CSC263

Data Structures and Analysis

Sinan Li

2023

CONTENTS

Part I

Data Structure

CHAPTER

PRIORITY QUEUES AND HEAPS 1

Data

- Collection of elements
- Each element x has a priority $x.priority$

Operations

- INSERT(Q, x)

  Add x to Q

  Note: $x.priority$ can be non-unique
- MAX(Q)

  Return the element with max priority

  Note: Q is unchanged
- EXTRACT-MAX(Q)

  Remove and return the element with the max priority

1.1 Implementation

1.1.1 Attempts

Implementation 1: Unsorted Array / Linked List

- INSERT takes $\Theta(1)$ time in the worst case
- MAX takes $\Theta(n)$ time in the worst case
- EXTRACTMAX takes $\Theta(n)$ time in the worst case

Implementation 2: Sorted Array / Linked List

- INSERT takes $\Theta(n)$ time in the worst case
- MAX takes $\Theta(1)$ time in the worst case
- EXTRACTMAX takes $\Theta(1)$ time in the worst case

1.1.2 Implementation

We want to combine the advantages of both data structures by having a "partially sorted" ADT – a (binary) *heap*.

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a **heap property**, the specifics of which depend on the kind of heap.

- Max heap property: the key of every node x is *larger* than or equal to the keys of its children.

  The largest element in a max-heap is stored at the root.

- Min heap property: the key of every node x is *smaller* than or equal to the keys of its children

  The smallest element in a min-heap is stored at the root,

These are called *heap orders*. There is no ordering between the siblings. A max/min heap is valid if it is a nearly complete binary tree and it satisfies the max/min heap property.

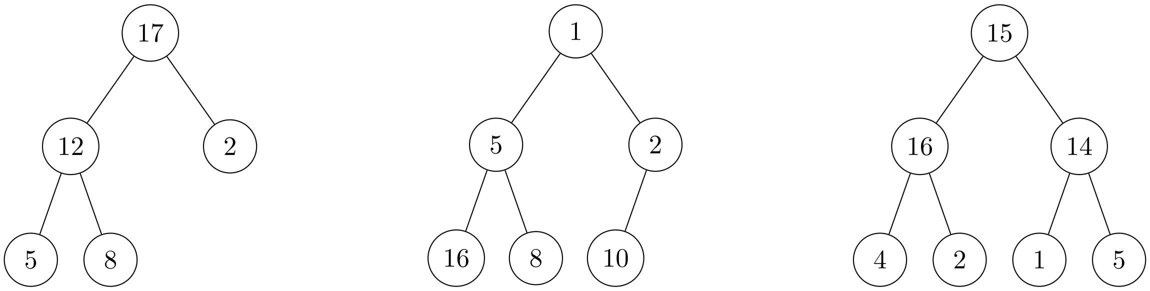


Figure 1.1: A valid max-heap (left), a valid min-heap (middle), and an invalid heap (right)

Although a heap is an **almost complete binary tree** [1], in practice, we usually use an array to store the data in memory. An array H that represents a heap is an object with two attributes: $H.length$, which (as usual) gives the number of elements in the array, and $H.heap\text{-}size$, which represents how many elements in the heap are stored within array H. The root of the tree is $H[1]$, and given the index i of a node, we can compute the indices of its parent, left child, and right child:

- PARENT

  return $\lfloor i/2 \rfloor$

- LEFT

  return $2i$

- RIGHT

  return $2i + 1$

[1] That is, the tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

1.2 Operations

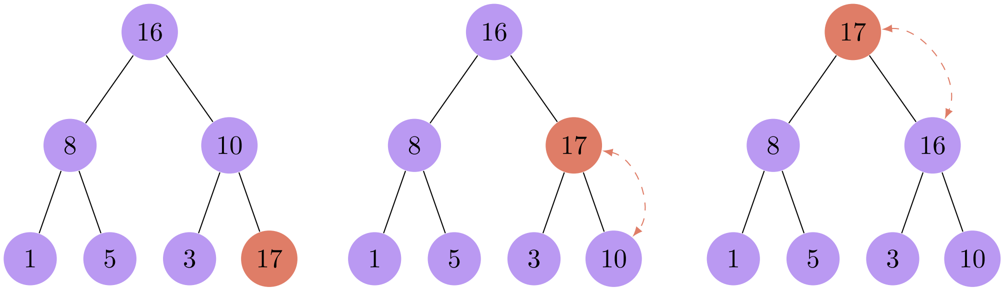
1.2.1 Insert

To insert element with key p into the heap H,

- Increment $H.\mathit{heap\text{-}size}$ and add a new node with key p to the next available position
- Repeatedly swap the new item with its parent until the heap property is satisfied

  This swapping process is called **bubbling up**
- Worst-case runtime: $\Theta(\lg n)$

For example, consider Insert$(H, 17)$ where $H = [16, 8, 10, 1, 5, 3]$

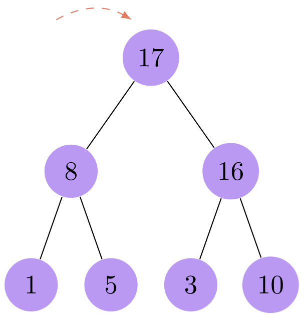


```
procedure Max-Heap-Insert($H$, $p$)
    $i \leftarrow H.\mathit{heap\text{-}size} \leftarrow H.\mathit{heap\text{-}size} + 1$
    $H[i] = p$
    while Parent$(i) > 0$ and $H[i] > H[\text{Parent}(i)]$ do
        swap $H[i]$ with $H[\text{Parent}[i]]$
        $i \leftarrow \text{Parent}(i)$
    end while
end procedure
```

1.2.2 Find-Max

To find the maximum key in the heap H,

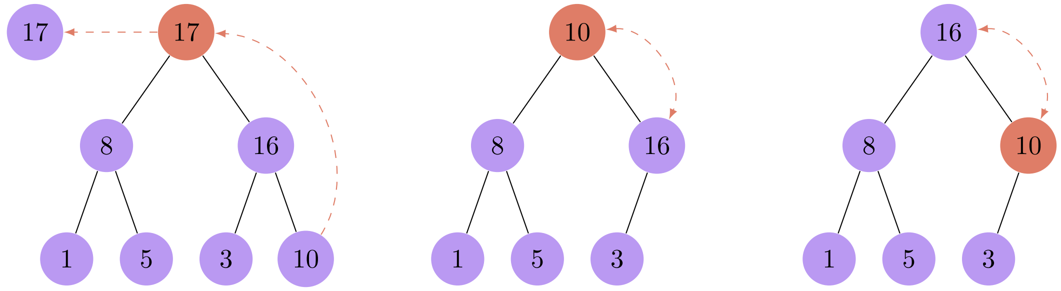
- Simply return the item in the root
- Worst-case runtime: $\Theta(1)$



```
procedure FIND-MAX($H$)
    return $H[1]$
end procedure
```

1.2.3 Extract-Max

- Save the item from the root in a temporary variable
- Replace the root with the rightmost item in the lowest level of the tree and decrement $H.heap\text{-}size$
- Repeatedly swap the item we moved with its largest child until the heap property is restored. This swapping process is called **bubble down**.
- Worst-case runtime: $\Theta(\lg n)$



```
procedure EXTRACT-MAX($H$)
    $max \leftarrow H[1]$
    $H[1] \leftarrow H[H.heap\text{-}size]$
    $H.heap\text{-}size \leftarrow H.heap\text{-}size - 1$
    MAX-HEAPIFY($H, 1$)
    return $max$
end procedure
```

```
procedure MAX-HEAPIFY($H$, $i$)
    $l \leftarrow$ LEFT($i$)
    $r \leftarrow$ RIGHT($i$)
    if $l \leq H.heap\text{-}size$ and $H[l] > H[i]$ then
        $largest \leftarrow l$
    else
        $largest \leftarrow i$
    end if
    if $r \leq H.heap\text{-}size$ and $H[r] > H[largest]$ then
        $largest \leftarrow r$
    end if
    if $largest \neq i$ then
        swap $H[i]$ with $H[largest]$
        MAX-HEAPIFY($H, largest$)
    end if
end procedure
```

1.2.4 Build-Max-Heap

- Takes an array A of length n and builds a max-heap H from it.
- Worst-case runtime: $\Theta(n)$

```
procedure BUILD-MAX-HEAP($A$)
    $H.heap\text{-}size \leftarrow A.length$
    for $i = \lfloor \frac{A.length}{2} \rfloor$ downto 1 do
        MAX-HEAPIFY($H, i$)
    end for
end procedure
```

CHAPTER 2

DICTIONARIES

Data

- A set S
- Each element x has a **unique** key $x.key$

Operations

- Search(S, k)

  Return x in S with $x.key = k$ (or NIL).

