CPSC 331: DATA STRUCTURES, ALGORITHMS, AND THEIR ANALYSIS

GO UEZONO

April 11, 2024

CONTENTS

LIST OF FIGURES

LIST OF TABLES

3.2 Red and Black Trees

3.2.1 *Properties*

A **red-black tree** is a concrete implementation of a **self-balancing binary-search tree** (reference here) that automatically maintains balance. Giving each node their respective color ensures that no path is more than twice as long as any other, thus is able to maintain approximate balance.

1. Every node is red/black
2. Root must be black
3. Leaves (*null*) are black
   - *null* vertices contain no values, while other (interior) do
4. If a node is red, then both its children are black
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes

The following **lemma** shows why red-black trees make good search trees:

Lemma 1. *A red-black tree with n internal nodes has height at most $2\log(n+1)$*

Proof. Start by showing subtree rooted at any ndoe x x contains at least a $2^{bh(x)} - 1$ internal nodes. We prove this by **mathematical induction** on the height of x.

CLAIM: If height of $x = 0$, then the leaf must be *T.null*, and the subtree rooted at x contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

INDUCTIVE STEP:
- Consider a node x that has positive height and is an internal node with two children.
- Each *child* has a black-height of either $bh(x)$ or $bh(x) - 1$ (depending on whether it is red or black respectively).
- Since height of a *child* of x is less than the height of x itself, we can apply the **I.H** to conlude that:
  - Each child has at least $2^{bh(x)-1} - 1$ internal nodes.

Thus, subtree rooted at x contains at least

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1$$

internal nodes, which proves the claim.

To complete the proof, let h be the height of thr tree. According to property 4 (reference above Properties), at least half the nodes from the root to a leaf (not including the root) must be black.

Consequently, the bh of the root must be at least $h/2$; thus,

$$n \geqslant 2^{h/2} - 1$$

Moving 1 to the left side and taking log on both sides yields:

$$\log(n+1) \geqslant h/2$$

or

$$h \leqslant 2\log(n+1)$$

□

3.2.2 *Rotational Properties*

Search operations *TREE-INSERT* and *TREE-DELETE* take $O(\log n)$ time. Since modifications are done to the tree, we must change the color of some of the nodes.

Definition (Rotation). Local operation that preserves the binary-tree property.

- **Left Rotation:** assume that its right child y is not *null*
- **Right Rotation:** assume that its left child y is not *null*
  - x can be any node on the tree whose respective child is not *null*
  - Left/Right rotations "pivots" around the link from x to y
  - Makes y the new root, x as y's left(right) child, y's left(right) child as x's right(left) child
- Both L/R rotates run in $O(1)$ time
- Only pointers are changed, all attributes in a node remain the same

Algorithm 1: Left-Rotate(T, x)

```
1  x = y.right ;                                                    // set y
2  x.right = y.left ;                    // Turn y's left subtree into x's right subtree
3  if y.left ≠ T.null then
4  |  y.left.p = x;
5  y.p = x.p;
6  if x.p == T.null then
7  |  T.root = y;
8  else if x == x.p.left then
9  |  x.p.left = y;
10 else
11 |  x.p.right = y;
12 y.left = x;
13 x.p = y;
```

3.2.3 *Insertion*

Inserting a node can be done in $O(\log n)$ time. Below is a pseudo-code that shows how insertion *RB-INSERT* works:

Algorithm 2: RB-INSERT(T, z)

Data: z node to insert,

```
1  y = T.null;
2  x = T.root;
3  while x ≠ T.null do
4  |   y = x;
5  |   if z.key < x.key then
6  |   |   x = x.left;
7  |   else
8  |   |   x = x.right;
9  |   end
10 end
11 z.p = y;
12 if y == T.null then
13 |   T.root = z;
14 else if z.key < y.key then
15 |   y.left = z;
16 else
17 |   y.right = z;
18 z.left = T.null;
19 z.right = T.null;
20 z.color = T.RED;
21 RB-INSERT(T, z);
```

To ensure we preserve the red-black properties,

3.3 Heaps

3.3.1 *Priority Queues*

- Priority queues are **NOT** FIFO
- These queues are interested in removing items (dequeue) with the **highest priority**
- Assume that higher priority value (HPV) entails higher priority
  - Not true in general (UNIX OS; smaller PV = higher priority)
- Similar operations as the standard Queue ADT:

```
// Queue ADT
public interface QueueADT<T> {
   public void enqueue(T item);
   public T dequeue(); // different implementation
   public boolean isEmpty();
   public boolean isFull();
}
```

PRIORITY QUEUE IMPLEMENTATIONS

- **Lists**
  - **Sorted** list by PV with array implementation
    - *Ascending order*: Remove **last** item
    - *Dequeue*: $O(1)$, n^{th} element of the array to be removed
    - *Enqueue*: $O(n)$, needs to be sorted after enqueue
  - **Sorted** list by PV with linked-list implementation
    - *Descending order*: Remove **first** item
    - *Ascending order*: Circular list implementation
    - *Dequeue*: $O(1)$
    - *Enqueue*: $O(n)$
  - **Unsorted** list
    - *Dequeue*: $O(n)$
    - *Enqueue*: $O(1)$
- **BST**
  - *Dequeue*: $O(\log(n))$ average-case, $O(n)$ worst-case
  - *Enqueue*: $O(\log(n))$ average-case, $O(n)$ worst-case
- **Heaps**
  - *Dequeue*: $O(\log(n))$ worst-case
  - *Enqueue*: $O(\log(n))$ worst-case

3.3.2 *Properties*

A heap is a **complete** binary tree that satisfies the Heap Property:

- Each node of a tree corresponds to an element of an array
  - Always stored **contiguously**
    - If there are blanks, they are on the rightside of the array
    - Otherwise, no blanks inbetween indices
- It is of height h and contains n nodes

Proof. Height h □

HEAP PROPERTIES (Figure 1)

