Lecture 7: Binary Heaps, Heapsort, Union-Find

Instructor: Saravanan Thirumuruganathan
Outline

1. Data Structures to speed up algorithms
   - Binary Heap
     - Heapsort
   - Union Find
In-Class Quizzes

- **URL:** http://m.socrative.com/
- **Room Name:** 4f2bb99e
Key Things to Know for Data Structures

- Motivation
- Distinctive Property
- Major operations
- Key Helper Routines
- Representation
- Algorithms for major operations
- Applications
Data Structures for Algorithmic Speedup

- BST and RBT are two examples of data structures to represent dynamic set.
- Today’s topic, Heap and Union-Find, can also be used to represent dynamic set.
- However, these are used more often to speed up algorithms.
Binary Heap
Motivation

- Heap Sort (CLRS is organized that way!)
- Priority Queue
- Most space efficient data structure
Priority Queue

- “Queue” data structure has a FIFO property
- Some times it is useful to consider priority
- Output element with highest priority first
Priority Queue - Major Operations

- Insert
- FindMin (resp. FindMax)
- DeleteMin (resp. DeleteMax)
- DecreaseKey (resp. IncreaseKey)
Priority Queue - Applications

- Dijkstra’s shortest path algorithm
- Prim’s MST algorithm
- Heapsort
- Online median
- Huffman Encoding
- A* Search (or any Best first search)
- Discrete event simulation
- CPU Scheduling
- ...
- See Wikipedia entry for priority for details

Kleinberg-Tardos Book and Wikipedia
Assume: for DeleteMin and DecreaseKey, pointer to element is given

- LinkedList
  - Insert:
Assume: for DeleteMin and DecreaseKey, pointer to element is given

- **Linked List**
  - Insert: $O(1)$
  - FindMin:
Assume: for DeleteMin and DecreaseKey, pointer to element is given

**Linked List**
- Insert: $O(1)$
- FindMin: $O(n)$
- DeleteMin:
Priority Queue - Candidate Implementations

- **Assume:** for DeleteMin and DecreaseKey, pointer to element is given

- **Linked List**
  - Insert: \( O(1) \)
  - FindMin: \( O(n) \)
  - DeleteMin: \( O(1) \)
  - DecreaseKey: 

- **Binary Heap**
  - Insert: \( O(\log n) \)
  - FindMin: \( O(1) \)
  - DeleteMin: \( O(\log n) \)
  - DecreaseKey: 

- **Binomial Heaps, Fibonacci Heaps etc.**
Priority Queue - Candidate Implementations

- Assume: for DeleteMin and DecreaseKey, pointer to element is given

- LinkedList
  - Insert: $O(1)$
  - FindMin: $O(n)$
  - DeleteMin: $O(1)$
  - DecreaseKey: $O(1)$

- Binary Heap
  - Insert: $O(lg n)$
  - FindMin: $O(1)$
  - DeleteMin: $O(lg n)$
  - DecreaseKey: $O(lg n)$

- Binomial Heaps, Fibonacci Heaps etc.
Binary Heaps

- Perfect data structure for implementing Priority Queue
- MaxHeap and MinHeap
- We will focus on MaxHeaps in this lecture
Complete Tree

- Perfectly balanced, except for bottom level
- Elements were inserted top-to-bottom and left-to-right

---

\(^2\)http://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/BinomialHeaps.pdf
Heap Property

- Heap is a binary tree (NOT BST)
- Heap:
  - **Completeness** Property: Heap has restricted structure. It must be a complete binary tree.
  - **Ordering** Property: Relates parent value with that of its children
- MaxHeap property: Value of parent must be greater than both its children
- MinHeap property: Value of parent must be less than both its children
- Heap with $n$ elements has height $O(\lg n)$
Max Heap Example

3 Wikipedia page for Heap
Major Operations

- Insert
- FindMax
- DeleteMax (aka ExtractMax)
- IncreaseKey
Key Helper Routines

- Max-Heapify (or Min-Heapify)
- Bubble-Up
- Bubble-Down
- Heapify
Representation: Arrays

- Very efficient implementation using arrays
- Possible due to completeness property
- Parent\( (i) \): return \( \lfloor i/2 \rfloor \)
- LeftChild\( (i) \): return \( 2i \)
- RightChild\( (i) \): return \( 2i + 1 \)
Representation: Arrays

