ICS 621: Analysis of Algorithms Fall 2019

Lecture 5

Prof. Nodari Sitchinava *Scribe: Honggen Zhang, Mojtaba Abolfazli, Sushil Shrestha*

1 Search Trees

1.1 Binary Search Trees

Each node of a binary search tree requires at least a variable to store key, left child pointer, right child pointer and possibly parent pointer. Binary search trees can be implemented in C programming language using:

```
struct Node{
    int key;
    Node *left;
    Node *right;
    Node *parent;
}

Node tree[totalNodes];
```

We can use an array of *Node* objects to store all the vertices in a binary tree. These data structures are stored sequentially in memory and can be represented as in Figure 1.

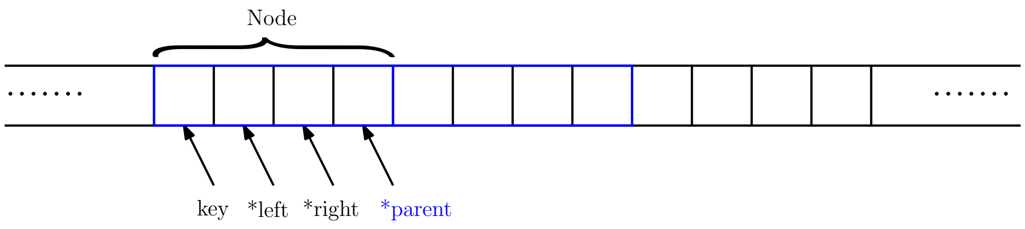

Figure 1: Struct representation in memory.

If we want to implement a binary search tree in a programming language that doesn't include the feature of struct or pointers, we can implement the same thing using 4 separate arrays to get the same feature as we have in our C implementation.

Both of the representations are identical except for the case when we consider caching effects on our data structure.

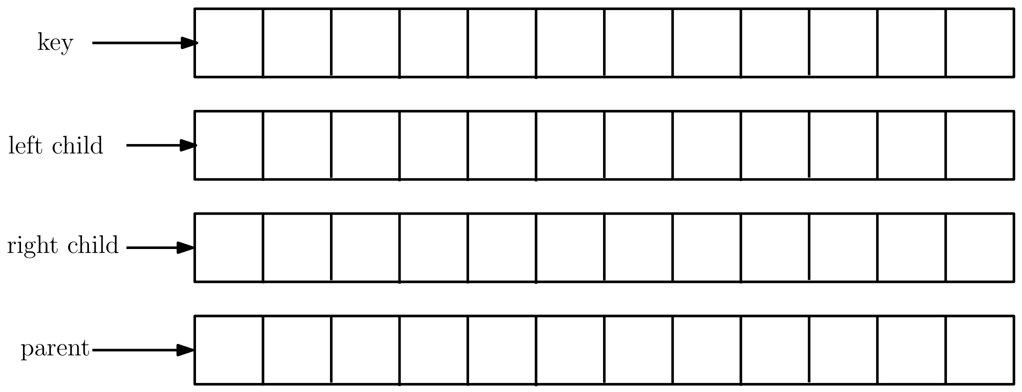

Figure 2: Array representation of binary tree

Balanced binary search tree

The cost of searching for an item in a balanced binary search tree is $\Theta(\log(n))$. Balanced binary tree can be maintained by rearranging the tree after certain number of inserts to the binary search tree.

Algorithm 1 Binary Search Tree Insertion

```
1: INSERT(80)
2: INSERT(50)
3: INSERT(10)
4: INSERT(5)
```

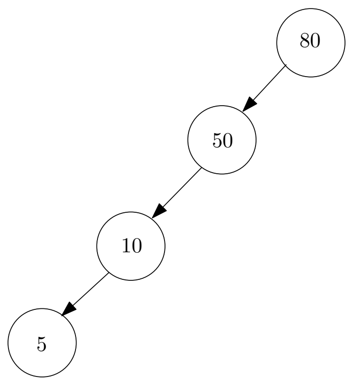

Figure 3: Unbalanced binary search tree

For example, consider a number of insert operations on a binary search tree as mentioned in Algorithm 1. The insertions result in a binary search tree corresponding to Figure 3. If we consider similar n insertions, the cost of search will be $O(n)$ in the worse case. Whereas if we balance the tree like in Figure 4, the cost of search can be reduced to $O(\log n)$.