- Insert(S, x)

  Add x to S – if S contains y with $y.key = x.key$, then *replace* y with x.

- Delete(S, x) [a]

  Remove element x from S.

[a] If we are given the key k instead of the element x, we could do Delete$(S,$ Search(S, k))

	Search	Insert	Delete$^{\#}$
unsorted array	n	n	1
sorted* array	$\lg n$	n	n
unsorted linked list	n	n	1
sorted* linked list	n	n	1
direct access table	1	1	1
hash table	n	n	n
binary search tree	height of the BST	height of the BST	height of the BST
balanced search tree	$\lg n$	$\lg n$	$\lg n$

*: these require the keys to be ordered
$^{\#}$: here we are only doing deletion (without having to search for x)

2.1 Binary Search Trees

A binary search tree is a binary tree with the *binary-search-tree property*:

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

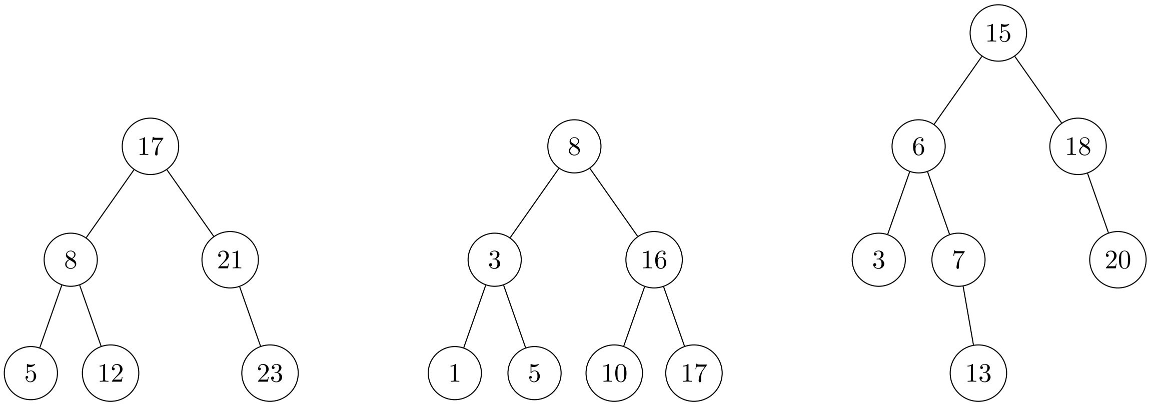


Figure 2.1: Examples of binary search trees

```
class Item:
    key: Any
    value: Any
```

```
class BST_Node:
    item: Item
    left: BST_Node
    right: BST_Node
```

```
class Dictionary:
    root: BST_Node
```

2.1.1 Insert

```
procedure INSERT($S$, $x$)
    $S.root \leftarrow$ BST-INSERT($S.root, x$)
end procedure
```

```
procedure BST-INSERT($root$, $x$)
    # Insert $x$ into stubree at root; return new root
    if $root =$ NIL then
        $root \leftarrow$ BST_NODE($x$) # Add $x$
    else if $x.key < root.item.key$ then
        $root.left \leftarrow$ BST-INSERT($root.left, x$)
    else if $x.key > root.item.key$ then
        $root.right \leftarrow$ BST-INSERT($root.right, x$)
    else # $x.key = root.item.key$
        $root.item \leftarrow x$ # replace with $x$
    end if
    return $root$
end procedure
```

2.1.2 Search

```
procedure Search($S$, $k$)
    $node \leftarrow$ BST-Search($S.root, k$)
    if $node =$ NIL then
        return NIL
    end if
    return $node.item$
end procedure
```

```
procedure BST-Search($root$, $k$)
    # Return node under root with key $k$ (or NIL)
    if $root =$ NIL then
        pass # $k$ not in tree
    else if $k < root.item.key$ then
        $root \leftarrow$ BST-Search($root.left, k$)
    else if $k > root.item.key$ then
        $root \leftarrow$ BST-Search($root.right, k$)
    else  # $k = root.item.key$
        pass
    end if
    return $root$
end procedure
```

2.1.3 Delete

```
procedure Delete($S$, $k$)
    $S.root \leftarrow$ BST-Delete($S.root, k$)
end procedure
```

```
procedure BST-Delete($root$, $x$)
    # Delete $x$ from stubree at root; return new root
    if $root =$ NIL then pass # $x$ not in tree
    else if $x < root.item.key$ then
        $root.left \leftarrow$ BST-Delete($root.left, x$)
    else if $x > root.item.key$ then
        $root.right \leftarrow$ BST-Delete($root.right, x$)
    else  # $x.key = root.item.key$
        if $root.left =$ NIL then
            $root \leftarrow root.right$ # could be NIL
        else if $root.right =$ NIL then
            $root \leftarrow root.left$
        else  # Replace $root.item$ with its successor
            $root.item, root.right \leftarrow$ BST-Del-Min($root.right$)
        end if
    end if
    return $root$
end procedure
```

```
procedure BST-Del-Min($root$)
    # Remove element with smallest key under root; return item and root of resulting subtree
Require: $root \neq$ NIL
    if $root.left =$ NIL then
        return $root.item, root.right$
    else
        $item, root.left \leftarrow$ BST-Del-Min($root.left$)
        return $item, root$
    end if
end procedure
```

2.2 Balanced Search trees

Despite the simplicity of the BST, it is not a very efficient data structure. The worst-case running time of the BST operations is proportional to the height of the tree, which is $\Theta(n)$, where n is the number of elements in the tree. The shape of a BST is determined by the order in which keys are inserted. If the keys are inserted in sorted order, the BST degenerates into a linked list.

We can improve the performance of the BST by making it more balanced. A *balanced BST* (also known as an *AVL tree* – Adelson-Velsky, Landis Tree) is one in which the heights of the two subtrees of any node differ by at most one. The height of a balanced BST is $\Theta(\log n)$, where n is the number of elements in the tree.

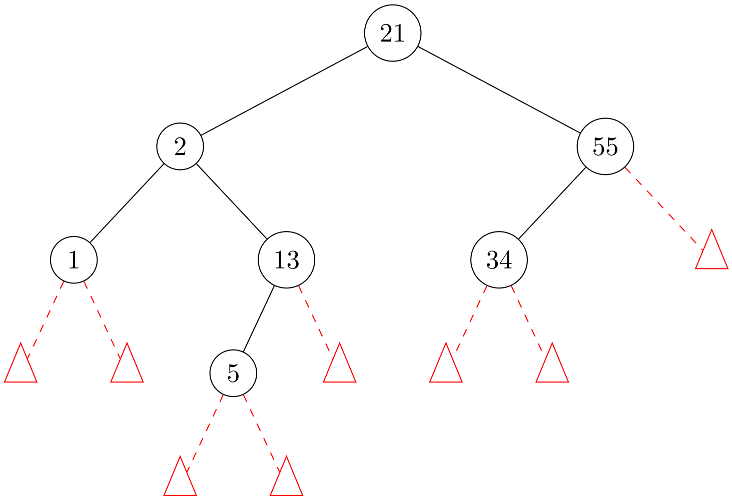


Figure 2.2: AVL balanced binary search tree

To implement an AVL tree, we need a mechanism to detect imbalance in the tree, and a way to restore balance. We will use the following definition of *balance factor* of a node x in a BST:

Definition 2.2.1 Balance Factor

An AVL balanced node x has a balance factor of -1, 0, or 1. If the height of its left subtree is h_L, and the height of its right subtree is h_R, then x has a balance factor of $h_L - h_R$.

- If $h_R - h_L = 0$, then x is balanced.
- If $h_R - h_L = 1$, then x is right-heavy.
- If $h_R - h_L = -1$, then x is left-heavy.

