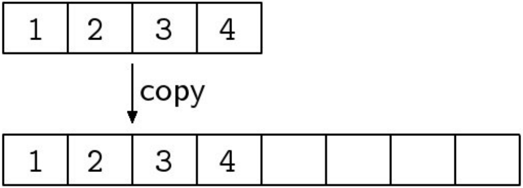
All costs will be covered if each push pays three coins:

- One for the initial insert
- One to copy itself, one to copy an existing item

Amortised Analysis

Represent the cost c by one coin

- Immediately after a copy the array is half full
- Assume all costs have been met so far



All costs will be covered if each push pays three coins:

- One for the initial insert
- One to copy itself, one to copy an existing item

Amortised Analysis

This amortised cost of $3c$ for each push shows that

- $T(N) \leq 3Nc$, so $T(N) = O(N)$
- The average cost of a push is $T(N)/N \leq 3c$
- The push method runs in amortised constant time

The same result can be reached by summing the costs for N pushes

- Assume stack has initial capacity 1 for simplicity
- $T(N) = Nc + (2^0 + \cdots + 2^j)c$, where $j = \lfloor log_2 N \rfloor$
- $2^0 + \cdots + 2^j = 2N - 1$, if N is a power of 2
- So, $T(N) = 3Nc - c$

The $-c$ is because the stack is empty before the first push

Queues

A (FIFO) *queue* is a first-in first-out list.

- Queues support only

  push (usually just 'add') for adding elements
  shift (usually 'remove') for removing elements

- Queues have many applications. e.g. breadth-first search of graphs
- Queues can also be implemented with arrays
- Queues are more naturally implemented using a linked-list structure

Queue Performance

Performance of queue operations

Operation	Array	Linked
add	amortised $O(1)$	$O(1)$
remove	amortised $O(1)$	$O(1)$
size	$O(1)$	$O(1)$
insert	$O(N)$	$O(N)$
delete	$O(N)$	$O(N)$
split	$O(N)$	$O(N)$
join	$O(N)$	$O(1)$

Priority Queues

A priority queue has different behaviour

- Each item has an associated key
- `remove()` returns the item with the lowest (ref. highest) key
- Such queues have many applications, e.g. best-first search

9, 8, 7, 6, 3 ← **Max Priority Queue** ← 6, 9, 7, 3, 8

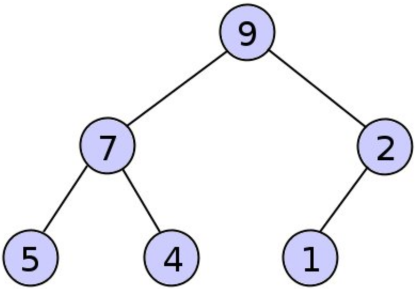
`remove items` `add items`

Binary Heap

A priority queue can be efficiently implemented using a binary heap

Definition (Binary Heap)

A binary tree is a *heap* iff the key at each node is less than or equal to the key of its parent

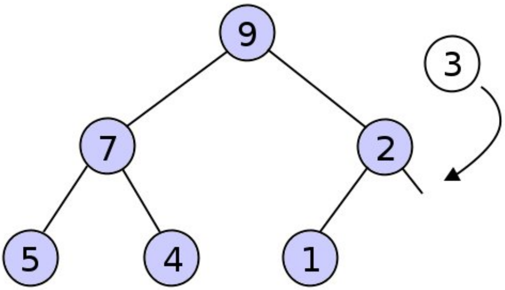


- Nodes are added and removed by traversing one branch: $O(log_2 N)$

Binary Heap Operations

add(key k, data d)

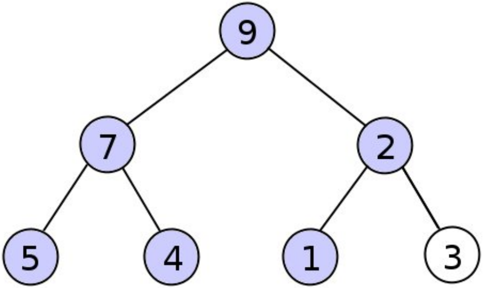
- insert a new node $n = (k, d)$ at the end of the heap
- while k is greater than the key of n's parent
  - swap n with its parent
- HALT



Binary Heap Operations

add(key k, data d)

