CPSC 223 Midterm #2: In-Depth Study Notes

Your Name

Contents

1 Hash Tables

Concept & Motivation

A *hash table* stores key–value pairs in an array of size m. A *hash function* $h(k)$ maps each key k to an index in $\{0, \ldots, m-1\}$. With n elements, the *load factor* $\alpha = n/m$ governs performance:

$$\text{Average cost of lookup/insert/delete} = \Theta\big(1 + \alpha\big)\,.$$

To keep operations $O(1)$ amortized, maintain $\alpha = O(1)$ by resizing when α exceeds a threshold (e.g. 0.75).

Collision Resolution

Separate chaining: Each bucket holds a linked list (or dynamic array) of entries.

$$\text{Cost} = \Theta\big(1 + \alpha\big)\,.$$

Open addressing: All entries live in the array; on collision probe:

- **Linear probing:** $h_i(k) = (h(k) + i) \bmod m$.
- **Quadratic probing:** $h_i(k) = (h(k) + c_1 i + c_2 i^2) \bmod m$.
- **Double hashing:** $h_i(k) = (h(k) + i \cdot h'(k)) \bmod m$.

Expected probes $\approx 1/(1 - \alpha)$.

Resizing

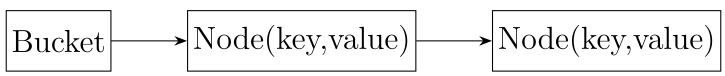
When $\alpha > \alpha_{\max}$ (e.g. 0.75), allocate new table of size $\approx 2m$ and rehash all n keys in $\Theta(n)$. This yields amortized $\Theta(1)$ insertion cost.

Example: Separate Chaining

```
#define TABLE_SIZE 101

typedef struct Node {
    int key;
    int value;
    struct Node *next;
} Node;

Node* table[TABLE_SIZE];

unsigned int hash(int key) {
    return (unsigned)key % TABLE_SIZE;
}
```

```c
void ht_insert(int key, int value) {
    unsigned idx = hash(key);
    Node* n = malloc(sizeof *n);
    n->key = key; n->value = value;
    n->next = table[idx];
    table[idx] = n;
}

Node* ht_search(int key) {
    unsigned idx = hash(key);
    for (Node* cur = table[idx]; cur; cur = cur->next)
        if (cur->key == key) return cur;
    return NULL;
}
```

Diagram: Separate Chaining Bucket



2 Binary Search Trees (BST)

Invariant

For every node x:

$$\forall\, y \in \mathrm{left}(x) : \; y.key < x.key, \quad \forall\, z \in \mathrm{right}(x) : \; z.key > x.key.$$

If the tree has height h, operations run in $O(h)$. In the average case $h \approx \log n$; in the worst case $h = n$.

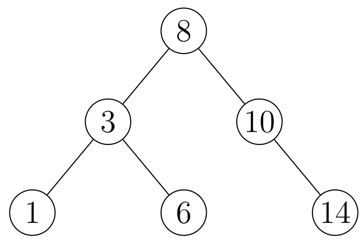
Operations

- **Search:** Compare at current node, recurse left or right.
- **Insert:** BST-search until a `NULL`, then link a new node.
- **Delete:** Three cases:
  1. Leaf: remove directly.
  2. One child: link parent to child.
  3. Two children: replace key with in-order successor (minimum of right subtree), then delete that node.

Example Code

```
typedef struct BSTNode {
    int key;
    struct BSTNode *left, *right;
} BSTNode;

BSTNode* bst_search(BSTNode* r, int k) {
    if (!r || r->key == k) return r;
    if (k < r->key) return bst_search(r->left, k);
    else return bst_search(r->right, k);
}

BSTNode* bst_insert(BSTNode* r, int k) {
    if (!r) {
        BSTNode* n = malloc(sizeof *n);
        n->key = k; n->left = n->right = NULL;
        return n;
    }
    if (k < r->key) r->left = bst_insert(r->left, k);
    else if (k > r->key) r->right = bst_insert(r->right, k);
    return r;
}
```

Diagram: BST Example



3 AVL Trees

Balance Factor

For node x:

$$bf(x) = \text{height}(x.\text{left}) - \text{height}(x.\text{right}), \quad |bf(x)| \leq 1.$$

Insertion or deletion may violate this; if $|bf(x)| = 2$, we perform rotations.

