18: Binary Search Trees
2.4 **Binary Heap Demo**
Things to remember about runtime complexity of heaps

- Insertion is $O(\log n)$. Why?
- Delete max is $O(\log n)$. Why?
- Space efficiency is $O(n)$. Why?
  - Array with complete tree
Lecture 18: Priority Queues, Heapsort, BST

- Binary Heaps
- Priority Queue
- Heapsort
Priority Queue ADT

- Service best element first
  - Compared to FIFO or LIFO
- Two operations:
  - Delete (return) the maximum
  - Insert
- Applications: load balancing and interruption handling in OS, Huffman codes for compression, A* search for AI, Dijkstra’s and Prim's algorithm for graph search, etc.
- How can we implement a priority queue efficiently?
  - Unordered array, Ordered array, Binary Heap
Option 1: Unordered array

- The lazy approach where we defer doing work (deleting the maximum) until necessary.
- Insert is $O(1)$ (will be implemented as push in stacks).
- Delete maximum is $O(n)$ (have to traverse the entire array to find the maximum element).
public class UnorderedArrayMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;     // elements
    private int n;         // number of elements

    // set initial size of heap to hold size elements
    public UnorderedArrayMaxPQ(int capacity) {
        pq = (Key[]) new Comparable[capacity];
        n = 0;
    }

    public boolean isEmpty() { return n == 0; }
    public int size() { return n; }
    public void insert(Key x) { pq[n++] = x; }   // Insert into index n

    public Key delMax() {
        int max = 0;
        for (int i = 1; i < n; i++)
            if (less(max, i)) max = i;    // Find max element
        exch(max, n-1);                   // Exchange max with last element

        return pq[-n];                    // Return last element
    }

    private boolean less(int i, int j) {
        return pq[i].compareTo(pq[j]) < 0;
    }

    private void exch(int i, int j) {
        Key swap = pq[i];
        pq[i] = pq[j];
        pq[j] = swap;
    }
}
Practice Time

- Given an empty array of capacity 10, perform the following operations in a priority queue based on an unordered array (lazy approach):
  1. Insert P
  2. Insert Q
  3. Insert E
  4. Delete max
  5. Insert X
  6. Insert A
  7. Insert M
  8. Delete max
  9. Insert P
  10. Insert L
  11. Insert E
  12. Delete max
**Answer**

- **Insert P**
- **Insert Q**
- **Insert E**
- **Delete-max -> Q**
- **Insert X**
- **Insert A**
- **Insert M**
- **Delete-max -> X**
- **Insert P**
- **Insert L**
- **Insert E**
- **Delete-max -> P**
Option 2: Ordered array

- The *eager* approach where we do the work (keeping the list sorted) up front to make later operations efficient.

- Insert is $O(n)$ (we have to find the index to insert and shift elements to perform insertion).

- Delete maximum is $O(1)$ (just take the last element which will be the maximum).
public class OrderedArrayMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;  // elements
    private int n;     // number of elements

    // set initial size of heap to hold size elements
    public OrderedArrayMaxPQ(int capacity) {
        pq = (Key[]) (new Comparable[capacity]);
        n = 0;
    }

    public boolean isEmpty() { return n == 0; }
    public int size() { return n; }
    public Key delMax() { return pq[--n]; }

    public void insert(Key key) {
        int i = n-1;
        while (i >= 0 && less(key, pq[i])) {
            pq[i+1] = pq[i];  // Empty element is at index i
            i--;
        }
        pq[i+1] = key;   // I+1 to get to the empty element
        n++;
    }

    private boolean less(Key v, Key w) {
        return v.compareTo(w) < 0;
    }
}
Given an empty array of capacity 10, perform the following operations in a priority queue based on an ordered array (eager approach):

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max
Answer
Option 3: Binary heap

- Will allow us to both insert and delete max in $O(\log n)$ running time.
- There is no way to implement a priority queue in such a way that insert and delete max can be achieved in $O(1)$ running time.
- Priority queues are synonyms to binary heaps.
Practice Time

- Given an empty binary heap that represents a priority queue, perform the following operations:

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max
Answer
Lecture 18: Priority Queues and Heapsort

- Priority Queue
- Heapsort
Basic plan for heap sort

- Use a priority queue to develop a sorting method that works in two steps:
  - 1) Heap construction: build a binary heap with all \( n \) keys that need to be sorted.
  - 2) Sortdown: repeatedly remove and return the maximum key.
**O(n)** Heap construction

- Construct complete binary tree with elements
- Ignore all leaves (indices \( n/2 + 1, \ldots, n \)).
- ```
   \textbf{for} (\textbf{int} \ k = n/2; \ k \geq 1; \ k--) 
   \text{sink}(a, k, n);
   ```
- **Key insight:** After `sink(a, k, n)` completes, the subtree rooted at \( k \) is a heap.