- **Min Heap Property**
  - Every node has a value $\leqslant$ than the value of its children
  - Root of any subtree has the *minimum* value in the subtree
- **Max Heap Property**
  - Every node has a value $\geqslant$ than the value of its children
  - Root of any subtree has the *maximum* value in the subtree
- **Child-to-Parent Relation**
  - Given a **child** a location *loc*, what is the index of **parent** *parent*?
  - If Child is right, index is **even**
    - $\text{loc}_r = 2 \times \text{parent} + 2$
  - If Child is left, index is **odd**
    - $\text{loc}_l = 2 \times \text{parent} + 1$
  - In either cases, $\text{parent} = (\text{loc} - 1) \div 2$

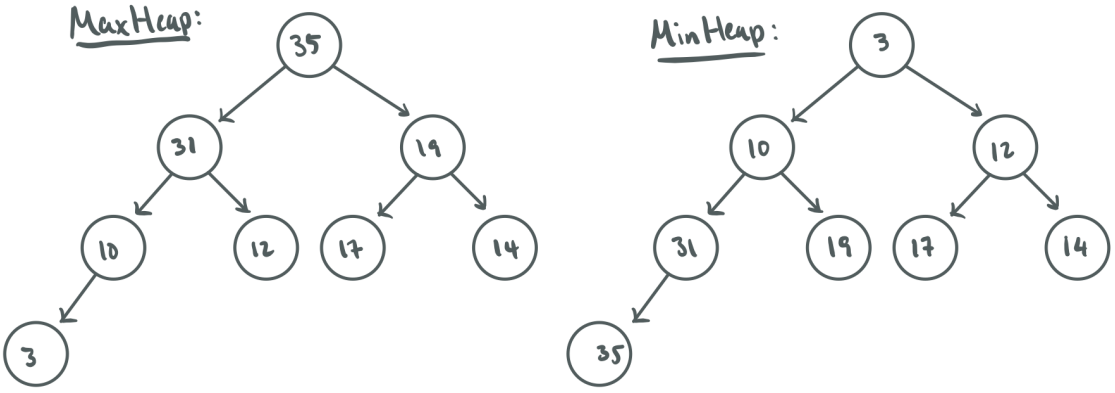

Figure 1: MaxHeap/MinHeap Diagram (3.3.2)

MAXHEAP CLASS Similar to the Queue ADT, differences in the enqueue and dequeue functions to maintain **heap** property

```
public class MaxHeap<T> {
   private T[] queue;
   private int size;

   public MaxHeap(Class<T> clazz, int maxSize) {
      queue = (T[]) Array.newInstance(clazz, maxSize);
      size = 0;
   }
   public boolean isEmpty() {
      return (size == 0);
   }
   public boolean isFull() {
      return (size == queue.length);
   }
}
// enqueue()/dequeue() functions shown later
```

3.3.3 *Insertion*

Definition. Inserting an element (Figure 2 on the following page)

- Must keep the tree **complete**
- Must keep the **max heap** property
- **Complexity of *Enqueue***
  - *Worst-case*: added node percolates from leaf to root
  - Since the tree is **complete**, height is $O(\log(n))$
  - Hence, **enqueue** is $O(\log(n))$

Enqueue:

```
public void enqueue(T item) {
   queue[size] = item;

   // Fix heap
   int loc = size;
   int parent = (loc - 1)/2;
   while (loc > 0 && queue[loc].compareTo(queue[parent]) > 0) {
      swap(loc, parent);
      loc = parent;
      parent = (loc - 1)/2;
   }
   size++;
}
```

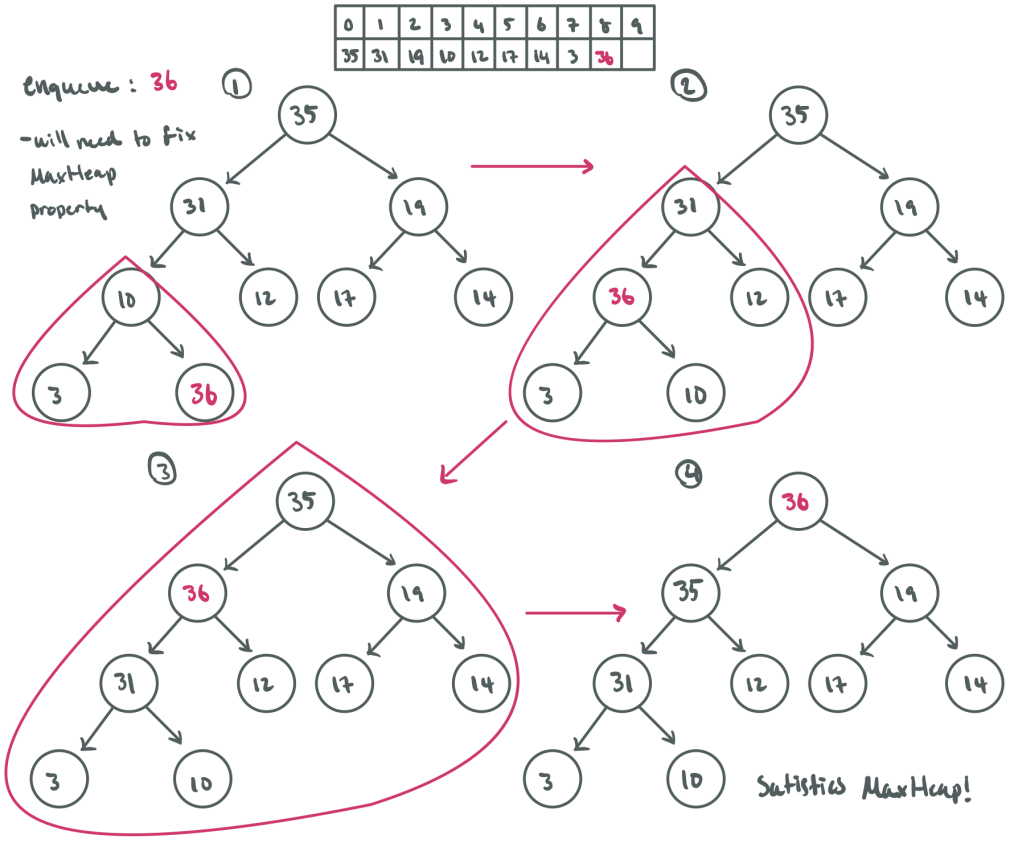

Figure 2: MaxHeap: Enqueue

Correctness: **Loop Invariant** for Enqueue

Proof. □

3.3.4 *Deletion*

Definition. Deleting an element (Figure 3 on the next page)