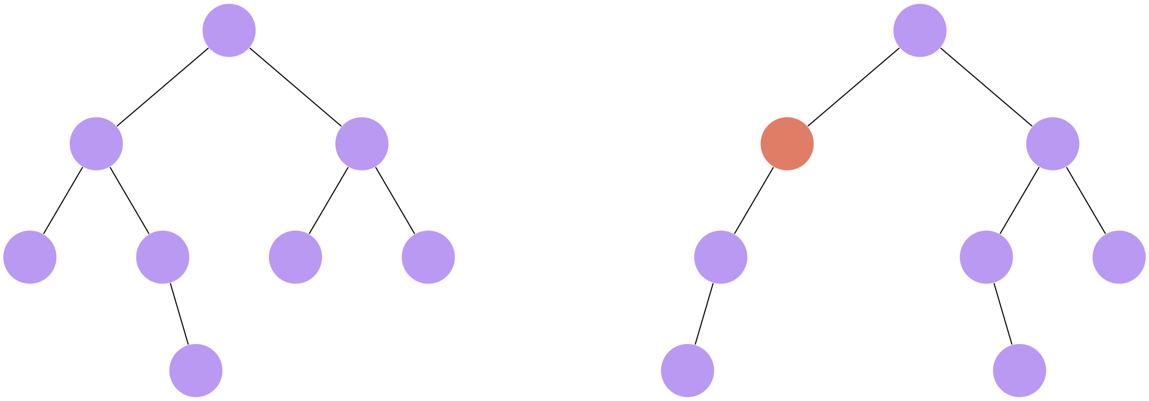


Figure 2.3: AVL balanced tree (left) and unbalanced tree (right)

Rotations

To restore balance, we need to perform a *rotation* on the tree. There are four types of rotations, depending on the balance factor of the node and its children. The following figure shows the four types of rotations.

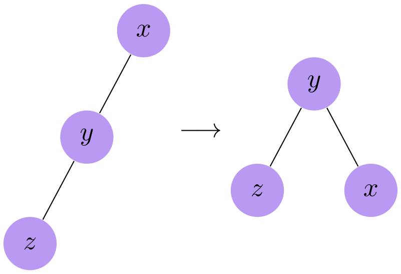


Figure 2.4: Single Left Rotation

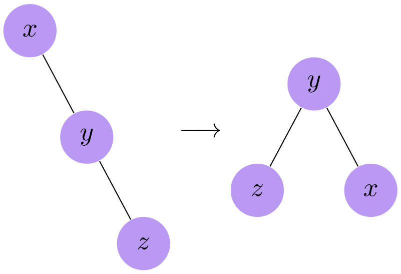


Figure 2.5: Single Left Rotation

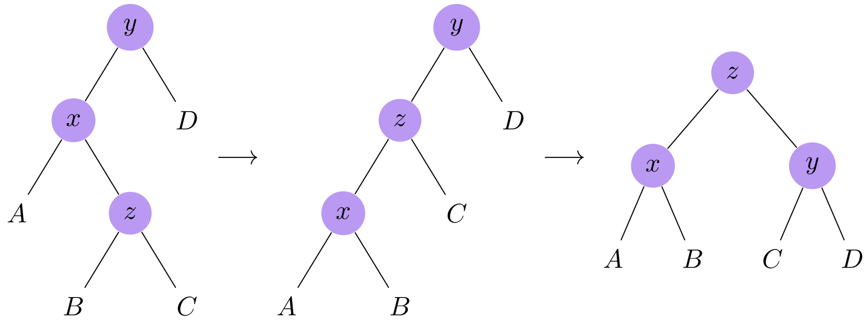


Figure 2.6: Double Left-Right Rotation

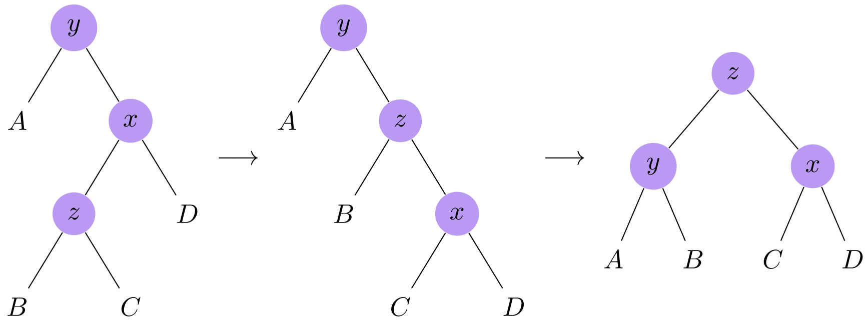


Figure 2.7: Double Right-Left Rotation

2.2.1 Insert

```
procedure AVL-INSERT($root$, $x$)
    # Insert $x$ into the tree at $root$, return new root
    if $root = \text{NIL}$ then
        $root \leftarrow$ AVL_NODE($x$) # add $x$
    else if $x.key < root.item.key$ then
        $root.left \leftarrow$ AVL-INSERT($root.left, x$)
        $root \leftarrow$ AVL-BALANCE-RIGHT($root$)
    else if $x.key > root.item.key$ then
        $root.right \leftarrow$ AVL-INSERT($root.right, x$)
        $root \leftarrow$ AVL-BALANCE-LEFT($root$)
    else# $x.key = root.item.key$
        $root.item \leftarrow x$ # replace with $x$
    end if
    return $root$
end procedure
```

2.2.2 Delete

```
procedure AVL-Delete($root$, $x$)
    # Delete $x$ from the tree at $root$, return new root
    if $root = \text{NIL}$ then
        pass # $x$ not in tree
    else if $x.key < root.item.key$ then
        $root.left \leftarrow$ AVL-Delete($root.left, x$)
        $root \leftarrow$ AVL-Balance-Left($root$)
    else if $x.key > root.item.key$ then
        $root.right \leftarrow$ AVL-Delete($root.right, x$)
        $root \leftarrow$ AVL-Balance-Right($root$)
    else  # $x.key = root.item.key$
        if $root.left = \text{NIL}$ then
            $root \leftarrow root.right$ # could be NIL
        else if $root.right = \text{NIL}$ then
            $root \leftarrow root.left$
        else
            if $root.left.height > root.right.height$ then
                $root.item, root.left \leftarrow$ AVL-Delete-Max($root.left$)
            else
                $root.item, root.right \leftarrow$ AVL-Delete-Min($root.right$)
            end if
        end if
        $root.height \leftarrow 1 +$ Max($root.left.height, root.right.height$)
    end if
    return $root$
end procedure
```

```
procedure AVL-Del-Max($root$)
    # Delete the maximum item from the tree at $root$, return new root and deleted item
Require: $root \neq \text{NIL}$
    if $root.right = \text{NIL}$ then
        return $root.item, root.left$
    else
        $item, root.right \leftarrow$ AVL-Delete-Max($root.right$)
        $root \leftarrow$ AVL-Balance-Right($root$)
        return $item, root$
    end if
end procedure
```

2.2.3 Rebalancing

```
procedure AVL-Balance-Left($root$)
Require: $root \neq$ NIL
    # First, recalculate height
    $root.height \leftarrow 1 + \textsc{Max}(root.left.height, root.right.height)$
    # Then, rebalance the left, if necessary
    if $root.right.height > root.left.height + 1$ then
        # Check for double rotation
        if $root.right.left.height > root.right.right.height$ then
            $root.right \leftarrow$ AVL-Rotate-Right($root.right$)
        end if
        $root \leftarrow$ AVL-Rotate-left($root$)
    end if
    return $root$
end procedure
```

```
procedure AVL-Rotate-Left($parent$)
Require: $parent \neq$ NIL, $parent.right \neq$ NIL
    # Rearrange references
    $child \leftarrow parent.right$
    $parent.right \leftarrow child.left$
    $child.left \leftarrow parent$
    # Update heights; parent first because it is now deeper
    $parent.height \leftarrow 1 + \textsc{Max}(parent.left.height, parent.right.height)$
    $child.height \leftarrow 1 + \textsc{Max}(child.left.height, child.right.height)$
    # Return new parent
    return $child$
end procedure
```

2.3 Hashing

- Universe U

  The set of all keys. We assume that $|U|$ is very large.

