COMP3600/6466 – Algorithms
Abstract Data Structures:
Binary Search Tree + Heaps
[CLRS 12.1-12.3, ch. 6]

Hanna Kurniawati

https://cs.anu.edu.au/courses/comp3600/
Comp_3600_6466@anu.edu.au
Abstract Data Structures

• Abstract Data Structures can be thought of as a mathematical model for data representation.

• An Abstract Data Structure consists of two components:
  • A container that holds the data
  • A set of operations on the data. These operations are defined based on their behavior (i.e., input-output and time (or sometimes space) complexity), rather than their exact implementation
Why bother with so many of them?

• Recall that algorithms transform input to output. This usually means the algorithm will perform certain operations on the input data, so as to transform them into the desired output.

• The way these data are represented in the algorithm could often influence the efficiency of the algorithm.

• Often, transforming the way the data are being represented makes solving becomes more efficient.
  • This technique is called Representation Change.
  • This is particularly useful when the data is dynamic, i.e., the input might change (the input size might increase, decrease, or the content changes) during run-time.
Topics

• Binary Search Tree
• Heaps
• AVL Tree
• Red-black Tree
Topics

• Binary Search Tree
  • What is it?
  • Tree walk & Querying
  • Insertion & Deletion
• Heaps
• AVL Tree
• Red-black Tree
What is Binary Search Tree (BST)?

An abstract data structure that represents the data as a binary tree with the following property:

- Each node in the tree contains the data itself, as key + satellite data, and pointers to its left child, right child, and parent node, usually denoted as left, right, p
  - Key: Data used for BST operations. In BST, all keys are distinct
  - Satellite data: Data carried around but not used in the implementation
  - The root node is the only node with $p = NIL$
- Suppose $x$ is a node of the BST. If $y$ is the left sub-tree of $x$ then, $y.key \leq x.key$. If $y$ is the right sub-tree of $x$ then, $y.key \geq x.key$
What is Binary Search Tree (BST)?

- Operations: Search, Min, Max, Successor, Predecessor, Insert, Delete in $O(h)$ time, where $h$ is the height of the tree
Topics

• Binary Search Tree
  ✓ What is it?
  • Tree walk & Querying
  • Insertion & Deletion
• Heaps
• AVL Tree
• Red-black Tree
Binary Search Tree (BST)

- Its property allows us to easily output all the keys in a sorted order in $\Theta(n)$ via *in-order tree walk*

\[
\text{INORDER-TREE-WALK}(x)
\]

1. if $x \neq \text{NIL}$
2. \text{INORDER-TREE-WALK}(x.left)
3. print $x.key$
4. \text{INORDER-TREE-WALK}(x.right)

**Example**
In-order tree walk: 2 3 4 6 7 9 13 15 17 18 20
Proof for $\Theta(n)$ time complexity

• Suppose $T(n)$ is the total time taken by inorder-tree-walk.
• The lower bound is straightforward, each node is visited at least once, hence $T(n) = \Omega(n)$.
• Now, we show $T(n) = O(n)$.
Proof for $\Theta(n)$ time complexity

- Suppose $T(n)$ is the time taken by inorder-tree-walk.
- The lower bound is straight-forward, each node is visited at least once, hence $T(n) = \Omega(n)$
- Now, we show $T(n) = O(n)$
  - Notice that $T(n) = T(k) + T(n-k - 1)$, where $k$ is #nodes in the left sub-tree. We can assume $T(1) = 1$
  - Now, let’s use substitution method, with base $T(1) = 1$
  - Inductive hypothesis: $T(n) \leq cn$
  - Induction step: $T(n + 1) = T(k) + T(n + 1 - k - 1)$
    \[
    \leq ck + c(n + 1 - k - 1)
    \]
    \[
    = ck + c(n + 1) - ck - c
    \]
    \[
    = c(n + 1) - c
    \]
    \[
    \leq c(n + 1)
    \]
Binary Search Tree (BST)

- Its property allows us to easily output all the keys in a sorted order in $\Theta(n)$ via \textit{in-order tree walk}

\begin{algorithm}
\caption{Inorder-Tree-Walk ($x$)}
\begin{algorithmic}[1]
\State \textbf{if} $x \neq \text{NIL}$
\State \textbf{Inorder-Tree-Walk} ($x$.\textit{left})
\State print $x$.\textit{key}$\quad$\text{3}
\State \textbf{Inorder-Tree-Walk} ($x$.\textit{right})\text{4}
\end{algorithmic}
\end{algorithm}

- What will the output be if we swap lines 2 & 3?
  - \textit{Pre-order tree walk}

- What will the output be if we swap lines 3 & 4?
  - \textit{Post-order tree walk}
Binary Search Tree: Search Operation

• To search for a key whose value is \( k \):

\[
\text{ITERATIVE-TREE-SEARCH}(x, k)
\]

1. while \( x \neq \text{NIL} \) and \( k \neq x.\text{key} \)
2. \hspace{1em} if \( k < x.\text{key} \)
3. \hspace{2em} \( x = x.\text{left} \)
4. \hspace{1em} else \( x = x.\text{right} \)
5. return \( x \)
Binary Search Tree: Min & Max

