CS 61B | Final Reference Sheet

Sorting

Algorithm	In place	Stable	Best	Worst	Space	Remarks
Selection Sort	✓		n^2	n^2	1	n exchanges; always pick next
Insertion Sort	✓	✓	n	n^2	1	for small/partially sorted arrays
Mergesort		✓	$n \log n$	$n \log n$	N	Divide-and-conquer
Quicksort	✓		$n \log n$	n^2	$\log N$	Shuffle first; fastest in practice
Heapsort	✓		$n^\dagger$	$n \log n$	1	† $n \log n$ if all keys distinct
LSD Radix		✓	$WN + WR$	$WN + WR$	$N + R$	N: Number of keys. R: Size of alphabet. W: Width of longest key
MSD Radix		✓	$N + R$	$WN + WR$	$N + WR$	

Symbol Tables

Data Structure	Worst Case			Average Case			
	Search	Insert	Delete	Search	Insert	Delete	Remarks
Binary Search Tree (Unbalanced)	n	n	n	$\log n$	$\log n$	$\sqrt{n}$	
LL Red-Black Tree	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	
Hash Table	n	n	n	$1^\dagger$	$1^\dagger$	$1^\dagger$	† uniform hashing

Graph Processing

Algorithm	Useful For	Time	Space	Remarks
DFS	Path; Cycle; Topological sort	$E + V$	V	
BFS	Shortest path (fewest edges)	$E + V$	V	
Kruskal	Minimum spanning tree	$E \log E$	$E + V$	
Prim	Minimum spanning tree	$E \log V$	V	
Dijkstra	Shortest paths (nonnegative weights)	$E \log V$	V	
Topological Sort	Shortest paths (no cycles)	$V + E$	V	

Algorithm	Union	Find	Algorithm	Union	Find
Quick-Find	N	1	**Weighted Quick-Union**	$\log N$	$\log N$
Quick-Union	$tree\ height$	$tree\ height$	**Weighted Quick-Union with Path Compression**	amortized close to 1	

Quick-Find: **p** and **q** are connected iff `id[p]` $=$ `id[q]`

Quick-Union: Each entry in `id[]` is the name of another element, i.e. a link

Runtime Analysis

$1 + 2 + 4 + 8 + \ldots + N \sim N$

$1 + 2 + 3 + 4 + \ldots + N \sim N^2$

$1 + 1/2 + 1/3 + \ldots + 1/N \sim \ln N$

Alternatively, replace sum with an integral!

Sorting

```
public class Selection { // Runtime is insensitive to input; Data movement is minimal
   public static void sort(Comparable[] a) {
      int N = a.length;              // array length
      for (int i = 0; i < N; I++) {  // Exchange a[i] with smallest entry in a[i+1 ... N)
         int min = i;                // index of minimal entry.
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min])) min = j;
         exch(a, i, min);
      }
   }
}
```

- Find the smallest item in the array and exchange it with the first entry
- Find the next smallest item and exchange it with the second entry
- Continue until the entire array is sorted

```
public class Insertion {
      public static void sort(Comparable[] a) {
            int N = a.length;
            for (int i = 1; i < N; i++) { // Insert a[i] among a[i-1], a[i-2], ...
                  for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
                        exch(a, j, j-1);
            }
      }
}
```

- Consider elements one at a time, inserting it to the appropriate position
- At each insertion, move larger items one position to the right

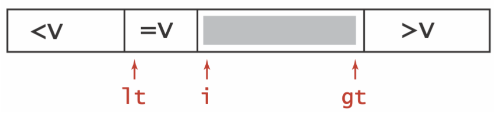
```
public class Merge {
      private static Comparable[] aux;         // auxiliary array for merges
      public static void sort(Comparable[] a) {
            aux = new Comparable[a.length];    // Allocate space just once.
            sort(a, 0, a.length - 1);
      }
      private static void sort(Comparable[] a, int lo, int hi) {  // Sort a[lo..hi].
            if (hi <= lo) return;
            int mid = lo + (hi - lo)/2;
            sort(a, lo, mid);       // Sort left half.
            sort(a, mid+1, hi);     // Sort right half.
            merge(a, lo, mid, hi);  // Merge results
      } // BottomUp mergesort is good for data organized in linked list, no recursion
} // Use insertion sort for small subarrays; Eliminate the auxiliary array
public static void merge(Comparable[] a, int lo, int mid, int hi) {
      int i = lo, j = mid+1;
      for (int k = lo; k <= hi; k++)   // Copy a[lo..hi] to aux[lo..hi].
            aux[k] = a[k];
      for (int k = lo; k <= hi; k++) { // Merge back to a[lo..hi].
            if      (i > mid)              a[k] = aux[j++]; // Left half exhausted
            else if (j > hi )              a[k] = aux[i++]; // Right half exhausted
            else if (less(aux[j], aux[i])) a[k] = aux[j++]; // Right key less than left
            else                           a[k] = aux[I++];
      }
}
```

```
public class Quick {
      public static void sort(Comparable[] a) {
            StdRandom.shuffle(a);          // Eliminate dependence on input.
            sort(a, 0, a.length - 1);
      }
      private static void sort(Comparable[] a, int lo, int hi) {
            if (hi <= lo) return;
            int j = partition(a, lo, hi);  // Partition (see page 291).
            sort(a, lo, j-1);              // Sort left part a[lo .. j-1].
            sort(a, j+1, hi);              // Sort right part a[j+1 .. hi].
      }
}
private static int partition(Comparable[] a, int lo, int hi) {
      int i = lo, j = hi+1;            // left and right scan indices
      Comparable v = a[lo];            // partitioning item
      while (true) {  // Scan right, scan left, check for scan complete, and exchange.
            while (less(a[++i], v)) if (i == hi) break;
            while (less(v, a[--j])) if (j == lo) break;
            if (i >= j) break;
            exch(a, i, j);
      }
      exch(a, lo, j);
      return j;
}
```

Quicksort (Dijkstra) partitioning: Repeat until **i** and **j** pointers cross

- Scan **i** from left to right so long as `a[i] < a[lo]`
- Scan **j** from left to left so long as `a[j] > a[lo]`
- Exchange `a[i]` with `a[j]`

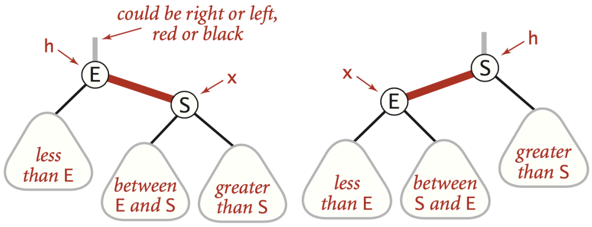
```
public class Quick3way {
      private static void sort(Comparable[] a, int lo, int hi) {
            if (hi <= lo) return;
            int lt = lo, i = lo+1, gt = hi;
            Comparable v = a[lo];
            while (i <= gt) {
                  int cmp = a[i].compareTo(v);
                  if      (cmp < 0) exch(a, lt++, i++);
                  else if (cmp > 0) exch(a, i, gt--);
                  else              i++;
            }  // Now a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi].
            sort(a, lo, lt - 1);
            sort(a, gt + 1, hi);
      }
} // Good for many duplicate keys; bad performance if keys are unique
```




```
public class HeapSort {
      public static void sort(Comparable[] a) {
            int N = a.length;
            for (int k = N/2; k >= 1; k--)
                  sink(a, k, N);
            while (N > 1) {
                  exch(a, 1, N--);
                  sink(a, 1, N);
            }
      }
}
```

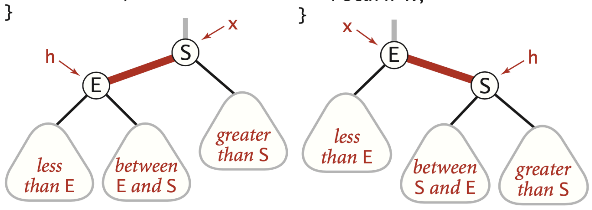
Quicksort better for primitive types (unstable doesn't matter) whereas mergesort better for objects (stability)

Symbol Tables



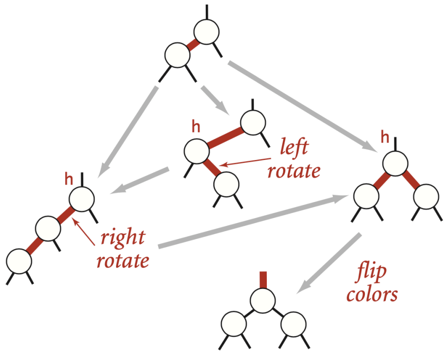

```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

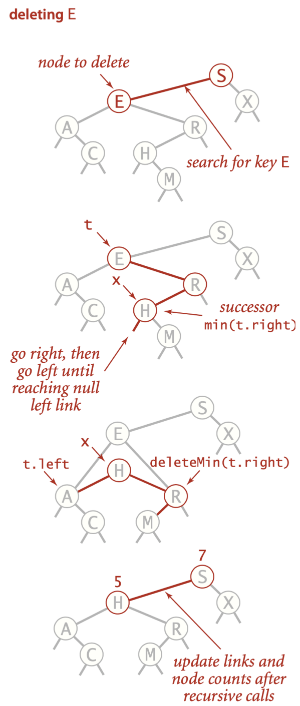


Left rotate (right link of h)

Right rotate (left link of h)



Passing a red link up a red-black BST



Deletion in a BST

Hibbard Deletion

- Save a link to the node to be deleted in **t**
- Set **x** to point to its successor `min(`**t**`.right)`
- Set the right link of **x** to `deleteMin(`**t**`.right)`, the link to the BST containing all the keys that are larger than **x**`.key` after the deletion
- Set the left link of **x** to **t**`.left`

```
public class SeparateChainingHashST<Key, Value> {
      private int N, M;  // number of key-value pairs, hash table size
      private SequentialSearchST<Key, Value>[] st;  // array of ST objects
      public SeparateChainingHashST(int M) {  // Create M linked lists.
            this.M = M;
            st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[M];
            for (int i = 0; i < M; i++)
                  st[i] = new SequentialSearchST();
      }
      private int hash(Key key) {  return (key.hashCode() & 0x7fffffff) % M; }
      public Value get(Key key) {  return (Value) st[hash(key)].get(key);  }
      public void put(Key key, Value val) {  st[hash(key)].put(key, val);  }
}
```

`hashCode()` must be consistent with equals → if `a.equals(b)`, `a.hashCode()` must equal `b.hashCode()`
→ Do not override `equals()` without overriding `hashcode()`

```
public interface Iterable<Item> {
      Iterator<Item> iterator();
}
```

```
public interface Comparator<Key> {
      int compare(Key v, Key w)
}
```

Priority Queue

Insert = Add new key at the end of the array; `size++`; swim up through the heap
Remove max = Pop the largest key (index 0) and put item from end of heap at the top; `size--`; sink

```
private void sink(int k) {
   while (2*k <= N) {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
private void swim(int k) {
   while (k > 1 && less(k/2, k)) {
      exch(k/2, k);
      k = k/2;
   }
}
```

Graphs

Kruskal Algorithm: $E \log E$

Consider edges in ascending order of weight

- Add next edge to tree **T** unless doing so would create a cycle

Operation	Frequency	Time Per Op
Build PQ	1	E
Delete Min	E	$\log E$
Union	V	$\log^\star V^\dagger$
Connected	E	$\log^\star V^\dagger$

† amortized of WQU with path compression;
If edges are already sorted, $E \log^\star V$ time

Prim's Algorithm: $E \log V$

- Start with some vertex and greedily grow tree **T**
- Add to **T** the min weight edge with exactly one endpoint in **T**
- Repeat until **V-1** edges

Operation	Frequency	Time Per Op
Build PQ	1	V
Insert	V	$\log V$
Delete Min	V	$\log V$
Δ Priority	E	$\log V$

Assuming $E > V$

Dijkstra's Algorithm: $E \log V$ (assuming $E > V$)

Operation	Frequency	Time Per Op
PQ add	V	$\log V$
PQ removeSmallest	V	$\log V$
PQ changePriority	E	$\log V$

Stacks = Last-in-First-out
Queue = First-in-First-out

Underlying Data Structure	Space	Add Edge	Check whether w is adjacent to v	Iterate through vertices adjacent to v
List of Edges	E	1	1	1
Adjacency Matrix	V^2	1	1	V
Adjacency Lists	$E+V$	1	$degree(V)$	$degree(V)$
Adjacency Sets	$E+V$	$\log V$	$\log V$	$\log V + degree(V)$

```java
public class PrimMST {
      private Edge[] edgeTo;          // shortest edge from tree vertex
      private double[] distTo;        // distTo[w] = edgeTo[w].weight()
      private boolean[] marked;       // true if v on tree
      private IndexMinPQ<Double> pq; // eligible crossing edges
      public PrimMST(EdgeWeightedGraph G) {
            edgeTo = new Edge[G.V()];
            distTo = new double[G.V()];
            marked = new boolean[G.V()];
            for (int v = 0; v < G.V(); v++) distTo[v] = Double.POSITIVE_INFINITY;
            pq = new IndexMinPQ<Double>(G.V());
            distTo[0] = 0.0;
            pq.insert(0, 0.0);             // Initialize pq with 0, weight 0.
            while (!pq.isEmpty())
                  visit(G, pq.delMin());  // Add closest vertex to tree.
      }
      private void visit(EdgeWeightedGraph G, int v) { // Add v to tree; update
            marked[v] = true;
            for (Edge e : G.adj(v)) {
                  int w = e.other(v);
                  if (marked[w]) continue;
                  if (e.weight() < distTo[w]) { // Edge e is new best connection
                        edgeTo[w] = e;
                        distTo[w] = e.weight();
                        if (pq.contains(w)) pq.change(w, distTo[w]);
                        else                pq.insert(w, distTo[w]);
                  }
            }
      }
      public Iterable<Edge> edges()
      public double weight()
}
```

- Start with vertex **`0`** and greedily grow tree **`T`**
- Add to **`T`** the min weight edge with exactly one endpoint in **`T`**
- Repeat until **`V-1`** edges

```java
public class KruskalMST {
      private Queue<Edge> mst;
      public KruskalMST(EdgeWeightedGraph G) {
            mst = new Queue<Edge>();
            MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());
            UF uf = new UF(G.V());
            while (!pq.isEmpty() && mst.size() < G.V()-1) {
                  Edge e = pq.delMin();                 // Get min weight edge on pq
                  int v = e.either(), w = e.other(v); //   and its vertices.
                  if (uf.connected(v, w)) continue;   // Ignore ineligible edges.
                  uf.union(v, w);                       // Merge components.
                  mst.enqueue(e);                       // Add edge to mst.
            }
      }
      public Iterable<Edge> edges() {  return mst;  }
}
```

- choose the edge of lightest weight in the graph
- add it to the MST if adding that edge does not create a cycle

```
public class DijkstraSP {
      private DirectedEdge[] edgeTo;
      private double[] distTo;
      private IndexMinPQ<Double> pq;
      public DijkstraSP(EdgeWeightedDigraph G, int s) {
            edgeTo = new DirectedEdge[G.V()];
            distTo = new double[G.V()];
            pq = new IndexMinPQ<Double>(G.V());
            for (int v = 0; v < G.V(); v++)
                  distTo[v] = Double.POSITIVE_INFINITY;
            distTo[s] = 0.0;
            pq.insert(s, 0.0);
            while (!pq.isEmpty())
                  relax(G, pq.delMin())   // Always take the vertex that is closest
      }
      private void relax(EdgeWeightedDigraph G, int v) {
            for (DirectedEdge e : G.adj(v)) {
                  int w = e.to();
                  if (distTo[w] > distTo[v] + e.weight()) { // Update shortest distance
                        distTo[w] = distTo[v] + e.weight();
                        edgeTo[w] = e;
                        if (pq.contains(w)) pq.change(w, distTo[w]);
                        else                pq.insert(w, distTo[w]);
                  }
            }
      }
}
```

- Start with `distTo` and `edgeTo`, and consider vertices in increasing distance from **s**
- Always pick the next shortest path
- For every adjacent edge, update the minPQ

```
bfs(graphNode s) {
   queue = new Queue!<>()
   mark s as visited;
   queue.add(s);
   while (!queue.isEmpty()):
      v = queue.dequeue();
      for each unmarked neighbor of v:
         edgeTo[neighbor] = v;
         marked[neighbor] = true;
         queue.add(neighbor); }
```

```
dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v))
        if (!marked[w])
            dfs(G, w); }
```

```
public class DepthFirstSearch {
      private boolean[] marked;
      public DepthFirstSearch(Graph G, int s) {
            marked = new boolean[G.V()];
            dfs(G, s);
      }
      private void dfs(Graph G, int v) {
            marked[v] = true;
            for (int w : G.adj(v))
                  if (!marked[w]) dfs(G, w);
      }
}
```

```
hoare_partition(arr[], lo, hi)
   pivot = arr[lo]
   i = lo - 1  // Initialize left index
   j = hi + 1  // Initialize right index
   // Find a value in left side greater
   // than pivot
   while (arr[I] < pivot)
      i = i + 1
  // Find a value in right side smaller
  // than pivot
   while (arr[j] > pivot);
      j--;
   if i ⩾ j then
      return j
   swap arr[i] with arr[j]
```

Strings

```
String[] arr1 = new String[5];
int[] arr2 = new int[] {3, 10, 7};
char[] arr3 = {‘a’, ‘e’, ‘i’, ‘o’, ‘u’};
```

```
public class LSD {
      public static void sort(String[] a, int W) {
            int N = a.length;
            int R = 256;
            String[] aux = new String[N];
            for (int d = W-1; d ≥ 0; d--) {
                  int[] count = new int[R+1];
                  for (int i = 0; i < N; i++)         // Compute frequency counts.
                        count[a[i].charAt(d) + 1]++;
                  for (int r = 0; r < R; r++)         // Transform counts to indices.
                        count[r+1] += count[r];
                  for (int i = 0; i < N; i++)         // Distribute the data
                        aux[count[a[i].charAt(d)]++] = a[i];
                  for (int i = 0; i < N; i++)         // Copy back
                        a[i] = aux[i];
            }
      }
}
```

MSD: Use recursion to sort sub-arrays of the same prefix

- Performance relies on small subarrays that have the same prefix
- Need to switch to insertion sort for small subarrays
- MSD runtime varies from sublinear (average) to linear (worst case where all keys are equal)

When considering a number of LSD / MSD Radix sort, Bitstring is equal to $\log(N)$ where N is the largest number

Trie = Nodes with links; each node represents a letter
Search hit takes time proportional to the length of the search key
Search miss involves examining only a few characters
Not suitable for large numbers of long keys taken from large alphabets (otherwise takes excessive space)