CLRS Fig 6.1
Max-Heapify

- **Objective:** Maintain heap property
- **Invocation:** Max-Heapify($A, i$)
- **Assume:** Left($i$) and Right($i$) are valid max-heaps
- $A[i]$ might violate max-heap property
- **Bubble-Down** the violation
- **Analysis:** $O(\lg n)$
Max-Heapify: Example

6 CLRS Fig 6.2
Max-Heapify: Example

CLRS Fig 6.2
Max-Heapify: Example

CLRS Fig 6.2
Heap: Insert

18

16

14

8

2

4

1

7

9

10

3
Heap: Insert

16

14

8

2

4

1

10

9

3

18
Heap: Insert

The diagram shows a binary max heap with the following structure:

- The root node is 16.
- The left child of the root node is 14, and its left child is 8, with 2 as its leftmost child.
- The right child of the root node is 10, with 9 as its leftmost child.
- The root node has a dashed outline, indicating it has been inserted into the heap.

The heap maintains the property that each parent node is greater than or equal to its children.
Heap: Insert
Heap: Insert

- Insert element at first available slot (no completeness property violation!)
- Fix heap property violations by bubbling up the violation till it is fixed
- Complexity:
Heap: Insert

- Insert element at first available slot (no completeness property violation!)
- Fix heap property violations by bubbling up the violation till it is fixed
- Complexity: $O(lg n)$
Heap: FindMax

- Look at the root element
- Time complexity: $O(1)$
Delete the maximum element (root)
Fix the heap
Heap: DeleteMax

18

16

8

2

4

1

14

7

9

3

10
Heap: DeleteMax
Heap: DeleteMax

Diagram of a heap with elements 7, 16, 14, 8, 2, 4, 1, 10, 9, 3.
Heap: DeleteMax

Diagram of a heap with nodes labeled 16, 7, 8, 4, 2, 14, 9, 3, and 10.
Heap: DeleteMax
Heap: DeleteMax

Diagram of a max heap with keys 16, 14, 8, 7, 2, 4, 1, 10, 9, 3.
Heap: DeleteMax

- Remove root
- Replace root with last element (does not affect Completeness property)
- Fix heap violations by bubbling it down till it is fixed
- Complexity:
Heap: DeleteMax

- Remove root
- Replace root with last element (does not affect Completeness property)
- Fix heap violations by bubbling it down till it is fixed
- Complexity: $O(\lg n)$
Heap: IncreaseKey

- Given a node, increase its priority to a new, higher value
- Fix heap property violations
**Heap: IncreaseKey**

**IncreaseKey**: Increase value of 4 to 15

---

9 CLRS Fig 6.5
Heap: IncreaseKey

\[ CLRS \text{ Fig 6.5} \]
Heap: IncreaseKey

CLRS Fig 6.5
Heap: IncreaseKey

- Update element
- Fix heap violations by bubbling it up till it is fixed
- Complexity: $O(\log n)$
Heap : IncreaseKey

- Update element
- Fix heap violations by bubbling it up till it is fixed
- Complexity: $O(\lg n)$
Build-Max-Heap

- Given an array \( A \), convert it to a max-heap
- \( A.length \): Length of the array
- \( A.heapSize \): Elements from 1 ... \( A.heapSize \) form a heap
- **Build-Max-Heap**\( (A) \):
  - \( A.heapSize = A.length \)
  - for \( i = \lfloor A.length/2 \rfloor \) down to 1
    - Max-Heapify\( (A, i) \)
- **Analysis**: \( O(n) \) (See book for details)
Build-Max-Heap: Example

CLRS Fig 6.3
Build-Max-Heap: Example

\[ \text{CLRS Fig 6.3} \]
Build-Max-Heap: Example

CLRS Fig 6.3
Build-Max-Heap: Example \(^{16}\)

\(^{16}\)CLRS Fig 6.3
Build-Max-Heap: Example

CLRS Fig 6.3
Build-Max-Heap: Example

CLRS Fig 6.3
HeapSort(A):
    Build-Max-Heap(A)
    for i = A.length down to 2
        A.heapSize = A.heapSize - 1
    Max-Heapify(A, 1)
Heap Sort: Example