- Must keep the tree **complete**
- Must keep the **max heap** property
- This process is called *sinking*, where the leaf moves to location (index) 0
- We then *sink* the leaf back to its respective location
- **Complexity of *Dequeue***
  - Deleted root "hole" always sinks to a leaf
  - Since the tree is complete, height is $O(\log(n))$
  - Hence, **dequeue** is $O(\log(n))$

Dequeue

```
public T dequeue() {
   T max = queue[0];
   queue[0] = queue[size - 1];
   sink(0);
   queue[size - 1] = null;
   size--;
   return max;
 }
```

Sink

```
int swapWith
int left = 2*loc + 1;
int right = 2*loc + 2;
T tmp = queue[loc];

if (left > (size - 1)) { return;} // Reached a leaf
else if (left == size - 1) { // Node with one child
   if (tmp.compareTo(queue[left] < 0)) {
      swap(loc, left);
   }
} else { // Node with two children
   if (queue[left].compareTo(queue[right]) < 0) {
      swapWith = right;
   } else {
      swapWith = left;
   }
   if (tmp.compareTo(queue[left]) < 0) {
      swap(loc, swapWith);
   }
   sink(swapWith);
}
```

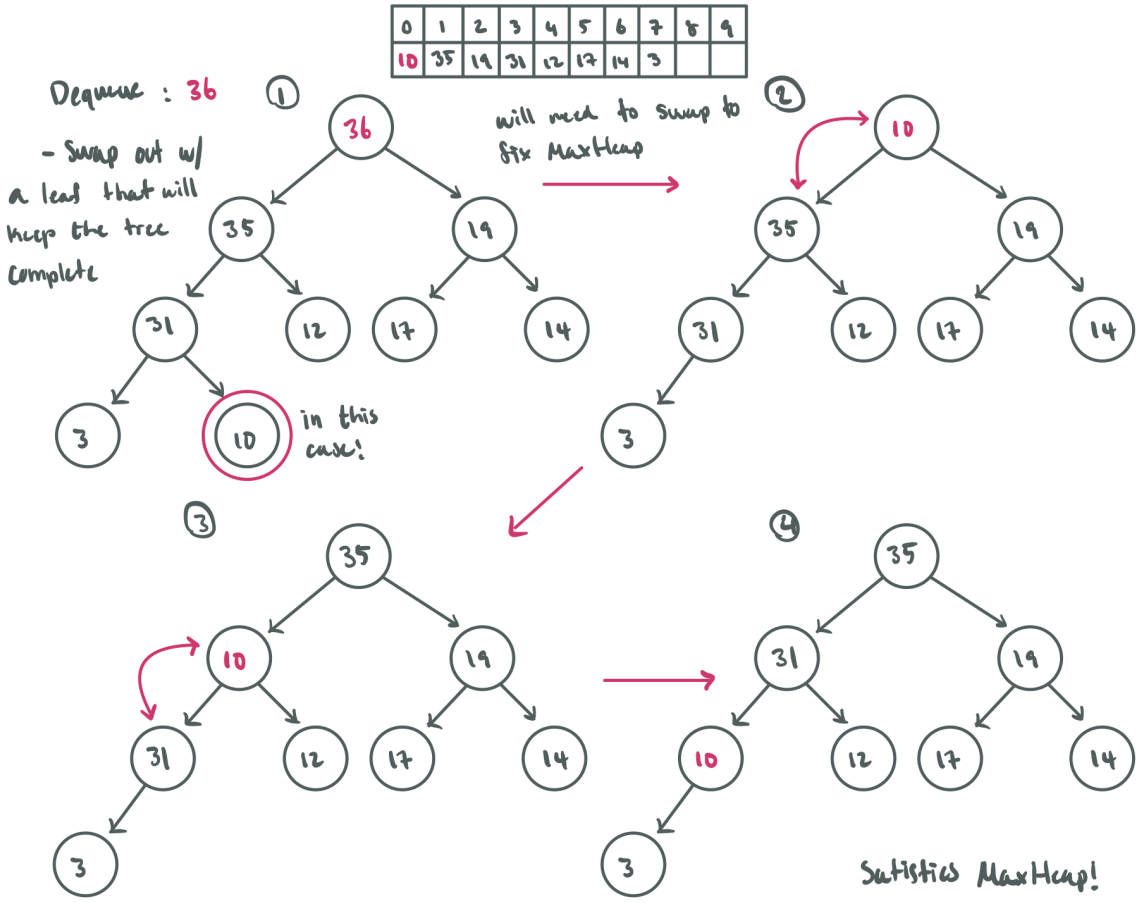

Figure 3: MaxHeap: Dequeue

Correctness: **Bound Function** for Dequeue

Proof. Prove that sink() maintains heap property □

3.4 Hash Table

3.4.1 *Properties*

Definition. *Hash Table ADT*

- Allows adding, removing, searching for items in $O(1)$ on average
- Implementation is called *hashing*
- Objects are mapped to a hash table based on a value calculated by a **hash function** using an object's key
  - e.g. ID of a Student object as a *key*
- Hash function:

$$h(x) : U \to \{0, 1, ..., m-1\}$$

  - $U \to$ set of all Objects
  - $|U| = n$
  - $m \to$ hash table size (buckets)
  - Typically $n > m$ (much larger)
  - Given an object $x \in U$, $h(x)$ maps x to a bucket in the hash table
- Performance of the hash table is dependent on the quality of the hash function
- A **Good** hash function distributes items *evenly* among buckets in the hash table