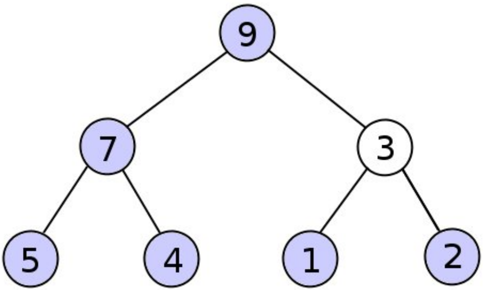
- insert a new node $n = (k, d)$ at the end of the heap
- while k is greater than the key of n's parent
  - swap n with its parent
- HALT



Binary Heap Operations

add(key k, data d)

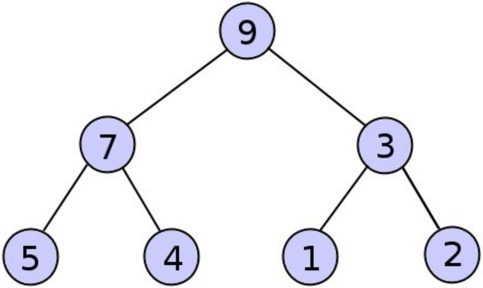
- insert a new node $n = (k, d)$ at the end of the heap
- while k is greater than the key of n's parent
  - swap n with its parent
- HALT



Binary Heap Operations

add(key k, data d)

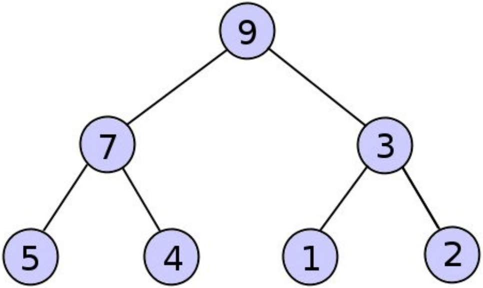
- insert a new node $n = (k, d)$ at the end of the heap
- while k is greater than the key of n's parent
  - swap n with its parent
- HALT



Binary Heap Operations

remove

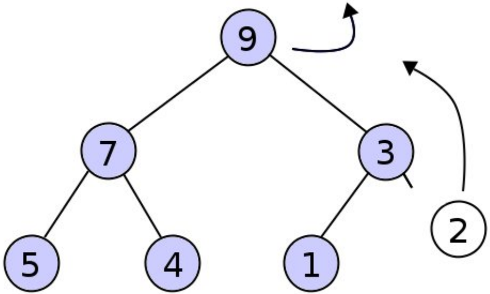
- Store the top node n_0
- Move node $n = (k, d)$ from the end of the heap to the top
- While k is less than the key of a child of n
  - swap n with the child with the highest key
- Return n_0 and HALT



Binary Heap Operations

remove

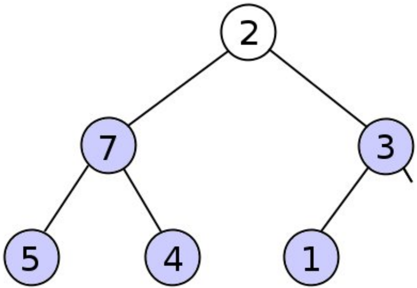
- Store the top node n_0
- Move node $n = (k, d)$ from the end of the heap to the top
- While k is less than the key of a child of n
  - swap n with the child with the highest key
- Return n_0 and HALT



Binary Heap Operations

remove

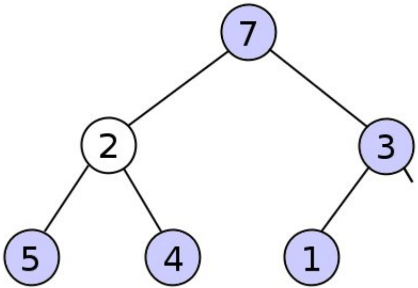
- Store the top node n_0
- Move node $n = (k, d)$ from the end of the heap to the top
- While k is less than the key of a child of n
  - swap n with the child with the highest key
- Return n_0 and HALT



Binary Heap Operations

remove

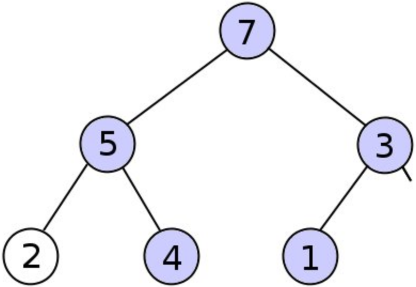
- Store the top node n_0
- Move node $n = (k, d)$ from the end of the heap to the top
- While k is less than the key of a child of n
  - swap n with the child with the highest key
- Return n_0 and HALT