Given the probability of access of nodes, cost of search operation in a binary tree can be further improved using an optimal binary search tree instead of a balanced binary tree. Before moving into the optimal binary search tree, let's review dynamic programming.

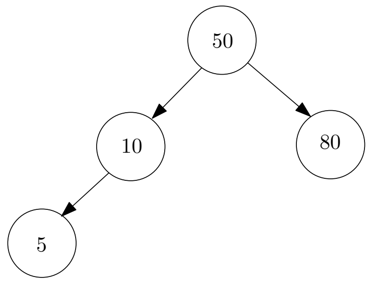

Figure 4: Balanced binary search tree

2 Dynamic Programming

Dynamic Programming (DP) solves problems by combining the solutions to subproblems. In other words, it is an optimization method over plain recursion. It means that if we have the solutions to some subproblems, we can use them later to solve other problems in a recursive manner. In DP, we solve each subproblem just once and record the solution in a table for later usage. By that, we can save the running time at a cost of using additional memory.

We will look at DP in the context of specific examples including Subset Sum, Knapsack, and Longest Increasing Subsequence. In addition, we introduce two approaches in DP: top-down and bottom-up and elaborate how they differ from each other in terms of implementation while their time complexity is usually of the same order.

2.1 Subset Sum

Problem: *Given a set of S of $1 \le n \le 20$ integers, and a positive integer x, is there a subset of S that sums to x?*

Example:

$$S = 17, 5, 7, 15, 3, 8 \qquad x = 16$$

Then, the subset $\{5, 3, 8\}$ of S that sums to 16.

2.1.1 Top-down Solution: Prune search space as you generate partial solutions

For the problem, let $\mathbf{SubsetSum(S[i:n], x)}$ be the function to find whether there is a subset of $\mathbf{S[i:n]}$ with sum equal to x. $S[i]$ is the *ith* number of elements in the set. The problem can be divided into two subproblems:

- Include the element $S[i]$, recurse on $i = i + 1, x = x - S[i]$.
- Exclude the element $S[i]$, recurse on $i = i + 1, x = x$.

If any of the subproblems return true, then return true. Because empty set is the subset of any set, when the $x = 0$, return true.

The following is the recursive formula for this problem.

$$SubsetSum(S[i:n], x) = SubsetSum(S[i+1:n], x);$$

or

$$SubsetSum(S[i:n], x) = SubsetSum(S[i+1:n], x - S[i]);$$

The following is a recursive algorithm that follows the recursive structure mentioned above.

Algorithm 2 Subset-sum: recursive

```
function MAIN()
    Initialize $S[1..n]$
    Initialize $x$
    return SUBSETSUM($S, x, 1, n$)

                                        ▷ Returns true iff exists subset within $S[i:n]$ that adds up to $x$;
function SUBSETSUM($S, x, i, n$)
    if $x = 0$ then
        return TRUE
    if $i > n$ or $x < 0$ then
        return FALSE
    if $S[i] > x$ then
        return SUBSETSUM($S, x, i+1, n$)
    else
        return SUBSETSUM($S, x, i+1, n$) or SUBSETSUM($S, x - S[i], i+1, n$)
```

The above solution may try all subsets of given set in the worst case. Therefore, time complexity of the above solution is exponential.

2.1.2 Dynamic Programming Solution

In the simple recursive solution, every time we calculate the function SUBSETSUM(S, x, i, n), we need to calculate the SUBSETSUM$(S, x, i+1, n)$ or SUBSETSUM$(S, x - S[i], i+1, n)$ over and over again. So if we can store the SUBSETSUM$(S, x, i+1, n)$ before we calculate the SUBSETSUM(S, x, i, n), we need calculate the SUBSETSUM$(S, x, i+1, n)$ or SUBSETSUM$(S, x - S[i], i+1, n)$ at most one time. Therefore, we create a 2D table $M[0..n][0..x]$ and use it to memoize the return values. The value of $S[i][x]$ will be true if there is a subset of $S[i:n]$ with sum equal to x, otherwise false. Finally, we return $S[i][x]$.