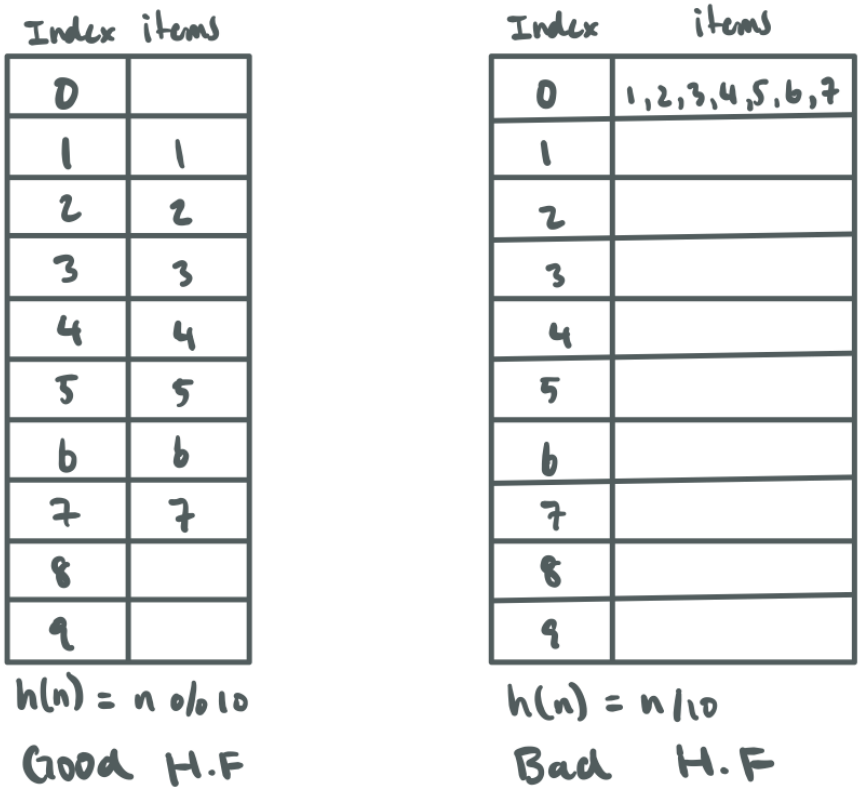

Figure 4: Good vs Bad Hash Functions

Collisions

- Occurs when two objects map to the same hash value
  - Objects o_1 and $o_2 \rightarrow h(o_1) = h(o_2)$
  - Unavoidable since $n > m$
- Can be dealt with:
  - Chaining
  - Probing/Open addressing
  - Double hashing

3.4.2 *Chaining*

Definition. *Chaining*

- Group items with the same hash value (via a hash function) into a bucket (AKA a chain)
- e.g. Figure 4 on the preceding page
- **Time Complexity**:
  - *Worst-case*: Hash table becomes a single LL if elements are mapped to one bucket - $O(n)$ search
  - *Best-case*: Search can become $O(1)$, but is actually $O(n/m)$
    - m size of the table, n number of elements to be hashed

Hashable Objects

```
public class HashableObject implements Hashable {
   private String key;
   // Other aspects of the object
   public HashableObject(String s) {
      key = new String(s);
   }
   public String key() {
      return key;
   }
}
```

3 Hash Functions, each used for different scenarios

- *Hash Function 1*
- Fast, but not good for large table sizes

```
public class HF1 extends HF {
   public int hash(String s, int size) {
      int hashVal = 0;
      for (int i = 0; i < s.length; i++) {
         hashVal += s.charAt(i);
         // Returns int code, e.g. A 65, B 66, etc
      }
      return hashVal % size;
   }
}
```

- *Hash Function 2*
- Fast, generally better than HF1, but horrible distributions of values where the first 3 chars are frequent

```
public class HF2 extends HF {
   public int hash(String s, int size) {
      int hashVal = 0;
      for (int i = 0; i < s.length; i++) {
         if (i == 0) { hashVal += s.charAt(0);}
         else if (i == 1) {hashVal += 27 * s.charAt(1);}
         else if (i == 2) {hashVal += 729 & s.charAt(2);}
      }
      return hashVal % size;
   }
}
```

- *Hash Function 3*
- Fast and has good distribution

```
public class HF3 extends HF {
   public int hash(String s, int size) {
      int hashVal = 0;
      for (int i = 0; i < s.length; i++) {
         hashVal += 37 * hashVal + s.charAt(i);
      }
   hashVal % size;
   if (hashVal < 0) {hashVal += size;}
   return hashVal;
   }
}
```

Hash Table implementation with Chaining

- The interface includes clear(), add(), remove(), and contain()

```
public class HashTableSC<T extends Hashable> implements HashTableInterface<T> {
   private LinkedList<T>[] hashTable;
   private HashFunction f;

   public HashTableSC(HashFunction f, int m) {
      hashTable = new LinkedList[m];
      this.f = f;
   }
   // Clear
   public void clear() {
      for (int i = 0; i < hashTable.length; i++) {
         hashTable[i] = null;
      }
   }
   // Insertion
   public void add(T item) {
      int i = f.hash(item.key(), hashTable.length);
      if (hashTable[i] == null) {
         hashTable[i] = new LinkedList<T>();
      }
      hashTable[i].add(item);
   }
```

```
    // Deletion
    public void remove(T item) {
        hashTable[f.hash(item.key(), hashTable.length)].remove(item.key());
        // Code before .remove will give you the Chain
    }
    // Search
    public boolean contains(T item) {
        int i = f.hash(item.key(), hashTable.length);
        if (hashTable[i] == null) {return false; }
        else {return hashTable[i].contains(item); }
    }
}
```