![Diagrams](image.png)
Practice Time

- Run the first step of heapsort, heap construction, on the array [2,9,7,6,5,8].
Answer: Heap construction

1. Initial heap (arbitrary order):
   - 9, 6, 5, 8, 3, 7, 2
   - Starting point: 9

2. Sink operation at index 3:
   - k = n/2 = 6/2 = 3
   - Sink(3, 6)

3. Result after sink operation:
   - 9, 6, 5, 8, 3, 7, 2
   - New heap: 9, 6, 5, 8, 3, 7, 2

4. Final heap (heap-ordered):
   - 1, 2, 3, 4, 5, 6, 7
Sortdown

- Remove the maximum, one at a time, but leave in array instead of nulling out.

```java
while(n>1){
    exch(a, 1, n--);
    sink(a, 1, n);
}
```

- **Key insight**: After each iteration the array consists of a heap-ordered subarray followed by a sub-array in final order.
HEAPSORT

Sortdown

- while (n > 1) {
  exch(a, 1, n--);
  sink(a, 1, n);
}

result (sorted)
Heapsort demo

**Sortdown.** Repeatedly delete the largest remaining item.
Practice Time

- Given the heap you constructed before, run the second step of heapsort, sortdown, to sort the array [2,9,7,6,5,8].
Answer: Sortdown

HEAPSORT

```
starting point (heap-ordered)
```

```
exch(1,6)
sink(1,5)
```

```
exch(1,5)
sink(1,4)
```

```
exch(1,4)
sink(1,3)
```

```
exch(1,3)
sink(1,2)
```

```
exch(1,2)
sink(1,1)
```

```
result (sorted)
```
Heapsort analysis

- Heap construction makes $O(n)$ exchanges and $O(n)$ compares.
- Sortdown and therefore the entire heap sort $O(n \log n)$ exchanges and compares.
- In-place sorting algorithm with $O(n \log n)$ worst-case!
- Remember:
  - mergesort: not in place, requires linear extra space.
  - quicksort: quadratic time in worst case.
- Heapsort is optimal both for time and space in terms of Big-O, but:
  - Inner loop longer than quick sort.
  - Poor use of cache. Why?
  - Not stable.
## Sorting: Everything you need to remember about it!

<table>
<thead>
<tr>
<th>Which Sort</th>
<th>In place</th>
<th>Stable</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>x</td>
<td></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$n$ exchanges</td>
</tr>
<tr>
<td>Insertion</td>
<td>x</td>
<td>x</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>Use for small arrays or partially ordered</td>
</tr>
<tr>
<td>Merge</td>
<td></td>
<td>x</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>Guaranteed performance; stable</td>
</tr>
<tr>
<td>Quick</td>
<td>x</td>
<td></td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$n \log n$ probabilistic guarantee; fastest!</td>
</tr>
<tr>
<td>Heap</td>
<td>x</td>
<td></td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>Guaranteed performance; in place</td>
</tr>
</tbody>
</table>
Lecture 18: Priority Queues and Heapsort

- Priority Queue
- Heapsort
Readings:

- Textbook:
  - Chapter 2.4 (Pages 308-327), 2.5 (336-344)

- Website:
  - Priority Queues: https://algs4.cs.princeton.edu/24pq/

- Visualization:
  - Create (nlogn) and heapsort: https://visualgo.net/en/heap

Practice Problems:

- 2.4.1-2.4.11. Also try some creative problems.
Readings:

- Textbook:
  - Chapter 2.4 (Pages 308-327)

- Website:
  - Priority Queues: [https://algs4.cs.princeton.edu/24pq/](https://algs4.cs.princeton.edu/24pq/)

Visualization:

- Insert and ExtractMax: [https://visualgo.net/en/heap](https://visualgo.net/en/heap)

Practice Problems:

- Practice with traversals of trees and insertions and deletions in binary heaps
Lecture 18: Search

- Dictionaries (Symbol Tables)
- Binary Search Trees
Dictionaries

- Also known as: symbol tables, maps, indices, associative arrays.
- Key-value pair abstractions that support two operations:
  - Insert a key-value pair.
  - Given a key, search for the corresponding value.
- Supported either with built-in or external libraries by the majority of programming languages.
Basic symbol table API

- **public class** `ST <Key extends Comparable<Key>, Value>`
  - Key needs to implement the Comparable interface, but it is a generic (use extends)
  - `ST()`: create an empty symbol table. By convention, values are not null.
- **void** `put(Key key, Value val)`: insert key-value pair.
  - Overwrites old value with new value if key already exists.
- **Value** `get(Key key)`: return value associated with key.
  - Returns null if key not present. Can’t distinguish between null values and non-existent pairs
- **boolean** `contains(Key key)`: is there a value associated with key?
- **Iterable** `keys()`: all the keys in the symbol table.
- **void** `delete(Key key)`: delete key and associated value.
- **boolean** `isEmpty()`: is the symbol table empty?
- **int** `size()`: number of key-value pairs.
Ordered symbol tables

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>09:00:00</td>
</tr>
<tr>
<td></td>
<td>09:00:03</td>
</tr>
<tr>
<td></td>
<td>09:00:13</td>
</tr>
<tr>
<td>get(09:00:13)</td>
<td>09:00:59</td>
</tr>
<tr>
<td></td>
<td>09:01:10</td>
</tr>
<tr>
<td>floor(09:05:00)</td>
<td>09:03:13</td>
</tr>
<tr>
<td></td>
<td>09:10:11</td>
</tr>
<tr>
<td>select(7)</td>
<td>09:10:25</td>
</tr>
<tr>
<td></td>
<td>09:14:25</td>
</tr>
<tr>
<td></td>
<td>09:19:32</td>
</tr>
<tr>
<td></td>
<td>09:19:46</td>
</tr>
<tr>
<td>keys(09:15:00, 09:25:00)</td>
<td>09:21:05</td>
</tr>
<tr>
<td></td>
<td>09:22:43</td>
</tr>
<tr>
<td></td>
<td>09:22:54</td>
</tr>
<tr>
<td></td>
<td>09:25:52</td>
</tr>
<tr>
<td>ceiling(09:30:00)</td>
<td>09:35:21</td>
</tr>
<tr>
<td></td>
<td>09:36:14</td>
</tr>
<tr>
<td>max()</td>
<td>09:37:44</td>
</tr>
</tbody>
</table>

size(09:15:00, 09:25:00) is 5
rank(09:10:25) is 7
Ordered symbol table API

- Key `min()`: smallest key.
- Key `max()`: largest key.
- Key `floor(Key key)`: largest key less than or equal to given key.
- Key `ceiling(Key key)`: smallest key greater than or equal to given key.
- int `rank(Key key)`: number of keys less that given key.
- Key `select(int k)`: key with rank `k`.
- Iterable `keys()`: all keys in symbol table in sorted order.
- Iterable `keys(int lo, int hi)`: keys in `[lo, ..., hi]` in sorted order.
Printed symbol tables are all around us

- **Dictionary**: key = word, value = definition.
- **Encyclopedia**: key = term, value = article.
- **Phonebook**: key = name, value = phone number.
- **Math table**: key = math functions and input, value = function output.

**Unsupported operations:**
- Add a new key and associated value.
- Remove a given key and associated value.
- Change value associated with a given key.
Lecture 23: Binary Search Trees

- Dictionaries
  - Unordered linked lists (Node with key and value)
    - Insertion and search are linear
  - Sorted array for keys and parallel array for values
    - Search is logarithmic, but insertion is linear
- Binary search Trees
Definitions

- **Binary Search Tree**: A binary tree in symmetric order.

- **Symmetric order**: Each node has a key, and every node’s key is:
  
  - Larger than all keys in its left subtree.
  
  - Smaller than all keys in its right subtree.

- Our textbook uses BSTs to implement dictionaries, therefore each node holds a key-value pair. Other implementations hold only a key.
Differences between heaps and BSTs

<table>
<thead>
<tr>
<th></th>
<th>Heap</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used to implement</td>
<td>Priority queues</td>
<td>Dictionaries</td>
</tr>
<tr>
<td>Supported operations</td>
<td>Insert, delete max</td>
<td>insert, search, delete, ordered operations</td>
</tr>
<tr>
<td>What is inserted</td>
<td>Keys</td>
<td>Key-value pairs</td>
</tr>
<tr>
<td>Underlying data structure</td>
<td>(Resizing) array</td>
<td>Linked nodes</td>
</tr>
<tr>
<td>Tree shape</td>
<td>Complete binary tree</td>
<td>Depends on data</td>
</tr>
<tr>
<td>Ordering of keys</td>
<td>Heap-ordered</td>
<td>Symmetrically-ordered</td>
</tr>
<tr>
<td>Duplicate keys allowed?</td>
<td>Yes</td>
<td>No*</td>
</tr>
</tbody>
</table>

*: when BSTs used to implement dictionaries.
BST representation of dictionaries

- We will use an inner class `Node` that is composed by:
  - A Key that is comparable and a Value
  - A reference to the root nodes of the left (smaller keys) and right (larger keys) subtrees.
  - Potentially, the total number of nodes in the subtree that has root at this node.
- A BST has a reference to a Node `root`.
BST and Node implementation

```java
public class BST<Key extends Comparable<Key>, Value> {
    private Node root; // root of BST

    private class Node {
        private Key key; // sorted by key
        private Value val; // associated value
        private Node left, right; // roots of left and right subtrees
        private int size; // #nodes in subtree rooted at this

        public Node(Key key, Value val, int size) {
            this.key = key;
            this.val = val;
            this.size = size;
        }
    }
}
```
Search for a key

- If less than key in node go to left subtree.
- If greater than key in node go to right subtree.
- If given key and key at examined node are equal, search hit.
- Return value corresponding to given key, or null if no such key.
  - In other implementations, you return the last node you reached.
- Number of compares is equal to the depth of the node + 1.
Search example

Successful (left) and unsuccessful (right) search in a BST
Search - iterative implementation

```java
public Value get(Key key) {
    Node x = root;
    while (x != null) {
        int cmp = key.compareTo(x.key);
        if (cmp < 0)
            x = x.left;
        else if (cmp > 0)
            x = x.right;
        else if (cmp == 0)
            return x.val;
    }
    return null;
}
```
Search - recursive implementation

```java
public Value get(Key key) {
    return get(root, key);
}

private Value get(Node x, Key key) {
    if (x == null)
        return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        return get(x.left, key);
    else if (cmp > 0)
        return get(x.right, key);
    else
        return x.val;
}
```
Practice Time

- Search for the keys 4 and 9 in the following BST:
BINARY SEARCH TREES

Insert

- If less than key in node go left.
- If greater than key in node go right.
- If null, insert.
- If already exists, update value.
- Number of compares is equal to the depth of the node + 1.
Insert example

Insertion into a BST
Insert

- **public** `void put(Key key, Value val) {`
  
  `root = put(root, key, val);`

`}`

- **private** `Node put(Node x, Key key, Value val) {`
  
  `if (x == null)`
  
  `return new Node(key, val, 1);`

  `int cmp = key.compareTo(x.key);`

  `if (cmp < 0)`
  
  `x.left = put(x.left, key, val);`

  `else if (cmp > 0)`
  
  `x.right = put(x.right, key, val);`

  `else`
  
  `x.val = val;`

  `x.size = 1 + size(x.left) + size(x.right);`

  `return x;`

`}`
Practice Time

- Add the key-value pairs (4,3) and (9,2) in the following BST:
3.2 Binary Search Tree Demo
Tree shape

- The same set of keys can result in different BSTs based on their order of insertion.
- Number of compares for search/insert is equal to depth of node +1.
BSTs mathematical analysis

- If $n$ distinct keys are inserted into a BST in random order, the expected number of compares of search/insert is $O(\log n)$.
  - If $n$ distinct keys are inserted into a BST in random order, the expected height of tree is $O(\log n)$. [Reed, 2003].
- Worst case height is $n$ but highly unlikely.
  - Keys would have to come (reversely) sorted!
- All ordered operations in a dictionary implemented with a BST depend on the height of the BST.
Hibbard deletion: Delete node which is a leaf

- Simply delete node.
- Example: delete 52 locates a node which is a leaf and removes it.
Hibbard deletion: Delete node with one child

- Delete node and replace it with its child.
- Example: delete 70 locates a node which has one child and replaces it with the child.
Hibbard deletion: Delete node with two children

- Delete node and replace it with successor (node with smallest of the larger keys). Move successor’s child (if any) where successor was.

- Example: delete 50 locates a node which has two children. Successor is 51.
public void delete(Key key) {
    root = delete(root, key);
}

private Node delete(Node x, Key key) {
    if (x == null) return null;

    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = delete(x.left, key);
    else if (cmp > 0)
        x.right = delete(x.right, key);
    else {
        if (x.right == null)
            return x.left;
        if (x.left == null)
            return x.right;
        Node t = x; // replace with successor
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    x.size = size(x.left) + size(x.right) + 1;
    return x;
Practice Time

- Delete the node 21 following Hibbard’s deletion
Answer

- Delete the node 21 following Hibbard’s deletion
Hibbard’s deletion

- Unsatisfactory solution. If we were to perform many insertions and deletions the BST ends up being not symmetric and skewed to the left.

- Extremely complicated analysis, but average cost of deletion ends up being $\sqrt{n}$. Let’s simplify things by saying it stays $O(\log n)$.

- No one has proven that alternating between the predecessor and successor will fix this.

- Hibbard devised the algorithm in 1962. Still no algorithm for efficient deletion in Binary Search Trees!

- Overall, BSTs can have $O(n)$ worst-case for search, insert, and delete. We want to do better (see future lectures).
Lecture 23: Binary Search Trees

- Dictionaries
- Binary Search Trees
ASSIGNED READINGS AND PRACTICE PROBLEMS

Readings:

- Textbook: Chapters 3.1 (Pages 362–386) and 3.2 (Pages 396-414)
- Website:
  - https://algs4.cs.princeton.edu/31elementary/
  - https://algs4.cs.princeton.edu/32bst/
- Visualization:
  - https://visualgo.net/en/bst

Practice Problems:

- 3.1.1-3.1.6, 3.2.1-3.2.13