Binary Heap Operations

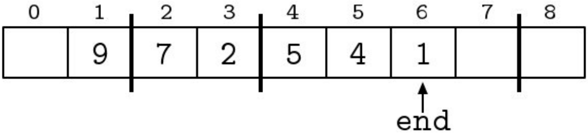
remove

- Store the top node n_0
- Move node $n = (k, d)$ from the end of the heap to the top
- While k is less than the key of a child of n
  - swap n with the child with the highest key
- Return n_0 and HALT



Heap Implementation

A heap can be implemented using a dynamic array



- Easy to keep track of the end of the heap
- Changes never leave a gap in the heap, so no extra copying
- Leaving `a[0]` blank simplifies navigation:
  - parent of `a[n]` is `a[n/2]`
  - children of `a[n]` are `a[2*n]` and `a[2*n+1]`

Heapsort

Heaps also provide us with the Heapsort algorithm (JWJ Williams, 1964)

Heapsort (given a list L)

- Create an empty heap H
- Remove each element of L and add it to H
- Remove each element of H and add it to L
- HALT

- What could be simpler?!
- Performance is again $O(N log_2 N)$
- Can also be implemented in place by setting up list and heap partitions within a single array

Sets

A *set* is an unordered collection of unique {keys, elements}.

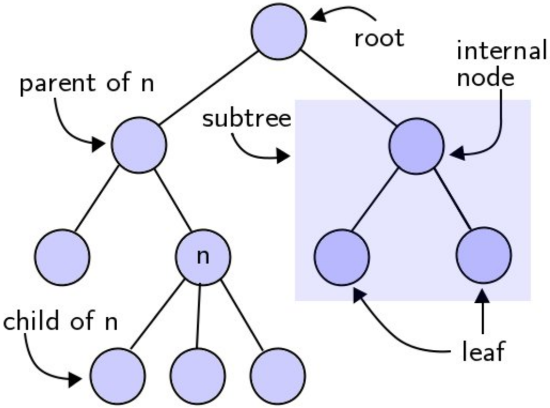
- A set might support operations such as

add	adds an element to the set
delete	removes an element from the set
contains	is the element in the set?
union	combines two sets
iterate	returns all elements of the set in any order

- The fundamental operation is search
- When each key in a set is associated with some other value the structure is called a map, dictionary or hash
- Sets are frequently implemented using trees and hash tables

Trees

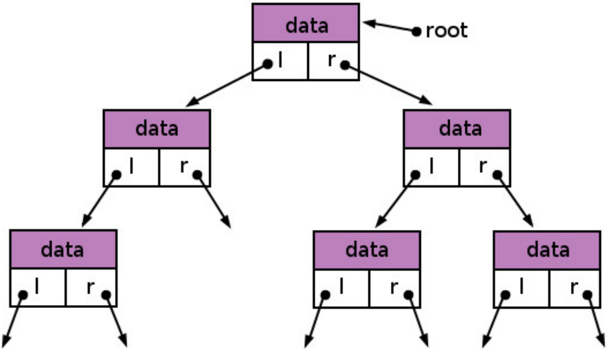
Trees enable efficient search using the same principle as binary search



Trees

A binary tree of node objects

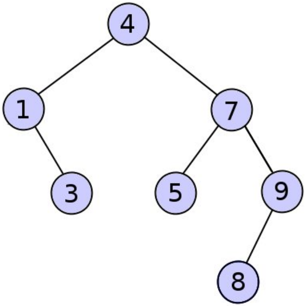
- Each node has two children, which may be null



Binary Search Trees

Definition

A binary tree is a binary search tree iff all keys in the left subtree are less than (or equal to) the root key and all keys in the right subtree are greater than (or equal to) the root key

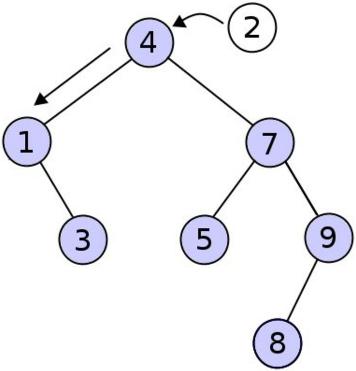


- I will assume duplicate keys are not allowed

Adding A Key

A new key is always added as a leaf node

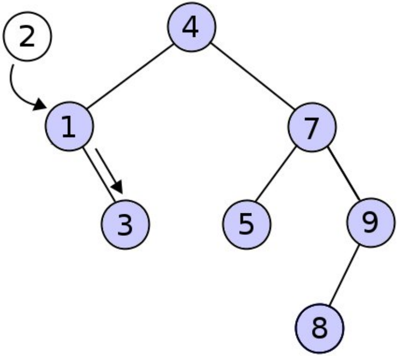
- Start at the root, move down comparing against each key