- Where’s the Min in a BST?
- Where’s the Max in a BST?

\[
\text{Tree-Minimum}(x)
\]

1. \textbf{while} \ x.left \neq \text{NIL} \\
2. \hspace{1cm} x = x.left \\
3. \hspace{1cm} \text{return} \ x

\[
\text{Tree-Maximum}(x)
\]

1. \textbf{while} \ x.right \neq \text{NIL} \\
2. \hspace{1cm} x = x.right \\
3. \hspace{1cm} \text{return} \ x
Topics

• Binary Search Tree
  ✓ What is it?
  ✓ Tree walk & Querying
  • Insertion & Deletion

• Heaps

• AVL Tree

• Red-black Tree
Binary Search Tree: Insertion

• Suppose we want to insert a new node \( z \), where \( z.key = v, z.left = NIL, z.right = NIL \)
• Traverse the tree until the right position of \( z \) is found
• Add \( z \) to the tree
Binary Search Tree: Insertion

\textbf{Tree-Insert}(T, z)

1 \hspace{0.5em} y = \text{NIL}
2 \hspace{0.5em} x = T.root
3 \hspace{0.5em} \textbf{while} x \neq \text{NIL}
4 \hspace{1.5em} y = x
5 \hspace{1.5em} \textbf{if} z.key < x.key
6 \hspace{2em} x = x.left
7 \hspace{1.5em} \textbf{else} x = x.right
8 \hspace{0.5em} z.p = y
9 \hspace{0.5em} \textbf{if} y == \text{NIL}
10 \hspace{1.5em} T.root = z \hspace{1em} // \text{tree } T \text{ was empty}
11 \hspace{0.5em} \textbf{elseif} z.key < y.key
12 \hspace{1.5em} y.left = z
13 \hspace{0.5em} \textbf{else} y.right = z
Binary Search Tree: Successor & Predecessor

• Successor of a node $x$ is the node with the smallest key greater than $x.key$ –that is, the node visited right after $x$ in in-order tree walk
  • If $x$ has a non-empty right sub-tree, its successor is the minimum of its right sub-tree
  • Otherwise, $x$’s successor is the lowest ancestor of $x$ whose left child is also an ancestor of $x$

• Predecessor of a node $x$ is the node with the largest key smaller than $x.key$ –that is, the node visited right before $x$ in inorder tree walk
  • If $x$ has a non-empty left sub-tree, its predecessor is the maximum of its left sub-tree
  • Otherwise, $x$’s predecessor is the lowest ancestor of $x$ whose right child is also an ancestor of $x$
Binary Search Tree: Deletion

- Suppose we want to delete $z$. There’s 3 cases:
  - When $z$ has no children: Remove $z$ and modify its parent to replace $z$ with NIL
  - When $z$ has only 1 child: Elevate the child to take $z$’s position in the tree by modify its parent to replace $z$ with $z$’s child
Binary Search Tree: Deletion

- Suppose we want to delete $z$. There’s 3 cases:
  - When $z$ has no children: Remove $z$ and modify its parent to replace $z$ with NIL
  - When $z$ has only 1 child: Elevate the child to take $z$’s position in the tree by modify its parent to replace $z$ with $z$’s child
  - When $z$ has 2 children, consider its successor $y$ (since $z$ has 2 children, $y$ must be in its right subtree and $y$ does not have a left child):
    - If $y$ is $z$’s right child, replace $z$ with $y$
      - $z$’s left child becomes $y$’s left child, and $y$ replaces $z$’s position
    - If $y$ is not $z$’s right child, first replace $y$ by its right child and then replace $z$ with $y$
  - The requirement to find a successor cause deletion to take $O(h)$, rather than constant
Is Deletion Commutative?

• Suppose T is a binary search tree and we want to delete two elements, \( a_1 \) and \( a_2 \) from T. Would the resulting tree always be the same if we delete \( a_1 \) first and then \( a_2 \), compared to if we delete \( a_2 \) first and then \( a_1 \)?

• No, it’s not commutative. Example: Deleting 2 and then 1 in the tree below would result in a different tree compared to if we delete 1 and then 2.
Is Deletion Commutative?

Delete 2

Delete 1

Delete 1

Delete 2
Topics

• Binary Search Tree
  ✓ What is it?
  ✓ Tree walk & Querying
  ✓ Insertion & Deletion

• Heaps

• AVL Tree

• Red-black Tree
Topics

- Binary Search Tree
- Heaps
- AVL Tree
- Red-black Tree
Topics

✓ Binary Search Tree

• Heaps
  • What is it?
  • Heapify
  • Insertion & Building
  • Extract/Deletion

• AVL Tree

• Red-black Tree
What is A Heap?

• A heap is a **binary tree** that satisfies heap property:
  • A heap is a complete binary tree
  • The nodes contain data similar to binary search tree:
    • Each node has a key, in which node order are based on
    • Each node may contain additional information
  • The parent node has key greater than the keys in its children
• Complete binary tree: A perfect binary tree where at the last level, some rightmost leaves may be missing
  • Perfect binary tree: A tree where all interior nodes have 2 children and all leaves are at the same level
• Since a heap is a complete tree, it can be implemented easily with an array
  • Left & right children becomes index of the array that holds the left & right child nodes
What is A Heap?