The following is a recursive algorithm that follows the recursive structure mentioned above.

Algorithm 3 Subset-sum: DP

```
function MAIN()
    Initialize $S[1..n]$
    Initialize $x$
    Initialize $M[1..n][1..x]$ to $UNDEFINED$ values
    return SUBSETSUMDP($S, x, 1, n, M$)

                                ▷ Returns true iff exists subset within $S[i : n]$ that adds up to $x$
function SUBSETSUMDP($S, x, i, n, M$)
    if $x = 0$ then
        return $M[i][x] =$ TRUE
    if $i > n$ or $x < 0$ then
        return FALSE
    if $M[i][x] \neq UNDEFINED$ then
        return $M[i][x]$
    if $S[i] > x$ then
        return $M[i][x] =$ SUBSETSUMDP($S, x, i + 1, n, M$)
    else
        return $M[i][x] =$ SUBSETSUMDP($S, x, i + 1, n, M$) or SUBSETSUMDP($S, x - S[i], i + 1, n, M$)
```

In this algorithm, in the worst case, we need calculate the $M[i][x]$ for every i and x at most once, so the worst cost is $O(xn)$.

2.2 0-1 Knapsack

Problem (0-1 Knapsack). *Given a set S of n items, each with its own value V_i and weight W_i for all $1 \leq i \leq n$ and a maximum knapsack capacity C, compute the maximum value of the items that you can carry. You cannot take fractions of items.*

Example:

$$\{(V_i, W_i)\} = \{(10, 17), (5, 7), (3, 8), (9, 15)\} \qquad C = 16$$

Then, the best value is 9 in the case of (V_4, W_4).

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. This problem is similar to the subset-sum problem. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the subset with the maximum value.

2.2.1 Recursive Solution

Like the subset-sum problem, MAXV(i, C) returns the maximum value among items $S[i : n]$ with remaining knapsack capacity of C. When $i = 1$, that is the initial problem, if the first item is not included in the knapsack, then this problem becomes the problem of finding the maximum value among items $S[2 : n]$ with remaining knapsack capacity of C. When the first item is included in the knapsack, then this problem becomes the problem of finding the maximum value among items $S[2:n]$ with remaining knapsack capacity of $C - W_1$. To consider all subsets of items, the Knapsack problem can be divided into two subproblems:

- The item is not included in the optimal subset, maximum value obtained by $S[i+1, n]$ and the maximum capacity C.
- The item is included in the optimal set, maximum value obtained by the value of *ith* item plus the maximum value obtained by $S[i+1, n]$ and $C - W_i$.

And when the $C \leq 0$, return false, because there is no capacity to store any item. When $i \geq n$, return false, because that is out of the range.

If $W_i > C$, then V_i cannot be included in the optimal subset and case 1 is the only possibility. Else, the V_i can be either included in the optimal subset or not included in the optimal subset.

The following is the recursive formula for this problem:

$$\textsc{maxV}(i, C) = \begin{cases} 0, & \text{if } i > n \\ 0, & \text{if } C \leq 0 \\ \textsc{maxV}(i+1, C), & \text{if } W_i > C \\ \max\{\textsc{maxV}(i+1, C), V_i + \textsc{maxV}(i+1, C - W_i)\}, & \text{if } W_i \leq C \end{cases}$$

The following is a recursive algorithm that follows the recursive structure mentioned above.

Algorithm 4 ksnapsack:recursive
```
function MAIN()
    Initialize $V[1..n], W[1..n]$ and $C$
    return MAXV($V, W, C, 1, n$)

function MAXV($V, W, C, i, n$)
    if $i > n$ or $C \leq 0$ then
        return 0
    if $W[i] > C$ then
        return MAXV($V, W, C, i+1, n$)
    else
        return $\max\{$MAXV$(V, W, C, i+1, n), V[i] +$ MAXV$(V, W, C - W[i], i+1, n)\}$
```

2.2.2 DP solution

This is similar to the Subset Sum problem. We create a 2D table $M[1..n][1..C]$ and use it to memoize the return values.