Adding A Key

A new key is always added as a leaf node

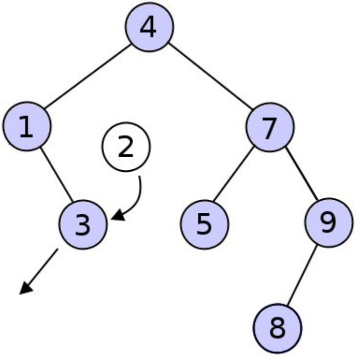
- Start at the root, move down comparing against each key



Adding A Key

A new key is always added as a leaf node

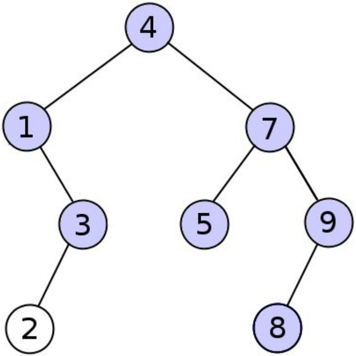
- Start at the root, move down comparing against each key



Adding A Key

A new key is always added as a leaf node

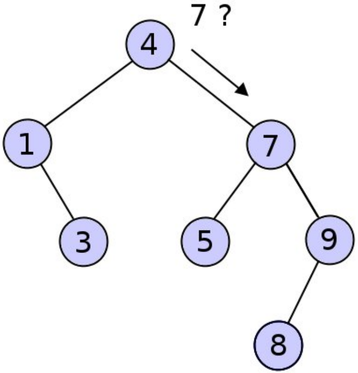
- Start at the root, move down comparing against each key



Search

Searching for some key k works in exactly the same way

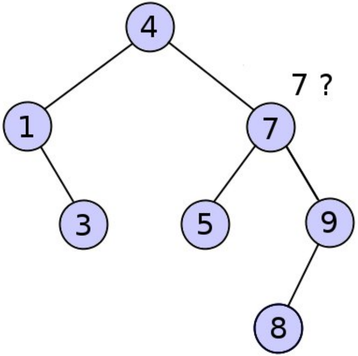
- Start at the root and at each node:
  - If the node contains k then return true or the data in the node
  - Otherwise, if k is less than the node's key move on to the left child
  - Otherwise move on to the right child



Search

Searching for some key k works in exactly the same way

- Start at the root and at each node:
  - If the node contains k then return true or the data in the node
  - Otherwise, if k is less than the node's key move on to the left child
  - Otherwise move on to the right child



A Java BST

```
public class BinarySearchTree<Key extends Comparable<Key>, Val>
implements Iterable<Val>{

    public static class BSTNode<K extends Comparable<K>, V> {
        K                key;
        V              value;
        BSTNode<K, V>  left;
        BSTNode<K, V> right;

        public BSTNode(K k, V v) {
            key   = k;
            value = v;
            left  = null;
            right = null;
        }
    }

    private BSTNode<Key, Val> root;

    public BinarySearchTree() {
        root = null;
    }
```

- BSTNode has four fields including two node pointers
- The BST class itself has a single node pointer field root

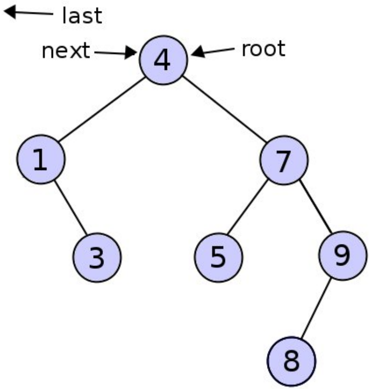
Add method

```
public void add(Key k, Val v) {
    BSTNode<Key, Val> next = root;
    BSTNode<Key, Val> last = null;
    int               compare = 0;

    while(next != null) {
        last = next;
        compare = k.compareTo(next.key);
        if (compare == 0) { return; }
        if (compare < 0)  { next = next.left; }
        else              { next = next.right; }
    }

    BSTNode<Key, Val> n = new BSTNode<Key, Val>(k, v);
    if      (last == null) { root = n; }
    else if (compare < 0)  { last.left = n; }
    else                   { last.right = n; }
}
```