CLRS Fig 6.4
Heap Sort: Example

CLRS Fig 6.4
Heap Sort: Example\textsuperscript{21}

\textsuperscript{21}CLRS Fig 6.4
Heapsort: Example

CLRS Fig 6.4
Heap Sort: Example

CLRS Fig 6.4
Heap Sort: Example

\[24\] CLRS Fig 6.4
Heap Sort: Example\textsuperscript{25}

\textsuperscript{25}CLRS Fig 6.4
Heap Sort: Example

---

26 CLRS Fig 6.4
Heap Sort: Example

CLRS Fig 6.4
Heap Sort: Example

28 CLRS Fig 6.4
Heap Sort: Example

CLRS Fig 6.4
HeapSort: Analysis

- Operations:
  - Build-Max-Heap:
Operations:
- Build-Max-Heap: $O(n)$
- $n$ Max-Heapify:
HeapSort: Analysis

- Operations:
  - Build-Max-Heap: $O(n)$
  - $n$ Max-Heapify: $n \times \lg n = O(n \lg n)$
  - Complexity: $O(n) + O(n \lg n) = O(n \lg n)$
HeapSort

- Very efficient in practice - often competitive with QuickSort
- In-Place but not stable (why?)
- Requires constant extra space
- Best, average and worst case complexity is $O(n \lg n)$ (unlike Quicksort)
Disjoint Sets ADT

Objective: Represent and manipulate disjoint sets (sets that do not overlap)

Required Operations
- MakeSet(x): Create a new set \( \{x\} \) with single element \( x \)
- Find(x): Find the set containing \( x \)
- Union(x, y): Merge sets containing \( x \) and \( y \)
Disjoint Sets: Example

- **Objects.**
  
  0 1 2 3 4 5 6 7 8 9

- **Disjoint sets of objects.**
  
  0 1 {2 3 9} {5 6} 7 {4 8}

- **Find query:** are objects 2 and 9 in the same set?
  
  0 1 {2 3 9} {5 6} 7 {4 8}

- **Union command:** merge sets containing 3 and 8.
  
  0 1 {2 3 4 8 9} {5 6} 7

---

30 [https://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf](https://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf)
Disjoint Sets: Applications

- Network connectivity: are two computers connected?
- Compilers: are two variables aliases?
- Image segmentation: are both pixels in same segment?
- Chip design: are two transistors connected to each other?
- Maze design
- Speeding up Kruskal’s MST algorithm
- Many many more

Disjoint Sets: Naming

- Represent each disjoint set by a unique name
- For convenience, the name is one of its elements
- This element is called the leader of the set
- Find(x) returns the leader of set containing x
- Typically, Union takes leaders as input. For eg, Union(a, b). If not easily fixable by Union(Find(a), Find(b))
Data Structures for Disjoint Sets

Objective:

- Design an efficient data structure to represent DS ADT
- Assume that there are $N$ elements - represented by $1, \ldots, N$
- $M$ operations (any mixture of union and find)

Candidate Representations:

- Array based
- Linked List based
- Tree based
Disjoint Sets Implementation: Arrays

Idea:
- Maintain an array $A$ with $N$ elements
- $A[i]$ stores the leader for set containing element $i$

Find: Find(3) = 4, Find(6) = 8

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i]</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

Union: Union(4, 8)

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i]</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>
Disjoint Sets Implementation: Arrays

Analysis:

- Find:
  \( O(1) \)

- Union:
  \( O(N) \)

Complexity for \( M \) operations:
\( O(MN) \)
Analysis:
- Find: $O(1)$
- Union:
Analysis:
- Find: $O(1)$
- Union: $O(N)$
- Complexity for $M$ operations: $O(MN)$
Disjoint Sets Implementation: Linked List

Idea:
- Represent each set as a linked list
- Set first element of each linked list as the leader

\[\text{CLRS Fig 21.2}\]
Analysis:

- Find:

\[ \mathcal{O}(1) \]

- Union:

\[ \mathcal{O}(N) \]

Complexity for \( M \) operations:

\[ \mathcal{O}(MN) \]
Analysis:

- Find: $O(1)$
- Union:
Disjoint Sets Implementation: Linked List

Analysis:

- Find: $O(1)$
- Union: $O(N)$
- Complexity for $M$ operations: $O(MN)$
Disjoint Sets Implementation: Tree

- Also called as Union-Find data structure
- Idea: Represent each set as a tree
- Store all sets as a forest (collection of disconnected trees)
- Allow each node to have arbitrary number of children
- Root of each tree is the leader
Union-Find: Up-Tree Representation

Initial state
1 2 3 4 5 6 7

Intermediate state
1 3
2
5
4

Roots are the names of each set.

Union-Find: Up-Tree Representation

Initial state

1  2  3  4  5  6  7

Intermediate state

1  3

2

7

5  4

6

Roots are the names of each set.

Union-Find: Find Operation

Find(6) = 7

Union-Find: Union Operation

[Diagram of a tree structure with nodes labeled 1, 2, 3, 4, 5, 6, 7, and an arrow indicating a union operation between nodes 1 and 7]

Union-Find: Up-Tree Implementation

Up[x] = -1 means x is a root.

Union-Find: Worst Case

Union(1,2)

Union(2,3)

Union(n-1,n)

Find(1)  n steps!!

Union-Find: Candidate Improvements

- Improve Union
  - Union by Size
  - Union by Rank (depth)

- Improve Find

39http://www.eecs.wsu.edu/~ananth/CptS223/Lectures/UnionFind.pdf
Union by Size (Weighted Union): Always point smaller tree to root of larger tree. Break ties arbitrarily.

Union-By-Size:

An arbitrary union could end up unbalanced like this:

More details at:

http://www.eecs.wsu.edu/~ananth/CptS223/Lectures/UnionFind.pdf
Union by Rank: Always point tree with smaller rank (depth) to root of larger tree. Break ties arbitrarily.

Union(3,7)

So, connect root 3 to root 4

http://www.eecs.wsu.edu/~ananth/CptS223/Lectures/UnionFind.pdf
Union by Size: Best Case

Complexity:

Union by Size: Best Case

Complexity: Union and Find take $O(1)$ time

Union by Size: Worst Case

Complexity:

Union by Size: Worst Case

Complexity: Union takes $O(1)$ and Find take $O(lg \ n)$ time

Union-Find: Improved Find

Path Compression: When doing find, point all nodes on search path to root.

Path Compression:

(a)

(b)

46 CLRS Fig 21.5
Amortized analysis

Similarity with Red-Black trees

Worst Case Analysis of Union-by Size + Path Compression

- Single Union-by-Size: $O(1)$
- Single Find with Path Compression: $O(lg \, n)$
- Amortized Complexity for $M \geq N$ operations: $O(m \log^* \, n)$
Log* Function

- \( \log^* 2 = 1 \)
- \( \log^* 4 = \log^* 2^2 = 2 \)
- \( \log^* 16 = \)

\[^{47}\text{http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf}\]
Log* Function

- $\log^* 2 = 1$
- $\log^* 4 = \log^* 2^2 = 2$
- $\log^* 16 = \log^* 2^{2^2} = 3$ (i.e. $\log \log \log 16 = 1$)
- $\log^* 65536 =$

---

Log* Function

- \( \log^* 2 = 1 \)
- \( \log^* 4 = \log^* 2^2 = 2 \)
- \( \log^* 16 = \log^* 2^{2^2} = 3 \) (i.e. \( \log \log \log 16 = 1 \))
- \( \log^* 65536 = \log^* 2^{2^{2^2}} = 4 \) (i.e. \( \log \log \log \log 65536 = 1 \))
- \( \log^* 2^{65536} = \)

\[47\text{http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf}\]
Log* Function

- $\log^* 2 = 1$
- $\log^* 4 = \log^* 2^2 = 2$
- $\log^* 16 = \log^* 2^{2^2} = 3$ (i.e. $\log \log \log 16 = 1$)
- $\log^* 65536 = \log^* 2^{2^{2^2}} = 4$ (i.e. $\log \log \log \log 65536 = 1$)
- $\log^* 2^{65536} = \ldots = 5$ (i.e. $\log \log \log \log \log 2^{65536} = 1$)
- In summary, for all reasonable $n$, $\log^* n \leq 5$

---

Tightener Bound

- Tarjan’s tighter bound when $M \geq N$, $\Theta(M \alpha(M, N))$
- $\alpha(a, b)$ is the inverse Ackerman function
- It grows even slower than $\log^* n$!
Summary

Major Concepts:

- Binary Heap
- Heapsort
- Disjoint set data structures
- Union-Find