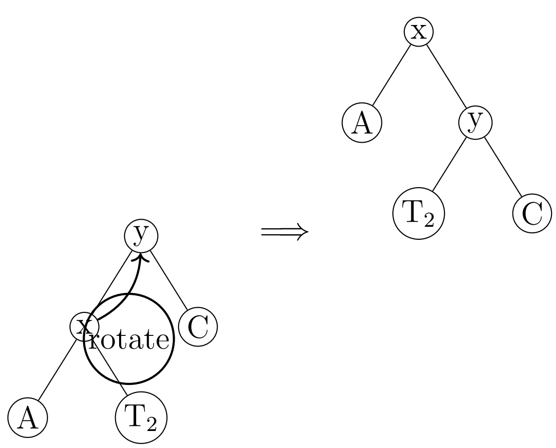
Rotations

Right Rotation (LL Case) Applicable when $bf(y) = +2$ and $bf(y.\text{left}) \geq 0$.

$$\begin{aligned}
&x = y.\text{left}, \quad T_2 = x.\text{right}, \\
&x.\text{right} := y, \quad y.\text{left} := T_2, \\
&\text{update heights of } y, \text{ then } x.
\end{aligned}$$

```
AVLNode* rightRotate(AVLNode* y) {
    AVLNode* x = y->left;
    AVLNode* T2 = x->right;
    // Perform rotation
    x->right = y;
    y->left = T2;
    // Update heights
    y->height = 1 + max(height(y->left), height(y->right));
    x->height = 1 + max(height(x->left), height(x->right));
    return x; // new root
}
```

Diagram: Right Rotation



Left Rotation (RR Case) Mirror of right rotation when $bf(y) = -2$ and $bf(y.\text{right}) \leq 0$:

```
AVLNode* leftRotate(AVLNode* x) {
    AVLNode* y = x->right;
    AVLNode* T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = 1 + max(height(x->left), height(x->right));
    y->height = 1 + max(height(y->left), height(y->right));
    return y;
}
```

Double Rotations

- **Left–Right (LR) Case:** $bf(y) = +2$ and $bf(y.\text{left}) = -1$: leftRotate(y.left); then rightRotate(y);.
- **Right–Left (RL) Case:** $bf(y) = -2$ and $bf(y.\text{right}) = +1$: rightRotate(y.right); then leftRotate(y);.

Insertion Algorithm

1. Insert as in a regular BST.
2. On unwind, update each ancestor's height and compute bf.
3. At first unbalanced node, apply the appropriate single or double rotation.

All steps cost $O(\log n)$.

4 Red–Black Trees

Properties

1. Every node is **red** or **black**.
2. The root is black.
3. Red nodes have only black children.
4. Every path from a node to its NIL leaves has the same number of black nodes (the *black height*).

These ensure height $O(\log n)$.

Insertion Fix-Up

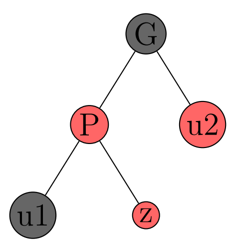
After inserting a red node z:

Case 1 (Uncle red): Recolor parent and uncle to black, grandparent to red, then repeat fix-up on grandparent.

Case 2 (Uncle black, Triangle): Rotate parent toward z to form a line (converts to Case 3).

Case 3 (Uncle black, Line): Rotate grandparent opposite direction, swap colors of parent and grandparent, finish.

Diagram: RB Insertion Case 1



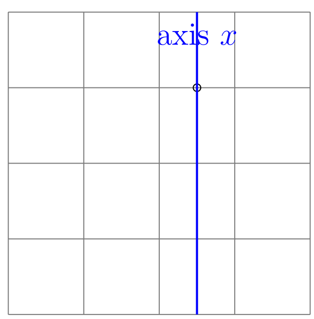
5 k-d Trees

Structure & Build

Stores n points in $\mathbb{R}^k$. At depth d, split on axis $d \bmod k$. Build in $O(n \log n)$ via median-of-axis selection (e.g. `nth_element`).

```
// Build a 2D k-d tree
kdnode* build(Point pts[], int l, int r, int depth) {
    if (l >= r) return NULL;
    int axis = depth % 2;
    int m = (l + r) / 2;
    nth_element(pts + l, pts + m, pts + r,
      [axis](const Point &a, const Point &b){
        return axis ? a.y < b.y : a.x < b.x;
      });
    kdnode* node = malloc(sizeof *node);
    node->pt = pts[m];
    node->left = build(pts, l, m, depth + 1);
    node->right = build(pts, m + 1, r, depth + 1);
    return node;
}
```

