Module 6: BST, Greedy Algorithms and Computational Complexity

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Learning Objectives

After completing this module, students will be able to do the following:

1. Describe Search Trees and its different operations.
2. Explain and apply Greedy Algorithm.
3. Describe what is the computational complexity and its different categories of decision problem sets.

Module 6 Study Guide and Deliverables
June 10 - June 16

Theme: BST, Greedy Algorithms and Computational Complexity

Topics:
- Parenthesization, edit distance, knapsack (Dynamic Programing)
- Recursive Activity Selector (Greedy)
- Computational complexity
- P and NP, NP-Completeness, NP-Hard Problems

Readings: Module 6 online content

Assignments:
- Term Project Presentation due **Tuesday, June 17 at 6:00 AM ET**
  - Share video presentation at “Media Gallery” on the left-hand course menu.
  - **How to record a video and share at the "Media Gallery" section?** Check out the direction to use Kaltura to capture and post or submit video.
  - Submit presentation slides and programming files at “Assignments” on the left-hand course menu.

Live Classrooms:
- **Wednesday, June 11 at 7:00 – 9:00 PM ET**
- Live office hours with a facilitator: TBD

Course Evaluation: Please complete the course evaluation once you receive an email or Blackboard notification indicating the evaluation is open. Your feedback is important to MET, as it helps us make improvements to the program and the course for future students.

Binary Search Trees

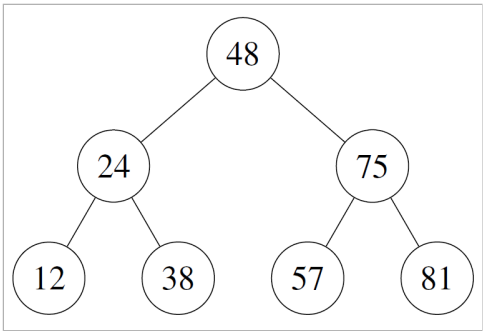
Reading from CLRS Book (Introduction to Algorithms, 3rd Edition):

Chapter 12 Binary Search Trees

- Sec. 12.1 What is a binary search tree?
- Sec. 12.2 Querying a binary search tree
- Sec. 12.3 Insertion and deletion

A binary search tree (BST) is organized as the name is stating in a binary tree structure. Each node of the tree is an object that has a key and a payload data (satellite data) and each node contains 3 important attributes or pointers to left node, right node and its parent node. Lef and right nodes are its child nodes. Figure 6.1 illustrates an example of a Binary Search Tree.

Figure 6.1: An Example of a Binary Search Tree



The keys in BST always satisfy the BST property.

For any given node in the tree, the keys of the left child node is smaller than the keys of the right child node.

The search tree data structure supports many dynamic-set operations, including the following operations.

- SEARCH – searches for a given key
- INSERT – Insert a new item to BST
- DELETE – Delete an item from BST
- PREDECESSOR – find the predecessor of a given key
- SUCCESSOR – find the successor of given key
- MINIMUM – find the min of the entire tree
- MAXIMUM – find the max of the entire tree

By using the BST property, we can have print out of all tree keys in a sorted order by a simple recursive algorithm.

Algorithm 25 INORDER-TREE-WALK(x)

```
1 : if x ≠ NIL then
2 :     INORDER-TREE-WALK(x.left)
3 :     print x.key
4 :     INORDER-TREE-WALK(x.right)
5 : end if
```

What is the run time of the INORDER-TREE-WALK algorithm?

$\Theta(n)$

- $T(n) \leq T(k) + T(n-k+1) + d$
- Assume that node x has a left subtree with k nodes.
- For some constant $d > 0$

Note: See CLRS Theorem 12.1.

Search in a BST

Algorithm 26 TREE-SEARCH(x, k)

```
1 : if x == NIL or k == x.key then
2 :     return x
3 : end if
4 : if x < x.key then
5 :     return TREE-SEARCH(x.left, k)
6 : else
7 :     return TREE-SEARCH(x.right, k)
8 : end if
```

We can rewrite the recursive BST search algorithm in an iterative form by “Opening-up and unrolling” the recursion into a while loop.

Algorithm 27 ITERATIVE-SEARCH(x, k)

```
1 : while (x ≠ NIL and k ≠ x.key) do
2 :     if (k < x.key) then
3 :         x = x.left
4 :     else
5 :         x = x.right
6 :     end if
7 : end while
8 : return x
```

- The running time of TREE-SEARCH is $O(h)$, where h is the height of the tree, or $O(\log(n))$.
- In worst case, $O(n)$.