The following is recursive algorithm that follows the recursive structure mentioned above.

Algorithm 5 ksnapsack:DP

```
function MAIN()
    Initialize $V[1..n], W[1..n]$ and $C$
    Initialize $M[1..n][1..C]$ to $UNDEFINED$
    return MAXV$(V, W, C, 1, n, M)$

function MAXV$(V, W, C, i, n, M)$
    if $i > n$ or $C \leq 0$ then
        return 0
    if $M[i][C] \neq UNDEFINED$ then
        return $M[i][C]$
    if $W[i] > C$ then
        return $M[i][C] =$ MAXV$(V, W, C, i+1, n, M)$
    else
        return $M[i][C] = \max\{$MAXV$(V, W, C, i+1, n, M), V[i] +$ MAXV$(V, W, C - W[i], i+1, n, M)\}$
```

2.3 Longest Increasing Subsequence

The Longest Increasing Subsequence (LIS) problem asks to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for $A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$ is 4 and LIS $= \{-7, 2, 3, 8\}$. We employ DP to solve this problem in polynomial time.

Let's consider that LIS(i) returns the size of the longest increasing subsequence of the array $A[1:i]$. Then, we can write LIS(i) recursively by deciding whether to add $A[i]$ to the solution of LIS($i-1$) or not. Since we don't know if including/excluding $A[i]$ will block LIS to have longer subsequence later or not, we need to modify this recursion in a little smarter way. To deal with that, we always compare $A[i]$ with all previous $A[j]$ where $j < i$ since LIS(i) must be the longest increasing subsequence that ends at i, i.e.:

$$\text{LIS}(i) = \begin{cases} 1 & \text{if } \ i = 1 \\ \max \begin{cases} 1 & \\ \text{LIS}(1) + 1 & \text{if } \ A[i] > A[1] \\ \text{LIS}(2) + 1 & \text{if } \ A[i] > A[2] \\ \text{LIS}(3) + 1 & \text{if } \ A[i] > A[3] \\ \dots & \quad\dots \\ \text{LIS}(i-1) + 1 & \text{if } \ A[i] > A[i-1] \end{cases} & \text{if } \ i > 1 \end{cases}$$

The pseudocode for implementing the above steps is given in the following:

Algorithm 6 Longest Increasing Subsequence

```
function MAIN()
    Initialize $A[1..n]$
    Initialize $L[1..n]$ to $UNDEFINED$
    $best = 0$
    for $i = 1$ to $n$ do
        $current = \text{LIS}(A, i, L)$
        if $best < current$ then
            $best = current$
    return $best$

function LIS($A, i, L$)                    ▷ Returns longest common subsequence of $A[1..i]$
    if $L[i] \neq UNDEFINED$ then
        return $L[i]$
    if $i = 1$ then
        return $L[i] = 1$
    $best = 1$
    for $j = 1$ to $i - 1$ do
        if $A[i] > A[j]$ then
            $current = \text{LIS}(A, j, L) + 1$
            if $best < current$ then
                $best = current$
    return $L[i] = best$
```

The complexity to find LIS depends on nested loop and of the order $O(n^2)$. In other words, the entry LIS(i) takes $O(i)$ time to compute which gives $O(1) + \ldots + O(n) = O(n^2)$ overall complexity.

2.4 Bottom-up Approach in DP

The bottom-up approach of DP consists of first looking at the smaller subproblems, and then solving the larger subproblems using the solution to the smaller subproblems. We can think of that as a table filling algorithm where we start by filling out lower cells and then going up. For example, if we need to calculate fifth Fibonacci number, we calculate first, then second then third all the way up to the fifth number. This approach does not use recursion but it consumes more space, since we will iteratively compute all subproblems until we reach the main problem. For example in subset sum problem, we start by using only the first element. We can either use it or not, so the possible sums are $\{0, A[1]\}$. Now we keep both of these, and also try adding $A[2]$ to each one. We get $\{0, A[1], A[2], A[1] + A[2]\}$. Then we repeat until all of the elements are included. Figure 5 illustrates how bottom-up approach is like filling out a table where at each step we need to calculate the applicable underneath cells.