Diagram: 2D Split



6 Binary Heaps

Array Representation

A complete binary tree in an array $A[0..n-1]$. For index i:

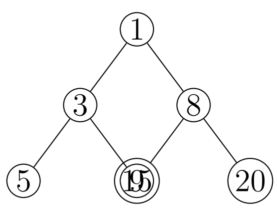
$$\text{parent}(i) = \lfloor (i-1)/2 \rfloor, \quad \text{children}(i) = 2i+1,\ 2i+2.$$

Operations

- **Insert:** append at end, *bubble up* in $O(\log n)$.
- **Extract-Min:** swap root with last, remove last, *bubble down* via minHeapify in $O(\log n)$.
- **Build-Heap:** call minHeapify from $\lfloor n/2 \rfloor$ down to $0 \to \Theta(n)$.

```
// Min-heapify at index i
void minHeapify(int A[], int n, int i) {
    int l = 2*i+1, r = 2*i+2, smallest = i;
    if (l < n && A[l] < A[smallest]) smallest = l;
    if (r < n && A[r] < A[smallest]) smallest = r;
    if (smallest != i) {
        swap(&A[i], &A[smallest]);
        minHeapify(A, n, smallest);
    }
}
```

Diagram: Heap Tree & Array



1	3	8	5	9	15	20

7 Graphs: BFS & DFS

Representations

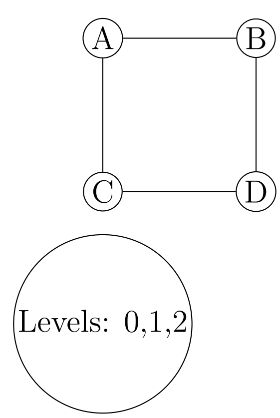
- *Adjacency list:* $O(V + E)$ space.
- *Adjacency matrix:* $O(V^2)$ space.

Breadth-First Search (BFS)

Visits vertices by increasing distance from source. Computes shortest paths in unweighted graphs in $O(V + E)$.

```
void bfs(Graph* g, int s) {
    bool seen[g->V]; int dist[g->V];
    for(int i=0;i<g->V;i++){seen[i]=false; dist[i]=INT_MAX;}
    Queue q; init(&q);
    seen[s]=true; dist[s]=0; enqueue(&q,s);
    while(!empty(&q)) {
        int u = dequeue(&q);
        for(Edge* e = g->adj[u]; e; e = e->next) {
            int v = e->to;
            if (!seen[v]) {
                seen[v] = true;
                dist[v] = dist[u] + 1;
                enqueue(&q, v);
            }
        }
    }
}
```

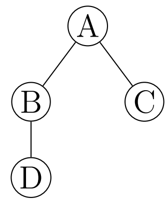
Diagram: BFS Levels



Depth-First Search (DFS)

Explores as deep as possible before backtracking. Useful for cycle detection, topological sort, SCCs.

Diagram: DFS Tree



8 Shortest-Path Algorithms

BFS for Unweighted Graphs

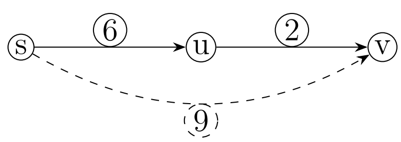
Since BFS explores by layers, it finds shortest paths in unweighted graphs in $O(V + E)$.

Dijkstra's Algorithm

For non-negative weights. Uses a min-priority queue. Complexity $O((V + E)\log V)$.

```
void dijkstra(Graph* g, int src, int dist[]) {
    bool vis[g->V];
    for(int i=0;i<g->V;i++){dist[i]=INT_MAX; vis[i]=false;}
    dist[src] = 0;
    PriorityQueue pq;
    pq.push({0,src});
    while (!pq.empty()) {
        auto [d,u] = pq.pop();
        if (d > dist[u]) continue;
        for (Edge* e = g->adj[u]; e; e = e->next) {
            int v = e->to, w = e->w;
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
}
```

Diagram: Dijkstra's Relaxation



Bellman–Ford

Handles negative weights (no negative cycles). Relax all edges $V - 1$ times in $O(VE)$. A Vth pass detecting further relaxation signals a negative cycle.