- Hash Table T

  An array of fixed size m. Each location $T[i]$ is called a *bucket*.

- Hash Function h

  The hash function $h : U \to \{0, 1, \ldots, m-1\}$ maps each key in U to an index in $\{0, 1, \ldots, m-1\}$. For each key $k \in U$, $h(k)$ is called the *home bucket* of k.

  To access item with key k, examine $T[h(k)]$.

A hash table is an effective data structure for implementing dictionaries. Although SEARCH for an element in a hes table can take as long as searching for an element in a linked list – $\Theta(n)$ time in the worst case – in practice, hashing preforms extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $\mathcal{O}(1)$.

2.3.1 Direct Access Tables

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. If U is small, then we can use an array T of size $|U|$ to implement a dictionary, called a *direct access table*. The key k is used as an index into T to access the item with key k.

2.3.2 Hash Tables

The downside of direct addressing is apparent: if the universe U is large or infinite. storing a table T of size $|U|$ is impractical, and the set K of keys *actually stored* may be so small relative to Y that most of the space allocated for T would be wasted. Instead, we use a hash table.

However, when $m << |U|$, collisions are unavoidable. A *collision* occurs when two keys k_1 and k_2 (with $k_1 \neq k_2$) are mapped to the same bucket $h(k_1) = h(k_2)$. There are two ways to handle collisions: *open addressing* and *closed addressing / chaining*.

Open Addressing

In open addressing, if $T[h(k)]$ is occupied, then we search for the next available location in T to store the item with key k. We call the original hash function h_1 the *primary hash function*, such that $h_1(k)$ is the home bucket of k. We use the *probe sequence* $h(k, i)$ to determine the bucket to try after i collisions.

- *Linear Probing*

  $h(k, i) = (h_1(k) + i) \mod m$

  Note that long clusters of occupied buckets can occur.

- *Quadratic Probing*

  $h(k, i) = (h_1(k) + c_1 i + c_2 i^2) \mod m$

  c_1 and c_2 are constants dependent on m.

- *Double Hashing*

  $h(k, i) = (h_i(k) + i \cdot (h_2(k))) \mod m$, where $h_2(k)$ is a secondary hash function.

Close Addressing / Chaining

In close addressing, we use a linked list to store the items in each bucket. Each nonempty slot points to a linked list, and all the elements that hash to the same slot go into that slot's linked list.

The average-case performance of the hash table depends on how evenly the hash function h distributes the keys across the buckets in the table. The *simple uniform hashing assumption*(SUHA) states that any given key is equally likely to hash into any of the m slots of the table, independently of where any other elements has hashed to. Under this assumption, the expected number of keys in each bucket is the same.

The expected number of keys in a bucket is $\frac{n}{m}$, where n is the number of items in the table and m is the size of the table. This ratio is called the *load factor* of the hash table, and we denote it by α.

CHAPTER

3 DYNAMIC ARRAY

C styled arrays are static, meaning that they have a fixed size. In this chapter, we will learn how to implement a dynamic array, which is a data structure that can grow and shrink in size.

```
class DynamicArray {
    capacity: integer  # room for elements
    size:     integer  # actual number of elements
};
```

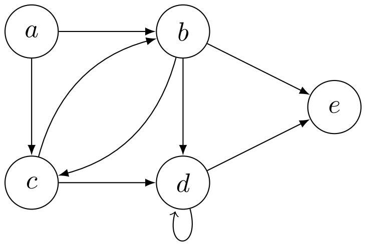
```
procedure INSERT($A, x$)
    if $A.size = A.capacity$ then
        $A \leftarrow$ RESIZE($A$)
    end if
    $A.size \leftarrow A.size + 1$
    $A[A.size] \leftarrow x$
end procedure
```

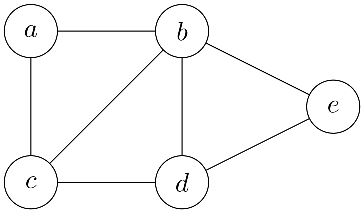
```
procedure RESIZE($A$)
    $B \leftarrow$ DYNAMICARRAY($2 \times A.capacity$)
    for $i = 1$ to $A.size$ do
        INSERT($B, A[i]$)
    end for
    return $B$
end procedure
```

To analyze the running time of the above algorithm, see this example using accounting method for amortized analysis. The amortized running time of the above algorithm is $\Theta(1)$.

CHAPTER 4

GRAPHS





4.1 Graphs

4.1.1 Graphs

Define a graph $G = \{V, E\}$

Representations

- Adjacency matrix

	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	1	1
c	1	1	0	1	0
d	0	1	1	0	1
e	0	1	0	1	0

Complexity: let $n = |V|$ and $m = |E|$

  - Space: $O(n^2)$
  - Edge query: $\Theta(1)$ time

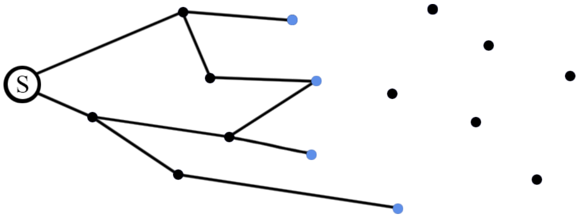
- Adjacency list

  Complexity: let $n = |V|$ and $m = |E|$

  - Space: $\Theta(n + m)$
  - Edge query: $\Theta(n)$ in worst-case time

4.1.2 Breadth-First Search

In breadth first search, we start at a source $s \in V$, and explore every vertex reachable from a, using only edges.



We assign each vertex $v \in V$ a color, which can be one of the following:

- White: v has not been discovered
- Gray: v has been discovered, but not explored
- Black: v has been discovered and explored

Define $\pi[v]$ to be the predecessor of v in the breadth-first search tree.
Define $d[v]$ to be the distance from s to v in the breadth-first search tree.
We use a queue to keep track of the vertices that we have discovered, but not yet explored.

```
procedure BFS($G$, $s$)
    # Initialize tracking info for all vertices
    for $v \in G.V$ do
        colour[$v$] $=$ white
        $\pi[v] \leftarrow$ NIL
        $d[v] \leftarrow \infty$
    end for
    # Initialize empty queue and source vertex tracking info
    $Q \leftarrow$ MAKE-QUEUE()
    colour[$s$] $\leftarrow$ gray
    $\pi[s] \leftarrow$ NIL
    $d[s] \leftarrow 0$
    ENQUEUE($Q$, $s$)
    # Main loop. Loop Invariant: $Q$ contains all (and only) gery vertices
    while $Q \neq$ EMPTY-QUEUE do
        $u \leftarrow$ DEQUEUE()
        for $v \in G.Adj[u]$ do
            if colour[$v$] $\leftarrow$ white then
                colour[$v$] $\leftarrow$ gray
                $\pi[v] \leftarrow u$
                $d[v] \leftarrow d[u] + 1$
                ENQUEUE($Q$, $v$)
            end if
        end for
        colour[$u$] $=$ black
    end while
end procedure
```

In breath first search,

- Each vertex is enqueued at most once
- Each vertex is dequeued at most once
- Each adjacent list is examined at most once
- The time complexity is $\Theta(n + m)$

BFS Finds Shortest Paths

Define $\delta(s, v)$ to be the length of the shortest path from vertex s to vertex v (i.e. the smallest number of edges in any path from s to v). If there is no path from s to v, then $\delta(s, v) = \infty$. Note that this definition will change when we consider **weighted** graphs.

Theorem 4.1.1 **Let $G = \{V, E\}$ be a graph, and let $s \in V$. Then, after BFS(G, s), $\forall v \in V$, $\delta(s, v) = d[v]$**

To prove this theorem, we will need to prove the following lemmas first.

Lemma 4.1.1 $\forall (u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$

Proof. (idea)

If $\delta(s, u) = \infty$, then the claim holds trivially.

If $\delta(s, u) \neq \infty$, then u is reachable from s.

Thus, v is also reachable from s.

Thus, the shortest path from s to v is no longer than the shortest path from s to u, plus the edge (u, v).