• There’s 2 types of heap: Max-heap and Min-heap
• The one we discussed in the previous slide is Max-heap
• Throughout this class, we’ll assume the heap is Max-heap unless stated otherwise
• Min-heap is similar to Max-heap but, the parent node has key less than the keys in its children
Example [CLRS sec. 6.1]
What is A Heap?

• Main Operations:
  • Heapify to ensure heap properties are satisfied
  • Insert a node to an existing heap
  • ExtractMax (for Min-heap, this operation is ExtractMin) to retrieve an element from the heap
• All of the above operations are $O(\log n)$, where $n$ is the number of nodes in the tree
Topics

✓ Binary Search Tree

• Heaps
  ✓ What is it?
  • Heapify
  • Insertion & Building
  • Extract/Deletion

• AVL Tree

• Red-black Tree
Heapify: Maintaining Heap Property

• Intuitively, heapify a node means traversing the tree from root until a suitable place (to ensure heap order) is found for the node

• Pseudo-code [CLRS sec. 6.2]

\[
\text{MAX-HEAPIFY}(A, i) \\
1 \quad l = \text{LEFT}(i) \\
2 \quad r = \text{RIGHT}(i) \\
3 \quad \text{if } l \leq A.\text{heap-size} \text{ and } A[l] > A[i] \quad \text{largest} = l \\
4 \quad \text{else } \text{largest} = i \\
5 \quad \text{if } r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}] \\
6 \quad \quad \text{largest} = r \\
7 \quad \text{if } \text{largest} \neq i \\
8 \quad \quad \text{exchange } A[i] \text{ with } A[\text{largest}] \\
9 \quad \text{MAX-HEAPIFY}(A, \text{largest})
\]
Example
Does max-heapify takes $O(\log n)$ time?

• The total time $T(n)$ is a recurrence
• The question is what’s the size of the sub-problem
• Since a heap is a complete binary tree, the largest imbalance, in which one sub-problem is maximised (in proportion to the total size) happens when the last level is half full. When this happens, we have:

  \[ n = 1 + \text{SizeLeftSubTree} + \text{SizeRightSubTree} \]

  \[ \text{SizeLeftSubTree} = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \]

  \[ \text{SizeRightSubTree} = \sum_{i=0}^{h-1} 2^i = 2^h - 1 \]

  Where $h$ is the height of the tree
Does max-heapify takes $O(\log n)$ time?

\[ n = 1 + \text{SizeLeftSubTree} + \text{SizeRightSubTree} \]
\[ n = 1 + 2^{h+1} - 1 + 2^h - 1 \]
\[ n = 2^h (2 + 1) - 1 \]
\[ 2^h = \frac{n + 1}{3} \]

\[ \text{SizeLeftSubTree} = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \]
\[ = 2 \frac{n+1}{3} - 1 \leq \frac{2n}{3} \]
Proof that max-heapify takes $O(\log n)$

- Therefore the total time for max-heapify is
  \[ T(n) \leq T\left(\frac{2n}{3}\right) + C \]

- Solving the recurrence gives us $T(n) = O(\log n)$

- We can also specify the time complexity in terms of the tree height, $h$, in which case $T(n) = O(h)$
Topics

✓ Binary Search Tree

• Heaps
  ✓ What is it?
  ✓ Heapify
  • Insertion & Building
  • Extract/Deletion
  • Applications

• AVL Tree

• Red-black Tree
Inserting a node to a heap

• Can also be used to build a heap when data is received dynamically

• Intuitively, add a new node at the bottom right most heap and then find a suitable position for the new node

• Pseudo-code [CLRS pp.164]

```
MAX-HEAP-INSERT(A, key)
1 A.heap-size = A.heap-size + 1
2 A[A.heap-size] = -\infty
3 HEAP-INCREASE-KEY(A, A.heap-size, key)

HEAP-INCREASE-KEY(A, i, key)
1 if key < A[i]
2 error "new key is smaller than current key"
3 A[i] = key
4 while i > 1 and A[PARENT(i)] < A[i]
5 exchange A[i] with A[PARENT(i)]
6 i = PARENT(i)
```
Example

Add 15

Apply heap-increase-key
• In the worst case, line 4—6 of heap-increase-key is called as many as the height of the tree. Therefore, the time complexity is $O(h) = O(\log n)$
Building a heap

• We can insert elements to the heap one by one using the insertion method we’ve discussed
  • Time complexity: $O(n \log n)$

• If we have all the elements a priori, can we do better?
  • Yes: $\text{BUILD-MAX-HEAP}(A)$

1. $A.\text{heap-size} = A.\text{length}$
2. for $i = \lceil A.\text{length}/2 \rceil$ downto 1
3. $\text{MAX-HEAPIFY}(A, i)$
Example

A 4 1 3 2 16 9 10 14 8 7

heapify
Time complexity for build-max-heap

• We’ll compute the total time as the number of times max-heapify is called multiply by the cost of each max-heapify: