Lecture 10 - Heaps and Binary Search Trees

Fall 2025, Korea University

Instructor: Gabin An (gabin_an@korea.ac.kr)

Course Outline (Before Midterm) - Recap

- Part 1: Basics
  - ~~Divide and Conquer~~
  - ~~Basic Sorting Algorithms (Insertion Sort & Merge Sort)~~
  - ~~Asymptotic Analysis (Big-O, Big-Theta, Big-Omega)~~
  - ~~Solving Recurrences Using Master Method~~
- Part 2: Advanced Selection and Sorting
  - ~~Median and Selection Algorithm~~
  - ~~Solving Recurrences Using Substitution Method~~
  - ~~Quicksort, Counting Sort, Radix Sort~~
- Part 3: Data Structures
  - **Heaps, Binary Search Trees, Balanced BSTs** - *Now we are here!* 📍

Data Structures

- So far, we've ignored *how* data structures are implemented.
- But operation runtimes can vary drastically depending on the choice of structure!

Motivation for New Data Structures

Operation	Unsorted Linked List	Sorted Array
Search	$\Theta(n)$	$\Theta(\log n)$
Select k-th	$\Theta(n)$	$\Theta(1)$
Rank	$\Theta(n)$	$\Theta(\log n)$
Predecessor/Successor	$\Theta(n)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(n)$ 🤔
Delete	$\Theta(n)$	$\Theta(n)$ 🤔

- Sorted arrays are great for **static data**. However, what if data changes often?
- Need a data structure with **logarithmic** time for most operations.

Preliminary - Complete Binary Tree

- Definition: **Complete Binary Tree**
  A complete binary tree is a rooted binary tree where each level is full except maybe the last level, and all nodes on the last level are **as far left as they can be**.

```
1)        A        2)        A        3)        A        4)        A
        /   \              /   \              /   \              /   \
      B       D          B       D          B       D          B       D
     / \     / \        / \     /          /         \          \     / \
    E   F H   I        E   F H            E           H          E F   H
```

  - What are the complete binary trees?

Binary Min Heap 🏔️

A binary min heap is a complete binary tree in which **the children nodes have a higher value (lesser priority) than the parent nodes**, i.e., any path from the root to the leaf nodes, has an ascending order of elements.

```
       1                1                9                1
     /   \            /   \            /   \            /   \
    2     3          2     6          2     6          2     6
   / \   / \        / \   /          / \   / \        / \     \
  8   4 7   6      3   4 7          3   4 7   8      3   4     8

Binary Min Heap  Binary Min Heap
```

Formulation

- A binary heap stores elements in a complete binary tree with a root `r`.
- Each node `x` has
  - `key(x)` (the key of the element stored in `x`),
  - `p(x)` (the parent of `x`, where `p(r) = NIL`),
  - `left(x)` (the left child of `x`),
  - `right(x)` (the right child of `x`).

  > The children of `x` are either other nodes or `NIL`.
- Suppose that we always maintain a pointer to the last node in the heap, as well as a pointer to the next node to be created.

Basic Operations

- Binary min-heaps (which we call heaps for short) support two operations: `insert(i)` and `extract-min`.
    - `extract-min` outputs the **minimum** element, and then deletes it from the heap.
- Heap excels at `insert` + `extract-min` workflows, which are particularly useful if you want to have a **priority queue** where elements arrive in an arbitrary order, but always leave in order of their key/priority.
- You can implement other operations such as `search(i)` and `delete(i)` on a heap, but they would not be efficient (take $\Theta(n)$ time).

`insert(i)`

- To insert, add the new node at the bottom and “bubble up” by swapping with its parent until the heap property holds.

```
      2       ||       2              |       2          |       1 <-
    /   \     ||     /   \            |     /   \        |     /   \
   3     5    ||    3     5           |    3     1 <-    |    3     2
  / \   /     ||   / \   / \          |   / \   / \      |   / \   / \
 8   4 7      ||  8   4 7   1 <-      |  8   4 7   5     |  8   4 7   5
              ||                      |                  |
  Original    ||      Add 1           |  Swap 1 & 5      |   Swap 1 & 2
```

`extract-min`

- To extract-min, we save the key of the root (min), replace it with the key of the last node, and delete the last node.
- Then we recursively propagate the key copied from the last node down the tree.

```
      1         ||        5       min  |         2       min  |         2
    /   \       ||      /   \     ---  |       /   \     ---  |       /   \
   3     2      ||     3     2     1   |      3     5     1   |      3     5
  / \   / \     ||    / \   /     ---  |     / \   /     ---  |     / \   /
 8   4 7   5    ||   8   4 7           |    8   4 7           |    8   4 7
                ||                     |                      |
                ||     Save 1 and      |                      |
   Original     ||   move 5 to root    |      Swap 5 & 2      |     return 1
```

Time Complexity of `insert(i)` and `extract-min`

- A binary min-heap is a complete binary tree, so its height is $O(\log n)$.
- Thus, both operations take $\Theta(\log n)$ swaps in the worst case.

Operation	Unsorted Linked List	Sorted Array	Binary Min Heap
Insert	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
Extract-Min	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$

Binary Search Tree (BST) 🌲

A binary search tree is a binary tree where all keys in a node’s left subtree are less than the node’s key, and all keys in its right subtree are greater.

```
     4                4                9                1
   /   \            /   \            /   \            /   \
  2     5          2     6          2     6          2     6
 / \     \        / \   / \        / \   / \        / \
1   3     6      1   3 5   7      1   4 7   8      3   4

    BST              BST           Not a BST        Not a BST
```

BST Property 1: Relationship to Quicksort

In a BST, each node `x` acts like a pivot in Quicksort for the keys in its subtree:

- Left subtree: keys < `key(x)`
- Right subtree: keys > `key(x)`

```
      4          - 1, 2, 3 are less that 4
    /   \        - 5, 6 are greater than 4
   2     5       - 1 is less than 2
  / \     \      - 3 is greater than 2
 1   3     6     - 6 is greater than 5
```

BST Property 2: Sorting with Inorder Traversal

An *inorder* traversal outputs keys in sorted order:

1. Traverse the left subtree (`inorder(left(x))`) if `left(x) != NIL`
2. Output `key(x)`
3. Traverse the right subtree (`inorder(right(x))`) if `right(x) != NIL`

With this approach, for every `x`, all keys in its left subtree will be output before `x`, then `x` will be output and then every element in its right subtree.

```
       4
     /   \
    2     5     1 -> 2 -> 3 -> 4 -> 5 -> 6
   / \     \
  1   3     6
```

BST Property 3: Subtree Property

```
1)        x                2)      x
        /   \                    /   \
       y    ...               ...     y
      / \                            / \
   ...   z                          z   ...
        / \                        / \
```

1. If `left(x)` is `y` and `right(y)` is `z`, then all keys in `z`'s subtree satisfy:
   - `y` < keys < `x`
2. If `right(x)` is `y` and `left(y)` is `z`, then all keys in `z`'s subtree satisfy:
   - `x` < keys < `y`

Basic Operations

- The three core operations on a BST are `search(i)`, `insert(i)`, and `delete(i)`.

`search(i)`

- To search for an element, we start at the root and compare the key of the node we are looking at to the element we are searching for.
    - If the node’s key matches, then we are done.
    - If the node's key is larger than the element, recursively search in the left tree
    - If the node's key is smaller than the element, recursively search in the right tree

```
       4
     /   \        search(3)
    2     5       -> 4 (large! search in the left tree)
   / \     \      -> 2 (small! search in the right tree)
  1   3     6     -> 3 (found!!)
```

`search(i)` - Continued

- What if the element does not exist in our BST?
  - We simply return the node that would be the parent of this node if we inserted it into our tree.

```
        4
      /   \        search(3.5)
     2     5       -> 4 (large! search in the left tree)
    / \     \      -> 2 (small! search in the right tree)
   1   3     6     -> 3 (small! but the right child is NIL; return node 3)
        \
       (3.5) <- does not exist
```

`insert(i)`

- Assume all keys are distinct.
- Find the parent node `x` where `i` should be inserted using `search(i)`.
- Create a new node `y` with `key(y)=i` and no children.
- Attach `y` as the left or right child of `x` according to the BST property:

```
x = search(i)  # parent of new node
y = Node(key=i, left=NIL, right=NIL, parent=x)
if i < key(x):
    left(x) = y
else:
    right(x) = y
```

Example: `insert(3)`

```
      4
    /   \        -> search(3)
   2     5       -> 4 (large! search in the left tree)
  /       \      -> 2 (small! search in the right tree)
 1         6     -> NIL (parent node found)
```

Create a new node 3 and attach it as the right child of 2

```
      4
    /   \
   2     5
  / \     \
 1   3     6
```

`delete(i)`

- Delection is a bit more complicated.
- To delete a node `x` that exists in our tree, we consider several cases:
  - Case 1: `x` has no childeren
  - Case 2: `x` has only one child `c`
  - Case 3: `x` has two childern, a left child `c1` and right child `c2`

`delete(i)` - Case 1: `x` has no childeren

We simply remove it.

```
      4                                         4
    /   \                                     /   \
   2     5      --- delete(6) --->           2     5
  / \     \                                 / \
 1   3     6                               1   3
```

`delete(i)` - Case 2: `x` has only one child `c`

We **elevate** `c` to take `x`'s position in the tree.

```
      4                                              4
    /   \                                          /   \
   2     5      --- delete(5) --->                2     6
  / \     \                                      / \
 1   3     6                                    1   3
```

`delete(i)` - Case 3: `x` has two childern, a left child `c1` and right child `c2`

We find `x` 's **immediate successor** `z` and have `z` take `x` 's position in the tree.

- `z` is in the subtree under `x` ’s right child `c2` and we can find it by running `z <- search(c2, key(x))` (i.e., searching `key` in the `c2` 's subtree)
- Since `z` is `x` 's successor, it doesn’t have a left child, but it might have a right child. Therefore, deleting the original `z` is either Case 1 or Case 2.

```
      2                                          3               3
    /   \                                      /   \           /   \
   1     5  -- delete(2) --> successor = 3 -->  1     5   -->   1     5
        /                                            /               /
       3                                           3*               4
        \                                            \
         4                                            4
```

Time Complexity of `search(i)`, `insert(i)`, and `delete(i)`

- **Search** runs in $O(\text{height of tree})$ in the worst case.
- **Insert** and **delete** call `search` a constant number of times and do $O(1)$ extra work, so their runtimes are also $O(\text{height of tree})$.
- Height of tree:
  - **Best case** (completely balanced): $O(\log n)$, **Worst case** (long chain): $O(n)$

```
   2                     3                       1
  / \                   /                         \
 1   3                 2                           2
                      /                             \
                     1                               3
Balanced        Long Lefttward Path    Long Rightward Path
```

Next Class: Self-Balancing BSTs

- To guarantee $O(\log n)$ height, we must **rebalance** after operations.
  - Examples of **self-balancing BSTs**: AVL tree, red–black tree, etc.
- In the next class, we'll explore **red–black trees**, the most popular self-balancing BST!

Quiz #2 is coming up after Chuseok (14th October).

- It will cover material from Lectures 8, 9, and 10 (Open book).
- 4 questions
- 15 minutes

Midterm Exam: 23rd October (Mark your calendar! 🗓️)

Credits & Resources

Lecture materials adapted from:

- Stanford CS161 slides and lecture notes
  - https://stanford-cs161.github.io/winter2025/
- *Algorithms Illuminated* by Tim Roughgarden
  - https://algorithmsilluminated.com/