- **Correctness**: follows the correctness of regular LL operations
- **Time Complexity** (with Chaining):
  - *Load factor* λ of a hash table is the average length of chains

$$(n_0 + n_1 + ... + n_{m-1})/m = n/m = \lambda$$

  - Search analysis: $O(1+\lambda)$, but can be proven $\theta(1+\lambda)$
  - Insertion analysis: **Average**: $\theta(1)$, **W.C**: $O(n)$
  - Deletion analysis: **Average**: $O(1+\lambda)$, **W.C**: $O(n)$

3.4.3 *Open Addressing*

Definition. Open Addressing

- Method for handling collisions
- All elements are stored in the hash table
- When *collision* occurs, probe the table until a free (null) entry is found

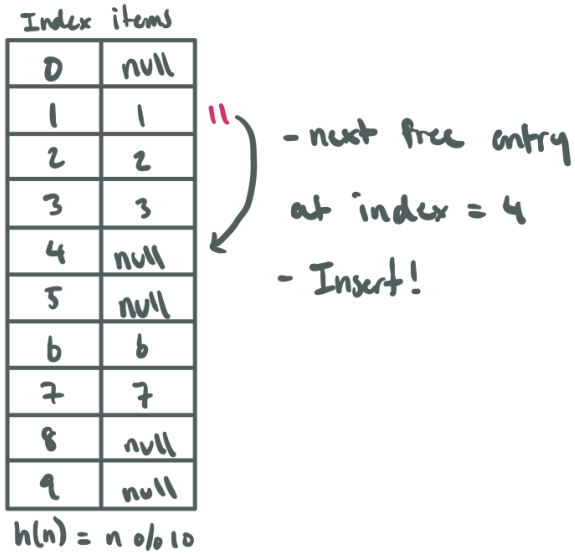

Figure 5: Insert 11 in hash table (via hash function)

- **Linear Probing**
  - Prior to inserting 11 at index 4, we had to go through previous indicies to check if they were free
  - That is $\rightarrow h(k)+0, h(k)+1, h(k)+2, h(k)+3$
  - Can be represented by: $< h(k), 0 >, < h(k), 1 >, < h(k), 2 >, ...$
  - This is the **probing sequence**, which gives us a new hashing function

$$h'(k) = h(k) + (i \mod 10)$$

  - This is called **Linear Probing**
    - Variations: Quadratic, double-hashing, etc.

Algorithms for Linear Probing Example of Insertion

Algorithm 3: Inserting a key *k*

```
1 i = 0;
2 while i < m do
3 |   j = h(k, i);
4 |   if (T[j] == null or T[j] == Deleted) then
5 |   |   T[j] = k;
6 |   |   return j;
7 |   i = i + 1;
8 end
9 Throw a TableFullException;
```

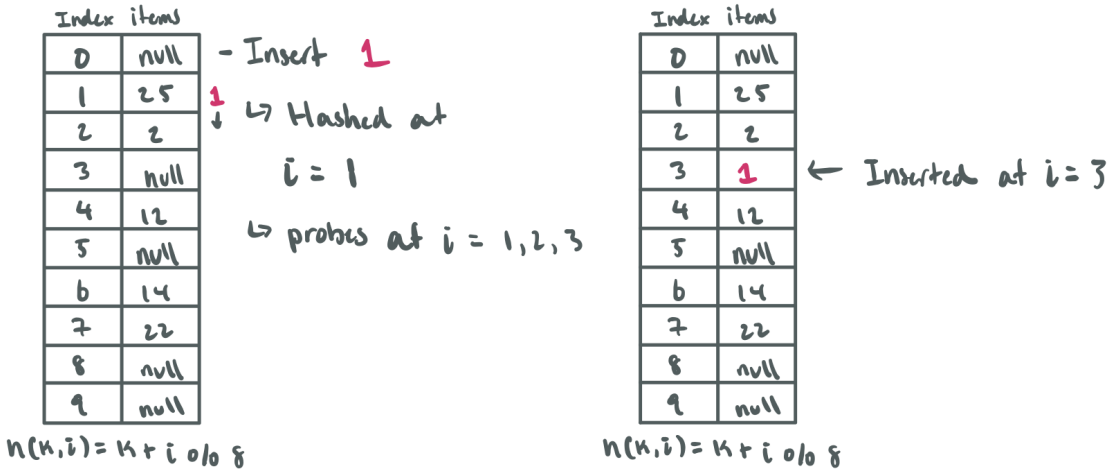

Figure 6: Insert 1

Algorithm 4: Searching a key *k*

```
1  i = 0;
2  while i < m do
3  |   j = h(k, i);
4  |   if (T[j] == k) then
5  |   |   return j;
6  |   else if T[j] == null then
7  |   |   Throw a NoSuchElementException;
8  |   i = i + 1;
9  end
```

Example of Searching

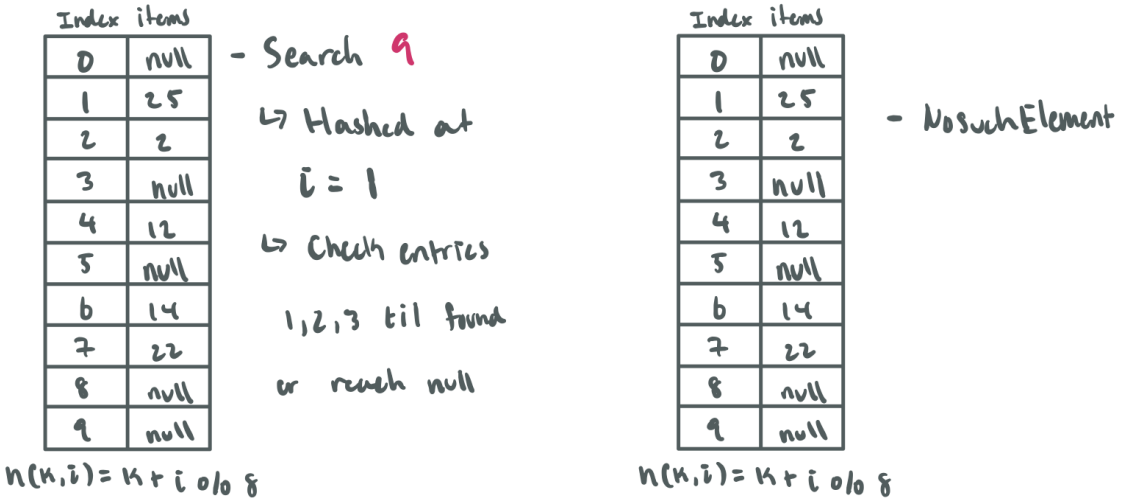

Figure 7: Search 9

Algorithm 5: Deleting a key *k*

```
1  i = 0;
2  while i < m do
3  |   j = h(k, i);
4  |   if (T[j] == k) then
5  |   |   T[j] == deleted;
6  |   |   return;
7  |   else if (T[j] == null) then
8  |   |   Throw a NoSuchElementException;
9  |   i = i + 1;
10 end
11 Throw a NoSuchElementException;
```

Example of Deletion

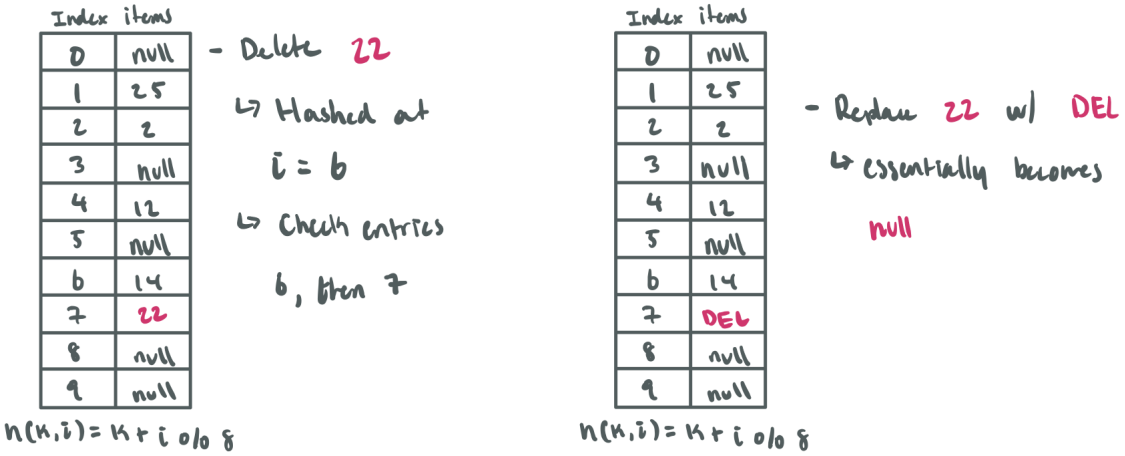

Figure 8: Delete 22

3.5 Graphs

3.6 Dijkstra's Algorithm

4 SORTING ALGORITHMS

4.1 Bubble Sort

4.1.1 *Bubble Sort Algorithm*

```
// Bubble Sort in Sorting
public void sort() {
   for (i = 0; i < size; i++) {
      for (j = size - 1; j > i; j--) {
         if (list[j].compareTo(list[j-1]) < 0) {
            T tmp = list[j];
            list[j] = list[j-1];
            list[j-1] = tmp;
         }
      }
   }
}
// Bubble Sort in BoundedList
public static void sort(BoundedList list) {
   for (i = 0; i < list.size(); i++) {
      for (j = list.size() - 1; j > i; j--) {
         if (list.getItem(j).compareTo(list.getItem(j-1)) < 0) {
            list.swap(j, j-1);
         }
      }
   }
}
// Main algorithm
Bubble-Sort(L = list of items, n = size) {
   for (i = 0; i < n; i++) {
      for (j = n - 1; j > i; j--) {
         if (L[j].key < L[j-1].key) {
            swap(L[j], L[j-1]);
         }
      }
   }
}
// Where swap() is...
swap(item n1, item n2) {
   tmp = n1;
   n1 = n2;
   n2 = tmp;
}
```

Example:

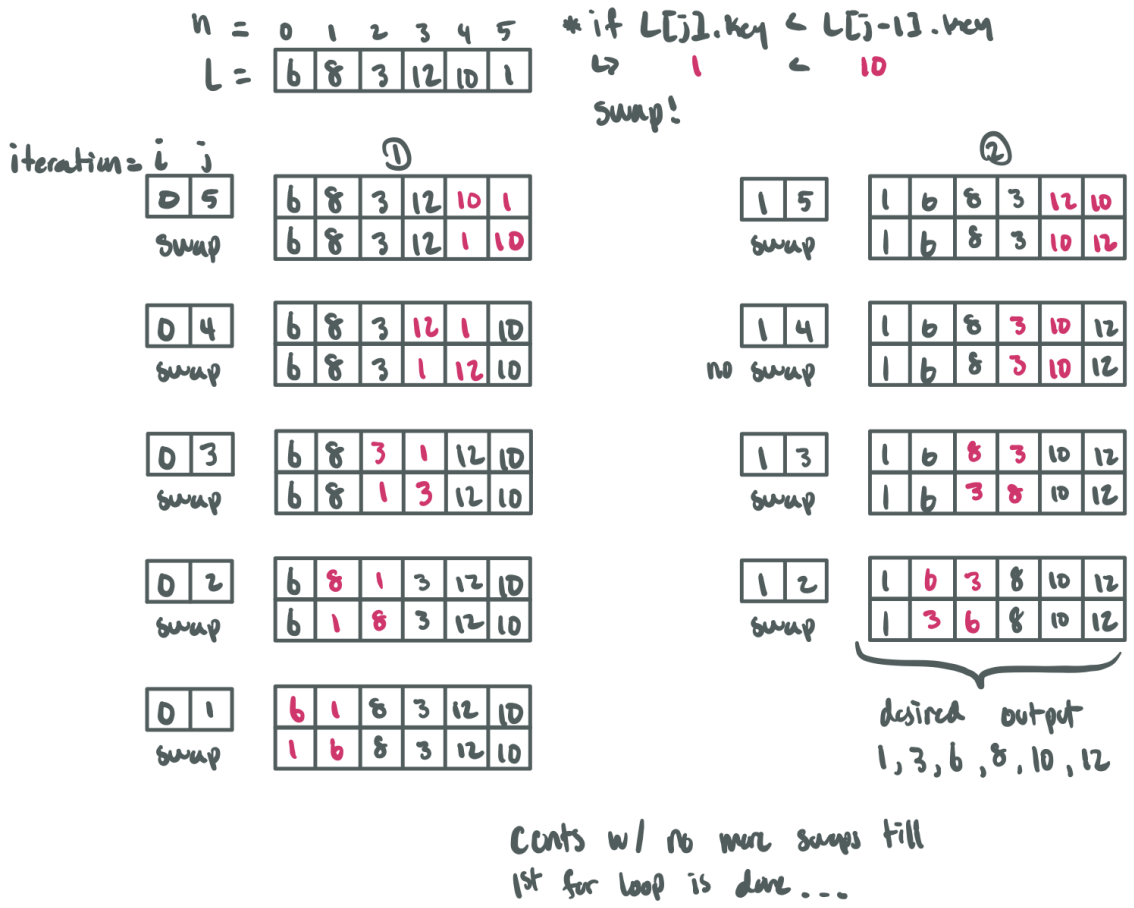

Figure 9: Bubble sort following the main algorithm

4.1.2 *Time Complexity*

Definition. *Time Complexity*

- **swap()**: $\theta(1)$
- **j-loop**: executed $n-i-1$ times
- **i-loop**: executed n times
- **Bubble-Sort**: $\Sigma_{i=0}^{n-1} n-i-1$

$$\Sigma_{i=0}^{n-1} n-i-1 = (n-1)+(n-2)+\ldots+1+0 = \frac{n^2-n}{2}$$

- **Hence**, Bubble-Sort is quadratic, $\theta(n^2)$

4.1.3 *Correctness*

Definition. *Loop Invariant*

- **Inner Loop**
  - *Initialization*:
    - Initially, $j = n - 1$ and $L[n-1...n-1]$ is a singleton
      - LI is vacuously satsified
  - *Maintenance*:
    - Assume LI is correct before iteration $j = k$
    - That is, $L[k]$ is the min in $L[k...n-1]$
    - In iteration $j = k$
      - If $L[k].key < L[k-1].key$, then swap($L[k], L[k-1]$), making $L[k-1]$ the new min in $L[k-1...n-1]$
      - Else, item being considered $L[k-1]$ is not bigger than $L[k]$, meaning that $L[k-1]$ is the min in $L[k-1...n-1]$
      - Hence, before the next iteration $j = k-1$, $L[k-1]$ is the min in $L[k-1...n-1]$
  - *Termination*:
    - Last iteration of the loop is when $j = i - 1$
    - Hence, by LI, $L[i]$ is the min item in $L[i...n-1]$
- **Outer Loop**
  - *Initialization*:
    - Before the first iteration, $i = 0$ and $L[0...i-1]$ is empty and LI is vacuously true
  - *Maintenance*:
    - Assume LI is correct before iteration $i = k$
    - Hence, $L[0...k-1]$ is a sorted list of the smallest *k* items in L
    - In iteration $i = k$, by the LI of the **inner loop**:
      - $L[k]$ is the min item in $L[k...n-1]$
    - Since $L[0...k-1]$ has the smallest *k* items of L in sorted order,
    - $L[0...k]$ will have the smallest $k+1$ items in L in sorted order
  - *Termination*:
    - Last iteration of the loop is when $i = n - 1$
    - Hence, when the loop terminates, $L[0...n-2]$ is a sorted permutation of the smallest $n-1$ elements in L
    - Since $L[n-1]$ was considered the last iteration and compared with $L[n-2]$, it must be the case that $L[n-1]$ is the largest item in L
    - Therefore, when the loop terminates, $L[0...n-1]$ is a sorted permutation of the original elements in L

Figure 10: Selection sort following main algorithm

4.2 Selection Sort

```
// Selection sort in Sorting
public static void selectionSort(BoundedList list) {
   for (int i = 0; i < list.size(); i++) {
      int minIndex = list.getMinIndex(i, list.size()-1);
      list.swap(i, minIndex);
   }
}
// Main algorithm
Selection-Sort(L = list of items, n = size) {
   for (i = 0; i < n; i++) {
      min = Find-Min(L[i...n-1]);
      swap(L[i], min);
   }
}
// Find minimum
Find-Min(L = list of items, n = size) { // Must be non-empty
   currMin = L[0];
   for (i = 1; i < n; i++) {
      if (L[i] < currMin) {
         currMin = L[i];
      }
   }
   return currMin;
}
```

Example:

Definition. *Time Complexity*

- **loop**: executed $n-1$ times
- **Find-Min()**: $\theta(n)$ on all cases
  - List is reduced by 1 after every iteration
  - 1st: $|L| = n$
  - 2nd: $|L| = n-1$
  - ith: $|L| = n-i-1$
- **Selection-Sort**: $\Sigma_{i=1}^{n-1} n-i-1$

$$\Sigma_{i=1}^{n-1} n-i-1 = (n-2)+\ldots+1 = \frac{n^2-2n+2}{2}$$

- **Hence**, Selection-Sort is quadratic, $\theta(n^2)$

4.3 Insertion Sort

```
// Insertion Sort in Sorting
public static void sort(BoundedList list) {
   for (i = 1; i < list.size(); i++) {
      for (j = i; j > 0; j--) {
         if (list.getItem(j).compareTo(list.getItem(j-1)) < 0) {
            list.swap(j, j-1);
         }
      }
   }
}
// Main algorithm
Insertion-Sort(L = list of items, n = size) {
   for (i = 1; i < n; i++) {
      for (j = i; j > 0; j--) {
         if (L[j].key < L[j-1].key) {
            swap(L[j], L[j-1]);
         }
      }
   }
}
// Where swap() is...
swap(item n1, item n2) {
   tmp = n1;
   n1 = n2;
   n2 = tmp;
}
```

Example:

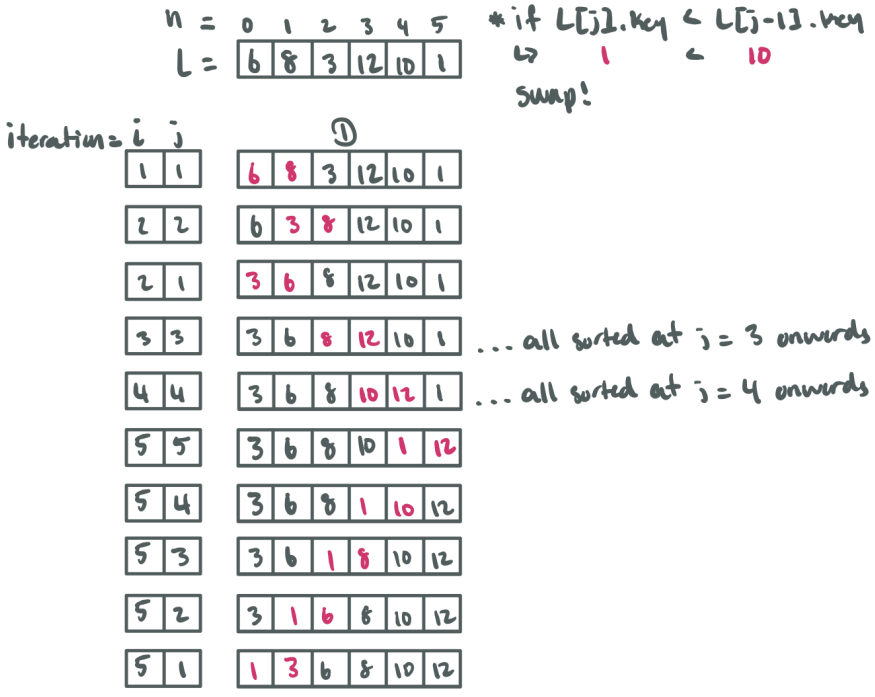

Figure 11: Insertion sort following the main algorithm

Definition. *Time Complexity*

- **swap()**: $\theta(1)$
- **j-loop**: executed i times
- **i-loop**: executed $n-1$ times, $\theta(n)$
- **Insertion-Sort**: $\Sigma_{i=1}^{n-1} n-i-1$

$$\Sigma_{i=1}^{n-1} n-i-1 = (n-1)+(n-2)+...+1+0 = \frac{n^2-n}{2}$$

- **Hence**, Insertion-Sort is quadratic, $\theta(n^2)$

4.3.1 *Correctness*

Definition. *Loop Invariant*

- **Inner Loop**
  - ***Initialization***:
    - Initially, $j = i = 1$, $L[0...0]$ is a sorted permutation of $S[0]$
      - LI is vacuously satsified
  - ***Maintenance***:
    - Assume LI is correct before iteration $j = k$
    - Hence, $L[k-1...i-1]$ is a sorted permutation of $S[k-1...i-1]$
    - We show LI still holds at the end of this iteration before $j = k-1$
      - Loop compares $L[k-1], L[k-2]$
      - If $L[k-2].key < L[k-1].key$, swap($L[k-2].key, L[k-1].key$)
      - Now, $L[k-2...i-1]$ is also a sorted permutation of $S[k-2...i-1]$
      - Else, $L[k-2].key$ is larger than $L[k-1].key$, loop does nothing and $L[k-2...i-1]$ is still a sorted permutation of $S[k-2...i-1]$
    - In both cases, the loop makes sure $L[k-2...i-1]$ has the same items as $S[k-2...i-1]$
    - Hence, LI still holds at the end of the iteration
  - ***Termination***:
    - Last iteration of the loop is when $j = 1$
    - Always terminates because j will eventually reach o, which is the condition for the for-loop to be false
    - Hence, at the end of it, $L[0...i-1]$ is a sorted permutation of $S[0...i-1]$ where $i \in [1, n]$
- **Outer Loop**
  - ***Initialization***:
    - Before the first iteration, $i = 1$ and $L[1...n-1]$ is empty and LI is vacuously true
  - ***Maintenance***:
    - Assume LI is correct before iteration $i = k$
    - Hence, by the inner LI, $L[0...n-2]$ is a sorted permutation of $S[0...n-2]$ after the last iteration of the outer loop
    - Suffices to note that when $i = n-1$ (last iteration of the outer loop)
    - If $L[n-2].key < L[n-1].key$, swap
    - Else, don't do anything

    - Hence, $L[0...n-1]$ is a sorting of $S[0...n-1]$
- ***Termination***:
    - Last iteration of the loop is when $i = n-1$
    - Starts with $i = 1$
    - i doesn't change anywhere other than incrementation
    - i will eventually become n
    - Hence, Insert-Sort() always terminates

4.4 Heap Sort

5 ADVANCED SORTING ALGORITHMS

5.1 Merge Sort

5.2 Quick Sort

6 SEARCHING

6.1 Linear Search

6.2 Binary Search

7 GRAPH TRAVERSAL

7.1 Depth-First Search

7.2 Breadth-First Search