Unlike bottom-up approach, in top-down approach we try to include the first element in the subset, and recursively see if any combination of the rest of elements add up to the rest of the sum. If we don't find a solution, leave out the first element and continue. In other words, we first break the problem into subproblems and then calculate and store values which is the opposite of bottom-up approach where we solve smaller subproblems first, then solve larger subproblems from them.

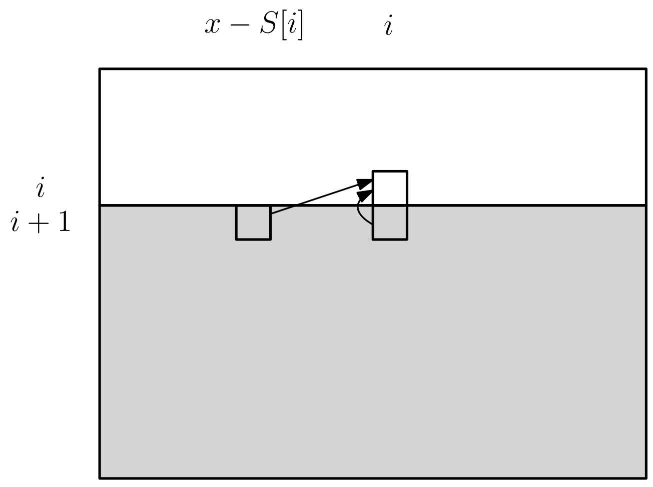

Figure 5: Tabular representation of bottom-up approach for subset sum problem.

The pseudocode for implementation is as follows:

Algorithm 7 Subset-sum: Bottom-Up

```
1: function SubsetSumBottomUp($S[1..n], x$)
2:     Allocate new $(n+1) \times (x+1)$ array $M[0..n][0..x]$
3:     for $i = 0$ to $x$ do
4:         $M[n][i]$ = False
5:     for $i = n$ downto 0 do
6:         for $j = S[i]$ to $x$ do
7:             $M[i][j] = M[i+1][j]$ or $M[i+1][j - S[i]]$
8:     return $M[0][x]$
```

The time complexity of the above pseudocode is dependent on nested loop which is a function of $|S|$ as the size of the set and x as the desired summation. Therefore, we have $O(|S| \cdot x)$ as the running time for the worst-case which is much better than exponential in the case without DP.

3 Optimal Binary Search Tree

The problem of a Optimal Binary Search Tree can be rephrased as:

Given a list of n keys ($A[1, ..., n]$) and their frequencies of access ($F[1, ..., n]$), construct a optimal binary search tree in which the cost of search is minimum.

We will start with a list of keys in a tree and their frequencies. Now to find the best binary tree with minimum cost to search in the tree, we can create all possible binary trees out of our list of keys and find the cost of searching items in those binary trees and pick one of the best binary tree as our optimal binary search tree.

Further we can use divide and conquer strategy to solve our problem. Note that if we have an optimal binary tree then at any node of that tree, it should have an optimal binary search tree on its left child tree and also an optimal binary search tree on its right child tree. Thus we can break our original problem into two sub problems and use same method to solve for those sub problems.

Our objective for this problem is to find the tree with minimum cost for searching. So we can use the cost of searching keys in the tree as our objective function for our recursive function. And before we move on to our recursive solution, let's look at the cost of searching a binary search tree in detail.

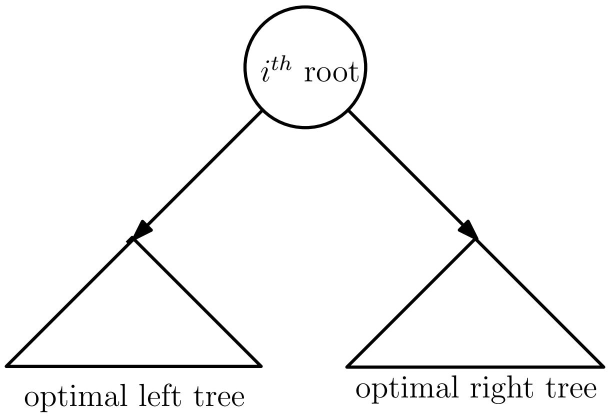

Figure 6: Dividing Optimal Binary Search Tree problem to sub problems

Given any binary search tree T, the total cost of searching can be calculated by calculating sum of product of frequencies of the nodes ($f[v]$) and depth of the nodes in the tree ($d[v]$).

$$cost_{total} = \sum_{v \in T} f[v] \cdot d[v] \tag{1}$$

Thus, the problem of finding the cost of searching the binary search tree can be solved recursively as follows (see Figure 6) for reference).