Hence, $\delta(s, v) \leq \delta(s, u) + 1$. ■

Lemma 4.1.2 **At ant point during BFS,** $\forall v \in V$, $d[v] \geq \delta(s, v)$

Proof. (idea)

Use induction on the number of Enqueue operations.

Immediately after we do the first Enqueue operation, $d[s] = 0$, and $\delta(s, s) = 0$.

We also have $d[v] = \infty$, and $\delta(s, v) = \infty$ for all $v \in V - \{s\}$.

Now, consider some vertex v that is first discovered while visiting a vertex u.

By the IH, we have $d[u] \geq \delta(s, u)$.

Hence, $d[v] = d[u] + 1 \geq \delta(s, u) + 1$ by Lemma 4.2.1.

Then v is painted grey and $d[v]$ is not changed for the rest of the algorithm. ■

Lemma 4.1.3 **If** $Q = \langle v_1, \ldots, v_r \rangle$, **then** $d[v_i] \leq d[v_{i+1}]$ **for all** $i \in \{1, \ldots, r-1\}$ **and** $d[v_r] \leq d[v_1] + 1$

Proof. (sketch)

Use induction on the number of Dequeue / Enqueue operations.

When $Q = \langle s \rangle$, the claim holds trivially.

To prove the inductive step, we need to show that the lemma hold after applying Dequeue / Enqueue to $Q = \langle v_1, \ldots, v_3 \rangle$.

- Case 1

  If we perform a Dequeue operation, then $Q = \langle v_2, \ldots, v_3 \rangle$ afterwards.

  By the IH, $d[v_r] \leq d[v_1] + 1$ and $d[v_1] \leq d[v_2]$.

  Hence, $d[v_r] \leq d[v_2] + 1$.

  All other inequalities are unaffected.

- Case 2

  If we perform a Enqueue operation, then $Q = \langle v_1, \dots, v_{r+1} \rangle$ afterwards.

  We discover v_{r+1} while visiting some vertex u, so $d[v_{r+1}] = d[u] + 1$.

  Vertex u must have been the previous vertex dequeued from the queue.

  Hence, either v_1 was discovered while visiting u, in which case $[v_1] = d[u] + 1$, or Q was equal to $\langle u_2, v_1, \dots \rangle$ at some prior point, in which case $d[u] \leq d[v_1]$ by IH.

  Hence, $d[v_{r+1}] = d[u] + 1 \leq d[v_1] + 1$.

  Otherwise, $d[v_r] \leq d[u] + 1 = d[v_{r+1}]$ by the IH.

■

Now, we can prove the theorem.

Proof. To derive a contradiction, suppose $d[v] \neq \delta(s, v)$ for some vertex $v \in V$.
Suppose v is a vertex with minimal $\delta(s, v)$ for which this is satisfied.
By Lemma 4.1.2, we have $d[v] > \delta(s, v)$.
Because we chose v with minimal $\delta(s, v)$, we have $d[u] = \delta(s, u)$.
Hence, $d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1$.
Consider the colour of v when we first dequeue u from Q.

- If v is painted **white**, then we set $d[v] = d[u] + 1$, which is a contradiction.
- If v is painted **black**, then v was in the queue before u. By Lemma 4.1.3, we have $d[v] \leq d[u]$, which is a contradiction.
- If v is painted **grey**, then v was discovered while visiting some vertex w that was dequeued earlier than u. Hence, $d[v] = d[w] + 1$, and by Lemma 4.1.3 we have $d[w] \leq d[u]$. So $d[v] \leq d[u] + 1$, which is a contradiction.

■

Time complexity of BFS

- Initialization (painting vertices white, setting entries of d to ∞ and entries of π to NIL) takes $\Theta(|V|)$ time.
- After initialization, we never paint a vertex white.
- Thus, each vertex is enqueued/dequeued at most once.
- Hence, we spend $\mathcal{O}(|V|)$ time doing queue operations.
- Every time we dequeue a vertex, we scan its out-neighbourhood to discover its neighbours:
  - With an **adjacency list**, we consider each edge at most once (or at most twice in an undirected graph), so in total we need at most $\mathcal{O}(|E|)$ time to consider all of the edges.
  - With an adjacency matrix, we scan each row of the matrix at most once, so in total we need at most $\mathcal{O}(|V|^2)$ time to consider all of the edges.

  Using adjacency list: $\mathcal{O}(|V| + |E|)$ Using adjacency matrix: $\mathcal{O}(|V|^2)$

4.1.3 Depth-First Search

- In DFS, we walk through the graph as far as possible until we hit a dead end – when this happens, we backtrack to an undiscovered vertex.
- Similar to BFS, we paint vertices as we go:
    - Painted white: undiscovered
    - Painted grey: discovered but not yet visited
    - Painted black: visited
- Instead of storing distances/depths, we store timestamps:
- - $disc[v]$ = time at which v is first discovered
    - $vis[v]$ = time at which we finish visiting v
- One approach is to simply replace the *queue* from the BFS algorithm with a *stack*. This gives us an **iterative** DFS algorithm.
- However, it is more natural to write DFS as a recursive algorithm.

```
procedure DFS(G)
    # Initialization
    for each $v \in G.V$ do
        $d[v] \leftarrow f[v] \leftarrow \infty$
        $\pi[v] \leftarrow$ NIL
    end for
    time ← 0 # global
    # Main loop
    for each $v \in G.V$ do
        if $d[v] = \infty$ then
            # colour[$v$] = white
            DFS-VISIT($G$, $v$)
        end if
    end for
end procedure
```

```
procedure DFS-VISIT($G$, $v$)
    # Discovered (colour[v] = grey)
    $d[v] \leftarrow$ time ← time + 1
    # Do something with $v$, if desired
    # Explore $v$'s adjacency list
    for each $u \in G.adj[v]$ do
        if $d[u] = \infty$ then
            # colour[$u$] = white
            $\pi[u] = v$
            DFS-VISIT($G$, $u$)
        end if
    end for
    # Finished (colour[$v$] = black)
    $f[v] \leftarrow$ time ← time + 1
end procedure
```

DFS Forests

We classify each edge (u, v) based on the colour of v when we consider this edge:

- *Tree edges* are the edges u, $v \in E$ that form the DFS forest stored by π

  The vertex v is painted **white** when (u, v) is considered
- *Back edges* are the edges $(u, v) \in E$ such that v is an ancestor of u in the DFS forest

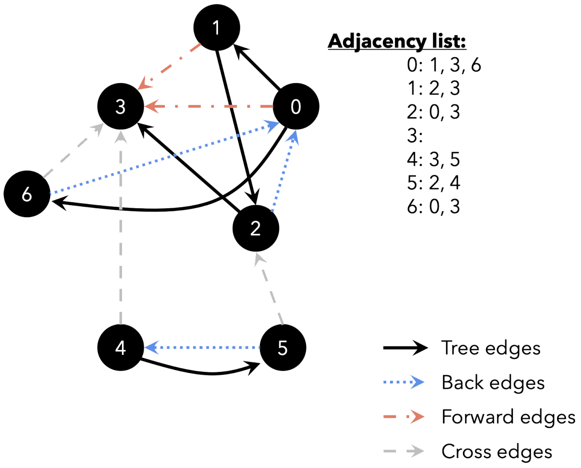
  The vertex v is painted **grey** when (u, v) is considered

- *Forward edges* are the edges $(u, v) \in E$ such that v is a descendant of u in the DFS forest

  The vertex v is painted **black** when (u, v) is considered

- *Cross edges* are all other edges $(u, v) \in E$ that are not part of the DFS forest (i.e. v is neither an ancestor nor a descendant of u)

  The vertex v is painted **black** when (u, v) is considered



Note that for undirected graphs, there are NO forward edges and NO cross edges.

Time Complexity

The run-time of DFS is similar to BFS.