Insertion in a BST

The insertion operation may cause that the BST need to reorganize and change to satisfy the BST property.

Givens for the insertion operation:

- A binary tree T
- A node z for which $z.key = v$, $z.left = NIL$ and $z.right = NIL$

Algorithm 28 INSET-SEARCH(T, z)

```
1 :  y = NIL                          ▷ trailing pointer y as the parent of x.
2 :  x = T.root                       ▷ Take the root of the given Tree T as x
3 :  while (x ≠ NIL) do               ▷ As long as x is not null do ...
4 :      y = x
5 :      if (z.key < x.key) then
6 :          x = x.left
7 :      else
8 :          x = x.right
9 :      end if
10 :  end while
11 :  z.p = y                         ▷ z.p is the parent of the z. Here we found a parent
12 :  if (y == NIL) then
13 :      T.root = z                  ▷ Our Tree T was an empty tree
14 :  else if (z.key < y.key) then
15 :      y.left = z                  ▷ Insert to left
16 :  else
17 :      y.right = z                 ▷ Insert to right
18 :  end if
```

- TREE-INSERT runs in $O(h)$ time on a tree of height h. If the tree is balanced $O(\log(n))$ and in worst case $O(n)$.

Deletion of a Key from BST

The goal is to delete a given key from BST so that the BST property holds after removal.

Three Cases are Possible:

- **Case 1 – z has no children**. We remove it by modifying its parent to replace z with NIL as its child.
- **Case 2 – z has only one child**. We would then elevate its childe to take z's position in the tree by modifying z's parent to replace z's child with z.
- **Case 3 – z has two children**. We need to find the z's successor y. In this case y, has to tbe in the z's right subtree so that we can swap position of z with y and remove z. This case is a bit tricky and 4 further sub-cases are possible.

  Sub-cases of Case 3:

  - **Case 3.1. z has no left child**. We replace z by its right child r.
  - **Case 3.2. z has no right child**. We replace z by its left child l.
  - **Case 3.3. z has two children and right sucessor y has no left child**. z has two children l and y. y has no left child and its right child is z. We remove z and replace it with y.
  - **Case 3.4. z has two children left child l and right child r, and r has a left child y**. We replace y by its right child x, and we set y to be r's parent. Then we can remove z and replace it with y, and make left child of y be l.

Figure 6.2: BST Delete Operation Case 3.1

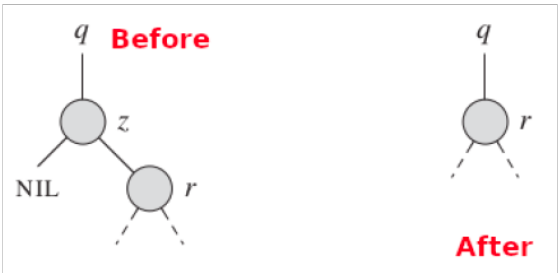


Figure 6.3: BST Delete Operation Case 3.2

Figure 6.4: BST Delete Operation Case 3.3

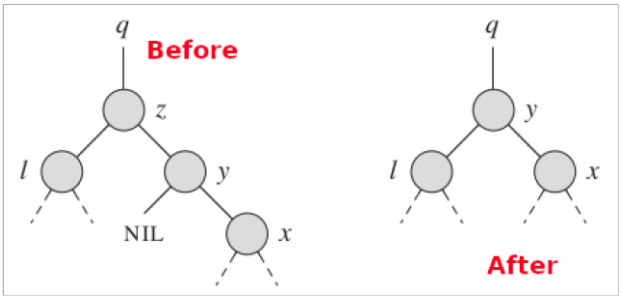
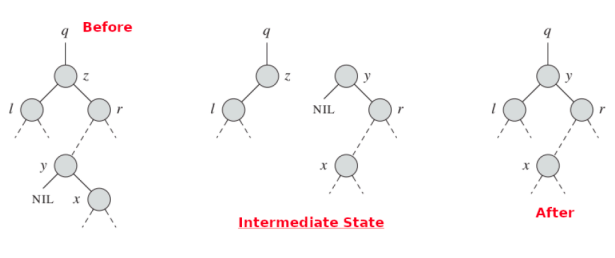


Figure 6.5: BST Delete Operation Case 3.4



To be able to move subtrees around within the binary search tree, we use a subroutine named TRANSPLANT that replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent then becomes node v's parent, and u's parent will be having u as its child.