For the left optimal tree (T_{left}), the cost of searching in the left optimal subtree is given by:

$$cost_{left} = \sum_{v \in T_{left}} f[v] \cdot d[v] \tag{2}$$

Similarly for right optimal subtree (T_{right}), the cost of searching in the right optimal subtree is given by:

$$cost_{right} = \sum_{v \in T_{right}} f[v] \cdot d[v] \tag{3}$$

Now, if we combine left and right optimal trees with the root node just like in Figure 6, the depth for nodes in left and right subtree are increased by 1 and the updated cost for left and right subtree is given by:

$$\begin{aligned}cost_{left\_new} &= \sum_{v\in T_{left}} f[v]\cdot(d[v]+1)\\ &= \sum_{v\in T_{left}} f[v]\cdot d[v] + \sum_{v\in T_{left}} f[v]\\ &= cost_{left} + \sum_{v\in T_{left}} f[v]\end{aligned}$$

$$\begin{aligned}cost_{right\_new} &= \sum_{v\in T_{right}} f[v]\cdot(d[v]+1)\\ &= cost_{right} + \sum_{v\in T_{right}} f[v]\end{aligned}$$

And, the final cost is sum of the cost of search on right, cost of search on left and the number of times we search for the root node (equal to the frequency of the key in the root node).

$$\begin{aligned}cost_{total} &= cost_{left\_new} + cost_{right\_new} + f[root]\\ &= \left(cost_{left} + \sum_{v\in T_{left}} f[v]\right) + \left(cost_{right} + \sum_{v\in T_{right}} f[v]\right) + f[root]\\ &= cost_{left} + cost_{right} + \left(\sum_{v\in T_{left}} f[v] + \sum_{v\in T_{right}} f[v] + f[root]\right)\end{aligned}$$

For our implementation with array of keys and frequencies, if we sort the keys and frequencies based on keys then we can say that if we pick any of the array element as a root node for our tree, the sub-array in the left of the key can be used as nodes for left binary sub-tree and the sub-array in the right of the key can be used as nodes for right sub-tree. For example if we want to create tree out of array that is bound by indices L and R with root node at index i, the cost of searching in the tree can be obtained using:

$$\begin{aligned}cost_{total} &= \sum_{v=L}^{i-1} f[v]\cdot d[v] + \sum_{v=i+1}^{R} f[v]\cdot d[v] + \left(\sum_{v=L}^{i-1} f[v] + \sum_{v=i+1}^{R} f[v] + f[i]\right)\\ &= \sum_{v=L}^{i-1} f[v]\cdot d[v] + \sum_{v=i+1}^{R} f[v]\cdot d[v] + \sum_{v=L}^{R} f[v]\\ &= cost_{left} + cost_{right} + \sum_{v=L}^{R} f[v]\end{aligned}$$

So using the cost function we can implement recursive solution for the problem as:

Algorithm 8 MAIN

```
1: Sort $A[1..n]$ and $F[1..n]$ by the keys
2: return findOptimalCost($A$, $F$, 1, $n$)
```

Algorithm 9 findOptimalCost(A, F, $left$, $right$)

```
1: $best = +\infty$
2: for $i = left$ to $right$ do                                ▷ try each as the root
3:     $cost_{left} = $ findOptimalCost$(A, F, left, i - 1)$
4:     $cost_{right} = $ findOptimalCost$(A, F, i + 1, right)$
5:     $cost_{total} = cost_{left} + cost_{right} + \sum_{v=left}^{right} F[v]$
6:     if $cost_{total} < cost_{best}$ then
7:         $best = cost_{total}$
8: return $best$
```