Using adjacency list: $\mathcal{O}(|V| + |E|)$ Using adjacency matrix: $\mathcal{O}(|V|^2)$

Parenthesis theorem

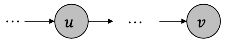
Theorem 4.1.2 **After performing DFS$(G = (V, E))$, for any two vertices u, $v \in V$, exactly one of the following statements holds:**

1. **The intervals $[\boldsymbol{disc}[u], \boldsymbol{vis}[u]]$ and $[\boldsymbol{disc}[v], \boldsymbol{vis}[v]]$ are disjoint, and neither u nor v is a descendant of the other in the DFS forest.**
2. **The interval $[\boldsymbol{disc}[u], \boldsymbol{vis}[u]]$ is contained entirely in the interval $[\boldsymbol{disc}[v], \boldsymbol{vis}[v]]$, and u is a descendant of v in the DFS forest.**
3. **The interval $[\boldsymbol{disc}[v], \boldsymbol{vis}[v]]$ is contained entirely in the interval $[\boldsymbol{disc}[u], \boldsymbol{vis}[u]]$, and v is a descendant of u in the DFS forest.**

Proof. (sketch)

Suppose that $disc[u] < disc[v]$.

- Case 1: $disc[v] < vis[u]$



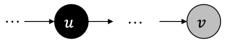
v is first discovered while u is painted grey.

So v is a descendant of u.

We don't backtrack to u until we have finished visiting v.

Therefore, we paint v black and set $vis[v]$ before backtracking to u. Hence, $vis[v] < vis[u]$.

- Case 2: $vis[u] < disc[v]$



v is first discovered while u is painted black.

So v is not a descendant of u.

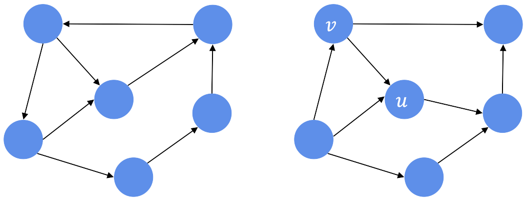
Since $disc[u] < dics[v]$, u is not a descendant of v.

Since $disc[u] < vis[u]$ and $disc[v] < vis[v]$, we have $disc[u] < vis[u] < disc[v] < vis[v]$.

When $disc[v] < disc[u]$, proof is symmetric. ■

4.1.4 Strongly connected

Recall that an undirected graph is connected if and only if there is a path of edges between any pair of vertices in V. What about directed graphs? A directed graph $G = (V, E)$ is strongly connected if and only if, for all u, $v \in V$, there is a path of edges from u to v in G.



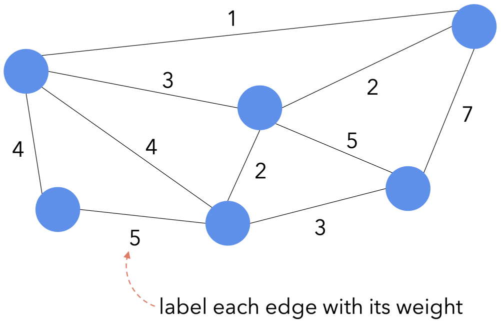
A strongly connected graph (left) and a non-strongly connected graph (right, where v is not reachable from u)

- Run DFS on the **transpose** of G (reverse edge direction of all edges in G)
- *Reorder* vertices in decreasing order of finish time (in $G.V$ and within each adjacency list)

  Run *DFS* on G using the new ordering of vertices
- Each DFS tree is one strongly connected component

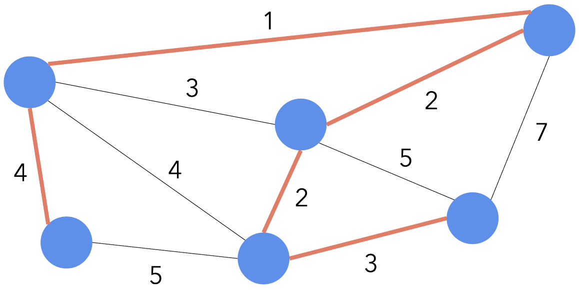
4.2 Minimum Spanning Trees

In a *weighted graph* $G = (V, E)$, each edge $e \in E$ has a weight $w(e)$



label each edge with its weight

A *minimum spanning tree* (MST) T of a weighted, undirected, connected graph $G = (v, E)$ is an acyclic subset of E that connects all of the vertices in V and whose total weight $\sum_{\{u,v\}\in T} w(\{u,v\})$ is minimized.

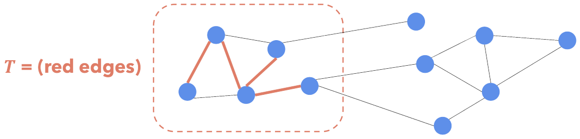


Above, we define an MST as a set of edges. We also often call the graph $M = (V, T)$ an MST of G.

The graph $M = (V, T)$ belongs to a special class of graphs called **trees**, which can be defined in many ways.

4.2.1 Prim's Algorithm

In Prim's Algorithm, we start with an empty tree T, add edges one at a time to T until we have an MST. At each step, pick the **smallest** edge that connects some vertex in T to a vertex outside T.



T = (red edges)

```
procedure Prim($G$, $w : E \to \mathbb{R}$, $r \in G.V$)
    # Initialization
    $T = \varnothing$
    $Q =$ Make-Queue($G.V$)
    # Place all vertices in the queue
    for $v \in G.V$ do
        $pri[v] = \infty$
        $\pi[v] =$ NIL
        $Q$.Insert($v$)
    end for
    # Now, set $r$ as the root
    $pri[r] = 0$
    $Q$.Decrease-Key($r$)
    # Main loop
    while not $Q$.Empty() do
        # Connect a vertex with minimum priority (edge weight)
        $u = Q$.Extract-Min()
        if $\pi[u] \neq NIL$ then $T = T \cup \{\pi[u], u\}$
        end if
        # Update priorities of neighbors of $u$
        for $v \in G.Adj[u]$ do
            if $v \in Q$ and $w(u, v) < pri[v]$ then
                $\pi[v] = u$
                $pri[v] = w(u, v)$
                $Q$.Decrease-Key($v$)
            end if
        end for
    end while
    return $T$
end procedure
```

Runtime

- Initialization: $\Theta(n)$

- Main loop: n iterations; one Extract-Min each time

  $\Theta(n \lg n)$

- Inner loop: examine each edge twice; at most Decrease-Key per edge

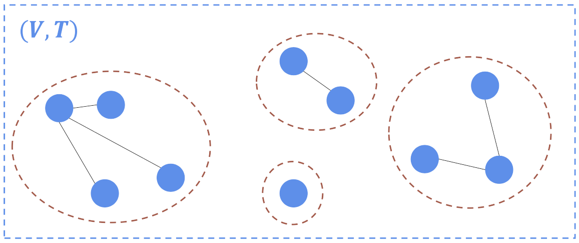
  $\Theta(m \lg n)$

- Overall: $\Theta((m + n) \lg n)$

Note that to make Decrease-Key efficient, we need to track $index[v]$ – the position of v in the min-heap – and update the index values during each swap.

4.2.2 Kruskal's algorithm

In Kruskal's algorithm, we repeatedly pick the **cheapest edge** of the graph G that **does not create a cycle** in the current set of edges.



```
procedure KRUSKAL($G$, $w$)
    # Sort edges: $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
    $T = \varnothing$
    for each $e_i = e_1, e_2, \ldots, e_m$ do
        let $(u_i, v_i) = e_i$
        if $u_i$, $v_i$ are not already connected by an edge in $T$ then
            $T = T \cup \{e_i\}$
        end if
    end for
    return $T$
end procedure
```

To determine if u_i, v_i are already connected, we can use BFS / DFS, which takes $\Theta(n)$ time [1]. The overall run-time would be $\Theta(m \cdot n)$. To make this algorithm more efficient, we can use disjoint sets.

4.3 Disjoint Sets

Data

- A collection $\{S, \ldots, S\}$ of disjoint, dynamic sets where $S_i \neq \varnothing$

  Each set in this collection contains a single *representative*

Operations

- MAKE-SET(x)

  Precondition: $x \in U$, $x \notin S_1, \ldots, x \notin S_n$

  Create a new set $S_x = \{x\}$, add S_x to the collection

- FIND-SET(x)

  Return the representative element for set S s.t. $x \in S$, or NIL if x no such S exists

[1] BFS / DFS takes $\Theta(m + n)$ time. Since $m \leq n - 1$, this step takes $\Theta(n)$ time.