Algorithm 29 TRANSPLANT(T, u, v)

```
                                  ▷ Tree T, replate subtree u with v
1 :  if u.p == NIL then           ▷ (If u is the root, replace it with v)
2 :      T.root = v
3 :  else if u == u.p.left then   ▷ If u is the left subtree, replace the left
4 :      u.p.left = v
5 :  else
6 :      u.p.right = v            ▷ If u is the right subtree, replace the right
7 :  end if
8 :  if v ≠ NIL then              ▷ We allow v to be null, if not null set its parent to th
9 :      v.p = u.p
10 : end if
```

Algorithm 30 TREE-DELETE(T, z)

```
                                                    ▷ Delete z from Tree T
1 :  if z.left == NIL then                          ▷ if we have case 3.1
2 :      TRANSPLANT (T, z, z.right)
3 :  else if z.right == NIL then
4 :      TRANSPLANT (T, z, z.left)                  ▷ if we have case 3.2
5 :  else
6 :      y = TREE − MINIMUM(z.right)                ▷ if we have case 3.3 and 3.4
7 :      if y.p ≠ z then
8 :          TRANSPLANT (T, y, y.right)
9 :          y.right = z.right
10 :         y.right.p = y
11 :     end if
12 :     TRANSPLANT (T, z, y)
13 :     y.left = z.left
14 :     y.left.p = y
15 :  end if
```

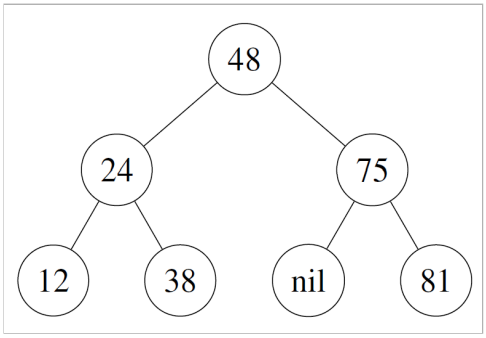
BST Additional Examples

The following three additional BST examples are for you to practice on your own. Please read the question, think carefully, figure out on your own first what the output would be, and then click “Show Answer” to compare yours to the suggested output.

BST Additional Example 1 - Test Yourself Practice 6.1

Insert Key into BST: What is the output result of the following BST when you insert 90?

Figure: Insert 90 in a Binary Search Tree

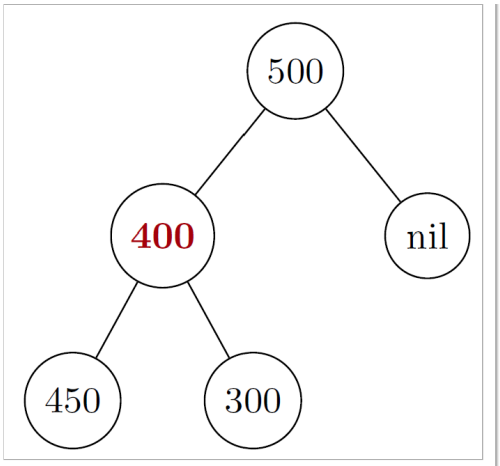


Suggested Solution:

BST Additional Example 2 - Test Yourself Practice 6.2

Remove Key from BST: What is the output result of the following BST when you remove 400?

Figure: Remove 400 in a Binary Search Tree

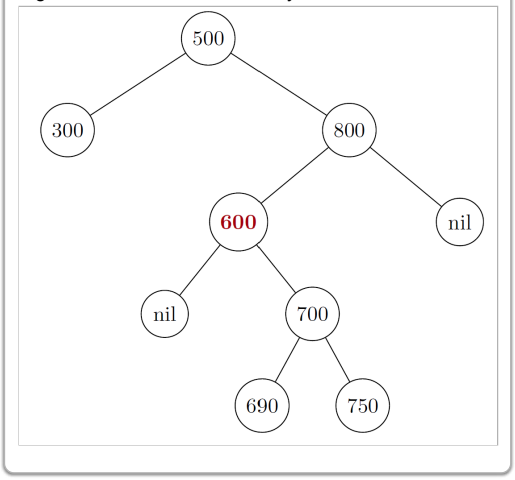


Suggested Solution:

BST Additional Example 3 - Test Yourself Practice 6.3

Remove Key from BST: What is the output result of the following BST when you remove 600?

Figure: Remove 600 in a Binary Search Tree



Suggested Solution:

Greedy Algorithms

Reading from CLRS Book (Introduction to Algorithms, 3rd Edition): Chapter 16 Greedy Algorithms

- Sec. 16.1 An Activity-Selection Problem

- We solve optimization problems by going over a set of steps.
- For many of the optimization problems using DP is an overkill.
- Simpler and more efficient algorithms can do the job as well.

A Greedy Algorithm always makes the choice that looks best at the current step/state.

It hopes that a locally optimal choice will lead to a globally optimal solution.

An Activity-Selection Problem

- We have a set of activities each with a specific start and end time.
- We have a classroom that the organizers of the activities want to use for the run the activities.
- Manager wants to give the classroom to maximum number of activities.
- Two activities a_i, a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$ (when they are after each other or before) and intervals do not overlap.
- We want to select mutually compatible activities.

Table 6.1: Set of Activities with their Start and Finish Time

$Activity_i$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
$Start_i$	1	3	0	5	3	5	6	8	8	2	12
$Finish_i$	4	5	6	7	9	9	10	11	12	14	16

For this example, we can select:

- $\{a_3, a_9, a_{11}\}$ of mutually compatible activities, but the set is not maximized.
- We can also select $\{a_1, a_4, a_9, a_{11}\}$ and subset of activities are larger.
- Another subset is $\{a_2, a_4, a_9, a_{11}\}$.

Dynamic Programing or Greedy Algorithm

- We can think about a dynamic-programming solution.
- Several choices when determining that subproblems to use in an optimal solution.
- We observe we need to consider only one choice (named the greedy choice), and if we make that choice, only one subproblem remains.
- We can then think of developing a recursive greedy algorithm to solve the activity-scheduling problem.
- We can then convert the recursive algorithm to an iterative one that is better to understand.

Dynamic Programming Subset

S_{ij} is the set of activities that start after activity a_i finishes and that finish before activity a_j starts.

We want to find a maximum set of mutually compatible activities in S_{ij} and maximize the size of the set A_{ij} that includes activity a_k.

We include a_k in the optimal solution. When we do so, the remaining is two sub-problems:

1. Find all mutually compatible activities S_{ik}. All activities that start after activity a_i finishes and that finish before activity a_k starts.
2. Find all mutually compatible activities S_{kj}. All activities that start after activity a_k finishes and that finish before activity a_j starts.

$A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$

The optimal solution contains a_k.

$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

We need to maximize the A_{ij}.

$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

Let us denote the size of an optimal solution $S_i j$ by $c[i, j]$, we can have the recurrence:

$$c[i, j] = c[i, k] + c[k, j] + 1$$

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ MAX_{a_k \in S_{ij}}\{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

We can then use dynamic programming to develop a recursive algorithm and memoize it, or we could work bottom-up and fill in table entries as we go along.

Making the Greedy Choice

- Think if we can have a greedy choice here?
- It can help to not consider all the choices inherent in recurrence.
- The activity-selection problem, can have a greedy choice.

Trust your Intuition!

- Choose an activity that leaves the resource available for as many other activities as possible.
- So, we would select an activity with the earliest finish time, since to have time/resources available for as many of the activities that follow it.

In our example, select a_1 because it would leave us as much possible time for other activities. After this greedy choice we have only one sub-problem to solve.

- After we select a_1, we do not need to consider activities that finish before a_1 starts. Why?
- Let define $S_k = \{a_i \in S : s_i \geq f_k\}$.
- If a_1 is in the optimal solution, then an optimal solution to the original problem consists of activity a_1 and all the activities in an optimal solution to the subproblem S.

> One important Big Question here is if our intuition is correct or not.
>
> Note: Choosing the a_{1} is not the only greedy choice for this problem.
>
> - Our greedy algorithm does not need to work bottom-up, like a table-based dynamic-programming algorithm.
> - It can be top-down, meaning to select an activity to put into the optimal solution and then solving the subproblem.
> - Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem.
> - Remember bottom-up technique means solving subproblems before making a choice.

Algorithm 31 RECURSIVE-ACTIVITY-SELECTOR(s, f ,k , n)

```
1 : m = k + 1
2 : while m ≤ n and s[m] < f[k] do          ▷ Try to find the first activity in S_k to finish
3 :     m = m + 1
4 : end while
5 : if m ≤ n then                           ▷ then call it recursively
6 :     return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m , n)
7 : else
8 :     return ∅
9 : end if
```

- s is set of activities,
- f is the set of finish times,
- k is the initialization,
- n is the number of activities.

How to start it up?

- Fictitious activity a_0 with $f_0 = O$ so that the subproblem S_0 is the entire problem set.
- The initial call of the above algorithm would be then RECURSIVE-ACTIVITY-SELECTOR(s, f, O, n).

Algorithm 32 GREEDY-ACTIVITY-SELECTOR(s, f)

```
1 : n = s.length
2 : A = {a_1}                                  ▷ Our Greedy Choice - Include it in the solution
3 : k = 1
4 :                ▷ Activities are sorted in order of monotonically increasing of their finish time.
5 : for m = 2 to n do
6 :     if s[m] ≥ f[k] then                    ▷ if the start of m is bigger than finish of k.n
7 :                    ▷ Remember f_k is always maximize the finish time of any activity in A.
8 :         A = A ∪ {a_m}                      ▷ Include this in the solution
9 :         k = m                              ▷ Swap m is the new k
10 :    end if
11 : end for
12 : return A
```

- The for-loop finds the earliest activity in S_k to finish.
- The loop considers each activity a_m in turn and adds a_m to A if it is compatible with all previously selected activities.
- Such an activity is the earliest in S_k to finish.

The run time of the above greedy solution is then $\Theta(n)$.

Other examples of Problems that can be solved by Greedy Algorithms.

- Fractional Knapsack Problem.
- Determine minimum number of money coins to give while making a change. (for example returning 37 cents, in 1 coin of 20 cents, 1 coin of 10 cents, 1 coin of 5 cents and 2 one cent coins.)

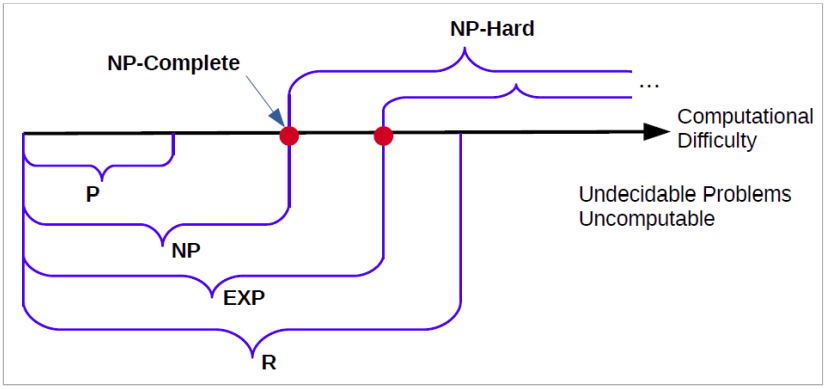
Computational Complexity

In this last section, we describe in a nutshell what is the computational complexity and its different categories of decision problem sets.

We can categorize the computation of complexity of all problems that we face in following categories.

Most problems that we may face are uncomputable.

Figure 6.6: Computation Complexity - P, NP, EXP, R



P – Polynomial. Set of all problems that we can solve in Polynomial (n^c) Time. All problems that we had in this course.

EXP – Exponential. Set of all problems that we can solve in exponential (2^{n^c}) time.

R – Recursive (finite time). Set of all problems that we can solve in finite time.

Example Problems

- Many problems that we saw in this course are in P.
- Shortest path in a weighted directed graph is in P.
- Detection of cycles in directed graph is in P.
- Playing Chess on an $n \times n$ size board is in Exponential complexity but not in P.
- Play a game named Tetris is in Exponential time.

Halting-Problem

Give an arbitrary computer program, knowing if the program will ever terminate and stop. The return value is a True/False, e.g. True if terminates and False if not.

The problem is uncomputable. You can say the problem is not in R. There is not algorithm that solves this problem in finite time on all given inputs.

NP – Nondeterministic Polynomial. NP is the set of all problems that we can solve by guessing the solution and following a set of steps in Polynomial time.

What is Nondeterministic? A nondeterministic algorithm makes random guesses and the answers the decision problem with YES or NO.

For example, the game Tetris is in set of NP problems, because we can show that we can solve Tetris by guessing in polynomial time.

Is $P \neq NP$? This is a billion dollar question.

Can we proof that we can engineer the luck in its general form. Can we make a lucky machine?

NP-Complete

All problems that are in the intersection of NP and NP-Hard problems.

For example Tetris is a NP-Complete problem.

Problem Reduction means that we can somehow convert our problem into other type of problems that we know how to solve it. We saw in our course a couple of these techniques.

NP-Complete problems are problems that are interreducible using polynomial time reduction techniques. For example, the 0-1 Knapsack Problem.