- Two node pointers are used to iterate down the tree
- The `next` pointer will fall off the tree at the addition point
- The `last` pointer identifies the parent of the new node
- If `last` is null the new node is the root

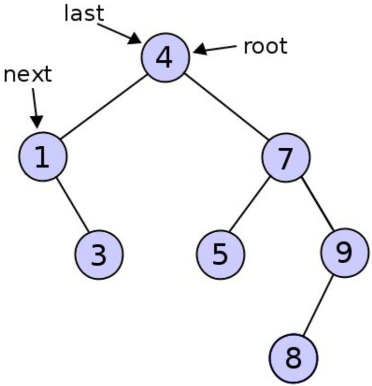
Adding A Key

The iterative add method



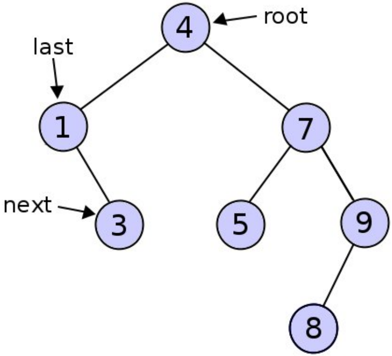
Adding A Key

The iterative add method



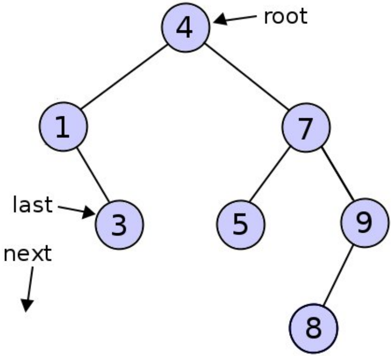
Adding A Key

The iterative add method



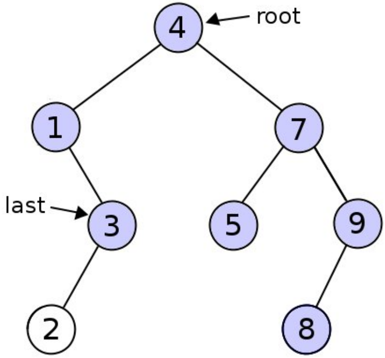
Adding A Key

The iterative add method



Adding A Key

The iterative add method



Recursive Add Method

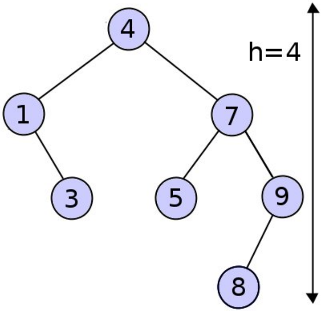
```java
public void addRecursive(Key k, Val v) {
    root = addToSubtree(root, k, v);
}

private BSTNode<Key, Val> addToSubtree(
        BSTNode<Key, Val> node, Key key, Val val) {

    if (node == null)  { return new BSTNode<Key, Val>(key, val); }

    int compare = key.compareTo(node.key);
    if (compare < 0)  { node.left =  addToSubtree(node.left, key, val); }
    if (compare > 0)  { node.right = addToSubtree(node.right, key, val); }
    return node;
}
```

- The add method can also be implemented recursively
- The `addToSubtree` method returns a pointer to the new subtree

Performance

`add` and `search` run in time proportional to the height h of the tree

- The methods follow a single branch of the tree
- h depends on the order the keys are added
- What is the worst case?



Traversing the Tree

The tree can be traversed in-order without checking any keys by

- Visiting the nodes in the left subtree
- Visiting the root node
- Visiting the nodes in the right subtree

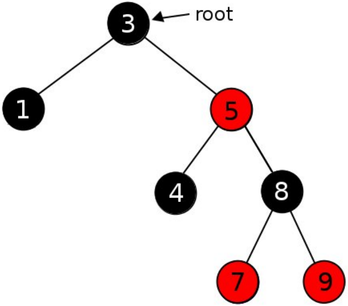
```
public void printKeysInOrder() {
    printKeysInOrder(root);
}

private void printKeysInOrder(BSTNode<Key, Val> n) {
    if (n != null) {
        printKeysInOrder(n.left);
        System.out.print(n.key + " ");
        printKeysInOrder(n.right);
    }
}
```

- Reordering the steps produces other orders

Red-Black Trees

Red-Black Trees are binary search trees that maintain balance

- A BST can become (very) unbalanced, resulting in long branches
- Searching a BST takes $O(N)$ time in the worst case
- The branches of a balanced tree remain as short as possible



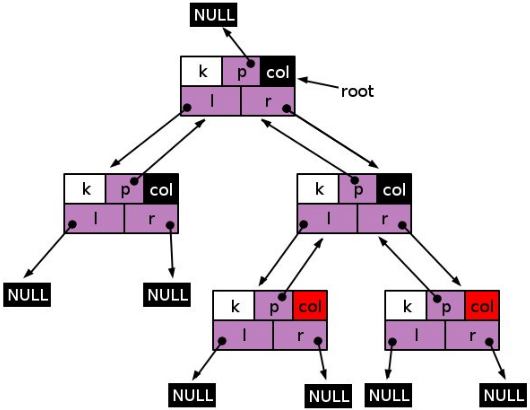
Red-Black Tree Properties

Definition (Red-Black Tree)

A binary search tree T is a red-black tree iff T satisfies the following five properties:

1. All nodes (including nulls) are either red or black
2. The root node is black
3. Every leaf (all null) is black
4. Both children of a red node are black
5. All paths from a node to a descendant leaf contain the same number of black nodes

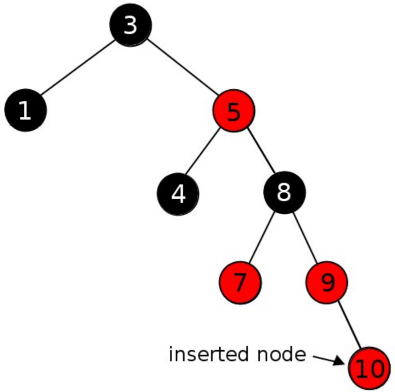
Red-Black Tree Attributes



- Nodes have an additional field for colour
- A parent pointer is also needed

Insertion

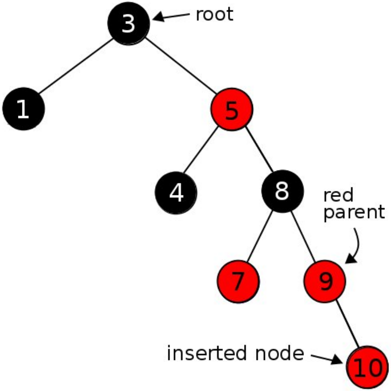
A node is inserted using the ordinary BST procedure



- A new node is always colored red

Insertion

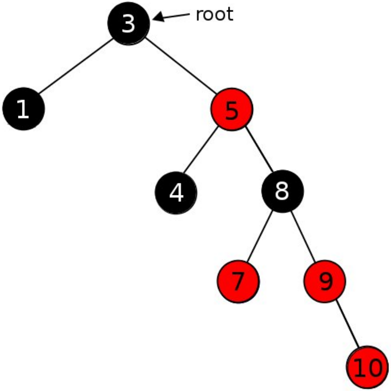
The insertion may result in a violation of the red-black tree properties



- The root might be coloured red
- A red node might have a red child
- Insertion must ensure the properties are restored

Insertion

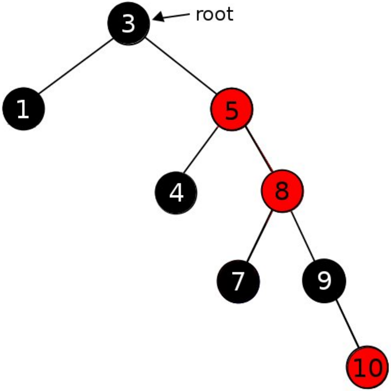
Step 1: recolour nodes 7, 8 and 9



- The black node is pushed down into its subtrees

Insertion

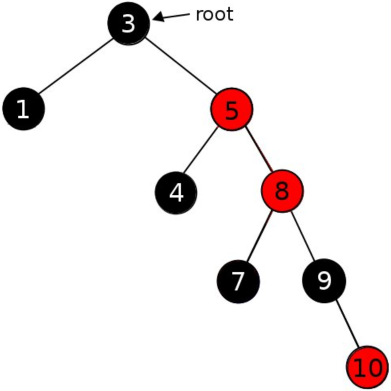
Step 1: recolour nodes 7, 8 and 9



- There is still a red node with a red parent

Insertion

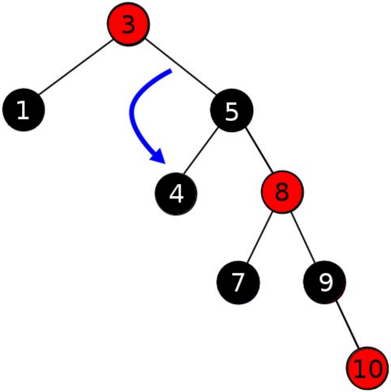
Step 2: recolour nodes 3 and 5 and left rotate node 3



- The red parent node (5), and its left subtree, move past the root

Insertion

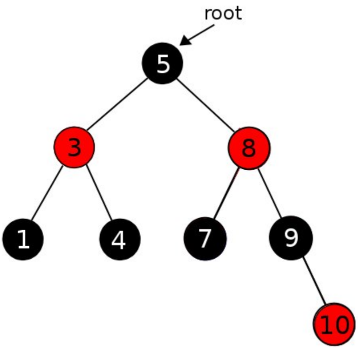
Step 2: recolour nodes 3 and 5 and left rotate node 3



- The red parent node (5), and its left subtree, move past the root

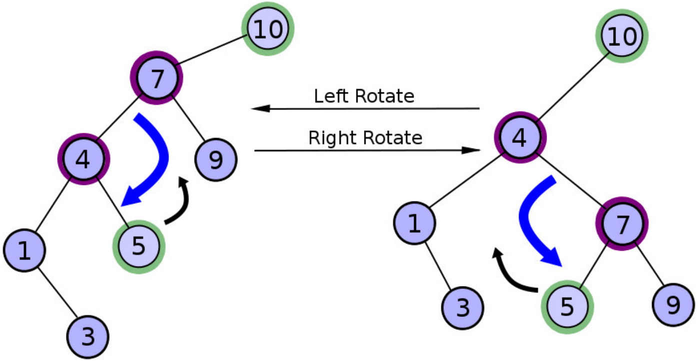
Insertion

Step 2: recolour nodes 3 and 5 and left rotate node 3



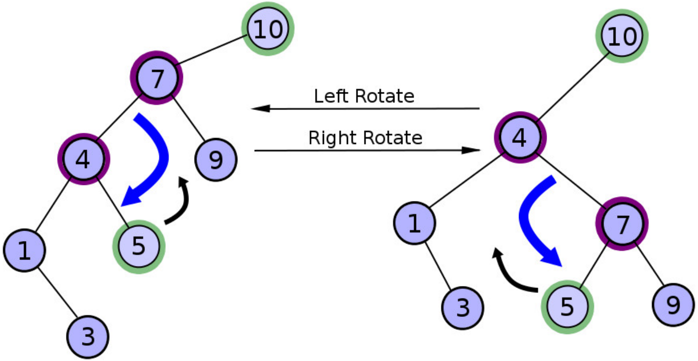
- The red parent node (5), and its left subtree, move past the root
- All the properties are now satisfied

Rotations



A rotation is a localised reorganisation of a binary search tree that maintains the correct key order

Rotations



- In a right rotation a node n becomes the right child of its left child c
- In a left rotation a node n becomes the left child of its right child c
- Neither n nor c can be an empty node
- The displaced child of c is transferred to n

Performance

Insertion and Search (and Deletion) run in $O(log_2(N))$ time

- In an ordinary BST the operations take $O(h)$ time
- In a red-black tree $h \leq 2log_2(N+1)$
- The Search method is the same
- For Insertion, only the last part is different

In each step of the tidy up part of Insertion

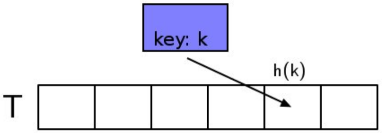
- Either the node with a red parent moves up the tree, or
- A rotation ends the insertion

So, Insertion runs in $O(log_2(N))$ time

Hash Tables

Hash Tables provide set implementations with average case $O(1)$ performance for Insertion, Search and Deletion

- A hash table T is (like) an array with m slots
- A hash function h maps every possible key to one of the slots
- So, an object with key k is stored at $T[h(k)]$



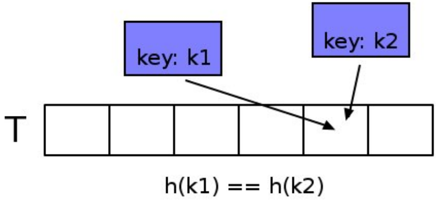
How many slots do we need if there are N_k keys?