Part II

Algorithms

CHAPTER 5

SORTING

5.1 Heap Sort

Implementation 1:

- Run BUILD-MAX-HEAP(A)
- Run EXTRACT-MAX(A) for $n-1$ times

Implementation 2:

- We modify the EXTRACT-MAX algorithm to swap the root with the last item

5.2 Quick Sort

```
procedure QUICKSORT($A$)
    if thenLEN($A$) $\leq 1$
        return $A$
    end if
    $L, p, G \leftarrow$ PARTITION($A$)
    return QUICKSORT($L$) $+ [p] +$ QUICKSORT($G$)
end procedure
```

5.2.1 Deterministic Quick Sort

In deterministic quick sort, we choose the pivot to be a certain index in the array. We can choose the pivot to be the first element, the last element, or the middle element.

- Run-time depends on input ordering

- Bad ordering would yield bad run-time, while random ordering would generally yield better run-time

5.2.2 Randomized Quick Sort

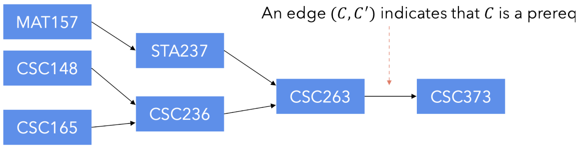
In randomized quick sort, we choose the pivot to be a random index in the array. Intuitively, the expected run-time would be the same as deterministic quick sort – $\Theta(n \lg n)$ – but the worst case run-time would be much better.

5.3 Topology Sort

Example. We wish to determine the sequence of courses to take during our undergraduate program

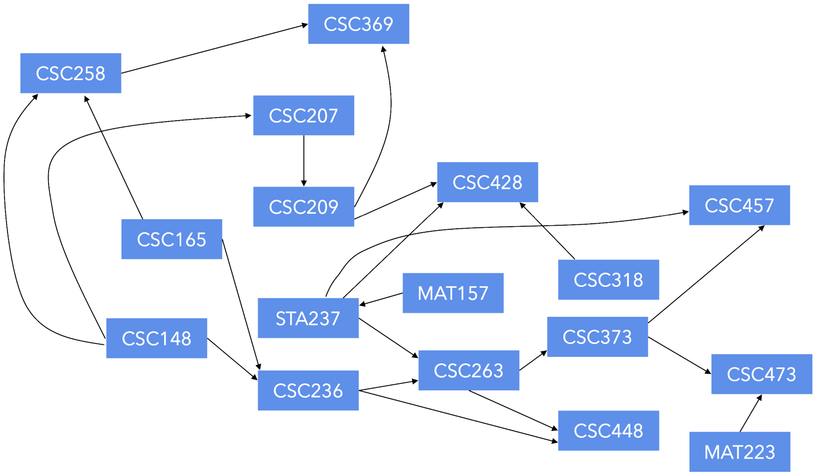
- A university course C may have a set of prerequisite courses that must be completed before C

- How can we determine a valid sequence of courses to take?



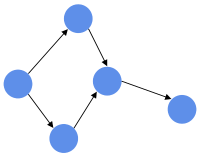
An edge (C, C') indicates that C is a prereq

When we add more nodes/edges, things become less clear

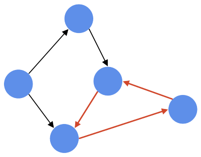


5.3.1 Directed Acyclic Graphs

A directed acyclic graph (or DAG) is a directed graph with no cycles



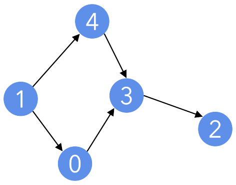
this **is** a DAG



this is **not** a DAG

5.3.2 Topological Sort

A topological sort/ordering of a DAG $G = (V, E)$ is a sequence of its vertices such that, if $(u, v) \in E$, then u appears before v in the sequence



1, 0, 4, 3, 2 is a topological ordering of these vertices

Given a DAG $G = (V, E)$, how can we efficiently compute a topological ordering of G?

- Claim: if we perform DFS on a DAG G, the resulting DFS forest contains no back edges.
- Claim: every out-neighbour of every vertex v has an earlier 'visited' time than v.
  - If (v, u) is a tree / forward edge, then u is a descendant of v. By the Parenthesis Theorem, $[disc[u], vis[u]]$ is contained in $[disc[v], vis[v]]$. Hence, $vis[u] < vis[v]$.
  - If (v, u) is a cross edge, then u was already visited by the time we finish visiting v. Hence, $vis[u] < vis[v]$.

We claim that a directed graph G can be topologically sorted iff G contains no cycles iff DFS finds no back edges.

1. Maintain a linked list L while performing DFS on $G = (V, E)$
2. When we finish visiting a node v (by setting $vis[v]$ and painting v black), add v to the head of the linked list L
3. Once DFS terminates, L contains a list of all vertices in V sorted in decreasing order by their 'visited' times

Part III

Analysis

CHAPTER 6

AVERAGE CASE ANALYSIS

In *average case analysis*, we are interested in the average performance of an algorithm. we take the average, or the expected value, over the distribution of the possible inputs.

For each n, define $S_n = \{\text{all inputs of size } n\}$, and if we consider the inputs to be random, then S_n is the sample space. For each $x \in S_n$, define $P(x)$ to be the probability that x will be chosen as the input. Define $t(x)$ as the number of steps preformed on input x. t is the random variable.

Then, the average case running time is defined as

$$\begin{aligned} T(n) &= E[t] \\ &= \sum_{x \in S_n} P(x) \cdot t(x) \end{aligned}$$

Example. Consider the linear search algorithm on a linked list L.

```
procedure LinSearch($L$, $x$)
    $z \leftarrow L.head$
    while $z \neq$ NIL and $z.data! = x$ do
        $z \leftarrow z.next$
    end while
    return $z$
end procedure
```

Let S_n be the sample space of all linked lists of size n. Let $P(x)$ be the probability that x is chosen as the input. Let $t(x)$ be the number of steps performed on input x.

- We need to know S_n with probability

  Consider the inputs $\texttt{input}_1 = ([1, 2, 3], 2)$ and $\texttt{input}_2 = ([\text{“}a\text{”}, \text{“}b\text{”}, \text{“}c\text{”}], \text{“}b\text{”})$, note that they will take the same steps. We only need one input for each possible value of t.

Define $S_n = \{([1,2,\ldots,n],1),([1,2,\ldots,n],2),\ldots,([1,2,\ldots,n],n),([1,2,\ldots,n],0)\}$.

- We assume all the inputs happen equally likely, then $P(x) = \frac{1}{n+1}$.
- We need an exact formula for $t(x)$.

  In practice, we choose some "key operations" s.t. counting **only** these operations is within a constant factor of total time – then set $t(x) =$ number of key operations.

  Here, we choose line 3, $z.data \neq x$, as the key operation.

Then, we have
$$\begin{aligned}T(n) &= \sum_{(L,i)\in S_n} t(L,i)\cdot P(L,i)\\ &= \frac{1}{n+1}\sum_{i=0}^{n} t([1,2,\ldots,n],i)\\ &= \frac{1}{n+1}\left(t([1,2,\ldots,n],0)+\sum_{i=1}^{n} i\right)\\ &= \frac{1}{n+1}\left(n+\frac{n(n+1)}{2}\right)\\ &= \frac{n}{n+1}+\frac{n}{2}\end{aligned}$$

Example. Consider Search(T,k) on a hash table T for a key k.

Assume t has m slots, and uses chaining to resolve collisions. Assume that prior to applying the textscSearch algorithm, the hash table contains n keys.

Assume the key k is samples uniformly at random from U.

Let $N(k)$ be the number of keys examined during search for k. $N(k)$ is the key operation.

$$\begin{aligned}E[N(k)] &= \sum_{k\in U} P[k]\cdot N(k)\\ &= \sum_{i=0}^{m-1}\sum_{\substack{k\in U\\ h(k)=i}} P[k]\cdot N(k) && \text{regroup terms}\\ &\leq \sum_{i=0}^{m-1}\sum_{\substack{k\in U\\ h(k)=i}} P[k]\cdot L_i && \text{since } N(k)\leq L_i \text{ when } h(k)=i\\ &= \sum_{i=0}^{m-1} L_i\cdot \sum_{\substack{k\in U\\ h(k)=i}} P[k]\\ &= \sum_{i=0}^{m-1} L_i\cdot P[h(k)=i]\\ &= \sum_{i=0}^{m-1} L_i\cdot\frac{1}{m}\end{aligned}$$

$$= \frac{1}{m}\sum_{i=0}^{m-1} L_i$$
$$= \frac{n}{m}$$

Example. Consider QUICKSORT on an array A of size n.

Let $S_n = \{\text{all permutations of } [1, 2, dots, n]\}$

We assume an uniform distribution of the inputs, then $P(x) = \frac{1}{n!}$.

Let the random variable $T(A)$ be the total number of comparisons between elements of A.

Define $X_{i,j} = \begin{cases} 1 & \text{if } i \text{ is compared to } j \\ 0 & \text{otherwise} \end{cases}$ for $1 \le i < j \le n$.

Then, $T(A) = \displaystyle\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{i,j}$.

The probability $P(X_{i,j} = 1) = P(i \text{ or } j \text{ appear in } A \text{ before all other values in range } [i...j])$

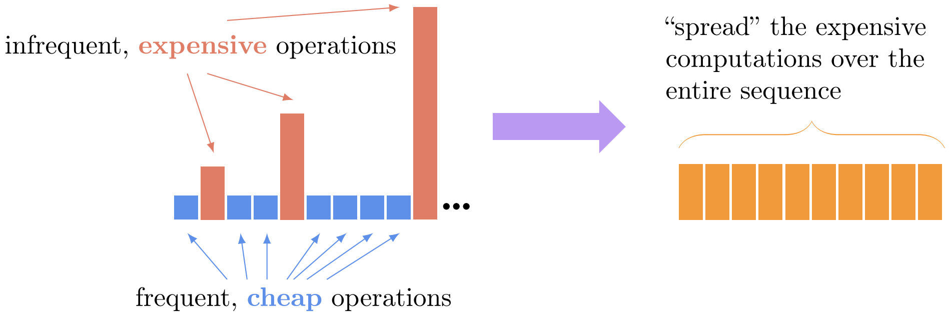
$$= \frac{1}{j-i+1} + \frac{1}{j-i+1}$$
$$= \frac{2}{j-i+1}$$

$$\begin{aligned}
\text{Then, } E[T(A)] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{i,j}] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} P(X_{i,j} = 1) \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{\alpha+1} && \text{substitute } j-i \text{ with } \alpha \\
&< \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} \Theta(\lg n) \\
&= \Theta(n \lg n)
\end{aligned}$$

CHAPTER 7

AMORTIZED ANALYSIS

In amortized analysis, we analyze the cost of a sequence of operations, not just a single operation. The amortized cost of a sequence of operations is the average cost per operation.


infrequent, expensive operations
frequent, cheap operations
“spread” the expensive computations over the entire sequence

- The *worst-case sequence complexicy* of a sequence S of k operations is the maximum possible total steps performed by S (taken over all possible inputs to the operations in S).

- The worst-case sequence complexity is **at most** $k \times$ the worst-case complexity of any individual operation in S.

- Suppose that the worst-case sequence complexity of a sequence of k operations is $T(k)$. Then the (worst-case) *amortized complexity* per operation of this sequence is $\frac{T(k)}{k}$.

- With amortized analysis, we take the average of the costs of multiple opertions (as opposed to average-case analysis, where we calculate the cost of a single operation by averaging over the input distribution)

7.1 Aggregated Method

In the *aggregated method*, we determine the upper bound $T(n)$ on the total cost of a sequence of N operations, then calculate the average cost per operation as $\frac{T(n)}{n}$.

Example. Consider when we insert into an array. We increase the size of the array by 4 when it is $\frac{3}{4}$ full.

For simplicity, suppose that each insert with no resizing requires c steps, for some constant $c \in \mathbb{N}^+$ (so each insert with resizing requires $c \cdot n + c$ steps, where n is the number of items in the array prior to resizing).

Operation Number	Cost
1	c
2	c
3	c
4	$4c$
5	c
6	c
7	c
8	$7c$
9	c
10	$10c$
$\vdots$	$\vdots$

Within a sequence of k insertions, we need to resize $\left\lfloor \frac{k-1}{3} \right\rfloor$ times. For all k insert operations, the total cost is

$$T(k) = c \cdot \sum_{i=1}^{\left\lfloor \frac{k-1}{3} \right\rfloor} (3i+1) + c \cdot \left(k - \left\lfloor \frac{k-1}{3} \right\rfloor\right)$$

$$\begin{aligned} \text{Note that } c \cdot \sum_{i=1}^{\left\lfloor \frac{k-1}{3} \right\rfloor} (3i+1) &= c \cdot \sum_{i=1}^{\left\lfloor \frac{k-1}{3} \right\rfloor} 3i + \sum_{i=1}^{\left\lfloor \frac{k-1}{3} \right\rfloor} 1 \\ &= 3c \cdot \sum_{i=1}^{\left\lfloor \frac{k-1}{3} \right\rfloor} i + c \cdot \left\lfloor \frac{k-1}{3} \right\rfloor \\ &= \frac{3}{2} c \cdot \left\lfloor \frac{k-1}{3} \right\rfloor \cdot \left(\left\lfloor \frac{k-1}{3} \right\rfloor + 1 \right) + c \cdot \left\lfloor \frac{k-1}{3} \right\rfloor \in \Theta(k^2) \end{aligned}$$

Thus, $T(k)$ is $\Theta(k^2)$. The amortized cost per operation is $\frac{T(k)}{k} = \Theta(k)$.

Example. Consider a k digit binary counter.

In this problem, we count the total number of bits changed in the counter. We can do this by counting the number of times each bit changes.

Bit Number	Number of Changes
0	m
1	$\approx \frac{m}{2}$
2	$\approx \frac{m}{4}$
$\vdots$	$\vdots$
i	$\approx \frac{m}{2^i}$
$\vdots$	$\vdots$

$$\begin{aligned}\text{Then, } T &= \sum_{i=0}^{(\lg m)-1} \frac{m}{2^i} \\ &= m \cdot \sum_{i=0}^{(\lg m)} \frac{1}{2^i} \\ &< m \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= 2m\end{aligned}$$

7.2 Accounting Method

The *accounting method* is a form of aggregate analysis which assigns to each operation an amortized cost which may differ from its actual cost. Early operations have an amortized cost higher than their actual cost, which accumulates a saved "credit" that pays for later operations having an amortized cost lower than their actual cost. Because the credit begins at zero, the actual cost of a sequence of operations equals the amortized cost minus the accumulated credit. Because the credit is required to be non-negative, the amortized cost is an upper bound on the actual cost. Usually, many short-running operations accumulate such credit in small increments, while rare long-running operations decrease it drastically.

Example. Consider a sequence of m INSERT for dynamic array.

Recall that the "cost" is the actual run-time, while the "charge" is the estimated amortized time.

Note that for $k = 2^n + 1$, we need $2^n = k - 1$ for reading and $2^n + 1 = k$ for writing. Thus, the cost is $2k + 1$.

Then, we know that $\text{cost}(\text{INSERT}(k)) = \begin{cases} 2k+1 & \text{if } k = 2^n + 1 \\ 1 & \text{otherwise} \end{cases}$

Define $\text{charge}(\text{INSERT}(k)) = \$5$. We need \$1 for writing the new element, and save \$4 as credit.

We need to prove our credit invariant: every element in the second half of the array has \$4 credit.

Proof. Proof by induction on the number of operations done.

- init: 0 elements, 0 credits
- Consider one INSERT Assuming credit invariant holds

  - If the array does not grow, then the new element gets \$4 credit. The credit invariant holds.
  - If the array does grow, it must be full, the total credits is $\$4 \cdot \frac{n}{2} = \$2n$, enough toi cover copying n elements. The new element gets \$4 credit. The credit invariant holds.

■

$$\begin{aligned} \text{Thus, Amortized} \leq \frac{\text{WCSC}}{m} &= \frac{\text{total cost}}{m} \\ &\leq \frac{\text{total charge}}{m} \quad \text{because of credit invariant} \\ &= \frac{5m}{m} \\ &= 5 \end{aligned}$$

Part IV

Appendices

BIBLIOGRAPHY

Books

INDEX