Hash Tables

If N_k is very large, m would also need to be very large

- We will have a massive table
- The table might only contain a few actual values
- This is not efficient, so we must reduce m and allow collisions

A collision occurs when two keys 'hash' to the same value

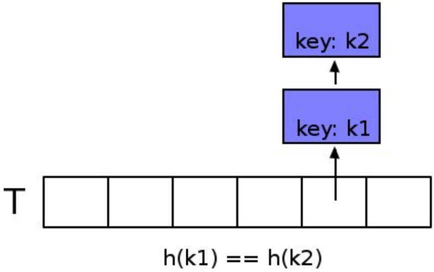


We need a method for collision resolution

Chaining

Chaining is a simple way to resolve collisions

- All objects whose key hashes to $h(k)$ are placed into a linked list
- The table contains a pointer to the list
- So, $T[i]$ contains a list of objects x where $h(x.key) = i$



Performance of Chaining

Insert object x

- Add x to the head of the list at $T[h(x.key)]$
- HALT

Insertion runs in $O(1)$ time

Search for key k

- Search list at $T[h(k)]$ for an object where $x.key == k$
- HALT

Assuming a table containing N values

- The performance of Search depends on the hash function
- If every key hashes to the same value, then running time $= \Theta(N)$

Performance is optimised when we have simple uniform hashing

Performance of Chaining

Definition (Simple Uniform Hashing)

Given a hash table T with m slots, a hash function h produces simple uniform hashing if, for an unknown key k, the probability that $h(k) = i$, is the same for all i such that $1 \leq i \leq m$.

Assuming simple uniform hashing, the average case running time of Search in a hash table with chaining is $O(N/m)$

- N is the number of objects stored in the table
- N/m is called the load factor
- The average length of a chain is N/m

If N is proportional to m, then Search runs in $O(1)$ time

Hash Functions

A good hash function should

- Run in $O(1)$ time
- Approximate simple uniform hashing
- Map related keys, like `"a"` and `"aa"`, to unrelated values

Two stages are involved

1. Convert the key to a natural number (0, 1, 2 ...)
2. Map the number to the range $0 \ldots m-1$

The first stage

- Will be different for each type of object
- The result should depend on as many bits as possible

Java Hashcodes

In Java, every object has a method (inherited from the class `Object`)

- `public int hashCode()`
  returns a hash code value for the object

This method is called if the object is used as a key in a hash table

- The result is the numerical representation of the object
- Any class that could be used as a key should override `hashCode`
- if `a.equals(b)` succeeds `a` and `b` must return same `hashCode`

Example (A class with two fields)

```
public int hashCode() {
  int hash = 1;
  hash = hash * 31 + nonNullObjectField.hashCode();
  hash = hash * 31 + intField;
  return hash;
}
```

A Hash Function

To map any number to $0 \dots m-1$:

- $h(k) = k \bmod m$
- Choice of m is important

Example

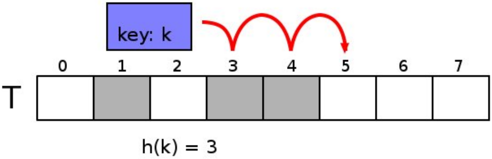
A hash table has keys that are strings. In order to convert every string s to a different number k the ASCII values of the characters in s are used to produce a radix-128 integer. So, $k = s[0] + s[1] * 128 + s[2] * 128^2 + \dots$. If m is also 128, then every key beginning with 'a' will be stored in the same slot, every key beginning with 'b' will be stored in the same slot and so on.

- Similar effect in a $m = 31$ table using the `hashCode` code above
- Since 31 is prime we will be OK as long as $m \neq 31$
- However, hash tables are vulnerable to a deliberate attack

Probing

In an open address hash table objects are stored directly in the table

- We use probing to resolve collisions
- To insert an object we probe the table until we find a space
- The hash function generates a sequence $\langle h(k,0),\ldots,h(k,m-1)\rangle$



The simplest form (above) is linear probing

- Consecutive slots are probed, beginning with $h(k)$, up to $h(k)-1$

Performance of Probing

Definition (Uniform Hashing)

Given a hash table with m slots, a hash function produces uniform hashing if, for an unknown key k, the probability that the probe sequence of k is p, where p is a permutation of $\langle 0, \ldots, m-1 \rangle$ is the same for all such p.

- Linear probing does not produce uniform hashing

Assuming uniform hashing, the average number of probes required to insert an element into a hash table with probing is at most $1/(1 - N/m)$

- Analysis is beyond the scope of this course
- Each probe is to a random slot, with probability N/m it is occupied

If N is proportional to m, then insertion (and search) runs in $O(1)$ time

Limitations

Hash tables do not support operations such as:

- In order iteration
- Next key / object
- Minimum key
- Maximum key

since objects are stored, by design, in random order.